Principia Softwarica: The Plan 9 Kernel 9

Raspberry Pi (ARM 32 bits) edition

version 0.1

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
Ken Thompson and Richard Miller

April 28, 2026

The text and figures are Copyright © 2014–2026 Yoann Padioleau.
All rights reserved.

The source code is Copyright © 2021 Plan 9 Foundation.
MIT license.

Contents

1 Introduction **15**
- 1.1 Motivations . . . 15
- 1.2 The Plan 9 kernel: `9` . . . 16
- 1.3 Other kernels . . . 16
- 1.4 Getting started . . . 18
- 1.5 Requirements . . . 19
- 1.6 About this document . . . 19
- 1.7 Copyright . . . 19
- 1.8 Acknowledgments . . . 20

2 Overview **21**
- 2.1 Operating system principles . . . 21
  - 2.1.1 Kernel mode vs user mode . . . 21
  - 2.1.2 Virtual memory and address spaces . . . 22
  - 2.1.3 Paging versus segmentation . . . 23
  - 2.1.4 Interrupts, exceptions, traps, and syscalls . . . 24
  - 2.1.5 Concurrency, races, and locks . . . 25
  - 2.1.6 User stack and kernel stack . . . 25
  - 2.1.7 Processes, threads, and context switching . . . 26
  - 2.1.8 Files, descriptors, and the file-system interface . . . 27
  - 2.1.9 Monolithic versus microkernel . . . 28
  - 2.1.10 Capabilities versus access-control lists . . . 29
  - 2.1.11 The kernel as a concurrent program . . . 29
- 2.2 9's syscall interface . . . 30
  - 2.2.1 Process syscalls . . . 30
  - 2.2.2 Memory syscalls . . . 31
  - 2.2.3 File syscalls . . . 31
  - 2.2.4 Directory syscalls . . . 32
  - 2.2.5 Namespace syscalls . . . 32
  - 2.2.6 Time syscalls . . . 33
  - 2.2.7 IPC syscalls . . . 33
  - 2.2.8 Concurrency syscalls . . . 34
  - 2.2.9 Security syscalls . . . 34
- 2.3 `helloraw.c` and `helloraw.s` (ARM) . . . 34
- 2.4 `minishell.c` . . . 36
- 2.5 The Raspberry Pi machine (ARM) . . . 36
  - 2.5.1 ARM CPU . . . 36
  - 2.5.2 ARM coprocessors . . . 39
  - 2.5.3 ARM MMU . . . 40
  - 2.5.4 Raspberry Pi Input/Output devices . . . 42

2.6 C and assembly (ARM) . . . 45
2.7 Code organization . . . 48
2.7.1 Portable versus arch-specific code . . . 49
2.7.2 Main headers . . . 49
2.7.3 `dat.h` and `fns.h` . . . 49
2.8 Software architecture . . . 50
2.8.1 Booting overview (ARM) . . . 50
2.8.2 Memory overview (ARM) . . . 51
2.8.3 Syscall overview (ARM) . . . 56
2.8.4 FAQ (ARM) . . . 59
2.8.5 FAQ . . . 60
2.9 Book structure . . . 61

3 Core Data Structures **62**
3.1 `Conf` and `conf` . . . 62
3.2 `Cpu` and `cpu` . . . 63
3.3 `Proc` and `up` . . . 65
3.4 `Chan` . . . 68
3.5 `Dev` and `devtab` . . . 71
3.6 `Soc` and `soc` (ARM) . . . 71

4 Concurrency **73**
4.1 Critical regions . . . 73
4.2 Interrupts disabling: `arch_splhi()`, `arch_spllo()` (ARM) . . . 73
4.3 Test and set: `arch_tas()` (ARM) . . . 75
4.4 Atomic count: `arch_xinc()` and `arch_xdec()` (ARM) . . . 76
4.5 Spin `Lock` . . . 76
4.6 Uninterruptable `ILock` . . . 79
4.7 Waiting `QLock` . . . 80
4.8 Readers/Writer `RWLock` . . . 83
4.9 Atomic `Ref`erences and `Counter` . . . 86
4.10 Safeguards . . . 87
4.10.1 `Lock.p` . . . 88
4.10.2 `Proc.nlocks` . . . 88
4.10.3 `Cpu.ilockdepth` . . . 89
4.11 Synchronization . . . 89
4.11.1 Rendez-vous . . . 89
4.11.2 Producer/Consumer Queue . . . 90
4.11.3 Semaphore . . . 90

I Virtualizing the Machine **91**

5 Memory **92**
5.1 Overview . . . 92
5.2 Long-lived structure allocator: `xalloc()` . . . 93
5.2.1 Data structures . . . 93
5.2.2 `xalloc()` . . . 95
5.2.3 `xfree()` . . . 96
5.2.4 `xmerge()` . . . 98

5.3 Memory pools . . . 99
5.3.1 Pool . . . 99
5.3.2 plock() . . . 100
5.3.3 pmainmem . . . 101
5.3.4 poolprint() . . . 102
5.4 Kernel malloc() . . . 103
5.4.1 malloc() . . . 103
5.4.2 Sleeping smalloc() . . . 104
5.4.3 mallocxxx() . . . 104
5.4.4 malloc tags . . . 105
5.5 User malloc() . . . 107
5.6 Page and palloc . . . 107
5.6.1 Page . . . 107
5.6.2 Palloc . . . 108
5.6.3 newpage() . . . 109
5.7 Segment and Pagetable . . . 112
5.7.1 Segment . . . 113
5.7.2 Pagetable . . . 114
5.7.3 newseg() . . . 116
5.7.4 smallpagedir . . . 117
5.7.5 Adding a page to a segment: segpage() . . . 118
5.7.6 Process segments . . . 118
5.7.7 /proc/<pid>/segment . . . 122
5.8 sysbrk() . . . 123
5.8.1 Extending a segment . . . 124
5.8.2 Shrinking a segment . . . 125

6 Processes **127**
6.1 Overview . . . 127
6.2 Process allocation . . . 127
6.2.1 Process ID: Proc.pid . . . 128
6.2.2 Procalloc . . . 128
6.2.3 PID hashing . . . 128
6.2.4 newproc() . . . 129
6.2.5 No more processes . . . 130
6.3 The user . . . 132
6.4 sysrfork() . . . 133
6.4.1 Fork: RFPROC . . . 134
6.4.2 Memory sharing: RFMEM . . . 135
6.4.3 arch_forkchild() (ARM) . . . 135
6.4.4 File descriptors: RFFDG, RFCFDG . . . 137
6.4.5 Namespace: RFNAMEG, RFCNAMEG . . . 137
6.4.6 Environment variables: RFENVG, RFCENVG . . . 138
6.4.7 Rendezvous group: RFREND . . . 138
6.4.8 Note group: RFNOTEG . . . 138
6.4.9 Process sharing settings (no RFPROC) . . . 139
6.5 sysexec() . . . 140
6.5.1 Text segment . . . 143
6.5.2 Data segment . . . 144
6.5.3 Bss segment . . . 144

6.5.4 Stack segment . . . 144
6.5.5 arch_execregs() (ARM) . . . 148
6.6 sysawait() . . . 148
6.6.1 Parent and children . . . 148
6.6.2 Waitmsg and Waitq . . . 149
6.6.3 sysawait() . . . 150
6.6.4 Rendez-vous: Proc.waitr . . . 151
6.7 sysexits() . . . 151
6.7.1 pexit() . . . 152
6.7.2 pexit() and schedinit() . . . 154
6.7.3 Wait message . . . 154
6.7.4 Freeing resources . . . 155
6.7.5 No waiting for child: RFNOWAIT . . . 156
6.8 Kernel processes . . . 157
6.8.1 arch_kprocchild() (ARM) . . . 158
6.9 Process control /proc/<pid>/ctl . . . 159

7 Interrupts **160**
7.1 Overview . . . 160
7.2 Exceptions (ARM) . . . 161
7.3 Exception vectors (ARM) . . . 161
7.4 System calls . . . 162
7.4.1 _vsvc() (ARM) . . . 162
7.4.2 Sargs . . . 163
7.4.3 arch__syscall() (ARM) . . . 163
7.5 Exception handlers . . . 166
7.5.1 _vxxx() (ARM) . . . 166
7.5.2 _vswitch() (ARM) . . . 166
7.5.3 arch__trap() (ARM) . . . 168
7.5.4 Undefined instruction fault (ARM) . . . 170
7.5.5 Prefetch fault (ARM) . . . 171
7.5.6 Data fault (ARM) . . . 171
7.6 IRQs (ARM) . . . 174
7.6.1 Vctl (ARM) . . . 174
7.6.2 Intregs (ARM) . . . 175
7.6.3 arch_intrenable() (ARM) . . . 175
7.6.4 Basic IRQs (ARM) . . . 176
7.6.5 irq() (ARM) . . . 177

8 Virtual Memory **178**
8.1 Overview . . . 178
8.2 The ARM MMU (ARM) . . . 178
8.3 Pagination and mmupd/mmupt (ARM) . . . 179
8.4 Page fault . . . 182
8.4.1 faultarm() (ARM) . . . 182
8.4.2 fault() . . . 183
8.4.3 fixfault() . . . 183
8.4.4 arch_putmmu() (ARM) . . . 186
8.5 Copy on write . . . 188
8.6 Page cache . . . 189

8.7 Demand loading . . . 193
8.7.1 Kimage and the image cache . . . 193
8.7.2 Segment and KImage . . . 194
8.7.3 Paging in . . . 197
8.7.4 duppage() . . . 199
8.7.5 Collecting images . . . 201
8.8 Swapping . . . 204
8.8.1 Swapalloc . . . 204
8.8.2 /dev/swap . . . 205
8.8.3 PageOrSwap . . . 206
8.8.4 Swapping plan . . . 207
8.8.5 The pager . . . 210
8.8.6 Paging out . . . 215
8.8.7 Paging in . . . 216
8.8.8 Setting up the swap file . . . 216

9 Scheduling 218
9.1 Scheduling principles . . . 218
9.2 Overview . . . 219
9.3 Process priority . . . 219
9.4 The runqueue . . . 221
9.5 The scheduler . . . 223
9.5.1 sched() . . . 223
9.5.2 schedinit() . . . 224
9.5.3 runproc() . . . 225
9.6 Context switch . . . 226
9.6.1 Label . . . 226
9.6.2 Kernel stacks . . . 226
9.6.3 arch_gotolabel() (ARM) . . . 227
9.6.4 sched() and schedinit() dance . . . 227
9.6.5 Delayed sched() . . . 228
9.6.6 arch_mmuswitch() (ARM) . . . 230
9.6.7 arch_procsave() (ARM) . . . 230
9.7 Cooperative scheduling . . . 231
9.7.1 yield() . . . 231
9.7.2 ready() . . . 231
9.7.3 Rendez . . . 232
9.7.4 sleep() . . . 233
9.7.5 wakeup() . . . 235
9.8 Preemptive scheduling . . . 235
9.8.1 hzsched() . . . 236
9.8.2 hzclock() . . . 236
9.9 Customized scheduling . . . 237
9.10 Real-time scheduling . . . 238
9.11 Adjusting priority . . . 238
9.11.1 reprioritize() . . . 238
9.11.2 rebalance() . . . 241
9.12 Halting (ARM) . . . 242

10 Time **243**
10.1 Time principles . . . 243
10.2 Overview (ARM) . . . 244
10.3 Time units . . . 244
10.3.1 Seconds . . . 244
10.3.2 Ticks . . . 244
10.3.3 Fastticks (ARM) . . . 245
10.4 Clock interrupt (ARM) . . . 246
10.5 Timers . . . 247
10.5.1 `Timer` . . . 247
10.5.2 `Timers` and CPU `timers` . . . 248
10.5.3 `tadd()/tdel()` . . . 248
10.5.4 `timeradd()//timerdel()` . . . 250
10.5.5 `arch_timerset()` (ARM) . . . 251
10.5.6 `timerintr()` . . . 252
10.5.7 Clock callbacks: `addclock0link()` . . . 253
10.5.8 Delay (ARM) . . . 254
10.6 `syssleep()` . . . 254
10.7 Alarms . . . 256
10.8 `sysalarm()` . . . 257
10.8.1 `alarmkproc()` . . . 258
10.8.2 `checkalarms()` . . . 259
10.9 Time of day . . . 259
10.10`sysnsec()` . . . 264
10.11`/dev/time` . . . 264
10.12`/dev/bintime` . . . 266
10.13Watchdog . . . 268

11 Initialization **269**
11.1 Booting the kernel (ARM) . . . 269
11.1.1 Loading the kernel in physical memory (ARM) . . . 269
11.1.2 First instructions (ARM) . . . 269
11.1.3 Basic pagination (ARM) . . . 270
11.1.4 Jumping to C `main()` (ARM) . . . 272
11.2 Kernel initialization . . . 272
11.2.1 `cpu` initialization (ARM) . . . 274
11.2.2 Memory detection (ARM) . . . 275
11.2.3 `conf` initialization (ARM) . . . 276
11.2.4 Exception vector initialization (ARM) . . . 278
11.2.5 Clock initialization (ARM) . . . 280
11.2.6 Timer initialization . . . 281
11.2.7 Virtual memory initialisation (ARM) . . . 281
11.2.8 Memory initialisation . . . 282
11.2.9 Device initialisation . . . 285
11.2.10 Displaying hardware information (ARM) . . . 286
11.3 The first process (ARM) . . . 287
11.4 Booting the user . . . 292

II Everything is a File 293

12 Files 294
12.1 Overview . . . 294
12.2 Channels . . . 294
12.3 Devices table devtab . . . 297
12.4 File descriptors, Fgrp . . . 298
12.5 sysopen(), sysclose() . . . 301
12.6 namec() . . . 304
12.6.1 Checking valid filename string . . . 307
12.6.2 Parsing filename . . . 309
12.6.3 Error management . . . 311
12.7 syspread(), syspwrite() . . . 312
12.8 sysseek() . . . 315
12.9 sysdup() . . . 317
12.10 sysfd2path() (a.k.a pwd()) . . . 318
12.11 Collecting unclosed files . . . 318
12.12 Special opening modes . . . 321
12.12.1 Close on exec . . . 321
12.12.2 Remove on close . . . 322

13 Directories 323
13.1 Overview . . . 323
13.2 Path . . . 324
13.3 '..' . . . 326
13.4 walk() . . . 327
13.5 syscreate() . . . 332
13.6 sysremove() . . . 335
13.7 syschdir() . . . 336
13.8 DirEntry . . . 336
13.9 sysstat() . . . 338

14 Namespaces 342
14.1 Namespace principles . . . 342
14.2 Overview . . . 343
14.3 Mount point . . . 344
14.4 Mount table . . . 346
14.5 Device path . . . 348
14.6 sysbind() . . . 348
14.7 /proc/<pid>/ns . . . 351
14.8 findmount(), domount() . . . 354
14.9 Path mount points and '..' . . . 356
14.10 Union mount . . . 357
14.10.1 Union walk . . . 358
14.10.2 ->umh . . . 358
14.10.3 syspread() for directories . . . 359
14.10.4 MCREATE . . . 361
14.10.5 Union of union . . . 361
14.11 Sandboxing . . . 362
14.12 sysmount() . . . 363

14.13 `sysumount()` . . . 365
14.14 XTODO . . . 367
14.15 XTODO . . . 368

15 Devices **374**
15.1 Driver model principles . . . 374
15.2 The keyboard . . . 375
15.2.1 Overview . . . 375
15.2.2 Scan code . . . 376
15.2.3 Esc . . . 380
15.2.4 `kbd` . . . 381
15.2.5 `echo()` . . . 383
15.2.6 PC Keyboard (x86) . . . 385
15.2.7 Raspberry Pi USB Keyboard (ARM) . . . 385
15.2.8 Changing keyboard: `/dev/kbmap` . . . 385
15.2.9 External keyboard input: `/dev/kbin` . . . 386
15.3 The screen . . . 386
15.3.1 Overview . . . 386
15.3.2 Text output on a PC (x86) . . . 386
15.3.3 Text output on a Raspberry Pi (ARM) . . . 386
15.4 The console device `/dev/cons` . . . 388
15.4.1 Overview . . . 388
15.4.2 Buffered input . . . 388
15.4.3 Output . . . 390
15.4.4 `print()`, `iprint()`, `pprint()` . . . 391
15.4.5 `/dev/cons` . . . 393
15.4.6 `/dev/consctl` and raw input . . . 396
15.5 The mouse . . . 397
15.5.1 Mouse and keyboard . . . 397
15.5.2 PS2 mouse (x86) . . . 398
15.5.3 USB mouse (ARM) . . . 398
15.6 Other devices . . . 398

16 `Queue` **399**

17 Filesystems **422**
17.1 The root filesystem . . . 422
17.2 The `/boot/` filesystem . . . 426
17.3 Helpers . . . 427
17.4 SimpleFS . . . 434
17.5 `Block` . . . 434
17.6 Fileservers . . . 445
17.7 `Fcall` . . . 446
17.8 XTODO . . . 446

III Beyond the Basics **447**

18 IPC **448**
18.1 Notes (signals) . . . 449

18.1.1 sysnotify() (ARM) . . . 450
18.1.2 sysnoted() (ARM) . . . 453
18.1.3 postnote() . . . 455
18.2 Pipes . . . 457
18.2.1 syspipe() . . . 457
18.2.2 /dev/pipe . . . 458
18.3 Shared segment and semaphore . . . 467
18.3.1 Segment . . . 467
18.3.2 Semaphore . . . 473
18.4 sysrendezvous() . . . 479
18.5 Network . . . 480

19 Debugging Support **481**
19.1 Core dumps and broken processes . . . 482
19.2 Stopping a process: /proc/<pid>/ctl . . . 484
19.3 Inspecting process memory: /proc/<pid>/mem . . . 487
19.4 /bin/ratrace . . . 490
19.5 Scheduler events: /proc/trace and /bin/trace . . . 498
19.6 /bin/db . . . 502
19.7 Other . . . 502

20 Profiling Support **503**
20.1 /dev/cputime and /proc/<pid>/status . . . 503
20.2 /bin/time . . . 504
20.3 /bin/tprof . . . 505
20.4 /bin/prof . . . 507

21 Userspace System Programs **510**
21.1 /boot/boot . . . 510
21.2 /root/init . . . 510
21.3 The shell . . . 510
21.3.1 #! . . . 510

22 Advanced Topics **513**
22.1 Floating point . . . 513
22.1.1 FPSaveStatus . . . 513
22.1.2 Arch_FPsave (ARM) . . . 514
22.1.3 Floating point hooks . . . 515
22.1.4 ARM floating point coprocessors (ARM) . . . 518
22.1.5 coproc.c (ARM) . . . 518
22.1.6 vfp.c (ARM) . . . 521
22.1.7 mathxxx() (ARM) . . . 527
22.1.8 /proc/<pid>/fpregs . . . 528
22.2 CPU cache . . . 528
22.2.1 Instruction and data cache (ARM) . . . 529
22.2.2 Cache policy . . . 529
22.2.3 Paging and caching (ARM) . . . 529
22.2.4 Memory barriers (ARM) . . . 530
22.2.5 Cache operations (ARM) . . . 530
22.2.6 cachexxx() (ARM) . . . 531

22.3 Memory model . . . 533
22.3.1 arch_coherence() . . . 534
22.3.2 Memory barriers (ARM) . . . 534
22.4 The Raspberry Pi 2 machine . . . 534
22.4.1 CPU ID . . . 535
22.4.2 MMU . . . 535
22.4.3 TTB . . . 536
22.4.4 Local registers . . . 536
22.4.5 Timer . . . 537
22.4.6 Performance monitor . . . 537
22.5 SMP (ARM) . . . 537
22.5.1 Initialization adjustments . . . 537
22.5.2 Setting cpu . . . 545
22.5.3 Concurrency adjustments . . . 546
22.5.4 Timer adjustments . . . 547
22.5.5 arch_coherence() (ARM) . . . 547
22.5.6 Cache adjustments . . . 548
22.5.7 IRQ adjustments . . . 552
22.5.8 Cycle counter adjustments . . . 553
22.6 Advanced virtual memory . . . 553
22.6.1 Page coloring . . . 554
22.6.2 Extra segment types . . . 556
22.6.3 arch_kmap() . . . 558
22.6.4 Extra checks . . . 558
22.7 Fast interrupts (FIQ) (ARM) . . . 560
22.7.1 Exception PsrMfiq and associated processor mode . . . 560
22.7.2 IRQfiq and vfiq . . . 561
22.7.3 fiq() . . . 562
22.8 Power management (ARM) . . . 562
22.9 Alignements (ARM) . . . 563
22.10 Real-time scheduling . . . 564
22.11 Fileservers . . . 579
22.11.1 Mnt . . . 580
22.11.2 MntRpc . . . 582
22.11.3 Fcall . . . 582
22.11.4 mntversion() . . . 582
22.11.5 mountrpc() . . . 585
22.11.6 devmnt . . . 588
22.12 Network . . . 590
22.13 Security . . . 590
22.14 Graphics . . . 593

23 Special Fileservers **594**
23.1 Cmdtab . . . 594
23.2 /env/ . . . 596
23.2.1 Egrp . . . 597
23.2.2 /env/ . . . 598
23.3 /proc/ . . . 599
23.3.1 /proc/<pid>/ctl . . . 599
23.3.2 /proc/<pid>/mem . . . 599

23.3.3 /proc/<pid>/status . . . 599
23.3.4 /proc/<pid>/args . . . 600
23.3.5 /proc/<pid>/wait . . . 600
23.3.6 Other /proc/<pid>/xxx . . . 601
23.3.7 XXX . . . 601
23.4 /fd/ . . . 622
23.5 /sys/ . . . 622
23.5.1 /sys/osversion . . . 624
23.5.2 /sys/config . . . 624
23.5.3 /ys/hosthowner, /sys/hostdomain . . . 624
23.5.4 /sys/sysname . . . 625
23.5.5 /sys/drivers . . . 626
23.5.6 /sys/reboot . . . 626
23.6 /dev/ . . . 633
23.6.1 /dev/random . . . 634

24 Conclusion **638**
24.1 Patterns and techniques . . . 638
24.2 Connections to other books . . . 639
24.3 What this book does not cover . . . 639
24.4 Beyond the Plan 9 kernel . . . 640

A Debugging **641**
A.1 sysnop() . . . 641
A.2 Special keys and dumpers . . . 641
A.3 /bin/ktrace (ARM) . . . 648
A.4 /dev/kmesg . . . 649
A.5 /dev/kprint . . . 651
A.6 Remote debugger . . . 652
A.7 Serial line . . . 653
A.7.1 serialoq . . . 653
A.7.2 uartconsinit() . . . 654
A.7.3 uartputs() . . . 654
A.7.4 consuart . . . 654
A.7.5 Uart, PhysUart . . . 655

B Profiling **656**
B.1 /sys/sysstat (ARM) . . . 656
B.2 /dev/kprof . . . 660
B.3 Statistics . . . 660
B.3.1 Lock.lockcycles . . . 660
B.3.2 TaslockStats . . . 661
B.3.3 QlockStats . . . 661
B.3.4 Kimage statistics . . . 661
B.3.5 Scheduling . . . 662
B.3.6 Clock . . . 662

C Error Management **663**
C.1 panic() . . . 663
C.2 error() . . . 664
C.3 waserror(), nexterror(), poperror() . . . 666
C.4 syserrstr() . . . 667

D Utilities **668**
D.1 Memory areas operations . . . 668
D.2 String functions . . . 668
D.3 Conversion functions . . . 670
D.4 Pool allocation . . . 670
D.5 Printf and Fmt . . . 670
D.6 Runes, a.k.a unicode . . . 672

E Extra Portable Code **673**
E.1 ../include/ . . . 673
E.2 buses/ . . . 673
E.3 concurrency/ . . . 692
E.4 console/ . . . 695
E.5 core/ . . . 701
E.6 devices/ . . . 706
E.7 devices/keyboard/ . . . 708
E.8 devices/mouse/ . . . 716
E.9 devices/sys/ . . . 730
E.10 devices/storage/ . . . 764
E.11 files/ . . . 803
E.12 filesystems/ . . . 826
E.13 init/ . . . 848
E.14 interrupts/ . . . 852
E.15 lib/ . . . 853
E.16 memory/ . . . 855
E.17 misc/ . . . 867
E.18 processes/ . . . 868
E.19 security/ . . . 883
E.20 syscalls/ . . . 884
E.21 time/ . . . 886

F Extra ARM Code **888**
F.1 ../include/arch/arm/ . . . 888
F.2 arch/arm/ . . . 888
F.3 buses/arm/ . . . 898
F.4 concurrency/arm/ . . . 970
F.5 core/arm/ . . . 972
F.6 devices/arm/ . . . 976
F.7 devices/sys/arm/ . . . 977
F.8 devices/storage/arm/ . . . 981
F.9 init/arm/ . . . 990
F.10 interrupts/arm/ . . . 1002
F.11 memory/arm/ . . . 1004
F.12 processes/arm/ . . . 1008

F.13 time/arm/ 1010
F.14 syscalls/arm/ 1012

Glossary **1014**

Indexes **1016**

References **1080**

Chapter 1

Introduction

The goal of this book is to explain with full details the source code of a kernel.

1.1 Motivations

Why a kernel? Because I think you are a better programmer if you fully understand how things work under the hood, and the kernel is the program that everything else rely on.

Indeed, every program you write—from a simple `helloworld` to a web server—relies on the kernel for its most basic operations: reading files, allocating memory, communicating with other programs, and even printing text on the screen. Understanding what happens below those system calls removes an entire layer of mystery and gives you a far more accurate mental model of how software actually executes on hardware.

Moreover, the kernel is a fascinating program in its own right. It must manage concurrency (multiple processes sharing the CPU), virtual memory (giving each process the illusion of having all memory for itself), and devices (keyboards, disks, screens)—all while remaining small enough to be trustworthy and fast enough to stay out of the way. The tension between generality and efficiency produces elegant solutions that are worth studying even if you never write kernel code yourself. Finally, many concepts introduced by the kernel—locks, scheduling, caches, interrupt handling—reappear in user-space programs such as databases, web servers, and even build systems. Understanding the kernel is thus also a way to understand those programs better.

Here are a few questions I hope this book will answer:

- What happens when the user turn on the computer? Which code gets executed? How does the kernel initialise itself?
- What happens when the user types `ls` in a terminal? What are the set of programs involved in such a command? What is the trace of such a command through the different layers of the software stack, from the keyboard interrupt to the display of text glyphs on the screen?
- Where does the kernel store meta data about the user processes?
- Why do the kernel code and some kernel data structures reside in the virtual memory space of every processes?
- What is the trace of a system call?
- What is the memory image of the kernel and the user processes?

1.2 The Plan 9 kernel: 9

Like for most books in Principia Softwarica, I chose a Plan 9 program for this book. The Plan 9 kernel, called 9, was written by Ken Thompson—the creator of UNIX—together with Dave Presotto and others at Bell Labs. If UNIX was "version 1," then Plan 9 is "version 2": a chance to redesign an operating system with decades of hindsight. The result is a kernel that is remarkably small, clean, and coherent—a good basis for the other Principia Softwarica books.

I will focus on the code of 9 for the Raspberry Pi[1], which uses the ARM architecture. ARM is a RISC design, far simpler than x86, making it a better target for a teaching book. Sections covering ARM-specific code are marked with "(ARM)" in their title, but most of the code in this book is portable C—the same source is shared between the ARM and x86 ports of 9. When talking specificially about the ARM and Raspberry Pi port of 9, I will sometimes use the name 9pi.

The graphics stack and the networking stack, which are parts of the 9 kernel, are covered in separate Principia Softwarica volumes (Graphics book [Pad16d] and Network book [Pad16e]), so this book focuses on the kernel's core: processes, memory, files, and devices.

To give a sense of scale: the 9pi kernel—portable C plus ARM-specific code—is about 70,000 lines (about 98% C and headers, only 2% assembly), with another 34,000 lines of userspace helpers under each subsystem's user/ subdirectory; see Table 2.1 for the per-subdirectory breakdown. The x86 port of the kernel adds another 28,000 lines. Stripping out the graphics stack (devices/screen/, covered in Graphics book [Pad16d]) and the networking stack (network/, covered in Network book [Pad16e]) brings the 9pi kernel down to about 47,000 lines, most of which is in devices/ and buses/—the device drivers for the Raspberry Pi board. The portable C of the core subsystems—processes, memory, files, scheduling, interrupts, syscalls, initialization—is smaller still, about 20,000 lines.

I have been praising 9 for being small, clean, and coherent, but that simplicity comes with real trade-offs that are worth naming before we dive in. The "everything is a file" UNIX philosophy pushed to its limit in Plan 9 is elegant but can be less efficient than specialized interfaces—for instance, Plan 9 has no buffer cache for its file servers (unlike Linux's page cache). The graphics stack lives in the kernel (devdraw), which adds complexity to a component that could arguably be in user space. The physical memory is split between kernel, image cache, and user segments with hardcoded ratios, rather than dynamically balanced as in Linux. These are conscious choices: Plan 9 optimizes for simplicity and uniformity, sometimes at the cost of performance.

1.3 Other kernels

Here are a few kernels that I considered for this book, but which I ultimately discarded:

- UNIX V6[2] (Ken Thompson et al.) runs only on the PDP-11, an architecture long since dead, and predates multiprocessor hardware, networking, and graphical displays. It is also already famously covered line-by-line in the Lions book [Lio77], so there is little point redoing that work here. V6 is Bell Labs research UNIX from 1975, the sixth edition of the operating system Thompson and Ritchie had been building since 1969. The whole kernel fits in about 9,000 lines of C and PDP-11 assembly; Lions' 1977 commentary was the first published open source-code reading in the industry.

- xv6[3], V6's modern incarnation (MIT's teaching rewrite for x86 in 2006, ported to RISC-V in 2019, used in the 6.828 operating-systems course), fixes all of the above: it runs on today's hardware, supports SMP, and is small—around 13,000 lines of C. Since xv6 has no graphics and no networking, the fair comparison is against 9's portable core—processes, memory, files, scheduling, interrupts, syscalls, initialization—which comes to about 20,000 lines of C (the device drivers and bus controllers for the Raspberry Pi sit on top

[1] https://www.raspberrypi.org/
[2] http://minnie.tuhs.org/cgi-bin/utree.pl
[3] http://pdos.csail.mit.edu/6.828/2014/xv6.html

of that core but are not really what xv6 is competing with; conversely, much of `9`'s USB stack and several filesystem drivers such as `dossrv` live in user space outside the kernel altogether). For those ~7,000 extra lines over xv6, `9` offers considerably more: per-process namespaces via `bind`, `mount`, and `rfork`; the 9P protocol for mounting remote file servers; light-weight threads sharing memory through `rfork(RFMEM)`; note-based signal handling; a demand-paged memory system with copy-on-write `fork` and an image cache; and a priority/EDF scheduler. xv6 by contrast stays deliberately simple—round-robin scheduling, no copy-on-write, UNIX-style signals—exactly what makes it a clean classroom exercise but not a system anyone would actually run.

- XINU[4] ("Xinu Is Not Unix", Douglas Comer) has a network stack, but the kernel itself is too simple for this book: the process model does not go much beyond "run a routine, wait on a semaphore", there are no namespaces or mounts, and the whole system is aimed at embedded devices rather than a general-purpose workstation. XINU was first written at Purdue in 1984 as the companion code to his two-volume textbook *Operating System Design* [Com84, Com87]. The original kernel fits in around 4,000 lines of C—an order of magnitude smaller than `9`'s core—which is exactly what makes it such a good classroom kernel. The second volume is devoted to its TCP/IP implementation, which is why XINU is often read in networking classes. A new edition in 2015 added virtual memory.

- Minix[5] (Andrew Tanenbaum et al.) is fairly small but relies on X11 for its windowing system, which is far more complicated than Plan 9's rio. Minix was first written at the Vrije Universiteit Amsterdam as the companion code to *Operating Systems: Design and Implementation* [Tan87] (1987)—the book Linus Torvalds was reading when he started Linux four years later. Minix 3 (2005) rewrote the system as a proper microkernel with fault-isolated user-space drivers; at around 13,000 LOC it is in the same ballpark as `9`'s 20,000-line portable core (though much smaller than a full `9pi` kernel with its drivers), but has never had a user base beyond its own classroom.

- Linux[6] (Linus Torvalds et al.) is really big. Of course the Linux kernel contains thousands of specific device drivers and those things could be discarded when presenting the core of the Linux kernel, but this core is still far bigger than Plan 9 equivalent core. Linux was announced by Linus Torvalds on comp.os.minix in August 1991 ("I'm doing a (free) operating system, just a hobby...") and now runs most of the world's servers, phones, and embedded devices. It is documented at book length in Bovet & Cesati's *Understanding the Linux Kernel* [BC05] and in Love's *Linux Kernel Development* [Lov10], but even those cover only a fraction of what is by now a very large system: just the architecture-independent core (memory, scheduler, VFS, signals), stripped of drivers and filesystems, is well over 400k LOC today.

- GNU Hurd[7] over Mach[8] (Richard Stallman, Thomas Bushnell et al.) is a pure microkernel design—a tiny message-passing core and a cloud of user-space servers for filesystem, authentication, networking, ...—that would have been tempting on paper. Mach itself came out of Richard Rashid's group at Carnegie Mellon in the mid-1980s (and later became the foundation of NeXTSTEP and, via XNU, of macOS and iOS); the GNU project announced Hurd on top of Mach in 1990, shipped Hurd 0.1 in 1996, and the system is still marked experimental today. And despite the "micro" label, Mach itself is not particularly small: around 60–100k lines depending on what you count, big enough that Jochen Liedtke wrote L4 in the early 1990s (around 10k LOC) as an explicit reaction to Mach's bloat. Add the Hurd servers on top and the whole stack ends up several times the size of `9`'s core, for something nobody actually runs. Plan 9 itself borrows some microkernel ideas—everything-is-a-file as a unified IPC bus, and pushing services out of the kernel: the window system `rio`, the authentication agent `factotum`, and—years before FUSE landed

[4] http://minnie.tuhs.org/cgi-bin/utree.pl?file=Xinu7
[5] http://minnie.tuhs.org/cgi-bin/utree.pl?file=Minix1.1
[6] https://kernel.org
[7] https://www.gnu.org/software/hurd/
[8] https://www.gnu.org/software/hurd/microkernel/mach.html

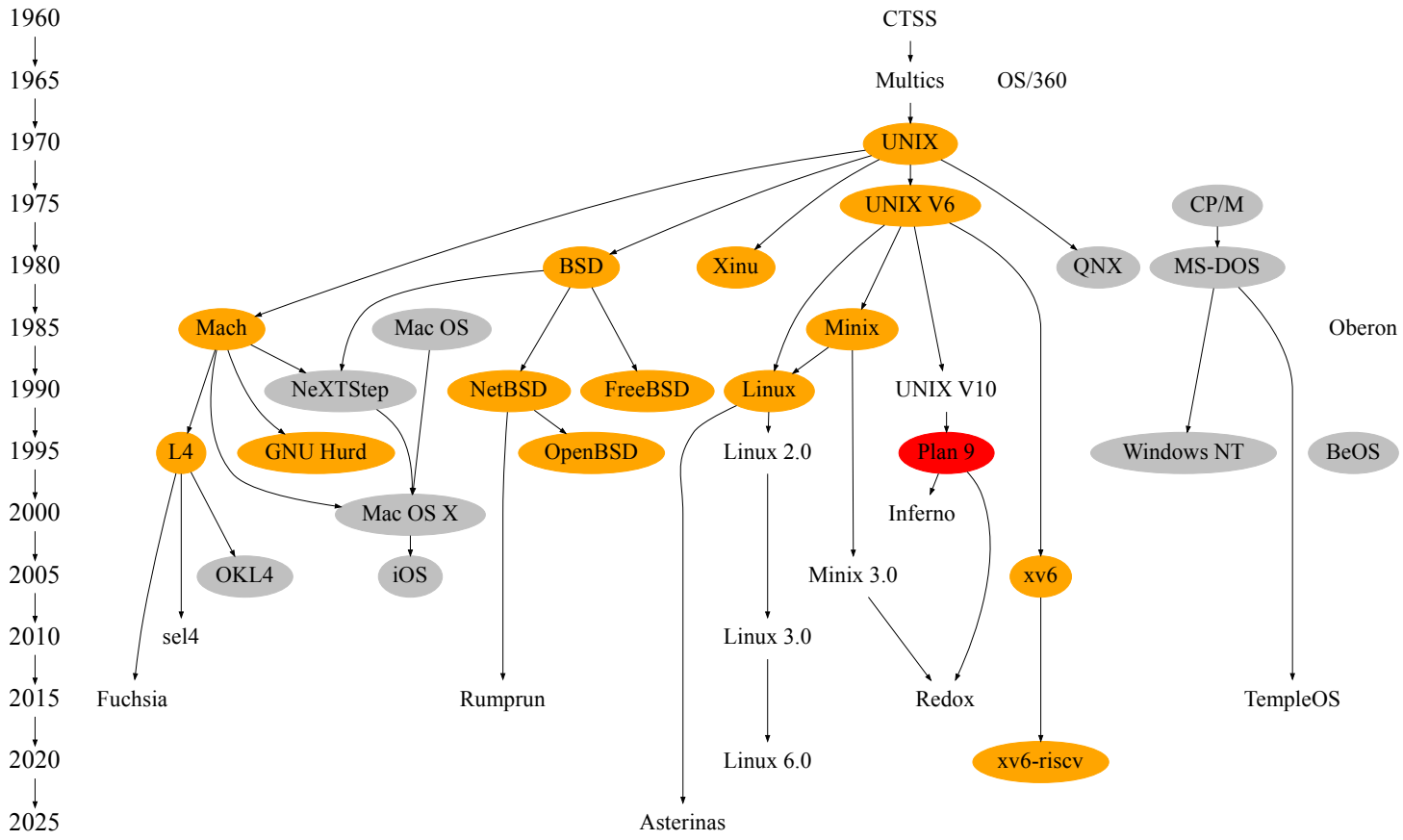


Figure 1.1: Kernels timeline

on Linux in 2006—entire filesystems like `fossil` and `dossrv` running as ordinary user-space file servers. Plan 9 nonetheless keeps drivers and scheduling in a single address space, trading full process isolation for simplicity.

Figure 1.1 presents a timeline of major kernels. I think `9` represents the best compromise for this book: it implements the essential features of an operating system kernel—processes, virtual memory, file systems, device drivers—while still having a small and understandable codebase. `9` is obviously not as used as the Linux kernel, but it is a production-quality kernel that was used to run all machines at Bell Labs and that still runs today on hobbyist hardware such as the Raspberry Pi.

1.4 Getting started

To get started you first need to get the source of the Plan 9 version described in this book at https://github.com/aryx/principia-softwarica. Then you need to compile using a regular C compiler (e.g. gcc, clang) the Plan 9 C cross compilers written by Ken Thompson: https://github.com/aryx/goken9cc. Then you need to cross compile, e.g. from Linux or macOS, the Plan 9 kernel using the Plan 9 C cross compilers installed in the previous step. You can then cross compile the whole Plan 9 operating system with all its libraries and utilities. Finally you can test this compiled kernel and utilities by installing them on a SD card and by booting this SD card on a Raspberry Pi.

All of this is explained in more details at https://www.principia-softwarica.org/getting-started.html, which also explains how to cross-compile instead the x86 version of the kernel and boot it using Qemu[9]

[9] http://www.qemu.org

1.5 Requirements

I assume you actually know a lot:

- C: most of this book is made of C source code, so you'll need to have a good knowledge of the C programming language [KR88]. In fact, kernel C is often more demanding than application C: the code makes heavy use of bit manipulation, pointer arithmetic, C idioms for implementing data structures (linked lists via embedded pointers, bitsets, union-based tagged structures), and subtle interactions with the hardware. A very good knowledge of C is required.

- Operating system: This book is not an introduction to operating systems; its goal is to explain a real kernel in full detail rather than to teach OS theory from scratch. That said, the Overview chapter opens with a short OS-agnostic recap of the concepts that come up repeatedly in the code to follow—privilege levels, virtual memory, interrupts, stacks, concurrency and races—so you do not need to keep a textbook at your side just to get through the first few chapters. Beyond that recap, familiarity with virtual memory, critical regions, interrupts, and system calls is helpful, and standard OS textbooks such as Tanenbaum [TB14] and Silberschatz [SGG18] provide the full theoretical treatment for readers who want it. For kernel source code walkthroughs similar in spirit to this book, see Lions [Lio77] (UNIX V6), Bach [Bac86] (System V, whose algorithms Plan 9 inherited), Tanenbaum [Tan87] (MINIX), Bovet and Cesati [BC05] (Linux), and McKusick et al. [MNNW14] (FreeBSD).

The following are helpful but not strictly required:

- The ARM architecture: some familiarity with the ARM processor helps for the architecture-specific chapters, but most of the kernel code is portable C. The EMULATOR book [Pad15b] and ASSEMBLER book [Pad15a] can serve as references.

- Assembly: a few key routines (context switch, interrupt entry, boot code) are written in assembly. The ASSEMBLER book [Pad15a] covers the Asm5 language used here.

If, while reading this book, you have specific questions on the syscall and device interfaces implemented by the kernel, I suggest you to consult the manual pages in my Plan 9 repository. Section 2 (`docs/man/2/`) documents the system calls described in various chapters (e.g., `2/open`, `2/bind`, `2/rfork`, `2/notify`, `2/segattach`, `2/sleep`, `2/intro`); Section 3 (`docs/man/3/`) documents the kernel device files (e.g., `3/proc`, `3/cons`, `3/pipe`, `3/root`, `3/mnt`, `3/draw`, `3/ip`); Section 8 (`docs/man/8/`) documents administration commands such as `8/boot`, `8/plan9.ini`, and `8/init`. The `kernel/docs/` directory contains two papers useful for the chapters on synchronization and storage: `kernel/docs/sleep.ps` ("Process Sleep and Wakeup on a Shared-memory Multiprocessor") and `kernel/docs/fossil.pdf` (the Fossil file server paper). The top-level `docs/articles/` directory also contains several papers that motivate design decisions discussed in this book: `docs/articles/9.pdf` ("Plan 9 from Bell Labs"), `docs/articles/names.ps` ("The Use of Name Spaces in Plan 9"), and `docs/articles/utf.ps` ("Hello World").

1.6 About this document

This document is a *literate program* [Knu92]. It derives from a set of files processed by a tool, `syncweb` [Pad09], generating either this book or the actual source code of the program. The code and its documentation are thus strongly connected.

1.7 Copyright

Most of this document is made of source code from Plan 9, so those parts are copyright by Plan 9 Foundation. The prose is mine and is licensed under the All Rights Reserved License.

1.8 Acknowledgments

I would like to acknowledge of course Plan9's authors who wrote most of this book: Ken Thompson, Rob Pike, Dave Presotto, Phil Winterbottom, Russ Cox, and many other people from Bell Labs. Thanks also to Federic Balesteros who wrote a lengthy note about the Plan 9 kernel. Finally, thanks to Richard Miller for porting the Plan 9 kernel to the Raspberry Pi machine.

Chapter 2

Overview

In this chapter, I will give a bird's-eye view of the `9` kernel before diving into the code in later chapters. I will first recall general operating system principles, then present the syscall interface of `9`, the ARM hardware the kernel runs on, its memory layout, and finally the overall software architecture and boot sequence.

2.1 Operating system principles

An operating system is a program that manages other programs. It provides convenient abstractions of the hardware and multiplex the hardware resources for the benefit of user programs. The operating system usually provides:

- a virtual CPU, so that each process thinks it has all the CPU for itself (via preemptive scheduling and the timer interrupt)
- a virtual memory, so that each process thinks it has all memory for itself (via hardware virtual memory features such as paging)
- a virtual screen, so that each process thinks it has all the screen for itself (via the windowing system)
- a file abstraction, so a process can read and store data whether it's on a disk, network, floppy, etc
- a directory abstraction, so a process can organize data whether it's on a disk, network, floppy, etc

The common thread is virtualization: each of these abstractions gives a process the illusion of exclusive access to a resource. Making this illusion work is the kernel's main job, and it requires solving several hard problems: concurrency (multiplexing the CPU safely), memory management (giving each process its own address space), interrupt handling (reacting to hardware events), and persistence (storing data reliably on disk). These are exactly the topics the rest of this book will cover.

Entire books have been written on each of those topics, and standard OS textbooks such as Tanenbaum [TB14] and Silberschatz [SGG18] provide the full theoretical treatment. I will not try to compete with them. But there are a handful of concepts that come up repeatedly in the code to follow, and rather than assume the reader has the right textbook open in another tab, I devote the next few subsections to a brief recap of each—just enough to make the rest of the book readable on its own. These recaps are deliberately OS-agnostic; each subsection then points at the chapter where the concrete Plan 9 version lives.

2.1.1 Kernel mode vs user mode

Every mainstream CPU provides at least two *privilege levels*: a privileged mode (the kernel runs here) with unrestricted access to every hardware register, memory page, and instruction; and a restricted mode (user

programs run here) where many instructions are forbidden, many memory regions are inaccessible, and every attempt to touch hardware goes through the kernel. The hardware enforces the split: user code physically cannot disable interrupts, reload the page-table pointer, or poke the disk controller.

The reason is isolation. If every program could touch every register, a single buggy or malicious process would take down the machine or leak data from every other process. Privilege levels let the kernel be the sole arbiter of who gets to do what, which in turn makes it possible to multiplex the CPU, memory, and I/O devices safely among untrusting programs. It is also what makes the kernel's job so delicate: a bug in kernel code cannot be sandboxed, because the kernel is already below the sandbox.

Transitions between modes are never voluntary from the user side. The CPU switches to privileged mode automatically in response to one of a fixed set of hardware events—a timer interrupt, a page fault, a deliberate trap instruction such as a system call—and each event vectors execution into a preconfigured kernel entry point. There is no "please switch me to supervisor mode" instruction available in user mode; the only way in is through an entry point the kernel itself chose. Different architectures spell the modes differently (`ring 0` vs `ring 3` on x86, `SVC` vs `USR` on ARM, `M/S/U` on RISC-V), and some define more than two levels (ARM and RISC-V each have a hypervisor mode for running VMs), but the core idea is the same everywhere. Plan 9's kernel runs in supervisor mode and user processes run in user mode; see Chapter 7 for the actual entry and return code on ARM.

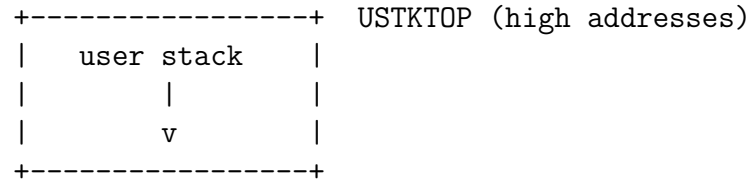
The privilege-ring idea was pioneered by Multics (MIT/GE/Bell Labs, 1965), which had eight rings; x86's four-ring design was directly inspired by it, and most modern systems collapse the four to two because the intermediate rings were rarely used. The same two-mode concept appears under different names in every mainstream kernel: Linux and Plan 9 use "kernel mode" and "user mode," Windows calls them the same (though internal documentation says ring 0 and ring 3), and macOS inherits the Mach "kernel task" and "user task" distinction from its NeXTSTEP lineage.

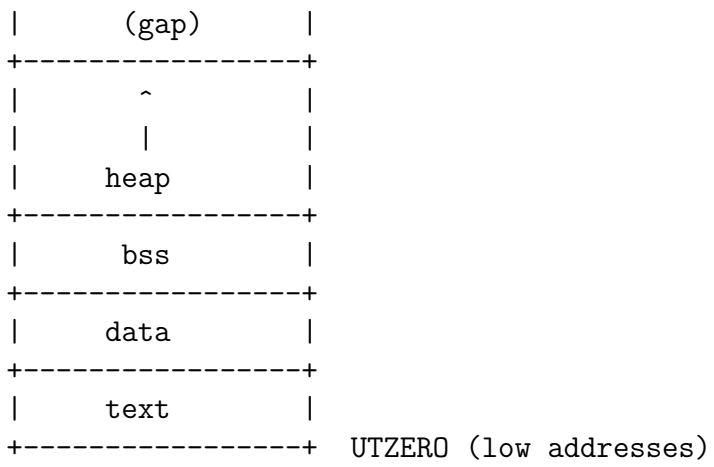
2.1.2 Virtual memory and address spaces

Modern CPUs almost never let software touch physical memory directly. Instead, every memory access goes through a hardware translation unit called the *MMU*, which maps a *virtual address* (what the program sees) to a *physical address* (what the RAM chips see). The translation is driven by per-process data structures called *page tables* that the kernel sets up and the MMU walks on every access (with a cache, the TLB, to make it fast). At any moment the MMU is using the page tables of a specific process—its *address space*—and will fault if code touches a virtual address that has no mapping.

This apparatus does three things at once. It gives each process the illusion of owning the whole address space from address 0 upward, which makes linking and loading trivial (every program can be linked at the same fixed address). It *isolates* processes from each other: process A has no mapping for process B's pages, so a stray pointer in A can at worst corrupt A. And it lets the kernel implement *on-demand* memory management—pages can be created lazily on first access (demand-zero), shared between processes (copy-on-write fork), backed by a file (memory-mapped I/O), or even written out to disk (swap)—all by editing page-table entries rather than moving bytes around.

A typical user process's virtual address space is organized into a handful of *segments*: a read-only text segment holding the code, an initialized-data segment (data), a zero-initialized segment (bss), a dynamically-growing heap, and a downward-growing stack. The layout is conventionally drawn with low addresses at the bottom and high addresses at the top (a convention I will follow for every memory-layout diagram throughout this book):





The kernel lives in an entirely separate region of the virtual address space, typically at high addresses above some `KZERO` boundary, and is mapped identically in every process so that a trap from any user process lands in the same kernel without changing page tables. Only the kernel can read or write its own region thanks to the privilege check from Section 2.1.1. Different operating systems arrange the details differently—Linux on x86-64 splits the canonical 48-bit address space in half at `0xffff800000000000`, Windows splits 32-bit space at `0x80000000`—but the "user low, kernel high, kernel shared" shape is near-universal. Plan 9's specific layout is shown in the memory-layout diagrams later in this chapter, and the full virtual memory implementation lives in Chapter 8.

Virtual memory was invented for the Atlas computer at Manchester University in 1962—the first machine to give each program the illusion of a large private memory backed by a smaller physical store—and was adopted by Multics (1965), which added the per-process page-table concept. The UNIX lineage picked it up gradually: early UNIX had no VM at all (programs were swapped whole), BSD 3 (1979) added demand-paging, and BSD 4.3's `mmap` (1986) gave user programs explicit control over the mapping.

2.1.3 Paging versus segmentation

Paging is not the only way hardware can provide virtual memory. The alternative, *segmentation*, divides the address space into variable-length *segments*, each with a name, a base address, a length, and access permissions. A pointer in a segmented system is a (segment, offset) pair, and the hardware checks the offset against the segment's length on every access. The idea is seductive: segments map to the logical units programmers actually think about—one segment for the code, one for the stack, one per module, one per shared data structure—so protection boundaries align naturally with program structure. Multics (MIT/GE/Bell Labs, 1965) built its entire security model around this: up to 256K segments, each tagged with a ring-level access mask, forming the most fine-grained hardware-enforced protection system ever deployed.

Paging won for a prosaic reason: *fixed-size pages do not fragment*. Variable-length segments leave gaps in physical memory that are too small to reuse (*external fragmentation*), eventually forcing expensive compaction. Fixed-size pages—typically 4 KB—have no external fragmentation at all: any free frame can hold any page, and the kernel just maintains a free-frame list. Paging is also invisible to the compiler: the program emits flat addresses, the MMU translates them, and no one has to produce segment-relative code. The combination was so much simpler to implement and so much more predictable at runtime that every mainstream system—UNIX, Windows, Linux, macOS, Plan 9—chose paging and ignored segmentation entirely.

Intel's x86 is the monument to what might have been. The 80286 (1982) added protected-mode segmentation with per-segment permissions—a microprocessor-scale Multics. The 80386 (1985) layered paging *on top of* segmentation, giving the programmer both. And then… nobody used the segmentation half. UNIX, Windows, and every other serious OS set all segment bases to 0 and all limits to the maximum, creating a *flat* address space and relying entirely on paging. Segments survived only for two niche uses: the `FS` and `GS` registers for thread-local storage (Linux still uses `FS` for the TLS base), and the code-segment descriptor for distinguishing

32-bit from 64-bit mode. When AMD designed x86-64 in 2003, it made the surrender official: in 64-bit long mode, segment bases are forced to 0 except for `FS` and `GS`. ARM and RISC-V never had segmentation at all—they went straight to paging, and Plan 9's kernel does the same. The elegant Multics vision of "one segment per logical module, hardware-enforced" was never fulfilled; modern systems achieve similar goals through software isolation (WebAssembly sandboxes, language-level capabilities) or page-level permissions instead.

2.1.4 Interrupts, exceptions, traps, and syscalls

These four words get thrown around interchangeably in casual conversation, but the underlying hardware mechanisms split them along two clean axes: synchronous vs asynchronous and deliberate vs accidental. The usual taxonomy is:

- An *interrupt* is **asynchronous** and always comes from outside the CPU: a timer tick, a network packet arriving, a keystroke on a keyboard, a disk finishing a read. It can land between any two user instructions, and the running program is not involved in requesting it.
- An *exception* is **synchronous** and caused accidentally by the currently executing instruction: dividing by zero, executing an illegal opcode, touching a virtual address with no mapping. The instruction is the cause, so the faulting `PC` is reproducible.
- A *trap* (also called a *software interrupt*) is **synchronous** and **deliberate**: the program executes a special instruction—`SWI` on ARM, `INT` on x86, `ECALL` on RISC-V—to request service from the kernel. Breakpoints (the debugger instruction `BKPT`/`INT3`) are another flavor of trap.
- A *system call* is a specific kind of trap whose purpose is to ask the kernel to perform an operation on the process's behalf: open a file, send on a pipe, fork. The "syscall number" passed in a register tells the kernel which operation.

The two axes fit naturally into a 2 x 2 table:

	Synchronous (caused by current instruction)	Asynchronous (from outside the CPU)
Accidental (not asked for)	exception div by zero, page fault, illegal opcode	interrupt timer tick, keystroke, disk done, network packet
Deliberate (requested by the program)	trap SWI / INT / ECALL, breakpoint, system call	IPI (inter-processor interrupt: one CPU signals another)

The deliberate/asynchronous cell is the rarest of the four and only matters on multi-processor systems, where one CPU can ask another to run a handler via an IPI (used, for example, to tell every other core "flush your TLB, I just changed a page table").

All four route through the same mechanism from the CPU's point of view: a hardware *vector table* (ARM has eight entries, x86 has 256, RISC-V's `mtvec` points at a single handler that demultiplexes). When any of these events fires, the CPU switches to privileged mode (see Section 2.1.1), jumps to the corresponding entry

point, and starts running kernel code. The kernel's first job is always the same: save enough of the interrupted program's state to be able to resume it later. Its next job depends on which kind of event fired—a timer interrupt drives the scheduler, a page-fault exception walks the page tables, a syscall trap dispatches through a table of function pointers—but the entry, state-save, and exit machinery are shared.

The synchronous/asynchronous distinction has a far-reaching consequence: because interrupts can land between any two instructions, every piece of kernel code that touches shared state must either disable interrupts around the critical section or use a lock that interrupts respect. This is why the book has an entire chapter on concurrency (Chapter 4) before you see any "real" kernel code—getting interrupt safety right is the first thing a kernel must do. The actual mechanics of ARM exception vectors, syscall dispatch, and IRQ delivery then live in Chapter 7.

2.1.5 Concurrency, races, and locks

Consider two pieces of code that both execute `counter = counter + 1` on a shared variable. On a real machine this compiles to something like load, add 1, store. If an interrupt fires between the load and the store, or if a second CPU runs its own load in the middle, both executions can end up observing the same old value, each adding 1, and each writing it back—losing one of the two increments. This kind of bug is called a *data race*, and it is the foundational problem of concurrent programming. The region of code that must execute "all-or-nothing" relative to other parallel executions (here, the three instructions around `counter`) is called a *critical section*.

Kernels have two sources of concurrency to worry about. The first is *asynchronous interrupts*: even on a single CPU with a single process, an interrupt handler can fire between any two instructions of kernel code (see Section 2.1.4), so any kernel data structure also touched by a handler is potentially racing with that handler. The simplest fix on a uniprocessor is to disable interrupts for the duration of the critical section—brutal but effective. The second source is *multiple CPUs*: on an SMP machine, two physical cores can be running kernel code simultaneously, and disabling interrupts on one core does nothing to stop the other. For that you need a proper *lock*: a piece of shared state that can be held by at most one CPU at a time, combined with a hardware atomic instruction (test-and-set, compare-and-swap) that guarantees the "check if free, then claim" step itself executes all-or-nothing.

Kernels typically offer a small zoo of lock variants because the cost of waiting is context-dependent. A *spinlock* simply busy-waits ("spin") until the lock is free, which is the right choice for very short critical sections where yielding the CPU would cost more than a few wasted cycles. A *sleeping lock* parks the waiter on a wait queue and schedules another process, which is the right choice for long critical sections or when the lock holder itself may sleep. An *interrupt-disabling lock* combines a spinlock with local interrupt-disable, for data structures shared between ordinary kernel code and interrupt handlers. Plan 9 has one of each: `Lock` (spinlock), `ILock` (spinlock that also disables interrupts), `QLock` (sleeping lock), and `RWLock` (reader/writer sleeping lock). See Chapter 4 for the implementations.

The same zoo appears under different names in every production kernel. Linux has `spin_lock` (spinlock), `mutex` (sleeping lock), `rw_semaphore` (reader/writer sleeping lock), and `raw_spin_lock` (spinlock with interrupts disabled on RT kernels). FreeBSD has `mtx` (mutex), `sx` (shared/exclusive sleeping lock), and `rwlock`. Windows has `KSPIN_LOCK`, `KMUTEX`, and the user-space `CRITICAL_SECTION`. The naming differs; the underlying design—a fast lock for short critical sections, a sleeping lock for long ones, and a variant that also disables interrupts—is the same everywhere.

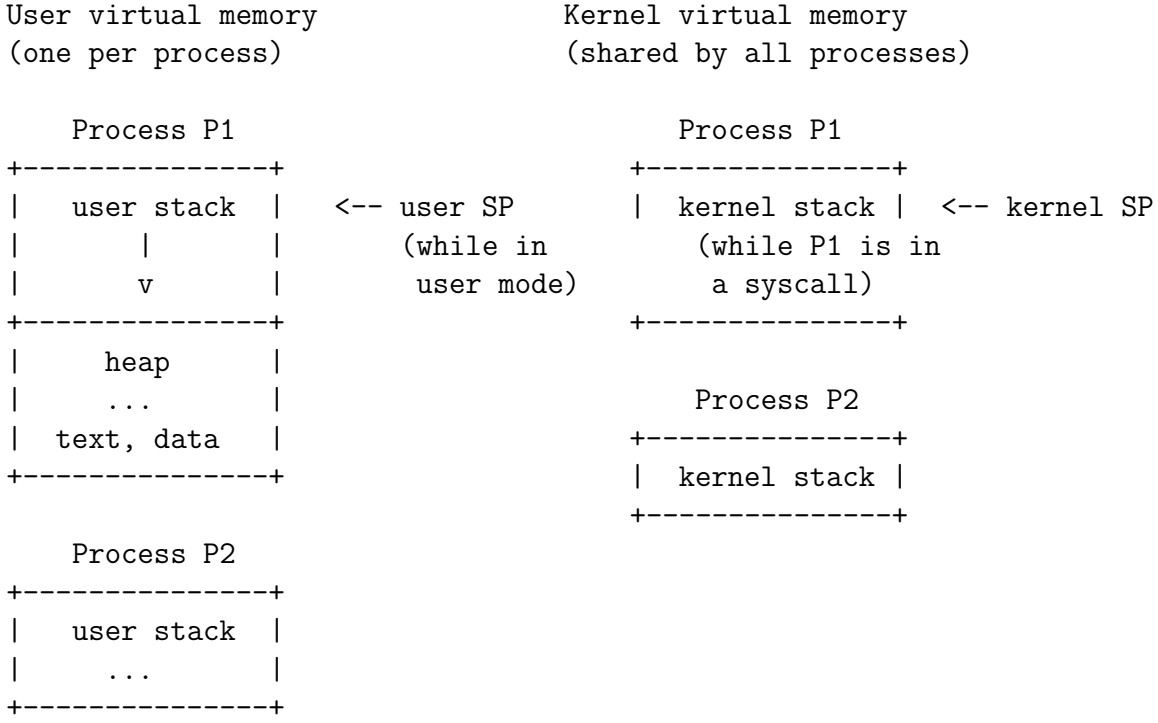
2.1.6 User stack and kernel stack

Every process in a modern operating system owns two stacks. The user stack lives in user-mode virtual memory and is manipulated by ordinary user code using the CPU's stack pointer register. When a trap, exception, or interrupt (Section 2.1.4) vectors execution into the kernel, the CPU switches the stack pointer to a separate kernel stack that lives in kernel memory, and every stack frame pushed by kernel code while it is handling the

event lands on that kernel stack. When the kernel returns to user mode, the user stack pointer is restored and the user stack resumes where it left off.

Two reasons for the split. First, safety: when a trap fires, the user stack pointer could be anything—pointing into unmapped memory, into a deliberately crafted region, or into a valid but nearly-exhausted stack. The kernel refuses to trust it at all; it switches to a known-good kernel stack as its very first action. Second, re-entrancy: if the kernel were to keep running on the user stack, then a trap taken while the kernel is already running (because a timer interrupt arrived during a syscall) would have nowhere to push its state. A dedicated kernel stack solves this cleanly.

There is one more design choice: whose kernel stack does the kernel use? Some historical systems had a single kernel stack per CPU, which forced kernel code to run to completion without blocking—once it started, nothing else could enter the kernel on that CPU until it was done. Modern systems, including Plan 9, instead give every process its own kernel stack, living either inside or next to the process's `Proc` structure. A process that has trapped into the kernel can therefore sleep mid-syscall (e.g., waiting for a disk read to complete); its kernel-side execution state—saved registers, return addresses, local variables—sits frozen on its own kernel stack, and a different process can safely be scheduled onto the CPU. When the sleeper is eventually woken, the scheduler restores its stack pointer and execution resumes exactly where it left off.



Plan 9's concrete version of this picture, including the `Ureg` structure saved on entry and the `Label` structure used to park a blocked kernel-side execution, is shown later in this chapter (in the calling-convention diagram of the C-and-assembly section) and in full detail in Chapter 6.

2.1.7 Processes, threads, and context switching

Everything above—the user/kernel split, the per-process address space, the user and kernel stacks—adds up to the kernel's most fundamental abstraction: the *process*. A process is a bundle of state that makes it look, from the program's own perspective, as if it owns the entire machine. The state has two halves. The static half is the address space: the page tables that define which virtual addresses exist and what they map to (see Section 2.1.2). The dynamic half is the execution state: the program counter, the general-purpose registers, the user stack pointer, and, whenever the process has trapped into the kernel, the kernel stack pointer and whatever kernel-side frames are currently parked on it.

A brief word on *threads*. In the usual textbook picture, multiple threads can live inside the same address space—sharing the static half, each with its own dynamic half—so that switching between them avoids reloading the page tables. This picture is not how Plan 9 uses the word: in Plan 9 the kernel-schedulable entity is always called a *process*, and "thread" (in the sense of `libthread`) is a *user-space cooperative* construct built inside a single process (see LIBCORE book [Pad16c]). For the rest of this book, "process" means the kernel-schedulable entity with its own address space and its own dynamic state; when two such processes want to share memory, Plan 9 has `rfork` flags for it, but each remains its own process from the kernel's point of view.

Other kernels draw the process/thread boundary differently. Linux's `clone()` syscall with the `CLONE_VM` flag creates a new schedulable entity that shares the parent's address space—what POSIX calls a "thread"—and `pthreads` is a user-space library built on top of it. Windows has always had first-class in-process threads (via `CreateThread`), each with its own kernel stack but sharing the process's address space. Plan 9's `rfork` with `RFMEM` achieves the same effect but keeps the word "process" for the kernel-schedulable entity, leaving "thread" to `libthread` in user space (see LIBCORE book [Pad16c]).

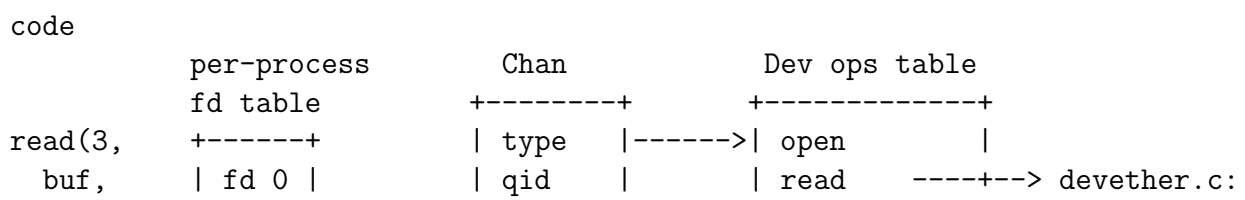
A *context switch* is the act of pausing one process and resuming another on the same CPU. Mechanically it is just a save-and-restore of the dynamic half: store the current registers into a per-process save area, load another process's saved registers, update the page-table base register if the address space is changing, and jump to the new PC. Context switches come in two flavors. A *voluntary* switch happens when the running process explicitly gives up the CPU—for instance because it is waiting on I/O and calls a `sleep()`-style kernel routine. Because the process knows the switch is about to happen, only a small save area is needed (essentially `SP`+`PC` plus a handful of bookkeeping registers). An *involuntary* switch happens when an interrupt fires and the scheduler decides to run something else; the interrupted process had no say in the matter, so the kernel must save *every* register before switching. The combination of voluntary and involuntary switches is what makes scheduling preemptive: a misbehaving process cannot hog the CPU because the timer interrupt will eventually take it away.

Plan 9's concrete objects for all of this are `Proc` (the process structure, one per process), `Label` (the small save area used by voluntary switches), `Ureg` (the full-register save area used by involuntary switches), and `Segment` (per-address-space region records). `Proc`, `Label`, and `Segment` are introduced in Chapter 3; `Ureg` is shown later in this chapter. Process creation, `sleep`/`wakeup`, and the scheduler itself live in Chapter 6 and Chapter 9.

2.1.8 Files, descriptors, and the file-system interface

A process's only window onto persistent storage, devices, networks, and inter-process communication is a small set of calls on *file descriptors*—small integers returned by `open` that the kernel uses as indices into a per-process table of opaque handles. Each handle ultimately points at a kernel-internal object (Linux's `struct file`, BSD's *vnode*, Windows' `HANDLE`) that carries enough state to route the next `read`/`write`/`close` to the right implementation. The calls themselves arrive via the usual syscall trap (Section 2.1.4); the small-integer form of the descriptor is what makes `fork`, `dup`, and `exec` cheap, because nothing but the integer gets copied.

The whole point of the descriptor is *uniformity*. The same handful of calls that read a regular file also read a pipe, a TTY, a network socket, a tape drive, or a kernel-internal data structure exposed as a pseudo-file (Linux `/proc` and `/sys`, BSD `sysctl` proxied through `/dev`). The kernel achieves this by routing every call through an *operations table*—a struct of function pointers, one per supported operation, that every file-system or device type fills in. Touching a regular file walks to the ext4 ops table; touching a socket walks to the TCP ops table; the user-side code is identical. This is the C-style object dispatch the book will keep meeting:



```
n)        | fd 1 |          | flag   |        | write        |    etherread(...)
          | fd 2 |          | offset |        | stat         |
          | fd 3 |--------->| ops   -+-----/  | close        |
          | ...  |          +--------+        +--------------+
          +------+
```

Where UNIX stops at "everything that can be read or written is a file", Plan 9 pushes further: *every service is a file server*. The process table, the window system, the network stack, the debugger interface, and every in-kernel device appear as directories and files in a namespace, served either by in-kernel code or by an out-of-kernel user process speaking the *9P* protocol over a pipe. There is no `kill` syscall—you write the string "kill" into `/proc/pid/note`; there is no `ps` or `netstat`—you `read /proc` or `/net`. The consequence is that the kernel's total syscall surface shrinks to around 30 calls (against Linux's 400-odd) while the set of things programs can do grows, because adding a new service means writing a file server, not extending the kernel. It also makes the system trivially distributable: since every service already speaks 9P, reaching a remote service is the same code path as reaching a local one (see NETWORK book [Pad16e]).

Plan 9's concrete objects for this story are `Chan` (the in-kernel handle behind a user file descriptor), `Dev` (its operations table), `Qid` (the type-plus-instance identifier, matching UNIX's inode number but carrying a version tag that lets clients detect stale caches), and `Fgrp` (the per-process descriptor table mapping small integers to `Chan` pointers). `Chan`, `Qid`, `Fgrp`, and the `Dev` structure are introduced in Chapter 3; the device chapters that follow are essentially one `Dev` implementation each, and 9P itself along with the `devmnt` driver that turns a remote file server into a local `Chan` are covered in Chapter 23.

2.1.9 Monolithic versus microkernel

Once you have kernel mode and user mode (Section 2.1.1), and a set of services the kernel provides—file systems, networking, device drivers, memory management—the question is: which of those services must run *inside* the kernel, and which could run as ordinary user-space processes? A *monolithic* kernel puts everything in kernel mode: device drivers, file systems, the network stack, and the scheduler all share one address space and call each other as ordinary functions. A *microkernel* reduces the kernel to the bare minimum—scheduling, IPC, and memory management—and runs everything else as user-space servers that communicate via message passing.

The trade-off is *speed* versus *isolation*. Monolithic kernels are fast because a syscall that touches the file system, then the network stack, then a device driver never leaves kernel mode—each step is just a C function call. But a single buggy driver can corrupt the entire kernel's memory and crash the machine. Microkernels isolate each service in its own address space, so a buggy driver takes down only itself—but every inter-service call requires crossing the user/kernel boundary twice (out and back), which historically made microkernels significantly slower for real workloads.

The debate came to a head publicly in 1992 when Andrew Tanenbaum (author of MINIX) called Torvalds's monolithic Linux "a giant step back to the 1970s" on the `comp.os.minix` newsgroup; Torvalds replied that Linux worked *now* and performance mattered. History proved both partially right. First-generation microkernels (Mach, CMU 1985) were indeed slow, and macOS and iOS today run on a Mach-derived kernel (XNU) but keep most services in-kernel for performance—a "hybrid" that is really monolithic in practice. But second-generation microkernels rewrote the rules: Jochen Liedtke's L4 (1993) brought IPC latency down to roughly one microsecond, and SEL4 (2009) became the first operating-system kernel with a complete formal proof of correctness—something that remains out of reach for any monolithic kernel. Today SEL4 runs in high-assurance embedded systems, and Google's Fuchsia uses the Zircon microkernel for its consumer OS.

Plan 9 sits at an interesting point in this spectrum. Its kernel is monolithic—device drivers, the network stack, the file-system layer, and the scheduler all share one address space—but its "everything is a file server" philosophy (Section 2.1.8) means that many services which would be in-kernel on Linux run as ordinary user-space processes: `rio` (the window manager), `cs` (the connection server), `dns`, `exportfs`, and the authentication daemons. These user-space servers communicate with the kernel and with each other via 9P, so they get the

microkernel benefit of isolation and replaceability without imposing the IPC cost on the *fast path*: the in-kernel `devXxx` drivers handle the performance-critical work as function calls, and only the configuration and policy services move out to user space. The boundary is drawn by a pragmatic rule: if it needs fast-path access to hardware or kernel data structures, it stays inside; everything else moves out.

The microkernel lost the mainstream battle, but its *ideas* keep returning inside monolithic systems. Linux's eBPF lets user-written programs run safely inside the kernel—which is a microkernel idea (user-provided code in the kernel's address space) implemented without a microkernel. Containers (Docker, Kubernetes) give per-process isolation that microkernels promised, using Linux namespaces and cgroups instead of separate address spaces. Unikernels (MirageOS, IncludeOS) collapse kernel and application into one image—the opposite of a microkernel, yet motivated by the same desire for minimal, isolated services. The microkernel lost the war but won the peace: its goals survived by being absorbed into the system that beat it.

2.1.10 Capabilities versus access-control lists

Operating systems need to decide who can access what. Two fundamentally different models answer this question. An *access-control list* (ACL) attaches permissions to the *resource*: a file has an owner, a group, and a set of read/write/execute bits (UNIX's `rwxrwxrwx`), or a richer list of entries (Windows' DACL/SACL). The system checks the caller's identity against the list on every access. A *capability* is the mirror image: an unforgeable *token* held by the *caller* that grants a specific right to a specific resource. If you hold the token, you can use it; if you don't, you can't—no identity check needed. The idea was introduced by Jack Dennis and Earl Van Horn at MIT in 1966, three years before UNIX existed.

Capabilities are theoretically elegant: they obey the *principle of least privilege* naturally (you give a process exactly the tokens it needs and nothing more), they compose cleanly (passing a capability to another process is just passing a value), and they are amenable to formal verification. But they never caught on in mainstream operating systems, mainly because *revoking* a capability is hard—once you hand someone a token, taking it back requires cooperation or indirection. UNIX chose ACLs (the `rwx` bits), Windows chose ACLs (DACLs), and the capability idea retreated to research systems like Hydra, KeyKOS, and EROS. Then it started coming back: SEL4 (2009) built its entire security model on formal capabilities; WebAssembly uses capabilities for sandboxing (a module can only access the functions and memory regions it was explicitly given handles to); and CloudFlare Workers use a capability-like model for isolating tenant code.

Plan 9 takes a third path that sidesteps the ACL-vs-capability dichotomy altogether. Its security model is the *per-process namespace*: instead of checking whether you are *allowed* to access a file, Plan 9 controls whether you can *see* the file at all. If `/net/tcp/clone` is not in your namespace, you cannot open a TCP connection—no ACL check needed, because the name does not exist for you. This is a softer version of the capability idea (a namespace is, in effect, a set of capabilities to see certain file trees), achieved without unforgeable tokens or a formal capability model, using only the `bind`/`mount`/`rfork` machinery already described in Section 2.1.8.

2.1.11 The kernel as a concurrent program

Kernels were arguably the first widely-deployed *concurrent* programs: they had to juggle multiple processes sharing a single CPU long before multi-threaded user applications were common. Most of the vocabulary and algorithms that user-space programmers now take for granted—semaphores (Dijkstra, 1965, for the THE operating system), monitors (Hoare, 1974, explicitly framed as an OS-structuring primitive), atomic test-and-set, wait-free data structures—were invented for kernel programming and only later adopted by application developers. The same story repeated with SMP: early commercial multiprocessors (VAX/VMS in 1980, Sequent in 1983) forced kernel authors to confront multi-core correctness decades before ordinary application programmers had to worry about it, which only happened around 2005 when Intel hit the frequency wall and multi-core CPUs became standard in desktops. The kernel's solutions still inform most of today's user-space patterns.

That history is more than trivia: it is also the right reading lens for the chapters that follow. A kernel can be read as an unusual concurrent program whose "threads" are the per-process kernel stacks from Section 2.1.6:

one per process, each with its own call stack and saved state, all sharing the same heap (the kernel address space with its globals and dynamically-allocated data structures—which is why the kernel needs locks in the first place). What is unusual, compared to a Java server or a Go program, is that most of these threads are dormant almost all of the time: either the process is running in user mode and its kernel-side thread has nothing to do, or it is blocked in a syscall with its kernel stack frozen on a wait queue. A kernel "thread" only comes alive when an event on its process's behalf (a syscall, a signal, an I/O completion) wakes it up—which is also why the kernel's `sleep/wakeup` primitives look a bit different from user-space `pthread_cond_wait`: the underlying idea is the same, just adapted to this wake-on-event shape.

With those basics in hand, we can now focus on the concrete services—the API—that a kernel provides to user programs.

2.2 9's syscall interface

Plan 9 is the successor to UNIX and inherits its philosophy that "everything is a file." Plan 9 pushes this further: "everything is a file server". Devices, the console, the screen, the process table—all are accessed through `open/read/write/close`, served by in-kernel or user-space file servers. This design lets the syscall interface remain remarkably small.

The entire kernel interface fits in a small single header, `sys.h`, which lives in `lib_core/libc/9syscall/`—it is shared between the kernel and the C library (see LIBCORE book [Pad16c]) because it defines the interface between them. The same directory also contains the auto-generated assembly wrappers (one per syscall) that user programs link against. Each syscall is a small integer used as an index into a dispatch table (the `systab`[57b] array shown later).

⟨`lib_core/libc/9syscall/sys.h` 30a⟩≡
```
#define NOP     0
⟨sys.h process syscalls 30b⟩
⟨sys.h memory syscalls 31a⟩
⟨sys.h file syscalls 31b⟩
⟨sys.h directory syscalls 32a⟩
⟨sys.h namespace syscalls 32b⟩
⟨sys.h time syscalls 33a⟩
⟨sys.h ipc syscalls 33b⟩
⟨sys.h concurrency syscalls 34a⟩
⟨sys.h security syscalls 34b⟩
#define ERRSTR      39
```

Only 39 syscalls! Modern Linux has more than 400. The reason Plan 9 can get away with so few is the "everything is a file server" philosophy: operations that would require dedicated syscalls in other systems (e.g., `getpid`, `ioctl`) are instead handled by reading or writing special files.

The `NOP` syscall at position 0 does nothing useful in production—here it just prints "Hello World" for testing purposes. The `ERRSTR` syscall at position 39 is Plan 9's error-reporting mechanism. Unlike UNIX, which uses the opaque numeric error code `errno` (where you need a man page to find out what `ENOENT` means), Plan 9 uses human-readable error strings that programs retrieve via `errstr()`—one of the many small cleanups Plan 9 made over UNIX.

I now describe each category in turn, with the C prototypes from `syscall.h` (see LIBCORE book [Pad16c] for the library wrappers built on top of these).

2.2.1 Process syscalls

The process syscalls mirror the UNIX model: `RFORK` creates a new process, `EXEC` replaces the current program, `EXITS` terminates, and `AWAIT` waits for a child.

⟨`sys.h` *process syscalls* 30b⟩≡ (30a)

```
#define RFORK       1
#define EXEC        2
#define EXITS       3
#define AWAIT       4
```

The C prototypes are:

```
int   rfork(int flags);
int   exec(char *name, char *argv[]);
void  _exits(char *status);
int   await(char *s, int n);
```

Plan 9 cleans up UNIX in several ways here. Instead of having separate `fork()` and `clone()` syscalls (as Linux does), Plan 9 has a single `RFORK` ("resource fork") that takes flags specifying which resources to copy and which to share: memory, file descriptors, environment, namespace, etc. A traditional `fork` is `RFORK` with everything copied; a thread (shared memory) is `RFORK` with memory shared. Linux's `clone()` was in fact inspired by Plan 9's `rfork`.

Similarly, `EXITS` takes an error string (or `nil` for success), not a numeric exit code. The parent reads this string via `AWAIT`. No need to memorize that exit code 2 means `ENOENT`—the string says `"file not found"` directly.

2.2.2 Memory syscalls

There is only one memory syscall: `BRK` (for "break"), which moves the boundary between the data segment and free memory. The name comes from the traditional UNIX term for the end of the process's data area. There is no `malloc` or `free` syscall—those are implemented entirely in user space (in the C library, see LIBCORE book [Pad16c]) on top of `BRK`.

⟨`sys.h` *memory syscalls* 31a⟩≡ (30a)

```
#define BRK     5
```

The C prototype is:

```
int   brk(void *addr);
```

2.2.3 File syscalls

The file syscalls are the workhorses of the interface. Because so many things in Plan 9 are files, most of the kernel's services are accessed through these seven calls. They split into two groups: fd-slot lifecycle (`OPEN`, `CLOSE`, `DUP`, `FD2PATH`) and channel I/O (`PREAD`, `PWRITE`, `SEEK`).

⟨`sys.h` *file syscalls* 31b⟩≡ (30a)

```
#define OPEN        6
#define CLOSE       7
#define DUP         8
#define FD2PATH     9
#define PREAD       10
#define PWRITE      11
#define SEEK        12
```

The C prototypes are:

```
int   open(char *file, int omode);
int   close(int fd);
int   dup(int oldfd, int newfd);            // -1 newfd = allocate
int   fd2path(int fd, char *buf, int nbuf);
```

```
long  pread(int fd, void *buf, long n, vlong offset);
long  pwrite(int fd, void *buf, long n, vlong offset);
vlong seek(int fd, vlong n, int type);
```

`dup` matches UNIX's `dup2` (with `-1` for "allocate a new slot" in place of the UNIX`dup` form). `fd2path` has no UNIX equivalent: in Plan 9, with per-process namespaces, the same file can be reached via different paths in different processes and `..` is resolved lexically, so the kernel remembers the path used to open each fd.

Note `PREAD` and `PWRITE` instead of the traditional UNIX `read`/`write`: the 'p' stands for "positional"—they take an explicit file offset, making seek-then-read atomic. In UNIX, `read` and `seek` are separate syscalls, so a concurrent `seek`+`read` from two threads can race. Plan 9 avoids this by combining them into a single `pread` call. (UNIX later adopted `pread`/`pwrite` in POSIX.1-2001, but the traditional `read`/`write` remain the primary interface.)

2.2.4 Directory syscalls

The directory syscalls include `CREATE`, `REMOVE`, `CHDIR`, and a family of `STAT` variants. The proliferation of `STAT`/`FSTAT`/`WSTAT`/`FWSTAT` comes from two axes: `STAT` vs. `FSTAT` (take a path name vs. a file descriptor), and read vs. write ("w" for write, i.e., change metadata).

⟨`sys.h` *directory syscalls* 32a⟩≡ (30a)
```
#define CREATE      13
#define REMOVE      14
#define CHDIR       15
#define STAT        16
#define FSTAT       17
#define WSTAT       18
#define FWSTAT      19
```

The C prototypes are:

```
int   create(char *file, int omode, ulong perm);
int   remove(char *file);
int   chdir(char *dir);
int   stat(char *file, uchar *buf, int nbuf);
int   fstat(int fd, uchar *buf, int nbuf);
int   wstat(char *file, uchar *buf, int nbuf);
int   fwstat(int fd, uchar *buf, int nbuf);
```

Note `CREATE` with the full spelling—in UNIX, the syscall was famously called `creat` without the final 'e'. Ken Thompson once said the one thing he would change about UNIX is "I'd spell creat with an e."

2.2.5 Namespace syscalls

The namespace syscalls are specific to Plan 9. `BIND` and `MOUNT` let each process customize its own view of the file system by overlaying directories or attaching file servers at arbitrary mount points.

⟨`sys.h` *namespace syscalls* 32b⟩≡ (30a)
```
#define BIND        20
#define MOUNT       21
#define UNMOUNT     22
```

The C prototypes are:

```
int   bind(char *new, char *old, int flags);
int   mount(int fd, int afd, char *old, int flags, char *aname);
int   unmount(char *new, char *old);
```

UNIX's `mount` is a privileged operation (root only) that attaches a disk partition. Plan 9's `mount` is unprivileged and far more general: any process can mount any 9P file server at any point in its own namespace. This enables user-space filesystems—Linux's FUSE arrived two decades later with a narrower version of the same idea—but also much more: `rio`, the windowing system, uses it to give each window its own private `/dev/cons`, `/dev/mouse`, and `/dev/draw` served by `rio` itself (see WINDOWS book [Pad16f]); `import` grafts part of another machine's namespace into the local one for transparent remote access; and sandboxing is just a restricted namespace.

`bind` has no UNIX equivalent—the closest is Linux's `mount --bind`, added decades later. UNIX's `chroot` provides a crude approximation of per-process namespaces, but it is all-or-nothing. Linux eventually added mount namespaces (2002), which form the basis of containers (Docker), but they arrived 20 years after Plan 9 had the same capability.

2.2.6 Time syscalls

⟨sys.h *time syscalls* 33a⟩≡ (30a)
```
#define SLEEP       23
#define ALARM       24
```

The C prototypes are:

```
int   sleep(long millisecs);
long  alarm(ulong millisecs);
```

`sleep` suspends the process for `n` milliseconds (if zero or negative, it yields). `alarm` sets a timer: after `n` milliseconds, the kernel sends an `"alarm"` note to the process.

UNIX's `sleep()` and `alarm()` take seconds, not milliseconds. For sub-second timing, UNIX needs `nanosleep` or `setitimer`. Plan 9 uses milliseconds everywhere; there is no sub-millisecond sleep.

2.2.7 IPC syscalls

The IPC syscalls cover pipes, notes (signals), and shared memory segments.

⟨sys.h *ipc syscalls* 33b⟩≡ (30a)
```
#define PIPE        27

#define NOTIFY      25
#define NOTED       26

#define SEGATTACH   28
#define SEGDETACH   29
#define SEGFREE     30
#define SEGFLUSH    31
#define SEGBRK      32
```

The C prototypes are:

```
int   pipe(int fd[2]);
int   notify(void (*f)(void*, char*));
int   noted(int v);
void* segattach(int attr, char *class, void *va, ulong len);
int   segdetach(void *addr);
int   segfree(void *va, ulong len);
int   segflush(void *va, ulong len);
void* segbrk(void *saddr, void *addr);
```

Plan 9's pipes are bidirectional (both ends can read and write), unlike UNIX's unidirectional pipes. Notes carry a string message (`"alarm"`, `"sys: write on closed pipe"`) rather than just a number (`SIGALRM`, `SIGPIPE`). `segattach` is Plan 9's equivalent of UNIX`mmap` for shared memory, but uses named segment classes rather than anonymous mappings.

2.2.8 Concurrency syscalls

The concurrency syscalls provide low-level primitives for user-level synchronization. `RENDEZVOUS` is a meeting-point mechanism, and `SEMACQUIRE`/`SEMRELEASE` provide kernel-backed semaphores. These are the building blocks on top of which the user-level thread library (`libthread`, see LIBCORE book [Pad16c]) is implemented.

⟨`sys.h` *concurrency syscalls* 34a⟩≡ (30a)
```
#define RENDEZVOUS  33
#define SEMACQUIRE  34
#define SEMRELEASE  35
#define TSEMACQUIRE 36
```

The C prototypes are:

```
void* rendezvous(void *tag, void *value);
int   semacquire(long *addr, int block);
long  semrelease(long *addr, long delta);
int   tsemacquire(long *addr, ulong ms);
```

UNIX has no equivalent of `rendezvous`. The closest is Linux's `futex`, which also provides user-space fast paths with kernel fallback, but is more complex.

2.2.9 Security syscalls

The two security-related syscalls handle Plan 9's authentication protocol. `FVERSION` negotiates the 9P protocol version with a file server, and `FAUTH` opens an authentication channel. These are used when mounting remote file servers and are part of Plan 9's security model, which relies on per-connection authentication rather than on a global user database.

⟨`sys.h` *security syscalls* 34b⟩≡ (30a)
```
#define FVERSION    37
#define FAUTH       38
```

The C prototypes are:

```
int   fversion(int fd, int bufsize, char *version, int nversion);
int   fauth(int fd, char *aname);
```

And that is the entire syscall interface. There is no `getpid`—a process reads `/dev/pid` instead. There is no `ioctl`—device control is done by writing commands to a file. There is no `select`—`libthread`'s `alt()` uses `rendezvous` instead (see LIBCORE book [Pad16c]). The file server abstraction absorbs what would otherwise be dozens of special-purpose syscalls.

2.3 `helloraw.c` and `helloraw.s` (ARM)

To see the syscall interface concretely, it is always good to start with the simplest possible program. Here is a "Hello World" that uses `pwrite()` directly, without going through libc `print()`:

⟨`helloraw.c` 34c⟩≡
```
#include <u.h>
#include <libc.h>
```

```
void main() {
  pwrite(1, "hello world\n", 12, 0LL);
  exits(nil);
}
```

The `pwrite()` call writes 12 bytes to file descriptor `1` (standard output) at offset 0.[1] Under the hood, `pwrite()` is a thin assembly wrapper that on ARM places the syscall number (`PWRITE`, which is `9` from `sys.h`) in register `R0` and executes a `SWI` (ARM software interrupt) instruction to trap into the kernel. The ARM assembly (see the ASSEMBLER book [Pad15a]) version makes this mechanism fully visible:

⟨helloraw.s 35⟩≡

```
TEXT _main(SB), $20
        /* pwrite(1, &hello, 12, 0LL) */
        MOVW $1, R1
        MOVW R1, 4(R13)
        MOVW $hello(SB), R1
        MOVW R1, 8(R13)
        MOVW $12, R1
        MOVW R1, 12(R13)
        MOVW $0, R1
        MOVW R1, 16(R13)
        MOVW R1, 20(R13)
        MOVW $9 /*PWRITE*/, R0
        SWI $0

        /* exits(0) */
        MOVW $0, R1
        MOVW R1, 4(R13)
        MOVW $3 /*EXITS*/, R0
        SWI $0

GLOBL   hello(SB), $12
DATA    hello+0(SB)/8, $"hello wo"
DATA    hello+8(SB)/4, $"rld\n"
```

To compile and run the C version:

```
$ 5c helloraw.c
$ 5l helloraw.5 -o helloraw
$ ./helloraw
hello world
```

For the assembly version:

```
$ 5a helloraw.s
$ 5l helloraw.5 -o helloraw
$ ./helloraw
hello world
```

You can also use `ratrace`, the Plan 9 equivalent of `strace`, to see the syscalls executed by the program:

```
$ ratrace -c ./helloraw
...
Pwrite 9, "hello world\n", 12
...
Exits  0
```

[1] The `0LL` suffix makes the offset a 64-bit `long long`, because Plan 9's `pwrite` takes a 64-bit file offset, allowing files larger than 4 GiB.

This confirms that our small program generates exactly the two system calls we expected: `PWRITE` and `EXITS`. We will see in Chapters 6 and 7 how the kernel handles the `SWI` instruction, dispatches the syscall number through `systab`[57b], and executes the corresponding kernel function.

2.4 `minishell.c`

A slightly more interesting example is a minimal shell that illustrates several syscalls working together:

⟨minishell.c 36⟩≡
```
#include <u.h>
#include <libc.h>

void main() {
  char buf[256];
  char *args[2];
  int n;

  for(;;) {
    pwrite(1, "$ ", 2, 0LL);
    n = pread(0, buf, sizeof(buf)-1, 0LL);
    if(n <= 0)
      break;
    buf[n-1] = '\0';  /* strip newline */
    args[0] = buf;
    args[1] = nil;
    if(rfork(RFPROC|RFFDG) == 0) {
      // child
      exec(buf, args);
      exits("exec failed");
    }
    // else parent
    await(buf, sizeof(buf));
  }
  exits(nil);
}
```

This program reads commands from standard input (file descriptor `0`), forks a child with `rfork`, loads the requested program with `exec`, and waits for it to finish with `await`. Running it under `ratrace` reveals the full sequence of syscalls for a single command—a preview of the machinery this book will explain.

2.5 The Raspberry Pi machine (ARM)

The target hardware for this book is the Raspberry Pi Model B, a cheap hobbyist board designed for education. It is built around a Broadcom BCM2835 System-on-Chip (SoC) with an ARM1176JZF-S processor implementing the ARMv6 instruction set (32 bits). In the rest of this book, "Raspberry Pi" and "ARM" refer specifically to this board and this processor unless noted otherwise.

2.5.1 ARM CPU

The ARM is a RISC processor, far simpler than the x86 CISC architecture. This makes it a better target for a teaching book. ARM is also the most popular architecture by number of devices shipped—it powers most smartphones—so understanding it is useful beyond this book.

ARM is a 32 bits architecture, hence the constant definition below "bytes per word". A word is the natural unit of data for a processor—the size of a general-purpose register and typically the width of the data bus. The

term dates back to early computing where a "word" was the fundamental chunk of data a machine could process in one operation. On a 32-bit ARM, a word is 4 bytes.

⟨*constant* BY2WD*(arm)* 37a⟩≡ (1005)
```
#define BY2WD       4
```

BY2V stands for "bytes per vlong." A `vlong` ("very long") is Plan 9's name for a 64-bit integer. In Plan 9, `long` is always 32 bits regardless of the architecture (unlike `gcc` where `long` is 64 bits on 64-bit systems). So `vlong` (equivalent to C99's `long long`) is 8 bytes.

⟨*constant* BY2V*(arm)* 37b⟩≡ (1005)
```
#define BY2V        8           /* only used in xalloc.c */
```

Main registers, `Ureg`

The ARM has 16 general-purpose 32-bit registers (`R0`–`R15`), of which `R13` is the stack pointer (`SP`), `R14` the link register (return address), and `R15` the program counter (`PC`). The `Ureg` structure below captures the full register state of a user process. The kernel saves a `Ureg` on the stack every time it takes control from a user process (on a syscall, interrupt, or exception), and restores it when returning.

⟨*struct* Ureg*(arm)* 37c⟩≡ (888a)
```
//coupling: do not change the order! some assembly code assumes this order.
struct Ureg {
    ulong   r0;
    ulong   r1;
    ulong   r2;
    ulong   r3;
    ulong   r4;
    ulong   r5;
    ulong   r6;
    ulong   r7;
    ulong   r8;
    ulong   r9;
    ulong   r10;
    ulong   r11;
    ulong   r12;    /* sb */

    union {
        ulong   r13;
        ulong   sp;
    };
    union {
        ulong   r14;
        ulong   link;
    };

    ulong   type;   /* of exception */
    ulong   psr;

    ulong   pc; /* interrupted addr */ // r15
};
```

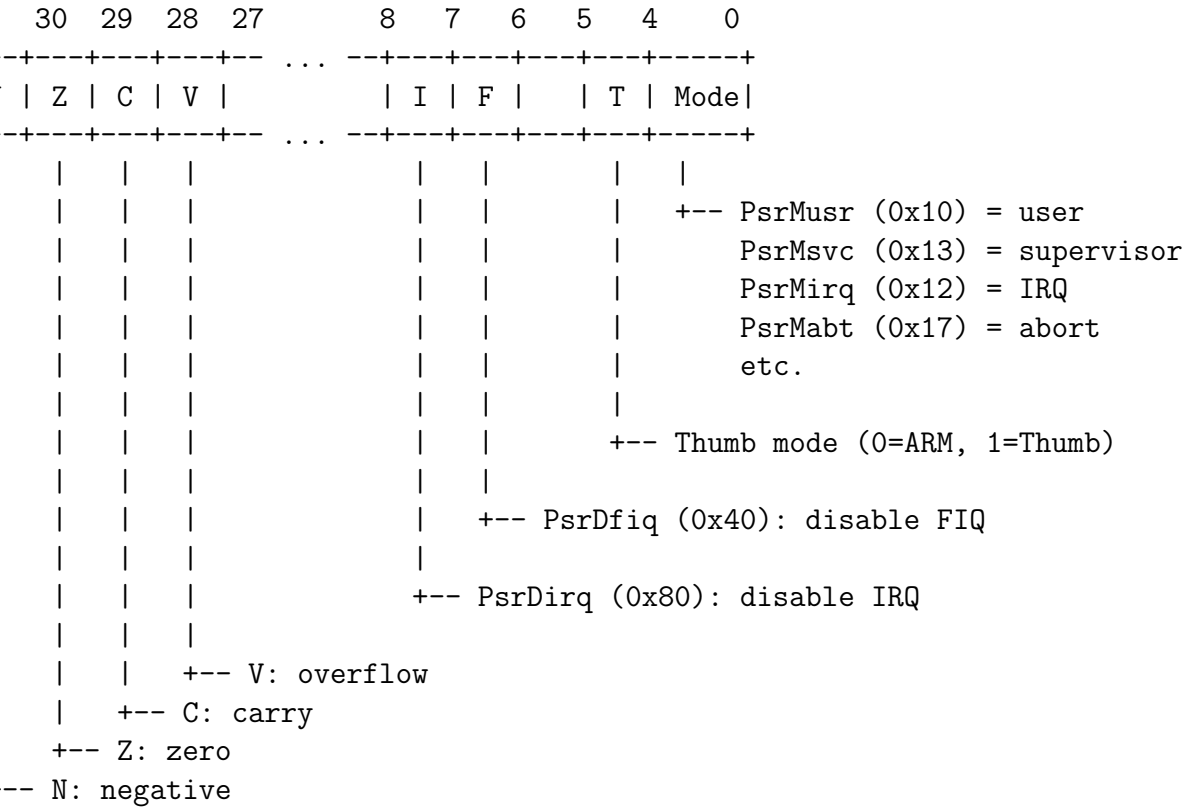
The last two fields, `type` and `psr`, are not general registers. `psr` holds the Program Status Register, which I cover in the next subsection. `type` records which exception pushed this `Ureg`—a data abort, a system call, an IRQ—and is what `arch__trap()` later switches on to pick a handler (Chapter 7).

Program Status Register (`PSR`)

In addition to the general-purpose registers, the ARM has a Program Status Register (PSR) (the `Ureg.psr` field above) that contains both the arithmetic flags (negative, zero, carry, overflow) and control bits (processor mode,

interrupt disable). Normal user programs never access the PSR directly, but the kernel must save and restore it (hence `Ureg.psr` above) because the flags are part of the process state. The control bits are critical too: they determine whether the processor is in user or supervisor mode, and whether interrupts are enabled.

The ARM actually has two copies of the PSR. The CPSR (Current Program Status Register) reflects the current state. The SPSR (Saved Program Status Register) is a shadow copy that the processor fills automatically when entering an exception: it saves the old CPSR so the handler can restore it when returning. Both have the same layout:



Processor modes

The low bits of the PSR encode the processor mode. The ARM supports several modes, but the kernel cares mainly about two: user mode (`PsrMusr`), where normal programs run with restricted privileges, and supervisor mode (`PsrMsvc`), where the kernel runs with full access to hardware. When an exception (e.g., a `SWI`) occurs, the processor automatically switches to supervisor mode; the kernel must switch back to user mode before returning to the user program. The switch back is not a regular store to `CPSR` but a dedicated atomic return-from-exception sequence: a single instruction that simultaneously restores `PC` from the saved link register and copies the banked `SPSR` back into `CPSR`, so that mode and flags are restored together with the return address. ARMv6 introduced a dedicated mnemonic for this, `RFE` (Return From Exception); older ARM code uses a block-load instruction with the "load PSR" bit set to achieve the same effect.

The mode field is 5 bits wide (bits 4–0), but Bit 4 is always set in the defined modes (it distinguishes "32-bit modes" from legacy 26-bit modes). Bits 3–0 select the specific mode.

Bit 5 (the T bit) selects Thumb vs. ARM instruction mode. Thumb is a compact 16-bit instruction encoding that ARM processors can use to reduce code size. The Plan 9 kernel does not use Thumb mode—all kernel and user code runs in ARM mode (32-bit instructions)—so we can ignore the T bit in this book.

⟨*type* `PsrMode` *(arm)* 38⟩≡ (888c)

```
// Last 5 bits of PSR
#define PsrMusr     0x00000010  /* user mode */
#define PsrMsvc     0x00000013  /* 'protected mode for OS' */
⟨PsrMode other cases(arm) 161a⟩
```

```
#define PsrMask     0x0000001F
```

The PSR also controls whether interrupts are enabled. Setting `PsrDirq` disables IRQ interrupts. The kernel uses this to protect *critical sections* where it must not be preempted—for instance, while manipulating the scheduler's run queue.

⟨*type* `PsrDisable` *(arm)* 39a⟩≡ (888c)
```
#define PsrDirq     0x00000080      /* disable IRQ interrupts */
⟨PsrDisable other cases(arm) 560d⟩
```

Exceptions

A note on terminology: the ARM uses the word exception as the general term for "any event that diverts the processor from normal execution". This includes interrupts (external hardware signals, called IRQ for Interrupt Request), software interrupts (the `SWI` instruction used for syscalls), and aborts (page faults, undefined instructions).

When an exception occurs, the ARM processor automatically saves the current CPSR into the SPSR, switches to the appropriate privileged mode, and jumps to a fixed address in the *exception vector table*. This table is normally at physical address 0, but the kernel configures the ARM (via the `CpChv` bit in the system control register we will see soon) to use "high vectors" at `0xFFFF0000`. This is more convenient both because as we will see soon it lives in the kernel's virtual address space, and because it frees physical address 0 for the null pointer trap (the first page of user virtual memory is left unmapped).

Note that `0xFFFF0000` is a virtual address, so once the MMU is enabled the hardware itself relies on the MMU to translate it to the physical memory containing the vector code. This is not a chicken-and-egg problem: the kernel sets up the page tables and copies the vector code to physical page 0 first (while still using physical addresses or identity-mapped memory during early boot), then enables the MMU and `CpChv` together. From that point on, any exception causes the ARM to fetch the handler from virtual `0xFFFF0000`, which the MMU translates to physical 0.

⟨*constant* `HVECTORS` *(arm)* 39b⟩≡ (888c)
```
#define HVECTORS    0xffff0000
```

2.5.2 ARM coprocessors

Beyond the main registers and the PSR, the ARM provides access to coprocessors that control hardware features such as the MMU, caches, and performance counters. Coprocessor registers are accessed via special instructions (`MRC` to read, `MCR` to write) and are only available in privileged mode. Normal user programs never touch them—they are the kernel's private interface to the hardware.

⟨*type* `Coprocessors` *(arm)* 39c⟩≡ (888c)
```
#define CpSC        15          /* System Control */
⟨Coprocessors other cases(arm) 518b⟩
```

System control coprocessor (`CpSC`)

The system control coprocessor (`CpSC`, coprocessor 15) is the most important for the kernel. Its registers control the MMU (`CpTTB` for the page table base, `CpTLB` for TLB management), report page faults (`CpFSR` for fault status, `CpFAR` for the faulting address), and configure miscellaneous features such as caches and the exception vector location.

⟨*type CpSC primary registers (arm)* 39d⟩≡ (888c)
```
#define CpCONTROL   1           /* miscellaneous control */
#define CpTTB       2           /* Translation Table Base(s) */
#define CpDAC       3           /* Domain Access Control */
```

```
#define CpFSR       5           /* Fault Status */
#define CpFAR       6           /* Fault Address */
#define CpTLB       8           /* TLB control */
#define CpSPM       15          /* system performance monitor (arm1176) */
⟨CpSC primary registers other cases(arm) 529a⟩
```

⟨*type CpCONTROL opcode2(arm)* 40a⟩≡ (888c)

```
/*
 * CpCONTROL op2 codes, op1==0, Crm==0.
 */
#define CpMainctl   0
⟨CpCONTROL.opcode2 other cases(arm) 522b⟩
```

These two bits are the ones the kernel flips when it wakes the MMU up. `CpCmmu` (the "M" bit) turns the MMU on: until it is set, every virtual address is used directly as a physical address. `CpChv` (the "V" bit) is the "high vectors" bit I already mentioned in the Exceptions subsection above: setting it moves the exception vector table from `0x00000000` to `0xFFFF0000`, so the kernel does not have to keep page 0 mapped. Unlike normal registers, coprocessor registers are not addressed by name. The main control register lives on coprocessor `CpSC` (15) at primary register `CpCONTROL`, secondary register 0, opcode2 `CpMainctl`, and is read or written only through the `MRC` / `MCR` coprocessor-move instructions. In Plan 9 ARM assembly this looks like:

```
MRC CpSC, 0, R1, C(CpCONTROL), C(0), CpMainctl  /* R1 := ctlreg */
MCR CpSC, 0, R1, C(CpCONTROL), C(0), CpMainctl  /* ctlreg := R1 */
```

The boot code will later read the register, OR in `CpCmmu|CpChv`, and write it back—one instruction away from having a virtual address space.

⟨*type CpCONTROL CpMainctl(arm)* 40b⟩≡ (888c)

```
/*
 * CpCONTROL: op1==0, CRm==0, op2==CpMainctl.
 * main control register.
 * cortex/armv7 has more ops and CRm values.
 */
#define CpCmmu      0x00000001  /* M: MMU enable */
#define CpChv       0x00002000  /* V: high vectors */
⟨CpCONTROL.CpMainctl other cases(arm) 529b⟩
```

Floating point coprocessors

The ARM has two coprocessors dedicated to floating-point arithmetic: one for single precision and one for double precision. Floating point adds considerable complexity to a kernel because the floating-point registers must be saved and restored on context switches, and some ARM chips lack hardware floating-point entirely (requiring software emulation). These issues are discussed in the Advanced Topics chapter.

2.5.3 ARM MMU

The ARM includes a Memory Management Unit (MMU) that translates virtual addresses to physical addresses using *page tables*. The page table base address is stored in the `CpTTB` coprocessor register—this is a physical address, since the MMU itself needs to walk the page table before any virtual-to-physical translation is available. When a translation fails, the MMU generates a fault and records the faulting address in `CpFAR` and the reason in `CpFSR`.

The kernel uses the MMU to give each process its own virtual address space and to protect kernel memory from user access. More details are in Chapter 8.

The ARM MMU supports two granularities of mapping. An L1 (first-level) table entry can either map a full 1 MiB *section* directly to a physical address, or point to an L2 (second-level) table that maps individual 4 KiB *pages*:

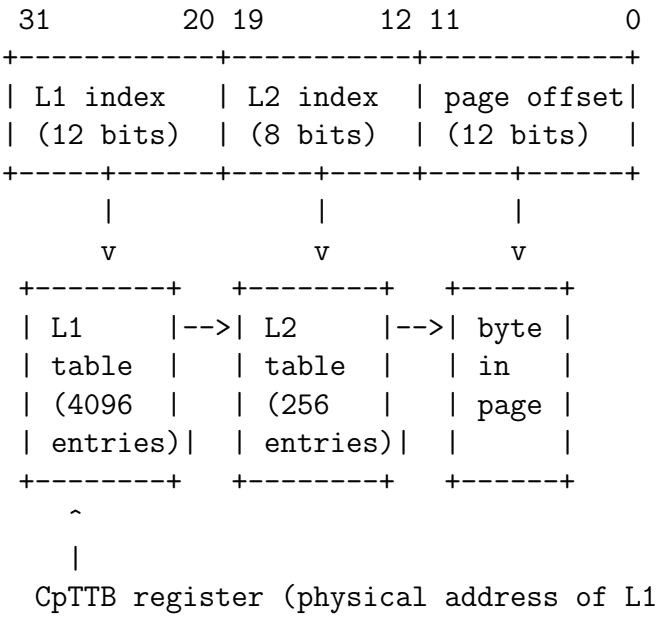
```
L1 entry types:
  Section  --> maps 1 MiB directly (no L2 table needed)
  Coarse   --> points to an L2 table (256 entries x 4 KiB = 1 MiB)
```

Sections are efficient for large contiguous mappings (the kernel maps all of RAM this way), while pages are needed for fine-grained control (individual user pages, the exception vector page at `HVECTORS`).

The basic unit of memory managed at the finest level is a page, which on ARM (and on x86) is 4 KiB:

⟨*constant* `BY2PG` *(arm)* 41a⟩≡ (1005)
```
#define BY2PG       (4*KiB)         /* bytes per page */
```

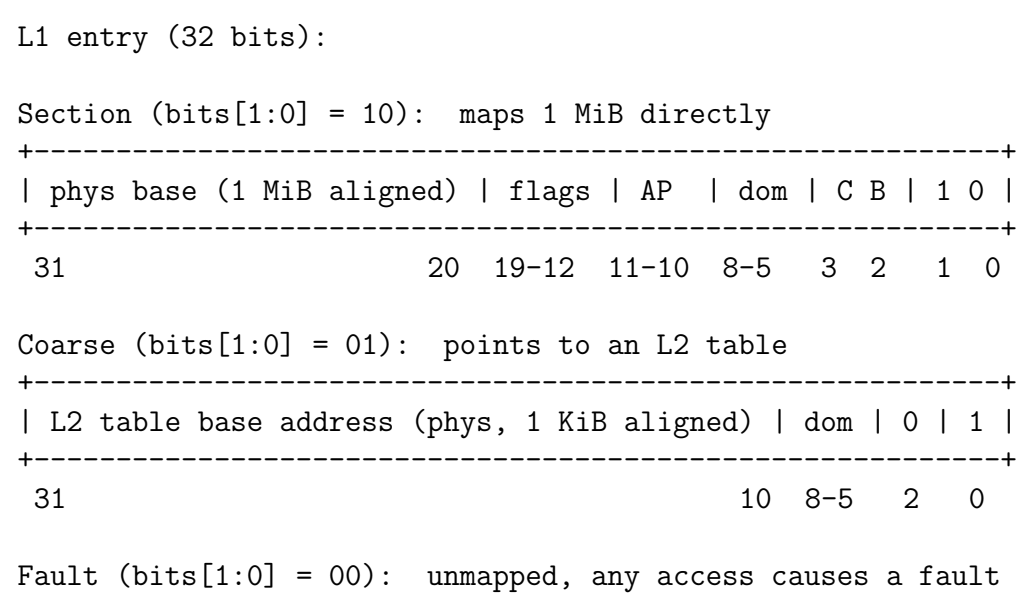
When using pages (L1 entry of type Coarse), the MMU translates virtual addresses using a two-level walk. The virtual address is split into three fields:



⟨*constant* `PGSHIFT` *(arm)* 41b⟩≡ (1005)
```
#define PGSHIFT     12          /* log(BY2PG) */
```

When using sections (L1 entry of type Section), the L2 step is skipped entirely: the L1 entry contains the physical base address of a 1 MiB region directly, and bits 19–0 of the virtual address become the offset within that region.

The lowest 2 bits of each L1 entry determine its type:



```
+-------------------------------------------------------+
|                       ignored                         | 0 0 |
+-------------------------------------------------------+
```

As I mentioned before, the `Section` type is used for the kernel's RAM mapping (512 entries covering 512 MiB) and the I/O register mapping. The `Coarse` type is used for the exception vectors at `HVECTORS` and for user-process pages, where fine-grained 4 KiB mapping is needed.

For completeness, here is the matching L2 entry format—what each of the 256 entries in an L2 table looks like when pointed to by an L1 `Coarse`:

```
L2 entry (32 bits):

Small (bits[1:0] = 10):  maps 4 KiB directly
+-------------------------------------------------------+
| phys base (4 KiB aligned) | AP3 AP2 AP1 AP0 | C B | 1 0 |
+-------------------------------------------------------+
 31                         12  11-10 9-8 7-6 5-4  3 2   1 0

Large (bits[1:0] = 01):  maps 64 KiB directly
Fault (bits[1:0] = 00):  unmapped
```

The one new wrinkle compared to the L1 format is that `AP` appears four times—an ARMv4 sub-page legacy in which each 4 KiB page could be split into four independently-protected 1 KiB regions. Plan 9 does not use that granularity: every sub-field ends up with the same value.

A few of the bit fields in these entries are worth knowing about now. `AP` (Access Permission, bits 11–10) controls whether the page is readable/writable by user code or only by the kernel—this is how user/kernel memory protection is enforced.

`dom` (domain, bits 8–5) is a coarse-grained permission layer above `AP` that Plan 9 does not really use: it pins every page to domain 0 in "client" mode, which just means "check the `AP` bits normally".[2]

The `C` and `B` bits (bits 3 and 2) control caching: `C`=1 enables the data cache, `B`=1 enables write buffering. Kernel RAM is mapped with both set (write-back cached); I/O registers are mapped with both clear (uncached, so reads/writes go directly to hardware).

2.5.4 Raspberry Pi Input/Output devices

The Raspberry Pi is more than just a CPU. The BCM2835 SoC includes a GPU with framebuffer, a USB controller (for keyboard and mouse), an SD card controller (for storage), GPIO pins, timers, and an interrupt controller. The kernel must initialize and manage all of these devices.

Memory-mapped IO

On ARM, devices are accessed through *memory-mapped I/O*: each device exposes its control registers at fixed physical addresses, and the kernel reads and writes them using ordinary load/store instructions (`MOVW` in Asm5, which is really `LDR`/`STR` under the hood, see the Assembler book [Pad15a] and Machine book [?]). This is simpler than the x86 approach, which uses special `IN`/`OUT` instructions for I/O ports.

[2]For the interested reader: the ARM supports 16 memory domains, each globally configured via the domain access register as "client" (check `AP` bits), "manager" (ignore `AP`, always allow), or "no access" (always fault). This was designed as a cheap way to enable or disable large groups of pages without walking page tables—useful, for example, for fast context switches between a few preconfigured protection domains. Modern operating systems, Plan 9 included, ignore it and rely entirely on `AP`.

The kernel maps the physical I/O region (starting at `0x20000000` on the Raspberry Pi 1) to a virtual address `VIRTIO`, so that all device registers are accessed as offsets from `VIRTIO`:

⟨*constant* `VIRTIO`*(arm)* 43a⟩≡ (1005)
```
#define VIRTIO      0x7E000000      /* i/o registers */
```

The remaining constants define the addresses of specific device register blocks, all at fixed offsets from `VIRTIO`:

⟨*constant* `SYSTIMERS`*(arm)* 43b⟩≡ (1011b)
```
#define SYSTIMERS (VIRTIO+0x3000)
```

⟨*constant* `ARMTIMER`*(arm)* 43c⟩≡ (1011b)
```
#define ARMTIMER (VIRTIO+0xB400)
```

⟨*constant* `INTREGS`*(arm)* 43d⟩≡ (1002c)
```
#define INTREGS     (VIRTIO+0xB200)
```

⟨*constant* `DMAREGS`*(arm)* 43e⟩≡ (960)
```
#define DMAREGS (VIRTIO+0x7000)
```

⟨*constant* `POWERREGS`*(arm)* 43f⟩≡ (1002a)
```
#define POWERREGS   (VIRTIO+0x100000)
```

⟨*constant* `AUXREGS`*(arm)* 43g⟩≡ (969b)
```
#define AUXREGS     (VIRTIO+0x215000)
```

Output device: VideoCore GPU

The Raspberry Pi's BCM2835 contains a VideoCore IV GPU that is actually more powerful than the ARM CPU itself.[3]

The GPU manages the screen and provides a *framebuffer*: a region of memory where each pixel is represented by a few bytes. The kernel writes pixels to this framebuffer, and the GPU continuously scans it to produce the video signal.

Communication between the ARM CPU and the GPU goes through a *mailbox* mechanism—a set of hardware registers at a fixed address where the ARM can send requests and receive responses:

⟨*constant* `MAILBOX`*(arm)* 43h⟩≡ (897b)
```
#define MAILBOX     (VIRTIO+0xB880)
```

To set up the framebuffer, the kernel fills an `Fbinfo` structure with the desired resolution and pixel depth, sends its physical address to the GPU via the mailbox, and the GPU responds by filling in the `pitch`, `base`, and `screensize` fields. The `base` field is the physical address of the framebuffer memory that the GPU has allocated. Note how the field order in the C struct must exactly match the hardware's expected memory layout:

⟨*struct* `Fbinfo`*(arm)* 43i⟩≡ (897b)
```
// Framebuffer
// The order matters! the fields match the memory-mapped external registers.
struct Fbinfo {
    u32int  xres;
    u32int  yres;
    u32int  xresvirtual;
    u32int  yresvirtual;
    u32int  pitch;          /* returned by gpu */
    u32int  bpp;
    u32int  xoffset;
```

[3] In two senses. First, raw throughput: VideoCore IV is a dual-core SIMD vector processor rated at roughly 24 GFLOPS, while the original Pi's single-core ARM11 runs below 1 GFLOPS. Second, and more surprisingly, the GPU is the *boot* processor: on power-up the VideoCore comes up first, reads its own firmware from the SD card, initializes RAM and clocks, loads the ARM kernel image into memory, and only then releases the ARM from reset. From the ARM's point of view the GPU is the "real" CPU and the ARM is a coprocessor.

```
    u32int  yoffset;
    u32int  base;           /* returned by gpu */
    u32int  screensize;     /* returned by gpu */
};
```

The kernel then maps this GPU-allocated physical memory to the virtual address `FRAMEBUFFER` so it can write pixels directly. See the Graphics book [Pad16d] for the full story of how the kernel system uses this framebuffer.

⟨*constant* `FRAMEBUFFER` *(arm)* 44a⟩≡ (1005)
```
#define FRAMEBUFFER 0xC0000000      /* video framebuffer */
```

Input devices: USB keyboard and mouse

Keyboard and mouse reach the kernel through the Raspberry Pi's on-board USB controller (a Synopsys DesignWare OTG). The controller is programmed by reading and writing registers at `USBREGS`:

⟨*constant* `USBREGS` *(arm)* 44b⟩≡ (953c)
```
#define  USBREGS (VIRTIO+0x980000)
```

Storage device: SD card

Persistent storage is an SD card attached to the Broadcom eMMC (embedded MultiMediaCard) controller. The kernel programs the controller through its memory-mapped registers at `EMMCREGS`:

⟨*constant* `EMMCREGS` *(arm)* 44c⟩≡ (989b)
```
#define EMMCREGS    (VIRTIO+0x300000)
```

Other devices

The remaining bank, `GPIOREGS`, exposes the Raspberry Pi's general-purpose I/O pins. These pins drive the UART console, activity LEDs, the I2C/SPI buses, and the 40-pin expansion header that makes the Raspberry Pi useful as a hobbyist platform:

⟨*constant* `GPIOREGS` *(arm)* 44d⟩≡ (969b)
```
#define GPIOREGS    (VIRTIO+0x200000)
```

Summary

The following diagram summarizes the I/O register layout as offsets from `VIRTIO` (`0x7E000000`):

```
VIRTIO + IOSIZE    (0x7E000000 + 16 MiB)
                   +-----------------+
                   | MAILBOX (GPU)   |
VIRTIO + 0xB880    +-----------------+
                   | ARMTIMER        |
VIRTIO + 0xB400    +-----------------+
                   | INTREGS         |
VIRTIO + 0xB200    +-----------------+
                   | USBREGS         |
VIRTIO + 0x980000  +-----------------+
                   | EMMCREGS (SD)   |
VIRTIO + 0x300000  +-----------------+
                   | AUXREGS (UART)  |
VIRTIO + 0x215000  +-----------------+
```

```
                    | GPIOREGS        |
  VIRTIO + 0x200000  +-----------------+
                    | POWERREGS       |
  VIRTIO + 0x100000  +-----------------+
                    | DMAREGS         |
  VIRTIO + 0x007000  +-----------------+
                    | SYSTIMERS       |
  VIRTIO + 0x003000  +-----------------+
                    |   (base)        |
  VIRTIO + 0x000000  +-----------------+
```

The total I/O space is `IOSIZE` (16 MiB), though only a small fraction of that range is actually used by device registers. The rest is simply unmapped.

⟨*constant* `IOSIZE`*(arm)* 45⟩≡ (1005)

```
#define IOSIZE      (16*MiB)
```

This is the first hardware memory-layout diagram in the book. It follows the same convention as the user-process address-space diagram shown back in Section 2.1.2, and as every memory-layout diagram that follows: low addresses at the bottom, high addresses at the top, so memory grows upward on the page.

2.6 C and assembly (ARM)

The kernel is written almost entirely in C, compiled in the case of `9pi` by the Plan 9 ARM C compiler `5c` (see the Compiler book [Pad16b]) and linked by the Plan 9 ARM linker `5l` (see the Linker book [Pad15c]). Some parts are written in assembly, assembled in the case of `9pi` by the Plan 9 ARM assembler `5a` (see the Assembler book [Pad15a]). A small amount of assembly is necessary for operations that C cannot express: saving and restoring registers, switching between processor modes, manipulating coprocessor registers, and performing the initial boot sequence.

The calling convention used by `5c` is straightforward: parameters are passed on the stack, the return value goes in `R0`, and all registers are *caller-saved* (see the Compiler book [Pad16b]).

Here is the stack layout when a C function `foo(a, b, c)` calls `bar(x, y)`:

```
          +-------------+
          |  c (arg 3)  |  12(FP)  for foo
          +-------------+
          |  b (arg 2)  |   8(FP)
          +-------------+
          |  a (arg 1)  |   4(FP)
          +-------------+
          | return addr |   0(FP)  = saved R14
  SP ---> +-------------+
          |  y (arg 2)  |   8(SP)  for bar
          +-------------+
          |  x (arg 1)  |   4(SP)
          +-------------+
          | (R14 save)  |   0(SP)
          +-------------+
```

One subtlety: in the Plan 9 toolchain (`5a`, `5c`, `5l`), `FP` is a *virtual* register—no real ARM register is dedicated to it, and the linker rewrites each `n(FP)` into the appropriate `SP`+offset at link time, based on the function's local-variable layout. The diagram above shows the conceptual view the programmer sees through `0(FP)`, `4(FP)`...;

the actual SP at any point inside the callee sits lower than drawn, below any locals the callee has reserved. What matters is the relative ordering: saved link register, then arguments, then locals.

For syscalls, the convention is different: the syscall number goes in R0 (not on the stack), and the SWI instruction traps into the kernel. The kernel then reads the user's stack to find the syscall arguments.

Concretely, the user stack just before the SWI $0 in the pwrite(1, \&hello, 12, 0LL) call from helloraw.s (Section 2.3) looks like this:

```
          +-------------+
          | 0LL (high)  |  20(R13)   offset, high word
          +-------------+
          | 0LL (low)   |  16(R13)   offset, low word
          +-------------+
          |  n = 12     |  12(R13)   arg 3
          +-------------+
          | &hello      |   8(R13)   arg 2
          +-------------+
          | fd = 1      |   4(R13)   arg 1
          +-------------+
R13 -->   |   (unused)  |   0(R13)
          +-------------+

R0 = 9  (= PWRITE)
```

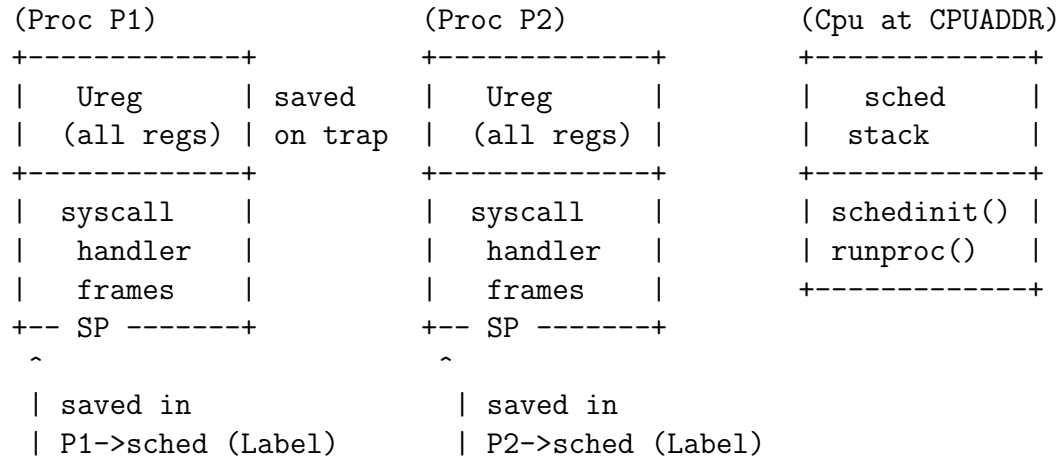
The slot at 0(R13) is left unused because the Plan 9 assembler reserves it for a future saved link register (the function prologue would store LR there on entry, but _main does not call another C function—it only traps via SWI). The kernel's syscall() handler, seeing R0 = 9, dispatches to syspwrite and passes it a pointer into this stack so it can fetch arg[0], arg[1], etc.

The caller-saved convention of 5c simplifies the kernel's context-switch code, because the Label[226d] structure (shown later) only needs to store SP (the stack pointer) and PC (the program counter)—the caller has already saved any registers it cares about. Note that this Label (just SP+PC) is used for voluntary context switches between processes: in Plan 9, a user process that has trapped into the kernel (for a syscall or a page fault) can block and yield mid-syscall, and the scheduler will then pick a different process to run.[4] When the kernel takes control from a user process (via a syscall or interrupt), it saves all registers into a Ureg on the kernel stack—because the user did not follow any calling convention that the kernel could rely on.

The following diagram shows where all of these structures live for two processes P1 and P2 running on CPU 0. Each process owns two stacks: its *user stack* at the top of user virtual memory, and its *kernel stack* (part of the Proc structure, in kernel memory). CPU 0 also owns a per-CPU *scheduler stack* inside the Cpu structure at CPUADDR, used only while the scheduler itself is running between processes.

```
   Process P1              Process P2           CPU 0

+-------------+         +-------------+      +-------------+
|  user       |         |  user       |      |             |
|  stack      |         |  stack      |      |             |
|  (USTKTOP-) |         |  (USTKTOP-) |      |             |
+-------------+         +-------------+      +-------------+
       .                       .                    .
       .                       .                    .
```

[4] The same Label mechanism is also used for kprocs, processes that live entirely in kernel mode—we will meet them in Chapter 6.



When `P1` voluntarily yields via `sched()`, its current `SP`/`PC` are stored into `P1->sched` (a `Label`), the CPU switches to the per-CPU scheduler stack, picks `P2`, and restores `SP`/`PC` from `P2->sched` to resume `P2` exactly where it previously yielded. Involuntary entry from user mode is different: the hardware pushes a `Ureg` at the top of the trapped process's kernel stack (because the user side did not follow any C calling convention and all registers must be preserved), then calls into C code.

One utility function deserves a mention here because it appears throughout the kernel: `getcallerpc` returns the `PC` from which the current function was called, i.e., one frame up the call stack. In GCC land this would be `__builtin_return_address(0)`; in the Plan 9 toolchain it is a three-instruction assembly routine that exploits the calling convention shown above. Because `5c` stores the caller's return address at `0(R13)` (see the stack diagram earlier in this section), `getcallerpc` just loads that word into `R0` and returns. The `$-4` frame-size directive tells the assembler to emit no prologue—no stack adjustment, no `LR` save—so the function runs with the caller's `R13`, and `0(R13)` therefore holds the caller's saved return address (the `PC` of the function that called the caller). Plan 9 uses `getcallerpc` all over the place: locking code records it to identify who last took a lock (so `unlock: not locked: pc \%\#p` panics can be tracked back to the offender), the allocator tags blocks with the caller's `PC` for leak debugging, and several other subsystems do similar things.

⟨*function* `getcallerpc` *(arm)* 47a⟩≡ (1010b)
```
TEXT getcallerpc(SB), 1, $-4
    MOVW    0(R13), R0
    RET
```

Kernel code makes heavy use of bit manipulation—rounding addresses to page boundaries, extracting bit fields from hardware registers, etc. The following macros are used throughout the kernel:

⟨*portdat.h macros* 47b⟩≡ (703b) 47c▷
```
#define ROUND(s, sz)  (((s)+(sz-1)) & ~(sz-1))
```

⟨*portdat.h macros* 47c⟩+≡ (703b) ◁47b 47d▷
```
#define MASK(v) ((1UL << (v)) - 1)  /* mask 'v' bits wide */
```

⟨*portdat.h macros* 47d⟩+≡ (703b) ◁47c 47e▷
```
#define MIN(a, b) ((a) < (b)? (a): (b))
```

⟨*portdat.h macros* 47e⟩+≡ (703b) ◁47d 47f▷
```
// used in bcm/mmu.c
#define FEXT(d, o, w) (((d)>>(o)) & ((1<<(w))-1))
```

⟨*portdat.h macros* 47f⟩+≡ (703b) ◁47e 120d▷
```
#define HOWMANY(x, y) (((x)+((y)-1))/(y))
#define ROUNDUP(x, y) (HOWMANY((x), (y))*(y)) /* ceiling */
```

Here are a few convenient constants used throughout the code.

⟨*constant* `KiB` *(arm)* 47g⟩≡ (1005)
```
#define KiB     1024u           /* Kibi 0x0000000000000400 */
```

⟨*constant* `MiB`*(arm)* 48a⟩≡ (1005)

```
#define MiB     1048576u        /* Mebi 0x0000000000100000 */
```

⟨*constant* `GiB`*(arm)* 48b⟩≡ (1005)

```
#define GiB     1073741824u     /* Gibi 0000000000040000000 */
```

2.7 Code organization

The `9` kernel source is split into many subsystem directories. Within each subsystem the portable C code sits at the top level; architecture-specific code lives in `arm/` (for the Raspberry Pi port `9pi`) and `386/` (for the x86 port) subdirectories; and userspace utilities that exercise the kernel interfaces live under `user/`. Those include small helper daemons (`disk`, `usbd`, `swap`) and full 9P file servers such as `dossrv`, which mounts a DOS/FAT disk image and serves it as a plain filesystem—without any FAT code ever entering the kernel.

Table 2.1 gives a bird's-eye view of the whole kernel tree: one row per subdirectory, the chapter(s) that discuss that code, a few representative source files, and the LOC split into portable, ARM-specific, and userspace parts. Since this book targets the Plan 9 kernel for the Raspberry Pi (`9pi`), the x86-specific code under every `386/` subdirectory is omitted; altogether that amounts to roughly 28K additional lines. Some subsystems span more than one chapter—the `memory/` code, for example, underlies both the Memory Management and the Virtual Memory chapters. The graphics stack (`devices/screen/` with `devdraw.c` and friends) and the networking stack (`network/`) are covered in the companion Graphics book [Pad16d] and Network book [Pad16e] volumes; I list them here for completeness.

Directory	Chapters	Representative files	Port	ARM	User	Total
`core/`	3	`error.c`, `portdat.c`	632	325	0	957
`concurrency/`	4	`qlock.c`, `ref.c`	866	252	0	1118
`memory/`	5, 8	`alloc.c`, `page.c`, `fault.c`	4020	1026	66	5112
`processes/`	6, 9, 18	`edf.c`, `pgrp.c`, `sysproc.c`	5335	905	0	6240
`interrupts/`	7	arch-specific trap entry	9	1109	0	1118
`syscalls/`	7	`systab.c`, `syscallfmt.c`	567	397	186	1150
`time/`	10	`alarm.c`, `portclock.c`	951	359	0	1310
`init/`	11	`rebootcmd.c`, arch `main.c`	196	1895	661	2752
`files/`	12, 13, 14	`chan.c`, `dev.c`, `allocb.c`	7001	0	195	7196
`devices/`	15	`keyboard/`, `mouse/`, `storage/`	9322	720	6152	16194
`devices/screen/`	Graphics book [Pad16d]	`devdraw.c`, `swconsole.c`	3755	254	4956	8965
`buses/`	15	`devuart.c`, USB, I2C, SPI	1093	4012	16767	21872
`console/`	15	`devcons.c`, `print.c`	1561	0	0	1561
`filesystems/`	17	`devfs.c`	1299	0	5100	6399
`network/`	Network book [Pad16e]	`netif.c`, `etherif.h`	18532	582	0	19114
`arch/`	(arch headers)	arch-specific constants	0	1121	0	1121
`lib/`, `misc/`	(utilities)	small helpers	686	0	0	686
`security/`	(security stubs)	`auth.c`	164	0	0	164
`conf/`	(build config)	`master`, `mkdevc`, `mksystab`	697	348	0	1045
Total			56686	13305	34083	104074

Table 2.1: Kernel subdirectories, representative files, and LOC split across portable, ARM-specific, and userspace parts. The x86-specific code under each `386/` subdirectory (roughly 27.6K LOC in total) is omitted, as this book covers it separately in its x86 chapters.

2.7.1 Portable versus arch-specific code

To make the boundary visible in the code, arch-specific functions called from portable code are prefixed with `arch_` (e.g., `arch_coherence()`), and arch-specific structures embedded in portable ones with `Arch_` (e.g., `Arch_Cpu`, `Arch_Proc`).

2.7.2 Main headers

Every kernel source file starts with the same set of includes:

⟨*kernel basic includes* 49a⟩≡ (1012d 1011b 1008 1006b 1002 999b 994 991c 989b 981a 977a 976 972 969b 960 953c 933 897b 892c 886a 88

```
#include <u.h>
#include "../port/lib.h"
#include "../port/error.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"
```

The three short filenames follow an old Plan 9 kernel convention: `mem.h` holds memory-related constants (the virtual-address map, page sizes, section boundaries); `dat.h` holds the data-structure and type definitions; `fns.h` holds the function (fns) declarations. The next subsection looks at `dat.h` and `fns.h` more closely.

2.7.3 `dat.h` and `fns.h`

The master header `dat.h` pulls in all data-structure definitions. In the original Plan 9 source, `dat.h` was a single long file and all portable code lived in a flat `port/` directory. I split it into separate `portdat_xxx.h` headers organized by subsystem (concurrency, memory, files, etc.) and created matching subdirectories. Reading the include list below is a good way to see the overall structure of the kernel:

⟨*dat.h includes(arm)* 49b⟩≡ (973a)

```
#include "dat_core.h"                    // Arch_Cpu
#include "../port/portdat_core.h"        // Cpu!!, Conf, Label
#include "../port/portdat_concurrency.h" // Lock
#include "dat_memory.h"                  // Arch_Proc
#include "../port/portdat_memory.h"      // Page, Pagetable, Segment, KImage
#include "../port/portdat_files.h"       // Chan!! Dev!!
#include "dat_time.h"                    // Arch_HZ
#include "../port/portdat_time.h"        // Timer
#include "dat_processes.h"               // Arch_FPsave
#include "../port/portdat_processes.h"   // Proc!!!
#include "../port/portdat_console.h"     // keyboard queue, consdevtab
#include "../port/portdat_devices.h"     // DevConf, DevPort (not that used)
#include "../port/portdat_misc.h"        // Cmd
#include "../port/portdat_buses.h"       // Uart
#include "dat_arch.h"                    // Soc
```

⟨*fns.h includes(arm)* 49c⟩≡ (974)

```
#include "../port/portfns_core.h"
#include "../port/portfns_concurrency.h"
#include "../port/portfns_memory.h"
#include "../port/portfns_processes.h"
#include "../port/portfns_files.h"
#include "../port/portfns_time.h"
#include "../port/portfns_interrupts.h"
#include "../port/portfns_console.h"
#include "../port/portfns_devices.h"
#include "../port/portfns_buses.h"
#include "../port/portfns_security.h"
#include "../port/portfns_network.h"
```

```
#include "../port/portfns_syscalls.h"
#include "../port/portfns_misc.h"
#include "../port/portfns_init.h"
```

2.8 Software architecture

Unlike a compiler or an assembler, a kernel is not a pipeline that transforms an input into an output. It is a *reactive* program: while user programs run, the kernel is dormant. It only wakes up when an *event* occurs—a syscall, an interrupt, a timer tick—handles it, and then lets the user program resume (see the historical aside in Section 2.1.5 for more on this).

The Plan 9 kernel is *monolithic*—device drivers, memory management, and scheduling all run in kernel space. But thanks to the 9P protocol, much functionality that would live in the kernel on other systems is pushed out to *user-space file servers* (e.g., the window system `rio`, the authentication agent `factotum`, the filesystems `fossil` and `dossrv`). This borrows a key idea from *microkernel* designs: keep the kernel small and move services to user space. In fact, the entire `9pi` kernel is about 70 000 lines of C and ARM assembly—remarkably small for a complete operating system (compare Linux's architecture-independent core at 400k+ LOC, graphics and network drivers excluded).

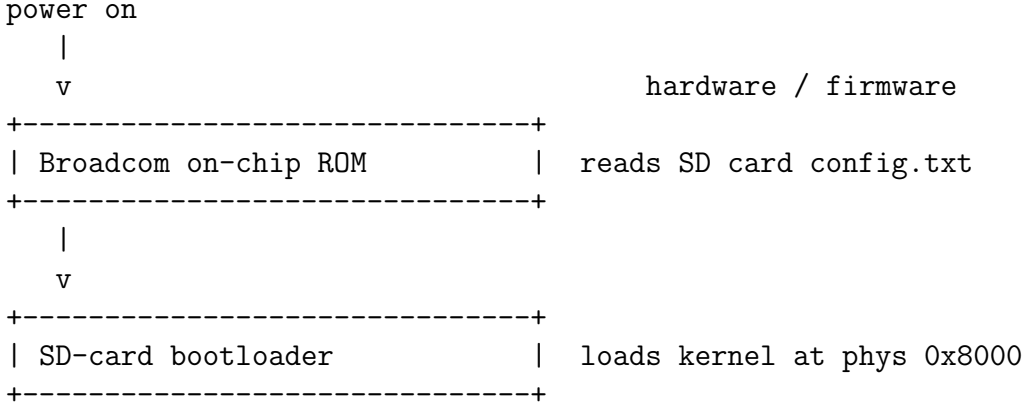
The kernel's execution can be understood along three axes. First, the boot sequence: how the kernel starts from a bare machine and ends up with a running user shell. Second, the memory layout: how physical memory is divided between the kernel, user processes, devices, and the framebuffer. Third, the steady-state operation: how syscalls flow from user space into the kernel and back, how interrupts propagate from hardware to sleeping processes, and how the scheduler picks which process runs next. The following subsections give an overview of the first two; the normal operation (once the system is up and running) is the subject of the rest of the book.

2.8.1 Booting overview (ARM)

On the Raspberry Pi, the Broadcom firmware (closed source, stored in ROM) loads a *bootloader* from the SD card, which in turn loads the kernel binary at physical address `0x8000`. The bootloader reads a configuration file (`config.txt`) from the SD card's boot partition, which specifies the path to the kernel binary. The details of this early firmware stage are specific to the Broadcom SoC and not very interesting; what matters is that all architectures share the same idea of *bootstrap code* loading the kernel into memory at a known address.

Once the ARM starts executing the kernel, the boot proceeds in three phases: (1) hardware detection, populating the `Conf`[62a] structure with the amount of physical memory and other parameters; (2) initialization of the kernel environment (memory allocators, process table, device drivers); and (3) creation of the first user process, which runs `/boot/boot`—the program responsible for mounting the root file system, setting up the user environment, and eventually launching `init` and then `rc` (the shell). See Chapter 11 for the full code.

Putting everything together, here is the full boot flow from power-on to shell prompt:



```
   |
   v                                  kernel
+------------------------------+
| 1. hardware detection        |  populate Conf (RAM, #CPUs, ...)
+------------------------------+
   |
   v
+------------------------------+
| 2. environment init          |  allocators, MMU, drivers
+------------------------------+
   |
   v
+------------------------------+
| 3. create first user process |  Proc[0] execs /boot/boot
+------------------------------+
   |
   v                                  user space
/boot/boot    mounts root file system, spawns init
   |
   v
init          sets up environment, spawns rc
   |
   v
rc            shell prompt
```

Chapter 11 covers the kernel-side phases in full detail; the userspace end of the flow is in Chapter 21.

2.8.2 Memory overview (ARM)

Understanding the kernel's memory layout requires thinking along several dimensions: *physical* vs. *virtual* addresses, *kernel* vs. *user* memory, and *static* vs. *dynamic* allocation.

On the Raspberry Pi, the physical memory starts at address 0.[5] As mentioned before the kernel binary is loaded at physical `0x8000`[6] and as we will see soon the corresponding virtual memory address will be `KTZERO`.

⟨*constant* `KTZERO` *(arm)* 51a⟩≡ (1005)
```
#define KTZERO      (KZERO+0x8000)      /* kernel text start */
```

The following constants define the virtual address space layout. `KZERO` at `0x80000000` (2 GiB) is the boundary: below is user space, above is kernel space.

⟨*constant* `KSEG0` *(arm)* 51b⟩≡ (1005)
```
#define KSEG0       0x80000000      /* kernel segment */
```

⟨*constant* `KZERO` *(arm)* 51c⟩≡ (1005)
```
#define KZERO       KSEG0           /* kernel address space */
```

[5] On some architectures, physical memory does not start at 0—for example, on many x86 systems the first megabyte is reserved for legacy BIOS structures. But on the Raspberry Pi's ARM SoC, physical RAM genuinely begins at address 0.

[6] Why `0x8000` and not 0? Address 0 is reserved for the ARM exception vector table. But the vectors only occupy a few dozen bytes, so why not `0x1000`? The `0x8000` address is a Broadcom firmware convention—the firmware maybe uses the memory below `0x8000` for its own data structures during boot. As we will see later the kernel in turn reuses some of this space (e.g., `CPUADDR` at `0x2000`) for early kernel structures like the first `Cpu`.

Everything below KZERO (0x80000000) in the virtual address space belongs to user processes; everything at or above KZERO belongs to the kernel. The kernel's virtual addresses are simply the physical addresses plus KZERO. Concretely, the Raspberry Pi bootloader drops the kernel binary at physical 0x8000, but the linker has linked it at virtual KTZERO (= KZERO + 0x8000) via a -T flag in the mkfile. Early boot code—still running with the MMU off—sets up the page tables so that KTZERO maps to physical 0x8000, then enables the MMU; from that point on, branches and calls resolve to the linked virtual addresses.

On the user side, the address space starts at UZERO (0). The first page is left unmapped so that null pointer dereferences cause a fault. User code starts at UTZERO ("User Text Zero"—one page above zero, the starting address of the user text segment), and the user stack grows downward from USTKTOP:

⟨*constant* UZERO*(arm)* 52a⟩≡ (1005)
```
#define UZERO       0           /* user segment */
```

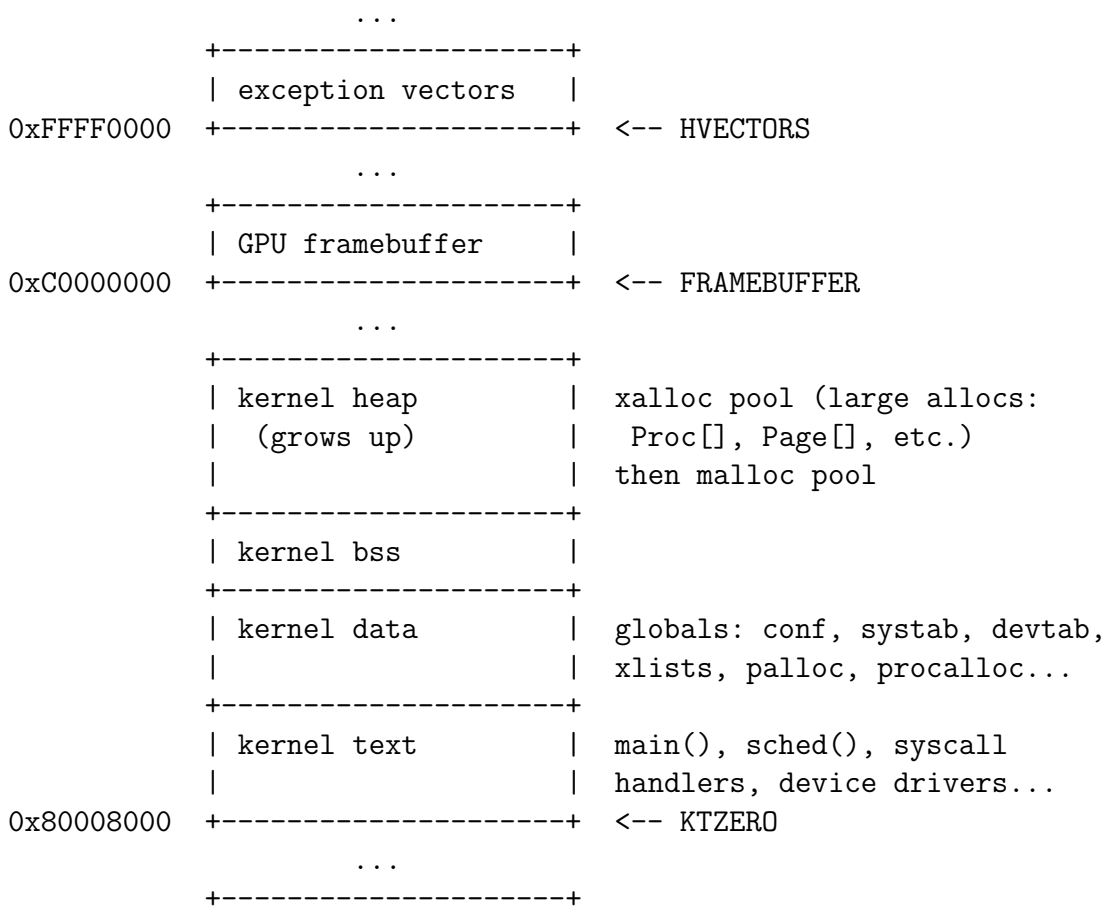
⟨*constant* UTZERO*(arm)* 52b⟩≡ (1005)
```
#define UTZERO      (UZERO+BY2PG)       /* user text start */
```

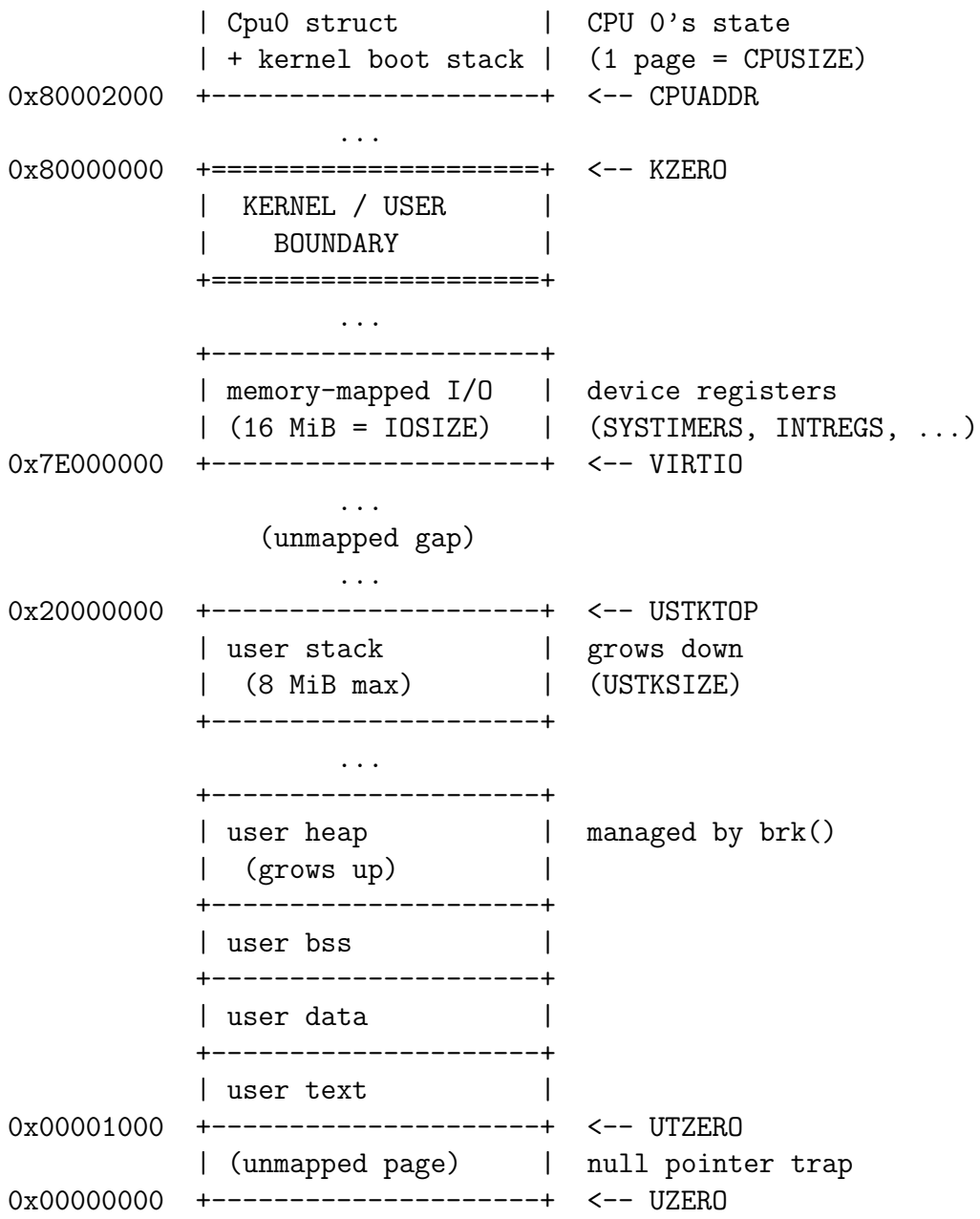
⟨*constant* USTKTOP*(arm)* 52c⟩≡ (1005)
```
#define USTKTOP     0x20000000      /* user segment end +1 */
```

⟨*constant* USTKSIZE*(arm)* 52d⟩≡ (1005)
```
#define USTKSIZE    (8*1024*1024)       /* user stack size */
```

Virtual address space (ARM)

The following diagram shows the full *virtual* address space (same convention as in Section 2.1.2: low addresses at the bottom, high addresses at the top). The kernel and user portions coexist in every process's page table (though the kernel portion is shared and identical across all processes):





A few things to notice. First, the kernel follows the standard binary layout—text, then data, then bss—just like a user program. The kernel heap (managed by `xalloc()`[95d] and `malloc()`[103c]) grows upward from the end of bss. Second, the first page at address zero is intentionally left unmapped so that null pointer dereferences cause an immediate fault. Third, the I/O registers (`VIRTIO`) sit in the gap between the user stack top and `KZERO`—they are mapped here for convenience, even though they are only accessed by the kernel. Fourth, the kernel's data section contains important global variables such as `systab`[57b] (the syscall dispatch table), `devtab`[71c] (the device driver table), `conf`[62b] (hardware configuration), `palloc`[109b] (the page allocator state), and `procalloc`[128d] (the process table). Finally, the `Cpu`[64a] structure (for CPU 0) occupies a single page at `CPUADDR`[56a], below the kernel text, in the small gap between `KZERO`[51c] and `KTZERO`[51a]. The `Cpu` structure holds per-processor kernel state—the currently-running process, the scheduler's local run queue, saved registers for voluntary context switches—and is described in Chapter 3.

On multi-processor systems (e.g., Raspberry Pi 2), each additional CPU has its own `Cpu` struct allocated elsewhere; the `cpus`[65a] array and the dedicated register `R10` allow each processor to find its own. The details of SMP boot and per-CPU lookup are covered in Chapter 22.

Physical address space (ARM)

The *physical* memory layout is simpler. The Raspberry Pi 1 has 512 MiB of RAM from physical `0x00000000` to `0x20000000`. The I/O registers start immediately after, at `0x20000000` (`Soc.physio`). The GPU allocates the framebuffer somewhere within the RAM. Here is the physical address space:

```
             +------------------+
             | I/O registers    |  SYSTIMERS, INTREGS,
             |  (16 MiB)        |  MAILBOX, USBREGS, ...
0x20000000   +------------------+  PHYSIO
             |                  |
             |  free RAM        |  user pages,
             |  (user pages,    |  GPU framebuffer
             |   GPU framebuf)  |
             |                  |
             +------------------+
             |  kernel:         |
             |    bss + heap    |  xalloc, malloc pools
             +------------------+
             |    data          |  conf, systab, devtab,
             |                  |  palloc, procalloc, ...
             +------------------+
             |    text          |  main(), sched(), ...
0x00008000   +------------------+
             | L1 page table    |  16 KiB, 4096 entries
             |  (CpTTB points   |  (Section or Coarse)
             |   here)          |
0x00004000   +------------------+
             | L2 page table    |  for HVECTORS mapping
0x00003000   +------------------+
             | Cpu0 struct +    |  CPU 0's state
             |  boot stack      |
0x00002000   +------------------+
             |                  |
             | vector table     |  exception entry points
0x00000000   +------------------+
```

Virtual to physical mapping (ARM)

The kernel's `mmuinit()`[271c] function (Chapter 11) fills the L1 page table at physical `0x4000` and then the boot code stores this address in the `CpTTB` register and enables the MMU and `CpChv` (see Section 2.5.3 for the ARM MMU introduction).

Here is what the 4096-entry L1 table looks like after `mmuinit()`:

```
L1 page table (at phys 0x4000, pointed to by CpTTB)
Each entry covers 1 MiB of virtual address space.

index   virt range            entry type     maps to phys
-----   ----------            ----------     ------------
4095    0xFFF0_0000 HVECTORS   Coarse    -->  L2 at 0x3000
                                             (L2[0] = phys page 0)
```

```
4094    ...                     Fault           (unmapped)
 ...     (unmapped)             ...
3072    0xC000_0000 FRAMEBUF    Section   -->   GPU-chosen address
3071    ...                     Fault           (unmapped)
 ...     (unmapped)             ...
2559    0x9FF0_0000             Section   -->   0x1FF0_0000
 ...     (kernel RAM, contiguous)
2049    0x8010_0000             Section   -->   0x0010_0000
2048    0x8000_0000 KZERO       Section   -->   0x0000_0000
====     kernel / user boundary  ==================
2031    0x7EF0_0000             Section   -->   0x20F0_0000
 ...     (I/O registers, 16 entries)
2016    0x7E00_0000 VIRTIO      Section   -->   0x2000_0000 PHYSIO
2015    ...                     Fault           (unmapped)
 ...     (unmapped gap)         ...
 512    0x2000_0000 USTKTOP     Fault           (unmapped)
 511    0x1FF0_0000             Fault initially, then
 ...     (user space)           Coarse when process mapped
   1    0x0010_0000             (filled on demand by
   0    0x0000_0000 UZERO       arch_putmmu / fixfault)
```

The user entries (indices 0–511) start as `Fault` after `mmuinit()`. They are filled on demand by `arch_putmmu()`[186] when a process is created or when a page fault occurs: each entry becomes a `Coarse` pointer to a per-process L2 table. Within the first MiB (entry 0), the L2 table maps pages starting at `UTZERO` (`0x1000`) but leaves page 0 permanently unmapped—this is the null pointer trap. So it is not the entire first MiB that is inaccessible, only the first 4 KiB page.

The kernel mapping is a simple offset: virtual `KZERO+x` maps to physical `x`, using 512 consecutive L1 `Section` entries (indices 2048–2559). The L1 table itself, the `Cpu`[64a] structure, and the vector table are all within this range, so they are accessible at both their physical addresses and at `KZERO` plus that address. The three special mappings are the I/O registers (16 L1 sections mapping virtual `VIRTIO` to physical `PHYSIO`), the exception vectors (one L1 Coarse entry at index 4095 pointing to an L2 table that maps a single 4 KiB page of physical 0 to virtual `HVECTORS`—the ARM's `CpChv` bit directs the processor to fetch vectors from `0xFFFF0000`), and the GPU framebuffer (one L1 section at index 3072). User mappings occupy indices 0–511 and are filled per-process with Coarse entries pointing to L2 tables, so that each process has its own pages.

To make the distinction between physical and virtual addresses explicit in the code, the following typedefs are used:

⟨*pad memory pointer types(arm)* 55a⟩≡ (973a) 55b▷

```
// physical address
typedef uintptr phys_addr;
// virtual address (a user_addr or kern_addr)
typedef uintptr virt_addr;

// user address (under KZERO)
typedef uintptr user_addr;
// kernel address (mostly physical + KZERO)
typedef uintptr kern_addr;
```

⟨*pad memory pointer types(arm)* 55b⟩+≡ (973a) ◁55a

```
// wp = word pointer, vp = void pointer
typedef ulong* virt_wp;
typedef void* virt_vp;
typedef ulong* kern_wp;
typedef void* kern_vp;
```

```
typedef ulong* user_wp;
typedef void* user_vp;
```

The kernel text (code) starts at KTZERO, which is KZERO+0x8000—because the ARM firmware loads the kernel binary at physical 0x8000. The physical memory below 0x8000 is used for early kernel data structures such as CPU 0's Cpu structure at CPUADDR as mentioned in a few diagrams before:

⟨*constant* CPUADDR*(arm)* 56a⟩≡ (1005)
```
#define CPUADDR     (KZERO+0x2000)      /* Cpu structure */
```

Because the kernel's virtual addresses are simply the physical addresses plus KZERO, converting between the two is a matter of masking bits. PADDR strips the high bits to go from virtual to physical; KADDR adds KZERO to go from physical to virtual:

⟨*macro* PADDR*(arm)* 56b⟩≡ (974)
```
#define PADDR(va)   PTR2UINT(PHYSDRAM | ((uintptr)(va) & ~KSEGM))
```

⟨*macro PADDR bis(arm)* 56c⟩≡ (890a)
```
#define PADDR(va)   (((va) & ~KSEGM))
```

⟨*constant* PHYSDRAM*(arm)* 56d⟩≡ (1005)
```
#define PHYSDRAM    0
```

⟨*constant* KSEGM*(arm)* 56e⟩≡ (1005)
```
/* mask to check segment; good for 1GB dram */
#define KSEGM       0xC0000000
```

⟨*macro* KADDR*(arm)* 56f⟩≡ (974)
```
#define KADDR(pa)   UINT2PTR(KZERO    | ((uintptr)(pa) & ~KSEGM))
```

⟨*macro* PTR2UINT*(arm)* 56g⟩≡ (974)
```
#define PTR2UINT(p) ((uintptr)(p))
```

⟨*macro* UINT2PTR*(arm)* 56h⟩≡ (974)
```
#define UINT2PTR(i) ((void*)(i))
```

⟨*function* arch_cankaddr*(arm)* 56i⟩≡ (1006b)
```
/*
 * Return the number of bytes that can be accessed via KADDR(pa).
 * If pa is not a valid argument to KADDR, return 0.
 */
uintptr
arch_cankaddr(phys_addr pa)
{
    if(pa < memsize)        /* assumes PHYSDRAM is 0 */
        return memsize - pa;
    return 0;
}
```

Uses memsize 276c.

2.8.3 Syscall overview (ARM)

We saw in the helloraw.s example in Section 2.3 how a user program invokes a syscall on ARM: it places the syscall number in R0 and executes SWI $0. On the kernel side, the *interrupt handler* extracts the number from R0 and uses it as an index into systab[57b], the *syscall dispatch table*. Each entry in systab is a function pointer with a uniform signature:

⟨*typedef Syscall* 56j⟩≡ (57a)
```
typedef long Syscall(ulong*); // user_wp? or copied in Sargs?
```

The `ulong*` parameter is a pointer into the user's stack, where the syscall arguments have been laid out as a flat array of machine words. For example, `pwrite(fd, buf, n, off)` has four arguments, but on the kernel side they all arrive as consecutive `ulong` values starting at this pointer: `arg[0]` is the file descriptor, `arg[1]` the buffer address, `arg[2]` the count, and `arg[3]` (possibly two words for a 64-bit offset) the file position. Each `sys*()` function we will throughout this book casts and extracts what it needs from this generic array.

⟨syscalls/systab.h 57a⟩≡
```
// used by systab.c but also by trap.c for certain codes
#include "../../../../lib_core/libc/9syscall/sys.h"

⟨typedef Syscall 56j⟩

extern Syscall *systab[];
extern int nsyscall;
extern char *sysctab[];
```

The `systab` array is the central dispatch table. It maps each syscall number from `sys.h` (see Section 2.2) to the corresponding kernel function. This is the only point in the kernel where all syscalls are gathered together:

⟨*global* `systab` 57b⟩≡ (886a)
```
Syscall *systab[] = {
    [NOP]     sysnop,
    ⟨systab process syscalls 127⟩
    ⟨systab memory syscalls 92⟩
    ⟨systab file syscalls 294a⟩
    ⟨systab directory syscalls 323a⟩
    ⟨systab namespace syscalls 342⟩
    ⟨systab time syscalls 243⟩
    ⟨systab ipc syscalls 448a⟩
    ⟨systab concurrency syscalls 448b⟩
    ⟨systab security syscalls 591c⟩
    [ERRSTR]    syserrstr,
};
```
Uses `sysfversion()` 591e and `sysnop()` 57d.

⟨*global* `nsyscall` 57c⟩≡ (886a)
```
int nsyscall = nelem(systab);
```

Here is the simplest possible syscall, useful for testing that the whole trap-and-dispatch machinery works end to end:

⟨*syscall nop* 57d⟩≡ (881b)
```
// void nop(void);
long
sysnop(ulong*)
{
    print("Hello World\n");
    return 0;
}
```
Uses `print`.

The `print` call here is the kernel's own `print`, not the libc `print` that user programs invoke. The kernel binary is linked with libc—it pulls in concrete pieces like `pool.c` for the allocator and `vseprintf` for formatting—but it overrides key user-facing entry points with its own versions. `print` is declared as a function-pointer global and assigned at boot to `devcons_print` (Chapter 15), which writes the formatted string into the `/dev/cons` queue; from there the bytes reach either the serial UART or the framebuffer console (covered in Graphics book [Pad16d]).

To tie everything together, here is the complete trace of a `NOP` syscall, from the user program down to `sysnop()` and back:

```
USER SPACE                          KERNEL SPACE

1. nop()
   libc wrapper:
     MOVW $0, R0   (NOP=0)
     SWI $0
        |
        | ARM hardware:
        | - saves PC in R14_svc
        | - saves CPSR in SPSR_svc
        | - switches to SVC mode
        | - jumps to HVECTORS+0x08
        |   (SWI = vector #2)
        |
        +------------------------->  2. HVECTORS (0xFFFF0000):
                                        vectors[2]: LDR PC, vtable[2]
                                             |
                                             v
                                     3. vtable[2] = _vsvc
                                             |
                                             v
                                     4. _vsvc (assembly):
                                        - saves user regs -> Ureg
                                          (on kernel stack)
                                        - sets R12 = kernel SB
                                        - sets R(CPU), R(UP)
                                        - calls arch__syscall(ureg)
                                             |
                                             v
                                     5. arch__syscall (C):
                                        - scallnr = ureg->r0  (= 0)
                                        - sp = user stack pointer
                                          (the ulong* in Syscall)
                                        - ret = systab[0](sp)
                                             |
                                             v
                                     6. sysnop(ulong*):
                                        - print("Hello World\n")
                                        - return 0
                                             |
                                             v
                                     7. back in _vsvc:
                                        - restores user regs from Ureg
                                        - restores SPSR -> CPSR
                                        - RFE (return to user mode)
        |                                    |
        +<-----------------------------------+
        v
8. user program continues
```

```
after SWI instruction
```

The exception vector table at HVECTORS is set up by arch__trapinit()[278e] during initialization: it copies the vectors code (a series of LDR PC instructions) and the vtable data (an array of addresses like _vsvc, _virq, etc.) to the page at 0xFFFF0000. The CpChv bit (set during boot) tells the ARM to fetch vectors from there instead of from address 0. The "v" in _vsvc stands for "vector" and "svc" for the supervisor call (SWI) exception. Similarly, _virq handles IRQ interrupts, _vdabt handles data aborts, etc.

A few things about the registers used in step 4. R(CPU) is R10, which by convention always holds a pointer to the current processor's Cpu[64a] structure—this is the extern register Cpu* cpu declaration we will see soon. R(UP) is R9, which holds a pointer to the current process's Proc[701b] structure (up[68a] for "user process"). R12 is the static base register (SB), used by the Plan 9 C compiler to access global variables (see the LINKER book [Pad15c] for details). The kernel must reload R12 because the user program may have used it for something else.

2.8.4 FAQ (ARM)

The overview above introduced many addresses, registers, tables, and mappings. It is easy to be confused, especially because some aspects are imposed by the ARM hardware (e.g., the exception vector mechanism, the PSR layout, the MMU page table format) while others are design decisions made by the Plan 9 kernel authors (e.g., the value of KZERO, the placement of VIRTIO, the systab dispatch table). The following questions and answers try to clarify the most common sources of confusion.

The exception vector table is at HVECTORS and the syscall table is systab. How are they related?

They are different things. The exception vector table at HVECTORS is a hardware mechanism: when a SWI instruction executes, the ARM jumps to a fixed address in this table. That entry loads the address of _vsvc from vtable. Then _vsvc (assembly code) extracts the syscall number from R0 and calls arch__syscall() (C code), which finally indexes into systab to find the right kernel function. So HVECTORS is the hardware entry point, and systab is the software dispatch table—they are connected by the chain of code in between.

Why SWI $0? Does the argument to SWI matter?

On Plan 9, the argument to SWI is ignored. The syscall number is passed in R0 instead. The SWI instruction is used purely for its side effect of trapping into supervisor mode. Some other operating systems encode the syscall number in the SWI argument, but Plan 9 chose the simpler approach of using a register. Similarly, Linux on x86 uses int 0x80 (or the faster syscall instruction on x86-64) to trap into the kernel, with the syscall number passed in EAX.

Why split the 4 GiB address space between kernel and user? Could each have its own full 4 GiB?

Some systems do give user space the full address range and switch page tables entirely when entering the kernel. For example, xv6[7] on RISC-V maintains a separate kernel page table (with direct virtual=physical mapping) and per-process user page tables. On every syscall or interrupt, xv6 switches the page table base register from the user's table to the kernel's. To make this switch possible, both tables map a single *trampoline* page at the same high virtual address—this is where the switching code runs.

The Plan 9 approach is different and simpler: the kernel is mapped in the upper half of every process's page table. A syscall or interrupt does not need to switch page tables at all—the kernel's code and data are always mapped and immediately accessible. The trade-off is that user programs get only 2 GiB of virtual space instead of 4 GiB (but on a machine with 512 MiB of physical RAM this is not a practical limitation). The benefits are: trap entry is faster (no TLB flush from a page table switch), no trampoline is needed, and the kernel can access user memory directly (e.g., syscall arguments on the user stack, buffer addresses) without switching page tables first. The kernel still needs to validate user pointers (e.g., check they are below KZERO, handle page faults), but the shared mapping makes the common case simpler than with separate page tables where even reading a user byte requires a special copy routine.

[7] https://github.com/mit-pdos/xv6-riscv

Can a page fault occur inside the kernel? Can kernel memory be swapped out?

The kernel's own pages are never swapped and are always mapped (via the L1 section entries set up by `mmuinit()`[271c]). So the kernel does not page-fault when accessing its own data structures. A page fault can only occur for user-space addresses. However, a page fault can occur while the kernel is executing a syscall on behalf of a user process—for example, when the kernel copies data from a user buffer (the user's page might not be resident). The fault handler recognizes this case and handles it; see the page-fault section in Chapter 8 for the details.

Why is `VIRTIO` mapped below `KZERO`, in the gap between user space and kernel space?

The kernel RAM mapping (`KZERO+x` = physical `x`) covers exactly `Soc.dramsize` (512 MiB), filling L1 entries 2048–2559. Placing the I/O registers above `KZERO` would require additional L1 entries in the kernel region, which would overlap with the simple offset mapping. Putting them at `0x7E000000` (below `KZERO`) avoids this conflict. User processes cannot access these pages because the L1 entries are set with kernel-only permissions.

Why is `CPUADDR` at `0x2000`, below the kernel text at `0x8000`?

The ARM firmware loads the kernel binary at physical `0x8000`. The memory below that is unused by the binary, so the kernel reuses it for the `Cpu` structure (at `0x2000`), the L2 page table (at `0x3000`), and the L1 page table (at `0x4000`–`0x8000`). This avoids wasting the 32 KiB gap that the firmware leaves.

2.8.5 FAQ

The previous FAQ focused on ARM-specific mechanics. This second FAQ has the same spirit, but the questions are architecture-independent: they apply to any kernel on any CPU, and answering them clarifies design choices Plan 9 makes regardless of which hardware it runs on.

The following are more subtle questions that may not occur to every reader on first reading, but that help deepen the understanding of the kernel's design.

When an interrupt or syscall fires while a user process is running, whose stack does the kernel execute on? Not the user stack (it is untrusted—potentially corrupt, tiny, or read-only in a dying process) and not a single global kernel stack (which would break on SMP or on nested traps). Every `Proc` has its own kernel stack (`kstack`), allocated when the `Proc` is created. On every kernel entry the CPU switches `SP` into that per-process stack, the kernel runs on it until the handler returns, and control switches back to user mode. Two nice consequences follow: the kernel can sleep (block the current process while waiting for I/O) without disturbing any other process's state, and nested traps just work as long as `kstack` is deep enough.

Much of this chapter may seem complicated and very ARM-specific. But the essential concepts are the same across architectures. Every kernel needs to:

- split the address space between kernel and user,
- set up page tables (or their equivalent) so the MMU can translate virtual to physical addresses,
- place an exception/interrupt vector table at a known address,
- save and restore user registers on kernel entry/exit,
- dispatch syscalls through a table of function pointers,
- reserve memory for per-CPU state.

The concrete addresses (`KZERO`, `HVECTORS`, etc.), the register conventions (`R10` for `cpu`, `R9` for `up`), and the page table format (L1 sections, L2 pages) are ARM-specific details, but the architecture of the kernel would look very similar on x86 or RISC-V. In fact most of the kernel is portable: the ARM-specific code in `arch/arm/` amounts to only about 1000 lines, while everything else—memory management, scheduling, file system, devices, IPC—lives in subsystem directories that are reused across architectures.

2.9 Book structure

The rest of the book is organized as follows. Chapters 3 and 4 present the foundational building blocks: core data structures (`Conf`[62a], `Cpu`[64a], etc.) and concurrency primitives (locks, `sleep()`[??]/`wakeup()`[??], `Rendez`[701b]). Part I, *Virtualizing the Machine*, covers how the kernel gives each process the illusion of having the hardware to itself: memory allocators (Chapter 5), processes (Chapter 6), interrupts and syscalls (Chapter 7), virtual memory (Chapter 8), scheduling (Chapter 9), time (Chapter 10), and the boot sequence (Chapter 11). Part II, *Everything is a File*, covers Plan 9's central abstraction: files as the uniform interface to data, devices, and services. This includes the `Chan`[701b] and `Dev`[701b] structures (Chapter 12), directories (Chapter 13), per-process namespaces (Chapter 14), the device driver framework (Chapter 15), streaming I/O queues (Chapter 16), and on-disk file systems (Chapter 17). Part III, *Beyond the Basics*, covers the remaining subsystems: inter-process communication (Chapter 18), debugging and profiling support (Chapters 19 and 20), advanced topics (Chapter 22), and the conclusion (Chapter 24). Some appendices present non-functional code: debugging helpers, error management, and generic utilities.

Chapter 3

Core Data Structures

Show me your code and conceal your data structures, and I shall continue to be mystified. Show me your data structures, and I won't usually need your code; it'll be obvious.

Fred Brooks

In this chapter, I will present the core data structures of the kernel. Some have already been introduced in the Overview: `Ureg`[37c], the processor modes (`PsrMode`), the address typedefs, and the syscall dispatch table (`systab`[57b]). Here I will cover the remaining structures that appear everywhere in the kernel: `Conf`[62a] (hardware configuration), `Cpu`[64a] (per-processor state), `Label`[226d] (context switch), and `Ref`[701b] (reference counting).

3.1 `Conf` and `conf`

The `Conf` structure records what the kernel learned about the hardware during the boot sequence: how many processors, how much physical memory, and how many processes to support. It is filled once by `confinit()X` during initialization and then treated as read-only.

⟨*struct* `Conf` 62a⟩≡ (702)
```
struct Conf
{
    ulong ncpu;    /* processors */
    Confmem mem[4];   /* physical memory */

    ulong nproc;    /* processes */
    ⟨Conf other fields 188f⟩
};
```

The `mem` array holds up to 4 `Confmem` entries for non-contiguous regions of available physical memory. On the Raspberry Pi there is just one bank; on a PC there can be two (one below and one above the kernel binary). `confinit()X` excludes the memory occupied by the kernel itself.

⟨*global* `conf` 62b⟩≡ (705b)
```
Conf conf;
```

Each `Confmem` describes one memory bank—its physical base address and the number of usable pages. These entries are later consumed by the memory allocators (`xalloc()`[95d] and `palloc()`[109b]):

⟨*struct* `Confmem` 62c⟩≡ (702)
```
// memory "bank"
struct Confmem
{
    phys_addr base;
    uintptr limit; // used by bcm only. phys_addr too?
    ulong npage;
    ⟨Confmem other fields 488c⟩
};
```

On the Raspberry Pi, there is typically a single memory bank. The `base` is the start of *usable* physical RAM (after the kernel binary and its static allocations), and `npage` is the number of free pages. For example, on a Raspberry Pi 1 with 512 MiB of RAM:

```
Physical memory:

0x1FFF_FFFF +---------------------------+
            |                           |
            |  Confmem bank 0           |
            |  base = ~end              |
            |  npage = remaining pages  |
            |                           |
      ~end  +---------------------------+
            |  kernel text+data+bss     |
    0x8000  +---------------------------+
            |  firmware / Cpu struct     |
    0x0000  +---------------------------+
```

The exact values depend on the kernel binary size and are computed at boot time by `confinit()`.

The Raspberry Pi bootloader passes configuration parameters to the Plan 9 kernel as key-value string pairs (similar to a Linux kernel command line). These are stored in the `confname` and `confval` parallel arrays. One important parameter is `*kernelpercent*`, which controls what fraction of physical memory the kernel reserves for its own dynamic allocations (via `xalloc`[95d]). The default in the code is 60%, but in practice a value around 20% is more typical—most memory should be available for user processes.

⟨*globals confname and confvar* 63a⟩≡ (705b)
```
// conf (boot) parameters *e.g. { "*kernelpercent*" => "60" }
// hash<string, string>
char *confname[MAXCONF];
char *confval[MAXCONF];
```
Uses MAXCONF 63b.

⟨*constant* MAXCONF 63b⟩≡ (702)
```
#define MAXCONF         64
```

⟨*global* nconf 63c⟩≡ (705b)
```
// Hashtbl.length(confname)
int nconf;
```

⟨*function* getconf 63d⟩≡
```
char* getconf(char *name)
{
        int i;
        for(i = 0; i < nconf; i++)
                if(cistrcmp(confname[i], name) == 0)
                        return confval[i];
        return nil;
}
```

3.2 Cpu and cpu

The `Cpu` structure holds the per-processor state: which process is currently running, the clock tick count, MMU state, and a small kernel stack used during boot and interrupts. On a uniprocessor Raspberry Pi 1 there is a

single Cpu at CPUADDR; on a multiprocessor system each CPU has its own.

⟨*struct* Cpu 64a⟩≡ (702)
```
//coupling: do not change the order of the first 4 fields!
// Some assembly code assumes this order.
struct Cpu
{
    int cpuno;      /* physical id of processor (KNOWN TO ASSEMBLY) */
    ⟨Cpu second field 74f⟩
      // ref<Proc>, or None if halting?
    Proc* proc;     /* current process on this processor */

    struct Arch_Cpu;

    ⟨Cpu time fields 64c⟩
    ⟨Cpu stat fields 656a⟩
    ⟨Cpu other fields 89c⟩

    // must be at the end of the structure!
    int stack[1];
};
```
Uses Arch_Cpu.

The first fields have a fixed layout because assembly code accesses them by hardcoded offsets. The stack[1] at the end is a C idiom: the actual stack extends to the end of the CPUSIZE page, so the array's "one element" is really the start of a much larger area.

⟨*constant* CPUSIZE*(arm)* 64b⟩≡ (1005)
```
#define CPUSIZE BY2PG
```

We will see more about ticks and the clock in the Interrupts chapter (Chapter 7).

⟨Cpu *time fields* 64c⟩≡ (64a) 64d▷
```
ulong ticks;      /* of the clock since boot time */
```

⟨Cpu *time fields* 64d⟩+≡ (64a) ◁64c
```
uvlong  cpuhz;
int cpumhz; // cpuhz / 1_000_000
```

The Cpu structure shown above is portable across all architectures. But each architecture also has specific fields (MMU state, floating-point context, etc.) that are gathered in Arch_Cpu. Thanks to the Plan 9 C compiler extension (see COMPILER book [Pad16b]), struct Arch_Cpu; can be *embedded* in Cpu without giving it a field name—its fields are accessed directly as if they were part of Cpu itself.

⟨*struct* Arch_Cpu*(arm)* 64e⟩≡ (973b)
```
//coupling: do not change the first field! Some assembly code assumes this order
struct Arch_Cpu {
    ⟨Cpu Arch mmu fields(arm) 179k⟩
    ⟨Cpu Arch float fields(arm) 518d⟩
    ⟨Cpu Arch save of first five registers fields(arm) 279c⟩
};
```

The global cpu allows to access the CPU structure. It must be per-processor global: on a multiprocessor system, each CPU must see its own Cpu structure. Since each processor has its own set of registers, a dedicated register is a natural way to hold per-processor state. On ARM, this is achieved by dedicating register R10—the 5c compiler's extern register extension reserves R10 so it is never used for anything else.

⟨*global* cpu*(arm)* 64f⟩≡ (973a)
```
extern register Cpu* cpu;          /* R10 */
```

Each processor sets its `R10` to point to its own `Cpu` during boot. Note that `R10` is only reserved in kernel code, not in user-space programs; it is a kernel convention. That is why `_vsvc` sets `R(CPU)` early upon entering kernel mode (visible in the syscall trace diagram in Section 2.8.3). On x86, where registers are scarce, a different trick involving the page table is used instead.

⟨*global* `cpus` 65a⟩≡ (705b)
```
/*
 * Each processor sees its own Cpu structure at address CPUADDR.
 * However, the Cpu structures must also be available via the per-processor
 * MMU information array cpus, mainly for disambiguation and access to
 * the clock which is only maintained by the bootstrap processor (0).
 */
// array<ref<Cpu>>
Cpu* cpus[MAXCPUS];
```

⟨*constant* `MAXCPUS` *(arm)* 65b⟩≡ (1005)
```
#define MAXCPUS     4           /* max # cpus system can run */
```

`MAXCPUS` is an arch-specific ceiling—each port picks a number big enough to cover the hardware it expects to run on. The Raspberry Pi port fixes it at 4, which has been enough for every Raspberry Pi board from the Pi 1 (2012, single-core ARM11) through the Pi 5 (2023, quad-core Cortex-A76). On x86 the same constant is typically set higher (8 or 16) to accommodate desktop and server machines.

⟨*macro* `CPUS` 65c⟩≡ (702)
```
#define CPUS(n)  (cpus[n])
```

The `Active` structure tracks global multiprocessor state that does not belong to any single `Cpu`. The `Active.cpus` bitmap records which processors are online, and `exiting` signals a system-wide shutdown. While `cpus` (the array) gives access to each processor's `Cpu` structure, `Active` answers a different question: which processors are alive, and is the system shutting down? It also carries a `Lock`[701b] to serialize concurrent updates from multiple CPUs.

⟨*struct* `Active` 65d⟩≡ (868c)
```
struct Active
{
    // array<bool> (coupling: sizeof(int) must be >= MAXCPUS)
    int cpus;      /* bitmap of active CPUs */
    bool exiting;    /* shutdown */
    ⟨Active other fields 628a⟩
    // extra
    Lock;
};
```

⟨*global* `active` 65e⟩≡ (873a)
```
struct Active active;
```
Uses `Active` 65d.

3.3 `Proc` and up

The `Proc` structure is the largest and most important data structure in the kernel—it is what the kernel knows about a process. Each running program has one `Proc` containing its state, memory map, open files, scheduling priority, parent/child relationships, notes (signals), error handling context, and much more. It is one of the largest structures in the kernel.

The structure is organized in sections (visible in the comment banners below), each corresponding roughly to a chapter in this book. We will only show the overview here; the individual fields are introduced gradually in their respective chapters.

⟨*struct* `Proc` 66⟩≡ (868c)

```
struct Proc
{
//----------------------------------------------------------------------
//coupling: have to be first! Some assembly code assumes this order.
//----------------------------------------------------------------------
    ⟨Proc assembly fields 226e⟩
//----------------------------------------------------------------------
// State
//----------------------------------------------------------------------
    ⟨Proc state fields 67b⟩
//----------------------------------------------------------------------
// Memory
//----------------------------------------------------------------------
    ⟨Proc memory fields 119a⟩

    struct Arch_Proc;
//----------------------------------------------------------------------
// Scheduling
//----------------------------------------------------------------------
    ⟨Proc scheduling fields 219c⟩
//----------------------------------------------------------------------
// Files
//----------------------------------------------------------------------
    ⟨Proc files fields 69⟩
//----------------------------------------------------------------------
// Notes
//----------------------------------------------------------------------
    ⟨Proc notes fields 450b⟩
//----------------------------------------------------------------------
// Process hierarchy
//----------------------------------------------------------------------
    ⟨Proc hierarchy fields 148b⟩
//----------------------------------------------------------------------
// Synchronization
//----------------------------------------------------------------------
    ⟨Proc synchronization fields 88f⟩
//----------------------------------------------------------------------
// Error management
//----------------------------------------------------------------------
    ⟨Proc error managment fields 665b⟩
//----------------------------------------------------------------------
// Stats, profiling
//----------------------------------------------------------------------
    ⟨Proc stats and profiling fields 503b⟩
//----------------------------------------------------------------------
// Debugging (the kernel itself)
//----------------------------------------------------------------------
    ⟨Proc debugging fields 77b⟩
//----------------------------------------------------------------------
// For debugger, strace
//----------------------------------------------------------------------
    ⟨Proc debugger fields 165e⟩
//----------------------------------------------------------------------
// Other
//----------------------------------------------------------------------
```

```
    ⟨Proc other fields 157a⟩
//------------------------------------------------------------------------
// Extra
//------------------------------------------------------------------------
    ⟨Proc extra fields 81d⟩
};
```

Uses Arch_Proc.

Just as Cpu has an Arch_Cpu, Proc embeds an anonymous struct Arch_Proc for architecture-specific fields (on ARM, this is the per-process MMU state). We will detail these fields later.

⟨*struct* Arch_Proc*(arm)* 67a⟩≡ (1004b)

```
struct Arch_Proc
{
    ⟨Proc Arch memory fields(arm) 180c⟩
};
```

Perhaps the most important field in Proc is its state—whether the process is running, ready to run, sleeping, or dead:

⟨Proc *state fields* 67b⟩≡ (66) 128a▷

```
// enum<Procstate>
int state; // Dead, Queuing, etc, (used by /proc/#/status if psstate==nil)
```

A process is always in one of the following states. The most important ones are Running (executing on a CPU), Ready (runnable, waiting for the scheduler to pick it), and Wakeme (sleeping, waiting for an event):

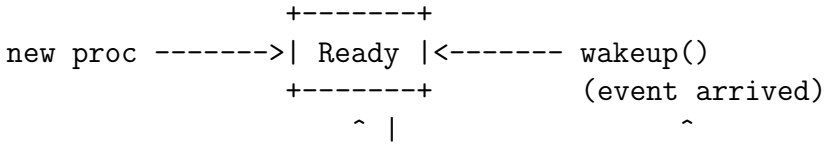
⟨*enum* procstate 67c⟩≡ (868c)

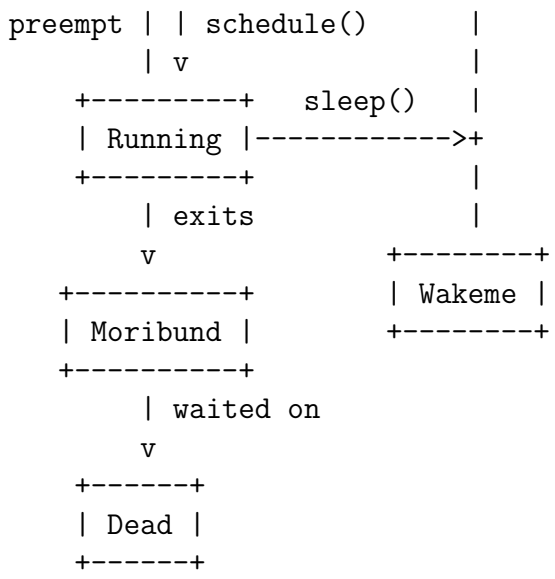
```
/* Process states, Proc.state */
enum Procstate
{
    Dead = 0,
    Running,
    ⟨Procstate cases 81e⟩
};
```

⟨*global* statename 67d⟩≡ (877c)

```
// hash<enum<Procstate>, string>, coupling: with enum Procstate
char *statename[] =
{
    [Dead]        = "Dead",
    [Running]     = "Running",
    [Queueing]    = "Queueing",
    [QueueingR]   = "QueueingR",
    [QueueingW]   = "QueueingW",
    [Moribund]    = "Moribund",
    [Ready]       = "Ready",
    [Scheding]    = "Scheding",
    [Wakeme]      = "Wakeme",
    [Broken]      = "Broken",
    [Stopped]     = "Stopped",
    [Rendezvous]  = "Rendezvous",
    [Waitrelease] = "Waitrelease",
};
```

The transitions between these states are driven by events in the process's life—being picked by the scheduler, blocking on a lock, waking up from a sleep, dying. The core loop is the small triangle Ready – Running – Wakeme:





A newly created process enters `Ready`, is promoted to `Running` by the scheduler, and falls back to `Ready` on a preemption (clock tick). A `Running` process moves to `Wakeme` when it calls `sleep()` to wait for a condition, and another process (or an interrupt handler) calls `wakeup()` to push it back to `Ready`. The three `Queueing` variants (`Queueing`, `QueueingR`, `QueueingW`) are subtypes of the sleeping state used specifically by `QLock` and `RWLock` queues—they are distinct so that `ps` can show which kind of lock a process is waiting on. `Moribund` and `Dead` are the two phases of death: a process exits into `Moribund` and stays there until its parent collects the exit status, after which its slot becomes `Dead` and can be reused.

Like `cpu`, the `up` pointer ("user process") is per-processor: it points to the `Proc` currently running on this CPU. On ARM it is stored in `R9`. The `5c` compiler assigns `extern register` variables to registers in decreasing order: the first declaration gets `R10` (which is `cpu`), the second gets `R9` (which is `up`), and a third would get `R8`, and so on.

⟨*global* `up` *(arm)* 68a⟩≡ (973a)
```
extern register Proc* up;          /* R9 */
```

3.4 Chan

A `Chan` ("channel") is the kernel's representation of an open file. It is the internal object behind a user's file descriptor—the file descriptor is just an integer index into a per-process table of `Chan` pointers.

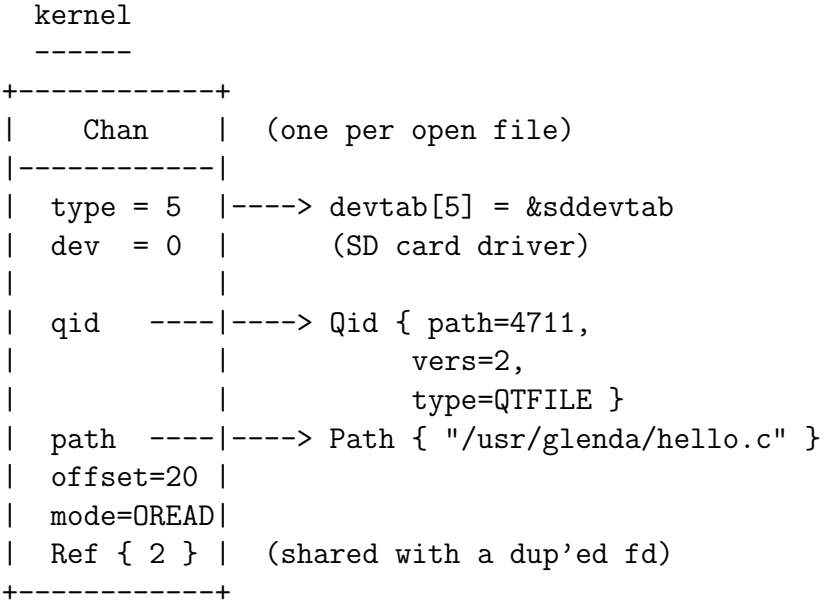
Because everything in Plan 9 is a file, channels are used everywhere: for real files on disk, for device control files like `/dev/cons`, for network connections, for mounted file servers, etc. The key fields are `type` (which device driver owns this channel), `dev` (a device-specific sub-identifier—together `type` and `dev` play the role of UNIX's major and minor device numbers), `qid` (the file's unique identity on that device, like an inode number), `path` (the file name), `offset` (the current read/write position), and `mode` (read, write, or both):

⟨*struct* `Chan` 68b⟩≡ (804b)
```
struct Chan
{
    ushort type; // idx in devtab
    ulong dev;
    Qid qid;

    Path* path;

    vlong offset;     /* in fd */

    //enum<open>, actually restricted to the OREAD|OWRITE|ORDWR of open type
```

```
    ushort mode;     /* read/write */

    bool ismtpt; // is a mount point

    union {
       void* aux; // generic pointer, for specific usages
       ⟨Chan union other fields 459g⟩
    };
    ⟨Chan other fields 295a⟩

    // extra
    Ref; /* the Lock in this Ref is also Chan's lock */
    ⟨Chan extra fields 295c⟩
};
```

Uses __anon_struct_93 68b.

Pictorially, this is the chain of pointers starting at a user's file descriptor and ending at the actual file object:



The file descriptor `3` is just an index into a small per-process array of `Chan` pointers. The `Chan` holds the "where am I reading, and in what mode" state that is private to this file descriptor—so that two independent `open` calls on the same file have independent `offset`s. The `Qid`, together with the `type`/`dev` pair, identifies the underlying file object, which is shared—two `open`s of the same file yield two distinct `Chan`s that carry identical `Qid`s. A `dup(2)`'ed descriptor, by contrast, does not allocate a new `Chan`: it bumps the `Ref` counter on the existing one, so both file descriptors share the same `offset`. In UNIX this chain is very similar, but the names and split differ. The kernel's "open file" object is called a `struct file`, and it points to a `struct inode` (Plan 9's `Qid` plus some metadata). The interesting difference is that in Plan 9 the `type` field inside `Chan` is a pure index into `devtab`, so there is no distinction between block and character devices—everything is just a file server.

Every process holds a few `Chan` pointers directly in its `Proc` structure. The two most fundamental ones are `slash` (the root of the process's namespace) and `dot` (its current directory):

⟨Proc *files fields* 69⟩≡ (66) 298b ▷

```
// ref<Chan>
Chan  *slash; // The root! used by namec()
// ref_counted<Chan>
Chan  *dot; // The current directory
```

A `Qid` ("q" for "unique") is Plan 9's equivalent of a UNIX inode number: it uniquely identifies a file within a device or file server. The `path` field is the unique identifier, `vers` is a version number for cache invalidation,

and `type` distinguishes files from directories:

⟨*struct* `Qid` 70a⟩≡ (853i)
```
struct Qid
{
  // note that this is not a string, but an int! it's kind of an inode
  uvlong  path;
  // for cache invalidation
  ulong vers;
  // enum<Qidtype>
  byte type;
};
```

Despite its name, `path` is not a string but a 64-bit integer—it is a unique numeric identifier for a file within a device, much like a UNIX inode number. The name is confusing because `Chan` also has a separate `path` field (of type `Path*`) that holds the actual string pathname. The `vers` field is more unusual: it is a version number that the file server increments whenever the file's content changes. The kernel uses it for cache invalidation—if a cached `Qid` has a stale `vers`, the kernel knows it must re-read the file's metadata.

⟨*enum* `qidtype` 70b⟩≡ (853i)
```
/* bits in Qid.type */
enum Qidtype {
  QTFILE = 0x00,    /* plain file */
  QTDIR = 0x80,    /* type bit for directories */
  ⟨Qidtype cases 427b⟩
};
```

⟨*function* `eqqid` 70c⟩≡ (808c)
```
bool
eqqid(Qid a, Qid b)
{
    return a.path==b.path && a.vers==b.vers;
}
```

⟨*function* `mkqid` 70d⟩≡ (810)
```
void
mkqid(Qid *q, vlong path, ulong vers, int type)
{
    q->type = type;
    q->vers = vers;
    q->path = path;
}
```

⟨*function* `eqchan` 70e⟩≡ (808c)
```
bool
eqchan(Chan *a, Chan *b, bool skipvers)
{
    if(a->qid.path != b->qid.path)
        return false;
    if(!skipvers && a->qid.vers!=b->qid.vers)
        return false;

    if(a->type != b->type)
        return false;
    if(a->dev != b->dev)
        return false;
    return true;
}
```
Uses `false` and `true`.

⟨*function* `eqchantdqid` 71a⟩≡ (808c)
```
bool
eqchantdqid(Chan *a, int type, int dev, Qid qid, bool skipvers)
{
    if(a->qid.path != qid.path)
        return false;
    if(!skipvers && a->qid.vers!=qid.vers)
        return false;
    if(a->type != type)
        return false;
    if(a->dev != dev)
        return false;
    return true;
}
```
Uses `false` and `true`.

3.5 `Dev` and `devtab`

A `Dev` is a device driver—or more precisely, a file server implemented in the kernel. Each `Dev` provides a set of methods (`open`, `read`, `write`, `close`, etc.) that implement the file interface for a particular kind of resource: the console (`devconsX`), the process table (`devprocX`), the root file system (`devrootX`), the SD card (`devsdX`), etc.

⟨*struct* `Dev` 71b⟩≡ (804b)
```
struct Dev
{
    Rune dc; // dev character code, e.g. '/' (devroot), 'e' (devenv), ...
    char* name;

    ⟨Dev methods 286b⟩
};
```

The single character `dc` (e.g., `'c'` for console, `'p'` for proc) identifies the device in Plan 9's `#c`, `#p` naming convention. In UNIX, devices are identified by major and minor numbers—opaque integers that require a lookup table to understand. Accessing a device requires creating special files with `mknod` and a populated `/dev` directory. Plan 9 replaces the major number with a human-readable Unicode character (a `Rune`): `'c'` for the console, `'p'` for the process table, `'e'` for the environment, etc. You can see these characters in Plan 9's device path syntax: `#c` for the console device, `#p` for the proc device.

This is much clearer than memorizing that major 4 means the console. And because `Rune` is a Unicode code point (not an ASCII byte), Plan 9 can support more than 256 different device drivers.

The global `devtab` is an array of pointers to all the `Dev` structures compiled into the kernel—it is the registry of all available device drivers. When a `Chan` is opened on a device, `Chan.type` is the index into `devtab`, so that `devtab[c->type]->read()` calls the right driver's read method.

⟨*global* `devtab` 71c⟩≡ (810)
```
// array<ref_own<Dev>>
Dev** devtab = nil;
```

While `Chan.type` is still a numeric index internally into `devtab`, the user-visible name uses the character from `Dev.dc`—so the human never sees the number.

3.6 `Soc` and `soc` (ARM)

The `Soc` structure captures the hardware specifics of the System-on-Chip: how much DRAM, where the I/O registers start, and various MMU and bus parameters. It is the single place where the differences between

Raspberry Pi 1 and Pi 2 are recorded:

⟨*struct* `Soc` *(arm)* 72a⟩≡ (888b)

```
struct Soc {            /* SoC dependent configuration */
    ulong   dramsize;
    phys_addr physio;
    ⟨Soc bus fields 957e⟩
    ⟨Soc mmu fields 529i⟩
    ⟨Soc other fields 536f⟩
};
```

⟨*global* `soc` *(raspberry pi1)(arm)* 72b⟩≡ (994b)

```
Soc soc = {
    .dramsize   = 512*MiB,    // 0 to 0x20000000
    .physio     = 0x20000000, // after RAM
    ⟨soc(raspberry pi1) other fields(arm) 529j⟩
};
```

Uses `MiB` 48a.

Stepping back from `Soc` and the other ARM-specific bits at the end of this chapter, notice how little of what we saw was actually arch-specific. The portable `Cpu`, `Proc`, and their relatives will be used unchanged in the next chapters; the arch-specific pieces live inside them as embedded `Arch_Cpu` and `Arch_Proc` fields. The handful of genuinely arch-specific structures—`Ureg` (the user-register save area), the page-table types, and `Soc`—can each be swapped for their x86 equivalents without touching anything else. In other words the kernel's data model is arch-independent; the arch differences live in small, named hooks.

Chapter 4

Concurrency

In this chapter, I will present the concurrency primitives used inside the kernel. These are foundational building blocks—almost every chapter that follows will use locks, interrupt disabling, or sleep/wakeup. In fact, we already encountered `Lock`[701b] (in `Active`[65d]) and `Ref`[701b] (in `Chan`[701b]) in the previous chapter without explaining them—they are defined here.

Kernels were arguably the first concurrent programs, long before user-level threads and multicore CPUs existed. Even on a uniprocessor, the kernel is inherently concurrent: a syscall can be interrupted at any point by a hardware interrupt (a timer tick, a keyboard press, a disk completion), and the interrupt handler may access the same data structures the syscall was modifying. On a multiprocessor, the situation is worse—two CPUs can execute kernel code simultaneously. Concurrency bugs in the kernel are especially dangerous because they can corrupt the data structures that all processes depend on. This chapter covers the kernel's own concurrency mechanisms. User-level concurrency (semaphores, rendezvous) is covered in Chapter 18.

4.1 Critical regions

A critical region is a section of code that accesses shared data and must not be executed by two threads of control at the same time. If two paths of execution read and modify the same data structure concurrently (for instance, the scheduler's run queue), the result is a race condition—the data structure can end up in an inconsistent state, leading to crashes or subtle corruption. In the kernel, the concurrent "threads" are not user-level threads but different paths of execution that can interleave:

- A syscall executing on behalf of process A can be interrupted by a timer tick, which triggers the scheduler, which switches to process B, which makes its own syscall.
- On a multiprocessor, two CPUs can execute two different syscalls simultaneously.
- An interrupt handler can fire while the kernel is in the middle of modifying a data structure that the handler also needs.

The kernel uses a hierarchy of mechanisms to protect critical regions, from coarsest to finest: `arch_splhi()` (disable all interrupts—the simplest but most heavy-handed), `Lock` (a spin lock using `arch_tas()`), `ILock` (a spin lock that also disables interrupts, to avoid deadlock with interrupt handlers), `QLock` (a lock where waiters sleep instead of spinning), and `RWLock` (a readers/writer lock). The following sections present each in turn.

4.2 Interrupts disabling: `arch_splhi()`, `arch_spllo()` (ARM)

The simplest way to protect a critical region is to disable interrupts: if no interrupt can fire, no other code can run on this processor. The "spl" name is a legacy from UNIX, standing for "Set Priority Level." `arch_splhi()`

disables IRQ interrupts by setting the `PsrDirq` bit in the CPSR; `arch_spllo()` re-enables them. `arch_splhi()` returns the old CPSR value so `arch_splx()` can restore it later. On a uniprocessor, `arch_splhi()` is sufficient for all mutual exclusion. But it does not scale to SMP: disabling interrupts on one CPU does not prevent another CPU from executing kernel code. For SMP, finer-grained locks are needed.

⟨*function* `arch_splhi` *(arm)* 74a⟩≡ (970)
```
// bool   arch_splhi(void);
TEXT arch_splhi(SB), 1, $-4
    ⟨arch_splxxx() save caller information(arm) 74e⟩
    MOVW    CPSR, R0            /* turn off irqs (but not fiqs) */
    ORR $(PsrDirq), R0, R1
    MOVW    R1, CPSR
    RET
```

⟨*function* `arch_spllo` *(arm)* 74b⟩≡ (970)
```
// bool   arch_spllo(void);
TEXT arch_spllo(SB), 1, $-4
    MOVW    CPSR, R0            /* turn on irqs and fiqs */
    BIC $(PsrDirq|PsrDfiq), R0, R1
    MOVW    R1, CPSR
    RET
```

⟨*function* `arch_splx` *(arm)* 74c⟩≡ (970)
```
// void    arch_splx(bool);
TEXT arch_splx(SB), 1, $-4
    ⟨arch_splxxx() save caller information(arm) 74e⟩
    MOVW    R0, R1              /* reset interrupt level */
    MOVW    CPSR, R0 //dead
    MOVW    R1, CPSR
    RET
```

⟨*function* `arch_islo` *(arm)* 74d⟩≡ (970)
```
// bool arch_islo(void);
TEXT arch_islo(SB), 1, $-4
    MOVW    CPSR, R0
    AND $(PsrDirq), R0
    EOR $(PsrDirq), R0
    RET
```

When `arch_splhi()` disables interrupts, it also records who called it, for debugging. Since `Cpu` lives at the fixed address `CPUADDR`, the assembly code can store the caller's return address (`R14`) directly into the second field of `Cpu` at offset 4:

```
CPUADDR+CPUSIZE  +--------------------+
                 | stack[1] ...       |
                 +--------------------+
                 | ...                |
                 +--------------------+
CPUADDR+0x04     | splpc              |  <- saved here
                 +--------------------+
CPUADDR          | cpuno              |
                 +--------------------+
```

⟨`arch_splxxx()` *save caller information(arm)* 74e⟩≡ (74)
```
MOVW    $(CPUADDR+4), R2        /* save caller pc in Cpu */
MOVW    R14, 0(R2)
```

⟨`Cpu` *second field* 74f⟩≡ (64a)
```
// must be second field at 0x04, used by arch_splhi()
ulong splpc;      /* pc of last caller to arch_splhi */
```

4.3 Test and set: `arch_tas()` (ARM)

Test-and-set is the hardware primitive on which all spin locks are built. The idea is simple: atomically write 1 into a memory word and return the old value. If the old value was 0, the caller has acquired the lock; if it was already 1, someone else holds it. In C terms, `lock()` will call `if(arch_tas(\&l->key) == 0)`. The key word is "atomically." On a RISC architecture like ARM, a naive implementation would need two separate instructions—a load and a store:

```
// hypothetical broken arch_tas:
MOVW  0(R1), R0       // load key into R0
MOVW  $1, 0(R1)       // store 1 into key
RET                    // return old value in R0
```

But an interrupt can fire between the load and the store:

```
Process A (syscall):    Process B (interrupt handler):

MOVW 0(R1), R0
  (reads key=0)
                        <--- timer interrupt fires --->
                        MOVW 0(R1), R0
                          (reads key=0, A hasn't stored yet)
                        MOVW $1, 0(R1)
                          (sets key=1)
                        R0==0 -> if() succeeds
                        ... enters critical section ...
                        ... done, returns from interrupt ...

MOVW $1, 0(R1)
  (sets key=1)
R0==0 -> if() succeeds
... also enters critical section! BUG
```

Both see `key=0` and both believe they acquired the lock. On the Raspberry Pi 1, the solution is the `SWPW` (swap word) instruction, which performs the load-and-store in a single indivisible operation—the interrupt sees either the state before or after, never in between. On the Pi 2 (ARMv7, SMP), a different mechanism using `LDREX/STREX` is needed because `SWPW` is deprecated and does not provide the memory ordering guarantees required for multiprocessor correctness.

⟨*function* `arch_tas` *(raspberry pi1)(arm)* 75⟩≡ (971a)

```
TEXT    arch_tas(SB), $-4
    MOVW    R0,R1
    MOVW    $1,R0
    SWPW    R0,(R1)         /* fix: deprecated in armv6 */
    RET
```

Why is `SWPW` needed instead of a plain `MOVW` load followed by a `MOVW` store? Consider the naive sequence: `MOVW (R1), R0` then `MOVW $1, (R1)`. If an interrupt fires between those two instructions and the interrupt handler (or another processor) also tries to acquire the same lock, it will read `0` (unlocked), set it to `1`, and proceed—but so will the original code when it resumes. Both believe they hold the lock, violating mutual exclusion. `SWPW` avoids this by atomically swapping the register with memory in a single instruction that cannot be interrupted.

4.4 Atomic count: `arch_xinc()` and `arch_xdec()` (ARM)

The other fundamental atomic operation is increment/decrement. These are used for reference counting (e.g., `Ref`). On the uniprocessor Pi 1, the implementation is trivial: disable interrupts, modify the counter, re-enable. On SMP, a more sophisticated approach using `LDREX`/`STREX` would be needed.

⟨*function* `arch_xinc` *(raspberry pi1)(arm)* 76a⟩≡ (972a)
```
void
arch_xinc(long *p)
{
    int s;

    s = arch_splhi();
    ++*p;
    arch_splx(s);
}
```

⟨*function* `arch_xdec` *(raspberry pi1)(arm)* 76b⟩≡ (972a)
```
long
arch_xdec(long *p)
{
    int s, v;

    s = arch_splhi();
    v = --*p;
    arch_splx(s);
    return v;
}
```

4.5 Spin `Lock`

A `Lock` (spin lock) is built on top of `arch_tas()`. Unlike `arch_splhi()`, a spin lock provides mutual exclusion across CPUs, not just against interrupts on one CPU. The caller spins in a tight loop until the lock becomes available. Spin locks are appropriate for very short critical regions (a few lines of code) where the wait is expected to be brief. On a uniprocessor, a spin lock still serves a purpose: while a process holds a lock, the kernel prevents it from being scheduled out (via `Proc.nlocks`), which effectively protects the critical region from other processes. However, a spin lock on a uniprocessor does not protect against interrupts—for that, `ILock` (next section) is needed. Note that `Lock` itself is portable, architecture-independent code (in `port/`)—it provides mutual exclusion on any architecture, uniprocessor or multiprocessor, by relying on the arch-specific `arch_tas()` primitive we just saw.

⟨*struct* `Lock` 76c⟩≡ (692)
```
// a.k.a Spinlock
struct Lock
{
    ulong key; // 0 when unset, 1 (or 0xDEADDEAD) when acquired, could be a bool

    ⟨Lock ilock fields 79b⟩
    ⟨Lock debugging fields 77a⟩
    ⟨Lock other fields 78a⟩
};
```

⟨*function* `lock` 76d⟩≡ (694a)
```
int
lock(Lock *l)
{
    int i;
```

```
    ⟨lock() other locals 78b⟩

    ⟨lock() start of lock, stat 661b⟩
    ⟨lock() increment nlocks 88g⟩
    if(arch_tas(&l->key) == 0){ // lock old value was 0, the lock was not held
        ⟨lock() got the lock, update other fields 78c⟩
        ⟨lock ifdef LOCKCYCLES 660c⟩
        return 0;
    }
    //else

    ⟨lock() decrement nlocks 88h⟩
    ⟨lock() when lock already held, stat 661c⟩

    // The lock was already held, need to spin
    for(;;){
    ⟨lock() in loop when lock already held, stat 661d⟩
        i = 0;
        while(l->key){
           ⟨lock() optional priority-inversion for real-time process 566f⟩
            if(i++ > 100000000){
                i = 0;
                lockloop(l, pc);
            }
        }
        // try again
        ⟨lock() increment nlocks 88g⟩
        if(arch_tas(&l->key) == 0){
            ⟨lock() got the lock, update other fields 78c⟩
            ⟨lock ifdef LOCKCYCLES 660c⟩
            return 1;
        }
        ⟨lock() decrement nlocks 88h⟩
    }
    return -1; // unreachable
}
```

Uses `lockloop()` 646a.

The connection between `lock()` and `arch_tas()` is straightforward:

```
lock(&l):
  arch_tas(&l->key)
       |
       v
  +----------+
  | key: 0   |  -> arch_tas returns 0, key set to 1: got it!
  +----------+
  | key: 1   |  -> arch_tas returns 1, key stays 1: spin...
  +----------+
```

For debugging, each `Lock` also records the program counter of the caller who acquired it (`pc`) and the CPU that holds it. Similarly, each `Proc` remembers its `lastlock`, making it easier to diagnose deadlocks or lock-ordering violations:

⟨`Lock` *debugging fields* 77a⟩≡ (76c) 88a▷

```
// for debugging, the caller who did the lock()
kern_addr pc;
```

⟨`Proc` *debugging fields* 77b⟩≡ (66) 79c▷

```
Lock* lastlock;
```

⟨Lock *other fields* 78a⟩≡ (76c)
```
// option<ref<Cpu>>, None when key = 0?
Cpu  *cpu; // not that used, only in iprintcanlock apparently
```

⟨lock() *other locals* 78b⟩≡ (76d)
```
ulong pc = getcallerpc(&l);
```

⟨lock() *got the lock, update other fields* 78c⟩≡ (76) 80d▷
```
if(up)
    up->lastlock = l;
l->pc = pc;
l->cpu = CPUS(cpu->cpuno); // pad's third bugfix
```
Uses CPUS 702 and up.

Unlocking is much simpler than locking—no atomicity is needed. The lock holder simply zeroes key:

⟨*function* unlock 78d⟩≡ (694a)
```
void
unlock(Lock *l)
{
    ⟨unlock ifdef LOCKCYCLES 660e⟩

    ⟨unlock() sanity checks 78e⟩
    l->cpu = nil;
    l->key = 0;
    ⟨unlock() after modified key, call coherence 534a⟩

    ⟨unlock() if delaysched 229c⟩
}
```

⟨unlock() *sanity checks* 78e⟩≡ (78d) 80f▷
```
if(l->key == 0)
    print("unlock: not locked: pc %#p\n", getcallerpc(&l));
```
Uses print.

⟨*function* canlock 78f⟩≡ (694a)
```
bool
canlock(Lock *l)
{
    ⟨lock() increment nlocks 88g⟩
    if(arch_tas(&l->key) != 0){ // lock old value != 0, lock was already held
       ⟨lock() decrement nlocks 88h⟩
        return false;
    }
    // else
    ⟨canlock() lock was not already held, update other fields 78g⟩
    ⟨lock ifdef LOCKCYCLES 660c⟩
    return true;
}
```
Uses false and true.

⟨canlock() *lock was not already held, update other fields* 78g⟩≡ (78f) 88c▷
```
if(up)
    up->lastlock = l;
l->pc = getcallerpc(&l);
l->cpu = CPUS(cpu->cpuno);
```
Uses CPUS 702 and up.

4.6 Uninterruptable `ILock`

An `ILock` is a spin lock that also disables interrupts before acquiring the lock. This is necessary when a data structure is accessed both from normal kernel code (syscalls) and from interrupt handlers. Without disabling interrupts, a deadlock can occur: a syscall acquires the lock, an interrupt fires on the same CPU, the interrupt handler tries to acquire the same lock, and spins forever because the interrupted syscall is the one holding it. `ILock` solves this by calling `arch_splhi()` before `arch_tas()`, and saving the old interrupt state in `Lock.sr` so `iunlock()` can restore it. In practice, `ILock` is used for structures accessed from interrupt handlers: the timer list, device driver queues, etc.

⟨*struct* `ILock` 79a⟩≡ (692)
```
typedef Lock ILock;
```

⟨`Lock` *ilock fields* 79b⟩≡ (76c)
```
bool_ushort isilock; // false when from lock(), true when from ilock()
ulong sr; // saved priority level when using ilock() to restore in iunlock()
```

⟨`Proc` *debugging fields* 79c⟩+≡ (66) ◁77b 81c▷
```
Lock  *lastilock;
```

⟨*function* `ilock` 79d⟩≡ (694a)
```
// To provide mutual exclusion with interrupt code and avoiding deadlock.
// By using arch_splhi() we disable interrupts while running the critical region
// code.
void
ilock(Lock *l)
{
    ulong x;
    ulong pc;

    pc = getcallerpc(&l);
    lockstats.locks++;

    x = arch_splhi();
    // no need to take care of up->nlock++ here, we have disabled interrupt
    // so no risk of getting scheduled
    if(arch_tas(&l->key) != 0){
        lockstats.glare++;
        /*
         * Cannot also check l->pc, l->cpu, or l->isilock here
         * because they might just not be set yet, or
         * (for pc and m) the lock might have just been unlocked.
         */
        for(;;){
            lockstats.inglare++;
            arch_splx(x);
            while(l->key)
                ;
            // let's try again
            x = arch_splhi();
            if(arch_tas(&l->key) == 0)
                goto acquire;
        }
    }
acquire:
    ⟨ilock() got the lock, update other fields 80a⟩
    ⟨lock ifdef LOCKCYCLES 660c⟩
}
```

Uses `lockstats` 661e.

⟨ilock() *got the lock, update other fields* 80a⟩≡ (79d) 88e▷
```
if(up)
    up->lastilock = l;
l->isilock = true;
l->sr = x;
l->pc = pc;
l->cpu = CPUS(cpu->cpuno);
```
Uses CPUS 702, true, and up.

⟨*function* iunlock 80b⟩≡ (694a)
```
void
iunlock(Lock *l)
{
    ulong sr;

    ⟨iunlock ifdef LOCKCYCLES 660f⟩
    ⟨iunlock() sanity checks 80c⟩
    sr = l->sr;
    l->cpu = nil;
    l->key = 0;
    ⟨iunlock() after modified key, call coherence 534b⟩
    ⟨iunlock() decrement ilockdepth 89e⟩
    if(up)
        up->lastilock = nil;
    arch_splx(sr);
}
```
Uses up.

⟨iunlock() *sanity checks* 80c⟩≡ (80b)
```
if(l->key == 0)
    print("iunlock: not locked: pc %#p\n", getcallerpc(&l));
if(!l->isilock)
    print("iunlock of lock: pc %#p, held by %#lux\n", getcallerpc(&l), l->pc);
if(arch_islo())
    print("iunlock while lo: pc %#p, held by %#lux\n", getcallerpc(&l), l->pc);
```
Uses print.

⟨lock() *got the lock, update other fields* 80d⟩+≡ (76) ◁78c 88b▷
```
l->isilock = false;
```
Uses false.

⟨canlock() *lock not held, update other fields* 80e⟩≡
```
l->isilock = false;
```

⟨unlock() *sanity checks* 80f⟩+≡ (78d) ◁78e 88d▷
```
if(l->isilock)
    print("unlock of ilock: pc %lux, held by %lux\n", getcallerpc(&l), l->pc);
```
Uses print.

4.7 Waiting QLock

A QLock (queuing lock, also known as a mutex) is a lock where the caller goes to sleep instead of spinning when the lock is not available. This is important for critical regions that may take a long time—for instance, I/O operations or file system lookups. Spinning for milliseconds would waste CPU time; sleeping lets the scheduler run other processes. The waiting processes form a queue (head/tail), linked through Proc.qnext. When the lock holder calls qunlock(), it wakes the first waiter in the queue. The internal coordination uses a Lock

(spin lock) to protect the queue itself, but the spin is only held for the few instructions needed to enqueue or dequeue—the actual wait is a sleep.

⟨*struct* QLock 81a⟩≡ (692)
```
// Kernel basic lock with Queue (renamed to avoid ambiguity with libc.h QLock)
// a.k.a Mutex
struct KQLock
{
    bool  locked;   /* flag */

    // list<ref<Proc>> (next = Proc.qnext)
    Proc  *head;    /* next process waiting for object */
    // option<ref<Proc>> (direct access to tail, queue)
    Proc  *tail;    /* last process waiting for object */

    ⟨KQLock debugging fields 81b⟩

    Lock  use;    /* to access Qlock structure */
};
```

⟨KQLock *debugging fields* 81b⟩≡ (81a)
```
kern_addr qpc;    /* pc of the holder */ // for debugging?
```

⟨Proc *debugging fields* 81c⟩+≡ (66) ◁79c
```
kern_addr qpc;    /* pc calling last blocking qlock */
```

⟨Proc *extra fields* 81d⟩≡ (66) 128f▷
```
// list<ref<Proc>> KQlock.head or RWLock.head (or Procalloc.free)
Proc  *qnext;   /* next process on queue for a QLock */
```

⟨Procstate *cases* 81e⟩≡ (67c) 83d▷
```
Queueing, // see qlock()
```

With the data structure in place, the `qlock()` function itself is straightforward. If the lock is free, take it immediately. Otherwise, add the current process to the wait queue and go to sleep:

⟨*function* qlock 81f⟩≡ (694b)
```
void
qlock(QLock *q)
{
    Proc *p;

    ⟨qlock() sanity checks 82b⟩
    lock(&q->use);
    ⟨qlock() start of qlock, stat 661h⟩
    if(!q->locked) {
        q->locked = true;
        q->qpc = getcallerpc(&q);
        unlock(&q->use);
        return;
    }
    // else
    ⟨qlock() sanity check up 82a⟩
    ⟨qlock() when qlock already held, stat 661i⟩

    // add_queue(q, up)
    p = q->tail;
    if(p == nil)
        q->head = up;
    else
        p->qnext = up;
```

```
        q->tail = up;
        // up->qnext could be non nil before? no otherwise that means
        // the process is already waiting for a lock and so had no occasion
        // to run another qlock() instruction.
        up->qnext = nil;

        up->state = Queueing;
        up->qpc = getcallerpc(&q);
        unlock(&q->use);

        // switch to another process!
        sched();
        // will resume here when another process unlock() the lock and ready() us
        q->qpc = getcallerpc(&q);
    }
```

Uses Queueing 81e, lock() 76d, sched, true, unlock() 78d, and up.

⟨qlock() *sanity check up* 82a⟩≡ (81f)

```
    if(up == nil)
        panic("qlock");
```

Uses up.

⟨qlock() *sanity checks* 82b⟩≡ (81f) 89b▷

```
    if(q->use.key == 0x55555555) // dead code??
        panic("qlock: q %#p, key 5*\n", q);
```

⟨*function* qunlock 82c⟩≡ (694b)

```
    void
    qunlock(QLock *q)
    {
        Proc *p;

        lock(&q->use);
        ⟨qunlock() sanity checks 82d⟩

        p = q->head;
        if(p){
            // dequeue(q)
            q->head = p->qnext;
            if(q->head == nil)
                q->tail = nil;

            unlock(&q->use);
            ready(p);
        }else{
            q->locked = false;
            q->qpc = nilptr;
            unlock(&q->use);
        }
    }
```

Uses false, lock() 76d, nilptr, ready, and unlock() 78d.

⟨qunlock() *sanity checks* 82d⟩≡ (82c)

```
    if(!q->locked)
        print("qunlock called with qlock not held, from %#p\n",
            getcallerpc(&q));
```

Uses print.

⟨*function* canqlock 83a⟩≡ (694b)
```
bool
canqlock(QLock *q)
{
    if(!canlock(&q->use))
        return false;
    if(q->locked){
        unlock(&q->use);
        return false;
    }
    // else

    q->locked = true;
    q->qpc = getcallerpc(&q);
    unlock(&q->use);
    return true;
}
```
Uses canlock() 78f, false, true, and unlock() 78d.

4.8 Readers/Writer RWLock

An RWlock allows multiple simultaneous readers or a single exclusive writer. This is useful for data structures that are read frequently but modified rarely, such as mount tables (Mhead[701b]), process groups (Pgrp[701b]), and environment groups (Egrp[701b]). The implementation is subtle: it must avoid starving either readers or writers, and like QLock, waiters sleep rather than spin.

⟨*struct* RWlock 83b⟩≡ (692)
```
struct RWlock
{
    int readers;  /* number of readers */
    bool writer;   /* have a writer? */

    // list<ref<Proc>> (next = Proc.qnext)
    Proc  *head;    /* list of waiting processes */
    // option<ref<Proc>> (direct access to tail, queue)
    Proc  *tail;
    // option<ref<Proc>>
    Proc  *wproc;   /* writing proc */

    ⟨RWlock debugging fields 83c⟩
    // Extra
    Lock  use;
};
```

⟨RWlock *debugging fields* 83c⟩≡ (83b)
```
kern_addr wpc;    /* pc of writer */
```

Like QLock, waiting for a read lock or a write lock puts the process into a dedicated state so the scheduler knows why it is blocked:

⟨Procstate *cases* 83d⟩+≡ (67c) ◁81e 85a▷
```
QueueingR, // see rlock()
```

⟨*function* rlock 83e⟩≡ (694b)
```
void
rlock(RWlock *q)
{
    Proc *p;
```

```
    lock(&q->use);
    rwstats.rlock++;
    if(!q->writer && q->head == nil){
        /* no writer, go for it */
        q->readers++;
        unlock(&q->use);
        return;
    }
    // else, a writer or a waiting queue??

    ⟨rlock() sanity check up 84a⟩
    rwstats.rlockq++;

    // add_queue(q, up)
    p = q->tail;
    if(p == nil)
        q->head = up;
    else
        p->qnext = up;
    q->tail = up;
    up->qnext = nil;

    up->state = QueueingR;
    unlock(&q->use);
    sched();
    // will resume here when another process unlock() the lock and ready() us
}
```

Uses QueueingR 83d, lock() 76d, rwstats 661g, sched, unlock() 78d, and up.

⟨rlock() *sanity check up* 84a⟩≡ (83e)

```
if(up == nil)
    panic("rlock");
```

Uses up.

⟨*function* runlock 84b⟩≡ (694b)

```
void
runlock(RWlock *q)
{
    Proc *p;

    lock(&q->use);
    p = q->head;
    if(--(q->readers) > 0 || p == nil){
        unlock(&q->use);
        return;
    }
    // else, p != nil && q->readers == 0

    /* start waiting writer */
    if(p->state != QueueingW)
        panic("runlock");

    // dequeue(q)
    q->head = p->qnext;
    if(q->head == nil)
        q->tail = nil;

    q->writer = true;
    unlock(&q->use);
    ready(p);
```

```
}
```

Uses QueueingW 85a, lock() 76d, ready, true, and unlock() 78d.

Writers wait separately from readers:

⟨Procstate *cases* 85a⟩+≡ (67c) ◁83d 129c▷
```
QueueingW, // see wlock()
```

⟨*function* wlock 85b⟩≡ (694b)
```
void
wlock(RWlock *q)
{
    Proc *p;

    lock(&q->use);
    rwstats.wlock++;
    if(q->readers == 0 && !q->writer){
        /* noone waiting, go for it */
        q->wpc = getcallerpc(&q);
        q->wproc = up;
        q->writer = true;
        unlock(&q->use);
        return;
    }
    // else, already a writer or readers

    /* wait */
    ⟨wlock() sanity check up 85c⟩
    rwstats.wlockq++;

    // add_queue(q, up)
    p = q->tail;
    if(p == nil)
        q->head = up;
    else
        p->qnext = up;
    q->tail = up;
    up->qnext = nil;

    up->state = QueueingW;
    unlock(&q->use);
    sched();
}
```

Uses QueueingW 85a, lock() 76d, rwstats 661g, sched, true, unlock() 78d, and up.

⟨wlock() *sanity check up* 85c⟩≡ (85b)
```
if(up == nil)
    panic("wlock");
```

Uses up.

⟨*function* wunlock 85d⟩≡ (694b)
```
void
wunlock(RWlock *q)
{
    Proc *p;

    lock(&q->use);
    p = q->head;
    if(p == nil){
        q->writer = false;
```

```
        unlock(&q->use);
        return;
    }
    if(p->state == QueueingW){
        /* start waiting writer */
        // dequeue(q)
        q->head = p->qnext;
        if(q->head == nil)
            q->tail = nil;

        unlock(&q->use);
        ready(p);
        return;
    }

    if(p->state != QueueingR)
        panic("wunlock");

    /* waken waiting readers */
    while(q->head != nil && q->head->state == QueueingR){
        p = q->head;
        q->head = p->qnext;
        q->readers++;
        ready(p);
    }

    if(q->head == nil)
        q->tail = nil;
    q->writer = false;
    unlock(&q->use);
}
```

Uses QueueingR 83d, QueueingW 85a, false, lock() 76d, ready, and unlock() 78d.

⟨*function* canrlock 86⟩≡ (694b)

```
/* same as rlock but punts if there are any writers waiting */
bool
canrlock(RWlock *q)
{
    lock(&q->use);
    rwstats.rlock++;
    if(q->writer == false && q->head == nil){
        /* no writer, go for it */
        q->readers++;
        unlock(&q->use);
        return true;
    }
    unlock(&q->use);
    return false;
}
```

Uses false, lock() 76d, rwstats 661g, true, and unlock() 78d.

4.9 Atomic References and Counter

We already saw arch_xinc()/arch_xdec(), which atomically adjust a raw counter. Ref below wraps a counter with a Lock instead of using arch_xinc directly. This may seem redundant. However, the Lock does allow decref() to additionnally and atomically check for underflow and panic. But the main reason is likely historical:

`Ref` is portable code (in `port/`) that predates the arch-specific atomic operations, and using a `Lock` keeps it independent of any particular architecture's atomic primitives.

⟨*struct* `Ref` 87a⟩≡ (692)
```
// For reference counting shared things (e.g. a Page)
struct Ref
{
    long ref;

    Lock;
};
```

⟨*function* `incref` 87b⟩≡ (695a)
```
long
incref(Ref *r)
{
    long x;

    lock(r);
    x = ++r->ref;
    unlock(r);
    return x;
}
```
Uses `lock()` 76d and `unlock()` 78d.

⟨*function* `decref` 87c⟩≡ (695a)
```
long
decref(Ref *r)
{
    long x;

    lock(r);
    x = --r->ref;
    unlock(r);
    if(x < 0)
        panic("decref pc=%#p", getcallerpc(&r));
    return x;
}
```
Uses `lock()` 76d and `unlock()` 78d.

Many kernel objects (pages, channels, segments) embed a `Ref` for reference counting—the object is freed when `decref()` returns 0.

⟨*struct* `Counter` 87d⟩≡ (692)
```
typedef struct Ref Counter;
```
Uses `Ref` 701b.

4.10 Safeguards

Concurrency bugs are notoriously hard to find, so the kernel includes several runtime checks to catch common mistakes: unlocking from the wrong process, sleeping while holding a spin lock, or taking a qlock while interrupts are disabled. We already saw several safeguards earlier: `Lock.pc` and `Proc.lastlock` recording who acquired a lock, `Lock.cpu` tracking which processor holds it, and `splpc` in `Cpu` remembering who disabled interrupts. This section covers additional checks.

4.10.1 Lock.p

Each Lock records which process acquired it (in Lock.p). On unlock, the kernel checks that the same process is releasing it—a mismatch indicates a bug (for instance, a context switch that should not have happened while the lock was held):

⟨Lock *debugging fields* 88a⟩+≡ (76c) ◁77a 660d▷
```
// option<ref<Proc>>, None when key == 0
Proc  *p; // the process who did the locking should be the same unlocking
```

⟨lock() *got the lock, update other fields* 88b⟩+≡ (76) ◁80d
```
l->p = up;
```
Uses up.

⟨canlock() *lock was not already held, update other fields* 88c⟩+≡ (78f) ◁78g
```
l->p = up;
```
Uses up.

⟨unlock() *sanity checks* 88d⟩+≡ (78d) ◁80f
```
if(l->p != up)
    print("unlock: up changed: pc %#p, acquired at pc %lux, lock p %#p, unlock up %#p\n", getcallerpc(&l), l->p
```
Uses print and up.

⟨ilock() *got the lock, update other fields* 88e⟩+≡ (79d) ◁80a 89d▷
```
l->p = up;
```
Uses up.

4.10.2 Proc.nlocks

Proc.nlocks counts how many spin locks the current process holds. This is crucial: a process holding a spin lock must not be scheduled out, because the lock protects a critical region that must complete without interruption. The scheduler checks nlocks and delays rescheduling until all locks are released. Similarly, qlock() checks nlocks and prints a warning if a process tries to sleep while holding a spin lock—that would deadlock, because no other process could run to release the lock. Note that nlocks itself uses arch_xinc()/arch_xdec() (via inccnt()/deccnt()) rather than incref()/decref(), because the latter use lock()/unlock()—which would modify nlocks, creating a circular dependency.

⟨Proc *synchronization fields* 88f⟩≡ (66) 233b▷
```
// As long as the current process hold spinlocks (to kernel data structures),
// we will not schedule another process in sched() or unlock();
// only the last unlock will eventually cause a rescheduling.
Ref nlocks;   /* number of locks held by proc */
```

⟨lock() *increment nlocks* 88g⟩≡ (78f 76)
```
if(up)
    inccnt(&up->nlocks);    /* prevent being scheded */
```
Uses inccnt() 88i and up.

⟨lock() *decrement nlocks* 88h⟩≡ (78f 76)
```
if(up)
    deccnt(&up->nlocks);
```
Uses deccnt() 89a and up.

⟨*function* inccnt 88i⟩≡ (694a)
```
// See also ref.c incref() and decref(), but we can't use them here as they
// themselves rely on lock() and unlock().
static void
inccnt(Ref *r)
{
    arch_xinc(&r->ref);
}
```

⟨*function* `deccnt` 89a⟩≡ (694a)
```
static int
deccnt(Ref *r)
{
    int x;

    x = arch_xdec(&r->ref);
    if(x < 0)
        panic("deccnt pc=%#p", getcallerpc(&r));
    return x;
}
```

⟨`qlock()` *sanity checks* 89b⟩+≡ (81f) ◁82b 89f▷
```
if(up != nil && up->nlocks.ref)
    print("qlock: %#p: nlocks %lud\n", getcallerpc(&q), up->nlocks.ref);
```
Uses `print` and `up`.

4.10.3 `Cpu.ilockdepth`

`Cpu.ilockdepth` counts how many `ILock`s are held on this CPU. Like `nlocks`, it is checked in `qlock()`: sleeping while holding an `ILock` (with interrupts disabled) would deadlock the entire CPU, because the timer interrupt that drives the scheduler could not fire. No atomic operation is needed here because interrupts are already disabled when `ilockdepth` is modified.

⟨`Cpu` *other fields* 89c⟩≡ (64a) 224e▷
```
int ilockdepth;
```

⟨`ilock()` *got the lock, update other fields* 89d⟩+≡ (79d) ◁88e
```
cpu->ilockdepth++;
```

⟨`iunlock()` *decrement ilockdepth* 89e⟩≡ (80b)
```
cpu->ilockdepth--;
```

⟨`qlock()` *sanity checks* 89f⟩+≡ (81f) ◁89b
```
if(cpu->ilockdepth != 0)
    print("qlock: %#p: ilockdepth %d\n", getcallerpc(&q), cpu->ilockdepth);
```
Uses `print`.

4.11 Synchronization

The locking primitives above provide mutual exclusion: preventing two threads from accessing the same data at the same time. But they do not solve synchronization: how does a process wait for something to happen (e.g., a disk read to complete, a pipe to have data)? For that, the kernel needs a way to put a process to sleep and wake it up later.

4.11.1 Rendez-vous

The fundamental synchronization mechanism in the kernel is `sleep()`/`wakeup()`, built around the `Rendez` structure (a "rendezvous point"). A process calls `sleep()` with a condition function; if the condition is not yet true, the process is put to sleep. Later, when another process or an interrupt handler makes the condition true, it calls `wakeup()` to resume the sleeper. The full code is in Chapter 6.

4.11.2 Producer/Consumer Queue

The I/O `Queue` structure (Chapter 16) uses `sleep()`/`wakeup()` to implement producer/consumer synchronization: a reader sleeps when the queue is empty, a writer sleeps when it is full, and each wakes the other.

4.11.3 Semaphore

The kernel also provides semaphores (`SEMACQUIRE`/`SEMRELEASE` syscalls), but these are primarily for user-level synchronization. The C library's `Lock` and `QLock` (in `libcore/libc/port/lock.c`) are implemented on top of these kernel semaphores. Inside the kernel itself, the spin locks and qlocks described above are used instead.

Part I

Virtualizing the Machine

The kernel's central job is to give each process the illusion of having the entire machine to itself. This part covers the mechanisms that make this illusion work: virtual memory (page tables and page faults) backed by a physical page allocator, process creation and management, interrupt and syscall handling (the boundary between the virtual and real machines), scheduling (virtualizing the CPU so each process gets its turn), time keeping, and finally the boot sequence that sets all of this up.

Chapter 5

Memory

From the user's point of view, memory is simple: call `brk()` (or `malloc()`) and get memory. Behind this simple interface lies a hierarchy of allocators, each solving a different problem.

⟨`systab` *memory syscalls* 92⟩≡ (57b)

```
[BRK]       sysbrk,
```

Uses `sysbrk()` 123a.

The C prototype is:

```
int   brk(void *addr);     // set heap end to addr
```

5.1 Overview

Memory management has a bootstrapping problem: you need memory to track which memory is free, but you do not have an allocator yet. The kernel solves this by building allocators in layers, each bootstrapped from the previous:

0. The kernel binary's data section contains statically allocated allocator metadata. The boot code also detects physical memory here and records it in `Conf.mem`.

1. `xalloc()`[95d] carves large structures from those banks. From here the hierarchy forks.

2a. On the kernel side, `malloc()`[103c] provides fine-grained allocation from a `Pool` backed by `xalloc`.

2b. On the user side, `palloc`[109b] manages physical pages. User `brk()` takes pages from `palloc`; user-space `malloc()` (in libc) subdivides them.

```
Layer 2:  kernel malloc()        palloc
          (Pool)                 (page allocator)
             |                      |
        kernel side            user brk()
                                    |
                               user malloc()
                               (in libc)
                                    |
                               user side

Layer 1:          xalloc
             (allocates Proc[], Page[],
              Pool arenas, from Confmem)
```

```
Layer 0:  kernel binary data section
          (xlists, conf, devtab, etc.)
          edata/end mark free memory start

Hardware: physical RAM (detected by confinit)
```

The following sections present each layer in detail.

5.2 Long-lived structure allocator: `xalloc()`

`xalloc()` is the bottom-level dynamic allocator. It manages the raw physical memory banks from `Conf.mem` using a simple first-fit, non-compacting algorithm. Its own metadata—the `xlists` global, containing the `Hole[128]` array—lives in the kernel's data section (layer 0), so it needs no memory allocation to get started. `xalloc` is called only during initialization to allocate a few large, long-lived structures—the `Proc` array, the `Page`[701b] array, and the `malloc` pool—that will never be freed. This simplicity is deliberate: the fixed-size hole array cannot handle heavy fragmentation, but it does not need to.

5.2.1 Data structures

`Xalloc`

The `Xalloc` structure manages free memory as a sorted linked list of `Hole`s.

⟨*struct* `Xalloc` 93a⟩≡ (855)

```
// Long lived data structure allocator (singleton)
struct Xalloc
{
    // array<Hole> where each Hole is linked to another hole
    Hole  hole[Nhole];

    // list<ref<Hole>> (next = Hole.next) list of free hole entries (addr=top=size=0)
    Hole* unused_slots;

    // list<ref<Hole>> (next = Hole.next) memory holes, sorted by their top addr
    Hole* sorted_holes;

    // extra
    Lock;
};
```

⟨`Hole` *extra fields* 93b⟩≡ (94)

```
Hole* next; // list<ref<Hole>> of Xalloc.sorted_holes or Xalloc.unused_slots
```

⟨*constants holes* 93c⟩≡ (855)

```
enum
{
    Nhole   = 128,
    ⟨constant Magichole 95c⟩
};
```

Each `Hole` describes a contiguous region of free memory (between `addr` and `top`). The `Hole` entries themselves are pre-allocated in a fixed-size array (`hole[Nhole]`), and those not currently describing a free region are kept in a separate "unused slots" list.

⟨*struct* `Hole` 94⟩≡ (855)

```
struct Hole
{
    // between addr and top the memory is free, this is the "hole"
    phys_addr addr;
    phys_addr top;
    ulong size; // top - addr (in bytes)

    // Extra
    ⟨Hole extra fields 93b⟩
};
```

Here is the layout after initialization (one memory bank) and after two allocations:

```
After xinit() with one Confmem bank:

  hole[0]:  addr=A  top=B  size=B-A   (the whole free bank)
  hole[1..127]: unused

  sorted_holes --> hole[0] --> nil
  unused_slots --> hole[1] --> hole[2] --> ... --> hole[127] --> nil

  Physical memory:
       B  +---------------------------+
          |     free (one big hole)   |
       A  +---------------------------+

After xalloc(s1) and xalloc(s2):

  hole[0]:  addr=A+s1+s2  top=B  size=B-A-s1-s2  (shrunk)

  sorted_holes --> hole[0] --> nil

  Physical memory:
       B  +---------------------------+
          |  free (hole[0], shrunk)   |
          +---------------------------+
          | Xhdr | data2 (s2 bytes)  |   <-- second xalloc
          +---------------------------+
          | Xhdr | data1 (s1 bytes)  |   <-- first xalloc
       A  +---------------------------+
```

Allocations grow upward from the bottom of the hole; the hole shrinks from below. Each allocation is prefixed with an `Xhdr` containing the size and a magic number (for corruption detection via `xmerge()`).

If the first allocation (s1) is later freed via `xfree()`, the sorted hole list would then contain two holes—the gap left by s1 and the original remaining free space at the top—which is why `xalloc` needs a list of holes rather than a single pointer.

`xlists`

There is a single global instance of `Xalloc`, statically allocated in the kernel's data section—this is layer 0, the bootstrap that makes everything else possible:

⟨*global* `xlists` 95a⟩≡ (859c)
```
static Xalloc   xlists;
```

`Xhdr`

Each block returned by `xalloc()` is preceded by an `Xhdr` header that records its size and a magic number. The magic number (`0x484F4C45`, which spells "HOLE" in ASCII) lets `xmerge()` detect memory corruption when a block is freed:

⟨*struct* `Xhdr` 95b⟩≡ (855)
```
// What is the connection with Hole? A Hole that get used will gets
// its top and size decremented, and this newly allocated part will describe
// a portion of used memory, and at this memory there will be a header
// and then just after the actual memory xalloc'ed by someone
struct Xhdr
{
    // bookkeeping area
    ulong size;
    ulong magix;

    char  data[]; // memory pointer returned by xalloc
};
```

⟨*constant* `Magichole` 95c⟩≡ (93c)
```
Magichole = 0x484F4C45,     /* HOLE */
```

5.2.2 `xalloc()`

`xalloc()` walks the sorted hole list looking for the first hole large enough (first-fit). It then carves the requested amount from the bottom of the hole, prepends an `Xhdr`, and shrinks the hole. If the hole is fully consumed, it is moved back to the unused slots list.

⟨*function* `xalloc` 95d⟩≡ (859c)
```
kern_vp
xalloc(ulong size)
{
    return xallocz(size, true);
}
```

Uses `true` and `xallocz()` 95e.

⟨*function* `xallocz` 95e⟩≡ (859c)
```
kern_vp
xallocz(ulong size, bool zero)
{
    Xhdr *p;
    Hole *h, **l;

    /* add room for magix & size overhead, round up to nearest vlong */
    size += BY2V + offsetof(Xhdr, data[0]);
    size &= ~(BY2V-1);

    ilock(&xlists);
    l = &xlists.sorted_holes;
    for(h = *l; h; h = h->next) {
```

```
        // found an appropriate hole
        if(h->size >= size) {
            p = (Xhdr*)KADDR(h->addr);
            h->addr += size; // shrink towards top
            h->size -= size;

            // This hole is now fully used (which is rare because one
            // rarely does an xalloc with the remaining size of a hole).
            // We can put it back in the list of free hole entries.
            if(h->size == 0) {
                *l = h->next;
                h->next = xlists.unused_slots;
                xlists.unused_slots = h;
            }

            iunlock(&xlists);
            if(zero)
                memset(p, 0, size);
            p->magix = Magichole;
            p->size = size;
            return p->data;
        }
        l = &h->next;
    }
    iunlock(&xlists);
    return nil; // panic?
}
```

Uses Magichole 94, ilock() 79d, iunlock() 80b, offsetof, and xlists-1849 95a.

5.2.3 xfree()

xfree()[96a] returns memory to the xalloc free list. It is rarely used in the kernel—most xalloc'd memory (page tables, device registers) is allocated once and never freed. The main caller is intrdisable(), which releases interrupt handler descriptors when a device is detached.

⟨*function* xfree 96a⟩≡ (859c)

```
void
xfree(kern_vp p)
{
    Xhdr *x;

    x = (Xhdr*)((ulong)p - offsetof(Xhdr, data[0]));
    ⟨xfree() sanity check magix mark 96b⟩
    xhole(PADDR((kern_addr)x), x->size);
}
```

Uses offsetof and xhole() 97.

⟨xfree() *sanity check magix mark* 96b⟩≡ (96a)

```
if(x->magix != Magichole) {
    xsummary();
    panic("xfree(%#p) %#ux != %#lux", p, Magichole, x->magix);
}
```

Uses Magichole 94 and xsummary() 643a.

hole() is the inverse of xalloc()[95d]: it adds a region of free memory to the sorted hole list. It tries to merge with adjacent holes (if the new region's bottom touches an existing hole's top, or vice versa) to avoid

fragmentation. This is also the function called by `xinit()`[282] to seed the allocator with the initial `Confmem` banks.

⟨*function* `xhole` 97⟩≡ (859c)

```
void
xhole(phys_addr addr, ulong size)
{
    phys_addr top;
    Hole *h, *c, **l;

    if(size == 0)
        return;

    top = addr + size;
    ilock(&xlists);
    l = &xlists.sorted_holes;
    for(h = *l; h; h = h->next) {
        if(h->top == addr) {
            h->size += size;
            h->top = h->addr+h->size;
            c = h->next;
            if(c && h->top == c->addr) {
                h->top += c->size;
                h->size += c->size;
                h->next = c->next;
                c->next = xlists.unused_slots;
                xlists.unused_slots = c;
            }
            iunlock(&xlists);
            return;
        }
        if(h->addr > addr)
            break;
        l = &h->next;
    }
    if(h && top == h->addr) {
        h->addr -= size;
        h->size += size;
        iunlock(&xlists);
        return;
    }

    if(xlists.unused_slots == nil) {
        iunlock(&xlists);
        print("xfree: no free holes, leaked %lud bytes\n", size);
        return;
    }

    h = xlists.unused_slots;
    xlists.unused_slots = h->next;
    h->addr = addr;
    h->top = top;
    h->size = size;
    h->next = *l;
    *l = h;
    iunlock(&xlists);
}
```

Uses `ilock()` 79d, `iunlock()` 80b, `print`, and `xlists-1849` 95a.

5.2.4 xmerge()

`xmerge()`[98a] tries to coalesce two adjacent free blocks into one. It is used by the `malloc` pool allocator (described next) as a callback to reduce fragmentation when blocks are freed.

⟨*function* `xmerge` 98a⟩≡ (859c)
```
bool
xmerge(kern_vp vp, kern_vp vq)
{
    Xhdr *p, *q;

    p = (Xhdr*)(((ulong)vp - offsetof(Xhdr, data[0])));
    q = (Xhdr*)(((ulong)vq - offsetof(Xhdr, data[0])));

    ⟨xmerge() sanity check magix 98b⟩
    if((byte*)p+p->size == (byte*)q) {
        p->size += q->size;
        return true;
    }
    return false;
}
```
Uses `false`, `offsetof`, and `true`.

⟨`xmerge()` *sanity check magix* 98b⟩≡ (98a)
```
if(p->magix != Magichole || q->magix != Magichole) {
    ⟨xmerge() debug info when not magichole 98c⟩
    panic("xmerge(%#p, %#p) bad magic %#lux, %#lux",
        vp, vq, p->magix, q->magix);
}
```
Uses `Magichole` 94.

⟨`xmerge()` *debug info when not magichole* 98c⟩≡ (98b)
```
int i;
ulong *wd;
void *badp;

xsummary();
badp = (p->magix != Magichole? p: q);
wd = (ulong *)badp - 12;
for (i = 24; i-- > 0; ) {
    print("%#p: %lux", wd, *wd);
    if (wd == badp)
        print(" <-");
    print("\n");
    wd++;
}
```
Uses `Magichole` 94, `print`, and `xsummary()` 643a.

⟨*function* `xspanalloc` 98d⟩≡ (859c)
```
kern_vp
xspanalloc(ulong size, int align, ulong span)
{
    ulong a, v, t;
    a = (kern_addr)xalloc(size+align+span);
    ⟨xspanalloc() sanity check a 99⟩

    if(span > 2) {
        v = (a + span) & ~(span-1);
        t = v - a;
        if(t > 0)
```

```
            xhole(PADDR(a), t);
        t = a + span - v;
        if(t > 0)
            xhole(PADDR(v+size+align), t);
    }
    else
        v = a;

    if(align > 1)
        v = (v + align) & ~(align-1);

    return (kern_vp)v;
}
```

Uses xalloc() 95d and xhole() 97.

⟨xspanalloc() *sanity check a* 99⟩≡ (98d)

```
if(a == nilptr)
    panic("xspanalloc: %lud %d %lux", size, align, span);
```

Uses nilptr.

5.3 Memory pools

xalloc is too coarse for general use—it can only handle a small number of holes (128). The kernel needs a finer-grained allocator for the many small, short-lived structures it creates (and free) during normal operation (channel tables, path strings, etc.). This is provided by the Pool allocator, which manages arenas obtained from xalloc and subdivides them using a best-fit algorithm.

5.3.1 Pool

The Pool structure is defined in lib_core/include/pool.h, not in the kernel, because the same allocation machinery is shared between the kernel's malloc() and the user-space malloc() in libc—only the backing allocator differs (xalloc for the kernel, brk for user space). The full definition is shown below as a verbatim excerpt but see the LIBCORE book [Pad16c] for more information:

```
// from pool.h
struct Pool {
  char* name;
  ulong maxsize;

  ulong cursize;
  ulong curfree;
  ulong curalloc;

  ulong minarena; /* smallest size of new arena */
  ulong quantum;  /* allocated blocks should be multiple of */
  ulong minblock; /* smallest newly allocated block */

  void* freeroot; /* actually Free* */
  void* arenalist;  /* actually Arena* */

  void* (*alloc)(ulong);
  int (*merge)(void*, void*);
```

```
  void  (*move)(void* from, void* to);

  int flags;
  int nfree;
  int lastcompact;

  void  (*lock)(Pool*);
  void  (*unlock)(Pool*);
  void  (*print)(Pool*, char*, ...);
  void  (*panic)(Pool*, char*, ...);
  void  (*logstack)(Pool*);

  void* private;
};
```

The important fields are `alloc` (the callback to get more memory—`xalloc` for the kernel), `merge` (to coalesce freed blocks—`xmerge` for the kernel), `freeroot` (the free-block tree), `arenalist` (the list of memory regions obtained from `alloc`), and the `lock`/`unlock` callbacks for thread safety.

The `Private` structure holds the per-pool lock and a message buffer. The message buffer is needed because the pool code cannot print while holding the lock (it would risk deadlock if `print` itself tries to allocate), so messages are saved and printed after unlock:

⟨*pool.c struct Private* 100a⟩≡ (860b)

```
struct Private {
    Lock        lk;
    char        msg[256]; /* a rock for messages to be printed at unlock */
};
```

5.3.2 `plock()`

The pool's lock callback uses `ilock()` (not a regular `lock()`) because interrupt handlers sometimes need to allocate memory. If a regular lock were used, a syscall holding the pool lock could be interrupted, and the interrupt handler's allocation attempt would deadlock.

⟨*function* `plock` 100b⟩≡ (860b)

```
static void
plock(Pool *p)
{
    Private *pv;

    pv = p->private;
    ilock(&pv->lk);
    pv->lk.pc = getcallerpc(&p);
    pv->msg[0] = 0;
}
```

Uses `ilock()` 79d.

Note the use of `ilock()` (interrupt-disabling lock) rather than a plain `lock()`. This is necessary because interrupt handlers occasionally allocate memory. If an interrupt fires while `malloc` holds the pool lock, and the handler calls `malloc` too, it would deadlock. By disabling interrupts while the lock is held, `plock()` prevents this scenario. On SMP, `ilock()` also provides mutual exclusion across processors.

⟨*function* `punlock` 100c⟩≡ (860b)

```
static void
punlock(Pool *p)
{
```

```
    Private *pv;
    char msg[sizeof pv->msg];

    pv = p->private;
    if(pv->msg[0] == 0){
        iunlock(&pv->lk);
        return;
    }

    memmove(msg, pv->msg, sizeof msg);
    iunlock(&pv->lk);
    iprint("%.*s", sizeof pv->msg, msg);
}
```

Uses iprint and iunlock() 80b.

5.3.3 pmainmem

The two pool instances, pmainmem (for general kernel allocations, 4 MB) and pimagmem (for graphics image data, 16 MB, see GRAPHICS book [Pad16d]), have their Pool structures statically allocated in the data section—another layer 0 bootstrap (the pool structures themselves, not the memory they manage).

Their alloc callback is xalloc, so when the pool needs more memory it asks xalloc for a new arena. The 4 MB maxsize is a cap, not a reservation—Pool only requests arenas as needed, leaving the remaining xalloc memory available for other direct allocations (the Proc array, the Page array, etc.). Here is how the kernel memory looks after initialization:

```
xalloc-managed kernel memory (kernelpercent of RAM):

            +--------------------------+
            | free (xalloc hole)       |
            +--------------------------+
            | pmainmem arena 2         |  (Pool, on demand)
            +--------------------------+
            | pimagmem arena 1         |  (Pool, on demand)
            +--------------------------+
            | pmainmem arena 1         |  (Pool, on demand)
            +--------------------------+
            | Page array               |  (direct xalloc)
            +--------------------------+
            | Proc array (~6 MB)       |  (direct xalloc)
            +--------------------------+
```

Arenas from different pools and direct xalloc calls can interleave—the order depends on when each allocation happens during boot and normal operation. Arenas are never returned to xalloc (no xfree call), so the pools only grow, which avoids fragmentation in xalloc's hole list.

⟨*global* pmainmem 101⟩≡ (860b)

```
static Private pmainpriv;
static Pool pmainmem = {
    .name=  "Main",
    .maxsize=   4*MB,
    .minarena=  128*KB,
    .quantum=   32,

    .alloc= xalloc,
```

```
    .merge= xmerge,

    .flags= POOL_TOLERANCE,

    .lock=plock,
    .unlock= punlock,

    .print= poolprint,
    .panic= ppanic,

    .private=   &pmainpriv,
};
```

Uses KB 867a, MB 867a, plock() 100b, pmainpriv-1838 101, poolprint() 103a, ppanic() 103b, punlock() 100c, xalloc() 95d, and xmerge() 98a.

⟨*global* pimagmem 102a⟩≡ (860b)

```
static Private pimagpriv;
static Pool pimagmem = {
    .name=  "Image",
    .maxsize=   16*MB,
    .minarena=  2*MB,
    .quantum=   32,

    .alloc= xalloc,
    .merge= xmerge,

    .flags= 0,

    .lock= plock,
    .unlock= punlock,

    .print= poolprint,
    .panic= ppanic,

    .private=   &pimagpriv,
};
```

Uses MB 867a, pimagpriv-1840 102a, plock() 100b, poolprint() 103a, ppanic() 103b, punlock() 100c, xalloc() 95d, and xmerge() 98a.

⟨*global mainmem and imagmem* 102b⟩≡ (860b)

```
// exported in include/pool.h, defined here!
Pool*   mainmem = &pmainmem;
Pool*   imagmem = &pimagmem;
```

Uses imagmem 102b, mainmem 102b, pimagmem-1841 102a, and pmainmem-1839 101.

The `maxsize` values above (4 MB for `mainmem`, 16 MB for `imagmem`) are just the static defaults compiled into the binary. At boot time, `arch__confinit()`[276d] overwrites both with `kmem`, the kernel memory left after subtracting space for the `Page`, `Proc`, and `KImage` arrays. On a terminal (non-cpuserver), `imagmem->maxsize` is also set to `kmem`, so both pools share the same cap and the dynamic allocator balances the load between general kernel allocations and graphics images.

5.3.4 poolprint()

`poolprint()`[103a] and `ppanic()`[103b] are the pool's diagnostic callbacks. They are called by the pool allocator code (in libc) when it detects corruption or wants to log warnings. Because the pool lock is held when these are called, they cannot print directly—`print()` calls `write()` which may acquire locks internally, risking deadlock.

Instead, `poolprint()` uses `vseprint()` (which does not allocate or take locks) to append to the message buffer in `Private`, and `punlock()` prints it after releasing the lock:

⟨*function* `poolprint` 103a⟩≡ (860b)

```
/*
 * because we can't print while we're holding the locks,
 * we have the save the message and print it once we let go.
 */
static void
poolprint(Pool *p, char *fmt, ...)
{
    va_list v;
    Private *pv;

    pv = p->private;
    va_start(v, fmt);
    vseprint(pv->msg+strlen(pv->msg), pv->msg+sizeof pv->msg, fmt, v);
    va_end(v);
}
```

⟨*function* `ppanic` 103b⟩≡ (860b)

```
static void
ppanic(Pool *p, char *fmt, ...)
{
    va_list v;
    Private *pv;
    char msg[sizeof pv->msg];

    pv = p->private;
    va_start(v, fmt);
    vseprint(pv->msg+strlen(pv->msg), pv->msg+sizeof pv->msg, fmt, v);
    va_end(v);
    memmove(msg, pv->msg, sizeof msg);
    iunlock(&pv->lk);
    panic("%s", msg);
}
```

Uses `iunlock()` 80b.

5.4 Kernel `malloc()`

The kernel's `malloc()` and `free()` are thin wrappers around `poolalloc()`/`poolfree()` on the `mainmem` pool. Unlike user-space `malloc`, the kernel version always zeroes the memory (via `memset`) and can optionally record the caller's PC in a tag area for debugging (when `Npadlong` is nonzero).

5.4.1 `malloc()`

⟨*function* `malloc` 103c⟩≡ (861b)

```
kern_vp
malloc(ulong size)
{
    kern_vp v;

    v = poolalloc(mainmem, size + Npadlong*sizeof(ulong));
    if(v == nil)
        return nil;
    ⟨malloc() if Npadlong 105c⟩
    memset(v, 0, size);
```

```
    return v;
}
```

Uses Npadlong-1813 861b and mainmem 102b.

⟨*function* free 104a⟩≡ (861b)

```
void
free(kern_vp v)
{
    if(v != nil)
        poolfree(mainmem, (ulong*)v-Npadlong);
}
```

Uses Npadlong-1813 861b and mainmem 102b.

5.4.2 Sleeping smalloc()

smalloc() ("sleeping malloc") is a variant of malloc() that never fails: if the pool is full, instead of returning nil, it sleeps for 100 ms and retries. This is useful for code paths where an allocation failure would be hard to recover from—for instance, when creating kernel data structures during a syscall where the caller expects success. The trade-off is that smalloc() cannot be called from interrupt context (sleeping in an interrupt handler would deadlock the system).

⟨*function* smalloc 104b⟩≡ (861b)

```
// non failing malloc! will repeat until it can
kern_vp
smalloc(ulong size)
{
    kern_vp v;

    for(;;) {
        v = poolalloc(mainmem, size + Npadlong*sizeof(ulong));
        if(v != nil)
            break;
        // else
        tsleep(&up->sleepr, returnfalse, 0, 100);
    }
    ⟨smalloc() if Npadlong 106c⟩
    memset(v, 0, size); // clear
    return v;
}
```

Uses Npadlong-1813 861b, mainmem 102b, returnfalse() 104c, tsleep, and up.

⟨*function* returnfalse 104c⟩≡ (705c)

```
// usually used as default callback for sleep/tsleep
bool
returnfalse(void*)
{
    return false;
}
```

Uses false.

5.4.3 mallocxxx()

⟨*function* mallocz 104d⟩≡ (861b)

```
kern_vp
mallocz(ulong size, bool clr)
{
```

```
    kern_vp v;

    v = poolalloc(mainmem, size+Npadlong*sizeof(ulong));
    ⟨mallocz() if Npadlong 106d⟩
    if(clr && v != nil)
        memset(v, 0, size);
    return v;
}
```

Uses Npadlong-1813 861b and mainmem 102b.

⟨*function* mallocalign 105a⟩≡ (861b)

```
kern_vp
mallocalign(ulong size, ulong align, long offset, ulong span)
{
    kern_vp v;

    v = poolallocalign(mainmem, size  +Npadlong*sizeof(ulong), align,
                               offset-Npadlong*sizeof(ulong), span);
    ⟨mallocalign() if Npadlong 106e⟩
    if(v)
        memset(v, 0, size);
    return v;
}
```

Uses Npadlong-1813 861b and mainmem 102b.

⟨*function* realloc 105b⟩≡ (861b)

```
kern_vp
realloc(kern_vp v, ulong size)
{
    kern_vp nv;

    if(v != nil)
        v = (ulong*)v - Npadlong;
    ⟨realloc() if Npadlong 106f⟩

    if(nv = poolrealloc(mainmem, v, size)){
       ⟨realloc() readjust nv if Npadlong 106g⟩
    }
    return nv;
}
```

Uses Npadlong-1813 861b and mainmem 102b.

5.4.4 malloc tags

When `Npadlong` is nonzero, `malloc()` reserves extra words before the returned pointer to store debugging tags: the PC of the caller who did the `malloc` and the PC of the last `realloc`. This lets the kernel (or a developer inspecting a crash dump) figure out who allocated a given block. The pool allocator in libc (see `lib_core/libc/port/pool.c`) provides the underlying bookkeeping; these tags are an additional layer on top.

⟨malloc() *if Npadlong* 105c⟩≡ (103c)

```
if(Npadlong){
    v = (ulong*)v+Npadlong;
    setmalloctag(v, getcallerpc(&size));
    setrealloctag(v, 0);
}
```

Uses Npadlong-1813 861b, setmalloctag() 106a, and setrealloctag() 106b.

⟨*function* setmalloctag 106a⟩≡ (861b)
```
void
setmalloctag(kern_vp v, kern_addr pc)
{
    kern_wp u;
    USED(v, pc); //??
    if(Npadlong <= MallocOffset || v == nil)
        return;
    u = v;
    u[-Npadlong+MallocOffset] = pc;
}
```
Uses MallocOffset-1814 861b and Npadlong-1813 861b.

⟨*function* setrealloctag 106b⟩≡ (861b)
```
void
setrealloctag(kern_vp v, kern_addr pc)
{
    kern_wp u;
    USED(v, pc);
    if(Npadlong <= ReallocOffset || v == nil)
        return;
    u = v;
    u[-Npadlong+ReallocOffset] = pc;
}
```
Uses Npadlong-1813 861b and ReallocOffset-1815 861b.

⟨smalloc() *if Npadlong* 106c⟩≡ (104b)
```
if(Npadlong){
    v = (ulong*)v+Npadlong;
    setmalloctag(v, getcallerpc(&size));
}
```
Uses Npadlong-1813 861b and setmalloctag() 106a.

⟨mallocz() *if Npadlong* 106d⟩≡ (104d)
```
if(Npadlong && v != nil){
    v = (ulong*)v+Npadlong;
    setmalloctag(v, getcallerpc(&size));
    setrealloctag(v, 0);
}
```
Uses Npadlong-1813 861b, setmalloctag() 106a, and setrealloctag() 106b.

⟨mallocalign() *if Npadlong* 106e⟩≡ (105a)
```
if(Npadlong && v != nil){
    v = (ulong*)v+Npadlong;
    setmalloctag(v, getcallerpc(&size));
    setrealloctag(v, 0);
}
```
Uses Npadlong-1813 861b, setmalloctag() 106a, and setrealloctag() 106b.

⟨realloc() *if Npadlong* 106f⟩≡ (105b)
```
if(Npadlong !=0 && size != 0)
    size += Npadlong*sizeof(ulong);
```
Uses Npadlong-1813 861b.

⟨realloc() *readjust nv if Npadlong* 106g⟩≡ (105b)
```
 nv = (ulong*)nv+Npadlong;
 setrealloctag(nv, getcallerpc(&v));
 if(v == nil)
     setmalloctag(nv, getcallerpc(&v));
```
Uses Npadlong-1813 861b, setmalloctag() 106a, and setrealloctag() 106b.

5.5 User `malloc()`

User-space `malloc()` is entirely in the C library (in `lib_core/libc/port/malloc.c`). It uses the same `Pool` allocator but backed by `brk()` instead of `xalloc()`. The kernel only sees page-granularity requests via `brk()`; the library subdivides those pages for the user.

5.6 `Page` and `palloc`

The previous allocators (`xalloc`, `Pool`, `malloc`) manage kernel memory. User memory is managed differently: the unit of allocation is a 4 KiB page, and the kernel tracks every physical page with a `Page` metadata structure. The `Palloc` structure maintains a free list of all available pages. When a user calls `brk()` or triggers a page fault, the kernel takes a page from `palloc` and maps it into the process's address space. Here is the relationship between the allocators:

```
Physical memory (one Confmem bank):

     +-------------------------------------------+
     | user pages (managed by palloc)            |
     |  each page has a Page struct in palloc.pages|
     +-------------------------------------------+
     | kernel malloc pool (Pool/mainmem)         |
     +-------------------------------------------+
     | Page[] array (xalloc'ed)                  |
     +-------------------------------------------+
     | Proc[] array (xalloc'ed)                  |
     +-------------------------------------------+  <-- end (linker symbol)
     | kernel bss                                |
     +-------------------------------------------+  <-- edata
     | kernel data:                              |
     |   xlists (Xalloc + Hole[128])  \          |
     |   pmainmem, pimagmem (Pool)     > layer 0  |
     |   conf, devtab, soc, etc.      /          |
     +-------------------------------------------+
     | kernel text                               |
0x8000 +-------------------------------------------+
     | L1/L2 tables, Cpu struct                  |
0x0  +-------------------------------------------+
```

The kernel data section (between `edata` and the text) contains the statically allocated globals that bootstrap the memory system: `xlists` with its 128-entry `Hole` array, the `pmainmem` and `pimagmem` pool structures, `conf`, `devtab`, etc. These are layer 0—they exist before any dynamic allocator runs. The `end` symbol marks where dynamic memory begins; `xalloc` starts carving from there.

5.6.1 `Page`

A `Page` is the metadata for one 4 KiB physical page. The kernel allocates one `Page` for every physical page available for user processes—this is a large array (`palloc.pages`), xalloc'ed during initialization. The key fields are `pa` (the physical address of the page, set once and never changed) and `ref` (the reference count, because

pages can be shared between processes via copy-on-write):

⟨*struct* Page 108a⟩≡ (855)
```
// Page metadata. We will allocate as many Page as to cover all physical memory
// available for the user. xalloc'ed in Palloc.pages
struct Page
{
    phys_addr pa;     /* Physical address in memory */
    user_addr va;     /* Virtual address for user */

    // Why not Ref? to save space (same reason for using 'char' below)
    // but that means needs to use Lock below to access this non-atomic ref.
    ushort  ref;      /* Reference count */ // Pages are shared!

    // bitset<enum<Modref>>
    char  modref;     /* Simulated modify/reference bits */

    ⟨Page other fields 190a⟩

    // extra
    Lock;
    ⟨Page extra fields 109a⟩
};
```

This structure must remain compact: there is one `Page` per physical page in the system, so if `Page` were 2 KiB, a third of memory would be wasted on bookkeeping for 4 KiB pages. In practice, `Page` is only a few dozen bytes.

The constants below are software modified/referenced bits (the ARM MMU does not provide hardware dirty bits like x86 does). `PG_REF` is set the first time a page is accessed (in `fixfault()`[185]); `PG_MOD` is set when a page is written.[1]

⟨*enum* modref 108b⟩≡ (855)
```
enum Modref
{
    PG_NOTHING = 0x00, // nothing

    PG_MOD    = 0x01,   /* software modified bit */
    PG_REF    = 0x02,   /* software referenced bit */
};
```

The swapper uses these bits to decide what to do when reclaiming a page: if `PG_MOD` is not set, the page is clean and can be discarded (or retrieved from the image cache) without writing it to swap. If `PG_REF` is set, the page was recently used and should be kept longer.

5.6.2 Palloc

The `Palloc` is a singleton that manages the free page list. Free pages are kept in a doubly-linked list (through `Page.next`/`Page.prev`), ordered so that the most recently used page is at the head and the least recently used at the tail. This LRU ordering is used by the swapper to decide which pages to evict first.

⟨*struct* Palloc 108c⟩≡ (855)
```
// Page Allocator (singleton)
struct Palloc
{
    // array<Page>, xalloc'ed in pageinit(), huge, cover physical user space
    Page  *pages; /* array of all pages */
```

[1] How can the kernel track user-space memory accesses? By initially mapping pages as invalid (or read-only for writes). When the user program touches the page, the MMU generates a page fault, giving the kernel a chance to set `PG_REF` or `PG_MOD` before mapping the page with the actual permissions.

```
    // list<ref<Page>> (next = Page.next), list of free pages
    Page  *head;      /* most recently used */
    // list<ref<Page>> (prev = Page.prev), list of free pages (backward)
    Page  *tail;      /* least recently used */

    ulong freecount;    /* how many pages on free list now */

    ⟨Palloc capacity fields 109c⟩
    ⟨Palloc other fields 190c⟩

    // extra
    Lock; // LOCK ORDERING: always do lock(&palloc); lock(&page)!!
    ⟨Palloc extra fields 210c⟩
};
```

Uses PGHLOG.

⟨Page *extra fields* 109a⟩≡ (108a) 190d▷

```
// list<ref<Page>> Palloc.head, or Proc.mmuused or Proc.mmufree, or in mfreeseg
Page  *next; /* Lru free list */
// list<ref<Page>> Palloc.tail
Page  *prev;
```

⟨*global* palloc 109b⟩≡ (859a)

```
// Page allocator
struct Palloc palloc;
```

Uses Palloc 701b.

Palloc has its own Pallocmem array, which looks redundant with Confmem. The difference is that Confmem describes all physical memory (including the kernel's portion), while Pallocmem describes only the user pages—the subset left after the kernel has taken its share via xalloc. The user field is the total count of user pages across all banks.

⟨Palloc *capacity fields* 109c⟩≡ (108c)

```
Pallocmem mem[4]; // = Conf.mem minus memory allocated for the kernel
// sum of mem.npage (which should be conf.upages)
ulong user;     /* how many user pages */
```

⟨*struct* Pallocmem 109d⟩≡ (855)

```
// memory banks for user memory, similar to Confmem (and RMap)
struct Pallocmem
{
    phys_addr base;
    ulong npage;
};
```

How much memory does all this bookkeeping consume? By default, confinit()X gives 80% of physical pages to user space and 20% to the kernel. On a 512 MiB Raspberry Pi, that is roughly 100 MiB for the kernel side: the Proc array (about 6 MiB for 2000 processes at 3 KiB each), the Page array (one Page struct per user page), the mainmem pool (4 MiB), the imagmem pool (16 MiB for graphics), etc. The remaining 400 MiB is managed by palloc as user pages.

5.6.3 newpage()

newpage() allocates a physical page from palloc. The clear flag requests the page be zeroed (needed for bss and stack pages). The s parameter is the segment the page belongs to—it may be set to nil on return if the

segment was invalidated while waiting for memory. The `va` is the virtual address where the page will be mapped:

⟨*constructor* `newpage` 110a⟩≡ (863d)
```
Page*
newpage(bool clear, Segment **s, user_addr va)
{
    Page *p;
    ⟨newpage() other locals 110b⟩

    lock(&palloc);

    ⟨newpage() loop waiting freecount ¿ highwater 210b⟩

    //when no color: p = palloc.head;
    ⟨newpage() find free page p, set color and ct 554i⟩

    pageunchain(p);

    lock(p);
    if(p->ref != 0)
        panic("newpage: p->ref %d != 0", p->ref);

    ⟨newpage() uncachepage 191c⟩
    p->ref = 1;
    p->va = va;
    p->modref = PG_NOTHING;
    ⟨newpage() set cachectl using ct 555b⟩

    unlock(p);
    unlock(&palloc);
    ⟨newpage() if clear 110c⟩
    return p;
}
```

The subtle part is the `if(s)/qunlock` dance: if the caller holds a segment lock (`*s != nil`), it must be released before sleeping (to avoid deadlock), and re-acquired after waking up. But while asleep, another process may have invalidated the segment, so `*s` is set to `nil` to signal the caller to retry.

The `Arch_KMap` is needed to temporarily map the physical page into the kernel's virtual address space so `memset()` can zero it. On ARM with less than 2 GiB of RAM, all physical memory is accessible via the `KZERO` offset, so this is trivial; on machines with more memory, a temporary mapping is required.

⟨`newpage()` *other locals* 110b⟩≡ (110a) 210a▷
```
Arch_KMap *k;
```

⟨`newpage()` *if clear* 110c⟩≡ (110a)
```
if(clear) {
    k = arch_kmap(p);
    memset((void*)VA(k), 0, BY2PG);
    arch_kunmap(k);
}
```

The following functions manage `palloc`'s doubly-linked free page list via the `prev`/`next` pointers in `Page` and the `head`/`tail` pointers in `Palloc`:

⟨*function* `pageunchain` 110d⟩≡ (863d)
```
// assumes palloc is held
static void
pageunchain(Page *p)
{
    if(canlock(&palloc))
        panic("pageunchain (palloc %p)", &palloc);
```

```
    // remove_queue(p, palloc);
    if(p->prev)
        p->prev->next = p->next;
    else
        palloc.head = p->next;
    if(p->next)
        p->next->prev = p->prev;
    else
        palloc.tail = p->prev;
    p->prev = p->next = nil;

    palloc.freecount--;
}
```

Uses canlock() 78f and palloc 109b.

⟨*destructor* putpage 111a⟩≡ (863d)

```
void
putpage(PageOrSwap *p)
{
    ⟨putpage if p is a swap address 206e⟩

    lock(&palloc);
    lock(p);

    if(p->ref == 0)
        panic("putpage");

    if(--p->ref > 0) {
        unlock(p);
        unlock(&palloc);
        return;
    }
    // else
    ⟨putpage if p has an image 191b⟩
    else
        pagechainhead(p);

    ⟨putpage wakeup process waiting for freepage if any 211a⟩
    unlock(p);
    unlock(&palloc);
}
```

Uses lock() 76d, pagechainhead() 111b, palloc 109b, and unlock() 78d.

⟨*function* pagechainhead 111b⟩≡ (863d)

```
// assumes palloc is held
void
pagechainhead(Page *p)
{
    if(canlock(&palloc))
        panic("pagechainhead");

    // add_head(p, palloc)
    if(palloc.head) {
        p->next = palloc.head;
        palloc.head->prev = p;
    }
    else {
        palloc.tail = p;
        p->next = nil;
```

```
    }
    palloc.head = p;
    p->prev = nil;

    palloc.freecount++;
}
```

Uses canlock() 78f and palloc 109b.

⟨*function* pagechaintail 112a⟩≡ (863d)

```
// assumes palloc is held
void
pagechaintail(Page *p)
{
    if(canlock(&palloc))
        panic("pagechaintail");

    // add_tail(p, palloc)
    if(palloc.tail) {
        p->prev = palloc.tail;
        palloc.tail->next = p;
    }
    else {
        palloc.head = p;
        p->prev = nil;
    }
    palloc.tail = p;
    p->next = nil;

    palloc.freecount++;
}
```

Uses canlock() 78f and palloc 109b.

5.7 Segment and Pagetable

A Segment represents a contiguous region of a process's virtual address space (text, data, bss, or stack). Each segment contains a two-level page directory that maps virtual addresses within the segment to physical Pages. These are the kernel's segment tables—separate from the ARM's hardware L1/L2 page tables, although the two must be kept in sync.

Why maintain segment tables when the hardware already has page tables? Because the hardware tables are architecture-specific (different format on ARM vs. x86), while Segment/Pagetable are portable. They also store information the hardware does not: reference counts (for shared pages and copy-on-write), swap addresses (for paged-out pages), and segment types (text vs. data vs. stack). The cost is extra memory, but the segment tables are sparse (only populated on demand via page faults), so the overhead is proportional to the pages actually used, not to the full address space.

⟨*enum* segtype 112b⟩≡ (855)

```
/* Segment types */
enum Segtype
{
    SG_TEXT   = 00,
    SG_DATA   = 01,
    SG_BSS    = 02,
    SG_STACK  = 03,
    ⟨Segtype other cases 556a⟩
    SG_TYPE   = 07,   /* Mask type of segment */
```

```
    ⟨Segtype other flags 120a⟩
};
```

⟨*global* sname 113a⟩≡ (740b)

```
// hash<enum<Segtype, string>>
static char *sname[]={
    [SG_TEXT]     = "Text",
    [SG_DATA]     = "Data",
    [SG_BSS]      = "Bss",
    [SG_STACK]    = "Stack",

    [SG_SHARED]   = "Shared",
    [SG_PHYSICAL] = "Phys"
};
```

5.7.1 Segment

Each segment tracks its virtual address range (`base`–`top`) and owns a `pagedir` array of `Pagetable`[701b] pointers:

⟨*struct* Segment 113b⟩≡ (855)

```
// smalloc'ed by newseg()
struct Segment
{
    // enum<Segtype>
    ushort  type;   /* segment type */

    user_addr base;   /* virtual base */
    user_addr top;    /* virtual top */
    ulong size;   /* size in pages */ // top - base / BY2PG

    // array<option<ref_own<Pagetable>>>, smalloc'ed (or smallpagedir alias)
    Pagetable **pagedir; // array of PAGEDIRSIZE max
    int pagedirsize; // nelem(pagedir)

    ⟨Segment other fields 117b⟩

    // extra
    Ref; // LOCK ORDERING: always do lock(img); lock(s) ??
    QLock lk; // FOR WHAT???
};
```

Here is the structure:

```
Proc
 |
 +-- seg[NSEG]:  array of Segment pointers
       |
       +-- seg[SG_TEXT]  --> Segment (base=UTZERO, top=...)
       +-- seg[SG_DATA]  --> Segment (base=..., top=...)
       +-- seg[SG_BSS]   --> Segment (base=..., top=...)
       +-- seg[SG_STACK] --> Segment (base=USTKTOP-USTKSIZE, top=USTKTOP)
                               |
                               +-- map[]:  array of Pagetable pointers
                                     |
                                     +-- Pagetable
                                           |
                                           +-- pages[]:  array of Page pointers
```

```
|
+-- Page (pa=..., va=...)
```

PAGEDIRSIZE is the maximum number of Pagetable pointers in a segment's page directory. Each Pagetable covers 1 MiB (256 pages × 4 KiB), so 1984 entries can map 1984 × 1 MiB ≈ 2 GiB—just under the user address space limit at KZERO. The value 1984 (not 2048) leaves room for the I/O region mapped at VIRTIO just below KZERO.

⟨*constant* PAGEDIRSIZE*(arm)* 114a⟩≡ (1005)
```
#define PAGEDIRSIZE 1984
```

5.7.2 Pagetable

A Pagetable maps 1 MiB of virtual address space (256 pages of 4 KiB each). It is the second level of the software page table—the Segment.map array is the first level. The first/last pointers optimize iteration: for a stack segment that uses only a few pages at the top of a large range, they avoid scanning hundreds of nil entries:

⟨*struct* Pagetable 114b⟩≡ (855)
```
// ptalloc'ed (malloc'ed)
struct Pagetable
{
    //array<option<ref<PageOrSwap>> will map 1M of memory
    PageOrSwap  *pagetab[PAGETABSIZE];

    //to avoid iterate over all entries in pagetab
    // ref<ref<Page>> in Pagetable.pages
    Page  **first;    /* First used entry */
    // ref<ref<Page>> in Pagetable.pages
    Page  **last;     /* Last used entry */
};
```

⟨*constant* PAGETABMAPMEM*(arm)* 114c⟩≡ (1005)
```
#define PAGETABMAPMEM   (1024*1024) // 1MB
```

⟨*constant* PAGETABSIZE*(arm)* 114d⟩≡ (1005)
```
#define PAGETABSIZE 256 // (PAGETABMAPMEM/BY2PG)
```

Here is how the constants relate:

```
Segment.map[0..PAGEDIRSIZE-1]      (up to 1984 entries)
     |
     +--[0]--> Pagetable.pagetab[0..255]  (PAGETABSIZE)
     |              |
     |              +--[0]--> Page (4 KiB = BY2PG)
     |              +--[1]--> Page
     |              ...
     |              +--[255]--> Page
     |         covers 256 * 4 KiB = 1 MiB (PAGETABMAPMEM)
     |
     +--[1]--> Pagetable (another 1 MiB)
     ...
     +--[1983]--> Pagetable
          covers 1984 * 1 MiB ~= 2 GiB (user address space)
```

⟨*constructor* ptalloc 115a⟩≡ (863d)
```
Pagetable*
ptalloc(void)
{
    Pagetable *new;

    new = smalloc(sizeof(Pagetable));
    new->first = &new->pagetab[PAGETABSIZE];
    new->last = new->pagetab;
    return new;
}
```
Uses smalloc() 104b.

Note the use of smalloc()—a sleeping variant of malloc() that blocks if memory is not available instead of returning nil.

⟨*destructor* freept 115b⟩≡ (863d)
```
void
freept(Segment *s, Pagetable *p)
{
    PageOrSwap **pte;
    ⟨freept() other locals 557b⟩

    switch(s->type&SG_TYPE) {
    ⟨freept() SG_PHYSICAL case 557c⟩
    default:
        for(pte = p->first; pte <= p->last; pte++)
            if(*pte) {
                putpage(*pte);
                *pte = nil;
            }
    }
    free(p);
}
```
Uses SG_TYPE, free() 104a, and putpage() 111a.

⟨*function* ptcpy 115c⟩≡ (863d)
```
Pagetable*
ptcpy(Pagetable *old)
{
    Pagetable *new;
    PageOrSwap **src, **dst;

    new = ptalloc();
    dst = &new->pagetab[old->first - old->pagetab];
    new->first = dst;
    for(src = old->first; src <= old->last; src++, dst++)
        if(*src) {
            ⟨ptcpy() if src is a swap page 207b⟩
            else {
                lock(*src);
                (*src)->ref++;
                unlock(*src);
            }
            new->last = dst;
            *dst = *src;
        }
    return new;
}
```
Uses lock() 76d, ptalloc() 115a, and unlock() 78d.

5.7.3 newseg()

We have seen newpage()[110a] for allocating physical pages, and the Segment and Pagetable[701b] data structures. Now we can see how a new segment is created:

newseg()[116a] creates a new Segment of the given type (text, data, bss, stack, or shared), starting at virtual address base and covering size pages. It is called by sysexec()X (to set up a new program's segments) and by sysbrk()[123a] (to extend the bss). The page directory is allocated but the actual Pagetables and Pages are not—they are populated lazily on page faults:

⟨*constructor* newseg 116a⟩≡ (864b)

```
Segment *
newseg(int type, user_addr base, ulong size)
{
    Segment *s;
    int pagedirsize;

    ⟨newseg() sanity check size 116b⟩
    s = smalloc(sizeof(Segment));

    s->ref = 1; // Segment.Ref.ref
    s->type = type;
    s->base = base;
    s->top = base+(size*BY2PG);
    s->size = size;

    ⟨newseg() sema initialization 474b⟩

    pagedirsize = ROUND(size, PAGETABSIZE)/PAGETABSIZE;

    ⟨newseg() if pagedirsize small 117d⟩
    else{
        pagedirsize *= 2; // room for growing
        if(pagedirsize > PAGEDIRSIZE)
            pagedirsize = PAGEDIRSIZE; // pad's first bugfix :)
        s->pagedir = smalloc(pagedirsize * sizeof(Pagetable*));
        s->pagedirsize = pagedirsize;
    }
    return s;
}
```

Uses ROUND 47b and smalloc() 104b.

Again smalloc() is used, so if the kernel is low on memory the caller will sleep until some is freed.

⟨newseg() *sanity check size* 116b⟩≡ (116a)

```
if(size > (PAGEDIRSIZE*PAGETABSIZE))
    error(Enovmem);
```

Uses Enovmem 664d.

⟨*destructor* putseg 116c⟩≡ (864b)

```
void
putseg(Segment *s)
{
    Pagetable **pde, **emap;
    ⟨putseg() other locals 201a⟩

    ⟨putseg() sanity check s 117a⟩
    ⟨putseg() if s has an image 201b⟩
    else
        lock(s);
```

```
    s->ref--;
    if(s->ref != 0) {
        unlock(s);
        return;
    }
    // else
    unlock(s);

    qlock(&s->lk);
    ⟨putseg() if s had an image 201c⟩

    emap = &s->pagedir[s->pagedirsize];
    for(pde = s->pagedir; pde < emap; pde++)
        if(*pde)
            freept(s, *pde);

    qunlock(&s->lk);

    ⟨putseg() if smallpagedir 117e⟩
    else
        free(s->pagedir);
    ⟨putseg() free profile 505e⟩
    free(s);
}
```

Uses `free()` 104a, `freept()` 115b, `lock()` 76d, `qlock()` 81f, `qunlock()` 82c, and `unlock()` 78d.

⟨putseg() *sanity check s* 117a⟩≡ (116c)

```
if(s == nil)
    return; // TODO: panic("putseg") instead?
```

5.7.4 smallpagedir

As a small optimization, each `Segment` embeds a fixed-size `smallpagedir` array. If the segment is small enough (at most `SMALLPAGEDIRSIZE` entries), `pagedir` points to this embedded array, avoiding an extra `smalloc()` call. Most segments are small enough to benefit from this.

⟨`Segment` *other fields* 117b⟩≡ (113b) 194c▷

```
// array<option<ref_own<Pagetable>>
Pagetable *smallpagedir[SMALLPAGEDIRSIZE];
```

⟨*constant* `SMALLPAGEDIRSIZE` *(arm)* 117c⟩≡ (1005)

```
#define SMALLPAGEDIRSIZE    16
```

⟨`newseg()` *if pagedirsize small* 117d⟩≡ (116a)

```
if(pagedirsize <= nelem(s->smallpagedir)){
    s->pagedir = s->smallpagedir;
    s->pagedirsize = nelem(s->smallpagedir);
}
```

Uses `nelem` 853i.

⟨`putseg()` *if smallpagedir* 117e⟩≡ (116c)

```
if(s->pagedir == s->smallpagedir) {
}
```

5.7.5 Adding a page to a segment: segpage()

segpage() attaches a physical Page to a segment's page table at the virtual address stored in p->va. It is called during process creation (userinit()X), page fault handling (fixfault()[185]), and segment growth (ibrk()[123b] for the bss).

⟨*function* segpage 118a⟩≡ (864b)
```
void
segpage(Segment *s, Page *p)
{
    Pagetable **pt;
    ulong off;
    Page **pg;

    ⟨segpage() sanity check page in range of segment 118b⟩

    off = p->va - s->base;
    pt = &s->pagedir[off/PAGETABMAPMEM]; // PDX
    if(*pt == nil)
        *pt = ptalloc();

    pg = &(*pt)->pagetab[(off&(PAGETABMAPMEM-1))/BY2PG]; // PTX
    *pg = p;

    if(pg < (*pt)->first)
        (*pt)->first = pg;
    if(pg > (*pt)->last)
        (*pt)->last = pg;
}
```
Uses ptalloc() 115a.

⟨segpage() *sanity check page in range of segment* 118b⟩≡ (118a)
```
if(p->va < s->base || p->va >= s->top)
    panic("segpage");
```

5.7.6 Process segments

We saw SG_TEXT, SG_DATA, etc. above—those are segment types stored in Segment.type. The Procseg enum below is different: it defines slot indices into Proc.seg[], the per-process array of segment pointers. TSEG is the slot for the text segment, DSEG for data, etc. A process always has the four core slots, plus extra slots for segattach()'d shared memory:

Procseg

⟨*enum* procseg 118c⟩≡ (868c)
```
/*
 *  process memory segments - NSEG always last !
 */
//coupling: must start with SSEG, see sysexec loop
enum Procseg
{
    SSEG, TSEG, DSEG, BSEG, // Stack, Text, Data, Bss

    ESEG, // E = Extra (used for temporary stack segment),
    _SEG0, _SEG1, _SEG2, _SEG3, _SEG4, // free slots for for segattach

    NSEG // to count, see Proc.seg array
};
```

And here is the field that ties it all together—the `seg[]` array in `Proc`, indexed by `Procseg`:

⟨`Proc` *memory fields* 119a⟩≡ (66) 181f▷
```
// hash<enum<procseg>, option<ref_own<Segment>>>, elt smalloc'ed? ref_counted?
Segment *seg[NSEG];
QLock seglock;  /* locked whenever seg[] changes */
```

`seg()`

Given a virtual address, `seg()` finds which segment it belongs to by scanning `up->seg[]`. This is used throughout the kernel whenever a user-space address must be validated or its segment located (e.g., in page fault handling, `read`/`write` syscalls, debugging):

⟨*function* `seg` 119b⟩≡ (863b)
```
Segment*
seg(Proc *p, user_addr addr, bool dolock)
{
    Segment **s, **et, *sg;

    et = &p->seg[NSEG];
    for(s = p->seg; s < et; s++) {
        sg = *s;
        if(sg == nil)
            continue;
        if(addr >= sg->base && addr < sg->top) {
            if(!dolock)
                return sg;
            // else

            qlock(&sg->lk);
            // can have a race, need to check again
            if(addr >= sg->base && addr < sg->top)
                return sg;
            qunlock(&sg->lk);
        }
    }
    return nil;
}
```
Uses `NSEG` 118c, `qlock()` 81f, and `qunlock()` 82c.

Addresses covering multiple segments, `validaddr()`

⟨*function* `okaddr` 119c⟩≡ (863b)
```
/*
 * Called only in a system call
 */
bool
okaddr(user_addr addr, ulong len, bool write)
{
    Segment *s;

    if((long)len >= 0) {
        for(;;) {
            s = seg(up, addr, false);
            if(s == nil || (write && (s->type&SG_RONLY)))
                break;

            if(addr+len > s->top) {
                len -= s->top - addr;
```

```
                addr = s->top;
            }else{
                return true;
            }
        }
    }
    pprint("suicide: invalid address %#lux/%lud in sys call pc=%#lux\n", addr, len, arch_userpc());
    return false;
}
```

Uses SG_RONLY 855, false, pprint, seg() 119b, true, and up.

⟨Segtype *other flags* 120a⟩≡ (112b) 467a▷

```
SG_RONLY  = 0040,   /* Segment is read only */
```

⟨*function* validaddr 120b⟩≡ (863b)

```
void
validaddr(user_addr addr, ulong len, bool write)
{
    if(!okaddr(addr, len, write)){
        postnote(up, 1, "sys: bad address in syscall", NDebug);
        error(Ebadarg);
    }
}
```

Uses Ebadarg 664d, NDebug 449a, okaddr() 119c, postnote, and up.

vmemchr() is a validated memchr()—it searches for character c in the user buffer starting at s for up to n bytes, checking that each page is valid as it crosses page boundaries. This is needed because a user buffer may span multiple pages, some of which might not be mapped yet:

⟨*function* vmemchr 120c⟩≡ (863b)

```
/*
 * &s[0] is known to be a valid address.
 */
void*
vmemchr(user_vp s, int c, int n)
{
    int m;
    user_addr a;
    user_vp t;

    a = (user_addr)s;
    while(PGROUND(a) != PGROUND(a+n-1)){
        /* spans pages; handle this page */
        m = BY2PG - (a & (BY2PG-1));
        t = memchr((void*)a, c, m);
        if(t)
            return t;
        a += m;
        n -= m;
        if(a < KZERO)
            validaddr(a, 1, false);
    }

    /* fits in one page */
    return memchr((void*)a, c, n);
}
```

Uses PGROUND 120d, false, and validaddr() 120b.

⟨*portdat.h macros* 120d⟩+≡ (703b) ◁47f

```
// BY2PG is defined in mem.h, which should always be included before "dat.h"!
#define PGROUND(s)  ROUNDUP(s, BY2PG)
```

dupseg()

dupseg() duplicates a segment for RFORK. Depending on the segment type, it either creates a full copy or sets up copy-on-write sharing (for data and bss segments):

⟨*function* dupseg 121a⟩≡ (864b)
```
Segment*
dupseg(Segment **seg, int segno, bool share)
{
    Segment *s;
    Segment *n; // when allocate a new segment
    ⟨dupseg() other locals 121e⟩

    SET(n);

    s = seg[segno];

    qlock(&s->lk);
    if(waserror()){
        qunlock(&s->lk);
        nexterror();
    }
    switch(s->type&SG_TYPE) {
    ⟨dupseg() switch segment type cases 121b⟩
    }
    // not sameseg, we have allocated a new seg in n above
    ⟨dupseg() when not goto sameseg, when allocated a new segment 122a⟩

    poperror();
    qunlock(&s->lk);
    return n;

sameseg:
    ⟨dupseg() in sameseg, when share a segment 121c⟩

    poperror();
    qunlock(&s->lk);
    return s;
}
```
Uses SG_TYPE, nexterror, poperror 666d, qlock() 81f, qunlock() 82c, and waserror 666d.

⟨dupseg() *switch segment type cases* 121b⟩≡ (121a) 121d▷
```
case SG_TEXT:       /* New segment shares pt set */
    goto sameseg;
```
Uses SG_TEXT.

⟨dupseg() *in sameseg, when share a segment* 121c⟩≡ (121a)
```
incref(s);
```
Uses incref() 87b.

⟨dupseg() *switch segment type cases* 121d⟩+≡ (121a) ◁121b 122b▷
```
case SG_STACK:
    n = newseg(s->type, s->base, s->size);
    break;
```
Uses SG_STACK 112b and newseg() 116a.

⟨dupseg() *other locals* 121e⟩≡ (121a)
```
int i;
int size; // pagedir
Pagetable *pt;
```

⟨dupseg() *when not goto sameseg, when allocated a new segment* 122a⟩≡ (121a)

```
size = s->pagedirsize;
for(i = 0; i < size; i++)
    if(pt = s->pagedir[i])
        n->pagedir[i] = ptcpy(pt); // will actually share the pages
```

⟨dupseg() *copy other fields* 555j⟩
⟨dupseg() *if original segment was shared, flush it* 471b⟩

Uses ptcpy() 115c.

⟨dupseg() *switch segment type cases* 122b⟩+≡ (121a) ◁121d 122c▷

```
case SG_DATA:       /* Copy on write plus demand load info */
    ⟨dupseg() when SG_DATA, if Text segment 196d⟩
    if(share)
        goto sameseg; // threads! clone()
    // else
    n = newseg(s->type, s->base, s->size);
    ⟨dupseg() SG_DATA case, attach image to new segment n 196c⟩
    break;
```

Uses SG_DATA 556b and newseg() 116a.

⟨dupseg() *switch segment type cases* 122c⟩+≡ (121a) ◁122b 556c▷

```
case SG_BSS:        /* Just copy on write */
    if(share)
        goto sameseg; // threads! clone()
    // else
    n = newseg(s->type, s->base, s->size);
    break;
```

Uses SG_BSS and newseg() 116a.

5.7.7 /proc/<pid>/segment

Reading /proc/\emphpid/segment shows the memory layout of a process—the base, top, and type of each segment. This is implemented in devprocX (see Chapter 6):

⟨procread() *locals* 122d⟩≡ (605d) 351b▷

```
Segment *sg;
```

⟨procread() *cases* 122e⟩≡ (605d) 352c▷

```
case Qsegment:
    j = 0;
    for(i = 0; i < NSEG; i++) {
        sg = p->seg[i];
        if(sg == nil)
            continue;
        j += snprint(statbuf+j, sizeof statbuf - j,
            "%-6s %c%c %.8lux %.8lux %4ld\n",
            sname[sg->type&SG_TYPE],
            sg->type&SG_RONLY ? 'R' : ' ',
            sg->profile ? 'P' : ' ',
            sg->base, sg->top, sg->ref);
    }
    if(offset >= j)
        return 0;
    if(offset+n > j)
        n = j-offset;
    if(n == 0 && offset == 0)
        exhausted("segments");
```

```
    memmove(a, &statbuf[offset], n);
    return n;
```

Uses NSEG 118c, Qsegment-873 602a, SG_RONLY 855, SG_TYPE, exhausted() 666c, and sname-909 113a.

5.8 sysbrk()

We now have all the pieces: `xalloc` for bulk allocation, `palloc` for page management, `Segment` and `Pagetable` for mapping virtual addresses to pages. We can finally see how the `brk()` syscall ties them together. This is the only memory-related syscall. When a user program calls `brk(addr)`, the kernel extends (or shrinks) the BSS segment to the requested address by allocating (or freeing) pages from `palloc` and updating the segment's page directory. The user-space `malloc()` library calls `brk()` when it needs more memory from the kernel.

⟨*syscall brk* 123a⟩≡ (881b)

```
// int brk(void*);
long
sysbrk(ulong* arg)
{
    return ibrk(arg[0], BSEG); // BSS, the heap size is changed
}
```

Uses BSEG 118c and ibrk() 123b.

⟨*function* ibrk 123b⟩≡ (864b)

```
long
ibrk(user_addr addr, int seg)
{
    Segment *s;
    user_addr newtop;
    ulong newsize;
    ⟨ibrk() other locals 124b⟩

    s = up->seg[seg];
    ⟨ibrk() sanity check s 123c⟩
    ⟨ibrk() if addr nil 123d⟩

    qlock(&s->lk);

    ⟨ibrk() if addr below base 124a⟩

    newtop = PGROUND(addr);
    newsize = (newtop - s->base)/BY2PG; // in nb pages

    ⟨ibrk() if newtop lower than oldtop, shrink the segment 125b⟩
    // else
    ⟨ibrk() if newtop more than oldtop, extend the segment 124c⟩

}
```

Uses PGROUND 120d, qlock() 81f, and up.

⟨ibrk() *sanity check s* 123c⟩≡ (123b)

```
if(s == nil)
    error(Ebadarg);
```

Uses Ebadarg 664d.

⟨ibrk() *if addr nil* 123d⟩≡ (123b)

```
if(addr == nilptr)
    return s->base;
```

Uses nilptr.

⟨ibrk() *if addr below base* 124a⟩≡ (123b)
```
/* We may start with the bss overlapping the data */
if(addr < s->base) {
    if(seg != BSEG || up->seg[DSEG] == nil || addr < up->seg[DSEG]->base) {
        qunlock(&s->lk);
        error(Enovmem);
    }
    addr = s->base;
}
```
Uses BSEG 118c, DSEG 118c, Enovmem 664d, qunlock() 82c, and up.

5.8.1 Extending a segment

The rest of ibrk() handles two cases: extending a segment (growing top upward, allocating new Pagetable entries in pagedir) and shrinking it (freeing pages and page tables):

⟨ibrk() *other locals* 124b⟩≡ (123b) 124d▷
```
int mapsize;
Pagetable **map;
```

⟨ibrk() *if newtop more than oldtop, extend the segment* 124c⟩≡ (123b)
```
⟨ibrk() sanity check newtop does not overlap other segments 124e⟩
⟨ibrk() sanity check newsize 125a⟩

// similar to code in newseg()
mapsize = ROUND(newsize, PAGETABSIZE)/PAGETABSIZE;
// realloc
if(mapsize > s->pagedirsize){
    map = smalloc(mapsize*sizeof(Pagetable*));
    memmove(map, s->pagedir, s->pagedirsize*sizeof(Pagetable*));
    if(s->pagedir != s->smallpagedir)
        free(s->pagedir);
    s->pagedir = map;
    s->pagedirsize = mapsize;
}

s->top = newtop;
s->size = newsize;
qunlock(&s->lk);
return OK_0;
```
Uses OK_0, ROUND 47b, free() 104a, qunlock() 82c, and smalloc() 104b.

⟨ibrk() *other locals* 124d⟩+≡ (123b) ◁124b
```
Segment *ns;
int i;
```

⟨ibrk() *sanity check newtop does not overlap other segments* 124e⟩≡ (124c)
```
for(i = 0; i < NSEG; i++) {
    ns = up->seg[i];
    if(ns == nil || ns == s)
        continue;
    if(newtop >= ns->base && newtop < ns->top) {
        qunlock(&s->lk);
        error(Esoverlap);
    }
}
```
Uses Esoverlap 664d, NSEG 118c, qunlock() 82c, and up.

⟨ibrk() *sanity check newsize* 125a⟩≡ (124c)
```
if(newsize > (PAGEDIRSIZE*PAGETABSIZE)) {
    qunlock(&s->lk);
    error(Enovmem);
}
```
Uses Enovmem 664d and qunlock() 82c.

5.8.2 Shrinking a segment

⟨ibrk() *if newtop lower than oldtop, shrink the segment* 125b⟩≡ (123b)
```
if(newtop < s->top) {
    /*
     * do not shrink a segment shared with other procs, as the
     * to-be-freed address space may have been passed to the kernel
     * already by another proc and is past the validaddr stage.
     */
    if(s->ref > 1){
        qunlock(&s->lk);
        error(Einuse);
    }
    mfreeseg(s, newtop, (s->top-newtop)/BY2PG);
    s->top = newtop;
    s->size = newsize;
    qunlock(&s->lk);
    arch_flushmmu();
    return OK_0;
}
```
Uses OK_0, mfreeseg() 125c, and qunlock() 82c.

⟨*function* mfreeseg 125c⟩≡ (864b)
```
/*
 *  called with s->lk locked
 */
void
mfreeseg(Segment *s, ulong start, int pages)
{
    int i, j, size;
    ulong soff;
    PageOrSwap *pg;
    Page *list;

    soff = start-s->base;
    j = (soff&(PAGETABMAPMEM-1))/BY2PG; // PTX

    size = s->pagedirsize;
    list = nil;
    for(i = soff/PAGETABMAPMEM; i < size; i++) { // PDX
        if(pages <= 0)
            break;
        if(s->pagedir[i] == nil) { // space was never accessed, good, easier
            pages -= PAGETABSIZE-j;
            j = 0;
            continue;
        }
        while(j < PAGETABSIZE) {
            pg = s->pagedir[i]->pagetab[j];
            /*
             * We want to zero s->pagedir[i]->page[j] and putpage(pg),
```

```
             * but we have to make sure other processors flush the
             * entry from their TLBs before the page is freed.
             * We construct a list of the pages to be freed, zero
             * the entries, then (below) call procflushseg, and call
             * putpage on the whole list.
             *
             * Swapped-out pages don't appear in TLBs, so it's okay
             * to putswap those pages before procflushseg.
             */
            if(pg){
                ⟨mfreeseg if pg is a swap address 206f⟩
                else{
                    pg->next = list;
                    list = pg;
                }
                s->pagedir[i]->pagetab[j] = nil;
            }
            if(--pages == 0)
                goto out;
            j++;
        }
        j = 0;
    }
out:
    ⟨mfreeseg() if segment was shared, flush it 471c⟩
    /* free the pages */
    for(pg = list; pg != nil; pg = list){
        list = list->next;
        putpage(pg);
    }
}
```

Uses putpage() 111a.

Chapter 6

Processes

In this chapter, I will present the process lifecycle: how processes are allocated, created (`rfork`), loaded with a new program (`exec`), how a parent waits for children (`await`), and how processes terminate (`exits`). These four syscalls are the foundation of Plan 9's process model.

⟨`systab` *process syscalls* 127⟩≡ (57b)

```
[RFORK]     sysrfork,
[EXEC]      sysexec,
[AWAIT]     sysawait,
[EXITS]     sysexits,
```

Uses `sysawait()` 150a, `sysexits()` 151e, and `sysrfork()` 133c.

The C prototypes (from `syscall.h`) are:

```
int   rfork(int flags);     // RFPROC|RFMEM|... see Rfork_flag
int   exec(char *name, char *argv[]);
void  _exits(char *status); // nil or "" for success
int   await(char *buf, int n);  // "pid t1 t2 t3 exitstr"
```

6.1 Overview

Plan 9 inherits the UNIX fork/exec model: `rfork` creates a child process (a copy of the parent), and `exec` replaces the child's memory with a new program. This two-step approach is powerful—between the fork and the exec, the child can set up pipes, redirect files, or modify its namespace before running the new program. Plan 9 generalizes UNIX `fork` into `rfork` ("resource fork"): instead of always copying everything, `rfork` takes flags that independently control which resources to copy, share, or create fresh—memory, file descriptors, namespace, environment, notes, and rendezvous group. A traditional `fork` is `rfork` with everything copied; a thread (shared memory) is `rfork` with `RFMEM` set. Sandboxing is trivial with `RFNOMNT`: the child gets a fresh namespace where `mount` and `bind` are forbidden, so it cannot reach any file server the parent does not explicitly pass. This achieves what Linux needs `chroot` jails or Docker containers for—but as a simple flag to `rfork`, with no extra kernel infrastructure.

6.2 Process allocation

The `Proc` structure is large and long-lived, so processes are pre-allocated in a pool during boot rather than malloc'd on each fork. The pool is an array of `conf.nproc` entries (up to 2000), allocated by `xalloc` during initialization. A free list threads through unused entries, and a hash table provides fast pid-to-process lookup.

6.2.1 Process ID: `Proc.pid`

Every process needs a unique identifier. The `pid` field holds it, and `pidalloc` (a `Counter`, i.e., a `Ref`) ensures uniqueness: each `newproc()` call atomically increments it to get a fresh value.

⟨`Proc` *state fields* 128a⟩+≡ (66) ◁67b 130d▷
```
ulong pid;
```

⟨*global* `pidalloc` 128b⟩≡ (877c)
```
static Counter  pidalloc;
```

6.2.2 `Procalloc`

The process allocator is a simple pool: an `xalloc`'ed array of `Proc` structures (`arena`) and a free list (`free`) threaded through `Proc.qnext`. When a process exits, it is returned to the free list for reuse:

⟨*struct* `Procalloc` 128c⟩≡ (868c)
```
// Process allocator
struct Procalloc
{
    // array<Proc>, xalloc'ed in procinit() (conf.nproc)
    Proc* arena;
    // list<ref<Proc>> (next = Proc.qnext)
    Proc* free;

    ⟨Procalloc other fields 128e⟩

    // extra
    Lock;
};
```

⟨*global* `procalloc` 128d⟩≡ (877c)
```
static struct Procalloc procalloc;
```
Uses `Procalloc` 128c.

6.2.3 PID hashing

The kernel needs to look up a `Proc` by pid (e.g., when accessing `/proc/x/...`). A linear scan of the arena would be slow, so `Procalloc` includes a hash table of 128 buckets chained through `Proc.pidhash`:

⟨`Procalloc` *other fields* 128e⟩≡ (128c)
```
// hash<Proc.pid, ref<Proc>> (next = Proc.pidhash)>
Proc* ht[128];
```

⟨`Proc` *extra fields* 128f⟩+≡ (66) ◁81d 221d▷
```
// hash<Proc.pid, ref<Proc>> Procalloc.ht
Proc  *pidhash; /* next proc in pid hash */
```

⟨*function* `pidhash` 128g⟩≡ (877c)
```
static void
pidhash(Proc *p)
{
    int h;

    h = p->pid % nelem(procalloc.ht);
    lock(&procalloc);
    // add_hash(procalloc.ht, p->pid, p)
    p->pidhash = procalloc.ht[h];
    procalloc.ht[h] = p;
```

```
    unlock(&procalloc);
}
```

Uses lock() 76d, nelem 853i, procalloc-2458 128d, and unlock() 78d.

⟨*function* pidunhash 129a⟩≡ (877c)

```
static void
pidunhash(Proc *p)
{
    int h;
    Proc **l;

    h = p->pid % nelem(procalloc.ht);
    lock(&procalloc);
    // remove_hash(procalloc.ht, p->pid, p)
    for(l = &procalloc.ht[h]; *l != nil; l = &(*l)->pidhash)
        if(*l == p){
            *l = p->pidhash;
            break;
        }
    unlock(&procalloc);
}
```

Uses lock() 76d, nelem 853i, procalloc-2458 128d, and unlock() 78d.

procindex() translates a pid into an index in the procalloc.arena array. It is used by devproc to implement /proc/\emphpid/...:

⟨*function* procindex 129b⟩≡ (877c)

```
int
procindex(ulong pid)
{
    Proc *p;
    int h;
    int s;

    s = -1;
    // hash_lookup(pid, procalloc.ht)
    h = pid % nelem(procalloc.ht);
    lock(&procalloc);
    for(p = procalloc.ht[h]; p != nil; p = p->pidhash)
        if(p->pid == pid){
            s = p - procalloc.arena;
            break;
        }
    unlock(&procalloc);
    return s;
}
```

Uses lock() 76d, nelem 853i, procalloc-2458 128d, and unlock() 78d.

6.2.4 newproc()

newproc() pops a Proc from the free list, assigns it a fresh pid (by incrementing the global pidalloc counter), and initializes its fields to safe defaults. The new process starts in state Scheding—not Ready—because its segments are not yet set up; the caller (sysrfork()[133c] or kproc()[157c]) must finish the initialization before calling ready()[??].

⟨Procstate *cases* 129c⟩+≡ (67c) ◁85a 153a▷

```
Scheding,
```

⟨*constructor* `newproc` 130a⟩≡ (877c)
```
Proc*
newproc(void)
{
    Proc *p;
    ⟨newproc() other locals 130g⟩

    lock(&procalloc);
    // =~ p = procalloc.free
    ⟨newproc() set p to procalloc.free and wait if nil 131a⟩
    procalloc.free = p->qnext;
    unlock(&procalloc);
    p->qnext = nil;

    p->state = Scheding;
    p->pid = incref(&pidalloc);
    ⟨newproc() sanity check pid 130b⟩
    pidhash(p);

    ⟨newproc() set fields of p 130c⟩

    return p;
}
```
Uses `Scheding` 129c, `incref()` 87b, `lock()` 76d, `pidalloc-2459` 128b, `pidhash()` 128g, `procalloc-2458` 128d, and `unlock()` 78d.

⟨`newproc()` *sanity check pid* 130b⟩≡ (130a)
```
if(p->pid==0)
    panic("pidalloc");
```

⟨`newproc()` *set fields of* `p` 130c⟩≡ (130a) 130e▷
```
memset(p->seg, nilptr, sizeof p->seg);
```
Uses `nilptr`.

Two string fields help with debugging and the `ps` command: `psstate` is a human-readable description of what the process is doing (e.g., "New", "PageOut", or the name of the syscall it is blocked in), and `text` is the name of the executable. Both appear in `/proc/\emphpid/status`.

⟨`Proc` *state fields* 130d⟩+≡ (66) ◁128a 132b▷
```
// some debugging information, e.g. "New", "PageOut", or name of syscall
char  *psstate; /* used by /proc/#/status */
// e.g. "*init*", or name of executable
char  *text;
```

⟨`newproc()` *set fields of* `p` 130e⟩+≡ (130a) ◁130c 130f▷
```
p->psstate = "New";
kstrdup(&p->text, "*notext");
```
Uses `kstrdup()` 669b.

⟨`newproc()` *set fields of* `p` 130f⟩+≡ (130a) ◁130e 132c▷
```
p->fgrp = nil;
p->pgrp = nil;
p->egrp = nil;
p->rgrp = nil;
```

6.2.5 No more processes

If the free list is empty when `newproc()`[130a] is called, the kernel sleeps and retries, hoping another process will exit soon. If the situation persists, it panics and reboots.

⟨`newproc()` *other locals* 130g⟩≡ (130a)
```
char msg[64];
```

⟨newproc() *set* p *to* procalloc.free *and wait if nil* 131a⟩≡ (130a)

```
while((p = procalloc.free) == nil) {
    unlock(&procalloc);

    snprint(msg, sizeof msg, "no procs; %s forking", up?up->text:"kernel");
    /*
     * the situation is unlikely to heal itself.
     * dump the proc table and restart by default.
     * *noprocspersist in plan9.ini will yield the old
     * behaviour of trying forever.
     */
    if(getconf("*noprocspersist") == nil)
        noprocpanic(msg);
    resrcwait(msg);
    lock(&procalloc);
}
```

Uses lock() 76d, noprocpanic() 131b, procalloc-2458 128d, resrcwait() 131c, unlock() 78d, and up.

`resrcwait()` (defined later in this chapter) prints a warning and sleeps for 300 ms before retrying—it is the kernel's standard "resource temporarily unavailable" back-off.

⟨*function* noprocpanic 131b⟩≡ (877c)

```
void
noprocpanic(char *msg)
{
    /*
     * setting exiting will make hzclock() on each processor call exit(0).
     * clearing our bit in cpus avoids calling exit(0) from hzclock()
     * on this processor.
     */
    lock(&active);
    active.cpus &= ~(1<<cpu->cpuno);
    active.exiting = true;
    unlock(&active);

    procdump();
    arch_delay(1000);
    panic(msg);
}
```

Uses active 65e, arch_delay, lock() 76d, procdump() 645a, true, and unlock() 78d.

⟨*function* resrcwait 131c⟩≡ (875)

```
void
resrcwait(char *reason)
{
    ulong now;
    char *p;
    static ulong lastwhine;

    if(up == nil)
        panic("resrcwait");

    p = up->psstate;
    if(reason) {
        up->psstate = reason;
        now = seconds();
        /* don't tie up the console with complaints */
        if(now - lastwhine > Whinesecs) {
            lastwhine = now;
            print("%s\n", reason);
```

```
        }
    }
    tsleep(&up->sleepr, returnfalse, 0, 300);
    up->psstate = p;
}
```

Uses Whinesecs-2604 132a, print, returnfalse() 104c, seconds() 263b, tsleep, and up.

⟨*constant* Whinesecs 132a⟩≡ (875)

```
Whinesecs = 10,     /* frequency of out-of-resources printing */
```

6.3 The user

In UNIX, the process owner is a numeric uid. In Plan 9, it is a string (`Proc.user`): typically "eve" for the system owner. There is no `/etc/passwd`—authentication is handled per-connection by the file server protocol (9P). In Plan 9, file permissions also use strings: `struct Dir` has `uid` and `gid` fields that are character strings, not numeric ids. The file server itself decides whether to grant access by comparing the connection's authenticated user name against these strings—there is no kernel-level uid/gid translation.

⟨Proc *state fields* 132b⟩+≡ (66) ◁130d 136a▷

```
// e.g.. "eve" (no uid/gid in plan9, because of its distributed nature?)
char  *user;
```

⟨newproc() *set fields of* p 132c⟩+≡ (130a) ◁130f 144d▷

```
kstrdup(&p->user, "*nouser");
```

Uses kstrdup() 669b.

`hostownerwrite()` is called during boot (from `boot.rc`) to set the "host owner"—the user who owns the terminal. This is the Plan 9 equivalent of setting the system's primary user account.

⟨*function* hostownerwrite 132d⟩≡ (884a)

```
/*
 *  called by devcons() for host owner/domain
 *
 *  writing hostowner also sets user
 */
long
hostownerwrite(char *a, int n)
{
    char buf[128];

    if(!iseve())
        error(Eperm);
    if(n <= 0 || n >= sizeof buf)
        error(Ebadarg);
    memmove(buf, a, n);
    buf[n] = 0;

    renameuser(eve, buf);
    kstrdup(&eve, buf);
    kstrdup(&up->user, buf);
    up->basepri = PriNormal;
    return n;
}
```

Uses Ebadarg 664d, PriNormal 219b, eve 591a, iseve() 591b, kstrdup() 669b, renameuser() 133a, and up.

⟨*function* `renameuser` 133a⟩≡ (877c)

```
/*
 *  change ownership to 'new' of all processes owned by 'old'.  Used when
 *  eve changes.
 */
void
renameuser(char *old, char *new)
{
    Proc *p, *ep;

    ep = procalloc.arena + conf.nproc;
    for(p = procalloc.arena; p < ep; p++)
        if(p->user!=nil && strcmp(old, p->user)==0)
            kstrdup(&p->user, new);
}
```

Uses `conf` 62b, `kstrdup()` 669b, and `procalloc-2458` 128d.

6.4 `sysrfork()`

`sysrfork()`[133c] is the most complex syscall in the kernel. It takes a bitmask of flags that independently control which resources to fork, share, or create fresh. If `RFPROC` is set, a new child process is created; otherwise, the flags modify the current process's own resources (e.g., detach from the parent's namespace).

⟨*enum* `rfork` 133b⟩≡ (881b)

```
//coupling: with libc.h
enum Rfork
{
    RFPROC      = (1<<4), // fork a new process!! (if unset then set props for up)
    RFMEM       = (1<<5), // share data and bss (kinda thread, a la Linux clone)

    RFNOWAIT    = (1<<6), // child will not leave a waitmsg

    RFNAMEG     = (1<<0), // copy namespace (if unset then share)
    RFENVG      = (1<<1), // copy environment variables (if unset then share)
    RFFDG       = (1<<2), // copy file descriptor table (if unset then share)

    RFCNAMEG    = (1<<10), // clean new namespace
    RFCENVG     = (1<<11), // clean new empty environment variables
    RFCFDG      = (1<<12), // clean new file descriptor table

    RFNOTEG     = (1<<3), // start new group for notes
    RFREND      = (1<<13), // start a new group for rendezvous

    RFNOMNT     = (1<<14), // # paths forbidden, sandboxing
};
```

⟨*syscall rfork* 133c⟩≡ (881b)

```
// int rfork(int flags);
long
sysrfork(ulong* arg)
{
    ulong flag;
    ⟨sysrfork() other locals 134a⟩

    flag = arg[0];
    /* Check flags before we commit */
    ⟨sysrfork() sanity check flags, clean or copy 137a⟩
```

```
    ⟨sysrfork() if no RFPROC, no new proc but property settings 139b⟩
    // else
    ⟨sysrfork() when RFPROC, fork a new process 134b⟩
}
```

6.4.1 Fork: RFPROC

When RFPROC is set, the kernel allocates a new process, copies or shares the parent's resources according to the other flags, sets up the child's kernel stack so it will return 0 from the syscall (via arch_forkchild()[135e]), and makes it runnable. The parent returns the child's pid.

```
⟨sysrfork() other locals 134a⟩≡ (133c) 135c▷
  Proc *p;
  ulong pid;
```

```
⟨sysrfork() when RFPROC, fork a new process 134b⟩≡ (133c)
  p = newproc();
  pid = p->pid;

  ⟨sysrfork() when RFPROC propagate fields from up to p 135a⟩

  /* Make a new set of memory segments */
  ⟨sysrfork() copy or share memory segments 135d⟩

  /* File descriptors */
  ⟨sysrfork() copy, clean, or share file descriptors 137b⟩
  /* Process groups */ // Namespace
  ⟨sysrfork() copy, clean, or share namespace 137d⟩
  // Rendez vous group
  ⟨sysrfork() new or share rendezvous group 138c⟩
  /* Environment group */
  ⟨sysrfork() copy, clean, or share environment 138b⟩
  // Note group
  ⟨sysrfork() share or not note group 138d⟩

  /* Craft a return frame which will cause the child to pop out of
   * the scheduler in user mode with the return register zero
   */
  arch_forkchild(p, up->dbgreg);

  p->parent = up;
  p->parentpid = up->pid;
  ⟨sysrfork() wait or not for child 156d⟩

  ⟨sysrfork() setting time field 503c⟩

  /*
   *  since the bss/data segments are now shareable,
   *  any mmu info about this process is now stale
   *  (i.e. has bad properties) and has to be discarded.
   */
  arch_flushmmu();

  ⟨sysrfork() if parent is a wired proc 238b⟩

  ready(p);
  sched(); // !!!
  return pid;
```

Uses newproc() 130a, ready, sched, and up.

The `ready(p)` call puts the child on the run queue, and `sched()` immediately reschedules—the parent may or may not continue first. The `arch_forkchild()` call (defined further below) sets up the child's kernel stack so that when the scheduler picks it, it returns from `rfork` with a return value of 0 (how the child knows it is the child).

⟨`sysrfork()` *when* `RFPROC` *propagate fields from up to p* 135a⟩≡ (134b) 135b▷
```
kstrdup(&p->text, up->text);
kstrdup(&p->user, up->user);
```
Uses `kstrdup()` 669b and `up`.

⟨`sysrfork()` *when* `RFPROC` *propagate fields from up to p* 135b⟩+≡ (134b) ◁135a 163d▷
```
p->slash = up->slash;
p->dot = up->dot;
incref(p->dot);
```
Uses `incref()` 87b and `up`.

6.4.2 Memory sharing: `RFMEM`

If `RFMEM` is set, the child shares the parent's data and bss segments (thread-like behavior). Otherwise, `dupseg()`[121a] makes copy-on-write copies. The text segment is always shared (it is read-only).

⟨`sysrfork()` *other locals* 135c⟩+≡ (133c) ◁134a 139a▷
```
bool share;
int i;
```

⟨`sysrfork()` *copy or share memory segments* 135d⟩≡ (134b)
```
share = flag & RFMEM;

qlock(&p->seglock);
if(waserror()){
    qunlock(&p->seglock);
    nexterror();
}
for(i = 0; i < NSEG; i++)
    if(up->seg[i])
        p->seg[i] = dupseg(up->seg, i, share);
qunlock(&p->seglock);
poperror();
```
Uses `NSEG` 118c, `RFMEM-2550` 133b, `dupseg()` 121a, `nexterror`, `poperror` 666d, `qlock()` 81f, `qunlock()` 82c, `up`, and `waserror` 666d.

6.4.3 `arch_forkchild()` (ARM)

`arch_forkchild()`[135e] is the key architecture-specific piece of fork: it sets up the child's kernel stack so that when the scheduler picks the child, it will "return" from the syscall with `R0`=0 (the child's return value). It copies the parent's saved user registers (`Ureg`) onto the child's kernel stack and sets `p->sched` (the `Label`) to point to `arch__forkretX`, which restores these registers and returns to user mode.

⟨*function* `arch_forkchild`*(arm)* 135e⟩≡ (1012d)
```
/*
 *  Craft a return frame which will cause the child to pop out of
 *  the scheduler in user mode with the return register zero.  Set
 *  pc to point to a l.s return function.
 */
void
arch_forkchild(Proc *p, Ureg *ureg)
{
```

```
    Ureg *cureg;

    p->sched.sp = (ulong)p->kstack + KSTACK - sizeof(Ureg);
    p->sched.pc = (ulong)arch__forkret;

    cureg = (Ureg*)(p->sched.sp);
    memmove(cureg, ureg, sizeof(Ureg));
    /* syscall returns 0 for child */ // adjust
    cureg->r0 = 0;

    /* Things from bottom of syscall which were never executed */
    p->psstate = nil;
    p->insyscall = false;

    ⟨arch_forkchild() floating point(arm) 515a⟩
}
```

Uses `false`.

⟨`Proc` *state fields* 136a⟩+≡ (66) ◁132b 144c▷

```
bool insyscall; // true when process inside a syscall
```

⟨*function* `arch__forkret` *(arm)* 136b⟩≡ (1010a)

```
/*
 *  here to jump to a newly forked process
 */
TEXT arch__forkret(SB), 1, $-4
    ADD $(4*15), R13        /* make r13 point to ureg->type */
    MOVW    8(R13), R14     /* restore link */
    MOVW    4(R13), R0      /* restore SPSR */
    MOVW    R0, SPSR        /* ... */
    MOVM.DB.S (R13), [R0-R14]   /* restore registers */
    ADD $8, R13         /* pop past ureg->{type+psr} */
    RFE             /* MOVM.IA.S.W (R13), [R15] */
```

Here is the memory layout after `arch_forkchild()` has set up the child, showing how the parent and child relate:

```
PARENT (up)                          CHILD (p)

user stack:                          user stack:
(shared or COW                       (shared or COW
 copy of parent)                      copy of parent)
+-----------+                        +-----------+
|  user     |                        |  user     |
|  code/data|                        |  code/data|
+-----------+                        +-----------+

kernel stack (up->kstack, 8 KiB):    kernel stack (p->kstack, 8 KiB):
+-----------+ <-- kstack+KSTACK      +-----------+ <-- kstack+KSTACK
|  Ureg     |  (saved by _vsvc       |  Ureg     |  (copied from parent,
|  r0..r14  |   on syscall entry)    |  r0=0!    |   r0 set to 0)
|  type,psr |                        |  type,psr |
|  pc       |  (return to user)      |  pc       |  (same user PC)
+-----------+ <-- up->dbgreg         +-----------+ <-- p->sched.sp
|           |                        |           |
|  (used by |                        |  (empty)  |
```

```
|  syscall  |                      |           |
|  handler) |                      |           |
+-----------+ <-- kstack           +-----------+ <-- kstack

up->sched.pc = (in sched code)     p->sched.pc = arch__forkret
up->sched.sp = (in sched code)     p->sched.sp = &Ureg on kstack

When scheduler picks child:
  1. gotolabel(p->sched) -> jump to arch__forkret
  2. arch__forkret restores Ureg from kstack
  3. RFE -> user mode, R0=0 (child sees rfork()=0)
```

6.4.4 File descriptors: RFFDG, RFCFDG

The next subsections handle resource sharing. They all follow the same three-way pattern: if the "G" flag is set, copy the parent's resource; if the "CG" flag is set, create a clean (empty) one; if neither, share the parent's (just increment the reference count). The two flags are mutually exclusive.

⟨sysrfork() *sanity check flags, clean or copy* 137a⟩≡ (133c) 137c▷

```
if((flag & (RFFDG|RFCFDG)) == (RFFDG|RFCFDG))
    error(Ebadarg);
```

Uses Ebadarg 664d, RFCFDG-2557 133b, and RFFDG-2554 133b.

⟨sysrfork() *copy, clean, or share file descriptors* 137b⟩≡ (134b)

```
if(flag & (RFFDG|RFCFDG)) {
    if(flag & RFFDG)
        p->fgrp = dupfgrp(up->fgrp);
    else
        p->fgrp = dupfgrp(nil);
}
else {
    p->fgrp = up->fgrp;
    incref(p->fgrp);
}
```

Uses RFCFDG-2557 133b, RFFDG-2554 133b, dupfgrp() 298c, incref() 87b, and up.

6.4.5 Namespace: RFNAMEG, RFCNAMEG

Same pattern as file descriptors, but for the process's view of the file system (the namespace). RFNOMNT is an additional flag that, when set, prevents the child from mounting anything—useful for sandboxing untrusted programs.

⟨sysrfork() *sanity check flags, clean or copy* 137c⟩+≡ (133c) ◁137a 138a▷

```
if((flag & (RFNAMEG|RFCNAMEG)) == (RFNAMEG|RFCNAMEG))
    error(Ebadarg);
```

Uses Ebadarg 664d, RFCNAMEG-2555 133b, and RFNAMEG-2552 133b.

⟨sysrfork() *copy, clean, or share namespace* 137d⟩≡ (134b)

```
if(flag & (RFNAMEG|RFCNAMEG)) {
    p->pgrp = newpgrp();
    if(flag & RFNAMEG)
        pgrpcpy(p->pgrp, up->pgrp);
    ⟨sysrfork() inherit noattach, RFPROC==1 case 363d⟩
}
else {
    p->pgrp = up->pgrp;
```

```
        incref(p->pgrp);
    }
    ⟨sysrfork() set noattach to true when RFNOMNT, RFPROC==1 case 363f⟩
```

Uses RFCNAMEG-2555 133b, RFNAMEG-2552 133b, incref() 87b, newpgrp() 347d, pgrpcpy() 368a, and up.

6.4.6 Environment variables: RFENVG, RFCENVG

```
⟨sysrfork() sanity check flags, clean or copy 138a⟩+≡ (133c) ◁137c
    if((flag & (RFENVG|RFCENVG)) == (RFENVG|RFCENVG))
        error(Ebadarg);
```

Uses Ebadarg 664d, RFCENVG-2556 133b, and RFENVG-2553 133b.

```
⟨sysrfork() copy, clean, or share environment 138b⟩≡ (134b)
    if(flag & (RFENVG|RFCENVG)) {
        p->egrp = smalloc(sizeof(Egrp)); // newegrp
        p->egrp->ref = 1;
        if(flag & RFENVG)
            envcpy(p->egrp, up->egrp);
    }
    else {
        p->egrp = up->egrp;
        incref(p->egrp);
    }
```

Uses RFCENVG-2556 133b, RFENVG-2553 133b, envcpy() 598a, incref() 87b, smalloc() 104b, and up.

6.4.7 Rendezvous group: RFREND

Rendezvous is Plan 9's lightweight IPC mechanism: two processes that share a rendezvous group can synchronize by calling `rendezvous()` with a matching tag (see Chapter 18). RFREND gives the child a fresh group, isolating it from the parent's rendezvous partners.

```
⟨sysrfork() new or share rendezvous group 138c⟩≡ (134b)
    if(flag & RFREND)
        p->rgrp = newrgrp();
    else {
        incref(up->rgrp);
        p->rgrp = up->rgrp;
    }
```

Uses RFREND-2559 133b, incref() 87b, newrgrp() 479d, and up.

6.4.8 Note group: RFNOTEG

Notes are Plan 9's equivalent of UNIX signals (see Chapter 7). Processes that share a note group can send notes to each other; RFNOTEG gives the child its own group, so notes sent to the parent's group will not reach the child.

```
⟨sysrfork() share or not note group 138d⟩≡ (134b)
    if((flag&RFNOTEG) == 0)
        p->noteid = up->noteid; // rectify what was done in newproc
```

Uses RFNOTEG-2558 133b and up.

6.4.9 Process sharing settings (no RFPROC)

When RFPROC is not set, `rfork` does not create a child. Instead, it modifies the current process's own resources—for example, a program can call `rfork(RFNAMEG)` to get its own private copy of the namespace before mounting something, without forking a new process. The code below mirrors the RFPROC branch but operates on `up` instead of a new `p`, and must carefully close the old resource groups after switching.

⟨sysrfork() *other locals* 139a⟩+≡ (133c) ◁135c 238a▷
```
Fgrp *ofg;
Pgrp *opg;
Egrp *oeg;
Rgrp *org;
```

⟨sysrfork() *if no* RFPROC*, no new proc but property settings* 139b⟩≡ (133c)
```
if((flag&RFPROC) == 0) { // not a fork, just setting properties for up
    ⟨sysrfork() when no RFPROC, sanity check flag 139c⟩

    ⟨sysrfork() when no RFPROC, clean or copy file descriptors 139d⟩

    ⟨sysrfork() when no RFPROC, clean or copy namespace 139e⟩

    ⟨sysrfork() when no RFPROC, clean or copy environment 140a⟩

    ⟨sysrfork() when no RFPROC, new rendez vous group 140b⟩

    ⟨sysrfork() when no RFPROC, new note group 140c⟩

    return 0;
}
```
Uses RFPROC-2549 133b.

⟨sysrfork() *when no* RFPROC*, sanity check flag* 139c⟩≡ (139b)
```
if(flag & (RFMEM|RFNOWAIT))
    error(Ebadarg);
```
Uses Ebadarg 664d, RFMEM-2550 133b, and RFNOWAIT-2551 133b.

⟨sysrfork() *when no* RFPROC*, clean or copy file descriptors* 139d⟩≡ (139b)
```
if(flag & (RFFDG|RFCFDG)) {
    ofg = up->fgrp;
    if(flag & RFFDG)
        up->fgrp = dupfgrp(ofg);
    else
        up->fgrp = dupfgrp(nil);
    closefgrp(ofg);
}
```
Uses RFCFDG-2557 133b, RFFDG-2554 133b, closefgrp() 299b, dupfgrp() 298c, and up.

⟨sysrfork() *when no* RFPROC*, clean or copy namespace* 139e⟩≡ (139b)
```
if(flag & (RFNAMEG|RFCNAMEG)) {
    opg = up->pgrp;
    up->pgrp = newpgrp();
    if(flag & RFNAMEG)
        pgrpcpy(up->pgrp, opg);
    ⟨sysrfork() inherit noattach, RFPROC==0 case 363c⟩
    closepgrp(opg);
}
⟨sysrfork() set noattach to true when RFNOMNT, RFPROC==0 case 363e⟩
```
Uses RFCNAMEG-2555 133b, RFNAMEG-2552 133b, closepgrp() 347e, newpgrp() 347d, pgrpcpy() 368a, and up.

⟨sysrfork() *when no* RFPROC*, clean or copy environment* 140a⟩≡ (139b)
```
if(flag & (RFENVG|RFCENVG)) {
    oeg = up->egrp;
    up->egrp = smalloc(sizeof(Egrp)); // newegrp()
    up->egrp->ref = 1;
    if(flag & RFENVG)
        envcpy(up->egrp, oeg);
    closeegrp(oeg);
}
```
Uses RFCENVG-2556 133b, RFENVG-2553 133b, closeegrp() 597d, envcpy() 598a, smalloc() 104b, and up.

⟨sysrfork() *when no* RFPROC*, new rendez vous group* 140b⟩≡ (139b)
```
if(flag & RFREND) {
    org = up->rgrp;
    up->rgrp = newrgrp();
    closergrp(org);
}
```
Uses RFREND-2559 133b, closergrp() 479e, newrgrp() 479d, and up.

⟨sysrfork() *when no* RFPROC*, new note group* 140c⟩≡ (139b)
```
if(flag & RFNOTEG)
    up->noteid = incref(&noteidalloc);
```
Uses RFNOTEG-2558 133b, incref() 87b, noteidalloc 449c, and up.

6.5 sysexec()

sysexec()X is the second half of the fork/exec pair. It replaces the current process's memory image with a new program loaded from a file. This is a complex operation because the old program's memory must be freed while building the new one—and if anything goes wrong (bad binary, out of memory), the process must survive with its old image intact. The logical chain is: file name → open channel → read a.out header → create text/data/bss segments → count arguments → build arguments on temporary stack → free old segments → relocate stack to final position → set PC and SP → return to user mode in new program. Script files starting with #! are handled by re-opening with the interpreter.

⟨*constant* AOUT_MAGIC*(arm)* 140d⟩≡ (1008a)
```
/*
 *  parameters for sysproc.c
 */
// E_MAGIC is defined in include/a.out.h
#define AOUT_MAGIC  (E_MAGIC)
```

The logical chain—filename string → channel → binary header → segments—is reflected in the local variables:

⟨sysexec() *locals* 140e⟩≡ (141c) 140f▷
```
char *file;
char *elem; // last element of binary, for up->text
char *file0;
```

⟨sysexec() *locals* 140f⟩+≡ (141c) ◁140e 140g▷
```
Chan *tc;
```

⟨sysexec() *locals* 140g⟩+≡ (141c) ◁140f 140h▷
```
Exec exec;
ulong magic;
user_addr text, entry, data, bss;
```

⟨sysexec() *locals* 140h⟩+≡ (141c) ◁140g 141a▷
```
user_addr t, d, b; // end of text/data/bss in address rounded to page
Segment *s, *ts;
```

⟨sysexec() *locals* 141a⟩+≡ (141c) ◁140h 141b▷
```
ulong ssize;
ulong nargs;
```

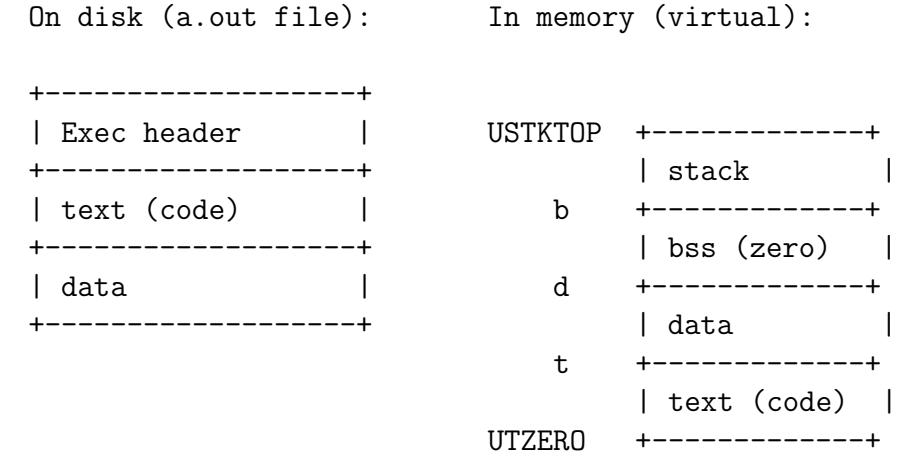
⟨sysexec() *locals* 141b⟩+≡ (141c) ◁141a 144e▷
```
int i, n;
```

⟨*syscall exec* 141c⟩≡ (881b)
```
// void* exec(char *name, char* argv[]);
long
sysexec(ulong* arg)
{
    ⟨sysexec() locals 140e⟩

    elem = nil;
    validaddr(arg[0], 1, false);
    file0 = validnamedup((char*)arg[0], true);
    if(waserror()){
        free(file0);
        free(elem);
        nexterror();
    }
    file = file0;
    for(;;){
        // this will also adjust up->genbuf to contain the last elt of file path
        ⟨sysexec() call namec() to get a channel in tc from file 304c⟩
        if(waserror()){
            cclose(tc);
            nexterror();
        }
        if(!indir)
            kstrdup(&elem, up->genbuf);

        n = devtab[tc->type]->read(tc, &exec, sizeof(Exec), 0);
        if(n < 2) // need at least 2 bytes to decide if a #! or real binary
            error(Ebadexec);
        magic = l2be(exec.magic);
        text = l2be(exec.text);
        entry = l2be(exec.entry);

        if(n==sizeof(Exec) && (magic == AOUT_MAGIC)){
            if(text >= USTKTOP-UTZERO
            || entry < UTZERO+sizeof(Exec)
            || entry >= UTZERO+sizeof(Exec)+text)
                error(Ebadexec);

            // else
            break; /* for binary */
        }
        ⟨sysexec() process sharpbang 510b⟩
    }
    // from break above
    data = l2be(exec.data);
    bss = l2be(exec.bss);

    ⟨sysexec() compute end of text, data, bss segments 143a⟩


    ⟨sysexec() build args count 145a⟩
```

```
    ⟨sysexec() build temporary stack segment and qlock seglock 145e⟩

    ⟨sysexec() build args 146b⟩

    ⟨sysexec() free old memory segments 143d⟩

    ⟨sysexec() close files marked as opened with close on exec 321f⟩


    ⟨sysexec() set Text segment 143f⟩

    ⟨sysexec() set Data segment 144a⟩

    ⟨sysexec() set Bss segment 144b⟩

    ⟨sysexec() relocate temporary stack segment, unlock seglock 147a⟩

    ⟨sysexec() set prioriry of root processes 220h⟩

    poperror();
    cclose(tc); // tc has still a reference in the img

    /*
     *  At this point, the mmu contains info about the old address
     *  space and needs to be flushed
     */
    arch_flushmmu();

    qlock(&up->debug);
    ⟨sysexec() when hold debug lock, reset some fields 450e⟩
    arch_procsetup(up);
    qunlock(&up->debug);

    ⟨sysexec() if hang 487f⟩

    return arch_execregs(entry, ssize, nargs);
}
```

`sysexec()` is one of the longest functions in the kernel. The key steps are: open the binary via `namec()`[305f] (the name-to-channel resolution function, covered in Chapter 14), read its header to get the text/data/bss sizes, tear down the old segments, create new ones, and build the stack.

The binary file (a.out format) starts with an `Exec` header followed by the text and data sections. On disk the layout is contiguous, but in memory the segments are placed at different virtual addresses:



The constants `t`, `d`, `b` below are the virtual addresses of the boundaries between segments, all page-aligned:

⟨`sysexec()` *compute end of text, data, bss segments* 143a⟩≡ (141c)
```
t = UTROUND(UTZERO+sizeof(Exec)+text);
// data is put at page boundary after text
d = ROUND(t + data, BY2PG);
// note that this is not d + bss here, but t + data + bss!
b = ROUND(t + data + bss, BY2PG);

if(t >= KZERO || d >= KZERO || b >= KZERO)
    error(Ebadexec);
```

⟨*function* `l2be` 143b⟩≡ (881b)
```
ulong
l2be(long l)
{
    byte *cp;

    cp = (byte*)&l;
    return (cp[0]<<24) | (cp[1]<<16) | (cp[2]<<8) | cp[3];
}
```

⟨*macro* `UTROUND`*(arm)* 143c⟩≡ (1005)
```
#define UTROUND(t)  ROUNDUP((t), BY2PG)
```

⟨`sysexec()` *free old memory segments* 143d⟩≡ (141c)
```
/*
 * Committed.
 * Free old memory.
 * Special segments are maintained across exec
 */
for(i = SSEG; i <= BSEG; i++) {
    putseg(up->seg[i]);
    /* prevent a second free if we have an error */
    up->seg[i] = nil;
}
```
⟨`sysexec()` *free special segments if close on exec* 467b⟩

`arch_procsetup()` initializes the architecture-specific parts of a process after exec: it sets up the MMU page table for the new address space and initializes the floating-point state.

⟨*function* `arch_procsetup`*(arm)* 143e⟩≡ (1008c)
```
/*
 *  set mach dependent process state for a new process
 */
void
arch_procsetup(Proc* p)
{
    ⟨arch_procsetup() floating point(arm) 515c⟩
}
```

6.5.1 Text segment

The text (code) segment is shared and demand-loaded: the pages are not read from disk now but will be faulted in on first access. If another process is already running the same executable, the text pages are shared via a cache image.

⟨`sysexec()` *set Text segment* 143f⟩≡ (141c)
```
/* Text.  Shared. Attaches to cache image if possible */
⟨sysexec() get text segment ts via demand loading on tc 196a⟩
up->seg[TSEG] = ts;
```

6.5.2 Data segment

⟨sysexec() *set Data segment* 144a⟩≡ (141c)
```
/* Data. Shared. */
s = newseg(SG_DATA, t, (d-t)>>PGSHIFT);
up->seg[DSEG] = s;
⟨sysexec() adjust data segment s for demand loading on tc 196b⟩
```

6.5.3 Bss segment

The bss is the simplest: it starts where data ends, and its pages are zero-filled on demand by the page fault handler—no disk I/O needed.

⟨sysexec() *set Bss segment* 144b⟩≡ (141c)
```
/* BSS. Zero fill on demand */
up->seg[BSEG] = newseg(SG_BSS, d, (b-d)>>PGSHIFT);
```

6.5.4 Stack segment

Each process has two stacks: a user stack (in the `SSEG` segment, growing down from `USTKTOP`) and a kernel stack (the per-process `kstack`, used during syscalls and interrupts). The exec code builds the new user stack with the program's arguments:

```
USTKTOP  +--------------------+
         | Tos                |  (time-of-stack info)
         +--------------------+
         | argv strings       |  "ls\0" "-l\0" ...
         +--------------------+
         | argv[n]=0          |  (nil terminator)
         | argv[1]            |  (pointer to "-l")
         | argv[0]            |  (pointer to "ls")
         +--------------------+
         | nargs              |  (argument count)
   SP --> +--------------------+
         |                    |
         | (free stack space) |
         |                    |
         +--------------------+
```

Arguments count

⟨Proc *state fields* 144c⟩+≡ (66) ◁136a 149b▷
```
char  *args;
int nargs;    /* number of bytes of args */
```

⟨newproc() *set fields of* p 144d⟩+≡ (130a) ◁132c 149a▷
```
kstrdup(&p->args, "");
p->nargs = 0;
```
Uses kstrdup() 669b.

⟨sysexec() *locals* 144e⟩+≡ (141c) ◁141b 145d▷
```
char **argp;
char *a, *args;
ulong nbytes;
```

⟨sysexec() *build args count* 145a⟩≡ (141c)
```
/*
 * Args: pass 1: count
 */
nargs = 0;
nbytes = 0;

⟨sysexec() nbytes tos adjustments 509h⟩
⟨sysexec() if indir arg adjustments 511b⟩

arch_validalign(arg[1], sizeof(char**));
argp = (char**)arg[1];
validaddr((ulong)argp, BY2WD, false);
while(*argp){
    a = *argp++;
    if(((ulong)argp&(BY2PG-1)) < BY2WD)
        validaddr((ulong)argp, BY2WD, false);
    validaddr((ulong)a, 1, false);
    nbytes += ((char*)vmemchr(a, 0, 0x7FFFFFFF) - a) + 1;
    nargs++;
}
ssize = BY2WD*(nargs+1) + ROUND(nbytes, BY2WD);
```

Temporary stack segment

Building the new stack is the trickiest part of exec. The kernel must write the argument strings and `argv` pointers into the new program's stack, but it cannot overwrite the current stack—the old program's syscall frame is still there. So the kernel creates a temporary stack segment at a different virtual address (`TSTKTOP`), builds the arguments there, and only after the old segments are freed does it relocate this segment to its final position at `USTKTOP`.

⟨*constant* `TSTKTOP` *(arm)* 145b⟩≡ (1005)
```
// end of new stack in sysexec
#define TSTKTOP     (USTKTOP-USTKSIZE)  /* sysexec temporary stack */
```

⟨*constant* `TSTKSIZ` *(arm)* 145c⟩≡ (1005)
```
#define TSTKSIZ     256 // pages in new stack; limits exec args
```

The kernel allocates a temporary stack segment at `TSTKTOP` (well below `USTKTOP`) so the old and new stacks coexist briefly:

```
USTKTOP   +--------------------+
          | old stack (SSEG)   |  <- still in use by syscall
          +--------------------+
          |        ...         |
TSTKTOP   +--------------------+
          | temp stack (ESEG)  |  <- new args built here
          +--------------------+
```

⟨sysexec() *locals* 145d⟩+≡ (141c) ◁144e 146a▷
```
ulong spage;
```

⟨sysexec() *build temporary stack segment and qlock seglock* 145e⟩≡ (141c)
```
/*
 * 8-byte align SP for those (e.g. sparc) that need it.
 * arch_execregs() will subtract another 4 bytes for argc.
 */
if((ssize+4) & 7)
```

```
    ssize += 4;
spage = (ssize+(BY2PG-1)) >> PGSHIFT;

/*
 * Build the stack segment, putting it in kernel virtual for the moment
 */
if(spage > TSTKSIZ)
    error(Enovmem);

qlock(&up->seglock);
if(waserror()){
    qunlock(&up->seglock);
    nexterror();
}
// why ESEG? and why not TSTKTOP?? ESEG because will free later
// the current SSEG. If we have an error in sysexec we don't want
// to have messed up with the current stack. TSTKTOP-USTKSIZE for
// the same reason, because we don't want overwrite old stack
// all of that will be relocated later.
up->seg[ESEG] = newseg(SG_STACK, TSTKTOP-USTKSIZE, USTKSIZE/BY2PG);
```

Arguments

The arguments are copied into the temporary stack in two passes: first the string data, then the `argv` pointer array pointing back into the strings.

⟨sysexec() *locals* 146a⟩+≡ (141c) ◁145d 195c▷

```
char **argv;
char *charp;
```

⟨sysexec() *build args* 146b⟩≡ (141c)

```
/*
 * Args: pass 2: assemble; the pages will be faulted in
 */
⟨sysexec() tos settings 509a⟩

argv = (char**)(TSTKTOP - ssize);
charp = (char*)(TSTKTOP - nbytes);
args = charp;
⟨sysexec() if indir argp adjustments 511c⟩
else
    argp = (char**)arg[1];

for(i=0; i<nargs; i++){
    ⟨sysexec() if indir argp adjustments again 512⟩
    *argv++ = charp + (USTKTOP-TSTKTOP);
    n = strlen(*argp) + 1;
    memmove(charp, *argp++, n);
    charp += n;
}
free(file0);
// file0 = nil? to avoid double free?

free(up->text);
up->text = elem;
elem = nil; /* so waserror() won't free elem */
USED(elem);

/* copy args; easiest from new process's stack */
```

```
n = charp - args;
if(n > 128) /* don't waste too much space on huge arg lists */
    n = 128;
a = up->args;
up->args = nil;
free(a);
up->args = smalloc(n);
memmove(up->args, args, n);

if(n>0 && up->args[n-1]!='\0'){
    /* make sure last arg is NUL-terminated */
    /* put NUL at UTF-8 character boundary */
    for(i=n-1; i>0; --i)
        if(fullrune(up->args+i, n-i))
            break;
    up->args[i] = 0;
    n = i+1;
}
up->nargs = n;
```

Note another use of `smalloc()` above for the argument buffer—it sleeps if memory is unavailable.

Relocate temporary stack segment

⟨`sysexec()` *relocate temporary stack segment, unlock seglock* 147a⟩≡ (141c)

```
/*
 * Move the stack
 */
s = up->seg[ESEG];
up->seg[ESEG] = nil;
up->seg[SSEG] = s;
qunlock(&up->seglock);

poperror(); /* seglock */
poperror(); /* elem */ // really? I think this matches more the cclose(tc)
s->base = USTKTOP-USTKSIZE;
s->top = USTKTOP;
relocateseg(s, USTKTOP-TSTKTOP);
```

⟨*function* `relocateseg` 147b⟩≡ (864b)

```
void
relocateseg(Segment *s, ulong offset)
{
    Page **pg, *x;
    Pagetable *pt, **p, **endpt;

    endpt = &s->pagedir[s->pagedirsize];
    for(p = s->pagedir; p < endpt; p++) {
        if(*p == nil)
            continue;
        pt = *p;
        for(pg = pt->first; pg <= pt->last; pg++) {
            if(x = *pg)
                x->va += offset;
        }
    }
}
```

6.5.5 arch_execregs() (ARM)

The final call in sysexec() is arch_execregs(), which sets the user registers (PC to the entry point, SP to the new stack) so that when the kernel returns to user mode, the new program starts. It also reserves space at the top of the stack for the Tos (Time of Stack) structure, which the kernel uses to pass timing and profiling information to user space (described in Chapter 10).

⟨*function* arch_execregs*(arm)* 148a⟩≡ (1012d)

```
long
arch_execregs(user_addr entry, ulong ssize, ulong nargs)
{
    user_wp sp;
    Ureg *ureg;

    sp = (user_wp)(USTKTOP - ssize);
    *--sp = nargs;

    ureg = up->dbgreg;
    ureg->r13 = (ulong)sp;
    ureg->pc = entry;

    /*
     * return the address of kernel/user shared data
     * (e.g. clock stuff)
     */
    return USTKTOP-sizeof(Tos);
}
```

Uses up.

6.6 sysawait()

We now look at the parent's side of process termination before seeing the child's (sysexits()). It is easier to understand the exit code once you know what the parent expects to receive. When a child exits, it leaves a wait message in the parent's queue (Proc.waitq). There is no race: even if the child exits before the parent calls await(), the message is preserved (up to 128 per parent). The parent blocks on Proc.waitr until a message is available, then returns it as a formatted string.

There are no zombie processes in Plan 9. When a child exits, it writes a wait message (containing its exit string and resource usage) to the parent's queue and immediately dies. The parent collects these messages via await(). If the parent has already exited, the messages are simply discarded. In UNIX, a zombie is created when a child exits but its parent has not yet called wait() —the child's Proc entry lingers, holding a process slot, until the parent collects the exit status. If the parent never calls wait, zombies accumulate. Plan 9 avoids this entirely: the exit status is copied into the parent's wait queue (a bounded buffer, not a Proc slot), and the child's Proc is freed immediately.

6.6.1 Parent and children

Each process keeps a pointer to its parent and a count of living children. But the parent pointer alone is not safe: if the parent exits, its Proc slot is reused for a new process. The parentpid field solves this—before accessing parent, the child checks that parent->pid == parentpid. If not, the parent has died and been replaced, so the child acts as an orphan. Orphans and zombies are different concepts: a zombie is a dead child whose exit status has not been collected; an orphan is a living child whose parent has died. Plan 9 has orphans but no zombies.

⟨Proc *hierarchy fields* 148b⟩≡ (66) 149c▷

```
  // option<ref<Proc>> (nil for the boot process)
  Proc  *parent;
  int nchild;   /* Number of living children */
```

⟨newproc() *set fields of* p 149a⟩+≡ (130a) ◁144d 149d▷
```
  p->parent = nil;
  p->nchild = 0;
```

⟨Proc *state fields* 149b⟩+≡ (66) ◁144c
```
  // option<pid> (None = 0)
  ulong parentpid;
```

6.6.2 Waitmsg and Waitq

This is the data structure behind Plan 9's "no zombies" design. Instead of keeping the dead child's Proc around (as UNIX does with zombie processes), the child writes a small Waitmsg (containing its pid, CPU time, and exit string) to the parent's waitq and then frees its Proc immediately. The parent collects these messages at its leisure via await().

⟨Proc *hierarchy fields* 149c⟩+≡ (66) ◁148b 151b▷
```
  // list<ref_own<Waitq>>> =~ list<ref_own<Waitmsg>>
  Waitq *waitq;   /* Exited processes wait children */
  int nwait;    /* Number of uncollected wait records */ // len(waitq)

  Lock  exl;    /* Lock count and waitq */ // and nchild
```

⟨newproc() *set fields of* p 149d⟩+≡ (130a) ◁149a 157b▷
```
  p->waitq = nil;
  p->nwait = 0;
```

⟨*struct* Waitmsg 149e⟩≡ (853i)
```
  struct Waitmsg
  {
    int pid;    /* of loved one */ // pid of the child
    ⟨Waitmsg time field 504d⟩
    char  msg[ERRMAX];  /* actually variable-size in user mode */
  };
```

⟨*struct* Waitq 149f⟩≡ (868c)
```
  // essentially a stack<ref_own<Waitmsg>>
  struct Waitq
  {
      Waitmsg w;

      // extra
      // list<ref_own<Waitq>> Proc.waitq
      Waitq *next;
  };
```

A Waitq entry is allocated (via smalloc) by the dying child in pexit()[??] and linked into the parent's Proc.waitq list. The 128 limit is enforced by Proc.nwait—if it reaches the maximum, the parent must drain some messages before more can be queued.

6.6.3 sysawait()

With the data structures in place, `sysawait()` itself is straightforward: it sleeps until a `Waitq` entry appears in the parent's queue, then formats it into the user-provided buffer:

⟨*syscall await* 150a⟩≡ (881b)
```
// int await(char *s, int n);
long
sysawait(ulong* arg)
{
    int i;
    int pid;
    Waitmsg w; // allocated in stack!
    ulong n;

    n = arg[1];
    validaddr(arg[0], n, true);
    pid = pwait(&w);
    ⟨sysawait() sanity check pid 150b⟩
    i = snprint((char*)arg[0], n, "%d %lud %lud %lud %q",
        w.pid,
        ⟨sysawait() snprint time field arguments 505b⟩
        w.msg);

    return i;
}
```
Uses `pwait()` 150c, `true`, and `validaddr()` 120b.

⟨`sysawait()` *sanity check pid* 150b⟩≡ (150a)
```
if(pid < 0)
    return -1;
```

⟨*function* `pwait` 150c⟩≡ (877c)
```
ulong
pwait(Waitmsg *w)
{
    ulong cpid; // child pid
    Waitq *wq;

    ⟨pwait() lock qwaitr 601a⟩

    ⟨pwait() sanity check process has children 151a⟩

    // sleep!
    ⟨pwait() wait until has a waitq 151c⟩
    // ok, got a waitmsg

    lock(&up->exl);
    // wq = pop(up->waitq), // can't be null, see haswaitq()
    wq = up->waitq;
    up->waitq = wq->next;
    up->nwait--;
    unlock(&up->exl);

    ⟨pwait() unlock qwaitr 601b⟩

    if(w)
        memmove(w, &wq->w, sizeof(Waitmsg));
    cpid = wq->w.pid;
    free(wq);
```

```
    return cpid;
}
```

Uses free() 104a, lock() 76d, unlock() 78d, and up.

⟨pwait() *sanity check process has children* 151a⟩≡ (150c)

```
lock(&up->exl);
if(up->nchild == 0 && up->waitq == nil) {
    unlock(&up->exl);
    error(Enochild);
}
unlock(&up->exl);
```

Uses Enochild 664d, lock() 76d, unlock() 78d, and up.

6.6.4 Rendez-vous: Proc.waitr

Here is the first concrete use of Rendez (introduced in the Concurrency chapter): waitr is the rendezvous point where a parent sleeps in sysawait() until a child exits and wakes it up.

⟨Proc *hierarchy fields* 151b⟩+≡ (66) ◁149c 600a▷

```
Rendez  waitr;    /* Place to hang out in wait */
```

⟨pwait() *wait until has a waitq* 151c⟩≡ (150c)

```
sleep(&up->waitr, haswaitq, up); // qwaitr is still locked
```

Uses haswaitq() 151d, sleep, and up.

⟨*function* haswaitq 151d⟩≡ (877c)

```
int
haswaitq(void *x)
{
    Proc *p;
    p = (Proc *)x;
    return p->waitq != nil;
}
```

The sleep() call (from the Concurrency chapter) takes a Rendez and a condition function. It acquires the Rendez's internal lock, checks the condition, and if false, puts the process to sleep. When another process calls wakeup() on the same Rendez, the sleeping process is woken and re-checks the condition. Here the condition is haswaitq()—"is there a Waitq entry in my queue?"

6.7 sysexits()

sysexits() is a thin wrapper that copies the exit string from user space and calls pexit()[??], the real termination function. The exit string is optional (nil means success). pexit() never returns—the process is destroyed.

⟨*syscall exits* 151e⟩≡ (881b)

```
// void exits(char *msg);
long
sysexits(ulong* arg)
{
    char *status;
    ⟨sysexits() other locals 152a⟩

    status = (char*)arg[0];
    ⟨sysexits() sanity check status, copy string in buf 152b⟩
    pexit(status, /*freemem*/true);

    panic("pexit: should never reach this point");
```

```
    return -1; // unreachable
}
```

Uses pexit and true.

⟨sysexits() *other locals* 152a⟩≡ (151e)

```
char *inval = "invalid exit string";
char buf[ERRMAX];
```

Uses ERRMAX.

⟨sysexits() *sanity check status, copy string in buf* 152b⟩≡ (151e)

```
if(status){
    if(waserror())
        status = inval;
    else{
        // validaddr() and vmemchr() can generate error()
        validaddr((ulong)status, 1, false);
        if(vmemchr(status, '\0', ERRMAX) == 0){
            memmove(buf, status, ERRMAX);
            buf[ERRMAX-1] = '\0';
            status = buf;
        }
        poperror();
    }

}
```

Uses ERRMAX, false, poperror 666d, validaddr() 120b, vmemchr() 120c, and waserror 666d.

6.7.1 pexit()

pexit() is the real termination function, and its design is split into two phases for a subtle reason. Phase 1 runs on the dying process's own kernel stack: it frees resources (file descriptors, namespace, segments), builds the wait message for the parent, and wakes the parent up. Phase 2 runs on the CPU's main stack (inside schedinit()[224f]): it releases the MMU state and returns the Proc to the free pool. The split is necessary because the process cannot free its own kernel stack while still using it—it must switch to another stack first. The process's state goes through Moribund (between the two phases) and finally Dead.

The two-phase design is necessary because phase 1 runs on the process's own kernel stack (Proc.kstack), which cannot be freed while it is in use. Phase 2 runs on the CPU's boot stack (inside the Cpu structure at CPUADDR):

```
Phase 1 (Proc.kstack):        Phase 2 (Cpu.stack):

kstack+KSTACK                 CPUADDR+CPUSIZE
    +---------------+             +---------------+
    | pexit() frame |             | schedinit()   |
    | free fds,     |             | frees kstack  |
    | segments,     |             | sets Dead     |
    | namespace...  |             | recycles Proc |
    +---------------+             +---------------+
kstack                        CPUADDR
```

⟨*function* pexit 152c⟩≡ (877c)

```
void
proc_pexit(char *exitstr, bool freemem)
{
    Proc *p; // parent
```

```
    ⟨pexit() other locals 154b⟩

    // free 1
    ⟨pexit() at the start, free resources 250d⟩
    ⟨pexit() hook proctrace 499i⟩
    // free 2
    ⟨pexit() nil out resources under lock and free them 156c⟩

    ⟨pexit() if a kernel process, do nothing about parent and child 158a⟩
    else {
        /*
         * if not a kernel process and have a parent,
         * do some housekeeping.
         */
        p = up->parent;
        ⟨pexit() sanity check parent pointer p 153b⟩
        while(waserror())
            ; // ????

        // exit/await and your parent
        ⟨pexit() build waitq message wq 154c⟩
        ⟨pexit() add wq message in parent process p waitq 154d⟩
    }
    ⟨pexit() if not freemem, add process to broken processes list 483c⟩

    // free 3, must be after built waitq because exitstr in one of the segment
    ⟨pexit() free memory segments 156a⟩

    // exit/await and your (orphan) children
    ⟨pexit() wakeup process waiting on waitr and unhash pid 155a⟩

    // free 4
    ⟨pexit() free waitq 155c⟩
    // free 5
    ⟨pexit() release debuggers 486a⟩

    /* Sched must not loop for these locks */
    lock(&procalloc);
    lock(&palloc);

    ⟨pexit() optional edfstop() for real-time scheduling 566b⟩
    up->state = Moribund;
    // will arch_gotolabel() to schedinit() which has special code around Moribund
    sched();
    panic("pexit: should never reach this point");
}
```

Uses Moribund 153a, lock() 76d, palloc 109b, procalloc-2458 128d, sched, up, and waserror 666d.

⟨Procstate *cases* 153a⟩+≡ (67c) ◁129c 231c▷

```
Moribund,
```

⟨pexit() *sanity check parent pointer* p 153b⟩≡ (152c)

```
// no parent pointer, must be the very first process
if(p == nil) {
    if(exitstr == nil)
        exitstr = "unknown";
    panic("boot process died: %s", exitstr);
}
```

6.7.2 `pexit()` and `schedinit()`

Here is phase 2: `pexit()`[??] set the state to `Moribund` and called `sched()`[??], which jumped to `schedinit()`[224f] on the CPU's main stack. Now `schedinit()` finishes the cleanup: release the MMU state, push the `Proc` back onto the free list, and unlock the locks that `pexit()` was holding.

⟨`schedinit()` *switch process state cases* 154a⟩≡ (228a) 228b▷

```
case Moribund:
    up->state = Dead;
    ⟨schedinit() optional real-time edfstop() 564h⟩
    /*
     * Holding locks from pexit:
     *  procalloc
     *  palloc
     */
    arch_mmurelease(up);

    up->qnext = procalloc.free;
    procalloc.free = up;

    unlock(&palloc);
    unlock(&procalloc);
    break;
```

Uses `Dead` 67c, `Moribund` 153a, `palloc` 109b, `procalloc-2458` 128d, `unlock()` 78d, and `up`.

6.7.3 Wait message

We saw the data structures for wait messages earlier, but the code that creates them lives inside `pexit()`—this is where the dying process builds the `Waitq` entry and links it into its parent's queue.

Wait message and your parent

⟨`pexit()` *other locals* 154b⟩≡ (152c) 155b▷

```
Waitq *wq;
```

⟨`pexit()` *build waitq message* `wq` 154c⟩≡ (152c)

```
wq = smalloc(sizeof(Waitq));
poperror(); // of ????

wq->w.pid = up->pid;
⟨pexit() set wait msg time field 504f⟩
if(exitstr && exitstr[0])
    snprint(wq->w.msg, sizeof(wq->w.msg), "%s %lud: %s", up->text, up->pid, exitstr);
else
    wq->w.msg[0] = '\0';
```

Uses `poperror` 666d, `smalloc()` 104b, and `up`.

Again `smalloc()`—the dying process may sleep here if memory is tight.

⟨`pexit()` *add* `wq` *message in parent process* `p` *waitq* 154d⟩≡ (152c)

```
lock(&p->exl);
/*
 * Check that parent is still alive.
 */
// the parent pointer may not be your parent anymore, the parent
// could have died and its slot reallocated to another process
if(p->pid == up->parentpid && p->state != Broken) {
    p->nchild--;
    ⟨pexit() update TC time of parent 505a⟩
```

```
    /*
     * If there would be more than 128 wait records
     * processes for my parent, then don't leave a wait
     * record behind.  This helps prevent badly written
     * daemon processes from accumulating lots of wait
     * records.
     */
    if(p->nwait < 128) {
        // push(wq, p->waitq)
        wq->next = p->waitq;
        p->waitq = wq;
        p->nwait++;

        wq = nil; // the parent will do the free
        wakeup(&p->waitr); // haswaitq() is true now
    }
  }
  unlock(&p->exl);
  if(wq)
      free(wq);
```

Uses Broken 482d 483a, free() 104a, lock() 76d, unlock() 78d, up, and wakeup.

Wait and my (orphan) children

When a process exits, it must also deal with its own children: wake up anyone waiting on this process's waitr, unhash its pid, and discard any uncollected wait messages from its own children.

⟨pexit() *wakeup process waiting on waitr and unhash pid* 155a⟩≡ (152c)

```
  lock(&up->exl);     /* Prevent my children from leaving waits */
  pidunhash(up);
  // so my children will not generate a waitq, I will not be here anymore
  up->pid = 0;
  wakeup(&up->waitr);
  unlock(&up->exl);
```

Uses lock() 76d, pidunhash() 129a, unlock() 78d, up, and wakeup.

⟨pexit() *other locals* 155b⟩+≡ (152c) ◁154b 155d▷

```
  Waitq *f, *next;
```

⟨pexit() *free waitq* 155c⟩≡ (152c)

```
  for(f = up->waitq; f; f = next) {
      next = f->next;
      free(f);
  }
```

Uses free() 104a and up.

6.7.4 Freeing resources

With the wait message safely queued in the parent, the dying process can now free its own resources: segments, file descriptors, namespace, environment, and rendezvous group. This must happen after building the wait message, because the exit string lives in one of the process's segments.

⟨pexit() *other locals* 155d⟩+≡ (152c) ◁155b 156b▷

```
  Segment **s, **es;
```

⟨pexit() *free memory segments* 156a⟩≡ (152c)
```
qlock(&up->seglock);
es = &up->seg[NSEG];
for(s = up->seg; s < es; s++) {
    if(*s) {
        putseg(*s);
        *s = nil;
    }
}
qunlock(&up->seglock);
```
Uses NSEG 118c, putseg() 116c, qlock() 81f, qunlock() 82c, and up.

⟨pexit() *other locals* 156b⟩+≡ (152c) ◁155d 499h▷
```
Fgrp *fgrp;
Egrp *egrp;
Rgrp *rgrp;
Pgrp *pgrp;
Chan *dot;
```

⟨pexit() *nil out resources under lock and free them* 156c⟩≡ (152c)
```
/* nil out all the resources under lock (free later) */
qlock(&up->debug);
fgrp = up->fgrp;
up->fgrp = nil;
egrp = up->egrp;
up->egrp = nil;
rgrp = up->rgrp;
up->rgrp = nil;
pgrp = up->pgrp;
up->pgrp = nil;
dot = up->dot;
up->dot = nil;
qunlock(&up->debug);

if(fgrp)
    closefgrp(fgrp);
if(egrp)
    closeegrp(egrp);
if(rgrp)
    closergrp(rgrp);
if(dot)
    cclose(dot);
if(pgrp)
    closepgrp(pgrp);
```
Uses cclose, closeegrp() 597d, closefgrp() 299b, closepgrp() 347e, closergrp() 479e, qlock() 81f, qunlock() 82c, and up.

6.7.5 No waiting for child: RFNOWAIT

Now that we have seen sysawait() and pexit(), we can understand the RFNOWAIT flag in sysrfork(). When set, parentpid is cleared to 0, so when the child exits, pexit()[??] will see that the parent's pid does not match parentpid and skip the wait message entirely. The child becomes an orphan: no one will collect its exit status. If RFNOWAIT is not set, the parent's nchild count is incremented normally. This is used by programs that spawn helper processes they never need to wait for—for example, rio (the window system) and lib9p use RFNOWAIT when forking worker processes.

⟨sysrfork() *wait or not for child* 156d⟩≡ (134b)
```
if(flag&RFNOWAIT)
    p->parentpid = 0;
```

```
else {
    lock(&up->exl);
    up->nchild++;
    unlock(&up->exl);
}
```

Uses RFNOWAIT-2551 133b, lock() 76d, unlock() 78d, and up.

6.8 Kernel processes

Not all processes run user programs. The kernel creates a few background kernel processes for housekeeping tasks: the pager (reclaims pages when memory is low), the alarm handler, etc. A kernel process has no user segments—it runs entirely in kernel memory. It is created by `kproc()`[157c], which is simpler than `sysrfork()`[133c]: it calls `newproc()`[130a], sets `kp=true`, stores the entry function in `kpfun`, and makes the process runnable. When scheduled, the process enters `linkproc()`[158c], which enables interrupts and calls the function.

⟨Proc *other fields* 157a⟩≡ (66) 163c▷

```
bool kp;   /* true if a kernel process */
void  (*kpfun)(void*);
void  *kparg;
```

⟨newproc() *set fields of* p 157b⟩+≡ (130a) ◁149d 159d▷

```
p->kp = false;
```

Uses false.

⟨*function* kproc 157c⟩≡ (877c)

```
// kernel process (aka kernel_thread in Linux?)
void
kproc(char *name, void (*func)(void *), void *arg)
{
    Proc *p;
    static Pgrp *kpgrp;

    p = newproc();
    p->kp = true; // Kernel Process!

    p->psstate = nil;
    p->procmode = 0640;
    p->ureg = nil;
    p->noswap = true;

    p->slash = up->slash;
    p->dot = up->dot;
    if(p->dot)
        incref(p->dot);

    p->fpsave = up->fpsave;
    p->sargs = up->sargs;

    memmove(p->note, up->note, sizeof(p->note));
    p->nnote = up->nnote;
    p->lastnote = up->lastnote;
    p->notify = up->notify;

    procpriority(p, PriKproc, false);

    arch_kprocchild(p, func, arg);
```

```
    kstrdup(&p->user, eve);
    kstrdup(&p->text, name);

    if(kpgrp == nil)
        kpgrp = newpgrp();
    p->pgrp = kpgrp;
    incref(kpgrp);

    ⟨kproc() setting time field 503d⟩
    ready(p);
}
```

Uses PriKproc 219b, eve 591a, false, incref() 87b, kstrdup() 669b, newpgrp() 347d, newproc() 130a, procpriority() 220a, ready, true, and up.

Unlike sysrfork(), kproc() calls ready(p) but not sched()—it does not voluntarily yield, so the new kernel process will run when the scheduler next picks it. The kp flag marks it as a kernel process.

Kernel processes have no user segments—they run entirely in kernel address space. This is why pexit() checks up->kp early and skips the parent/child wait message machinery:

⟨pexit() *if a kernel process, do nothing about parent and child* 158a⟩≡ (152c)

```
if(up->kp) {
}
```

Uses up.

6.8.1 arch_kprocchild() (ARM)

arch_kprocchild() is the kernel-process counterpart of arch_forkchild()—it sets up the child's kernel stack so that it starts executing func(arg) when scheduled. It is simpler because there is no user-mode return to arrange:

⟨*function* arch_kprocchild*(arm)* 158b⟩≡ (1008b)

```
/*
 *  setup stack and initial PC for a new kernel proc.  This is architecture
 *  dependent because of the starting stack location
 */
void
arch_kprocchild(Proc *p, void (*func)(void*), void *arg)
{
    p->sched.pc = PTR2UINT(linkproc);
    p->sched.sp = PTR2UINT(p->kstack+KSTACK);

    p->kpfun = func;
    p->kparg = arg;
}
```

Uses PTR2UINT 56g.

linkproc() is the entry point for all kernel processes—arch_kprocchild() sets the child's PC to linkproc above. It calls the actual function, then pexit() when it returns:

⟨*function* linkproc*(arm)* 158c⟩≡ (1008b)

```
/*
 *  this is the body for all kproc's
 */
static void
linkproc(void)
{
    arch_spllo();
    up->kpfun(up->kparg);
    // should never reach this place?? kernel processes are supposed
```

```
        // to run forever??
        pexit("kproc exiting", false);
    }
```

Uses false, pexit, and up.

6.9 Process control /proc/<pid>/ctl

Processes can be controlled from outside by writing commands to /proc/<pid>/ctl. For example, echo kill > /proc/42/ctl sends a "sys: killed" note to process 42. This is how the shell implements job control and how administrators kill runaway processes. The full /proc file system is covered in Chapter 15. The shell rc itself does not use /proc/\emphpid/ctl directly—it sends notes via postnote()—but tools like kill(1) and slay(1) do write to ctl.

⟨Procctl *cases* 159a⟩≡ (484d) 491a▷

```
Proc_exitme,
```

The Procctl enum values correspond to commands that can be written to /proc/\emphpid/ctl. The full /proc device is covered in Chapter 15, but procctl() is shown here because its cases use pexit() and other helpers defined in this chapter. It is fine to skim this code for now.

⟨procctl() Proc_exitme *case* 159b⟩≡ (485b)

```
case Proc_exitme:
    arch_spllo();        /* pexit has locks in it */
    pexit("Killed", true);
```

Uses Proc_exitme 159a, pexit, and true.

⟨procctlreq() *CMkill case* 159c⟩≡ (620b)

```
case CMkill:
    switch(p->state) {
    ⟨procctlreq() CMkill case, Broken case 484b⟩
    case Stopped:
        p->procctl = Proc_exitme;
        postnote(p, 0, "sys: killed", NExit);
        ready(p);
        break;
    default:
        p->procctl = Proc_exitme;
        postnote(p, 0, "sys: killed", NExit);
    }
    break;
```

Uses CMkill-882 602b, NExit 449a, Proc_exitme 159a, Stopped 484c, postnote, and ready.

⟨newproc() *set fields of* p 159d⟩+≡ (130a) ◁157b 213c▷

```
p->pdbg = nil;
p->procctl = Proc_nothing;
p->syscalltrace = nil;
p->trace = false;
p->dbgreg = nil;
```

Uses Proc_nothing 484d and false.

Chapter 7

Interrupts

This chapter shows how the kernel handles events that interrupt normal execution: system calls (which we already traced in the Overview), hardware interrupts (timer, keyboard, disk), and faults (page faults, undefined instructions). Most of the code is architecture-specific, but the overall structure is the same on any architecture: save registers, dispatch to a handler, restore registers, return.

7.1 Overview

There are two main paths through this chapter. The syscall path (a software interrupt via `SWI`) goes through `_vsvc` → `arch__syscall` → `systab[n]` → return. The exception/IRQ path goes through `_vxxx` → `_vswitch` → `arch__trap`, which dispatches based on the exception type: page faults, undefined instructions, or hardware IRQs.

```
SWI (syscall):          Other exceptions:

user code               user code (or kernel)
    |                       |
    v                       v
vectors[2]              vectors[x]
    |                       |
    v                       v
_vsvc                   _vund / _vdabt / _virq / ...
    |                       |
    | (save Ureg,           | (save R0-R4, set type)
    |  set R10/R9)          v
    v                   _vswitch
arch__syscall               | (switch to SVC mode,
    |                       |  save full Ureg,
    v                       |  set R10/R9)
systab[n]()                 v
    |                   arch__trap
    v                       |
return to user          dispatch by type:
                          - PsrMabt -> faultarm (page fault)
                          - PsrMund -> postnote (signal)
                          - PsrMirq -> irq()   (devices)
```

7.2 Exceptions (ARM)

We saw `PsrMusr` and `PsrMsvc` in the Overview. Here are the remaining processor modes, one for each exception type. Each mode has its own banked R13/R14, which is why `_vswitch` must save those registers before switching to SVC mode:

⟨`PsrMode` *other cases(arm)* 161a⟩≡ (38) 560b▷

```
#define PsrMirq     0x00000012
#define PsrMabt     0x00000017
#define PsrMund     0x0000001B
```

⟨*global* `trapnames` *(arm)* 161b⟩≡ (1002c)

```
static char *trapnames[PsrMask+1] = {
    [ PsrMusr ] "user mode",
    [ PsrMsvc ] "svc/swi exception",
    [ PsrMirq ] "irq interrupt",
    [ PsrMabt ] "prefetch abort/data abort",
    [ PsrMabt+1 ] "data abort",
    [ PsrMund ] "undefined instruction",
    ⟨trapnames() other entries(arm) 560c⟩
};
```

Uses `PsrMask` 38.

7.3 Exception `vectors` (ARM)

The ARM has 8 exception entries at fixed offsets (0x00 for reset, 0x04 for undefined, 0x08 for SWI, etc.). Each entry is a single instruction that loads the PC from a nearby table (`vtable`). The `vectors` and `vtable` code below is copied to `HVECTORS` (`0xFFFF0000`) by `arch__trapinit()`[278e] during boot. The indirection through `vtable` is necessary because the ARM vectors are only one instruction wide (4 bytes)—not enough room for a full 32-bit address. Each vector entry does `MOVW 0x18(R15), R15`, which loads the PC from 8 instructions ahead (where `vtable` starts), effectively jumping to the handler. This works because `MOVW smallint(R15), R15` encodes as a single 4-byte ARM instruction (a PC-relative load with a small offset), whereas `MOVW $_vsvc(SB), R15` would require 8 bytes or more since the full address of `_vsvc` is not known until link time (see Linker book [Pad15c] for details on ARM machine code generation):

⟨*global* `vectors` *(arm)* 161c⟩≡ (1004a)

```
/*
 *  exception vectors, copied by trapinit() to somewhere useful
 */
TEXT vectors(SB), 1, $-4
    MOVW    0x18(R15), R15      /* reset */
    MOVW    0x18(R15), R15      /* undefined instr. */
    MOVW    0x18(R15), R15      /* SWI & SMC */
    MOVW    0x18(R15), R15      /* prefetch abort */
    MOVW    0x18(R15), R15      /* data abort */
    MOVW    0x18(R15), R15      /* reserved */
    MOVW    0x18(R15), R15      /* IRQ */
    MOVW    0x18(R15), R15      /* FIQ */
```

⟨*global* `vtable` *(arm)* 161d⟩≡ (1004a)

```
//coupling: must be just after vectors in memory (at +0x18 = 8 instructions)
TEXT vtable(SB), 1, $-4
    WORD    $_vsvc(SB)      /* reset, in svc mode already */
    WORD    $_vund(SB)      /* undefined, switch to svc mode */
    WORD    $_vsvc(SB)      /* swi, in svc mode already */
    WORD    $_vpabt(SB)     /* prefetch abort, switch to svc mode */
    WORD    $_vdabt(SB)     /* data abort, switch to svc mode */
```

```
    WORD    $_vsvc(SB)      /* reserved */
    WORD    $_virq(SB)      /* IRQ, switch to svc mode */
    WORD    $_vfiq(SB)      /* FIQ, switch to svc mode */
```

7.4 System calls

We already traced the syscall path in the Overview chapter. Here we see the actual code. The `SWI` instruction puts the ARM in SVC mode and jumps to `vectors[2]`, which loads `_vsvc` from `vtable[2]`.

7.4.1 `_vsvc()` (ARM)

`_vsvc` builds a `Ureg` on the kernel stack by pushing registers in the right order, then calls `arch__syscall()` in C. On return, it restores the registers and executes `RFE` (Return From Exception) to resume user mode. Why the kernel stack? When the ARM enters SVC mode, it automatically switches to the SVC-mode stack pointer (R13_svc). This stack pointer was set up earlier during process creation: each `Proc` has a kernel stack page, and `arch__kprocchild()` (see Chapter 6) sets `R13\_svc` to the top of that page. So when a user process does `SWI`, the hardware switches to the per-process kernel stack, and `_vsvc` can safely push the `Ureg` there. The key steps are:

1. Push ureg->pc (= R14, the return address)
2. Push ureg->psr (= SPSR, the saved user CPSR)
3. Push ureg->type (= PsrMsvc, to identify this as a syscall)
4. Push R0-R14 (all user registers, via MOVM.DB.S)
5. Set R12 = kernel SB, R10 = cpu, R9 = up
6. Call arch__syscall(ureg)
7. Restore R0-R14, SPSR from the Ureg
8. RFE -> back to user mode

⟨*constant* CPU*(arm)* 162a⟩≡ (1005)
```
  #define CPU     10      /* R10 is cpu-> */
```

⟨*constant* UP*(arm)* 162b⟩≡ (1005)
```
  #define UP      9       /* R9 is up-> */
```

⟨*function* _vsvc*(arm)* 162c⟩≡ (1004a)
```
  TEXT _vsvc(SB), 1, $-4          /* SWI */
      CLREX
      MOVW.W  R14, -4(R13)        /* ureg->pc = interrupted PC */
      MOVW    SPSR, R14       /* ureg->psr = SPSR */
      MOVW.W  R14, -4(R13)        /* ... */
      MOVW    $PsrMsvc, R14       /* ureg->type = PsrMsvc */
      MOVW.W  R14, -4(R13)        /* ... */

      /* avoid the ambiguity described in notes/movm.w. */
      MOVM.DB.S [R0-R14], (R13)   /* save user level registers */
      SUB $(15*4), R13        /* r13 now points to ureg */

      MOVW    $setR12(SB), R12    /* Make sure we've got the kernel's SB loaded */

      /* get R(CPU) for this cpu */
      ⟨_vsvc() set cpu register(arm) 545d⟩

      MOVW    8(R(CPU)), R(UP)        /* up */ // Cpu->proc
```

```
MOVW    R13, R0         /* first arg is pointer to ureg */
SUB $8, R13         /* space for argument+link */

BL  arch__syscall(SB)

ADD $(8+4*15), R13      /* make r13 point to ureg->type */
MOVW    8(R13), R14     /* restore link */
MOVW    4(R13), R0      /* restore SPSR */
MOVW    R0, SPSR        /* ... */
MOVM.DB.S (R13), [R0-R14]   /* restore registers */
ADD $8, R13         /* pop past ureg->{type+psr} */
RFE             /* MOVM.IA.S.W (R13), [R15] */
```

7.4.2 Sargs

The syscall arguments live on the user stack, but the kernel copies them into `Proc.sargs` before dispatching. This avoids repeated user-space accesses (which could fault) and ensures the arguments cannot be modified by another thread while the syscall is running. `MAXSYSARG` is 5 because `mount()` has the most parameters of any syscall: `mount(fd, afd, mpt, flag, arg)`.

⟨*constant* `MAXSYSARG` 163a⟩≡ (868c)
```
#define MAXSYSARG  5 /* for mount(fd, afd, mpt, flag, arg) */
```

⟨*struct* `Sargs` 163b⟩≡ (868c)
```
// syscall arguments copied from user stack
struct Sargs
{
    ulong args[MAXSYSARG];
};
```

⟨`Proc` *other fields* 163c⟩+≡ (66) ◁157a 256b▷
```
Sargs sargs;    /* address of this is known by db */
```

⟨`sysrfork()` *when* `RFPROC` *propagate fields from up to p* 163d⟩+≡ (134b) ◁135b 213d▷
```
p->sargs = up->sargs;
```
Uses `up`.

7.4.3 `arch__syscall()` (ARM)

`arch__syscall()` is the C-level syscall dispatcher, called by `_vsvc` with a pointer to the saved `Ureg`. It extracts the syscall number from `ureg->r0`, copies the arguments from the user stack into `up->sargs`, calls `systab[scallnr]()`, stores the return value in `ureg->r0`, and handles errors. Before returning, it checks for pending notes (`notify`) and deferred scheduling.

⟨*function* `arch__syscall`*(arm)* 163e⟩≡ (1012d)
```
void
arch__syscall(Ureg* ureg)
{
    int scallnr;
    long ret;
    user_addr sp;
    ⟨syscall() other locals(arm) 165a⟩

    ⟨syscall() sanity check call from userspace(arm) 165g⟩
    ⟨syscall() adjust kentry 509d⟩

    cpu->syscall++;
    up->insyscall = true;
```

```
    up->pc = ureg->pc;
    up->dbgreg = ureg;

    // syscall number!
    scallnr = ureg->r0;
    //up->scallnr = scallnr;

    ⟨syscall() if RFORK, handle floating point(arm) 516d⟩

    arch_spllo();
    sp = ureg->sp;

    ⟨syscall() Proc_tracesyscall if, syscall entry(arm) 491c⟩

    up->nerrlab = 0;
    ret = -1;

    startns = todget(nil);
    if(!waserror()){
        ⟨syscall() sanity check scallnr(arm) 165b⟩
        ⟨syscall() sanity check sp(arm) 165c⟩

        // copy syscall arguments from user stack to up->sargs
        up->sargs = *((Sargs*)(sp+BY2WD)); // extra BY2WD for?
        up->psstate = sysctab[scallnr];

        //!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        //IMPORTANT: The actual system call
        ret = systab[scallnr](up->sargs.args);
        //!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

        poperror();
    }else{
        /* failure: save the error buffer for errstr */
        e = up->syserrstr;
        up->syserrstr = up->errstr;
        up->errstr = e;
    }
    ⟨syscall() if nerrlab(arm) 165d⟩

    /*
     *  Put return value in frame.  On the x86 the syscall is
     *  just another trap and the return value from syscall is
     *  ignored.  On other machines the return value is put into
     *  the results register by caller of syscall.
     */
    ureg->r0 = ret;

    ⟨syscall() Proc_tracesyscall if, syscall exit(arm) 491e⟩

    up->insyscall = false;
    up->psstate = nil;

    ⟨syscall() call noted() 453b⟩
    ⟨syscall() call notify(arm) 451a⟩

    ⟨syscall() if delaysched(arm) 229e⟩
    arch__kexit(ureg);
}
```

Uses arch_kexit() 169, false, poperror 666d, sysctab 885, systab 57b, todget() 262, true, up, and waserror 666d.

⟨syscall() *other locals(arm)* 165a⟩≡ (163e) 491d▷
```
char *e;
u32int s;
int i;
```

⟨syscall() *sanity check scallnr(arm)* 165b⟩≡ (163e)
```
if(scallnr >= nsyscall){
    pprint("bad sys call number %d pc %#lux\n",
        scallnr, ureg->pc);
    postnote(up, 1, "sys: bad sys call", NDebug);
    error(Ebadarg);
}
```
Uses Ebadarg 664d, NDebug 449a, nsyscall 886a, postnote, pprint, and up.

⟨syscall() *sanity check sp(arm)* 165c⟩≡ (163e)
```
if(sp < (USTKTOP-BY2PG) || sp > (USTKTOP-sizeof(Sargs)-BY2WD))
    validaddr(sp, sizeof(Sargs)+BY2WD, 0);
```
Uses validaddr() 120b.

⟨syscall() *if nerrlab(arm)* 165d⟩≡ (163e)
```
if(up->nerrlab){
    print("bad errstack [%d]: %d extra\n", scallnr, up->nerrlab);
    for(i = 0; i < NERR; i++)
        print("sp=%#p pc=%#p\n",
            up->errlab[i].sp, up->errlab[i].pc);
    panic("error stack");
}
```
Uses NERR 666a, print, and up.

⟨Proc *debugger fields* 165e⟩≡ (66) 485a▷
```
void  *dbgreg;  /* User registers for devproc */
```

⟨*function* arch_userpc*(arm)* 165f⟩≡ (1008b)
```
/*
 *  return the userpc the last exception happened at
 */
uintptr
arch_userpc(void)
{
    Ureg *ureg = up->dbgreg;
    return ureg->pc;
}
```
Uses up.

Before dispatching, arch__syscall() checks that the SWI came from user mode—a syscall from kernel mode would be a bug. arch_userureg() tests whether the saved processor mode in ureg->psr is PsrMusr:

⟨syscall() *sanity check call from userspace(arm)* 165g⟩≡ (163e)
```
if(!arch_userureg(ureg))
    panic("syscall: from kernel: pc %#lux r14 %#lux psr %#lux",
        ureg->pc, ureg->r14, ureg->psr);
```

⟨*function* arch_userureg*(arm)* 165h⟩≡ (1008b)
```
bool
arch_userureg(Ureg* ureg)
{
    return (ureg->psr & PsrMask) == PsrMusr;
}
```
Uses PsrMask 38 and PsrMusr 38.

7.5 Exception handlers

All exceptions other than SWI (which has its own fast path via `_vsvc`) go through the same two-step mechanism: a short handler `_vxxx` that saves a few registers and records the exception type, then a common `_vswitch` routine that switches to SVC mode and builds the full `Ureg`.

7.5.1 `_vxxx()` (ARM)

Each exception type has a tiny entry point that saves `R0`–`R4` to make working room, loads the exception type constant into `R0`, and falls through to `_vswitch`. Note that each exception mode has its own banked R13, so the `MOVM.IA` writes to that mode's stack—a small scratch area set up during boot, not the kernel's SVC stack:

⟨*function* `_vund` *(arm)* 166a⟩≡ (1004a)

```
TEXT _vund(SB), 1, $-4          /* undefined */
    MOVM.IA [R0-R4], (R13)      /* free some working space */
    MOVW    $PsrMund, R0
    B   _vswitch
```

⟨*function* `_vpabt` *(arm)* 166b⟩≡ (1004a)

```
TEXT _vpabt(SB), 1, $-4         /* prefetch abort */
    MOVM.IA [R0-R4], (R13)      /* free some working space */
    MOVW    $PsrMabt, R0        /* r0 = type */
    B   _vswitch
```

⟨*function* `_vdabt` *(arm)* 166c⟩≡ (1004a)

```
TEXT _vdabt(SB), 1, $-4         /* data abort */
    MOVM.IA [R0-R4], (R13)      /* free some working space */
    MOVW    $(PsrMabt+1), R0    /* r0 = type */
    B   _vswitch
```

⟨*function* `_virq` *(arm)* 166d⟩≡ (1004a)

```
TEXT _virq(SB), 1, $-4          /* IRQ */
    MOVM.IA [R0-R4], (R13)      /* free some working space */
    MOVW    $PsrMirq, R0        /* r0 = type */
    B   _vswitch
```

7.5.2 `_vswitch()` (ARM)

`_vswitch` is the heart of the exception mechanism. It must switch from the exception's processor mode (IRQ, ABT, UND) to SVC mode, because the kernel runs in SVC mode and needs access to the SVC-mode stack. But switching modes changes the banked R13 and R14, so `_vswitch` saves them first (SPSR, R14, R13) into general registers (`R1`–`R3`), then switches:

```
In exception mode (e.g., IRQ):
  R0 = exception type (set by _vxxx)
  R1 = SPSR (saved user/kernel CPSR)
  R2 = R14  (interrupted PC)
  R3 = R13  (pointer to saved R0-R4)
      |
      v  switch CPSR to SVC mode
      |
Now in SVC mode:
  R13 = SVC stack (kernel stack)
  Build Ureg on this stack, then call arch__trap()
```

It then checks whether the exception interrupted user code or kernel code (`AND.S \$0xf, R1`)—the handling differs because a kernel-mode exception must preserve the kernel's R10 (CPU) and R9 (up), and may have been nested (an interrupt inside an interrupt handler).

⟨*label* `_vswitch`*(arm)* 167a⟩≡ (1004a)

```
      /*
       *  come here with type in R0 and R13 pointing above saved [r0-r4].
       *  we'll switch to SVC mode and then call trap.
       */
  _vswitch:
      CLREX
      MOVW    SPSR, R1        /* save SPSR for ureg */
      MOVW    R14, R2         /* save interrupted pc for ureg */
      MOVW    R13, R3         /* save pointer to where the original [R0-R4] are */

      /*
       * switch processor to svc mode.  this switches the banked registers
       * (r13 [sp] and r14 [link]) to those of svc mode.
       */
      MOVW    CPSR, R14
      BIC $PsrMask, R14
      ORR $(PsrDirq|PsrMsvc), R14
      MOVW    R14, CPSR       /* switch! */

      AND.S   $0xf, R1, R4        /* interrupted code kernel or user? */
      BEQ _userexcep
      // else
      /* here for trap from SVC mode */
      ⟨_vswitch trap from SVC mode 168a⟩
      /* here for trap from USER mode */
      ⟨_vswitch trap from USER mode 167b⟩
```

⟨`_vswitch` *trap from USER mode* 167b⟩≡ (167a)

```
  _userexcep:
      MOVM.DB.W [R0-R2], (R13)    /* set ureg->{type, psr, pc}; r13 points to ureg->type  */
      MOVM.IA   (R3), [R0-R4]     /* restore [R0-R4] from previous mode's stack */

      /* avoid the ambiguity described in notes/movm.w. */
      MOVM.DB.S [R0-R14], (R13)   /* save kernel level registers */
      SUB $(15*4), R13        /* r13 now points to ureg */

      MOVW    $setR12(SB), R12    /* Make sure we've got the kernel's SB loaded */

      /* get R(CPU) for this cpu */
      ⟨_vswitch() set cpu register(arm) 546a⟩

      MOVW    8(R(CPU)), R(UP)        /* up */

      MOVW    R13, R0         /* first arg is pointer to ureg */
      SUB $(4*2), R13     /* space for argument+link (for debugger) */

      BL  arch__trap(SB)

      ADD $(4*2+4*15), R13    /* make r13 point to ureg->type */
      MOVW    8(R13), R14     /* restore link */
      MOVW    4(R13), R0      /* restore SPSR */
      MOVW    R0, SPSR        /* ... */
      MOVM.DB.S (R13), [R0-R14]   /* restore registers */
      ADD $(4*2), R13     /* pop past ureg->{type+psr} */
      RFE             /* MOVM.IA.S.W (R13), [R15] */
```

The SVC-mode (kernel) trap path is trickier. Since the kernel was already in SVC mode when the exception occurred, R13 and R14 were not banked—the saved R0–R4 are on the exception mode's stack, not the SVC stack. The code restores them, then saves all registers onto the SVC stack. On return, it must also overwrite the saved `R10` with the current `R(CPU)`, because if a reschedule happened during the trap, this processor might now be running a different process whose R10 was stale.

⟨_vswitch *trap from SVC mode* 168a⟩≡ (167a)

```
MOVM.DB.W [R0-R2], (R13)    /* set ureg->{type, psr, pc}; r13 points to ureg->type  */
MOVM.IA   (R3), [R0-R4]     /* restore [R0-R4] from previous mode's stack */

/*
 * avoid the ambiguity described in notes/movm.w.
 * In order to get a predictable value in R13 after the stores,
 * separate the store-multiple from the stack-pointer adjustment.
 * We'll assume that the old value of R13 should be stored on the stack.
 */
/* save kernel level registers, at end r13 points to ureg */
MOVM.DB [R0-R14], (R13)
SUB $(15*4), R13        /* SP now points to saved R0 */

MOVW    $setR12(SB), R12    /* Make sure we've got the kernel's SB loaded */

MOVW    R13, R0         /* first arg is pointer to ureg */
SUB $(4*2), R13     /* space for argument+link (for debugger) */
MOVW    $0xdeaddead, R11    /* marker */

BL  arch__trap(SB)

MOVW    $setR12(SB), R12    /* reload kernel's SB (ORLY?) */
ADD $(4*2+4*15), R13    /* make r13 point to ureg->type */
/*
 * if we interrupted a previous trap's handler and are now
 * returning to it, we need to propagate the current R(CPU) (R10)
 * by overriding the saved one on the stack, since we may have
 * been rescheduled and be on a different processor now than
 * at entry.
 */
MOVW    R(CPU), (-(15-CPU)*4)(R13) /* restore current cpu's CPU */
MOVW    8(R13), R14     /* restore link */
MOVW    4(R13), R0      /* restore SPSR */
MOVW    R0, SPSR        /* ... */

MOVM.DB (R13), [R0-R14]     /* restore registers */

ADD $(4*2), R13     /* pop past ureg->{type+psr} to pc */
RFE             /* MOVM.IA.S.W (R13), [R15] */
```

7.5.3 `arch__trap()` (ARM)

`arch__trap()` is the C-level exception dispatcher, analogous to `arch__syscall()`[163e] for system calls. It receives the saved `Ureg` and dispatches based on `ureg->type`: data abort and prefetch abort go to the page fault handler, undefined instruction to a trap handler, and IRQ to the interrupt dispatcher. Before returning to user mode, it checks for pending notes (signals) and deferred scheduling, just like `arch__syscall()`. One subtle detail: the ARM pipeline means the saved PC points past the faulting instruction. For most exceptions, the interrupted PC is at `ureg->pc - 4`; for data aborts it is `ureg->pc - 8` (because the ARM has already advanced past the faulting instruction and the one after it). The code below corrects for this.

⟨*function* `arch__trap` *(arm)* 168b⟩≡ (1002c)

```
/*
 *  here on all exceptions other than syscall (SWI) and fiq
 */
void
arch__trap(Ureg *ureg)
{
    bool user;
    bool clockintr;
    ⟨trap() other locals(arm) 170b⟩

    ⟨trap() sanity check interrupts are disabled(arm) 170a⟩
    ⟨trap() sanity check enough space in kernel stack(arm) 170c⟩

    user = (ureg->psr & PsrMask) == PsrMusr; // TODO: arch_userureg
    if(user){
        up->dbgreg = ureg;
        ⟨trap() adjust kentry when interrupt user 509e⟩
    }

    /*
     * All interrupts/exceptions should be resumed at ureg->pc-4,
     * except for Data Abort which resumes at ureg->pc-8.
     */
    if(ureg->type == (PsrMabt+1))
        ureg->pc -= 8;
    else
       ureg->pc -= 4;

    clockintr = false;      /* if set, may call sched() before return */

    switch(ureg->type){
    ⟨trap() switch exception type cases(arm) 170e⟩
    default:
        panic("unknown trap; type %#lux, psr mode %#lux pc %lux", ureg->type,
            ureg->psr & PsrMask, ureg->pc);
        break;
    }
    arch_splhi();

    ⟨trap() if delaysched and clockintr 229d⟩

    if(user){
        ⟨trap() call possibly notify(arm) 450g⟩
        arch__kexit(ureg);
    }
}
```

Uses PsrMabt 161a, PsrMask 38, PsrMusr 38, arch__kexit() 169, false, and up.

arch__kexit() is called just before returning from the kernel to user space (after both syscalls and exceptions). It updates the Tos structure with the current CPU frequency and flushes the write-back cache so that user code sees consistent memory.

⟨*function* arch__kexit*(arm)* 169⟩≡ (1008b)

```
/* go to user space */
void
arch__kexit(Ureg*)
{
    ⟨arch__kexit() tos adjustments 509b⟩
    tos->cyclefreq = cpu->cpuhz;
    ⟨arch__kexit() write back cache(arm) 531m⟩
```

```
}
```

⟨trap() *sanity check interrupts are disabled(arm)* 170a⟩≡ (168b)
```
assert(!arch_islo());
```
Uses assert 853f.

⟨trap() *other locals(arm)* 170b⟩≡ (168b) 170d▷
```
int rem;
```

⟨trap() *sanity check enough space in kernel stack(arm)* 170c⟩≡ (168b)
```
if(up != nil)
    rem = ((char*)ureg)-up->kstack;
else
    rem = ((char*)ureg)-((char*)cpu + sizeof(Cpu));
if(rem < 256) {
    iprint("trap: %d stack bytes left, up %#p ureg %#p at pc %#lux\n",
        rem, up, ureg, ureg->pc);
    arch_delay(1000);
    arch_dumpstack();
    panic("trap: %d stack bytes left, up %#p ureg %#p at pc %#lux",
        rem, up, ureg, ureg->pc);
}
```
Uses arch_delay, arch_dumpstack, iprint, and up.

7.5.4 Undefined instruction fault (ARM)

An undefined instruction in user mode sends a note to the process (which can handle it or die). In kernel mode, it is a bug and causes a panic. The handler also checks for floating-point instructions, which may trap if the FPU was not enabled for this process.

⟨trap() *other locals(arm)* 170d⟩+≡ (168b) ◁170b 171b▷
```
char buf[ERRMAX];
```
Uses ERRMAX.

⟨trap() *switch exception type cases(arm)* 170e⟩≡ (168b) 171c▷
```
case PsrMund:           /* undefined instruction */
    if(user){
        ⟨trap() when undefined instruction, if breakpoint(arm) 482a⟩
        else{
            ⟨trap() when undefined instruction, if float instr(arm) 527a⟩
            if(rv == 0){
                snprint(buf, sizeof buf,
                    "undefined instruction: pc %#lux\n",
                    ureg->pc);
                postnote(up, 1, buf, NDebug);
            }
        }
    }else{
        ⟨trap() when undefined instruction, round pc(arm) 171a⟩
        iprint("undefined instruction: pc %#lux inst %#ux\n",
            ureg->pc, *(u32int*)ureg->pc);
        panic("undefined instruction");
    }
    break;
```
Uses NDebug 449a, PsrMund 161a, iprint, postnote, and up.

⟨trap() *when undefined instruction, round pc(arm)* 171a⟩≡ (170e)
```
if (ureg->pc & 3) {
    iprint("rounding fault pc %#lux down to word\n",
        ureg->pc);
    ureg->pc &= ~3;
}
```
Uses iprint.

7.5.5 Prefetch fault (ARM)

A prefetch abort means the CPU tried to fetch an instruction from an unmapped or protected address. If it is a valid user address, this triggers a page fault (handled by `faultarm()`[182b], which calls the virtual memory system). Otherwise, it is a fatal error. The Instruction Fault Status Register (`IFSR`) encodes the reason; the code extracts a 4-bit fault code by concatenating non-contiguous bits (an ARMv6/v7 encoding quirk).

⟨trap() *other locals(arm)* 171b⟩+≡ (168b) ◁170d 171f▷
```
int x;
ulong fsr;
```

⟨trap() *switch exception type cases(arm)* 171c⟩+≡ (168b) ◁170e 172a▷
```
case PsrMabt:           /* prefetch fault */
    x = ifsrget();
    fsr = (x>>7) & 0x8 | x & 0x7;
    switch(fsr){
    ⟨trap() when prefetch fault, switch fst cases(arm) 482b⟩
    default:
        // Page fault on code!!
        faultarm(ureg, ureg->pc, user, true);
        break;
    }
    break;
```
Uses PsrMabt 161a, faultarm() 182b, and true.

⟨*function* ifsrget*(arm)* 171d⟩≡ (1003)
```
TEXT ifsrget(SB), 1, $-4            /* instruction fault status */
    MRC CpSC, 0, R0, C(CpFSR), C(0), CpFSRinst
    RET
```

⟨*type* CpFSR*(arm)* 171e⟩≡ (888c)
```
/*
 * CpFSR opcode2 values.
 */
#define CpFSRdata   0           /* armv6, armv7 */
#define CpFSRinst   1           /* armv6, armv7 */
```

7.5.6 Data fault (ARM)

A data abort is the most common exception: it fires when a load or store accesses an unmapped page. This is the entry point for demand paging—the page fault handler (`faultarm()`[182b]) checks whether the address belongs to a valid segment, allocates a page if needed, fills it from disk or zeros it, and returns. The faulting instruction is then re-executed. Data aborts can also occur when the kernel accesses a user address during a syscall (e.g., copying arguments); this case is handled via the error/waserror mechanism. The Fault Status Register (`FSR`) tells the kernel why the fault occurred. The code extracts a 6-bit fault code from the FSR and dispatches on it: translation faults (0x5, 0x7) and permission faults (0xd, 0xf) go to the page fault handler; domain faults (0x9, 0xb) are access violations; alignment faults and external aborts are fatal errors.

⟨trap() *other locals(arm)* 171f⟩+≡ (168b) ◁171b 526d▷
```
virt_addr va; // can be a kern_addr?
ulong inst;
```

⟨trap() *switch exception type cases(arm)* 172a⟩+≡ (168b) ◁171c 177a▷
```
case PsrMabt+1:         /* data fault */
    va = farget();
    inst = *(ulong*)(ureg->pc);
    /* bits 12 and 10 have to be concatenated with status */
    x = fsrget();
    fsr = (x>>7) & 0x20 | (x>>6) & 0x10 | x & 0xf;
    switch(fsr){
    ⟨trap() when data fault, switch fst cases(arm) 172d⟩
    default:
    case 0xa:       /* ? was under external abort */
        panic("unknown data fault, 6b fsr %#lux", fsr);
        break;
    }
    break;
```
Uses PsrMabt 161a.

⟨*function* farget*(arm)* 172b⟩≡ (1003)
```
TEXT farget(SB), 1, $-4             /* fault address */
    MRC CpSC, 0, R0, C(CpFAR), C(0x0)
    RET
```

⟨*function* fstget*(arm)* 172c⟩≡ (1003)
```
TEXT fsrget(SB), 1, $-4             /* data fault status */
    MRC CpSC, 0, R0, C(CpFSR), C(0), CpFSRdata
    RET
```

⟨trap() *when data fault, switch fst cases(arm)* 172d⟩≡ (172a) 173a▷
```
case 0xd:
case 0xf:
    /* permission error, copy on write or real permission error */
    faultarm(ureg, va, user, !writetomem(inst));
    break;
```
Uses faultarm() 182b and writetomem() 172e.

The page fault handler needs to know whether the faulting instruction was a load or a store, to set PG_MOD correctly. writetomem() decodes the faulting ARM instruction: bit 20 distinguishes loads (1) from stores (0) for most instructions, and swap instructions (detected by a specific bit pattern) always write.

⟨*function* writetomem*(arm)* 172e⟩≡ (1002c)
```
/*
 *  returns true if the instruction writes memory, false otherwise
 */
bool
writetomem(ulong inst)
{
    /* swap always write memory */
    if((inst & 0x0FC00000) == 0x01000000)
        return true;

    /* loads and stores are distinguished by bit 20 */
    if(inst & (1<<20))
        return false;

    return true;
}
```
Uses false and true.

⟨trap() *when data fault, switch fst cases(arm)* 173a⟩+≡ (172a) ◁172d 173b▷
```
  case 0x5:       /* translation fault, no section entry */
  case 0x7:       /* translation fault, no page entry */
      // Page fault on data!!
      faultarm(ureg, va, user, !writetomem(inst));
      break;
```
Uses faultarm() 182b and writetomem() 172e.

⟨trap() *when data fault, switch fst cases(arm)* 173b⟩+≡ (172a) ◁173a 173c▷
```
  case 0x9:
  case 0xb:
      /* domain fault, accessing something we shouldn't */
      if(user){
          snprint(buf, sizeof buf,
              "sys: access violation: pc %#lux va %#p\n",
              ureg->pc, va);
          postnote(up, 1, buf, NDebug);
      } else
          panic("kernel access violation: pc %#lux va %#p",
              ureg->pc, va);
      break;
```
Uses NDebug 449a, postnote, and up.

⟨trap() *when data fault, switch fst cases(arm)* 173c⟩+≡ (172a) ◁173b 173d▷
```
  case 0x0:
      panic("vector exception at %#lux", ureg->pc);
      break;
```

⟨trap() *when data fault, switch fst cases(arm)* 173d⟩+≡ (172a) ◁173c 173e▷
```
  case 0x1:       /* alignment fault */
  case 0x3:       /* access flag fault (section) */
      if(user){
          snprint(buf, sizeof buf,
              "sys: alignment: pc %#lux va %#p\n",
              ureg->pc, va);
          postnote(up, 1, buf, NDebug);
      } else
          panic("kernel alignment: pc %#lux va %#p", ureg->pc, va);
      break;
```
Uses NDebug 449a, postnote, and up.

⟨trap() *when data fault, switch fst cases(arm)* 173e⟩+≡ (172a) ◁173d 173f▷
```
  case 0x2:
      panic("terminal exception at %#lux", ureg->pc);
      break;
```

⟨trap() *when data fault, switch fst cases(arm)* 173f⟩+≡ (172a) ◁173e 174a▷
```
  case 0x4:       /* icache maint fault */
  case 0x6:       /* access flag fault (page) */
  case 0x8:       /* precise external abort, non-xlat'n */
  case 0x28:
  case 0xc:       /* l1 translation, precise ext. abort */
  case 0x2c:
  case 0xe:       /* l2 translation, precise ext. abort */
  case 0x2e:
  case 0x16:      /* imprecise ext. abort, non-xlt'n */
  case 0x36:
      panic("external abort %#lux pc %#lux addr %#p",
          fsr, ureg->pc, va);
      break;
```

⟨trap() *when data fault, switch fst cases(arm)* 174a⟩+≡ (172a) ◁173f
```
  case 0x1c:      /* l1 translation, precise parity err */
  case 0x1e:      /* l2 translation, precise parity err */
  case 0x18:      /* imprecise parity or ecc err */
      panic("translation parity error %#lux pc %#lux addr %#p",
          fsr, ureg->pc, va);
      break;
```

7.6 IRQs (ARM)

Hardware interrupts (IRQs) come from external devices: the timer, USB controller, SD card, UART, etc. On the Raspberry Pi, the BCM2835 has an interrupt controller at `INTREGS` with three 32-bit pending registers (covering up to 72 interrupt sources). Each device driver registers its handler via `arch_intrenable()`[175e], and when an IRQ fires, `irq()`[177b] scans the pending bits and dispatches to the registered handler. The overall pattern is simple: at boot, each device driver calls `arch_intrenable(IRQxxx, handler, arg)` to register itself. At runtime, when a device raises an interrupt, the hardware sets a bit in the pending register, the ARM jumps to `_virq`, which builds a `Ureg` and calls `arch__trap`, which calls `irq()`, which walks the `vctl` list and calls every handler whose pending bit is set.

⟨*enum* `IRQ` *(arm)* 174b⟩≡ (890b)
```
  enum IRQ {
      IRQtimer0   = 0, // reserved by GPU?
      IRQtimer1   = 1, // reserved by GPU?
      IRQtimer2   = 2, // reserved by GPU?
      IRQtimer3   = 3,

      IRQclock    = IRQtimer3,

      IRQusb      = 9,  // also IRQfiq
      IRQdma0     = 16, // IRQdma1, IRQdma2, ... via IRQDMA() macro

      IRQaux      = 29,
      IRQi2c      = 53,
      IRQspi      = 54,
      IRQmmc      = 62,

      ⟨IRQ basic cases(arm) 176c⟩
      ⟨IRQ local cases(arm) 552⟩
      ⟨IRQ other cases(arm) 561b⟩
  };
```
Uses `IRQtimer3` 174b.

7.6.1 `Vctl` (ARM)

A `Vctl` ("vector control") records one registered interrupt handler: which IRQ number it handles, the hardware register and bitmask to check whether this IRQ is pending, the handler function, and its argument. All registered handlers form a linked list (`vctl`):

⟨*struct* `Vctl` *(arm)* 174c⟩≡ (1002c)
```
  struct Vctl {
      // enum<IRQ>
      int irq;

      u32int  *reg; // VIRTIO address
      u32int  mask;
```

```
    void    (*f)(Ureg*, void*); /* handler to call */
    void    *a; /* argument to call it with */

    ⟨Vctl other fields(arm) 553a⟩

    // Extra
    ⟨Vctl extra fields(arm) 175b⟩
};
```

⟨*global* `vctl`*(arm)* 175a⟩≡ (1002c)
```
// list<ref_own<Vctl>>> (next = Vctl.next)
static Vctl *vctl;
```

⟨`Vctl` *extra fields(arm)* 175b⟩≡ (174c)
```
Vctl    *next;
```

⟨*global* `vctllock`*(arm)* 175c⟩≡ (1002c)
```
static Lock vctllock;
```

7.6.2 `Intregs` (ARM)

The BCM2835 interrupt controller is accessed via memory-mapped registers at `INTREGS`. It has separate pending/enable/disable registers for ARM-specific and GPU-shared interrupts. The structure fields are laid out to match the hardware registers exactly—writing to `GPUenable[0]` directly enables the corresponding interrupt in hardware:

⟨*struct* `Intregs`*(arm)* 175d⟩≡ (1002c)
```
/*
 * interrupt control registers
 */
// The order matters! the fields match the memory-mapped external registers.
struct Intregs {
    u32int  ARMpending;
    u32int  GPUpending[2];

    u32int  FIQctl;

    u32int  GPUenable[2];
    u32int  ARMenable;

    u32int  GPUdisable[2];
    u32int  ARMdisable;
};
```

7.6.3 `arch_intrenable()` (ARM)

Device drivers call `arch_intrenable()`[175e] during initialization to register their interrupt handler. It allocates a `Vctl`, records the handler and the appropriate pending register/mask, and enables the interrupt in the hardware controller:

⟨*macro* `arch_intrenable`*(arm)* 175e⟩≡ (974)
```
#define arch_intrenable(i, f, a, b, n) irqenable((i), (f), (a))
```

⟨*function* `irqenable` *(arm)* 176a⟩≡ (1002c)

```
void
irqenable(int irq, void (*f)(Ureg*, void*), void* a)
{
    Vctl *v;
    Intregs *ip;
    u32int *enable;

    ip = (Intregs*)INTREGS;
    v = (Vctl*)malloc(sizeof(Vctl));
    ⟨irqenable() sanity check v(arm) 176b⟩
    v->irq = irq;
    v->cpu = 0;
    ⟨irqenable() if IRQlocal(arm) 553b⟩
    ⟨irqenable() if IRQbasic(arm) 176d⟩
    else{
        enable = &ip->GPUenable[irq/32];
        v->reg = &ip->GPUpending[irq/32];
        v->mask = 1 << (irq % 32);
    }
    v->f = f;
    v->a = a;
    lock(&vctllock);
    ⟨irqenable() if IRQfiq(arm) 561c⟩
    else{
        v->next = vctl;
        vctl = v;
        ⟨irqenable() if IRQlocal adjust enable(arm) 553c⟩
        else
            *enable = v->mask;
    }
    unlock(&vctllock);
}
```

Uses `INTREGS-2403` 43d, `lock()` 76d, `malloc()` 103c, `unlock()` 78d, `vctl-2412` 175a, and `vctllock-2411` 175c.

⟨`irqenable()` *sanity check v(arm)* 176b⟩≡ (176a)

```
if(v == nil)
    panic("irqenable: no mem");
```

7.6.4 Basic IRQs (ARM)

The BCM2835 splits interrupts into two groups: 64 GPU interrupts (in `GPUpending[0..1]`) and a handful of ARM-specific interrupts (in `ARMpending`, starting at number 64). The ARM timer interrupt (`IRQtimerArm`) is in this second group:

⟨`IRQ` *basic cases(arm)* 176c⟩≡ (174b)

```
IRQbasic    = 64,
IRQtimerArm = IRQbasic + 0,
```

Uses `IRQbasic` 176c.

⟨`irqenable()` *if IRQbasic(arm)* 176d⟩≡ (176a)

```
else if(irq >= IRQbasic){
    enable = &ip->ARMenable;
    v->reg = &ip->ARMpending;
    v->mask = 1 << (irq - IRQbasic);
}
```

Uses `IRQbasic` 176c.

7.6.5 irq() (ARM)

`irq()`[177b] is called from `arch__trap` when the exception type is `PsrMirq`. It walks the `vctl` list and, for each handler whose pending bit is set, calls the handler. It returns `true` if a clock interrupt was among them—the caller uses this to decide whether to trigger a reschedule (the clock is what drives preemptive scheduling).

⟨`trap()` *switch exception type cases(arm)* 177a⟩+≡ (168b) ◁172a
```
case PsrMirq:
    clockintr = irq(ureg);
    cpu->intr++;
    break;
```
Uses `PsrMirq` 161a and `irq()` 177b.

The dispatch loop is brute-force: walk the entire `vctl` list and check each handler's pending bit. This is acceptable because the list is short (a handful of device drivers). If no handler matches, the interrupt is counted as spurious. The return value tells `arch__trap()` whether the clock was among the interrupts, which triggers a possible reschedule.

⟨*function* `irq`*(arm)* 177b⟩≡ (1002c)
```
/*
 *  called by trap to handle irq interrupts.
 *  returns true iff a clock interrupt, thus maybe reschedule.
 */
static bool
irq(Ureg* ureg)
{
    Vctl *v;
    bool clockintr;
    bool found;

    cpu->perf.intrts = arch_perfticks();
    clockintr = false;
    found = false;
    for(v = vctl; v; v = v->next)
        if(v->cpu == cpu->cpuno && (*v->reg & v->mask) != 0){
            found = true;
            ⟨irq() before dispatch(arm) 534c⟩
            // Dispatch
            v->f(ureg, v->a);
            ⟨irq() after dispatch(arm) 534d⟩
            if(v->irq == IRQclock || v->irq == IRQcntps || v->irq == IRQcntpns)
                clockintr = true;
        }
    if(!found)
        cpu->spuriousintr++;
    intrtime();
    return clockintr;
}
```
Uses `IRQclock` 174b, `IRQcntpns` 552, `IRQcntps` 552, `false`, `true`, and `vctl-2412` 175a.

Chapter 8

Virtual Memory

8.1 Overview

This chapter connects the interrupt machinery of the previous chapter to the memory structures of the Memory chapter. When a process accesses an unmapped page, the ARM MMU generates a data abort (or prefetch abort), which reaches `arch__trap()` and dispatches to `faultarm()`[182b]. The page fault handler then walks the software segment tables to decide what to do: allocate a zero page (bss), load from disk (demand loading), copy a shared page (copy on write), or page in from swap. The chapter is organized in layers: first the ARM MMU constants and hardware page table manipulation (`arch_putmmu`), then the portable page fault handler (`fault`/`fixfault`), then the higher-level mechanisms built on top: copy on write, the page cache, demand loading from executables, and swapping.

8.2 The ARM MMU (ARM)

We saw the ARM's two-level page table structure in the Overview chapter. Here are the macros that extract the L1 index (bits 31–20, selecting one of 4096 entries in the page directory) and the L2 index (bits 19–12, selecting one of 256 entries in a page table):

⟨*macro* `L1X`*(arm)* 178a⟩≡ (1006b)
```
#define L1X(va)     FEXT((va), 20, 12)
```

⟨*macro* `L2X`*(arm)* 178b⟩≡ (1006b)
```
#define L2X(va)     FEXT((va), 12, 8)
```

⟨*macro L1X bis(arm)* 178c⟩≡ (890a)
```
#define L1X(va)     (((((va))>>20) & 0x0fff)<<2)
```

The L1 entry type determines the granularity of the mapping. `Section` maps 1 MiB directly (used for kernel RAM). `Coarse` points to an L2 table of 256 entries, each mapping a 4 KiB page (used for user process pages):

⟨*type* `PageDirGranularity`*(arm)* 178d⟩≡ (888c)
```
#define Coarse      (Mbz|1)         /* L1 */
#define Section     (Mbz|2)         /* L1 1MB */
```

⟨*constant* `Mbz`*(arm)* 178e⟩≡ (888c)
```
/* Mbz (0x10) bit is implementation-defined and must be 0 on the cortex. */
#define Mbz     (0<<4)
```

⟨*type* `PageTableGranularity`*(arm)* 178f⟩≡ (888c)
```
#define Large       0x00000001      /* L2 64KB */
#define Small       0x00000002      /* L2 4KB */
```

The Domain Access Control (DAC) and Access Permission (AP) bits together control who can access a page. Client means the MMU checks the AP bits; Manager means the domain bypasses all permission checks. For AP: Krw = kernel read/write only, Uro = user read-only, Urw = user read/write:

⟨*type* DAC*(arm)* 179a⟩≡ (888c)
```
#define Client      1           /* DAC */
#define Manager     3           /* DAC */
```

⟨*type* PageTableEntryAttribute3*(arm)* 179b⟩≡ (888c)
```
#define Noaccess    0       /* AP, DAC */
#define Krw     1           /* AP */
/* armv7 deprecates AP[2] == 1 & AP[1:0] == 2 (Uro), prefers 3 (new in v7) */
#define Uro     2           /* AP */
#define Urw     3           /* AP */
```

⟨*constant* Fault*(arm)* 179c⟩≡ (888c)
```
#define Fault       0x00000000      /* L[12] pte: unmapped */
```

⟨*constant* Dom0*(arm)* 179d⟩≡ (888c)
```
#define Dom0        0
```

⟨*macro* L1AP*(arm)* 179e⟩≡ (888c)
```
#define L1AP(ap)    (AP(3, (ap)))
```

⟨*macro* AP*(arm)* 179f⟩≡ (888c)
```
#define AP(n, v)    F((v), ((n)*2)+4, 2)
```

⟨*macro* F*(arm)* 179g⟩≡ (888c)
```
#define F(v, o, w)  (((v) & ((1<<(w))-1))<<(o))
```

⟨*macro* DAC*(arm)* 179h⟩≡ (888c)
```
#define DAC(n, v)   F((v), (n)*2, 2)
```

⟨*macro* L2AP*(arm)* 179i⟩≡ (1006b)
```
#define L2AP(ap)    l2ap(ap)
```

⟨*function* l2ap*(raspberry pi1)(arm)* 179j⟩≡ (994b)
```
int
l2ap(int ap)
{
    return (AP(3, (ap))|AP(2, (ap))|AP(1, (ap))|AP(0, (ap)));
}
```
Uses AP 179f.

8.3 Pagination and mmupd/mmupt (ARM)

Now we see how Plan 9 uses the ARM MMU hardware. Each CPU has a single L1 page directory (cpu->mmul1), shared between the kernel (upper half, mapped with Section entries during boot and never changed) and the current user process (lower half, populated on demand by arch_putmmu() and cleared on context switch by mmul1empty()). Each process keeps its own list of L2 page table pages in Proc.mmul2. When the scheduler switches to a new process, arch_mmuswitch() invalidates the old user mappings from the L1 and the TLB; the new process's pages are faulted in on demand.

⟨Cpu Arch *mmu fields(arm)* 179k⟩≡ (64e) 188a▷
```
PTE*    mmul1;      /* l1 for this processor */
```

⟨*type* PTE*(arm)* 179l⟩≡ (973b)
```
typedef u32int      PTE;
```

`L1lo` and `L1hi` delimit the range of L1 entries used for user-space mappings. Only entries between `L1lo` (index 0, at `UZERO`) and `L1hi` (just past `USTKTOP`) are cleared on context switch; the kernel entries above `KZERO` are left untouched:

⟨*constant* `L1lo` *(arm)* 180a⟩≡ (1006a)
```
L1lo        = UZERO/MiB,        /* L1X(UZERO)? */
```
Uses `MiB` 48a.

⟨*constant* `L1hi` *(arm)* 180b⟩≡ (1006a)
```
L1hi        = (USTKTOP+MiB-1)/MiB,  /* L1X(USTKTOP+MiB-1)? */
```
Uses `MiB` 48a.

⟨`Proc Arch` *memory fields(arm)* 180c⟩≡ (67a) 181b▷
```
// list<ref<Page> > pages used to store page tables
Page*   mmul2;
```

⟨*macro* `PPN` *(arm)* 180d⟩≡ (1005)
```
#define PPN(x)      ((x)&~(BY2PG-1))
```

⟨*macro* `VA` *(arm)* 180e⟩≡ (1004b)
```
#define VA(k)       ((uintptr)(k))
```

When a process dies (`Moribund` state), `schedinit()` calls `arch_mmurelease()` to free all its hardware page table resources: the L2 pages are zeroed and returned to the free page list, the L1 user entries are cleared, and the TLB is invalidated.

⟨*function* `arch_mmurelease` *(arm)* 180f⟩≡ (1006b)
```
void
arch_mmurelease(Proc* proc)
{
    Page *page, *next;

    ⟨arch_mmurelease() invalidate cache before adjusting page tables(arm) 531i⟩
    mmul2empty(proc, false);
    ⟨arch_mmurelease() free l2 page cache(arm) 181d⟩
    mmul1empty();
    ⟨arch_mmurelease() write back cache after adjusting page tables(arm) 531b⟩
    /* lose any possible stale tlb entries */
    mmuinvalidate();
}
```
Uses `false`, `mmul1empty()` 181e, and `mmul2empty()` 181a.

⟨*function* `mmuinvalidate` *(arm)* 180g⟩≡ (1007a)
```
/*
 * invalidate tlb
 */
TEXT mmuinvalidate(SB), 1, $-4
    MOVW    $0, R0
    MCR CpSC, 0, R0, C(CpTLB), C(CpTLBinvu), CpTLBinv
    BARRIERS
    RET
```

⟨*type CpTLB secondary registers(arm)* 180h⟩≡ (888c)
```
/*
 * CpTLB Secondary (CRm) registers and opcode2 fields.
 */
#define CpTLBinvu   7           /* unified */
```

⟨*type CpTLB opcode2(arm)* 180i⟩≡ (888c)
```
#define CpTLBinv    0           /* invalidate all */
#define CpTLBinvse  1           /* invalidate single entry */
```

`mmul2empty()` walks the per-process list of L2 page table pages. For each one, it sets the corresponding L1 entry to `Fault` so the MMU will fault on any access to that region. When `clear` is true (process exit), the L2 page contents are also zeroed. The freed pages are not returned to `palloc` immediately—they are spliced onto `mmul2cache` for reuse by the same process, avoiding the cost of allocating fresh pages on the next context switch.

⟨*function* `mmul2empty`*(arm)* 181a⟩≡ (1006b)
```
static void
mmul2empty(Proc* proc, bool clear)
{
    PTE *l1;
    Page **l2, *page;

    l1 = cpu->mmul1;
    l2 = &proc->mmul2;
    for(page = *l2; page != nil; page = page->next){
        if(clear)
            memset(UINT2PTR(page->va), 0, BY2PG);
        l1[page->daddr] = Fault;
        l2 = &page->next;
    }
    ⟨mmul2empty() remember free pages(arm) 181c⟩
}
```
Uses `Fault` 179c and `UINT2PTR` 56h.

⟨`Proc Arch` *memory fields(arm)* 181b⟩+≡ (67a) ◁180c
```
// list<ref<Page> > (next = Page.next)
Page*   mmul2cache; /* free mmu pages */
```

⟨`mmul2empty()` *remember free pages(arm)* 181c⟩≡ (181a)
```
*l2 = proc->mmul2cache;
proc->mmul2cache = proc->mmul2;
proc->mmul2 = nil;
```

⟨`arch_mmurelease()` *free l2 page cache(arm)* 181d⟩≡ (180f)
```
for(page = proc->mmul2cache; page != nil; page = next){
    next = page->next;
    if(--page->ref)
        panic("mmurelease: page->ref %d", page->ref);
    pagechainhead(page);
}
if(proc->mmul2cache && palloc.freememr.p)
    wakeup(&palloc.freememr);
proc->mmul2cache = nil;
```
Uses `pagechainhead()` 111b, `palloc` 109b, and `wakeup`.

⟨*function* `mmul1empty`*(arm)* 181e⟩≡ (1006b)
```
static void
mmul1empty(void)
{
    memset(&cpu->mmul1[L1lo], 0, (L1hi - L1lo)*sizeof(PTE));
}
```
Uses `L1hi-1848` 180b and `L1lo-1847` 180a.

When the pager modifies a process's page tables (e.g., swapping out a page), it sets `newtlb` to signal that the process must flush its TLB entries next time it runs. `arch_flushmmu()` forces an immediate flush:

⟨`Proc` *memory fields* 181f⟩+≡ (66) ◁119a 213b▷
```
bool newtlb;    /* Pager has changed my pte's, I must flush */
```

⟨*function* `arch_flushmmu`*(arm)* 182a⟩≡ (1006b)

```
void
arch_flushmmu(void)
{
    int s;

    s = arch_splhi();
    up->newtlb = true;
    arch_mmuswitch(up);
    arch_splx(s);
}
```

Uses `true` and `up`.

8.4 Page fault

This is where interrupt handling meets memory management. When a data abort or prefetch abort reaches `arch__trap()`, it calls `faultarm()` with the faulting address. The call chain is:

```
faultarm(ARM) -> fault(portable) -> fixfault(portable)
                                        -> arch_putmmu(ARM)
```

The ARM-specific code extracts the faulting address and determines read vs. write; the portable code decides what to do about it.

8.4.1 `faultarm()` (ARM)

`faultarm()` is the ARM-specific wrapper: it calls the portable `fault()` function with the faulting address. If `fault()` returns an error in user mode, a note is posted (the process can handle it or will be killed); if it is a kernel fault, the system panics.

⟨*function* `faultarm`*(arm)* 182b⟩≡ (1002c)

```
/*
 *  called by trap to handle access faults
 */
static void
faultarm(Ureg *ureg, virt_addr va, bool user, bool read)
{
    bool insyscall;
    int n;
    char buf[ERRMAX];

    if(up == nil) {
        //dumpregs(ureg);
        panic("fault: nil up in faultarm, pc %#p accessing %#p", ureg->pc, va);
    }
    insyscall = up->insyscall;
    up->insyscall = true; // ???
    ⟨faultarm() if debug 647d⟩
    n = fault(va, read); // portable code

    if(n < 0){
        if(!user){
            dumpregs(ureg);
            panic("fault: kernel accessing %#p", va);
        }
        /* don't dump registers; programs suicide all the time */
```

```
            snprint(buf, sizeof buf, "sys: trap: fault %s va=%#p",
                read? "read": "write", va);
            postnote(up, 1, buf, NDebug);
        }
        up->insyscall = insyscall;
    }
```

Uses ERRMAX, NDebug 449a, fault() 183, postnote, true, and up.

8.4.2 fault()

The portable fault() function finds which segment contains the faulting address (via seg()[119b]) and calls fixfault() to resolve it. The for(;;) loop handles the case where fixfault() returns an error because the segment was invalidated while waiting for I/O—it simply retries.

⟨*function* fault 183⟩≡ (863b)

```
    int
    fault(virt_addr addr, bool read)
    {
        Segment *s;
        char *sps;

        if(up == nil)
            panic("fault: nil up");
        if(up->nlocks.ref)
            print("fault: addr %#p: nlocks %ld\n", addr, up->nlocks.ref);

        sps = up->psstate;
        up->psstate = "Fault";
        arch_spllo();

        cpu->pfault++;
        for(;;) {
            s = seg(up, addr, /*dolock*/true); /* leaves s->lk qlocked if seg != nil */
            if(s == nil) {
                up->psstate = sps;
                return -1;
            }
            if(!read && (s->type&SG_RONLY)) {
                qunlock(&s->lk);
                up->psstate = sps;
                return -1;
            }

            if(fixfault(s, addr, read, /*putmmu*/true) == 0) /* qunlocks s->lk */
                break;

            // else? try again?
        }

        up->psstate = sps;
        return 0;
    }
```

Uses SG_RONLY 855, fixfault() 185, print, qunlock() 82c, seg() 119b, true, and up.

8.4.3 fixfault()

fixfault() is the heart of virtual memory. It walks the software segment tables to find (or create) the Page

for the faulting address, then calls `arch_putmmu()` to install the mapping in the hardware page table. What it does depends on the segment type:

```
SG_TEXT:  demand-load from executable (read-only)
SG_BSS:  allocate a zero-filled page
SG_STACK: allocate a zero-filled page
SG_DATA:  demand-load, or copy-on-write if shared
```

The `PTEVALID`, `PTERONLY`, and `PTEWRITE` flags are software-defined bits that `fixfault()` passes to `arch_putmmu()`, which translates them into the hardware AP bits (`Uro` or `Urw`):

⟨*type* `PTExxx` 184⟩≡ (855)

```
#define PTEVALID    (1<<0)
#define PTERONLY    0
#define PTEWRITE    (1<<1)
```

⟨`PTExxx` *other cases* 558a⟩

Before reading the code, it helps to see the structure that `fixfault` walks. A `Segment` owns a two-level software page table that is independent from the hardware L1/L2 tables maintained by `arch_putmmu`. The first level, `pagedir`, has one entry per `PAGETABMAPMEM`-sized chunk of virtual address space and each entry either is `nil` or points at a `Pagetable`; the second level, `pagetab[]`, has one slot per page, each either `nil` (page not yet materialised), a pointer to a `Page` (resident), or an opaque swap/daddr marker (paged out):

```
Segment s (e.g. the BSS of /bin/ls)
  base  = 0x00400000
  top   = 0x00410000
  pagedir ----> +-----------+
                |    [0] ---|---> Pagetable
                |    [1] ---|---> Pagetable
                |    ...    |
                |   [255]nil|
                +-----------+


Pagetable (one per 1 MiB chunk)
  pagetab[0]  ----> Page { pa=0x3fa000, ref=2, va=..., next, ... }
  pagetab[1]  ----> nil                (will fault on first touch)
  pagetab[2]  ----> <daddr>|PG_ONSWAP  (will fault, then swap in)
  pagetab[..] ....
  first, last  (bounds for fast iteration during teardown)
```

A concrete first-touch write fault on a BSS page makes this less abstract. Suppose the user does `*(int*)0x404050 = 1;` on a process that has never touched that address before:

1. The ARM writes to `0x404050`, the MMU has no L2 entry, hardware raises a data abort at virtual address `0x404050`. Control ends up in `faultarm`, then `fault`, then `fixfault` with `addr=0x404050`.

2. `soff = 0x404050 - 0x400000 = 0x4050`. Dividing by `PAGETABMAPMEM` (1 MiB) gives `pagedir` index 0. `*pde` is `nil` on first access, so `ptalloc` allocates a new `Pagetable` on the fly.

3. Inside the new `Pagetable`, the per-page index is `(0x4050 \& 0xFFFFF) / 0x1000 = 4`. `pagetab[4]` is `nil`—this is the telltale of "untouched BSS," which falls into the `SG_BSS` branch.

4. `newpage(true, ...)` returns a fresh zeroed `Page` and stores it in `pagetab[4]`. The code falls through `goto common`: the page has one reference, no image, so copy-on-write does not apply. `mmupte = PPN(page->pa) | PTEWRITE | PTEVALID`.

5. `arch_putmmu` installs the hardware L1/L2 entry for `0x404000` and invalidates the TLB for that address.

6. The user instruction is restarted by `RFE`; this time the MMU finds the mapping and the write completes.

The `SG_TEXT` branch is similar, except `pagedout` is possible (the text page is on disk in the executable) and the resulting PTE is `PTERONLY`. The `SG_DATA` branch adds the copy-on-write check: if the faulting access is a write and the refcount is above one, a fresh private copy is allocated before the new mapping is installed.

⟨*function* `fixfault` 185⟩≡ (863b)

```
errorneg1
fixfault(Segment *s, virt_addr addr, bool read, bool doputmmu)
{
    int type;
    int ref;
    Pagetable **pde, *pt;
    Page **pte, *lkp, *new;
    ulong mmupte = nilptr;
    ulong soff;

    addr &= ~(BY2PG-1);
    soff = addr - s->base;

    // walk
    pde = &s->pagedir[soff/PAGETABMAPMEM];
    if(*pde == nil)
        *pde = ptalloc();
    pt = *pde;
    pte = &pt->pagetab[(soff&(PAGETABMAPMEM-1))/BY2PG];
    if(pte < pt->first)
        pt->first = pte;
    if(pte > pt->last)
        pt->last = pte;

    type = s->type&SG_TYPE;

    switch(type) {
    case SG_TEXT:            /* Demand load */
        ⟨fixfault() page in for SG_TEXT pte if pagedout 197b⟩

        mmupte = PPN((*pte)->pa) | PTERONLY|PTEVALID;
        (*pte)->modref = PG_REF;
        break;

    case SG_BSS:
    case SG_SHARED:          /* Zero fill on demand */
    case SG_STACK:
        if(*pte == nil) {
            new = newpage(true, &s, addr); // true so clear! zeroed BSS & heap!
            if(s == nil) //?? when can be nil at exit?
                return ERROR_NEG1;
            *pte = new;
        }
        goto common;

    case SG_DATA:
    common:          /* Demand load/pagein/copy on write */
        ⟨fixfault() page in for SG_DATA or swapin (SG_BSS, etc) pte if pagedout 197c⟩

        ⟨fixfault() if read and copy on write, adjust mmupte and break 189a⟩
        // else
```

```
            lkp = *pte;

            lock(lkp);

            if(lkp->image == &swapimage)
                ref = lkp->ref + swapcount(lkp->daddr);
            else
                ref = lkp->ref;
            ⟨fixfault() if one ref and page has an image 199b⟩
            unlock(lkp);

            ⟨fixfault() if write and more than one ref then copy page 189b⟩

            mmupte = PPN((*pte)->pa) | PTEWRITE | PTEVALID;
            (*pte)->modref = PG_MOD|PG_REF;
            break;

        ⟨fixfault() SG_PHYSICAL case 557d⟩

        default:
            panic("fault");
            break;
        }
        qunlock(&s->lk);

        if(doputmmu)
            arch_putmmu(addr, mmupte, *pte);

        return OK_0;
    }
```

Uses ERROR_NEG1, OK_0, PG_MOD 108b, PG_REF 108b, PTERONLY 184, PTEVALID 184, PTEWRITE 184, SG_BSS, SG_DATA 556b, SG_SHARED, SG_STACK 112b, SG_TEXT, SG_TYPE, lock() 76d, nilptr, ptalloc() 115a, qunlock() 82c, swapcount() 207c, swapimage 205b, true, and unlock() 78d.

8.4.4 arch_putmmu() (ARM)

arch_putmmu() installs a mapping in the hardware page table. If the L1 entry for this virtual address is Fault (no L2 table yet), it allocates a page for the L2 table (reusing one from the per-process mmul2cache if available). It then writes the L2 entry with the physical page address and the appropriate permission bits, and invalidates the TLB entry for this address so the MMU picks up the new mapping.

⟨*function* arch_putmmu*(arm)* 186⟩≡ (1006b)

```
  void
  arch_putmmu(virt_addr va, phys_addr pa, Page* page)
  {
      int x;
      Page *pg;
      PTE *l1, *pte;

      x = L1X(va);
      l1 = &cpu->mmul1[x];
      if(*l1 == Fault){
          /* wasteful - l2 pages only have 256 entries - fix */
          ⟨arch_putmmu() if mmul2cache not empty(arm) 187a⟩
          else{
              /* auxpg since we don't need much? memset if so */
              pg = newpage(true, nil, 0);
```

```
            pg->va = VA(arch_kmap(pg));
        }
        pg->daddr = x;

        //add_list(pg, up->mmul2)
        pg->next = up->mmul2;
        up->mmul2 = pg;

        ⟨arch_putmmu() write back cache before fixing va fault(arm) 531h⟩

        *l1 = PPN(pg->pa)|Dom0|Coarse;
        ⟨arch_putmmu() write back cache after changing pde(arm) 531f⟩

        ⟨arch_putmmu() maintain mmul1lo and mmul1hi(arm) 188b⟩
    }
    pte = UINT2PTR(KADDR(PPN(*l1)));

    /* protection bits are
     *  PTERONLY|PTEVALID;
     *  PTEWRITE|PTEVALID;
     *  PTEWRITE|PTEUNCACHED|PTEVALID;
     */
    x = Small;
    ⟨arch_putmmu() if PTEUNCACHED, do not change x(arm) 558b⟩
    else
        x |= L2ptedramattrs;

    if(pa & PTEWRITE)
        x |= L2AP(Urw);
    else
        x |= L2AP(Uro);

    pte[L2X(va)] = PPN(pa)|x;
    ⟨arch_putmmu() write back cache after changing pte(arm) 531g⟩

    /* clear out the current entry */
    mmuinvalidateaddr(PPN(va));

    /*  write back dirty entries - we need this because the pio() in
     *  fault.c is writing via a different virt addr and won't clean
     *  its changes out of the dcache.  Page coloring doesn't work
     *  on this mmu because the virtual cache is set associative
     *  rather than direct mapped.
     */
    ⟨arch_putmmu() if PG_TXTFLUSH, invalidate cache(arm) 555c⟩
    arch_checkmmu(va, PPN(pa));
}
```

Uses Coarse 178d, Dom0 179d, Fault 179c, L1X-1842 178a, L2AP-1844 179i, L2X-1843 178b, L2ptedramattrs-1846 530b, PTEWRITE 184, Small 178f, UINT2PTR 56h, Uro 179b, Urw 179b, true, and up.

⟨arch_putmmu() *if mmul2cache not empty(arm)* 187a⟩≡ (186)

```
if(up->mmul2cache != nil){
    pg = up->mmul2cache;
    up->mmul2cache = pg->next;
    memset(UINT2PTR(pg->va), 0, BY2PG);
}
```

Uses UINT2PTR 56h and up.

⟨*function* mmuinvalidateaddr*(arm)* 187b⟩≡ (1007a)

```
/*
```

```
 * mmuinvalidateaddr(va)
 *   invalidate tlb entry for virtual page address va, ASID 0
 */
TEXT mmuinvalidateaddr(SB), 1, $-4
    MCR CpSC, 0, R0, C(CpTLB), C(CpTLBinvu), CpTLBinvse
    BARRIERS
    RET
```

⟨Cpu Arch *mmu fields(arm)* 188a⟩+≡ (64e) ◁179k 230a▷

```
// index in mmul1
int mmul1lo;
int mmul1hi;
```

⟨arch_putmmu() *maintain mmul1lo and mmul1hi(arm)* 188b⟩≡ (186)

```
if(x >= cpu->mmul1lo && x < cpu->mmul1hi){
    if(x+1 - cpu->mmul1lo < cpu->mmul1hi - x)
        cpu->mmul1lo = x+1;
    else
        cpu->mmul1hi = x;
}
```

⟨*function* arch_checkmmu*(arm)* 188c⟩≡ (1006b)

```
void
arch_checkmmu(virt_addr va, phys_addr pa)
{
    USED(va);
    USED(pa);
}
```

`arch_kmap()` makes a physical page accessible to the kernel. On ARM with less than 2 GiB RAM, this is trivial: add `KZERO` to get the virtual address. `arch_kunmap()` flushes the cache for that page:

⟨*function* arch_kmap*(arm)* 188d⟩≡ (1006b)

```
Arch_KMap*
arch_kmap(Page *p) {
  return (Arch_KMap*)((p)->pa|KZERO);
}
```

⟨*function* arch_kunmap*(arm)* 188e⟩≡ (1006b)

```
void
arch_kunmap(Arch_KMap *k)
{
    ⟨arch_kunmap() write back cache for page(arm) 531j⟩
}
```

8.5 Copy on write

When `rfork` duplicates a data segment, it does not immediately copy all pages. Instead, both parent and child share the same physical pages, marked read-only. When either process writes to a shared page, the write triggers a permission fault. `fixfault()` sees that the page has `ref > 1` (multiple users), allocates a fresh page, copies the content, and gives the writer the private copy with write permission. This is copy on write (COW)—it avoids copying pages that are never modified. There is a subtlety: if the segment has only one user (`s->ref == 1`) and the access is a read, the page can be mapped read-only without copying, even if `conf.copymode` would normally trigger a copy. This optimization is controlled by `conf.copymode`:

⟨Conf *other fields* 188f⟩≡ (62a) 205a▷

```
int copymode; /* 0 is copy on write, 1 is copy on reference */
```

⟨fixfault() *if read and copy on write, adjust mmupte and break* 189a⟩≡ (185)

```
/*
 *  It's only possible to copy on write if
 *  we're the only user of the segment.
 */
if(read && conf.copymode == false && s->ref == 1) {
    mmupte = PPN((*pte)->pa)|PTERONLY|PTEVALID;
    (*pte)->modref |= PG_REF;
    break;
}
```

Uses PG_REF 108b, PTERONLY 184, PTEVALID 184, conf 62b, and false.

⟨fixfault() *if write and more than one ref then copy page* 189b⟩≡ (185)

```
if(ref > 1){
    new = newpage(false, &s, addr);
    if(s == nil)
        return -1;
    *pte = new;
    copypage(lkp, *pte);
    putpage(lkp); //why??
}
```

Uses copypage() 189c, false, and putpage() 111a.

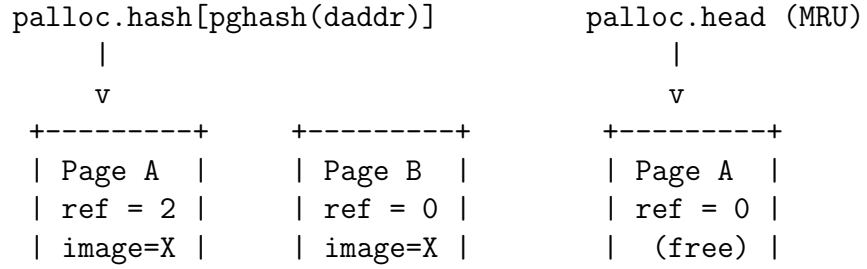
copypage() performs the actual page copy via temporary kernel mappings (arch_kmap):

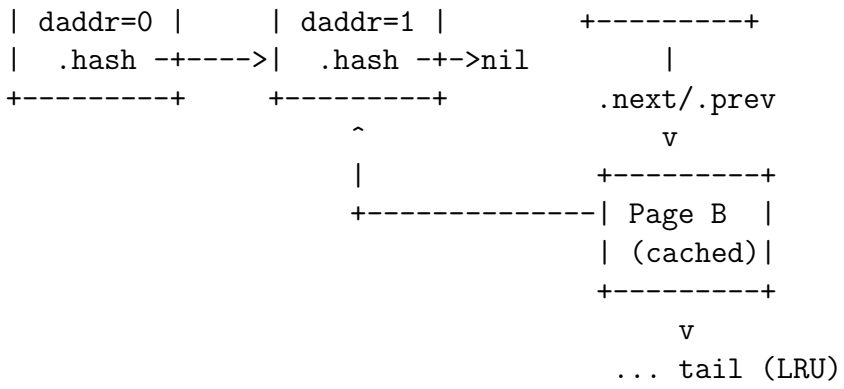
⟨*function* copypage 189c⟩≡ (863d)

```
void
copypage(Page *f, Page *t)
{
    Arch_KMap *ks, *kd;

    ks = arch_kmap(f);
    kd = arch_kmap(t);
    memmove((void*)VA(kd), (void*)VA(ks), BY2PG);
    arch_kunmap(ks);
    arch_kunmap(kd);
}
```

8.6 Page cache

The page cache avoids redundant disk reads. When the kernel loads a page from an executable (demand loading) or from the swap file, it registers the page in a hash table keyed by (image, disk address). If the same page is needed again—e.g., two instances of rc sharing the same text—lookpage() finds it in the cache instead of reading from disk. Pages in the cache can be simultaneously in use (referenced by a segment) or free (on palloc's free list but still in the hash table, ready to be reclaimed if needed before being recycled). The same Page can be chained into two lists at once: palloc.hash (the page cache, keyed by daddr) and palloc.head/palloc.tail (the LRU free list). The chains use different link fields— Page.hash for the cache, Page.next/Page.prev for the LRU—so they do not collide:





Page A has `ref == 2` (two segments map it) so it is only in the hash chain. Page B has `ref == 0`—it was freed but its `image`/`daddr` are still set, so it lives in both the hash chain and the free list. `lookpage()`[191a] on `(X, 1)` still finds Page B and calls `pageunchain()` to pull it off the LRU, bumping `ref` back to 1—a cache hit with zero disk I/O. This is why `putpage()`[111a] puts cached pages at the tail of the free list (via `pagechaintail()`) while uncached ones go to the head: tail pages are the last to be recycled, giving the cache the longest possible window.

Each `Page` can be associated with a `KImage` (an open executable or the swap file) and a disk address (`daddr`). Together, these form the cache key:

⟨`Page` *other fields* 190a⟩≡ (108a) 214d▷
```
// option<ref<Kimage>>
KImage  *image;     /* Associated binary image or swap image */
ulong daddr;      /* Disc address on image */
```

⟨*function* `pghash` 190b⟩≡ (855)
```
enum
{
    PGHLOG  = 9, // 2^9 = 512
    PGHSIZE = 1<<PGHLOG,  /* Page hash for image lookup */
};
#define pghash(daddr) palloc.hash[(daddr>>PGSHIFT)&(PGHSIZE-1)]
```

⟨`Palloc` *other fields* 190c⟩≡ (108c)
```
// hash<Page.daddr, ref<Page>> (next = Page.hash)>
Page  *hash[PGHSIZE];
Lock  hashlock;
```

⟨`Page` *extra fields* 190d⟩+≡ (108a) ◁109a
```
// hash<daddr, ref<Page>> Palloc.hash
Page  *hash; /* Image hash chains */
```

Note that a page can be simultaneously in the page cache and on the free list. When a process exits, its text pages are freed (`ref` goes to 0) but remain in the cache. If another process execs the same binary before those pages are recycled, `lookpage()` finds them and bumps `ref` back to 1—avoiding a disk read. This is why `putpage()` places pages with images at the tail of the free list: they are the last to be recycled, maximizing the chance of a cache hit.

`cachepage()` adds a page to the cache after loading it from disk. `lookpage()` searches the cache before attempting a disk read. `uncachepage()` removes a page when it is recycled for a different purpose:

⟨*function* `cachepage` 190e⟩≡ (863d)
```
void
cachepage(Page *p, KImage *i)
{
    Page **l; // list

    /* If this ever happens it should be fixed by calling
```

```
         * uncachepage instead of panic. I think there is a race
         * with pio in which this can happen. Calling uncachepage is
         * correct - I just wanted to see if we got here.
         */
        if(p->image)
            panic("cachepage");

        incref(i);
        lock(&palloc.hashlock);
        p->image = i;
        // add_hash(palloc.hash, p->daddr, p)
        l = &pghash(p->daddr);
        p->hash = *l;
        *l = p;
        unlock(&palloc.hashlock);
    }
```

Uses `incref()` 87b, `lock()` 76d, `palloc` 109b, `pghash` 108c, and `unlock()` 78d.

`lookpage()` searches the hash table for a cached page. If found and the page was free (ref was 0), it unchains it from the free list—the page is now in use again without any disk I/O:

⟨*function* `lookpage` 191a⟩≡ (863d)

```
Page *
lookpage(KImage *i, ulong daddr)
{
    Page *pg;

    lock(&palloc.hashlock);
    for(pg = pghash(daddr); pg; pg = pg->hash) {
        if(pg->image == i && pg->daddr == daddr) {
            unlock(&palloc.hashlock);

            lock(&palloc);
            lock(pg);
            if(pg->image != i || pg->daddr != daddr) { // race, not the one anymore
                unlock(pg);
                unlock(&palloc);
                return nil;
            }
            if(++pg->ref == 1)
                pageunchain(pg);
            unlock(&palloc);
            unlock(pg);

            return pg;
        }
    }
    unlock(&palloc.hashlock);

    return nil;
}
```

Uses `lock()` 76d, `pageunchain()` 110d, `palloc` 109b, `pghash` 108c, and `unlock()` 78d.

⟨`putpage` *if p has an image* 191b⟩≡ (111a)

```
if(p->image && p->image != &swapimage)
    pagechaintail(p);
```

Uses `pagechaintail()` 112a and `swapimage` 205b.

⟨`newpage()` *uncachepage* 191c⟩≡ (110a)

```
uncachepage(p);
```

⟨*function* uncachepage 192a⟩≡ (863d)
```
/* Always called with a locked page */
void
uncachepage(Page *p)
{
    Page **l, *f;

    if(p->image == nil)
        return;

    lock(&palloc.hashlock);
    // remove_hash(palloc.hash, p->daddr, p)
    l = &pghash(p->daddr);
    for(f = *l; f; f = f->hash) {
        if(f == p) {
            *l = p->hash;
            break;
        }
        l = &f->hash;
    }
    unlock(&palloc.hashlock);
    putimage(p->image);
    p->image = nil;
    p->daddr = 0;
}
```
Uses `lock()` 76d, `palloc` 109b, `pghash` 108c, `putimage()` 201d, and `unlock()` 78d.

`cachedel()` removes a page from the cache by (image, daddr) key rather than by `Page` pointer. It is used by `pagepte()` when allocating a swap slot: if a stale cache entry already exists at that swap address (from a previous eviction cycle), it must be cleared before the slot is reused. The double-check pattern (test, lock, retest) handles the race where another thread modifies the page between the hash lookup and the page lock:

⟨*function* cachedel 192b⟩≡ (863d)
```
void
cachedel(KImage *i, ulong daddr)
{
    Page *f, **l;

    lock(&palloc.hashlock);
    l = &pghash(daddr);
    for(f = *l; f; f = f->hash) {
        if(f->image == i && f->daddr == daddr) {
            lock(f);
            // can have a race? things could have changed, so rested under lock
            if(f->image == i && f->daddr == daddr){
                *l = f->hash;
                putimage(f->image); // =~ decref
                f->image = nil;
                f->daddr = 0;
            }
            unlock(f);
            break;
        }
        l = &f->hash;
    }
    unlock(&palloc.hashlock);
}
```
Uses `lock()` 76d, `palloc` 109b, `pghash` 108c, `putimage()` 201d, and `unlock()` 78d.

8.7 Demand loading

When `sysexec()` creates the text and data segments, it does not load any pages from disk. Instead, the segment tables are left empty (`nil` page entries), and the segment records which file to read from (`Segment.image`, `Segment.fstart`, `Segment.flen`). When the process first touches a page, `fixfault()` sees `*pte == nil`, calls `pio()` to load the page from the executable on disk, and installs it. This is demand loading—pages are loaded only when actually accessed. Demand loading also enables text sharing: if two processes run the same executable, `attachimage()` finds the existing `KImage` in the image cache and shares its segment. The pages loaded by the first process are found via `lookpage()` by the second, avoiding duplicate reads.

8.7.1 `Kimage` and the image cache

A `KImage` wraps a `Chan` (open file handle) to an executable binary or the swap file. It also holds a back pointer to the `Segment` that uses it, enabling text sharing: when a second process execs the same binary, `attachimage()` finds the existing `KImage` and shares its segment (and thus its already-loaded pages):

⟨*struct* `KImage` 193a⟩≡ (855)
```
// a KImage is essentially a channel to an executable or swapfile
struct KImage
{
    Chan  *c;     /* channel to text file */
    bool  notext;     /* no file associated */ // for swapfile
    ⟨Kimage other fields 195a⟩

    // extra
    Ref;
    ⟨Kimage extra fields 193c⟩
};
```

`Imagealloc` is a fixed-size pool of `KImage` structures. The hash table maps `Qid` paths to cached images, enabling `attachimage()` to find an existing image for an executable. The free list holds unused `KImage` slots. When the free list is empty, `imagereclaim()` evicts pages from the page cache, which may free their associated `KImage` structures:

⟨*struct* `Imagealloc` 193b⟩≡ (855)
```
// Image allocator (internal to segment.c, but important so here, singleton)
struct Imagealloc
{
    // hash<qid.path, ref<Kimage>> (next = Kimage.hash)
    KImage  *hash[IHASHSIZE];

    // list<ref<Kimage> (next = Kimage.next)
    KImage  *free; // originally  xalloc'ed in imageinit() (conf.nimage)
    QLock ireclaim; /* mutex on reclaiming free images */

    ⟨Imagealloc other fields 202a⟩

    // extra
    Lock;
};
```

⟨`Kimage` *extra fields* 193c⟩≡ (193a) 194d▷
```
// list<ref<Kimage>> of Imagealloc.free
KImage  *next; /* Free list */
// hash<qid.path, ref<Kimage>> Imagealloc.hash
KImage  *hash; /* Qid hash chains */
```

⟨*function* ihash 194a⟩≡ (855)
```
#define IHASHSIZE 64
// actually internal to page.c, but important so here
#define ihash(qidpath)  imagealloc.hash[qidpath%IHASHSIZE]
```

⟨*global* imagealloc 194b⟩≡ (864b)
```
static struct Imagealloc imagealloc;
```
Uses Imagealloc 193b.

8.7.2 Segment and KImage

Each segment that supports demand loading has a pointer to its KImage and records the file offset (fstart) and length (flen) of the segment's data within the executable:

⟨Segment *other fields* 194c⟩+≡ (113b) ◁117b 195b▷
```
KImage  *image;   /* text in file attached to this segment */
```

⟨Kimage *extra fields* 194d⟩+≡ (193a) ◁193c
```
// option<ref<Segment>>?
Segment *s;     /* TEXT segment for image if running */
```

⟨*constructor* attachimage 194e⟩≡ (864b)
```
KImage*
attachimage(int type, Chan *c, virt_addr base, ulong len)
{
    KImage *img, **l;

    /* reclaim any free channels from reclaimed segments */
    if(imagealloc.nfreechan)
        imagechanreclaim();

    lock(&imagealloc);

    /*
     * Search the image cache for remains of the text from a previous
     * or currently running incarnation
     */
    for(img = ihash(c->qid.path); img; img = img->hash) {
        if(c->qid.path == img->qid.path) {
            lock(img);
            if(eqqid(c->qid, img->qid) &&
               eqqid(c->mqid, img->mqid) &&
               c->mchan == img->mchan &&
               c->type == img->type) {
                goto found;
            }
            unlock(img);
        }
    }

    /*
     * imagereclaim dumps pages from the free list which are cached by image
     * structures. This should free some image structures.
     */
    while(!(img = imagealloc.free)) {
        unlock(&imagealloc);
        imagereclaim();
        sched();
        lock(&imagealloc);
```

```
    }

    imagealloc.free = img->next;

    lock(img);
    incref(c);
    img->c = c;
    img->type = c->type;
    img->qid = c->qid;
    img->mqid = c->mqid;
    img->mchan = c->mchan;
    //add_hash(imagealloc.hash, c->qid.path, img)
    l = &ihash(c->qid.path);
    img->hash = *l;
    *l = img;

found:
    unlock(&imagealloc);

    if(img->s == nil) {
        /* Disaster after commit in exec */
        if(waserror()) {
            unlock(img);
            pexit(Enovmem, /*freemem*/true);
        }
        img->s = newseg(type, base, len);
        img->s->image = img;
        img->ref++;
        poperror();
    }
    else
        incref(img->s);

    return img;
}
```

Uses Enovmem 664d, ihash 193b, imagealloc-1803 194b, imagechanreclaim() 202b, incref() 87b, lock() 76d, newseg() 116a, pexit, poperror 666d, sched, true, unlock() 78d, and waserror 666d.

⟨Kimage *other fields* 195a⟩≡ (193a)

```
Qid   qid;      /* Qid for page cache coherence */
Chan  *mchan;
Qid   mqid;
ushort  type;     /* Device type of owning channel */
```

`attachimage()` is called by `sysexec()` to find or create a `KImage` for the executable. It searches the image hash table by `Qid`—if the same binary is already cached (from a previous or concurrent exec), the existing `KImage` and its shared segment are reused. Otherwise, a fresh `KImage` is allocated from the free list and a new segment is created.

⟨Segment *other fields* 195b⟩+≡ (113b) ◁194c 474a▷

```
ulong fstart;   /* start address in file for demand load */
ulong flen;   /* length of segment in file */
```

⟨sysexec() *locals* 195c⟩+≡ (141c) ◁146a 321e▷

```
KImage *img;
```

Here is how `sysexec()` sets up demand loading. It calls `attachimage()` which either finds a cached `KImage` for this executable or creates a new one. The text segment's `fstart` is 0 (text starts at the beginning of the

binary, after the `Exec` header) and `flen` covers the header plus text bytes. The data segment's `fstart` is set just past the text, so `pio()` reads from the right offset when a data page is first touched:

⟨`sysexec()` *get text segment ts via demand loading on tc* 196a⟩≡ (143f)
```
/* attachimage returns a locked cache image */
img = attachimage(SG_TEXT|SG_RONLY, tc, UTZERO, (t-UTZERO)>>PGSHIFT);
ts = img->s;
ts->fstart = 0;
ts->flen = sizeof(Exec)+text;
⟨sysexec() initialize other fields 555f⟩
unlock(img);
```

⟨`sysexec()` *adjust data segment s for demand loading on tc* 196b⟩≡ (144a)
```
/* Attached by hand */
incref(img);
s->image = img;
s->fstart = ts->fstart+ts->flen;
s->flen = data;
// data is also in binary
```

When `rfork` duplicates a data segment, the new segment inherits the parent's `KImage` reference and file offsets, so demand loading still works for pages that have not yet been touched:

⟨`dupseg()` `SG_DATA` *case, attach image to new segment n* 196c⟩≡ (122b)
```
incref(s->image); // how sure non nil? data always attached to an img
n->image = s->image;
n->fstart = s->fstart;
n->flen = s->flen;
```
Uses `incref()` 87b.

There is a subtle case when duplicating the text segment slot (`TSEG`): the process may hold it as an `SG_DATA` segment (from a previous `exec` that shared text via `KImage`). `data2txt()` converts it back to `SG_TEXT` with proper demand-loading attributes so the child can share the read-only text pages:

⟨`dupseg()` *when* `SG_DATA`*, if Text segment* 196d⟩≡ (122b)
```
if(segno == TSEG){ // why not SG_TEXT then?
    poperror();
    qunlock(&s->lk);
    return data2txt(s);// ????
}
```
Uses `TSEG` 118c, `data2txt()` 196e, `poperror` 666d, and `qunlock()` 82c.

⟨*function* `data2txt` 196e⟩≡ (864b)
```
Segment*
data2txt(Segment *s)
{
    Segment *ps;

    ps = newseg(SG_TEXT, s->base, s->size);
    ps->image = s->image;
    incref(ps->image);
    ps->fstart = s->fstart;
    ps->flen = s->flen;
    ⟨data2txt() initialize other segment fields 555h⟩

    return ps;
}
```
Uses `SG_TEXT`, `incref()` 87b, and `newseg()` 116a.

8.7.3 Paging in

A page table entry is "paged out" if it is either `nil` (never loaded—demand loading) or marked as a swap address (`onswap`). In both cases, `pio()` must read data from disk before the page can be used:

⟨*function* `pagedout` 197a⟩≡ (855)
```
#define pagedout(s) (((ulong)s)==0 || onswap(s))
```

⟨`fixfault()` *page in for* `SG_TEXT` *pte if pagedout* 197b⟩≡ (185)
```
if(pagedout(*pte))
    pio(s, addr, soff, pte);
```
Uses `pagedout` 855 and `pio()` 197d.

⟨`fixfault()` *page in for* `SG_DATA` *or swapin (*`SG_BSS`*, etc) pte if pagedout* 197c⟩≡ (185)
```
if(pagedout(*pte))
    pio(s, addr, soff, pte);
```
Uses `pagedout` 855 and `pio()` 197d.

`pio()` ("page I/O") loads a page from disk. It first checks the page cache (`lookpage()`)—if the page is already cached, no I/O is needed. Otherwise, it allocates a fresh page, reads the data from the executable's channel (or the swap file), and adds the page to the cache. The segment lock (`s->lk`) is released during the I/O to avoid blocking other processes; this creates a race (another process might fault on the same page), which is handled by rechecking after re-acquiring the lock.

⟨*function* `pio` 197d⟩≡ (863b)
```
void
pio(Segment *s, virt_addr addr, ulong soff, PageOrSwap **p)
{
    Page *new;
    Arch_KMap *k;
    Chan *c;
    int n, ask;
    char *kaddr;
    ulong daddr; // disk address
    Page *loadrec;

retry:
    loadrec = *p;

    if(loadrec == nil) {  /* from a text/data image */
        daddr = s->fstart+soff;
        new = lookpage(s->image, daddr);
        if(new != nil) {
            *p = new;
            return;
        }

        c = s->image->c;
        ask = s->flen-soff;
        if(ask > BY2PG)
            ask = BY2PG;
    }else{          /* from a swap image */
        daddr = swapaddr(loadrec);
        new = lookpage(&swapimage, daddr);
        if(new != nil) {
            putswap(loadrec);
            *p = new;
            return;
        }
        c = swapimage.c;
```

```
    ask = BY2PG;
}
qunlock(&s->lk);

new = newpage(false, nil, addr);
k = arch_kmap(new);
kaddr = (char*)VA(k);

while(waserror()) {
    if(strcmp(up->errstr, Eintr) == 0)
        continue;
    arch_kunmap(k);
    putpage(new);
    faulterror(Eioload, c, false);
}

// reading the Page!! slow! which is why it's done without s->lk locked
n = devtab[c->type]->read(c, kaddr, ask, daddr);

if(n != ask)
    faulterror(Eioload, c, false);
if(ask < BY2PG)
    memset(kaddr+ask, 0, BY2PG-ask);

poperror();
arch_kunmap(k);
qlock(&s->lk);

if(loadrec == nil) {  /* This is demand load */
    /*
     *  race, another proc may have gotten here first while
     *  s->lk was unlocked
     */
    if(*p == nil) {
        new->daddr = daddr;
        cachepage(new, s->image);
        *p = new;
    }
    else
        putpage(new);
}else{          /* This is paged out */
    /*
     *  race, another proc may have gotten here first
     *  (and the pager may have run on that page) while
     *  s->lk was unlocked
     */
    if(*p != loadrec){
        if(!pagedout(*p)){
            /* another process did it for me */
            putpage(new);
            goto done; // =~ return, just need adjust cachectl before
        } else {
            /* another process and the pager got in */
            putpage(new);
            goto retry;
        }
    }

    new->daddr = daddr;
    cachepage(new, &swapimage);
```

```
            *p = new;
            putswap(loadrec);
        }
    done:
    ⟨pio() set cachectl if flushme segment 555l⟩
    }
```

Uses Eioload 664d, cachepage() 190e, false, faulterror() 199a, lookpage() 191a, pagedout 855, poperror 666d, putpage() 111a, putswap() 206d, qlock() 81f, qunlock() 82c, swapaddr 556d, swapimage 205b, up, and waserror 666d.

⟨*function* faulterror 199a⟩≡ (863b)

```
static void
faulterror(char *s, Chan *c, bool freemem)
{
    char buf[ERRMAX];

    if(c && c->path){
        snprint(buf, sizeof buf, "%s accessing %s: %s", s, c->path->s, up->errstr);
        s = buf;
    }
    if(up->nerrlab) {
        postnote(up, 1, s, NDebug);
        error(s);
    }
    pexit(s, freemem);
}
```

Uses ERRMAX, NDebug 449a, pexit, postnote, and up.

8.7.4 duppage()

duppage() handles a subtle case in copy-on-write. When a page has ref == 1 but is cached (has an image), the page is about to be modified—but the cache still holds a reference to the original content. duppage() makes a copy of the page for the cache, so the original can be safely modified. This preserves the invariant that cached pages always reflect their disk content.

⟨fixfault() *if one ref and page has an image* 199b⟩≡ (185)

```
if(ref == 1 && lkp->image){
    /* save a copy of the original for the image cache */
    duppage(lkp);
    ref = lkp->ref;
}
```

Uses duppage() 199d.

⟨*global* dupretries 199c⟩≡ (863d)

```
static int dupretries = 15000;
```

Uses dupretries-1850 199c.

⟨*function* duppage 199d⟩≡ (863d)

```
/* Always call with p locked */
int
duppage(Page *p)
{
    Page *np;
    int retries;
 int color;

    retries = 0;
retry:
```

```
    if(retries++ > dupretries){
        print("duppage %d, up %p\n", retries, up);
        dupretries += 100;
        if(dupretries > 100000)
            panic("duppage\n");
        uncachepage(p);
        return 1;
    }


    /* don't dup pages with no image */
    if(p->ref == 0 || p->image == nil || p->image->notext)
        return 0;

    /*
     *  normal lock ordering is to call
     *  lock(&palloc) before lock(p).
     *  To avoid deadlock, we have to drop
     *  our locks and try again.
     */
    if(!canlock(&palloc)){
        unlock(p);
        if(up)
            sched();
        lock(p);
        goto retry;
    }

    /* No freelist cache when memory is very low */
    if(palloc.freecount < swapalloc.highwater) {
        unlock(&palloc);
        uncachepage(p);
        return 1;
    }

 color = getpgcolor(p->va);
 for(np = palloc.head; np; np = np->next)
  if(np->color == color)
   break;

 /* No page of the correct color */
 if(np == 0) {
  unlock(&palloc);
  uncachepage(p);
  return 1;
 }
    //when no color: np = palloc.head;

    pageunchain(np);
    pagechaintail(np);
/*
* XXX - here's a bug? - np is on the freelist but it's not really free.
* when we unlock palloc someone else can come in, decide to
* use np, and then try to lock it.  they succeed after we've
* run copypage and cachepage and unlock(np).  then what?
* they call pageunchain before locking(np), so it's removed
* from the freelist, but still in the cache because of
* cachepage below.  if someone else looks in the cache
* before they remove it, the page will have a nonzero ref
* once they finally lock(np).
```

```
 */
    lock(np);
    unlock(&palloc);

    /* Cache the new version */
    uncachepage(np);
    np->va = p->va;
    np->daddr = p->daddr;
    copypage(p, np);
    cachepage(np, p->image);
    unlock(np);
    uncachepage(p);

    return 0;
}
```

Uses `cachepage()` 190e, `canlock()` 78f, `copypage()` 189c, `dupretries-1850` 199c, `lock()` 76d, `pagechaintail()` 112a, `pageunchain()` 110d, `palloc` 109b, `print`, `sched`, `uncachepage()` 192a, `unlock()` 78d, and `up`.

8.7.5 Collecting images

When a segment is freed (`putseg()`) and it was the last reference, the `KImage` is detached (its `s` pointer set to `nil`) and its reference count decremented. If the `KImage` itself reaches zero references, it is removed from the image hash table and returned to the free list. Channel closing is deferred to avoid blocking while holding spin locks.

⟨`putseg()` *other locals* 201a⟩≡ (116c)

```
KImage *img;
```

⟨`putseg()` *if s has an image* 201b⟩≡ (116c)

```
img = s->image;
if(img != nil) {
    lock(img);
    lock(s);
    if(img->s == s && s->ref == 1) // race?
        img->s = nil;
    unlock(img);
}
```

Uses `lock()` 76d and `unlock()` 78d.

⟨`putseg()` *if s had an image* 201c⟩≡ (116c)

```
if(img)
    putimage(img);
```

Uses `putimage()` 201d.

`putimage()` decrements the `KImage` reference count. When it reaches zero, the image is removed from the hash table and returned to the free list. Closing the underlying `Chan` is deferred into `imagealloc.freechan` because `cclose()` can block (it may involve I/O to a file server), and we are holding spin locks at this point:

⟨*destructor* `putimage` 201d⟩≡ (864b)

```
void
putimage(KImage *img)
{
    Chan *c, **cp;
    KImage *f, **l;

    if(img->notext)
        return;
```

```
    lock(img);
    if(--img->ref == 0) {
        l = &ihash(img->qid.path);
        mkqid(&img->qid, ~0, ~0, QTFILE);
        unlock(img);
        c = img->c;

        lock(&imagealloc);
        //remove_hash(imagealloc.hash, img)
        for(f = *l; f; f = f->hash) {
            if(f == img) {
                *l = img->hash;
                break;
            }
            l = &f->hash;
        }

        //add_list(imagealloc.free, img)
        img->next = imagealloc.free;
        imagealloc.free = img;

        /* defer freeing channel till we're out of spin lock's */

        if(imagealloc.nfreechan == imagealloc.szfreechan){
            //realloc(imagealloc.freenchan, szfreechan+NFREECHAN)
            imagealloc.szfreechan += NFREECHAN;
            cp = malloc(imagealloc.szfreechan*sizeof(Chan*));
            if(cp == nil)
                panic("putimage");
            memmove(cp, imagealloc.freechan, imagealloc.nfreechan*sizeof(Chan*));
            free(imagealloc.freechan);
            imagealloc.freechan = cp;
        }

        imagealloc.freechan[imagealloc.nfreechan++] = c;
        unlock(&imagealloc);

        return;
    }
    unlock(img);
}
```

Uses NFREECHAN 194a, QTFILE, free() 104a, ihash 193b, imagealloc-1803 194b, lock() 76d, malloc() 103c, and unlock() 78d.

⟨Imagealloc *other fields* 202a⟩≡ (193b)

```
Chan  **freechan; /* free image channels */
int nfreechan;  /* number of free channels */
int szfreechan; /* size of freechan array */
QLock fcreclaim;  /* mutex on reclaiming free channels */
```

`imagechanreclaim()` drains the deferred channel close queue. It is called at the start of `attachimage()`, a safe point where no spin locks are held. The `fcreclaim` qlock ensures only one thread does the cleanup:

⟨*function* imagechanreclaim 202b⟩≡ (864b)

```
/*
 *  since close can block, this has to be called outside of
 *  spin locks.
 */
static void
imagechanreclaim(void)
{
```

```
    Chan *c;

    /* Somebody is already cleaning the image chans */
    if(!canqlock(&imagealloc.fcreclaim))
        return;

    /*
     * We don't have to recheck that nfreechan > 0 after we
     * acquire the lock, because we're the only ones who decrement
     * it (the other lock contender increments it), and there's only
     * one of us thanks to the qlock above.
     */
    while(imagealloc.nfreechan > 0){
        lock(&imagealloc);
        imagealloc.nfreechan--;
        c = imagealloc.freechan[imagealloc.nfreechan];
        unlock(&imagealloc);
        cclose(c);
    }

    qunlock(&imagealloc.fcreclaim);
}
```

Uses canqlock() 83a, cclose, imagealloc-1803 194b, lock() 76d, qunlock() 82c, and unlock() 78d.

imagereclaim() is the last resort when attachimage() cannot find a free KImage structure. It walks the free page list from the tail (where putpage() places pages that still have cache associations), uncaching up to 1000 pages. Each uncachepage() call decrements the image's reference count, potentially freeing the KImage for reuse:

⟨*function* imagereclaim 203⟩≡ (864b)

```
static void
imagereclaim(void)
{
    int n;
    Page *p;
    uvlong ticks; // fastticks

    irstats.calls++;
    /* Somebody is already cleaning the page cache */
    if(!canqlock(&imagealloc.ireclaim))
        return;

    lock(&palloc);
    ticks = arch_fastticks(nil);
    n = 0;
    /*
     * All the pages with images backing them are at the
     * end of the list (see putpage) so start there and work
     * backward.
     */
    for(p = palloc.tail; p && p->image && n<1000; p = p->prev) {
        if(p->ref == 0 && canlock(p)) {
            if(p->ref == 0) {
                n++;
                uncachepage(p); // will call putimage()
            }
            unlock(p);
        }
    }
    ticks = arch_fastticks(nil) - ticks;
```

```
    unlock(&palloc);
    irstats.loops++;
    irstats.ticks += ticks;
    if(ticks > irstats.maxt)
        irstats.maxt = ticks;
    //print("T%llud+", ticks);
    qunlock(&imagealloc.ireclaim);
}
```

8.8 Swapping

When physical memory runs low, the kernel must evict pages to disk to make room. This is swapping (or paging out). The evicted page's content is written to a swap file, and the page table entry is replaced with the swap address. When the process touches that page again, a page fault triggers `pio()` to read it back. The swap subsystem mirrors the page allocator: where `Palloc` manages physical pages, `Swapalloc` manages slots in the swap file. Each slot is one page (4 KiB), and the `swmap` array is a reference count for each slot (because forked processes can share swap entries via copy-on-write).

8.8.1 Swapalloc

`Swapalloc` mirrors `Palloc` for the swap file. Where `Palloc.pages` tracks physical pages, `Swapalloc.swmap` is a byte array where each entry is a reference count for the corresponding 4 KiB slot in the swap file. A byte suffices because a single swap slot is shared at most by 254 processes (copy-on-write forks). The `last` pointer speeds allocation by remembering where the previous scan stopped, so `newswap()` does not always search from the beginning.

⟨*struct* Swapalloc 204a⟩≡ (855)

```
// Swap allocator (singleton)
struct Swapalloc
{
    // array<byte> xalloc'ed in swapinit()
    // each idx represents a chunk of swapimage, each value a ref count
    byte*  swmap;      /* Base of swap map in memory */

    int free;     /* currently free swap pages */

    // ref<byte> in swmap
    byte*  alloc;     /* Round robin allocator */
    // ref<byte> in swmap
    byte*  top;      /* Top of swap map */

    // ref<byte> in swmap
    byte*  last;     /* Speed swap allocation */

    ulong highwater;    /* Pager start threshold */ // = 5% conf.upages
    ulong headroom;   /* Space pager frees under highwater */ // = 1.25*hw

    //extra
    Lock;       /* Free map lock */
    Rendez r;      /* Pager kproc idle sleep */ // needpages()
};
```

⟨*global* swapalloc 204b⟩≡ (859a)

```
// Swap allocator
struct Swapalloc swapalloc;
```

Uses `Swapalloc` 204a.

⟨Conf *other fields* 205a⟩+≡ (62a) ◁188f 208a▷
```
  ulong nswap;    /* number of swap pages */
```

The swap file is represented as a KImage just like executable binaries, which lets the page cache machinery work uniformly for both demand loading and swap-in. The notext flag distinguishes it: swapimage has no associated text file, just a raw swap device:

⟨*global* swapimage 205b⟩≡ (859a)
```
  KImage  swapimage;
```

newswap() allocates a slot in the swap file by scanning swmap for a zero entry (unused slot) starting from last. It sets the reference count to 1 and returns the byte offset in the swap file (slot index times BY2PG). If no free slots remain, it returns 0 and the pager gives up on this page:

⟨*function* newswap 205c⟩≡ (866b)
```
  ulong
  newswap(void)
  {
      byte *look;

      lock(&swapalloc);

      if(swapalloc.free == 0){
          unlock(&swapalloc);
          return ~0; //???
      }

      look = memchr(swapalloc.last, 0, swapalloc.top-swapalloc.last);
      if(look == nil)
          panic("inconsistent swap"); // swapalloc.free != 0, should find a page

      *look = 1;
      swapalloc.last = look;
      swapalloc.free--;
      unlock(&swapalloc);
      return (look-swapalloc.swmap) * BY2PG; // offset in swapfile
  }
```
Uses lock() 76d, swapalloc 204b, and unlock() 78d.

8.8.2 /dev/swap

Reading /dev/swap gives a summary of all the memory subsystems we have seen: total physical memory, page size, kernel vs. user pages, swap usage, and malloc pool sizes. It is a convenient diagnostic:

⟨consread() *cases* 205d⟩≡ (394c) 264d▷
```
  case Qswap:
      snprint(tmp, sizeof tmp,
          "%lud memory\n"
          "%d pagesize\n"
          "%lud kernel\n"
          "%lud/%lud user\n"
          "%lud/%lud swap\n"
          "%lud/%lud kernel malloc\n"
          "%lud/%lud kernel draw\n",
          conf.npage*BY2PG,
          BY2PG,
          conf.npage-conf.upages,
          palloc.user - palloc.freecount, palloc.user,
          conf.nswap - swapalloc.free, conf.nswap,
          mainmem->cursize, mainmem->maxsize,
```

```
        imagmem->cursize, imagmem->maxsize);

    return readstr((ulong)offset, buf, n, tmp);
```

Uses Qswap-1790 633a, conf 62b, imagmem 102b, mainmem 102b, palloc 109b, readstr() 670d, and swapalloc 204b.

8.8.3 PageOrSwap

How does a page table entry indicate "this page is on swap"? By a clever encoding trick: since `Page*` pointers are word-aligned, their lowest bit is always 0. Setting bit 0 (`PG_ONSWAP`) marks the value as a swap address instead of a `Page` pointer. The remaining bits encode the offset in the swap file. This is the `PageOrSwap` type—a union disguised as a pointer:

⟨*constant* PG_ONSWAP 206a⟩≡ (855)
```
#define PG_ONSWAP 1
```

⟨*type* PageOrSwap 206b⟩≡ (855)
```
typedef Page PageOrSwap;
```

⟨*function* onswap 206c⟩≡ (855)
```
#define onswap(s) (((kern_addr)s)&PG_ONSWAP)
```

`putswap()`, `dupswap()`, and `swapcount()` manage the reference counts in `swmap`. `putswap()` decrements the count and frees the slot when it reaches zero, updating `last` so the freed slot is found quickly by `newswap()`. `dupswap()` increments the count when `ptcpy()` duplicates a page table entry that points to swap (i.e., during `rfork`):

⟨*function* putswap 206d⟩≡ (866b)
```
void
putswap(Page *p)
{
    byte *idx;

    lock(&swapalloc);
    idx = &swapalloc.swmap[((ulong)p)/BY2PG];
    if(--(*idx) == 0) {
        swapalloc.free++;
        if(idx < swapalloc.last)
            swapalloc.last = idx;
    }
    if(*idx >= 254)
        panic("putswap %#p == %ud", p, *idx);
    unlock(&swapalloc);
}
```

Uses lock() 76d, swapalloc 204b, and unlock() 78d.

⟨putpage *if p is a swap address* 206e⟩≡ (111a)
```
    if(onswap(p)) {
        putswap(p);
        return;
    }
```

Uses onswap 216a and putswap() 206d.

⟨mfreeseg *if pg is a swap address* 206f⟩≡ (125c)
```
if(onswap(pg))
    putswap(pg);
```

Uses onswap 216a and putswap() 206d.

⟨*function* dupswap 207a⟩≡ (866b)
```
void
dupswap(Page *p)
{
    lock(&swapalloc);
    if(++swapalloc.swmap[((ulong)p)/BY2PG] == 0)
        panic("dupswap");
    unlock(&swapalloc);
}
```
Uses lock() 76d, swapalloc 204b, and unlock() 78d.

⟨ptcpy() *if src is a swap page* 207b⟩≡ (115c)
```
if(onswap(*src))
    dupswap(*src);
```
Uses dupswap() 207a and onswap 216a.

⟨*function* swapcount 207c⟩≡ (866b)
```
int
swapcount(ulong daddr)
{
    return swapalloc.swmap[daddr/BY2PG];
}
```
Uses swapalloc 204b.

8.8.4 Swapping plan

When the pager decides to evict a page, `pagepte()` handles the details per segment type. For text pages, eviction is free: the page is simply discarded (it can be demand-loaded again from the executable). For data, bss, and stack pages, the content must be written to the swap file: `pagepte()` allocates a swap slot, caches the page under the swap image (so a concurrent fault can find it during the write), replaces the `Page*` in the page table with the swap address, and adds the page to `iolist` for the actual disk write.

⟨*function* pagepte 207d⟩≡ (866b)
```
static void
pagepte(int type, Page **pte)
{
    ulong daddr;
    Page *outp;

    outp = *pte;
    switch(type) {
    case SG_TEXT:               /* Revert to demand load */
        putpage(outp);
        *pte = nil;
        break;

    case SG_DATA:
    case SG_BSS:
    case SG_STACK:
    case SG_SHARED:
        /*
         *  get a new swap address and clear any pages
         *  referring to it from the cache
         */
        daddr = newswap();
        if(daddr == ~0)
            break; // return;
        cachedel(&swapimage, daddr);
```

```
        lock(outp);

        /* forget anything that it used to cache */
        uncachepage(outp);

        /*
         *  incr the reference count to make sure it sticks around while
         *  being written
         */
        outp->ref++;

        /*
         *  enter it into the cache so that a fault happening
         *  during the write will grab the page from the cache
         *  rather than one partially written to the disk
         */
        outp->daddr = daddr;
        cachepage(outp, &swapimage);
        *pte = (Page*)(daddr|PG_ONSWAP); // turn a Page pte into a Swap pte!
        unlock(outp);

        /* Add page to IO transaction list */
        iolist[ioptr++] = outp;
        break;
    }
}
```

Uses PG_ONSWAP, SG_BSS, SG_DATA 556b, SG_SHARED, SG_STACK 112b, SG_TEXT, cachedel() 192b, cachepage() 190e, iolist-1807 208b, ioptr-1808 208c, lock() 76d, newswap() 205c, putpage() 111a, swapimage 205b, uncachepage() 192a, and unlock() 78d.

⟨Conf *other fields* 208a⟩+≡ (62a) ◁205a 276b▷

```
int nswppo;   /* max # of pageouts per segment pass */
```

⟨*global* iolist 208b⟩≡ (866b)

```
// array<option<ref<Page>>>, xalloc'ed in swapinit(), size = Conf.nswppo
static  Page    **iolist;
```

⟨*global* ioptr 208c⟩≡ (866b)

```
// index in iolist
static  int ioptr;
```

`pageout()` walks a segment's page tables looking for pages to evict. It uses an aging algorithm: each page records when it was last referenced (`gen`). Pages older than `genage` are candidates for eviction. Recently referenced pages (`PG_REF` set) get their generation reset, giving them a second chance.

⟨*function* pageout 208d⟩≡ (866b)

```
static void
pageout(Proc *p, Segment *s)
{
    int type, i, size;
    ulong age;
    Pagetable *pt;
    Page **pg, *entry;

    if(!canqlock(&s->lk))   /* We cannot afford to wait, we will surely deadlock */
        return;

    if(s->steal) {      /* Protected by /dev/proc */
        qunlock(&s->lk);
```

```
        return;
    }

    if(!canflush(p, s)) {   /* Able to invalidate all tlbs with references */
        qunlock(&s->lk);
        putseg(s);
        return;
    }

    if(waserror()) {
        qunlock(&s->lk);
        putseg(s);
        return;
    }

    /* Pass through the tables looking for memory pages to swap out */
    type = s->type&SG_TYPE;
    size = s->pagedirsize;
    for(i = 0; i < size; i++) {
        pt = s->pagedir[i];
        if(pt == nil)
            continue;
        for(pg = pt->first; pg < pt->last; pg++) {
            entry = *pg;
            if(pagedout(entry)) // already swapped, or nil
                continue;

            if(entry->modref & PG_REF) {
                entry->modref &= ~PG_REF;
                entry->gen = genclock;
            }


            if(genclock < entry->gen)
                age = ~(entry->gen - genclock);
            else
                age = genclock - entry->gen;
            gensum += age;
            gencount++;
            if(age <= genage)
                continue;


            pagepte(type, pg);

            if(ioptr >= conf.nswppo)
                goto out;
        }
    }
out:
    poperror();
    qunlock(&s->lk);
    putseg(s);
}
```

Uses PG_REF 108b, SG_TYPE, canflush() 213f, canqlock() 83a, conf 62b, genage-1809 214c, genclock-1810 214c, gencount-1811 214c, gensum-1812 214c, ioptr-1808 208c, pagedout 855, pagepte() 207d, poperror 666d, putseg() 116c, qunlock() 82c, and waserror 666d.

8.8.5 The pager

The pager is a kernel process (started by `kproc()` during initialization) that runs when free memory drops below `swapalloc.highwater`. When `newpage()` cannot find enough free pages, it calls `kickpager()` to wake the pager, which scans all processes' segments looking for old pages to evict. Once enough pages are freed, the pager goes back to sleep. This is the classic "page daemon" found in all UNIX-like systems.

⟨`newpage()` *other locals* 210a⟩+≡ (110a) ◁110b 554h▷
```
bool dontalloc;
```

When free memory is below `highwater`, `newpage()` must wait for the pager to free pages. But it cannot hold the segment lock while sleeping—that would deadlock, since the pager needs to lock segments to evict their pages. So `newpage()` drops the segment lock (setting `*s = nil`) and returns `nil` after the pager runs. The caller (`fixfault()`) detects the `nil` segment, returns an error, and `fault()`'s `for(;;)` loop retries with fresh locks.

⟨`newpage()` *loop waiting freecount ¿ highwater* 210b⟩≡ (110a)
```
for(;;) {
    if(palloc.freecount > swapalloc.highwater)
        break;
    if(up->kp && palloc.freecount > 0)
        break;

    // in highwater, not so many free pages, need to wait

    unlock(&palloc);
    dontalloc = false;
    if(s && *s) {
        qunlock(&((*s)->lk));
        *s = nil;// !!
        dontalloc = true;
    }
    qlock(&palloc.pwait);   /* Hold memory requesters here */

    while(waserror())   /* Ignore interrupts */
        ;

    kickpager();
    tsleep(&palloc.freememr, hasfreepages, 0, 1000);

    poperror();
    qunlock(&palloc.pwait);

    /*
     * If called from fault and we lost the segment from
     * underneath don't waste time allocating and freeing
     * a page. Fault will call newpage again when it has
     * reacquired the segment locks
     */
    if(dontalloc)
        return nil;

    lock(&palloc);
}
```

⟨`Palloc` *extra fields* 210c⟩≡ (108c)
```
Rendez  freememr; /* Sleep for free mem */ // hasfreepages()
QLock pwait; /* Queue of procs waiting for memory */
```

⟨putpage *wakeup process waiting for freepage if any* 211a⟩≡ (111a)
```
if(palloc.freememr.p != nil)
    wakeup(&palloc.freememr);
```
Uses palloc 109b and wakeup.

Notice the hysteresis between `highwater` and `headroom`: `newpage()` blocks when free pages drop below `highwater` (5% of user pages), but the pager keeps running until free pages rise above `headroom` (6.25%). This gap prevents the system from oscillating between "need pages" and "have enough"—the pager frees a batch of pages before going back to sleep.

⟨*function* hasfreepages 211b⟩≡ (863d)
```
int
hasfreepages(void*)
{
    return palloc.freecount >= swapalloc.highwater;
}
```
Uses palloc 109b and swapalloc 204b.

`kickpager()` lazily creates the pager kernel process on first call, then simply wakes it up on subsequent calls. This avoids running the pager at all on systems that never run low on memory:

⟨*function* kickpager 211c⟩≡ (866b)
```
void
kickpager(void)
{
    static bool started;

    if(started)
        wakeup(&swapalloc.r);
    else {
        kproc("kpager", pager, nil);
        started = true;
    }
}
```
Uses kproc() 157c, pager() 211d, swapalloc 204b, true, and wakeup.

The pager's main loop is a round-robin scan over all processes. Each iteration advances to the next process and calls `pageout()` on each of its segments. For non-text segments, the evicted pages are batched in `iolist` and then written to the swap file by `executeio()`. After a full pass through the process table, `gentick()` updates the generation clock and recalculates the average page age, which adjusts the eviction threshold for the next pass. When no swap file is configured, the pager falls back to `killbig()`—it simply kills the largest process.

⟨*function* pager 211d⟩≡ (866b)
```
static void
pager(void *junk)
{
    int i;
    Segment *s;
    Proc *p, *ep;

    if(waserror())
        panic("pager: os error");

    p = proctab(0);
    ep = &p[conf.nproc];

loop:
    up->psstate = "Idle";
    wakeup(&palloc.freememr);
```

```
    sleep(&swapalloc.r, needpages, nil);

    while(needpages(junk)) {
        if(swapimage.c) {
            p++;
            if(p >= ep){
                p = proctab(0);
                gentick();
            }

            if(p->state == Dead || p->noswap)
                continue;

            if(!canqlock(&p->seglock))
                continue;       /* process changing its segments */

            for(i = 0; i < NSEG; i++) {
                if(!needpages(junk)){
                    qunlock(&p->seglock);
                    goto loop;
                }

                if(s = p->seg[i]) {
                    switch(s->type&SG_TYPE) {
                    case SG_TEXT:
                        pageout(p, s);
                        break;
                    case SG_DATA:
                    case SG_BSS:
                    case SG_STACK:
                    case SG_SHARED:
                        up->psstate = "Pageout";
                        pageout(p, s);
                        if(ioptr != 0) {
                            up->psstate = "I/O";
                            executeio();
                        }
                        break;
                    default:
                        break;
                    }
                }
            }
            qunlock(&p->seglock);
        } else {
            print("out of memory\n");
            killbig("out of memory");
            freebroken();       /* can use the memory */

            /* Emulate the old system if no swap channel */
            if(!swapimage.c)
                tsleep(&up->sleepr, returnfalse, 0, 5000);
        }
    }
    goto loop;
}
```

Uses Dead 67c, NSEG 118c, SG_BSS, SG_DATA 556b, SG_SHARED, SG_STACK 112b, SG_TEXT, SG_TYPE, canqlock() 83a, conf 62b, executeio() 215a, freebroken() 484a, gentick() 214b, ioptr-1808 208c, killbig() 876d, needpages() 213a, pageout() 208d, palloc 109b, print, proctab, qunlock() 82c, returnfalse() 104c, sleep, swapalloc 204b, swapimage 205b,

tsleep, up, wakeup, and waserror 666d.

⟨*function* needpages 213a⟩≡ (866b)
```
static bool
needpages(void*)
{
    return palloc.freecount < swapalloc.headroom;
}
```
Uses palloc 109b and swapalloc 204b.

⟨Proc *memory fields* 213b⟩+≡ (66) ◁181f
```
bool noswap;   /* process is not swappable */
```

⟨newproc() *set fields of* p 213c⟩+≡ (130a) ◁159d 219d▷
```
p->noswap = false;
```
Uses false.

⟨sysrfork() *when* RFPROC *propagate fields from up to p* 213d⟩+≡ (134b) ◁163d 220c▷
```
p->noswap = up->noswap;
```
Uses up.

⟨procctlreq() *CMnoswap case* 213e⟩≡ (620b)
```
case CMnoswap:
    p->noswap = true;
    break;
```
Uses CMnoswap-884 602b and true.

Before evicting pages from a segment, the pager must ensure that no processor has stale TLB entries pointing to those pages. `canflush()` checks whether all processes sharing this segment can be paged—i.e., they are not currently running on any CPU. `canpage()` does the actual check: if `p->cpu == nil`, the process is not running and its `newtlb` flag is set so the TLB will be flushed before it resumes. If the process is currently executing, the pager skips this segment rather than blocking.

⟨*function* canflush 213f⟩≡ (866b)
```
static bool
canflush(Proc *p, Segment *s)
{
    int i;
    Proc *ep;

    lock(s);
    if(s->ref == 1) {       /* Easy if we are the only user */
        s->ref++;
        unlock(s);
        return canpage(p);
    }
    s->ref++;
    unlock(s);

    /* Now we must do hardwork to ensure all processes which have tlb
     * entries for this segment will be flushed if we succeed in paging it out
     */
    p = proctab(0);
    ep = &p[conf.nproc];
    while(p < ep) {
        if(p->state != Dead) {
            for(i = 0; i < NSEG; i++)
                if(p->seg[i] == s)
                    if(!canpage(p))
```

```
                return false;
            }
            p++;
        }
        return true;
    }
```

Uses Dead 67c, NSEG 118c, canpage() 214a, conf 62b, false, lock() 76d, proctab, true, and unlock() 78d.

⟨*function* canpage 214a⟩≡ (877c)

```
int
canpage(Proc *p)
{
    bool ok = false;

    arch_splhi();
    lock(runq);
    /* Only reliable way to see if we are Running */
    if(p->cpu == nil) {
        p->newtlb = true;
        ok = true;
    }
    unlock(runq);
    arch_spllo();

    return ok;
}
```

Uses false, lock() 76d, runq 221b, true, and unlock() 78d.

The page aging algorithm uses a simple generation clock. Each page records the genclock value when it was last referenced (PG_REF set). gentick() advances the clock and computes genage as the average age of all pages examined in the last pass. Pages older than genage are eviction candidates—this is an adaptive threshold that automatically adjusts to the workload: under heavy memory pressure many pages are old, so genage is high and eviction is more aggressive.

⟨*function* gentick 214b⟩≡ (866b)

```
static void
gentick(void)
{
    genclock++;
    if(gencount)
        genage = gensum / gencount;
    else
        genage = 0;
    gensum = gencount = 0;
}
```

Uses genage-1809 214c, genclock-1810 214c, gencount-1811 214c, and gensum-1812 214c.

⟨*global* genxxx 214c⟩≡ (866b)

```
static  ulong   genage, genclock, gencount;
static  uvlong  gensum;
```

⟨Page *other fields* 214d⟩+≡ (108a) ◁190a 554a▷

```
ulong gen;      /* Generation counter for swap */
```

8.8.6 Paging out

`executeio()` writes the batch of evicted pages to the swap file. It sorts the pages by `daddr` first, so the writes are sequential on disk (minimizing seek time on spinning disks). After each page is written, its extra reference (added by `pagepte()`) is dropped and the page is freed via `putpage()`.

⟨*function* executeio 215a⟩≡ (866b)

```
static void
executeio(void)
{
    Page *out;
    int i, n;
    Chan *c;
    char *kaddr;
    Arch_KMap *k;

    c = swapimage.c;
    qsort(iolist, ioptr, sizeof iolist[0], pageiocomp);
    for(i = 0; i < ioptr; i++) {
        if(ioptr > conf.nswppo)
            panic("executeio: ioptr %d > %d", ioptr, conf.nswppo);
        out = iolist[i];
        k = arch_kmap(out);
        kaddr = (char*)VA(k);

        if(waserror())
            panic("executeio: page out I/O error");

        n = devtab[c->type]->write(c, kaddr, BY2PG, out->daddr);// swap 1 page
        if(n != BY2PG)
            nexterror();

        arch_kunmap(k);
        poperror();

        /* Free up the page after I/O */
        lock(out);
        out->ref--;
        unlock(out);
        putpage(out);
    }
    ioptr = 0;
}
```

Uses `conf` 62b, `iolist-1807` 208b, `ioptr-1808` 208c, `lock()` 76d, `nexterror`, `pageiocomp()` 215b, `poperror` 666d, `putpage()` 111a, `swapimage` 205b, `unlock()` 78d, and `waserror` 666d.

⟨*function* pageiocomp 215b⟩≡ (866b)

```
static int
pageiocomp(void *a, void *b)
{
    Page *p1, *p2;

    p1 = *(Page **)a;
    p2 = *(Page **)b;
    if(p1->daddr > p2->daddr)
        return 1;
    else
        return -1;
}
```

8.8.7 Paging in

⟨*function* swapaddr 216a⟩≡ (855)
```
#define swapaddr(s) (((ulong)s)&~PG_ONSWAP)
```

8.8.8 Setting up the swap file

The swap file is configured at runtime by writing to `/dev/swap`. The boot scripts open a file (typically on the local disk) and write its file descriptor number to `/dev/swap`, which calls `setswapchan()`. Writing "start" just kicks the pager without changing the swap file. This late-binding approach means swapping is optional—embedded systems can run without a swap file, and the pager will fall back to killing processes when memory runs out.

⟨conswrite() *cases* 216b⟩≡ (395a) 265b▷
```
case Qswap:
    if(n >= sizeof buf)
        error(Egreg);
    memmove(buf, va, n);    /* so we can NUL-terminate */
    buf[n] = 0;
    /* start a pager if not already started */
    if(strncmp(buf, "start", 5) == 0){
        kickpager();
        break;
    }
    if(!iseve())
        error(Eperm);
    if(buf[0]<'0' || '9'<buf[0])
        error(Ebadarg);
    fd = strtoul(buf, 0, 0);
    swc = fdtochan(fd, -1, true, true);
    setswapchan(swc);
    break;
```
Uses `Ebadarg` 664d, `Egreg` 664d, `Qswap-1790` 633a, `fdtochan()` 302d, `iseve()` 591b, `kickpager()` 211c, `setswapchan()` 216c, and `true`.

`setswapchan()` installs the swap file channel. If the device is not a mount (`L'M'`), it stats the file to check its size and may reduce `conf.nswap` if the file is smaller than the pre-allocated swap map. The swap file can only be replaced when all swap slots are free (no pages currently swapped out):

⟨*function* setswapchan 216c⟩≡ (866b)
```
void
setswapchan(Chan *c)
{
    byte dirbuf[sizeof(DirEntry)+100];
    DirEntry d;
    int n;

    if(swapimage.c) {
        if(swapalloc.free != conf.nswap){
            cclose(c);
            error(Einuse);
        }
        cclose(swapimage.c);
    }

    /*
     *  if this isn't a file, set the swap space
     *  to be at most the size of the partition
     */
```

```
    if(devtab[c->type]->dc != L'M'){
        n = devtab[c->type]->stat(c, dirbuf, sizeof dirbuf);
        if(n <= 0){
            cclose(c);
            error("stat failed in setswapchan");
        }
        convM2D(dirbuf, n, &d, nil);
        if(d.length < conf.nswap*BY2PG){
            conf.nswap = d.length/BY2PG;
            swapalloc.top = &swapalloc.swmap[conf.nswap];
            swapalloc.free = conf.nswap;
        }
    }

    swapimage.c = c;
}
```

Uses `cclose`, `conf` 62b, `swapalloc` 204b, and `swapimage` 205b.

Chapter 9

Scheduling

9.1 Scheduling principles

The scheduler is the piece of the kernel that decides, many times a second, which of the currently ready processes should run next on a given CPU. Its core loop is small: pick a winner from a *run queue* of ready processes, context-switch to it (Section 2.1.7), let it run until it blocks on I/O or is forcibly preempted, then repeat. Every user-visible performance property of a system—interactive responsiveness, background-job throughput, fairness between users, the ability of a real-time process to meet its deadline—emerges from the scheduler's choice of who runs next, which makes this the most heavily studied subsystem in any kernel.

The first question a scheduler asks is whether control returns to it voluntarily or must be taken back by force. *Cooperative* schedulers wait for the running process to call `yield()` or to block on I/O, and only then pick the next one; this was the model in classic Mac OS through version 9, in Windows 3.x, and— by deliberate choice—in modern Go's goroutine runtime. *Preemptive* schedulers use a hardware timer interrupt to forcibly stop the running process after a fixed *quantum*, save its state, and run the scheduler whether the process agreed or not. Preemptive scheduling has been the default in every serious kernel since UNIX V1 in 1971, because the alternative is that a single buggy or malicious process hangs the entire machine—a property that was a daily experience on Windows 3.x. Plan 9, like every modern kernel, is preemptive, but it also exposes the cooperative primitives (`sched()`, `sleep()`, `wakeup()`) because most of its own internal yielding is voluntary: I/O-bound processes almost always block *before* their quantum runs out.

The interesting variety in schedulers is not the "save and restore the context" mechanism—which we already covered in Section 2.1.7—but the *policy* that picks *which* ready process runs next. The simplest policy is round-robin over a single FIFO: good for fairness, terrible for interactive latency. Priority-based scheduling assigns each process a number and always picks the highest, and is the classic UNIX design; to avoid starving low-priority processes it requires *dynamic* priority adjustment. *Multilevel feedback queues*, first introduced in the CTSS system at MIT in 1962 and standard in historical UNIX, keep one queue per priority and move processes between queues based on recent CPU usage—an interactive process that blocks often floats up, a compute-bound one sinks. *Fair-share* schedulers such as Linux's CFS (2007–2023) track each process's accumulated "virtual runtime" and always pick the one that has had the least, using a red-black tree as their run queue instead of a priority array; Linux 6.6 (2023) replaced CFS with EEVDF, a refinement adding deadline-like virtual eligible time. Orthogonal to all of this, *real-time* schedulers provide hard priority classes (POSIX `SCHED_FIFO` and `SCHED_RR`, Windows' real-time class) where a high-priority process really does run to the exclusion of every lower one, for code that cannot tolerate any scheduling jitter.

Plan 9's scheduler is a classic multilevel-priority design, deliberately kept simple enough to be read end-to-end in a single chapter—no red-black trees, no virtual runtime, no control groups. There are twenty priority levels (the constant `Npriq`), with normal user processes starting at `PriNormal = 10` and kernel processes at `PriKproc = 13`; higher numbers win. The run queue is an array of twenty doubly-linked lists, one per level, and `sched()` simply scans them from the top down and takes the head of the first non-empty one:

```
runq[]:  array of Npriq (=20) doubly-linked lists

pri 19  ------> (empty)
pri 18  ------> (empty)
 ...
pri 13  ------> [ kproc A ] -> [ kproc B ]          <- PriKproc
 ...
pri 10  ------> [ proc X  ] -> [ proc Y ] -> [ Z ]  <- PriNormal
pri  9  ------> [ proc W  ]
 ...
pri  0  ------> (empty)

sched() scans top-to-bottom and picks the head
of the first non-empty queue.
```

Priorities are adjusted dynamically to keep things fair: a process that has just woken up from sleeping on I/O is boosted (reward for interactivity); a process that has consumed its full quantum is demoted (penalty for being compute-bound). Preemption is driven by `hzclock()`, introduced in Chapter 10, which decrements the running process's quantum and calls `sched()` when it reaches zero. The chapter that follows is split into those two halves: the cooperative path (`sched()`, `sleep()`, `wakeup()`) first, then the preemptive path (`hzclock()` and the timer-driven context switch).

9.2 Overview

The chapter's central function is `sched()` (with its twin `schedinit()`); everything else is plumbing around it. We begin with the priority mechanism: how each process gets a priority number, and how those numbers are adjusted at runtime.

9.3 Process priority

There are 20 priority levels (`Npriq`). Normal user processes start at priority 10 (`PriNormal`), kernel processes at 13 (`PriKproc`). Higher numbers mean higher priority. The priority can be adjusted at runtime via `/proc/\emphpid/ctl`:

⟨*constant* `Npriq` 219a⟩≡ (868c)
```
Npriq   = 20,   /* number of scheduler priority levels */
```

⟨*enum* `priority` 219b⟩≡ (868c)
```
enum Priority
{
    PriNormal = 10,   /* base priority for normal processes */
    PriKproc  = 13,   /* base priority for kernel processes */
    ⟨Priority cases 220g⟩
    ⟨Priority cases for real-time priority 564a⟩
};
```

⟨`Proc` *scheduling fields* 219c⟩≡ (66) 220b▷
```
// enum<Priority>
ulong priority; /* priority level */
```

⟨`newproc()` *set fields of* `p` 219d⟩+≡ (130a) ◁213c 224c▷
```
/* sched params */
procpriority(p, PriNormal, false);
```
Uses `PriNormal` 219b, `false`, and `procpriority()` 220a.

⟨*function* `procpriority` 220a⟩≡ (877c)
```
void
procpriority(Proc *p, int pri, bool fixed)
{
    ⟨procpriority() sanity check pri 220d⟩
    p->priority = pri;

    p->basepri = pri;
    p->fixedpri = fixed;
}
```

⟨`Proc` *scheduling fields* 220b⟩+≡ (66) ◁219c 224b▷
```
// enum<Priority>
ulong basepri;  /* base priority level */
bool fixedpri; /* priority level doesn't change */
```

⟨`sysrfork()` *when* `RFPROC` *propagate fields from up to p* 220c⟩+≡ (134b) ◁213d 226c▷
```
p->priority = up->basepri;
p->basepri = up->basepri;
p->fixedpri = up->fixedpri;
```
Uses `up`.

⟨`procpriority()` *sanity check pri* 220d⟩≡ (220a)
```
if(pri >= Npriq)
    pri = Npriq - 1;
else if(pri < 0)
    pri = 0;
```
Uses `Npriq` 219a.

⟨`procctlreq()` *CMpri case* 220e⟩≡ (620b)
```
case CMpri:
    pri = atoi(cb->f[1]);
    if(pri > PriNormal && !iseve())
        error(Eperm);
    procpriority(p, pri, false);
    break;
```
Uses `CMpri-887` 602b, `PriNormal` 219b, `false`, `iseve()` 591b, and `procpriority()` 220a.

⟨`procctlreq()` *CMfixedpri case* 220f⟩≡ (620b)
```
case CMfixedpri:
    pri = atoi(cb->f[1]);
    if(pri > PriNormal && !iseve())
        error(Eperm);
    procpriority(p, pri, true);
    break;
```
Uses `CMfixedpri-888` 602b, `PriNormal` 219b, `iseve()` 591b, `procpriority()` 220a, and `true`.

⟨`Priority` *cases* 220g⟩≡ (219b)
```
PriRoot   = 13,   /* base priority for root processes */
```

⟨`sysexec()` *set prioriry of root processes* 220h⟩≡ (141c)
```
/*
 *  '/' processes are higher priority (hack to make /ip more responsive).
 */
if(devtab[tc->type]->dc == L'/')
    up->basepri = PriRoot;
up->priority = up->basepri;
```

9.4 The runqueue

The run queue is an array of `Nrq` queues, one per priority level. Each queue is a FIFO linked list of `Proc` pointers. The bitmap `runvec` has bit i set if `runq[i]` is non-empty, allowing `runproc()` to quickly find the highest-priority non-empty queue:

```
runq[19]:  (highest priority)
runq[18]:
  ...
runq[13]:  kproc1 -> kproc2
  ...
runq[10]:  proc_A -> proc_B -> proc_C
  ...
runq[0]:   (lowest priority)

runvec = ... 0010_0000_0100_0000_0000  (bits 10, 13 set)
```

⟨*constant* `Nrq` 221a⟩≡ (868c)
```
Nrq   = Npriq+2,  /* number of priority levels including real time */
```
Uses `Npriq` 219a.

⟨*global* `runq` 221b⟩≡ (877c)
```
// The run queue!!
// hash<enum<priority>, queue<ref<Proc>>>
Schedq  runq[Nrq];
```
Uses `Nrq` 221a.

⟨*struct* `Schedq` 221c⟩≡ (868c)
```
// essentially a queue<ref<Proc>>
struct Schedq
{
    // list<ref<Proc>> (next = Proc.rnext)
    Proc* head;
    // ref<Proc>, the tail
    Proc* tail;
    // size of list
    int n;

    // extra
    Lock; // protects also the globals nrdy and runvec
};
```

⟨`Proc` *extra fields* 221d⟩+≡ (66) ◁128f 256g▷
```
// list<ref<Proc>> of Schedq.head
Proc  *rnext;   /* next process in run queue */
```

⟨*global* `runveq` 221e⟩≡ (877c)
```
// array<bool>, each bit i represents whether the runq at pri i has processes
ulong   runvec; // coupling: sizeof(ulong) must be >= Nrq
```

⟨*function* `anyready` 221f⟩≡ (877c)
```
bool
anyready(void)
{
    return runvec;
}
```
Uses `runvec` 221e.

⟨*global* `nrdy` 222a⟩≡ (877c)
```
int nrdy;
```

⟨*function* `queueproc` 222b⟩≡ (877c)
```
/*
 * add a process to a scheduling queue
 */
void
queueproc(Schedq *rq, Proc *p)
{
    int pri;

    pri = rq - runq;
    lock(runq);
    p->priority = pri; // because a priority can evolve

    // add_queue(p, rq)
    p->rnext = nil;
    if(rq->tail)
        rq->tail->rnext = p;
    else
        rq->head = p;
    rq->tail = p;
    rq->n++;

    nrdy++;
    runvec |= 1<<pri;
    unlock(runq);
}
```
Uses `lock()` 76d, `nrdy` 222a, `runq` 221b, `runvec` 221e, and `unlock()` 78d.

⟨*function* `dequeueproc` 222c⟩≡ (877c)
```
/*
 *  try to remove a process from a scheduling queue (called arch_splhi)
 */
Proc*
dequeueproc(Schedq *rq, Proc *tp)
{
    Proc *l, *p;

    if(!canlock(runq))
        return nil;

    /*
     *  the queue may have changed before we locked runq,
     *  refind the target process.
     */
    l = nil;
    for(p = rq->head; p; p = p->rnext){
        if(p == tp)
            break;
        l = p;
    }
    ⟨dequeueproc() return if process not found or already associated 223a⟩

    // remove_queue(p, rq)
    if(p->rnext == nil)
        rq->tail = l;
    if(l)
        l->rnext = p->rnext;
```

```
    else
        rq->head = p->rnext;

    if(rq->head == nil)
        runvec &= ~(1<<(rq-runq));
    rq->n--;
    nrdy--;
    ⟨dequeueproc() sanity check p 223b⟩
    unlock(runq);
    return p;
}
```

Uses canlock() 78f, nrdy 222a, runq 221b, runvec 221e, and unlock() 78d.

⟨dequeueproc() *return if process not found or already associated* 223a⟩≡ (222c)

```
/*
 *  p->cpu==nil only when process state is saved
 */
if(p == nil || p->cpu){
    unlock(runq);
    return nil;
}
```

Uses runq 221b and unlock() 78d.

⟨dequeueproc() *sanity check p* 223b⟩≡ (222c)

```
if(p->state != Ready)
    print("dequeueproc %s %lud %s\n", p->text, p->pid, statename[p->state]);
```

Uses Ready 231c, print, and statename 67d.

9.5 The scheduler

9.5.1 sched()

sched() is the central function. When called with up != nil (a running process wants to yield), it saves the current process's context and jumps to schedinit(). When called with up == nil (from schedinit(), on the CPU's boot stack), it picks the next process via runproc() and jumps to it.

⟨*function* sched 223c⟩≡ (877c)

```
/*
 *  If changing this routine, look also at sleep().  It
 *  contains a copy of the guts of sched().
 */
void
proc_sched(void)
{
    ⟨sched() locals 223d⟩

    ⟨sched() sanity check ilockdepth 224d⟩
    ⟨sched() if up 227c⟩
    // else
    ⟨sched() if no up 224a⟩
}
```

⟨sched() *locals* 223d⟩≡ (223c)

```
Proc *p;
```

⟨sched() *if no up* 224a⟩≡ (223c)
```
// We should execute this code using the main kernel stack, as
// we should arrive here from schedinit()

p = runproc();
⟨sched() after found process p to schedule, possibly update priority 238c⟩
⟨sched() if readied process, do not adjust schedticks 236e⟩
else {
    ⟨sched() after found a new proc, adjust schedticks 236b⟩
}
⟨sched() reset readied 232e⟩

cpu->proc = p;
up = p; // same instruction than previous line on some archi (e.g. x86)

up->state = Running;
up->cpu = CPUS(cpu->cpuno);

arch_mmuswitch(up);
arch_gotolabel(&up->sched);
```
Uses CPUS 702, Running 67c, runproc() 225, and up.

⟨Proc *scheduling fields* 224b⟩+≡ (66) ◁220b 226a▷
```
// option<ref<Cpu>>, null when not associated to a processor
Cpu  *cpu;    /* processor running this proc */
```

⟨newproc() *set fields of* p 224c⟩+≡ (130a) ◁219d 226b▷
```
p->cpu = nil;
```

⟨sched() *sanity check ilockdepth* 224d⟩≡ (223c)
```
if(cpu->ilockdepth)
    panic("cpu%d: ilockdepth %d, last lock %#p at %#p, sched called from %#p",
        cpu->cpuno, cpu->ilockdepth, up? up->lastilock: nil,
        (up && up->lastilock)? up->lastilock->pc: 0, getcallerpc(&p+2));
```

9.5.2 schedinit()

schedinit() runs on the CPU's boot stack (set up during initialization). It calls arch_setlabel() to save the CPU's "return here" point in cpu->sched, then handles the outgoing process (putting it back on the run queue if Running, freeing it if Moribund), and falls through to sched() with up == nil. When sched() later finds a new process, it calls arch_gotolabel(\&cpu->sched) to return here.

⟨Cpu *other fields* 224e⟩+≡ (64a) ◁89c 232c▷
```
Label sched;      /* scheduler wakeup */ // address of schedinit()
```

⟨*function* schedinit 224f⟩≡ (877c)
```
/*
 * Always arch_splhi()'ed.
 */
void
schedinit(void)     /* never returns */
{
    ⟨schedinit() locals 564f⟩

    arch_setlabel(&cpu->sched);
    ⟨schedinit() after setlabel, if up 228a⟩
    // ok at this point up is nil
    sched();
}
```
Uses sched.

9.5.3 runproc()

runproc() finds the next process to run. It scans the run queue from highest to lowest priority, looking for a process with CPU affinity to this processor (or any process on a second pass). If no process is ready, it calls arch_idlehands() to halt the CPU until an interrupt arrives.

⟨*function* runproc 225⟩≡ (877c)

```
/*
 *  pick a process to run
 */
Proc*
runproc(void)
{
    Proc *p;
    Schedq *rq;
    int i;
    ⟨runproc() other locals 499f⟩

    ⟨runproc() start 656d⟩
    ⟨runproc() cooperative scheduling if process readied 232d⟩
    //else

    preempts++;

loop:
    /*
     *  find a process that last ran on this processor (affinity),
     *  or one that hasn't moved in a while (load balancing). Every
     *  time around the loop affinity goes down.
     */
    arch_spllo(); // from schedinit()

    for(i = 0;; i++){
        /*
         *  find the highest priority target process that this
         *  processor can run given affinity constraints.
         *
         */
        for(rq = &runq[Nrq-1]; rq >= runq; rq--){
            for(p = rq->head; p; p = p->rnext){
                if(p->lastcpu == nil || p->lastcpu == CPUS(cpu->cpuno)
                   || (p->wired == nil && i > 0)) // favor affinity for first round
                    goto found;
            }
        }
        // nothing found
        /* waste time or halt the CPU */
        arch_idlehands();
        ⟨runproc() after arch_idlehands, before restarting loop 656e⟩
    }

found:
    arch_splhi();
    p = dequeueproc(rq, p);
    if(p == nil) // the runq may have changed between spllo and splhi
        goto loop;

    p->state = Scheding;
    p->lastcpu = CPUS(cpu->cpuno);
```

```
    ⟨runproc() test if p is a real-time process 564c⟩
    ⟨runproc() hook proctrace 499g⟩
    return p;
}
```

Uses CPUS 702, Nrq 221a, Scheding 129c, arch_idlehands(), dequeueproc() 222c, preempts 662b, and runq 221b.

```
⟨Proc scheduling fields 226a⟩+≡                                (66) ◁224b 229a▷
  Cpu *lastcpu;    /* processor this process last ran on */
```

```
⟨newproc() set fields of p 226b⟩+≡                             (130a) ◁224c 226f▷
  p->lastcpu = nil;
```

```
⟨sysrfork() when RFPROC propagate fields from up to p 226c⟩+≡      (134b) ◁220c 449g▷
  p->lastcpu = up->lastcpu;
```

Uses up.

9.6 Context switch

A context switch saves one process's execution state and restores another's. Unlike the full register save done by _vsvc on a syscall (saving all 15 registers into a Ureg), a kernel-to-kernel context switch only needs to save SP and PC—because kernel code follows the calling convention, and the compiler has already saved any registers it needs.

9.6.1 Label

```
⟨struct Label 226d⟩≡                                                  (702)
  // =~ a jumpbuf in C, for coroutines
  struct Label
  {
      // or virt_addr? used also for saving context of user code?
      kern_addr sp;
      kern_addr pc;
  };
```

```
⟨Proc assembly fields 226e⟩≡                                          (66)
  Label sched;    /* known to l.s */
  char  *kstack;  /* known to l.s */
```

9.6.2 Kernel stacks

Each process has its own 8 KiB kernel stack (kstack), allocated once when the Proc is first used and reused across process lifetimes. This per-process stack is where syscall handlers, page fault handlers, and sched() run. It is separate from the CPU's boot stack (inside the Cpu structure), which is used only by schedinit().

```
⟨newproc() set fields of p 226f⟩+≡                             (130a) ◁226b 229b▷
  if(p->kstack == nil)
      p->kstack = smalloc(KSTACK);
```

Uses smalloc() 104b.

```
⟨constant KSTACK(arm) 226g⟩≡                                          (1005)
  #define KSTACK      (8*KiB)
```

9.6.3 `arch_gotolabel()` (ARM)

`arch_setlabel()` saves `SP` and `PC` into a `Label` and returns 0. `arch_gotolabel()` restores them and returns 1. This is the kernel's equivalent of C's `setjmp/longjmp`—it is how the scheduler switches between processes. The different return values let the caller distinguish "just saved" from "resumed after being switched out":

⟨*function* `arch_gotolabel` *(arm)* 227a⟩≡ (1010b)
```
TEXT arch_gotolabel(SB), 1, $-4
    MOVW    0(R0), R13      /* sp */
    MOVW    4(R0), R14      /* pc */
    MOVW    $1, R0
    RET
```

⟨*function* `arch_setlabel` *(arm)* 227b⟩≡ (1010b)
```
TEXT arch_setlabel(SB), 1, $-4
    MOVW    R13, 0(R0)      /* sp */
    MOVW    R14, 4(R0)      /* pc */
    MOVW    $0, R0
    RET
```

9.6.4 `sched()` and `schedinit()` dance

The interplay between `sched()` and `schedinit()` is the trickiest part of the scheduler. Here is the flow when a running process calls `sched()`:

```
  Process A's kstack:          CPU's boot stack:

  1. sched() called
     arch_setlabel(&up->sched)
     returns 0 (first time)
         |
         v
  2. arch_gotolabel(&cpu->sched)
     ----jumps to-------->  3. schedinit() resumes
                               up->state == Running
                               -> ready(up)
                               up = nil
                               -> sched()
                                    |
                                    v
                            4. p = runproc()
                               (picks process B)
                               arch_gotolabel(&p->sched)
                               ----jumps to-------->
                                                    Process B resumes
                                                    from its setlabel
```

When process A is eventually picked again by `runproc()`, `arch_gotolabel(\&A->sched)` returns 1 (into step 1 above), and `sched()` calls `arch_procrestore()` and returns—process A continues where it left off.

⟨`sched()` *if up* 227c⟩≡ (223c)
```
if(up){
    ⟨sched() when up, possibly increment delaysched and return 228c⟩
    arch_splhi(); // schedinit requires this (also sensitive moment)
    cpu->cs++;
```

```
        arch_procsave(up);
        if(arch_setlabel(&up->sched)){
            //
            // here when the process has been scheduled back
            // from a arch_gotolabel(up->sched) by another process, see below
            //
            arch_procrestore(up);
            arch_spllo();
            return;
        }else{
            //
            // here to go to schedinit() (which will call sched() back)
            //
            arch_gotolabel(&cpu->sched); // goto schedinit()
            panic("sched: should never reach this point");
        }
    }
```

Uses up.

⟨schedinit() *after setlabel, if up* 228a⟩≡ (224f)

```
    if(up) {
        ⟨schedinit() optional real-time edfrecord() 564g⟩
        //old: cpu->proc = nil;
        // but now that on x86 up = cpu->proc and not cpu->externup
        // we can't do that anymore. Is there a place that rely on
        // cpu->proc and cpu->externup to not be in sync?
        switch(up->state) {
        ⟨schedinit() switch process state cases 154a⟩
        }
        up->cpu = nil;
        ⟨schedinit() updatecpu before reseting up 239b⟩
        cpu->proc = nil;
        up = nil; // same instruction than previous line on some archi (e.g. PC)
    }
```

Uses up.

⟨schedinit() *switch process state cases* 228b⟩+≡ (228a) ◁154a

```
    case Running:
        ready(up);
        break;
```

Uses Running 67c, ready, and up.

9.6.5 Delayed sched()

A process holding spin locks must not be switched out—the lock would remain held, potentially deadlocking other CPUs. So when sched() is called while up->nlocks > 0, it increments delaysched and returns immediately. The actual reschedule happens later, when unlock() sees delaysched > 0 and calls sched() after releasing the last lock. As a safety valve, if delaysched exceeds 20 (a lock was probably leaked), the scheduler forces a switch anyway.

⟨sched() *when up, possibly increment delaysched and return* 228c⟩≡ (227c)

```
    /*
     * Delay the sched until the process gives up the locks
     * it is holding.  This avoids dumb lock loops.
     *
     * But don't delay if the process is Moribund.
     * It called sched to die.
```

```
   * But do sched eventually. This avoids a missing unlock
   * from hanging the entire kernel.
   * But don't reschedule procs holding palloc or procalloc.
   * Those are far too important to be holding while asleep.
   *
   * This test is not exact.  There can still be a few instructions
   * in the middle of taslock when a process holds a lock
   * but Lock.p has not yet been initialized.
   */
  if(up->nlocks.ref)
    if(up->state != Moribund)
      if(up->delaysched < 20
        || palloc.Lock.p == up
        || procalloc.Lock.p == up){

          up->delaysched++;
          delayedscheds++;
          return;
      }
  // else
  up->delaysched = 0;
```

Uses Moribund 153a, delayedscheds 662b, palloc 109b, procalloc-2458 128d, and up.

⟨Proc *scheduling fields* 229a⟩+≡ (66) ◁226a 237d▷

```
  ulong delaysched;
```

⟨newproc() *set fields of* p 229b⟩+≡ (130a) ◁226f 237e▷

```
  p->nlocks.ref = 0;
  p->delaysched = 0;
```

⟨unlock() *if delaysched* 229c⟩≡ (78d)

```
  if(up && deccnt(&up->nlocks) == 0 && up->delaysched && arch_islo()){
      /*
       * Call sched if the need arose while locks were held
       * But, don't do it from interrupt routines, hence the arch_islo() test
       */
      sched();
  }
```

Uses deccnt() 89a, sched, and up.

⟨trap() *if delaysched and clockintr* 229d⟩≡ (168b)

```
  /* delaysched set because we held a lock or because our quantum ended */
  if(up && up->delaysched && clockintr){
      sched();
      arch_splhi();
  }
```

Uses sched and up.

⟨syscall() *if delaysched(arm)* 229e⟩≡ (163e)

```
  /* if we delayed sched because we held a lock, sched now */
  if(up->delaysched){
      sched();
      arch_splhi();
  }
```

Uses sched and up.

9.6.6 arch_mmuswitch() (ARM)

On each context switch, `arch_mmuswitch()` installs the new process's hardware page tables. It clears the user portion of the L1, repopulates it from the process's `mmul2` list, and invalidates the TLB. If the pager modified the process's pages while it was asleep (`newtlb` flag), the L2 pages are also cleared.

⟨Cpu Arch *mmu fields(arm)* 230a⟩+≡ (64e) ◁188a
```
int mmupid;
```

⟨*function* arch_mmuswitch*(arm)* 230b⟩≡ (1006b)
```
void
arch_mmuswitch(Proc* proc)
{
    int x;
    PTE *l1;
    Page *page;

    /* do kprocs get here and if so, do they need to? */
/*** "This is plausible, but wrong" - Charles Forsyth 1 Mar 2015
    if(cpu->mmupid == proc->pid && !proc->newtlb)
        return;
***/
    cpu->mmupid = proc->pid;

    ⟨arch_mmuswitch() write back and invalidate cache before switch(arm) 531a⟩

    if(proc->newtlb){
        mmul2empty(proc, 1);
        proc->newtlb = false;
    }

    mmul1empty();

    /* move in new map */
    l1 = cpu->mmul1;
    for(page = proc->mmul2; page != nil; page = page->next){
        x = page->daddr;
        l1[x] = PPN(page->pa)|Dom0|Coarse;
        /* know here that L1lo < x < L1hi */
        if(x+1 - cpu->mmul1lo < cpu->mmul1hi - x)
            cpu->mmul1lo = x+1;
        else
            cpu->mmul1hi = x;
    }

    ⟨arch_mmuswitch() write back cache after adjusting page tables(arm) 531c⟩

    /* lose any possible stale tlb entries */
    mmuinvalidate();
}
```
Uses `Coarse` 178d, `Dom0` 179d, `false`, `mmul1empty()` 181e, and `mmul2empty()` 181a.

9.6.7 arch_procsave() (ARM)

`arch_procsave()` saves architecture-specific state before a context switch (floating-point registers, cycle counters). `arch_procrestore()` restores them when the process resumes:

⟨*function* arch_procsave*(arm)* 230c⟩≡ (1008c)
```
/*
```

```
 *  Save the mach dependent part of the process state.
 */
void
arch_procsave(Proc* p)
{
    ⟨arch_procsave() cycles adjustments 509f⟩
    ⟨arch_procsave() floating point(arm) 515e⟩
}
```

⟨*function* `arch_procrestore` *(arm)* 231a⟩≡ (1008c)

```
void
arch_procrestore(Proc* p)
{
    uvlong t;

    if(p->kp)
        return;
    ⟨arch_procrestore cycles adjustments 509g⟩
    ⟨arch_procrestore() floating point(arm) 516b⟩
}
```

9.7 Cooperative scheduling

In cooperative scheduling, a process voluntarily gives up the CPU by calling `sched()`, `sleep()`, or `yield()`. This is the most common path: syscalls that block (waiting for I/O, a lock, or a child process) call `sleep()` on a `Rendez`, which calls `sched()` internally.

9.7.1 `yield()`

⟨*function* `yield` 231b⟩≡ (877c)

```
/*
 *  yield the processor and drop our priority
 */
void
yield(void)
{
    if(anyready()){
        ⟨yield() adjust lastupdate before scheduling another process 239g⟩
        sched();
    }
}
```

Uses `anyready()` 221f and `sched`.

9.7.2 `ready()`

`ready()` puts a process on the run queue at its current priority. It is called from many places: `schedinit()` (when the current process is preempted), `wakeup()` (when a sleeping process's condition becomes true), `qunlock()` (when a waiting process gets the lock), `sysrfork()` (for the child), and `kproc()` (for new kernel processes).

⟨`Procstate` *cases* 231c⟩+≡ (67c) ◁153a 233d▷

```
Ready,
```

⟨*function* `ready` 232a⟩≡ (877c)

```
/*
 *  ready(p) picks a new priority for a process and sticks it in the
 *  runq for that priority.
 */
void
proc_ready(Proc *p)
{
    int pri;
    int s; // spl
    Schedq *rq;
    ⟨ready() other locals 499d⟩

    s = arch_splhi();
    ⟨ready() optional edfready() for real-time scheduling 565a⟩

    ⟨ready() possibly set cpu->readied 232b⟩
    ⟨ready() update priority 239a⟩

    rq = &runq[pri];
    p->state = Ready;
    queueproc(rq, p);
    ⟨ready() hook proctrace 499e⟩
    arch_splx(s);
}
```

Uses `Ready` 231c, `queueproc()` 222b, and `runq` 221b.

⟨`ready()` *possibly set* `cpu->readied` 232b⟩≡ (232a)

```
if(up != p && (p->wired == nil || p->wired == CPUS(cpu->cpuno))) // pad fix
    cpu->readied = p; /* group scheduling */
```

Uses `CPUS` 702 and `up`.

⟨`Cpu` *other fields* 232c⟩+≡ (64a) ◁224e 236a▷

```
Proc* readied;    /* for runproc */
```

⟨`runproc()` *cooperative scheduling if process readied* 232d⟩≡ (225)

```
/* cooperative scheduling until the clock ticks */
p=cpu->readied;
if(p && p->cpu==nil && p->state==Ready &&
  (p->wired == nil || p->wired == CPUS(cpu->cpuno)) && // pad's bugfix!
  ⟨runproc() test for empty real-time scheduling queue 564b⟩
){
    skipscheds++;
    rq = &runq[p->priority];
    goto found;
}
```

Uses `CPUS` 702, `Ready` 231c, `runq` 221b, and `skipscheds` 662b.

⟨`sched()` *reset readied* 232e⟩≡ (224a)

```
cpu->readied = nil;
```

9.7.3 Rendez

A `Rendez` (rendezvous point) is a condition variable for kernel code. It pairs a sleeping process (`r->p`) with a lock. `sleep()` puts the current process to sleep on a `Rendez` if a condition is false; `wakeup()` wakes it when the

condition becomes true. This is how all blocking in the kernel works: waiting for a page, a lock, a child exit, a pipe read, etc.

⟨*struct* Rendez 233a⟩≡ (692)
```
struct Rendez
{
    // option<ref<Proc>>
    Proc  *p; // sleeping process

    Lock; // LOCK ORDERING: always do lock(r); lock(p->rlock);
};
```

⟨Proc *synchronization fields* 233b⟩+≡ (66) ◁88f 233c▷
```
// option<ref<Rendez>>, can point to waitr, freememr, sleepr, etc
Rendez  *r;   /* rendezvous point slept on */
```

⟨Proc *synchronization fields* 233c⟩+≡ (66) ◁233b 254f▷
```
Lock  rlock;    /* sync sleep/wakeup with postnote */
```

9.7.4 sleep()

`sleep()` checks the condition function `f(arg)`. If true (or a note is pending), it returns immediately. Otherwise, it sets the process state to `Wakeme`, records the `Rendez`, and performs a context switch (the same `arch_setlabel`/`arch_gotolabel` dance as `sched()`). The locks on both the `Rendez` and the process's `rlock` are held while committing to sleep, then released just before jumping to `schedinit()`.

⟨Procstate *cases* 233d⟩+≡ (67c) ◁231c 480a▷
```
Wakeme,
```

⟨*function* sleep 233e⟩≡ (877c)
```
/*
 *  sleep if a condition is not true.  Another process will
 *  awaken us after it sets the condition.  When we awaken
 *  the condition may no longer be true.
 *
 *  we lock both the process and the rendezvous to keep r->p
 *  and p->r synchronized.
 */
void
proc_sleep(Rendez *r, bool (*f)(void*), void *arg)
{
    int s; // spl
    ⟨sleep() other locals 499j⟩

    s = arch_splhi();
    ⟨sleep() sanity check nlocks 234a⟩
    lock(r);
    lock(&up->rlock);
    ⟨sleep() sanity check double sleep on this rendez vous 234b⟩

    /*
     *  Wakeup only knows there may be something to do by testing
     *  r->p in order to get something to lock on.
     *  Flush that information out to memory in case the sleep is
     *  committed.
     */
    r->p = up;

    if((*f)(arg) || up->notepending){
```

```
        /*
         *  if condition happened or a note is pending
         *  never mind
         */
        r->p = nil;
        unlock(&up->rlock);
        unlock(r);
    } else {
        /*
         *  now we are committed to
         *  change state and call scheduler
         */
        ⟨sleep() hook proctrace 499k⟩
        up->state = Wakeme;
        up->r = r;

        // similar code to sched(), why not call sched()? because of the unlock?

        /* statistics */
        cpu->cs++;

        arch_procsave(up);
        if(arch_setlabel(&up->sched)) {
            /*
             *  here when the process is awakened
             */
            arch_procrestore(up);
            arch_spllo();
        } else {
            /*
             *  here to go to sleep (i.e. stop Running)
             */
            unlock(&up->rlock);
            unlock(r);

            arch_gotolabel(&cpu->sched);
            panic("sleep: should never reach this point");
        }
    }
    ⟨sleep() if notepending 450a⟩
    arch_splx(s);
}
```

Uses Wakeme 233d, lock() 76d, unlock() 78d, and up.

⟨sleep() *sanity check nlocks* 234a⟩≡ (233e)

```
if(up->nlocks.ref)
    print("process %lud sleeps with %lud locks held, last lock %#p locked at pc %#lux, sleep called from %#p\n"
        up->pid, up->nlocks.ref, up->lastlock, up->lastlock->pc, getcallerpc(&r));
```

Uses print and up.

⟨sleep() *sanity check double sleep on this rendez vous* 234b⟩≡ (233e)

```
if(r->p){
    print("double sleep called from %#p, %lud %lud\n", getcallerpc(&r), r->p->pid, up->pid);
    arch_dumpstack();
}
```

Uses arch_dumpstack, print, and up.

9.7.5 wakeup()

`wakeup()` checks if anyone is sleeping on the `Rendez`. If so, it clears the `Rendez` link, calls `ready()` to put the process back on the run queue, and returns the process. The woken process will resume inside `sleep()` at the `arch_setlabel` return point:

⟨*function* `wakeup` 235a⟩≡ (877c)
```
/*
 *  Expects that only one process can call wakeup for any given Rendez.
 *  We hold both locks to ensure that r->p and p->r remain consistent.
 *  Richard Miller has a better solution that doesn't require both to
 *  be held simultaneously, but I'm a paranoid - presotto.
 */
Proc*
proc_wakeup(Rendez *r)
{
    Proc *p;
    int s; // spl

    s = arch_splhi();
    lock(r);
    p = r->p;

    if(p != nil){
        lock(&p->rlock);
        ⟨wakeup() sanity check process state 235b⟩
        r->p = nil;
        p->r = nil;
        ready(p);
        unlock(&p->rlock);
    }
    unlock(r);
    arch_splx(s);
    return p;
}
```
Uses `lock()` 76d, `ready`, and `unlock()` 78d.

⟨`wakeup()` *sanity check process state* 235b⟩≡ (235a)
```
if(p->state != Wakeme || p->r != r){
    iprint("%p %p %d\n", p->r, r, p->state);
    panic("wakeup: state");
}
```
Uses `Wakeme` 233d and `iprint`.

9.8 Preemptive scheduling

Cooperative scheduling is not enough—a CPU-bound process that never calls `sleep()` would monopolize the processor. Preemptive scheduling uses the clock interrupt (firing `Arch_HZ` times per second, i.e., every 10 ms) to force context switches. Each process gets a time quantum of 100 ms (`Arch_HZ/10` ticks); when the quantum expires, `hzsched()` sets `delaysched` to trigger a reschedule at the next safe point.

⟨*constant* `Arch_HZ`*(arm)* 235c⟩≡ (1010c)
```
#define Arch_HZ     100         /* clock frequency */
```

9.8.1 hzsched()

`hzsched()` is called from `hzclock()` on every timer tick while a process is running. It checks two conditions for preemption: is there a higher-priority process ready (`anyhigher()`)? or has the current process's quantum expired (`cpu->ticks > cpu->schedticks`)? If either is true, it sets `delaysched` rather than calling `sched()` directly—because we are inside an interrupt handler and cannot safely switch context here. The actual switch happens when the interrupt returns and checks `delaysched`.

⟨Cpu *other fields* 236a⟩+≡ (64a) ◁232c 472b▷
```
ulong schedticks;   /* next forced context switch */
```

⟨sched() *after found a new proc, adjust schedticks* 236b⟩≡ (224a)
```
cpu->schedticks = cpu->ticks + Arch_HZ/10; // 100ms of allocated time
```

⟨*function* hzsched 236c⟩≡ (877c)
```
/*
 *  here once per clock tick to see if we should resched
 */
void
hzsched(void)
{
    ⟨hzsched() rebalance ready procs 241a⟩
    /* unless preempted, get to run for at least 100ms */
    if(anyhigher()
    || (!up->fixedpri && cpu->ticks > cpu->schedticks && anyready())){

        cpu->readied = nil;   /* avoid cooperative scheduling */
        up->delaysched++;
    }
}
```
Uses anyhigher() 236d, anyready() 221f, and up.

⟨*function* anyhigher 236d⟩≡ (877c)
```
int
anyhigher(void)
{
    return runvec & ~((1<<(up->priority+1))-1);
}
```
Uses runvec 221e and up.

⟨sched() *if readied process, do not adjust schedticks* 236e⟩≡ (224a)
```
if(p == cpu->readied) {
}
```

9.8.2 hzclock()

`hzclock()` is the clock interrupt handler, called 100 times per second. It increments `cpu->ticks`, accounts CPU time to the current process, checks alarms, and calls `hzsched()` to decide whether to preempt.

⟨*clock callback hzclock* 236f⟩≡ (874c)
```
void
hzclock(Ureg *ur)
{
    cpu->ticks++;
    ⟨hzclock() adjust pc of proc 237a⟩
    ⟨hzclock() if flushmmu 472a⟩
    ⟨hzclock() accounttime 657a⟩
    ⟨hzclock() if kproftimer 660a⟩
```

```
    ⟨hzclock() if not active cpu, return 237b⟩
    // else
    ⟨hzclock() if exiting, exit 237c⟩
    // else

    ⟨hzclock() check the alarms 259a⟩
    if(up && up->state == Running)
        hzsched();  /* in proc.c */
}
```

Uses Running 67c, hzsched() 236c, and up.

⟨hzclock() *adjust pc of proc* 237a⟩≡ (236f)
```
if(cpu->proc) // why not using up here? why cpu->proc?
    cpu->proc->pc = ur->pc;
```

⟨hzclock() *if not active cpu, return* 237b⟩≡ (236f)
```
if((active.cpus & (1<<cpu->cpuno)) == 0)
    return;
```

Uses active 65e.

⟨hzclock() *if exiting, exit* 237c⟩≡ (236f)
```
if(active.exiting) {
    print("someone's exiting\n");
    arch_exit(0);
}
```

Uses active 65e, arch_exit, and print.

9.9 Customized scheduling

A process can be wired to a specific CPU via `echo wired \emphn > /proc/\emphpid/ctl`. Once wired, `runproc()` will only run it on that CPU. This is used for I/O-intensive processes that benefit from cache locality on a particular processor:

⟨Proc *scheduling fields* 237d⟩+≡ (66) ◁229a 239c▷
```
Cpu  *wired;
```

⟨newproc() *set fields of* p 237e⟩+≡ (130a) ◁229b 239d▷
```
p->wired = nil;
```

⟨procctlreq() *CMwired case* 237f⟩≡ (620b)
```
case CMwired:
    procwired(p, atoi(cb->f[1]));
    break;
```

Uses CMwired-889 602b and procwired() 237g.

⟨*function* procwired 237g⟩≡ (877c)
```
/*
 * wire this proc to a processor
 */
void
procwired(Proc *p, int bm)
{
    Proc *pp;
    int i;
    char nwired[MAXCPUS];
    Cpu *wm;
```

```
    if(bm < 0){
        /* pick a processor to wire to */
        memset(nwired, 0, sizeof(nwired));
        p->wired = nil;
        pp = proctab(0);
        for(i=0; i<conf.nproc; i++, pp++){
            wm = pp->wired;
            if(wm && pp->pid)
                nwired[wm->cpuno]++;
        }
        bm = 0;
        for(i=0; i<conf.ncpu; i++)
            if(nwired[i] < nwired[bm])
                bm = i;
    } else {
        /* use the virtual processor requested */
        bm = bm % conf.ncpu;
    }

    p->wired = CPUS(bm);
    p->lastcpu = p->wired;
}
```

Uses CPUS 702, conf 62b, and proctab.

⟨sysrfork() *other locals* 238a⟩+≡ (133c) ◁139a

```
Cpu *wm;
```

⟨sysrfork() *if parent is a wired proc* 238b⟩≡ (134b)

```
wm = up->wired;
if(wm)
    procwired(p, wm->cpuno);
```

Uses procwired() 237g and up.

9.10 Real-time scheduling

9.11 Adjusting priority

A fixed-priority scheduler would starve low-priority processes. To prevent this, the kernel dynamically adjusts each process's priority based on how much CPU time it has used recently. `updatecpu()` decays the process's CPU usage counter, and `reprioritize()` maps that usage to a priority level. Processes that have used a lot of CPU time recently get a lower priority; those that have been sleeping get a higher one. This ensures that I/O-bound processes (which sleep often) remain responsive, while CPU-bound processes gradually drop in priority.

9.11.1 reprioritize()

⟨sched() *after found process* p *to schedule, possibly update priority* 238c⟩≡ (224a)

```
⟨sched() if real-time process then do nothing, do not reprioritize 566a⟩
else {
    updatecpu(p);
    p->priority = reprioritize(p);
}
```

Uses reprioritize() 240 and updatecpu() 239h.

⟨ready() *update priority* 239a⟩≡ (232a)
```
updatecpu(p);
pri = reprioritize(p);
p->priority = pri;
```
Uses reprioritize() 240 and updatecpu() 239h.

⟨schedinit() *updatecpu before reseting up* 239b⟩≡ (228a)
```
updatecpu(up);
```
Uses up and updatecpu() 239h.

⟨Proc *scheduling fields* 239c⟩+≡ (66) ◁237d 566c▷
```
ulong lastupdate; // dimension?? ticks * Scaling;
ulong cpuavg;    /* cpu average */
```

⟨newproc() *set fields of* p 239d⟩+≡ (130a) ◁237e 449e▷
```
p->cpuavg = 0;
p->lastupdate = CPUS(0)->ticks * Scaling;
```
Uses CPUS 702 and Scaling-2461 239e.

⟨*constant* Scaling 239e⟩≡ (877c)
```
Scaling=2,
```

⟨*constant* Schedagain 239f⟩≡ (877c)
```
schedgain = 30, /* units in seconds */
```

⟨yield() *adjust lastupdate before scheduling another process* 239g⟩≡ (231b)
```
/* pretend we just used 1/2 tick */
up->lastupdate -= Scaling/2;
```
Uses Scaling-2461 239e and up.

⟨*function* updatecpu 239h⟩≡ (877c)
```
/*
 * Update the cpu time average for this particular process,
 * which is about to change from up -> not up or vice versa.
 * p->lastupdate is the last time an updatecpu happened.
 *
 * The cpu time average is a decaying average that lasts
 * about D clock ticks.  D is chosen to be approximately
 * the cpu time of a cpu-intensive "quick job".  A job has to run
 * for approximately D clock ticks before we home in on its
 * actual cpu usage.  Thus if you manage to get in and get out
 * quickly, you won't be penalized during your burst.  Once you
 * start using your share of the cpu for more than about D
 * clock ticks though, your p->cpu hits 1000 (1.0) and you end up
 * below all the other quick jobs.  Interactive tasks, because
 * they basically always use less than their fair share of cpu,
 * will be rewarded.
 *
 * If the process has not been running, then we want to
 * apply the filter
 *
 *  cpu = cpu * (D-1)/D
 *
 * n times, yielding
 *
 *  cpu = cpu * ((D-1)/D)^n
 *
 * but D is big enough that this is approximately
 *
```

```
 *  cpu = cpu * (D-n)/D
 *
 * so we use that instead.
 *
 * If the process has been running, we apply the filter to
 * 1 - cpu, yielding a similar equation.  Note that cpu is
 * stored in fixed point (* 1000).
 *
 * Updatecpu must be called before changing up, in order
 * to maintain accurate cpu usage statistics.  It can be called
 * at any time to bring the stats for a given proc up-to-date.
 */
void
updatecpu(Proc *p)
{
    int n, t, ocpu;
    int D = schedgain * Arch_HZ * Scaling;

    ⟨updatecpu() return if real-time process 564e⟩

    t = CPUS(0)->ticks*Scaling + Scaling/2;
    n = t - p->lastupdate;
    p->lastupdate = t;

    if(n == 0)
        return;
    if(n > D)
        n = D;

    ocpu = p->cpuavg;
    if(p != up)
        p->cpuavg = (ocpu*(D-n))/D;
    else{
        t = 1000 - ocpu;
        t = (t*(D-n))/D;
        p->cpuavg = 1000 - t;
    }

}
```

Uses CPUS 702, Scaling-2461 239e, schedgain-2460 239f, and up.

⟨*function* reprioritize 240⟩≡ (877c)

```
/*
 * On average, p has used p->cpuavg of a cpu recently.
 * Its fair share is conf.ncpu/cpu->load of a cpu.  If it has been getting
 * too much, penalize it.  If it has been getting not enough, reward it.
 * I don't think you can get much more than your fair share that
 * often, so most of the queues are for using less.  Having a priority
 * of 3 means you're just right.  Having a higher priority (up to p->basepri)
 * means you're not using as much as you could.
 */
int
reprioritize(Proc *p)
{
    int fairshare, n, load, ratio;

    load = CPUS(0)->load;
    if(load == 0)
        return p->basepri;
```

```
    /*
     *  fairshare = 1.000 * conf.nproc * 1.000/load,
     * except the decimal point is moved three places
     * on both load and fairshare.
     */
    fairshare = (conf.ncpu*1000*1000)/load;
    n = p->cpuavg;
    if(n == 0)
        n = 1;
    ratio = (fairshare+n/2) / n;
    if(ratio > p->basepri)
        ratio = p->basepri;
    if(ratio < 0)
        panic("reprioritize");
//iprint("pid %d cpu %d load %d fair %d pri %d\n", p->pid, p->cpuavg, load, fairshare, ratio);
    return ratio;
}
```

Uses CPUS 702 and conf 62b.

9.11.2 rebalance()

⟨hzsched() *rebalance ready procs* 241a⟩≡ (236c)

```
/* once a second, rebalance will reprioritize ready procs */
if(cpu->cpuno == 0)
    rebalance();
```

Uses rebalance() 241b.

⟨*function* rebalance 241b⟩≡ (877c)

```
static void
rebalance(void)
{
    int pri, npri, t, x;
    Schedq *rq;
    Proc *p;

    ⟨rebalance() return if less than one second since last rebalance 242b⟩

    for(pri=0, rq=runq; pri<Npriq; pri++, rq++){
another:
        p = rq->head;
        if(p == nil)
            continue;
        if(p->lastcpu != CPUS(cpu->cpuno)) // this has been the source of comments on 9fans, todo look if bug (
            continue;
        if(pri == p->basepri)
            continue;
        updatecpu(p);
        npri = reprioritize(p);
        if(npri != pri){
            x = arch_splhi();
            p = dequeueproc(rq, p);
            if(p)
                queueproc(&runq[npri], p);
            arch_splx(x);
            goto another;
        }
    }
}
```

Uses CPUS 702, Npriq 219a, dequeueproc() 222c, queueproc() 222b, reprioritize() 240, runq 221b, and updatecpu() 239h.

⟨*global* `balancetime` 242a⟩≡ (877c)
```
/*
 *  recalculate priorities once a second.  We need to do this
 *  since priorities will otherwise only be recalculated when
 *  the running process blocks.
 */
ulong balancetime;
```

⟨`rebalance()` *return if less than one second since last rebalance* 242b⟩≡ (241b)
```
t = cpu->ticks;
if(t - balancetime < Arch_HZ)
    return;
balancetime = t;
```
Uses `balancetime` 242a.

9.12 Halting (ARM)

When `runproc()` finds no ready process, it calls `arch_idlehands()`, which enables interrupts and executes the ARM `WFI` (Wait For Interrupt) instruction. The CPU halts until the next interrupt (typically the clock) arrives, saving power:

⟨*function* `arch_idlehands` *(raspberry pi1)(arm)* 242c⟩≡ (990b)
```
TEXT arch_idlehands(SB), $-4
    MOVW    CPSR, R3
    ORR $(PsrDirq|PsrDfiq), R3, R1      /* splfhi */
    MOVW    R1, CPSR

    DSB
    MOVW    nrdy(SB), R0
    CMP $0, R0
    MCR.EQ  CpSC, 0, R0, C(CpCACHE), C(CpCACHEintr), CpCACHEwait
    DSB

    MOVW    R3, CPSR            /* splx */
    RET
```

Chapter 10

Time

⟨systab *time syscalls* 243⟩≡ (57b)
```
  [SLEEP]     syssleep,
  [ALARM]     sysalarm,
```
Uses `sysalarm()` 257b.

The C prototypes are:

```
int   sleep(long millisecs);   // 0 = yield
long  alarm(ulong millisecs);  // 0 = cancel; returns previous
```

10.1 Time principles

Two separate concepts get lumped under the word "time" inside a kernel, and they are worth distinguishing before reading any code. The first is *elapsed-time measurement*: how long ago did this process start, how long has the current function been running, in how many microseconds must we retransmit that TCP packet. The second is *wall-clock time*: what date and time is it, so that a log line can be timestamped and a file's mtime can mean what users expect. The two are served by radically different hardware sources, have different precision and stability trade-offs, and—as we will see below—should never be confused with one another. Modern hardware usually offers several time sources at once: an x86 machine has the TSC (per-core cycle counter), the HPET (high-precision event timer), and the RTC (a real-time clock on a battery); a Raspberry Pi-class ARM machine has a free-running system counter at 1 MHZ, an ARM timer at 250 MHZ, and a per-CPU generic timer.

Every kernel needs a large number of software timers— `sleep`-style deadlines, `alarm`/timeout fires, TCP retransmits, scheduler preemption, filesystem delayed writeback—but it typically has only one hardware timer it can program to raise an interrupt. The standard answer is to multiplex: keep a sorted queue of (deadline, callback) pairs, and on each hardware tick advance a global counter and fire any callbacks whose deadlines have passed. The tick rate—the famous `HZ` constant—is a trade-off: a faster tick (1000 Hz) gives more accurate sleep and finer scheduler granularity but wastes power; a slower tick (100 Hz) is the opposite. Since Linux 2.6.21 (2007), the "tickless" or *dynamic tick* design has reprogrammed the hardware to fire at exactly the next deadline instead of every `HZ` tick, which lets idle laptops reach deep sleep states; FreeBSD 9 and Windows 8 followed. The underlying multiplexing structure is unchanged—only the firing schedule is.

The single most important distinction among kernel clocks is *monotonic time* versus *wall-clock time*. Monotonic time (POSIX `CLOCK_MONOTONIC`, `clock_gettime` with various flavors) counts elapsed time since some arbitrary boot-time zero and never goes backward. Wall-clock time (POSIX `CLOCK_REALTIME` and `gettimeofday`) tracks the real date and time, and can jump, either forward or backward, whenever NTP slews the clock, the sysadmin runs `date -s`, or DST starts. Any code measuring elapsed intervals— "how long did this take", "have we exceeded the timeout", "is it time to retransmit"—must use monotonic time. Any code exposing a human-readable timestamp (log lines, file mtimes, the `date` command) uses wall-clock time. Confusing the two is a famous bug class: cron jobs firing twice during autumn DST, distributed consensus systems electing two

leaders during an NTP slew, network retransmit timers misbehaving immediately after `ntpdate`. Plan 9 makes the same internal split: `fastticks` is monotonic; `/dev/time` is wall.

Plan 9's kernel heartbeat is `hzclock()`, a function called from the hardware-timer interrupt service routine `HZ` times per second. `hzclock()` is where the scheduler counts process CPU time, where `sleep` and `alarm` deadlines are checked, and where runaway-process detection happens. Alongside it sit two higher-resolution counters: `fastticks` (a 64-bit monotonic counter used for profiling and sub-tick sleep accounting) and `perfticks` (the ARM timer at 250 MHZ, used for cycle-level measurements). Wall-clock time, in keeping with the "everything is a file" principle (Section 2.1.8), is not exposed as a dedicated syscall like UNIX's `gettimeofday`; instead it is `read` from `/dev/time`, served by the `cons` device. The two time-related syscalls `sleep` and `alarm` above are the only ones. The rest of this chapter walks through the Raspberry Pi hardware-timer setup, `hzclock()`, `sleep/alarm`, and the generic "call this function in N milliseconds" mechanism built on top.

10.2 Overview (ARM)

We saw `hzclock()` briefly in the Scheduling chapter; here we see how it is actually wired up to the hardware, and what else runs on each clock tick.

The Raspberry Pi has several hardware timers running at different frequencies. The kernel uses system timer 3 (1 MHz, 64-bit) for `hzclock` interrupts and `fastticks`, and the ARM timer (250 MHz, 32-bit) for performance measurement.

10.3 Time units

The kernel uses several time units internally. The conversion between them is a frequent source of bugs, so the author introduced explicit typedefs:

10.3.1 Seconds

⟨*type* Txxx 244a⟩≡ (874a) 244c▷
```
typedef vlong   Tsec; // seconds
typedef int     Tms; // milliseconds
typedef vlong   Tmicro; // microseconds
typedef vlong   Tnano; // nanoseconds
```

⟨*macro* MS2NS 244b⟩≡ (873b)
```
#define MS2NS(n) (((vlong)(n))*1000000LL)
```

10.3.2 Ticks

A tick is the kernel's basic time unit: `cpu->ticks` is incremented once per `hzclock()` interrupt, i.e., `Arch_HZ` times per second (every 10 ms on the Raspberry Pi). Ticks are coarse but cheap to read. The `TK2MS` macro converts ticks to milliseconds:

⟨*type* Txxx 244c⟩+≡ (874a) ◁244a 245e▷
```
typedef vlong   Tval; // ticks
```

⟨*macro* TK2MS 244d⟩≡ (873b)
```
#define TK2MS(x) ((x)*(1000/Arch_HZ))
```

⟨*function* `tk2ms` 245a⟩≡ (874c)

```
/*
 *  This tk2ms avoids overflows that the macro version is prone to.
 *  It is a LOT slower so shouldn't be used if you're just converting
 *  a delta.
 */
ulong
tk2ms(ulong ticks)
{
    uvlong t, hz;

    t = ticks;
    hz = Arch_HZ;
    t *= 1000L;
    t = t/hz;
    ticks = t;
    return ticks;
}
```

⟨*function* `ms2tk` 245b⟩≡ (874c)

```
ulong
ms2tk(ulong ms)
{
    ⟨ms2tk() overflow approximation 245c⟩
    return (ms * Arch_HZ + 500) / 1000;
}
```

⟨`ms2tk()` *overflow approximation* 245c⟩≡ (245b)

```
/* avoid overflows at the cost of precision */
if(ms >= 1000000000 / Arch_HZ)
    return (ms / 1000) * Arch_HZ;
```

10.3.3 Fastticks (ARM)

Fastticks are much finer-grained than ticks: on the Raspberry Pi, the system timer runs at 1 MHz, so one fasttick is one microsecond (vs. 10 ms for a tick—10,000 times finer). Fastticks are used for the software timer system (`Timer.twhen`) and for precise time measurement.

⟨*constant* `SystimerFreq`*(arm)* 245d⟩≡ (1010d)

```
SystimerFreq    = 1*Mhz,
```

Uses `Mhz` 874a.

⟨*type* `Txxx` 245e⟩+≡ (874a) ◁244c

```
typedef vlong   Tfast; // fastticks
typedef uvlong   Tufast; // fastticks
```

⟨*function* `arch_us`*(arm)* 245f⟩≡ (1011b)

```
ulong // Tmicro
arch_us(void)
{
    ⟨arch_us() if non-standard systimer frequency 263c⟩
    return arch_fastticks(nil);
}
```

Uses `arch_fastticks`.

⟨*function* `clock_arch_fastticks(arm)` 246a⟩≡ (1011b)
```
Tufast
clock_arch_fastticks(uvlong *hz)
{
    Systimers *tn;
    ulong lo, hi;
    Tufast now;
    int s;

    ⟨arch_fastticks() if hz pointer given 246b⟩

    tn = (Systimers*)SYSTIMERS;
    s = arch_splhi();

    do{
        hi = tn->chi;
        lo = tn->clo;
    }while(tn->chi != hi);

    now = (uvlong)hi<<32 | lo;
    arch_splx(s);
    return now;
}
```
Uses `SYSTIMERS-2647` 43b.

⟨`arch_fastticks()` *if hz pointer given* 246b⟩≡ (246a)
```
if(hz)
    *hz = SystimerFreq;
```
Uses `SystimerFreq-2652` 245d.

10.4 Clock interrupt (ARM)

The clock interrupt is the heartbeat of the kernel. On the Raspberry Pi, system timer 3 fires an IRQ (`IRQclock`) which calls `clockintr()`. This dismisses the interrupt by writing to the timer's control/status register, then calls the portable `timerintr()` which processes all elapsed software timers (including `hzclock`).

⟨*struct* `Systimers(arm)` 246c⟩≡ (1011b)
```
// The order matters! the fields match the memory-mapped external registers.
struct Systimers {
    u32int  cs;

    u32int  clo;
    u32int  chi;

    u32int  c0;
    u32int  c1;
    u32int  c2;
    u32int  c3;
};
```

⟨*function* `clockintr(arm)` 246d⟩≡ (1011b)
```
static void
clockintr(Ureg *ureg, void*)
{
    Systimers *tn;

    tn = (Systimers*)SYSTIMERS;
    /* dismiss interrupt */
```

```
    tn->cs = 1<<3;
    // portable code
    timerintr(ureg, 0);
}
```

Uses SYSTIMERS-2647 43b and timerintr() 252.

10.5 Timers

The hardware provides a single compare register per timer channel—it can only trigger at one point in time. But the kernel needs many independent timers (for `sleep`, alarms, periodic callbacks, etc.). The solution is a sorted linked list of software `Timer` structures: only the earliest one is programmed into the hardware. When it fires, `timerintr()` processes it and programs the next. Here is a snapshot of CPU 0's list after the kernel has armed a `hzclock` at 10 ms, a process `sleep(50)`, and an Ethernet poll at 80 ms (times in fasttick units, relative to `now`):

```
timers[0].head
     |
     v
  +-------+     +-------+     +-------+
  | twhen |     | twhen |     | twhen |
  |  10ms |---->|  50ms |---->|  80ms |----> nil
  | hzclk |     | wakeup|     | ethpol|
  +-------+     +-------+     +-------+
     ^
     |
 arch_timerset(10ms)   <-- only the head is loaded
                           into the hardware compare
                           register (c3 on ARM)
```

When the hardware fires at 10 ms, `timerintr()`[252] pops the head, invokes its callback, re-adds it (for periodic timers like `hzclock`) or drops it, and then calls `arch_timerset()` on the new head (now 50 ms away). This is the classic timer wheel collapsed to a list: cheap when only a few timers are outstanding, but $O(n)$ to insert. The kernel gets away with the linear scan because the list is usually short.

10.5.1 Timer

⟨*struct* Timer 247⟩≡ (874a)

```
struct Timer
{
    /* Public interface */
    // enum<Timermode>
    int tmode;    /* See above */
    Tnano tns;    /* meaning defined by mode */ //nanosecond

    void  (*tf)(Ureg*, Timer*);
    void  *ta; // extra argument (generic pointer)

    /* Internal */
    Tfast  twhen;    /* ns represented in fastticks */
    ILock; // LOCK ORDERING: ilock(timer); ilock(timers)

    ⟨Timer extra fields 248c⟩
    };
```

⟨*enum* `timermode` 248a⟩≡ (874a)
```
/*
 * fasttick timer interrupts
 */
enum Timermode
{
    Trelative,  /* timer programmed in ns from now */
    Tperiodic,  /* periodic timer, period in ns */
};
```

10.5.2 `Timers` and CPU `timers`

Each CPU has its own sorted timer list (`timers[cpuno]`), so timers fire on the CPU that created them. The list is protected by an `ILock` (not a regular `Lock`) because timer manipulation happens in interrupt context:

⟨*struct* `Timers` 248b⟩≡ (874a)
```
struct Timers
{
    // list<Timer> (next = Timer.tnext)
    Timer *head;
    // extra
    ILock;
};
```

⟨`Timer` *extra fields* 248c⟩≡ (247) 248d▷
```
// list<Timer> of Timers.head
Timer *tnext;
```

⟨`Timer` *extra fields* 248d⟩+≡ (247) ◁248c
```
// option<ref<list<Timer>> Timers.head (of one of global timers[MAXCPUS])
Timers  *tt;    /* Timers queue this timer runs on */
```

⟨*global* `timers` 248e⟩≡ (874c)
```
static Timers timers[MAXCPUS];
```

10.5.3 `tadd()/tdel()`

`tadd()` inserts a timer into the sorted list (by `twhen`). If the new timer becomes the head (earliest), it returns the new deadline so the caller can reprogram the hardware. `tdel()` removes a timer:

⟨*function* `tadd` 248f⟩≡ (874c)
```
static Tfast
tadd(Timers *tt, Timer *nt)
{
    Timer *t, **last;

    /* Called with tt locked */
    ⟨tadd() sanity check nt 249a⟩
    ⟨tadd() set nt->twhen 249b⟩

    // insert_sorted(nt, tt, (a->twhen <=> b->twhen))
    for(last = &tt->head; t = *last; last = &t->tnext){
        if(t->twhen > nt->twhen)
            break;
    }
    nt->tnext = *last;
    *last = nt;
```

```
    nt->tt = tt;
    if(last == &tt->head)
        return nt->twhen;
    return 0;
}
```

⟨tadd() *sanity check nt* 249a⟩≡ (248f)
```
assert(nt->tt == nil);
```
Uses assert 853f.

⟨tadd() *set* nt->twhen 249b⟩≡ (248f)
```
switch(nt->tmode){
case Trelative:
    if(nt->tns <= 0)
        nt->tns = 1;
    nt->twhen = arch_fastticks(nil) + ns2fastticks(nt->tns);
    break;
case Tperiodic:
    assert(nt->tns >= 100000);  /* At least 100 Âµs period */
    if(nt->twhen == 0){
        ⟨tadd() look if can combine with identical timer 249c⟩
        else
            nt->twhen = arch_fastticks(nil);
    }
    nt->twhen += ns2fastticks(nt->tns);
    break;
default:
    panic("timer: impossible");
    break;
}
```
Uses Tperiodic 248a, Trelative 248a, arch_fastticks, assert 853f, and ns2fastticks() 264b.

⟨tadd() *look if can combine with identical timer* 249c⟩≡ (249b)
```
/* look for another timer at same frequency for combining */
for(t = tt->head; t; t = t->tnext){
    if(t->tmode == Tperiodic && t->tns == nt->tns)
        break;
}
if (t)
    nt->twhen = t->twhen;
```
Uses Tperiodic 248a.

⟨*function* tdel 249d⟩≡ (874c)
```
static Tfast
tdel(Timer *dt)
{
    Timer *t, **last;
    Timers *tt;

    tt = dt->tt;
    ⟨tdel() sanity check tt 250a⟩
    // remove_list(dt, tt)
    for(last = &tt->head; t = *last; last = &t->tnext){
        if(t == dt){
            assert(dt->tt);
            dt->tt = nil;
            *last = t->tnext;
            break;
        }
```

```
    }
    if(last == &tt->head && tt->head)
        return tt->head->twhen;
    return 0;
}
```

Uses assert 853f.

⟨tdel() *sanity check tt* 250a⟩≡ (249d)

```
if (tt == nil)
    return 0; //todo: panic instead
```

10.5.4 timeradd()//timerdel()

timeradd() and timerdel() are the public interface. They lock the Timer first, then the Timers list (respecting the lock ordering), and call tadd()/tdel(). If the new timer becomes the earliest, arch_timerset() is called to reprogram the hardware:

⟨*function* timeradd 250b⟩≡ (874c)

```
/* add or modify a timer */
void
timeradd(Timer *nt)
{
    Timers *tt;
    Tfast when;

    /* Must lock Timer struct before Timers struct */
    ilock(nt);

    ⟨timeradd() if timer was already in a list 250c⟩
    tt = &timers[cpu->cpuno];
    ilock(tt);
    when = tadd(tt, nt);
    if(when)
        arch_timerset(when);
    iunlock(tt);
    iunlock(nt);
}
```

Uses ilock() 79d, iunlock() 80b, tadd() 248f, and timers-2645 248e.

⟨timeradd() *if timer was already in a list* 250c⟩≡ (250b)

```
if(tt = nt->tt){
    ilock(tt);
    tdel(nt);
    iunlock(tt);
}
```

Uses ilock() 79d, iunlock() 80b, and tdel() 249d.

⟨pexit() *at the start, free resources* 250d⟩≡ (152c) 257d▷

```
if (up->tt)
    timerdel(up);
```

Uses timerdel() 250e and up.

⟨*function* timerdel 250e⟩≡ (874c)

```
void
timerdel(Timer *dt)
{
    Timers *tt;
    Tfast when;
```

```
    ilock(dt);
    if(tt = dt->tt){
        ilock(tt);
        when = tdel(dt);
        if(when && tt == &timers[cpu->cpuno])
            arch_timerset(tt->head->twhen);
        iunlock(tt);
    }
    iunlock(dt);
}
```

Uses `ilock()` 79d, `iunlock()` 80b, `tdel()` 249d, and `timers-2645` 248e.

10.5.5 `arch_timerset()` (ARM)

`arch_timerset()` programs the hardware timer to fire at the given fasttick value by writing to system timer 3's compare register (`c3`). The period is clamped to avoid either firing too fast (overwhelming the CPU) or too slow (missing the next `hzclock`):

⟨*function* `arch_timerset` *(arm)* 251a⟩≡ (1011b)

```
void
arch_timerset(Tfast next)
{
    Systimers *tn;
    Tufast now;
    long period;

    now = arch_fastticks(nil);
    period = next - now;
    ⟨arch_timerset() sanitize period(arm) 251b⟩
    ⟨arch_timerset() if not cpu0(arm) 547g⟩
    else{
        tn = (Systimers*)SYSTIMERS;
        tn->c3 = (ulong)(now + period);
    }
}
```

Uses `SYSTIMERS-2647` 43b and `arch_fastticks`.

⟨`arch_timerset()` *sanitize period(arm)* 251b⟩≡ (251a)

```
if(period < MinPeriod)
    period = MinPeriod;
else if(period > MaxPeriod)
    period = MaxPeriod;
```

Uses `MaxPeriod-2653` 251c and `MinPeriod-2654` 251d.

⟨*constant* `MaxPeriod` *(arm)* 251c⟩≡ (1010d)

```
MaxPeriod   = SystimerFreq / Arch_HZ,
```

Uses `SystimerFreq-2652` 245d.

⟨*constant* `MinPeriod` *(arm)* 251d⟩≡ (1010d)

```
MinPeriod   = SystimerFreq / (100*Arch_HZ),
```

Uses `SystimerFreq-2652` 245d.

10.5.6 timerintr()

timerintr() is the portable timer interrupt handler. It walks the sorted timer list, popping and calling every timer whose twhen has passed. Periodic timers are re-added to the list after firing. hzclock() is called last (via the callhzclock flag) to ensure all other timer callbacks complete first:

⟨*interrupt callback timerintr* 252⟩≡ (874c)

```
void
timerintr(Ureg *u, Tval)
{
    Timer *t;
    Timers *tt;
    Tufast when, now;
    bool callhzclock = false;
    ⟨timerintr() other locals 253b⟩

    ⟨timerintr() start, stat 662d⟩
    tt = &timers[cpu->cpuno];
    now = arch_fastticks(nil);
    ⟨timerintr() sanity check now 253a⟩
    ilock(tt);
    ⟨timerintr() before loop, initialize count 253c⟩
    while((t = tt->head) != nil){
        /*
         * No need to ilock t here: any manipulation of t
         * requires tdel(t) and this must be done with a
         * lock to tt held.  We have tt, so the tdel will
         * wait until we're done
         */
        when = t->twhen;
        if(when > now){
            arch_timerset(when);
            iunlock(tt);
            if(callhzclock)
                hzclock(u);
            return;
        }
        // else
        // pop_list(tt)
        tt->head = t->tnext;
        assert(t->tt == tt);
        t->tt = nil;
        ⟨timerintr() in timers loop, stat 662e⟩
        iunlock(tt);

        // call the callback
        if(t->tf)
            (*t->tf)(u, t);
        else
            callhzclock = true;

        ilock(tt);
        if(t->tmode == Tperiodic)
            tadd(tt, t);
        ⟨timerintr() end of timers loop, sanity check count 253e⟩
    }
    iunlock(tt); // unreachable?
}
```

Uses Tperiodic 248a, arch_fastticks, assert 853f, false, hzclock() 236f, ilock() 79d, iunlock() 80b, tadd() 248f, timers-2645 248e, and true.

⟨timerintr() *sanity check now* 253a⟩≡ (252)
```
if(now == 0)
    panic("timerintr: zero arch_fastticks()");
```

⟨timerintr() *other locals* 253b⟩≡ (252)
```
int count;
```

⟨timerintr() *before loop, initialize count* 253c⟩≡ (252)
```
count = Maxtimerloops;
```
Uses Maxtimerloops-2644 253d.

⟨*constant* Maxtimerloops 253d⟩≡ (874c)
```
Maxtimerloops = 20*1000,
```

⟨timerintr() *end of timers loop, sanity check count* 253e⟩≡ (252)
```
if (--count <= 0) {
    count = Maxtimerloops;
    iprint("timerintr: probably stuck in while loop; "
        "scrutinise clock.c or use faster cycle "
        "counter\n");
}
```
Uses Maxtimerloops-2644 253d and iprint.

10.5.7 Clock callbacks: addclock0link()

addclock0link() registers a periodic callback on CPU 0's timer list. Device drivers use this to run polling functions at regular intervals (e.g., checking the Ethernet controller, updating the console). If ms is 0, the callback runs at the hzclock frequency (every 10 ms):

⟨*function* addclock0link 253f⟩≡ (874c)
```
Timer*
addclock0link(void (*f)(void), Tms ms)
{
    Timer *nt;
    Tfast when;

    ⟨addclock0link() sanity check timersinit 254a⟩

    nt = malloc(sizeof(Timer));
    ⟨addclock0link() sanity check nt 254b⟩
    ⟨addclock0link() sanitize ms 254c⟩

    nt->tns = (Tnano)ms * 1000000LL; // MS2NS(ms)
    nt->tmode = Tperiodic;
    nt->tt = nil;
    nt->tf = (void (*)(Ureg*, Timer*))f;

    // those clock callbacks are all done on the bootstrap processor
    ilock(&timers[0]);
    when = tadd(&timers[0], nt);
    if(when)
        arch_timerset(when);
    iunlock(&timers[0]);
    return nt;
}
```
Uses Tperiodic 248a, ilock() 79d, iunlock() 80b, malloc() 103c, tadd() 248f, and timers-2645 248e.

⟨addclock0link() *sanity check timersinit* 254a⟩≡ (253f)
```
if(!timersinited)
    panic("addclock0link: timersinit not called yet");
```
Uses timersinited-2646 281a.

⟨addclock0link() *sanity check nt* 254b⟩≡ (253f)
```
if(nt == nil)
    error(Enomem);
```

⟨addclock0link() *sanitize ms* 254c⟩≡ (253f)
```
/* Synchronize to hztimer if ms is 0 */
if(ms == 0)
    ms = 1000 / Arch_HZ; // tk2ms(1)
```

10.5.8 Delay (ARM)

`arch_delay()` and `arch_microdelay()` are busy-wait loops used by device drivers that need to wait for hardware to respond (e.g., after writing a control register). They spin-wait on the system timer counter—no interrupt is involved:

⟨*function* clock_arch_microdelay*(arm)* 254d⟩≡ (1011b)
```
void
clock_arch_microdelay(int n) // Tmicro
{
    Systimers *tn;
    u32int now, diff;

    diff = n + 1;
    tn = (Systimers*)SYSTIMERS;
    now = tn->clo;
    while(tn->clo - now < diff)
        ;
}
```
Uses SYSTIMERS-2647 43b.

⟨*function* clock_arch_delay*(arm)* 254e⟩≡ (1011b)
```
void
clock_arch_delay(Tms n)
{
    while(--n >= 0)
        arch_microdelay(1000);
}
```
Uses arch_microdelay.

10.6 syssleep()

The `sleep(ms)` syscall puts the calling process to sleep for `ms` milliseconds. It works by combining the `Rendez`/`sleep()`/`wakeup()` mechanism from the Scheduling chapter with a one-shot timer: `tsleep()` adds a timer that will call `twakeup()` after the requested delay, then calls `sleep()` with a condition that is always false (so the process sleeps until the timer wakes it).

⟨Proc *synchronization fields* 254f⟩+≡ (66) ◁233c 479c▷
```
Rendez  sleepr;    /* place for syssleep/debug/tsleep */
```

⟨*syscall sleep* 255a⟩≡ (881b)

```
// int sleep(long millisecs);
long
syssleep(ulong* arg)
{
    int n;
    n = arg[0];
    if(n <= 0) {
        ⟨syssleep() optional edfyield() for real-time scheduling 565b⟩
        yield();
        return 0;
    }
    ⟨syssleep() sanitize n 255b⟩
    tsleep(&up->sleepr, returnfalse, 0, n);
    return 0;
}
```

Uses returnfalse() 104c, tsleep, up, and yield() 231b.

⟨syssleep() *sanitize n* 255b⟩≡ (255a)

```
if(n < TK2MS(1))
    n = TK2MS(1);
```

Uses TK2MS 244d.

⟨*function* tsleep 255c⟩≡ (877c)

```
void
proc_tsleep(Rendez *r, int (*fn)(void*), void *arg, ulong ms)
{
    ⟨tsleep() sanity check up timer 256a⟩
    up->tns = MS2NS(ms);
    up->tmode = Trelative;
    up->tf = twakeup;

    up->ta = up; // so can access r from wakeup
    up->trend = r;

    up->tfn = fn; // needed?

    timeradd(up);

    if(waserror()){
        timerdel(up);
        nexterror();
    }
    sleep(r, tfn, arg);
    // back after sleep
    ⟨tsleep() reset timer after sleep 255d⟩
    poperror();
}
```

Uses MS2NS 244b, Trelative 248a, nexterror, poperror 666d, sleep, tfn() 256d, timeradd() 250b, timerdel() 250e, twakeup() 256e, up, and waserror 666d.

⟨tsleep() *reset timer after sleep* 255d⟩≡ (255c)

```
if(up->tt)
    timerdel(up);
up->twhen = 0;
```

Uses timerdel() 250e and up.

⟨tsleep() *sanity check up timer* 256a⟩≡ (255c)
```
if (up->tt){
    print("tsleep: timer active: mode %d, tf %#p\n", up->tmode, up->tf);
    timerdel(up);
}
```
Uses print, timerdel() 250e, and up.

⟨Proc *other fields* 256b⟩+≡ (66) ◁163c 256c▷
```
Timer;      /* For tsleep and real-time */
```

⟨Proc *other fields* 256c⟩+≡ (66) ◁256b 257c▷
```
Rendez  *trend;
int (*tfn)(void*);
```

⟨*function* tfn 256d⟩≡ (877c)
```
static int
tfn(void *arg)
{
    return up->trend == nil || up->tfn(arg);
}
```
Uses up.

⟨*function* twakeup 256e⟩≡ (877c)
```
void
twakeup(Ureg*, Timer *t)
{
    Proc *p;
    Rendez *trend;

    p = t->ta;
    trend = p->trend;
    p->trend = nil;
    if(trend)
        wakeup(trend);
}
```
Uses wakeup.

10.7 Alarms

Alarms are a simpler, coarser mechanism than timers. The `alarm(ms)` syscall arranges for a "sys: alarm" note to be posted to the calling process after `ms` milliseconds. Unlike `sleep()`, the process continues running—the note arrives asynchronously, like a UNIX `SIGALRM`. Alarms are stored as a sorted linked list of `Proc` pointers, checked on every `hzclock()` tick by `checkalarms()`:

⟨*struct* Alarms 256f⟩≡ (868c)
```
struct Alarms
{
    // sorted_list<ref<Proc> (next = Proc.palarm, sort_key = Proc.alarm)
    Proc  *head;

    // Extra
    QLock;
};
```

⟨Proc *extra fields* 256g⟩+≡ (66) ◁221d
```
// Alarms.head chain?
Proc  *palarm;  /* Next alarm time */
```

⟨*global* `alarms` 257a⟩≡ (874b)
```
static Alarms   alarms;
```

10.8 sysalarm()

⟨*syscall alarm* 257b⟩≡ (881b)
```
// long alarm(unsigned long millisecs);
long
sysalarm(ulong* arg)
{
    return procalarm(arg[0]);
}
```
Uses `procalarm()` 257e.

⟨`Proc` *other fields* 257c⟩+≡ (66) ◁256c 307c▷
```
// option<ticks>
ulong alarm;    /* Time of call */ // Tval ticks
```

⟨`pexit()` *at the start, free resources* 257d⟩+≡ (152c) ◁250d 492c▷
```
up->alarm = 0;
```
Uses `up`.

⟨*function* `procalarm` 257e⟩≡ (874b)
```
ulong
procalarm(ulong time) // Tms
{
    Proc **l, *f;
    ulong when, old; // Tval ticks

    ⟨procalarm() compute remaining time in old alarm 258a⟩
    ⟨procalarm() if time is zero, reset alarm 258b⟩
    when = ms2tk(time) + CPUS(0)->ticks;
    ⟨procalarm() sanitize when 258c⟩

    qlock(&alarms);
    l = &alarms.head;
    // remove_list(up, alarms)
    for(f = *l; f; f = f->palarm) {
        if(up == f){
            *l = up->palarm;
            break;
        }
        l = &f->palarm;
    }
    up->palarm = nil;

    // add_sorted_list(up, alarms)
    if(alarms.head) {
        l = &alarms.head;
        for(f = *l; f; f = f->palarm) {
            if((long)(f->alarm - when) >= 0) {
                up->palarm = f;
                *l = up;
                goto done;
            }
            l = &f->palarm;
        }
        *l = up;
```

```
    }
    else
        alarms.head = up;

done:
    up->alarm = when;
    qunlock(&alarms);

    return old;
}
```

Uses CPUS 702, alarms-2607 257a, ms2tk() 245b, qlock() 81f, qunlock() 82c, and up.

⟨procalarm() *compute remaining time in old alarm* 258a⟩≡ (257e)

```
if(up->alarm)
    old = tk2ms(up->alarm - CPUS(0)->ticks);
else
    old = 0;
```

Uses CPUS 702, tk2ms() 245a, and up.

⟨procalarm() *if time is zero, reset alarm* 258b⟩≡ (257e)

```
if(time == 0) {
    up->alarm = 0;
    return old;
}
```

Uses up.

⟨procalarm() *sanitize when* 258c⟩≡ (257e)

```
if(when == 0)       /* ticks have wrapped to 0? */
    when = 1;   /* distinguish a wrapped alarm from no alarm */
```

10.8.1 alarmkproc()

The alarm kernel process runs in an infinite loop: it sleeps until checkalarms() (called from hzclock()) wakes it, then walks the sorted alarm list posting "alarm" notes to processes whose alarm time has passed. The indirection through a kernel process avoids doing the postnote() work inside the interrupt handler:

⟨*global* alarmr 258d⟩≡ (874b)

```
static Rendez alarmr;
```

⟨*kernel process alarmkproc* 258e⟩≡ (874b)

```
// Kernel Process for alarm management
void
alarmkproc(void*)
{
    Proc *rp;
    ulong now;

    for(;;){
        now = CPUS(0)->ticks;

        qlock(&alarms);
        /*
         * the odd test of now vs. rp->alarm is to cope with
         * now wrapping around.
         */
        while((rp = alarms.head) && (long)(now - rp->alarm) >= 0){
            if(rp->alarm != 0L){
                if(canqlock(&rp->debug)){
```

```
                    if(!waserror()){
                        postnote(rp, 0, "alarm", NUser);
                        poperror();
                    }
                    qunlock(&rp->debug);
                    rp->alarm = 0L;
                }else
                    break;
            }
            alarms.head = rp->palarm;
        }
        qunlock(&alarms);

        sleep(&alarmr, returnfalse, 0);
    }
}
```

Uses CPUS 702, NUser 449a, alarmr-2608 258d, alarms-2607 257a, canqlock() 83a, poperror 666d, postnote, qlock() 81f, qunlock() 82c, returnfalse() 104c, sleep, and waserror 666d.

10.8.2 checkalarms()

checkalarms() is called on every clock tick. It just checks whether the earliest alarm has expired and, if so, wakes up alarmkproc() to do the actual work. This keeps the interrupt handler fast:

⟨hzclock() *check the alarms* 259a⟩≡ (236f)

```
checkalarms();
```

Uses checkalarms() 259b.

⟨*function* checkalarms 259b⟩≡ (874b)

```
/*
 *  called every clock tick
 */
void
checkalarms(void)
{
    Proc *p;
    ulong now;

    p = alarms.head;
    now = CPUS(0)->ticks;

    if(p && (long)(now - p->alarm) >= 0)
        wakeup(&alarmr);
}
```

Uses CPUS 702, alarmr-2608 258d, alarms-2607 257a, and wakeup.

10.9 Time of day

The time of day (wall clock) is maintained by the TOD structure, which converts fastticks into nanoseconds since the epoch using a fixed-point multiplier. The multiplier is calibrated from cpu->cpuhz (the fasttick frequency). A periodic callback (todfix) prevents the 64-bit intermediate values from overflowing, and todset() allows the time to be adjusted gradually (via a delta applied over several ticks) to avoid jumps.

⟨*struct* TOD 259c⟩≡ (880b)

```
struct TOD {
    bool init;      /* true if initialized */
```

```
    ulong   cnt;

    uvlong  multiplier; /* ns = off + (multiplier*ticks)>>31 */
    uvlong  divider;    /* ticks = (divider*(ns-off))>>31 */
    uvlong  umultiplier;/* Âµs = (Âµmultiplier*ticks)>>31 */
    uvlong  udivider;   /* ticks = (Âµdivider*Âµs)>>31 */

    vlong   hz;     /* frequency of fast clock */
    vlong   last;       /* last reading of fast clock */

    vlong   off;        /* offset from epoch to last */
    vlong   lasttime;   /* last return value from todget */
    vlong   delta;  /* add 'delta' each slow clock tick from sstart to send */

    ulong   sstart;     /* ... */
    ulong   send;       /* ... */

    // Extra
    Lock;
};
```

⟨*global* `tod` 260a⟩≡ (880b)
```
struct TOD tod;
```
Uses `TOD` 259c.

⟨*devcons.c Exxx errors* 260b⟩≡ (696b)
```
char *Ebadtimectl = "bad time control";
```
Uses `Ebadtimectl` 260b.

⟨*constant* `VLNUMSIZE` 260c⟩≡ (696b)
```
VLNUMSIZE=  22,
```

`todinit()` bootstraps the wall clock. It reads the fasttick counter once (which also fills in `tod.hz`, the fasttick frequency), computes the multipliers, and registers `todfix()` as a periodic callback (every 100 ms) to prevent overflow in the tick-to-nanosecond conversion.

⟨*function* `todinit` 260d⟩≡ (880b)
```
void
todinit(void)
{
    if(tod.init)
        return;
    ilock(&tod);
    tod.init = true;            /* prevent reentry via arch_fastticks */
    tod.last = arch_fastticks((uvlong *)&tod.hz);
    iunlock(&tod);
    todsetfreq(tod.hz);
    addclock0link(todfix, 100);
}
```
Uses `addclock0link()` 253f, `arch_fastticks`, `ilock()` 79d, `iunlock()` 80b, `tod` 260a, `todfix()` 263a, `todsetfreq()` 261a, and `true`.

`todsetfreq()` precomputes the fixed-point multipliers used by `todget()` and friends. The key idea is to avoid floating-point arithmetic in the kernel: `mk64fract(TODFREQ, f)` produces a 64-bit fraction whose upper

32 bits are the integer part and lower 32 bits are the fractional part, so multiplying by it and shifting right 31 gives an accurate conversion from fastticks to nanoseconds without any division at runtime.

⟨*function* `todsetfreq` 261a⟩≡ (880b)

```
/*
 *  calculate multiplier
 */
void
todsetfreq(vlong f)
{
    if (f <= 0)
        panic("todsetfreq: freq %lld <= 0", f);
    ilock(&tod);
    tod.hz = f;

    /* calculate multiplier for time conversion */
    tod.multiplier  = mk64fract(TODFREQ, f);
    tod.divider     = mk64fract(f, TODFREQ) + 1;
    tod.umultiplier = mk64fract(MicroFREQ, f);
    tod.udivider    = mk64fract(f, MicroFREQ) + 1;
    iunlock(&tod);
}
```

Uses `MicroFREQ-2670` 880b, `TODFREQ-2669` 880b, `ilock()` 79d, `iunlock()` 80b, `mk64fract()` 264c, and `tod` 260a.

`todset()` has two modes. When `t >= 0`, it sets the time absolutely (used at boot or when writing `/dev/time`): `tod.off` becomes the new epoch offset and `tod.last` is reset to the current fasttick. When `t < 0`, it sets up a gradual correction: over the next `n` clock ticks, `delta` nanoseconds will be spread evenly so the clock smoothly drifts to the correct time instead of jumping. This prevents applications that measure elapsed time from seeing negative intervals.

⟨*function* `todset` 261b⟩≡ (880b)

```
/*
 *  Set the time of day struct
 */
void
todset(vlong t, vlong delta, int n)
{
    if(!tod.init)
        todinit();

    ilock(&tod);
    if(t >= 0){
        tod.off = t;
        tod.last = arch_fastticks(nil);
        tod.lasttime = 0;
        tod.delta = 0;
        tod.sstart = tod.send;
    } else {
        if(n <= 0)
            n = 1;
        n *= Arch_HZ;
        if(delta < 0 && n > -delta)
            n = -delta;
        if(delta > 0 && n > delta)
            n = delta;
        if (n == 0) {
            iprint("todset: n == 0, delta == %lld\n", delta);
            delta = 0;
        } else
            delta /= n;
```

```
        tod.sstart = CPUS(0)->ticks;
        tod.send = tod.sstart + n;
        tod.delta = delta;
    }
    iunlock(&tod);
}
```

Uses CPUS 702, arch_fastticks, ilock() 79d, iprint, iunlock() 80b, tod 260a, and todinit() 260d.

`todget()` is the workhorse: it reads the current fasttick, applies the fixed-point multiplier to convert to nanoseconds, adds the epoch offset, and applies any in-progress delta correction. The monotonicity guarantee (`x >= tod.lasttime`) ensures time never goes backward even if the fasttick counter wraps or the delta is negative. The optional `ticksp` output gives callers the raw fasttick value synchronized with the nanosecond reading, which is important for `/dev/time` and `/dev/bintime`.

⟨*function* todget 262⟩≡ (880b)

```
/*
 *  get time of day
 */
vlong
todget(vlong *ticksp)
{
    uvlong x;
    vlong ticks, diff;
    ulong t;

    if(!tod.init)
        todinit();

    /*
     * we don't want time to pass twixt the measuring of arch_fastticks
     * and grabbing tod.last.  Also none of the vlongs are atomic so
     * we have to look at them inside the lock.
     */
    ilock(&tod);
    tod.cnt++;
    ticks = arch_fastticks(nil);

    /* add in correction */
    if(tod.sstart != tod.send){
        t = CPUS(0)->ticks;
        if(t >= tod.send)
            t = tod.send;
        tod.off = tod.off + tod.delta*(t - tod.sstart);
        tod.sstart = t;
    }

    /* convert to epoch */
    diff = ticks - tod.last;
    if(diff < 0)
        diff = 0;
    arch_mul64fract(&x, diff, tod.multiplier);
    x += tod.off;

    /* time can't go backwards */
    if(x < tod.lasttime)
        x = tod.lasttime;
    else
        tod.lasttime = x;

    iunlock(&tod);
```

```
    if(ticksp != nil)
        *ticksp = ticks;

    return x;
}
```

Uses CPUS 702, arch_fastticks, ilock() 79d, iunlock() 80b, tod 260a, and todinit() 260d.

`todfix()` is called every 100 ms by the clock callback. Its job is to "rebase" the time-of-day calculation. Without it, the difference `ticks - tod.last` would grow unboundedly, and the multiplication `diff * tod.multiplier` would eventually overflow 64 bits. By periodically converting the accumulated ticks into nanoseconds and folding them into `tod.off`, the `diff` stays small.

⟨*function* todfix 263a⟩≡ (880b)

```
/*
 *  called regularly to avoid calculation overflows
 */
static void
todfix(void)
{
    vlong ticks, diff;
    uvlong x;

    ticks = arch_fastticks(nil);

    diff = ticks - tod.last;
    if(diff > tod.hz){
        ilock(&tod);

        /* convert to epoch */
        arch_mul64fract(&x, diff, tod.multiplier);
if(x > 30000000000ULL) iprint("todfix %llud\n", x);
        x += tod.off;

        /* protect against overflows */
        tod.last = ticks;
        tod.off = x;

        iunlock(&tod);
    }
}
```

Uses arch_fastticks, ilock() 79d, iprint, iunlock() 80b, and tod 260a.

⟨*function* seconds 263b⟩≡ (880b)

```
long
seconds(void)
{
    return (vlong)todget(nil) / TODFREQ;
}
```

Uses TODFREQ-2669 880b and todget() 262.

⟨arch_us() *if non-standard systimer frequency* 263c⟩≡ (245f)

```
if(SystimerFreq != 1*Mhz)
    return fastticks2us(arch_fastticks(nil));
```

Uses Mhz 874a, SystimerFreq-2652 245d, arch_fastticks, and fastticks2us() 264a.

⟨*function* `fastticks2us` 264a⟩≡ (880b)
```
uvlong
fastticks2us(uvlong ticks)
{
    uvlong res;

    if(!tod.init)
        todinit();
    arch_mul64fract(&res, ticks, tod.umultiplier);
    return res;
}
```
Uses `tod` 260a and `todinit()` 260d.

⟨*function* `ns2fastticks` 264b⟩≡ (880b)
```
/*
 *  convert nanoseconds to fast ticks
 */
uvlong
ns2fastticks(uvlong ns)
{
    uvlong res;

    if(!tod.init)
        todinit();
    arch_mul64fract(&res, ns, tod.divider);
    return res;
}
```
Uses `tod` 260a and `todinit()` 260d.

`mk64fract()` builds a fixed-point multiplier for the ratio `to/from`. Shifting `to` left by 32 before dividing puts the fractional bits in the low 32 bits. The companion `arch_mul64fract()` multiplies a 64-bit value by this fraction and shifts right 31, giving the effect of multiplying by `to/from` without any floating-point. For example, `mk64fract(1000000000, 1000000)` produces a multiplier that converts microsecond ticks to nanoseconds.

⟨*function* `mk64fract` 264c⟩≡ (880b)
```
/*
 * Make a 64 bit fixed point number that has a decimal point
 * to the left of the low order 32 bits.  This is used with
 * arch_mul64fract for converting twixt nanoseconds and arch_fastticks.
 *
 *  multiplier = (to<<32)/from
 */
uvlong
mk64fract(uvlong to, uvlong from)
{
    return (to<<32) / from;
}
```

10.10 `sysnsec()`

10.11 `/dev/time`

Reading `/dev/time` returns the current time in seconds since the epoch (used by `date(1)`). Writing to it sets the time—this is how the system clock is initialized from the network or a real-time clock at boot:

⟨`consread()` *cases* 264d⟩+≡ (394c) ◁205d 266a▷
```
case Qtime:
```

```
    return readtime((ulong)offset, buf, n);
```

Uses Qtime-1791 633a and readtime() 265c.

⟨*global* fasthz 265a⟩≡ (696b)

```
vlong   fasthz;
```

⟨conswrite() *cases* 265b⟩+≡ (395a) ◁216b 266b▷

```
case Qtime:
    if(!iseve())
        error(Eperm);
    return writetime(a, n);
```

Uses Qtime-1791 633a, iseve() 591b, and writetime() 265d.

⟨*function* readtime 265c⟩≡ (696b)

```
/*
 *  like the old #c/time but with added info.  Return
 *
 *  secs    nanosecs    fastticks   fasthz
 */
static int
readtime(ulong off, char *buf, int n)
{
    vlong   nsec, ticks;
    long sec;
    char str[7*NUMSIZE];

    nsec = todget(&ticks);
    if(fasthz == 0LL)
        arch_fastticks((uvlong*)&fasthz);
    sec = nsec/1000000000ULL;
    snprint(str, sizeof(str), "%*lud %*llud %*llud %*llud ",
        NUMSIZE-1, sec,
        VLNUMSIZE-1, nsec,
        VLNUMSIZE-1, ticks,
        VLNUMSIZE-1, fasthz);
    return readstr(off, buf, n, str);
}
```

Uses NUMSIZE 867a, VLNUMSIZE-1796 260c, arch_fastticks, fasthz 265a, readstr() 670d, and todget() 262.

⟨*function* writetime 265d⟩≡ (696b)

```
/*
 *  set the time in seconds
 */
static int
writetime(char *buf, int n)
{
    char b[13];
    long i;
    vlong now;

    if(n >= sizeof(b))
        error(Ebadtimectl);
    strncpy(b, buf, n);
    b[n] = 0;
    i = strtol(b, 0, 0);
    if(i <= 0)
        error(Ebadtimectl);
    now = i*1000000000LL;
    todset(now, 0, 0);
```

```
    return n;
}
```
Uses Ebadtimectl 260b and todset() 261b.

10.12 /dev/bintime

/dev/bintime is the binary counterpart of /dev/time. Instead of a text string, it returns three little-endian 64-bit integers: nanoseconds since the epoch, the fasttick counter, and the fasttick frequency. Because nsec and ticks come from the same todget() call, they are synchronized—userspace can use the pair to convert between the two time bases without kernel round-trips. Writing to /dev/bintime supports three commands, selected by a one-byte prefix: 'n' sets the time absolutely (nanoseconds), 'd' applies a gradual delta correction over a period (used by NTP daemons), and 'f' recalibrates the fasttick frequency. The le2vlong/vlong2le helpers handle byte-order conversion using uvorder, a constant whose byte representation encodes the mapping from little-endian to native order.

⟨consread() *cases* 266a⟩+≡ (394c) ◁264d 504b▷
```
case Qbintime:
    return readbintime(buf, n);
```
Uses Qbintime-1783 633a and readbintime() 266c.

⟨conswrite() *cases* 266b⟩+≡ (395a) ◁265b 397a▷
```
case Qbintime:
    if(!iseve())
        error(Eperm);
    return writebintime(a, n);
```
Uses Qbintime-1783 633a, iseve() 591b, and writebintime() 267a.

⟨*function* readbintime 266c⟩≡ (696b)
```
/*
 *  read binary time info.  all numbers are little endian.
 *  ticks and nsec are syncronized.
 */
static int
readbintime(char *buf, int n)
{
    int i;
    vlong nsec, ticks;
    byte *b = (byte*)buf;

    i = 0;
    if(fasthz == 0LL)
        arch_fastticks((uvlong*)&fasthz);
    nsec = todget(&ticks);
    if(n >= 3*sizeof(uvlong)){
        vlong2le(b+2*sizeof(uvlong), fasthz);
        i += sizeof(uvlong);
    }
    if(n >= 2*sizeof(uvlong)){
        vlong2le(b+sizeof(uvlong), ticks);
        i += sizeof(uvlong);
    }
    if(n >= 8){
        vlong2le(b, nsec);
        i += sizeof(vlong);
    }
    return i;
}
```
Uses arch_fastticks, fasthz 265a, todget() 262, and vlong2le() 268a.

⟨*function* writebintime 267a⟩≡ (696b)

```
/*
 *  set any of the following
 *  - time in nsec
 *  - nsec trim applied over some seconds
 *  - clock frequency
 */
static int
writebintime(char *buf, int n)
{
    byte *p;
    vlong delta;
    long period;

    n--;
    p = (byte*)buf + 1;
    switch(*buf){
    case 'n':
        if(n < sizeof(vlong))
            error(Ebadtimectl);
        le2vlong(&delta, p);
        todset(delta, 0, 0);
        break;
    case 'd':
        if(n < sizeof(vlong)+sizeof(long))
            error(Ebadtimectl);
        p = le2vlong(&delta, p);
        le2long(&period, p);
        todset(-1, delta, period);
        break;
    case 'f':
        if(n < sizeof(uvlong))
            error(Ebadtimectl);
        le2vlong(&fasthz, p);
        if(fasthz <= 0)
            error(Ebadtimectl);
        todsetfreq(fasthz);
        break;
    }
    return n;
}
```

Uses Ebadtimectl 260b, fasthz 265a, le2long() 268c, le2vlong() 267c, todset() 261b, and todsetfreq() 261a.

⟨*global* uvorder 267b⟩≡ (696b)

```
static uvlong uvorder = 0x0001020304050607ULL;
```

Uses uvorder-1799 267b.

⟨*function* le2vlong 267c⟩≡ (696b)

```
static byte*
le2vlong(vlong *to, byte *f)
{
    byte *t, *o;
    int i;

    t = (byte*)to;
    o = (byte*)&uvorder;
    for(i = 0; i < sizeof(vlong); i++)
        t[o[i]] = f[i];
    return f+sizeof(vlong);
}
```

Uses uvorder-1799 267b.

⟨*function* vlong2le 268a⟩≡ (696b)
```
static byte*
vlong2le(byte *t, vlong from)
{
    byte *f, *o;
    int i;

    f = (byte*)&from;
    o = (byte*)&uvorder;
    for(i = 0; i < sizeof(vlong); i++)
        t[i] = f[o[i]];
    return t+sizeof(vlong);
}
```
Uses uvorder-1799 267b.

⟨*global* order 268b⟩≡ (696b)
```
static long order = 0x00010203;
```
Uses order-1800 268b.

⟨*function* le2long 268c⟩≡ (696b)
```
static byte*
le2long(long *to, byte *f)
{
    byte *t, *o;
    int i;

    t = (byte*)to;
    o = (byte*)&order;
    for(i = 0; i < sizeof(long); i++)
        t[o[i]] = f[i];
    return f+sizeof(long);
}
```
Uses order-1800 268b.

10.13 Watchdog

A watchdog timer is a hardware safety net: if the kernel does not periodically "pet" it (reset its counter), the hardware assumes the system is hung and forces a reboot. On the Raspberry Pi, the watchdog is also used to trigger a deliberate reboot via `arch_reboot()`.

Chapter 11

Initialization

11.1 Booting the kernel (ARM)

The boot process has three stages: the Broadcom firmware loads the kernel binary at physical `0x8000`, the assembly startup code (`_start`) sets up a minimal environment (SVC mode, stack, basic page tables), and then `main()` in C initializes every kernel subsystem we have seen in the previous chapters—memory, processes, devices, timers —and creates the first user process.

11.1.1 Loading the kernel in physical memory (ARM)

11.1.2 First instructions (ARM)

The very first instructions set up the minimum needed to run C code: the static base register `R12` (adjusted for physical addresses since the MMU is still off), SVC mode with interrupts disabled, and a stack at the top of the `Cpu` structure at `CPUADDR`. Then it jumps to `armstart` to set up the page tables:

⟨*function* `_start` *(arm)* 269⟩≡ (990a)

```
/*
 * on bcm2836, only cpu0 starts here
 * other cpus enter at cpureset in armv7.s
 */
TEXT _start(SB), 1, $-4
    /*
     * load physical base for SB addressing while mmu is off
     * keep a handy zero in R0 until first function call
     */
    MOVW    $setR12(SB), R12
    SUB $KZERO, R12
    MOVW    $0, R0

    /*
     * SVC mode, interrupts disabled
     */
    MOVW    $(PsrDirq|PsrDfiq|PsrMsvc), R1
    MOVW    R1, CPSR

    /*
     * start stack at top of 'cpu' (physical addr)
     */
    MOVW    $PADDR(CPUADDR+CPUSIZE-4), R13

    /*
     * do arch-dependent startup (no return)
```

```
     */
    BL  armstart(SB)

    // Unreached
    B   0(PC)
    RET
```

11.1.3 Basic pagination (ARM)

Before calling C `main()`, `armstart` sets up a minimal page table so that the MMU can be enabled. It first disables the MMU and caches, clears the memory between `CPUADDR` and `KTZERO` (which will hold the `Cpu` struct and the L1/L2 tables), then calls `mmuinit()` to populate the L1 page directory with three sets of mappings:

```
1. RAM at KZERO:  pa 0 -> va KZERO  (Section entries)
2. Identity map:  pa 0 -> va 0      (so PC works after MMU on)
3. I/O at VIRTIO: pa soc.physio -> va VIRTIO
4. Vectors:       pa 0 -> va HVECTORS (via L2, Coarse entry)
```

After enabling the MMU and caches, `armstart` adjusts `R12` (SB) and `R13` (SP) to their virtual (KZERO) addresses and jumps to `_startpg`, which calls C `main()`.

⟨*function* `armstart` *(raspberry pi1)(arm)* 270⟩≡ (990b)

```
TEXT armstart(SB), 1, $-4

    /*
     * disable the mmu and L1 caches
     * invalidate caches and tlb
     */
    MRC CpSC, 0, R1, C(CpCONTROL), C(0), CpMainctl
    BIC $(CpCmmu), R1
    ⟨armstart() disable features(arm) 529c⟩
    MCR CpSC, 0, R1, C(CpCONTROL), C(0), CpMainctl

    ⟨armstart() invalidate caches(arm) 529d⟩
    MCR CpSC, 0, R0, C(CpTLB), C(CpTLBinvu), CpTLBinv
    ISB

    /*
     * clear mach and page tables
     */
    MOVW    $PADDR(CPUADDR), R1
    MOVW    $PADDR(KTZERO), R2
_ramZ:
    MOVW    R0, (R1)
    ADD $4, R1
    CMP R1, R2
    BNE _ramZ

    /*
     * start stack at top of CPUADDR (physical addr)
     * set up page tables for kernel
     */
    MOVW    $PADDR(CPUADDR+CPUSIZE-4), R13 // done already in start.s
    MOVW    $PADDR(L1), R0

    BL  mmuinit(SB)

    /*
```

```
     * set up domain access control and page table base
     */
    MOVW    $Client, R1
    MCR CpSC, 0, R1, C(CpDAC), C(0)

    MOVW    $PADDR(L1), R1
    MCR CpSC, 0, R1, C(CpTTB), C(0)

    /*
     * enable caches, mmu, and high vectors
     */
    MRC CpSC, 0, R0, C(CpCONTROL), C(0), CpMainctl
    ORR $(CpCmmu|CpChv), R0
⟨armstart() reenable features(arm) 529e⟩
    MCR CpSC, 0, R0, C(CpCONTROL), C(0), CpMainctl
    ISB

    /*
     * switch SB, SP, and PC into KZERO space
     */
    MOVW    $setR12(SB), R12
    MOVW    $(CPUADDR+CPUSIZE-4), R13

    MOVW    $_startpg(SB), R15
```

⟨constant L1(arm) 271a⟩≡ (1005)

```
#define L1          (KZERO+0x4000)      /* ttb ptes: 16KiB aligned */
```

⟨constant L2(arm) 271b⟩≡ (1005)

```
#define L2          (KZERO+0x3000)      /* L2 ptes for vectors etc */
```

⟨function mmuinit(arm) 271c⟩≡ (1006b)

```
/*
 * Set up initial PTEs for this cpu (called with mmu off)
 */
void
mmuinit(void *a)
{
    PTE *l1, *l2;
    phys_addr pa;
    kern_addr va;

    l1 = (PTE*)a; // PADDR(L1) for cpu0

    /*
     * map all of ram at KZERO
     */
    va = KZERO;
    for(pa = 0; pa < soc.dramsize; pa += MiB, va += MiB)
    {
        l1[L1X(va)] = pa|Dom0|L1AP(Krw)|Section|L1ptedramattrs;
    }

    /*
     * identity map first MB of ram so mmu can be enabled
     */
    // ???
    l1[L1X(0)] = 0|Dom0|L1AP(Krw)|Section|L1ptedramattrs;

    /*
```

```
     * map i/o registers
     */
    va = VIRTIO;
    for(pa = soc.physio; pa < soc.physio + IOSIZE; pa += MiB, va += MiB){
        // No L1ptedramattrs (Cached | Buffered) here, because volatile!
        l1[L1X(va)] = pa|Dom0|L1AP(Krw)|Section;
    }
    ⟨mmuinit() set armlocal virtual memory mapping(arm) 536i⟩

    /*
     * double map exception vectors at top of virtual memory
     */
    // ???
    l2 = (PTE*)PADDR(L2);
    va = HVECTORS;
    l1[L1X(va)] = (uintptr)l2|Dom0|Coarse;
    l2[L2X(va)] = L2AP(Krw)|Small|L2ptedramattrs;
}
```

Uses Coarse 178d, Dom0 179d, HVECTORS 39b, IOSIZE 45, Krw 179b, L1AP 179e, L1X-1842 178a, L1ptedramattrs-1845 530a, L2 271b, L2AP-1844 179i, L2X-1843 178b, L2ptedramattrs-1846 530b, MiB 48a, Section 178d, Small 178f, and VIRTIO 43a.

11.1.4 Jumping to C main() (ARM)

⟨*function* _startpg*(raspberry pi1)(arm)* 272a⟩≡ (990b)

```
TEXT _startpg(SB), 1, $-4

    ⟨_startpg() enable cycle counter(raspberry pi1)(arm) 508f⟩
    /*
     * call main and loop forever if it returns
     */
    BL  main(SB)

    // Unreached
    B   0(PC)
    BL  _div(SB)        /* hack to load _div, etc. */
```

11.2 Kernel initialization

`main()` initializes every kernel subsystem in the right order—a chicken-and-egg problem, since each subsystem depends on others. The sequence is roughly: (1) CPU and memory detection, (2) kernel memory allocator (`xinit`), (3) console output (`uartconsinit`, `screeninit`), (4) exception vectors and clock interrupts, (5) process and page allocators (`procinit`, `pageinit`), (6) device drivers (`chandevreset`), (7) first user process (`userinit`), (8) enter the scheduler (`schedinit`).

The ordering reflects hard dependencies. You cannot call `print()` until the UART is initialized, so console setup must come before any diagnostic output. You cannot allocate memory with `xalloc()` until `xinit()` has carved physical memory into kernel and user pools, so `xinit()` must precede `procinit()`, `pageinit()`, and `imageinit()`. Interrupts must be disabled (`arch__trapinit()`) before the clock is started (`clockinit()`), otherwise a timer interrupt could fire before the handler is installed. Device drivers (`chandevreset()`) come after the process and page allocators because some drivers allocate memory during `reset()`. And `userinit()` must be the very last step before `schedinit()`, because it calls `ready()` on the first process—if the scheduler ran before everything else was initialized, that process would crash.

⟨*function* main*(arm)* 272b⟩≡ (991c)

```
void
main(void)
```

```
{
    ⟨main() locals(arm) 286c⟩

    ⟨main() initial assignments for backward deps(arm) ??⟩

    ⟨main() clear bss(arm) 274a⟩

    // Let's go!

    cpu = (Cpu*)CPUADDR;
    arch__cpu0init(); // cpu0 initialization (calls arch__cpuinit())
    mmuinit1((void*)L1); // finish mmu initialization started in mmuinit0
    machon(0); // enable cpu0 for scheduling

    ⟨main() parsing options and boot arguments(arm) 276a⟩
    arch__confinit();     /* figures out amount of memory */
    xinit();

    uartconsinit();

    arch__screeninit(); // screenputs = swconsole_screenputs
    quotefmtinstall(); // libc printf initialization

    print("\nPlan 9 from Bell Labs\n"); // yeah!

    ⟨main() print board and firmware information(arm) 286d⟩
    /* set clock rate to arm_freq from config.txt (default pi1:700Mhz pi2:900MHz) */
    setclkrate(ClkArm, 0);

    arch__trapinit();
    clockinit(); // enable timer interrupts

    //TODO? kbdqinit(); // setup kbdq
    lineqinit(); // setup lineq

    timersinit(); // set initial timer for timer interrupt
    swcursor_init();

    arch_cpuidprint();
    archreset();

    procinit();
    imageinit();

    links();

    // initialize all devices
    chandevreset();         /* most devices are discovered here */

    pageinit(); // setup palloc.pages and swapalloc.highwater
    swapinit(); // setup swapalloc

    // let's craft our first process (that will then exec("boot/boot"))
    arch__userinit();

    ⟨main() start the other processors(arm) 539g⟩

    // schedule the only ready user process (the one created by userinit)
    schedinit();
```

```
    assert(0);          /* shouldn't have returned */
}
```

Uses ClkArm 896a, L1 271a, arch__confinit() 276d, arch__cpu0init() 274c, arch__userinit() 287d, assert 853f, chandevreset() 286a, clockinit() 280, imageinit() 285c, lineqinit() 389b, machon() 275a, mmuinit1() 281c, pageinit() 283c, print, procinit() 283a, schedinit() 224f, setclkrate() 896c, swapinit() 284b, swcursor_init(), timersinit() 281b, uartconsinit() 968a, and xinit() 282.

⟨main() *clear bss(arm)* 274a⟩≡ (272b)

```
memset(edata, 0, end - edata);  /* clear bss */
```

⟨*lib.h exxx decl* 274b⟩≡ (853i)

```
extern  char  etext[];
//@Scheck: Assembly, not dead used by 386/l.s and arm/main.c
extern  char  edata[];
extern  char  end[];
```

11.2.1 cpu initialization (ARM)

arch__cpu0init() initializes the boot CPU's Cpu structure: sets cpuno to 0, registers it in the cpus array, and clears up (no process running yet). machon() then marks this CPU as active in the Active bitmap so the scheduler knows it exists.

Notice that conf.ncpu is set to 0 here and only incremented inside machon(), which is called separately after mmuinit1(). This separation matters: the CPU structure must be fully initialized before calling mmuinit1() (which stores the L1 page table pointer in cpu->mmul1), and the CPU must only be marked as schedulable after the MMU setup is complete. Setting up to nil is critical—any trap or interrupt before userinit() creates the first process would see that no process is running.

⟨*function* arch__cpu0init*(arm)* 274c⟩≡ (991c)

```
void
arch__cpu0init(void)
{
    conf.ncpu = 0; // set in machon() instead (machon() is called after cpuinit)

    cpu->cpuno = 0;
    cpus[cpu->cpuno] = cpu;

    arch__cpuinit();
    active.exiting = false;

    up = nil;
}
```

Uses active 65e, arch__cpuinit() 274d, conf 62b, cpus 65a, false, and up.

⟨*function* arch__cpuinit*(arm)* 274d⟩≡ (991c)

```
void
arch__cpuinit(void)
{
    Cpu *cpu0;

    cpu->ticks = 1;
    cpu->perf.period = 1;

    cpu0 = CPUS(0);
    ⟨arch__cpuinit() it not cpu0 544b⟩
}
```

Uses CPUS 702.

`machon()` is the gate that makes a CPU visible to the scheduler. It sets the CPU's bit in `active.cpus` and increments `conf.ncpu` under a lock—the lock matters for SMP, where secondary CPUs call `machon()` concurrently after their own initialization.

⟨*function* `machon` *(arm)* 275a⟩≡ (991c)
```
/* enable scheduling of this cpu */
void
machon(uint xcpu)
{
    ulong cpubit;

    cpubit = 1 << xcpu;
    lock(&active);
    if ((active.cpus & cpubit) == 0) {  /* currently off? */
        conf.ncpu++;
        active.cpus |= cpubit;
    }
    unlock(&active);
}
```
Uses `active` 65e, `conf` 62b, `lock()` 76d, and `unlock()` 78d.

⟨*function* `arch_cpuidprint` *(raspberry pi1)(arm)* 275b⟩≡ (994b)
```
void
arch_cpuidprint(void)
{
    char name[64];

    cputype2name(name, sizeof name);
    arch_delay(50);             /* let uart catch up */
    print("cpu%d: %dMHz ARM %s\n", cpu->cpuno, cpu->cpumhz, name);
}
```
Uses `arch_delay` and `print`.

⟨*function* `cputype2name` *(raspberry pi1)(arm)* 275c⟩≡ (994b)
```
char*
cputype2name(char *buf, int size)
{
    seprint(buf, buf + size, "1176JZF-S");
    return buf;
}
```

11.2.2 Memory detection (ARM)

The Raspberry Pi does not have a BIOS memory map like x86. Instead, `getramsize()` queries the Broadcom GPU via the mailbox interface (`vcreq`) to find the base and limit of ARM-accessible RAM:

⟨*function* `getramsize` *(arm)* 275d⟩≡ (897b)
```
/*
 * Get ARM ram
 */
void
getramsize(Confmem *mem)
{
    u32int buf[2];

    if(vcreq(TagGetram, buf, 0, sizeof buf) != sizeof buf)
        return;
    mem->base = buf[0];
    mem->limit = buf[1];
```

```
}
```
Uses TagGetram-2364 893d and vcreq() 895a.

11.2.3 conf initialization (ARM)

arch__confinit() determines how much physical memory is available (via the Broadcom mailbox interface), splits it into kernel and user portions (80% user by default), and computes the sizes of all the large kernel data structures: the Proc array, Page array, KImage array, swap map, and malloc pools.

The function works in two phases. First, it carves the kernel image out of the physical memory map: pa is set to the physical address just past end (the linker symbol marking the end of BSS), and any memory region containing the kernel has its base bumped past pa. This ensures the kernel's own code and data are never handed out by xalloc().

Second, it computes the sizes of every large array the kernel will allocate. The formulas are empirical: 5 extra process slots per megabyte of RAM, 3 swap slots per physical page, and a fixed 200 image cache headers. The key subtlety is the kmem calculation at the end: after reserving 80% of pages for users, the remaining 20% must fit all the kernel arrays (Page, Proc, KImage, swap map) plus the malloc heap. The function subtracts the array sizes from kmem to find how much is left for mainmem and imagmem—the two memory pools used by malloc() and smalloc().

⟨main() *parsing options and boot arguments(arm)* 276a⟩≡ (272b)
```
//optionsinit("/boot/boot boot"); // setup values for getconf()
//ataginit((Atag*)BOOTARGS); // from bootloader config
// example of manual config:
// TODO confname(``console'') = 1?
```

⟨Conf *other fields* 276b⟩+≡ (62a) ◁208a 459e▷
```
ulong npage;    /* total physical pages of memory */

ulong upages;   /* user page pool */
// kpages = npage - upages

ulong nimage;   /* number of page cache image headers */
ulong ialloc;   /* max interrupt time allocation in bytes */
```

⟨*global* memsize*(arm)* 276c⟩≡ (991c)
```
// used also in mmu.c
ulong   memsize = 128*1024*1024;
```
Uses memsize 276c.

⟨*function* arch__confinit*(arm)* 276d⟩≡ (991c)
```
void
arch__confinit(void)
{
    int i;
    char *p;
    phys_addr pa;
    ulong kpages;
    ulong kmem;

    ⟨arch__confinit() if maxmem boot parameter(arm) 277⟩
    // simpler than for x86 :)
    getramsize(&conf.mem[0]);
    ⟨arch__confinit() sanity check conf.mem[0](arm) 278a⟩

    conf.npage = 0;
    pa = PADDR(PGROUND(PTR2UINT(end)));
```

```
    /*
     *  we assume that the kernel is at the beginning of one of the
     *  contiguous chunks of memory and fits therein.
     */
    for(i=0; i<nelem(conf.mem); i++){
        /* take kernel out of allocatable space */
        if(pa > conf.mem[i].base && pa < conf.mem[i].limit)
            conf.mem[i].base = pa;

        conf.mem[i].npage = (conf.mem[i].limit - conf.mem[i].base)/BY2PG;
        conf.npage += conf.mem[i].npage;
    }

    conf.upages = (conf.npage*80)/100;
    kpages = conf.npage - conf.upages;

    /* set up other configuration parameters */
    conf.ialloc = (kpages/2)*BY2PG; // max bytes for iallocb

    conf.nproc = 100 + ((conf.npage*BY2PG)/MB)*5;
    if(cpuserver)
        conf.nproc *= 3;
    if(conf.nproc > 2000)
        conf.nproc = 2000;

    conf.nswap = conf.npage*3;
    conf.nswppo = 4096;
    conf.nimage = 200;

    conf.copymode = 1;      /* copy on reference, not copy on write */

    /*
     * Guess how much is taken by the large permanent
     * datastructures. Mntcache and Mntrpc are not accounted for
     * (probably ~300KB).
     */
    kmem = kpages * BY2PG;
    kmem -=
          conf.upages * sizeof(Page)
        + conf.nproc  * sizeof(Proc)
        + conf.nimage * sizeof(KImage)
        + conf.nswap
        + conf.nswppo * sizeof(Page*); // pad's second bugfix :)

    // memory pool
    mainmem->maxsize = kmem;

    if(!cpuserver)
        /*
         * give terminals lots of image memory, too; the dynamic
         * allocation will balance the load properly, hopefully.
         * be careful with 32-bit overflow.
         */
        imagmem->maxsize = kmem;
}
```

Uses MB 867a, PGROUND 120d, PTR2UINT 56g, conf 62b, getramsize() 275d, imagmem 102b, mainmem 102b, and nelem 853i.

⟨arch__confinit() *if maxmem boot parameter(arm)* 277⟩≡ (276d)

```
if((p = getconf("*maxmem")) != nil){
```

```
    memsize = strtoul(p, 0, 0);
    if (memsize < 16*MB)        /* sanity */
        memsize = 16*MB;
}
```

Uses MB 867a and memsize 276c.

⟨arch__confinit() *sanity check* conf.mem[0]*(arm)* 278a⟩≡ (276d)

```
if(conf.mem[0].limit == 0){
    conf.mem[0].base = 0;
    conf.mem[0].limit = memsize;
}
⟨arch__confinit() if maxmem parameter and limit mem is zero(arm) 278b⟩
```

Uses conf 62b and memsize 276c.

⟨arch__confinit() *if maxmem parameter and limit mem is zero(arm)* 278b⟩≡ (278a)

```
else if(p != nil)
    conf.mem[0].limit = conf.mem[0].base + memsize;
```

Uses conf 62b and memsize 276c.

11.2.4 Exception vector initialization (ARM)

arch__trapinit() copies the vectors and vtable code (from the Interrupts chapter) to HVECTORS (0xFFFF0000), disables all interrupts, and sets up scratch stacks for each exception mode (IRQ, ABT, UND) so that _vxxx handlers have somewhere to save R0–R4.

ARM has a fixed exception vector table at either address 0 or 0xFFFF0000 (high vectors, selected by the CpChv bit set during boot). The Vpage0 structure mirrors the hardware layout: 8 branch instructions followed by 8 target addresses. memmove copies both from the linker-placed originals into the high-vector page, then a cache flush ensures the instruction cache sees the new code. The intrsoff() call writes all-ones to the Broadcom interrupt controller's disable registers, masking every interrupt source—they will be selectively re-enabled later as individual drivers register handlers via arch_intrenable().

⟨*struct* Vpage0*(arm)* 278c⟩≡ (1002c)

```
/*
 *   Layout at virtual address KZERO (double mapped at HVECTORS).
 */
struct Vpage0 {
    void    (*vectors[Nvec])(void);
    u32int  vtable[Nvec];
};
```

Uses Nvec-2406 278d.

⟨*constant* Nvec*(arm)* 278d⟩≡ (1002b)

```
Nvec = 8,       /* # of vectors at start of lexception.s */
```

⟨*function* arch__trapinit*(arm)* 278e⟩≡ (1002c)

```
/*
 *  set up for exceptions
 */
void
arch__trapinit(void)
{
    Vpage0 *vpage0;

    if (cpu->cpuno == 0) {
        /* disable everything */
        intrsoff();
        /* set up the exception vectors */
```

```
        vpage0 = (Vpage0*)HVECTORS;
        memmove(vpage0->vectors, vectors, sizeof(vpage0->vectors));
        memmove(vpage0->vtable, vtable, sizeof(vpage0->vtable));
        ⟨arch__trapinit() invalidate cache after adjust vectors(arm) 531k⟩
    }

    /* set up the stacks for the interrupt modes */
    ⟨arch__trapinit() set stack for other exception/processor-modes 279b⟩
    ⟨arch__trapinit() coherence(arm) 534e⟩
}
```

Uses HVECTORS 39b and intrsoff() 279a.

⟨*function* intrsoff*(arm)* 279a⟩≡ (1002c)

```
void
intrsoff(void)
{
    Intregs *ip;
    int disable;

    ip = (Intregs*)INTREGS;
    disable = ~0;
    ip->GPUdisable[0] = disable;
    ip->GPUdisable[1] = disable;
    ip->ARMdisable = disable;
    ip->FIQctl = 0;
}
```

Uses INTREGS-2403 43d.

ARM has banked R13 (SP) registers: each processor mode (IRQ, ABT, UND, SVC) has its own private R13. When an interrupt fires, the CPU switches to IRQ mode and uses IRQ's R13—if it was never initialized, the handler would corrupt random memory. setr13() temporarily switches to each mode, installs the 5-word scratch area from the Cpu structure as that mode's R13, then switches back. The SVC stack was already set up by _start.

⟨arch__trapinit() *set stack for other exception/processor-modes* 279b⟩≡ (278e) 560f▷

```
setr13(PsrMirq, cpu->sirq);
setr13(PsrMabt, cpu->sabt);
setr13(PsrMund, cpu->sund);
```

Uses PsrMabt 161a, PsrMirq 161a, and PsrMund 161a.

⟨Cpu Arch *save of first five registers fields(arm)* 279c⟩≡ (64e)

```
/* save areas for exceptions, hold R0-R4 */
u32int  sirq[5];
u32int  sund[5];
u32int  sabt[5];
```

⟨*function* setr13*(arm)* 279d⟩≡ (1004a)

```
/*
 *  set the stack value for the mode passed in R0
 */
TEXT setr13(SB), 1, $-4
    MOVW    4(FP), R1

    MOVW    CPSR, R2
    BIC $PsrMask, R2, R3
    ORR $(PsrDirq|PsrDfiq), R3
    ORR R0, R3
    MOVW    R3, CPSR        /* switch to new mode */
```

```
MOVW    R13, R0         /* return old sp */
MOVW    R1, R13         /* install new one */

MOVW    R2, CPSR        /* switch back to old mode */
RET
```

11.2.5 Clock initialization (ARM)

`clockinit()` measures the CPU frequency by timing cycles against the 1 MHz system timer, then registers `clockintr` as the handler for `IRQtimer3`. After this, the hardware will generate an interrupt when system timer 3's compare register matches the counter.

The frequency measurement is a neat calibration trick. The Broadcom system timer runs at a known 1 MHz, so the code waits for the counter to advance by exactly 10,000 ticks (10 ms) and counts how many CPU cycles elapse during that window. Since 10,000 microseconds is 1/100th of a second, multiplying the cycle count by 100 gives `cpuhz`. This is necessary because the ARM clock frequency can be set by `config.txt` and is not a compile-time constant.

Note that `clockinit()` only enables the hardware interrupt source—it does not start the periodic tick. That happens in `timersinit()`, which creates a `Timer` structure and calls `timeradd()` to program the compare register. The two-step design separates hardware setup (architecture-specific) from the timer policy (portable code).

⟨*function* `clockinit` *(arm)* 280⟩≡ (1011b)

```
void
clockinit(void)
{
    Systimers *tn;
    Armtimer *tm;
    u32int t0, t1, tstart, tend;

    ⟨clockinit() if many processors 547c⟩

    tn = (Systimers*)SYSTIMERS;
    tstart = tn->clo;
    do{
        t0 = arch_lcycles();
    }while(tn->clo == tstart);
    tend = tstart + 10000;
    do{
        t1 = arch_lcycles();
    }while(tn->clo != tend);
    t1 -= t0;

    cpu->cpuhz = 100 * t1;
    cpu->cpumhz = (cpu->cpuhz + Mhz/2 - 1) / Mhz;

    cpu->cyclefreq = cpu->cpuhz;

    if(cpu->cpuno == 0){
        tn->c3 = tn->clo - 1;

        tm = (Armtimer*)ARMTIMER;
        tm->load = 0;
        tm->ctl = TmrPrescale1|CntEnable|CntWidth32;

        arch_intrenable(IRQtimer3, clockintr, nil, 0, "clock");
    }
    ⟨clockinit() if not cpu0 547d⟩
```

```
}
```

Uses ARMTIMER-2648 43c, CntEnable-2657 1011a, CntWidth32-2665 1011a, Mhz 874a, SYSTIMERS-2647 43b, and TmrPrescale1-2661 1011a.

11.2.6 Timer initialization

`timersinit()` creates the periodic timer that drives `hzclock()`. The timer's `tf` callback is set to `nil`, which `timerintr()` interprets as "call `hzclock()`". This is the timer that makes the kernel tick:

⟨*global* `timersinited` 281a⟩≡ (874c)

```
static bool timersinited;
```

⟨*function* `timersinit` 281b⟩≡ (874c)

```
void
timersinit(void)
{
    Timer *t;

    timersinited = true;
    todinit();

    t = malloc(sizeof(Timer));
    if(t == nil)
        error(Enomem);
    t->tmode = Tperiodic;
    t->tt = nil;
    t->tns = 1000000000 / Arch_HZ;
    /*
     * T->tf == nil means the HZ clock for this processor.
     */
    t->tf = nil;
    timeradd(t);
}
```

Uses `Tperiodic` 248a, `malloc()` 103c, `timeradd()` 250b, `timersinited-2646` 281a, `todinit()` 260d, and `true`.

11.2.7 Virtual memory initialisation (ARM)

`mmuinit1()` finishes what `mmuinit()` started: it records the L1 table in `cpu->mmul1` and removes the identity mapping of physical address 0 that was needed only while the MMU was being enabled. The identity map was a bootstrapping trick: when the MMU is first turned on, the next instruction fetch uses virtual addresses, so the code must be mapped at its current physical address too. Now that we are running in KZERO space, the identity map is a security hole—clearing it ensures that virtual address 0 is unmapped, so null pointer dereferences fault.

⟨*function* `mmuinit1`*(arm)* 281c⟩≡ (1006b)

```
void
mmuinit1(void *a)
{
    PTE *l1;

    l1 = (PTE*)a;
    cpu->mmul1 = l1;

    /*
     * undo identity map of first MB of ram
     */
    l1[L1X(0)] = 0;
    ⟨mmuinit1() write back cache after adjusting page tables(arm) 531d⟩
```

```
    mmuinvalidateaddr(0);
}
```

Uses L1X-1842 178a.

11.2.8 Memory initialisation

The memory initialization functions allocate the large kernel data structures using `xalloc` from the Memory chapter. `xinit()` splits physical memory between the kernel (`xhole`) and user pages (`palloc.mem`). Then `procinit()`, `pageinit()`, `swapinit()`, and `imageinit()` each allocate their respective arrays.

The split works as follows. `arch__confinit()` already decided that 20% of pages go to the kernel and 80% to users. `xinit()` walks the `conf.mem` array and for each contiguous region, feeds the first `kpages` worth to `xhole()` (the kernel free list), and records the rest in `palloc.mem` (the user page pool). After `xinit()`, `xalloc()` can hand out kernel memory, and `procinit()`, `pageinit()`, `swapinit()`, and `imageinit()` each call `xalloc()` to grab their arrays. `pageinit()` then threads the user pages into `palloc`'s doubly-linked free list and prints the familiar "128M memory: 26M kernel data, 102M user, 307M swap" diagnostic.

`xinit()`

⟨*function* `xinit` 282⟩≡ (859c)

```
void
xinit(void)
{
    int i;
    int nkpages;
    int kpages;
    ulong maxpages;
    Confmem *m;
    Pallocmem *pm;
    Hole *h, *eh;

    eh = &xlists.hole[Nhole-1];
    for(h = xlists.hole; h < eh; h++)
        h->next = h+1;

    xlists.unused_slots = xlists.hole;

    kpages = conf.npage - conf.upages;

    pm = palloc.mem;
    for(i=0; i<nelem(conf.mem); i++){
        m = &conf.mem[i];
        nkpages = m->npage;
        if(nkpages > kpages)
            nkpages = kpages; // will be zero once kpages has been filled
        /* don't try to use non-KADDR-able memory for kernel */
        maxpages = arch_cankaddr(m->base)/BY2PG;
        if(nkpages > maxpages)
            nkpages = maxpages;

        /* first give to kernel */
        if(nkpages > 0){
            xhole(m->base, nkpages*BY2PG);
            kpages -= nkpages;
            ⟨xinit() nkpages kernel memory in m 488b⟩
        }
```

```
        /* if anything left over, give to user */
        if(m->npage > nkpages){
            if(pm >= palloc.mem+nelem(palloc.mem)){
                print("xinit: losing %lud pages\n", m->npage-nkpages);
                continue;
            }
            pm->base = m->base+nkpages*BY2PG;
            pm->npage = m->npage - nkpages;
            pm++;
        }
    }
}
```

Uses Nhole 855, conf 62b, nelem 853i, palloc 109b, print, xhole() 97, and xlists-1849 95a.

procinit()

⟨*function* procinit 283a⟩≡ (877c)

```
void
procinit(void)
{
    Proc *p;
    int i;

    procalloc.free = xalloc(conf.nproc * sizeof(Proc));
    ⟨procinit() sanity check procalloc.free 283b⟩
    procalloc.arena = procalloc.free;

    p = procalloc.free;
    for(i=0; i<conf.nproc-1; i++,p++)
        p->qnext = p+1;
    p->qnext = nil;
}
```

Uses conf 62b, procalloc-2458 128d, and xalloc() 95d.

⟨procinit() *sanity check* procalloc.free 283b⟩≡ (283a)

```
if(procalloc.free == nil){
    xsummary();
    panic("cannot allocate %lud procs (%ludMB)\n",
                conf.nproc, conf.nproc*sizeof(Proc)/MB);
}
```

Uses procalloc-2458 128d and xsummary() 643a.

pageinit()

⟨*function* pageinit 283c⟩≡ (863d)

```
void
pageinit(void)
{
    int i, j;
    Page *p;
    Pallocmem *pm;
    ulong m, np, k, vkb, pkb;
    ⟨pageinit() other locals 554b⟩

    np = 0;
    for(i=0; i<nelem(palloc.mem); i++){
        pm = &palloc.mem[i];
        np += pm->npage;
```

```
    }

    palloc.pages = xalloc(np * sizeof(Page));
    ⟨pageinit() sanity check palloc.pages 284a⟩

    palloc.head = palloc.pages;
    p = palloc.head;
    for(i=0; i<nelem(palloc.mem); i++){
        pm = &palloc.mem[i];
        for(j=0; j < pm->npage; j++){
            p->prev = p-1;
            p->next = p+1;
            p->pa = pm->base + j*BY2PG;
            palloc.freecount++;
            ⟨pageinit() set page color 554c⟩
            p++;
        }
    }
    palloc.tail = p - 1;
    palloc.head->prev = nil;
    palloc.tail->next = nil;

    palloc.user = p - palloc.pages;
    assert(palloc.user == np);

    pkb = palloc.user*BY2PG/KB;
    vkb = pkb + (conf.nswap*BY2PG)/KB;

    /* Paging numbers */
    swapalloc.highwater = (palloc.user*5)/100;
    swapalloc.headroom = swapalloc.highwater + (swapalloc.highwater/4);

    // diagnostic
    m = 0;
    for(i=0; i<nelem(conf.mem); i++)
        if(conf.mem[i].npage)
            m += conf.mem[i].npage*BY2PG;
    k = PGROUND(end - (char*)KTZERO);
    print("%ldM memory: ", (m+k+1024*1024-1)/(1024*1024));
    print("%ldM kernel data, ", (m+k-pkb*1024+1024*1024-1)/(1024*1024));
    print("%ldM user, ", pkb/1024);
    print("%ldM swap\n", vkb/1024);
}
```

Uses KB 867a, PGROUND 120d, assert 853f, conf 62b, nelem 853i, palloc 109b, print, swapalloc 204b, and xalloc() 95d.

⟨pageinit() *sanity check* palloc.pages 284a⟩≡ (283c)

```
if(palloc.pages == nil)
    panic("pageinit");
```

Uses palloc 109b.

swapinit()

⟨*function* swapinit 284b⟩≡ (866b)

```
void
swapinit(void)
{
    swapalloc.swmap = xalloc(conf.nswap);

    swapalloc.top = &swapalloc.swmap[conf.nswap];
```

```
    swapalloc.alloc = swapalloc.swmap;
    swapalloc.last = swapalloc.swmap;
    swapalloc.free = conf.nswap;

    iolist = xalloc(conf.nswppo*sizeof(Page*));

    ⟨swapinit() sanity check xalloc return values 285a⟩
    swapimage.notext = true;
}
```

Uses conf 62b, iolist-1807 208b, swapalloc 204b, swapimage 205b, true, and xalloc() 95d.

⟨swapinit() *sanity check xalloc return values* 285a⟩≡ (284b)

```
if(swapalloc.swmap == nil || iolist == nil)
    panic("swapinit: not enough memory");
```

Uses iolist-1807 208b and swapalloc 204b.

imageinit()

⟨*constant* NFREECHAN 285b⟩≡ (855)

```
#define NFREECHAN 64
```

⟨*function* imageinit 285c⟩≡ (864b)

```
void
imageinit(void)
{
    KImage *i, *ie;

    imagealloc.free = xalloc(conf.nimage * sizeof(KImage));
    ⟨imageinit() sanity check imagealloc.free 285d⟩
    ie = &imagealloc.free[conf.nimage-1];
    for(i = imagealloc.free; i < ie; i++)
        i->next = i+1;
    i->next = nil;
    imagealloc.freechan = malloc(NFREECHAN * sizeof(Chan*));
    imagealloc.szfreechan = NFREECHAN;
}
```

Uses NFREECHAN 194a, conf 62b, imagealloc-1803 194b, malloc() 103c, and xalloc() 95d.

⟨imageinit() *sanity check* imagealloc.free 285d⟩≡ (285c)

```
if (imagealloc.free == nil)
    panic("imageinit: no memory");
```

Uses imagealloc-1803 194b.

11.2.9 Device initialisation

`chandevreset()` iterates over `devtab` and calls each device driver's `reset()` method. This is where most hardware is discovered and initialized: the USB controller, SD card, Ethernet, console, etc. A second pass via `chandevinit()` (called later from `init0()`) runs each driver's `init()` method in a process context.

The two-pass design (`reset` then `init`) exists because `reset()` runs during `main()` before any process exists—it cannot sleep or use `waserror()`. `init()` runs later inside `init0()`, where there is a real process

context with a kernel stack and error labels. Drivers that need to sleep during initialization (e.g., waiting for USB enumeration) must defer that work to `init()`.

⟨*function* `chandevreset` 286a⟩≡ (808c)
```
void
chandevreset(void)
{
    int i;

    todinit();  /* avoid later reentry causing infinite recursion */

    for(i=0; devtab[i] != nil; i++) {
        print("reset %d, %s\n", i, devtab[i]->name);
        devtab[i]->reset();
    }
}
```
Uses `print` and `todinit()` 260d.

⟨`Dev` *methods* 286b⟩≡ (71b) 290a ▷
```
void  (*reset)(void); // done once at boot time
```

11.2.10 Displaying hardware information (ARM)

⟨`main()` *locals(arm)* 286c⟩≡ (272b)
```
uint firmware, board;
```

⟨`main()` *print board and firmware information(arm)* 286d⟩≡ (272b)
```
board = getboardrev();
firmware = getfirmware();
print("board rev: %#ux firmware rev: %d\n", board, firmware);
if(firmware < Minfirmrev){
    print("Sorry, firmware (start*.elf) must be at least rev %d"
          " or newer than %s\n", Minfirmrev, Minfirmdate);
    for(;;)
        ;
}
```
Uses `Minfirmdate-2432` 286f, `Minfirmrev-2431` 286e, `getboardrev()` 286g, `getfirmware()` 287a, and `print`.

⟨*constant* `Minfirmrev`*(arm)* 286e⟩≡ (991c)
```
/* Firmware compatibility */
#define Minfirmrev  326770
```

⟨*constant* `Minfirmdate`*(arm)* 286f⟩≡ (991c)
```
#define Minfirmdate "19 Aug 2013"
```

⟨*function* `getboardrev`*(arm)* 286g⟩≡ (897b)
```
/*
 * Get board revision
 */
uint
getboardrev(void)
{
    u32int buf[1];

    if(vcreq(TagGetrev, buf, 0, sizeof buf) != sizeof buf)
        return 0;
    return buf[0];
}
```
Uses `TagGetrev-2362` 893d and `vcreq()` 895a.

⟨*function* `getfirmware` *(arm)* 287a⟩≡ (897b)

```
/*
 * Get firmware revision
 */
uint
getfirmware(void)
{
    u32int buf[1];

    if(vcreq(TagGetfwrev, buf, 0, sizeof buf) != sizeof buf)
        return 0;
    return buf[0];
}
```

Uses `TagGetfwrev-2361` 893d and `vcreq()` 895a.

11.3 The first process (ARM)

`arch__userinit()` creates the first process by hand—there is no parent to `rfork` from. It allocates a `Proc`, creates a text segment containing `initcode` (a tiny assembly program that calls `exec("/boot/boot")`), creates a stack segment, sets `sched.pc` to `init0` (the kernel-side startup code), and calls `ready()`. When `schedinit()` picks this process, it jumps to `init0`, which sets up the root namespace (`/`), initializes environment variables, starts the alarm kernel process, and finally calls `arch_touser()` to drop into user mode and execute `initcode`.

This is the most delicate piece of the boot sequence, because the first process has no parent to inherit from. Every subsequent process is created by `rfork`, which copies segments, file descriptors, and namespace from a parent. Here, everything must be built from scratch: a fresh process group, environment group, file descriptor group, and rendez group. The process gets two segments—a text segment with the hand-assembled `initcode` and a stack segment—and two program counters: `sched.pc` points to `init0` (the kernel-side setup that will run first), and the user-mode entry point is `UTZERO+0x20` (set by `arch_touser()`).

⟨*global* `sp` *(arm)* 287b⟩≡ (991c)

```
static user_addr sp;      /* XXX - must go - user stack of init proc */
```

⟨*macro* `STACKALIGN` *(arm)* 287c⟩≡ (1005)

```
#define STACKALIGN(sp)  ((sp) & ~3)     /* bug: assure with alloc */
```

⟨*function* `arch__userinit` *(arm)* 287d⟩≡ (991c)

```
/*
 *  create the first process
 */
void
arch__userinit(void)
{
    Proc *p;
    Segment *s;
    Arch_KMap *k;
    Page *pg;

    /* no processes yet */
    up = nil;

    p = newproc();
    p->pgrp = newpgrp();
    p->egrp = smalloc(sizeof(Egrp));
    p->egrp->ref = 1;
    p->fgrp = dupfgrp(nil);
    p->rgrp = newrgrp();
```

```
    p->procmode = 0640;

    kstrdup(&eve, "");
    kstrdup(&p->text, "*init*");
    kstrdup(&p->user, eve);

    /*
     * Kernel Stack
     */
    p->sched.pc = PTR2UINT(init0);
    p->sched.sp = PTR2UINT(p->kstack + KSTACK
                           - sizeof(up->sargs.args)
                           - sizeof(uintptr));
    p->sched.sp = STACKALIGN(p->sched.sp);

    /*
     * User Stack
     *
     * Technically, newpage can't be called here because it
     * should only be called when in a user context as it may
     * try to sleep if there are no pages available, but that
     * shouldn't be the case here.
     */
    s = newseg(SG_STACK, USTKTOP-USTKSIZE, USTKSIZE/BY2PG);
    ⟨arch__userinit() set flushme for stack segment 555g⟩
    p->seg[SSEG] = s;
    pg = newpage(true, nil, USTKTOP-BY2PG);
    segpage(s, pg);

    k = arch_kmap(pg);
    sp = bootargs(VA(k));
    arch_kunmap(k);

    /*
     * Text
     */
    s = newseg(SG_TEXT, UTZERO, 1); // initcode needs only 1 page
    p->seg[TSEG] = s;
    pg = newpage(true, nil, UTZERO);
    ⟨arch__userinit() set cachectl(arm) 555d⟩
    segpage(s, pg);

    k = arch_kmap(s->pagedir[0]->pagetab[0]);
    memmove(UINT2PTR(VA(k)), initcode, sizeof initcode);
    arch_kunmap(k);

    // ready to go!
    ready(p);
}
```

Uses PTR2UINT 56g, SG_STACK 112b, SG_TEXT, SSEG 118c, STACKALIGN 287c, TSEG 118c, UINT2PTR 56h, dupfgrp() 298c, eve 591a, initcode, kstrdup() 669b, newpgrp() 347d, newproc() 130a, newrgrp() 479d, newseg() 116a, ready, segpage() 118a, smalloc() 104b, sp-2433 287b, true, and up.

`init0()` is the kernel-side half of the first process. When the scheduler picks the process created by `userinit()`, it "returns" to `init0` via the saved `sched.pc`. The function runs in kernel mode with interrupts initially disabled. Its job is to build the minimal environment that `startboot()` will need: a root namespace (`up->slash`), device driver second-phase initialization (`chandevinit()`), environment variables like `cputype` and `service`, and the `alarmkproc` kernel thread. Only then does `arch_touser(sp)` fake a return-from-interrupt to drop into user mode at `UTZERO+0x20`, where `initcode` calls `exec("/boot/boot")`.

⟨*function* `init0(arm)` 289a⟩≡ (991c)

```
/*
 *  starting place for first process
 */
void
init0(void)
{
    int i;
    char buf[2*KNAMELEN]; // has to be the same than in ksetenv?

    up->nerrlab = 0;

    ⟨init0() coherence(arm) 534f⟩
    arch_spllo(); // Allow interrupts!!!

    /*
     * These are o.k. because rootinit is null.
     * Then early kproc's will have a root and dot.
     */
    up->slash = namec("#/", Atodir, 0, 0);
    pathclose(up->slash->path);
    up->slash->path = newpath("/");
    up->dot = cclone(up->slash);

    chandevinit();

    if(!waserror()){
        snprint(buf, sizeof(buf), "%s %s", "ARM", conffile);
        ksetenv("terminal", buf, 0);
        ksetenv("cputype", "arm", 0); // used by mkfile!
        if(cpuserver)
            ksetenv("service", "cpu", 0);
        else
            ksetenv("service", "terminal", 0);
        snprint(buf, sizeof(buf), "-a %s", getethermac());
        ksetenv("etherargs", buf, 0);

        /* convert plan9.ini variables to #e and #ec */
        for(i = 0; i < nconf; i++) {
            ksetenv(confname[i], confval[i], 0);
            ksetenv(confname[i], confval[i], 1);
        }
        poperror();
    }
    kproc("alarm", alarmkproc, nil); // ??

    // !! Go to user space !!
    arch_touser(sp);

    assert(0);          /* shouldn't have returned */
}
```

Uses `Atodir` 305b, `KNAMELEN` 853b, `alarmkproc()` 258e, `assert` 853f, `cclone()` 332a, `chandevinit()` 289b, `confname` 63a, `confval` 63a, `getethermac()` 895b, `kproc()` 157c, `ksetenv()` 598b, `namec()` 305f, `nconf` 63c, `newpath()` 324f, `pathclose()` 325c, `poperror` 666d, `sp-2433` 287b, `up`, and `waserror` 666d.

⟨*function* `chandevinit` 289b⟩≡ (808c)

```
void
chandevinit(void)
{
    int i;
```

```
    for(i=0; devtab[i] != nil; i++) {
        devtab[i]->init();
    }
}
```

⟨Dev *methods* 290a⟩+≡ (71b) ◁286b 304a▷
```
void  (*init)(void);
```

`arch_touser()` is the ceremony that crosses the kernel/user boundary for the very first time. It cannot simply branch to user code, because the CPU must switch from SVC mode to USR mode and simultaneously set the user stack pointer. The trick is to fake a return-from-interrupt: it loads `sp` into the user-mode `R13` (via the `.S` suffix which accesses banked registers), sets `SPSR` to user mode, pushes `UTZERO+0x20` as the return address, and executes `RFE`—which atomically loads `PC` from the stack and `CPSR` from `SPSR`, dropping to user mode. The `0x20` offset skips the Plan 9 a.out header.

⟨*function* `arch_touser` *(arm)* 290b⟩≡ (1010a)
```
/*
 *  This is the first jump from kernel to user mode.
 *  Fake a return from interrupt.
 *
 *  Enter with R0 containing the user stack pointer.
 *  UTZERO + 0x20 is always the entry point.
 *
 */
TEXT arch_touser(SB), 1, $-4
    /* store the user stack pointer into the USR_r13 */
    MOVM.DB.W [R0], (R13)
    /* avoid the ambiguity described in notes/movm.w. */
//  MOVM.S.IA.W (R13), [R13]
    MOVM.S  (R13), [R13]
    ADD $4, R13

    /* set up a PSR for user level */
    MOVW    $(PsrMusr), R0
    MOVW    R0, SPSR

    /* save the PC on the stack */
    MOVW    $(UTZERO+0x20), R0
    MOVM.DB.W [R0], (R13)

    /*
     * note that 5a's RFE is not the v6 arch. instruction (0xe89d0a00,
     * I think), which loads CPSR from the word after the PC at (R13),
     * but rather the pre-v6 simulation 'MOVM.IA.S.W (R13), [R15]'
     * (0xe8fd8000 since MOVM is LDM in this case), which loads CPSR
     * not from memory but from SPSR due to '.S'.
     */
    RFE
```

`initcode` is a tiny user-mode program compiled separately and embedded in the kernel as a byte array. It is the very first code to run in user space. On ARM, the entry point is in assembly (`init9.s`) because `R12` (the static base register) must be set before any C code can access globals—and doing that in C would pull in the full C runtime. The assembly stub sets `R12`, then calls `startboot()`, which opens `/dev/cons` for file descriptors 0, 1, and 2, binds essential device trees (`#c`, `#e`, `#s`) into the namespace, and calls `exec("/boot/boot")`. After this `exec`, the first process is running `boot.rc`—the kernel's work is done.

⟨init9.c 290c⟩≡
```
//@Scheck: def can actually be found initcode.c, but this file is in skip list
extern void startboot(char*, char**);
```

```
//@Scheck: entry point looked for by the linker 8l
void
_main(char *argv0)
{
    startboot(argv0, &argv0);
}
```

⟨init/user/preboot/arm/init9.s 291a⟩≡

```
/*
 * This is the same as the C program:
 *
 *  void
 *  main(char* argv0)
 *  {
 *      startboot(argv0, &argv0);
 *  }
 *
 * It is in assembler because SB needs to be
 * set and doing this in C drags in too many
 * other routines.
 */
TEXT main(SB), 1, $8
    MOVW    $setR12(SB), R12        /* load the SB */
    MOVW    $boot(SB), R0

    ADD $12, R13, R1            /* pointer to 0(FP) */

    MOVW    R0, 4(R13)          /* pass argc, argv */
    MOVW    R1, 8(R13)

    BL  startboot(SB)
_loop:
    B   _loop
```

⟨initcode.c 291b⟩≡

```
/*
 * IMPORTANT!  DO NOT ADD LIBRARY CALLS TO THIS FILE.
 * The entire text image must fit on one page
 * (and there's no data segment, so any read/write data must be on the stack).
 */

#include <u.h>
#include <libc.h>

char cons[] = "#c/cons";
char boot[] = "/boot/boot";
char dev[] = "/dev";
char c[] = "#c";
char e[] = "#e";
char ec[] = "#ec";
char s[] = "#s";
char srv[] = "/srv";
char env[] = "/env";

// called from init9.s
void
startboot(char *argv0, char **argv)
{
    char buf[200];  /* keep this fairly large to capture error details */
```

```
    /* in case boot is a shell script */
    open(cons, OREAD);
    open(cons, OWRITE);
    open(cons, OWRITE);

    bind(c, dev, MAFTER);
    bind(ec, env, MAFTER);
    bind(e, env, MCREATE|MAFTER);
    bind(s, srv, MREPL|MCREATE);

    USED(argv0);
    exec(boot, argv);

    rerrstr(buf, sizeof buf);
    buf[sizeof buf - 1] = '\0';
    _exits(buf);
}
```

11.4 Booting the user

Once `arch_touser()` drops into user mode, `initcode` calls `exec("/boot/boot")`, which replaces the first process with the boot program. `boot.rc` is a simple `rc` script that mounts the root file system, sets up the user environment, and eventually launches `init` and then the shell. At this point, the kernel is fully operational—all the machinery from the previous chapters is running.

Notice how `startboot()` carefully bootstraps the namespace before calling `exec`. It opens `#c/cons` three times to create file descriptors 0, 1, and 2 (stdin, stdout, stderr)—without these, `boot.rc` could not print anything. Then it binds `#c` (the console device) into `/dev`, `#ec` and `#e` (the environment devices) into `/env`, and `#s` (the srv device) into `/srv`. These binds build just enough of the conventional namespace for `rc` to function. The full namespace—with `/proc`, `/net`, and the root file system—is assembled later by `boot.rc`.

```
#!/boot/rc -m /boot/rcmain
/boot/echo booooooooting...
...
```

Part II

Everything is a File

In Plan 9, the file is not just a storage abstraction—it is the universal interface to everything: devices, network connections, the process table, even the windowing system. This part covers how the kernel implements this idea: the `Chan` and `Dev` structures that represent open files and device drivers, the directory and namespace operations that let each process assemble its own view of the file system, the `Queue` structure used for streaming I/O, and the on-disk file system.

Chapter 12

Files

⟨systab *file syscalls* 294a⟩≡ (57b)
```
[OPEN]      sysopen,
[CLOSE]     sysclose,
[DUP]       sysdup,
[FD2PATH]   sysfd2path, // =~ pwd
[PREAD]     syspread,
[PWRITE]    syspwrite,
[SEEK]      sysseek,
```
Uses `syspread()` 312b, `syspwrite()` 314a, and `sysseek()` 315.

The C prototypes are:

```
int   open(char *file, int omode);
int   close(int fd);
int   dup(int oldfd, int newfd);            // -1 newfd = allocate
int   fd2path(int fd, char *buf, int nbuf);
long  pread(int fd, void *buf, long n, vlong offset);  // -1 = current
long  pwrite(int fd, void *buf, long n, vlong offset);
vlong seek(int fd, vlong n, int type);
```

12.1 Overview

In this chapter, I present the file-related system calls: `sysopen()`, `sysclose()`, `sysdup()`, `sysfd2path()`, `syspread()`, `syspwrite()`, and `sysseek()`. In Plan 9, files are the universal abstraction — not just for storage but for devices, network connections, and system services. The key data structures were introduced in Chapter ??: `\hlinkChan` is the kernel's representation of an open file, `\hlinkDev` defines the file server interface, and `devtab` maps device types to their method tables. This chapter shows how user-visible file descriptors connect to channels, how the syscalls dispatch through the device table, and how the `\hlinkPath` structure tracks human-readable filenames.

12.2 Channels

A `\hlinkChan` carries a `flag` bitset that tracks its lifecycle state. The two fundamental flags are `COPEN` (the channel has been opened for I/O via a device's `open()` method) and `CFREE` (the channel is on the free list and available for reuse). Other flags like `CCEXEC` and `CRCLOSE` are added later via the open mode bits.

⟨*enum* `channelflag` 294b⟩≡ (804b)
```
/*
 * channel flags
```

```
 */
enum ChannelFlag
{
  COPEN = 0x0001,   /* for i/o */
  CFREE = 0x0010,   /* not in use */
  ⟨ChannelFlag cases 321b⟩
};
```

⟨Chan *other fields* 295a⟩≡ (68b) 358d▷

```
// enum<channelflag>> (actually a bitset for certain properties)
ushort  flag;
```

Channels are managed by a custom allocator rather than being individually malloced and freed. The `\hlinkChanalloc` structure maintains two linked lists: `free` holds recycled channels ready for reuse, and `list` holds every channel ever allocated (used for debugging and the `fid` counter). Channels are never truly freed back to the heap — once allocated, they cycle between the free list and active use. This avoids the cost of repeated `malloc`/`free` pairs for what is one of the most frequently allocated kernel objects.

⟨*struct* Chanalloc 295b⟩≡ (808c)

```
// Channel allocator
struct Chanalloc
{
    //list<ref<Chan>> (next = Chan.next), the free one
    Chan    *free;
    //list<ref<Chan>> (next = Chan.link), the used one
    Chan    *list;

    int fid; // type could be 'Counter', but already have a Lock anyway

    // extra
    Lock;
};
```

⟨Chan *extra fields* 295c⟩≡ (68b)

```
// list<ref<Chan> of Chanalloc.free
Chan* next;     /* allocation */
// list<ref<Chan> of Chanalloc.list
Chan* link;
```

⟨*global* chanalloc 295d⟩≡ (808c)

```
struct Chanalloc chanalloc;
```

Uses Chanalloc 295b.

`\hlinknewchan` first tries the free list; if empty, it allocates a fresh channel and assigns it a unique `fid` (used by the mount device `devmnt` to multiplex 9P sessions). The channel is initialized with `type` 0 (the root device), so if an error occurs before the caller associates it with a real device, `cclose()` will call `rootclose()`, which is a no-op — a safe default.

⟨*constructor* newchan 295e⟩≡ (808c)

```
Chan*
newchan(void)
{
    Chan *c;

    lock(&chanalloc);
    c = chanalloc.free;
    if(c != nil)
        chanalloc.free = c->next;
    unlock(&chanalloc);
```

```
    if(c == nil){
        c = smalloc(sizeof(Chan));
        lock(&chanalloc);
        c->fid = ++chanalloc.fid;
        c->link = chanalloc.list;
        chanalloc.list = c;
        unlock(&chanalloc);
    }

    /* if you get an error before associating with a dev,
       close calls rootclose, a nop */
    c->type = 0;
    c->dev = 0;

    c->path = nil;

    c->offset = 0;
    c->devoffset = 0;
    // c->mode?

    c->flag = 0;
    c->ref = 1;

    c->ismtpt = false;
    c->umh = nil;
    // c->umc?
    c->uri = 0;

    c->mux = nil;
    c->mchan = nil;
    memset(&c->mqid, 0, sizeof(c->mqid));
    c->mcp = nil;
    c->iounit = 0;
    c->dri = 0;

    c->aux = nil;

    return c;
}
```

Uses `false`, `lock()` 76d, `smalloc()` 104b, and `unlock()` 78d.

⟨*destructor* `chanfree` 296⟩≡ (808c)

```
void
chanfree(Chan *c)
{
    c->flag = CFREE;

    ⟨chanfree() optional free 358e⟩

    pathclose(c->path);
    c->path = nil;

    lock(&chanalloc);
    c->next = chanalloc.free;
    chanalloc.free = c;
    unlock(&chanalloc);
}
```

Uses `CFREE` 294b, `lock()` 76d, `pathclose()` 325c, and `unlock()` 78d.

`\hlinkuniquechan` implements copy-on-write for channels: if the channel has multiple references, it clones

it to get an exclusive copy. This is needed before operations that modify the channel in place (like `open` or `create`).

⟨*function* `uniquechan` 297a⟩≡ (808c)
```
/*
 * Make sure we have the only copy of c.  (Copy on write.)
 */
Chan*
uniquechan(Chan *c)
{
    Chan *nc;

    if(c->ref != 1){
        nc = cclone(c);
        cclose(c);
        return nc;
    }else{
      return c;
    }
}
```
Uses `cclone()` 332a and `cclose`.

12.3 Devices table `devtab`

Each device driver provides a standard `attach()` method. Most drivers implement it by calling the generic helper `\hlinkdevattach`, which allocates a fresh channel, sets its `type` to the index of the driver in `devtab`, and constructs a path string like "`#c`" or "`#I`" from the device's rune character.

⟨*function* `devattach` 297b⟩≡ (810)
```
Chan*
devattach(Rune tc, char *spec)
{
    int n;
    Chan *c;
    char *buf;

    c = newchan();
    mkqid(&c->qid, 0, 0, QTDIR); // root of a device is a QTDIR
    c->type = devno(tc, false);

    if(spec == nil)
        spec = "";
    n = 1+UTFmax+strlen(spec)+1; // '#' + Rune + spec + '\0'
    buf = smalloc(n);
    snprint(buf, n, "#%C%s", tc, spec);
    c->path = newpath(buf);
    free(buf);

    return c;
}
```
Uses `QTDIR` 70b, `UTFmax`, `devno()` 297c, `false`, `free()` 104a, `newchan()` 295e, `newpath()` 324f, and `smalloc()` 104b.

⟨*function* `devno` 297c⟩≡ (810)
```
int
devno(Rune c, bool user)
{
    int i;
```

```
    for(i = 0; devtab[i] != nil; i++) {
        if(devtab[i]->dc == c)
            return i;
    }
    if(!user)
        panic("devno %C %#ux", c, c);

    return -1;
}
```

12.4 File descriptors, Fgrp

A file descriptor is simply an integer index into the `fd` array of an `\hlinkFgrp` (file descriptor group). Each slot is either `nil` (unused) or a reference-counted pointer to a `\hlinkChan`. The array grows dynamically in `DELTAFD` increments, capped at 5000. The `\hlinkFgrp` itself is reference-counted and stored in `Proc.fgrp`, allowing processes created with `RFFDG` to share file descriptor tables.

⟨*struct* Fgrp 298a⟩≡ (868c)
```
struct Fgrp
{
    // array<option<ref_counted<Chan>>>
    Chan  **fd;
    // nelem(fd)
    int nfd;      /* number allocated */
    int maxfd;      /* highest fd in use */ // <= nfd

    ⟨Fgrp other fields 301a⟩
    // extra
    Ref;
};
```

⟨Proc *files fields* 298b⟩+≡ (66) ◁69 318c▷
```
// ref_counted<Fgrp>
Fgrp  *fgrp;    /* File descriptor group */
```

`\hlinkdupfgrp` serves double duty: called with `nil` it acts as a constructor (allocating a fresh table of `DELTAFD` slots), and called with an existing `\hlinkFgrp` it duplicates the table, incrementing the reference count on each channel. This is the function behind `RFFDG`: when a child process gets its own file descriptor group, all current fds are shared (via reference counting on the channels), but future opens in either process are independent.

⟨*function* dupfgrp 298c⟩≡ (875)
```
Fgrp*
dupfgrp(Fgrp *f)
{
    Fgrp *new;
    Chan *c;
    int i;

    new = smalloc(sizeof(Fgrp));
    if(f == nil){
        new->fd = smalloc(DELTAFD*sizeof(Chan*));
        new->nfd = DELTAFD;
        new->ref = 1;
        return new;
    }

    lock(f);
```

```
    /* Make new fd list shorter if possible, preserving quantization */
    new->nfd = f->maxfd+1;
    i = new->nfd%DELTAFD;
    if(i != 0)
        new->nfd += DELTAFD - i;
    new->fd = malloc(new->nfd*sizeof(Chan*));
    if(new->fd == nil){
        unlock(f);
        free(new);
        error("no memory for fgrp");
    }
    new->ref = 1;

    new->maxfd = f->maxfd;
    for(i = 0; i <= f->maxfd; i++) {
        if(c = f->fd[i]){
            incref(c);
            new->fd[i] = c;
        }
    }
    unlock(f);

    return new;
}
```

Uses DELTAFD 299a, free() 104a, incref() 87b, lock() 76d, malloc() 103c, smalloc() 104b, and unlock() 78d.

⟨*constant* DELTAFD 299a⟩≡ (868c)

```
DELTAFD = 20    /* incremental increase in Fgrp.fd's */
```

⟨*function* closefgrp 299b⟩≡ (875)

```
void
closefgrp(Fgrp *f)
{
    int i;
    Chan *c;

    if(f == nil)
        return;

    if(decref(f) != 0)
        return;

    /*
     * If we get into trouble, forceclosefgrp
     * will bail us out.
     */
    up->closingfgrp = f;
    for(i = 0; i <= f->maxfd; i++)
        if(c = f->fd[i]){
            f->fd[i] = nil;
            cclose(c);
        }
    up->closingfgrp = nil;

    free(f->fd);
    free(f);
}
```

Uses cclose, decref() 87c, free() 104a, and up.

\hlinknewfd assigns a channel to the lowest available file descriptor. It scans the fd array for a nil slot via \hlinkfindfreefd, growing the array if needed. The fd table uses the same amortized growth strategy (allocate in DELTAFD-sized chunks) seen throughout the kernel.

⟨*function* newfd 300a⟩≡ (815)
```
int
newfd(Chan *c)
{
    int fd;
    Fgrp *f;

    f = up->fgrp;
    lock(f);
    fd = findfreefd(f, 0);
    if(fd < 0){
        unlockfgrp(f);
        return -1;
    }
    if(fd > f->maxfd)
        f->maxfd = fd;
    f->fd[fd] = c;
    unlockfgrp(f);
    return fd;
}
```
Uses findfreefd() 300b, lock() 76d, unlockfgrp() 301b, and up.

⟨*function* findfreefd 300b⟩≡ (815)
```
/*
 *  this assumes that the fgrp is locked
 */
int
findfreefd(Fgrp *f, int start)
{
    int fd;

    for(fd=start; fd<f->nfd; fd++)
        if(f->fd[fd] == nil)
            break;
    if(fd >= f->nfd && growfd(f, fd) < 0)
        return -1;
    return fd;
}
```
Uses growfd() 300c.

⟨*function* growfd 300c⟩≡ (815)
```
int
growfd(Fgrp *f, int fd) /* fd is always >= 0 */
{
    Chan **newfd, **oldfd;

    if(fd < f->nfd)
        return 0; // should never happen no?
    if(fd >= f->nfd+DELTAFD)
        return -1;  /* out of range */ // when can this happen? from sysdup?
    /*
     * Unbounded allocation is unwise
     */
    if(f->nfd >= 5000){
    Exhausted:
        print("no free file descriptors\n");
```

```
        return -1;
    }
    // realloc
    newfd = malloc((f->nfd+DELTAFD)*sizeof(Chan*));
    if(newfd == nil)
        goto Exhausted;
    oldfd = f->fd;
    memmove(newfd, oldfd, f->nfd*sizeof(Chan*));
    f->fd = newfd;
    free(oldfd);
    f->nfd += DELTAFD;
    if(fd > f->maxfd){
        if(fd/100 > f->maxfd/100)
            f->exceed = (fd/100)*100;
        f->maxfd = fd;
    }
    return 1;
}
```

Uses DELTAFD 299a, free() 104a, malloc() 103c, and print.

⟨Fgrp *other fields* 301a⟩≡ (298a)

```
int exceed;     /* debugging */
```

⟨*function* unlockfgrp 301b⟩≡ (815)

```
static void
unlockfgrp(Fgrp *f)
{
    int ex;

    ex = f->exceed;
    f->exceed = 0;
    unlock(f);
    if(ex)
        pprint("warning: process exceeds %d file descriptors\n", ex);
}
```

Uses pprint and unlock() 78d.

12.5 sysopen(), sysclose()

The open syscall is deceptively simple: it calls \hlinknamec to resolve the filename into a channel (walking through mount points and device tables), then assigns that channel to the lowest available fd. The heavy lifting is in namec() (presented later in this chapter) and in the device's open() method. Close is even simpler: it checks the fd is valid via \hlinkfdtochan, then calls \hlinkfdclose which removes it from the table and decrements the channel's reference count.

⟨*enum* open 301c⟩≡ (853i)

```
enum OpenFlags {
    OREAD = 0, /* open for read */
    OWRITE = 1, /* write */
    ORDWR = 2, /* read and write */
    ⟨OpenFlags cases 305a⟩
};
```

⟨*syscall open* 302a⟩≡ (815)

```
// int open(char *file, int omode);
long
sysopen(ulong* arg)
{
    int fd;
    Chan *c;

    openmode(arg[1]);   /* error check only */
    validaddr(arg[0], 1, false);
    c = namec((char*)arg[0], Aopen, arg[1], 0);
    if(waserror()){
        cclose(c);
        nexterror();
    }
    fd = newfd(c);
    if(fd < 0)
        error(Enofd);
    poperror();
    return fd;
}
```

Uses `Aopen` 305b, `Enofd` 664d, `cclose`, `false`, `namec()` 305f, `newfd()` 300a, `nexterror`, `openmode()` 302b, `poperror` 666d, `validaddr()` 120b, and `waserror` 666d.

⟨*function* `openmode` 302b⟩≡ (812b)

```
int
openmode(ulong o)
{
    o &= ~(OTRUNC|OCEXEC|ORCLOSE);
    if(o > OEXEC)
        error(Ebadarg);
    if(o == OEXEC)
        return OREAD;
    return o;
}
```

Uses `Ebadarg` 664d, `OCEXEC`, `OEXEC` 301c, `ORCLOSE`, `OREAD` 853i, and `OTRUNC`.

⟨*syscall close* 302c⟩≡ (815)

```
// int close(int fd);
long
sysclose(ulong* arg)
{
    fdtochan(arg[0], -1, false, false); // for its error checking side effect
    fdclose(arg[0], 0);
    return 0;
}
```

Uses `false`, `fdclose()` 303b, and `fdtochan()` 302d.

`\hlinkfdtochan` converts a user-space file descriptor integer into a kernel `\hlinkChan` pointer, with bounds checking and optional access mode validation. The `mode` parameter of `-1` means "skip mode checking". When `iref` is true, it increments the channel's reference count so the caller can safely use the channel even if another thread closes the fd concurrently.

⟨*function* `fdtochan` 302d⟩≡ (812b)

```
Chan*
fdtochan(int fd, int mode, bool chkmnt, bool iref)
{
    Chan *c;
    Fgrp *f;
```

```
    c = nil;
    f = up->fgrp;

    lock(f);
    if(fd<0 || f->nfd<=fd || (c = f->fd[fd])==nil) {
        unlock(f);
        error(Ebadfd);
    }
    // c = f->fd[fd] != nil

    if(iref)
        incref(c);
    unlock(f);

    if(chkmnt && (c->flag&CMSG)) {
        ⟨fdtochan() undo iref incref 303a⟩
        error(Ebadusefd);
    }

    if(mode<0 || c->mode==ORDWR) // mode < 0? mean no check?
        return c;

    if((mode&OTRUNC) && c->mode==OREAD) { // BUG impossible, dead code
        ⟨fdtochan() undo iref incref 303a⟩
        error(Ebadusefd);
    }
    // the access mode must match the opening mode
    if((mode&~OTRUNC) != c->mode) {
        ⟨fdtochan() undo iref incref 303a⟩
        error(Ebadusefd);
    }

    return c;
}
```

Uses CMSG 582c, Ebadfd 664d, Ebadusefd 664d, ORDWR 301c, OREAD 853i, OTRUNC, incref() 87b, lock() 76d, unlock() 78d, and up.

⟨fdtochan() *undo iref incref* 303a⟩≡ (302)

```
if(iref)
    cclose(c);
```

Uses cclose.

\hlinkfdclose removes a channel from the fd table and closes it. The flag parameter supports selective closing: when non-zero (e.g., CCEXEC), only fds whose channel has that flag set are closed. This is how "close on exec" works — sysexec() calls fdclose(i, CCEXEC) for every fd. After clearing the slot, fdclose walks backward to update maxfd, keeping it accurate so that closefgrp() does not scan empty slots.

⟨*function* fdclose 303b⟩≡ (812b)

```
void
fdclose(int fd, int flag)
{
    int i;
    Chan *c;
    Fgrp *f = up->fgrp;

    lock(f);
    c = f->fd[fd]; // bound checking??
    if(c == nil){
        /* can happen for users with shared fd tables */
        unlock(f);
```

```
        return;
    }
    if(flag){
        if(c==nil || !(c->flag&flag)){ // dead test c==nil? see above
            unlock(f);
            return;
        }
    }
    f->fd[fd] = nil;

    if(fd == f->maxfd)
        for(i=fd; --i>=0 && f->fd[i]==nil; )
            f->maxfd = i;

    unlock(f);
    cclose(c);
}
```

Uses cclose, lock() 76d, unlock() 78d, and up.

⟨Dev *methods* 304a⟩+≡ (71b) ◁290a 307a▷

```
void  (*close)(Chan*);
```

\hlinkcclose is the central channel destructor. It decrements the reference count; if it reaches zero, it calls the device's close() method (wrapped in waserror since the close may involve I/O to a remote file server that could fail) and then returns the channel to the free list via \hlinkchanfree.

⟨*function* cclose 304b⟩≡ (808c)

```
void
chan_cclose(Chan *c)
{
    if(c->flag&CFREE)
        panic("cclose %#p", getcallerpc(&c));

    DBG("cclose %p name=%s ref=%ld\n", c, c->path->s, c->ref);
    if(decref(c))
        return;

    if(!waserror()){
        devtab[c->type]->close(c);
        poperror();
    }
    chanfree(c);
}
```

Uses CFREE 294b, DBG-765 808b, chanfree() 296, decref() 87c, poperror 666d, and waserror 666d.

12.6 namec()

\hlinknamec ("name to channel") is the central function that translates a file path string into a \hlinkChan. It is called by every file syscall: sysopen(), syscreate(), sysstat(), sysremove(), sysbind(), syschdir(), and sysexec(). The amode parameter (Aopen, Acreate, Aremove, Aaccess, Atodir, Abind, Amount) tells namec() what the caller intends to do with the result, which affects how mount points are traversed. The overall flow is: (1) pick a starting channel (up->slash for "/", up->dot for relative paths, or a device attach for "#" paths), (2) parse the name into path elements, (3) call \hlinkwalk to resolve those elements through devices and mount points, and (4) call the device's open() or create() method.

⟨sysexec() *call namec() to get a channel in* tc *from file* 304c⟩≡ (141c)

```
tc = namec(file, Aopen, OEXEC, 0);
```

⟨OpenFlags *cases* 305a⟩≡ (301c) 321a▷
```
OEXEC = 3, /* execute, == read but check execute permission */
```

⟨*enum* accessnamec 305b⟩≡ (804b)
```
/*
 * Access types in namec
 */
enum AccessFlags
{
  Aopen,        /* for i/o */
  Acreate,      /* is to be created */
  Aremove,      /* will be removed by caller */
  Aaccess,      /* as in stat, wstat */
  Atodir,       /* as in chdir */
  ⟨AccessFlags cases 349c⟩
};
```

⟨namec() *locals* 305c⟩≡ (305f) 305d▷
```
char *name;
```

⟨namec() *locals* 305d⟩+≡ (305f) ◁305c 305e▷
```
Elemlist e;
```

⟨namec() *locals* 305e⟩+≡ (305f) ◁305d 307d▷
```
Chan *c;
```

⟨*function* namec 305f⟩≡ (808c)
```
/*
 * Turn a name into a channel.
 * &name[0] is known to be a valid address.  It may be a kernel address.
 *
 * Opening with amode Aopen, Acreate, Aremove, or Aaccess guarantees
 * that the result will be the only reference to that particular fid.
 * This is necessary since we might pass the result to
 * devtab[]->remove().
 *
 * Opening Atodir or Amount does not guarantee this.
 *
 * Under certain circumstances, opening Aaccess will cause
 * an unnecessary clone in order to get a uniquechan Chan so it
 * can attach the correct name.  Sysstat and sys_stat need the
 * correct name so they can rewrite the stat info.
 */
Chan*
namec(char *aname, int amode, int omode, ulong perm)
{
    ⟨namec() locals 305c⟩

    if(aname[0] == '\0')
        error("empty file name");

    aname = validnamedup(aname, true);

    if(waserror()){
        free(aname);
        nexterror();
    }
    DBG("namec %s %d %d\n", aname, amode, omode);
    name = aname;
```

```
sharppath = (name[0] == '#');

/*
 * Find the starting off point (the current slash, the root of
 * a device tree, or the current dot) as well as the name to
 * evaluate starting there.
 */
switch(name[0]){
⟨namec() if name[0] is a sharp 348b⟩
case '/':
    c = up->slash;
    incref(c);
    break;

default:
    c = up->dot;
    incref(c);
    break;
}

⟨namec() initializes Elemlist e 309c⟩

if(waserror()){
    cclose(c);
    ⟨namec() if waserror free Elemlist e and prepare nice error 311c⟩
}

/*
 * Build a list of elements in the name.
 */
parsename(name, &e);

/*
 * On create, ....
 */
⟨namec() adjust Elemlist e if Acreate 333a⟩

if(walk(&c, e.elems, e.nelems, sharppath, &e.nerror) < 0){
    if(e.nerror < 0 || e.nerror > e.nelems){
        print("namec %s walk error nerror=%d\n", aname, e.nerror);
        e.nerror = 0;
    }
    nexterror();
}

⟨namec() error if not a directory or cannot exec directory 324c⟩

switch(amode){
case Aopen:
case Aremove:
case Aaccess:
Open:
    ⟨namec() case Aopen, Acreate, Aremove, Aaccess, handle mountpoint part1 354b⟩

    switch(amode){
    case Aopen:
    case Acreate:
        ⟨namec() print error if c-¿umh != nil 358f⟩
        ⟨namec() case Aopen, Acreate, handle mountpoint part2 358c⟩
```

```
            ⟨namec() set channel flag before open 746a⟩
            c = devtab[c->type]->open(c, omode&~OCEXEC);
            ⟨namec() set channel flag after open 321c⟩
            break;

        case Aaccess:
        case Aremove:
            ⟨namec() case Aremove, Aaccess, handle mountpoint part2 354c⟩
            break;

        }
        break;
    ⟨namec() other cases 333d⟩
    default:
        panic("unknown namec access %d\n", amode);
    }

    ⟨namec() set genbuf from Elemlist e 307b⟩

    ⟨namec() free Elemlist e 310a⟩
    poperror(); /* e c */
    free(aname);
    poperror(); /* aname */

    return c;
}
```

Uses Aaccess 305b, Acreate 305b, Aopen 305b, Aremove 305b, DBG-765 808b, OCEXEC, cclose, free() 104a, incref() 87b, nexterror, poperror 666d, print, true, up, validnamedup() 307e, walk() 328f, and waserror 666d.

⟨Dev *methods* 307a⟩+≡ (71b) ◁304a 313a▷

```
Chan* (*open)(Chan*, int);
```

⟨namec() *set genbuf from Elemlist e* 307b⟩≡ (305f)

```
/* place final element in genbuf for e.g. exec */
if(e.nelems > 0)
    kstrcpy(up->genbuf, e.elems[e.nelems-1], sizeof up->genbuf);
else
    kstrcpy(up->genbuf, ".", sizeof up->genbuf);
```

Uses kstrcpy() 669a and up.

⟨Proc *other fields* 307c⟩+≡ (66) ◁257c 513b▷

```
char  genbuf[128];  /* buffer used e.g. for last name element from namec */
```

⟨namec() *locals* 307d⟩+≡ (305f) ◁305e 311b▷

```
int len, n, t;
```

12.6.1 Checking valid filename string

Before parsing, namec() validates the filename string: it must be shorter than 2^{16} bytes (to fit in a 9P string field) and must not contain control characters or other "frog" characters. The dup variant copies the string out of user space before the second scan, preventing a malicious thread from modifying the string after validation.

⟨*function* validnamedup 307e⟩≡ (808c)

```
char*
validnamedup(char *aname, bool slashok)
{
    return validname0(aname, slashok, true, getcallerpc(&aname));
}
```

Uses true and validname0() 308b.

⟨*function* `validname` 308a⟩≡ (808c)

```
void
validname(char *aname, bool slashok)
{
    validname0(aname, slashok, false, getcallerpc(&aname));
}
```

Uses `false` and `validname0()` 308b.

⟨*function* `validname0` 308b⟩≡ (808c)

```
/*
 * Check that the name
 *  a) is in valid memory.
 *  b) is shorter than 2^16 bytes, so it can fit in a 9P string field.
 *  c) contains no frogs.
 * The first byte is known to be addressible by the requester, so the
 * routine works for kernel and user memory both.
 * The parameter slashok flags whether a slash character is an error
 * or a valid character.
 *
 * The parameter dup flags whether the string should be copied
 * out of user space before being scanned the second time.
 * (Otherwise a malicious thread could remove the NUL, causing us
 * to access unchecked addresses.)
 */
static char*
validname0(char *aname, bool slashok, bool dup, kern_addr pc)
{
    char *ename, *name, *s;
    int n;
    int c;
    Rune r;

    name = aname;

    if((virt_addr)name < KZERO){
        if(!dup)
            print("warning: validname called from %#p with user pointer", pc);
        ename = vmemchr(name, 0, (1<<16));
    }else
        ename = memchr(name, 0, (1<<16));

    if(ename==nil || ename-name>=(1<<16))
        error("name too long");

    s = nil;
    if(dup){
        n = ename-name;
        s = smalloc(n+1);
        memmove(s, name, n);
        s[n] = '\0';
        aname = s;
        name = s;
        setmalloctag(s, pc);
    }

    // checking validity of name
    while(*name){
        /* all characters above '~' are ok */
        c = *(byte*)name;
        if(c >= Runeself)
```

```
            name += chartorune(&r, name);
        else{
            if(isfrog[c])
                if(!slashok || c!='/'){
                    snprint(up->genbuf, sizeof(up->genbuf), "%s: %q", Ebadchar, aname);
                    free(s);
                    error(up->genbuf);
                }
            name++;
        }
    }
    return s;
}
```

Uses Ebadchar 664d, Runeself 672a, free() 104a, print, setmalloctag() 106a, smalloc() 104b, up, and vmemchr() 120c.

⟨*global* isfrog 309a⟩≡ (808c)

```
char isfrog[256]={
    /*NUL*/ 1, 1, 1, 1, 1, 1, 1, 1,
    /*BKS*/ 1, 1, 1, 1, 1, 1, 1, 1,
    /*DLE*/ 1, 1, 1, 1, 1, 1, 1, 1,
    /*CAN*/ 1, 1, 1, 1, 1, 1, 1, 1,
    ['/']   1,
    [0x7f]  1,
};
```

12.6.2 Parsing filename

Before walking, namec() parses the path string into an \hlinkElemlist: an array of individual name components ("usr", "pad", "foo.c") with offsets back into the original string for error reporting. The mustbedir flag records whether the path ended with "/", which means the final element must be a directory.

⟨*struct* Elemlist 309b⟩≡ (808c)

```
struct Elemlist
{
    char    *aname; /* original name */
    char    *name;  /* copy of name, so '/' can be overwritten */

    //array<string> pointing to subparts of name
    char    **elems; // subparts
    // size(elems)
    int nelems;
    //array<int>, offset in name for the subparts of name in elems
    int *off;

    bool mustbedir; // if last part of filename is a '/'

    int nerror;
    int prefix;
};
```

⟨namec() *initializes Elemlist* e 309c⟩≡ (305f)

```
e.aname = aname;
e.prefix = name - aname;
e.name = nil;
e.elems = nil;
e.off = nil;
e.nelems = 0;
e.nerror = 0;
```

⟨namec() *free Elemlist e* 310a⟩≡ (305f)
```
free(e.name);
free(e.elems);
free(e.off);
```
Uses free() 104a.

⟨*function* parsename 310b⟩≡ (808c)
```
/*
 * The name is known to be valid.
 * Copy the name so slashes can be overwritten.
 * An empty string will set nelem=0.
 * A path ending in / or /. or /.//./ etc. will have
 * e.mustbedir = true, so that we correctly
 * reject, e.g., "/adm/users/." when /adm/users is a file
 * rather than a directory.
 */
static void
parsename(char *aname, Elemlist *e)
{
    char *name, *slash;

    kstrdup(&e->name, aname);
    name = e->name;
    e->nelems = 0;
    e->elems = nil;
    e->off = smalloc(sizeof(int));
    e->off[0] = skipslash(name) - name;
    for(;;){
        name = skipslash(name);
        if(*name == '\0'){
            e->off[e->nelems] = name+strlen(name) - e->name;
            e->mustbedir = true;
            break;
        }
        growparse(e);
        e->elems[e->nelems++] = name;
        slash = utfrune(name, '/');
        if(slash == nil){
            e->off[e->nelems] = name+strlen(name) - e->name;
            e->mustbedir = false;
            break;
        }
        e->off[e->nelems] = slash - e->name;
        *slash++ = '\0';
        name = slash;
    }
}
```
Uses false, growparse() 310c, kstrdup() 669b, skipslash() 311a, smalloc() 104b, and true.

⟨*function* growparse 310c⟩≡ (808c)
```
static void
growparse(Elemlist *e)
{
    char **new;
    int *inew;
    enum { Delta = 8 };

    if(e->nelems % Delta == 0){
        new = smalloc((e->nelems+Delta) * sizeof(char*));
        memmove(new, e->elems, e->nelems*sizeof(char*));
```

```
        free(e->elems);
        e->elems = new;
        inew = smalloc((e->nelems+Delta+1) * sizeof(int));
        memmove(inew, e->off, (e->nelems+1)*sizeof(int));
        free(e->off);
        e->off = inew;
    }
}
```

Uses free() 104a and smalloc() 104b.

⟨*function* skipslash 311a⟩≡ (808c)

```
/*
 * name is valid. skip leading / and ./ as much as possible
 */
char*
skipslash(char *name)
{
    while(name[0]=='/' || (name[0]=='.' && (name[1]=='\0' || name[1]=='/')))
        name++;
    return name;
}
```

12.6.3 Error management

When namec() fails partway through a path, it constructs an error message showing the prefix of the path that was successfully resolved, helping the user identify which component was not found.

⟨namec() *locals* 311b⟩+≡ (305f) ◁307d 328e▷

```
char tmperrbuf[ERRMAX];
```

Uses ERRMAX.

⟨namec() *if waserror free Elemlist e and prepare nice error* 311c⟩≡ (305f)

```
free(e.name);
free(e.elems);
/*
 * Prepare nice error, showing first e.nerror elements of name.
 */
if(e.nerror == 0)
    nexterror();
strcpy(tmperrbuf, up->errstr);
if(e.off[e.nerror]==0)
    print("nerror=%d but off=%d\n",e.nerror, e.off[e.nerror]);
len = e.prefix+e.off[e.nerror];
free(e.off);
namelenerror(aname, len, tmperrbuf);
```

Uses free() 104a, namelenerror() 311d, nexterror, print, and up.

⟨*function* namelenerror 311d⟩≡ (808c)

```
void
namelenerror(char *aname, int len, char *err)
{
    char *ename, *name, *next;
    int i, errlen;

    /*
     * If the name is short enough, just use the whole thing.
     */
    errlen = strlen(err);
```

```
    if(len < ERRMAX/3 || len+errlen < 2*ERRMAX/3)
        snprint(up->genbuf, sizeof up->genbuf, "%.*s",
            utfnlen(aname, len), aname);
    else{
        /*
         * Print a suffix of the name, but try to get a little info.
         */
        ename = aname+len;
        next = ename;
        do{
            name = next;
            next = memrchr(aname, '/', name-aname);
            if(next == nil)
                next = aname;
            len = ename-next;
        }while(len < ERRMAX/3 || len + errlen < 2*ERRMAX/3);

        /*
         * If the name is ridiculously long, chop it.
         */
        if(name == ename){
            name = ename-ERRMAX/4;
            if(name <= aname)
                panic("bad math in namelenerror");
            /* walk out of current UTF sequence */
            for(i=0; (*name&0xC0)==0x80 && i<UTFmax; i++)
                name++;
        }
        snprint(up->genbuf, sizeof up->genbuf, "...%.*s",
            utfnlen(name, ename-name), name);
    }
    snprint(up->errstr, ERRMAX, "%#q %s", up->genbuf, err);
    nexterror();
}
```

Uses ERRMAX, UTFmax, memrchr() 668b, nexterror, and up.

⟨*function* nameerror 312a⟩≡ (808c)

```
void
nameerror(char *name, char *err)
{
    namelenerror(name, strlen(name), err);
}
```

Uses namelenerror() 311d.

12.7 syspread(), syspwrite()

Plan 9 uses pread and pwrite (with an explicit offset) rather than separate read/write/seek syscalls. This matches the 9P protocol, where every read and write message carries an offset, making the operations atomic and stateless. When the offset argument is ~0ULL~ ("no offset"), the kernel falls back to maintaining the channel's internal c->offset. The actual I/O dispatches through devtab[c->type]->read() or devtab[c->type]->write(), reaching the appropriate device driver.

⟨*syscall pread* 312b⟩≡ (815)

```
// long pread(int fd, void *buf, long nbytes, vlong offset);
long
syspread(ulong* arg)
{
```

```
    vlong v;
    va_list list;

    /* use varargs to guarantee alignment of vlong */
    va_start(list, arg[2]);
    v = va_arg(list, vlong);
    va_end(list);

    if(v == ~0ULL)
        return read(arg, nil);

    return read(arg, &v);
}
```

Uses `read()` 313b.

⟨`Dev` *methods* 313a⟩+≡ (71b) ◁307a 314b▷

```
long  (*read)(Chan*, void*, long, vlong);
```

The internal `read()` function handles two cases depending on the channel type. For regular files, it dispatches directly to the device's `read()` method. For directories (`QTDIR`), the logic is more involved: the kernel must track two offsets — `c->offset` counts bytes as the user sees them (after mount table rewriting by `mountfix()`), while `c->devoffset` counts bytes as the device sees them. This dual accounting is needed because `mountfix()` can change the size of directory entries (rewriting their names to reflect bind/mount translations), so the user-visible byte count diverges from the device-level byte count. The `cclose()` at the end releases the extra reference taken by `fdtochan()` with `iref` true.

⟨*function* `read` 313b⟩≡ (815)

```
// long pread(int fd, void *buf, long nbytes, vlong offset);
static long
read(ulong *arg, vlong *offp)
{
    long n, nn, nnn;
    byte *p;
    Chan *c;
    vlong off;

    n = arg[2];
    validaddr(arg[1], n, true);
    p = (void*)arg[1];
    c = fdtochan(arg[0], OREAD, true, true);

    if(waserror()){
        cclose(c);
        nexterror();
    }

    if(offp == nil) /* use and maintain channel's offset */
        off = c->offset;
    else
        off = *offp;

    if(off < 0)
        error(Enegoff);

    if(off == 0){   /* rewind to the beginning of the directory */
        if(offp == nil){
            c->offset = 0;
            c->devoffset = 0;
        }
```

```
        ⟨read() rewind when off == 0 360a⟩
    }

    ⟨read() if c is a QTDIR 359c⟩
    else
        nnn = nn = devtab[c->type]->read(c, p, n, off);

    lock(c);
    c->offset += nnn;
    c->devoffset += nn;
    unlock(c);

    poperror();
    cclose(c);

    return nnn;
}
```

Uses Enegoff 664d, OREAD 853i, cclose, fdtochan() 302d, lock() 76d, nexterror, poperror 666d, true, unlock() 78d, validaddr() 120b, and waserror 666d.

⟨*syscall pwrite* 314a⟩≡ (815)

```
// long pwrite(int fd, void *buf, long nbytes, vlong offset);
long
syspwrite(ulong* arg)
{
    vlong v;
    va_list list;

    /* use varargs to guarantee alignment of vlong */
    va_start(list, arg[2]);
    v = va_arg(list, vlong);
    va_end(list);

    if(v == ~0ULL)
        return write(arg, nil);

    return write(arg, &v);
}
```

Uses write() 314c.

⟨Dev *methods* 314b⟩+≡ (71b) ◁313a 327b▷

```
long  (*write)(Chan*, void*, long, vlong);
```

The write() function uses an optimistic offset strategy: it advances c->offset by the requested count n before calling the device, then rolls back if the device wrote fewer bytes than requested. This differs from read(), which advances the offset after the device call. The asymmetry exists because a partial write is a normal condition (the caller may retry), while a partial read means the data simply was not there. Writing to a directory is forbidden — unlike reads, which return serialized directory entries, there is no meaningful write operation on a directory.

⟨*function* write 314c⟩≡ (815)

```
// long pwrite(int fd, void *buf, long nbytes, vlong offset);
static long
write(ulong *arg, vlong *offp)
{
    Chan *c;
    long m, n;
    vlong off;
```

```
    validaddr(arg[1], arg[2], false);
    n = 0;
    c = fdtochan(arg[0], OWRITE, true, true);
    if(waserror()) {
        if(offp == nil){
            lock(c);
            c->offset -= n;
            unlock(c);
        }
        cclose(c);
        nexterror();
    }

    if(c->qid.type & QTDIR)
        error(Eisdir);

    n = arg[2];

    if(offp == nil){    /* use and maintain channel's offset */
        lock(c);
        off = c->offset;
        c->offset += n;
        unlock(c);
    }else
        off = *offp;

    if(off < 0)
        error(Enegoff);

    m = devtab[c->type]->write(c, (void*)arg[1], n, off);

    if(offp == nil && m < n){
        lock(c);
        c->offset -= n - m;
        unlock(c);
    }

    poperror();
    cclose(c);

    return m;
}
```

Uses Enegoff 664d, OWRITE, QTDIR 70b, cclose, false, fdtochan() 302d, lock() 76d, nexterror, poperror 666d, true, unlock() 78d, validaddr() 120b, and waserror 666d.

12.8 sysseek()

There is no `seek` method in the `\hlinkDev` interface — seek is handled entirely by the kernel by updating `c->offset`. The device never needs to know about seeks because it receives the offset on every read and write call. For type-2 seeks (from end of file), the kernel calls `stat()` to learn the file length. Directories have restricted seek: you can only rewind to offset 0.

⟨*syscall seek* 315⟩≡ (815)

```
// vlong seek(int fd, vlong n, int type);
long
sysseek(ulong* arg)
{
    validaddr(arg[0], BY2V, true);
```

```
    arch_validalign(arg[0], sizeof(vlong));
    sseek(arg);
    return 0;
}
```

Uses sseek() 316, true, and validaddr() 120b.

⟨*function* sseek 316⟩≡ (815)

```
union v_or_u2 {
  vlong v;
  ulong u[2];
};

// vlong seek(int fd, vlong n, int type);
static void
sseek(ulong *arg)
{
    Chan *c;
    byte buf[sizeof(DirEntry)+100];
    DirEntry dir;
    int n;
    vlong off;
    union v_or_u2 o;

    c = fdtochan(arg[1], -1, true, true);
    if(waserror()){
        cclose(c);
        nexterror();
    }
    ⟨sseek() and pipes 458b⟩

    off = 0;
    o.u[0] = arg[2];
    o.u[1] = arg[3];
    switch(arg[4]){
    case 0: // from the start
        off = o.v;
        ⟨sseek() ensures off is 0 for directories 324a⟩
        if(off < 0)
            error(Enegoff);
        c->offset = off; // just write, no need for lock
        break;

    case 1: // from the current location
        ⟨sseek() disallows seek type 1 or 2 for directories 324b⟩
        lock(c);    /* lock for read/write update */
        off = o.v + c->offset;
        if(off < 0){
            unlock(c);
            error(Enegoff);
        }
        c->offset = off;
        unlock(c);
        break;

    case 2: // from the end of the file
        ⟨sseek() disallows seek type 1 or 2 for directories 324b⟩
        n = devtab[c->type]->stat(c, buf, sizeof buf);
        if(convM2D(buf, n, &dir, nil) == 0)
            error("internal error: stat error in seek");
        off = dir.length + o.v;
```

```
        if(off < 0)
            error(Enegoff);
        c->offset = off;
        break;

    default:
        error(Ebadarg);
    }
    *(vlong*)arg[0] = off;
    c->uri = 0;
    c->dri = 0;
    cclose(c);
    poperror();
}
```

Uses Ebadarg 664d, Enegoff 664d, cclose, fdtochan() 302d, lock() 76d, nexterror, poperror 666d, true, unlock() 78d, v_or_u2 316, and waserror 666d.

12.9 sysdup()

The dup() system call duplicates a file descriptor. If newfd is −1, a new descriptor is allocated. Otherwise, the specified slot is used, and any existing channel there is closed *after* the dup—this ordering matters for idioms like date > #d/1 where stdout is being replaced.

⟨*syscall dup* 317⟩≡ (815)

```
// int dup(int oldfd, int newfd);
long
sysdup(ulong* arg)
{
    int fd;
    Chan *c, *oc;
    Fgrp *f = up->fgrp;

    /*
     * Close after dup'ing, so date > #d/1 works
     */
    c = fdtochan(arg[0], -1, false, true);
    fd = arg[1];
    if(fd != -1){
        lock(f);
        if(fd<0 || growfd(f, fd)<0) {
            unlockfgrp(f);
            cclose(c);
            error(Ebadfd);
        }
        if(fd > f->maxfd)
            f->maxfd = fd;

        oc = f->fd[fd];
        f->fd[fd] = c;
        unlockfgrp(f);
        if(oc)
            cclose(oc);
    }else{
        if(waserror()) {
            cclose(c);
            nexterror();
        }
        fd = newfd(c);
```

```
        if(fd < 0)
            error(Enofd);
        poperror();
    }

    return fd;
}
```

Uses Ebadfd 664d, Enofd 664d, cclose, false, fdtochan() 302d, growfd() 300c, lock() 76d, newfd() 300a, nexterror, poperror 666d, true, unlockfgrp() 301b, up, and waserror 666d.

12.10 sysfd2path() (a.k.a pwd())

fd2path returns the full pathname associated with an open file descriptor. This is why the kernel must track \hlinkPath objects on every channel — without them, there would be no way to answer "what file is fd 3 pointing to?". The implementation is trivial: look up the fd, return c->path->s.

⟨*syscall fd2path* 318a⟩≡ (815)

```
// int fd2path(int fd, char *buf, int nbuf);
long
sysfd2path(ulong* arg)
{
    Chan *c;

    validaddr(arg[1], arg[2], true);
    c = fdtochan(arg[0], -1, false, true);
    snprint((char*)arg[1], arg[2], "%s",
              chanpath(c));
    cclose(c);
    return 0;
}
```

Uses cclose, chanpath() 318b, false, fdtochan() 302d, true, and validaddr() 120b.

⟨*function* chanpath 318b⟩≡ (808c)

```
char*
chanpath(Chan *c)
{
    if(c == nil)
        return "<nil chan>";
    if(c->path == nil)
        return "<nil path>";
    if(c->path->s == nil)
        return "<nil path.s>";
    return c->path->s;
}
```

12.11 Collecting unclosed files

When a process exits, \hlinkclosefgrp iterates over all open file descriptors and calls \hlinkcclose on each. But closing a channel that refers to a remote file server (via the mount device) can block or even deadlock — for instance, if the file server is the dying process itself. If a kill signal arrives during this teardown, \hlinkforceclosefgrp hands the remaining unclosed channels to a background "clunk queue" served by a dedicated kernel process, breaking the deadlock.

⟨Proc *files fields* 318c⟩+≡ (66) ◁298b 346c▷

```
Fgrp  *closingfgrp; /* used during teardown */
```

⟨sleep() *forceclosefgrp* 319a⟩≡ (450a)
```
if(up->procctl == Proc_exitme && up->closingfgrp)
    forceclosefgrp();
```
Uses Proc_exitme 159a, forceclosefgrp() 319b, and up.

⟨*function* forceclosefgrp 319b⟩≡ (875)
```
/*
 * Called from sleep because up is in the middle
 * of closefgrp and just got a kill ctl message.
 * This usually means that up has wedged because
 * of some kind of deadly embrace with mntclose
 * trying to talk to itself.  To break free, hand the
 * unclosed channels to the close queue.  Once they
 * are finished, the blocked cclose that we've
 * interrupted will finish by itself.
 */
void
forceclosefgrp(void)
{
    int i;
    Chan *c;
    Fgrp *f;

    if(up->procctl != Proc_exitme || up->closingfgrp == nil){
        print("bad forceclosefgrp call");
        return;
    }

    f = up->closingfgrp;
    for(i = 0; i <= f->maxfd; i++)
        if(c = f->fd[i]){
            f->fd[i] = nil;
            ccloseq(c);
        }
}
```
Uses Proc_exitme 159a, ccloseq() 320a, print, and up.

The \hlinkClunkq is a simple producer-consumer queue: dying processes enqueue channels via ccloseq(), and background kernel processes (closeproc) dequeue and close them. The nqueued and nclosed counters are for debugging. The QLock serializes consumer access (only one closeproc drains at a time), while the Rendez lets the consumer sleep when the queue is empty.

⟨*struct* Clunkq 319c⟩≡ (808c)
```
/*
 * Queue a chan to be closed by one of the clunk procs.
 */
struct Clunkq {
    Chan *head;
    Chan *tail;
    int nqueued;
    int nclosed;
    Lock l;
    QLock q;
    Rendez r;
};
```

⟨*global* clunkq 319d⟩≡ (808c)
```
struct Clunkq clunkq;
```
Uses Clunkq 319c.

\hlinkccloseq ("close, queued") is the deferred variant of cclose(). Instead of calling the device's close() method directly — which might block talking to a remote file server — it enqueues the channel on the clunk queue. If no closeproc worker is sleeping, it spawns a new one.

⟨*function* ccloseq 320a⟩≡ (808c)
```
void
ccloseq(Chan *c)
{
    if(c->flag&CFREE)
        panic("cclose %#p", getcallerpc(&c));

    DBG("ccloseq %p name=%s ref=%ld\n", c, c->path->s, c->ref);

    if(decref(c))
        return;

    lock(&clunkq.l);
    clunkq.nqueued++;
    c->next = nil;
    if(clunkq.head)
        clunkq.tail->next = c;
    else
        clunkq.head = c;
    clunkq.tail = c;
    unlock(&clunkq.l);

    if(!wakeup(&clunkq.r))
        kproc("kcloseproc", closeproc, nil);
}
```
Uses CFREE 294b, DBG-765 808b, closeproc() 320c, clunkq 319d, decref() 87c, kproc() 157c, lock() 76d, unlock() 78d, and wakeup.

⟨*function* clunkwork 320b⟩≡ (808c)
```
static int
clunkwork(void*)
{
    return clunkq.head != nil;
}
```
Uses clunkq 319d.

The closeproc worker is a self-managing kernel process: ccloseq() spawns one via kproc() only if wakeup() finds no existing sleeper, so workers are created on demand. Each worker loops draining the queue; if it finds nothing after a 5-second timeout, it exits via pexit(). This means the system only spends a kernel process slot on clunk work when there is actual work to do, and the worker disappears when idle.

⟨*function* closeproc 320c⟩≡ (808c)
```
void
closeproc(void*)
{
    Chan *c;

    for(;;){
        qlock(&clunkq.q);
        if(clunkq.head == nil){
            if(!waserror()){
                tsleep(&clunkq.r, clunkwork, nil, 5000);
                poperror();
            }
            if(clunkq.head == nil){
                qunlock(&clunkq.q);
```

```
                pexit("no work", /*freemem*/true);
            }
        }
        lock(&clunkq.l);
        c = clunkq.head;
        clunkq.head = c->next;
        clunkq.nclosed++;
        unlock(&clunkq.l);
        qunlock(&clunkq.q);
        if(!waserror()){
            devtab[c->type]->close(c);
            poperror();
        }
        chanfree(c);
    }
}
```

Uses chanfree() 296, clunkq 319d, clunkwork() 320b, lock() 76d, pexit, poperror 666d, qlock() 81f, qunlock() 82c, true, tsleep, unlock() 78d, and waserror 666d.

12.12 Special opening modes

Two optional flags can be OR'd into the open mode: OCEXEC causes the fd to be automatically closed when the process calls exec() (preventing fd leaks into child programs), and ORCLOSE causes the file to be removed when the fd is closed (useful for temporary files). These flags are stored in the channel's flag field as CCEXEC and CRCLOSE respectively, since c->mode only holds the base access mode.

12.12.1 Close on exec

⟨OpenFlags *cases* 321a⟩+≡ (301c) ◁305a 322a▷

```
OCEXEC = 32,  /* or'ed in, close on exec */
```

⟨ChannelFlag *cases* 321b⟩≡ (294b) 322b▷

```
CCEXEC  = 0x0008,   /* close on exec */
```

⟨namec() *set channel flag after open* 321c⟩≡ (305f) 322c▷

```
if(omode & OCEXEC)
    c->flag |= CCEXEC;
```

Uses CCEXEC 321b and OCEXEC.

⟨namec() *set channel flag after create* 321d⟩≡ (333d) 322d▷

```
if(omode & OCEXEC)
    cnew->flag |= CCEXEC;
```

Uses CCEXEC 321b and OCEXEC.

⟨sysexec() *locals* 321e⟩+≡ (141c) ◁195c 508j▷

```
Fgrp *f;
```

⟨sysexec() *close files marked as opened with close on exec* 321f⟩≡ (141c)

```
/*
 * Close on exec
 */
f = up->fgrp;
for(i=0; i<=f->maxfd; i++)
    fdclose(i, CCEXEC);
```

12.12.2 Remove on close

⟨OpenFlags *cases* 322a⟩+≡ (301c) ◁321a 335a▷
```
  ORCLOSE = 64,  /* or'ed in, remove on close */
```

⟨ChannelFlag *cases* 322b⟩+≡ (294b) ◁321b 582c▷
```
  CRCLOSE = 0x0020,   /* remove on close */
```

⟨namec() *set channel flag after open* 322c⟩+≡ (305f) ◁321c
```
  if(omode & ORCLOSE)
      c->flag |= CRCLOSE;
```
Uses CRCLOSE 322b and ORCLOSE.

⟨namec() *set channel flag after create* 322d⟩+≡ (333d) ◁321d
```
  if(omode & ORCLOSE)
      cnew->flag |= CRCLOSE;
```
Uses CRCLOSE 322b and ORCLOSE.

Chapter 13

Directories

⟨systab *directory syscalls* 323a⟩≡ (57b)
```
[CREATE]    syscreate,
[REMOVE]    sysremove,

[CHDIR]     syschdir,

[STAT]      sysstat,
[FSTAT]     sysfstat,
[WSTAT]     syswstat,
[FWSTAT]    sysfwstat,
```
Uses `syschdir()` 336a, `sysfd2path()` 318a, `sysfstat()` 339a, `sysfwstat()` 340a, and `syswstat()` 339d.

The C prototypes are:

```
int   create(char *file, int omode, ulong perm);
int   remove(char *file);
int   chdir(char *dir);
int   stat(char *file, uchar *buf, int nbuf);   // 9P wire format
int   fstat(int fd, uchar *buf, int nbuf);
int   wstat(char *file, uchar *buf, int nbuf);   // w = write metadata
int   fwstat(int fd, uchar *buf, int nbuf);
```

13.1 Overview

This chapter covers the directory-related syscalls: `create`, `remove`, `chdir`, `stat`/`fstat`, and `wstat`/`fwstat`. Directories in Plan 9 are not fundamentally different from files — they are channels whose `qid.type` has the `QTDIR` bit set. The kernel enforces a few directory-specific rules (no writing, no exec, restricted seek), but most directory operations are delegated to device drivers via the same `\hlinkDev` method table. This chapter also presents the `\hlinkPath` structure for tracking human-readable pathnames, the `\hlinkwalk` function that resolves path elements one by one through device drivers, and the `\hlinkDirEntry` structure (Plan 9's equivalent of UNIX's `struct stat`).

⟨*function* `error_if_not_dir` 323b⟩≡ (808c)
```
void
error_if_not_dir(Chan *c)
{
    if(c->qid.type & QTDIR)
        return;
    error(Enotdir);
}
```
Uses `QTDIR` 70b.

⟨sseek() *ensures off is 0 for directories* 324a⟩≡ (316)
```
if((c->qid.type & QTDIR) && off != 0)
    error(Eisdir);
```
Uses QTDIR 70b.

⟨sseek() *disallows seek type 1 or 2 for directories* 324b⟩≡ (316)
```
if(c->qid.type & QTDIR)
    error(Eisdir);
```
Uses QTDIR 70b.

⟨namec() *error if not a directory or cannot exec directory* 324c⟩≡ (305f)
```
if(e.mustbedir && !(c->qid.type & QTDIR))
    error(Enotdir);

if((amode == Aopen) && (omode & OEXEC) && (c->qid.type & QTDIR))
    error("cannot exec directory");
```
Uses Aopen 305b, OEXEC 301c, and QTDIR 70b.

13.2 Path

Every \hlinkChan carries a \hlinkPath that records the human-readable filename used to reach it (e.g., "/usr/pad/foo.c"). The path is needed for fd2path() (the Plan 9 equivalent of "pwd" for arbitrary fds) and for good error messages. One might think the kernel could reconstruct the path by walking ".." back to the root, but mount points make that impossible — the ".." of a mounted directory leads to a different place than the ".." of the underlying device. So the path must be tracked explicitly as the name is resolved. \hlinkPath is reference-counted and uses copy-on-write via \hlinkuniquepath.

⟨*struct* Path 324d⟩≡ (804b)
```
struct Path
{
    char  *s;
    int len;      /* strlen(s) */
    int alen;     /* allocated length of s */

    ⟨Path mtpt fields 356a⟩

    // extra
    Ref;
};
```

⟨*global* npath 324e⟩≡ (808c)
```
Ref npath;
```

⟨*constructor* newpath 324f⟩≡ (808c)
```
Path*
newpath(char *s)
{
    int i;
    Path *p;

    p = smalloc(sizeof(Path));
    i = strlen(s);
    p->len = i;
    p->alen = i+PATHSLOP;
    p->s = smalloc(p->alen);
    memmove(p->s, s, i+1);
    p->ref = 1;
```

```
    incref(&npath);

    ⟨newpath() mtpt handling 356c⟩
    return p;
}
```

Uses PATHSLOP-766 325a, incref() 87b, npath 324e, and smalloc() 104b.

⟨*constant* PATHSLOP 325a⟩≡ (808c)

```
PATHSLOP    = 20,
```

⟨*function* copypath 325b⟩≡ (808c)

```
static Path*
copypath(Path *p)
{
    int i;
    Path *pp;

    pp = smalloc(sizeof(Path));
    pp->ref = 1;
    incref(&npath);
    DBG("copypath %s %p => %p\n", p->s, p, pp);

    pp->len = p->len;
    pp->alen = p->alen;
    pp->s = smalloc(p->alen);
    memmove(pp->s, p->s, p->len+1);

    ⟨copypath() mtpt handling 356d⟩
    return pp;
}
```

Uses DBG-765 808b, incref() 87b, npath 324e, and smalloc() 104b.

⟨*destructor* pathclose 325c⟩≡ (808c)

```
void
pathclose(Path *p)
{
    int i;

    if(p == nil)
        return;
    ⟨pathclose() debugging 356f⟩
    if(decref(p))
        return;

    decref(&npath);
    free(p->s);

    ⟨pathclose() mtpt handling 356e⟩
    free(p);
}
```

Uses decref() 87c, free() 104a, and npath 324e.

⟨*function* uniquepath 325d⟩≡ (808c)

```
static Path*
uniquepath(Path *p)
{
    Path *new;

    if(p->ref > 1){
```

```
        /* copy on write */
        new = copypath(p);
        pathclose(p);
        p = new;
    }
    return p;
}
```

Uses copypath() 325b and pathclose() 325c.

\hlinkaddelem appends a path element to a \hlinkPath, growing the string buffer if needed. For ".." it calls \hlinkfixdotdotname to clean the path in place. The from parameter records the mount point channel at this position (or nil), which is stored in the mtpt array for later ".." handling.

⟨*function* addelem 326a⟩≡ (808c)

```
static Path*
addelem(Path *p, char *s, Chan *from)
{
    char *t;
    int a, i;
    Chan *c, **tt;

    if(s[0]=='.' && s[1]=='\0')
        return p;

    p = uniquepath(p);

    i = strlen(s);
    // +1 for the '\0'
    if(p->len+1+i+1 > p->alen){
        a = p->len+1+i+1 + PATHSLOP;
        t = smalloc(a);
        memmove(t, p->s, p->len+1);
        free(p->s);
        p->s = t;
        p->alen = a;
    }
    /* don't insert extra slash if one is present */
    if(p->len>0 && p->s[p->len-1]!='/' && s[0]!='/')
        p->s[p->len++] = '/';
    memmove(p->s+p->len, s, i+1);
    p->len += i;

    if(isdotdot(s)){
        fixdotdotname(p);
    }
    ⟨addelem() mtpt handling 357a⟩

    return p;
}
```

Uses PATHSLOP-766 325a, fixdotdotname() 327a, free() 104a, isdotdot() 326b, smalloc() 104b, and uniquepath() 325d.

13.3 '..'

⟨*function* isdotdot 326b⟩≡ (808c)

```
int
isdotdot(char *p)
{
    return p[0]=='.' && p[1]=='.' && p[2]=='\0';
}
```

⟨*function* `fixdotdotname` 327a⟩≡ (808c)

```
/*
 * In place, rewrite name to compress multiple /, eliminate ., and process ..
 * (Really only called to remove a trailing .. that has been added.
 * Otherwise would need to update n->mtpt as well.)
 */
static void
fixdotdotname(Path *p)
{
    char *r;

    if(p->s[0] == '#'){
        r = strchr(p->s, '/');
        if(r == nil)
            return;
        cleanname(r);

        /*
         * The correct name is #i rather than #i/,
         * but the correct name of #/ is #/.
         */
        if(strcmp(r, "/")==0 && p->s[1] != '/')
            *r = '\0';
    }else
        cleanname(p->s);
    p->len = strlen(p->s);
}
```

13.4 `walk()`

The `walk` `\hlinkDev` method takes an array of path element names and walks through them as far as possible within one device. It returns a `\hlinkWalkqid` containing the cloned channel and an array of `Qid`s for each element successfully walked. Taking multiple names in a single call avoids a 9P round-trip per path element.

⟨`Dev` *methods* 327b⟩+≡ (71b) ◁314b 333c▷

```
Walkqid*(*walk)(Chan*, Chan*, char**, int);
```

⟨*struct* `Walkqid` 327c⟩≡ (804b)

```
struct Walkqid
{
  Chan  *clone;
  int nqid;
  // variable array length, size = nqid
  Qid qid[1];
};
```

⟨`walk()` *locals* 327d⟩≡ (328f) 327e▷

```
Chan *c;
Path *path;
Walkqid *wq;
Chan *nc;
```

⟨`walk()` *locals* 327e⟩+≡ (328f) ◁327d 328a▷

```
Mhead *mh;
Mhead *nmh;
int type, dev;
```

⟨walk() *locals* 328a⟩+≡ (328f) ◁327e 328b▷
```
Chan *mtpt;
bool didmount;
```

⟨walk() *locals* 328b⟩+≡ (328f) ◁328a 328c▷
```
bool dotdot;
```

⟨walk() *locals* 328c⟩+≡ (328f) ◁328b 328d▷
```
int nhave, ntry;
```

⟨walk() *locals* 328d⟩+≡ (328f) ◁328c 358a▷
```
int i, n;
```

⟨namec() *locals* 328e⟩+≡ (305f) ◁311b 333b▷
```
bool sharppath;
```

The kernel-level `\hlinkwalk` is the mount-point-aware layer above the device walk method. Its loop processes batches of path elements: it calls the device's walk for a run of names (stopping before any ".." element), then checks each intermediate `Qid` for mount points via `\hlinkfindmount`. When it hits a mount point mid-path, it crosses to the mounted channel and continues walking in the new device. The ".." element is handled specially: it may need to uncross a mount point via `\hlinkundomount`, returning to the channel that was originally mounted upon. The `didmount` flag tracks which side of a mount point the current channel is on.

A concrete example helps. Suppose the namespace has `/usr/glenda` bound to an on-disk filesystem served by `devsd` (the SD-card driver), and the user opens the path `/usr/glenda/src/hello.c`. The caller has already split that path into the four elements `usr, glenda, src, hello.c`, and `walk` starts with `c` pointing at the root channel. The loop then executes:

```
iter  nhave  c (device)      ewalk names returned   mount?
----  -----  --------------  --------------------   ------
 1      0    devroot         "usr","glenda"       -> yes,
                                                      cross to devsd
 2      2    devsd           "src","hello.c"      -> no
 3      4    -- loop exits, nnames reached --
```

In the first iteration, `ewalk` asks `devroot` to walk `usr,glenda`. `devroot` succeeds for both. The mount-point check then sees that the `Qid` of `glenda` is mounted: `findmount` returns `nc` pointing at the `devsd` root of the mounted tree, `didmount` is set, and `n` is set to 2 so the second iteration starts after "glenda". In the second iteration `c` is on `devsd`; `ewalk` walks `src,hello.c` within that device, finds no further mounts, and the loop exits after four elements. The resulting `Path` string is `/usr/glenda/src/hello.c` —but the channel `c` now lives in `devsd`, not `devroot`. A later `..` from here would call `undomount` to return to the original `devroot` channel that `glenda` was mounted on.

⟨*function* walk 328f⟩≡ (808c)
```
/*
 * Either walks all the way or not at all.  No partial results in *cp.
 * *nerror is the number of names to display in an error message.
 */
int
walk(Chan **cp, char **names, int nnames, bool sharppath, int *nerror)
{
    ⟨walk() locals 327d⟩

    c = *cp;
    incref(c);
    path = c->path;
    incref(path);
```

```
mh = nil;

/*
 * While we haven't gotten all the way down the path:
 *    1. step through a mount point, if any
 *    2. send a walk request for initial dotdot or initial prefix without dotdot
 *    3. move to the first mountpoint along the way.
 *    4. repeat.
 *
 * Each time through the loop:
 *
 *  If didmount==false, c is on the undomount side of the mount point.
 *  If didmount==true, c is on the domount side of the mount point.
 *  Either way, c's full path is path.
 */
didmount = false;
for(nhave=0; nhave<nnames; nhave+=n){

    ⟨walk() return error if channel c is not a dir 331a⟩

    ⟨walk() set dotdot and ntry to index of next '..' or end 330b⟩

    if(!sharppath && !dotdot && !didmount)
        domount(&c, &mh, &path);

    type = c->type;
    dev = c->dev;

    if((wq = ewalk(c, names+nhave, ntry)) == nil){
        // didn't find the element, look in union mount
        ⟨walk() if c is a union mount, find and adjust type and dev 358b⟩
        ⟨walk() if wq is still nil, close and return 331b⟩
    }

    didmount = false;
    if(dotdot){
        assert(wq->nqid == 1);
        assert(wq->clone != nil);
        path = addelem(path, "..", nil);
        nc = undomount(wq->clone, path);
        nmh = nil;
        n = 1;
    }else{
        nc = nil;
        nmh = nil;
        if(!sharppath){
            // mount points along path?
            for(i=0; i<wq->nqid && i<ntry-1; i++){
                if(findmount(&nc, &nmh, type, dev, wq->qid[i])){
                    didmount = true;
                    break;
                }
            }
        }

        if(!didmount){  /* no mount points along path */
            ⟨walk() if nc == nil and wq-¿clone == nil return error 331c⟩
            n = wq->nqid;
            nc = wq->clone;
```

```
            }else{      /* stopped early, at a mount point */
                if(wq->clone != nil){
                    cclose(wq->clone);
                    wq->clone = nil;
                }
                n = i+1;
            }

            for(i=0; i<n; i++){
                mtpt = nil;
                if(i==n-1 && nmh)
                    mtpt = nmh->from;
                path = addelem(path, names[nhave+i], mtpt);
            }
        }
        cclose(c);
        c = nc;
        putmhead(mh);
        mh = nmh;
        free(wq);
    }

    putmhead(mh);

    c = uniquechan(c); // late? why now? we've modified in place c already

    ⟨walk() print error if c-¿umh != nil 358g⟩

    pathclose(c->path);
    c->path = path;

    cclose(*cp);
    *cp = c;
    if(nerror)
        *nerror = nhave;
    return 0;
}
```

Uses addelem() 326a, assert 853f, cclose, domount() 355, ewalk() 330a, false, findmount() 354d, free() 104a, incref() 87b, pathclose() 325c, putmhead() 344c, true, undomount() 357b, and uniquechan() 297a.

⟨*function* ewalk 330a⟩≡ (808c)

```
/*
 * Call dev walk but catch errors.
 */
static Walkqid*
ewalk(Chan *c, char **names, int nnames)
{
    Walkqid *wq;

    if(waserror())
        return nil;
    wq = devtab[c->type]->walk(c, nil, names, nnames);
    poperror();
    return wq;
}
```

Uses poperror 666d and waserror 666d.

⟨walk() *set dotdot and ntry to index of next '..' or end* 330b⟩≡ (328f)

```
dotdot = false;
// remaining names
```

```
ntry = nnames - nhave;
if(ntry > MAXWELEM)
    ntry = MAXWELEM;

for(i=0; i<ntry; i++){
    if(isdotdot(names[nhave+i])){
        if(i==0){
            dotdot = true;
            ntry = 1;
        }else
            ntry = i;
        break;
    }
}
// ntry is now the index of the next '..' or the remaining names to walk
```

Uses false, isdotdot() 326b, and true.

⟨walk() *return error if channel c is not a dir* 331a⟩≡ (328f)

```
if(!(c->qid.type & QTDIR)){
    pathclose(path);
    cclose(c);
    if(nerror)
        *nerror = nhave;
    if(mh != nil)
        putmhead(mh);
    strcpy(up->errstr, Enotdir);
    return -1;
}
```

Uses QTDIR 70b, cclose, pathclose() 325c, putmhead() 344c, and up.

⟨walk() *if wq is still nil, close and return* 331b⟩≡ (328f)

```
if(wq == nil){
    pathclose(path);
    cclose(c);
    if(nerror)
        *nerror = nhave+1;
    if(mh != nil)
        putmhead(mh);
    return -1;
}
```

Uses cclose, pathclose() 325c, and putmhead() 344c.

⟨walk() *if nc == nil and wq-¿clone == nil return error* 331c⟩≡ (328f)

```
if(wq->clone == nil){
    pathclose(path);
    cclose(c);
    if(wq->nqid==0 || (wq->qid[wq->nqid-1].type&QTDIR)){
        if(nerror)
            *nerror = nhave+wq->nqid+1;
        strcpy(up->errstr, Edoesnotexist);
    }else{
        if(nerror)
            *nerror = nhave+wq->nqid;
        strcpy(up->errstr, Enotdir);
    }
    free(wq);
    if(mh != nil)
        putmhead(mh);
    return -1;
```

```
}
```

Uses Edoesnotexist-768 665a, QTDIR 70b, cclose, free() 104a, pathclose() 325c, putmhead() 344c, and up.

⟨*function* cclone 332a⟩≡ (808c)

```
Chan*
cclone(Chan *c)
{
    Chan *nc;
    Walkqid *wq;

    wq = devtab[c->type]->walk(c, nil, nil, 0);

    if(wq == nil)
        error("clone failed");
    nc = wq->clone;
    free(wq);
    nc->path = c->path;
    if(c->path)
        incref(c->path);
    return nc;
}
```

Uses free() 104a and incref() 87b.

13.5 syscreate()

The create syscall is structurally similar to open, but calls namec() with Acreate mode. The interesting complexity is in the namec() Acreate case, which must handle the "create/create race": if two processes try to create the same file simultaneously, one create(5) 9P message will succeed and the other will fail. The losing process retries with open+OTRUNC instead, providing the expected semantics where create on an existing file truncates it. The OEXCL flag lets applications that care about atomicity get the raw "fail if exists" behavior.

⟨*syscall create* 332b⟩≡ (815)

```
// int create(char *file, int omode, ulong perm);
long
syscreate(ulong* arg)
{
    int fd;
    Chan *c;

    openmode(arg[1]&~OEXCL);    /* error check only; OEXCL okay here */
    validaddr(arg[0], 1, false);
    c = namec((char*)arg[0], Acreate, arg[1], arg[2]);
    if(waserror()){
        cclose(c);
        nexterror();
    }
    fd = newfd(c);
    if(fd < 0)
        error(Enofd);
    poperror();
    return fd;
}
```

Uses Acreate 305b, Enofd 664d, OEXCL, cclose, false, namec() 305f, newfd() 300a, nexterror, openmode() 302b, poperror 666d, validaddr() 120b, and waserror 666d.

⟨namec() *adjust Elemlist e if Acreate* 333a⟩≡ (305f)
```
if(amode == Acreate){
    /* perm must have DMDIR if last element is / or /. */
    if(e.mustbedir && !(perm&DMDIR)){
        e.nerror = e.nelems;
        error("create without DMDIR");
    }

    /* don't try to walk the last path element just yet. */
    if(e.nelems == 0)
        error(Eexist);
    e.nelems--;
}
```
Uses Acreate 305b and DMDIR.

⟨namec() *locals* 333b⟩+≡ (305f) ◁328e 348a▷
```
Chan *cnew;
char *createerr;
```

⟨Dev *methods* 333c⟩+≡ (71b) ◁327b 335c▷
```
void  (*create)(Chan*, char*, int, ulong);
```

⟨namec() *other cases* 333d⟩≡ (305f) 336b▷
```
case Acreate:
    /*
     * We've already walked all but the last element.
     * If the last exists, try to open it OTRUNC.
     * If omode&OEXCL is set, just give up.
     */
    e.nelems++;
    e.nerror++;
    if(walk(&c, e.elems+e.nelems-1, 1, sharppath, nil) == 0){
        if(omode&OEXCL)
            error(Eexist);
        omode |= OTRUNC;
        goto Open;
    }

    /*
     * The semantics of the create(2) system call are that if the
     * file exists and can be written, it is to be opened with truncation.
     * On the other hand, the create(5) message fails if the file exists.
     * If we get two create(2) calls happening simultaneously,
     * they might both get here and send create(5) messages, but only
     * one of the messages will succeed.  To provide the expected create(2)
     * semantics, the call with the failed message needs to try the above
     * walk again, opening for truncation.  This correctly solves the
     * create/create race, in the sense that any observable outcome can
     * be explained as one happening before the other.
     * The create/create race is quite common.  For example, it happens
     * when two rc subshells simultaneously update the same
     * environment variable.
     *
     * The implementation still admits a create/create/remove race:
     * (A) walk to file, fails
     * (B) walk to file, fails
     * (A) create file, succeeds, returns
     * (B) create file, fails
     * (A) remove file, succeeds, returns
     * (B) walk to file, return failure.
```

```
 *
 * This is hardly as common as the create/create race, and is really
 * not too much worse than what might happen if (B) got a hold of a
 * file descriptor and then the file was removed -- either way (B) can't do
 * anything with the result of the create call.  So we don't care about this race.
 *
 * Applications that care about more fine-grained decision of the races
 * can use the OEXCL flag to get at the underlying create(5) semantics;
 * by default we provide the common case.
 *
 * We need to stay behind the mount point in case we
 * need to do the first walk again (should the create fail).
 *
 * We also need to cross the mount point and find the directory
 * in the union in which we should be creating.
 *
 * The channel staying behind is c, the one moving forward is cnew.
 */
m = nil;
cnew = nil; /* is this assignment necessary? */
if(!waserror()){    /* try create */
    ⟨namec() Acreate case, if cnew is a mountpoint 361a⟩
    else{
        cnew = c;
        incref(cnew);
    }

    /*
     * We need our own copy of the Chan because we're
     * about to send a create, which will move it.  Once we have
     * our own copy, we can fix the name, which might be wrong
     * if findmount gave us a new Chan.
     */
    cnew = uniquechan(cnew);
    pathclose(cnew->path);
    cnew->path = c->path;
    incref(cnew->path);

    devtab[cnew->type]->create(cnew, e.elems[e.nelems-1], omode&~(OEXCL|OCEXEC), perm);
    poperror();
    ⟨namec() set channel flag after create 321d⟩
    if(m)
        putmhead(m);
    cclose(c);
    c = cnew;
    c->path = addelem(c->path, e.elems[e.nelems-1], nil);
    break; // switch
}

/* create failed */
cclose(cnew);
if(m)
    putmhead(m);
if(omode & OEXCL)
    nexterror();

/* save error */
createerr = up->errstr;
up->errstr = tmperrbuf;
/* note: we depend that walk does not error */
```

```
    if(walk(&c, e.elems+e.nelems-1, 1, sharppath, nil) < 0){
        up->errstr = createerr;
        error(createerr);   /* report true error */
    }
    up->errstr = createerr;
    omode |= OTRUNC;
    goto Open;
```

Uses Acreate 305b, OCEXEC, OEXCL, OTRUNC, addelem() 326a, cclose, incref() 87b, nexterror, pathclose() 325c, poperror 666d, putmhead() 344c, uniquechan() 297a, up, walk() 328f, and waserror 666d.

⟨OpenFlags *cases* 335a⟩+≡ (301c) ◁322a 335b▷

```
OTRUNC = 16,  /* or'ed in (except for exec), truncate file first */
```

⟨OpenFlags *cases* 335b⟩+≡ (301c) ◁335a

```
OEXCL = 0x1000,  /* or'ed in, exclusive create */
```

13.6 sysremove()

Remove takes a filename, resolves it via namec() with Aremove, and calls the device's remove() method. Removing a mount point is explicitly disallowed — the ambiguity of whether to remove the mount point or the mounted channel is resolved by simply forbidding it. After remove(), the channel's type is reset to 0 (root device) because the 9P remove clunks the fid, so the channel must not call the original device's close() again.

⟨Dev *methods* 335c⟩+≡ (71b) ◁333c 338b▷

```
void  (*remove)(Chan*);
```

⟨*syscall remove* 335d⟩≡ (815)

```
// int remove(char *file);
long
sysremove(ulong* arg)
{
    Chan *c;

    validaddr(arg[0], 1, false);
    c = namec((char*)arg[0], Aremove, 0, 0);
    /*
     * Removing mount points is disallowed to avoid surprises
     * (which should be removed: the mount point or the mounted Chan?).
     */
    if(c->ismtpt){
        cclose(c);
        error(Eismtpt);
    }
    if(waserror()){
        c->type = 0;    /* see below */
        cclose(c);
        nexterror();
    }
    devtab[c->type]->remove(c);
    /*
     * Remove clunks the fid, but we need to recover the Chan
     * so fake it up.  rootclose() is known to be a nop.
     */
    c->type = 0;
    poperror();
    cclose(c);
    return 0;
}
```

Uses Aremove 305b, Eismtpt 664d, cclose, false, namec() 305f, nexterror, poperror 666d, validaddr() 120b, and waserror 666d.

13.7 syschdir()

Changing directory is simple: resolve the path via namec() with Atodir mode (which verifies it is a directory), then replace up->dot.

⟨*syscall chdir* 336a⟩≡ (815)
```
// int chdir(char *dirname);
long
syschdir(ulong* arg)
{
    Chan *c;

    validaddr(arg[0], 1, false);
    c = namec((char*)arg[0], Atodir, 0, 0);
    cclose(up->dot);
    up->dot = c;
    return 0;
}
```
Uses Atodir 305b, cclose, false, namec() 305f, up, and validaddr() 120b.

⟨namec() *other cases* 336b⟩+≡ (305f) ◁333d 349d▷
```
case Atodir:
    /*
     * Directories (e.g. for cd) are left before the mount point,
     * so one may mount on / or . and see the effect.
     */
    if(!(c->qid.type & QTDIR))
        error(Enotdir);
    break;
```
Uses Atodir 305b and QTDIR 70b.

13.8 DirEntry

Where UNIX uses numeric uid/gid, Plan 9 uses string usernames directly in \hlinkDirEntry, eliminating the need for /etc/passwd. The structure is serialized into a platform-independent wire format matching the 9P stat message.

⟨*struct* DirEntry 336c⟩≡ (853i)
```
struct DirEntry {
  /* system-modified data */
  ushort  type; /* server type */
  uint  dev;  /* server subtype */

  /* file data */
  Qid qid;  /* unique id from server */
  // bitset<enum<dirmode>>
  ulong mode; /* permissions */

  ulong atime;  /* last read time */
  ulong mtime;  /* last write time */

  vlong length; /* file length: see <u.h> */
  char  *name;  /* last element of path */
```

```
    char  *uid; /* owner name */
    char  *gid; /* group name */
    char  *muid;  /* last modifier name */
  };
```

⟨*enum* dirmode 337a⟩≡ (853i)

```
  /* bits in DirEntry.mode */
  enum Dirmode {
      DMDIR = 0x80000000,  /* mode bit for directories */

      DMREAD = 0x4,   /* mode bit for read permission */
      DMWRITE = 0x2,   /* mode bit for write permission */
      DMEXEC = 0x1,   /* mode bit for execute permission */
      ⟨Dirmode cases 432a⟩
  };
```

Before the two helper functions below, it helps to know what the serialized form of a `DirEntry` looks like in memory—because those helpers poke at raw byte offsets, not C fields. When a device's `stat` method fills a buffer, it writes a byte sequence that exactly matches the 9P "Rstat" message wire format. Every field has a fixed encoding (`BIT16SZ` = 2, `BIT32SZ` = 4, `BIT64SZ` = 8):

```
offset  size   field
------  -----  ------------------------
     0    2    total size (not counted in itself)
     2    2    type
     4    4    dev
     8    1    qid.type
     9    4    qid.vers
    13    8    qid.path
    21    4    mode
    25    4    atime
    29    4    mtime
    33    8    length
    41    2    len(name)   <-+
    43    n    name          |  variable-length
               +2   len(uid)    strings, each
                    ...         preceded by a
                                2-byte length
```

So `offsetof_name_direntry` is not computing a C `offsetof`—it is walking the fixed-size prefix (2+2+4 + 1+4+8 + 4+4+4 + 8 = 41 bytes) to land on the 2-byte length of the `name` string, and then returning a pointer to the first character of the name itself. The `BIT*SZ` constants are written out one per group so the reader can match them against the four field groups in the diagram above (header, qid, mode/times, length). `direntry_setname` uses this to rewrite the name field in place after a stat, when the kernel wants to replace whatever name the device reported with the last element of the user-visible path. Because `name` has variable length, the trailing strings (`uid`, `gid`, `muid`) must be shifted by `len - olen` bytes, and the overall length header at offset 0 must be updated accordingly.

⟨*function* direntry_setname 337b⟩≡ (815)

```
  static long
  direntry_setname(char *name, int len, byte *p, long n, long maxn)
  {
      char *oname;
      int olen;
```

```
    long nn;

    if(n == BIT16SZ)
        return BIT16SZ;

    oname = offsetof_name_direntry(p, &olen);

    nn = n+len-olen;
    PBIT16(p, nn-BIT16SZ);
    if(nn > maxn)
        return BIT16SZ;

    if(len != olen)
        memmove(oname+len, oname+olen, p+n-(byte*)(oname+olen));
    PBIT16((byte*)(oname-2), len);
    memmove(oname, name, len);
    return nn;
}
```

Uses offsetof_name_direntry() 338a.

⟨*function* offsetof_name_direntry 338a⟩≡ (815)

```
static char*
offsetof_name_direntry(byte *p, int *n)
{
    p += BIT16SZ+BIT16SZ+BIT32SZ +
         BIT8SZ+BIT32SZ+BIT64SZ +
         BIT32SZ+BIT32SZ+BIT32SZ +
         BIT64SZ;
    *n = GBIT16(p);
    return (char*)p+BIT16SZ;
}
```

13.9 sysstat()

The stat/fstat syscalls call the device's stat() method to fill a buffer with a serialized \hlinkDirEntry. After getting the raw stat, sysstat() rewrites the name field to match the last component of the path the user supplied, since the device may return a different name after crossing mount points.

⟨Dev *methods* 338b⟩+≡ (71b) ◁335c 339c▷

```
int (*stat)(Chan*, byte*, int);
```

⟨*syscall stat* 338c⟩≡ (815)

```
// int stat(char *name, byte *edir, int nedir);
long
sysstat(ulong* arg)
{
    char *name;
    Chan *c;
    uint l;

    l = arg[2];
    validaddr(arg[1], l, true);
    validaddr(arg[0], 1, false);
    c = namec((char*)arg[0], Aaccess, 0, 0);
    if(waserror()){
        cclose(c);
        nexterror();
    }
```

```
    l = devtab[c->type]->stat(c, (byte*)arg[1], l);

    name = pathlast(c->path);
    if(name)
        l = direntry_setname(name, strlen(name), (byte*)arg[1], l, arg[2]);

    poperror();
    cclose(c);
    return l;
}
```

Uses Aaccess 305b, cclose, direntry_setname() 337b, false, namec() 305f, nexterror, pathlast() 339b, poperror 666d, true, validaddr() 120b, and waserror 666d.

⟨*syscall fstat* 339a⟩≡ (815)

```
// int fstat(int fd, byte *edir, int nedir);
long
sysfstat(ulong* arg)
{
    Chan *c;
    uint l;

    l = arg[2];
    validaddr(arg[1], l, true);
    c = fdtochan(arg[0], -1, false, true);
    if(waserror()) {
        cclose(c);
        nexterror();
    }
    l = devtab[c->type]->stat(c, (byte*)arg[1], l);
    poperror();
    cclose(c);
    return l;
}
```

Uses cclose, false, fdtochan() 302d, nexterror, poperror 666d, true, validaddr() 120b, and waserror 666d.

⟨*function* pathlast 339b⟩≡ (815)

```
static char*
pathlast(Path *p)
{
    char *s;

    if(p == nil)
        return nil;
    if(p->len == 0)
        return nil;
    s = strrchr(p->s, '/');
    if(s)
        return s+1;
    return p->s;
}
```

⟨Dev *methods* 339c⟩+≡ (71b) ◁338b 348c▷

```
int (*wstat)(Chan*, byte*, int);
```

The wstat syscalls modify file metadata. The kernel validates the stat buffer and checks that mount points cannot be renamed (to avoid ambiguity about what is being renamed — the mount point or the mounted channel).

⟨*syscall wstat* 339d⟩≡ (815)

```
// int wstat(char *name, byte *edir, int nedir);
```

```
long
syswstat(ulong* arg)
{
    Chan *c;
    uint l;

    l = arg[2];
    validaddr(arg[1], l, false);
    validstat((byte*)arg[1], l);
    validaddr(arg[0], 1, false);
    c = namec((char*)arg[0], Aaccess, 0, 0);
    return wstat(c, (byte*)arg[1], l);
}
```

Uses Aaccess 305b, false, namec() 305f, validaddr() 120b, validstat() 340b, and wstat() 341.

⟨*syscall fwstat* 340a⟩≡ (815)

```
// int fwstat(int fd, byte *edir, int nedir);
long
sysfwstat(ulong* arg)
{
    Chan *c;
    uint l;

    l = arg[2];
    validaddr(arg[1], l, false);
    validstat((byte*)arg[1], l);
    c = fdtochan(arg[0], -1, true, true);
    return wstat(c, (byte*)arg[1], l);
}
```

Uses false, fdtochan() 302d, true, validaddr() 120b, validstat() 340b, and wstat() 341.

⟨*function* validstat 340b⟩≡ (812b)

```
void
validstat(byte *s, int n)
{
    int m;
    char buf[64];

    if(statcheck(s, n) < 0)
        error(Ebadstat);
    /* verify that name entry is acceptable */
    s += STATFIXLEN - 4*BIT16SZ;    /* location of first string */
    /*
     * s now points at count for first string.
     * if it's too long, let the server decide; this is
     * only for his protection anyway. otherwise
     * we'd have to allocate and waserror.
     */
    m = GBIT16(s);
    s += BIT16SZ;
    if(m+1 > sizeof buf)
        return;
    memmove(buf, s, m);
    buf[m] = '\0';
    /* name could be '/' */
    if(strcmp(buf, "/") != 0)
        validname(buf, false);
}
```

Uses false and validname() 308a.

⟨*function* `wstat` 341⟩≡ (815)

```
static long
wstat(Chan *c, byte *d, int nd)
{
    long l;
    int namelen;

    if(waserror()){
        cclose(c);
        nexterror();
    }
    if(c->ismtpt){
        /*
         * Renaming mount points is disallowed to avoid surprises
         * (which should be renamed? the mount point or the mounted Chan?).
         */
        offsetof_name_direntry(d, &namelen);
        if(namelen)
            nameerror(chanpath(c), Eismtpt);
    }
    l = devtab[c->type]->wstat(c, d, nd);
    poperror();
    cclose(c);
    return l;
}
```

Uses `Eismtpt` 664d, `cclose`, `chanpath()` 318b, `nameerror()` 312a, `nexterror`, `offsetof_name_direntry()` 338a, `poperror` 666d, and `waserror` 666d.

Chapter 14

Namespaces

⟨`systab` *namespace syscalls* 342⟩≡ (57b)

```
[BIND]      sysbind,
[MOUNT]     sysmount,
[UNMOUNT]   sysunmount,
```

Uses `sysmount()` 363g and `sysunmount()` 365a.

The C prototypes are:

```
int   bind(char *new, char *old, int flags);
int   mount(int fd, int afd, char *old, int flags, char *aname);
int   unmount(char *new, char *old);
```

In `mount`: `fd` is the pipe to a 9P server, `afd` is an authentication fd (-1 if none), `old` is the mount point, `flags` is `MBEFORE`/`MAFTER`/`MREPL`, and `aname` names which tree to attach to.

14.1 Namespace principles

Every operating system kernel has to solve the same problem dozens of times a second: a user program hands it a path like `/usr/bin/ls`, and the kernel has to turn that string into a kernel object (a file, a device, a pipe, a socket) that subsequent `read`/`write` calls can be dispatched on. The function that does this goes by many names—`namei` in historical UNIX and BSD, `path_walk` (or `filename_lookup`) in Linux, `ObReferenceObjectByName` in Windows, and `namec` in Plan 9. The algorithm is everywhere the same: split the path at "/", and walk the directory tree one component at a time, asking "does this directory contain an entry named X?" at each step, until the components are exhausted or an error is reported.

On top of that plain walk sits one crucial twist: at some directories, the walk is intercepted and redirected into a different subtree. Such a directory is called a *mount point*, and the rule governing it lives in an auxiliary data structure called a *mount table*. The classic use is attaching a filesystem: you plug in a USB disk and tell the kernel "attach this disk's root at `/mnt/usb`"; thereafter, when the name-resolution walk reaches `/mnt/usb`, it jumps into the disk's root instead of continuing in the original tree, and the remaining components (say, `/photos/2024`) are resolved *there*. `chroot` is a degenerate version of the same idea, one that redirects only the start of the walk. The same mechanism exists everywhere under different names: UNIX `mount`, Linux bind mounts, Windows junction points and reparse points, BSD `nullfs`, macOS synthetic firmlinks.

A subtler generalization turns the mount table from a global thing into a *per-process* thing: different processes can have different mount tables, so that the same string `/home/alice` resolves to different kernel objects depending on who is asking. This is how modern containers work: Linux *mount namespaces* (introduced in 2002, used by Docker, LXC, Kubernetes, `systemd-nspawn`) give each container its own copy of the mount table, so one container's `/etc/passwd` is invisible to another's. FreeBSD jails and Solaris zones are earlier, coarser versions of the same idea. In every case, the operating system is generalizing "where does `/` live" into a per-process attribute—which Plan 9 had from day one, not as a late bolt-on.

In Plan 9, per-process namespaces were the whole *point* of the system: the primitives that define a process's file-system view—`bind`, `mount`, `unmount`, and the implicit namespace-copy that happens at `rfork`—are the ones Plan 9 programs use every day to rearrange their view of the world. The kernel data structures behind this are `Pgrp` (the process group's mount table), `Mhead` (the "from" channel at a mount point), `Mount` (a linked list of "to" channels), and `namec` (the walker itself). A distinctive feature is the *union mount*: multiple "to" channels can be attached at the same mount point and walked in order, so that `/bin` can simultaneously expose programs from several filesystems:

```
bind -a /usr/alice/bin /bin
bind -a /rc/bin        /bin

/bin  ---> [ original /bin  ]
      ---> [ /usr/alice/bin ]   (MAFTER)
      ---> [ /rc/bin        ]   (MAFTER)
```

Opening `/bin/ls` then walks each attached channel in order and returns the first `ls` it finds. This is Plan 9's equivalent of UNIX's `$PATH` search, except that it is built into name resolution itself instead of being a shell convention. The rest of this chapter is the implementation of `Pgrp`, `Mhead`, `Mount`, and `namec`.

14.2 Overview

With those ideas in hand we can dive into the Plan 9 code. We start with the `MountFlags` that control the order-of-attachment at a mount point, then look at `Mhead` and `Mount`, and finally `namec`—the walker that ties it all together.

⟨*enum* `mount` 343⟩≡ (853i)

```
enum MountFlags {
    MREPL = 0x0000,  /* mount replaces object */

    MBEFORE = 0x0001,  /* mount goes before others in union directory */
    MAFTER = 0x0002,  /* mount goes after others in union directory */

    MCREATE = 0x0004,  /* permit creation in mounted directory */
    ⟨MountFlags cases 364d⟩

    MORDERMASK =  0x0003,  /* mask for bits defining order of mounting */
    MMASK = 0x0017,  /* all bits on */
};
```

An `\hlinkMhead` stores the "from" channel (the directory being mounted upon) and a linked list of `\hlinkMount` structures representing what is mounted there. For a simple bind or mount, there is one `\hlinkMount` entry. For a union mount, there are multiple entries in the list, and the order matters: when walking a name through a union directory, the kernel tries each mount's channel in list order.

Pictorially, a `Pgrp` with three bindings—one simple (`#c` replacing `/dev`) and one union (`/386/bin` and `/rc/bin` both appearing as `/bin`)—looks like this. Note how the `Pgrp.mnthash` is a small hash keyed by `qid.path`, each bucket chains `Mhead`s via `Mhead.hash`, and each `Mhead` owns its own list of `Mount`s via `Mount.next`:

```
Pgrp                              Mhead (for /dev)
 mnthash[k1] ---hash---> from --> Chan (/dev directory, devroot)
     |                   mount --> Mount { to: Chan(#c),
     |                                     next: nil }
     |
 mnthash[k2] ---hash---> Mhead (for /bin)
                          from --> Chan (/bin directory)
```

```
mount --> Mount { to: Chan(/386/bin)
                  flag: MBEFORE,
                  next: ----+
                            |
          Mount { to: Chan(/rc/bin)<-+
                  flag: MAFTER,
                  next: nil }
```

When `namec` walks into `/bin/ls`, it looks up `/bin`'s `qid` in `mnthash` via `MOUNTH`, finds the `Mhead`, and iterates the `Mount` list until some device reports that it has a file called `ls`. If `/386/bin/ls` exists, it wins because `MBEFORE` made its `Mount` the first in the list; otherwise the walk falls through to `/rc/bin/ls`. For `MREPL`, there is only one `Mount` and the original `from` directory is not added to the list—so the "dev" directory `Mount` above has no link back to the real `/dev`.

Two implementation details justify the split into `Mhead` and `Mount`. First, multiple places in the kernel want to keep a pointer to "the top of this mount stack" (the `Pgrp` hash, the `Chan.mh` field returned by `findmount`, transiently some `createdir` paths). A shared `Mhead` with its own `Ref` makes this sound. Second, the `RWlock` on `Mhead` protects modifications to the `Mount` list while readers (walkers) traverse it without holding the `Pgrp` write lock—this is critical because walks are common and mount changes are rare.

14.3 Mount point

⟨*struct* `Mhead` 344a⟩≡ (804b)

```
struct Mhead
{
    // ref<Chan>
    Chan* from;     /* channel mounted upon */
    // list<ref<Mount>> list because of union mount (ordered via MBEFORE|MAFTER)
    Mount*  mount;      /* what's mounted upon it */

    Ref;
    RWlock  lock;
    ⟨Mhead extra fields 347a⟩
};
```

⟨*constructor* `newmhead` 344b⟩≡ (808c)

```
Mhead*
newmhead(Chan *from)
{
    Mhead *mh;

    mh = smalloc(sizeof(Mhead));
    mh->ref = 1;
    mh->from = from;
    incref(from);
    return mh;
}
```

Uses `incref()` 87b and `smalloc()` 104b.

⟨*destructor* `putmhead` 344c⟩≡ (808c)

```
/*
 * This is necessary because there are many
 * pointers to the top of a given mount list:
 *
 *  - the mhead in the namespace hash table
 *  - the mhead in chans returned from findmount:
```

```
 *    used in namec and then by unionread.
 *  - the mhead in chans returned from createdir:
 *    used in the open/create race protect, which is gone.
 *
 * The RWlock in the Mhead protects the mount list it contains.
 * The mount list is deleted when we cunmount.
 * The RWlock ensures that nothing is using the mount list at that time.
 *
 * It is okay to replace c->mh with whatever you want as
 * long as you are sure you have a unique reference to it.
 *
 * This comment might belong somewhere else.
 */
void
putmhead(Mhead *m)
{
    if(m && decref(m) == 0){
        m->mount = (Mount*)0xCafeBeef;
        free(m);
    }
}
```

Uses `decref()` 87c and `free()` 104a.

A `\hlinkMount` holds the "to" channel and flags. The `MBEFORE/MAFTER` ordering is implicit in list position.

⟨*struct* `Mount` 345a⟩≡ (804b)

```
struct Mount
{
    // ref<Chan>
    Chan* to;     /* channel replacing channel */

    // enum<mount>, but mainly for MCREATE, the MBEFORE|MAFTER is encoded via list
    int mflag;

    // option<string>
    char  *spec; // for mount(), nil for bind()
    ⟨Mount other fields 367a⟩

    ⟨Mount extra fields 345c⟩
};
```

⟨*constructor* `newmount` 345b⟩≡ (875)

```
Mount*
newmount(Chan *to, int flag, char *spec)
{
    Mount *m;

    m = smalloc(sizeof(Mount));
    m->to = to;
    incref(to);
    m->mountid = incref(&mountid);
    m->mflag = flag;
    if(spec != nil)
        kstrdup(&m->spec, spec);

    return m;
}
```

Uses `incref()` 87b, `kstrdup()` 669b, `mountid-2606` 367b, and `smalloc()` 104b.

⟨`Mount` *extra fields* 345c⟩≡ (345a) 367d▷

```
Mount*  next;
```

⟨*destructor* `mountfree` 346a⟩≡ (875)
```
void
mountfree(Mount *m)
{
    Mount *f;

    while(m) {
        f = m->next;
        cclose(m->to);
        m->mountid = 0;
        free(m->spec);
        free(m);
        m = f;
    }
}
```
Uses `cclose` and `free()` 104a.

14.4 Mount table

The mount table `\hlinkPgrp` ("process group") is the per-process namespace. It is a hash table of `\hlinkMhead` entries, keyed by `qid.path`. When `\hlinknamec` encounters a directory, it checks this table for mount points. The `Pgrp` is reference-counted; processes created with `RFNAMEG` get their own copy.

⟨*struct* `Pgrp` 346b⟩≡ (868c)
```
// Mount table, aka Namespace, aka process group
struct Pgrp
{
    // hash<qid.path, list<ref<Mhead> (next = Mhead.next)>
    Mhead *mnthash[MNTHASH];

    ⟨Pgrp other fields 347b⟩

    // extra
    Ref;        /* also used as a lock when mounting */

    // ORDER OF LOCK: first debug and then ns
    QLock debug;      /* single access via devproc.c */
    RWlock  ns;     /* Namespace n read/one write lock */
};
```

⟨`Proc` *files fields* 346c⟩+≡ (66) ◁318c
```
// ref_counted<pgrp>
Pgrp  *pgrp;    /* Process group for namespace */
```

The mount table hash has only 32 buckets (`MNTHASH`), indexed by the low bits of `qid.path`. Collisions are handled by chaining `\hlinkMhead` entries via `Mhead.hash`. The small size is adequate because mount tables are typically small (a few dozen entries at most).

⟨*function* `MOUNTH` 346d⟩≡ (868c)
```
enum
{
    MNTLOG  = 5,
    MNTHASH = 1<<MNTLOG,  /* Hash to walk mount table */
};
#define MOUNTH(p,qid) ((p)->mnthash[(qid).path&((1<<MNTLOG)-1)])
```
Uses `MNTLOG` 346d.

⟨Mhead *extra fields* 347a⟩≡ (344a)
```
// hash<qid.path, list<ref<Mhead>> of Pgrp.mnthash
Mhead*  hash;     /* Hash chain */
```

⟨Pgrp *other fields* 347b⟩≡ (346b) 362d▷
```
ulong pgrpid;
```

⟨*global* pgrpid 347c⟩≡ (875)
```
static Counter pgrpid;
```

⟨*constructor* newpgrp 347d⟩≡ (875)
```
Pgrp*
newpgrp(void)
{
    Pgrp *p;

    p = smalloc(sizeof(Pgrp));
    p->ref = 1;
    p->pgrpid = incref(&pgrpid);
    return p;
}
```
Uses incref() 87b, pgrpid-2605 347c, and smalloc() 104b.

When a process group's reference count drops to zero, \hlinkclosepgrp walks every bucket in the hash, closing all "from" channels and freeing all mount chains. The lock ordering (debug then ns) must be respected even during cleanup.

⟨*destructor* closepgrp 347e⟩≡ (875)
```
void
closepgrp(Pgrp *p)
{
    Mhead **h, **e, *f, *next;

    if(decref(p) != 0)
        return;

    qlock(&p->debug);
    wlock(&p->ns);
    p->pgrpid = -1;

    e = &p->mnthash[MNTHASH];
    for(h = p->mnthash; h < e; h++) {
        for(f = *h; f; f = next) {
            wlock(&f->lock);
            cclose(f->from);
            mountfree(f->mount);
            f->mount = nil;
            next = f->hash;
            wunlock(&f->lock);
            putmhead(f);
        }
    }
    wunlock(&p->ns);
    qunlock(&p->debug);
    free(p);
}
```
Uses MNTHASH 346d, cclose, decref() 87c, free() 104a, mountfree() 346a, putmhead() 344c, qlock() 81f, qunlock() 82c, wlock() 85b, and wunlock() 85d.

14.5 Device path

The "#" prefix in path names is how the kernel bootstraps the namespace. A path like "`#c`" directly attaches to the console device driver, bypassing the mount table entirely. During boot, `init0()` uses `bind("\#c", "/dev", MREPL)` to make device drivers accessible at conventional paths. This is Plan 9's alternative to UNIX's major/minor device numbers and static `/dev` entries — here the namespace is constructed dynamically by bind commands.

⟨`namec()` *locals* 348a⟩+≡ (305f) ◁333b 354a▷
```
Rune r;
```

⟨`namec()` *if name[0] is a sharp* 348b⟩≡ (305f)
```
case '#':
    up->genbuf[0] = '\0';
    n = 0;
    while(*name != '\0' && (*name != '/' || n < 2)){ // "#/" is ok hence n < 2
        if(n >= sizeof(up->genbuf)-1)
            error(Efilename);
        up->genbuf[n++] = *name++;
    }
    up->genbuf[n] = '\0';

    n = chartorune(&r, up->genbuf+1)+1; // +1 for the leading '#'

    ⟨namec() noattach checking 363b⟩

    t = devno(r, true);
    if(t == -1)
        error(Ebadsharp);

    c = devtab[t]->attach(up->genbuf+n);
    break;
```
Uses `Ebadsharp` 664d, `Efilename` 664d, `devno()` 297c, `true`, and `up`.

⟨`Dev` *methods* 348c⟩+≡ (71b) ◁339c 433b▷
```
Chan* (*attach)(char*);
```

14.6 `sysbind()`

Both `bind` and `mount` go through `\hlinkbindmount`, which resolves two paths and calls `\hlinkcmount` to insert a mount point.

⟨*syscall bind* 348d⟩≡ (815)
```
// int bind(char *to, char *from, int flag);
long
sysbind(ulong* arg)
{
    return bindmount(false, -1, -1, (char*)arg[0], (char*)arg[1], arg[2], nil);
}
```
Uses `bindmount()` 349a and `false`.

⟨`bindmount()` *locals* 348e⟩≡ (349a) 349b▷
```
Chan *c0;
Chan *c1;
```

⟨*function* bindmount 349a⟩≡ (815)

```
long
bindmount(bool ismount, int fd, int afd, char* arg0, char* arg1, ulong flag, char* spec)
{
    ⟨bindmount() locals 348e⟩

    if((flag&~MMASK) || (flag&MORDERMASK)==(MBEFORE|MAFTER))
        error(Ebadarg);

    ⟨bindmount() if ismount 364b⟩
    else{
        spec = nil;
        validaddr((ulong)arg0, 1, false);
        c0 = namec(arg0, Abind, 0, 0);
    }

    if(waserror()){
        cclose(c0);
        nexterror();
    }

    validaddr((ulong)arg1, 1, false);
    c1 = namec(arg1, Amount, 0, 0);

    if(waserror()){
        cclose(c1);
        nexterror();
    }

    ret = cmount(c0, c1, flag, spec);

    poperror();
    cclose(c1);
    poperror();
    cclose(c0);
    ⟨bindmount() if ismount free 364c⟩
    return ret;
}
```

Uses Abind 349c, Amount 349c, Ebadarg 664d, MAFTER 343, MBEFORE 343, MMASK, MORDERMASK 364d, cclose, cmount() 350c, false, namec() 305f, nexterror, poperror 666d, validaddr() 120b, and waserror 666d.

⟨bindmount() *locals* 349b⟩+≡ (349a) ◁348e 364a▷

```
int ret;
```

⟨AccessFlags *cases* 349c⟩≡ (305b)

```
Abind,        /* for left-hand-side of bind */
Amount,       /* to be mounted or mounted upon */
```

⟨namec() *other cases* 349d⟩+≡ (305f) ◁336b 350a▷

```
case Abind:
    m = nil;
    if(!sharppath)
        /* no need to maintain path - cannot dotdot an Abind */ // so pass nil
        domount(&c, &m, nil);
    ⟨namec() set c-¿umh in Abind if mounted point 361c⟩
    break;
```

Uses Abind 349c and domount() 355.

⟨namec() *other cases* 350a⟩+≡ (305f) ◁349d
```
case Amount:
    /*
     * When mounting on an already mounted upon directory,
     * one wants subsequent mounts to be attached to the
     * original directory, not the replacement.  Don't domount.
     */
    break;
```
Uses Amount 349c.

⟨cmount() *locals* 350b⟩≡ (350c) 362a▷
```
Pgrp *pg;
Mhead *mh;
Mount *m;
int order;
```

\hlinkcmount looks up old's qid in the mount table hash; if no \hlinkMhead exists, it creates one. Then it inserts a new \hlinkMount according to the flag: MREPL replaces the existing list, MBEFORE prepends, MAFTER appends. For union mounts on a fresh \hlinkMhead, the original channel is also added to the list so the union includes the original directory contents.

⟨*function* cmount 350c⟩≡ (808c)
```
int
cmount(Chan *new, Chan *old, int flag, char *spec)
{
    ⟨cmount() locals 350b⟩

    if((old->qid.type ^ new->qid.type) & QTDIR)
        error(Emount);
    ⟨cmount() print error if old-¿umh != nil 359a⟩

    order = flag&MORDERMASK;
    if(!(old->qid.type & QTDIR) && order != MREPL)
        error(Emount);

    ⟨cmount() if new is itself a mount point, error if cant create there 362b⟩

    pg = up->pgrp;
    wlock(&pg->ns);

    // mh = lookup(old->gdid, pg.mnthash)
    l = &MOUNTH(pg, old->qid);
    for(mh = *l; mh; mh = mh->hash){
        if(eqchan(mh->from, old, true))
            break;
        l = &mh->hash;
    }

    if(mh == nil){
        /*
         *  nothing mounted here yet.  create a mount
         *  head and add to the hash table.
         */
        mh = newmhead(old);
        *l = mh;

        /*
         *  if this is a union mount, add the old
         *  node to the mount chain.
         */
```

```
        if(order != MREPL)
            mh->mount = newmount(old, 0, nil);
    }

    wlock(&mh->lock);
    if(waserror()){
        wunlock(&mh->lock);
        nexterror();
    }
    wunlock(&pg->ns);

    m = newmount(new, flag, spec);

    ⟨cmount() if new is itself a mount point, copy mounts 362c⟩

    if(mh->mount && order == MREPL){
        mountfree(mh->mount);
        mh->mount = nil;
    }
    if(flag & MCREATE)
        m->mflag |= MCREATE;

    if(mh->mount && order == MAFTER){
        for(f = mh->mount; f->next; f = f->next)
            ;
        f->next = m;
    }else{ // MBEFORE or MREPL
        // when new was itself a mount point, can have union of mount again
        for(f = m; f->next; f = f->next)
            ;
        f->next = mh->mount; // =~ m->next = mh->mount for the most common case
        mh->mount = m;
    }

    wunlock(&mh->lock);
    poperror();
    return m->mountid;
}
```

Uses Emount 664d, MAFTER 343, MCREATE 343, MORDERMASK 364d, MOUNTH 346d, MREPL 853i, QTDIR 70b, eqchan() 70e, mountfree() 346a, newmhead() 344b, newmount() 345b, nexterror, poperror 666d, true, up, waserror 666d, wlock() 85b, and wunlock() 85d.

14.7 /proc/<pid>/ns

A process's namespace can be inspected by reading `/proc/<pid>/ns`, which produces a sequence of `bind` and `mount` commands that reconstruct the namespace. The `\hlinkmntscan` function walks the mount table hash in `mountid` order (not hash order), so the output is deterministic and reflects the temporal ordering of mounts.

⟨*struct* Mntwalk 351a⟩≡ (740b)

```
struct Mntwalk              /* state for /proc/#/ns */
{
    bool cddone;
    Mhead*  mh;
    Mount*  cm;
};
```

⟨procread() *locals* 351b⟩+≡ (605d) ◁122d 605c▷

```
Mntwalk *mw;
char flag[10];
```

⟨procopen() *cases* 352a⟩≡ (611)

```
case Qns:
    if(omode != OREAD)
        error(Eperm);
    c->aux = malloc(sizeof(Mntwalk));
    break;
```

Uses OREAD 853i, Qns-870 602a, and malloc() 103c.

⟨procclose() *hooks* 352b⟩≡ (617)

```
if(QID(c->qid) == Qns && c->aux != nil)
    free(c->aux);
```

Uses QID-911 604a, Qns-870 602a, and free() 104a.

⟨procread() *cases* 352c⟩+≡ (605d) ◁122e 488a▷

```
case Qns:
    qlock(&p->debug);
    if(waserror()){
        qunlock(&p->debug);
        nexterror();
    }
    if(p->pgrp == nil || p->pid != PID(c->qid))
        error(Eprocdied);

    mw = c->aux;
    if(mw == nil)
        error(Enomem);
    if(mw->cddone){
        qunlock(&p->debug);
        poperror();
        return 0;
    }

    mntscan(mw, p);

    if(mw->mh == nil){
        mw->cddone = true;
        i = snprint(a, n, "cd %s\n", p->dot->path->s);
        qunlock(&p->debug);
        poperror();
        return i;
    }

    int2flag(mw->cm->mflag, flag);
    if(strcmp(mw->cm->to->path->s, "#M") == 0){
        srv = srvname(mw->cm->to->mchan);
        i = snprint(a, n, "mount %s %s %s %s\n", flag,
            srv==nil? mw->cm->to->mchan->path->s : srv,
            mw->mh->from->path->s, mw->cm->spec? mw->cm->spec : "");
        free(srv);
    }else
        i = snprint(a, n, "bind %s %s %s\n", flag,
            mw->cm->to->path->s, mw->mh->from->path->s);
    qunlock(&p->debug);
    poperror();
    return i;
```

Uses Eprocdied 664d, PID-913 604a, Qns-870 602a, free() 104a, int2flag() 353a, mntscan() 353b, nexterror, poperror 666d, qlock() 81f, qunlock() 82c, srvname() 758c, true, and waserror 666d.

⟨*function* `int2flag` 353a⟩≡ (740b)

```
static void
int2flag(int flag, char *s)
{
    if(flag == 0){
        *s = '\0';
        return;
    }
    *s++ = '-';
    if(flag & MAFTER)
        *s++ = 'a';
    if(flag & MBEFORE)
        *s++ = 'b';
    if(flag & MCREATE)
        *s++ = 'c';
    if(flag & MCACHE)
        *s++ = 'C';
    *s = '\0';
}
```

Uses `MAFTER` 343, `MBEFORE` 343, `MCACHE` 343, and `MCREATE` 343.

`\hlinkmntscan` iterates through the mount table in `mountid` order. Since the hash table has no inherent ordering, it does a full scan on every call, finding the mount entry with the smallest `mountid` greater than the last one returned. This is $O(n^2)$ overall but namespaces are small, and it avoids holding locks across reads.

⟨*function* `mntscan` 353b⟩≡ (740b)

```
void
mntscan(Mntwalk *mw, Proc *p)
{
    Pgrp *pg;
    Mount *t;
    Mhead *f;
    int i;
    bool nxt;
    ulong last, bestmid;

    pg = p->pgrp;
    rlock(&pg->ns);

    nxt = false;
    bestmid = ~0;

    last = 0;
    if(mw->mh)
        last = mw->cm->mountid;

    for(i = 0; i < MNTHASH; i++) {
        for(f = pg->mnthash[i]; f; f = f->hash) {
            for(t = f->mount; t; t = t->next) {
                if(mw->mh == nil ||
                  (t->mountid > last && t->mountid < bestmid)) {
                    mw->cm = t;
                    mw->mh = f;
                    bestmid = mw->cm->mountid;
                    nxt = true;
                }
            }
        }
    }
    if(nxt == false)
```

```
            mw->mh = nil;

        runlock(&pg->ns);
    }
```

Uses MNTHASH 346d, false, rlock() 83e, runlock() 84b, and true.

14.8 findmount(), domount()

\hlinkfindmount looks up a channel's qid in the \hlinkPgrp hash and returns the first mount's "to" channel. \hlinkdomount wraps it and also updates the \hlinkPath mount point array for ".." handling.

⟨namec() *locals* 354a⟩+≡ (305f) ◁348a

```
Mhead *m;
Path *path;
```

⟨namec() *case Aopen, Acreate, Aremove, Aaccess, handle mountpoint part1* 354b⟩≡ (305f)

```
/* save&update the name; domount might change c */
path = c->path;
incref(path);

m = nil;
if(!sharppath)
    domount(&c, &m, &path);

/* our own copy to open or remove */
c = uniquechan(c);

/* now it's our copy anyway, we can put the name back */
pathclose(c->path);
c->path = path;

/* record whether c is on a mount point */
c->ismtpt = (m!=nil);
```

Uses domount() 355, incref() 87b, pathclose() 325c, and uniquechan() 297a.

⟨namec() *case Aremove, Aaccess, handle mountpoint part2* 354c⟩≡ (305f)

```
putmhead(m);
```

Uses putmhead() 344c.

⟨*function* findmount 354d⟩≡ (808c)

```
/* also used by sysfile.c:/^mountfix */
bool
findmount(Chan **cp, Mhead **mp, int type, int dev, Qid qid)
{
    Pgrp *pg;
    Mhead *m;

    pg = up->pgrp;
    rlock(&pg->ns);

    // *mp = lookup(gqid, pg.mnthash); *cp = mp->mount->to
    for(m = MOUNTH(pg, qid); m; m = m->hash){
        rlock(&m->lock);
        if(m->from == nil){
            print("m %p m->from 0\n", m);
            runlock(&m->lock);
            continue;
        }
```

```
        if(eqchantdqid(m->from, type, dev, qid, true)){
            runlock(&pg->ns);
            if(mp != nil){
                if(*mp != nil)
                    putmhead(*mp);
                *mp = m;
                incref(m);
            }
            if(*cp != nil)
                cclose(*cp);
            *cp = m->mount->to;
            incref(m->mount->to);
            runlock(&m->lock);
            return true;
        }
        runlock(&m->lock);
    }

    runlock(&pg->ns);
    return false;
}
```

Uses MOUNTH 346d, cclose, eqchantdqid() 71a, false, incref() 87b, print, putmhead() 344c, rlock() 83e, runlock() 84b, true, and up.

⟨*function* domount 355⟩≡ (808c)

```
/*
 * Calls findmount but also updates path.
 */
static void
domount(Chan **cp, Mhead **mp, Path **path)
{
    Chan **lc;
    Path *p;

    if(!findmount(cp, mp, (*cp)->type, (*cp)->dev, (*cp)->qid))
        return;

    if(path){
        p = *path;
        p = uniquepath(p);
        if(p->mlen <= 0)
            print("domount: path %s has mlen==%d\n", p->s, p->mlen);
        else{
            lc = &p->mtpt[p->mlen-1];
            DBG("domount %p %s => add %p (was %p)\n", p, p->s, (*mp)->from, p->mtpt[p->mlen-1]);
            if(*lc)
                cclose(*lc);
            *lc = (*mp)->from;
            incref((*mp)->from);
        }
        *path = p;
    }
    return;
}
```

Uses DBG-765 808b, cclose, findmount() 354d, incref() 87b, print, and uniquepath() 325d.

14.9 Path mount points and ’..’

The trickiest part of namespace handling is “..” (dot-dot). If `/usr` is a mount point, then “`cd /usr/pad; cd ..`” should return to `/`, not to the root of whatever device is mounted on `/usr`. Each `\hlinkPath` carries an `mtpt` array that records, for each path element, the “from” channel of any mount point crossed. When `\hlinkwalk` processes “..”,`\hlinkundomount` uses this array to return to the channel that was mounted upon.

⟨Path *mtpt fields* 356a⟩≡ (324d)
```
// array<option<ref_counted<Chan>>, nil for elements which are not mount point
Chan  **mtpt;     /* mtpt history */
int mlen;     /* number of path elements */
int malen;      /* allocated length of mtpt */
```

⟨*constant* PATHMSLOP 356b⟩≡ (808c)
```
PATHMSLOP   = 20,
```

⟨newpath() *mtpt handling* 356c⟩≡ (324f)
```
/*
 * Cannot use newpath for arbitrary names because the mtpt
 * array will not be populated correctly.  The names #/ and / are
 * allowed, but other names with / in them draw warnings.
 */
if(strchr(s, '/') && strcmp(s, "#/") != 0 && strcmp(s, "/") != 0)
    print("newpath: %s from %#p\n", s, getcallerpc(&s));

p->mlen = 1;
p->malen = PATHMSLOP;
p->mtpt = smalloc(p->malen*sizeof p->mtpt[0]); // smalloc set p->mtpt[0] = nil
```
Uses PATHMSLOP-767 356b, print, and smalloc() 104b.

⟨copypath() *mtpt handling* 356d⟩≡ (325b)
```
pp->mlen = p->mlen;
pp->malen = p->malen;
pp->mtpt = smalloc(p->malen*sizeof pp->mtpt[0]);
for(i=0; i<pp->mlen; i++){
    pp->mtpt[i] = p->mtpt[i];
    if(pp->mtpt[i])
        incref(pp->mtpt[i]);
}
```
Uses incref() 87b and smalloc() 104b.

⟨pathclose() *mtpt handling* 356e⟩≡ (325c)
```
for(i=0; i<p->mlen; i++)
    if(p->mtpt[i])
        cclose(p->mtpt[i]);
free(p->mtpt);
```
Uses cclose and free() 104a.

⟨pathclose() *debugging* 356f⟩≡ (325c)
```
    DBG("pathclose %p %s ref=%ld =>", p, p->s, p->ref);
    for(i=0; i<p->mlen; i++)
        DBG(" %p", p->mtpt[i]);
    DBG("\n");
```
Uses DBG-765 808b.

⟨addelem() *mtpt handling* 357a⟩≡ (326a)

```
if(isdotdot(s)){
    DBG("addelem %s .. => rm %p\n", p->s, p->mtpt[p->mlen-1]);
    if(p->mlen>1 && (c = p->mtpt[--p->mlen])){
        p->mtpt[p->mlen] = nil;
        cclose(c);
    }
}else{
    if(p->mlen >= p->malen){
        // realloc
        p->malen = p->mlen+1+PATHMSLOP;
        tt = smalloc(p->malen*sizeof tt[0]);
        memmove(tt, p->mtpt, p->mlen*sizeof tt[0]);
        free(p->mtpt);
        p->mtpt = tt;
    }
    DBG("addelem %s %s => add %p\n", p->s, s, from);
    p->mtpt[p->mlen++] = from;
    if(from)
        incref(from);
}
```

Uses DBG-765 808b, PATHMSLOP-767 356b, cclose, incref() 87b, isdotdot() 326b, and smalloc() 104b.

⟨*function* undomount 357b⟩≡ (808c)

```
/*
 * If c is the right-hand-side of a mount point, returns the left hand side.
 * Changes name to reflect the fact that we've uncrossed the mountpoint,
 * so name had better be ours to change!
 */
static Chan*
undomount(Chan *c, Path *path)
{
    Chan *nc;

    if(path->ref != 1 || path->mlen == 0)
        print("undomount: path %s ref %ld mlen %d caller %#p\n",
            path->s, path->ref, path->mlen, getcallerpc(&c));

    if(path->mlen>0 && (nc=path->mtpt[path->mlen-1]) != nil){
        DBG("undomount %p %s => remove %p\n", path, path->s, nc);
        cclose(c);
        path->mtpt[path->mlen-1] = nil;
        // c = nc = mh->from
        c = nc;
    }
    return c;
}
```

Uses DBG-765 808b, cclose, and print.

14.10 Union mount

Union mounts allow multiple directories to appear merged at a single mount point. When walking a name through a union directory, if the first mount's device walk fails (the name is not found there), `\hlinkwalk` tries the next mount in the list, and so on. Reading a union directory is handled by `\hlinkunionread`, which reads from each mount's channel in sequence, advancing to the next when one is exhausted.

14.10.1 Union walk

⟨walk() *locals* 358a⟩+≡ (328f) ◁328d
```
Mount *f;
```

⟨walk() *if c is a union mount, find and adjust type and dev* 358b⟩≡ (328f)
```
/* try a union mount, if any */
if(mh && !sharppath){
    /*
     * mh->mount->to == c, so start at mh->mount->next
     */
    rlock(&mh->lock);
    f = mh->mount;
    for(f = (f? f->next: f); f; f = f->next)
        if((wq = ewalk(f->to, names+nhave, ntry)) != nil)
            break;
    runlock(&mh->lock);
    if(f != nil){
        type = f->to->type;
        dev = f->to->dev;
    }
}
```
Uses ewalk() 330a, rlock() 83e, and runlock() 84b.

14.10.2 ->umh

⟨namec() *case Aopen, Acreate, handle mountpoint part2* 358c⟩≡ (305f)
```
/* only save the mount head if it's a multiple element union */
if(m && m->mount && m->mount->next)
    c->umh = m;
else
    putmhead(m);
```
Uses putmhead() 344c.

⟨Chan *other fields* 358d⟩+≡ (68b) ◁295a 359b▷
```
Mhead*  umh;      /* mount point that derived Chan; used in unionread */
```

⟨chanfree() *optional free* 358e⟩≡ (296) 359e▷
```
if(c->umh != nil){
    putmhead(c->umh);
    c->umh = nil;
}
```
Uses putmhead() 344c.

⟨namec() *print error if c-¿umh != nil* 358f⟩≡ (305f)
```
if(c->umh != nil){
    print("uniquechan umh Open\n");
    putmhead(c->umh);
    c->umh = nil;
}
```
Uses print and putmhead() 344c.

⟨walk() *print error if c-¿umh != nil* 358g⟩≡ (328f)
```
if(c->umh != nil){  //BUG
    print("walk umh\n");
    putmhead(c->umh);
    c->umh = nil;
}
```
Uses print and putmhead() 344c.

⟨cmount() *print error if old-¿umh != nil* 359a⟩≡ (350c)
```
if(old->umh)
    print("cmount: unexpected umh, caller %#p\n", getcallerpc(&new));
```
Uses print.

14.10.3 syspread() for directories

Reading a directory on a mount point requires extra bookkeeping. The channel has two offsets: c->offset (user-visible) and c->devoffset (what the device sees), because \hlinkmountfix rewrites entries to reflect the current namespace, changing sizes. Overflow entries are saved in c->dirrock for the next read.

⟨Chan *other fields* 359b⟩+≡ (68b) ◁358d 359d▷
```
vlong devoffset;    /* in underlying device; see read */
```

⟨read() *if c is a QTDIR* 359c⟩≡ (313b)
```
    if(c->qid.type & QTDIR){
        if(mountrockread(c, p, n, &nn)){
            /* do nothing: mountrockread filled buffer */
        }else if(c->umh){
            nn = unionread(c, p, n);
        }else{
            if(off != c->offset)
                error(Edirseek);
            nn = devtab[c->type]->read(c, p, n, c->devoffset);
        }
        nnn = mountfix(c, p, nn, n);
    }
```
Uses Edirseek 664d, QTDIR 70b, mountfix() 371, mountrockread() 370a, and unionread() 359f.

⟨Chan *other fields* 359d⟩+≡ (68b) ◁359b 365b▷
```
Chan* umc;      /* channel in union; held for union read */
QLock umqlock;    /* serialize unionreads */
int uri;      /* union read index */
```

⟨chanfree() *optional free* 359e⟩+≡ (296) ◁358e 368b▷
```
if(c->umc != nil){
    cclose(c->umc);
    c->umc = nil;
}
```
Uses cclose.

\hlinkunionread reads a union directory by iterating through the mount list. It lazily opens each mount's channel (c->umc) as needed and reads until empty, then advances to the next mount via c->uri. If a mount's device raises an error, that component is silently skipped—a pragmatic choice since one bad server should not break the whole union. The umqlock serializes reads because the state (uri, umc) is shared across concurrent readers.

⟨*function* unionread 359f⟩≡ (815)
```
long
unionread(Chan *c, virt_vp va, long n)
{
    int i;
    long nr;
    Mhead *m;
    Mount *mount;

    qlock(&c->umqlock);
    m = c->umh;
```

```
    rlock(&m->lock);
    mount = m->mount;

    /* bring mount in sync with c->uri and c->umc */
    for(i = 0; mount != nil && i < c->uri; i++)
        mount = mount->next;

    nr = 0;
    while(mount != nil){
        /* Error causes component of union to be skipped */
        if(mount->to && !waserror()){
            if(c->umc == nil){
                c->umc = cclone(mount->to);
                c->umc = devtab[c->umc->type]->open(c->umc, OREAD);
            }

            nr = devtab[c->umc->type]->read(c->umc, va, n, c->umc->offset);
            c->umc->offset += nr;
            poperror();
        }
        if(nr > 0)
            break;

        /* Advance to next element */
        c->uri++;
        if(c->umc){
            cclose(c->umc);
            c->umc = nil;
        }
        mount = mount->next;
    }
    runlock(&m->lock);
    qunlock(&c->umqlock);
    return nr;
}
```

Uses OREAD 853i, cclone() 332a, cclose, poperror 666d, qlock() 81f, qunlock() 82c, rlock() 83e, runlock() 84b, and waserror 666d.

⟨read() *rewind when off == 0* 360a⟩≡ (313b)

```
/*
 * The offset is passed through on directories, normally.
 * Sysseek complains, but pread is used by servers like exportfs,
 * that shouldn't need to worry about this issue.
 *
 * Notice that c->devoffset is the offset that c's dev is seeing.
 * The number of bytes read on this fd (c->offset) may be different
 * due to rewritings in rockfix.
 */
    mountrewind(c);
    unionrewind(c);
```

Uses mountrewind() 370b and unionrewind() 360b.

⟨*function* unionrewind 360b⟩≡ (815)

```
static void
unionrewind(Chan *c)
{
    qlock(&c->umqlock);
    c->uri = 0;
    if(c->umc){
        cclose(c->umc);
```

```
            c->umc = nil;
        }
        qunlock(&c->umqlock);
    }
```

Uses cclose, qlock() 81f, and qunlock() 82c.

14.10.4 MCREATE

In a union directory, only the mount entry with the MCREATE flag can receive new files. \hlinkcreatedir walks the union mount list to find it. This is why bind -c exists: without the -c flag, you cannot create files in the mounted directory.

⟨namec() *Acreate case, if cnew is a mountpoint* 361a⟩≡ (333d)

```
    if(!sharppath && findmount(&cnew, &m, c->type, c->dev, c->qid))
        cnew = createdir(cnew, m);
```

Uses createdir() 361b and findmount() 354d.

⟨*function* createdir 361b⟩≡ (808c)

```
    /*
     * c is a mounted non-creatable directory.  find a creatable one.
     */
    Chan*
    createdir(Chan *c, Mhead *m)
    {
        Chan *nc;
        Mount *f;

        rlock(&m->lock);
        if(waserror()){
            runlock(&m->lock);
            nexterror();
        }
        for(f = m->mount; f; f = f->next){
            if(f->mflag&MCREATE){
                nc = cclone(f->to);
                runlock(&m->lock);
                poperror();
                cclose(c);
                return nc;
            }
        }
        error(Enocreate);
        panic("createdir: should not reach this point");
        return nil; // unreachable
    }
```

Uses Enocreate 664d, MCREATE 343, cclone() 332a, cclose, nexterror, poperror 666d, rlock() 83e, runlock() 84b, and waserror 666d.

14.10.5 Union of union

When binding a union directory onto another location, \hlinkcmount must copy the entire union mount list from the source, so the new location inherits all the union members. The umh field on a channel records the \hlinkMhead that produced this channel, so \hlinkcmount can walk the source union's mount chain.

⟨namec() *set c-¿umh in Abind if mounted point* 361c⟩≡ (349d)

```
        if(c->umh != nil)
            putmhead(c->umh);
```

```
        c->umh = m;
```
Uses putmhead() 344c.

⟨cmount() *locals* 362a⟩+≡ (350c) ◁350b
```
  int flg;
  Mhead *umh, **l;
  Mount *f, *um, **h;
```

⟨cmount() *if new is itself a mount point, error if cant create there* 362b⟩≡ (350c)
```
  umh = new->umh;
  /*
   * Not allowed to bind when the old directory is itself a union.
   * (Maybe it should be allowed, but I don't see what the semantics
   * would be.)
   *
   * We need to check umh->mount->next to tell unions apart from
   * simple mount points, so that things like
   *  mount -c fd /root
   *  bind -c /root /
   * work.
   *
   * The check of mount->mflag allows things like
   *  mount fd /root
   *  bind -c /root /
   *
   * This is far more complicated than it should be, but I don't
   * see an easier way at the moment.
   */
  if((flag&MCREATE) && umh && umh->mount
     && (umh->mount->next || !(umh->mount->mflag&MCREATE)))
      error(Emount);
```
Uses Emount 664d and MCREATE 343.

⟨cmount() *if new is itself a mount point, copy mounts* 362c⟩≡ (350c)
```
  if(umh != nil && umh->mount != nil){
      /*
       *  copy a union when binding it onto a directory
       */
      flg = order;
      if(order == MREPL)
          flg = MAFTER;
      h = &m->next;
      um = umh->mount;
      for(um = um->next; um; um = um->next){
          f = newmount(um->to, flg, um->spec);
          *h = f;
          h = &f->next;
      }
  }
```
Uses MAFTER 343, MREPL 853i, and newmount() 345b.

14.11 Sandboxing

The noattach flag provides namespace sandboxing: when set via RFNOMNT, the process cannot mount new file servers or access most "#" device paths. Only safe devices (pipes, fds, environment, console, proc) are allowed through.

⟨Pgrp *other fields* 362d⟩+≡ (346b) ◁347b
```
  bool noattach;
```

⟨bindmount() *error if noattach* 363a⟩≡ (364b)
```
if(up->pgrp->noattach)
    error(Enoattach);
```
Uses Enoattach 664d and up.

⟨namec() *noattach checking* 363b⟩≡ (348b)
```
/*
 *  noattach is sandboxing.
 *
 *  the OK exceptions are:
 *  |  it only gives access to pipes you create
 *  d  this process's file descriptors
 *  e  this process's environment
 *  the iffy exceptions are:
 *  c  time and pid, but also cons and consctl
 *  p  control of your own processes (and unfortunately
 *     any others left unprotected)
 */
/* actually / is caught by parsing earlier */
if(utfrune("M", r))
    error(Enoattach);
if(up->pgrp->noattach && utfrune("|decp", r)==nil)
    error(Enoattach);
```
Uses Enoattach 664d and up.

⟨sysrfork() *inherit noattach, RFPROC==0 case* 363c⟩≡ (139e)
```
/* inherit noattach */
up->pgrp->noattach = opg->noattach;
```
Uses up.

⟨sysrfork() *inherit noattach, RFPROC==1 case* 363d⟩≡ (137d)
```
/* inherit noattach */
p->pgrp->noattach = up->pgrp->noattach;
```
Uses up.

⟨sysrfork() *set noattach to true when RFNOMNT, RFPROC==0 case* 363e⟩≡ (139e)
```
if(flag & RFNOMNT)
    up->pgrp->noattach = true;
```
Uses RFNOMNT-2560 133b, true, and up.

⟨sysrfork() *set noattach to true when RFNOMNT, RFPROC==1 case* 363f⟩≡ (137d)
```
if(flag & RFNOMNT)
    p->pgrp->noattach = true;
```
Uses RFNOMNT-2560 133b and true.

14.12 sysmount()

While bind connects two local paths, mount connects a file descriptor (typically a pipe to a file server) to a path, using the mount device (devmnt) to speak 9P over the fd.

⟨*syscall mount* 363g⟩≡ (815)
```
// int mount(int fd, int afd, char *old, int flag, char *aname);
long
sysmount(ulong* arg)
{
    return bindmount(true, arg[0], arg[1], nil, (char*)arg[2], arg[3], (char*)arg[4]);
}
```
Uses bindmount() 349a and true.

⟨bindmount() *locals* 364a⟩+≡ (349a) ◁349b
```
Chan *ac, *bc;

struct Bogus bogus;
```
Uses Bogus 804b.

The mount case is more complex than bind. It converts the file descriptor `fd` into a channel, then attaches to the mount device (`#M`) which will speak 9P over that channel. The `afd` parameter is an optional authentication channel. The `spec` string lets you select a subtree from the remote server (e.g., mounting just `/usr` from a file server).

⟨bindmount() *if ismount* 364b⟩≡ (349a)
```
if(ismount){
    validaddr((ulong)spec, 1, false);
    spec = validnamedup(spec, true);
    if(waserror()){
        free(spec);
        nexterror();
    }
    ⟨bindmount() error if noattach 363a⟩

    ac = nil;
    bc = fdtochan(fd, ORDWR, false, true);
    if(waserror()) {
        if(ac)
            cclose(ac);
        cclose(bc);
        nexterror();
    }

    if(afd >= 0)
        ac = fdtochan(afd, ORDWR, false, true);

    bogus.flags = flag & MCACHE;
    bogus.chan = bc;
    bogus.authchan = ac;
    bogus.spec = spec;
    ret = devno('M', false);
    c0 = devtab[ret]->attach((char*)&bogus);
    poperror(); /* ac bc */
    if(ac)
        cclose(ac);
    cclose(bc);
}
```
Uses MCACHE 343, ORDWR 301c, cclose, devno() 297c, false, fdtochan() 302d, free() 104a, nexterror, poperror 666d, true, validaddr() 120b, validnamedup() 307e, and waserror 666d.

⟨bindmount() *if ismount free* 364c⟩≡ (349a)
```
if(ismount){
    fdclose(fd, 0);
    poperror();
    free(spec);
}
```
Uses fdclose() 303b, free() 104a, and poperror 666d.

⟨MountFlags *cases* 364d⟩≡ (343)
```
MCACHE = 0x0010,  /* cache some data */
```

14.13 sysumount()

\hlinkcunmount searches the \hlinkPgrp hash and removes either the entire mount point or a specific entry from the union list.

⟨*syscall unmount* 365a⟩≡ (815)
```
// int unmount(char *name, char *old);
long
sysunmount(ulong* arg)
{
    Chan *cmount, *cmounted;

    cmounted = nil;

    validaddr(arg[1], 1, false);
    cmount = namec((char *)arg[1], Amount, 0, 0);
    if(waserror()) {
        cclose(cmount);
        if(cmounted)
            cclose(cmounted);
        nexterror();
    }

    if(arg[0]) {
        /*
         * This has to be namec(..., Aopen, ...) because
         * if arg[0] is something like /srv/cs or /fd/0,
         * opening it is the only way to get at the real
         * Chan underneath.
         */
        validaddr(arg[0], 1, false);
        cmounted = namec((char*)arg[0], Aopen, OREAD, 0);
    }
    cunmount(cmount, cmounted);
    poperror();
    cclose(cmount);
    if(cmounted)
        cclose(cmounted);
    return 0;
}
```
Uses Amount 349c, Aopen 305b, OREAD 853i, cclose, cunmount() 365d, false, namec() 305f, nexterror, poperror 666d, validaddr() 120b, and waserror 666d.

⟨Chan *other fields* 365b⟩+≡ (68b) ◁359d 369b▷
```
Chan* mchan;      /* channel to mounted server */
```

⟨cunmount() *print error if mnt-¿umh* 365c⟩≡ (365d)
```
if(mnt->umh)    /* should not happen */
    print("cunmount newp extra umh %p has %p\n", mnt, mnt->umh);
```
Uses print.

\hlinkcunmount has two modes: if mounted is nil, it removes the entire mount point (the \hlinkMhead and all its mounts). If mounted is given, it removes only that specific entry from the union list. When the last entry in a union is removed, the \hlinkMhead itself is cleaned up.

⟨*function* cunmount 365d⟩≡ (808c)
```
void
cunmount(Chan *mnt, Chan *mounted)
{
    Pgrp *pg;
```

```
Mhead *m, **l;
Mount *f, **p;
```

⟨cunmount() *print error if mnt-¿umh* 365c⟩

```
/*
 * It _can_ happen that mounted->umh is non-nil,
 * because mounted is the result of namec(Aopen)
 * (see sysfile.c:/^sysunmount).
 * If we open a union directory, it will have a umh.
 * Although surprising, this is okay, since the
 * cclose will take care of freeing the umh.
 */

pg = up->pgrp;
wlock(&pg->ns);

l = &MOUNTH(pg, mnt->qid);
for(m = *l; m; m = m->hash){
    if(eqchan(m->from, mnt, true))
        break;
    l = &m->hash;
}

if(m == nil){
    wunlock(&pg->ns);
    error(Eunmount);
}

wlock(&m->lock);
if(mounted == nil){
    *l = m->hash;
    wunlock(&pg->ns);
    mountfree(m->mount);
    m->mount = nil;
    cclose(m->from);
    wunlock(&m->lock);
    putmhead(m);
    return;
}

p = &m->mount;
for(f = *p; f; f = f->next){
    /* BUG: Needs to be 2 pass */
    if(eqchan(f->to, mounted, true) ||
      (f->to->mchan && eqchan(f->to->mchan, mounted, true))){
        *p = f->next;
        f->next = nil;
        mountfree(f);
        if(m->mount == nil){
            *l = m->hash;
            cclose(m->from);
            wunlock(&m->lock);
            wunlock(&pg->ns);
            putmhead(m);
            return;
        }
        wunlock(&m->lock);
        wunlock(&pg->ns);
        return;
```

```
        }
        p = &f->next;
    }
    wunlock(&m->lock);
    wunlock(&pg->ns);
    error(Eunion);
}
```

Uses Eunion 664d, Eunmount 664d, MOUNTH 346d, cclose, eqchan() 70e, mountfree() 346a, putmhead() 344c, true, up, wlock() 85b, and wunlock() 85d.

14.14 XTODO

Each \hlinkMount gets a globally unique mountid from a counter. This serves two purposes: \hlinkmntscan uses it to iterate the mount table in creation order (for /proc/\#/ns), and \hlinkpgrpcpy uses it to preserve ordering when duplicating a namespace for RFNAMEG. The order and copy fields are scratch pointers used only during \hlinkpgrpcpy.

⟨Mount *other fields* 367a⟩≡ (345a) 367c▷

```
ulong mountid;
```

⟨*global* mountid 367b⟩≡ (875)

```
static Counter mountid;
```

⟨Mount *other fields* 367c⟩+≡ (345a) ◁367a

```
// list<ref<Mount>> sorted list by mount->mountid
Mount*  order;
```

⟨Mount *extra fields* 367d⟩+≡ (345a) ◁345c

```
Mount*  copy;
```

⟨*function* pgrpinsert 367e⟩≡ (875)

```
void
pgrpinsert(Mount **order, Mount *m)
{
    Mount *f;

    m->order = nil;
    if(*order == nil) {
        *order = m;
        return;
    }
    for(f = *order; f; f = f->order) {
        if(m->mountid < f->mountid) {
            m->order = f;
            *order = m;
            return;
        }
        order = &f->order;
    }
    *order = m;
}
```

`\hlinkpgrpcpy` duplicates a namespace for `RFNAMEG`. It walks the entire mount table hash, cloning every `\hlinkMhead` and `\hlinkMount`. The tricky part is preserving the `mountid` order: mount ids are allocated globally and must increase monotonically, so the copy collects all old mounts in sorted order via `\hlinkpgrpinsert` and then assigns new ids in the same sequence.

⟨*function* pgrpcpy 368a⟩≡ (875)

```
/*
 * pgrpcpy MUST preserve the mountid allocation order of the parent group
 */
void
pgrpcpy(Pgrp *to, Pgrp *from)
{
    int i;
    Mount *n, *m, **link, *order;
    Mhead *f, **tom, **l, *mh;

    wlock(&from->ns);
    order = nil;
    tom = to->mnthash;
    for(i = 0; i < MNTHASH; i++) {
        l = tom++;
        for(f = from->mnthash[i]; f; f = f->hash) {
            rlock(&f->lock);
            mh = newmhead(f->from);
            *l = mh;
            l = &mh->hash;
            link = &mh->mount;
            for(m = f->mount; m; m = m->next) {
                n = newmount(m->to, m->mflag, m->spec);

                m->copy = n;
                pgrpinsert(&order, m);

                *link = n;
                link = &n->next;
            }
            runlock(&f->lock);
        }
    }
    /*
     * Allocate mount ids in the same sequence as the parent group
     */
    lock(&mountid);
    for(m = order; m; m = m->order)
        m->copy->mountid = mountid.ref++;
    unlock(&mountid);
    wunlock(&from->ns);
}
```

Uses `MNTHASH` 346d, `lock()` 76d, `mountid-2606` 367b, `newmhead()` 344b, `newmount()` 345b, `pgrpinsert()` 367e, `rlock()` 83e, `runlock()` 84b, `unlock()` 78d, `wlock()` 85b, and `wunlock()` 85d.

14.15 XTODO

⟨`chanfree()` *optional free* 368b⟩+≡ (296) ◁359e 369a▷

```
if(c->mux != nil){
    muxclose(c->mux);
    c->mux = nil;
}
```

```
if(c->mchan != nil){
    cclose(c->mchan);
    c->mchan = nil;
}
```

Uses cclose and muxclose() 582a.

⟨chanfree() *optional free* 369a⟩+≡ (296) ◁368b

```
if(c->dirrock != nil){
    free(c->dirrock);
    c->dirrock = nil;
    c->nrock = 0;
    c->mrock = 0;
}
```

Uses free() 104a.

⟨Chan *other fields* 369b⟩+≡ (68b) ◁365b 432b▷

```
byte*  dirrock;    /* directory entry rock for translations */
int nrock;
int mrock;
QLock rockqlock;
```

The "rock" mechanism handles a subtle problem with directory reads across mount points. \hlinkmountfix rewrites directory entries to reflect the current namespace (e.g., changing device-relative names to mount-relative names), which can change entry sizes. If the rewritten entries overflow the user buffer, the excess is stashed in c->dirrock and returned on the next read call via \hlinkmountrockread. The name "rock" comes from Plan 9 slang for a stash of saved state.

⟨*function* mountrock 369c⟩≡ (815)

```
/*
 * Mountfix might have caused the fixed results of the directory read
 * to overflow the buffer.  Catch the overflow in c->dirrock.
 */
static void
mountrock(Chan *c, byte *p, byte **pe)
{
    byte *e, *r;
    int len, n;

    e = *pe;

    /* find last directory entry */
    for(;;){
        len = BIT16SZ+GBIT16(p);
        if(p+len >= e)
            break;
        p += len;
    }

    /* save it away */
    qlock(&c->rockqlock);
    if(c->nrock+len > c->mrock){
        n = ROUND(c->nrock+len, 1024);
        r = smalloc(n);
        memmove(r, c->dirrock, c->nrock);
        free(c->dirrock);
        c->dirrock = r;
        c->mrock = n;
    }
    memmove(c->dirrock+c->nrock, p, len);
    c->nrock += len;
```

```
    qunlock(&c->rockqlock);

    /* drop it */
    *pe = p;
}
```

Uses ROUND 47b, free() 104a, qlock() 81f, qunlock() 82c, and smalloc() 104b.

⟨*function* mountrockread 370a⟩≡ (815)

```
/*
 * Satisfy a directory read with the results saved in c->dirrock.
 */
static bool
mountrockread(Chan *c, byte *op, long n, long *nn)
{
    long dirlen;
    byte *rp, *erp, *ep, *p;

    /* common case */
    if(c->nrock == 0)
        return false;

    /* copy out what we can */
    qlock(&c->rockqlock);
    rp = c->dirrock;
    erp = rp+c->nrock;
    p = op;
    ep = p+n;
    while(rp+BIT16SZ <= erp){
        dirlen = BIT16SZ+GBIT16(rp);
        if(p+dirlen > ep)
            break;
        memmove(p, rp, dirlen);
        p += dirlen;
        rp += dirlen;
    }

    if(p == op){
        qunlock(&c->rockqlock);
        return false;
    }

    /* shift the rest */
    if(rp != erp)
        memmove(c->dirrock, rp, erp-rp);
    c->nrock = erp - rp;

    *nn = p - op;
    qunlock(&c->rockqlock);
    return true;
}
```

Uses false, qlock() 81f, qunlock() 82c, and true.

⟨*function* mountrewind 370b⟩≡ (815)

```
static void
mountrewind(Chan *c)
{
    c->nrock = 0;
}
```

`\hlinkmountfix` is the postprocessor for directory reads. For each entry in the read buffer, it checks if that entry corresponds to a mount point. If so, it re-stats the mounted channel to get fresh metadata (size, permissions, modification time) while preserving the original name. This is necessary because a directory listing of `/dev` should show the attributes of the mounted device files, not the underlying directory entries. The buffer surgery (shifting entries to accommodate size changes) can overflow into the "rock" buffer via `\hlinkmountrock`.

⟨*function* `mountfix` 371⟩≡ (815)

```
/*
 * Rewrite the results of a directory read to reflect current
 * name space bindings and mounts.  Specifically, replace
 * directory entries for bind and mount points with the results
 * of statting what is mounted there.  Except leave the old names.
 */
static long
mountfix(Chan *c, byte *op, long n, long maxn)
{
    char *name;
    int nbuf, nname;
    Chan *nc;
    Mhead *mh;
    Mount *m;
    byte *p;
    int dirlen, rest;
    long l;
    byte *buf, *e;
    DirEntry d;

    p = op;
    buf = nil;
    nbuf = 0;
    for(e=&p[n]; p+BIT16SZ<e; p+=dirlen){
        dirlen = dirfixed(p, e, &d);
        if(dirlen < 0)
            break;
        nc = nil;
        mh = nil;
        if(findmount(&nc, &mh, d.type, d.dev, d.qid)){
            /*
             * If it's a union directory and the original is
             * in the union, don't rewrite anything.
             */
            for(m=mh->mount; m; m=m->next)
                if(eqchantdqid(m->to, d.type, d.dev, d.qid, true))
                    goto Norewrite;

            name = offsetof_name_direntry(p, &nname);
            /*
             * Do the stat but fix the name.  If it fails, leave old entry.
             * BUG: If it fails because there isn't room for the entry,
             * what can we do?  Nothing, really.  Might as well skip it.
             */
            if(buf == nil){
                buf = smalloc(4096);
                nbuf = 4096;
            }
            if(waserror())
                goto Norewrite;
            l = devtab[nc->type]->stat(nc, buf, nbuf);
            l = direntry_setname(name, nname, buf, l, nbuf);
            if(l == BIT16SZ)
```

```
                error("direntry_setname");
            poperror();

            /*
             * Shift data in buffer to accomodate new entry,
             * possibly overflowing into rock.
             */
            rest = e - (p+dirlen);
            if(l > dirlen){
                while(p+l+rest > op+maxn){
                    mountrock(c, p, &e);
                    if(e == p){
                        dirlen = 0;
                        goto Norewrite;
                    }
                    rest = e - (p+dirlen);
                }
            }
            if(l != dirlen){
                memmove(p+l, p+dirlen, rest);
                dirlen = l;
                e = p+dirlen+rest;
            }

            /*
             * Rewrite directory entry.
             */
            memmove(p, buf, l);

            Norewrite:
            cclose(nc);
            putmhead(mh);
        }
    }
    if(buf)
        free(buf);

    if(p != e)
        error("oops in rockfix");

    return e-op;
}
```

Uses cclose, direntry_setname() 337b, dirfixed() 372, eqchantdqid() 71a, findmount() 354d, free() 104a, mountrock() 369c, offsetof_name_direntry() 338a, poperror 666d, putmhead() 344c, smalloc() 104b, true, and waserror 666d.

⟨*function* dirfixed 372⟩≡ (815)

```
static int
dirfixed(byte *p, byte *e, DirEntry *d)
{
    int len;

    len = GBIT16(p)+BIT16SZ;
    if(p + len > e)
        return -1;

    p += BIT16SZ;   /* ignore size */
    d->type = devno(GBIT16(p), true);
    p += BIT16SZ;
    d->dev = GBIT32(p);
```

```
        p += BIT32SZ;
        d->qid.type = GBIT8(p);
        p += BIT8SZ;
        d->qid.vers = GBIT32(p);
        p += BIT32SZ;
        d->qid.path = GBIT64(p);
        p += BIT64SZ;
        d->mode = GBIT32(p);
        p += BIT32SZ;
        d->atime = GBIT32(p);
        p += BIT32SZ;
        d->mtime = GBIT32(p);
        p += BIT32SZ;
        d->length = GBIT64(p);

        return len;
    }
```

Uses `devno()` 297c and `true`.

Chapter 15

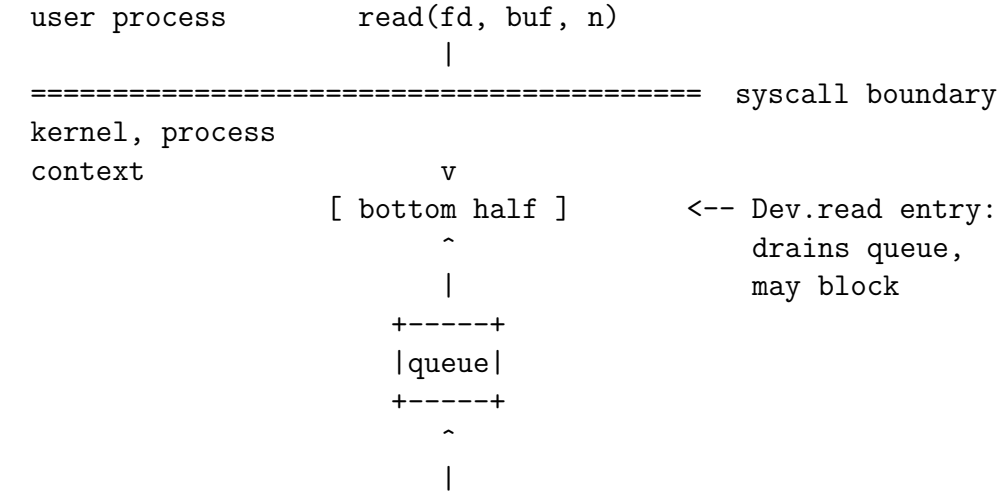
Devices

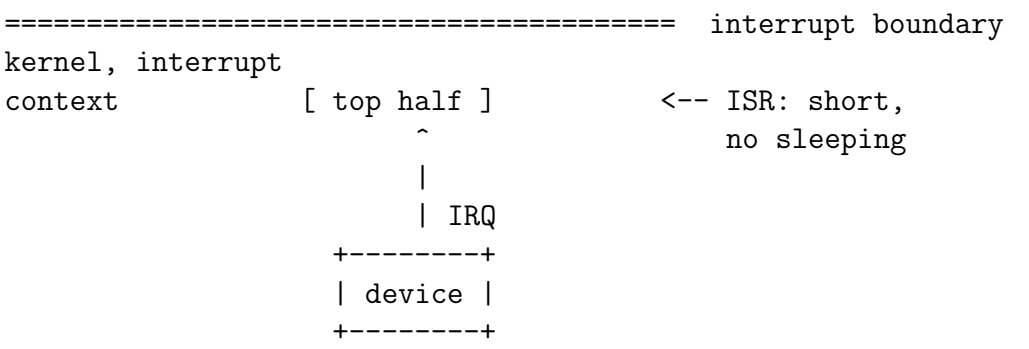
Device drivers are the largest part of any kernel, but also the most repetitive—each one follows the same `Dev` interface (see Chapter 17). This chapter presents only a few representative drivers: the keyboard (interrupt-driven scan code processing), the screen (framebuffer and display initialization), and the console device (`devcons`, which ties keyboard input to `/dev/cons`). For the keyboard and screen, I use the PC (x86) version rather than the Raspberry Pi version, because the PC hardware interface is simpler (direct I/O ports vs. USB).

15.1 Driver model principles

Every device driver sits between two worlds. On one side is the generic file-like interface every program sees: the `open/read/write/close` calls routed through the `Dev` operations table from Section 2.1.8. On the other side is one specific, usually baroque piece of hardware: the chipset's memory-mapped registers, its interrupt line, its DMA quirks, its bus-protocol handshake. The driver's job is to translate one side to the other. At this structural level, Linux, BSD, and Windows drivers are all identical—a struct of function pointers registered with the kernel at boot, plus an interrupt handler registered with the interrupt controller; only the calling conventions and the names differ.

The single most important design idea in a driver is that its code runs in two radically different contexts. The *top half*—also called the *interrupt service routine* (ISR) or *fast path*—runs the instant hardware fires an interrupt (Section 2.1.4). It runs with most interrupts disabled, cannot sleep, cannot block on locks that another driver might hold, and must return as fast as possible because every cycle it spends keeps the rest of the system frozen. The *bottom half*—called a *deferred procedure call* on Windows (DPC), *softirq/tasklet/workqueue* on Linux, *ithread* on FreeBSD—runs later, in ordinary kernel context, where it can sleep, grab sleeping locks, and do the heavy lifting the ISR could not. The two halves communicate via a lock-free or interrupt-safe queue that the top half produces into and the bottom half consumes from:





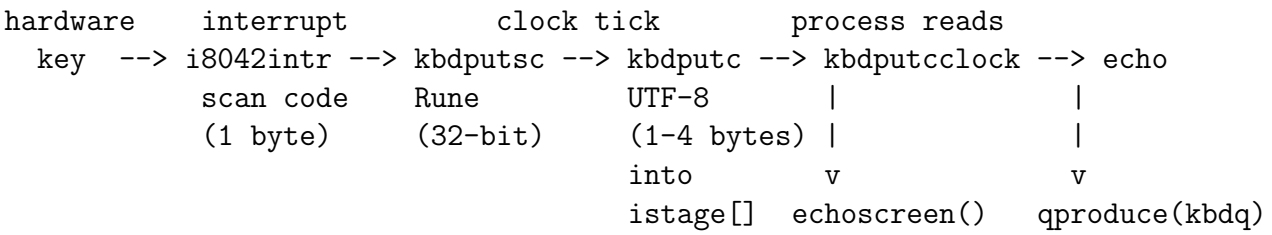
A driver has three ways to actually move data between the device and the kernel, and most real drivers combine all three. *Polling* (the CPU repeatedly reads a status register in a loop until the device says "ready") is the oldest and simplest scheme; it is still used for very-rare or latency-critical events, and during early boot before the interrupt controller itself is up. *Interrupt-driven* I/O (the device raises its IRQ line when something happens; the CPU runs the handler) is the default for keyboards, timers, and most network and storage devices, because it lets the CPU do useful work while the device is idle. DMA (*direct memory access*: the device is programmed with a RAM address and a byte count, and then transfers bytes *directly* between itself and memory, bypassing the CPU) is the choice for high-bandwidth devices such as disks, Ethernet, and graphics, because it moves data at memory speed rather than at the cost of a per-byte PIO load-store loop.

Plan 9's concrete version of this model is the one the Files/VFS subsection already previewed. Every driver is a struct of type `Dev` registered in a global `devtab` (Chapter 17), with a short prefix-letter name (`'#c'` for `devcons`, `'#l'` for `devether`, `'#m'` for `devmouse`, and so on). Its top half is an ordinary C function registered with the platform's interrupt code (via `intrenable` on ARM); its bottom half is the `read`/`write` entry in the `Dev` struct, called from user syscalls. The two halves communicate through `Queue`s (Chapter 16)—a Plan 9-specific, flow-controlled blocking byte queue designed for exactly this pattern. Of the three drivers this chapter walks through, the keyboard is the canonical ISR-plus-queue driver (top half, bottom half, and `Queue` all visible); the screen has no interrupts at all (it is purely a memory-mapped framebuffer); and the console is the glue that ties the first two into `/dev/cons`.

15.2 The keyboard

15.2.1 Overview

Keyboard input passes through a multi-stage pipeline before reaching a user process. When you press a key, the hardware generates a scan code (a raw byte identifying the physical key). The keyboard interrupt handler calls [kbdputsc] which translates that scan code into a Rune (a Unicode code point) by looking it up in one of several tables depending on the modifier state (Shift, Ctrl, Alt, Altgr). The Rune is then passed to [kbdputc], which converts it to UTF-8 bytes and stores them in a circular buffer (`kbd.istage`). A periodic clock callback ([kbdputcclock]) drains that buffer, calling [echo] which both echoes characters to the screen and enqueues them into `kbdq` via [qproduce]. The pipeline looks like this:



The reason for the intermediate `istage` circular buffer and the clock-driven drain is to amortize the cost of queue operations: calling [qproduce] on every single keystroke interrupt would be wasteful, so instead characters accumulate at interrupt time and get flushed in batch every 22 ms by the clock.

15.2.2 Scan code

The scan code tables map raw hardware bytes to Unicode code points (Runes). Special keys like Shift, Ctrl, and arrow keys cannot be represented as normal characters, so they are assigned values in Unicode's Private Use Area (starting at `0xF000`). This is a clever trick: modifier keys and function keys live in the same Rune space as regular characters, so the entire translation can be done with a single array lookup rather than complex conditional logic. The `Nscan` constant is 128 because the 8th bit of a scan code encodes key-up vs. key-down, leaving 7 bits for the key identity.

⟨*enum* `specialkey` 376a⟩≡ (708)

```
enum {
    KF=         0xF000,     /* function key (begin Unicode private space) */
    Spec=       0xF800,     /* Unicode private space */


    PF=         Spec|0x20,  /* num pad function key */
    View=       Spec|0x00,  /* view (shift window up) */

    Shift=      Spec|0x60,
    Break=      Spec|0x61,
    Ctrl=       Spec|0x62,
    Alt=        Spec|0x63,
    Caps=       Spec|0x64,
    Num=        Spec|0x65,
    Middle=     Spec|0x66,
    Altgr=      Spec|0x67,

    Kmouse=     Spec|0x100,

    No=         0x00,       /* peter */

    /* KF|1, KF|2, ..., KF|0xC is F1, F2, ..., F12 */

    Home=       KF|13,
    Up=         KF|14,
    Pgup=       KF|15,
    Print=      KF|16,
    Left=       KF|17,
    Right=      KF|18,
    End=        KF|24,
    Down=       View,
    Pgdown=     KF|19,
    Ins=        KF|20,
    Del=        0x7F,
    Scroll=     KF|21,

    Nscan=  128,
};
```

Uses `KF-959` 376a, `Spec-960` 376a, and `View-962` 376a.

⟨*global* `kbtab` 376b⟩≡ (708)

```
Rune kbtab[Nscan] =
{
[0x00]  No, 0x1b,   '1',    '2',    '3',    '4',    '5',    '6',
[0x08]  '7',    '8',    '9',    '0',    '-',    '=',    '\b',   '\t',
[0x10]  'q',    'w',    'e',    'r',    't',    'y',    'u',    'i',
[0x18]  'o',    'p',    '[',    ']',    '\n',   Ctrl,   'a',    's',
[0x20]  'd',    'f',    'g',    'h',    'j',    'k',    'l',    ';',
[0x28]  '\'',   '`',    Shift,  '\\',   'z',    'x',    'c',    'v',
```

```
  [0x30]  'b',    'n',    'm',    ',',    '.',    '/',    Shift,  '*',
  [0x38]  Alt,  ' ',    Ctrl,   KF|1,   KF|2,   KF|3,   KF|4,   KF|5,
  [0x40]  KF|6,   KF|7,   KF|8,   KF|9,   KF|10,  Num,    Scroll, '7',
  [0x48]  '8',    '9',    '-',    '4',    '5',    '6',    '+',    '1',
  [0x50]  '2',    '3',    '0',    '.',    No, No, No, KF|11,
  [0x58]  KF|12,  No, No, No, No, No, No, No,
  [0x60]  No, No, No, No, No, No, No, No,
  [0x68]  No, No, No, No, No, No, No, No,
  [0x70]  No, No, No, No, No, No, No, No,
  [0x78]  No, No, No, No, No, No, No, No,
  };
```

Uses Alt-966 376a, Ctrl-965 376a, KF-959 376a, No-972 376a, Nscan-985 376a, Num-968 376a, Scroll-984 376a, and Shift-963 376a.

⟨*global* kbtabshift 377a⟩≡ (708)

```
  Rune kbtabshift[Nscan] =
  {
  [0x00]  No, 0x1b,   '!',    '@',    '#',    '$',    '%',    '^',
  [0x08]  '&',    '*',    '(',    ')',    '_',    '+',    '\b',   '\t',
  [0x10]  'Q',    'W',    'E',    'R',    'T',    'Y',    'U',    'I',
  [0x18]  'O',    'P',    '{',    '}',    '\n',   Ctrl,   'A',    'S',
  [0x20]  'D',    'F',    'G',    'H',    'J',    'K',    'L',    ':',
  [0x28]  '"',    '~',    Shift,  '|',    'Z',    'X',    'C',    'V',
  [0x30]  'B',    'N',    'M',    '<',    '>',    '?',    Shift,  '*',
  [0x38]  Alt,  ' ',    Ctrl,   KF|1,   KF|2,   KF|3,   KF|4,   KF|5,
  [0x40]  KF|6,   KF|7,   KF|8,   KF|9,   KF|10,  Num,    Scroll, '7',
  [0x48]  '8',    '9',    '-',    '4',    '5',    '6',    '+',    '1',
  [0x50]  '2',    '3',    '0',    '.',    No, No, No, KF|11,
  [0x58]  KF|12,  No, No, No, No, No, No, No,
  [0x60]  No, No, No, No, No, No, No, No,
  [0x68]  No, No, No, No, No, No, No, No,
  [0x70]  No, No, No, No, No, No, No, No,
  [0x78]  No, No, No, No, No, No, No, No,
  };
```

Uses Alt-966 376a, Ctrl-965 376a, KF-959 376a, No-972 376a, Nscan-985 376a, Num-968 376a, Scroll-984 376a, and Shift-963 376a.

⟨*global kbtabctrl decl* 377b⟩≡ (708)

```
  extern Rune kbtabctrl[];
```

⟨*global* kbtabaltgr 377c⟩≡ (708)

```
  Rune kbtabaltgr[Nscan] =
  {
  [0x00]  No, No, No, No, No, No, No, No,
  [0x08]  No, No, No, No, No, No, No, No,
  [0x10]  No, No, No, No, No, No, No, No,
  [0x18]  No, No, No, No, '\n',   Ctrl,   No, No,
  [0x20]  No, No, No, No, No, No, No, No,
  [0x28]  No, No, Shift,  No, No, No, No, No,
  [0x30]  No, No, No, No, No, '/',    No, Print,
  [0x38]  Altgr,  No, No, No, No, No, No, No,
  [0x40]  No, No, No, No, No, No, Break,  Home,
  [0x48]  Up, Pgup,   No, Left,   No, Right,  No, End,
  [0x50]  Down,   Pgdown, Ins,    Del,    No, No, No, No,
  [0x58]  No, No, No, No, No, No, No, No,
  [0x60]  No, No, No, No, No, No, No, No,
  [0x68]  No, No, No, No, No, No, No, No,
  [0x70]  No, No, No, No, No, No, No, No,
  [0x78]  No, No, No, No, No, No, No, No,
  };
```

Uses Altgr-970 376a, Break-964 376a, Ctrl-965 376a, Del-983 376a, Down-980 376a, End-979 376a, Home-973 376a, Ins-982 376a, Left-977 376a, No-972 376a, Nscan-985 376a, Pgdown-981 376a, Pgup-975 376a, Print-976 376a, Right-978 376a, Shift-963 376a, and Up-974 376a.

⟨*struct* Kbscan 378a⟩≡ (706c)
```
struct Kbscan {
    bool ctl;
    bool shift;
    bool caps;
    bool alt;
    bool altgr;
    bool num;
    ⟨Kbscan other fields 381a⟩
};
```

The Kbscan structure tracks the current modifier key state for the scan code state machine. When a Shift key-down event arrives, kbscan->shift is set to true; when the key-up event arrives, it is cleared. The next regular key press then consults these flags to pick the right translation table (kbtab, kbtabshift, kbtabctrl, or kbtabaltgr). The kbscans array has two entries: one for the built-in keyboard (KbInt) and one for an external USB keyboard (KbExt), so each can track its modifier state independently.

⟨*enum* kbscan 378b⟩≡ (706c)
```
/* kbscans indices */
enum KbscanKind {
    KbInt=    0,
    KbExt,

    KbNscans,
};
```

⟨*global* kbscans 378c⟩≡ (708)
```
// hash<enum<KbscanKind>, Kbscan>
Kbscan kbscans[KbNscans]; /* kernel and external scan code state */
```
Uses KbNscans 378b.

[kbdputsc] is the heart of the scan code state machine. It takes a raw byte from the hardware interrupt and converts it to a Rune. The high bit of the byte indicates key-up (release) vs. key-down (press). For key-up events, only modifier keys matter – releasing a regular key has no effect, since we already processed it on key-down. For key-down events of modifier keys (Ctrl, Shift, Alt, Caps), the function updates the Kbscan state without producing any character. Only non-modifier key-down events result in a call to [kbdputc] to deposit the character.

⟨*function* kbdputsc 378d⟩≡ (708)
```
/*
 * Scan code processing
 */
void
kbdputsc(byte k, int external)
{
    bool keyup;
    Kbscan *kbscan;
    // Rune is (uint) but actually 'c' can get the result of functions
    // like latin1() which can return negative numbers.
    long c = k;

    if(external)
        kbscan = &kbscans[KbExt];
    else
        kbscan = &kbscans[KbInt];
```

⟨kbdputsc() *debugging* 641c⟩

⟨kbdputsc() *esc key handling part1 and possible return* 381b⟩

```
keyup = c & 0x80; // key released (1 bit)
c &= 0x7f; // 128
```

⟨kbdputsc() *ensures c is in boundary* 380b⟩

⟨kbdputsc() *esc key handling part2 and possible return* 381c⟩

```
else if(kbscan->shift)
    c = kbtabshift[c];
else if(kbscan->altgr)
    c = kbtabaltgr[c];
else if(kbscan->ctl)
    c = kbtabctrl[c];
else
    c = kbtab[c];

if(kbscan->caps && c<='z' && c>='a')
    c += 'A' - 'a';

/*
 *  keyup only important for shifts
 */
if(keyup){
    switch(c){
    case Ctrl:
        kbscan->ctl = false;
        break;
    case Alt:
        kbscan->alt = false;
        break;
    case Altgr:
        kbscan->altgr = false;
        break;
    case Shift:
        kbscan->shift = false;
        ⟨kbdputsc() reset mouseshift 397c⟩
        ⟨kbdputsc() debugging up shift 641d⟩
        break;
    ⟨kbdputsc() mouse keyup cases 397e⟩
    }
    return;
}

/*
 *  normal character
 */
if(!(c & (Spec|KF))){
    ⟨kbdputsc() reboot if ctl-alt-del 627c⟩
    ⟨kbdputsc() if collecting 672d⟩
    else {
        kbdputc(c); //!! adding the character in kbd staging area
    }
}else{
    switch(c){
    case Ctrl:
        kbscan->ctl = true;
        break;
```

```
        case Alt:
            kbscan->alt = true;
            ⟨kbdputsc() start collecting 672e⟩
            break;
        case Altgr:
            kbscan->altgr = true;
            break;
        case Shift:
            kbscan->shift = true;
            ⟨kbdputsc() set mouseshift 397d⟩
            ⟨kbdputsc() debugging down shift 641e⟩
            break;
        case Caps:
            kbscan->caps ^= true;
            break;
        case Num:
            kbscan->num ^= true;
            if(!external)
                arch_setleds(kbscan);
            break;
        ⟨kbdputsc() mouse keydown cases 398⟩
        ⟨kbdputsc() special keyboard debug keys cases 641b⟩
        // e.g. Left, Right or other special key
        default:
            kbdputc(c);
            break;
        }
    }
}
```

Uses Alt-966 376a, Altgr-970 376a, Caps-967 376a, Ctrl-965 376a, KF-959 376a, KbExt 378b, KbInt 378b, Num-968 376a, Shift-963 376a, Spec-960 376a, false, kbdputc() 382c, kbscans 378c, kbtab 376b, kbtabaltgr 377c, kbtabctrl, kbtabshift 377a, and true.

⟨*function* arch_setleds *(arm)* 380a⟩≡ (976)

```
// empty callback for portkbd.c
void arch_setleds(Kbscan *kbscan) { USED(kbscan); }
```

⟨kbdputsc() *ensures c is in boundary* 380b⟩≡ (378d)

```
if(c > sizeof kbtab){
    // how could reach that? kbtab has 0x80 elts and do c&=0x7f.
    c |= keyup;
    if(c != 0xFF)   /* these come fairly often: CAPSLOCK U Y */
        print("unknown key %ux\n", c);
    return;
}
```

Uses kbtab 376b and print.

15.2.3 Esc

Despite the name, this subsection has nothing to do with the Esc key (which is just scan code 0x01). The "esc" here refers to escape sequences in the PS/2 scan code protocol: certain keys (arrow keys, Home, End, etc.) generate multi-byte sequences prefixed by 0xe0 (2-byte) or 0xe1 (3-byte). The state machine in [kbdputsc] handles this by setting esc1 or esc2 flags when it sees the prefix byte and returning immediately, then on the next interrupt it uses kbtabesc1 instead of kbtab for the lookup. This is why arrow keys like Left and Right appear in kbtabesc1 rather than in the main kbtab.

⟨*global* kbtabesc1 380c⟩≡ (708)

```
Rune kbtabesc1[Nscan] =
```

```
{
[0x00]  No, No, No, No, No, No, No, No,
[0x08]  No, No, No, No, No, No, No, No,
[0x10]  No, No, No, No, No, No, No, No,
[0x18]  No, No, No, No, '\n',   Ctrl,   No, No,
[0x20]  No, No, No, No, No, No, No, No,
[0x28]  No, No, Shift,  No, No, No, No, No,
[0x30]  No, No, No, No, No, '/',    No, Print,
[0x38]  Altgr,  No, No, No, No, No, No, No,
[0x40]  No, No, No, No, No, No, Break,  Home,
[0x48]  Up, Pgup,   No, Left,   No, Right,  No, End,
[0x50]  Down,   Pgdown, Ins,    Del,    No, No, No, No,
[0x58]  No, No, No, No, No, No, No, No,
[0x60]  No, No, No, No, No, No, No, No,
[0x68]  No, No, No, No, No, No, No, No,
[0x70]  No, No, No, No, No, No, No, No,
[0x78]  No, No, No, No, No, No, No, No,
};
```

Uses Altgr-970 376a, Break-964 376a, Ctrl-965 376a, Del-983 376a, Down-980 376a, End-979 376a, Home-973 376a, Ins-982 376a, Left-977 376a, No-972 376a, Nscan-985 376a, Pgdown-981 376a, Pgup-975 376a, Print-976 376a, Right-978 376a, Shift-963 376a, and Up-974 376a.

⟨Kbscan *other fields* 381a⟩≡ (378a) 397b▷

```
bool esc1;
int esc2;
```

⟨kbdputsc() *esc key handling part1 and possible return* 381b⟩≡ (378d)

```
/*
 *  e0's is the first of a 2 character sequence, e1 the first
 *  of a 3 character sequence (on the safari)
 */
if(c == 0xe0){
    kbscan->esc1 = true;
    return;
} else if(c == 0xe1){
    kbscan->esc2 = 2;
    return;
}
```

Uses true.

⟨kbdputsc() *esc key handling part2 and possible return* 381c⟩≡ (378d)

```
if(kbscan->esc1){
    c = kbtabesc1[c];
    kbscan->esc1 = false;
} else if(kbscan->esc2){
    kbscan->esc2--;
    return;
}
```

Uses false and kbtabesc1 380c.

15.2.4 kbd

The ConsKbd structure is the staging area between the keyboard interrupt and the queue system. Its istage circular buffer accumulates UTF-8 bytes deposited by [kbdputc] at interrupt time. The circular buffer uses three

pointers: `iw` (write position), `ir` (read position), and `ie` (end sentinel). This is a classic lock-free single-producer single-consumer ring buffer – the interrupt handler writes at `iw` and the clock callback reads from `ir`.

⟨*struct* `ConsKbd` 382a⟩≡ (696b)
```
struct ConsKbd
{
    /* a place to save up characters at interrupt time before dumping them in the queue */
    char    istage[1024];
    // pointers into istage to implement a circular buffer
    char    *iw; // write
    char    *ir; // read
    char    *ie; // end
    ILock    lockputc;

    ⟨ConsKbd other fields 389c⟩
    // extra
    QLock;
};
```

⟨*global* `kbd` 382b⟩≡ (696b)
```
static struct ConsKbd kbd =
{
    .iw = kbd.istage,
    .ir = kbd.istage,
    .ie = kbd.istage + sizeof(kbd.istage),
};
```
Uses `ConsKbd` 382a and `kbd-1778` 382b.

[kbdputc] bridges the Rune world and the byte world. It takes the Rune produced by [kbdputsc] and converts it to UTF-8 via `runetochar`, then writes those bytes into the `istage` circular buffer. This is where the `/dev/cons` interface becomes UTF-8 encoded – even though the keyboard hardware deals in scan codes and the internal processing uses 32-bit Runes, what reaches user processes through the queue is a stream of UTF-8 bytes. Note the use of `ilock` rather than `qlock`: this function runs at interrupt time, so it must use a spin lock rather than a sleeping mutex.

⟨*function* `kbdputc` 382c⟩≡ (696b)
```
/*
 *  Put character, possibly a rune, into kbd at interrupt time.
 *  Called at interrupt time to process a character.
 */
void
kbdputc(Rune ch)
{
    int i, n;
    char buf[UTFmax]; // pad's fourth bugfix :)
    Rune r;
    char *next;

    ⟨kbdputc() debugging 642a⟩

    if(kbd.ir == nil)
        return;       /* in case we're not inited yet */

    ilock(&kbd.lockputc);       /* just a mutex */
    r = ch;
    n = runetochar(buf, &r);
    for(i = 0; i < n; i++){
        next = kbd.iw+1;
        // circular buffer
        if(next >= kbd.ie)
```

```
            next = kbd.istage;
        if(next == kbd.ir) // full
            break;
        *kbd.iw = buf[i];
        kbd.iw = next;
    }
    iunlock(&kbd.lockputc);
    return;
}
```

Uses UTFmax, ilock() 79d, iunlock() 80b, and kbd-1778 382b.

[kbdputcclock] is called every 22 ms from the clock interrupt. It drains the istage circular buffer by calling [echo] on whatever bytes have accumulated. The wraparound case (when iw < ir) requires two calls to [echo]: one for the tail of the buffer (from ir to ie) and one for the head (from istage to iw). There is no explicit lock here because the clock callback and the keyboard interrupt are carefully ordered: the clock reads iw once into a local variable, creating a consistent snapshot.

⟨*clock callback kbdputcclock* 383a⟩≡ (696b)

```
/*
 *  we save up input characters till clock time to reduce
 *  per character interrupt overhead.
 */
static void
kbdputcclock(void)
{
    char *iw;

    /* this amortizes cost of qproduce */
    if(kbd.iw != kbd.ir){
        iw = kbd.iw;
        if(iw < kbd.ir){
            echo(kbd.ir, kbd.ie-kbd.ir);
            kbd.ir = kbd.istage;
        }
        if(kbd.ir != iw){
            echo(kbd.ir, iw-kbd.ir);
            kbd.ir = iw;
        }
    }
}
```

Uses echo() 384a and kbd-1778 382b.

⟨consinit() *initializing things* 383b⟩≡ (393b) 634a▷

```
/*
 * at 115200 baud, the 1024 char buffer takes 56 ms to process,
 * processing it every 22 ms should be fine
 */
addclock0link(kbdputcclock, 22);
```

Uses addclock0link() 253f and kbdputcclock() 383a.

15.2.5 echo()

[echo] serves a dual purpose: it enqueues the keyboard bytes into kbdq via [qproduce] so that processes reading /dev/cons can consume them, and it echoes them back to the screen via the screenputs hook so the user can see what they typed. The screenputs function pointer is initially nil and gets set later when the graphics subsystem initializes (e.g., to swconsole_screenputs for framebuffer output or cgascreenputs on x86). In raw

mode, echoing is suppressed – the characters still go into kbdq, but nothing is displayed, which is exactly what programs like rio need when they handle their own display.

⟨*function* echo 384a⟩≡ (696b)
```
static void
echo(char *buf, int n)
{
    ⟨echo() locals 642b⟩

    if(n == 0)
        return;

    ⟨echo() special keys handler 642c⟩

    qproduce(kbdq, buf, n); //!! add in kbd queue to be read from /dev/cons

    ⟨echo() return before any echoscreen if raw mode 396c⟩

    ⟨echo() hooks 384b⟩
}
```
Uses kbdq 388c and qproduce() 407.

⟨echo() *hooks* 384b⟩≡ (384a) 650c▷
```
if(screenputs != nil)
   echoscreen(buf, n);
```
Uses echoscreen() 384d and screenputs 384c.

⟨*hook screenputs* 384c⟩≡ (696b)
```
void    (*screenputs)(char*, int) = nil;
```
Uses screenputs 384c.

⟨*function* echoscreen 384d⟩≡ (696b)
```
static void
echoscreen(char *buf, int n)
{
    char *e, *p;
    char ebuf[128];
    int x;

    p = ebuf;
    e = ebuf + sizeof(ebuf) - 4;
    while(n-- > 0){
        if(p >= e){
            screenputs(ebuf, p - ebuf);
            p = ebuf;
        }
        x = *buf++;
        if(x == 0x15){ // ??
            *p++ = '^';
            *p++ = 'U';
            *p++ = '\n';
        } else
            *p++ = x;
    }
    if(p != ebuf)
        screenputs(ebuf, p - ebuf);
}
```
Uses screenputs 384c.

15.2.6 PC Keyboard (x86)

15.2.7 Raspberry Pi USB Keyboard (ARM)

15.2.8 Changing keyboard: /dev/kbmap

⟨*function* kbdgetmap 385a⟩≡ (708)

```
int
kbdgetmap(uint offset, int *t, int *sc, Rune *r)
{
    if ((int)offset < 0)
        error(Ebadarg);
    *t = offset/Nscan;
    *sc = offset%Nscan;
    switch(*t) {
    default:
        return 0;
    case 0:
        *r = kbtab[*sc];
        return 1;
    case 1:
        *r = kbtabshift[*sc];
        return 1;
    case 2:
        *r = kbtabesc1[*sc];
        return 1;
    case 3:
        *r = kbtabaltgr[*sc];
        return 1;
    case 4:
        *r = kbtabctrl[*sc];
        return 1;
    }
}
```

Uses Ebadarg 664d, Nscan-985 376a, kbtab 376b, kbtabaltgr 377c, kbtabctrl, kbtabesc1 380c, and kbtabshift 377a.

⟨*function* kbdputmap 385b⟩≡ (708)

```
void
kbdputmap(ushort m, ushort scanc, Rune r)
{
    if(scanc >= Nscan)
        error(Ebadarg);
    switch(m) {
    default:
        error(Ebadarg);
    case 0:
        kbtab[scanc] = r;
        break;
    case 1:
        kbtabshift[scanc] = r;
        break;
    case 2:
        kbtabesc1[scanc] = r;
        break;
    case 3:
        kbtabaltgr[scanc] = r;
        break;
    case 4:
        kbtabctrl[scanc] = r;
        break;
```

```
    }
}
```

Uses Ebadarg 664d, Nscan-985 376a, kbtab 376b, kbtabaltgr 377c, kbtabctrl, kbtabesc1 380c, and kbtabshift 377a.

15.2.9 External keyboard input: /dev/kbin

15.3 The screen

15.3.1 Overview

The screen subsystem provides the low-level text output that the console device dispatches to via the `screenputs` function pointer. On x86, this is straightforward CGA text-mode output—writing character/attribute pairs to a memory-mapped buffer at `0xB8000`. On ARM, there is no hardware text mode, so the kernel must negotiate a framebuffer with the VideoCore GPU through a mailbox protocol, then render characters in software (covered in Graphics book [Pad16d]).

15.3.2 Text output on a PC (x86)

15.3.3 Text output on a Raspberry Pi (ARM)

On the Raspberry Pi, the CPU does not have direct access to the display hardware. Instead, it communicates with the VideoCore GPU through a mailbox interface: a shared memory buffer at a fixed physical address (`VCBUFFER` at `0x3400`) where the CPU writes a framebuffer request (resolution, color depth) and the GPU writes back the physical address of the allocated framebuffer. The kernel then calls [mmukmap] to map that physical framebuffer into the kernel's virtual address space at the `FRAMEBUFFER` address, creating 1 MiB section entries in the L1 page table. This two-pass approach in `mmukmap`—first checking that no existing mappings conflict, then installing the new entries—avoids accidentally clobbering live mappings.

⟨*constant* `VCBUFFER` *(arm)* 386a⟩≡ (1005)

```
#define VCBUFFER    (KZERO+0x3400)      /* videocore mailbox buffer */
```

⟨*function* `fbinit` *(arm)* 386b⟩≡ (897b)

```
void*
fbinit(bool set, int *width, int *height, int *depth)
{
    Fbinfo *fi = (Fbinfo*)(VCBUFFER);
    uintptr va;

    if(!set)
        fbdefault(width, height, depth);
    /* Screen width must be a multiple of 16 */
    *width &= ~0xF;

    memset(fi, 0, sizeof(Fbinfo));
    fi->xres = fi->xresvirtual = *width;
    fi->yres = fi->yresvirtual = *height;
    fi->bpp = *depth;
    ⟨fbinit() write back cache after modified fb info(arm) 531l⟩
    vcwrite(ChanFb, dmaaddr(fi));

    if(vcread(ChanFb) != 0)
        return nil;
    //TODO: fi->base is in virtual space?? how vcore knows about that? PADDR?
    va = mmukmap(FRAMEBUFFER, PADDR(fi->base), fi->screensize);
    // make it a blue screen
```

```
    if(va)
        memset((char*)va, 0x7F, fi->screensize);
    return (void*)va;
}
```

Uses ChanFb-2355 893b, FRAMEBUFFER 44a, VCBUFFER 386a, dmaaddr() 957h, fbdefault() 387b, mmukmap() 387a, vcread() 894c, and vcwrite() 894b.

⟨*function* mmukmap*(arm)* 387a⟩≡ (1006b)

```
uintptr
mmukmap(uintptr va, uintptr pa, usize size)
{
    int o;
    usize n;
    PTE *pte, *pte0;

    assert((va & (MiB-1)) == 0);
    o = pa & (MiB-1);
    pa -= o;
    size += o;
    pte = pte0 = &cpu->mmul1[L1X(va)];
    for(n = 0; n < size; n += MiB)
        if(*pte++ != Fault)
            return 0;
    pte = pte0;
    for(n = 0; n < size; n += MiB){
        *pte++ = (pa+n)|Dom0|L1AP(Krw)|Section;
        mmuinvalidateaddr(va+n);
    }
    ⟨mmukmap() write back cache after changing page tables(arm) 531e⟩
    return va + o;
}
```

Uses Dom0 179d, Fault 179c, Krw 179b, L1AP 179e, L1X-1842 178a, MiB 48a, Section 178d, and assert 853f.

⟨*function* fbdefault*(arm)* 387b⟩≡ (897b)

```
/*
 * Framebuffer
 */
static errorneg1
fbdefault(int *width, int *height, int *depth)
{
    u32int buf[3];

    if(vcreq(TagGetres, &buf[0], 0, 2*4) != 2*4 ||
       vcreq(TagGetdepth, &buf[2], 0, 4) != 4)
        return ERROR_NEG1;
    *width = buf[0];
    *height = buf[1];
    *depth = buf[2];
    return OK_0;
}
```

Uses ERROR_NEG1, OK_0, TagGetdepth-2379 893d, TagGetres-2375 893d, and vcreq() 895a.

⟨*function* fbblank*(arm)* 387c⟩≡ (897b)

```
int
fbblank(bool blank)
{
    u32int buf[1];

    buf[0] = blank;
```

```
    if(vcreq(TagFbblank, buf, sizeof buf, sizeof buf) != sizeof buf)
        return ERROR_NEG1;
    return buf[0] & 1;
}
```

Uses ERROR_NEG1, TagFbblank-2374 893d, and vcreq() 895a.

15.4 The console device /dev/cons

The console device exposes keyboard input and screen output through the file system as /dev/cons and /dev/consctl. Reading /dev/cons in cooked mode gives line-edited input: the kernel buffers characters until a newline or Control-D, handling backspace and Control-U (kill line) internally. This is much cleaner than Unix's layered tty/pty/termios system – in Plan 9, all line discipline lives in this single device driver.

15.4.1 Overview

⟨*devcons.c enum Qxxx* 388a⟩≡ (696b)

```
enum{
    Qdir,

    Qcons,
    Qconsctl,

    ⟨devcons.c enum Qxxx cases 633a⟩
};
```

⟨*global* consdir 388b⟩≡ (696b)

```
static Dirtab consdir[]={
    ".",    {Qdir, 0, QTDIR},   0,      DMDIR|0555,

    "cons",     {Qcons},    0,      0660,
    "consctl",  {Qconsctl}, 0,      0220,

    ⟨consdir fields 633b⟩
};
```

Uses DMDIR, QTDIR 70b, Qcons-1781 388a, Qconsctl-1782 388a, and Qdir-1780 388a.

15.4.2 Buffered input

The console uses a two-queue design. Raw keystrokes arrive in kbdq (the "unprocessed" queue, fed by [echo] via [qproduce]). When a process reads /dev/cons in cooked mode, the [consread] code pulls bytes one at a time from kbdq, applies line editing (backspace, Control-U, newline detection), and when a complete line is ready, writes it into lineq (the "processed" queue). The process then reads from lineq. In raw mode, this two-stage pipeline is short-circuited: bytes go from kbdq directly to the reader without line editing.

⟨*global* kbdq 388c⟩≡ (696b)

```
Queue*  kbdq;           /* unprocessed console input */
```

⟨*function* kbdqinit 388d⟩≡ (696b)

```
void
kbdqinit(void)
{
    kbdq = qopen(4*1024, 0, 0, 0);
    if(kbdq == nil)
        panic("kbdinit");
    qnoblock(kbdq, true);
```

```
}
```

Uses kbdq 388c, qnoblock() 420c, qopen() 409a, and true.

⟨*global* lineq 389a⟩≡ (696b)

```
Queue*  lineq;          /* processed console input */
```

⟨*function* lineqinit 389b⟩≡ (696b)

```
void
lineqinit(void)
{

    lineq = qopen(2*1024, 0, nil, nil);
    if(lineq == nil)
        panic("lineqinit");
    qnoblock(lineq, true);
}
```

Uses lineq 389a, qnoblock() 420c, qopen() 409a, and true.

⟨ConsKbd *other fields* 389c⟩≡ (382a) 396a▷

```
char    line[1024]; /* current input line */
int x;      /* index into line */
```

This is the core of console input. The outer `while` loop spins until `lineq` has data. Inside, it reads one byte at a time from `kbdq` and implements a simple line editor: backspace decrements the index, Control-U resets it to zero, newline marks the line as complete (sets `send`), and Control-D sends whatever has been accumulated (even if empty, which the reader sees as EOF). When `send` is true or the line buffer is full, the accumulated bytes are written to `lineq` in one shot. This batching means a `read` on `/dev/cons` always returns a complete line in cooked mode.

⟨consread() *Qcons case* 389d⟩≡ (394c)

```
case Qcons:
    qlock(&kbd);
    if(waserror()) {
        qunlock(&kbd);
        nexterror();
    }
    while(!qcanread(lineq)){
        if(qread(kbdq, &ch, 1) == 0)
            continue;
        send = false;
        if(ch == 0){
            /* flush output on rawoff -> rawon */
            if(kbd.x > 0)
                send = !qcanread(kbdq);

        ⟨consread() else if raw mode 396b⟩
        }else{
            switch(ch){
            case '\b':
                if(kbd.x > 0)
                    kbd.x--;
                break;
            case 0x15:  /* ^U */
                kbd.x = 0;
                break;
            case '\n':
                send = true;
                kbd.line[kbd.x++] = ch;
                break;
```

```
            case 0x04:  /* ^D */
                send = true;
                break;
            default:
                kbd.line[kbd.x++] = ch;
                break;
            }
        }
        if(send || kbd.x == sizeof kbd.line){
            qwrite(lineq, kbd.line, kbd.x);
            kbd.x = 0;
        }
    }
    n = qread(lineq, buf, n);
    qunlock(&kbd);
    poperror();
    return n;
```

Uses Qcons-1781 388a, false, kbd-1778 382b, kbdq 388c, lineq 389a, nexterror, poperror 666d, qcanread() 420a, qlock() 81f, qread() 413, qunlock() 82c, qwrite() 401b, true, and waserror 666d.

15.4.3 Output

Writing to /dev/cons calls [putstrn0], which dispatches to whatever output backend is active: the screenputs hook for display output, kmesg for the kernel message buffer, or the serial port. The data is copied to a local buffer first to avoid page faults while holding locks inside the output path.

⟨conswrite() *Qcons case* 390a⟩≡ (395a)

```
case Qcons:
    /*
     * Can't page fault in putstrn, so copy the data locally.
     */
    l = n;
    while(l > 0){
        bp = l;
        if(bp > sizeof buf)
            bp = sizeof buf;
        memmove(buf, a, bp);
        putstrn0(buf, bp, 1);
        a += bp;
        l -= bp;
    }
    break;
```

Uses Qcons-1781 388a and putstrn0() 390b.

⟨*function* putstrn0 390b⟩≡ (696b)

```
/*
 *   Print a string on the console.
 */
static void
putstrn0(char *str, int n, bool usewrite)
{
    int m;
    char *t;

    if(!arch_islo())
        usewrite = false;

    ⟨putstrn0() kmesg handling 650d⟩
```

```
    ⟨putstrn0() if kprint 652c⟩
    else if(screenputs != nil)
        screenputs(str, n);

    ⟨putstrn0() serialoq handling 654a⟩
}
```

Uses false and screenputs 384c.

15.4.4 print(), iprint(), pprint()

The kernel has three distinct print functions, each tailored to a different context. print is the normal one: it formats into a stack buffer and calls [putstrn0] with usewrite false, meaning output goes to the screen and serial port but not through the file system. iprint (interrupt print) is for use in interrupt handlers and other contexts where locks may already be held: it raises the interrupt priority level with arch_splhi, uses a try-lock on iprintlock rather than blocking, and outputs directly to screen, serial, and the kernel message buffer. The try-lock with a bounded retry loop (1000 attempts with 100 μs delays) is a pragmatic compromise—on multiprocessor systems it prevents interleaved output without risking deadlock if the lock holder has crashed. pprint (process print) is different altogether: it writes to the calling process's stderr (file descriptor 2) through the normal device write path, prepending the process name and pid. This is how the kernel reports per-process errors (e.g., "db 43: unknown qid") to the right terminal window rather than the system console.

print()

⟨*constant* PRINTSIZE 391a⟩≡ (867a)

```
PRINTSIZE = 256,
```

⟨*function* print 391b⟩≡ (696b)

```
bool noprint; // to debug?

int
devcons_print(char *fmt, ...)
{
    int n;
    va_list arg;
    char buf[PRINTSIZE];

    if(noprint)
        return -1;

    va_start(arg, fmt);
    n = vseprint(buf, buf+sizeof(buf), fmt, arg) - buf;
    va_end(arg);
    putstrn(buf, n);

    return n;
}
```

Uses PRINTSIZE 391a, noprint 391b, and putstrn() 391c.

⟨*function* putstrn 391c⟩≡ (696b)

```
void
putstrn(char *str, int n)
{
    putstrn0(str, n, false);
}
```

Uses false and putstrn0() 390b.

iprint()

⟨*global* iprintscreenputs 392a⟩≡ (696b)
```
bool iprintscreenputs = true;
```
Uses iprintscreenputs 392a and true.

⟨*function* iprint 392b⟩≡ (696b)
```
int
devcons_iprint(char *fmt, ...)
{
    int n, s, locked;
    va_list arg;
    char buf[PRINTSIZE];

    s = arch_splhi();

    va_start(arg, fmt);
    n = vseprint(buf, buf+sizeof(buf), fmt, arg) - buf;
    va_end(arg);

    locked = iprintcanlock(&iprintlock);

    kmesgputs(buf, n); // addon
    if(screenputs != nil && iprintscreenputs)
        screenputs(buf, n);
    uartputs(buf, n);

    if(locked)
        unlock(&iprintlock);

    arch_splx(s);

    return n;
}
```
Uses PRINTSIZE 391a, iprintcanlock() 392d, iprintlock-1779 392c, iprintscreenputs 392a, kmesgputs() 650e, screenputs 384c, uartputs() 676d, and unlock() 78d.

⟨*global* iprintlock 392c⟩≡ (696b)
```
/*
 * Want to interlock iprints to avoid interlaced output on
 * multiprocessor, but don't want to deadlock if one processor
 * dies during print and another has something important to say.
 * Make a good faith effort.
 */
static Lock iprintlock;
```

⟨*function* iprintcanlock 392d⟩≡ (696b)
```
static bool
iprintcanlock(Lock *l)
{
    int i;

    for(i=0; i<1000; i++){
        if(canlock(l))
            return true;
        if(l->cpu == CPUS(cpu->cpuno))
            return false;
        arch_microdelay(100);
    }
    return false;
```

```
}
```

Uses CPUS 702, arch_microdelay, canlock() 78f, false, and true.

pprint()

⟨*function* pprint 393a⟩≡ (696b)

```
int
devcons_pprint(char *fmt, ...)
{
    int n;
    Chan *c;
    va_list arg;
    char buf[2*PRINTSIZE];

    if(up == nil || up->fgrp == nil)
        return 0;

    c = up->fgrp->fd[2]; // stderr
    if(c==nil || (c->mode!=OWRITE && c->mode!=ORDWR))
        return 0;

    n = snprint(buf, sizeof buf, "%s %lud: ", up->text, up->pid);
    va_start(arg, fmt);
    n = vseprint(buf+n, buf+sizeof(buf), fmt, arg) - buf;
    va_end(arg);

    if(waserror())
        return 0;
    devtab[c->type]->write(c, buf, n, c->offset);
    poperror();

    lock(c);
    c->offset += n;
    unlock(c);

    return n;
}
```

Uses ORDWR 301c, OWRITE, PRINTSIZE 391a, lock() 76d, poperror 666d, unlock() 78d, up, and waserror 666d.

15.4.5 /dev/cons

The console device driver follows the standard Plan 9 device interface: a Dev structure populated with method pointers for each file operation. Most methods delegate to generic helpers (devopen, devgen, devdirread) with the real logic concentrated in consread and conswrite, which switch on c->qid.path to dispatch to the right sub-file handler (Qcons, Qconsctl, Quser, Qnull, etc.). The device character 'c' identifies this as the console device in the devtab array.

⟨*method consinit* 393b⟩≡ (696b)

```
static void
consinit(void)
{
    ⟨consinit() initializing things 383b⟩
}
```

⟨*method consopen* 394a⟩≡ (696b)

```
static Chan*
consopen(Chan *c, int omode)
{
    c->aux = nil;
    c = devopen(c, omode, consdir, nelem(consdir), devgen);

    switch((ulong)c->qid.path){
    ⟨consopen() cases 396d⟩
    }
    return c;
}
```

Uses consdir-1797 388b, devgen() 428, and nelem 853i.

⟨*method consclose* 394b⟩≡ (696b)

```
static void
consclose(Chan *c)
{
    switch((ulong)c->qid.path){
    ⟨consclose() cases 396e⟩
    }
}
```

⟨*method consread* 394c⟩≡ (696b)

```
static long
consread(Chan *c, void *buf, long n, vlong off)
{
    ulong l;
    char ch;
    char tmp[256];      /* must be >= 18*NUMSIZE (Qswap) */
    int i, k, send;
    vlong offset = off;

    if(n <= 0)
        return n;

    switch((ulong)c->qid.path){
    case Qdir:
        return devdirread(c, buf, n, consdir, nelem(consdir), devgen);

    ⟨consread() Qcons case 389d⟩

    ⟨consread() cases 205d⟩


    case Quser:
        return readstr((ulong)offset, buf, n, up->user);

    case Qpid:
        return readnum((ulong)offset, buf, n, up->pid, NUMSIZE);

    case Qppid:
        return readnum((ulong)offset, buf, n, up->parentpid, NUMSIZE);

    case Qpgrpid:
        return readnum((ulong)offset, buf, n, up->pgrp->pgrpid, NUMSIZE);

    case Qnull:
        return 0;
```

```
    case Qzero:
        memset(buf, 0, n);
        return n;

    default:
        print("consread %#llux\n", c->qid.path);
        error(Egreg);
    }
    panic("consread: should not reach this point");
    return -1;
}
```

Uses Egreg 664d, NUMSIZE 867a, Qdir-1780 388a, Qnull-1785 633a, Qpgrpid-1786 633a, Qpid-1787 633a, Qppid-1788 633a, Quser-1792 633a, Qzero-1793 633a, consdir-1797 388b, devdirread() 432c, devgen() 428, nelem 853i, print, readnum() 670c, readstr() 670d, and up.

⟨*method conswrite* 395a⟩≡ (696b)

```
static long
conswrite(Chan *c, void *va, long n, vlong off)
{
    char *a;

    ⟨conswrite locals 395b⟩

    a = va;
    USED(off);

    switch((ulong)c->qid.path){

    ⟨conswrite() Qcons case 390a⟩

    ⟨conswrite() cases 216b⟩

    case Quser:
        return userwrite(a, n);


    // > /dev/null :)
    case Qnull:
        break;

    default:
        print("conswrite: %#llux\n", c->qid.path);
        error(Egreg);
    }
    return n;
}
```

Uses Egreg 664d, Qnull-1785 633a, Quser-1792 633a, print, and userwrite() 592a.

⟨conswrite *locals* 395b⟩≡ (395a)

```
    char buf[256], ch;
    long l, bp;
    int fd;
    Chan *swc;
```

⟨*global* consdevtab 395c⟩≡ (696b)

```
Dev consdevtab = {
    .dc       =    'c',
    .name     =    "cons",

    .reset    =    devreset,
```

```
    .init     =    consinit,
    .shutdown =    devshutdown,
    .attach   =    consattach,
    .walk     =    conswalk,
    .stat     =    consstat,
    .open     =    consopen,
    .create   =    devcreate,
    .close    =    consclose,
    .read     =    consread,
    .bread    =    devbread,
    .write    =    conswrite,
    .bwrite   =    devbwrite,
    .remove   =    devremove,
    .wstat    =    devwstat,
};
```

Uses consattach() 696b, consclose() 394b, consinit() 393b, consopen() 394a, consread() 394c, consstat() 696b, conswalk() 696b, conswrite() 395a, devbread() 434a, devbwrite() 434b, devcreate() 810, devremove() 810, devreset() 810, devshutdown() 810, and devwstat() 810.

15.4.6 /dev/consctl and raw input

Raw mode is controlled by writing "rawon" or "rawoff" to /dev/consctl. When raw mode is active, [consread] bypasses line editing entirely: each byte from kbdq is passed straight through without waiting for a newline, and [echo] skips the screen echo. The ctl reference count tracks how many processes have /dev/consctl open; when the last one closes it, raw mode is automatically turned off. This is important because programs like rio that need raw keyboard input must not leave the console in a broken state if they crash.

⟨ConsKbd *other fields* 396a⟩+≡ (382a) ◁389c 627d▷

```
bool raw;        /* true if we shouldn't process input */
Ref ctl;        /* number of opens to the control file */
```

⟨consread() *else if raw mode* 396b⟩≡ (389d)

```
}else if(kbd.raw){
    kbd.line[kbd.x++] = ch;
    send = !qcanread(kbdq);
```

Uses kbd-1778 382b, kbdq 388c, and qcanread() 420a.

⟨echo() *return before any echoscreen if raw mode* 396c⟩≡ (384a)

```
if(kbd.raw)
    return;
```

Uses kbd-1778 382b.

⟨consopen() *cases* 396d⟩≡ (394a) 651g▷

```
case Qconsctl:
    incref(&kbd.ctl);
    break;
```

Uses Qconsctl-1782 388a, incref() 87b, and kbd-1778 382b.

⟨consclose() *cases* 396e⟩≡ (394b) 652a▷

```
/* last close of control file turns off raw */
case Qconsctl:
    if(c->flag&COPEN){
        if(decref(&kbd.ctl) == 0)
            kbd.raw = false;
    }
    break;
```

Uses COPEN 294b, Qconsctl-1782 388a, decref() 87c, false, and kbd-1778 382b.

⟨conswrite() *cases* 397a⟩+≡ (395a) ◁266b
```
case Qconsctl:
    if(n >= sizeof(buf))
        n = sizeof(buf)-1;
    strncpy(buf, a, n);
    buf[n] = 0;
    for(a = buf; a;){
        if(strncmp(a, "rawon", 5) == 0){
            kbd.raw = true;
            /* clumsy hack - wake up reader */
            ch = 0;
            qwrite(kbdq, &ch, 1);
        } else if(strncmp(a, "rawoff", 6) == 0){
            kbd.raw = false;
        }
        ⟨conswrite() Qconsctl other ifs 627e⟩
        if(a = strchr(a, ' '))
            a++;
    }
    break;
```
Uses `Qconsctl-1782` 388a, `false`, `kbd-1778` 382b, `kbdq` 388c, `qwrite()` 401b, and `true`.

15.5 The mouse

15.5.1 Mouse and keyboard

The keyboard and mouse are tightly coupled in Plan 9. The `Kbscan` structure tracks mouse button state alongside modifier keys because some keyboards generate mouse button events (e.g., via USB HID). When the scan code falls in the `Kmouse` range, [kbdputsc] updates the `buttons` bitmask and calls the `kbdmouse` hook, which injects the event into the mouse device driver.

⟨`Kbscan` *other fields* 397b⟩+≡ (378a) ◁381a 672c▷
```
int buttons;
```

⟨`kbdputsc()` *reset mouseshift* 397c⟩≡ (378d)
```
mouseshifted = false;
```
Uses `false` and `mouseshifted` 719.

⟨`kbdputsc()` *set mouseshift* 397d⟩≡ (378d)
```
mouseshifted = true;
```
Uses `mouseshifted` 719 and `true`.

⟨`kbdputsc()` *mouse keyup cases* 397e⟩≡ (378d)
```
case Kmouse|1:
case Kmouse|2:
case Kmouse|3:
case Kmouse|4:
case Kmouse|5:
    kbscan->buttons &= ~(1<<(c-Kmouse-1));
    if(kbdmouse)
        kbdmouse(kbscan->buttons);
    break;
```
Uses `Kmouse-971` 376a and `kbdmouse` 719.

⟨kbdputsc() *mouse keydown cases* 398⟩≡ (378d)

```
case Kmouse|1:
case Kmouse|2:
case Kmouse|3:
case Kmouse|4:
case Kmouse|5:
    kbscan->buttons |= 1<<(c-Kmouse-1);
    if(kbdmouse)
        kbdmouse(kbscan->buttons);
    break;
```

Uses Kmouse-971 376a and kbdmouse 719.

15.5.2 PS2 mouse (x86)

The PS2 mouse on x86 shares the i8042 keyboard controller: the auxiliary port (port 0x60/0x64) delivers 3-byte mouse packets that encode button state and relative X/Y deltas. On ARM, there is no PS2 hardware; the mouse is a USB HID device handled by the USB stack, which converts USB reports into the same internal mouse event format.

15.5.3 USB mouse (ARM)

15.6 Other devices

Beyond the keyboard, screen, console, and mouse, the kernel includes several other device drivers exposed through /dev: devpipe for in-kernel pipes, devdup for file descriptor manipulation, devmnt for mounted file systems, devsrv for the service registry, and devaudio for sound. Each follows the same Dev interface pattern, making them interchangeable from the file system's perspective.

Chapter 16

Queue

Queues are the kernel's general-purpose buffering mechanism for I/O data flow. Unlike simple byte ring buffers, these queues are block-based: data is stored in linked lists of `Block` structures, each containing a contiguous byte range. This design is optimized for networking where data arrives in packets that should not be split unnecessarily. Queues provide flow control (back-pressure when a producer is faster than a consumer), support both message mode (preserving packet boundaries) and stream mode (treating data as a byte stream), and offer two APIs: a process-context API ([qread]/[qwrite]) that can sleep, and an interrupt-context API ([qproduce]/[qconsume]) that never blocks. The state bits encode the queue's current condition. The two most important for flow control are `Qstarve` and `Qflow`: `Qstarve` is set by the consumer when it finds an empty queue, signaling the producer to wake it up when data arrives; `Qflow` is set by the producer when the queue exceeds its limit, signaling it to stop and wait. `Qmsg` changes the read semantics: in stream mode, partial blocks can be returned; in message mode, each read returns exactly one block (or discards excess data), preserving packet boundaries for protocols like UDP.

⟨*enum* `queuestate` 399a⟩≡ (804b)
```
/* queue state bits,  Qmsg, Qcoalesce, and Qkick can be set in qopen */
enum QueueState
{
  /* Queue.state */
  Qstarve   = (1<<0), /* consumer starved */
  Qmsg    = (1<<1), /* message stream */
  Qclosed   = (1<<2), /* queue has been closed/hungup */
  Qflow   = (1<<3), /* producer flow controlled */
  Qcoalesce = (1<<4), /* coalesce packets on read */
  Qkick   = (1<<5), /* always call the kick routine after qwrite */
};
```

The `Queue` structure uses separate locks for different concerns. The embedded `Lock` (a spin lock) protects the block list and state bits – it is used with `ilock`/`iunlock` so it can be held at interrupt level. The `rlock` and `wlock` QLocks (sleeping mutexes) serialize multiple reader or writer processes respectively. The `Rendez` variables `rr` and `wr` are the sleep/wakeup rendezvous points for blocked readers and writers. The `kick` callback is called when the queue drains below the flow control threshold, which is used by network drivers to restart hardware transmission.

⟨*struct* `Queue` 399b⟩≡ (804b)
```
struct Queue
{
  Lock;

  Block*  bfirst;   /* buffer */
  Block*  blast;

  int len;    /* bytes allocated to queue */
  int dlen;   /* data bytes in queue */
```

```
    int limit;    /* max bytes in queue */
    int inilim;   /* initial limit */
    int state;
    int noblock;  /* true if writes return immediately when q full */
    int eof;    /* number of eofs read by user */

    void  (*kick)(void*); /* restart output */
    void  (*bypass)(void*, Block*); /* bypass queue altogether */
    void* arg;    /* argument to kick */

    QLock rlock;    /* mutex for reading processes */
    Rendez  rr;   /* process waiting to read */
    QLock wlock;    /* mutex for writing processes */
    Rendez  wr;   /* process waiting to write */

    char  err[ERRMAX];
};
```

Pictorially, a queue is the head of a singly-linked list of `Block`s. `bfirst` points at the oldest unread block, `blast` at the newest, and each block carries a read pointer `rp` and a write pointer `wp` into its own byte array. In message mode each block is exactly one message; in stream mode the block boundaries are invisible to the reader:

```
Queue {
   bfirst ---+                              blast ---+
   blast  ---|------------------------------+       |
   dlen=9,   |                               |       |
   limit=64  |                               |       |
}            v                               v
             +-------+    +-------+    +-------+
             | Block |--->| Block |--->| Block |--> nil
             +-------+    +-------+    +-------+
             |rp wp  |    |rp wp  |    |rp wp  |
             | "he"  |    | "llo" |    | " wo" |
             +-------+    +-------+    +-------+

BLEN(b) = b->wp - b->rp     (bytes readable from b)
q->dlen = sum of BLEN over all blocks  = 9
q->len  = sum of BALLOC    over all blocks (allocated bytes)
```

Producer and consumer communicate flow control purely through the `Qflow` and `Qstarve` bits, which form a tiny two-state synchronization protocol:

```
producer side                    consumer side
-------------                    -------------
q->dlen >= limit ?               q->bfirst == nil ?
     |                                |
     | yes: set Qflow                 | yes: set Qstarve
     |      sleep on q->wr            |      sleep on q->rr
     v                                v
wait until a qget drops          wait until a qwrite adds
q->len below limit/2             a block; qwrite sees
and clears Qflow, then           Qstarve and wakeup(&q->rr)
wakeup(&q->wr)
```

Two details are worth calling out. First, the flow-control threshold is hysteretic: the producer stops at `limit` but is only resumed once `q->len` falls below `limit/2`. Without this gap, a queue that hovers near the limit would thrash—every consumed byte would wake the producer, which would immediately refill the queue and re-block. Second, `Qstarve` is set lazily by the consumer, not eagerly by the producer: the producer does not look at anything the consumer maintains, it just checks `Qstarve` after adding a block and wakes the sleeper if the flag was set. This keeps `qwrite` cheap in the common case when the consumer is already running.

⟨*function* `pullupqueue` 401a⟩≡ (813c)

```
/*
 *  make sure the first block has at least n bytes
 */
Block*
pullupqueue(Queue *q, int n)
{
    Block *b;

    if(BLEN(q->bfirst) >= n)
        return q->bfirst;
    q->bfirst = pullupblock(q->bfirst, n);
    for(b = q->bfirst; b != nil && b->next != nil; b = b->next)
        ;
    q->blast = b;
    return q->bfirst;
}
```

Uses `BLEN` 435b and `pullupblock()` 441b.

[qwrite] is the process-context write API. It breaks the data into `Maxatomic`-sized chunks, wraps each in a `Block`, and calls [qbwrite] which handles flow control and sleeping. In message mode (`Qmsg`), only the first chunk is sent – the loop exits after one iteration – preserving the message as a single block. In stream mode, all chunks are sent sequentially.

⟨*function* `qwrite` 401b⟩≡ (813c)

```
/*
 *  write to a queue.  only Maxatomic bytes at a time is atomic.
 */
int
qwrite(Queue *q, void *vp, int len)
{
    int n, sofar;
    Block *b;
    byte *p = vp;

//    QDEBUG if(!arch_islo())  print("qwrite hi %#p\n", getcallerpc(&q));

    sofar = 0;
    do {
        n = len-sofar;
        if(n > Maxatomic)
            n = Maxatomic;

        b = allocb(n);
        setmalloctag(b, (up->text[0]<<24)|(up->text[1]<<16)|(up->text[2]<<8)|up->text[3]);
        if(waserror()){
            freeb(b);
            nexterror();
        }
        memmove(b->wp, p+sofar, n);
        poperror();
        b->wp += n;
```

```
        qbwrite(q, b);

        sofar += n;
    } while(sofar < len && (q->state & Qmsg) == 0);

    return len;
}
```

Uses Maxatomic-779 813b, Qmsg 399a, allocb() 436b, freeb() 438a, nexterror, poperror 666d, qbwrite() 414b, setmalloctag() 106a, up, and waserror 666d.

\hlinkqget is a non-blocking dequeue for interrupt context: it returns the first block or nil (setting Qstarve so the producer knows to wake the consumer later). If the queue drops below half its limit, it clears Qflow and wakes the producer. The half-limit hysteresis prevents oscillation at the boundary.

⟨*function* qget 402a⟩≡ (813c)

```
/*
 *  get next block from a queue, return null if nothing there
 */
Block*
qget(Queue *q)
{
    int dowakeup;
    Block *b;

    /* sync with qwrite */
    ilock(q);

    b = q->bfirst;
    if(b == nil){
        q->state |= Qstarve;
        iunlock(q);
        return nil;
    }
    q->bfirst = b->next;
    b->next = 0;
    q->len -= BALLOC(b);
    q->dlen -= BLEN(b);
//    QDEBUG checkb(b, "qget");

    /* if writer flow controlled, restart */
    if((q->state & Qflow) && q->len < q->limit/2){
        q->state &= ~Qflow;
        dowakeup = 1;
    } else
        dowakeup = 0;

    iunlock(q);

    if(dowakeup)
        wakeup(&q->wr);

    return b;
}
```

Uses BALLOC 435c, BLEN 435b, Qflow 399a, Qstarve 399a, ilock() 79d, iunlock() 80b, and wakeup.

⟨*function* qdiscard 402b⟩≡ (813c)

```
/*
 *  throw away the next 'len' bytes in the queue
 */
```

```
int
qdiscard(Queue *q, int len)
{
    Block *b;
    int dowakeup, n, sofar;

    ilock(q);
    for(sofar = 0; sofar < len; sofar += n){
        b = q->bfirst;
        if(b == nil)
            break;
//        QDEBUG checkb(b, "qdiscard");
        n = BLEN(b);
        if(n <= len - sofar){
            q->bfirst = b->next;
            b->next = 0;
            q->len -= BALLOC(b);
            q->dlen -= BLEN(b);
            freeb(b);
        } else {
            n = len - sofar;
            b->rp += n;
            q->dlen -= n;
        }
    }

    /*
     *  if writer flow controlled, restart
     *
     *  This used to be
     *  q->len < q->limit/2
     *  but it slows down tcp too much for certain write sizes.
     *  I really don't understand it completely.  It may be
     *  due to the queue draining so fast that the transmission
     *  stalls waiting for the app to produce more data.  - presotto
     */
    if((q->state & Qflow) && q->len < q->limit){
        q->state &= ~Qflow;
        dowakeup = 1;
    } else
        dowakeup = 0;

    iunlock(q);

    if(dowakeup)
        wakeup(&q->wr);

    return sofar;
}
```

Uses BALLOC 435c, BLEN 435b, Qflow 399a, freeb() 438a, ilock() 79d, iunlock() 80b, and wakeup.

[qproduce] and [qconsume] are the interrupt-safe queue API. They use `ilock` (spin lock with interrupts disabled) instead of `qlock` (sleeping mutex), so they can be called from interrupt handlers. [qconsume] copies data out of the first block. If the queue is empty it returns `-1` and sets `Qstarve` rather than blocking – an interrupt handler cannot sleep. These are the functions used by the keyboard driver ([echo] calls [qproduce] to feed `kbdq`) and by network device drivers that process packets at interrupt time.

⟨*function* qconsume 403⟩≡ (813c)

```
/*
 *  Interrupt level copy out of a queue, return # bytes copied.
```

```
 */
int
qconsume(Queue *q, void *vp, int len)
{
    Block *b;
    int n, dowakeup;
    byte *p = vp;
    Block *tofree = nil;

    /* sync with qwrite */
    ilock(q);

    for(;;) {
        b = q->bfirst;
        if(b == 0){
            q->state |= Qstarve;
            iunlock(q);
            return -1;
        }
//        QDEBUG checkb(b, "qconsume 1");

        n = BLEN(b);
        if(n > 0)
            break;
        q->bfirst = b->next;
        q->len -= BALLOC(b);

        /* remember to free this */
        b->next = tofree;
        tofree = b;
    };

    if(n < len)
        len = n;
    memmove(p, b->rp, len);
    consumecnt += n;
    b->rp += len;
    q->dlen -= len;

    /* discard the block if we're done with it */
    if((q->state & Qmsg) || len == n){
        q->bfirst = b->next;
        b->next = 0;
        q->len -= BALLOC(b);
        q->dlen -= BLEN(b);

        /* remember to free this */
        b->next = tofree;
        tofree = b;
    }

    /* if writer flow controlled, restart */
    if((q->state & Qflow) && q->len < q->limit/2){
        q->state &= ~Qflow;
        dowakeup = 1;
    } else
        dowakeup = 0;

    iunlock(q);
```

```
    if(dowakeup)
        wakeup(&q->wr);

    if(tofree != nil)
        freeblist(tofree);

    return len;
}
```

Uses BALLOC 435c, BLEN 435b, Qflow 399a, Qmsg 399a, Qstarve 399a, consumecnt-774 812d, freeblist() 439a, ilock() 79d, iunlock() 80b, and wakeup.

\hlinkqpass and \hlinkqpassnolim append an entire block list to a queue at interrupt level. qpass drops the list if the queue is full (returning -1), while qpassnolim always accepts. These are used by network drivers that have already allocated blocks and want to hand them off without copying.

⟨*function* qpass 405⟩≡ (813c)

```
int
qpass(Queue *q, Block *b)
{
    int dlen, len, dowakeup;

    /* sync with qread */
    dowakeup = 0;
    ilock(q);
    if(q->len >= q->limit){
        freeblist(b);
        iunlock(q);
        return -1;
    }
    if(q->state & Qclosed){
        len = BALLOC(b);
        freeblist(b);
        iunlock(q);
        return len;
    }

    /* add buffer to queue */
    if(q->bfirst)
        q->blast->next = b;
    else
        q->bfirst = b;
    len = BALLOC(b);
    dlen = BLEN(b);
//    QDEBUG checkb(b, "qpass");
    while(b->next){
        b = b->next;
//        QDEBUG checkb(b, "qpass");
        len += BALLOC(b);
        dlen += BLEN(b);
    }
    q->blast = b;
    q->len += len;
    q->dlen += dlen;

    if(q->len >= q->limit/2)
        q->state |= Qflow;

    if(q->state & Qstarve){
        q->state &= ~Qstarve;
        dowakeup = 1;
```

```
    }
    iunlock(q);

    if(dowakeup)
        wakeup(&q->rr);

    return len;
}
```

Uses BALLOC 435c, BLEN 435b, Qclosed 399a, Qflow 399a, Qstarve 399a, freeblist() 439a, ilock() 79d, iunlock() 80b, and wakeup.

⟨*function* qpassnolim 406⟩≡ (813c)

```
int
qpassnolim(Queue *q, Block *b)
{
    int dlen, len, dowakeup;

    /* sync with qread */
    dowakeup = 0;
    ilock(q);

    if(q->state & Qclosed){
        freeblist(b);
        iunlock(q);
        return BALLOC(b);
    }

    /* add buffer to queue */
    if(q->bfirst)
        q->blast->next = b;
    else
        q->bfirst = b;
    len = BALLOC(b);
    dlen = BLEN(b);
//    QDEBUG checkb(b, "qpass");
    while(b->next){
        b = b->next;
//        QDEBUG checkb(b, "qpass");
        len += BALLOC(b);
        dlen += BLEN(b);
    }
    q->blast = b;
    q->len += len;
    q->dlen += dlen;

    if(q->len >= q->limit/2)
        q->state |= Qflow;

    if(q->state & Qstarve){
        q->state &= ~Qstarve;
        dowakeup = 1;
    }
    iunlock(q);

    if(dowakeup)
        wakeup(&q->rr);

    return len;
}
```

Uses BALLOC 435c, BLEN 435b, Qclosed 399a, Qflow 399a, Qstarve 399a, freeblist() 439a, ilock() 79d, iunlock() 80b, and wakeup.

[qproduce] is the interrupt-safe counterpart of [qbwrite]. It allocates a block with `iallocb` (the interrupt-safe allocator that does not sleep), copies the data in, and appends it to the queue. If the queue is full, it returns `-1` and sets `Qflow` rather than blocking. If a consumer was starved (waiting for data), it clears `Qstarve` and wakes it up.

⟨*function* qproduce 407⟩≡ (813c)

```
int
qproduce(Queue *q, void *vp, int len)
{
    Block *b;
    int dowakeup;
    byte *p = vp;

    /* sync with qread */
    dowakeup = 0;
    ilock(q);

    /* no waiting receivers, room in buffer? */
    if(q->len >= q->limit){
        q->state |= Qflow;
        iunlock(q);
        return -1;
    }

    /* save in buffer */
    b = iallocb(len);
    if(b == 0){
        iunlock(q);
        return 0;
    }
    memmove(b->wp, p, len);
    producecnt += len;
    b->wp += len;
    if(q->bfirst)
        q->blast->next = b;
    else
        q->bfirst = b;
    q->blast = b;
    /* b->next = 0; done by iallocb() */
    q->len += BALLOC(b);
    q->dlen += BLEN(b);
//    QDEBUG checkb(b, "qproduce");

    if(q->state & Qstarve){
        q->state &= ~Qstarve;
        dowakeup = 1;
    }

    if(q->len >= q->limit)
        q->state |= Qflow;
    iunlock(q);

    if(dowakeup)
        wakeup(&q->rr);

    return len;
}
```

Uses BALLOC 435c, BLEN 435b, Qflow 399a, Qstarve 399a, iallocb() 437c, ilock() 79d, iunlock() 80b, producecnt-775 812d, and wakeup.

⟨*function* qcopy 408⟩≡ (813c)

```
/*
 *  copy from offset in the queue
 */
Block*
qcopy(Queue *q, int len, ulong offset)
{
    int sofar;
    int n;
    Block *b, *nb;
    byte *p;

    nb = allocb(len);

    ilock(q);

    /* go to offset */
    b = q->bfirst;
    for(sofar = 0; ; sofar += n){
        if(b == nil){
            iunlock(q);
            return nb;
        }
        n = BLEN(b);
        if(sofar + n > offset){
            p = b->rp + offset - sofar;
            n -= offset - sofar;
            break;
        }
//        QDEBUG checkb(b, "qcopy");
        b = b->next;
    }

    /* copy bytes from there */
    for(sofar = 0; sofar < len;){
        if(n > len - sofar)
            n = len - sofar;
        memmove(nb->wp, p, n);
        qcopycnt += n;
        sofar += n;
        nb->wp += n;
        b = b->next;
        if(b == nil)
            break;
        n = BLEN(b);
        p = b->rp;
    }
    iunlock(q);

    return nb;
}
```

Uses BLEN 435b, allocb() 436b, ilock() 79d, iunlock() 80b, and qcopycnt-776 812d.

[qopen] creates a new queue. The `limit` parameter sets the flow control threshold: when the total allocated bytes exceed this limit, producers get back-pressured. The `msg` parameter is actually the initial state bits – passing `Qmsg` here creates a message-mode queue. The `kick` callback, if provided, is called whenever the queue

drains below the flow control mark, giving network drivers a chance to restart hardware transmission. The queue starts in `Qstarve` state because there is no data yet.

⟨*function* `qopen` 409a⟩≡ (813c)

```
/*
 *  called by non-interrupt code
 */
Queue*
qopen(int limit, int msg, void (*kick)(void*), void *arg)
{
    Queue *q;

    q = malloc(sizeof(Queue));
    if(q == nil)
        return nil;

    q->limit = q->inilim = limit;
    q->kick = kick;
    q->arg = arg;
    q->state = msg;

    q->state |= Qstarve;
    q->eof = 0;
    q->noblock = 0;

    return q;
}
```

Uses `Qstarve` 399a and `malloc()` 103c.

⟨*function* `qbypass` 409b⟩≡ (813c)

```
/* open a queue to be bypassed */
Queue*
qbypass(void (*bypass)(void*, Block*), void *arg)
{
    Queue *q;

    q = malloc(sizeof(Queue));
    if(q == nil)
        return nil;

    q->limit = 0;
    q->arg = arg;
    q->bypass = bypass;
    q->state = 0;

    return q;
}
```

Uses `malloc()` 103c.

⟨*function* `notempty` 409c⟩≡ (813c)

```
static int
notempty(void *a)
{
    Queue *q = a;

    return (q->state & Qclosed) || q->bfirst != 0;
}
```

Uses `Qclosed` 399a.

[qwait] implements the sleep/wakeup protocol for readers. It is called with the queue's spin lock held. If the queue is empty, it sets `Qstarve`, drops the lock, and sleeps on the `rr` rendezvous until [notempty] returns true. When it wakes up, it re-acquires the lock and checks again (the loop handles spurious wakeups). For closed queues, the `eof` counter allows up to 3 reads to return 0 (EOF) before returning an error, matching the Unix convention that EOF can be read multiple times.

⟨*function* `qwait` 410a⟩≡ (813c)

```
/*
 *  wait for the queue to be non-empty or closed.
 *  called with q ilocked.
 */
static int
qwait(Queue *q)
{
    /* wait for data */
    for(;;){
        if(q->bfirst != nil)
            break;

        if(q->state & Qclosed){
            if(++q->eof > 3)
                return -1;
            if(*q->err && strcmp(q->err, Ehungup) != 0)
                return -1;
            return 0;
        }

        q->state |= Qstarve;    /* flag requesting producer to wake me */
        iunlock(q);
        sleep(&q->rr, notempty, q);
        ilock(q);
    }
    return 1;
}
```

Uses `Ehungup` 664d, `Qclosed` 399a, `Qstarve` 399a, `ilock()` 79d, `iunlock()` 80b, `notempty()` 409c, and `sleep`.

⟨*function* `qaddlist` 410b⟩≡ (813c)

```
/*
 * add a block list to a queue
 */
void
qaddlist(Queue *q, Block *b)
{
    /* queue the block */
    if(q->bfirst)
        q->blast->next = b;
    else
        q->bfirst = b;
    q->len += blockalloclen(b);
    q->dlen += blocklen(b);
    while(b->next)
        b = b->next;
    q->blast = b;
}
```

Uses `blockalloclen()` 440b and `blocklen()` 440a.

⟨*function* `qremove` 410c⟩≡ (813c)

```
/*
 *  called with q ilocked
```

```
 */
Block*
qremove(Queue *q)
{
    Block *b;

    b = q->bfirst;
    if(b == nil)
        return nil;
    q->bfirst = b->next;
    b->next = nil;
    q->dlen -= BLEN(b);
    q->len -= BALLOC(b);
//    QDEBUG checkb(b, "qremove");
    return b;
}
```

Uses BALLOC 435c and BLEN 435b.

⟨*function* qputback 411a⟩≡ (813c)

```
/*
 *  put a block back to the front of the queue
 *  called with q ilocked
 */
void
qputback(Queue *q, Block *b)
{
    b->next = q->bfirst;
    if(q->bfirst == nil)
        q->blast = b;
    q->bfirst = b;
    q->len += BALLOC(b);
    q->dlen += BLEN(b);
}
```

Uses BALLOC 435c and BLEN 435b.

⟨*function* qwakeup_iunlock 411b⟩≡ (813c)

```
/*
 *  flow control, get producer going again
 *  called with q ilocked
 */
static void
qwakeup_iunlock(Queue *q)
{
    int dowakeup = 0;

    /* if writer flow controlled, restart */
    if((q->state & Qflow) && q->len < q->limit/2){
        q->state &= ~Qflow;
        dowakeup = 1;
    }

    iunlock(q);

    /* wakeup flow controlled writers */
    if(dowakeup){
        if(q->kick)
            q->kick(q->arg);
        wakeup(&q->wr);
    }
}
```

Uses Qflow 399a, iunlock() 80b, and wakeup.

[qbread] and [qread] are the process-context read API. They acquire `rlock` to serialize readers, then call [qwait] which sleeps on `rr` until data arrives or the queue is closed. A key difference between message and stream mode appears here: in stream mode, if the first block is larger than requested, it is split and the remainder is put back with [qputback]; in message mode, excess data is silently discarded, preserving packet boundaries.

⟨*function* qbread 412⟩≡ (813c)

```
/*
 *  get next block from a queue (up to a limit)
 */
Block*
qbread(Queue *q, int len)
{
    Block *b, *nb;
    int n;

    qlock(&q->rlock);
    if(waserror()){
        qunlock(&q->rlock);
        nexterror();
    }

    ilock(q);
    switch(qwait(q)){
    case 0:
        /* queue closed */
        iunlock(q);
        qunlock(&q->rlock);
        poperror();
        return nil;
    case -1:
        /* multiple reads on a closed queue */
        iunlock(q);
        error(q->err);
    }

    /* if we get here, there's at least one block in the queue */
    b = qremove(q);
    n = BLEN(b);

    /* split block if it's too big and this is not a message queue */
    nb = b;
    if(n > len){
        if((q->state&Qmsg) == 0){
            n -= len;
            b = allocb(n);
            memmove(b->wp, nb->rp+len, n);
            b->wp += n;
            qputback(q, b);
        }
        nb->wp = nb->rp + len;
    }

    /* restart producer */
    qwakeup_iunlock(q);

    poperror();
    qunlock(&q->rlock);
```

```
    return nb;
}
```

Uses BLEN 435b, Qmsg 399a, allocb() 436b, ilock() 79d, iunlock() 80b, nexterror, poperror 666d, qlock() 81f, qputback() 411a, qremove() 410c, qunlock() 82c, qwait() 410a, qwakeup_iunlock() 411b, and waserror 666d.

⟨*function* qread 413⟩≡ (813c)

```
/*
 *  read a queue.  if no data is queued, post a Block
 *  and wait on its Rendez.
 */
long
qread(Queue *q, void *vp, int len)
{
    Block *b, *first, **l;
    int m, n;

    qlock(&q->rlock);
    if(waserror()){
        qunlock(&q->rlock);
        nexterror();
    }

    ilock(q);
again:
    switch(qwait(q)){
    case 0:
        /* queue closed */
        iunlock(q);
        qunlock(&q->rlock);
        poperror();
        return 0;
    case -1:
        /* multiple reads on a closed queue */
        iunlock(q);
        error(q->err);
    }

    /* if we get here, there's at least one block in the queue */
    if(q->state & Qcoalesce){
        /* when coalescing, 0 length blocks just go away */
        b = q->bfirst;
        if(BLEN(b) <= 0){
            freeb(qremove(q));
            goto again;
        }

        /*  grab the first block plus as many
         *  following blocks as will completely
         *  fit in the read.
         */
        n = 0;
        l = &first;
        m = BLEN(b);
        for(;;) {
            *l = qremove(q);
            l = &b->next;
            n += m;

            b = q->bfirst;
            if(b == nil)
```

```
                break;
            m = BLEN(b);
            if(n+m > len)
                break;
        }
    } else {
        first = qremove(q);
        n = BLEN(first);
    }

    /* copy to user space outside of the ilock */
    iunlock(q);
    b = bl2mem(vp, first, len);
    ilock(q);

    /* take care of any left over partial block */
    if(b != nil){
        n -= BLEN(b);
        if(q->state & Qmsg)
            freeb(b);
        else
            qputback(q, b);
    }

    /* restart producer */
    qwakeup_iunlock(q);

    poperror();
    qunlock(&q->rlock);
    return n;
}
```

Uses BLEN 435b, Qcoalesce 399a, Qmsg 399a, bl2mem() 445, freeb() 438a, ilock() 79d, iunlock() 80b, nexterror, poperror 666d, qlock() 81f, qputback() 411a, qremove() 410c, qunlock() 82c, qwait() 410a, qwakeup_iunlock() 411b, and waserror 666d.

⟨*function* qnotfull 414a⟩≡ (813c)

```
static int
qnotfull(void *a)
{
    Queue *q = a;

    return q->len < q->limit || (q->state & Qclosed);
}
```

Uses Qclosed 399a.

[qbwrite] is the workhorse behind [qwrite]. It handles the full lifecycle: check for closed/bypass queues, append the block, wake up starved readers, call the `kick` callback for network drivers, and then – critically – block the writer if the queue exceeds its limit. The flow control sleep happens *after* the data is queued, not before. This is deliberate: it ensures that 9P flush messages and SSL records are fully queued before the process can be interrupted, preventing partial writes from corrupting protocol state.

⟨*function* qbwrite 414b⟩≡ (813c)

```
/*
 *  add a block to a queue obeying flow control
 */
long
qbwrite(Queue *q, Block *b)
{
    int n, dowakeup;
    Proc *p;
```

```
    n = BLEN(b);

    if(q->bypass){
        (*q->bypass)(q->arg, b);
        return n;
    }

    dowakeup = 0;
    qlock(&q->wlock);
    if(waserror()){
        if(b != nil)
            freeb(b);
        qunlock(&q->wlock);
        nexterror();
    }

    ilock(q);

    /* give up if the queue is closed */
    if(q->state & Qclosed){
        iunlock(q);
        error(q->err);
    }

    /* if nonblocking, don't queue over the limit */
    if(q->len >= q->limit){
        if(q->noblock){
            iunlock(q);
            freeb(b);
            noblockcnt += n;
            qunlock(&q->wlock);
            poperror();
            return n;
        }
    }

    /* queue the block */
    if(q->bfirst)
        q->blast->next = b;
    else
        q->bfirst = b;
    q->blast = b;
    b->next = 0;
    q->len += BALLOC(b);
    q->dlen += n;
//    QDEBUG checkb(b, "qbwrite");
    b = nil;

    /* make sure other end gets awakened */
    if(q->state & Qstarve){
        q->state &= ~Qstarve;
        dowakeup = 1;
    }
    iunlock(q);

    /*  get output going again */
    if(q->kick && (dowakeup || (q->state&Qkick)))
        q->kick(q->arg);
```

```
    /* wakeup anyone consuming at the other end */
    if(dowakeup){
        p = wakeup(&q->rr);

        /* if we just wokeup a higher priority process, let it run */
        if(p != nil && p->priority > up->priority)
            sched();
    }

    /*
     *  flow control, wait for queue to get below the limit
     *  before allowing the process to continue and queue
     *  more.  We do this here so that postnote can only
     *  interrupt us after the data has been queued.  This
     *  means that things like 9p flushes and ssl messages
     *  will not be disrupted by software interrupts.
     *
     *  Note - this is moderately dangerous since a process
     *  that keeps getting interrupted and rewriting will
     *  queue infinite crud.
     */
    for(;;){
        if(q->noblock || qnotfull(q))
            break;

        ilock(q);
        q->state |= Qflow;
        iunlock(q);
        sleep(&q->wr, qnotfull, q);
    }
    USED(b);

    qunlock(&q->wlock);
    poperror();
    return n;
}
```

Uses BALLOC 435c, BLEN 435b, Qclosed 399a, Qflow 399a, Qkick 399a, Qstarve 399a, freeb() 438a, ilock() 79d, iunlock() 80b, nexterror, noblockcnt-777 812d, poperror 666d, qlock() 81f, qnotfull() 414a, qunlock() 82c, sched, sleep, up, wakeup, and waserror 666d.

[qiwrite] is a special variant used by `print()` and other kernel output functions. Unlike [qbwrite], it does not use `qlock` because it may be called from interrupt context or from contexts where no process is running (e.g., during boot). It also uses `iallocb` instead of `allocb` and has a hardcoded 16KB limit to prevent runaway kernel messages from consuming all memory.

⟨*function* `qiwrite` 416⟩≡ (813c)

```
/*
 *  used by print() to write to a queue.  Since we may be arch_splhi or not in
 *  a process, don't qlock.
 *
 *  this routine merges adjacent blocks if block n+1 will fit into
 *  the free space of block n.
 */
int
qiwrite(Queue *q, void *vp, int len)
{
    int n, sofar, dowakeup;
    Block *b;
    byte *p = vp;
```

```
    dowakeup = 0;

    sofar = 0;
    do {
        n = len-sofar;
        if(n > Maxatomic)
            n = Maxatomic;

        b = iallocb(n);
        if(b == nil)
            break;
        memmove(b->wp, p+sofar, n);
        b->wp += n;

        ilock(q);

        /* we use an artificially high limit for kernel prints since anything
         * over the limit gets dropped
         */
        if(q->dlen >= 16*1024){
            iunlock(q);
            freeb(b);
            break;
        }

//        QDEBUG checkb(b, "qiwrite");
        if(q->bfirst)
            q->blast->next = b;
        else
            q->bfirst = b;
        q->blast = b;
        q->len += BALLOC(b);
        q->dlen += n;

        if(q->state & Qstarve){
            q->state &= ~Qstarve;
            dowakeup = 1;
        }

        iunlock(q);

        if(dowakeup){
            if(q->kick)
                q->kick(q->arg);
            wakeup(&q->rr);
        }

        sofar += n;
    } while(sofar < len && (q->state & Qmsg) == 0);

    return sofar;
}
```

Uses BALLOC 435c, Maxatomic-779 813b, Qmsg 399a, Qstarve 399a, freeb() 438a, iallocb() 437c, ilock() 79d, iunlock() 80b, and wakeup.

⟨*function* qfree 417⟩≡ (813c)

```
/*
 *  be extremely careful when calling this,
 *  as there is no reference accounting
 */
```

```
void
qfree(Queue *q)
{
    qclose(q);
    free(q);
}
```

Uses `free()` 104a and `qclose()` 418a.

[qclose] and [qhangup] both mark a queue as `Qclosed`, but they differ in an important way: [qclose] immediately frees all queued blocks (hard close), while [qhangup] leaves the data in place so the consumer can drain remaining bytes before seeing the hangup (graceful shutdown). The `eof` counter in [qwait] allows a closed queue to be read a few times before returning an error, giving the reader a chance to process the final data. [qreopen] reverses a close, restoring the queue to its initial state – this is used when a network connection is recycled.

⟨*function* `qclose` 418a⟩≡ (813c)

```
/*
 *  Mark a queue as closed.  No further IO is permitted.
 *  All blocks are released.
 */
void
qclose(Queue *q)
{
    Block *bfirst;

    if(q == nil)
        return;

    /* mark it */
    ilock(q);
    q->state |= Qclosed;
    q->state &= ~(Qflow|Qstarve);
    strcpy(q->err, Ehungup);
    bfirst = q->bfirst;
    q->bfirst = 0;
    q->len = 0;
    q->dlen = 0;
    q->noblock = 0;
    iunlock(q);

    /* free queued blocks */
    freeblist(bfirst);

    /* wake up readers/writers */
    wakeup(&q->rr);
    wakeup(&q->wr);
}
```

Uses `Ehungup` 664d, `Qclosed` 399a, `Qflow` 399a, `Qstarve` 399a, `freeblist()` 439a, `ilock()` 79d, `iunlock()` 80b, and `wakeup`.

⟨*function* `qhangup` 418b⟩≡ (813c)

```
/*
 *  Mark a queue as closed.  Wakeup any readers.  Don't remove queued
 *  blocks.
 */
void
qhangup(Queue *q, char *msg)
{
    /* mark it */
    ilock(q);
    q->state |= Qclosed;
    if(msg == 0 || *msg == 0)
```

```
        strcpy(q->err, Ehungup);
    else
        strncpy(q->err, msg, ERRMAX-1);
    iunlock(q);

    /* wake up readers/writers */
    wakeup(&q->rr);
    wakeup(&q->wr);
}
```

Uses ERRMAX, Ehungup 664d, Qclosed 399a, ilock() 79d, iunlock() 80b, and wakeup.

⟨*function* qiclosed 419a⟩≡ (813c)

```
/*
 *  return non-zero if the q is hungup
 */
int
qisclosed(Queue *q)
{
    return q->state & Qclosed;
}
```

Uses Qclosed 399a.

⟨*function* qreopen 419b⟩≡ (813c)

```
/*
 *  mark a queue as no longer hung up
 */
void
qreopen(Queue *q)
{
    ilock(q);
    q->state &= ~Qclosed;
    q->state |= Qstarve;
    q->eof = 0;
    q->limit = q->inilim;
    iunlock(q);
}
```

Uses Qclosed 399a, Qstarve 399a, ilock() 79d, and iunlock() 80b.

⟨*function* qlen 419c⟩≡ (813c)

```
/*
 *  return bytes queued
 */
int
qlen(Queue *q)
{
    return q->dlen;
}
```

⟨*function* qwindow 419d⟩≡ (813c)

```
/*
 * return space remaining before flow control
 */
int
qwindow(Queue *q)
{
    int l;

    l = q->limit - q->len;
    if(l < 0)
```

```
        l = 0;
    return l;
}
```

⟨*function* qcanread 420a⟩≡ (813c)

```
/*
 *  return true if we can read without blocking
 */
int
qcanread(Queue *q)
{
    return q->bfirst!=0;
}
```

⟨*function* qsetlimit 420b⟩≡ (813c)

```
/*
 *  change queue limit
 */
void
qsetlimit(Queue *q, int limit)
{
    q->limit = limit;
}
```

⟨*function* qnoblock 420c⟩≡ (813c)

```
/*
 *  set blocking/nonblocking
 */
void
qnoblock(Queue *q, bool onoff)
{
    q->noblock = onoff;
}
```

⟨*function* qflush 420d⟩≡ (813c)

```
/*
 *  flush the output queue
 */
void
qflush(Queue *q)
{
    Block *bfirst;

    /* mark it */
    ilock(q);
    bfirst = q->bfirst;
    q->bfirst = 0;
    q->len = 0;
    q->dlen = 0;
    iunlock(q);

    /* free queued blocks */
    freeblist(bfirst);

    /* wake up readers/writers */
    wakeup(&q->wr);
}
```

Uses `freeblist()` 439a, `ilock()` 79d, `iunlock()` 80b, and `wakeup`.

⟨*function* `qfull` 421⟩≡ (813c)

```
int
qfull(Queue *q)
{
    return q->state & Qflow;
}
```

Uses `Qflow` 399a.

Chapter 17

Filesystems

In Plan 9, every device driver implements the `Dev` interface, which is essentially a file server interface. This chapter covers two in-kernel "filesystems" — the root device (`#/`) and the boot device — plus the generic helper functions ([devdir], [devgen], [devwalk], [devstat], [devdirread]) that factor out the boilerplate every driver needs to present its files as a directory tree. The chapter ends with `Block`, the I/O buffer structure used by queues and networking.

`Dirtab` is a compact in-memory directory entry used by device drivers to describe their synthetic files. Unlike a full `DirEntry` (which is the on-the-wire 9P format), `Dirtab` contains just the fields a driver needs to enumerate its files: a name, a `Qid`, a length, and permissions. The [devgen] family of helpers iterate over arrays of `Dirtab` entries to implement walk, stat, and directory reads.

⟨*struct* `Dirtab` 422a⟩≡ (804b)
```
struct Dirtab
{
  char  name[KNAMELEN];
  Qid qid;
  vlong length;
  long  perm;
};
```

17.1 The root filesystem

The root device (`#/`) is the in-kernel filesystem that provides the top-level directory structure: `/dev`, `/proc`, `/bin`, `/env`, `/net`, `/srv`, and so on. These are not real directories on disk — they are just entries in a static array that serve as mount points. When the system boots, other devices and file servers get bound onto these directories (e.g., `#c` onto `/dev`, `#p` onto `/proc`). Notice how many of the `Dev` methods delegate to default implementations (`devinit`, `devshutdown`, `devcreate`, `devbread`, `devbwrite`, `devremove`, `devwstat`). The root device only needs custom implementations for `rootreset` (to populate the directory entries), `rootread`/`rootwrite` (to serve file content), and the walk/stat/open methods that use its specific `Dirtab` arrays.

⟨*global* `rootdevtab` 422b⟩≡ (730c)
```
Dev rootdevtab = {
    .dc       = '/',
    .name     = "root",

    .reset    = rootreset,
    .init     = devinit,
    .shutdown = devshutdown,
    .attach   = rootattach,
    .walk     = rootwalk,
    .stat     = rootstat,
    .open     = rootopen,
```

```
    .create   = devcreate,
    .close    = rootclose,
    .read     = rootread,
    .bread    = devbread,
    .write    = rootwrite,
    .bwrite   = devbwrite,
    .remove   = devremove,
    .wstat    = devwstat,
};
```

Uses devbread() 434a, devbwrite() 434b, devcreate() 810, devinit() 810, devremove() 810, devshutdown() 810, devwstat() 810, rootattach() 730c, rootclose() 730b, rootopen() 730c, rootread() 425b, rootreset() 424c, rootstat() 730c, rootwalk() 730c, and rootwrite() 730c.

⟨*devroot enum Qxxx* 423a⟩≡ (730c)

```
enum
{
    Qdir = 0,
    Qboot = 0x1000,

    Nrootfiles = 32,
    Nbootfiles = 32,
};
```

The root device stores its files in two parallel arrays: `dir` holds the `Dirtab` entries (names, Qids, permissions) and `data` holds pointers to file contents. The `Dirlist` groups these arrays together with counters. There are two instances: `rootlist` for the root directory and `bootlist` for `/boot/`.

⟨*struct* `Dirlist` 423b⟩≡ (730c)

```
struct Dirlist
{
    uint base; // for unique qids
    Dirtab *dir;
    byte **data;
    int ndir; // number of dir used
    int mdir; // max dir entries
};
```

⟨*globals rootdir, rootdata, rootlist* 423c⟩≡ (730c)

```
static Dirtab rootdir[Nrootfiles] = {
  {
    .name = "#/",
    .qid = {Qdir, 0, QTDIR},
    .length = 0,
    .perm = DMDIR|0555,
  },
  {
    .name = "boot",
    .qid = {Qboot, 0, QTDIR},
    .length = 0,
    .perm = DMDIR|0555,
  }
};
static byte *rootdata[Nrootfiles];
static Dirlist rootlist =
{
  .base = 0,
  .dir = rootdir,
  .data = rootdata,
  .ndir = 2,
  .mdir = Nrootfiles
```

```
};
```

Uses DMDIR, Nrootfiles-828 423a, QTDIR 70b, Qboot-827 423a, Qdir-826 423a, rootdata-831 423c, and rootdir-830 423c.

⟨*function* addlist 424a⟩≡ (730c)

```
/*
 *  add a file to the list
 */
static void
addlist(Dirlist *l, char *name, byte *contents, ulong len, int perm)
{
    Dirtab *d;

    if(l->ndir >= l->mdir)
        panic("too many root files");
    l->data[l->ndir] = contents;
    d = &l->dir[l->ndir];
    strcpy(d->name, name);
    d->length = len;
    d->perm = perm;
    d->qid.type = 0;
    d->qid.vers = 0;
    d->qid.path = ++l->ndir + l->base;
    if(perm & DMDIR)
        d->qid.type |= QTDIR;
}
```

Uses DMDIR and QTDIR 70b.

⟨*function* addrootdir 424b⟩≡ (730c)

```
/*
 *  add a root directory
 */
static void
addrootdir(char *name)
{
    addlist(&rootlist, name, nil, 0, DMDIR|0555);
}
```

Uses DMDIR, addlist() 424a, and rootlist-832 423c.

[rootreset] is called once at boot to populate the root directory with the standard Plan 9 top-level directories. These are all empty directories that will later be populated via `bind` and `mount` operations. For example, `/dev` will have `#c` (the console device) bound onto it, `/proc` will get `#p`, and so on.

⟨*method rootreset* 424c⟩≡ (730c)

```
static void
rootreset(void)
{
    addrootdir("bin");
    addrootdir("dev");
    addrootdir("env");
    addrootdir("fd");
    addrootdir("mnt");
    addrootdir("net");
    addrootdir("proc");
    addrootdir("root");
    addrootdir("srv");
    addrootdir("sys");
}
```

Uses addrootdir() 424b.

[rootgen] is the `Devgen` callback for the root device. It dispatches on the Qid path to decide whether to list root directories or boot files. When called with `Qdir` it delegates to [devgen] with the `rootlist` table; when called with `Qboot` it uses `bootlist`. For individual files (the default case), it computes an index by subtracting the base Qid and looks up the corresponding `Dirtab` entry directly.

⟨*function* `rootgen` 425a⟩≡ (730c)

```
static int
rootgen(Chan *c, char *name, Dirtab*, int, int s, DirEntry *dp)
{
    int t;
    Dirtab *d;
    Dirlist *l;

    switch((int)c->qid.path){
    case Qdir:
        if(s == DEVDOTDOT){
            devdir(c, (Qid){Qdir, 0, QTDIR}, "#/", 0, eve, 0555, dp);
            return 1;
        }
        return devgen(c, name, rootlist.dir, rootlist.ndir, s, dp);
    case Qboot:
        if(s == DEVDOTDOT){
            devdir(c, (Qid){Qdir, 0, QTDIR}, "#/", 0, eve, 0555, dp);
            return 1;
        }
        return devgen(c, name, bootlist.dir, bootlist.ndir, s, dp);
    default:
        if(s == DEVDOTDOT){
            if((int)c->qid.path < Qboot)
                devdir(c, (Qid){Qdir, 0, QTDIR}, "#/", 0, eve, 0555, dp);
            else
                devdir(c, (Qid){Qboot, 0, QTDIR}, "#/", 0, eve, 0555, dp);
            return 1;
        }
        if(s != 0)
            return -1;
        if((int)c->qid.path < Qboot){
            t = c->qid.path-1;
            l = &rootlist;
        }else{
            t = c->qid.path - Qboot - 1;
            l = &bootlist;
        }
        if(t >= l->ndir)
            return -1;
        d = &l->dir[t];
        devdir(c, d->qid, d->name, d->length, eve, d->perm, dp);
        return 1;
    }
}
```

Uses `DEVDOTDOT` 805, `QTDIR` 70b, `Qboot-827` 423a, `Qdir-826` 423a, `bootlist-835` 426, `devdir()` 427c, `devgen()` 428, `eve` 591a, and `rootlist-832` 423c.

[rootread] dispatches on the Qid. For directories (`Qdir` or `Qboot`) it delegates to [devdirread] with [rootgen]. For regular files, it computes an index from the Qid path, looks up the `data` array, and copies bytes. The content of root files comes from the `data2s` tool that converts binaries into C arrays at link time.

⟨*method rootread* 425b⟩≡ (730c)

```
static long
rootread(Chan *c, void *buf, long n, vlong off)
```

```
{
    ulong t;
    Dirtab *d;
    Dirlist *l;
    byte *data;
    ulong offset = off;

    t = c->qid.path;
    switch(t){
    case Qdir:
    case Qboot:
        return devdirread(c, buf, n, nil, 0, rootgen);
    }

    if(t<Qboot)
        l = &rootlist;
    else{
        t -= Qboot;
        l = &bootlist;
    }

    t--;
    if(t >= l->ndir)
        error(Egreg);

    d = &l->dir[t];
    data = l->data[t];
    if(offset >= d->length)
        return 0;
    if(offset+n > d->length)
        n = d->length - offset;
    memmove(buf, data+offset, n);
    return n;
}
```

Uses Egreg 664d, Qboot-827 423a, Qdir-826 423a, bootlist-835 426, devdirread() 432c, rootgen() 425a, and rootlist-832 423c.

17.2 The /boot/ filesystem

The /boot/ filesystem is a second in-kernel file store, separate from the root directory. At link time, essential binaries (the boot program, /bin/rc, /bin/ls, etc.) are converted to C data arrays by data2s and registered via [addbootfile]. This gives the kernel a minimal set of executables it needs to bootstrap without any external storage — enough to run the boot script, which then mounts the real filesystem and binds everything into place. Note how bootlist.base is set to Qboot (0x1000), ensuring that boot file Qids live in a different range from root directory Qids. This lets [rootgen] distinguish which Dirlist to consult based solely on the Qid path.

⟨*globals bootdir, bootdata, bootlist* 426⟩≡ (730c)

```
static Dirtab bootdir[Nbootfiles] = {
  {
    .name = "boot",
    .qid = {Qboot, 0, QTDIR},
    .length = 0,
    .perm = DMDIR|0555,
  }
};

static byte *bootdata[Nbootfiles];
```

```
static Dirlist bootlist =
{
    .base = Qboot,
    .dir = bootdir,
    .data = bootdata,
    .ndir = 1,
    .mdir = Nbootfiles
};
```

Uses DMDIR, Nbootfiles-829 423a, QTDIR 70b, Qboot-827 423a, bootdata-834 426, and bootdir-833 426.

⟨*function* addbootfile 427a⟩≡ (730c)

```
/*
 *  add a boot file
 */
void
addbootfile(char *name, byte *contents, ulong len)
{
    addlist(&bootlist, name, contents, len, 0555);
}
```

Uses addlist() 424a and bootlist-835 426.

17.3 Helpers

The dev.c helper functions — [devdir], [devgen], [devwalk], [devstat], [devdirread], [devopen] — implement the generic walk/stat/read-directory logic once, parameterized by a Dirtab array and a Devgen callback. A typical device driver just declares its file table as a static Dirtab array and passes it to these helpers.

⟨Qidtype *cases* 427b⟩≡ (70b) 446▷

```
QTMOUNT = 0x10,    /* type bit for mounted channel */
```

⟨*function* devdir 427c⟩≡ (810)

```
void
devdir(Chan *c, Qid qid, char *n, vlong length, char *user, long perm, DirEntry *db)
{
    db->name = n;
    if(c->flag&CMSG)
        qid.type |= QTMOUNT;
    db->qid = qid;
    db->type = devtab[c->type]->dc;
    db->dev = c->dev;
    db->mode = perm;
    db->mode |= qid.type << 24;
    db->atime = seconds();
    db->mtime = kerndate;
    db->length = length;
    db->uid = user;
    db->gid = eve;
    db->muid = user;
}
```

Uses CMSG 582c, QTMOUNT 70b, eve 591a, and seconds() 263b.

The long comment before [devgen] documents a subtle tension in the Devgen interface. Rule (i) says a generator should list a directory's children; rule (ii) says it should list siblings. [devgen] follows rule (i) only

when `c` is the root directory; otherwise it follows (ii). When `name` is non-nil, [devgen] does a linear search by name (for walks); when `name` is nil and `s` is a numeric index, it returns the s-th entry (for directory reads).

⟨*function* devgen 428⟩≡ (810)

```
/*
 * (here, Devgen is the prototype; devgen is the function in dev.c.)
 *
 *
 * a Devgen is expected to return the directory entry for ".."
 * if you pass it s==DEVDOTDOT (-1).  otherwise...
 *
 * there are two contradictory rules.
 *
 * (i) if c is a directory, a Devgen is expected to list its children
 * as you iterate s.
 *
 * (ii) whether or not c is a directory, a Devgen is expected to list
 * its siblings as you iterate s.
 *
 * devgen always returns the list of children in the root
 * directory.  thus it follows (i) when c is the root and (ii) otherwise.
 * many other Devgens follow (i) when c is a directory and (ii) otherwise.
 *
 * devwalk assumes (i).  it knows that devgen breaks (i)
 * for children that are themselves directories, and explicitly catches them.
 *
 * devstat assumes (ii).  if the Devgen in question follows (i)
 * for this particular c, devstat will not find the necessary info.
 * with our particular Devgen functions, this happens only for
 * directories, so devstat makes something up, assuming
 * c->name, c->qid, eve, DMDIR|0555.
 *
 * devdirread assumes (i).  the callers have to make sure
 * that the Devgen satisfies (i) for the chan being read.
 */
/*
 * the zeroth element of the table MUST be the directory itself for ..
*/
int
devgen(Chan *c, char *name, Dirtab *tab, int ntab, int i, DirEntry *dp)
{
    if(tab == 0)
        return -1;
    if(i == DEVDOTDOT){
        /* nothing */
    }else if(name){
        for(i=1; i<ntab; i++)
            if(strcmp(tab[i].name, name) == 0)
                break;
        if(i==ntab)
            return -1;
        tab += i;
    }else{
        /* skip over the first element, that for . itself */
        i++;
        if(i >= ntab)
            return -1;
        tab += i;
    }
    devdir(c, tab->qid, tab->name, tab->length, eve, tab->perm, dp);
```

```
    return 1;
}
```

Uses DEVDOTDOT 805, devdir() 427c, and eve 591a.

[devclone] creates a new Chan that copies c's type, device number, Qid, and auxiliary pointer. The clone must not be open (COPEN)—cloning is only used during walk, before the file is opened.

⟨*function* devclone 429a⟩≡ (810)

```
Chan*
devclone(Chan *c)
{
    Chan *nc;

    if(c->flag & COPEN)
        panic("clone of open file type %C\n", devtab[c->type]->dc);

    nc = newchan();

    nc->type = c->type;
    nc->dev = c->dev;
    nc->mode = c->mode;
    nc->qid = c->qid;
    nc->offset = c->offset;
    nc->umh = nil;
    nc->aux = c->aux;
    nc->mqid = c->mqid;
    nc->mcp = c->mcp;
    return nc;
}
```

Uses COPEN 294b and newchan() 295e.

[devwalk] is the generic walk implementation. Given a channel c and path components name, it clones c into nc, then walks each component by calling the Devgen callback. On partial success (some but not all names walked), it drops the clone and returns just the Qids reached so far — the 9P layer uses this to decide whether to retry with a different mount point.

⟨*function* devwalk 429b⟩≡ (810)

```
Walkqid*
devwalk(Chan *c, Chan *nc, char **name, int nname, Dirtab *tab, int ntab, Devgen *gen)
{
    int i, j;
    bool alloc;
    Walkqid *wq;
    char *n;
    DirEntry dir;

    if(nname > 0)
        error_if_not_dir(c);

    alloc = false;
    wq = smalloc(sizeof(Walkqid)+(nname-1)*sizeof(Qid));
    if(waserror()){
        if(alloc && wq->clone!=nil)
            cclose(wq->clone);
        free(wq);
        return nil;
    }
    if(nc == nil){
        nc = devclone(c);
        nc->type = 0;   /* device doesn't know about this channel yet */
```

```
        alloc = true;
    }
    wq->clone = nc;

    for(j=0; j<nname; j++){
        if(!(nc->qid.type&QTDIR)){
            if(j==0)
                error(Enotdir);
            goto Done;
        }
        n = name[j];
        if(strcmp(n, ".") == 0){
    Accept:
            wq->qid[wq->nqid++] = nc->qid;
            continue;
        }
        if(strcmp(n, "..") == 0){
            if((*gen)(nc, nil, tab, ntab, DEVDOTDOT, &dir) != 1){
                print("devgen walk .. in dev%s %llux broken\n",
                    devtab[nc->type]->name, nc->qid.path);
                error("broken devgen");
            }
            nc->qid = dir.qid;
            goto Accept;
        }
        /*
         * Ugly problem: If we're using devgen, make sure we're
         * walking the directory itself, represented by the first
         * entry in the table, and not trying to step into a sub-
         * directory of the table, e.g. /net/net. Devgen itself
         * should take care of the problem, but it doesn't have
         * the necessary information (that we're doing a walk).
         */
        if(gen==devgen && nc->qid.path!=tab[0].qid.path)
            goto Notfound;
        for(i=0;; i++) {
            switch((*gen)(nc, n, tab, ntab, i, &dir)){
            case -1:
            Notfound:
                if(j == 0)
                    error(Enonexist);
                kstrcpy(up->errstr, Enonexist, ERRMAX);
                goto Done;
            case 0:
                continue;
            case 1:
                if(strcmp(n, dir.name) == 0){
                    nc->qid = dir.qid;
                    goto Accept;
                }
                continue;
            }
        }
    }
    /*
     * We processed at least one name, so will return some data.
     * If we didn't process all nname entries succesfully, we drop
     * the cloned channel and return just the Qids of the walks.
     */
Done:
```

```
    poperror();
    if(wq->nqid < nname){
        if(alloc)
            cclose(wq->clone);
        wq->clone = nil;
    }else if(wq->clone){
        /* attach cloned channel to same device */
        wq->clone->type = c->type;
    }
    return wq;
}
```

Uses DEVDOTDOT 805, ERRMAX, QTDIR 70b, cclose, devclone() 429a, devgen() 428, error_if_not_dir() 323b, false, free() 104a, kstrcpy() 669a, poperror 666d, print, smalloc() 104b, true, up, and waserror 666d.

[devstat] iterates the Devgen to find the entry matching c's Qid path, then serializes it with convD2M. If the generator never finds a match and c is a directory, it fabricates a minimal entry from c->qid and c->path — this handles the case where the generator follows rule (i) but [devstat] needs rule (ii).

⟨*function* devstat 431⟩≡ (810)

```
int
devstat(Chan *c, byte *db, int n, Dirtab *tab, int ntab, Devgen *gen)
{
    int i;
    DirEntry dir;
    char *p, *elem;

    for(i=0;; i++){
        switch((*gen)(c, nil, tab, ntab, i, &dir)){
        case -1:
            if(c->qid.type & QTDIR){
                if(c->path == nil)
                    elem = "???";
                else if(strcmp(c->path->s, "/") == 0)
                    elem = "/";
                else
                    for(elem=p=c->path->s; *p; p++)
                        if(*p == '/')
                            elem = p+1;
                devdir(c, c->qid, elem, 0, eve, DMDIR|0555, &dir);
                n = convD2M(&dir, db, n);
                if(n == 0)
                    error(Ebadarg);
                return n;
            }

            error(Enonexist);
        case 0:
            break;
        case 1:
            if(c->qid.path == dir.qid.path) {
                if(c->flag&CMSG)
                    dir.mode |= DMMOUNT;
                n = convD2M(&dir, db, n);
                if(n == 0)
                    error(Ebadarg);
                return n;
            }
            break;
        }
    }
```

```
    return -1; // unreachable
}
```

Uses CMSG 582c, DMDIR, DMMOUNT 337a, Ebadarg 664d, QTDIR 70b, devdir() 427c, and eve 591a.

⟨Dirmode *cases* 432a⟩≡ (337a) 651d▷

```
DMMOUNT = 0x10000000,  /* mode bit for mounted channel */
```

⟨Chan *other fields* 432b⟩+≡ (68b) ◁369b 745b▷

```
int dri;      /* devdirread index */
```

[devdirread] is the generic `read` implementation for directories. It iterates the `Devgen` starting from `c->dri` (the saved read index), serializing entries with `convD2M` until the output buffer is full or the generator returns −1. The `dri` field in `Chan` remembers the position across successive `read` calls, so the user-space `dirread` library function can consume directory entries incrementally.

⟨*function* devdirread 432c⟩≡ (810)

```
long
devdirread(Chan *c, char *d, long n, Dirtab *tab, int ntab, Devgen *gen)
{
    long m, dsz;
    DirEntry dir;

    for(m=0; m<n; c->dri++) {
        switch((*gen)(c, nil, tab, ntab, c->dri, &dir)){
        case -1:
            return m;

        case 0:
            break;

        case 1:
            dsz = convD2M(&dir, (byte*)d, n-m);
            if(dsz <= BIT16SZ){ /* <= not < because this isn't stat; read is stuck */
                if(m == 0)
                    error(Eshort);
                return m;
            }
            m += dsz;
            d += dsz;
            break;
        }
    }

    return m;
}
```

Uses Eshort 664d.

[devpermcheck] implements Plan 9's simplified permission model. There are only three categories: the file owner, the host owner (`eve`), and everyone else. The `access` array maps the four open modes (`OREAD`, `OWRITE`, `ORDWR`, `OEXEC`) to the corresponding permission bits (0400, 0200, 0600, 0100). Shifting the file's permission bits left by 0, 3, or 6 positions aligns the relevant group's bits with the check mask.

⟨*function* devpermcheck 432d⟩≡ (810)

```
/*
 * error(Eperm) if open permission not granted for up->user.
 */
void
devpermcheck(char *fileuid, ulong perm, int omode)
{
    ulong t;
```

```
    static int access[] = { 0400, 0200, 0600, 0100 };

    if(strcmp(up->user, fileuid) == 0)
        perm <<= 0;
    else
    if(strcmp(up->user, eve) == 0)
        perm <<= 3;
    else
        perm <<= 6;

    t = access[omode&3];
    if((t&perm) != t)
        error(Eperm);
}
```

Uses eve 591a and up.

[devopen] iterates the generator to find the entry matching c's Qid, checks permissions via [devpermcheck], then marks the channel as open. If the generator exhausts without finding a match (e.g., for a directory not explicitly listed), it skips the permission check—directories are always readable. Directories may only be opened for reading.

⟨*function* devopen 433a⟩≡ (810)

```
Chan*
devopen(Chan *c, int omode, Dirtab *tab, int ntab, Devgen *gen)
{
    int i;
    DirEntry dir;

    for(i=0;; i++) {
        switch((*gen)(c, nil, tab, ntab, i, &dir)){
        case -1:
            goto Return;
        case 0:
            break;
        case 1:
            if(c->qid.path == dir.qid.path) {
                devpermcheck(dir.uid, dir.mode, omode);
                goto Return;
            }
            break;
        }
    }
Return:
    c->offset = 0;
    if((c->qid.type&QTDIR) && omode!=OREAD)
        error(Eperm);
    c->mode = openmode(omode);
    c->flag |= COPEN;
    return c;
}
```

Uses COPEN 294b, OREAD 853i, QTDIR 70b, devpermcheck() 432d, and openmode() 302b.

⟨Dev *methods* 433b⟩+≡ (71b) ◁348c 628c▷

```
Block* (*bread)(Chan*, long, ulong);
long   (*bwrite)(Chan*, Block*, ulong);
```

[devbread] and [devbwrite] are default Dev methods that bridge between the block-oriented interface (used by devmnt for 9P communication) and the byte-oriented read/write methods that most devices implement.

They allocate or consume a `Block`, calling the device's `read` or `write` method on its data region.

⟨*function* `devbread` 434a⟩≡ (810)
```
Block*
devbread(Chan *c, long n, ulong offset)
{
    Block *bp;

    bp = allocb(n);
    if(bp == 0)
        error(Enomem);
    if(waserror()) {
        freeb(bp);
        nexterror();
    }
    bp->wp += devtab[c->type]->read(c, bp->wp, n, offset);
    poperror();
    return bp;
}
```
Uses `allocb()` 436b, `freeb()` 438a, `nexterror`, `poperror` 666d, and `waserror` 666d.

⟨*function* `devbrwrite` 434b⟩≡ (810)
```
long
devbwrite(Chan *c, Block *bp, ulong offset)
{
    long n;

    if(waserror()) {
        freeb(bp);
        nexterror();
    }
    n = devtab[c->type]->write(c, bp->rp, BLEN(bp), offset);
    poperror();
    freeb(bp);

    return n;
}
```
Uses `BLEN` 435b, `freeb()` 438a, `nexterror`, `poperror` 666d, and `waserror` 666d.

17.4 SimpleFS

17.5 `Block`

A `Block` is the fundamental I/O buffer in Plan 9, used by queues, pipes, and the networking stack. Data lives between `rp` (read pointer) and `wp` (write pointer), with header space before `rp` that lets protocol layers prepend headers without copying. Blocks chain via `next` for scatter/gather I/O.

⟨*enum* `blockflag` 434c⟩≡ (804b)
```
/* flag values */
enum BlockFlags
{
  BINTR = (1<<0),
  BFREE = (1<<1),
  Bipck = (1<<2),   /* ip checksum */
  Budpck  = (1<<3),   /* udp checksum */
  Btcpck  = (1<<4),   /* tcp checksum */
  Bpktck  = (1<<5),   /* packet checksum */
};
```

⟨*struct* Block 435a⟩≡ (804b)

```
struct Block
{
  long  ref;
  Block*  next;
  Block*  list;
  byte*  rp;     /* first unconsumed byte */
  byte*  wp;     /* first empty byte */
  byte*  lim;      /* 1 past the end of the buffer */
  byte*  base;     /* start of the buffer */
  void  (*free)(Block*);
  ushort  flag;
  ushort  checksum;   /* IP checksum of complete packet (minus media header) */
};
```

The layout of a `Block` buffer is:

```
base          rp               wp            lim
 |             |                |             |
 v             v                v             v
 +-------------+----------------+-------------+
 | header      | data           | free        |
 | space       | (BLEN bytes)   | trailer     |
 | (Hdrspc)    |                | space       |
 +-------------+----------------+-------------+
 <--------- BALLOC = lim - base ------------>
```

The region before `rp` is header space (`Hdrspc` bytes). Protocol layers prepend headers by decrementing `rp`, avoiding copies—an IP header gets pushed in front of a TCP segment that already sits in front of the payload, all without touching the data bytes. `BLEN` gives the number of data bytes currently between `rp` and `wp`; `BALLOC` gives the total allocated size. When the initial `Hdrspc` runs out, [padblock] allocates a bigger buffer and copies; the common case skips the copy.

⟨*function* BLEN 435b⟩≡ (804b)

```
#define BLEN(s) ((s)->wp - (s)->rp)
```

⟨*function* BALLOC 435c⟩≡ (804b)

```
#define BALLOC(s) ((s)->lim - (s)->base)
```

⟨*constant* BLOCKALIGN*(arm)* 435d⟩≡ (1005)

```
// possible alternative fix: git show 2342e706e42d1e6653c51cc1433f44816cf53fe6
#define BLOCKALIGN  32          /* only used in allocb.c */
```

`_allocb` allocates a `Block` and its data buffer in a single `mallocz` call. The data region is aligned to `BLOCKALIGN` (32 bytes for DMA compatibility). The key subtlety is that `rp` starts near the end of the buffer, leaving `Hdrspc` at the front so protocol layers can prepend headers by decrementing `rp` without copying.

⟨*function* _allocb 435e⟩≡ (811)

```
static Block*
_allocb(int size)
{
    Block *b;
    ulong addr;

    if((b = mallocz(sizeof(Block)+size+Hdrspc, false)) == nil)
        return nil;

    b->next = nil;
    b->list = nil;
```

```
    b->free = 0;
    b->flag = 0;
    b->ref = 0;
    arch_xinc(&b->ref); // why not incref?? or at least inccnt?

    /* align start of data portion by rounding up */
    addr = (ulong)b;
    addr = ROUND(addr + sizeof(Block), BLOCKALIGN);
    b->base = (byte*)addr;

    /* align end of data portion by rounding down */
    b->lim = ((byte*)b) + msize(b);
    addr = (ulong)(b->lim);
    addr = addr & ~(BLOCKALIGN-1);
    b->lim = (byte*)addr;

    /* leave sluff at beginning for added headers */
    b->rp = b->lim - ROUND(size, BLOCKALIGN);
    if(b->rp < b->base)
        panic("_allocb");
    b->wp = b->rp;

    return b;
}
```

Uses Hdrspc-780 811, ROUND 47b, false, mallocz() 104d, and msize() 436a.

⟨*function* msize 436a⟩≡ (861b)

```
ulong
msize(void *v)
{
    return poolmsize(mainmem, (ulong*)v-Npadlong)-Npadlong*sizeof(ulong);
}
```

Uses Npadlong-1813 861b and mainmem 102b.

[allocb] is for process context: it panics if memory is exhausted. [iallocb] below is for interrupt context: it returns nil on failure and tracks memory against a quota (`conf.ialloc`) to prevent interrupt handlers from consuming all memory. Blocks from [iallocb] are flagged `BINTR` so [freeb] can decrement the interrupt allocation counter.

⟨*function* allocb 436b⟩≡ (811)

```
Block*
allocb(int size)
{
    Block *b;

    /*
     * Check in a process and wait until successful.
     * Can still error out of here, though.
     */
    if(up == nil)
        panic("allocb without up: %#p", getcallerpc(&size));
    if((b = _allocb(size)) == nil){
        arch_splhi();
        xsummary();
        mallocsummary();
        arch_delay(500);
        panic("allocb: no memory for %d bytes; caller %#p", size,
            getcallerpc(&size));
    }
    setmalloctag(b, getcallerpc(&size));
```

```
    return b;
}
```

Uses _allocb() 435e, arch_delay, mallocsummary() 643b, setmalloctag() 106a, up, and xsummary() 643a.

⟨*struct* Ialloc 437a⟩≡ (811)

```
struct Ialloc
{
    Lock;
    ulong   bytes;
};
```

⟨*global* ialloc 437b⟩≡ (811)

```
struct Ialloc ialloc;
```

Uses Ialloc 437a.

⟨*function* iallocb 437c⟩≡ (811)

```
Block*
iallocb(int size)
{
    Block *b;
    static int m1, m2, mp;

    if(ialloc.bytes > conf.ialloc){
        if((m1++%10000)==0){
            if(mp++ > 1000){
                active.exiting = true;
                arch_exit(0);
            }
            iprint("iallocb: limited %lud/%lud\n",
                ialloc.bytes, conf.ialloc);
        }
        return nil;
    }

    if((b = _allocb(size)) == nil){
        if((m2++%10000)==0){
            if(mp++ > 1000){
                active.exiting = true;
                arch_exit(0);
            }
            iprint("iallocb: no memory %lud/%lud\n",
                ialloc.bytes, conf.ialloc);
        }
        return nil;
    }
    setmalloctag(b, getcallerpc(&size));
    b->flag = BINTR;

    ilock(&ialloc);
    ialloc.bytes += b->lim - b->base;
    iunlock(&ialloc);

    return b;
}
```

Uses BINTR 434c, _allocb() 435e, active 65e, arch_exit, conf 62b, ialloc 437b, ilock() 79d, iprint, iunlock() 80b, setmalloctag() 106a, and true.

[freeb] uses atomic reference counting: `arch_xdec` returns the new count, and only the caller who decrements to zero actually frees. Before freeing, it poisons all pointers with `Bdead` — a debugging aid that makes use-after-free crash immediately with a recognizable address rather than silently corrupting data.

⟨*function* `freeb` 438a⟩≡ (811)

```
void
freeb(Block *b)
{
    void *dead = (void*)Bdead;
    long ref;

    if(b == nil || (ref = arch_xdec(&b->ref)) > 0)
        return;

    if(ref < 0){
        arch_dumpstack();
        panic("freeb: ref %ld; caller pc %#p", ref, getcallerpc(&b));
    }

    /*
     * drivers which perform non cache coherent DMA manage their own buffer
     * pool of uncached buffers and provide their own free routine.
     */
    if(b->free) {
        b->free(b);
        return;
    }
    if(b->flag & BINTR) {
        ilock(&ialloc);
        ialloc.bytes -= b->lim - b->base;
        iunlock(&ialloc);
    }

    /* poison the block in case someone is still holding onto it */
    b->next = dead;
    b->rp = dead;
    b->wp = dead;
    b->lim = dead;
    b->base = dead;

    free(b);
}
```

Uses `BINTR` 434c, `Bdead-781` 811, `arch_dumpstack`, `free()` 104a, `ialloc` 437b, `ilock()` 79d, and `iunlock()` 80b.

⟨*function* `checkb` 438b⟩≡ (811)

```
void
checkb(Block *b, char *msg)
{
    void *dead = (void*)Bdead;

    if(b == dead)
        panic("checkb b %s %#p", msg, b);
    if(b->base == dead || b->lim == dead || b->next == dead
      || b->rp == dead || b->wp == dead){
        print("checkb: base %#p lim %#p next %#p\n",
            b->base, b->lim, b->next);
        print("checkb: rp %#p wp %#p\n", b->rp, b->wp);
        panic("checkb dead: %s", msg);
    }
```

```
    if(b->base > b->lim)
        panic("checkb 0 %s %#p %#p", msg, b->base, b->lim);
    if(b->rp < b->base)
        panic("checkb 1 %s %#p %#p", msg, b->base, b->rp);
    if(b->wp < b->base)
        panic("checkb 2 %s %#p %#p", msg, b->base, b->wp);
    if(b->rp > b->lim)
        panic("checkb 3 %s %#p %#p", msg, b->rp, b->lim);
    if(b->wp > b->lim)
        panic("checkb 4 %s %#p %#p", msg, b->wp, b->lim);
}
```

Uses Bdead-781 811 and print.

⟨*function* freeblist 439a⟩≡ (813c)

```
/*
 *  free a list of blocks
 */
void
freeblist(Block *b)
{
    Block *next;

    for(; b != 0; b = next){
        next = b->next;
        if(b->ref == 1)
            b->next = nil;
        freeb(b);
    }
}
```

Uses freeb() 438a.

[padblock] ensures there is enough space to prepend (positive `size`) or append (negative `size`) data to a block. If the existing header or trailer space is sufficient, it simply adjusts `rp` or does nothing. Otherwise it allocates a new block, copies the data, and frees the old one. This is the function that makes the "prepend headers without copying" design work in practice — when the initial header space runs out, [padblock] transparently reallocates.

⟨*function* padblock 439b⟩≡ (813c)

```
/*
 *  pad a block to the front (or the back if size is negative)
 */
Block*
padblock(Block *bp, int size)
{
    int n;
    Block *nbp;

//    QDEBUG checkb(bp, "padblock 1");
    if(size >= 0){
        if(bp->rp - bp->base >= size){
            bp->rp -= size;
            return bp;
        }

        if(bp->next)
            panic("padblock %#p", getcallerpc(&bp));
        n = BLEN(bp);
        padblockcnt++;
        nbp = allocb(size+n);
        nbp->rp += size;
```

```
            nbp->wp = nbp->rp;
            memmove(nbp->wp, bp->rp, n);
            nbp->wp += n;
            freeb(bp);
            nbp->rp -= size;
        } else {
            size = -size;

            if(bp->next)
                panic("padblock %#p", getcallerpc(&bp));

            if(bp->lim - bp->wp >= size)
                return bp;

            n = BLEN(bp);
            padblockcnt++;
            nbp = allocb(size+n);
            memmove(nbp->wp, bp->rp, n);
            nbp->wp += n;
            freeb(bp);
        }
    //    QDEBUG checkb(nbp, "padblock 1");
        return nbp;
    }
```

Uses BLEN 435b, allocb() 436b, freeb() 438a, and padblockcnt-770 812d.

⟨*function* blocklen 440a⟩≡ (813c)

```
    /*
     *  return count of bytes in a string of blocks
     */
    int
    blocklen(Block *bp)
    {
        int len;

        len = 0;
        while(bp) {
            len += BLEN(bp);
            bp = bp->next;
        }
        return len;
    }
```

Uses BLEN 435b.

⟨*function* blockalloclen 440b⟩≡ (813c)

```
    /*
     * return count of space in blocks
     */
    int
    blockalloclen(Block *bp)
    {
        int len;

        len = 0;
        while(bp) {
            len += BALLOC(bp);
            bp = bp->next;
        }
        return len;
    }
```

Uses BALLOC 435c.

⟨*function* concatblock 441a⟩≡ (813c)
```
/*
 *  copy the  string of blocks into
 *  a single block and free the string
 */
Block*
concatblock(Block *bp)
{
    int len;
    Block *nb, *f;

    if(bp->next == 0)
        return bp;

    nb = allocb(blocklen(bp));
    for(f = bp; f; f = f->next) {
        len = BLEN(f);
        memmove(nb->wp, f->rp, len);
        nb->wp += len;
    }
    concatblockcnt += BLEN(nb);
    freeblist(bp);
//    QDEBUG checkb(nb, "concatblock 1");
    return nb;
}
```
Uses BLEN 435b, allocb() 436b, blocklen() 440a, concatblockcnt-771 812d, and freeblist() 439a.

[pullupblock] consolidates the first n bytes of a block chain into a single contiguous block. Protocol processing often needs to examine a fixed-size header (e.g., an IP header) without it being split across blocks.

⟨*function* pullupblock 441b⟩≡ (813c)
```
/*
 *  make sure the first block has at least n bytes
 */
Block*
pullupblock(Block *bp, int n)
{
    int i;
    Block *nbp;

    /*
     *  this should almost always be true, it's
     *  just to avoid every caller checking.
     */
    if(BLEN(bp) >= n)
        return bp;

    /*
     *  if not enough room in the first block,
     *  add another to the front of the list.
     */
    if(bp->lim - bp->rp < n){
        nbp = allocb(n);
        nbp->next = bp;
        bp = nbp;
    }

    /*
```

```
     *  copy bytes from the trailing blocks into the first
     */
    n -= BLEN(bp);
    while(nbp = bp->next){
        i = BLEN(nbp);
        if(i > n) {
            memmove(bp->wp, nbp->rp, n);
            pullupblockcnt++;
            bp->wp += n;
            nbp->rp += n;
//            QDEBUG checkb(bp, "pullupblock 1");
            return bp;
        } else {
            /* shouldn't happen but why crash if it does */
            if(i < 0){
                print("pullup negative length packet, called from %#p\n",
                    getcallerpc(&bp));
                i = 0;
            }
            memmove(bp->wp, nbp->rp, i);
            pullupblockcnt++;
            bp->wp += i;
            bp->next = nbp->next;
            nbp->next = 0;
            freeb(nbp);
            n -= i;
            if(n == 0){
//                QDEBUG checkb(bp, "pullupblock 2");
                return bp;
            }
        }
    }
    freeb(bp);
    return nil;
}
```

Uses BLEN 435b, allocb() 436b, freeb() 438a, print, and pullupblockcnt-772 812d.

⟨*function* packblock 442⟩≡ (813c)

```
/*
 *  if the allocated space is way out of line with the used
 *  space, reallocate to a smaller block
 */
Block*
packblock(Block *bp)
{
    Block **l, *nbp;
    int n;

    for(l = &bp; *l; l = &(*l)->next){
        nbp = *l;
        n = BLEN(nbp);
        if((n<<2) < BALLOC(nbp)){
            *l = allocb(n);
            memmove((*l)->wp, nbp->rp, n);
            (*l)->wp += n;
            (*l)->next = nbp->next;
            freeb(nbp);
        }
    }
```

```
    return bp;
}
```
Uses BALLOC 435c, BLEN 435b, allocb() 436b, and freeb() 438a.

⟨*function* trimblock 443a⟩≡ (813c)
```
/*
 *  trim to len bytes starting at offset
 */
Block *
trimblock(Block *bp, int offset, int len)
{
    ulong l;
    Block *nb, *startb;

//    QDEBUG checkb(bp, "trimblock 1");
    if(blocklen(bp) < offset+len) {
        freeblist(bp);
        return nil;
    }

    while((l = BLEN(bp)) < offset) {
        offset -= l;
        nb = bp->next;
        bp->next = nil;
        freeb(bp);
        bp = nb;
    }

    startb = bp;
    bp->rp += offset;

    while((l = BLEN(bp)) < len) {
        len -= l;
        bp = bp->next;
    }

    bp->wp -= (BLEN(bp) - len);

    if(bp->next) {
        freeblist(bp->next);
        bp->next = nil;
    }

    return startb;
}
```
Uses BLEN 435b, blocklen() 440a, freeb() 438a, and freeblist() 439a.

⟨*function* copyblock 443b⟩≡ (813c)
```
/*
 *  copy 'count' bytes into a new block
 */
Block*
copyblock(Block *bp, int count)
{
    int l;
    Block *nbp;

//    QDEBUG checkb(bp, "copyblock 0");
    nbp = allocb(count);
    for(; count > 0 && bp != 0; bp = bp->next){
```

```
        l = BLEN(bp);
        if(l > count)
            l = count;
        memmove(nbp->wp, bp->rp, l);
        nbp->wp += l;
        count -= l;
    }
    if(count > 0){
        memset(nbp->wp, 0, count);
        nbp->wp += count;
    }
    copyblockcnt++;
//    QDEBUG checkb(nbp, "copyblock 1");

    return nbp;
}
```

Uses BLEN 435b, allocb() 436b, and copyblockcnt-773 812d.

⟨*function* adjustblock 444a⟩≡ (813c)

```
Block*
adjustblock(Block* bp, int len)
{
    int n;
    Block *nbp;

    if(len < 0){
        freeb(bp);
        return nil;
    }

    if(bp->rp+len > bp->lim){
        nbp = copyblock(bp, len);
        freeblist(bp);
//        QDEBUG checkb(nbp, "adjustblock 1");

        return nbp;
    }

    n = BLEN(bp);
    if(len > n)
        memset(bp->wp, 0, len-n);
    bp->wp = bp->rp+len;
//    QDEBUG checkb(bp, "adjustblock 2");

    return bp;
}
```

Uses BLEN 435b, copyblock() 443b, freeb() 438a, and freeblist() 439a.

⟨*function* pullblock 444b⟩≡ (813c)

```
/*
 *  throw away up to count bytes from a
 *  list of blocks.  Return count of bytes
 *  thrown away.
 */
int
pullblock(Block **bph, int count)
{
    Block *bp;
    int n, bytes;
```

```
    bytes = 0;
    if(bph == nil)
        return 0;

    while(*bph != nil && count != 0) {
        bp = *bph;
        n = BLEN(bp);
        if(count < n)
            n = count;
        bytes += n;
        count -= n;
        bp->rp += n;
//        QDEBUG checkb(bp, "pullblock ");
        if(BLEN(bp) == 0) {
            *bph = bp->next;
            bp->next = nil;
            freeb(bp);
        }
    }
    return bytes;
}
```

Uses BLEN 435b and freeb() 438a.

⟨*function* bl2mem 445⟩≡ (813c)

```
/*
 *  copy the contents of a string of blocks into
 *  memory.  emptied blocks are freed.  return
 *  pointer to first unconsumed block.
 */
Block*
bl2mem(byte *p, Block *b, int n)
{
    int i;
    Block *next;

    for(; b != nil; b = next){
        i = BLEN(b);
        if(i > n){
            memmove(p, b->rp, n);
            b->rp += n;
            return b;
        }
        memmove(p, b->rp, i);
        n -= i;
        p += i;
        b->rp += i;
        next = b->next;
        freeb(b);
    }
    return nil;
}
```

Uses BLEN 435b and freeb() 438a.

17.6 Fileservers

The chapters that follow (IPC, Special Fileservers) cover the more complex in-kernel file servers—**devpipe**, **devsrv**, **devmnt**—that layer on top of the **Dev** interface to provide rich inter-process communication. The **Fcall**

structure (covered in LIBCORE book [Pad16c]) is the 9P message format used by `devmnt` to communicate with external file servers.

17.7 Fcall

17.8 XTODO

⟨Qidtype *cases* 446⟩+≡ (70b) ◁427b

```
  QTAUTH = 0x08,    /* type bit for authentication file */
  QTAPPEND = 0x40,    /* type bit for append only files */
  QTEXCL = 0x20,    /* type bit for exclusive use files */
```

Part III

Beyond the Basics

The previous two parts covered the kernel's core: process virtualization and the file abstraction. This part covers the remaining subsystems that build on top of that foundation: inter-process communication (pipes, notes, shared memory), the debugging and profiling infrastructure, special file servers, and advanced topics that crosscut multiple subsystems.

Chapter 18

IPC

Plan 9 provides several IPC mechanisms beyond the filesystem (which is itself an IPC mechanism via `read`/`write` on shared channels). This chapter covers notes (the Plan 9 equivalent of Unix signals, but string-based), pipes (bidirectional, unlike Unix), shared memory segments, user-level semaphores with kernel backing, and rendezvous (a lightweight synchronization primitive for exchanging values between processes).

⟨`systab` *ipc syscalls* 448a⟩≡ (57b)
```
[NOTIFY]    sysnotify,
[NOTED]     sysnoted,

[PIPE]      syspipe,

[SEGATTACH] syssegattach,
[SEGDETACH] syssegdetach,
[SEGFREE]   syssegfree,
[SEGFLUSH]  syssegflush,
[SEGBRK]    syssegbrk,
```
Uses `sysnoted()` 453a, `syspipe()` 457, `syssegattach()` 468a, `syssegbrk()` 467c, `syssegdetach()` 470, `syssegflush()` 472c, and `syssegfree()` 471a.

The C prototypes are:

```
int   notify(void (*f)(void *ureg, char *note));
int   noted(int v);           // NCONT, NDFLT, NSAVE, NRSTR
int   pipe(int fd[2]);        // bidirectional
void* segattach(int attr, char *class, void *va, ulong len);
int   segdetach(void *addr);
```

⟨`systab` *concurrency syscalls* 448b⟩≡ (57b)
```
[RENDEZVOUS]    sysrendezvous,

[SEMACQUIRE]    syssemacquire,
[SEMRELEASE]    syssemrelease,
[TSEMACQUIRE]   systsemacquire,
```
Uses `syssemacquire()` 477b, `syssemrelease()` 478b, and `systsemacquire()` 478a.

The C prototypes are:

```
void* rendezvous(void *tag, void *value);  // exchange values
int   semacquire(long *addr, int block);   // decrement or sleep
long  semrelease(long *addr, long delta);  // increment and wake
int   tsemacquire(long *addr, ulong ms);   // with timeout
```

18.1 Notes (signals)

Notes are Plan 9's replacement for Unix signals. Instead of numbered signals (`SIGSEGV`, `SIGPIPE`), notes are arbitrary strings like `"sys: trap: fault read addr=0x0"`. A process registers a handler via `notify(handler)`; when a note arrives, the kernel builds a "note frame" on the user stack containing saved registers and the note string, then redirects execution to the handler. The handler calls `noted(NCONT)` to resume or `noted(NDFLT)` to terminate.

⟨*enum* `notekind` 449a⟩≡ (868c)
```
enum NoteKind
{
    NUser,        /* note provided externally */
    NExit,        /* deliver note quietly */
    NDebug,       /* print debug message */
};
```

⟨*enum* `note` 449b⟩≡ (853i)
```
enum NoteHook {
  NCONT = 0, /* continue after note */
  NDFLT = 1, /* terminate after note */
  NSAVE = 2, /* clear note but hold state */
  NRSTR = 3, /* restore saved state */
};
```

⟨*global* `noteidalloc` 449c⟩≡ (877c)
```
// also used by sysrfork()
Counter noteidalloc;
```

Each process has a circular buffer of up to `NNOTE` (5) pending notes. The `flag` field distinguishes user-posted notes (`NUser`) from system notes (`NExit`, `NDebug`). System notes with no handler cause immediate termination; user notes without a handler also terminate, but a handler can intercept them.

⟨*struct* `Note` 449d⟩≡ (868c)
```
// a kind of Unix signal
struct Note
{
    char  msg[ERRMAX];
    // enum<NoteKind>
    int flag;     /* whether system posted it */
};
```

⟨`newproc()` *set fields of* `p` 449e⟩+≡ (130a) ◁239d 449f▷
```
p->noteid = incref(&noteidalloc);
if(p->noteid==0)
    panic("pidalloc");
```
Uses `incref()` 87b and `noteidalloc` 449c.

⟨`newproc()` *set fields of* `p` 449f⟩+≡ (130a) ◁449e 450c▷
```
p->notepending = false;
p->notified = false;
```
Uses `false`.

⟨`sysrfork()` *when* `RFPROC` *propagate fields from up to p* 449g⟩+≡ (134b) ◁226c 450d▷
```
memmove(p->note, up->note, sizeof(p->note));
p->nnote = up->nnote;
p->lastnote = up->lastnote;
p->notify = up->notify;
```
Uses `up`.

⟨sleep() *if notepending* 450a⟩≡ (233e)
```
if(up->notepending) {
    up->notepending = false;
    arch_splx(s);
    ⟨sleep() forceclosefgrp 319a⟩
    error(Eintr);
}
```
Uses false and up.

The note machinery in Proc tracks several states: notify is the user handler, notified prevents recursive delivery, notepending wakes sleeping processes, and ureg saves registers for noted().

⟨Proc *notes fields* 450b⟩≡ (66)
```
Note  note[NNOTE];
short nnote;

int (*notify)(void*, char*);
bool_ushort notified; /* sysnoted is due */

ulong noteid;   /* Equivalent of note group */

bool notepending;  /* note issued but not acted on */

Note  lastnote;

void  *ureg;    /* User registers for notes */
```

⟨newproc() *set fields of* p 450c⟩+≡ (130a) ◁449f 513c▷
```
p->ureg = nil;
```

⟨sysrfork() *when* RFPROC *propagate fields from up to p* 450d⟩+≡ (134b) ◁449g 487e▷
```
p->ureg = up->ureg; // will be adjusted for R0??
```
Uses up.

⟨sysexec() *when hold debug lock, reset some fields* 450e⟩≡ (141c) 601h▷
```
up->nnote = 0;
up->notify = nil;
up->notified = false;
```

⟨*constant* NNOTE 450f⟩≡ (868c)
```
NNOTE = 5,
```

⟨trap() *call possibly notify(arm)* 450g⟩≡ (168b)
```
if(up->procctl || up->nnote)
    arch__notify(ureg);
```
Uses arch__notify() 452 and up.

18.1.1 sysnotify() (ARM)

⟨*syscall notify* 450h⟩≡ (881b)
```
// int notify(void (*f)(void*, char*));
long
sysnotify(ulong* arg)
{
    if(arg[0] != 0)
        validaddr(arg[0], sizeof(ulong), false);
    up->notify = (int(*)(void*, char*))(arg[0]);
    return 0;
}
```
Uses false, up, and validaddr() 120b.

⟨syscall() *call notify(arm)* 451a⟩≡ (163e)

```
arch_splhi();
if(scallnr != RFORK && (up->procctl || up->nnote))
    arch__notify(ureg);
```

Uses arch__notify() 452 and up.

The NFrame is the activation record that arch__notify builds on the user stack. It contains a copy of the Ureg (so the handler can inspect and modify registers), the note message string, and a pointer to the previous NFrame (old) for nested notes. The ip field is set to zero, acting as a return address that will cause a fault if the handler tries to return normally instead of calling noted().

Here is the user stack before and after arch__notify delivers a note. The trick is that the kernel returns to user mode with PC reset to the notify handler and SP pointing into a synthesized frame that looks like a normal C call frame:

```
Before arch__notify (interrupted user code):

  high  |     ...existing user stack...    |
        +----------------------------------+
  sp -> | (user's current stack frame)     |
        +----------------------------------+
  low   |     ...available stack...        |

After arch__notify (about to return to user):

  high  |     ...existing user stack...    |
        +----------------------------------+
        |  ureg: copy of interrupted regs  |  <-+
        |        (pc,sp,r0..r14,cpsr)      |    |
        +----------------------------------+    |
        |  old: previous up->ureg          |    |
        +----------------------------------+    |
        |  msg: "sys: trap: ... pc=0x1234" |    |
        +----------------------------------+    |
        |  arg1: &msg[0]  -----------------|----|----> used as 2nd arg
        +----------------------------------+    |
        |  arg0: &ureg    -----------------|----+    used as 1st arg
        +----------------------------------+
  sp -> |  ip = 0  (fault if handler RETs) |
        +----------------------------------+
        |     ...available stack...        |
  low
```

After the kernel restores user mode, execution resumes at up->notify with sp pointing at the fake return address (the zeroed ip) and r0 set to the Ureg*. From the handler's perspective it has been called as notify(ureg, msg), following the normal C calling convention. The handler must explicitly call noted(NCONT) or noted(NDFLT)—if it tries to return like a normal C function, it will jump to address 0 and crash. That is a deliberate safety check: the kernel needs to know how the handler wants to resume, and "resume as if nothing happened" is not a valid option. noted() finds the NFrame via up->ureg (which was just overwritten to point at nf), copies the saved Ureg back into the kernel's own trap frame, restores the previous up->ureg from nf->old, and returns—dropping through the usual trap-return path and resuming the interrupted code at its original pc.

⟨*struct* NFrame*(arm)* 451b⟩≡ (1012d)

```
struct NFrame {
    uintptr ip;
    Ureg*   arg0;
    char*   arg1;
    char    msg[ERRMAX];
    Ureg*   old;
    Ureg    ureg;
};
```

Uses ERRMAX.

`arch__notify` is the heart of note delivery. It builds an `NFrame` on the user stack containing the saved `Ureg`, the note message, and the handler address, then overwrites the trap frame's PC and SP so that returning to user mode jumps to the handler. The handler receives `(ureg*, msg)` as arguments and calls `noted()` to return control. If the process is already notified, has no handler, or the handler address is invalid, the process is killed.

⟨*function* `arch__notify` *(arm)* 452⟩≡ (1012d)

```
/*
 *  Call user, if necessary, with note.
 *  Pass user the Ureg struct and the note on his stack.
 */
int
arch__notify(Ureg* ureg)
{
    int l;
    Note *n;
    u32int s;
    uintptr sp;
    NFrame *nf;

    if(up->procctl)
        procctl(up);
    if(up->nnote == 0)
        return 0;

    fpunotify(ureg);

    s = arch_spllo();
    qlock(&up->debug);

    up->notepending = 0;
    n = &up->note[0];
    if(strncmp(n->msg, "sys:", 4) == 0){
        l = strlen(n->msg);
        if(l > ERRMAX-23)   /* " pc=0x0123456789abcdef\0" */
            l = ERRMAX-23;
        snprint(n->msg + l, sizeof n->msg - l, " pc=%#lux", ureg->pc);
    }

    if(n->flag != NUser && (up->notified || up->notify == 0)){
        if(n->flag == NDebug)
            pprint("suicide: %s\n", n->msg);
        qunlock(&up->debug);
        pexit(n->msg, n->flag != NDebug);
    }

    if(up->notified){
        qunlock(&up->debug);
        arch_splhi();
        return 0;
    }
```

```
    if(up->notify == nil){
        qunlock(&up->debug);
        pexit(n->msg, n->flag != NDebug);
    }
    if(!okaddr(PTR2UINT(up->notify), 1, 0)){
        pprint("suicide: notify function address %#p\n", up->notify);
        qunlock(&up->debug);
        pexit("Suicide", 0);
    }

    sp = ureg->sp - sizeof(NFrame);
    if(!okaddr(sp, sizeof(NFrame), 1)){
        qunlock(&up->debug);
        pprint("suicide: notify stack address %#p\n", sp);
        pexit("Suicide", 0);
    }

    nf = UINT2PTR(sp);
    memmove(&nf->ureg, ureg, sizeof(Ureg));
    nf->old = up->ureg;
    up->ureg = nf;
    memmove(nf->msg, up->note[0].msg, ERRMAX);
    nf->arg1 = nf->msg;
    nf->arg0 = &nf->ureg;
    ureg->r0 = PTR2UINT(nf->arg0);
    nf->ip = 0;

    ureg->sp = sp;
    ureg->pc = PTR2UINT(up->notify);
    up->notified = 1;
    up->nnote--;
    memmove(&up->lastnote, &up->note[0], sizeof(Note));
    memmove(&up->note[0], &up->note[1], up->nnote*sizeof(Note));

    qunlock(&up->debug);
    arch_splx(s);

    return 1;
}
```

Uses ERRMAX, NDebug 449a, NUser 449a, PTR2UINT 56g, UINT2PTR 56h, fpunotify() 517c, okaddr() 119c, pexit, pprint, procctl() 485b, qlock() 81f, qunlock() 82c, and up.

18.1.2 sysnoted() (ARM)

⟨*syscall noted* 453a⟩≡ (881b)

```
// int noted(int v);
long
sysnoted(ulong* arg)
{
    if(arg[0]!=NRSTR && !up->notified)
        error(Egreg);
    return 0;
}
```

Uses Egreg 664d, NRSTR 449b, and up.

⟨syscall() *call noted()* 453b⟩≡ (163e)

```
if(scallnr == NOTED)
    arch__noted(ureg, *(ulong*)(sp+BY2WD));
```

`arch__noted` restores process state after the note handler finishes. `NCONT`/`NRSTR` restore saved registers and pop the note frame. `NSAVE` re-enters the handler (for coroutine-like handling). `NDFLT` terminates the process. The PSR is masked to prevent user code from escalating to kernel mode by modifying the saved status register.

⟨*function* `arch__noted`*(arm)* 454⟩≡ (1012d)

```
/*
 *   Return user to state before notify()
 */
static void
arch__noted(Ureg* cur, uintptr arg0)
{
    NFrame *nf;
    Ureg *nur;

    qlock(&up->debug);
    if(arg0 != NRSTR && !up->notified){
        qunlock(&up->debug);
        pprint("call to noted() when not notified\n");
        pexit("Suicide", 0);
    }
    up->notified = 0;
    fpunoted();

    nf = up->ureg;

    /* sanity clause */
    if(!okaddr(PTR2UINT(nf), sizeof(NFrame), 0)){
        qunlock(&up->debug);
        pprint("bad ureg in noted %#p\n", nf);
        pexit("Suicide", 0);
    }

    /* don't let user change system flags */
    nur = &nf->ureg;
    nur->psr &= PsrMask|PsrDfiq|PsrDirq;
    nur->psr |= (cur->psr & ~(PsrMask|PsrDfiq|PsrDirq));

    memmove(cur, nur, sizeof(Ureg));

    switch((int)arg0){
    case NCONT:
    case NRSTR:
        if(!okaddr(nur->pc, BY2WD, 0) || !okaddr(nur->sp, BY2WD, 0)){
            qunlock(&up->debug);
            pprint("suicide: trap in noted\n");
            pexit("Suicide", 0);
        }
        up->ureg = nf->old;
        qunlock(&up->debug);
        break;
    case NSAVE:
        if(!okaddr(nur->pc, BY2WD, 0) || !okaddr(nur->sp, BY2WD, 0)){
            qunlock(&up->debug);
            pprint("suicide: trap in noted\n");
            pexit("Suicide", 0);
        }
        qunlock(&up->debug);

        arch_splhi();
        nf->arg1 = nf->msg;
```

```
        nf->arg0 = &nf->ureg;
        nf->ip = 0;
        cur->sp = PTR2UINT(nf);
        cur->r0 = PTR2UINT(nf->arg0);
        break;
    default:
        pprint("unknown noted arg %#p\n", arg0);
        up->lastnote.flag = NDebug;
        /*FALLTHROUGH*/
    case NDFLT:
        if(up->lastnote.flag == NDebug){
            qunlock(&up->debug);
            pprint("suicide: %s\n", up->lastnote.msg);
        }
        else
            qunlock(&up->debug);
        pexit(up->lastnote.msg, up->lastnote.flag != NDebug);
    }
}
```

Uses NCONT 853i, NDFLT, NDebug 449a, NRSTR 449b, NSAVE 449b, PTR2UINT 56g, PsrDfiq 560d, PsrDirq 39a, PsrMask 38, fpunoted() 518a, okaddr() 119c, pexit, pprint, qlock() 81f, qunlock() 82c, and up.

18.1.3 postnote()

[postnote] is one of the trickiest functions in the kernel due to lock ordering. To wake a sleeping process, it needs both `p->rlock` and the `Rendez` lock, but [sleep] acquires them in the opposite order. The solution is a retry loop: acquire `p->rlock`, try `canlock(r)`, and if that fails, drop `p->rlock`, call `sched()`, and retry. The second half handles rendezvous: if the target is blocked in [sysrendezvous], it gets pulled out of the hash table and made ready.

⟨*function* postnote 455⟩≡ (877c)

```
/*
 *  if waking a sleeping process, this routine must hold both
 *  p->rlock and r->lock.  However, it can't know them in
 *  the same order as wakeup causing a possible lock ordering
 *  deadlock.  We break the deadlock by giving up the p->rlock
 *  lock if we can't get the r->lock and retrying.
 */
int
proc_postnote(Proc *p, int dolock, char *n, int flag)
{
    int s, ret;
    Rendez *r;
    Proc *d, **l;

    if(dolock)
        qlock(&p->debug);

    if(flag != NUser && (p->notify == 0 || p->notified))
        p->nnote = 0;

    ret = 0;
    if(p->nnote < NNOTE) {
        strcpy(p->note[p->nnote].msg, n);
        p->note[p->nnote++].flag = flag;
        ret = 1;
    }
    p->notepending = true;
```

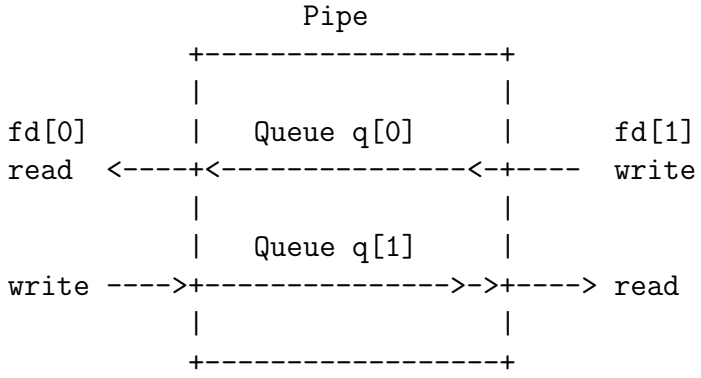
```
    if(dolock)
        qunlock(&p->debug);

    /* this loop is to avoid lock ordering problems. */
    for(;;){
        s = arch_splhi();
        lock(&p->rlock);
        r = p->r;

        /* waiting for a wakeup? */
        if(r == nil)
            break;  /* no */

        /* try for the second lock */
        if(canlock(r)){
            if(p->state != Wakeme || r->p != p)
                panic("postnote: state %d %d %d", r->p != p, p->r != r, p->state);
            p->r = nil;
            r->p = nil;
            ready(p);
            unlock(r);
            break;
        }

        /* give other process time to get out of critical section and try again */
        unlock(&p->rlock);
        arch_splx(s);
        sched();
    }
    unlock(&p->rlock);
    arch_splx(s);

    if(p->state != Rendezvous)
        return ret;

    /* Try and pull out of a rendezvous */
    lock(p->rgrp);
    if(p->state == Rendezvous) {
        p->rendval = ~0;
        l = &REND(p->rgrp, p->rendtag);
        for(d = *l; d; d = d->rendhash) {
            if(d == p) {
                *l = p->rendhash;
                break;
            }
            l = &d->rendhash;
        }
        ready(p);
    }
    unlock(p->rgrp);
    return ret;
}
```

Uses NNOTE 450f, NUser 449a, REND 479a, Rendezvous 480a, Wakeme 233d, canlock() 78f, lock() 76d, qlock() 81f, qunlock() 82c, ready, sched, true, and unlock() 78d.

18.2 Pipes

18.2.1 syspipe()

Plan 9 pipes are bidirectional: both ends can read and write. A pipe is a pair of Queues: writing to fd[0] enqueues on q[1] (which fd[1] reads from), and vice versa. A single pipe suffices for two-way communication, whereas Unix requires two. The cross-wiring is what makes one pipe act like two Unix pipes. Each endpoint's "read" and "write" paths hit different queues, so a byte written at one end always comes out at the other:



The qref[2] array counts how many open channels reference each side; when a side's refcount drops to zero, the opposite queue is hung up (sending EOF to the peer). Closing fd[0] thus delivers EOF to whoever is still reading at fd[1]—the same contract as a Unix pipe, just achieved symmetrically. [syspipe] creates a pipe by attaching to the pipe device (#|), cloning the channel, then walking each clone to data and data1.

⟨*syscall pipe* 457⟩≡ (815)

```
// int pipe(int fd[2]);
long
syspipe(ulong* arg)
{
    int fd[2];
    Chan *c[2];
    Dev *d;
    static char *datastr[] = {"data", "data1"};

    validaddr(arg[0], 2*BY2WD, true);
    arch_validalign(arg[0], sizeof(int));
    d = devtab[devno('|', false)];
    c[0] = namec("#|", Atodir, 0, 0);
    c[1] = nil;
    fd[0] = -1;
    fd[1] = -1;

    if(waserror()){
        cclose(c[0]);
        if(c[1])
            cclose(c[1]);
        nexterror();
    }
    c[1] = cclone(c[0]);
    if(walk(&c[0], datastr+0, 1, 1, nil) < 0)
        error(Egreg);
    if(walk(&c[1], datastr+1, 1, 1, nil) < 0)
        error(Egreg);
    c[0] = d->open(c[0], ORDWR);
    c[1] = d->open(c[1], ORDWR);
    if(newfd2(fd, c) < 0)
```

```
        error(Enofd);
    poperror();

    ((long*)arg[0])[0] = fd[0];
    ((long*)arg[0])[1] = fd[1];
    return 0;
}
```

Uses Atodir 305b, Egreg 664d, Enofd 664d, ORDWR 301c, cclone() 332a, cclose, devno() 297c, false, namec() 305f, newfd2() 458a, nexterror, poperror 666d, true, validaddr() 120b, walk() 328f, and waserror 666d.

⟨*function* newfd2 458a⟩≡ (815)

```
int
newfd2(int fd[2], Chan *c[2])
{
    Fgrp *f;

    f = up->fgrp;
    lock(f);
    fd[0] = findfreefd(f, 0);
    if(fd[0] < 0){
        unlockfgrp(f);
        return -1;
    }
    fd[1] = findfreefd(f, fd[0]+1);
    if(fd[1] < 0){
        unlockfgrp(f);
        return -1;
    }
    if(fd[1] > f->maxfd)
        f->maxfd = fd[1];
    f->fd[fd[0]] = c[0];
    f->fd[fd[1]] = c[1];
    unlockfgrp(f);

    return 0;
}
```

Uses findfreefd() 300b, lock() 76d, unlockfgrp() 301b, and up.

⟨sseek() *and pipes* 458b⟩≡ (316)

```
if(devtab[c->type]->dc == '|')
    error(Eisstream);
```

Uses Eisstream 664d.

18.2.2 /dev/pipe

The Pipe structure holds two Queues and reference counts. The qref array tracks how many open channels reference each side; when a side's reference count drops to zero, the opposite queue is hung up (sending EOF to readers) and the local queue is closed. This cross-wiring is key to bidirectional pipes: closing one end signals the other.

⟨*struct* Pipe 458c⟩≡ (739b)

```
struct Pipe
{
    int ref;
    ulong   path;
    long    perm;
    Queue   *q[2];
    int qref[2];
```

```
    // extra
    QLock;
    Pipe    *next;
};
```

⟨*struct* `Pipealloc` 459a⟩≡ (739b)
```
struct Pipealloc
{
    ulong   path;

    // extra
    Lock;
};
```

⟨*global* `pipealloc` 459b⟩≡ (739b)
```
struct Pipealloc pipealloc;
```
Uses `Pipealloc` 459a.

⟨*devpipe.c enum Qxxx* 459c⟩≡ (739b)
```
enum
{
    Qdir,
    Qdata0,
    Qdata1,
};
```

⟨*global* `pipedir` 459d⟩≡ (739b)
```
Dirtab pipedir[] =
{
    ".",        {Qdir,0,QTDIR}, 0,      DMDIR|0500,
    "data",     {Qdata0},   0,      0600,
    "data1",    {Qdata1},   0,      0600,
};
```
Uses `DMDIR`, `QTDIR` 70b, `Qdata0-856` 459c, `Qdata1-857` 459c, and `Qdir-855` 459c.

⟨`Conf` *other fields* 459e⟩+≡ (62a) ◁276b
```
ulong pipeqsize;  /* size in bytes of pipe queues */
```

⟨*method pipeinit* 459f⟩≡ (739b)
```
static void
pipeinit(void)
{
    if(conf.pipeqsize == 0){
        if(conf.ncpu > 1)
            conf.pipeqsize = 256*1024;
        else
            conf.pipeqsize = 32*1024;
    }
}
```
Uses `conf` 62b.

⟨`Chan` *union other fields* 459g⟩≡ (68b) 604c▷
```
/*Pipe*/void* chanpipe; // for pipes
```

⟨*method pipeattach* 460a⟩≡ (739b)

```
/*
 *  create a pipe, no streams are created until an open
 */
static Chan*
pipeattach(char *spec)
{
    Pipe *p;
    Chan *c;

    c = devattach('|', spec);
    p = malloc(sizeof(Pipe));
    if(p == 0)
        exhausted("memory");
    p->ref = 1;

    p->q[0] = qopen(conf.pipeqsize, 0, 0, 0);
    if(p->q[0] == 0){
        free(p);
        exhausted("memory");
    }
    p->q[1] = qopen(conf.pipeqsize, 0, 0, 0);
    if(p->q[1] == 0){
        free(p->q[0]);
        free(p);
        exhausted("memory");
    }

    lock(&pipealloc);
    p->path = ++pipealloc.path;
    unlock(&pipealloc);
    p->perm = pipedir[Qdata0].perm;

    mkqid(&c->qid, NETQID(2*p->path, Qdir), 0, QTDIR);
    c->aux = p;
    c->dev = 0;
    return c;
}
```

Uses `NETQID`, `QTDIR` 70b, `Qdata0-856` 459c, `Qdir-855` 459c, `conf` 62b, `devattach()` 297b, `exhausted()` 666c, `free()` 104a, `lock()` 76d, `malloc()` 103c, `pipealloc` 459b, `pipedir` 459d, `qopen()` 409a, and `unlock()` 78d.

[pipegen] is the custom directory generator for the pipe device. Unlike most devices that report static sizes from `Dirtab`, pipes report the current queue length for `data` and `data1` — so `stat` reflects how many bytes are buffered and waiting to be read.

⟨*function* `pipegen` 460b⟩≡ (739b)

```
static int
pipegen(Chan *c, char*, Dirtab *tab, int ntab, int i, DirEntry *dp)
{
    Qid q;
    int len;
    Pipe *p;

    if(i == DEVDOTDOT){
        devdir(c, c->qid, "#|", 0, eve, DMDIR|0555, dp);
        return 1;
    }
    i++;    /* skip . */
    if(tab==0 || i>=ntab)
        return -1;
```

```
    tab += i;
    p = c->aux;
    switch((ulong)tab->qid.path){
    case Qdata0:
        len = qlen(p->q[0]);
        break;
    case Qdata1:
        len = qlen(p->q[1]);
        break;
    default:
        len = tab->length;
        break;
    }
    mkqid(&q, NETQID(NETID(c->qid.path), tab->qid.path), 0, QTFILE);
    devdir(c, q, tab->name, len, eve, p->perm, dp);
    return 1;
}
```

Uses DEVDOTDOT 805, DMDIR, NETID, NETQID, QTFILE, Qdata0-856 459c, Qdata1-857 459c, devdir() 427c, eve 591a, and qlen() 419c.

[pipewalk] delegates to `devwalk` but must bump the pipe's reference count when the walk clones the channel. If the original channel was already open, the per-side `qref` count is also incremented — this matters because [pipeclose] uses `qref` to decide when to hang up the opposite end.

⟨*method pipewalk* 461a⟩≡ (739b)

```
static Walkqid*
pipewalk(Chan *c, Chan *nc, char **name, int nname)
{
    Walkqid *wq;
    Pipe *p;

    wq = devwalk(c, nc, name, nname, pipedir, NPIPEDIR, pipegen);
    if(wq != nil && wq->clone != nil && wq->clone != c){
        p = c->aux;
        qlock(p);
        p->ref++;
        if(c->flag & COPEN){
            print("channel open in pipewalk\n");
            switch(NETTYPE(c->qid.path)){
            case Qdata0:
                p->qref[0]++;
                break;
            case Qdata1:
                p->qref[1]++;
                break;
            }
        }
        qunlock(p);
    }
    return wq;
}
```

Uses COPEN 294b, NETTYPE, NPIPEDIR-858 739b, Qdata0-856 459c, Qdata1-857 459c, devwalk() 429b, pipedir 459d, pipegen() 460b, print, qlock() 81f, and qunlock() 82c.

⟨*method pipestat* 461b⟩≡ (739b)

```
static int
pipestat(Chan *c, byte *db, int n)
{
    Pipe *p;
```

```
    DirEntry dir;

    p = c->aux;

    switch(NETTYPE(c->qid.path)){
    case Qdir:
        devdir(c, c->qid, ".", 0, eve, DMDIR|0555, &dir);
        break;
    case Qdata0:
        devdir(c, c->qid, "data", qlen(p->q[0]), eve, p->perm, &dir);
        break;
    case Qdata1:
        devdir(c, c->qid, "data1", qlen(p->q[1]), eve, p->perm, &dir);
        break;
    default:
        panic("pipestat");
    }
    n = convD2M(&dir, db, n);
    if(n < BIT16SZ)
        error(Eshortstat);
    return n;
}
```

Uses DMDIR, Eshortstat 664d, NETTYPE, Qdata0-856 459c, Qdata1-857 459c, Qdir-855 459c, devdir() 427c, eve 591a, and qlen() 419c.

[pipewstat] only allows the host owner (eve) to change the pipe's permissions. Ownership changes are rejected — pipe endpoints have no meaningful owner identity.

⟨*method pipewstat* 462⟩≡ (739b)

```
static int
pipewstat(Chan* c, byte* db, int n)
{
    int m;
    DirEntry *dir;
    Pipe *p;

    p = c->aux;
    if(strcmp(up->user, eve) != 0)
        error(Eperm);
    if(NETTYPE(c->qid.path) == Qdir)
        error(Eisdir);

    dir = smalloc(sizeof(DirEntry)+n);
    if(waserror()){
        free(dir);
        nexterror();
    }
    m = convM2D(db, n, &dir[0], (char*)&dir[1]);
    if(m == 0)
        error(Eshortstat);
    if(!emptystr(dir[0].uid))
        error("can't change owner");
    if(dir[0].mode != ~0UL)
        p->perm = dir[0].mode;
    poperror();
    free(dir);
    return m;
}
```

Uses Eshortstat 664d, NETTYPE, Qdir-855 459c, emptystr() 670a, eve 591a, free() 104a, nexterror, poperror 666d, smalloc() 104b, up, and waserror 666d.

[pipeopen] increments the per-side reference count (`qref`) that tracks how many open file descriptors reference each endpoint. The `iounit` is set to `qiomaxatomic`, advertising the maximum size of an atomic write — writes up to this size are guaranteed not to be interleaved with writes from other processes.

⟨*method pipeopen* 463a⟩≡ (739b)
```
/*
 *  if the stream doesn't exist, create it
 */
static Chan*
pipeopen(Chan *c, int omode)
{
    Pipe *p;

    if(c->qid.type & QTDIR){
        if(omode != OREAD)
            error(Ebadarg);
        c->mode = omode;
        c->flag |= COPEN;
        c->offset = 0;
        return c;
    }

    p = c->aux;
    qlock(p);
    switch(NETTYPE(c->qid.path)){
    case Qdata0:
        p->qref[0]++;
        break;
    case Qdata1:
        p->qref[1]++;
        break;
    }
    qunlock(p);

    c->mode = openmode(omode);
    c->flag |= COPEN;
    c->offset = 0;
    c->iounit = qiomaxatomic;
    return c;
}
```
Uses `COPEN` 294b, `Ebadarg` 664d, `NETTYPE`, `OREAD` 853i, `QTDIR` 70b, `Qdata0-856` 459c, `Qdata1-857` 459c, `openmode()` 302b, `qiomaxatomic` 813b, `qlock()` 81f, and `qunlock()` 82c.

[pipeclose] implements the cross-hangup semantics: when the last reference to `data` closes, it hangs up `q[1]` (sending EOF to readers of `data1`) and closes `q[0]`. When both sides reach zero references, the queues are reopened for potential reuse.

⟨*method pipeclose* 463b⟩≡ (739b)
```
static void
pipeclose(Chan *c)
{
    Pipe *p;

    p = c->aux;
    qlock(p);

    if(c->flag & COPEN){
        /*
         *  closing either side hangs up the stream
         */
```

```
        switch(NETTYPE(c->qid.path)){
        case Qdata0:
            p->qref[0]--;
            if(p->qref[0] == 0){
                qhangup(p->q[1], 0);
                qclose(p->q[0]);
            }
            break;
        case Qdata1:
            p->qref[1]--;
            if(p->qref[1] == 0){
                qhangup(p->q[0], 0);
                qclose(p->q[1]);
            }
            break;
        }
    }


    /*
     *  if both sides are closed, they are reusable
     */
    if(p->qref[0] == 0 && p->qref[1] == 0){
        qreopen(p->q[0]);
        qreopen(p->q[1]);
    }

    /*
     *  free the structure on last close
     */
    p->ref--;
    if(p->ref == 0){
        qunlock(p);
        free(p->q[0]);
        free(p->q[1]);
        free(p);
    } else
        qunlock(p);
}
```

Uses COPEN 294b, NETTYPE, Qdata0-856 459c, Qdata1-857 459c, free() 104a, qclose() 418a, qhangup() 418b, qlock() 81f, qreopen() 419b, and qunlock() 82c.

⟨*method piperead* 464⟩≡ (739b)

```
static long
piperead(Chan *c, void *va, long n, vlong)
{
    Pipe *p;

    p = c->aux;

    switch(NETTYPE(c->qid.path)){
    case Qdir:
        return devdirread(c, va, n, pipedir, NPIPEDIR, pipegen);
    case Qdata0:
        return qread(p->q[0], va, n);
    case Qdata1:
        return qread(p->q[1], va, n);
    default:
        panic("piperead");
    }
```

```
    panic("piperead: should not reach this point");
    return -1; // unreachable
}
```

Uses NETTYPE, NPIPEDIR-858 739b, Qdata0-856 459c, Qdata1-857 459c, Qdir-855 459c, devdirread() 432c, pipedir 459d, pipegen() 460b, and qread() 413.

⟨*method pipebread* 465a⟩≡ (739b)

```
static Block*
pipebread(Chan *c, long n, ulong offset)
{
    Pipe *p;

    p = c->aux;

    switch(NETTYPE(c->qid.path)){
    case Qdata0:
        return qbread(p->q[0], n);
    case Qdata1:
        return qbread(p->q[1], n);
    }

    return devbread(c, n, offset);
}
```

Uses NETTYPE, Qdata0-856 459c, Qdata1-857 459c, devbread() 434a, and qbread() 412.

Note the cross-wiring in [pipewrite]: writing to Qdata0 enqueues on q[1], and writing to Qdata1 enqueues on q[0]. If the write fails (the other end is closed), a "sys: write on closed pipe" note is posted — the Plan 9 equivalent of Unix's SIGPIPE.

⟨*method pipewrite* 465b⟩≡ (739b)

```
/*
 *  a write to a closed pipe causes a note to be sent to
 *  the process.
 */
static long
pipewrite(Chan *c, void *va, long n, vlong)
{
    Pipe *p;

    if(!arch_islo())
        print("pipewrite hi %#p\n", getcallerpc(&c));
    if(waserror()) {
        /* avoid notes when pipe is a mounted queue */
        if((c->flag & CMSG) == 0)
            postnote(up, 1, "sys: write on closed pipe", NUser);
        nexterror();
    }

    p = c->aux;

    switch(NETTYPE(c->qid.path)){
    case Qdata0:
        n = qwrite(p->q[1], va, n);
        break;

    case Qdata1:
        n = qwrite(p->q[0], va, n);
        break;

    default:
```

```
        panic("pipewrite");
    }

    poperror();
    return n;
}
```

Uses CMSG 582c, NETTYPE, NUser 449a, Qdata0-856 459c, Qdata1-857 459c, nexterror, poperror 666d, postnote, print, qwrite() 401b, up, and waserror 666d.

⟨*method pipebwrite* 466a⟩≡ (739b)

```
static long
pipebwrite(Chan *c, Block *bp, ulong)
{
    long n;
    Pipe *p;

    if(waserror()) {
        /* avoid notes when pipe is a mounted queue */
        if((c->flag & CMSG) == 0)
            postnote(up, 1, "sys: write on closed pipe", NUser);
        nexterror();
    }

    p = c->aux;
    switch(NETTYPE(c->qid.path)){
    case Qdata0:
        n = qbwrite(p->q[1], bp);
        break;

    case Qdata1:
        n = qbwrite(p->q[0], bp);
        break;

    default:
        n = 0;
        panic("pipebwrite");
    }

    poperror();
    return n;
}
```

Uses CMSG 582c, NETTYPE, NUser 449a, Qdata0-856 459c, Qdata1-857 459c, nexterror, poperror 666d, postnote, qbwrite() 414b, up, and waserror 666d.

⟨*global* pipedevtab 466b⟩≡ (739b)

```
Dev pipedevtab = {
    .dc       =    '|',
    .name     =    "pipe",

    .reset    =    devreset,
    .init     =    pipeinit,
    .shutdown =    devshutdown,
    .attach   =    pipeattach,
    .walk     =    pipewalk,
    .stat     =    pipestat,
    .open     =    pipeopen,
    .create   =    devcreate,
    .close    =    pipeclose,
    .read     =    piperead,
    .bread    =    pipebread,
```

```
    .write    =    pipewrite,
    .bwrite   =    pipebwrite,
    .remove   =    devremove,
    .wstat    =    pipewstat,
};
```

Uses devcreate() 810, devremove() 810, devreset() 810, devshutdown() 810, pipeattach() 460a, pipebread() 465a, pipebwrite() 466a, pipeclose() 463b, pipeinit() 459f, pipeopen() 463a, piperead() 464, pipestat() 461b, pipewalk() 461a, pipewrite() 465b, and pipewstat() 462.

18.3 Shared segment and semaphore

Shared memory segments are created via `segattach()`, which maps a named physical memory region (`Physseg`) into the process's address space. Multiple processes can attach the same segment to share memory. The semaphore syscalls (`semacquire`, `semrelease`) operate on addresses within shared segments, providing the kernel-side backing for user-level locks.

18.3.1 Segment

⟨Segtype *other flags* 467a⟩+≡ (112b) ◁120a

```
SG_CEXEC  = 0100,   /* Detach at exec */
```

⟨sysexec() *free special segments if close on exec* 467b⟩≡ (143d)

```
for(i = BSEG+1; i < NSEG; i++) {
    s = up->seg[i];
    if(s != nil && (s->type&SG_CEXEC)) { // close on exec
        putseg(s);
        up->seg[i] = nil;
    }
}
```

`segbrk` extends a user-created segment (not text, data, or stack). It locates the segment containing `saddr`, verifies it is a custom segment type, then delegates to `ibrk` which grows its virtual address range. This lets user programs manage their own heaps in attached shared segments.

⟨*syscall segbrk* 467c⟩≡ (881b)

```
// void* segbrk(void *saddr, void *addr);
long
syssegbrk(ulong* arg)
{
    int i;
    ulong addr;
    Segment *s;

    addr = arg[0];
    for(i = 0; i < NSEG; i++) {
        s = up->seg[i];
        if(s == nil || addr < s->base || addr >= s->top)
            continue;
        switch(s->type&SG_TYPE) {
        case SG_TEXT:
        case SG_DATA:
        case SG_STACK:
            error(Ebadarg);
        default:
            return ibrk(arg[1], i);
        }
    }
```

```
    error(Ebadarg);
    panic("syssegbrk: should not reach this point");
    return -1; // unreachable
}
```

Uses Ebadarg 664d, NSEG 118c, SG_DATA 556b, SG_STACK 112b, SG_TEXT, SG_TYPE, ibrk() 123b, and up.

⟨*syscall segattach* 468a⟩≡ (881b)

```
// void* segattach(int attr, char *class, void *va, ulong len);
long
syssegattach(ulong* arg)
{
    return segattach(up, arg[0], (char*)arg[1], arg[2], arg[3]);
}
```

Uses segattach() 468c and up.

⟨*hook* _globalsegattach 468b⟩≡ (864b)

```
Segment* (*_globalsegattach)(Proc*, char*);
```

[segattach] first tries to find a globally registered segment with the matching name (via the _globalsegattach hook). If none exists, it searches the physseg table for a matching Physseg, finds a free slot in the process's segment array, and uses [isoverlap] to locate a non-conflicting virtual address range (growing downward from the stack). This is how device drivers expose memory-mapped hardware to user space.

⟨*function* segattach 468c⟩≡ (864b)

```
ulong
segattach(Proc *p, ulong attr, char *name, ulong va, ulong len)
{
    int sno;
    Segment *s, *os;
    Physseg *ps;

    if(va != 0 && va >= USTKTOP)
        error(Ebadarg);

    validaddr((ulong)name, 1, false);
    vmemchr(name, 0, ~0);

    for(sno = 0; sno < NSEG; sno++)
        if(p->seg[sno] == nil && sno != ESEG)
            break;

    if(sno == NSEG)
        error(Enovmem);

    /*
     *  first look for a global segment with the
     *  same name
     */
    if(_globalsegattach != nil){
        s = (*_globalsegattach)(p, name);
        if(s != nil){
            p->seg[sno] = s;
            return s->base;
        }
    }

    len = PGROUND(len);
    if(len == 0)
        error(Ebadarg);
```

```
    /*
     * Find a hole in the address space.
     * Starting at the lowest possible stack address - len,
     * check for an overlapping segment, and repeat at the
     * base of that segment - len until either a hole is found
     * or the address space is exhausted.  Ensure that we don't
     * map the zero page.
     */
    if(va == 0) {
        for (os = p->seg[SSEG]; os != nil; os = isoverlap(p, va, len)) {
            va = os->base;
            if(len >= va)
                error(Enovmem);
            va -= len;
        }
        va &= ~(BY2PG-1);
    } else {
        va &= ~(BY2PG-1);
        if(va == 0 || va >= USTKTOP)
            error(Ebadarg);
    }

    if(isoverlap(p, va, len) != nil)
        error(Esoverlap);


    for(ps = physseg; ps->name; ps++)
        if(strcmp(name, ps->name) == 0)
            goto found;

    error(Ebadarg);
found:
    if(len > ps->size)
        error(Enovmem);

    attr &= ~SG_TYPE;       /* Turn off what is not allowed */
    attr |= ps->attr;       /* Copy in defaults */

    s = newseg(attr, va, len/BY2PG);
    s->pseg = ps;
    p->seg[sno] = s;

    return va;
}
```

Uses ESEG 118c, Ebadarg 664d, Enovmem 664d, Esoverlap 664d, NSEG 118c, PGROUND 120d, SG_TYPE, SSEG 118c, _globalsegattach 468b, false, isoverlap() 469, newseg() 116a, physseg-1804 556f, validaddr() 120b, and vmemchr() 120c.

⟨*function* isoverlap 469⟩≡ (864b)

```
Segment*
isoverlap(Proc *p, ulong va, int len)
{
    int i;
    Segment *ns;
    ulong newtop;

    newtop = va+len;
    for(i = 0; i < NSEG; i++) {
        ns = p->seg[i];
        if(ns == 0)
```

```
            continue;
        if((newtop > ns->base && newtop <= ns->top) ||
           (va >= ns->base && va < ns->top))
            return ns;
    }
    return nil;
}
```

Uses NSEG 118c.

[segdetach] unmaps a previously attached segment from the process's address space. It acquires both the process segment lock and the individual segment lock, finds the segment containing the given address, then calls [putseg] to decrement the reference count. The stack segment (SSEG) cannot be detached. The final arch_flushmmu is essential: without it, stale TLB entries could allow continued access to the detached memory.

⟨*syscall segdetach* 470⟩≡ (881b)

```
// int segdetach(void *addr);
long
syssegdetach(ulong* arg)
{
    int i;
    ulong addr;
    Segment *s;

    qlock(&up->seglock);
    if(waserror()){
        qunlock(&up->seglock);
        nexterror();
    }

    s = 0;
    addr = arg[0];
    for(i = 0; i < NSEG; i++)
        if(s = up->seg[i]) {
            qlock(&s->lk);
            if((addr >= s->base && addr < s->top) ||
               (s->top == s->base && addr == s->base))
                goto found;
            qunlock(&s->lk);
        }

    error(Ebadarg);

found:
    /*
     * Check we are not detaching the initial stack segment.
     */
    if(s == up->seg[SSEG]){
        qunlock(&s->lk);
        error(Ebadarg);
    }
    up->seg[i] = 0;
    qunlock(&s->lk);
    putseg(s);
    qunlock(&up->seglock);
    poperror();

    /* Ensure we flush any entries from the lost segment */
    arch_flushmmu();
    return 0;
}
```

Uses Ebadarg 664d, NSEG 118c, SSEG 118c, nexterror, poperror 666d, putseg() 116c, qlock() 81f, qunlock() 82c, up, and waserror 666d.

segfree releases pages within a segment without detaching the segment itself. This is useful for returning memory to the system while keeping the virtual address range reserved. The address boundaries are page-aligned (rounding from up and to down), so only complete pages are freed.

⟨*syscall segfree* 471a⟩≡ (881b)
```
// int segfree(void *va, ulong len);
long
syssegfree(ulong* arg)
{
    Segment *s;
    ulong from, to;

    from = arg[0];
    s = seg(up, from, 1);
    if(s == nil)
        error(Ebadarg);
    to = (from + arg[1]) & ~(BY2PG-1);
    from = PGROUND(from);

    if(to > s->top) {
        qunlock(&s->lk);
        error(Ebadarg);
    }

    mfreeseg(s, from, (to - from) / BY2PG);
    qunlock(&s->lk);
    arch_flushmmu();

    return 0;
}
```
Uses Ebadarg 664d, PGROUND 120d, mfreeseg() 125c, qunlock() 82c, seg() 119b, and up.

⟨dupseg() *if original segment was shared, flush it* 471b⟩≡ (122a)
```
if(s->ref > 1)
    procflushseg(s); // ??
```
Uses procflushseg() 471d.

⟨mfreeseg() *if segment was shared, flush it* 471c⟩≡ (125c)
```
/* flush this seg in all other processes */
if(s->ref > 1)
    procflushseg(s);
```
Uses procflushseg() 471d.

When a shared segment is modified (e.g., pages freed via mfreeseg), all processes sharing it must flush their TLB entries. [procflushseg] iterates over all processes and CPUs, setting newtlb and flushmmu flags. It then busy-waits for each CPU to take a clock interrupt and perform the flush — an expensive but necessary operation for multiprocessor correctness.

⟨*function* procflushseg 471d⟩≡ (877c)
```
/*
 *  wait till all processes have flushed their mmu
 *  state about segment s
 */
void
procflushseg(Segment *s)
{
    int i, ns, nm;
```

```
    bool wait;
    Proc *p;

    /*
     *  tell all processes with this
     *  segment to flush their mmu's
     */
    wait = false;
    for(i=0; i<conf.nproc; i++) {
        p = &procalloc.arena[i];
        if(p->state == Dead)
            continue;
        for(ns = 0; ns < NSEG; ns++)
            if(p->seg[ns] == s){
                p->newtlb = true;
                for(nm = 0; nm < conf.ncpu; nm++){
                    if(CPUS(nm)->proc == p){
                        CPUS(nm)->flushmmu = true;
                        wait = true;
                    }
                }
                break;
            }
    }
    if(!wait)
        return;

    /*
     *  wait for all processors to take a clock interrupt
     *  and flush their mmu's
     */
    for(nm = 0; nm < conf.ncpu; nm++)
        if(CPUS(nm) != cpu)
            while(CPUS(nm)->flushmmu)
                sched();
}
```

Uses CPUS 702, Dead 67c, NSEG 118c, conf 62b, false, procalloc-2458 128d, sched, and true.

⟨hzclock() *if flushmmu* 472a⟩≡ (236f)

```
if(cpu->flushmmu){
    if(up)
        arch_flushmmu();
    cpu->flushmmu = false;
}
```

Uses false and up.

⟨Cpu *other fields* 472b⟩+≡ (64a) ◁236a 508a▷

```
bool flushmmu;   /* make current proc flush it's mmu state */
```

`segflush` invalidates the instruction cache for a range of addresses. This is needed on architectures with separate instruction and data caches (like ARM) after modifying code in memory — for example, a JIT compiler or dynamic linker must call `segflush` before executing newly written instructions. The function walks the page tables chunk by chunk, marking pages for cache flush via `ptflush`.

⟨*syscall segflush* 472c⟩≡ (866c)

```
long
syssegflush(ulong* arg)
{
    Segment *s;
    ulong addr, l;
```

```
    Pagetable *pt;
    int chunk, ps, pe, len;

    addr = arg[0];
    len = arg[1];

    while(len > 0) {
        s = seg(up, addr, 1);
        if(s == 0)
            error(Ebadarg);

    ⟨syssegflush() set flushme 555k⟩
    more:
        l = len;
        if(addr+l > s->top)
            l = s->top - addr;

        ps = addr-s->base;
        pt = s->pagedir[ps/PAGETABMAPMEM];
        ps &= PAGETABMAPMEM-1;
        pe = PAGETABMAPMEM;
        if(pe-ps > l){
            pe = ps + l;
            pe = ROUND(pe, BY2PG);
        }
        if(pe == ps) {
            qunlock(&s->lk);
            error(Ebadarg);
        }

        // when have cachectl
        if(pt)
            ptflush(pt, ps/BY2PG, pe/BY2PG);

        chunk = pe-ps;
        len -= chunk;
        addr += chunk;

        if(len > 0 && addr < s->top)
            goto more;

        qunlock(&s->lk);
    }
    arch_flushmmu();
    return 0;
}
```

Uses Ebadarg 664d, ROUND 47b, ptflush() 866c, qunlock() 82c, seg() 119b, and up.

18.3.2 Semaphore

Plan 9's semaphores are a hybrid user/kernel design. The semaphore value lives in user memory (shared segment), and the fast path uses compare-and-swap entirely in user space. Only when a process must block does it enter the kernel via `semacquire()`, which enqueues a `Sema` on the segment's wait list and calls [sleep]. This gives spin-lock performance in the uncontended case with proper blocking when needed.

⟨*struct* Sema 473⟩≡ (692)

```
// user level semaphores, used to implement user-level lock,
// see libc/port/lock.c
```

```
struct Sema
{
    long  *addr; // value stored in user space!
    bool waiting;

    //list<Sema> of Segment.sema
    Sema  *next;
    Sema  *prev;

    Rendez;
};
```

⟨Segment *other fields* 474a⟩+≡ (113b) ◁195b 488e▷
```
Sema sema;
```

⟨newseg() *sema initialization* 474b⟩≡ (116a)
```
    // no list, just one sema
    s->sema.prev = &s->sema;
    s->sema.next = &s->sema;
```

cmpswap() (ARM)

⟨*function* arch_cmpswap*(raspberry pi1)(arm)* 474c⟩≡ (972a)
```
bool
arch_cmpswap(long *addr, long old, long new)
{
    return cas32(addr, old, new);
}
```
Uses cas32() 474d.

⟨*function* cas32*(arm)* 474d⟩≡ (972a)
```
bool
cas32(void* addr, u32int old, u32int new)
{
    bool r;
    int s;

    s = arch_splhi();
    r = (*(u32int*)addr == old);
    if(r)
        *(u32int*)addr = new;
    arch_splx(s);
    ⟨cas32() if address contained old value, call coherence(arm) 534g⟩
    return r;
}
```

semxxx()

The semaphore wait list is a doubly-linked circular list rooted at s->sema. Each waiting process has a stack-allocated Sema node linked into this list. semqueue inserts at the tail (before the sentinel), semdequeue removes, and semwakeup walks the list looking for waiters on a specific address. Note that Sema embeds a Rendez, so each waiter has its own sleep/wakeup rendezvous point.

⟨*function* semqueue 474e⟩≡ (882)
```
/* Add semaphore p with addr a to list in seg. */
static void
semqueue(Segment *s, long *a, Sema *p)
{
```

```
    memset(p, 0, sizeof *p);
    p->addr = a;
    lock(&s->sema); /* uses s->sema.Rendez.Lock, but no one else is */
    p->next = &s->sema;
    p->prev = s->sema.prev;
    p->next->prev = p;
    p->prev->next = p;
    unlock(&s->sema);
}
```

Uses `lock()` 76d and `unlock()` 78d.

⟨*function* `semdequeue` 475a⟩≡ (882)

```
/* Remove semaphore p from list in seg. */
static void
semdequeue(Segment *s, Sema *p)
{
    lock(&s->sema);
    p->next->prev = p->prev;
    p->prev->next = p->next;
    unlock(&s->sema);
}
```

Uses `lock()` 76d and `unlock()` 78d.

⟨*function* `semwakeup` 475b⟩≡ (882)

```
/* Wake up n waiters with addr a on list in seg. */
static void
semwakeup(Segment *s, long *a, long n)
{
    Sema *p;

    lock(&s->sema);
    for(p=s->sema.next; p!=&s->sema && n>0; p=p->next){
        if(p->addr == a && p->waiting){
            p->waiting = false;
            arch_coherence();
            wakeup(p);
            n--;
        }
    }
    unlock(&s->sema);
}
```

Uses `arch_coherence`, `false`, `lock()` 76d, `unlock()` 78d, and `wakeup`.

⟨*function* `semrelease` 475c⟩≡ (882)

```
/* Add delta to semaphore and wake up waiters as appropriate. */
static long
semrelease(Segment *s, long *addr, long delta)
{
    long value;

    do
        value = *addr;
    while(!arch_cmpswap(addr, value, value+delta));
    semwakeup(s, addr, delta);
    return value+delta;
}
```

Uses `semwakeup()` 475b.

⟨*function* canacquire 476a⟩≡ (882)

```
/* Try to acquire semaphore using compare-and-swap */
static bool
canacquire(long *addr)
{
    long value;

    while((value=*addr) > 0)
        if(arch_cmpswap(addr, value, value-1))
            return true;
    return false;
}
```

Uses false and true.

⟨*function* semawoke 476b⟩≡ (882)

```
/* Should we wake up? */
static bool
semawoke(void *p)
{
    arch_coherence();
    return !((Sema*)p)->waiting;
}
```

Uses arch_coherence.

The [semacquire] loop is subtle: after enqueuing itself on the wait list, the process sets `phore.waiting = true`, calls `coherence()` to ensure the write is visible, then tries `canacquire` one more time. This recheck prevents a lost-wakeup race where `semrelease` could run between the failed `canacquire` and the [sleep]. If a note interrupts the sleep, the function cleans up and re-raises the error via [nexterror].

⟨*function* semacquire 476c⟩≡ (882)

```
/* Acquire semaphore (subtract 1). */
static bool
semacquire(Segment *s, long *addr, bool block)
{
    int acquired;
    Sema phore;

    if(canacquire(addr))
        return true;
    if(!block)
        return false;

    acquired = false;
    semqueue(s, addr, &phore);
    for(;;){
        phore.waiting = true;
        arch_coherence();
        if(canacquire(addr)){
            acquired = true;
            break;
        }
        if(waserror())
            break;
        sleep(&phore, semawoke, &phore);
        poperror();
    }
    semdequeue(s, &phore);
    arch_coherence();    /* not strictly necessary due to lock in semdequeue */
    if(!phore.waiting)
        semwakeup(s, addr, 1);
```

```
    if(!acquired)
        nexterror();
    return false;
}
```

Uses arch_coherence, canacquire() 476a, false, nexterror, poperror 666d, semawoke() 476b, semdequeue() 475a, semqueue() 474e, semwakeup() 475b, sleep, true, and waserror 666d.

[tsemacquire] adds a timeout to the acquire loop. It uses `tsleep` instead of `sleep` and tracks elapsed time via CPU ticks. The `ms` budget is decremented each iteration, so multiple spurious wakeups are handled correctly.

⟨*function* tsemacquire 477a⟩≡ (882)

```
/* Acquire semaphore or time-out */
static bool
tsemacquire(Segment *s, long *addr, ulong ms)
{
    int acquired, timedout;
    ulong t, elms;
    Sema phore;

    if(canacquire(addr))
        return true;
    if(ms == 0)
        return false;
    acquired = timedout = false;
    semqueue(s, addr, &phore);
    for(;;){
        phore.waiting = true;
        arch_coherence();
        if(canacquire(addr)){
            acquired = true;
            break;
        }
        if(waserror())
            break;
        t = cpu->ticks;
        tsleep(&phore, semawoke, &phore, ms);
        elms = TK2MS(cpu->ticks - t);
        poperror();
        if(elms >= ms){
            timedout = true;
            break;
        }
        ms -= elms;
    }
    semdequeue(s, &phore);
    arch_coherence();    /* not strictly necessary due to lock in semdequeue */
    if(!phore.waiting)
        semwakeup(s, addr, 1);
    if(timedout)
        return false;
    if(!acquired)
        nexterror();
    return false;
}
```

Uses TK2MS 244d, arch_coherence, canacquire() 476a, false, nexterror, poperror 666d, semawoke() 476b, semdequeue() 475a, semqueue() 474e, semwakeup() 475b, true, tsleep, and waserror 666d.

⟨*syscall semacquire* 477b⟩≡ (882)

```
// int semacquire(long *addr, int block);
long
```

```
syssemacquire(ulong* arg)
{
    int block;
    long *addr;
    Segment *s;

    validaddr(arg[0], sizeof(long), true);
    arch_validalign(arg[0], sizeof(long));
    addr = (long*)arg[0];
    block = arg[1];

    if((s = seg(up, (ulong)addr, 0)) == nil)
        error(Ebadarg);
    if(*addr < 0)
        error(Ebadarg);
    return semacquire(s, addr, block);
}
```

Uses Ebadarg 664d, seg() 119b, semacquire() 476c, true, up, and validaddr() 120b.

⟨*syscall tsemacquire* 478a⟩≡ (882)

```
// int tsemacquire(long *addr, ulong ms);
long
systsemacquire(ulong* arg)
{
    long *addr;
    ulong ms;
    Segment *s;

    validaddr(arg[0], sizeof(long), true);
    arch_validalign(arg[0], sizeof(long));
    addr = (long*)arg[0];
    ms = arg[1];

    if((s = seg(up, (ulong)addr, 0)) == nil)
        error(Ebadarg);
    if(*addr < 0)
        error(Ebadarg);
    return tsemacquire(s, addr, ms);
}
```

Uses Ebadarg 664d, seg() 119b, true, tsemacquire() 477a, up, and validaddr() 120b.

⟨*syscall semrelease* 478b⟩≡ (882)

```
// long semrelease(long *addr, long count);
long
syssemrelease(ulong* arg)
{
    long *addr, delta;
    Segment *s;

    validaddr(arg[0], sizeof(long), true);
    arch_validalign(arg[0], sizeof(long));
    addr = (long*)arg[0];
    delta = arg[1];

    if((s = seg(up, (ulong)addr, 0)) == nil)
        error(Ebadarg);
    /* delta == 0 is a no-op, not a release */
    if(delta < 0 || *addr < 0)
        error(Ebadarg);
    return semrelease(s, addr, delta);
```

```
}
```

Uses Ebadarg 664d, seg() 119b, semrelease() 475c, true, up, and validaddr() 120b.

18.4 sysrendezvous()

rendezvous() is a lightweight synchronization primitive that also exchanges a value. Two processes call rendezvous(tag, value) with the same tag: the first to arrive sleeps in a hash table; when the second arrives, they swap values and both continue. The Rgrp (rendezvous group) scopes which processes can rendezvous with each other.

⟨*function* REND 479a⟩≡ (868c)

```
enum
{
    RENDLOG = 5,
    RENDHASH =  1<<RENDLOG, /* Hash to lookup rendezvous tags */
};
#define REND(p,s) ((p)->rendhash[(s)&((1<<RENDLOG)-1)])
```

Uses RENDLOG 479a.

⟨*struct* Rgrp 479b⟩≡ (868c)

```
struct Rgrp
{
    // hash<??, list<ref<Proc>>>
    Proc  *rendhash[RENDHASH];  /* Rendezvous tag hash */

    // extra
    Ref;        /* the Ref's lock is also the Rgrp's lock */
};
```

⟨Proc *synchronization fields* 479c⟩+≡ (66) ◁254f

```
Rgrp  *rgrp;    /* Rendez group */

uintptr rendtag;  /* Tag for rendezvous */
uintptr rendval;  /* Value for rendezvous */
//??
Proc  *rendhash;  /* Hash list for tag values */
```

⟨*constructor* newrgrp 479d⟩≡ (875)

```
Rgrp*
newrgrp(void)
{
    Rgrp *r;

    r = smalloc(sizeof(Rgrp));
    r->ref = 1;
    return r;
}
```

Uses smalloc() 104b.

⟨*destructor* closergrp 479e⟩≡ (875)

```
void
closergrp(Rgrp *r)
{
    if(decref(r) == 0)
        free(r);
}
```

Uses decref() 87c and free() 104a.

⟨Procstate *cases* 480a⟩+≡ (67c) ◁233d 482d▷
```
Rendezvous,
```

[sysrendezvous] searches the hash table for a process with the same tag. If found, it swaps values and wakes the sleeper. If not found, the caller inserts itself into the hash chain, sets state to `Rendezvous`, and calls `sched()` to sleep until a partner arrives. A return value of `0` indicates interruption (e.g., by a note).

⟨*syscall rendezvous* 480b⟩≡ (881b)
```
// void* rendezvous(void* tag, void* value);
long
sysrendezvous(ulong* arg)
{
    uintptr tag, val;
    Proc *p, **l;

    tag = arg[0];
    l = &REND(up->rgrp, tag);
    up->rendval = ~(uintptr)0;

    lock(up->rgrp);
    for(p = *l; p; p = p->rendhash) {
        if(p->rendtag == tag) {
            *l = p->rendhash;
            val = p->rendval;
            p->rendval = arg[1];

            while(p->cpu != 0) // ????
                ;

            ready(p);
            unlock(up->rgrp);
            return val;
        }
        l = &p->rendhash;
    }

    /* Going to sleep here */
    up->rendtag = tag;
    up->rendval = arg[1];
    up->rendhash = *l;
    *l = up;
    up->state = Rendezvous;
    unlock(up->rgrp);

    sched();

    return up->rendval;
}
```
Uses `REND` 479a, `Rendezvous` 480a, `lock()` 76d, `ready`, `sched`, `unlock()` 78d, and `up`.

18.5 Network

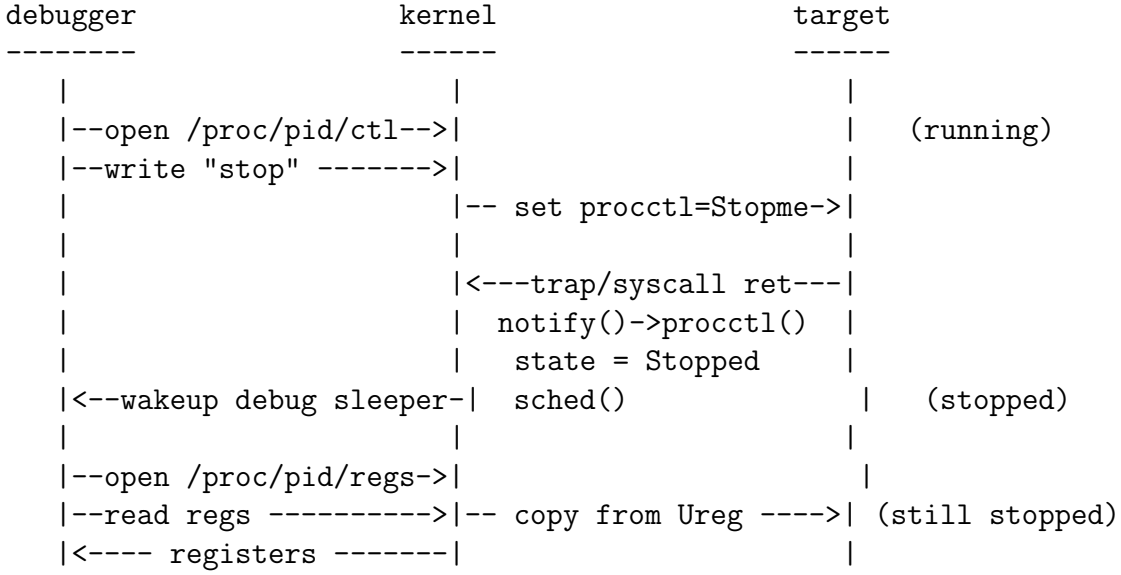
Chapter 19

Debugging Support

In this chapter, I will present the kernel infrastructure that makes debuggers and system call tracers possible. Building a tool like `/bin/db` (the Plan 9 debugger) or `/bin/ratrace` (Plan 9 equivalent of Linux `strace`) requires three capabilities from the kernel: the ability to *stop* another process, the ability to *inspect* its memory and registers, and the ability to *resume* it. In Plan 9, all of these are provided through the `/proc` filesystem—there is no `ptrace()` system call as in UNIX. The debugger simply reads and writes files like `/proc/\emphpid/mem`, `/proc/\emphpid/regs`, and `/proc/\emphpid/ctl`.

Other operating systems reach these same three primitives by very different routes. Linux and the BSDs add a dedicated system call, `ptrace`, with dozens of sub-commands (`PEEKDATA`, `POKEDATA`, `GETREGS`, `SYSCALL`, `SINGLESTEP`...) that in effect form a small RPC language—conceptually similar to reading and writing files, but encoded as opcodes and copied through kernel buffers. Darwin/macOS keeps `ptrace` for source compatibility but layers the real debugger interface on Mach exception ports: when a task hits a breakpoint the kernel synthesizes a Mach IPC message and delivers it to whatever port the debugger subscribed to—turning "stop and inspect" into "receive a message." Windows goes the other way and makes debugging a scheduling event: a debuggee is attached with `DebugActiveProcess`, which causes `WaitForDebugEvent` to be the thread's natural blocking point whenever the debuggee faults. Historically, debuggers on DOS and CP/M patched `INT 3` directly into the code and installed their own interrupt vector. Plan 9's choice—route everything through named files—is the most austere of the lot: there are no new system calls, no new IPC primitives, and no kernel-side protocol parser. `/proc/\emphpid/mem` is literally a `read`/`write` on a `Chan`, so any program that can open a file can, in principle, implement a debugger.

The stop/inspect/resume cycle then looks like this in terms of the three processes involved (the debugger, the kernel, and the target):



```
|                          |                          |
|--write "start" ------>|-- ready(target) ----->|
|                          |                          |   (running)
```

The point to notice is that the kernel never synchronously suspends another process: the "stop" write only sets the `procctl` flag on the target and returns. The target actually stops itself the next time it passes through `notify()`, i.e. on its next trap or syscall return. This is the same deferred-delivery mechanism used for notes (signals), and it means the debugger may have to wait a little (Plan 9 blocks the debugger on the `Proc.sleepr` rendezvous until `procctl()` actually fires). The benefit is huge: the kernel never has to preempt another CPU mid-instruction, never has to worry about what state the stopped process was in, and the "safe points" are exactly the points at which `Ureg` is already saved on the kernel stack.

Breakpoints are implemented using the ARM BKPT instruction. When the processor hits this instruction, it triggers a prefetch fault, and the trap handler recognizes it as a debug event. For user-mode breakpoints, the kernel posts a "sys: breakpoint" note to the process, which the debugger can intercept. A kernel-mode breakpoint is a fatal error.

⟨`trap()` *when undefined instruction, if breakpoint(arm)* 482a⟩≡ (170e)

```
if(seg(up, ureg->pc, false) != nil &&
   *(u32int*)ureg->pc == 0xD1200070)
    postnote(up, 1, "sys: breakpoint", NDebug);
```

Uses `NDebug` 449a, `false`, `postnote`, `seg()` 119b, and `up`.

⟨`trap()` *when prefetch fault, switch fst cases(arm)* 482b⟩≡ (171c)

```
case 0x02:      /* instruction debug event (BKPT) */
    if(user){
        snprint(buf, sizeof buf, "sys: breakpoint");
        postnote(up, 1, buf, NDebug);
    }else{
        iprint("kernel bkpt: pc %#lux inst %#ux\n",
            ureg->pc, *(u32int*)ureg->pc);
        panic("kernel bkpt");
    }
    break;
```

Uses `NDebug` 449a, `iprint`, `postnote`, and `up`.

⟨*function* `arch_setregisters`*(arm)* 482c⟩≡ (1008b)

```
/* This routine must save the values of registers the user is not permitted
 * to write from devproc and then restore the saved values before returning.
 */
void
arch_setregisters(Ureg* ureg, char* pureg, char* uva, int n)
{
    USED(ureg, pureg, uva, n);
}
```

19.1 Core dumps and broken processes

Instead of writing core dumps to disk like UNIX, Plan 9 keeps a crashed process alive in the `Broken` state. The process retains its address space and registers intact, so a debugger can attach to it with `db \emphpid` and do post-mortem analysis without any intermediate file. This is both more convenient (no multi-megabyte core files) and more faithful (the entire process state is preserved, not just what the kernel chose to dump). The kernel limits broken processes to `NBROKEN` (4) at a time; when a new one arrives and the table is full, the oldest is evicted and allowed to finish dying.

⟨`Procstate` *cases* 482d⟩+≡ (67c) ◁480a 484c▷

```
Broken,
```

⟨*struct* Broken 483a⟩≡ (877c)
```
/*
 * weird thing: keep at most NBROKEN around
 */
#define NBROKEN 4
struct Broken
{
    // array<ref<Proc>>
    Proc    *p[NBROKEN];
    // number of entries used in p
    int n;

    // extra
    QLock;
};
```
Uses NBROKEN-2462 483a.

⟨*global* broken 483b⟩≡ (877c)
```
struct Broken broken;
```
Uses Broken 482d 483a.

⟨pexit() *if not freemem, add process to broken processes list* 483c⟩≡ (152c)
```
// instead of a core dump we just keep the process around
if(!freemem)
    addbroken(up); // will call sched()
```
Uses addbroken() 483d and up.

When addbroken() is called (from pexit() with freemem false), the process is suspended in the Broken state and added to the fixed-size broken array. If the array is full, the oldest broken process is evicted by calling ready() on it, which resumes its pexit() and lets it finally free its resources and die. Killing a broken process (via "kill" in its ctl file) calls unbreak(), which also readies it to resume dying. The freebroken() function is a last-resort call from the pager when memory is critically low.

⟨*function* addbroken 483d⟩≡ (877c)
```
void
addbroken(Proc *p)
{
    qlock(&broken);
    if(broken.n == NBROKEN) {
        ready(broken.p[0]);
        memmove(&broken.p[0], &broken.p[1], sizeof(Proc*)*(NBROKEN-1));
        --broken.n;
    }
    broken.p[broken.n++] = p;
    qunlock(&broken);

    edfstop(up);
    p->state = Broken;
    p->psstate = nil;
    sched();
}
```
Uses Broken 482d 483a, NBROKEN-2462 483a, broken 483b, edfstop() 568c, qlock() 81f, qunlock() 82c, ready, sched, and up.

⟨*function* unbreak 483e⟩≡ (877c)
```
void
unbreak(Proc *p)
{
    int b;
```

```
    qlock(&broken);
    for(b=0; b < broken.n; b++)
        if(broken.p[b] == p) {
            broken.n--;
            memmove(&broken.p[b], &broken.p[b+1],
                    sizeof(Proc*)*(NBROKEN-(b+1)));
            ready(p);
            break;
        }
    qunlock(&broken);
}
```

Uses NBROKEN-2462 483a, broken 483b, qlock() 81f, qunlock() 82c, and ready.

⟨*function* freebroken 484a⟩≡ (877c)

```
int
freebroken(void)
{
    int i, n;

    qlock(&broken);
    n = broken.n;
    for(i=0; i<n; i++) {
        ready(broken.p[i]);
        broken.p[i] = nil;
    }
    broken.n = 0;
    qunlock(&broken);
    return n;
}
```

Uses broken 483b, qlock() 81f, qunlock() 82c, and ready.

⟨procctlreq() *CMkill case, Broken case* 484b⟩≡ (159c)

```
        case Broken:
            // will resume the broken process to finally die and free its mem
            unbreak(p);
            break;
```

Uses Broken 482d 483a and unbreak() 483e.

19.2 Stopping a process: /proc/<pid>/ctl

The Stopped state is the foundation for all debugging. A process enters this state when procctl() is called with Proc_stopme. But a process can only stop *itself*—the kernel cannot arbitrarily suspend a running process on another CPU. Instead, the debugger writes "stop" to /proc/\emphpid/ctl, which sets the target's procctl field to Proc_stopme. This field is checked at the next trap or system call return (via notify()), at which point the process stops itself. This is similar to how notes (signals) are delivered—they are deferred until a safe point.

⟨Procstate *cases* 484c⟩+≡ (67c) ◁482d 568b▷

```
Stopped,
```

⟨*enum* procctl 484d⟩≡ (868c)

```
enum Procctl
{
    Proc_nothing = 0,
    Proc_stopme,
    ⟨Procctl cases 159a⟩
};
```

⟨Proc *debugger fields* 485a⟩+≡ (66) ◁165e 485e▷
```
// enum<procctl>
int procctl;  /* Control for /proc debugging */
```

The `procctl()` function is where a process actually stops itself. It is called from `notify()` on return from a trap or system call, so it runs in the context of the process being controlled (always `up`). For `Proc_stopme`, the process sets its state to `Stopped`, wakes up any waiting debugger via `pdbg->sleepr`, and calls `sched()` to give up the CPU. It will remain in the scheduler's stopped list until the debugger writes "start" to its ctl file.

⟨*function* `procctl` 485b⟩≡ (877c)
```
/*
 *  called arch_splhi() by notify().  See comment in notify for the
 *  reasoning.
 */
void
procctl(Proc *p)
{
    char *oldstate;
    ulong s;

    switch(p->procctl) {
    ⟨procctl() Proc_traceme case (and fallthrough Proc_stopme) 502b⟩
    case Proc_stopme:
        p->procctl = Proc_nothing;
        oldstate = p->psstate;
        p->psstate = "Stopped";
        s = arch_spllo();
        ⟨procctl() wakeup waiting debugger 487a⟩
        arch_splhi();
        p->state = Stopped;
        sched();
        // here when something (debugger, strace, user) ready(p) back
        p->psstate = oldstate;
        arch_splx(s);
        return;
    ⟨procctl() Proc_exitbig case 876c⟩
    ⟨procctl() Proc_exitme case 159b⟩
    }
}
```
Uses `Proc_nothing` 484d, `Proc_stopme` 484d, `Stopped` 484c, and `sched`.

⟨procctlreq() *CMstop case* 485c⟩≡ (620b)
```
case CMstop:
    procstopwait(p, Proc_stopme);
    break;
```
Uses `CMstop-893` 602b, `Proc_stopme` 484d, and `procstopwait()` 486c.

⟨procctlreq() *CMstart case* 485d⟩≡ (620b)
```
case CMstart:
    if(p->state != Stopped)
        error(Ebadctl);
    ready(p);
    break;
```
Uses `CMstart-890` 602b, `Stopped` 484c, and `ready`.

⟨Proc *debugger fields* 485e⟩+≡ (66) ◁485a 486b▷
```
Proc  *pdbg;    /* the debugging process */
```

⟨pexit() *release debuggers* 486a⟩≡ (152c)
```
/* release debuggers */
qlock(&up->debug);
if(up->pdbg) {
    wakeup(&up->pdbg->sleepr);
    up->pdbg = nil;
}
qunlock(&up->debug);
```
Uses qlock() 81f, qunlock() 82c, up, and wakeup.

⟨Proc *debugger fields* 486b⟩+≡ (66) ◁485e 487b▷
```
QLock debug;    /* to access debugging elements */ // used for many things
```

The procstopwait() function is the debugger's half of the stop/resume handshake. The debugger (caller) sets the target's procctl field, then sleeps on its own sleepr rendez until the target actually reaches the Stopped state. The pdbg field records who is debugging whom, so only one debugger can control a process at a time. When the target stops (in procctl()), it wakes up the sleeping debugger.

⟨*function* procstopwait 486c⟩≡ (740b)
```
// assumes p->debug is held
void
procstopwait(Proc *p, int ctl)
{
    int pid;

    if(p->pdbg)
        error(Einuse);
    if(procstopped(p) || p->state == Broken)
        return;

    if(ctl != Proc_nothing)
        p->procctl = ctl;

    p->pdbg = up;
    pid = p->pid;

    qunlock(&p->debug);
    up->psstate = "Stopwait";
    if(waserror()) {
        p->pdbg = nil;
        qlock(&p->debug);
        nexterror();
    }

    sleep(&up->sleepr, procstopped, p);

    poperror();
    qlock(&p->debug);
    if(p->pid != pid) // p was reallocated to a new process
        error(Eprocdied);
}
```
Uses Broken 482d 483a, Eprocdied 664d, Proc_nothing 484d, nexterror, poperror 666d, procstopped() 486d, qlock() 81f, qunlock() 82c, sleep, up, and waserror 666d.

⟨*function* procstopped 486d⟩≡ (740b)
```
bool
procstopped(void *a)
{
    Proc *p = a;
    return p->state == Stopped;
```

```
}
```

Uses Stopped 484c.

⟨procctl() *wakeup waiting debugger* 487a⟩≡ (485b)

```
/* free a waiting debugger */
qlock(&p->debug);
if(p->pdbg) {
    wakeup(&p->pdbg->sleepr);
    p->pdbg = nil;
}
qunlock(&p->debug);
```

Uses qlock() 81f, qunlock() 82c, and wakeup.

To debug a program from the start, the debugger uses the "hang" mechanism. The sequence is: fork a child, write "hang" to the child's ctl file, then let the child exec the target program. During sysexec(), the kernel sees up->hang and sets procctl to Proc_stopme, so the child stops before executing its first instruction. The debugger can then set breakpoints, inspect the initial state, and resume with "start". The hang flag is inherited across rfork, so strace can trace children of the traced process as well.

⟨Proc *debugger fields* 487b⟩+≡ (66) ◁486b 492b▷

```
bool hang;   /* hang at next exec for debug */
```

⟨procctlreq() *CMhang case* 487c⟩≡ (620b)

```
case CMhang:
    p->hang = true;
    break;
```

Uses CMhang-881 602b and true.

⟨procctlreq() *CMnohang case* 487d⟩≡ (620b)

```
case CMnohang:
    p->hang = false;
    break;
```

Uses CMnohang-883 602b and false.

⟨sysrfork() *when* RFPROC *propagate fields from up to p* 487e⟩+≡ (134b) ◁450d 514b▷

```
p->hang = up->hang;
```

Uses up.

⟨sysexec() *if hang* 487f⟩≡ (141c)

```
if(up->hang)
    up->procctl = Proc_stopme;
```

⟨procctlreq() *CMwaitstop case* 487g⟩≡ (620b)

```
case CMwaitstop:
    procstopwait(p, Proc_nothing);
    break;
```

Uses CMwaitstop-894 602b, Proc_nothing 484d, and procstopwait() 486c.

19.3 Inspecting process memory: /proc/<pid>/mem

Reading /proc/\emphpid/mem lets a debugger inspect another process's address space. Reads at offsets below KZERO go through procctlmemio(), which walks the target's page tables and temporarily maps the physical page into kernel space to copy the data. Writes require the process to be stopped first. If the debugger writes to

a text segment, the kernel first converts it to a data segment via `text2data()`—this is copy-on-write for code, needed because text segments may be shared across processes.

⟨`procread()` *cases* 488a⟩+≡ (605d) ◁352c 492d▷
```
case Qmem:
    if(offset < KZERO)
        return procctlmemio(p, offset, n, va, true);

    if(!iseve())
        error(Eperm);

    // why allowing read access to kernel code or kernel memory?
    // who is using that?

    /* validate kernel addresses */
    if(offset < (kern_addr)end) {
        if(offset+n > (kern_addr)end)
            n = (kern_addr)end - offset;
        memmove(a, (char*)offset, n);
        return n;
    }
    for(i=0; i<nelem(conf.mem); i++){
        cm = &conf.mem[i];
        /* klimit-1 because klimit might be zero! */
        if(cm->kbase <= offset && offset <= cm->klimit-1){
            if(offset+n >= cm->klimit-1)
                n = cm->klimit - offset;
            memmove(a, (char*)offset, n);
            return n;
        }
    }
    error(Ebadarg);
```
Uses `Ebadarg` 664d, `Qmem-866` 602a, `conf` 62b, `iseve()` 591b, `nelem` 853i, `procctlmemio()` 488f, and `true`.

⟨`xinit()` *nkpages kernel memory in m* 488b⟩≡ (282)
```
m->kbase = (ulong)KADDR(m->base);
m->klimit = (ulong)KADDR(m->base+nkpages*BY2PG);
```

⟨`Confmem` *other fields* 488c⟩≡ (62c)
```
kern_addr kbase; // KADDR(Confmem.base)
kern_addr klimit; // KADDR(base+ x*BY2PG) where x <= Confmem.npage
```

⟨`procwrite()` *cases* 488d⟩≡ (608c)
```
case Qmem:
    if(p->state != Stopped)
        error(Ebadctl);

    n = procctlmemio(p, offset, n, va, false);
    break;
```
Uses `Qmem-866` 602a, `Stopped` 484c, `false`, and `procctlmemio()` 488f.

⟨`Segment` *other fields* 488e⟩+≡ (113b) ◁474a 505d▷
```
ushort  steal;    /* Page stealer lock */
```

⟨*function* `procctlmemio` 488f⟩≡ (740b)
```
int
procctlmemio(Proc *p, ulong offset, int n, virt_vp va, bool read)
{
    Arch_KMap *k;
    Pagetable *pt;
```

```
Page *pg;
Segment *s;
ulong soff, l;
char *a = va, *b;

for(;;) {
    s = seg(p, offset, 1);
    if(s == nil)
        error(Ebadarg);

    if(offset+n >= s->top)
        n = s->top-offset;

    if(!read && (s->type&SG_TYPE) == SG_TEXT)
        s = text2data(p, s);

    s->steal++;
    soff = offset-s->base;
    if(waserror()) {
        s->steal--;
        nexterror();
    }
    if(fixfault(s, offset, read, /*putmmu*/false) == 0)
        break;
    poperror();
    s->steal--;
}
poperror();
pt = s->pagedir[soff/PAGETABMAPMEM];
if(pt == nil)
    panic("procctlmemio");
pg = pt->pagetab[(soff&(PAGETABMAPMEM-1))/BY2PG];
if(pagedout(pg))
    panic("procctlmemio1");

l = BY2PG - (offset&(BY2PG-1));
if(n > l)
    n = l;

k = arch_kmap(pg);
if(waserror()) {
    s->steal--;
    arch_kunmap(k);
    nexterror();
}
b = (char*)VA(k);
b += offset&(BY2PG-1);
if(read == 1)
    memmove(a, b, n);   /* This can fault */
else
    memmove(b, a, n);
arch_kunmap(k);
poperror();
```

⟨`procctlmemio()` *set cachectl if flushme segment* 555m⟩

```
s->steal--;

if(read == 0)
    p->newtlb = false;
```

```
    return n;
}
```

Uses Ebadarg 664d, SG_TEXT, SG_TYPE, false, fixfault() 185, nexterror, pagedout 855, poperror 666d, seg() 119b, text2data() 490, and waserror 666d.

⟨*function* text2data 490⟩≡ (740b)

```
Segment*
text2data(Proc *p, Segment *s)
{
    int i;
    Segment *ps;

    ps = newseg(SG_DATA, s->base, s->size);
    ps->image = s->image;
    incref(ps->image);
    ps->fstart = s->fstart;
    ps->flen = s->flen;
    ⟨text2data() initialize other fields 555i⟩

    qlock(&p->seglock);
    for(i = 0; i < NSEG; i++)
        if(p->seg[i] == s)
            break;
    if(i == NSEG)
        panic("segment gone");

    qunlock(&s->lk);
    putseg(s);
    qlock(&ps->lk);
    p->seg[i] = ps;
    qunlock(&p->seglock);

    return ps;
}
```

Uses NSEG 118c, SG_DATA 556b, incref() 87b, newseg() 116a, putseg() 116c, qlock() 81f, and qunlock() 82c.

19.4 /bin/ratrace

System call tracing uses a ping-pong protocol between the tracer and the traced process. The tracer writes "startsyscall" to /proc/\emphpid/ctl, which readies the traced process and immediately sleeps waiting for it to stop again. The traced process runs until its next syscall entry, where it formats the syscall name and arguments into syscalltrace, sets itself to Proc_stopme, and stops. The tracer wakes up, reads /proc/\emphpid/syscall, then writes "startsyscall" again to let the traced process execute the actual syscall. The process stops a second time after the syscall completes, allowing the tracer to read the return value. Here is the sequence:

```
tracer                          traced (ls)
  |                                |
  |-- "startsyscall" --> ready     |
  |   (sleep on sleepr)            |
  |                          run until syscall
  |                          format args
  |                          stop (wakeup tracer)
  |   read /proc/pid/syscall <---|
  |-- "startsyscall" --> ready     |
```

```
|   (sleep on sleepr)          |
|                          execute syscall
|                          format return value
|                          stop (wakeup tracer)
|   read /proc/pid/syscall <---|
|         ... repeat ...       |
```

⟨Procctl *cases* 491a⟩+≡ (484d) ◁159a 502a▷
```
Proc_tracesyscall,
```

⟨procctlreq() *CMstartsyscall case* 491b⟩≡ (620b)
```
case CMstartsyscall:
    if(p->state != Stopped)
        error(Ebadctl);
    p->procctl = Proc_tracesyscall;
    ready(p);
    procstopwait(p, Proc_tracesyscall); // will sleep
    break;
```
Uses CMstartsyscall-892 602b, Proc_tracesyscall 491a, Stopped 484c, procstopwait() 486c, and ready.

⟨syscall() Proc_tracesyscall *if, syscall entry(arm)* 491c⟩≡ (163e)
```
if(up->procctl == Proc_tracesyscall){
    /*
     * Redundant validaddr.  Do we care?
     * Tracing syscalls is not exactly a fast path...
     * Beware, validaddr currently does a pexit rather
     * than an error if there's a problem; that might
     * change in the future.
     */
    if(sp < (USTKTOP-BY2PG) || sp > (USTKTOP-sizeof(Sargs)-BY2WD))
        validaddr(sp, sizeof(Sargs)+BY2WD, false);

    syscallfmt(scallnr, ureg->pc, (va_list)(sp+BY2WD));
    up->procctl = Proc_stopme;
    // this will call sched() and wakeup the tracer process
    procctl(up);
    // back here when the tracer process readied us back and
    // should have set procctl back to Proc_tracesyscall
    if(up->syscalltrace)
        free(up->syscalltrace);
    up->syscalltrace = nil;
}
```
Uses Proc_stopme 484d, Proc_tracesyscall 491a, false, free() 104a, procctl() 485b, syscallfmt() 492e, up, and validaddr() 120b.

⟨syscall() *other locals(arm)* 491d⟩+≡ (163e) ◁165a
```
vlong startns, stopns;
```

⟨syscall() Proc_tracesyscall *if, syscall exit(arm)* 491e⟩≡ (163e)
```
if(up->procctl == Proc_tracesyscall){
    stopns = todget(nil);
    up->procctl = Proc_stopme;
    sysretfmt(scallnr, (va_list)(sp+BY2WD), ret, startns, stopns);
    s = arch_splhi();
    procctl(up); // again, will call sched() and wakeup tracer process
    arch_splx(s);
    if(up->syscalltrace)
        free(up->syscalltrace);
    up->syscalltrace = nil;
}
```
Uses Proc_stopme 484d, Proc_tracesyscall 491a, free() 104a, procctl() 485b, sysretfmt() 496, todget() 262, and up.

⟨sysrfork() *inherit* Proc_tracesyscall 492a⟩≡
```
    if(up && up->procctl == Proc_tracesyscall)
        p->procctl = Proc_tracesyscall;
```

⟨Proc *debugger fields* 492b⟩+≡ (66) ◁487b 498b▷
```
  char  *syscalltrace;  /* syscall trace */
```

⟨pexit() *at the start, free resources* 492c⟩+≡ (152c) ◁257d
```
  if(up->syscalltrace)
      free(up->syscalltrace);
```
Uses free() 104a and up.

⟨procread() *cases* 492d⟩+≡ (605d) ◁488a 507a▷
```
  case Qsyscall:
      if(!p->syscalltrace)
          return 0;
      n = readstr(offset, a, n, p->syscalltrace);
      return n;
```
Uses Qsyscall-878 602a and readstr() 670d.

The syscallfmt() and sysretfmt() functions format system call entries and exits as human-readable strings for strace. Each syscall number has a custom case that knows how to decode its specific argument types—strings are read from user space and quoted, pointers are shown as hex, file descriptors as integers. The result is stored in up->syscalltrace, which the tracer reads via /proc/\emphpid/syscall.

⟨*function* syscallfmt 492e⟩≡ (884b)
```
  void
  syscallfmt(int syscallno, ulong pc, va_list list)
  {
      long l;
      Fmt fmt;
      void *v;
      vlong vl;
      uintptr p;
      int i[2], len;
      char *a, **argv;

      fmtstrinit(&fmt);
      fmtprint(&fmt, "%uld %s ", up->pid, up->text);

      if(syscallno > nsyscall)
          fmtprint(&fmt, " %d ", syscallno);
      else
          fmtprint(&fmt, "%s ", sysctab[syscallno]?
              sysctab[syscallno]: "huh?");

      fmtprint(&fmt, "%ulx ", pc);
      if(up->syscalltrace != nil)
          free(up->syscalltrace);

      switch(syscallno){
      case NOP:
          p = va_arg(list, uintptr);
          fmtprint(&fmt, "%#p", p);
          break;
      case CHDIR:
      case EXITS:
      case REMOVE:
          a = va_arg(list, char*);
          fmtuserstring(&fmt, a, "");
```

```
    break;
case BIND:
    a = va_arg(list, char*);
    fmtuserstring(&fmt, a, " ");
    a = va_arg(list, char*);
    fmtuserstring(&fmt, a, " ");
    i[0] = va_arg(list, int);
    fmtprint(&fmt, "%#ux",  i[0]);
    break;
case CLOSE:
case NOTED:
    i[0] = va_arg(list, int);
    fmtprint(&fmt, "%d", i[0]);
    break;
case DUP:
    i[0] = va_arg(list, int);
    i[1] = va_arg(list, int);
    fmtprint(&fmt, "%d %d", i[0], i[1]);
    break;
case ALARM:
    l = va_arg(list, unsigned long);
    fmtprint(&fmt, "%#lud ", l);
    break;
case EXEC:
    a = va_arg(list, char*);
    fmtuserstring(&fmt, a, "");
    argv = va_arg(list, char**);
    //validalign(PTR2UINT(argv), sizeof(char*)); ??
    for(;;){
        validaddr((ulong)argv, sizeof(char**), false);
        a = *(char **)argv;
        if(a == nil)
            break;
        fmtprint(&fmt, " ");
        fmtuserstring(&fmt, a, "");
        argv++;
    }
    break;
case FAUTH:
    i[0] = va_arg(list, int);
    a = va_arg(list, char*);
    fmtprint(&fmt, "%d", i[0]);
    fmtuserstring(&fmt, a, "");
    break;
case SEGBRK:
case RENDEZVOUS:
    v = va_arg(list, void*);
    fmtprint(&fmt, "%#p ", v);
    v = va_arg(list, void*);
    fmtprint(&fmt, "%#p", v);
    break;
case OPEN:
    a = va_arg(list, char*);
    fmtuserstring(&fmt, a, " ");
    i[0] = va_arg(list, int);
    fmtprint(&fmt, "%#ux", i[0]);
    break;
case SLEEP:
    l = va_arg(list, long);
    fmtprint(&fmt, "%ld", l);
```

```
    break;
case RFORK:
    i[0] = va_arg(list, int);
    fmtprint(&fmt, "%#ux", i[0]);
    break;
case PIPE:
case BRK:
    v = va_arg(list, int*);
    fmtprint(&fmt, "%#p", v);
    break;
case CREATE:
    a = va_arg(list, char*);
    fmtuserstring(&fmt, a, " ");
    i[0] = va_arg(list, int);
    i[1] = va_arg(list, int);
    fmtprint(&fmt, "%#ux %#ux", i[0], i[1]);
    break;
case FD2PATH:
case FSTAT:
case FWSTAT:
    i[0] = va_arg(list, int);
    a = va_arg(list, char*);
    l = va_arg(list, unsigned long);
    fmtprint(&fmt, "%d %#p %lud", i[0], a, l);
    break;
case NOTIFY:
case SEGDETACH:
    v = va_arg(list, void*);
    fmtprint(&fmt, "%#p", v);
    break;
case SEGATTACH:
    i[0] = va_arg(list, int);
    fmtprint(&fmt, "%d ", i[0]);
    a = va_arg(list, char*);
    fmtuserstring(&fmt, a, " ");
    /*FALLTHROUGH*/
case SEGFREE:
case SEGFLUSH:
    v = va_arg(list, void*);
    l = va_arg(list, unsigned long);
    fmtprint(&fmt, "%#p %lud", v, l);
    break;
case UNMOUNT:
    a = va_arg(list, char*);
    fmtuserstring(&fmt, a, " ");
    a = va_arg(list, char*);
    fmtuserstring(&fmt, a, "");
    break;
case SEMACQUIRE:
case SEMRELEASE:
    v = va_arg(list, int*);
    i[0] = va_arg(list, int);
    fmtprint(&fmt, "%#p %d", v, i[0]);
    break;
case TSEMACQUIRE:
    v = va_arg(list, long*);
    l = va_arg(list, ulong);
    fmtprint(&fmt, "%#p %ld", v, l);
    break;
case SEEK:
```

```
        v = va_arg(list, vlong*);
        i[0] = va_arg(list, int);
        vl = va_arg(list, vlong);
        i[1] = va_arg(list, int);
        fmtprint(&fmt, "%#p %d %#llux %d", v, i[0], vl, i[1]);
        break;
    case FVERSION:
        i[0] = va_arg(list, int);
        i[1] = va_arg(list, int);
        fmtprint(&fmt, "%d %d ", i[0], i[1]);
        a = va_arg(list, char*);
        fmtuserstring(&fmt, a, " ");
        l = va_arg(list, unsigned long);
        fmtprint(&fmt, "%lud", l);
        break;
    case WSTAT:
    case STAT:
        a = va_arg(list, char*);
        fmtuserstring(&fmt, a, " ");
        /*FALLTHROUGH*/
    case ERRSTR:
    case AWAIT:
        a = va_arg(list, char*);
        l = va_arg(list, unsigned long);
        fmtprint(&fmt, "%#p %lud", a, l);
        break;
    case MOUNT:
        i[0] = va_arg(list, int);
        i[1] = va_arg(list, int);
        fmtprint(&fmt, "%d %d ", i[0], i[1]);
        a = va_arg(list, char*);
        fmtuserstring(&fmt, a, " ");
        i[0] = va_arg(list, int);
        fmtprint(&fmt, "%#ux ", i[0]);
        a = va_arg(list, char*);
        fmtuserstring(&fmt, a, "");
        break;
    case PREAD:
        i[0] = va_arg(list, int);
        v = va_arg(list, void*);
        l = va_arg(list, long);
        fmtprint(&fmt, "%d %#p %ld", i[0], v, l);
        if(syscallno == PREAD){
            vl = va_arg(list, vlong);
            fmtprint(&fmt, " %lld", vl);
        }
        break;
    case PWRITE:
        i[0] = va_arg(list, int);
        v = va_arg(list, void*);
        l = va_arg(list, long);
        fmtprint(&fmt, "%d ", i[0]);
        len = MIN(l, 64);
        fmtrwdata(&fmt, v, len, " ");
        fmtprint(&fmt, "%ld", l);
        if(syscallno == PWRITE){
            vl = va_arg(list, vlong);
            fmtprint(&fmt, " %lld", vl);
        }
        break;
```

```
    }

    up->syscalltrace = fmtstrflush(&fmt);
}
```

Uses MIN 47d, false, fmtrwdata() 497, fmtuserstring() 498a, free() 104a, nsyscall 886a, sysctab 885, up, and validaddr() 120b.

⟨*function* sysretfmt 496⟩≡ (884b)

```
void
sysretfmt(int syscallno, va_list list, long ret, uvlong start, uvlong stop)
{
    long l;
    void* v;
    Fmt fmt;
    vlong vl;
    int i, len;
    char *a, *errstr;

    fmtstrinit(&fmt);

    if(up->syscalltrace)
        free(up->syscalltrace);

    errstr = "\"\"";
    switch(syscallno){
    default:
    case ALARM:
    case PWRITE:
        if(ret == -1)
            errstr = up->syserrstr;
        fmtprint(&fmt, " = %ld", ret);
        break;
    case EXEC:
    case SEGBRK:
    case SEGATTACH:
    case RENDEZVOUS:
        if((void *)ret == (void*)-1)
            errstr = up->syserrstr;
        fmtprint(&fmt, " = %#p", (void *)ret);
        break;
    case AWAIT:
        a = va_arg(list, char*);
        l = va_arg(list, unsigned long);
        if(ret > 0){
            fmtuserstring(&fmt, a, " ");
            fmtprint(&fmt, "%lud = %ld", l, ret);
        }
        else{
            fmtprint(&fmt, "%#p/\"\" %lud = %ld", a, l, ret);
            errstr = up->syserrstr;
        }
        break;
    case ERRSTR:
        a = va_arg(list, char*);
        l = va_arg(list, unsigned long);
        if(ret > 0){
            fmtuserstring(&fmt, a, " ");
            fmtprint(&fmt, "%lud = %ld", l, ret);
        }
        else{
```

```
            fmtprint(&fmt, "\"\" %lud = %ld", l, ret);
            errstr = up->syserrstr;
        }
        break;
    case FD2PATH:
        i = va_arg(list, int);
        USED(i);
        a = va_arg(list, char*);
        l = va_arg(list, unsigned long);
        if(ret > 0){
            fmtuserstring(&fmt, a, " ");
            fmtprint(&fmt, "%lud = %ld", l, ret);
        }
        else{
            fmtprint(&fmt, "\"\" %lud = %ld", l, ret);
            errstr = up->syserrstr;
        }
        break;
    case PREAD:
        i = va_arg(list, int);
        USED(i);
        v = va_arg(list, void*);
        l = va_arg(list, long);
        if(ret > 0){
            len = MIN(ret, 64);
            fmtrwdata(&fmt, v, len, "");
        }
        else{
            fmtprint(&fmt, "/\"\"");
            errstr = up->syserrstr;
        }
        fmtprint(&fmt, " %ld", l);
        if(syscallno == PREAD){
            vl = va_arg(list, vlong);
            fmtprint(&fmt, " %lld", vl);
        }
        fmtprint(&fmt, " = %ld", ret);
        break;
    }
    fmtprint(&fmt, " %s %#llud %#llud\n", errstr, start, stop);
    up->syscalltrace = fmtstrflush(&fmt);
}
```

Uses MIN 47d, fmtrwdata() 497, fmtuserstring() 498a, free() 104a, and up.

⟨*function* fmtrwdata 497⟩≡ (884b)

```
// WE ARE OVERRUNNING SOMEHOW
static void
fmtrwdata(Fmt* f, char* a, int n, char* suffix)
{
    int i;
    char *t;

    if(a == nil){
        fmtprint(f, "0x0%s", suffix);
        return;
    }
    validaddr((ulong)a, n, false);
    t = smalloc(n+1);
    for(i = 0; i < n; i++)
        if(a[i] > 0x20 && a[i] < 0x7f)  /* printable ascii? */
```

```
            t[i] = a[i];
        else
            t[i] = '.';

    fmtprint(f, " %#p/\"%s\"%s", a, t, suffix);
    free(t);
}
```

Uses false, free() 104a, smalloc() 104b, and validaddr() 120b.

⟨*function* fmtuserstring 498a⟩≡ (884b)

```
static void
fmtuserstring(Fmt* f, char* a, char* suffix)
{
    int n;
    char *t;

    if(a == nil){
        fmtprint(f, "0/\"\"%s", suffix);
        return;
    }
    validaddr((ulong)a, 1, false);
    n = ((char*)vmemchr(a, 0, 0x7fffffff) - a) + 1;
    t = smalloc(n+1);
    memmove(t, a, n);
    t[n] = 0;
    fmtprint(f, "%#p/\"%s\"%s", a, t, suffix);
    free(t);
}
```

Uses false, free() 104a, smalloc() 104b, validaddr() 120b, and vmemchr() 120c.

19.5 Scheduler events: /proc/trace and /bin/trace

The /proc/trace file provides a stream of scheduler events for processes that have tracing enabled. Each event records a process ID, an event type (SReady, SRun, SDead, SSleep, or SUser), and a timestamp. The events are stored in a circular buffer of Nevents (16384) entries. The tracing hooks are scattered throughout the scheduler: in ready(), runproc(), sleep(), and pexit(). To avoid races, each hook copies the proctrace function pointer to a local variable before calling it—if the pointer were read twice, it could become nil between the check and the call.

⟨Proc *debugger fields* 498b⟩+≡ (66) ◁492b 618a▷

```
bool trace;    /* process being traced? */
```

⟨procctlreq() *CMtrace case* 498c⟩≡ (620b)

```
case CMtrace:
    switch(cb->nf){
    case 1:
        p->trace ^= true;
        break;
    case 2:
        p->trace = (atoi(cb->f[1]) != 0);
        break;
    default:
        error("args");
    }
    break;
```

Uses CMtrace-895 602b and true.

⟨*hook proctrace* 499a⟩≡ (873a)
```
void (*proctrace)(Proc*, /*enum<tevent>*/int, vlong) = 0; // was in devproc.c
```
Uses proctrace 499a.

⟨*enum* Tevent 499b⟩≡ (673a)
```
enum Tevent {
    SReady = 0,  /* runnable but not running  */
    SRun,  /* running best effort */
    SDead,  /* proc dies */
    SSleep,  /* blocked */
    SUser,  /* user event */

    /* real-time extensions */
    ⟨enum Tevent real-time scheduling events 579⟩

    Nevent, // must be last
};
```

⟨*struct* Traceevent 499c⟩≡ (673a)
```
struct Traceevent {
    ulong pid;
    // enum<tevent>
    ulong etype; /* Event type */
    vlong time; /* time stamp  */ // dimension?
};
```

⟨ready() *other locals* 499d⟩≡ (232a)
```
void (*pt)(Proc*, int, vlong);
```

⟨ready() *hook proctrace* 499e⟩≡ (232a)
```
pt = proctrace;
if(pt)
    pt(p, SReady, 0);
```
Uses proctrace 499a.

⟨runproc() *other locals* 499f⟩≡ (225) 656c▷
```
void (*pt)(Proc*, int, vlong);
```

⟨runproc() *hook proctrace* 499g⟩≡ (225)
```
pt = proctrace;
if(pt)
    pt(p, SRun, 0);
```
Uses proctrace 499a.

⟨pexit() *other locals* 499h⟩+≡ (152c) ◁156b 504e▷
```
void (*pt)(Proc*, int, vlong);
```

⟨pexit() *hook proctrace* 499i⟩≡ (152c)
```
pt = proctrace;
if(pt)
    pt(up, SDead, 0);
```
Uses proctrace 499a.

⟨sleep() *other locals* 499j⟩≡ (233e)
```
void (*pt)(Proc*, int, vlong);
```

⟨sleep() *hook proctrace* 499k⟩≡ (233e)
```
pt = proctrace;
if(pt)
    pt(up, SSleep, 0);
```
Uses proctrace 499a.

⟨procctlreq() *CMevent case* 500a⟩≡ (620b)
```
case CMevent:
    pt = proctrace;
    if(up->trace && pt)
        pt(up, SUser, 0);
    break;
```
Uses CMevent-904 602b, proctrace 499a, and up.

⟨*global trace txxx* 500b⟩≡ (740b)
```
// array<Traceevent>
static Traceevent *tevents;
static int topens;
static int tproduced, tconsumed;
static Lock tlock;
```

⟨*function* eventsavailable 500c⟩≡ (740b)
```
static int
eventsavailable(void *)
{
    return tproduced > tconsumed;
}
```
Uses tconsumed-918 500b and tproduced-917 500b.

⟨*function* _proctrace 500d⟩≡ (740b)
```
static void
_proctrace(Proc* p, Tevent etype, vlong ts)
{
    Traceevent *te;

    if (p->trace == false || topens == 0 ||
        tproduced - tconsumed >= Nevents)
        return;

    te = &tevents[tproduced&Emask];
    te->pid = p->pid;
    te->etype = etype;
    if (ts == 0)
        te->time = todget(nil);
    else
        te->time = ts;
    tproduced++;
}
```
Uses Emask-906 500f, Nevents-905 500e, false, tconsumed-918 500b, tevents-915 500b, todget() 262, topens-916 500b, and tproduced-917 500b.

⟨*constant* Nevents 500e⟩≡ (740b)
```
Nevents = 0x4000,
```

⟨*constant* Emask 500f⟩≡ (740b)
```
Emask = Nevents - 1,
```
Uses Nevents-905 500e.

⟨procopen() *Qtrace if* 500g⟩≡ (611)
```
if(QID(c->qid) == Qtrace){
    if (omode != OREAD)
        error(Eperm);
    lock(&tlock);
    if (waserror()){
        unlock(&tlock);
```

```
        nexterror();
    }
    if (topens > 0)
        error("already open");
    topens++;
    if (tevents == nil){
        tevents = (Traceevent*)malloc(Nevents * sizeof(Traceevent));
        if(tevents == nil)
            error(Enomem);
        tproduced = tconsumed = 0;
    }
    proctrace = _proctrace;
    unlock(&tlock);
    poperror();

    c->mode = openmode(omode);
    c->flag |= COPEN;
    c->offset = 0;
    return c;
}
```

Uses COPEN 294b, Nevents-905 500e, OREAD 853i, QID-911 604a, Qtrace-860 602a, _proctrace() 500d, lock() 76d, malloc() 103c, nexterror, openmode() 302b, poperror 666d, proctrace 499a, tconsumed-918 500b, tevents-915 500b, tlock-919 500b, topens-916 500b, tproduced-917 500b, unlock() 78d, and waserror 666d.

⟨procclose() *Qtrace if* 501a⟩≡ (617)

```
if(QID(c->qid) == Qtrace){
    lock(&tlock);
    if(topens > 0)
        topens--;
    if(topens == 0)
        proctrace = nil;
    unlock(&tlock);
}
```

Uses QID-911 604a, Qtrace-860 602a, lock() 76d, proctrace 499a, tlock-919 500b, topens-916 500b, and unlock() 78d.

⟨procread() *Qtrace if* 501b⟩≡ (605d)

```
if(QID(c->qid) == Qtrace){
    if(!eventsavailable(nil))
        return 0;

    rptr = (byte*)va;
    navail = tproduced - tconsumed;
    if(navail > n / sizeof(Traceevent))
        navail = n / sizeof(Traceevent);
    while(navail > 0) {
        ne = ((tconsumed & Emask) + navail > Nevents)?
                Nevents - (tconsumed & Emask): navail;
        memmove(rptr, &tevents[tconsumed & Emask],
                ne * sizeof(Traceevent));

        tconsumed += ne;
        rptr += ne * sizeof(Traceevent);
        navail -= ne;
    }
    return rptr - (byte*)va;
}
```

Uses Emask-906 500f, Nevents-905 500e, QID-911 604a, Qtrace-860 602a, eventsavailable() 500c, tconsumed-918 500b, tevents-915 500b, and tproduced-917 500b.

19.6 /bin/db

The debugger db sets breakpoints by writing a BKPT instruction into the target's text segment via /proc/\emphpid/m (after stopping the process and converting text to data with text2data()). When the process resumes and hits that address, the hardware faults, the trap handler recognizes the BKPT opcode, and a "sys: breakpoint" note is posted. Since the debugger is waiting on the target's ctl file, it wakes up and can inspect the state. This is efficient because the breakpoint check happens in hardware, not on every instruction.

19.7 Other

⟨Procctl *cases* 502a⟩+≡ (484d) ◁491a 876b▷
```
  Proc_traceme,
```

⟨procctl() Proc_traceme *case (and fallthrough* Proc_stopme*)* 502b⟩≡ (485b)
```
  case Proc_traceme:
      if(p->nnote == 0)
          return;
      /* No break */
```
Uses Proc_traceme 502a.

⟨procctlreq() *CMstartstop case* 502c⟩≡ (620b)
```
  case CMstartstop:
      if(p->state != Stopped)
          error(Ebadctl);
      p->procctl = Proc_traceme;
      ready(p);
      procstopwait(p, Proc_traceme);
      break;
```
Uses CMstartstop-891 602b, Proc_traceme 502a, Stopped 484c, procstopwait() 486c, and ready.

Chapter 20

Profiling Support

In this chapter, I will present two forms of profiling support in the kernel: timing and sampling. The timing infrastructure tracks how much CPU time each process spends in user mode vs kernel mode, reported via `/dev/cputime` and `/proc/\emphpid/status`. The sampling infrastructure (`/bin/tprof`) uses a periodic timer to record which instruction the program is executing, building a coverage histogram over the text segment. A third mechanism, `/bin/prof`, requires cooperation between the compiler, linker, and kernel through the `Tos` structure at the top of the user stack.

20.1 /dev/cputime and /proc/<pid>/status

Each process tracks five time counters: user time, system time, real (wall-clock) time, and cumulative user/system time of children. The counters are incremented by `accountime()`, which is called on every clock tick and checks `p->insyscall` to decide whether to attribute the tick to user or system time. The real time is stored as the tick count at process creation, so the actual elapsed time is computed on-the-fly as `CPUS(0)->ticks - p->time[TReal]`.

⟨*enum* `proctimer` 503a⟩≡ (868c)
```
enum Proctime
{
    TUser = 0,    /* Proc.time */
    TSys,
    TReal,

    // to accumulate also the time of the children of the process
    TCUser,
    TCSys,
};
```

⟨`Proc` *stats and profiling fields* 503b⟩≡ (66) 508b▷
```
// hash<enum<proctime>, ulong>
ulong time[5];  /* User, Sys, Real; child U, S */
```

⟨`sysrfork()` *setting time field* 503c⟩≡ (134b)
```
memset(p->time, 0, sizeof(p->time));
p->time[TReal] = CPUS(0)->ticks;
```
Uses `CPUS` 702 and `TReal` 503a.

⟨`kproc()` *setting time field* 503d⟩≡ (157c)
```
memset(p->time, 0, sizeof(p->time));
p->time[TReal] = CPUS(0)->ticks;
```
Uses `CPUS` 702 and `TReal` 503a.

⟨accountime() *update time of current process* 504a⟩≡ (657b)
```
p->time[p->insyscall ? TSys : TUser]++;
```
Uses TSys 503a and TUser 503a.

⟨consread() *cases* 504b⟩+≡ (394c) ◁266a 637▷
```
case Qcputime:
    k = offset;
    if(k >= 5*NUMSIZE)
        return 0;
    if(k+n > 5*NUMSIZE)
        n = 5*NUMSIZE - k;
    /* easiest to format in a separate buffer and copy out */

    for(i=0; i<5 && NUMSIZE*i<k+n; i++){
        l = up->time[i];
        if(i == TReal)
            l = CPUS(0)->ticks - l;
        l = TK2MS(l);
        readnum(0, tmp+NUMSIZE*i, NUMSIZE, l, NUMSIZE);
    }
    memmove(buf, tmp+k, n);
    return n;
```
Uses CPUS 702, NUMSIZE 867a, Qcputime-1784 633a, TK2MS 244d, TReal 503a, readnum() 670c, and up.

⟨procread() *Qstatus case, time part* 504c⟩≡ (605d)
```
for(i = 0; i < 5; i++) {
    l = p->time[i];
    if(i == TReal)
        l = CPUS(0)->ticks - l;
    l = TK2MS(l);
    readnum(0, statbuf+j+NUMSIZE*i, NUMSIZE, l, NUMSIZE);
}
```
Uses CPUS 702, NUMSIZE 867a, TK2MS 244d, TReal 503a, and readnum() 670c.

20.2 /bin/time

The /bin/time command is trivially simple because the kernel already includes timing information in the wait message that a parent receives when a child exits. The Waitmsg carries three values—user, system, and real time—which pexit() computes from the process's counters. The TCUser and TCSys fields accumulate the children's times, so the reported values include all descendants, just like UNIX's /usr/bin/time.

⟨Waitmsg *time field* 504d⟩≡ (149e)
```
ulong time[3];  /* of loved one and descendants */
```

⟨pexit() *other locals* 504e⟩+≡ (152c) ◁499h
```
long utime, stime;
```

⟨pexit() *set wait msg time field* 504f⟩≡ (154c)
```
utime = up->time[TUser] + up->time[TCUser];
stime = up->time[TSys] + up->time[TCSys];

wq->w.time[TUser] = tk2ms(utime);
wq->w.time[TSys] = tk2ms(stime);
wq->w.time[TReal] = tk2ms(CPUS(0)->ticks - up->time[TReal]);
```
Uses CPUS 702, TCSys 503a, TCUser 503a, TReal 503a, TSys 503a, TUser 503a, tk2ms() 245a, and up.

⟨pexit() *update TC time of parent* 505a⟩≡ (154d)
```
p->time[TCUser] += utime;
p->time[TCSys] += stime;
```
Uses TCSys 503a and TCUser 503a.

⟨sysawait() *snprint time field arguments* 505b⟩≡ (150a)
```
w.time[TUser], w.time[TSys], w.time[TReal],
```
Uses TReal 503a, TSys 503a, and TUser 503a.

⟨procread() *Qwait case, snprint time field arguments* 505c⟩≡ (600b)
```
wq->w.time[TUser], wq->w.time[TSys], wq->w.time[TReal],
```
Uses TReal 503a, TSys 503a, and TUser 503a.

20.3 /bin/tprof

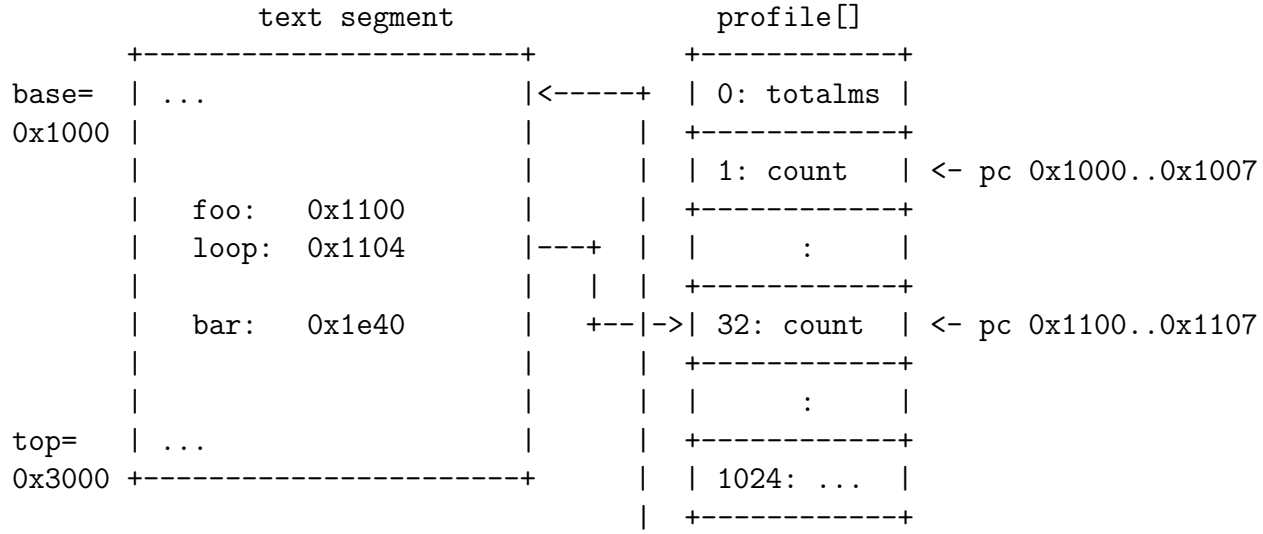
Tick-based profiling (tprof) works by sampling: a periodic timer fires at a frequency deliberately not aligned with HZ (to avoid aliasing), checks what PC the user process is executing, and increments a counter in the profile array attached to the text segment. The array covers the entire text segment with a resolution of $2^3 = 8$ bytes per bucket (LRESPROF). The profile is stored in the segment rather than the process so that multi-threaded programs sharing the same text accumulate a single combined profile. The first element profile[0] holds total execution time in milliseconds; subsequent elements hold per-bucket times. The /bin/tprof tool reads /proc/\emphpid/profile and correlates the buckets with symbol information from the binary.

⟨Segment *other fields* 505d⟩+≡ (113b) ◁488e 555e▷
```
kern_wp  profile;  /* Tick profile area */ // for TSEG only
```

⟨putseg() *free profile* 505e⟩≡ (116c)
```
if(s->profile != nil)
    free(s->profile);
```
Uses free() 104a.

⟨*constant* LRESPROF 505f⟩≡ (855)
```
LRESPROF  = 3,
```

The profile array is a fixed-resolution histogram of the text segment: each bucket covers $2^{\texttt{LRESPROF}} = 8$ bytes of code, and segclock maps a sampled program counter into a bucket by shifting (pc-base) >> LRESPROF. If a process's text segment runs from 0x1000 to 0x3000 (8 KiB), its profile will contain $0x2000/8 = 1024$ buckets, plus bucket 0 which is reserved for "total time" and does not correspond to any instruction:



The 8-byte granularity is a deliberate trade-off. Finer resolution (say, one bucket per instruction) would quadruple the memory overhead and would not add much accuracy, because the sampled PC can land anywhere inside a small function even if the function is really the "hot" spot. Coarser resolution would start merging unrelated functions into the same bucket. Eight bytes matches two ARM instructions—enough to keep individual basic blocks distinct without blowing up the array.

The next subtle point is the choice of sampling frequency. `profclock` is registered as a secondary clock callback via `addclock0link(profclock, 113)`. The `113` is a period in HZ units, and the comment "relative prime to `HZ`" hides one of the more important correctness conditions in a sampling profiler. Suppose `HZ` is `100` and we naively sampled at `100` or `50` or `25`: the profiler would fire exactly in sync with the main system clock, and any code that tends to run right after a clock tick (a handler, a scheduled task) would be massively over-represented—the profiler would systematically "see" the same instructions that the clock tick tends to interrupt. This is aliasing, the same phenomenon that makes a wagon wheel appear to spin backwards in a film. Picking a period that is relatively prime to `HZ` guarantees that, over time, the samples drift across every phase of the main clock, so that no region of code is systematically favoured. `113` is a prime just above `HZ` and is a standard choice in samplers inherited from the BSDs.

⟨`proc_init()` *other init* 506a⟩≡ (604d)
```
addclock0link((void (*)(void))profclock, 113);  /* Relative prime to HZ */
```
Uses `addclock0link()` 253f and `profclock()` 506b.

⟨*clock callback profclock* 506b⟩≡ (740b)
```
static void
profclock(Ureg *ur, Timer *)
{
    if(up == nil || up->state != Running)
        return;

    /* user profiling clock */
    if(arch_userureg(ur)){
        ⟨profclock tos adjustments when user 509c⟩
        segclock(ur->pc);
    }
}
```
Uses `Running` 67c, `segclock()` 506c, and `up`.

⟨*clock callback segclock* 506c⟩≡ (864b)
```
// called via profclock
void
segclock(ulong pc)
{
    Segment *s;

    s = up->seg[TSEG];
    if(s == nil || s->profile == nil)
        return;

    s->profile[0] += TK2MS(1);
    if(pc >= s->base && pc < s->top) {
        pc -= s->base;
        s->profile[pc>>LRESPROF] += TK2MS(1);
    }
}
```
Uses `LRESPROF` 113b, `TK2MS` 244d, `TSEG` 118c, and `up`.

⟨`procctlreq()` *CMprofile case* 506d⟩≡ (620b)
```
case CMprofile:
    s = p->seg[TSEG];
```

```
    if(s == nil || (s->type&SG_TYPE) != SG_TEXT)
        error(Ebadctl);
    if(s->profile != nil)
        free(s->profile);
    npc = (s->top-s->base)>>LRESPROF;
    s->profile = malloc(npc*sizeof(*s->profile));
    if(s->profile == nil)
        error(Enomem);
    break;
```

Uses CMprofile-886 602b, LRESPROF 113b, SG_TEXT, SG_TYPE, TSEG 118c, free() 104a, and malloc() 103c.

⟨procread() *cases* 507a⟩+≡ (605d) ◁492d 600b▷

```
case Qprofile:
    s = p->seg[TSEG];
    if(s == nil || s->profile == nil)
        error("profile is off");
    i = (s->top-s->base)>>LRESPROF;
    i *= sizeof(*s->profile);
    if(offset >= i)
        return 0;
    if(offset+n > i)
        n = i - offset;
    memmove(a, ((char*)s->profile)+offset, n);
    return n;
```

Uses LRESPROF 113b, Qprofile-877 602a, and TSEG 118c.

⟨procgen() *Qprofile case* 507b⟩≡ (614d)

```
case Qprofile:
    q = p->seg[TSEG];
    if(q && q->profile) {
        len = (q->top-q->base)>>LRESPROF;
        len *= sizeof(*q->profile);
    }
    break;
```

Uses LRESPROF 113b, Qprofile-877 602a, and TSEG 118c.

20.4 /bin/prof

Function-level profiling (prof) requires cooperation between the compiler, linker, and kernel. The compiler instruments each function entry with profiling code, and the linker allocates profiling data structures. But making a system call on every function call would be far too slow, so the kernel provides the Tos (Top of Stack) structure mapped at the top of every user stack. This structure gives user code direct read access to cycle counts and a millisecond clock without entering the kernel. The kernel updates tos->kcycles and tos->pcycles on every kernel entry/exit, and tos->clock on every profiling tick, so the profiling library can compute accurate per-function times entirely in user space.

⟨*struct* Tos 507c⟩≡ (673b)

```
struct Tos {
    ⟨Tos prof field 508h⟩
    uvlong cyclefreq; /* cycle clock frequency if there is one, 0 otherwise */

    vlong kcycles; /* cycles spent in kernel */
    vlong pcycles; /* cycles spent in process (kernel + user) */
    ulong clock; // in ms

    ulong pid;  /* might as well put the pid here */
    ⟨Tos other fields 508i⟩
```

```
    /* top of stack is here */
};
```

⟨Cpu *other fields* 508a⟩+≡ (64a) ◁472b
```
// cyclefreq == cpuhz if havetsc, 0 otherwise
uvlong  cyclefreq;    /* Frequency of user readable cycle counter */
```

⟨Proc *stats and profiling fields* 508b⟩+≡ (66) ◁503b
```
uvlong  kentry;   /* Kernel entry time stamp (for profiling) */
/*
 * pcycles: cycles spent in this process (updated on procsave/restore)
 * when this is the current proc and we're in the kernel
 * (procrestores outnumber procsaves by one)
 * the number of cycles spent in the proc is pcycles + cycles()
 * when this is not the current process or we're in user mode
 * (procrestores and procsaves balance), it is pcycles.
 */
vlong pcycles;
```

⟨*macro* arch_cycles*(arm)* 508c⟩≡ (974)
```
#define arch_cycles(ip) *(ip) = arch_lcycles()
```

⟨*type CpSPM secondary registers(arm)* 508d⟩≡ (888c)
```
/*
 * CpSPM Secondary (CRm) registers and opcode2 fields (armv6)
 */
#define CpSPMperf   12          /* various counters */
```

⟨*type CpSPM opcode2(arm)* 508e⟩≡ (888c)
```
#define CpSPMctl    0           /* performance monitor control */
#define CpSPMcyc    1           /* cycle counter register */
```

⟨_startpg() *enable cycle counter(raspberry pi1)(arm)* 508f⟩≡ (272a)
```
/*
 * enable cycle counter
 */
MOVW    $1, R1
MCR CpSC, 0, R1, C(CpSPM), C(CpSPMperf), CpSPMctl
```

⟨*function* arch_lcycles*(raspberry pi1)(arm)* 508g⟩≡ (1012a)
```
TEXT arch_lcycles(SB), 1, $-4
    MRC CpSC, 0, R0, C(CpSPM), C(CpSPMperf), CpSPMcyc
    RET
```

⟨Tos *prof field* 508h⟩≡ (507c)
```
struct   /* Per process profiling */
{
    Plink *pp; /* known to be 0(ptr) */
    Plink *next; /* known to be 4(ptr) */
    Plink *last;
    Plink *first;
    ulong pid;
    ulong what;
} prof;
```

⟨Tos *other fields* 508i⟩≡ (507c)
```
/* scratch space for kernel use (e.g., mips fp delay-slot execution) */
ulong kscr[4];
```

⟨sysexec() *locals* 508j⟩+≡ (141c) ◁321e 510a▷
```
Tos *tos;
```

⟨sysexec() *tos settings* 509a⟩≡ (146b)
```
tos = (Tos*)(TSTKTOP - sizeof(Tos));

tos->cyclefreq = cpu->cyclefreq;
arch_cycles((uvlong*)&tos->pcycles);
tos->pcycles = -tos->pcycles; // see comment above on Proc->pcycle
tos->kcycles = tos->pcycles;
tos->clock = 0;
// what about other fields? like pid? will be set in kexit! but could be
// done here? what about sysrfork? call kexit?
```

⟨arch__kexit() *tos adjustments* 509b⟩≡ (169)
```
uvlong t;
Tos *tos;

/* precise time accounting, kernel exit */
tos = (Tos*)(USTKTOP-sizeof(Tos));
arch_cycles(&t);
tos->kcycles += t - up->kentry;
tos->pcycles = up->pcycles;
tos->pid = up->pid;
```
Uses up.

⟨profclock *tos adjustments when user* 509c⟩≡ (506b)
```
Tos *tos;
tos = (Tos*)(USTKTOP-sizeof(Tos));
tos->clock += TK2MS(1);
```
Uses TK2MS 244d.

⟨syscall() *adjust kentry* 509d⟩≡ (163e)
```
arch_cycles(&up->kentry);
```

⟨trap() *adjust kentry when interrupt user* 509e⟩≡ (168b)
```
arch_cycles(&up->kentry);
```

⟨arch_procsave() *cycles adjustments* 509f⟩≡ (230c)
```
uvlong t;

arch_cycles(&t);
p->pcycles += t;
```

⟨arch_procrestore *cycles adjustments* 509g⟩≡ (231a)
```
arch_cycles(&t);
p->pcycles -= t;
```

⟨sysexec() *nbytes tos adjustments* 509h⟩≡ (145a)
```
nbytes += sizeof(Tos); /* hole for profiling clock at top of stack (and more) */
```

Chapter 21

Userspace System Programs

This short chapter covers the handful of programs that the kernel needs to get from bare hardware to a running shell. These programs are compiled into the kernel image itself (via the root filesystem) so they are available before any external storage is mounted. The most important mechanism here is the "sharpbang" (`#!`) support in `sysexec()`, which is needed because the boot script `boot.rc` is an `rc` script, and `sysexec()` must recognize the `#!` header to invoke the interpreter.

21.1 /boot/boot

21.2 /root/init

21.3 The shell

21.3.1 #!

When `sysexec()` reads the file header and finds `#!` instead of a valid binary magic number, it parses the interpreter path and arguments from the first line with `shargs()`, then restarts the exec with the interpreter as the target and the original script as an argument. The `indir` flag prevents recursive sharpbang processing—you cannot have a script whose interpreter is itself a script.

⟨sysexec() *locals* 510a⟩+≡ (141c) ◁508j

```
char line[sizeof(Exec)];
char *progarg[sizeof(Exec)/2+1];
char progelem[64];

bool indir = false;
```

⟨sysexec() *process sharpbang* 510b⟩≡ (141c)

```
/*
 * Process #! /bin/sh args ...
 */
memmove(line, &exec, sizeof(Exec));
if(indir || line[0]!='#' || line[1]!='!')
    error(Ebadexec);
n = shargs(line, n, progarg);
if(n == 0)
    error(Ebadexec);
indir = true;
/*
 * First arg becomes complete file name
 */
```

```
progarg[n++] = file;
progarg[n] = 0;
validaddr(arg[1], BY2WD, true);
arg[1] += BY2WD;
file = progarg[0];
if(strlen(elem) >= sizeof progelem)
    error(Ebadexec);
strcpy(progelem, elem);
progarg[0] = progelem;
poperror();
cclose(tc);
```

⟨*function* shargs 511a⟩≡ (881b)

```
int
shargs(char *s, int n, char **ap)
{
    int i;

    s += 2;
    n -= 2;     /* skip #! */
    for(i=0; s[i]!='\n'; i++)
        if(i == n-1)
            return 0;
    s[i] = 0;
    *ap = 0;
    i = 0;
    for(;;) {
        while(*s==' ' || *s=='\t')
            s++;
        if(*s == 0)
            break;
        i++;
        *ap++ = s;
        *ap = 0;
        while(*s && *s!=' ' && *s!='\t')
            s++;
        if(*s == 0)
            break;
        else
            *s++ = 0;
    }
    return i;
}
```

⟨sysexec() *if indir arg adjustments* 511b⟩≡ (145a)

```
if(indir){
    argp = progarg;
    while(*argp){
        a = *argp++;
        nbytes += strlen(a) + 1;
        nargs++;
    }
}
```

⟨sysexec() *if indir argp adjustments* 511c⟩≡ (146b)

```
if(indir)
    argp = progarg;
```

⟨sysexec() *if indir argp adjustments again* 512⟩≡ (146b)

```
if(indir && *argp==nil) {
    indir = false;
    argp = (char**)arg[1];
}
```

Chapter 22

Advanced Topics

In this chapter, I cover the remaining advanced aspects of the kernel: floating point management, CPU caches, the memory model, SMP support for the Raspberry Pi 2, real-time scheduling, and the 9P fileserver protocol. These topics were deferred from earlier chapters because they add complexity on top of the core mechanisms already presented.

22.1 Floating point

22.1.1 `FPSaveStatus`

The kernel uses lazy floating point context switching: the FPU is kept disabled by default, and only enabled when a process actually executes a floating point instruction. This avoids the cost of saving and restoring 32 double-precision registers on every context switch for processes that never use floating point. The `FPSaveStatus` enum tracks where each process stands in this lifecycle: `FPinit` means the process has never used the FPU, `FPactive` means the FPU is currently on for this process, and `FPinactive` means the FPU state has been saved and the hardware disabled. The `FPillegal` bit is or'd in during note (signal) handling to prevent note handlers from corrupting the interrupted code's FPU state.

⟨*enum* `fpsavestatus` 513a⟩≡ (868c)

```
/*
 * FPsave.status
 */
enum FPSaveStatus
{
    /* this is a state */
    FPinit=   0,

    FPactive= 1,
    FPinactive= 2,

    /* the following is a bit that can be or'd into the state */
    FPillegal=  0x100,
};
```

Each process carries an `fpstate` field that the scheduler hooks consult to decide whether FPU registers need saving or restoring. New processes start in `FPinit`, and a forked child inherits the parent's state minus the `FPillegal` bit so it is not penalized for something the parent did in a note handler.

⟨`Proc` *other fields* 513b⟩+≡ (66) ◁307c 514d▷

```
// enum<FPSaveStatus>
int fpstate;
```

⟨`newproc()` *set fields of* `p` 513c⟩+≡ (130a) ◁450c 564d▷

```
⟨newproc() fp init 514a⟩
```

⟨newproc() *fp init* 514a⟩≡ (513c)
```
p->fpstate = FPinit;
```
Uses FPinit 513a.

⟨sysrfork() *when* RFPROC *propagate fields from up to p* 514b⟩+≡ (134b) ◁487e 514e▷
⟨sysrfork() *propagate fpstate* 514c⟩

⟨sysrfork() *propagate fpstate* 514c⟩≡ (514b)
```
/* don't penalize the child, it hasn't done FP in a note handler. */
p->fpstate = up->fpstate & ~FPillegal;
```
Uses FPillegal 513a and up.

22.1.2 Arch_FPsave (ARM)

⟨Proc *other fields* 514d⟩+≡ (66) ◁513b 597c▷
```
Arch_FPsave  fpsave;   /* address of this is known by db */
```

⟨sysrfork() *when* RFPROC *propagate fields from up to p* 514e⟩+≡ (134b) ◁514b 601d▷
⟨sysrfork() *propagate fpsave* 514f⟩

⟨sysrfork() *propagate fpsave* 514f⟩≡ (514e)
```
p->fpsave = up->fpsave;
```
Uses up.

⟨*struct* Arch_FPsave*(arm)* 514g⟩≡ (1008a)
```
/*
 * emulated or vfp3 floating point
 */
struct Arch_FPsave
{
    /*
     * vfp3 with ieee fp regs; uvlong is sufficient for hardware but
     * each must be able to hold an Internal from fpi.h for sw emulation.
     */
    ulong   regs[Maxfpregs][3];
    ⟨Proc Arch FPsave other fields 514h⟩
};
```

⟨Proc Arch FPsave *other fields* 514h⟩≡ (514g)
```
int fpstate; // vs Proc.fpstate??

ulong   status;
ulong   control;

uintptr pc;     /* of failed fp instr. */
```

⟨*enum* _anon_*(*(arch/arm/dat_processes.h)(arm)*)* 514i⟩≡ (1008a)
```
enum {
    Maxfpregs   = 32,   /* could be 16 or 32, see Mach.fpnregs */
    Nfpctlregs  = 16,
};
```

22.1.3 Floating point hooks

The lazy FPU scheme plugs into the scheduler through a set of hooks called at key process lifecycle points. The flow is:

```
procsave()     --> fpuprocsave()        save FPU if FPactive, mark FPinactive
procrestore()  --> fpuprocrestore()    (no-op: restored lazily on fault)
sysrfork()     --> fpusysrfork()       save parent's FPU before fork
forkchild()    --> fpusysrforkchild()  copy state to child
sysexec()      --> fpusysprocsetup()   reset to FPinit, disable FPU
```

The key insight is that [fpuprocrestore] is intentionally empty. Rather than eagerly restoring FPU state when a process is rescheduled, the kernel leaves the FPU disabled. If the process tries to use floating point, it triggers an "undefined instruction" trap, and [fpuemu] restores the state at that point. This means processes that were using FPU before being descheduled but happen not to use it in their next quantum pay zero cost.

⟨`arch_forkchild()` *floating point(arm)* 515a⟩≡ (135e)
```
fpusysrforkchild(p, cureg, up);
```
Uses `fpusysrforkchild()` 515b and `up`.

⟨*function* `fpusysrforkchild`*(arm)* 515b⟩≡ (1008c)
```
/*
 * Called later in sysrfork() via the machine-dependent
 * sysrforkchild() routine.
 * Copy the parent FPU state to the child.
 */
void
fpusysrforkchild(Proc *p, Ureg *, Proc *up)
{
    /* don't penalize the child, it hasn't done FP in a note handler. */
    p->fpstate = up->fpstate & ~FPillegal;
}
```
Uses `FPillegal` 513a.

⟨`arch_procsetup()` *floating point(arm)* 515c⟩≡ (143e)
```
fpusysprocsetup(p);
```
Uses `fpusysprocsetup()` 515d.

⟨*function* `fpusysprocsetup`*(arm)* 515d⟩≡ (1008c)
```
/*
 * Disable the FPU.
 * Called from sysexec() via sysprocsetup() to
 * set the FPU for the new process.
 */
void
fpusysprocsetup(Proc *p)
{
    p->fpstate = FPinit;
    fpoff();
}
```
Uses `fpoff()` 523a.

⟨`arch_procsave()` *floating point(arm)* 515e⟩≡ (230c)
```
/* TODO: save and restore VFPv3 FP state once 5[cal] know the new registers.*/
fpuprocsave(p);
```
Uses `fpuprocsave()` 516a.

When a process is about to be descheduled, [fpuprocsave] saves the FPU registers only if the process was actively using them. A dying process (`Moribund`) gets its FPU cleared outright since its state will never be needed again. The subtle point in the comment is that [fpsave] cannot handle pending unmasked exceptions here because [postnote] requires the process lock, which [sleep] may already hold. Instead, exception handling is deferred until the process runs again and triggers a new FPU fault.

⟨*function* `fpuprocsave` *(arm)* 516a⟩≡ (1008c)

```
/*
 * Called from sched() and sleep() via the machine-dependent
 * procsave() routine.
 * About to go in to the scheduler.
 * If the process wasn't using the FPU
 * there's nothing to do.
 */
void
fpuprocsave(Proc *p)
{
    if(p->fpstate == FPactive){
        if(p->state == Moribund)
            fpclear();
        else{
            /*
             * Fpsave() stores without handling pending
             * unmasked exeptions. Postnote() can't be called
             * here as sleep() already has up->rlock, so
             * the handling of pending exceptions is delayed
             * until the process runs again and generates an
             * emulation fault to activate the FPU.
             */
            fpsave(&p->fpsave);
        }
        p->fpstate = FPinactive;
    }
}
```

Uses `FPactive` 513a, `FPinactive` 513a, `Moribund` 153a, and `fpclear()` 524d.

⟨`arch_procrestore()` *floating point(arm)* 516b⟩≡ (231a)

```
fpuprocrestore(p);
```

Uses `fpuprocrestore()` 516c.

⟨*function* `fpuprocrestore` *(arm)* 516c⟩≡ (1008c)

```
/*
 * The process has been rescheduled and is about to run.
 * Nothing to do here right now. If the process tries to use
 * the FPU again it will cause a Device Not Available
 * exception and the state will then be restored.
 */
void
fpuprocrestore(Proc *)
{
}
```

⟨`syscall()` *if RFORK, handle floating point(arm)* 516d⟩≡ (163e)

```
if(scallnr == RFORK)
    fpusysrfork(ureg);
```

Uses `fpusysrfork()` 517a.

⟨*function* `fpusysrfork`*(arm)* 517a⟩≡ (1008c)

```
/*
 * Called early in the non-interruptible path of
 * sysrfork() via the machine-dependent syscall() routine.
 * Save the state so that it can be easily copied
 * to the child process later.
 */
void
fpusysrfork(Ureg*)
{
    if(up->fpstate == FPactive){
        fpsave(&up->fpsave);
        up->fpstate = FPinactive;
    }
}
```

Uses `FPactive` 513a, `FPinactive` 513a, and `up`.

The actual register save loop iterates over all FPU registers (16 for VFPv2, 32 for VFPv3), reading each via [fpsavereg] which dynamically constructs a VSTR instruction. The FPSCR (floating point status and control register) is saved as both `status` and `control` so that pending exception flags and configuration bits are preserved separately.

⟨*function* `fpsave`*(arm)* 517b⟩≡ (1008c)

```
/* should only be called if p->fpstate == FPactive */
void
fpsave(Arch_FPsave *fps)
{
    int n;

    fpon();
    fps->control = fps->status = fprd(Fpscr);
    assert(cpu->fpnregs);
    for (n = 0; n < cpu->fpnregs; n++)
        fpsavereg(n, (uvlong *)fps->regs[n]);
    fpoff();
}
```

Uses `Fpscr-2564` 521a, `assert` 853f, `fpoff()` 523a, `fpon()` 524c, and `fprd()` 519b.

Note handlers are forbidden from using the FPU because their execution is interposed between the interrupted code and its eventual resumption. If a note handler used floating point, it would silently corrupt the interrupted code's FPU registers. The `FPillegal` bit enforces this: [fpunotify] sets it before delivering the note, and if the handler tries to execute an FP instruction, [fpuemu] will see the bit and post an error note. When the handler finishes, [fpunoted] clears the bit.

⟨*function* `fpunotify`*(arm)* 517c⟩≡ (1008c)

```
/*
 * Called when a note is about to be delivered to a
 * user process, usually at the end of a system call.
 * Note handlers are not allowed to use the FPU so
 * the state is marked (after saving if necessary) and
 * checked in the Device Not Available handler.
 */
void
fpunotify(Ureg*)
{
    if(up->fpstate == FPactive){
        fpsave(&up->fpsave);
        up->fpstate = FPinactive;
    }
    up->fpstate |= FPillegal;
```

```
}
```

Uses FPactive 513a, FPillegal 513a, FPinactive 513a, and up.

⟨*function* fpunoted*(arm)* 518a⟩≡ (1008c)

```
/*
 * Called from sysnoted() via the machine-dependent
 * noted() routine.
 * Clear the flag set above in fpunotify().
 */
void
fpunoted(void)
{
    up->fpstate &= ~FPillegal;
}
```

Uses FPillegal 513a and up.

22.1.4 ARM floating point coprocessors (ARM)

On ARM, floating point is accessed through coprocessors 10 (single precision) and 11 (double precision). The kernel detects whether the hardware implements VFPv2 (16 registers) or VFPv3 (32 registers) and records this in `cpu->fpnregs`. This fork of Plan 9 relies on the compiler (`5l -f`) generating native VFP3 instructions, so the old software FPU emulation code (fpiarm.c) has been removed.

⟨Coprocessors *other cases(arm)* 518b⟩≡ (39c)

```
#define CpFP        10          /* float FP, VFP cfg. */
#define CpDFP       11          /* double FP */
```

⟨*enum* _anon_*(*(processes/arm/vfp3.c)(arm)*)* 518c⟩≡ (1008c)

```
/* subarchitecture code in cpu->havefp */
enum VFPKind {
    VFPv2   = 2,
    VFPv3   = 3,
};
```

⟨Cpu Arch *float fields(arm)* 518d⟩≡ (64e) 518e▷

```
// option<VFPKind>, None = 0 /* vfp2 or vfp3 fpu */
int havefp;
bool havefpvalid;
// 16 for VFPv2, 32 for VFPv3
int fpnregs;
```

⟨Cpu Arch *float fields(arm)* 518e⟩+≡ (64e) ◁518d 518f▷

```
bool fpon;
```

⟨Cpu Arch *float fields(arm)* 518f⟩+≡ (64e) ◁518e 524a▷

```
ulong   fpscr;          /* sw copy */
```

22.1.5 coproc.c (ARM)

The ARM architecture does not provide general-purpose instructions to read and write individual VFP registers by index. Instead, the kernel uses a remarkable trick: it constructs a VMRS or VMSR instruction at runtime by writing the machine code into a local array, then calls that array as a function pointer. The `setupfpctlop` function patches the register number into the instruction encoding, flushes the data cache so the new instruction

is visible to the instruction fetcher, and invalidates the instruction cache. This self-modifying-code technique avoids needing 32 separate save/restore functions.

⟨*function* `setupfpctlop`*(arm)* 519a⟩≡ (892c)
```
/* fp coproc control */
static void
setupfpctlop(ulong instr[2], int opcode, int fpctlreg)
{
    ulong instrsz[2];

    fpctlreg &= Nfpctlregs - 1;
    instr[0] = opcode | fpctlreg << 16 | 0 << 12 | CpFP << 8;
    instr[1] = RETinst;

    ⟨setupfpctlop() write back cache for modified instr(arm) 532a⟩
    ⟨setupfpctlop() invalidate instruction cache(arm) 532c⟩
}
```
Uses `CpFP` 518b, `Nfpctlregs` 514i, and `RETinst-2383` 890c.

⟨*function* `fprd`*(arm)* 519b⟩≡ (892c)
```
ulong
fprd(int fpreg)
{
    int s, r;
    volatile ulong instr[2];
    Pufv fp;

    if (!cpu->fpon) {
        arch_dumpstack();
        panic("fprd: cpu%d fpu off", cpu->cpuno);
    }
    s = arch_splhi();
    /*
     * VMRS.  return value will be in R0, which is convenient.
     * Rt will be R0.
     */
    setupfpctlop(instr, 0xeef00010, fpreg);
    fp = (Pufv)instr;
    r = fp();
    arch_splx(s);
    return r;
}
```
Uses `arch_dumpstack` and `setupfpctlop()` 519a.

⟨*function* `fpwr`*(arm)* 519c⟩≡ (892c)
```
void
fpwr(int fpreg, ulong val)
{
    int s;
    volatile ulong instr[2];
    Pvfu fp;

    /* fpu might be off and this VMSR might enable it */
    s = arch_splhi();
    setupfpctlop(instr, 0xeee00010, fpreg);     /* VMSR, Rt is R0 */
    fp = (Pvfu)instr;
    fp(val);
    arch_coherence();
    arch_splx(s);
}
```
Uses `arch_coherence` and `setupfpctlop()` 519a.

⟨*function* setupfpop*(arm)* 520a⟩≡ (892c)

```
/* fp register access; don't bother with single precision */
static void
setupfpop(ulong instr[2], int opcode, int fpreg)
{
    ulong instrsz[2];

    instr[0] = opcode | 0 << 16 | (fpreg & (16 - 1)) << 12;
    if (fpreg >= 16)
        instr[0] |= 1 << 22;        /* high bit of dfp reg # */
    instr[1] = RETinst;

    ⟨setupfpop() write back cache for modified instr(arm) 532b⟩
    ⟨setupfpop() invalidate instruction cache(arm) 532d⟩
}
```

Uses RETinst-2383 890c.

⟨*function* fprestreg*(arm)* 520b⟩≡ (892c)

```
void
fprestreg(int fpreg, uvlong val)
{
    int s;
    volatile ulong instr[2];
    void (*fp)(uvlong *);

    if (!cpu->fpon)
        panic("fprestreg: cpu%d fpu off", cpu->cpuno);
    s = arch_splhi();
    setupfpop(instr, 0xed100000 | CpDFP << 8, fpreg); /* VLDR, Rt is R0 */
    fp = (void (*)(uvlong *))instr;
    fp(&val);
    arch_coherence();
    arch_splx(s);
}
```

Uses CpDFP 518b, arch_coherence, and setupfpop() 520a.

⟨*function* fpsavereg*(arm)* 520c⟩≡ (892c)

```
ulong
fpsavereg(int fpreg, uvlong *fpp)
{
    int s, r;
    volatile ulong instr[2];
    ulong (*fp)(uvlong *);

    if (!cpu->fpon)
        panic("fpsavereg: cpu%d fpu off", cpu->cpuno);
    s = arch_splhi();
    /*
     * VSTR.  pointer will be in R0, which is convenient.
     * Rt will be R0.
     */
    setupfpop(instr, 0xed000000 | CpDFP << 8, fpreg);
    fp = (ulong (*)(uvlong *))instr;
    r = fp(fpp);
    arch_splx(s);
    arch_coherence();
    return r;           /* not too meaningful */
}
```

22.1.6 vfp.c (ARM)

This subsection contains the VFP3 initialization and configuration code. The key function is [havefp], which probes the coprocessor access control register to determine what floating point hardware is present, then reads the FPSID register to distinguish VFPv2 from VFPv3. The [fpon]/[fpoff] functions enable and disable the FPU by writing the Fpexc (exception) register, while [fpcfg] sets the default FPSCR configuration: denormals-are-zero (Dn), flush-to-zero (Fz), and round-to-nearest (FPRNR).

⟨*enum* _anon_*(*(processes/arm/vfp3.c)2(arm)*)* 521a⟩≡ (1008c)
```
/* fp control regs.  most are read-only */
enum {
    Fpsid = 0,
    Fpscr = 1,          /* rw */
    Mvfr1 = 6,
    Mvfr0 = 7,
    Fpexc = 8,          /* rw */
    Fpinst= 9,          /* optional, for exceptions */
    Fpinst2=10,
};
```

⟨*enum* _anon_*(*(processes/arm/vfp3.c)3(arm)*)* 521b⟩≡ (1008c)
```
enum {
    /* Fpexc bits */
    Fpex =      1u << 31,
    Fpenabled = 1 << 30,
    Fpdex =     1 << 29,    /* defined synch exception */
    /* FSR bits appear here */
    Fpmbc =     Fpdex,      /* bits exception handler must clear */

    /* Fpscr bits; see u.h for more */
    Stride =    MASK(2) << 20,
    Len =       MASK(3) << 16,
    Dn=     1 << 25,
    Fz=     1 << 24,
    /* trap exception enables (not allowed in vfp3) */
    FPIDNRM =   1 << 15,    /* input denormal */
    Alltraps = FPIDNRM | FPINEX | FPUNFL | FPOVFL | FPZDIV | FPINVAL,
    /* pending exceptions */
    FPAIDNRM =  1 << 7,     /* input denormal */
    Allexc = FPAIDNRM | FPAINEX | FPAUNFL | FPAOVFL | FPAZDIV | FPAINVAL,
    /* condition codes */
    Allcc =     MASK(4) << 28,
};
```
Uses FPAIDNRM-2580 521b, FPIDNRM-2578 521b, Fpdex-2572 521b, and MASK 47c.

⟨*enum* _anon_*(*(processes/arm/vfp3.c)4(arm)*)* 521c⟩≡ (1008c)
```
enum {
    /* CpCPaccess bits */
    Cpaccnosimd =   1u << 31,
    Cpaccd16 =  1 << 30,
};
```

⟨*function* subarch*(arm)* 521d⟩≡ (1008c)
```
static char *
subarch(int impl, uint sa)
{
    static char *armarchs[] = {
        "VFPv1 (unsupported)",
        "VFPv2",
        "VFPv3+ with common VFP subarch v2",
```

```
        "VFPv3+ with null subarch",
        "VFPv3+ with common VFP subarch v3",
    };

    if (impl != 'A' || sa >= nelem(armarchs))
        return "GOK";
    else
        return armarchs[sa];
}
```

Uses nelem 853i.

⟨*function* implement*(arm)* 522a⟩≡ (1008c)

```
static char *
implement(uchar impl)
{
    if (impl == 'A')
        return "arm";
    else
        return "unknown";
}
```

⟨CpCONTROL.opcode2 *other cases(arm)* 522b⟩≡ (40a) 539d ▷

```
#define CpCPaccess  2
```

⟨*function* havefp*(arm)* 522c⟩≡ (1008c)

```
static int
havefp(void)
{
    int gotfp;
    ulong acc, sid;

    if (cpu->havefpvalid)
        return cpu->havefp;

    cpu->havefp = 0;
    gotfp = 1 << CpFP | 1 << CpDFP;
    cpwrsc(0, CpCONTROL, 0, CpCPaccess, MASK(28));
    acc = cprdsc(0, CpCONTROL, 0, CpCPaccess);
    if ((acc & (MASK(2) << (2*CpFP))) == 0) {
        gotfp &= ~(1 << CpFP);
        print("fpon: no single FP coprocessor\n");
    }
    if ((acc & (MASK(2) << (2*CpDFP))) == 0) {
        gotfp &= ~(1 << CpDFP);
        print("fpon: no double FP coprocessor\n");
    }
    if (!gotfp) {
        print("fpon: no FP coprocessors\n");
        cpu->havefpvalid = true;
        return 0;
    }
    cpu->fpon = true;          /* don't panic */
    sid = fprd(Fpsid);
    cpu->fpon = false;

    switch((sid >> 16) & MASK(7)){
    case 0:             /* VFPv1 */
        break;
    case 1:             /* VFPv2 */
        cpu->havefp = VFPv2;
```

```
            cpu->fpnregs = 16;
            break;
        default:            /* VFPv3 or later */
            cpu->havefp = VFPv3;
            cpu->fpnregs = (acc & Cpaccd16) ? 16 : 32;
            break;
        }
        if (cpu->cpuno == 0)
            print("fp: %d registers, %s simd\n", cpu->fpnregs,
                (acc & Cpaccnosimd? " no": ""));
        cpu->havefpvalid = true;
        return 1;
    }
```

Uses CpCONTROL 39d, CpCPaccess 522b, CpDFP 518b, CpFP 518b, Cpaccd16-2584 521c, Cpaccnosimd-2583 521c, Fpsid-2563 521a, MASK 47c, VFPv2-2561 518c, VFPv3-2562 518c, cprdsc() 892a, cpwrsc() 892b, false, fprd() 519b, print, and true.

⟨*function* fpoff*(arm)* 523a⟩≡ (1008c)

```
    /*
     * these can be called to turn the fpu on or off for user procs,
     * not just at system start up or shutdown.
     */
    void
    fpoff(void)
    {
        if (cpu->fpon) {
            fpwr(Fpexc, 0);
            cpu->fpon = false;
        }
    }
```

Uses Fpexc-2567 521a, false, and fpwr() 519c.

⟨*function* fponly*(arm)* 523b⟩≡ (1008c)

```
    void
    fponly(void)
    {
        if (!cpu->fpon && havefp()) {
            /* enable fp.  must be first operation on the FPUs. */
            fpwr(Fpexc, Fpenabled);
            cpu->fpon = true;
        }
    }
```

Uses Fpenabled-2571 521b, Fpexc-2567 521a, fpwr() 519c, havefp() 522c, and true.

⟨*function* fpcfg*(arm)* 523c⟩≡ (1008c)

```
    static void
    fpcfg(void)
    {
        int impl;
        ulong sid;
        static int printed;

        /* clear pending exceptions; no traps in vfp3; all v7 ops are scalar */
        cpu->fpscr = Dn | Fz | FPRNR | (FPINVAL | FPZDIV | FPOVFL) & ~Alltraps;
        fpwr(Fpscr, cpu->fpscr);
        cpu->fpconfiged = true;

        if (printed)
            return;
        sid = fprd(Fpsid);
```

```
        impl = sid >> 24;
        print("fp: %s arch %s; rev %ld\n", implement(impl),
            subarch(impl, (sid >> 16) & MASK(7)), sid & MASK(4));
        printed = 1;
    }
```

Uses Alltraps-2579 521b, Dn-2576 521b, Fpscr-2564 521a, Fpsid-2563 521a, Fz-2577 521b, MASK 47c, fprd() 519b, fpwr() 519c, implement() 522a, print, subarch() 521d, and true.

⟨Cpu Arch *float fields(arm)* 524a⟩+≡ (64e) ◁518f 525a▷

```
    bool fpconfiged;
```

⟨*function* fpinit*(arm)* 524b⟩≡ (1008c)

```
    void
    fpinit(void)
    {
        if (havefp()) {
            fpononly();
            fpcfg();
        }
    }
```

Uses fpcfg() 523c, fpononly() 523b, and havefp() 522c.

⟨*function* fpon*(arm)* 524c⟩≡ (1008c)

```
    void
    fpon(void)
    {
        if (havefp()) {
          fpononly();
            if (cpu->fpconfiged)
                fpwr(Fpscr, (fprd(Fpscr) & Allcc) | cpu->fpscr);
            else
                fpcfg();    /* 1st time on this fpu; configure it */
        }
    }
```

Uses Allcc-2582 521b, Fpscr-2564 521a, fpcfg() 523c, fpononly() 523b, fprd() 519b, fpwr() 519c, and havefp() 522c.

⟨*function* fpclear*(arm)* 524d⟩≡ (1008c)

```
    void
    fpclear(void)
    {
    //  ulong scr;

        fpon();
    //  scr = fprd(Fpscr);
    //  cpu->fpscr = scr & ~Allexc;
    //  fpwr(Fpscr, cpu->fpscr);

        fpwr(Fpexc, fprd(Fpexc) & ~Fpmbc);
    }
```

Uses Fpexc-2567 521a, Fpmbc-2573 521b, fpon() 524c, fprd() 519b, and fpwr() 519c.

⟨*function* fpstuck*(arm)* 524e⟩≡ (1008c)

```
    void
    fpstuck(uintptr pc)
    {
        if (cpu->fppc == pc && cpu->fppid == up->pid) {
            cpu->fpcnt++;
            if (cpu->fpcnt > 4)
                panic("fpuemu: cpu%d stuck at pid %ld %s pc %#p "
```

```
                "instr %#8.8lux", cpu->cpuno, up->pid, up->text,
                pc, *(ulong *)pc);
        } else {
            cpu->fppid = up->pid;
            cpu->fppc = pc;
            cpu->fpcnt = 0;
        }
    }
```

Uses up.

⟨Cpu Arch *float fields(arm)* 525a⟩+≡ (64e) ◁524a

```
    int fppid;          /* pid of last fault */
    uintptr fppc;       /* addr of last fault */
    int fpcnt;          /* how many consecutive at that addr */
```

⟨*enum* _anon_*(*(processes/arm/vfp3.c)5(arm)*)* 525b⟩≡ (1008c)

```
    enum {
        N = 1<<31,
        Z = 1<<30,
        C = 1<<29,
        V = 1<<28,
        REGPC = 15,
    };
```

⟨*function* condok*(arm)* 525c⟩≡ (1008c)

```
    static int
    condok(int cc, int c)
    {
        switch(c){
        case 0: /* Z set */
            return cc&Z;
        case 1: /* Z clear */
            return (cc&Z) == 0;
        case 2: /* C set */
            return cc&C;
        case 3: /* C clear */
            return (cc&C) == 0;
        case 4: /* N set */
            return cc&N;
        case 5: /* N clear */
            return (cc&N) == 0;
        case 6: /* V set */
            return cc&V;
        case 7: /* V clear */
            return (cc&V) == 0;
        case 8: /* C set and Z clear */
            return cc&C && (cc&Z) == 0;
        case 9: /* C clear or Z set */
            return (cc&C) == 0 || cc&Z;
        case 10:    /* N set and V set, or N clear and V clear */
            return (~cc&(N|V))==0 || (cc&(N|V)) == 0;
        case 11:    /* N set and V clear, or N clear and V set */
            return (cc&(N|V))==N || (cc&(N|V))==V;
        case 12:    /* Z clear, and either N set and V set or N clear and V clear */
            return (cc&Z) == 0 && ((~cc&(N|V))==0 || (cc&(N|V))==0);
        case 13:    /* Z set, or N set and V clear or N clear and V set */
            return (cc&Z) || (cc&(N|V))==N || (cc&(N|V))==V;
        case 14:    /* always */
            return 1;
        case 15:    /* never (reserved) */
```

```
        return 0;
    }
    return 0;   /* not reached */
}
```

Uses C-2587 525b, N-2585 525b, V-2588 525b, and Z-2586 525b.

⟨*macro* ISCPOP*(arm)* 526a⟩≡ (1008c)

```
#define ISCPOP(op)  ((op) == 0xE || ((op) & ~1) == 0xC)
```

⟨*macro* ISVFPOP*(arm)* 526b⟩≡ (1008c)

```
#define ISVFPOP(cp, op) (((cp) == CpDFP || (cp) == CpFP) && ISCPOP(op))
```

When a user process executes a floating point instruction with the FPU disabled, the CPU raises an "undefined instruction" exception. The trap handler calls [fpuemu], which is the entry point for the lazy FPU scheme. It checks whether the faulting instruction is actually a VFP operation (using ISVFPOP), and if so calls [mathemu] to enable the FPU and let the instruction be retried. The [fpstuck] function is a debugging safeguard: if the same process faults repeatedly at the same PC, something is wrong and the kernel panics rather than looping forever.

⟨*function* fpuemu*(arm)* 526c⟩≡ (1008c)

```
/* only called to deal with user-mode instruction faults */
int
fpuemu(Ureg* ureg)
{
    int s, nfp, cop, op;
    uintptr pc;

    if(waserror()){
        postnote(up, 1, up->errstr, NDebug);
        return 1;
    }

    if(up->fpstate & FPillegal)
        error("floating point in note handler");

    nfp = 0;
    pc = ureg->pc;
    validaddr(pc, 4, 0);
    if(!condok(ureg->psr, *(ulong*)pc >> 28))
        iprint("fpuemu: conditional instr shouldn't have got here\n");
    op  = (*(ulong *)pc >> 24) & MASK(4);
    cop = (*(ulong *)pc >>  8) & MASK(4);
    if(cpu->fpon)
        fpstuck(pc);        /* debugging; could move down 1 line */
    if (ISVFPOP(cop, op)) {  /* if vfp, fpu must be off */
        mathemu(ureg);      /* enable fpu & retry */
        nfp = 1;
    }

    poperror();
    return nfp;
}
```

Uses FPillegal 513a, ISVFPOP-2591 526b, MASK 47c, NDebug 449a, condok() 525c, fpstuck() 524e, iprint, poperror 666d, postnote, up, validaddr() 120b, and waserror 666d.

⟨trap() *other locals(arm)* 526d⟩+≡ (168b) ◁171f

```
int rv;
```

⟨`trap()` *when undefined instruction, if float instr(arm)* 527a⟩≡ (170e)
```
/* look for floating point instructions to interpret */
rv = fpuemu(ureg);
```
Uses `fpuemu()` 526c.

22.1.7 `mathxxx()` (ARM)

The [mathemu] function implements the FPU state machine. On `FPinit` (first-ever use), it calls [fpinit] to configure the hardware from scratch. On `FPinactive` (previously saved state), it checks for pending exceptions before restoring—if there are exceptions, it delivers a note to the process instead of restoring. On `FPactive` (FPU already on), the fault is a genuine illegal instruction. After handling any state transition, [fpclear] clears pending exceptions so the retried instruction starts clean.

⟨*function* `mathnote`*(arm)* 527b⟩≡ (1008c)
```
static void
mathnote(void)
{
    ulong status;
    char *msg, note[ERRMAX];

    status = up->fpsave.status;

    /*
     * Some attention should probably be paid here to the
     * exception masks and error summary.
     */
    if (status & FPAINEX)
        msg = "inexact";
    else if (status & FPAOVFL)
        msg = "overflow";
    else if (status & FPAUNFL)
        msg = "underflow";
    else if (status & FPAZDIV)
        msg = "divide by zero";
    else if (status & FPAINVAL)
        msg = "bad operation";
    else
        msg = "spurious";
    snprint(note, sizeof note, "sys: fp: %s fppc=%#p status=%#lux",
        msg, up->fpsave.pc, status);
    postnote(up, 1, note, NDebug);
}
```
Uses `ERRMAX`, `NDebug` 449a, `postnote`, and `up`.

⟨*function* `mathemu`*(arm)* 527c⟩≡ (1008c)
```
static void
mathemu(Ureg *)
{
    switch(up->fpstate){
    case FPinit:
        fpinit();
        up->fpstate = FPactive;
        break;
    case FPinactive:
        /*
         * Before restoring the state, check for any pending
         * exceptions.  There's no way to restore the state without
         * generating an unmasked exception.
```

```
             * More attention should probably be paid here to the
             * exception masks and error summary.
             */
            if(up->fpsave.status & (FPAINEX|FPAUNFL|FPAOVFL|FPAZDIV|FPAINVAL)){
                mathnote();
                break;
            }
            fprestore(up);
            up->fpstate = FPactive;
            break;
        case FPactive:
            error("illegal instruction: bad vfp fpu opcode");
            break;
        }
        fpclear();
    }
```

Uses FPactive 513a, FPinactive 513a, FPinit 513a, fpclear() 524d, fpinit() 524b, and up.

⟨*function* fprestore*(arm)* 528a⟩≡ (1008c)

```
    static void
    fprestore(Proc *p)
    {
        int n;

        fpon();
        fpwr(Fpscr, p->fpsave.control);
        cpu->fpscr = fprd(Fpscr) & ~Allcc;
        assert(cpu->fpnregs);
        for (n = 0; n < cpu->fpnregs; n++)
            fprestreg(n, *(uvlong *)p->fpsave.regs[n]);
    }
```

Uses Allcc-2582 521b, Fpscr-2564 521a, assert 853f, fpon() 524c, fprd() 519b, fprestreg() 520b, and fpwr() 519c.

22.1.8 /proc/<pid>/fpregs

⟨procread() *Qfpregs case* 528b⟩≡ (605d)

```
    case Qfpregs:
        rptr = (uchar*)&p->fpsave;
        rsize = sizeof(Arch_FPsave);
        goto regread;
```

Uses Qfpregs-864 602a.

⟨procwrite *Qfpregs case* 528c⟩≡ (608c)

```
    case Qfpregs:
        if(offset >= sizeof(Arch_FPsave))
            n = 0;
        else if(offset+n > sizeof(Arch_FPsave))
            n = sizeof(Arch_FPsave) - offset;
        memmove((uchar*)&p->fpsave+offset, va, n);
        break;
```

Uses Qfpregs-864 602a.

22.2 CPU cache

CPU caches are small fast memories that sit between the processor and main RAM. The ARM has separate L1 caches for instructions (I-cache) and data (D-cache), which means code modifications (like the self-modifying

FPU trick above, or writing exception vectors) require explicit cache maintenance: the D-cache must be written back so the new data reaches memory, and the I-cache must be invalidated so the processor fetches the updated instructions. The kernel must also manage caches around MMU operations, because changing page tables can make cached virtual-to-physical translations stale.

22.2.1 Instruction and data cache (ARM)

⟨CpSC primary registers *other cases(arm)* 529a⟩≡ (39d) 535a▷
```
#define CpCACHE     7           /* cache/write buffer control */
```

⟨CpCONTROL.CpMainctl *other cases(arm)* 529b⟩≡ (40b) 539a▷
```
#define CpCdcache   0x00000004  /* C: data cache on */
#define CpCicache   0x00001000  /* I: instruction cache on */
```

⟨armstart() *disable features(arm)* 529c⟩≡ (270)
```
BIC $(CpCdcache|CpCicache), R1
```

⟨armstart() *invalidate caches(arm)* 529d⟩≡ (270)
```
MCR CpSC, 0, R0, C(CpCACHE), C(CpCACHEinvu), CpCACHEall
```

⟨armstart() *reenable features(arm)* 529e⟩≡ (270)
```
ORR $(CpCdcache|CpCicache), R0
```

22.2.2 Cache policy

The cache can operate in write-through mode (every store goes immediately to memory) or write-back mode (stores stay in the cache and are written to memory later). Write-back is more efficient but requires explicit "writeback" operations before other bus masters (DMA, other CPUs) need to see the data. The `Buffered` and `Cached` page table entry attributes select write-back caching for normal DRAM pages. Memory-mapped I/O regions must not be cached, since reading a device register should always go to the hardware, not return a stale cached value.

⟨*type CpCACHERANGE opcode2(arm)* 529f⟩≡ (888c)
```
/*
 * CpCACHERANGE opcode2 fields for MCRR instruction (armv6)
 */
#define CpCACHERANGEinvd    6       /* invalidate data */
#define CpCACHERANGEdwb     12      /* writeback */
#define CpCACHERANGEdwbi    14      /* writeback+invalidate */
```

⟨*instruction MCRR(arm)* 529g⟩≡ (890a)
```
#define MCRR(coproc, op, rd, rn, crm) \
    WORD $(0xec400000|(rn)<<16|(rd)<<12|(coproc)<<8|(op)<<4|(crm))
```

22.2.3 Paging and caching (ARM)

⟨*type* PageTableEntryAttribute1*(arm)* 529h⟩≡ (888c)
```
#define Buffered    0x00000004      /* L[12]: write-back not -thru */
#define Cached      0x00000008      /* L[12] */
```

⟨Soc *mmu fields* 529i⟩≡ (72a)
```
u32int  l1ptedramattrs;
u32int  l2ptedramattrs;
```

⟨soc(raspberry pi1) *other fields(arm)* 529j⟩≡ (72b) 536g▷
```
.l1ptedramattrs = Cached | Buffered,
.l2ptedramattrs = Cached | Buffered,
```
Uses `Buffered` 529h and `Cached` 529h.

⟨*constant* L1ptedramattrs*(arm)* 530a⟩≡ (1006b)
```
#define L1ptedramattrs  soc.l1ptedramattrs
```

⟨*constant* L2ptedramattrs*(arm)* 530b⟩≡ (1006b)
```
#define L2ptedramattrs  soc.l2ptedramattrs
```

22.2.4 Memory barriers (ARM)

On ARMv6, ISB and DSB are implemented via MCR instructions to coprocessor 7, not as dedicated instructions. ISB invalidates the instruction prefetch buffer, DSB drains the write buffer. They are combined into the `BARRIERS` macro used in boot code.

⟨*instruction ISB(arm)* 530c⟩≡ (890a)
```
#define ISB \
    MOVW    $0, R0; \
    MCR CpSC, 0, R0, C(CpCACHE), C(CpCACHEinvi), CpCACHEwait
```

⟨*instruction DSB(arm)* 530d⟩≡ (890a)
```
#define DSB \
    MOVW    $0, R0; \
    MCR CpSC, 0, R0, C(CpCACHE), C(CpCACHEwb), CpCACHEwait
```

22.2.5 Cache operations (ARM)

The ARM cache operations are accessed through MCR/MRC instructions to coprocessor 7 (`CpCACHE`). The terminology is confusing: ARM's "flush" means invalidate (discard cached data without writing back), and "clean" means writeback (push dirty data to memory). The operations come in three granularities: entire cache (`CpCACHEall`), single cache line by virtual address (`CpCACHEse`), and by set/way index (`CpCACHEsi`). The kernel provides C-callable wrappers: `cachedwb` (writeback), `cachedwbinv` (writeback + invalidate), `cachedwbse` (writeback a range), `cacheiinv` (invalidate I-cache), and `cacheuwbinv` (unified: writeback D-cache + invalidate I-cache).

⟨*type CpCACHE secondary registers(arm)* 530e⟩≡ (888c)
```
/*
 * CpCACHE Secondary (CRm) registers and opcode2 fields.  op1==0.
 * In ARM-speak, 'flush' means invalidate and 'clean' means writeback.
 */
#define CpCACHEintr 0           /* interrupt (op2==4) */
#define CpCACHEinvi 5           /* instruction, branch table */
#define CpCACHEinvd 6           /* data or unified */
#define CpCACHEinvu 7           /* unified (not on cortex) */
#define CpCACHEwb   10          /* writeback to PoC */
#define CpCACHEwbi  14          /* writeback+invalidate (to PoC) */
```

⟨*type CpCACHE opcode2(arm)* 530f⟩≡ (888c)
```
#define CpCACHEall  0           /* entire (not for invd nor wb(i) on cortex) */
#define CpCACHEse   1           /* single entry */
#define CpCACHEsi   2           /* set/index (set/way) */
#define CpCACHEwait 4           /* wait (prefetch flush on cortex) */
```

⟨*constant CACHELINESZ(raspberry pi1)(ram)* 530g⟩≡ (1007b)
```
#define CACHELINESZ 32
```

22.2.6 cachexxx() (ARM)

The cache operations are called throughout the kernel at points where coherency between caches, memory, and the MMU must be maintained. The most common pattern is after modifying page table entries: `cachedwbse` writes back the modified PTEs so the MMU's page table walker sees them. Before an MMU switch (`arch_mmuswitch`), `cacheuwbinv` flushes everything because virtual addresses are about to change meaning. The self-modifying FPU code calls `cachedwbse` followed by `cacheiinv` to ensure the dynamically constructed instructions reach the I-cache.

⟨`arch_mmuswitch()` *write back and invalidate cache before switch(arm)* 531a⟩≡ (230b)
```
/* write back dirty and invalidate l1 caches */
cacheuwbinv();
```

⟨`arch_mmurelease()` *write back cache after adjusting page tables(arm)* 531b⟩≡ (180f)
```
/* make sure map is in memory */
/* could be smarter about how much? */
cachedwbse(&cpu->mmul1[L1X(UZERO)], (L1hi - L1lo)*sizeof(PTE));
```
Uses `L1X-1842` 178a, `L1hi-1848` 180b, and `L1lo-1847` 180a.

⟨`arch_mmuswitch()` *write back cache after adjusting page tables(arm)* 531c⟩≡ (230b)
```
/* make sure map is in memory */
/* could be smarter about how much? */
cachedwbse(&l1[L1X(UZERO)], (L1hi - L1lo)*sizeof(PTE));
```
Uses `L1X-1842` 178a, `L1hi-1848` 180b, and `L1lo-1847` 180a.

⟨`mmuinit1()` *write back cache after adjusting page tables(arm)* 531d⟩≡ (281c)
```
cachedwbse(&l1[L1X(0)], sizeof(PTE));
```
Uses `L1X-1842` 178a.

⟨`mmukmap()` *write back cache after changing page tables(arm)* 531e⟩≡ (387a)
```
cachedwbse(pte0, (uintptr)pte - (uintptr)pte0);
```

⟨`arch_putmmu()` *write back cache after changing pde(arm)* 531f⟩≡ (186)
```
cachedwbse(l1, sizeof *l1);
```

⟨`arch_putmmu()` *write back cache after changing pte(arm)* 531g⟩≡ (186)
```
cachedwbse(&pte[L2X(va)], sizeof pte[0]);
```
Uses `L2X-1843` 178b.

⟨`arch_putmmu()` *write back cache before fixing va fault(arm)* 531h⟩≡ (186)
```
/* force l2 page to memory */
cachedwbse((void *)pg->va, BY2PG);
```

⟨`arch_mmurelease()` *invalidate cache before adjusting page tables(arm)* 531i⟩≡ (180f)
```
/* write back dirty and invalidate l1 caches */
cacheuwbinv();
```

⟨`arch_kunmap()` *write back cache for page(arm)* 531j⟩≡ (188e)
```
cachedwbinvse(k, BY2PG);
```

⟨`arch__trapinit()` *invalidate cache after adjust vectors(arm)* 531k⟩≡ (278e)
```
cacheuwbinv();
l2cacheuwbinv();
```

⟨`fbinit()` *write back cache after modified fb info(arm)* 531l⟩≡ (386b)
```
cachedwbinvse(fi, sizeof(Fbinfo));
```

⟨`arch__kexit()` *write back cache(arm)* 531m⟩≡ (169)
```
/* make visible immediately to user proc */
cachedwbinvse(tos, sizeof *tos);
```

⟨setupfpctlop() *write back cache for modified instr(arm)* 532a⟩≡ (519a)
```
cachedwbse(instr, sizeof instrsz);
```

⟨setupfpop() *write back cache for modified instr(arm)* 532b⟩≡ (520a)
```
cachedwbse(instr, sizeof instrsz);
```

⟨setupfpctlop() *invalidate instruction cache(arm)* 532c⟩≡ (519a)
```
cacheiinv();
```

⟨setupfpop() *invalidate instruction cache(arm)* 532d⟩≡ (520a)
```
cacheiinv();
```

⟨*function cachexxx(raspberry pi1)(ram)* 532e⟩≡ (1007b)
```
/*
 * drain write buffer
 * writeback data cache
 */
TEXT cachedwb(SB), 1, $-4
    DSB
    MOVW    $0, R0
    MCR CpSC, 0, R0, C(CpCACHE), C(CpCACHEwb), CpCACHEall
    RET

/*
 * drain write buffer
 * writeback and invalidate data cache
 */
TEXT cachedwbinv(SB), 1, $-4
    DSB
    MOVW    $0, R0
    MCR CpSC, 0, R0, C(CpCACHE), C(CpCACHEwbi), CpCACHEall
    RET

/*
 * cachedwbinvse(va, n)
 *   drain write buffer
 *   writeback and invalidate data cache range [va, va+n)
 */
TEXT cachedwbinvse(SB), 1, $-4
    MOVW    R0, R1      /* DSB clears R0 */
    DSB
    MOVW    n+4(FP), R2
    ADD R1, R2
    SUB $1, R2
    BIC $(CACHELINESZ-1), R1
    BIC $(CACHELINESZ-1), R2
    MCRR(CpSC, 0, 2, 1, CpCACHERANGEdwbi)
    RET

/*
 * cachedwbse(va, n)
 *   drain write buffer
 *   writeback data cache range [va, va+n)
 */
TEXT cachedwbse(SB), 1, $-4
    MOVW    R0, R1      /* DSB clears R0 */
    DSB
    MOVW    n+4(FP), R2
    ADD R1, R2
    BIC $(CACHELINESZ-1), R1
```

```
    BIC $(CACHELINESZ-1), R2
    MCRR(CpSC, 0, 2, 1, CpCACHERANGEdwb)
    RET

/*
 * cachedinvse(va, n)
 *   drain write buffer
 *   invalidate data cache range [va, va+n)
 */
TEXT cachedinvse(SB), 1, $-4
    MOVW    R0, R1      /* DSB clears R0 */
    DSB
    MOVW    n+4(FP), R2
    ADD R1, R2
    SUB $1, R2
    BIC $(CACHELINESZ-1), R1
    BIC $(CACHELINESZ-1), R2
    MCRR(CpSC, 0, 2, 1, CpCACHERANGEinvd)
    RET

/*
 * drain write buffer and prefetch buffer
 * writeback and invalidate data cache
 * invalidate instruction cache
 */
TEXT cacheuwbinv(SB), 1, $-4
    BARRIERS
    MOVW    $0, R0
    MCR CpSC, 0, R0, C(CpCACHE), C(CpCACHEwbi), CpCACHEall
    MCR CpSC, 0, R0, C(CpCACHE), C(CpCACHEinvi), CpCACHEall
    RET

/*
 * L2 cache is not enabled
 */
TEXT l2cacheuwbinv(SB), 1, $-4
    RET

/*
 * invalidate instruction cache
 */
TEXT cacheiinv(SB), 1, $-4
    MOVW    $0, R0
    MCR CpSC, 0, R0, C(CpCACHE), C(CpCACHEinvi), CpCACHEall
    RET
```

22.3 Memory model

Modern processors do not necessarily perform memory operations in program order. The CPU may reorder loads and stores for performance, and with SMP the situation is worse because each processor has its own cache. Memory barriers force ordering: a Data Synchronization Barrier (DSB) ensures all preceding stores are visible before any following ones, and an Instruction Synchronization Barrier (ISB) flushes the instruction pipeline. The kernel's `arch_coherence` function issues both barriers and is called at critical points: after releasing a lock (so other processors see the unlocked state), around interrupt dispatch, and after compare-and-swap operations.

22.3.1 arch_coherence()

⟨unlock() *after modified key, call coherence* 534a⟩≡ (78d)
```
// for processor caches, to ensure the lock value is seen by other
// processors so that if they were doing while(l->key) { ... } they
// can finally exit the while loop.
arch_coherence();
```
Uses arch_coherence.

⟨iunlock() *after modified key, call coherence* 534b⟩≡ (80b)
```
arch_coherence();
```
Uses arch_coherence.

⟨irq() *before dispatch(arm)* 534c⟩≡ (177b)
```
arch_coherence();
```
Uses arch_coherence.

⟨irq() *after dispatch(arm)* 534d⟩≡ (177b)
```
arch_coherence();
```
Uses arch_coherence.

⟨arch__trapinit() *coherence(arm)* 534e⟩≡ (278e)
```
arch_coherence();
```
Uses arch_coherence.

⟨init0() *coherence(arm)* 534f⟩≡ (289a)
```
arch_coherence();
```
Uses arch_coherence.

⟨cas32() *if address contained old value, call coherence(arm)* 534g⟩≡ (474d)
```
if (r)
    arch_coherence();
```
Uses arch_coherence.

22.3.2 Memory barriers (ARM)

⟨*instruction BARRIERS(arm)* 534h⟩≡ (890a)
```
#define BARRIERS    ISB; DSB
```

22.4 The Raspberry Pi 2 machine

The Raspberry Pi 2 uses a BCM2836 SoC with four Cortex-A7 cores, 1 GB of RAM, and peripheral I/O remapped to 0x3F000000 (versus 0x20000000 on the Pi 1). Most of the kernel is portable between Pi 1 and Pi 2, but SMP support, the different cache line size (64 bytes vs 32), the generic ARM timer, per-CPU local registers, and the LDREX/STREX atomic instructions are Pi 2-specific. The Soc structure captures these hardware differences so the same kernel source can target both machines.

⟨*global* soc*(raspberry pi2)(arm)* 534i⟩≡ (994c)
```
Soc soc = {
    .dramsize   = 1024*MiB,
    .physio     = 0x3F000000,
     ⟨soc(raspberry pi2) other fields(arm) 535f⟩
};
```
Uses MiB 48a.

22.4.1 CPU ID

⟨CpSC primary registers *other cases(arm)* 535a⟩+≡ (39d) ◁529a 537a▷
```
#define CpID        0           /* ID and cache type */
```

⟨*type CpID secondary registers(arm)* 535b⟩≡ (888c)
```
/*
 * CpID Secondary (CRm) registers.
 */
#define CpIDidct    0
#define CpIDfeat    1
```

⟨*type CpID opcode2(arm)* 535c⟩≡ (888c)
```
/*
 * CpID op1==0 opcode2 fields.
 * the cortex has more op1 codes for cache size, etc.
 */
#define CpIDid      0           /* main ID */
#define CpIDmpid    5           /* multiprocessor id (cortex) */
```

⟨*instruction CPUID(arm)* 535d⟩≡ (890a)
```
/*
 * get cpu id, or zero if armv6
 */
#define CPUID(r) \
    MRC CpSC, 0, r, C(CpID), C(CpIDfeat), 7; \
    CMP $0, r; \
    B.EQ    2(PC); \
    MRC CpSC, 0, r, C(CpID), C(CpIDidct), CpIDmpid; \
    AND.S   $(MAXCPUS-1), r
```

22.4.2 MMU

On the Pi 2, page table entries for DRAM include additional attributes: write-allocate (so cache lines are allocated on writes, not just reads) and sharable (so the cache coherency protocol keeps the entry consistent across CPUs). These are critical for SMP correctness—without the sharable bit, one CPU's cache could hold stale data that another CPU has modified.

⟨*type* PageTableEntryAttribute2*(arm)* 535e⟩≡ (888c)
```
#define L1wralloc   (1<<12)         /* L1 TEX */
#define L1sharable  (1<<16)
#define L2wralloc   (1<<6)          /* L2 TEX (small pages) */
#define L2sharable  (1<<10)
```

⟨soc(raspberry pi2) *other fields(arm)* 535f⟩≡ (534i) 536h▷
```
.l1ptedramattrs = Cached | Buffered | L1wralloc | L1sharable,
.l2ptedramattrs = Cached | Buffered | L2wralloc | L2sharable,
```
Uses Buffered 529h, Cached 529h, L1sharable 535e, L1wralloc 535e, L2sharable 535e, and L2wralloc 535e.

⟨*function* l2ap*(raspberry pi2)(arm)* 535g⟩≡ (994c)
```
int
l2ap(int ap)
{
    return (AP(0, (ap)));
}
```
Uses AP 179f.

22.4.3 TTB

⟨*type* CpTTB*(arm)* 536a⟩≡ (888c)
```
/*
 * CpTTB op1==0, Crm==0 opcode2 values.
 */
#define CpTTB1      1           /* cortex */
```

⟨*type CpTTB cache control bits(arm)* 536b⟩≡ (888c)
```
/*
 * CpTTB cache control bits
 */
#define CpTTBs  (1<<1)  /* page table in shareable memory */
```

22.4.4 Local registers

The BCM2836 adds per-CPU "local" registers at physical address 0x40000000, above the main 1 GB RAM. These include per-CPU mailboxes used to start secondary processors during boot: CPU 0 writes the entry point address into a secondary CPU's `startcpu` mailbox, and that CPU begins executing at that address. The mailboxes are also used for inter-processor interrupts.

⟨*struct* Mbox*(arm)* 536c⟩≡ (994c)
```
/*
 * Arm local regs for SMP
 */
struct Mbox {
    u32int  doorbell;
    u32int  mbox1;
    u32int  mbox2;
    u32int  startcpu;
};
```

⟨*struct* Mboxes*(arm)* 536d⟩≡ (994c)
```
struct Mboxes {
    Mbox    set[4];
    Mbox    clr[4];
};
```

⟨*constant* ARMLOCAL*(arm)* 536e⟩≡ (1011b 994c)
```
#define ARMLOCAL    (VIRTIO+IOSIZE)
```

⟨Soc *other fields* 536f⟩≡ (72a)
```
uintptr armlocal;
```

⟨soc(raspberry pi1) *other fields(arm)* 536g⟩+≡ (72b) ◁529j 957f▷
```
.armlocal   = 0,
```

⟨soc(raspberry pi2) *other fields(arm)* 536h⟩+≡ (534i) ◁535f 957g▷
```
.armlocal   = 0x40000000,
```

⟨mmuinit() *set armlocal virtual memory mapping(arm)* 536i⟩≡ (271c)
```
// for raspi2
pa = soc.armlocal;
if(pa)
    l1[L1X(va)] = pa|Dom0|L1AP(Krw)|Section;
```
Uses Dom0 179d, Krw 179b, L1AP 179e, L1X-1842 178a, and Section 178d.

22.4.5 Timer

⟨CpSC primary registers *other cases(arm)* 537a⟩+≡ (39d) ◁535a 537c▷
```
#define CpTIMER     14          /* Generic timer (cortex-a7) */
```

⟨*type* CpTIMER*(arm)* 537b⟩≡ (888c)
```
/*
 * CpTIMER op1==0 Crm and opcode2 registers (cortex-a7)
 */
#define CpTIMERphys     2

#define CpTIMERphysval  0
#define CpTIMERphysctl  1
```

22.4.6 Performance monitor

⟨CpSC primary registers *other cases(arm)* 537c⟩+≡ (39d) ◁537a
```
#define CpCLD       9       // Performance monitor
```

⟨*type CpCLD secondary registers(arm)* 537d⟩≡ (888c)
```
/*
 * CpCLD Secondary (CRm) registers and opcode2 fields for op1==0. (cortex)
 */
#define CpCLDena    12          /* enables */
#define CpCLDcyc    13          /* cycle counter */
```

⟨*type CpCLD opcode2(arm)* 537e⟩≡ (888c)
```
#define CpCLDenapmnc    0
#define CpCLDenacyc 1
```

22.5 SMP (ARM)

Adding SMP support to the kernel requires changes at many levels: the boot sequence must start secondary processors, each processor needs its own `Cpu` structure and page tables, locks must use atomic instructions (LDREX/STREX instead of SWP), memory barriers must be inserted around lock operations, and the timer/interrupt subsystems must handle per-CPU local interrupts. The overall structure remains the same—each CPU ends up calling [schedinit] and picking up processes from the shared run queue.

22.5.1 Initialization adjustments

Basic pagination cpu0

The Pi 2 boot sequence for CPU 0 is almost identical to the Pi 1: disable MMU and caches, clear the page table area, call [mmuinit], then enable the MMU and jump to virtual addresses. The main difference is that the SMP coherency bit (`CpACsmp`) must be set in the auxiliary control register before enabling caches, so that L1 caches participate in the hardware coherency protocol. The very first thing `armstart` does is read the CPU ID: if this is not CPU 0, it branches to `reset` for secondary CPU startup.

⟨*function* armstart*(raspberry pi2)(arm)* 537f⟩≡ (990c)
```
TEXT armstart(SB), 1, $-4

    /*
     * if not cpu0, go to secondary startup
     */
    CPUID(R1)
    BNE reset
```

```
    /*
     * disable the mmu and caches
     * invalidate tlb
     */
    MRC CpSC, 0, R1, C(CpCONTROL), C(0), CpMainctl
    BIC $(CpCdcache|CpCicache|CpCmmu), R1
    ORR $(CpCsbo|CpCsw), R1
    BIC $CpCsbz, R1
    MCR CpSC, 0, R1, C(CpCONTROL), C(0), CpMainctl
    MCR CpSC, 0, R0, C(CpTLB), C(CpTLBinvu), CpTLBinv
    ISB

    /*
     * clear mach and page tables
     */
    MOVW    $PADDR(CPUADDR), R1
    MOVW    $PADDR(KTZERO), R2
_ramZ:
    MOVW    R0, (R1)
    ADD $4, R1
    CMP R1, R2
    BNE _ramZ

    /*
     * start stack at top of mach (physical addr)
     * set up page tables for kernel
     */
    MOVW    $PADDR(CPUADDR+CPUSIZE-4), R13
    MOVW    $PADDR(L1), R0
    BL  mmuinit(SB)

    /*
     * set up domain access control and page table base
     */
    MOVW    $Client, R1
    MCR CpSC, 0, R1, C(CpDAC), C(0)
    MOVW    $PADDR(L1), R1
    ORR     $(CpTTBs/*|CpTTBowba|CpTTBiwba*/), R1
    MCR CpSC, 0, R1, C(CpTTB), C(0)
    MCR CpSC, 0, R1, C(CpTTB), C(0), CpTTB1 /* cortex has two */

    /*
     * invalidate my caches before enabling
     */
    BL  cachedinv(SB)
    BL  cacheiinv(SB)
    BL  l2cacheuinv(SB)
    BARRIERS

    /*
     * enable caches, mmu, and high vectors
     */
    MRC CpSC, 0, R1, C(CpCONTROL), C(0), CpAuxctl
    ORR $CpACsmp, R1        /* turn SMP on */
    MCR CpSC, 0, R1, C(CpCONTROL), C(0), CpAuxctl
    BARRIERS

    MRC CpSC, 0, R0, C(CpCONTROL), C(0), CpMainctl
    ORR $(CpChv|CpCdcache|CpCicache|CpCmmu), R0
```

```
MCR CpSC, 0, R0, C(CpCONTROL), C(0), CpMainctl
BARRIERS

/*
 * switch SB, SP, and PC into KZERO space
 */
MOVW    $setR12(SB), R12
MOVW    $(CPUADDR+CPUSIZE-4), R13
MOVW    $_startpg(SB), R15
```

⟨CpCONTROL.CpMainctl *other cases(arm)* 539a⟩+≡ (40b) ◁529b 539b▷

```
#define CpCsw       (1<<10)     /* SW: SWP(B) enable (deprecated in v7) */
#define CpCpredict  0x00000800  /* Z: branch prediction (armv7) */
```

⟨CpCONTROL.CpMainctl *other cases(arm)* 539b⟩+≡ (40b) ◁539a 539c▷

```
#define CpCve       (1<<24)     /* VE: intr vectors enable */
#define CpCtre      (1<<28)     /* TRE: TEX remap enable */
```

⟨CpCONTROL.CpMainctl *other cases(arm)* 539c⟩+≡ (40b) ◁539b

```
#define CpCsbo (3<<22|1<<18|1<<16|017<<3)       /* must be 1 (armv7) */
#define CpCsbz (CpCtre|1<<26|CpCve|1<<15|7<<7)  /* must be 0 (armv7) */
```

⟨CpCONTROL.opcode2 *other cases(arm)* 539d⟩+≡ (40a) ◁522b

```
#define CpAuxctl    1
```

⟨*type CpCONTROL CpAuxctl(arm)* 539e⟩≡ (888c)

```
/*
 * CpCONTROL: op1==0, CRm==0, op2==CpAuxctl.
 * Auxiliary control register on cortex at least.
 */
/* cortex-a7 and cortex-a9 */
#define CpACsmp         (1<<6)  /* SMP l1 caches coherence; needed for ldrex/strex */
```

Jumping to C `main()` cpu0

⟨*function* `_startpg`*(raspberry pi2)(arm)* 539f⟩≡ (990c)

```
TEXT _startpg(SB), 1, $-4

    ⟨_startpg() enable cycle counter(raspberry pi2)(arm) 553h⟩
    /*
     * call main and loop forever if it returns
     */
    BL  main(SB)
    B   0(PC)

    BL  _div(SB)        /* hack to load _div, etc. */
```

Starting the other processors

After CPU 0 finishes its initialization and starts the first user process, it calls [launchinit] to bring up the secondary CPUs. For each secondary CPU, [launchinit] allocates a `Cpu` structure and a fresh L1 page table (cloned from CPU 0's), then calls [startcpus]. The startup handshake uses the BCM2836 mailbox mechanism: CPU 0 writes the physical address of [cpureset] into the target CPU's mailbox, which wakes it up. A per-CPU lock synchronizes the handshake—CPU 0 locks it before starting the CPU and waits for the secondary to unlock it once running.

⟨`main()` *start the other processors(arm)* 539g⟩≡ (272b)

```
launchinit(getncpus());
```

Uses `launchinit()` 540a.

⟨*function* `launchinit`*(arm)* 540a⟩≡ (991c)

```
void
launchinit(int ncpus)
{
    int xcpu;
    Cpu *mm;
    PTE *l1;

    ⟨launchinit() sanity check ncpus(arm) 540c⟩
    for(xcpu = 1; xcpu < ncpus; xcpu++){
        cpus[xcpu] = mm = mallocalign(CPUSIZE, CPUSIZE, 0, 0);
        l1 = mallocalign(L1SIZE, L1SIZE, 0, 0);
        if(mm == nil || l1 == nil)
            panic("launchinit");
        memset(mm, 0, CPUSIZE);
        mm->cpuno = xcpu;

        memmove(l1, cpu->mmul1, L1SIZE);  /* clone cpu0's l1 table */
        cachedwbse(l1, L1SIZE);
        mm->mmul1 = l1;
        cachedwbse(mm, CPUSIZE);

    }
    cachedwbse(cpus, sizeof cpus);
    xcpu = startcpus(ncpus);
    if(xcpu < ncpus)
            panic("only %d cpu%s started", xcpu, xcpu == 1? "" : "s");
}
```

Uses `L1SIZE` 540b, `cpus` 65a, and `mallocalign()` 105a.

⟨*constant* `L1SIZE`*(arm)* 540b⟩≡ (1005)

```
#define L1SIZE      (4 * BY2PG)
```

⟨`launchinit()` *sanity check ncpus(arm)* 540c⟩≡ (540a)

```
if(ncpus > MAXCPUS)
    ncpus = MAXCPUS;
```

⟨*function* `startcpus`*(raspberry pi1)(arm)* 540d⟩≡ (994b)

```
int
startcpus(uint)
{
    return 1;
}
```

⟨*function* `getncpus`*(raspberry pi1)(arm)* 540e⟩≡ (994b)

```
int
getncpus(void)
{
    return 1;
}
```

⟨*function* `getncpus`*(raspberry pi2)(arm)* 540f⟩≡ (994c)

```
int
getncpus(void)
{
    int n, max;
    char *p;

    n = 4;
    if(n > MAXCPUS)
```

```
        n = MAXCPUS;
    ⟨getncpus() adjust n if ncpu config parameter(arm) 541a⟩
    return n;
}
```

⟨getncpus() *adjust n if ncpu config parameter(arm)* 541a⟩≡ (540f)

```
p = getconf("*ncpu");
if(p && (max = atoi(p)) > 0 && n > max)
    n = max;
```

⟨*function* startcpus*(raspberry pi2)(arm)* 541b⟩≡ (994c)

```
int
startcpus(uint ncpu)
{
    int i;

    for(i = 0; i < ncpu; i++)
        lock(&startlock[i]);
    cachedwbse(startlock, sizeof startlock);
    for(i = 1; i < ncpu; i++){
        if(startcpu(i) < 0)
            return i;
        lock(&startlock[i]);
        unlock(&startlock[i]);
    }
    return ncpu;
}
```

Uses lock() 76d, startcpu() 541c, startlock-2448 994c, and unlock() 78d.

⟨*function* startcpu*(raspberry pi2)(arm)* 541c⟩≡ (994c)

```
static int
startcpu(uint xcpu)
{
    Mboxes *mb;
    int i;
    void cpureset();

    mb = (Mboxes*)(ARMLOCAL + Mboxregs);
    if(mb->clr[xcpu].startcpu)
        return -1;
    mb->set[xcpu].startcpu = PADDR(cpureset);
    for(i = 0; i < 1000; i++)
        if(mb->clr[xcpu].startcpu == 0)
            return 0;
    mb->clr[xcpu].startcpu = PADDR(cpureset);
    mb->set[xcpu].doorbell = 1;
    return 0;
}
```

Uses ARMLOCAL-2446 536e and Mboxregs-2447 994c.

Basic pagination other cpus

Secondary CPUs wake up executing at the physical address written to their mailbox. The [cpureset] code is similar to CPU 0's armstart but must find its own Cpu structure: it reads its CPU ID, indexes into the cpus array (set up by [launchinit]), and uses that CPU's pre-allocated L1 page table. After enabling the MMU, it

adds KZERO to all registers to switch into virtual address space, then jumps to `_startpg2`.

⟨*function* `cpureset` *(raspberry pi2)(arm)* 542⟩≡ (990c)

```
/*
 * startup entry for cpu(s) other than 0
 */
TEXT cpureset(SB), 1, $-4
reset:
    /*
     * load physical base for SB addressing while mmu is off
     * keep a handy zero in R0 until first function call
     */
    MOVW    $setR12(SB), R12
    SUB $KZERO, R12
    MOVW    $0, R0

    /*
     * SVC mode, interrupts disabled
     */
    MOVW    $(PsrDirq|PsrDfiq|PsrMsvc), R1
    MOVW    R1, CPSR

    /*
     * disable the mmu and caches
     * invalidate tlb
     */
    MRC CpSC, 0, R1, C(CpCONTROL), C(0), CpMainctl
    BIC $(CpCdcache|CpCicache|CpCmmu), R1
    ORR $(CpCsbo|CpCsw), R1
    BIC $CpCsbz, R1
    MCR CpSC, 0, R1, C(CpCONTROL), C(0), CpMainctl
    MCR CpSC, 0, R0, C(CpTLB), C(CpTLBinvu), CpTLBinv
    ISB

    /*
     * find Mach for this cpu
     */
    MRC CpSC, 0, R2, C(CpID), C(CpIDidct), CpIDmpid
    AND $(MAXCPUS-1), R2    /* mask out non-cpu-id bits */
    SLL $2, R2          /* convert to word index */
    MOVW    $cpus(SB), R0
    ADD R2, R0          /* R0 = &cpus[cpuid] */
    MOVW    (R0), R0        /* R0 = cpus[cpuid] */
    CMP $0, R0
    MOVW.EQ $CPUADDR, R0        /* paranoia: use CPUADDR if 0 */
    SUB $KZERO, R0      /* phys addr */
    MOVW    R0, R(CPU)      /* m = PADDR(cpus[cpuid]) */

    /*
     * start stack at top of local Mach
     */
    MOVW    R(CPU), R13
    ADD     $(CPUSIZE-4), R13

    /*
     * set up page tables for kernel
     */
    MOVW    12(R(CPU)), R0  /* m->mmul1 */
    SUB $KZERO, R0      /* phys addr */
    BL  mmuinit(SB)
```

```
/*
 * set up domain access control and page table base
 */
MOVW    $Client, R1
MCR CpSC, 0, R1, C(CpDAC), C(0)
MOVW    12(R(CPU)), R1  /* m->mmul1 */
SUB $KZERO, R1      /* phys addr */
ORR     $(CpTTBs/*|CpTTBowba|CpTTBiwba*/), R1
MCR CpSC, 0, R1, C(CpTTB), C(0)
MCR CpSC, 0, R1, C(CpTTB), C(0), CpTTB1 /* cortex has two */

/*
 * invalidate my caches before enabling
 */
BL  cachedinv(SB)
BL  cacheiinv(SB)
BARRIERS

/*
 * enable caches, mmu, and high vectors
 */
MRC CpSC, 0, R1, C(CpCONTROL), C(0), CpAuxctl
ORR $CpACsmp, R1        /* turn SMP on */
MCR CpSC, 0, R1, C(CpCONTROL), C(0), CpAuxctl
BARRIERS

MRC CpSC, 0, R0, C(CpCONTROL), C(0), CpMainctl
ORR $(CpChv|CpCdcache|CpCicache|CpCmmu), R0
MCR CpSC, 0, R0, C(CpCONTROL), C(0), CpMainctl
BARRIERS

/*
 * switch CPU, SB, SP, and PC into KZERO space
 */
ADD $KZERO, R(CPU)
MOVW    $setR12(SB), R12
ADD $KZERO, R13
MOVW    $_startpg2(SB), R15
```

Jumping to C `cpustart()`

⟨*function* `_startpg2` *(raspberry pi2)(arm)* 543⟩≡ (990c)

```
TEXT _startpg2(SB), 1, $-4

    ⟨_startpg2() enable cycle counter(raspberry pi2)(arm) 553i⟩
    /*
     * call cpustart and loop forever if it returns
     */
    MRC CpSC, 0, R0, C(CpID), C(CpIDidct), CpIDmpid
    AND $(MAXCPUS-1), R0            /* mask out non-cpu-id bits */
    BL  ,cpustart(SB)
    B   ,0(PC)
```

`main()` other processors (`cpustart()`)

The C entry point for secondary CPUs is much simpler than `main`: there is no need to initialize memory pools, the device table, or the first user process—all of that was done by CPU 0. Each secondary CPU just needs to

finalize its own MMU, set up its trap vectors and local timer, then call [schedinit] to enter the scheduler loop and start running processes. The `unlock` of `startlock` signals to CPU 0 that this CPU is ready.

⟨*function* `cpustart` *(raspberry pi2)(arm)* 544a⟩≡ (994c)
```
void
cpustart(int xcpu)
{
    Mboxes *mb;

    up = nil;
    arch__cpuinit();
    mmuinit1((void*)cpu->mmul1);

    mb = (Mboxes*)(ARMLOCAL + Mboxregs);
    mb->clr[xcpu].doorbell = 1;

    arch__trapinit();
    clockinit();
    timersinit();

    arch_cpuidprint();
    archreset();

    machon(cpu->cpuno);

    unlock(&startlock[xcpu]);

    // schedule a ready process (not necessarily the first one now)
    schedinit();

    panic("schedinit returned");
}
```
Uses `ARMLOCAL-2446` 536e, `Mboxregs-2447` 994c, `arch__cpuinit()` 274d, `clockinit()` 280, `machon()` 275a, `mmuinit1()` 281c, `schedinit()` 224f, `startlock-2448` 994c, `timersinit()` 281b, `unlock()` 78d, and `up`.

⟨`arch__cpuinit()` *it not cpu0* 544b⟩≡ (274d)
```
if (cpu->cpuno != 0) {
    /* synchronise with cpu 0 */
    cpu->ticks = cpu0->ticks;
}
```

Displaying processor information

⟨*function* `arch_cpuidprint` *(raspberry pi2)(arm)* 544c⟩≡ (994c)
```
void
arch_cpuidprint(void)
{
    char name[64];

    cputype2name(name, sizeof name);
    arch_delay(50);             /* let uart catch up */
    print("cpu%d: %dMHz ARM %s\n", cpu->cpuno, cpu->cpumhz, name);
}
```
Uses `arch_delay` and `print`.

⟨*function* `cputype2name` *(raspberry pi2)(arm)* 544d⟩≡ (994c)
```
char*
cputype2name(char *buf, int size)
{
```

```
    ulong r;

    r = cpidget();          /* main id register */
    assert((r >> 24) == 'A');
    seprint(buf, buf + size, "Cortex-A7 r%ldp%ld",
        (r >> 20) & MASK(4), r & MASK(4));
    return buf;
}
```

Uses MASK 47c and assert 853f.

⟨*function* cpidget*(raspberry pi2)(arm)* 545a⟩≡ (990c)

```
TEXT cpidget(SB), 1, $-4            /* main ID */
    MRC CpSC, 0, R0, C(CpID), C(0), CpIDid
    RET
```

Halting

When a CPU has nothing to run, `arch_idlehands` could use WFI (Wait For Interrupt) to save power. However, WFI is disabled on the Pi 2 because local timer interrupts can be lost, causing the CPU to hang.

⟨*function* arch_idlehands*(raspberry pi2)(arm)* 545b⟩≡ (990c)

```
TEXT arch_idlehands(SB), $-4
    MOVW    CPSR, R3
    ORR $(PsrDirq|PsrDfiq), R3, R1      /* splfhi */
    MOVW    R1, CPSR

    DSB
    MOVW    nrdy(SB), R0
    CMP $0, R0
/*** with WFI, local timer interrupts can be lost and dispatching stops
    WFI_EQ
***/
    DSB

    MOVW    R3, CPSR            /* splx */
    RET
```

⟨*instruction* WFI_EQ*(arm)* 545c⟩≡ (990c)

```
#define WFI_EQ  WORD    $0x0320f003 /* wait for interrupt if eq */
```

22.5.2 Setting cpu

On a uniprocessor, the `cpu` pointer lives at a fixed address (`CPUADDR`) and never changes. With SMP, each exception handler must figure out which CPU it is running on and load the correct `Cpu` pointer. The CPUID macro reads the multiprocessor ID register, and the result indexes into the `cpus` array. This code appears in both `_vsvc` (syscall entry) and `_vswitch` (trap from user mode).

⟨_vsvc() *set cpu register(arm)* 545d⟩≡ (162c)

```
CPUID(R1)
SLL $2, R1          /* convert to word index */
MOVW    $cpus(SB), R2
ADD R1, R2
MOVW    (R2), R(CPU)        /* m = cpus[cpuid] */
CMP $0, R(CPU)
MOVW.EQ $CPUADDR, R0        /* paranoia: use CPUADDR if 0 */
```

⟨_vswitch() *set cpu register(arm)* 546a⟩≡ (167b)
```
CPUID(R1)
SLL $2, R1          /* convert to word index */
MOVW    $cpus(SB), R2
ADD R1, R2
MOVW    (R2), R(CPU)        /* m = cpus[cpuid] */
CMP $0, R(CPU)
MOVW.EQ $CPUADDR, R0        /* paranoia: use CPUADDR if 0 */
```

22.5.3 Concurrency adjustments

The Pi 1's `arch_tas` uses the SWP instruction, which is deprecated on ARMv7 and does not work correctly on multiprocessors. The Pi 2 replaces it with the LDREX/STREX (load/store exclusive) pair, which provides true atomic compare-and-swap via the cache coherency protocol. LDREX loads a value and marks the cache line as "exclusive"; STREX stores a new value only if the line is still exclusive, returning success or failure. If another CPU touched the same cache line in between, STREX fails and the code retries. DMB (data memory barrier) instructions bracket the sequence to ensure proper ordering.

⟨*instruction LDREX(arm)* 546b⟩≡ (890a)
```
#define LDREX(fp,t)   WORD $(0xe<<28|0x01900f9f | (fp)<<16 | (t)<<12)
```

⟨*instruction STREX(arm)* 546c⟩≡ (890a)
```
/* 'The order of operands is from left to right in dataflow order' - asm man */
#define STREX(f,tp,r) WORD $(0xe<<28|0x01800f90 | (tp)<<16 | (r)<<12 | (f)<<0)
```

⟨*instruction CLREX(arm)* 546d⟩≡ (890a)
```
#define CLREX   WORD    $0xf57ff01f
```

⟨*instruction DMB(arm)* 546e⟩≡ (971b)
```
#define DMB WORD    $0xf57ff05f /* data mem. barrier; last f = SY */
```

⟨*function* `arch_tas` *(raspberry pi2)(arm)* 546f⟩≡ (971b)
```
TEXT    arch_tas(SB), $-4
    /* returns old (R0) after modifying (R0) */
    MOVW    R0,R5
    DMB

    MOVW    $1,R2       /* new value of (R0) */
    MOVW    $MAXSC, R8
tas1:
    LDREX(5,7)      /* LDREX 0(R5),R7 */
    CMP.S   $0, R7      /* old value non-zero (lock taken)? */
    BNE lockbusy    /* we lose */
    SUB.S   $1, R8
    BEQ lockloop2
    STREX(2,5,4)        /* STREX R2,(R5),R4 */
    CMP.S   $0, R4
    BNE tas1        /* strex failed? try again */
    DMB
    B   tas0
lockloop2:
    BL  abort(SB)
lockbusy:
    CLREX
tas0:
    MOVW    R7, R0      /* return old value */
    RET
```

22.5.4 Timer adjustments

On the Pi 1, there is a single system timer shared by the one CPU. On the Pi 2, CPU 0 still uses the system timer, but the secondary CPUs use the ARM generic timer (a per-CPU countdown timer accessed via coprocessor 14). CPU 0 configures the prescaler to produce a 1 MHz tick, and each secondary CPU enables its own local timer interrupt via `arch_intrenable` with the `IRQcntpns` vector.

⟨*constant* `Localctl` *(arm)* 547a⟩≡ (1010d)
```
Localctl    = 0x00,
```

⟨*constant* `Prescaler` *(arm)* 547b⟩≡ (1010d)
```
Prescaler   = 0x08,
```

⟨`clockinit()` *if many processors* 547c⟩≡ (280)
```
if(((cprdsc(0, CpID, CpIDfeat, 1) >> 16) & 0xF) != 0) {
    /* generic timer supported */
    if(cpu->cpuno == 0){
        *(ulong*)(ARMLOCAL + Localctl) = 0;             /* magic */
        *(ulong*)(ARMLOCAL + Prescaler) = 0x06aaaaab;   /* magic for 1 Mhz */
    }
    cpwrsc(0, CpTIMER, CpTIMERphys, CpTIMERphysctl, Imask);
}
```
Uses `ARMLOCAL-2649` 536e, `CpID` 535a, `CpIDfeat` 535b, `CpTIMER` 537a, `CpTIMERphys` 537b, `CpTIMERphysctl` 537b, `Imask-2667` 1011a, `Localctl-2650` 547a, `Prescaler-2651` 547b, `cprdsc()` 892a, and `cpwrsc()` 892b.

⟨`clockinit()` *if not cpu0* 547d⟩≡ (280)
```
else
    arch_intrenable(IRQcntpns, localclockintr, nil, 0, "clock");
```

⟨*function* `localclockintr` *(arm)* 547e⟩≡ (1011b)
```
static void
localclockintr(Ureg *ureg, void *)
{
    if(cpu->cpuno == 0)
        panic("cpu0: Unexpected local generic timer interrupt");
    cpwrsc(0, CpTIMER, CpTIMERphys, CpTIMERphysctl, Imask|Enable);
    timerintr(ureg, 0);
}
```
Uses `CpTIMER` 537a, `CpTIMERphys` 537b, `CpTIMERphysctl` 537b, `Enable-2666` 1011a, `Imask-2667` 1011a, `cpwrsc()` 892b, and `timerintr()` 252.

⟨*function* `arch_lcycles` *(raspberry pi2)(arm)* 547f⟩≡ (1012c)
```
TEXT arch_lcycles(SB), 1, $-4
    MRC CpSC, 0, R0, C(CpCLD), C(CpCLDcyc), 0
    RET
```

⟨`arch_timerset()` *if not cpu0(arm)* 547g⟩≡ (251a)
```
if(cpu->cpuno > 0){
    cpwrsc(0, CpTIMER, CpTIMERphys, CpTIMERphysval, period);
    cpwrsc(0, CpTIMER, CpTIMERphys, CpTIMERphysctl, Enable);
}
```
Uses `CpTIMER` 537a, `CpTIMERphys` 537b, `CpTIMERphysctl` 537b, `CpTIMERphysval` 537b, `Enable-2666` 1011a, and `cpwrsc()` 892b.

22.5.5 `arch_coherence()` (ARM)

⟨*function* `arm_arch_coherence` *(arm)* 547h⟩≡ (970)
```
TEXT arm_arch_coherence(SB), $-4
    BARRIERS
    RET
```

22.5.6 Cache adjustments

The Cortex-A7's cache line is 64 bytes (vs 32 on the ARM1176 in Pi 1), and the cache operations use the ARMv7 set/way interface rather than the simpler whole-cache MCR instructions. The [wholecache] routine iterates over all sets and ways at a given cache level, calling a function pointer (writeback, invalidate, or both) for each set/way combination. This is necessary because the ARMv7 architecture removed the "invalidate entire data cache" operation to prevent accidental data loss in multi-core systems.

⟨*constant CACHELINESZ(raspberry pi2)(ram)* 548a⟩≡ (1007c)
```
#define CACHELINESZ 64
```

⟨*instruction CPSID(arm)* 548b⟩≡ (890a)
```
#define CPSID   WORD    $0xf10c0080 /* intr disable: sets I bit */
```

⟨*type CpID CpIDid op1(arm)* 548c⟩≡ (888c)
```
/* CpIDid op1 values */
#define CpIDcsize   1           /* cache size (cortex) */
#define CpIDcssel   2           /* cache size select (cortex) */
```

⟨*function cachexxx(raspberry pi2)(ram)* 548d⟩≡ (1007c)
```
/*
 * 'single-element' cache operations.
 * in arm arch v7, they operate on all cache levels, so separate
 * l2 functions are unnecessary.
 */

TEXT cachedwbse(SB), $-4            /* D writeback SE */
    MOVW    R0, R2

    MOVW    CPSR, R3
    CPSID                   /* splhi */

    BARRIERS            /* force outstanding stores to cache */
    MOVW    R2, R0
    MOVW    4(FP), R1
    ADD R0, R1              /* R1 is end address */
    BIC $(CACHELINESZ-1), R0        /* cache line start */
_dwbse:
    MCR CpSC, 0, R0, C(CpCACHE), C(CpCACHEwb), CpCACHEse
    /* can't have a BARRIER here since it zeroes R0 */
    ADD $CACHELINESZ, R0
    CMP.S   R0, R1
    BGT _dwbse
    B   _wait

TEXT cachedwbinvse(SB), $-4         /* D writeback+invalidate SE */
    MOVW    R0, R2

    MOVW    CPSR, R3
    CPSID                   /* splhi */

    BARRIERS            /* force outstanding stores to cache */
    MOVW    R2, R0
    MOVW    4(FP), R1
    ADD R0, R1              /* R1 is end address */
    BIC $(CACHELINESZ-1), R0        /* cache line start */
_dwbinvse:
    MCR CpSC, 0, R0, C(CpCACHE), C(CpCACHEwbi), CpCACHEse
    /* can't have a BARRIER here since it zeroes R0 */
    ADD $CACHELINESZ, R0
```

```
    CMP.S   R0, R1
    BGT _dwbinvse
_wait:                      /* drain write buffer */
    BARRIERS

    MOVW    R3, CPSR            /* splx */
    RET

TEXT cachedinvse(SB), $-4           /* D invalidate SE */
    MOVW    R0, R2

    MOVW    CPSR, R3
    CPSID                   /* splhi */

    BARRIERS            /* force outstanding stores to cache */
    MOVW    R2, R0
    MOVW    4(FP), R1
    ADD R0, R1              /* R1 is end address */
    BIC $(CACHELINESZ-1), R0        /* cache line start */
_dinvse:
    MCR CpSC, 0, R0, C(CpCACHE), C(CpCACHEinvd), CpCACHEse
    /* can't have a BARRIER here since it zeroes R0 */
    ADD $CACHELINESZ, R0
    CMP.S   R0, R1
    BGT _dinvse
    B   _wait

/*
 * cortex arm arch v7 cache flushing and invalidation
 * shared by l.s and rebootcode.s
 */

TEXT cacheiinv(SB), $-4             /* I invalidate */
    MOVW    $0, R0
    MCR CpSC, 0, R0, C(CpCACHE), C(CpCACHEinvi), CpCACHEall /* ok on cortex */
    ISB
    RET

/*
 * set/way operators, passed a suitable set/way value in R0.
 */
TEXT cachedwb_sw(SB), $-4
    MCR CpSC, 0, R0, C(CpCACHE), C(CpCACHEwb), CpCACHEsi
    RET

TEXT cachedwbinv_sw(SB), $-4
    MCR CpSC, 0, R0, C(CpCACHE), C(CpCACHEwbi), CpCACHEsi
    RET

TEXT cachedinv_sw(SB), $-4
    MCR CpSC, 0, R0, C(CpCACHE), C(CpCACHEinvd), CpCACHEsi
    RET

    /* set cache size select */
TEXT setcachelvl(SB), $-4
    MCR CpSC, CpIDcssel, R0, C(CpID), C(CpIDidct), 0
    ISB
    RET

    /* return cache sizes */
```

```
TEXT getwayssets(SB), $-4
    MRC CpSC, CpIDcsize, R0, C(CpID), C(CpIDidct), 0
    RET

/*
 * l1 cache operations.
 * l1 and l2 ops are intended to be called from C, thus need save no
 * caller's regs, only those we need to preserve across calls.
 */

TEXT cachedwb(SB), $-4
    MOVW.W  R14, -8(R13)
    MOVW    $cachedwb_sw(SB), R0
    MOVW    $1, R8
    BL  wholecache(SB)
    MOVW.P  8(R13), R15

TEXT cachedwbinv(SB), $-4
    MOVW.W  R14, -8(R13)
    MOVW    $cachedwbinv_sw(SB), R0
    MOVW    $1, R8
    BL  wholecache(SB)
    MOVW.P  8(R13), R15

TEXT cachedinv(SB), $-4
    MOVW.W  R14, -8(R13)
    MOVW    $cachedinv_sw(SB), R0
    MOVW    $1, R8
    BL  wholecache(SB)
    MOVW.P  8(R13), R15

TEXT cacheuwbinv(SB), $-4
    MOVM.DB.W [R14], (R13)  /* save lr on stack */
    MOVW    CPSR, R1
    CPSID           /* splhi */

    MOVM.DB.W [R1], (R13)   /* save R1 on stack */

    BL  cachedwbinv(SB)
    BL  cacheiinv(SB)

    MOVM.IA.W (R13), [R1]   /* restore R1 (saved CPSR) */
    MOVW    R1, CPSR
    MOVM.IA.W (R13), [R14]  /* restore lr */
    RET

/*
 * l2 cache operations
 */

TEXT l2cacheuwb(SB), $-4
    MOVW.W  R14, -8(R13)
    MOVW    $cachedwb_sw(SB), R0
    MOVW    $2, R8
    BL  wholecache(SB)
    MOVW.P  8(R13), R15

TEXT l2cacheuwbinv(SB), $-4
    MOVW.W  R14, -8(R13)
    MOVW    CPSR, R1
```

```
    CPSID           /* splhi */

    MOVM.DB.W [R1], (R13)   /* save R1 on stack */

    MOVW    $cachedwbinv_sw(SB), R0
    MOVW    $2, R8
    BL  wholecache(SB)
    BL  l2cacheuinv(SB)

    MOVM.IA.W (R13), [R1]   /* restore R1 (saved CPSR) */
    MOVW    R1, CPSR
    MOVW.P  8(R13), R15

TEXT l2cacheuinv(SB), $-4
    MOVW.W  R14, -8(R13)
    MOVW    $cachedinv_sw(SB), R0
    MOVW    $2, R8
    BL  wholecache(SB)
    MOVW.P  8(R13), R15

/*
 * these shift values are for the Cortex-A8 L1 cache (A=2, L=6) and
 * the Cortex-A8 L2 cache (A=3, L=6).
 * A = log2(# of ways), L = log2(bytes per cache line).
 * see armv7 arch ref p. 1403.
 */
#define L1WAYSH 30
#define L1SETSH 6
#define L2WAYSH 29
#define L2SETSH 6

/*
 * callers are assumed to be the above l1 and l2 ops.
 * R0 is the function to call in the innermost loop.
 * R8 is the cache level (one-origin: 1 or 2).
 *
 * initial translation by 5c, then massaged by hand.
 */
TEXT wholecache+0(SB), $-4
    MOVW    R0, R1      /* save argument for inner loop in R1 */
    SUB $1, R8      /* convert cache level to zero origin */

    /* we may not have the MMU on yet, so map R1 to PC's space */
    BIC $KSEGM, R1  /* strip segment from address */
    MOVW    PC, R2      /* get PC's segment ... */
    AND $KSEGM, R2
    ORR R2, R1      /* combine them */

    /* drain write buffers */
    BARRIERS
    MCR CpSC, 0, R0, C(CpCACHE), C(CpCACHEwb), CpCACHEwait
    ISB

    MOVW    CPSR, R2
    MOVM.DB.W [R2,R14], (SP) /* save regs on stack */
    CPSID           /* splhi to make entire op atomic */

    /* get cache sizes */
    SLL $1, R8, R0  /* R0 = (cache - 1) << 1 */
    MCR CpSC, CpIDcssel, R0, C(CpID), C(CpIDidct), 0 /* set cache size select */
```

```
    ISB
    MRC CpSC, CpIDcsize, R0, C(CpID), C(CpIDidct), 0 /* get cache sizes */

    /* compute # of ways and sets for this cache level */
    SRA $3, R0, R5  /* R5 (ways) = R0 >> 3 */
    AND $1023, R5   /* R5 = (R0 >> 3) & MASK(10) */
    ADD $1, R5      /* R5 (ways) = ((R0 >> 3) & MASK(10)) + 1 */

    SRA $13, R0, R2 /* R2 = R0 >> 13 */
    AND $32767, R2  /* R2 = (R0 >> 13) & MASK(15) */
    ADD $1, R2      /* R2 (sets) = ((R0 >> 13) & MASK(15)) + 1 */

    /* precompute set/way shifts for inner loop */
    CMP $0, R8      /* cache == 1? */
    MOVW.EQ $L1WAYSH, R3    /* yes */
    MOVW.EQ $L1SETSH, R4
    MOVW.NE $L2WAYSH, R3    /* no */
    MOVW.NE $L2SETSH, R4

    /* iterate over ways */
    MOVW    $0, R7      /* R7: way */
outer:
    /* iterate over sets */
    MOVW    $0, R6      /* R6: set */
inner:
    /* compute set/way register contents */
    SLL R3, R7, R0  /* R0 = way << R3 (L?WAYSH) */
    ORR R8<<1, R0   /* R0 = way << L?WAYSH | (cache - 1) << 1 */
    ORR R6<<R4, R0  /* R0 = way<<L?WAYSH | (cache-1)<<1 |set<<R4 */

    BL  (R1)        /* call set/way operation with R0 */

    ADD $1, R6      /* set++ */
    CMP R2, R6      /* set >= sets? */
    BLT inner       /* no, do next set */

    ADD $1, R7      /* way++ */
    CMP R5, R7      /* way >= ways? */
    BLT outer       /* no, do next way */

    MOVM.IA.W (SP), [R2,R14] /* restore regs */
    MOVW    R2, CPSR    /* splx */

    /* drain write buffers */
    MCR CpSC, 0, R0, C(CpCACHE), C(CpCACHEwb), CpCACHEwait
    ISB
    RET
```

22.5.7 IRQ adjustments

The Pi 2 adds per-CPU local interrupts (timer, mailbox) that are separate from the shared peripheral interrupts. These local IRQs start at number 96 (`IRQlocal`) and are routed through per-CPU pending/enable registers in the ARM local register block, rather than the shared `Intregs` used for peripheral interrupts. Each `Vctl` gains a `cpu` field to record which processor the interrupt handler was registered on.

⟨`IRQ` *local cases(arm)* 552⟩≡ (174b)

```
  IRQlocal    = 96,
  IRQcntps    = IRQlocal + 0,
  IRQcntpns   = IRQlocal + 1,
```

Uses IRQlocal 552.

⟨Vctl *other fields(arm)* 553a⟩≡ (174c)
```
int cpu;
```

⟨irqenable() *if IRQlocal(arm)* 553b⟩≡ (176a)
```
if(irq >= IRQlocal){
    enable = (u32int*)(LOCALREGS + Localtimerint) + cpu->cpuno;
    v->reg = (u32int*)(LOCALREGS + Localintpending) + cpu->cpuno;
    v->mask = 1 << (irq - IRQlocal);
    v->cpu = cpu->cpuno;
}
```
Uses IRQlocal 552, LOCALREGS-2404 553g, Localintpending-2410 553e, and Localtimerint-2408 553d.

⟨irqenable() *if IRQlocal adjust enable(arm)* 553c⟩≡ (176a)
```
if(irq >= IRQlocal)
    *enable |= 1 << (irq - IRQlocal);
```
Uses IRQlocal 552.

⟨*constant* Localtimerint *(arm)* 553d⟩≡ (1002b)
```
Localtimerint   = 0x40,
```

⟨*constant* Localintpending *(arm)* 553e⟩≡ (1002b)
```
Localintpending = 0x60,
```

⟨*constant* Localmboxint *(arm)* 553f⟩≡ (1002b)
```
Localmboxint    = 0x50,
```

⟨*constant* LOCALREGS *(arm)* 553g⟩≡ (1002c)
```
#define LOCALREGS   (VIRTIO+IOSIZE)
```

22.5.8 Cycle counter adjustments

⟨_startpg() *enable cycle counter(raspberry pi2)(arm)* 553h⟩≡ (539f)
```
/*
 * enable cycle counter
 */
MOVW    $(1<<31), R1
MCR CpSC, 0, R1, C(CpCLD), C(CpCLDena), CpCLDenacyc
MOVW    $1, R1
MCR CpSC, 0, R1, C(CpCLD), C(CpCLDena), CpCLDenapmnc
```

⟨_startpg2() *enable cycle counter(raspberry pi2)(arm)* 553i⟩≡ (543)
```
/*
 * enable cycle counter
 */
MOVW    $(1<<31), R1
MCR CpSC, 0, R1, C(CpCLD), C(CpCLDena), CpCLDenacyc
MOVW    $1, R1
MCR CpSC, 0, R1, C(CpCLD), C(CpCLDena), CpCLDenapmnc
```

22.6 Advanced virtual memory

This section covers virtual memory features beyond the basic page fault handler presented in Chapter ??: page coloring (a cache optimization), physical memory segments (for memory-mapped I/O), and the `flushme` mechanism for maintaining instruction cache coherency when text pages are modified.

22.6.1 Page coloring

Page coloring is a technique to reduce cache conflicts in virtually-indexed caches. By assigning pages to "colors" based on their physical address, the kernel ensures that pages mapped at different virtual addresses do not compete for the same cache lines. On the ARM, the data cache is set-associative rather than direct-mapped, so coloring is less important—`NCOLOR` is set to 1, effectively disabling it. The machinery remains in the code because other architectures (like MIPS) benefit from it.

⟨`Page` *other fields* 554a⟩+≡ (108a) ◁214d 554g▷
```
// int [0..NCOLOR[
char color;   /* Cache coloring */
```

⟨`pageinit()` *other locals* 554b⟩≡ (283c)
```
int color = 0;
```

⟨`pageinit()` *set page color* 554c⟩≡ (283c)
```
p->color = color;
color = (color+1)%NCOLOR;
```

⟨*macro* `getpgcolor`*(arm)* 554d⟩≡ (974)
```
#define getpgcolor(a)   0
```

⟨*constant* `NCOLOR`*(arm)* 554e⟩≡ (1004b)
```
#define NCOLOR  1       /* 1 level cache, don't worry about VCE's */
```

Each page carries a per-CPU `cachectl` array that tracks whether the instruction cache needs to be invalidated when the page is mapped. `PG_TXTFLUSH` is set by `pio` whenever a text segment page is written (e.g., during `sysexec`). When `arch_putmmu` maps this page, it checks `cachectl` and calls `cacheiinv` if needed. The `flushme` flag on a `Segment` controls which segments participate in this mechanism.

⟨*enum* `Cachectl` 554f⟩≡ (855)
```
enum Cachectl {
    PG_NOFLUSH = 0,
    PG_TXTFLUSH = 1,  /* flush dcache and invalidate icache */

    PG_NEWCOL = 3,  /* page has been recolored */
};
```

⟨`Page` *other fields* 554g⟩+≡ (108a) ◁554a
```
// array<enum<Cachectl>>
char cachectl[MAXCPUS]; /* Cache flushing control for putmmu */
```

⟨`newpage()` *other locals* 554h⟩+≡ (110a) ◁210a 555a▷
```
int color;
// enum<Cachectl>
uchar ct;
```

⟨`newpage()` *find free page p, set color and ct* 554i⟩≡ (110a)
```
color = getpgcolor(va);

/* First try for our colour */
for(p = palloc.head; p; p = p->next)
     if(p->color == color)
         break;

ct = PG_NOFLUSH;
if(p == nil) {
    p = palloc.head;
    p->color = color;
    ct = PG_NEWCOL;
}
```

⟨newpage() *other locals* 555a⟩+≡ (110a) ◁554h
```
int i;
```

⟨newpage() *set cachectl using ct* 555b⟩≡ (110a)
```
for(i = 0; i < MAXCPUS; i++)
    p->cachectl[i] = ct;
```

⟨arch_putmmu() *if* PG_TXTFLUSH, *invalidate cache(arm)* 555c⟩≡ (186)
```
if(page->cachectl[cpu->cpuno] == PG_TXTFLUSH){
    /* pio() sets PG_TXTFLUSH whenever a text pg has been written */
    cacheiinv();
    page->cachectl[cpu->cpuno] = PG_NOFLUSH;
}
```
Uses PG_NOFLUSH 554f and PG_TXTFLUSH 554f.

⟨arch__userinit() *set cachectl(arm)* 555d⟩≡ (287d)
```
memset(pg->cachectl, PG_TXTFLUSH, sizeof(pg->cachectl));
```
Uses PG_TXTFLUSH 554f.

⟨Segment *other fields* 555e⟩+≡ (113b) ◁505d 556e▷
```
bool flushme; /* maintain icache for this segment */
```

⟨sysexec() *initialize other fields* 555f⟩≡ (196a)
```
ts->flushme = true;
```

⟨arch__userinit() *set flushme for stack segment* 555g⟩≡ (287d)
```
s->flushme = true;
```
Uses true.

⟨data2txt() *initialize other segment fields* 555h⟩≡ (196e)
```
ps->flushme = true;
```
Uses true.

⟨text2data() *initialize other fields* 555i⟩≡ (490)
```
ps->flushme = true;
```
Uses true.

⟨dupseg() *copy other fields* 555j⟩≡ (122a)
```
n->flushme = s->flushme;
```

⟨syssegflush() *set flushme* 555k⟩≡ (472c)
```
s->flushme = true;
```
Uses true.

⟨pio() *set cachectl if flushme segment* 555l⟩≡ (197d)
```
if(s->flushme)
    memset((*p)->cachectl, PG_TXTFLUSH, sizeof((*p)->cachectl));
```
Uses PG_TXTFLUSH 554f.

⟨procctlmemio() *set cachectl if flushme segment* 555m⟩≡ (488f)
```
/* Ensure the process sees text page changes */
if(s->flushme)
    memset(pg->cachectl, PG_TXTFLUSH, sizeof(pg->cachectl));
```
Uses PG_TXTFLUSH 554f.

22.6.2 Extra segment types

Beyond the standard text/data/bss/stack segments, Plan 9 supports `SG_SHARED` (memory shared between processes via [syssegattach]) and `SG_PHYSICAL` (direct mapping of physical memory, used for memory-mapped I/O like the VGA framebuffer). Both are handled specially during [dupseg]: they share the same `Segment` object rather than copying, since the whole point is that multiple processes see the same memory. For `SG_PHYSICAL`, the page fault handler ([fixfault]) creates pages whose physical addresses map directly to the device's physical memory, and marks them uncached (`PTEUNCACHED`) so reads and writes go straight to hardware.

⟨`Segtype` *other cases* 556a⟩≡ (112b) 556b▷
```
SG_SHARED = 04,
```

⟨`Segtype` *other cases* 556b⟩+≡ (112b) ◁556a
```
SG_PHYSICAL = 05,
```

⟨`dupseg()` *switch segment type cases* 556c⟩+≡ (121a) ◁122c
```
case SG_SHARED:
case SG_PHYSICAL:
    goto sameseg;
```

Uses `SG_PHYSICAL` and `SG_SHARED`.

⟨*struct* `Physseg` 556d⟩≡ (855)
```
struct Physseg
{
    ulong attr;     /* Segment attributes */
    char  *name;      /* Attach name */
    phys_addr pa;     /* Physical address */
    ulong size;     /* Maximum segment size in pages */
};
```

⟨`Segment` *other fields* 556e⟩+≡ (113b) ◁555e 559b▷
```
Physseg *pseg;
```

⟨*global* `physseg` 556f⟩≡ (864b)
```
/*
 * Attachable segment types
 */
static Physseg physseg[10] = {
    { .attr= SG_SHARED,
      .name= "shared",
      .pa= 0,
      .size= SEGMAXSIZE,
    },
    { .attr= SG_BSS,
      .name= "memory",
      .pa = 0,
      .size = SEGMAXSIZE,
    },
    { 0, 0, 0, 0},
};
```

Uses `SEGMAXSIZE` 556d, `SG_BSS`, and `SG_SHARED`.

⟨*global* `physseglock` 556g⟩≡ (864b)
```
static Lock physseglock;
```

⟨*function* addphysseg 557a⟩≡ (864b)
```
int
addphysseg(Physseg* new)
{
    Physseg *ps;

    /*
     * Check not already entered and there is room
     * for a new entry and the terminating null entry.
     */
    lock(&physseglock);
    for(ps = physseg; ps->name; ps++){
        if(strcmp(ps->name, new->name) == 0){
            unlock(&physseglock);
            return -1;
        }
    }
    if(ps-physseg >= nelem(physseg)-2){
        unlock(&physseglock);
        return -1;
    }

    *ps = *new;
    unlock(&physseglock);

    return 0;
}
```
Uses `lock()` 76d, `nelem` 853i, `physseg-1804` 556f, `physseglock-1805` 556g, and `unlock()` 78d.

⟨`freept()` *other locals* 557b⟩≡ (115b)
```
int ref;
Page *pt, **ptop;
```

⟨`freept()` `SG_PHYSICAL` *case* 557c⟩≡ (115b)
```
case SG_PHYSICAL:
    ptop = &p->pagetab[PAGETABSIZE];
    for(pte = p->pagetab; pte < ptop; pte++) {
        pt = *pte;
        if(pt == nil)
            continue;
        lock(pt);
        ref = --pt->ref;
        unlock(pt);
        if(ref == 0)
            free(pt); // because was smalloc'ed in fixfault
    }
    break;
```
Uses `SG_PHYSICAL`, `free()` 104a, `lock()` 76d, and `unlock()` 78d.

For `SG_PHYSICAL` segments, the page fault handler does not allocate real memory. Instead, it creates a `Page` whose physical address maps directly to the device's physical address (computed from the segment's base PA plus the offset). The page is marked `PTEUNCACHED` so that reads and writes bypass the CPU cache and go directly to the hardware.

⟨`fixfault()` `SG_PHYSICAL` *case* 557d⟩≡ (185)
```
case SG_PHYSICAL:
    if(*pte == nil) {
        new = smalloc(sizeof(Page));
        new->va = addr;
        new->pa = s->pseg->pa+(addr-s->base);
```

```
        new->ref = 1;
        *pte = new;
    }

    if (checkaddr && addr == addr2check)
        (*checkaddr)(addr, s, *pte);
    mmupte = PPN((*pte)->pa) |PTEWRITE|PTEUNCACHED|PTEVALID;
    (*pte)->modref = PG_MOD|PG_REF;
    break;
```

Uses PG_MOD 108b, PG_REF 108b, PTEUNCACHED 558a, PTEVALID 184, PTEWRITE 184, SG_PHYSICAL, addr2check 863b, checkaddr 863b, and smalloc() 104b.

⟨PTExxx *other cases* 558a⟩≡ (184)

```
#define PTEUNCACHED (1<<4)
```

⟨arch_putmmu() *if PTEUNCACHED, do not change x(arm)* 558b⟩≡ (186)

```
if(pa & PTEUNCACHED) {
}
```

Uses PTEUNCACHED 558a.

22.6.3 arch_kmap()

22.6.4 Extra checks

⟨*function* checkpagerefs 558c⟩≡ (863d)

```
void
checkpagerefs(void)
{
    int s;
    ulong i, np, nwrong;
    ulong *ref;

    np = palloc.user;
    ref = malloc(np*sizeof ref[0]);
    if(ref == nil){
        print("checkpagerefs: out of memory\n");
        return;
    }

    /*
     * This may not be exact if there are other processes
     * holding refs to pages on their stacks.  The hope is
     * that if you run it on a quiescent system it will still
     * be useful.
     */
    s = arch_splhi();
    lock(&palloc);
    arch_countpagerefs(ref, 0);
    portcountpagerefs(ref, 0);
    nwrong = 0;
    for(i=0; i<np; i++){
        if(palloc.pages[i].ref != ref[i]){
            iprint("page %#.8lux ref %d actual %lud\n",
                palloc.pages[i].pa, palloc.pages[i].ref, ref[i]);
            ref[i] = 1;
            nwrong++;
        }else
            ref[i] = 0;
```

```
    }
    arch_countpagerefs(ref, 1);
    portcountpagerefs(ref, 1);
    iprint("%lud mistakes found\n", nwrong);
    unlock(&palloc);
    arch_splx(s);
}
```

Uses iprint, lock() 76d, malloc() 103c, palloc 109b, portcountpagerefs() 559c, print, and unlock() 78d.

⟨*macro* arch_countpagerefs*(arm)* 559a⟩≡ (974)

```
// same signatures but optimized away
#define arch_countpagerefs(a, b)
```

⟨Segment *other fields* 559b⟩+≡ (113b) ◁556e

```
ulong mark;   /* portcountrefs */
```

⟨*function* portcountpagerefs 559c⟩≡ (863d)

```
void
portcountpagerefs(ulong *ref, int print)
{
    ulong i, j, k, ns, n;
    Page **pg, *entry;
    Proc *p;
    Pagetable *pt;
    Segment *s;

    /*
     * Pages in segments.  s->mark avoids double-counting.
     */
    n = 0;
    ns = 0;
    for(i=0; i<conf.nproc; i++){
        p = proctab(i);
        for(j=0; j<NSEG; j++){
            s = p->seg[j];
            if(s)
                s->mark = 0;
        }
    }
    for(i=0; i<conf.nproc; i++){
        p = proctab(i);
        for(j=0; j<NSEG; j++){
            s = p->seg[j];
            if(s == nil || s->mark++)
                continue;
            ns++;
            for(k=0; k<s->pagedirsize; k++){
                pt = s->pagedir[k];
                if(pt == nil)
                    continue;
                for(pg = pt->first; pg <= pt->last; pg++){
                    entry = *pg;
                    if(pagedout(entry))
                        continue;
                    if(print){
                        if(ref[pagenumber(entry)])
                            iprint("page %#.8lux in segment %#p\n", entry->pa, s);
                        continue;
                    }
                    if(ref[pagenumber(entry)]++ == 0)
```

```
                        n++;
                }
            }
        }
    }
    if(!print){
        iprint("%lud pages in %lud segments\n", n, ns);
        for(i=0; i<conf.nproc; i++){
            p = proctab(i);
            for(j=0; j<NSEG; j++){
                s = p->seg[j];
                if(s == nil)
                    continue;
                if(s->ref != s->mark){
                    iprint("segment %#p (used by proc %lud pid %lud) has bad ref count %lud actual %lud\n",
                        s, i, p->pid, s->ref, s->mark);
                }
            }
        }
    }
}
```

Uses NSEG 118c, conf 62b, iprint, pagedout 855, pagenumber() 560a, and proctab.

⟨*function* pagenumber 560a⟩≡ (863d)

```
ulong
pagenumber(Page *p)
{
    return p - palloc.pages;
}
```

Uses palloc 109b.

22.7 Fast interrupts (FIQ) (ARM)

ARM provides two interrupt levels: IRQ (normal) and FIQ (fast). FIQ has higher priority than IRQ and can preempt IRQ handlers. It also has its own banked registers (R8–R12), so a FIQ handler can execute without saving and restoring those registers, reducing latency. In this kernel, FIQ is used exclusively for USB interrupts (`IRQfiq = IRQusb`), and only one interrupt source can be routed to FIQ at a time. The `_vfiq` handler builds a full `Ureg` on the FIQ stack and calls the C function [fiq], which dispatches to the registered handler.

22.7.1 Exception PsrMfiq and associated processor mode

⟨PsrMode *other cases(arm)* 560b⟩+≡ (38) ◁161a

```
#define PsrMfiq     0x00000011
```

⟨trapnames() *other entries(arm)* 560c⟩≡ (161b)

```
[ PsrMfiq ] "fiq interrupt",
```

⟨PsrDisable *other cases(arm)* 560d⟩≡ (39a)

```
#define PsrDfiq     0x00000040      /* disable FIQ interrupts */
```

⟨*constant* FIQSTKTOP *(arm)* 560e⟩≡ (1005)

```
#define FIQSTKTOP   (KZERO+0x4000)      /* FIQ stack */
```

⟨arch__trapinit() *set stack for other exception/processor-modes* 560f⟩+≡ (278e) ◁279b

```
setr13(PsrMfiq, (u32int*)(FIQSTKTOP));
```

Uses FIQSTKTOP 560e and PsrMfiq 560b.

⟨*function* `_vfiq`*(arm)* 561a⟩≡ (1004a)
```
TEXT _vfiq(SB), 1, $-4          /* FIQ */
    CLREX
    MOVW    $PsrMfiq, R8        /* trap type */
    MOVW    SPSR, R9        /* interrupted psr */
    MOVW    R14, R10        /* interrupted pc */
    MOVM.DB.W [R8-R10], (R13)   /* save in ureg */
    MOVM.DB.S [R0-R14], (R13)   /* save interrupted regs */
    SUB $(15*4), R13
    MOVW    $setR12(SB), R12    /* Make sure we've got the kernel's SB loaded */

    /* get R(CPU) for this cpu */
    CPUID(R1)
    SLL $2, R1          /* convert to word index */
    MOVW    $cpus(SB), R2
    ADD R1, R2
    MOVW    (R2), R(CPU)        /* m = cpus[cpuid] */
    CMP $0, R(CPU)
    MOVW.EQ $CPUADDR, R(CPU)        /* paranoia: use CPUADDR if 0 */

    MOVW    8(R(CPU)), R(UP)        /* up */
    MOVW    R13, R0         /* first arg is pointer to ureg */
    SUB $(4*2), R13     /* space for argument+link (for debugger) */

    BL  fiq(SB)

    ADD $(8+4*15), R13      /* make r13 point to ureg->type */
    MOVW    8(R13), R14     /* restore link */
    MOVW    4(R13), R0      /* restore SPSR */
    MOVW    R0, SPSR        /* ... */
    MOVM.DB.S (R13), [R0-R14]   /* restore registers */
    ADD $8, R13         /* pop past ureg->{type+psr} */
    RFE             /* MOVM.IA.S.W (R13), [R15] */
```

22.7.2 `IRQfiq` and `vfiq`

⟨`IRQ` *other cases(arm)* 561b⟩≡ (174b)
```
IRQfiq      = IRQusb,   /* only one source can be FIQ */
```
Uses `IRQusb` 174b.

⟨`irqenable()` *if IRQfiq(arm)* 561c⟩≡ (176a)
```
if(irq == IRQfiq){
    ⟨irqenable() sanity check no previous FIQ(arm) 561f⟩
    vfiq = v;
    ip->FIQctl = Fiqenable | irq;
}
```
Uses `Fiqenable-2407` 561e, `IRQfiq` 561b, and `vfiq-2413` 561d.

⟨*global* `vfiq`*(arm)* 561d⟩≡ (1002c)
```
static Vctl *vfiq;
```

⟨*constant* `Fiqenable`*(arm)* 561e⟩≡ (1002b)
```
Fiqenable = 1<<7,
```

⟨`irqenable()` *sanity check no previous FIQ(arm)* 561f⟩≡ (561c)
```
assert((ip->FIQctl & Fiqenable) == 0);
assert((*enable & v->mask) == 0);
```
Uses `Fiqenable-2407` 561e and `assert` 853f.

22.7.3 `fiq()`

Unlike IRQ dispatch which walks a linked list of handlers, FIQ dispatch is trivial: there is at most one FIQ source, stored in the global `vfiq`. The `ureg->pc -= 4` adjustment is the standard ARM convention for interrupt return address correction.

⟨*function* `fiq`*(arm)* 562a⟩≡ (1002c)

```
/*
 * called direct from lexception.s to handle fiq interrupt.
 */
void
fiq(Ureg *ureg)
{
    Vctl *v;

    cpu->perf.intrts = arch_perfticks();
    v = vfiq;
    if(v == nil)
        panic("cpu%d: unexpected item in bagging area", cpu->cpuno);
    cpu->intr++;
    ureg->pc -= 4;
    arch_coherence();

    // Dispatch!
    v->f(ureg, v->a);

    arch_coherence();
    intrtime();
}
```

Uses `arch_coherence` and `vfiq-2413` 561d.

⟨*function* `splfhi`*(arm)* 562b⟩≡ (970)

```
TEXT splfhi(SB), 1, $-4
    MOVW    $(CPUADDR+4), R2        /* save caller pc in Mach */
    MOVW    R14, 0(R2)

    MOVW    CPSR, R0            /* turn off irqs and fiqs */
    ORR $(PsrDirq|PsrDfiq), R0, R1
    MOVW    R1, CPSR
    RET
```

22.8 Power management (ARM)

Power management on the Raspberry Pi is handled through the VideoCore mailbox interface (see [vcreq]). The kernel can turn individual peripherals on or off by sending a `TagSetpower` request. Currently only `PowerUsb` is used in practice.

⟨*enum* `Power`*(arm)* 562c⟩≡ (890b)

```
enum {
    PowerSd     = 0,
    PowerUart0,
    PowerUart1,
    PowerUsb,
    PowerI2c0,
    PowerI2c1,
    PowerI2c2,
    PowerSpi,
    PowerCcp2tx,
};
```

⟨*function* `setpower`*(arm)* 563a⟩≡ (897b)

```
/*
 * Power management
 */
void
setpower(int dev, bool on)
{
    u32int buf[2];

    buf[0] = dev;
    buf[1] = Powerwait | on;
    vcreq(TagSetpower, buf, sizeof buf, sizeof buf);
}
```

Uses `Powerwait-2367` 893d, `TagSetpower-2366` 893d, and `vcreq()` 895a.

⟨*function* `getpower`*(arm)* 563b⟩≡ (897b)

```
int
getpower(int dev)
{
    u32int buf[2];

    buf[0] = dev;
    buf[1] = 0;
    if(vcreq(TagGetpower, buf, sizeof buf[0], sizeof buf) != sizeof buf)
        return ERROR_NEG1;
    return buf[0] & 1;
}
```

Uses `ERROR_NEG1`, `TagGetpower-2365` 893d, and `vcreq()` 895a.

22.9 Alignements (ARM)

⟨*function* `arch_validalign`*(arm)* 563c⟩≡ (1008b)

```
/*
 * called in syscallfmt.c, sysfile.c, sysproc.c
 */
void
arch_validalign(uintptr addr, unsigned align)
{
    /*
     * Plan 9 is a 32-bit O/S, and the hardware it runs on
     * does not usually have instructions which move 64-bit
     * quantities directly, synthesizing the operations
     * with 32-bit move instructions. Therefore, the compiler
     * (and hardware) usually only enforce 32-bit alignment,
     * if at all.
     *
     * Take this out if the architecture warrants it.
     */
    if(align == sizeof(vlong))
        align = sizeof(long);

    /*
     * Check align is a power of 2, then addr alignment.
     */
    if((align != 0 && !(align & (align-1))) && !(addr & (align-1)))
        return;
    postnote(up, 1, "sys: odd address", NDebug);
    error(Ebadarg);
```

```
    /*NOTREACHED*/
  }
```

Uses Ebadarg 664d, NDebug 449a, postnote, and up.

22.10 Real-time scheduling

Plan 9 includes an Earliest Deadline First (EDF) real-time scheduler, configurable through /proc/<pid>/ctl commands (period, deadline, cost, admit, expel). An admitted EDF process is guaranteed a CPU time budget (C) within each period (T), with a deadline (D) by which the budget must be consumed. EDF processes run at priorities above all best-effort processes (PriEdf and PriRelease are above Npriq). The scheduler hooks into the normal scheduling path: [runproc] checks if the selected process is EDF and calls [edfrun] to start a deadline timer, [sched] skips priority adjustment for EDF processes, and [ready] calls [edfready] to handle release timing.

⟨Priority *cases for real-time priority* 564a⟩≡ (219b)

```
  PriRelease  = Npriq,  /* released edf processes */
  PriEdf    = Npriq+1,  /* active edf processes */
  PriExtra  = Npriq-1,  /* edf processes at high best-effort pri */
```

Uses Npriq 219a.

⟨runproc() *test for empty real-time scheduling queue* 564b⟩≡ (232d)

```
  runq[Nrq-1].head == nil && runq[Nrq-2].head == nil
```

Uses Nrq 221a and runq 221b.

⟨runproc() *test if p is a real-time process* 564c⟩≡ (225)

```
  if(edflock(p)){
      edfrun(p, rq == &runq[PriEdf]); /* start deadline timer and do admin */
      edfunlock();
  }
```

Uses PriEdf 564a, edflock() 568c, edfrun() 568c, edfunlock() 568c, and runq 221b.

⟨newproc() *set fields of* p 564d⟩+≡ (130a) ◁513c 601f▷

```
  p->edf = nil;
```

⟨updatecpu() *return if real-time process* 564e⟩≡ (239h)

```
  if(p->edf)
      return;
```

⟨schedinit() *locals* 564f⟩≡ (224f)

```
  Edf *e;
```

⟨schedinit() *optional real-time* edfrecord() 564g⟩≡ (228a)

```
  if((e = up->edf) && (e->flags & Admitted))
      edfrecord(up);
```

Uses Admitted 566g, edfrecord() 568c, and up.

⟨schedinit() *optional real-time* edfstop() 564h⟩≡ (154a)

```
  edfstop(up);
  if (up->edf)
      free(up->edf);
  up->edf = nil;
```

Uses edfstop() 568c, free() 104a, and up.

⟨ready() *optional* edfready() *for real-time scheduling* 565a⟩≡ (232a)

```
if(edfready(p)){
    arch_splx(s);
    return;
}
```

Uses edfready() 568c.

⟨syssleep() *optional* edfyield() *for real-time scheduling* 565b⟩≡ (255a)

```
if (up->edf && (up->edf->flags & Admitted))
    edfyield();
else
```

Uses Admitted 566g, edfyield() 568c, and up.

EDF parameters are set by writing commands to /proc/<pid>/ctl: "period 10ms", "cost 2ms", "deadline 8ms", then "admit" to activate the schedule. The kernel checks C ¡= D ¡= T before running the schedulability test.

⟨procctlreq() *optional real-time commands* 565c⟩≡ (620b)

```
/* real time */
case CMperiod:
    if(p->edf == nil)
        edfinit(p);
    if(e=parsetime(&time, cb->f[1]))    /* time in ns */
        error(e);
    edfstop(p);
    p->edf->T = time/1000;  /* Edf times are in microseconds */
    break;
case CMdeadline:
    if(p->edf == nil)
        edfinit(p);
    if(e=parsetime(&time, cb->f[1]))
        error(e);
    edfstop(p);
    p->edf->D = time/1000;
    break;
case CMcost:
    if(p->edf == nil)
        edfinit(p);
    if(e=parsetime(&time, cb->f[1]))
        error(e);
    edfstop(p);
    p->edf->C = time/1000;
    break;
case CMsporadic:
    if(p->edf == nil)
        edfinit(p);
    p->edf->flags |= Sporadic;
    break;
case CMdeadlinenotes:
    if(p->edf == nil)
        edfinit(p);
    p->edf->flags |= Sendnotes;
    break;
case CMadmit:
    if(p->edf == 0)
        error("edf params");
    if(e = edfadmit(p))
        error(e);
    break;
case CMextra:
```

```
        if(p->edf == nil)
            edfinit(p);
        p->edf->flags |= Extratime;
        break;
    case CMexpel:
        if(p->edf)
            edfstop(p);
        break;
```

Uses CMadmit-901 602b, CMcost-898 602b, CMdeadline-897 602b, CMdeadlinenotes-900 602b, CMexpel-903 602b, CMextra-902 602b, CMperiod-896 602b, CMsporadic-899 602b, Extratime 566g, Sendnotes 566g, Sporadic 566g, edfadmit() 568c, edfinit() 568c, edfstop() 568c, and parsetime() 620a.

⟨sched() *if real-time process then do nothing, do not reprioritize* 566a⟩≡ (238c)

```
if(p->edf) {
}
```

⟨pexit() *optional* edfstop() *for real-time scheduling* 566b⟩≡ (152c)

```
edfstop(up);
```

Uses edfstop() 568c and up.

⟨Proc *scheduling fields* 566c⟩+≡ (66) ◁239c 566d▷

⟨Proc *optional* edf *field for real-time scheduling* 566e⟩

⟨Proc *scheduling fields* 566d⟩+≡ (66) ◁566c 877a▷

```
ulong readytime;  /* time process came ready */
```

⟨Proc *optional* edf *field for real-time scheduling* 566e⟩≡ (566c)

```
// option<ref_own?<edf>>
Edf *edf; /* if non-null, real-time proc, edf contains scheduling params */
```

On a uniprocessor, if a real-time process spins on a lock held by a lower-priority process, the lower-priority process will never get to run and release the lock—classic priority inversion. The fix is to set the EDF deadline to "now", which causes the real-time process to be descheduled at the next scheduling point, giving the lock holder a chance to run. On a multiprocessor, this is not needed because the lock holder may be running on another CPU.

⟨lock() *optional priority-inversion for real-time process* 566f⟩≡ (76d)

```
if(conf.ncpu < 2 && up && up->edf && (up->edf->flags & Admitted)){
    /*
     * Priority inversion, yield on a uniprocessor; on a
     * multiprocessor, the other processor will unlock
     */
    print("inversion %#p pc %#lux proc %lud held by pc %#lux proc %lud\n",
        l, pc, up ? up->pid : 0, l->pc, l->p ? l->p->pid : 0);
    up->edf->d = todget(nil);   /* yield to process with lock */
}
```

Uses Admitted 566g, conf 62b, print, todget() 262, and up.

⟨*enum* edfflags 566g⟩≡ (868c)

```
enum EdfFlags
{
    /* Edf.flags field */
    Admitted    = 0x01,
    Sporadic    = 0x02,
    Yieldonblock    = 0x04,
    Sendnotes   = 0x08,
    Deadline    = 0x10,
    Yield     = 0x20,
    Extratime   = 0x40,
};
```

The `Edf` structure stores both the task parameters (period T, deadline D, cost C) and the per-instance state (current release time r, current deadline d, remaining slice S). It also embeds a `Timer` (via anonymous inclusion) so the scheduler can set deadline and release timers directly. The `testDelta`/`testtime` fields are used only during the admission test, which simulates the schedule to verify that adding a new real-time task will not cause any existing task to miss its deadline.

⟨*struct* `Edf` 567a⟩≡ (868c)

```
struct Edf {
    /* All times in µs */
    /* time intervals */
    long    D;    /* Deadline */
    long    Delta;    /* Inherited deadline */
    long    T;    /* period */
    long    C;    /* Cost */
    long    S;    /* Slice: time remaining in this period */
    /* times (only low-order bits of absolute time) */
    long    r;    /* (this) release time */
    long    d;    /* (this) deadline */
    long    t;    /* Start of next period, t += T at release */
    long    s;    /* Time at which this proc was last scheduled */

    /* for schedulability testing */
    long    testDelta;
    int   testtype; /* Release or Deadline */
    long    testtime;
    Proc    *testnext;

    /* other */
    // set<enum<edfflags>>
    ushort    flags;

    Timer;

    /* Stats */
    long    edfused;
    long    extraused;
    long    aged;
    ulong   periods;
    ulong   missed;
};
```

processes/edf.c

The EDF implementation centers on three interrupt-driven callbacks: [deadlineintr] fires when a process has consumed its time slice and forces a reschedule, [releaseintr] fires at the start of a new period to make the process eligible again, and [release] resets the per-period bookkeeping (remaining slice S, new deadline d). The [edfadmit] function performs an offline schedulability test before admitting a new real-time task: it simulates the combined schedule of all admitted tasks to verify that no deadlines will be missed. Sporadic tasks (those with the `Sporadic` flag) are released at most once per period, while periodic tasks are released at fixed intervals regardless of when they last ran.

⟨*edf.c debugging macro* 567b⟩≡ (568c)

```
enum {
    Dontprint = true,
};
#define DPRINT  if(!Dontprint) print
```

Uses `true`.

⟨*edf.c statistics* 568a⟩≡ (568c)

```
/* Statistics stuff */
ulong       edfnrun;
//ulong     nilcount;
//ulong     scheds;
//int       misseddeadlines;
```

⟨Procstate *cases* 568b⟩+≡ (67c) ◁484c

```
Waitrelease, // for real-time scheduling
```

⟨processes/edf.c 568c⟩≡

```
/* EDF scheduling */
⟨kernel basic includes 49a⟩

#include    <trace.h>

// in proc.c
extern Schedq   runq[Nrq];
extern ulong    delayedscheds;
extern int  nrdy;
extern ulong    runvec;
// in devcons.c
extern int panicking;
// used to be in edf.h
//unused: extern Lock edftestlock;  /* for atomic admitting/expelling */

⟨edf.c debugging macro 567b⟩

enum
{
    Maxsteps = 200 * 100 * 2, /* 100 periods of 200 procs */
};


static long now;    /* Low order 32 bits of time in µs */


⟨edf.c statistics 568a⟩

int     edfinited;

/* Edfschedlock protects modification of admission params */
QLock       edfschedlock;
static Lock thelock;

enum{
    Dl, /* Invariant for schedulability test: Dl < Rl */
    Rl,
};

static char *testschedulability(Proc*);
static Proc *qschedulability;

enum {
    Onemicrosecond =    1,
    Onemillisecond =    1000,
    Onesecond =     1000000,
    OneRound =      Onemillisecond/2,
};
```

```
static int
timeconv(Fmt *f)
{
    char buf[128], *sign;
    vlong t;

    buf[0] = 0;
    switch(f->r) {
    case 'U':
        t = va_arg(f->args, uvlong);
        break;
    case 't':           /* vlong in nanoseconds */
        t = va_arg(f->args, long);
        break;
    default:
        return fmtstrcpy(f, "(timeconv)");
    }
    if (t < 0) {
        sign = "-";
        t = -t;
    }
    else
        sign = "";
    if (t > Onesecond){
        t += OneRound;
        snprint(buf, sizeof buf, "%s%d.%.3ds", sign,
            (int)(t / Onesecond),
            (int)(t % Onesecond)/Onemillisecond);
    }else if (t > Onemillisecond)
        snprint(buf, sizeof buf, "%s%d.%.3dms", sign,
            (int)(t / Onemillisecond), (int)(t % Onemillisecond));
    else
        snprint(buf, sizeof buf, "%s%dµs", sign, (int)t);
    return fmtstrcpy(f, buf);
}

#ifdef EDFCYCLES
long edfcycles;
#endif

Edf*
edflock(Proc *p)
{
    Edf *e;

    if (p->edf == nil)
        return nil;
    ilock(&thelock);
    if((e = p->edf) && (e->flags & Admitted)){
        thelock.pc = getcallerpc(&p);
#ifdef EDFCYCLES
        edfcycles -= lcycles();
#endif
        now = arch_us();
        return e;
    }
    iunlock(&thelock);
    return nil;
}
```

```
void
edfunlock(void)
{

#ifdef EDFCYCLES
    edfcycles += lcycles();
#endif
    edfnrun++;
    iunlock(&thelock);
}

void
edfinit(Proc* p)
{
    if(!edfinited){
        fmtinstall('t', timeconv);
        edfinited++;
    }
    now = arch_us();
    DPRINT("%lud edfinit %lud[%s]\n", now, p->pid, statename[p->state]);
    p->edf = malloc(sizeof(Edf));
    if(p->edf == nil)
        error(Enomem);
    return;
}

static void
deadlineintr(Ureg*, Timer *t)
{
    /* Proc reached deadline */
    Proc *p;
    void (*pt)(Proc*, int, vlong);

    if(panicking || active.exiting)
        return;

    p = t->ta;
    now = arch_us();
    DPRINT("%lud deadlineintr %lud[%s]\n", now, p->pid, statename[p->state]);
    /* If we're interrupting something other than the proc pointed to by t->a,
     * we've already achieved recheduling, so we need not do anything
     * Otherwise, we must cause a reschedule, but if we call sched()
     * here directly, the timer interrupt routine will not finish its business
     * Instead, we cause the resched to happen when the interrupted proc
     * returns to user space
     */
    if(p == up){
        if(up->trace && (pt = proctrace))
            pt(up, SInts, 0);
        up->delaysched++;
        delayedscheds++;
    }
}

static void
release(Proc *p)
{
    /* Called with edflock held */
    Edf *e;
    void (*pt)(Proc*, int, vlong);
```

```
    long n;
    vlong nowns;

    e = p->edf;
    e->flags &= ~Yield;
    if(e->d - now < 0){
        e->periods++;
        e->r = now;
        if((e->flags & Sporadic) == 0){
            /*
             * Non sporadic processes stay true to their period;
             * calculate next release time.
             * Second test limits duration of while loop.
             */
            if((n = now - e->t) > 0){
                if(n < e->T)
                    e->t += e->T;
                else
                    e->t = now + e->T - (n % e->T);
            }
        }else{
            /* Sporadic processes may not be released earlier than
             * one period after this release
             */
            e->t = e->r + e->T;
        }
        e->d = e->r + e->D;
        e->S = e->C;
        DPRINT("%lud release %lud[%s], r=%lud, d=%lud, t=%lud, S=%lud\n",
            now, p->pid, statename[p->state], e->r, e->d, e->t, e->S);
        if(pt = proctrace){
            nowns = todget(nil);
            pt(p, SRelease, nowns);
            pt(p, SDeadline, nowns + 1000LL*e->D);
        }
    }else{
        DPRINT("%lud release %lud[%s], too late t=%lud, called from %#p\n",
            now, p->pid, statename[p->state], e->t, getcallerpc(&p));
    }
}

static void
releaseintr(Ureg*, Timer *t)
{
    Proc *p;
    Schedq *rq;

    if(panicking || active.exiting)
        return;

    p = t->ta;
    if((edflock(p)) == nil)
        return;
    DPRINT("%lud releaseintr %lud[%s]\n", now, p->pid, statename[p->state]);
    switch(p->state){
    default:
        edfunlock();
        return;
    case Ready:
        /* remove proc from current runq */
```

```
        rq = &runq[p->priority];
        if(dequeueproc(rq, p) != p){
            DPRINT("releaseintr: can't find proc or lock race\n");
            release(p); /* It'll start best effort */
            edfunlock();
            return;
        }
        p->state = Waitrelease;
        /* fall through */
    case Waitrelease:
        release(p);
        edfunlock();
        if(p->state == Wakeme){
            iprint("releaseintr: wakeme\n");
        }
        ready(p);
        if(up){
            up->delaysched++;
            delayedscheds++;
        }
        return;
    case Running:
        release(p);
        edfrun(p, 1);
        break;
    case Wakeme:
        release(p);
        edfunlock();
        if(p->trend)
            wakeup(p->trend);
        p->trend = nil;
        if(up){
            up->delaysched++;
            delayedscheds++;
        }
        return;
    }
    edfunlock();
}

void
edfrecord(Proc *p)
{
    long used;
    Edf *e;
    void (*pt)(Proc*, int, vlong);

    if((e = edflock(p)) == nil)
        return;
    used = now - e->s;
    if(e->d - now <= 0)
        e->edfused += used;
    else
        e->extraused += used;
    if(e->S > 0){
        if(e->S <= used){
            if(pt = proctrace)
                pt(p, SSlice, 0);
            DPRINT("%lud edfrecord slice used up\n", now);
            e->d = now;
```

```
            e->S = 0;
        }else
            e->S -= used;
    }
    e->s = now;
    edfunlock();
}

void
edfrun(Proc *p, int edfpri)
{
    Edf *e;
    void (*pt)(Proc*, int, vlong);
    long tns;

    e = p->edf;
    /* Called with edflock held */
    if(edfpri){
        tns = e->d - now;
        if(tns <= 0 || e->S == 0){
            /* Deadline reached or resources exhausted,
             * deschedule forthwith
             */
            p->delaysched++;
            delayedscheds++;
            e->s = now;
            return;
        }
        if(e->S < tns)
            tns = e->S;
        if(tns < 20)
            tns = 20;
        e->tns = 1000LL * tns;  /* µs to ns */
        if(e->tt == nil || e->tf != deadlineintr){
            DPRINT("%lud edfrun, deadline=%lud\n", now, tns);
        }else{
            DPRINT("v");
        }
        if(p->trace && (pt = proctrace))
            pt(p, SInte, todget(nil) + e->tns);
        e->tmode = Trelative;
        e->tf = deadlineintr;
        e->ta = p;
        timeradd(e);
    }else{
        DPRINT("<");
    }
    e->s = now;
}

char *
edfadmit(Proc *p)
{
    char *err;
    Edf *e;
    int i;
    Proc *r;
    void (*pt)(Proc*, int, vlong);
    long tns;
```

```
e = p->edf;
if (e->flags & Admitted)
    return "task state";    /* should never happen */

/* simple sanity checks */
if (e->T == 0)
    return "T not set";
if (e->C == 0)
    return "C not set";
if (e->D > e->T)
    return "D > T";
if (e->D == 0)  /* if D is not set, set it to T */
    e->D = e->T;
if (e->C > e->D)
    return "C > D";

qlock(&edfschedlock);
if (err = testschedulability(p)){
    qunlock(&edfschedlock);
    return err;
}
e->flags |= Admitted;

edflock(p);

if(p->trace && (pt = proctrace))
    pt(p, SAdmit, 0);

/* Look for another proc with the same period to synchronize to */
SET(r);
for(i=0; i<conf.nproc; i++) {
    r = proctab(i);
    if(r->state == Dead || r == p)
        continue;
    if (r->edf == nil || (r->edf->flags & Admitted) == 0)
        continue;
    if (r->edf->T == e->T)
            break;
}
if (i == conf.nproc){
    /* Can't synchronize to another proc, release now */
    e->t = now;
    e->d = 0;
    release(p);
    if (p == up){
        DPRINT("%lud edfadmit self %lud[%s], release now: r=%lud d=%lud t=%lud\n",
            now, p->pid, statename[p->state], e->r, e->d, e->t);
        /* We're already running */
        edfrun(p, 1);
    }else{
        /* We're releasing another proc */
        DPRINT("%lud edfadmit other %lud[%s], release now: r=%lud d=%lud t=%lud\n",
            now, p->pid, statename[p->state], e->r, e->d, e->t);
        p->ta = p;
        edfunlock();
        qunlock(&edfschedlock);
        releaseintr(nil, p);
        return nil;
    }
}else{
```

```
        /* Release in synch to something else */
        e->t = r->edf->t;
        if (p == up){
            DPRINT("%lud edfadmit self %lud[%s], release at %lud\n",
                now, p->pid, statename[p->state], e->t);
            edfunlock();
            qunlock(&edfschedlock);
            return nil;
        }else{
            DPRINT("%lud edfadmit other %lud[%s], release at %lud\n",
                now, p->pid, statename[p->state], e->t);
            if(e->tt == nil){
                e->tf = releaseintr;
                e->ta = p;
                tns = e->t - now;
                if(tns < 20)
                    tns = 20;
                e->tns = 1000LL * tns;
                e->tmode = Trelative;
                timeradd(e);
            }
        }
    }
    edfunlock();
    qunlock(&edfschedlock);
    return nil;
}

void
edfstop(Proc *p)
{
    Edf *e;
    void (*pt)(Proc*, int, vlong);

    if(e = edflock(p)){
        DPRINT("%lud edfstop %lud[%s]\n", now, p->pid, statename[p->state]);
        if(p->trace && (pt = proctrace))
            pt(p, SExpel, 0);
        e->flags &= ~Admitted;
        if(e->tt)
            timerdel(e);
        edfunlock();
    }
}

static int
yfn(void *)
{
    now = arch_us();
    return up->trend == nil || now - up->edf->r >= 0;
}

void
edfyield(void)
{
    /* sleep until next release */
    Edf *e;
    void (*pt)(Proc*, int, vlong);
    long n;
```

```
    if((e = edflock(up)) == nil)
        return;
    if(up->trace && (pt = proctrace))
        pt(up, SYield, 0);
    if((n = now - e->t) > 0){
        if(n < e->T)
            e->t += e->T;
        else
            e->t = now + e->T - (n % e->T);
    }
    e->r = e->t;
    e->flags |= Yield;
    e->d = now;
    if (up->tt == nil){
        n = e->t - now;
        if(n < 20)
            n = 20;
        up->tns = 1000LL * n;
        up->tf = releaseintr;
        up->tmode = Trelative;
        up->ta = up;
        up->trend = &up->sleepr;
        timeradd(up);
    }else if(up->tf != releaseintr)
        print("edfyield: surprise! %#p\n", up->tf);
    edfunlock();
    sleep(&up->sleepr, yfn, nil);
}

int
edfready(Proc *p)
{
    Edf *e;
    Schedq *rq;
    Proc *l, *pp;
    void (*pt)(Proc*, int, vlong);
    long n;

    if((e = edflock(p)) == nil)
        return 0;

    if(p->state == Wakeme && p->r){
        iprint("edfready: wakeme\n");
    }
    if(e->d - now <= 0){
        /* past deadline, arrange for next release */
        if((e->flags & Sporadic) == 0){
            /*
             * Non sporadic processes stay true to their period;
             * calculate next release time.
             */
            if((n = now - e->t) > 0){
                if(n < e->T)
                    e->t += e->T;
                else
                    e->t = now + e->T - (n % e->T);
            }
        }
        if(now - e->t < 0){
            /* Next release is in the future, schedule it */
```

```
        if(e->tt == nil || e->tf != releaseintr){
            n = e->t - now;
            if(n < 20)
                n = 20;
            e->tns = 1000LL * n;
            e->tmode = Trelative;
            e->tf = releaseintr;
            e->ta = p;
            timeradd(e);
            DPRINT("%lud edfready %lud[%s], release=%lud\n",
                now, p->pid, statename[p->state], e->t);
        }
        if(p->state == Running && (e->flags & (Yield|Yieldonblock)) == 0 && (e->flags & Extratime)){
            /* If we were running, we've overrun our CPU allocation
             * or missed the deadline, continue running best-effort at low priority
             * Otherwise we were blocked.  If we don't yield on block, we continue
             * best effort
             */
            DPRINT(">");
            p->basepri = PriExtra;
            p->fixedpri = 1;
            edfunlock();
            return 0;   /* Stick on runq[PriExtra] */
        }
        DPRINT("%lud edfready %lud[%s] wait release at %lud\n",
            now, p->pid, statename[p->state], e->t);
        p->state = Waitrelease;
        edfunlock();
        return 1;   /* Make runnable later */
    }
    DPRINT("%lud edfready %lud %s release now\n", now, p->pid, statename[p->state]);
    /* release now */
    release(p);
}
edfunlock();
DPRINT("^");
rq = &runq[PriEdf];
/* insert in queue in earliest deadline order */
lock(runq);
l = nil;
for(pp = rq->head; pp; pp = pp->rnext){
    if(pp->edf->d > e->d)
        break;
    l = pp;
}
p->rnext = pp;
if (l == nil)
    rq->head = p;
else
    l->rnext = p;
if(pp == nil)
    rq->tail = p;
rq->n++;
nrdy++;
runvec |= 1 << PriEdf;
p->priority = PriEdf;
p->readytime = cpu->ticks;
p->state = Ready;
unlock(runq);
if(p->trace && (pt = proctrace))
```

```
        pt(p, SReady, 0);
    return 1;
}


static void
testenq(Proc *p)
{
    Proc *xp, **xpp;
    Edf *e;

    e = p->edf;
    e->testnext = nil;
    if (qschedulability == nil) {
        qschedulability = p;
        return;
    }
    SET(xp);
    for (xpp = &qschedulability; *xpp; xpp = &xp->edf->testnext) {
        xp = *xpp;
        if (e->testtime - xp->edf->testtime < 0
        || (e->testtime == xp->edf->testtime && e->testtype < xp->edf->testtype)){
            e->testnext = xp;
            *xpp = p;
            return;
        }
    }
    assert(xp->edf->testnext == nil);
    xp->edf->testnext = p;
}

static char *
testschedulability(Proc *theproc)
{
    Proc *p;
    long H, G, Cb, ticks;
    int steps, i;

    /* initialize */
    DPRINT("schedulability test %lud\n", theproc->pid);
    qschedulability = nil;
    for(i=0; i<conf.nproc; i++) {
        p = proctab(i);
        if(p->state == Dead)
            continue;
        if ((p->edf == nil || (p->edf->flags & Admitted) == 0) && p != theproc)
            continue;
        p->edf->testtype = Rl;
        p->edf->testtime = 0;
        DPRINT("\tInit: edfenqueue %lud\n", p->pid);
        testenq(p);
    }
    H=0;
    G=0;
    for(steps = 0; steps < Maxsteps; steps++){
        p = qschedulability;
        qschedulability = p->edf->testnext;
        ticks = p->edf->testtime;
        switch (p->edf->testtype){
        case Dl:
```

```
            H += p->edf->C;
            Cb = 0;
            DPRINT("\tStep %3d, Ticks %lud, pid %lud, deadline, H += %lud → %lud, Cb = %lud\n",
                steps, ticks, p->pid, p->edf->C, H, Cb);
            if (H+Cb>ticks){
                DPRINT("not schedulable\n");
                return "not schedulable";
            }
            p->edf->testtime += p->edf->T - p->edf->D;
            p->edf->testtype = Rl;
            testenq(p);
            break;
        case Rl:
            DPRINT("\tStep %3d, Ticks %lud, pid %lud, release, G  %lud, C%lud\n",
                steps, ticks, p->pid, p->edf->C, G);
            if(ticks && G <= ticks){
                DPRINT("schedulable\n");
                return nil;
            }
            G += p->edf->C;
            p->edf->testtime += p->edf->D;
            p->edf->testtype = Dl;
            testenq(p);
            break;
        default:
            assert(0);
        }
    }
    DPRINT("probably not schedulable\n");
    return "probably not schedulable";
}
```

Uses Admitted 566g, DPRINT-2593 567b, Dead 67c, Dl-2597 568c, Extratime 566g, Maxsteps-2594 568c, OneRound-2603 568c, Onemillisecond-2601 568c, Onesecond-2602 568c, PriEdf 564a, PriExtra 564a, Ready 231c, Rl-2598 568c, Running 67c, Sporadic 566g, Trelative 248a, Waitrelease 568b, Wakeme 233d, Yield 566g, Yieldonblock 566g, active 65e, assert 853f, conf 62b, deadlineintr() 568c, delayedscheds 662b, dequeueproc() 222c, edfinited 568c, edflock() 568c, edfnrun 568a, edfrun() 568c, edfschedlock 568c, edfunlock() 568c, ilock() 79d, iprint, iunlock() 80b, lock() 76d, malloc() 103c, now-2595 568c, nrdy 222a, panicking 663a, print, proctab, proctrace 499a, qlock() 81f, qschedulability-2599 568c, qunlock() 82c, ready, release() 568c, releaseintr() 568c, runq 221b, runvec 221e, sleep, statename 67d, testenq() 568c, thelock-2596 568c, timeconv() 568c, timeradd() 250b, timerdel() 250e, todget() 262, unlock() 78d, up, wakeup, and yfn() 568c.

Tracing scheduling events

⟨enum Tevent *real-time scheduling events* 579⟩≡ (499b)

```
SAdmit, /* Edf admit */
SRelease, /* Edf release, waiting to be scheduled */
SYield,  /* blocked waiting for release */
SSlice,  /* slice exhausted */
SDeadline, /* proc's deadline */
SExpel,  /* Edf expel */
SInts,  /* Interrupt start */
SInte,  /* Interrupt end */
```

22.11 Fileservers

When a file server is mounted on a directory, the kernel forwards file operations (open, read, write, walk) to the server via the 9P protocol. The Mnt structure represents a connection to a file server: it holds the transport

channel (`c`), the negotiated message size, and a queue of pending RPCs. The key design is multiplexing: multiple processes can have files open on the same server, but 9P messages must be serialized onto a single connection. The `rip` (reader in progress) field gates access—only one process reads replies at a time, dispatching replies to other waiting processes via [mountmux].

22.11.1 Mnt

⟨*struct* Mnt 580a⟩≡ (804b)
```
struct Mnt
{
  Lock;
  /* references are counted using c->ref; channels on this mount point incref(c->mchan) == Mnt.c */
  Chan  *c;   /* Channel to file service */
  Proc  *rip;   /* Reader in progress */
  Mntrpc  *queue;   /* Queue of pending requests on this channel */
  ulong id;   /* Multiplexer id for channel check */
  Mnt *list;    /* Free list */
  int flags;    /* cache */
  int msize;    /* data + IOHDRSZ */
  char  *version; /* 9P version */
  Queue *q;   /* input queue */
};
```

⟨*struct* Mntalloc 580b⟩≡ (804b)
```
/*
 * References are managed as follows:
 * The channel to the server - a network connection or pipe - has one
 * reference for every Chan open on the server.  The server channel has
 * c->mux set to the Mnt used for muxing control to that server.  Mnts
 * have no reference count; they go away when c goes away.
 * Each channel derived from the mount point has mchan set to c,
 * and increfs/decrefs mchan to manage references on the server
 * connection.
 */
struct Mntalloc
{
    Mnt*    list;       /* Mount devices in use */
    Mnt*    mntfree;    /* Free list */
    Mntrpc* rpcfree;
    int nrpcfree;
    int nrpcused;
    ulong   id;
    ulong   tagmask[NMASK];

    // extra
    Lock;

};
```
Uses NMASK 580d.

⟨*global* mntalloc 580c⟩≡ (812c)
```
struct Mntalloc mntalloc;
```
Uses Mntalloc 580b.

⟨*constants tags* 580d⟩≡ (804b)
```
enum
{
    TAGSHIFT = 5,           /* ulong has to be 32 bits */
```

```
    TAGMASK = (1<<TAGSHIFT)-1,
    NMASK = (64*1024)>>TAGSHIFT,
};
```

Uses TAGSHIFT 580d.

⟨*function* freetag 581a⟩≡ (812c)

```
void
freetag(int t)
{
    mntalloc.tagmask[t>>TAGSHIFT] &= ~(1<<(t&TAGMASK));
}
```

Uses TAGMASK 580d, TAGSHIFT 580d, and mntalloc 580c.

⟨*function* mntfree 581b⟩≡ (812c)

```
void
mntfree(Mntrpc *r)
{
    if(r->b != nil)
        freeblist(r->b);
    lock(&mntalloc);
    if(mntalloc.nrpcfree >= 10){
        free(r->rpc);
        freetag(r->request.tag);
        free(r);
    }
    else{
        r->list = mntalloc.rpcfree;
        mntalloc.rpcfree = r;
        mntalloc.nrpcfree++;
    }
    mntalloc.nrpcused--;
    unlock(&mntalloc);
}
```

Uses free() 104a, freeblist() 439a, freetag() 581a, lock() 76d, mntalloc 580c, and unlock() 78d.

⟨*function* mntpntfree 581c⟩≡ (812c)

```
void
mntpntfree(Mnt *m)
{
    Mnt *f, **l;
    Queue *q;

    lock(&mntalloc);
    l = &mntalloc.list;
    for(f = *l; f; f = f->list) {
        if(f == m) {
            *l = m->list;
            break;
        }
        l = &f->list;
    }
    m->list = mntalloc.mntfree;
    mntalloc.mntfree = m;
    q = m->q;
    unlock(&mntalloc);

    qfree(q);
}
```

Uses lock() 76d, mntalloc 580c, qfree(), and unlock() 78d.

⟨*function* `muxclose` 582a⟩≡ (812c)

```
void
muxclose(Mnt *m)
{
    Mntrpc *q, *r;

    for(q = m->queue; q; q = r) {
        r = q->list;
        mntfree(q);
    }
    m->id = 0;
    free(m->version);
    m->version = nil;
    mntpntfree(m);
}
```

Uses `free()` 104a, `mntfree()` 581b, and `mntpntfree()` 581c.

22.11.2 MntRpc

Each in-flight 9P request is represented by an `Mntrpc`. It pairs the outgoing `Fcall` (request) with the space for the incoming reply, plus a `Rendez` for the requesting process to sleep on while waiting. The `flushed` pointer supports the 9P flush protocol: if a process is interrupted, the kernel sends a `Tflush` to cancel the original request.

⟨*struct* `Mntrpc` 582b⟩≡ (804b)

```
struct Mntrpc
{
  Chan* c;    /* Channel for whom we are working */
  Mntrpc* list;   /* Free/pending list */
  Fcall request;  /* Outgoing file system protocol message */
  Fcall   reply;    /* Incoming reply */
  Mnt*  m;    /* Mount device during rpc */
  Rendez  r;    /* Place to hang out */
  byte*  rpc;    /* I/O Data buffer */
  uint  rpclen;   /* len of buffer */
  Block *b;   /* reply blocks */
  char  done;   /* Rpc completed */
  uvlong  stime;    /* start time for mnt statistics */
  ulong reqlen;   /* request length for mnt statistics */
  ulong replen;   /* reply length for mnt statistics */
  Mntrpc* flushed;  /* message this one flushes */
};
```

22.11.3 Fcall

22.11.4 mntversion()

⟨`ChannelFlag` *cases* 582c⟩+≡ (294b) ◁322b 745d▷

```
CMSG  = 0x0002,   /* the message channel for a mount */
```

Before any 9P operations can proceed, the client and server must agree on a protocol version and maximum message size through the `Tversion`/`Rversion` exchange. This is the only 9P message that is not multiplexed—it must be the first message on a new connection. After the handshake, [mntversion] allocates the `Mnt` structure and attaches it to the channel via `c->mux`.

⟨*function* `mntversion` 582d⟩≡ (812c)

```
/*
```

```
 * Version is not multiplexed: message sent only once per connection.
 */
long
mntversion(Chan *c, char *version, int msize, int returnlen)
{
    Fcall f;
    uchar *msg;
    Mnt *m;
    char *v;
    long k, l;
    uvlong oo;
    char buf[128];

    qlock(&c->umqlock); /* make sure no one else does this until we've established ourselves */
    if(waserror()){
        qunlock(&c->umqlock);
        nexterror();
    }

    /* defaults */
    if(msize == 0)
        msize = MAXRPC;
    if(msize > c->iounit && c->iounit != 0)
        msize = c->iounit;
    v = version;
    if(v == nil || v[0] == '\0')
        v = VERSION9P;

    /* validity */
    if(msize < 0)
        error("bad iounit in version call");
    if(strncmp(v, VERSION9P, strlen(VERSION9P)) != 0)
        error("bad 9P version specification");

    m = c->mux;

    if(m != nil){
        qunlock(&c->umqlock);
        poperror();

        strecpy(buf, buf+sizeof buf, m->version);
        k = strlen(buf);
        if(strncmp(buf, v, k) != 0){
            snprint(buf, sizeof buf, "incompatible 9P versions %s %s", m->version, v);
            error(buf);
        }
        if(returnlen > 0){
            if(returnlen < k)
                error(Eshort);
            memmove(version, buf, k);
        }
        return k;
    }

    f.type = Tversion;
    f.tag = NOTAG;
    f.msize = msize;
    f.version = v;
    msg = malloc(8192+IOHDRSZ);
    if(msg == nil)
```

```
    exhausted("version memory");
if(waserror()){
    free(msg);
    nexterror();
}
k = convS2M(&f, msg, 8192+IOHDRSZ);
if(k == 0)
    error("bad fversion conversion on send");

lock(c);
oo = c->offset;
c->offset += k;
unlock(c);

l = devtab[c->type]->write(c, msg, k, oo);

if(l < k){
    lock(c);
    c->offset -= k - l;
    unlock(c);
    error("short write in fversion");
}

/* message sent; receive and decode reply */
k = devtab[c->type]->read(c, msg, 8192+IOHDRSZ, c->offset);
if(k <= 0)
    error("EOF receiving fversion reply");

lock(c);
c->offset += k;
unlock(c);

l = convM2S(msg, k, &f);
if(l != k)
    error("bad fversion conversion on reply");
if(f.type != Rversion){
    if(f.type == Rerror)
        error(f.ename);
    error("unexpected reply type in fversion");
}
if(f.msize > msize)
    error("server tries to increase msize in fversion");
if(f.msize<256 || f.msize>1024*1024)
    error("nonsense value of msize in fversion");
k = strlen(f.version);
if(strncmp(f.version, v, k) != 0)
    error("bad 9P version returned from server");

/* now build Mnt associated with this connection */
lock(&mntalloc);
m = mntalloc.mntfree;
if(m != 0)
    mntalloc.mntfree = m->list;
else {
    m = malloc(sizeof(Mnt));
    if(m == 0) {
        unlock(&mntalloc);
        exhausted("mount devices");
    }
}
```

```
    m->list = mntalloc.list;
    mntalloc.list = m;
    m->version = nil;
    kstrdup(&m->version, f.version);
    m->id = mntalloc.id++;
    m->q = qopen(10*MAXRPC, 0, nil, nil);
    m->msize = f.msize;
    unlock(&mntalloc);

    if(returnlen > 0){
        if(returnlen < k)
            error(Eshort);
        memmove(version, f.version, k);
    }

    poperror(); /* msg */
    free(msg);

    lock(m);
    m->queue = 0;
    m->rip = 0;

    c->flag |= CMSG;
    c->mux = m;
    m->c = c;
    unlock(m);

    poperror(); /* c */
    qunlock(&c->umqlock);

    return k;
}
```

Uses CMSG 582c, Eshort 664d, MAXRPC-782 812c, exhausted() 666c, free() 104a, kstrdup() 669b, lock() 76d, malloc() 103c, mntalloc 580c, nexterror, poperror 666d, qlock() 81f, qopen() 409a, qunlock() 82c, unlock() 78d, and waserror 666d.

22.11.5 mountrpc()

The [mountrpc]/[mountio] pair implements the core RPC mechanism. [mountio] serializes the `Fcall`, writes it to the server channel, then enters the reader gate: processes take turns being the "reader in progress" (`rip`). The reader calls [mntrpcread] to receive a reply, then [mountmux] to match the reply's tag to the correct waiting `Mntrpc` and wake up that process.

The reader gate deserves a picture, because it is what distinguishes `devmnt` from a naive RPC stub. Many processes can have an outstanding 9P request on the same `Mnt` at the same time, but only one of them can be blocked on `read(m->c)` at any given moment—the underlying network channel is a single byte stream. The `rip` ("reader in progress") field nominates that process; the others sleep on their own per-request `Rendez` and rely on whoever happens to be `rip` to deliver their reply and wake them up. When `rip` finishes its own call, `mntgate` elects a successor from the queue of sleeping callers:

```
client A      client B      kernel Mnt          server (over c)
--------      --------      ----------          --------------
    |             |             |                     |
mountrpc(A)       |             |                     |
    |             |             |                     |
    |---write----->|------Twrite Tattach(A)----------->
    |         (not rip, sleep on B->r)                 |
```

```
 (I am rip)        |              |                      |
     |             |              |                      |
     |----read m->c, recv reply-->|                      |
     |             |              |<---Rattach(B)-------|
     |---mountmux--+->tag==B ?                           |
     |             |                                     |
     |             | wakeup(&B->r)                       |
     |             |                                     |
     |             |                                     |
     |---read m->c, recv reply-->|<---Rattach(A)--------|
     |---mountmux--+->tag==A ?                           |
     |   done                                             |
     | mntgate(): pick next reader from queue            |
     |             |                                     |
     |             |-- becomes rip, resumes reading ----|
```

Three consequences of this design are worth noting. First, replies can arrive in any order—`mountmux` demultiplexes on the 9P `tag` field, so the server is free to reorder as long as it tags replies correctly. Second, the process that sent request X is not necessarily the process that will receive reply X: some other process may be `rip` when X's reply arrives, match the tag, and wake the original sender. Third, if `rip` is interrupted by a note, `mntgate` must hand the channel off cleanly, otherwise all the sleeping peers would deadlock. This is why the outer loop of `mountio` checks `waserror` and calls `mntgate` before re-entering.

⟨*function* `mountrpc` 586⟩≡ (746b)

```
void
mountrpc(Mnt *m, Mntrpc *r)
{
    char *sn, *cn;
    int t;

    r->reply.tag = 0;
    r->reply.type = Tmax;   /* can't ever be a valid message type */

    mountio(m, r);

    t = r->reply.type;
    switch(t) {
    case Rerror:
        error(r->reply.ename);
    case Rflush:
        error(Eintr);
    default:
        if(t == r->request.type+1)
            break;
        sn = "?";
        if(m->c->path != nil)
            sn = m->c->path->s;
        cn = "?";
        if(r->c != nil && r->c->path != nil)
            cn = r->c->path->s;
        print("mnt: proc %s %lud: mismatch from %s %s rep %#p tag %d fid %d T%d R%d rp %d\n",
            up->text, up->pid, sn, cn,
            r, r->request.tag, r->request.fid, r->request.type,
            r->reply.type, r->reply.tag);
        error(Emountrpc);
    }
}
```

Uses Emountrpc 664d, mountio() 587, print, and up.

⟨*function* mountio 587⟩≡ (746b)

```
void
mountio(Mnt *m, Mntrpc *r)
{
    int n;

    while(waserror()) {
        if(m->rip == up)
            mntgate(m);
        if(strcmp(up->errstr, Eintr) != 0){
            mntflushfree(m, r);
            nexterror();
        }
        r = mntflushalloc(r, m->msize);
    }

    lock(m);
    r->m = m;
    r->list = m->queue;
    m->queue = r;
    unlock(m);

    /* Transmit a file system rpc */
    if(m->msize == 0)
        panic("msize");
    n = convS2M(&r->request, r->rpc, m->msize);
    if(n < 0)
        panic("bad message type in mountio");
    if(devtab[m->c->type]->write(m->c, r->rpc, n, 0) != n)
        error(Emountrpc);
    r->stime = arch_fastticks(nil);
    r->reqlen = n;

    /* Gate readers onto the mount point one at a time */
    for(;;) {
        lock(m);
        if(m->rip == 0)
            break;
        unlock(m);
        sleep(&r->r, rpcattn, r);
        if(r->done){
            poperror();
            mntflushfree(m, r);
            return;
        }
    }
    m->rip = up;
    unlock(m);
    while(r->done == 0) {
        if(mntrpcread(m, r) < 0)
            error(Emountrpc);
        mountmux(m, r);
    }
    mntgate(m);
    poperror();
    mntflushfree(m, r);
}
```

Uses Emountrpc 664d, arch_fastticks, lock() 76d, mntflushalloc() 746b, mntflushfree() 746b, mntgate() 746b, mntrpcread() 746b, mountmux() 746b, nexterror, poperror 666d, rpcattn() 746b, sleep, unlock() 78d, up, and waserror 666d.

22.11.6 devmnt

The mount device (devmnt) implements the kernel's 9P client. Its methods translate Chan operations into 9P messages, send them via [mountrpc], and convert the replies back. [mntattach] sends Tattach to establish a session, and [mntwalk] sends Twalk to navigate the remote file tree.

mntattach()

⟨*function* mntattach 588⟩≡ (746b)

```
static Chan*
mntattach(char *muxattach)
{
    Mnt *m;
    Chan *c;
    Mntrpc *r;
    struct Bogus bogus;

    bogus = *((struct Bogus *)muxattach);
    c = bogus.chan;

    m = c->mux;

    if(m == nil){
        mntversion(c, nil, 0, 0);
        m = c->mux;
        if(m == nil)
            error(Enoversion);
    }

    c = mntchan();
    if(waserror()) {
        /* Close must not be called since it will
         * call mnt recursively
         */
        chanfree(c);
        nexterror();
    }

    r = mntralloc(0, m->msize);

    if(waserror()) {
        mntfree(r);
        nexterror();
    }

    r->request.type = Tattach;
    r->request.fid = c->fid;
    if(bogus.authchan == nil)
        r->request.afid = NOFID;
    else
        r->request.afid = bogus.authchan->fid;
    r->request.uname = up->user;
    r->request.aname = bogus.spec;
```

```
    mountrpc(m, r);

    c->qid = r->reply.qid;
    c->mchan = m->c;
    incref(m->c);
    c->mqid = c->qid;

    poperror(); /* r */
    mntfree(r);

    poperror(); /* c */

    if(bogus.flags&MCACHE)
        c->flag |= CCACHE;
    return c;
}
```

Uses Bogus 804b, CCACHE 745d, Enoversion 664d, MCACHE 343, chanfree() 296, incref() 87b, mntchan() 746b, mntfree() 581b, mntralloc() 746b, mntversion() 582d, mountrpc() 586, nexterror, poperror 666d, up, and waserror 666d.

mntwalk()

⟨*function* mntwalk 589⟩≡ (746b)

```
static Walkqid*
mntwalk(Chan *c, Chan *nc, char **name, int nname)
{
    int i, alloc;
    Mnt *m;
    Mntrpc *r;
    Walkqid *wq;

    if(nc != nil)
        print("mntwalk: nc != nil\n");
    if(nname > MAXWELEM)
        error("devmnt: too many name elements");
    alloc = 0;
    wq = smalloc(sizeof(Walkqid)+(nname-1)*sizeof(Qid));
    if(waserror()){
        if(alloc && wq->clone!=nil)
            cclose(wq->clone);
        free(wq);
        return nil;
    }

    alloc = 0;
    m = mntchk(c);
    r = mntralloc(c, m->msize);
    if(nc == nil){
        nc = devclone(c);
        /*
         * Until the other side accepts this fid, we can't mntclose it.
         * Therefore set type to 0 for now; rootclose is known to be safe.
         */
        nc->type = 0;
        alloc = 1;
    }
    wq->clone = nc;
    nc->flag |= c->flag&CCACHE;

    if(waserror()) {
```

```
            mntfree(r);
            nexterror();
        }
        r->request.type = Twalk;
        r->request.fid = c->fid;
        r->request.newfid = nc->fid;
        r->request.nwname = nname;
        memmove(r->request.wname, name, nname*sizeof(char*));

        mountrpc(m, r);

        if(r->reply.nwqid > nname)
            error("too many QIDs returned by walk");
        if(r->reply.nwqid < nname){
            if(alloc)
                cclose(nc);
            wq->clone = nil;
            if(r->reply.nwqid == 0){
                free(wq);
                wq = nil;
                goto Return;
            }
        }

        /* move new fid onto mnt device and update its qid */
        if(wq->clone != nil){
            if(wq->clone != c){
                wq->clone->type = c->type;
                wq->clone->mchan = c->mchan;
                incref(c->mchan);
            }
            if(r->reply.nwqid > 0)
                wq->clone->qid = r->reply.wqid[r->reply.nwqid-1];
        }
        wq->nqid = r->reply.nwqid;
        for(i=0; i<wq->nqid; i++)
            wq->qid[i] = r->reply.wqid[i];

        Return:
        poperror();
        mntfree(r);
        poperror();
        return wq;
    }
```

Uses CCACHE 745d, cclose, devclone() 429a, free() 104a, incref() 87b, mntchk() 746b, mntfree() 581b, mntralloc() 746b, mountrpc() 586, nexterror, poperror 666d, print, smalloc() 104b, and waserror 666d.

22.12 Network

22.13 Security

Plan 9 security is based on a per-process user identity string (`up->user`) rather than numeric UIDs. The "host owner" (`eve`) is the privileged user, analogous to root on Unix. The [iseve] function checks whether the current process is running as the host owner. Authentication is handled outside the kernel via the `fauth`/`fversion` system calls, which set up an authenticated channel to a file server—the kernel itself does not implement any authentication protocol but provides the plumbing for user-space auth servers. A notable security feature is

that any process can become "none" (the unprivileged identity) by writing to `/dev/user`, but no process can become anyone else. This one-way transition provides a simple sandboxing mechanism.

⟨*global* `eve` 591a⟩≡ (705b)
```
char    *eve;
```

⟨*function* `iseve` 591b⟩≡ (705b)
```
bool iseve(void) {
  return strcmp(eve, up->user) == 0;
}
```
Uses `eve` 591a and `up`.

⟨`systab` *security syscalls* 591c⟩≡ (57b)
```
[FAUTH]     sysfauth,
[FVERSION]  sysfversion,
```

The C prototypes are:

```
int   fversion(int fd, int bufsize, char *vers, int nvers);
int   fauth(int fd, char *aname);
```

⟨*global* `hostdomain` 591d⟩≡ (884a)
```
char    hostdomain[DOMLEN];
```

⟨*syscall fversion* 591e⟩≡ (884a)
```
long
sysfversion(ulong *arg)
{
    char *vers;
    uint arglen, m, msize;
    Chan *c;

    msize = arg[1];
    vers = (char*)arg[2];
    arglen = arg[3];
    validaddr(arg[2], arglen, true);
    /* check there's a NUL in the version string */
    if(arglen==0 || memchr(vers, 0, arglen)==0)
        error(Ebadarg);
    c = fdtochan(arg[0], ORDWR, false, true);
    if(waserror()){
        cclose(c);
        nexterror();
    }

    m = mntversion(c, vers, msize, arglen);

    cclose(c);
    poperror();
    return m;
}
```
Uses `Ebadarg` 664d, `ORDWR` 301c, `cclose`, `false`, `fdtochan()` 302d, `mntversion()` 582d, `nexterror`, `poperror` 666d, `true`, `validaddr()` 120b, and `waserror` 666d.

⟨*syscall fauth* 591f⟩≡ (884a)
```
long
sysfauth(ulong *arg)
{
    Chan *c, *ac;
    char *aname;
```

```
    int fd;

    validaddr(arg[1], 1, false);
    aname = validnamedup((char*)arg[1], true);
    if(waserror()){
        free(aname);
        nexterror();
    }
    c = fdtochan(arg[0], ORDWR, false, true);
    if(waserror()){
        cclose(c);
        nexterror();
    }

    ac = mntauth(c, aname);
    /* at this point ac is responsible for keeping c alive */
    poperror(); /* c */
    cclose(c);
    poperror(); /* aname */
    free(aname);

    if(waserror()){
        cclose(ac);
        nexterror();
    }

    fd = newfd(ac);
    if(fd < 0)
        error(Enofd);
    poperror(); /* ac */

    /* always mark it close on exec */
    ac->flag |= CCEXEC;

    return fd;
}
```

Uses CCEXEC 321b, Enofd 664d, ORDWR 301c, cclose, false, fdtochan() 302d, free() 104a, mntauth() 746b, newfd() 300a, nexterror, poperror 666d, true, validaddr() 120b, validnamedup() 307e, and waserror 666d.

⟨*function* userwrite 592a⟩≡ (884a)

```
/*
 *  called by devcons() for user device
 *
 *  anyone can become none
 */
long
userwrite(char *a, int n)
{
    if(n!=4 || strncmp(a, "none", 4)!=0)
        error(Eperm);
    kstrdup(&up->user, "none");
    up->basepri = PriNormal;
    return n;
}
```

Uses PriNormal 219b, kstrdup() 669b, and up.

⟨*function* hostdomainwrite 592b⟩≡ (884a)

```
long
hostdomainwrite(char *a, int n)
{
```

```
    char buf[DOMLEN];

    if(!iseve())
        error(Eperm);
    if(n >= DOMLEN)
        error(Ebadarg);
    memset(buf, 0, DOMLEN);
    strncpy(buf, a, n);
    if(buf[0] == 0)
        error(Ebadarg);
    memmove(hostdomain, buf, DOMLEN);
    return n;
}
```

Uses Ebadarg 664d, hostdomain 591d, and iseve() 591b.

22.14 Graphics

Chapter 23

Special Fileservers

23.1 Cmdtab

In this chapter, I will present the various special-purpose fileservers that the kernel provides: `/env/` for environment variables, `/proc/` for process inspection and control, `/fd/` for file descriptor access, `/sys/` for system configuration, and `/dev/` for miscellaneous devices. Each is implemented as a device driver following the `Dev` interface described in the Devices chapter. But first, the `Cmdtab` infrastructure—Plan 9's replacement for `ioctl()`. Instead of cryptic numeric ioctl codes, device control files accept human-readable text commands like "stop" or "hang". The `parsecmd()` function tokenizes the command string into a `Cmdbuf`, and `lookupcmd()` matches the first word against a table of `Cmdtab` entries, validating the argument count.

⟨*struct* `Cmdtab` 594a⟩≡ (867a)
```
struct Cmdtab
{
  int index;  /* used by client to switch on result */
  char  *cmd; /* command name */
  int narg; /* expected #args; 0 ==> variadic */
};
```

⟨*struct* `Cmdbuf` 594b⟩≡ (867a)
```
struct Cmdbuf
{
  char  *buf;
  char  **f;
  int nf;
};
```

The `parsecmd`[594c] function uses a single-allocation trick: the `Cmdbuf` struct, the array of string pointers, and the copy of the command text are all packed into one `smalloc` call. The layout in memory is: `Cmdbuf` header, then `nf char*` pointers (for `cb->f`), then the `n+1` byte string copy (for `cb->buf`). This avoids multiple allocations and simplifies cleanup—a single `free(cb)` releases everything.

⟨*function* `parsecmd` 594c⟩≡ (868a)
```
/*
 *  parse a command written to a device
 */
Cmdbuf*
parsecmd(char *p, int n)
{
    Cmdbuf *volatile cb;
    int nf;
    char *sp;

    nf = ncmdfield(p, n);
```

```
    /* allocate Cmdbuf plus string pointers plus copy of string including \0 */
    sp = smalloc(sizeof(*cb) + nf * sizeof(char*) + n + 1);
    cb = (Cmdbuf*)sp;
    cb->f = (char**)(&cb[1]);
    cb->buf = (char*)(&cb->f[nf]);

    if(up!=nil && waserror()){
        free(cb);
        nexterror();
    }
    memmove(cb->buf, p, n);
    if(up != nil)
        poperror();

    /* dump new line and null terminate */
    if(n > 0 && cb->buf[n-1] == '\n')
        n--;
    cb->buf[n] = '\0';

    cb->nf = tokenize(cb->buf, cb->f, nf-1);
    cb->f[cb->nf] = nil;

    return cb;
}
```

Uses `free()` 104a, `ncmdfield()` 596, `nexterror`, `poperror` 666d, `smalloc()` 104b, `up`, and `waserror` 666d.

⟨*function* `cmderror` 595a⟩≡ (868a)

```
/*
 * Reconstruct original message, for error diagnostic
 */
void
cmderror(Cmdbuf *cb, char *s)
{
    int i;
    char *p, *e;

    p = up->genbuf;
    e = p+ERRMAX-10;
    p = seprint(p, e, "%s \"", s);
    for(i=0; i<cb->nf; i++){
        if(i > 0)
            p = seprint(p, e, " ");
        p = seprint(p, e, "%q", cb->f[i]);
    }
    strcpy(p, "\"");
    error(up->genbuf);
}
```

Uses `ERRMAX` and `up`.

The `lookupcmd`[595b] function scans the command table linearly, matching on the first word of the parsed command. A table entry with `cmd` set to `"*"` acts as a wildcard catch-all, matching any command—useful for devices that accept freeform input. If `narg` is non-zero, the argument count must match exactly; a zero `narg` means the command is variadic.

⟨*function* `lookupcmd` 595b⟩≡ (868a)

```
/*
 * Look up entry in table
 */
Cmdtab*
```

```
lookupcmd(Cmdbuf *cb, Cmdtab *ctab, int nctab)
{
    int i;
    Cmdtab *ct;

    if(cb->nf == 0)
        error("empty control message");

    for(ct = ctab, i=0; i<nctab; i++, ct++){
        if(strcmp(ct->cmd, "*") !=0)    /* wildcard always matches */
        if(strcmp(ct->cmd, cb->f[0]) != 0)
            continue;
        if(ct->narg != 0 && ct->narg != cb->nf)
            cmderror(cb, Ecmdargs);
        return ct;
    }

    cmderror(cb, "unknown control message");
    return nil;
}
```

Uses `Ecmdargs` 664d and `cmderror()` 595a.

⟨*ncmdfield* 596⟩≡ (868a)

```
/*
 * Generous estimate of number of fields, including terminal nil pointer
 */
static int
ncmdfield(char *p, int n)
{
    int white, nwhite;
    char *ep;
    int nf;

    if(p == nil)
        return 1;

    nf = 0;
    ep = p+n;
    white = 1;  /* first text will start field */
    while(p < ep){
        nwhite = (strchr(" \t\r\n", *p++ & 0xFF) != 0); /* UTF is irrelevant */
        if(white && !nwhite)    /* beginning of field */
            nf++;
        white = nwhite;
    }
    return nf+1;    /* +1 for nil */
}
```

23.2 /env/

In Plan 9, environment variables are not an in-process data structure as in UNIX—they are files in the `/env/` filesystem. Each variable is a separate file; reading it gives the value, writing it sets it. This means any program in any language can access environment variables through the standard file API, and tools like `cat` and `echo` can inspect and modify them. The variables are grouped per-process in an `Egrp` (environment group), which can be shared or copied across `rfork`.

23.2.1 Egrp

An `Evalue` pairs a name with a byte array value. The value is typically a string but can be arbitrary bytes. The `Egrp` holds a dynamically-sized array of `Evalue` pointers, reference-counted so that processes sharing an environment group (via `RFENVG`) see the same variables. The `envcpy()` function performs a deep copy when a process gets its own environment group.

⟨*struct* `Evalue` 597a⟩≡ (804b)
```
struct Evalue
{
    // string
    char  *name;
    // option<array<byte>, usually a string but can be something else
    byte  *value;
    // nelem(Evalue.value),
    int len;

    Qid qid;
};
```

⟨*struct* `Egrp` 597b⟩≡ (804b)
```
struct Egrp
{
    // array<ref_counted<Evalue>>
    Evalue  **ent;
    // used entries, <= ment
    int nent;
    // nelem(Egrp.ent), malloc'ed entries
    int ment;

    ulong path; /* qid.path of next Evalue to be allocated */
    ulong vers; /* of Egrp */

    // extra
    Ref;
    RWlock;
};
```

⟨`Proc` *other fields* 597c⟩+≡ (66) ◁514d 601c▷
```
// ref_counted<Egrp>
Egrp  *egrp;    /* Environment group */
```

⟨*function* `closeegrp` 597d⟩≡ (812a)
```
void
closeegrp(Egrp *eg)
{
    int i;
    Evalue *e;

    if(decref(eg) == 0){
        for(i=0; i<eg->nent; i++){
            e = eg->ent[i];
            free(e->name);
            if(e->value)
                free(e->value);
            free(e);
        }
        free(eg->ent);
        free(eg);
    }
```

```
}
```

Uses decref() 87c and free() 104a.

⟨*function* envcpy 598a⟩≡ (734)

```
void
envcpy(Egrp *to, Egrp *from)
{
    int i;
    Evalue *ne, *e;

    rlock(from);
    to->ment = (from->nent+31)&~31;
    to->ent = smalloc(to->ment*sizeof(to->ent[0]));
    for(i=0; i<from->nent; i++){
        e = from->ent[i];
        ne = smalloc(sizeof(Evalue));
        ne->name = smalloc(strlen(e->name)+1);
        strcpy(ne->name, e->name);
        if(e->value){
            ne->value = smalloc(e->len);
            memmove(ne->value, e->value, e->len);
            ne->len = e->len;
        }
        ne->qid.path = ++to->path;
        to->ent[i] = ne;
    }
    to->nent = from->nent;
    runlock(from);
}
```

Uses rlock() 83e, runlock() 84b, and smalloc() 104b.

23.2.2 /env/

The ksetenv() function lets the kernel set environment variables during initialization. It works by opening a file through namec() using the device shorthand #e (or #ec for configuration variables) and writing the value to it—the kernel uses its own filesystem interface to talk to the environment device, treating itself as just another client.

⟨*function* ksetenv 598b⟩≡ (734)

```
/*
 *  to let the kernel set environment variables
 */
void
ksetenv(char *ename, char *eval, bool conf)
{
    Chan *c;
    char buf[2*KNAMELEN];

    snprint(buf, sizeof(buf), "#e%s/%s", conf?"c":"", ename);
    c = namec(buf, Acreate, OWRITE, 0600);
    devtab[c->type]->write(c, eval, strlen(eval), 0);
    cclose(c);
}
```

Uses Acreate 305b, KNAMELEN 853b, OWRITE, cclose, and namec() 305f.

⟨*global* envdevtab 599⟩≡ (734)

```
Dev envdevtab = {
    .dc       =    'e',
    .name     =    "env",

    .reset    =    devreset,
    .init     =    devinit,
    .shutdown =    devshutdown,
    .attach   =    envattach,
    .walk     =    envwalk,
    .stat     =    envstat,
    .open     =    envopen,
    .create   =    envcreate,
    .close    =    envclose,
    .read     =    envread,
    .bread    =    devbread,
    .write    =    envwrite,
    .bwrite   =    devbwrite,
    .remove   =    envremove,
    .wstat    =    devwstat,
};
```

Uses devbread() 434a, devbwrite() 434b, devinit() 810, devreset() 810, devshutdown() 810, devwstat() 810, envclose() 734, envcreate() 734, envopen() 734, envread() 734, envremove() 734, envstat() 734, envwalk() 734, and envwrite() 734.

23.3 /proc/

The /proc filesystem (device letter #p) is the most complex special fileserver in the kernel. Each running process appears as a directory /proc/\emphpid/ containing files for inspection and control: ctl for sending commands, mem for reading/writing the address space, status for state information, note for posting notifications, regs/fpregs for register access, wait for collecting child exit statuses, syscall for tracing output, and profile for tick profiling data. The qid encoding packs both the file type (5 bits) and the process slot number (26 bits) into the path, with the pid in the version field for consistency checking.

23.3.1 /proc/<pid>/ctl

Writing text commands to /proc/\emphpid/ctl controls a process: "kill", "stop", "start", "hang", "private", and others. The commands are parsed by procctlreq[620b] using the Cmdtab infrastructure. This is how kill works in Plan 9—it writes "kill" to the target's ctl file rather than sending a numeric signal.

23.3.2 /proc/<pid>/mem

Reading and writing /proc/\emphpid/mem provides direct access to the process's virtual address space. This is how debuggers like acid read variables, set breakpoints, and patch running code. The kernel translates the file offset into a virtual address, walks the process's segment table to find the corresponding page, and copies the data. If privatemem is set, access is denied.

23.3.3 /proc/<pid>/status

Reading /proc/\emphpid/status returns a fixed-format string containing the process name, user, state, CPU times, memory usage, and priority—this is what ps reads. The format is an array of fixed-width fields rather than whitespace-separated text, so ps can parse it without ambiguity even when process names contain spaces.

23.3.4 /proc/<pid>/args

The `args` file contains the process's command-line arguments, as they were at `exec` time. A process can also write to its own `args` file to change its display name in `ps`—useful for daemons that want to advertise their current state.

23.3.5 /proc/<pid>/wait

Reading `/proc/\emphpid/wait` is an alternative to the `await()` system call for collecting child exit statuses. The `qwaitr` lock ensures only one reader can block waiting for children at a time—if a second reader tries, it gets an "in use" error.

⟨`Proc` *hierarchy fields* 600a⟩+≡ (66) ◁151b
```
QLock qwaitr;
```

⟨`procread()` *cases* 600b⟩+≡ (605d) ◁507a 608a▷
```
case Qwait:
    if(!canqlock(&p->qwaitr))
        error(Einuse);

    if(waserror()) {
        qunlock(&p->qwaitr);
        nexterror();
    }

    ⟨procread() when Qwait case, sanity check process has children 600c⟩
    pid = p->pid;
    while(p->waitq == nil) {
        unlock(&p->exl);
        sleep(&p->waitr, haswaitq, p);
        if(p->pid != pid)
            error(Eprocdied);
        lock(&p->exl);
    }
    wq = p->waitq;
    p->waitq = wq->next;
    p->nwait--;
    unlock(&p->exl);

    qunlock(&p->qwaitr);
    poperror();

    n = snprint(a, n, "%d %lud %lud %lud %q",
        wq->w.pid,
        ⟨procread() Qwait case, snprint time field arguments 505c⟩
        wq->w.msg);
    free(wq);
    return n;
```
Uses `Eprocdied` 664d, `Qwait-876` 602a, `canqlock()` 83a, `free()` 104a, `haswaitq()` 151d, `lock()` 76d, `nexterror`, `poperror` 666d, `qunlock()` 82c, `sleep`, `unlock()` 78d, and `waserror` 666d.

⟨`procread()` *when Qwait case, sanity check process has children* 600c⟩≡ (600b)
```
// same in sysawait
lock(&p->exl);
if(up == p && p->nchild == 0 && p->waitq == 0) {
    unlock(&p->exl);
    error(Enochild);
}
```
Uses `Enochild` 664d, `lock()` 76d, `unlock()` 78d, and `up`.

⟨pwait() *lock qwaitr* 601a⟩≡ (150c)
```
if(!canqlock(&up->qwaitr))
    error(Einuse); // someone is reading /proc/pid/wait?
if(waserror()) {
    qunlock(&up->qwaitr);
    nexterror();
}
```
Uses canqlock() 83a, nexterror, qunlock() 82c, up, and waserror 666d.

⟨pwait() *unlock qwaitr* 601b⟩≡ (150c)
```
qunlock(&up->qwaitr);
poperror();
```
Uses poperror 666d, qunlock() 82c, and up.

23.3.6 Other /proc/<pid>/xxx

The remaining /proc files include note (for posting and reading notifications), noteid/notepg (for note group management), regs/fpregs/kregs (for register inspection), ns (for namespace listings), segment (for memory segment info), and text (which returns a channel to the executable binary, allowing debuggers to read symbol tables).

23.3.7 XXX

The procmode field controls the default permissions for files in a process's /proc directory. When procmode is 0, the per-file defaults from procdir apply. A process can also set privatemem to prevent anyone from reading its memory or posting notes—useful for processes handling sensitive data.

⟨Proc *other fields* 601c⟩+≡ (66) ◁597c 601e▷
```
ulong procmode; /* proc device default file mode */
```

⟨sysrfork() *when* RFPROC *propagate fields from up to p* 601d⟩+≡ (134b) ◁514e 601g▷
```
p->procmode = up->procmode;
```
Uses up.

⟨Proc *other fields* 601e⟩+≡ (66) ◁601c
```
bool privatemem; /* proc does not let anyone read mem */
```

⟨newproc() *set fields of* p 601f⟩+≡ (130a) ◁564d 618b▷
```
p->privatemem = false;
```
Uses false.

⟨sysrfork() *when* RFPROC *propagate fields from up to p* 601g⟩+≡ (134b) ◁601d
```
p->privatemem = up->privatemem;
```
Uses up.

⟨sysexec() *when hold debug lock, reset some fields* 601h⟩+≡ (141c) ◁450e
```
up->privatemem = false;
```

⟨procctlreq() *CMprivate case* 601i⟩≡ (620b)
```
case CMprivate:
    p->privatemem = true;
    break;
```
Uses CMprivate-885 602b and true.

⟨*devproc.c enum Qxxx* 602a⟩≡ (740b)

```
enum
{
    Qdir,
    Qtrace,

    Qargs,
    Qctl,
    Qfd,
    Qfpregs,
    Qkregs,
    Qmem,
    Qnote,
    Qnoteid,
    Qnotepg,
    Qns,
    Qproc,
    Qregs,
    Qsegment,
    Qstatus,
    Qtext,
    Qwait,
    Qprofile,
    Qsyscall,
};
```

⟨*devproc enum CMxxx* 602b⟩≡ (740b)

```
enum
{
    CMclose,
    CMclosefiles,
    CMhang,
    CMkill,
    CMnohang,
    CMnoswap,
    CMprivate,
    CMprofile,

    CMpri,
    CMfixedpri,
    CMwired,

    CMstart,
    CMstartstop,
    CMstartsyscall,
    CMstop,
    CMwaitstop,
    CMtrace,

    /* real time */
    CMperiod,
    CMdeadline,
    CMcost,
    CMsporadic,
    CMdeadlinenotes,
    CMadmit,
    CMextra,
    CMexpel,
    CMevent,
};
```

The `procdir` table defines the files visible in each `/proc/\emphpid/` directory, and the `proccmd` table below defines the text commands accepted by the `ctl` file. Files with permission 0000 use the process's `procmode` as their default; `status`, `fd`, and `ns` are world-readable (0444) because tools like `ps` need to read them for any process.

⟨*global* `procdir` 603a⟩≡ (740b)

```
/*
 * Status, fd, and ns are left fully readable (0444) because of their use in debugging,
 * particularly on shared servers.
 * Arguably, ns and fd shouldn't be readable; if you'd prefer, change them to 0000
 */
Dirtab procdir[] =
{
    "args",     {Qargs},    0,          0660,
    "ctl",      {Qctl},     0,          0000,
    "fd",       {Qfd},      0,          0444,
    "fpregs",   {Qfpregs},  sizeof(Arch_FPsave),     0000,
    "kregs",    {Qkregs},   sizeof(Ureg),       0400,
    "mem",      {Qmem},     0,          0000,
    "note",     {Qnote},    0,          0000,
    "noteid",   {Qnoteid},  0,          0664,
    "notepg",   {Qnotepg},  0,          0000,
    "ns",       {Qns},      0,          0444,
    "proc",     {Qproc},    0,          0400,
    "regs",     {Qregs},    sizeof(Ureg),       0000,
    "segment",  {Qsegment}, 0,          0444,
    "status",   {Qstatus},  STATSIZE,       0444,
    "text",     {Qtext},    0,          0000,
    "wait",     {Qwait},    0,          0400,
    "profile",  {Qprofile}, 0,          0400,
    "syscall",  {Qsyscall}, 0,          0400,
};
```

Uses `Qargs-861` 602a, `Qctl-862` 602a, `Qfd-863` 602a, `Qfpregs-864` 602a, `Qkregs-865` 602a, `Qmem-866` 602a, `Qnote-867` 602a, `Qnoteid-868` 602a, `Qnotepg-869` 602a, `Qns-870` 602a, `Qproc-871` 602a, `Qprofile-877` 602a, `Qregs-872` 602a, `Qsegment-873` 602a, `Qstatus-874` 602a, `Qsyscall-878` 602a, `Qtext-875` 602a, `Qwait-876` 602a, and `STATSIZE-907` 740b.

⟨*global* `proccmd` 603b⟩≡ (740b)

```
static
Cmdtab proccmd[] = {
    CMclose,        "close",        2,
    CMclosefiles,       "closefiles",       1,
    CMfixedpri,     "fixedpri",     2,
    CMhang,         "hang",         1,
    CMnohang,       "nohang",       1,
    CMnoswap,       "noswap",       1,
    CMkill,         "kill",         1,
    CMpri,          "pri",          2,
    CMprivate,      "private",      1,
    CMprofile,      "profile",      1,
    CMstart,        "start",        1,
    CMstartstop,        "startstop",        1,
    CMstartsyscall,     "startsyscall",     1,
    CMstop,         "stop",         1,
    CMwaitstop,     "waitstop",     1,
    CMwired,        "wired",        2,
    CMtrace,        "trace",        0,
    CMperiod,       "period",       2,
    CMdeadline,     "deadline",     2,
    CMcost,         "cost",         2,
    CMsporadic,     "sporadic",     1,
```

```
    CMdeadlinenotes,    "deadlinenotes",    1,
    CMadmit,        "admit",        1,
    CMextra,        "extra",        1,
    CMexpel,        "expel",        1,
    CMevent,        "event",        1,
};
```

Uses CMadmit-901 602b, CMclose-879 602b, CMclosefiles-880 602b, CMcost-898 602b, CMdeadline-897 602b, CMdeadlinenotes-900 602b, CMevent-904 602b, CMexpel-903 602b, CMextra-902 602b, CMfixedpri-888 602b, CMhang-881 602b, CMkill-882 602b, CMnohang-883 602b, CMnoswap-884 602b, CMperiod-896 602b, CMpri-887 602b, CMprivate-885 602b, CMprofile-886 602b, CMsporadic-899 602b, CMstart-890 602b, CMstartstop-891 602b, CMstartsyscall-892 602b, CMstop-893 602b, CMtrace-895 602b, CMwaitstop-894 602b, and CMwired-889 602b.

The qid encoding for `/proc` is cleverly packed: the low 5 bits of the path identify which file within the process directory (e.g., `Qctl`, `Qmem`, `Qstatus`), while the upper bits encode the process slot number. The version field carries the pid, so stale channels to dead-and-reallocated process slots can be detected. The `QID()`, `SLOT()`, and `PID()` macros extract these components.

⟨*devproc QXXX macros* 604a⟩≡ (740b)

```
/*
 * Qids are, in path:
 *   5 bits of file type (qids above)
 *  26 bits of process slot number + 1
 *       in vers,
 *  32 bits of pid, for consistency checking
 * If notepg, c->pgrpid.path is pgrp slot, .vers is noteid.
 */
#define QSHIFT  5   /* location in qid of proc slot # */

#define QID(q)      ((((ulong)(q).path) & ((1<<QSHIFT)-1)) >> 0)
#define SLOT(q)     (((((ulong)(q).path) & ~(1UL<<31)) >> QSHIFT) - 1)
#define PID(q)      ((q).vers)
#define NOTEID(q)   ((q).vers)
```

⟨*function* `proctab` 604b⟩≡ (877c)

```
Proc*
proc_proctab(int i)
{
    return &procalloc.arena[i];
}
```

Uses procalloc-2458 128d.

⟨`Chan` *union other fields* 604c⟩+≡ (68b) ◁459g 605b▷

```
Qid pgrpid;   /* for #p/notepg */
```

⟨*method procinit* 604d⟩≡ (740b)

```
static void
proc_init(void)
{
    if(conf.nproc >= (1<<(31-QSHIFT))-1)
        print("warning: too many procs for devproc\n");

    ⟨proc_init() other init 506a⟩
}
```

Uses QSHIFT-910 604a, conf 62b, and print.

⟨*global* `procdevtab` 605a⟩≡ (740b)

```
Dev procdevtab = {
    .dc       =    'p',
    .name     =    "proc",

    .reset    =    devreset,
    .init     =    proc_init,
    .shutdown =    devshutdown,
    .attach   =    procattach,
    .walk     =    procwalk,
    .stat     =    procstat,
    .open     =    procopen,
    .create   =    devcreate,
    .close    =    procclose,
    .read     =    procread,
    .bread    =    devbread,
    .write    =    procwrite,
    .bwrite   =    devbwrite,
    .remove   =    devremove,
    .wstat    =    procwstat,
};
```

Uses `devbread()` 434a, `devbwrite()` 434b, `devcreate()` 810, `devremove()` 810, `devreset()` 810, `devshutdown()` 810, `proc_init()` 604d, `procattach()` 740b, `procclose()` 617, `procopen()` 611, `procread()` 605d, `procstat()` 740b, `procwalk()` 740b, `procwrite()` 608c, and `procwstat()` 613.

⟨`Chan` *union other fields* 605b⟩+≡ (68b) ◁604c

```
ulong mid;    /* for ns in devproc */
```

⟨`procread()` *locals* 605c⟩+≡ (605d) ◁351b

```
/* NSEG*32 was too small for worst cases */
char *a, *sps, *srv, statbuf[NSEG*64];
int i, j, m, navail, ne, pid, rsize;
long l;
byte *rptr;
ulong offset;
Confmem *cm;
Proc *p;
Segment *s;
Ureg kur;
Waitq *wq;
```

Uses `NSEG` 118c.

The `procread()` function is a large switch on the file type extracted from the channel's qid. Most cases follow a similar pattern: validate that the target process still exists (comparing `p->pid` against `PID(c->qid)`), lock `p->debug` if needed, format the requested data into a buffer, and copy it to the caller. The `Qstatus` case assembles a fixed-format string with the process name, user, state, time counters, memory usage, and priority—this is what `/bin/ps` reads.

⟨*method procread* 605d⟩≡ (740b)

```
static long
procread(Chan *c, void *va, long n, vlong off)
{
    ⟨procread() locals 122d⟩

    a = va;
    offset = off;

    if(c->qid.type & QTDIR)
        return devdirread(c, a, n, 0, 0, procgen);
```

⟨procread() *Qtrace if* 501b⟩

```
p = proctab(SLOT(c->qid));
if(p->pid != PID(c->qid))
    error(Eprocdied);

switch(QID(c->qid)){
```

⟨procread() *cases* 122e⟩

```
case Qargs:
    qlock(&p->debug);
    j = procargs(p, up->genbuf, sizeof up->genbuf);
    qunlock(&p->debug);
    if(offset >= j)
        return 0;
    if(offset+n > j)
        n = j-offset;
    memmove(a, &up->genbuf[offset], n);
    return n;


case Qnote:
    qlock(&p->debug);
    if(waserror()){
        qunlock(&p->debug);
        nexterror();
    }
    if(p->pid != PID(c->qid))
        error(Eprocdied);
    if(n < 1)   /* must accept at least the '\0' */
        error(Etoosmall);
    if(p->nnote == 0)
        n = 0;
    else {
        m = strlen(p->note[0].msg) + 1;
        if(m > n)
            m = n;
        memmove(va, p->note[0].msg, m);
        ((char*)va)[m-1] = '\0';
        p->nnote--;
        memmove(p->note, p->note+1, p->nnote*sizeof(Note));
        n = m;
    }
    if(p->nnote == 0)
        p->notepending = false;
    poperror();
    qunlock(&p->debug);
    return n;

case Qproc:
    if(offset >= sizeof(Proc))
        return 0;
    if(offset+n > sizeof(Proc))
        n = sizeof(Proc) - offset;
    memmove(a, ((char*)p)+offset, n);
    return n;


case Qregs:
```

```
        rptr = (byte*)p->dbgreg;
        rsize = sizeof(Ureg);
        goto regread;
```

⟨procread() *Qfpregs case* 528b⟩

```
    regread:
        if(rptr == nil)
            error(Enoreg);
        if(offset >= rsize)
            return 0;
        if(offset+n > rsize)
            n = rsize - offset;
        memmove(a, rptr+offset, n);
        return n;


    // coupling with ps.c
/*
 * 0  text
 * 1  user
 * 2  state
 * 3  cputime[5]
 * 8  memory
 * 9 basepri
 * 10 pri
 */
    case Qstatus:
        if(offset >= STATSIZE)
            return 0;
        if(offset+n > STATSIZE)
            n = STATSIZE - offset;

        memset(statbuf, ' ', sizeof statbuf);
        readstr(0, statbuf+0*KNAMELEN, KNAMELEN-1, p->text);
        readstr(0, statbuf+1*KNAMELEN, KNAMELEN-1, p->user);
        sps = p->psstate;
        if(sps == nil)
            sps = statename[p->state];
        readstr(0, statbuf+2*KNAMELEN, 11, sps);

        j = 2*KNAMELEN + 12;
        ⟨procread() Qstatus case, time part 504c⟩

        /* ignore stack, which is mostly non-existent */
        l = 0;
        for(i=1; i<NSEG; i++){
            s = p->seg[i];
            if(s)
                l += s->top - s->base;
        }
        readnum(0, statbuf+j+NUMSIZE*5, NUMSIZE, l>>10, NUMSIZE);
        readnum(0, statbuf+j+NUMSIZE*6, NUMSIZE, p->basepri, NUMSIZE);
        readnum(0, statbuf+j+NUMSIZE*7, NUMSIZE, p->priority, NUMSIZE);
        memmove(a, statbuf+offset, n);
        return n;
```

```
        case Qnoteid:
            return readnum(offset, va, n, p->noteid, NUMSIZE);
        case Qfd:
            return procfds(p, va, n, offset);
        }
        error(Egreg);
        panic("procread: should not reach this point");
        return -1; // unreachable
    }
```

Uses Egreg 664d, Enoreg 664d, Eprocdied 664d, KNAMELEN 853b, NSEG 118c, NUMSIZE 867a, PID-913 604a, QID-911 604a, QTDIR 70b, Qargs-861 602a, Qfd-863 602a, Qnote-867 602a, Qnoteid-868 602a, Qproc-871 602a, Qregs-872 602a, Qstatus-874 602a, SLOT-912 604a, STATSIZE-907 740b, devdirread() 432c, false, nexterror, poperror 666d, procargs() 618c, procfds() 616b, procgen() 614d, proctab, qlock() 81f, qunlock() 82c, readnum() 670c, readstr() 670d, statename 67d, up, and waserror 666d.

⟨procread() *cases* 608a⟩+≡ (605d) ◁600b

```
    case Qkregs:
        memset(&kur, 0, sizeof(Ureg));
        arch_setkernur(&kur, p);
        rptr = (byte*)&kur;
        rsize = sizeof(Ureg);
        goto regread;
```

Uses Qkregs-865 602a.

⟨*function* arch_setkernur*(arm)* 608b⟩≡ (1008b)

```
    /* Give enough context in the ureg to produce a kernel stack for
     * a sleeping process
     */
    void
    arch_setkernur(Ureg* ureg, Proc* p)
    {
        ureg->pc = p->sched.pc;
        ureg->sp = p->sched.sp+4;
        ureg->r14 = PTR2UINT(sched);
    }
```

Uses PTR2UINT 56g and sched.

The procwrite() method handles writes to /proc/\emphpid/ctl (dispatched to procctlreq()), /proc/\emphpi (posting notifications to another process), /proc/\emphpid/noteid (changing the note group for signal broadcasting), /proc/\emphpid/regs (modifying registers of a stopped process), and /proc/\emphpid/mem (writing into the process's address space). The Qnotepg case is handled specially—it uses a remembered note group ID stored in the channel, not the target process's current group.

⟨*method procwrite* 608c⟩≡ (740b)

```
    static long
    procwrite(Chan *c, void *va, long n, vlong off)
    {
        int id, m;
        Proc *p, *t, *et;
        char *a, *arg, buf[ERRMAX];
        ulong offset = off;

        a = va;
        if(c->qid.type & QTDIR)
            error(Eisdir);

        p = proctab(SLOT(c->qid));

        /* Use the remembered noteid in the channel rather
```

```
 * than the process pgrpid
 */
if(QID(c->qid) == Qnotepg) {
    pgrpnote(NOTEID(c->pgrpid), va, n, NUser);
    return n;
}

qlock(&p->debug);
if(waserror()){
    qunlock(&p->debug);
    nexterror();
}
if(p->pid != PID(c->qid))
    error(Eprocdied);

switch(QID(c->qid)){

⟨procwrite() cases 488d⟩

case Qctl:
    procctlreq(p, va, n);
    break;

case Qargs:
    if(n == 0)
        error(Eshort);
    if(n >= ERRMAX)
        error(Etoobig);
    arg = malloc(n+1);
    if(arg == nil)
        error(Enomem);
    memmove(arg, va, n);
    m = n;
    if(arg[m-1] != 0)
        arg[m++] = 0;
    free(p->args);
    p->nargs = m;
    p->args = arg;
    p->setargs = true;
    break;


case Qregs:
    if(offset >= sizeof(Ureg))
        n = 0;
    else if(offset+n > sizeof(Ureg))
        n = sizeof(Ureg) - offset;
    if(p->dbgreg == nil)
        error(Enoreg);
    arch_setregisters(p->dbgreg, (char*)(p->dbgreg)+offset, va, n);
    break;

⟨procwrite Qfpregs case 528c⟩

case Qnote:
    if(p->kp)
        error(Eperm);
    if(n >= ERRMAX-1)
        error(Etoobig);
    memmove(buf, va, n);
```

```
            buf[n] = 0;
            if(!postnote(p, 0, buf, NUser))
                error("note not posted");
            break;
        case Qnoteid:
            id = atoi(a);
            if(id == p->pid) {
                p->noteid = id;
                break;
            }
            t = proctab(0);
            for(et = t+conf.nproc; t < et; t++) {
                if(t->state == Dead)
                    continue;
                if(id == t->noteid) {
                    if(strcmp(p->user, t->user) != 0)
                        error(Eperm);
                    p->noteid = id;
                    break;
                }
            }
            if(p->noteid != id)
                error(Ebadarg);
            break;

        default:
            pprint("unknown qid in procwrite\n");
            error(Egreg);
        }
        poperror();
        qunlock(&p->debug);
        return n;
    }
```

Uses Dead 67c, ERRMAX, Ebadarg 664d, Egreg 664d, Enoreg 664d, Eprocdied 664d, Eshort 664d, Etoobig 664d, NOTEID-914 604a, NUser 449a, PID-913 604a, QID-911 604a, QTDIR 70b, Qargs-861 602a, Qctl-862 602a, Qnote-867 602a, Qnoteid-868 602a, Qnotepg-869 602a, Qregs-872 602a, SLOT-912 604a, conf 62b, free() 104a, malloc() 103c, nexterror, pgrpnote() 610, poperror 666d, postnote, pprint, procctlreq() 620b, proctab, qlock() 81f, qunlock() 82c, true, and waserror 666d.

The `pgrpnote()` function broadcasts a note (notification) to all processes sharing a note group—Plan 9's equivalent of sending a signal to a process group. Writing to `/proc/\emphpid/notepg` triggers this, delivering the message to every non-kernel process with the matching `noteid`.

⟨*function* pgrpnote 610⟩≡ (875)

```
  void
  pgrpnote(ulong noteid, char *a, long n, int flag)
  {
      Proc *p, *ep;
      char buf[ERRMAX];

      if(n >= ERRMAX-1)
          error(Etoobig);

      memmove(buf, a, n);
      buf[n] = 0;
      p = proctab(0);
      ep = p+conf.nproc;
      for(; p < ep; p++) {
          if(p->state == Dead)
              continue;
          if(up != p && p->noteid == noteid && p->kp == 0) {
```

```
            qlock(&p->debug);
            if(p->pid == 0 || p->noteid != noteid){
                qunlock(&p->debug);
                continue;
            }
            if(!waserror()) {
                postnote(p, 0, buf, flag);
                poperror();
            }
            qunlock(&p->debug);
        }
    }
}
```

Uses Dead 67c, ERRMAX, Etoobig 664d, conf 62b, poperror 666d, postnote, proctab, qlock() 81f, qunlock() 82c, up, and waserror 666d.

The `procopen()` method enforces per-file access policies: `Qtext` and `Qprofile` are read-only, `Qnote` and `Qmem` are forbidden if the process has set `privatemem`, and the "none" user is blocked from most files via `nonone()`. The `Qtext` case is special—it returns a channel to the executable's binary file, not a synthetic proc file, allowing debuggers to read symbol tables.

⟨*method procopen* 611⟩≡ (740b)

```
static Chan*
procopen(Chan *c, int omode)
{
    Proc *p;
    Pgrp *pg;
    Chan *tc;
    int pid;

    if(c->qid.type & QTDIR)
        return devopen(c, omode, 0, 0, procgen);

    ⟨procopen() Qtrace if 500g⟩

    p = proctab(SLOT(c->qid));
    qlock(&p->debug);
    if(waserror()){
        qunlock(&p->debug);
        nexterror();
    }
    pid = PID(c->qid);
    if(p->pid != pid)
        error(Eprocdied);

    omode = openmode(omode);

    switch(QID(c->qid)){

    ⟨procopen() cases 352a⟩
    case Qtext:
        if(omode != OREAD)
            error(Eperm);
        tc = proctext(c, p);
        tc->offset = 0;
        qunlock(&p->debug);
        poperror();
        cclose(c);
        return tc;
```

```
case Qproc:
case Qkregs:
case Qsegment:
case Qprofile:
case Qfd:
    if(omode != OREAD)
        error(Eperm);
    break;

case Qnote:
    if(p->privatemem)
        error(Eperm);
    break;

case Qmem:
case Qctl:
    if(p->privatemem)
        error(Eperm);
    nonone(p);
    break;

case Qargs:
case Qnoteid:
case Qstatus:
case Qwait:
case Qregs:
case Qfpregs:
case Qsyscall:
    nonone(p);
    break;


case Qnotepg:
    nonone(p);
    pg = p->pgrp;
    if(pg == nil)
        error(Eprocdied);
    if(omode!=OWRITE || pg->pgrpid == 1)
        error(Eperm);
    c->pgrpid.path = pg->pgrpid+1;
    c->pgrpid.vers = p->noteid;
    break;

default:
    pprint("procopen %#lux\n", QID(c->qid));
    error(Egreg);
}

/* Affix pid to qid */
if(p->state != Dead)
    c->qid.vers = p->pid;

/* make sure the process slot didn't get reallocated while we were playing */
arch_coherence();
if(p->pid != pid)
    error(Eprocdied);

tc = devopen(c, omode, 0, 0, procgen);
qunlock(&p->debug);
poperror();
```

```
    return tc;
}
```

Uses Dead 67c, Egreg 664d, Eprocdied 664d, OREAD 853i, OWRITE, PID-913 604a, QID-911 604a, QTDIR 70b, Qargs-861 602a, Qctl-862 602a, Qfd-863 602a, Qfpregs-864 602a, Qkregs-865 602a, Qmem-866 602a, Qnote-867 602a, Qnoteid-868 602a, Qnotepg-869 602a, Qproc-871 602a, Qprofile-877 602a, Qregs-872 602a, Qsegment-873 602a, Qstatus-874 602a, Qsyscall-878 602a, Qtext-875 602a, Qwait-876 602a, SLOT-912 604a, arch_coherence, cclose, nexterror, nonone() 616a, openmode() 302b, poperror 666d, pprint, procgen() 614d, proctab, proctext() 618d, qlock() 81f, qunlock() 82c, and waserror 666d.

The `procwstat()` method allows changing the owner and default permissions of proc files. This is how `chmod +w /proc/96/mem` works to allow a debugger to write to a process's memory. Only the process owner or the host owner (`eve`) can change these settings.

⟨*method procwstat* 613⟩≡ (740b)

```
static int
procwstat(Chan *c, byte *db, int n)
{
    Proc *p;
    DirEntry *d;

    if(c->qid.type&QTDIR)
        error(Eperm);

    if(QID(c->qid) == Qtrace)
        return devwstat(c, db, n);

    p = proctab(SLOT(c->qid));
    nonone(p);
    d = nil;
    if(waserror()){
        free(d);
        qunlock(&p->debug);
        nexterror();
    }
    qlock(&p->debug);

    if(p->pid != PID(c->qid))
        error(Eprocdied);

    if(strcmp(up->user, p->user) != 0 && strcmp(up->user, eve) != 0)
        error(Eperm);

    d = smalloc(sizeof(DirEntry)+n);
    n = convM2D(db, n, &d[0], (char*)&d[1]);
    if(n == 0)
        error(Eshortstat);
    if(!emptystr(d->uid) && strcmp(d->uid, p->user) != 0){
        if(strcmp(up->user, eve) != 0)
            error(Eperm);
        else
            kstrdup(&p->user, d->uid);
    }
    /* p->procmode determines default mode for files in /proc */
    if(d->mode != ~0UL)
        p->procmode = d->mode&0777;

    poperror();
    free(d);
    qunlock(&p->debug);
```

```
    return n;
}
```

Uses Eprocdied 664d, Eshortstat 664d, PID-913 604a, QID-911 604a, QTDIR 70b, Qtrace-860 602a, SLOT-912 604a, devwstat() 810, emptystr() 670a, eve 591a, free() 104a, kstrdup() 669b, nexterror, nonone() 616a, poperror 666d, proctab, qlock() 81f, qunlock() 82c, smalloc() 104b, up, and waserror 666d.

⟨*function* procoffset 614a⟩≡ (740b)

```
static long
procoffset(long offset, char *va, int *np)
{
    if(offset > 0) {
        offset -= *np;
        if(offset < 0) {
            memmove(va, va+*np+offset, -offset);
            *np = -offset;
        }
        else
            *np = 0;
    }
    return offset;
}
```

⟨*function* procqidwidth 614b⟩≡ (740b)

```
static int
procqidwidth(Chan *c)
{
    char buf[32];

    return snprint(buf, sizeof buf, "%lud", c->qid.vers);
}
```

⟨*function* procfdprint 614c⟩≡ (740b)

```
int
procfdprint(Chan *c, int fd, int w, char *s, int ns)
{
    int n;

    if(w == 0)
        w = procqidwidth(c);
    n = snprint(s, ns, "%3d %.2s %C %4ld (%.16llux %*lud %.2ux) %5ld %8lld %s\n",
        fd,
        &"r w rw"[(c->mode&3)<<1],
        devtab[c->type]->dc, c->dev,
        c->qid.path, w, c->qid.vers, c->qid.type,
        c->iounit, c->offset, c->path->s);
    return n;
}
```

Uses procqidwidth() 614b.

The procgen() function generates directory entries for the /proc filesystem. At the top level (/proc/), it generates one entry per process (named by pid). Within each process directory, it generates entries from the procdir table. The permissions are computed from both the table defaults and the process's procmode, so a process can restrict access to its own proc files.

⟨*function* procgen 614d⟩≡ (740b)

```
static int
procgen(Chan *c, char *name, Dirtab *tab, int, int s, DirEntry *dp)
{
    Qid qid;
    Proc *p;
```

```
char *ename;
Segment *q;
ulong pid, path, perm, len;

if(s == DEVDOTDOT){
    mkqid(&qid, Qdir, 0, QTDIR);
    devdir(c, qid, "#p", 0, eve, 0555, dp);
    return 1;
}

if(c->qid.path == Qdir){
    if(s == 0){
        strcpy(up->genbuf, "trace");
        mkqid(&qid, Qtrace, -1, QTFILE);
        devdir(c, qid, up->genbuf, 0, eve, 0444, dp);
        return 1;
    }

    if(name != nil){
        /* ignore s and use name to find pid */
        pid = strtol(name, &ename, 10);
        if(pid==0 || ename[0]!='\0')
            return -1;
        s = procindex(pid);
        if(s < 0)
            return -1;
    }
    else if(--s >= conf.nproc)
        return -1;

    p = proctab(s);
    pid = p->pid;
    if(pid == 0)
        return 0;
    snprint(up->genbuf, sizeof up->genbuf, "%lud", pid);
    /*
     * String comparison is done in devwalk so name must match its formatted pid
    */
    if(name != nil && strcmp(name, up->genbuf) != 0)
        return -1;
    mkqid(&qid, (s+1)<<QSHIFT, pid, QTDIR);
    devdir(c, qid, up->genbuf, 0, p->user, DMDIR|0555, dp);
    return 1;
}
if(c->qid.path == Qtrace){
    strcpy(up->genbuf, "trace");
    mkqid(&qid, Qtrace, -1, QTFILE);
    devdir(c, qid, up->genbuf, 0, eve, 0444, dp);
    return 1;
}
if(s >= nelem(procdir))
    return -1;
if(tab)
    panic("procgen");

tab = &procdir[s];
path = c->qid.path&~(((1<<QSHIFT)-1));  /* slot component */

/* p->procmode determines default mode for files in /proc */
p = proctab(SLOT(c->qid));
```

```
    perm = tab->perm;
    if(perm == 0)
        perm = p->procmode;
    else    /* just copy read bits */
        perm |= p->procmode & 0444;

    len = tab->length;
    switch(QID(c->qid)) {
    case Qwait:
        len = p->nwait; /* incorrect size, but >0 means there's something to read */
        break;

    ⟨procgen() Qprofile case 507b⟩
    }

    mkqid(&qid, path|tab->qid.path, c->qid.vers, QTFILE);
    devdir(c, qid, tab->name, len, p->user, perm, dp);
    return 1;
}
```

Uses DEVDOTDOT 805, DMDIR, QID-911 604a, QSHIFT-910 604a, QTDIR 70b, QTFILE, Qdir-859 602a, Qtrace-860 602a, Qwait-876 602a, SLOT-912 604a, conf 62b, devdir() 427c, eve 591a, nelem 853i, procdir 603a, procindex() 129b, proctab, and up.

The nonone() function implements a security policy: the "none" user (the Plan 9 equivalent of "nobody") cannot inspect or control other processes. This prevents a compromised network server running as none from using /proc to attack other processes on the system.

⟨*function* nonone 616a⟩≡ (740b)

```
/*
 *  none can't read or write state on other
 *  processes.  This is to contain access of
 *  servers running as none should they be
 *  subverted by, for example, a stack attack.
 */
static void
nonone(Proc *p)
{
    if(p == up)
        return;
    if(strcmp(up->user, "none") != 0)
        return;
    if(iseve())
        return;
    error(Eperm);
}
```

Uses up.

The procfds[616b] function generates the content of /proc/\emphpid/fd, listing all open file descriptors with their mode, device type, qid, offset, and path. It formats into a local buffer first to avoid holding the Fgrp lock while copying to user space—a common pattern in the kernel to minimize lock hold times. The two-pass approach over the fd array (first to compute the widest qid field, then to format) ensures aligned columns in the output.

⟨*function* procfds 616b⟩≡ (740b)

```
static int
procfds(Proc *p, char *va, int count, long offset)
{
    Fgrp *f;
    Chan *c;
    char buf[256];
```

```
    int n, i, w, ww;
    char *a;

    /* print to buf to avoid holding fgrp lock while writing to user space */
    if(count > sizeof buf)
        count = sizeof buf;
    a = buf;

    qlock(&p->debug);
    f = p->fgrp;
    if(f == nil){
        qunlock(&p->debug);
        return 0;
    }
    lock(f);
    if(waserror()){
        unlock(f);
        qunlock(&p->debug);
        nexterror();
    }

    n = readstr(0, a, count, p->dot->path->s);
    n += snprint(a+n, count-n, "\n");
    offset = procoffset(offset, a, &n);
    /* compute width of qid.path */
    w = 0;
    for(i = 0; i <= f->maxfd; i++) {
        c = f->fd[i];
        if(c == nil)
            continue;
        ww = procqidwidth(c);
        if(ww > w)
            w = ww;
    }
    for(i = 0; i <= f->maxfd; i++) {
        c = f->fd[i];
        if(c == nil)
            continue;
        n += procfdprint(c, i, w, a+n, count-n);
        offset = procoffset(offset, a, &n);
    }
    unlock(f);
    qunlock(&p->debug);
    poperror();

    /* copy result to user space, now that locks are released */
    memmove(va, buf, n);

    return n;
}
```

Uses `lock()` 76d, `nexterror`, `poperror` 666d, `procfdprint()` 614c, `procoffset()` 614a, `procqidwidth()` 614b, `qlock()` 81f, `qunlock()` 82c, `readstr()` 670d, `unlock()` 78d, and `waserror` 666d.

⟨*method procclose* 617⟩≡ (740b)

```
static void
procclose(Chan* c)
{
    ⟨procclose() Qtrace if 501a⟩
    ⟨procclose() hooks 352b⟩
}
```

⟨Proc *debugger fields* 618a⟩+≡ (66) ◁498b
```
bool setargs;
```

⟨newproc() *set fields of* p 618b⟩+≡ (130a) ◁601f 665c▷
```
p->setargs = false;
```
Uses false.

⟨*function* procargs 618c⟩≡ (740b)
```
static int
procargs(Proc *p, char *buf, int nbuf)
{
    int j, k, m;
    char *a;
    int n;

    a = p->args;
    if(p->setargs){
        snprint(buf, nbuf, "%s [%s]", p->text, p->args);
        return strlen(buf);
    }
    n = p->nargs;
    for(j = 0; j < nbuf - 1; j += m){
        if(n <= 0)
            break;
        if(j != 0)
            buf[j++] = ' ';
        m = snprint(buf+j, nbuf-j, "%q",  a);
        k = strlen(a) + 1;
        a += k;
        n -= k;
    }
    return j;
}
```

The proctext[618d] function is unusual: instead of returning synthetic data, it returns a clone of the channel to the process's executable binary. This lets debuggers read the symbol table and text segment directly from the on-disk image, which is more efficient than reading through /proc/\emphpid/mem and avoids issues with demand-paged text segments that may not be fully resident.

⟨*function* proctext 618d⟩≡ (740b)
```
Chan*
proctext(Chan *c, Proc *p)
{
    Chan *tc;
    KImage *i;
    Segment *s;

    s = p->seg[TSEG];
    if(s == 0)
        error(Enonexist);
    if(p->state==Dead)
        error(Eprocdied);

    lock(s);
    i = s->image;
    if(i == 0) {
        unlock(s);
        error(Eprocdied);
    }
    unlock(s);
```

```
    lock(i);
    if(waserror()) {
        unlock(i);
        nexterror();
    }

    tc = i->c;
    if(tc == 0)
        error(Eprocdied);

    if(incref(tc) == 1 || (tc->flag&COPEN) == 0 || tc->mode!=OREAD) {
        cclose(tc);
        error(Eprocdied);
    }

    if(p->pid != PID(c->qid)){
        cclose(tc);
        error(Eprocdied);
    }

    unlock(i);
    poperror();

    return tc;
}
```

Uses COPEN 294b, Dead 67c, Eprocdied 664d, OREAD 853i, PID-913 604a, TSEG 118c, cclose, incref() 87b, lock() 76d, nexterror, poperror 666d, unlock() 78d, and waserror 666d.

⟨*function* procctlcloseone 619a⟩≡ (740b)

```
static void
procctlcloseone(Proc *p, Fgrp *f, int fd)
{
    Chan *c;

    c = f->fd[fd];
    if(c == nil)
        return;
    f->fd[fd] = nil;
    unlock(f);
    qunlock(&p->debug);
    cclose(c);
    qlock(&p->debug);
    lock(f);
}
```

Uses cclose, lock() 76d, qlock() 81f, qunlock() 82c, and unlock() 78d.

⟨*function* procctlclosefiles 619b⟩≡ (740b)

```
void
procctlclosefiles(Proc *p, bool all, int fd)
{
    int i;
    Fgrp *f;

    f = p->fgrp;
    if(f == nil)
        error(Eprocdied);

    lock(f);
    f->ref++;
```

```
    if(all)
        for(i = 0; i < f->maxfd; i++)
            procctlcloseone(p, f, i);
    else
        procctlcloseone(p, f, fd);
    unlock(f);
    closefgrp(f);
}
```

Uses Eprocdied 664d, closefgrp() 299b, lock() 76d, procctlcloseone() 619a, and unlock() 78d.

⟨*function* parsetime 620a⟩≡ (740b)

```
static char *
parsetime(vlong *rt, char *s)
{
    uvlong ticks;
    ulong l;
    char *e, *p;
    static int p10[] = {100000000, 10000000, 1000000, 100000, 10000, 1000, 100, 10, 1};

    if (s == nil)
        return "missing value";
    ticks=strtoul(s, &e, 10);
    if (*e == '.'){
        p = e+1;
        l = strtoul(p, &e, 10);
        if(e-p > nelem(p10))
            return "too many digits after decimal point";
        if(e-p == 0)
            return "ill-formed number";
        l *= p10[e-p-1];
    }else
        l = 0;
    if (*e == '\0' || strcmp(e, "s") == 0){
        ticks = 1000000000 * ticks + l;
    }else if (strcmp(e, "ms") == 0){
        ticks = 1000000 * ticks + l/1000;
    }else if (strcmp(e, "Âµs") == 0 || strcmp(e, "us") == 0){
        ticks = 1000 * ticks + l/1000000;
    }else if (strcmp(e, "ns") != 0)
        return "unrecognized unit";
    *rt = ticks;
    return nil;
}
```

Uses nelem 853i.

The procctlreq() function is the central dispatch for all /proc/\emphpid/ctl commands. It parses the text command using parsecmd()/lookupcmd() against the proccmd table, then switches on the command index. The function assumes the caller holds p->debug, which serializes all control operations on a given process.

⟨*function* procctlreq 620b⟩≡ (740b)

```
// assumes p->debug is held
void
procctlreq(Proc *p, char *va, int n)
{
    Segment *s;
    int npc, pri;
    Cmdbuf *cb;
    Cmdtab *ct;
    vlong time;
    char *e;
```

```
    void (*pt)(Proc*, int, vlong);

    if(p->kp)   /* no ctl requests to kprocs */
        error(Eperm);

    cb = parsecmd(va, n);
    if(waserror()){
        free(cb);
        nexterror();
    }

    ct = lookupcmd(cb, proccmd, nelem(proccmd));

    switch(ct->index){

    ⟨procctlreq() CMpri case 220e⟩
    ⟨procctlreq() CMfixedpri case 220f⟩
    ⟨procctlreq() CMwired case 237f⟩
    ⟨procctlreq() CMnoswap case 213e⟩

    ⟨procctlreq() CMprivate case 601i⟩

    ⟨procctlreq() CMhang case 487c⟩
    ⟨procctlreq() CMnohang case 487d⟩

    ⟨procctlreq() CMstop case 485c⟩
    ⟨procctlreq() CMstart case 485d⟩
    ⟨procctlreq() CMwaitstop case 487g⟩

    ⟨procctlreq() CMstartsyscall case 491b⟩
    ⟨procctlreq() CMstartstop case 502c⟩

    ⟨procctlreq() CMtrace case 498c⟩
    ⟨procctlreq() CMevent case 500a⟩

    ⟨procctlreq() CMprofile case 506d⟩


    ⟨procctlreq() CMkill case 159c⟩

    case CMclose:
        procctlclosefiles(p, false, atoi(cb->f[1]));
        break;
    case CMclosefiles:
        procctlclosefiles(p, true, 0);
        break;


    ⟨procctlreq() optional real-time commands 565c⟩

    }

    poperror();
    free(cb);
}
```

Uses CMclose-879 602b, CMclosefiles-880 602b, false, free() 104a, lookupcmd() 595b, nelem 853i, nexterror, parsecmd() 594c, poperror 666d, proccmd-908 603b, procctlclosefiles() 619b, true, and waserror 666d.

23.4 /fd/

The /fd/ device (also known as devdup) exposes the current process's file descriptors as files. Opening /fd/0 gives you a new reference to file descriptor 0. This is used during boot to set up standard I/O: the boot script opens the console device and then uses /fd/0, /fd/1, /fd/2 to wire up stdin, stdout, and stderr. The \hlinksysdup system call (see Chapter 12) provides programmatic access to the same functionality.

23.5 /sys/

The /sys/ device (device letter #k) provides system-wide configuration files that are independent of any particular process: the OS version string, the system name, the host owner, the list of registered device drivers, and a reboot/halt interface. Unlike /dev/ files which may return per-process information (e.g., /dev/pid), /sys/ files represent global system state. The distinction is: if the value depends on up, it goes in /dev; if it is a machine-wide setting, it goes in /sys.

⟨*devsys.c enum Qxxx* 622a⟩≡ (732e)
```
enum{
    Qdir,

    ⟨devsys.c enum Qxxx cases 624a⟩
};
```

⟨*global* sysdir 622b⟩≡ (732e)
```
static Dirtab sysdir[]={
    ".",    {Qdir, 0, QTDIR},   0,      DMDIR|0555,

    ⟨sysdir fields 624b⟩
};
```
Uses DMDIR, QTDIR 70b, and Qdir-836 622a.

⟨*method sysinit* 622c⟩≡ (732e)
```
static void
sysinit(void)
{
}
```

⟨*method sysattach* 622d⟩≡ (732e)
```
static Chan*
sysattach(char *spec)
{
    return devattach('k', spec);
}
```
Uses devattach() 297b.

⟨*method sysread* 622e⟩≡ (732e)
```
static long
sysread(Chan *c, void *buf, long n, vlong off)
{
    vlong offset = off;
    char tmp[256];
    ⟨sysread() locals 623a⟩

    if(n <= 0)
        return n;

    switch((ulong)c->qid.path){
    case Qdir:
```

```
        return devdirread(c, buf, n, sysdir, nelem(sysdir), devgen);

    ⟨sysread() cases 624c⟩

    default:
        print("sysread %#llux\n", c->qid.path);
        error(Egreg);
    }
    return -1;      /* never reached */
}
```

Uses Egreg 664d, Qdir-836 622a, devdirread() 432c, devgen() 428, nelem 853i, print, and sysdir-845 622b.

⟨sysread() *locals* 623a⟩≡ (622e)

```
    char *b, *bp;
    int i, k;
    Cpu *mp;
    int id;
```

⟨*method syswrite* 623b⟩≡ (732e)

```
static long
syswrite(Chan *c, void *va, long n, vlong off)
{
    ulong offset;
    char *a;
    ⟨syswrite() locals 623c⟩

    offset = off;
    a = va;

    switch((ulong)c->qid.path){

    ⟨syswrite() cases 625c⟩

    default:
        print("syswrite: %#llux\n", c->qid.path);
        error(Egreg);
    }
    return n;
}
```

Uses Egreg 664d and print.

⟨syswrite() *locals* 623c⟩≡ (623b) 627a ▷

```
    char buf[256];
    Cpu *mp;
    int id;
```

⟨*global* sysdevtab 623d⟩≡ (732e)

```
Dev sysdevtab = {
    .dc       =    'k',
    .name     =    "sys",

    .reset    =    devreset,
    .init     =    sysinit,
    .shutdown =    devshutdown,
    .attach   =    sysattach,
    .walk     =    syswalk,
    .stat     =    sysstat,
    .open     =    sysopen,
    .create   =    devcreate,
    .close    =    sysclose,
```

```
    .read     =    sysread,
    .bread    =    devbread,
    .write    =    syswrite,
    .bwrite   =    devbwrite,
    .remove   =    devremove,
    .wstat    =    devwstat,
};
```

Uses devbread() 434a, devbwrite() 434b, devcreate() 810, devremove() 810, devreset() 810, devshutdown() 810, and devwstat() 810.

23.5.1 /sys/osversion

⟨*devsys.c enum Qxxx cases* 624a⟩≡ (622a) 624e▷

```
Qosversion,
```

⟨sysdir *fields* 624b⟩≡ (622b) 624f▷

```
"osversion",    {Qosversion},    0,      0444,
```

Uses Qosversion-837 624a.

⟨sysread() *cases* 624c⟩≡ (622e) 624g▷

```
case Qosversion:
    snprint(tmp, sizeof tmp, "pad's version");
    n = readstr((ulong)offset, buf, n, tmp);
    return n;
```

Uses Qosversion-837 624a and readstr() 670d.

23.5.2 /sys/config

⟨*devsys.c decls and globals* 624d⟩≡ (732e) 626f▷

```
extern byte configfile[]; // in $CONF.c
```

⟨*devsys.c enum Qxxx cases* 624e⟩+≡ (622a) ◁624a 624h▷

```
Qconfig,
```

⟨sysdir *fields* 624f⟩+≡ (622b) ◁624b 625a▷

```
"config",   {Qconfig},   0,      0444,
```

Uses Qconfig-838 624e.

⟨sysread() *cases* 624g⟩+≡ (622e) ◁624c 625b▷

```
case Qconfig:
    return readstr((ulong)offset, buf, n, (char*) configfile);
```

Uses Qconfig-838 624e and readstr() 670d.

23.5.3 /ys/hostowner, /sys/hostdomain

The host owner (eve) is Plan 9's equivalent of root—the user who "owns" the machine. Reading /sys/hostowner returns this name, and only eve can change it. The host domain identifies the authentication domain. These are writable because the boot process sets them up after the kernel starts, and an administrator may need to change them.

⟨*devsys.c enum Qxxx cases* 624h⟩+≡ (622a) ◁624e 625e▷

```
Qhostowner,
Qhostdomain,
```

⟨sysdir *fields* 625a⟩+≡ (622b) ◁624f 625f▷
```
  "hostowner",    {Qhostowner},   0,      0664,
  "hostdomain",   {Qhostdomain},  DOMLEN,     0664,
```
Uses Qhostdomain-840 624h and Qhostowner-839 624h.

⟨sysread() *cases* 625b⟩+≡ (622e) ◁624g 625g▷
```
  case Qhostowner:
      return readstr((ulong)offset, buf, n, eve);

  case Qhostdomain:
      return readstr((ulong)offset, buf, n, hostdomain);
```
Uses Qhostdomain-840 624h, Qhostowner-839 624h, eve 591a, hostdomain 591d, and readstr() 670d.

⟨syswrite() *cases* 625c⟩≡ (623b) 625h▷
```
  case Qhostowner:
      return hostownerwrite(a, n);

  case Qhostdomain:
      return hostdomainwrite(a, n);
```
Uses Qhostdomain-840 624h, Qhostowner-839 624h, hostdomainwrite() 592b, and hostownerwrite() 132d.

23.5.4 /sys/sysname

⟨*global* sysname 625d⟩≡ (732e)
```
  char    *sysname;
```

⟨*devsys.c enum Qxxx cases* 625e⟩+≡ (622a) ◁624h 626a▷
```
  Qsysname,
```

⟨sysdir *fields* 625f⟩+≡ (622b) ◁625a 626b▷
```
  "sysname",  {Qsysname}, 0,      0664,
```
Uses Qsysname-841 625e.

⟨sysread() *cases* 625g⟩+≡ (622e) ◁625b 626c▷
```
  case Qsysname:
      if(sysname == nil)
          return 0;
      return readstr((ulong)offset, buf, n, sysname);
```
Uses Qsysname-841 625e, readstr() 670d, and sysname 625d.

⟨syswrite() *cases* 625h⟩+≡ (623b) ◁625c 627b▷
```
  case Qsysname:
      if(offset != 0)
          error(Ebadarg);
      if(n <= 0 || n >= sizeof buf)
          error(Ebadarg);
      strncpy(buf, a, n);
      buf[n] = 0;
      if(buf[n-1] == '\n')
          buf[n-1] = 0;
      kstrdup(&sysname, buf);
      break;
```
Uses Ebadarg 664d, Qsysname-841 625e, kstrdup() 669b, and sysname 625d.

23.5.5 /sys/drivers

Reading `/sys/drivers` lists all registered device drivers with their device letter and name—e.g., "`#c cons`", "`#e env`", "`#p proc`". This is useful for discovering what devices are compiled into the kernel without knowing the configuration at build time.

⟨*devsys.c enum Qxxx cases* 626a⟩+≡ (622a) ◁625e 626d▷
```
Qdrivers,
```

⟨`sysdir` *fields* 626b⟩+≡ (622b) ◁625f 626e▷
```
"drivers",  {Qdrivers}, 0,      0444,
```
Uses `Qdrivers-842` 626a.

⟨`sysread()` *cases* 626c⟩+≡ (622e) ◁625g 659a▷
```
case Qdrivers:
    b = malloc(READSTR);
    if(b == nil)
        error(Enomem);
    k = 0;
    for(i = 0; devtab[i] != nil; i++)
        k += snprint(b+k, READSTR-k, "#%C %s\n",
            devtab[i]->dc, devtab[i]->name);
    if(waserror()){
        free(b);
        nexterror();
    }
    n = readstr((ulong)offset, buf, n, b);
    free(b);
    poperror();
    return n;
```
Uses `Qdrivers-842` 626a, `READSTR` 867a, `free()` 104a, `malloc()` 103c, `nexterror`, `poperror` 666d, `readstr()` 670d, and `waserror` 666d.

23.5.6 /sys/reboot

Writing "halt", "reboot", or "panic" to `/sys/reboot` triggers a system shutdown. The reboot path is particularly interesting on multi-core systems: the kernel must migrate to CPU 0 (the boot processor), shut down all other CPUs, disable interrupts and caches, then copy a small trampoline (`rebootcode.s`) to a fixed physical address. This trampoline turns off the MMU and copies the new kernel image to its load address before jumping to it.

⟨*devsys.c enum Qxxx cases* 626d⟩+≡ (622a) ◁626a 658d▷
```
Qreboot,
```

⟨`sysdir` *fields* 626e⟩+≡ (622b) ◁626b 658e▷
```
"reboot",   {Qreboot},  0,      0660,
```
Uses `Qreboot-843` 626d.

⟨*devsys.c decls and globals* 626f⟩+≡ (732e) ◁624d
```
enum
{
    CMhalt,
    CMreboot,
    CMpanic,
};

Cmdtab rebootmsg[] =
{
    CMhalt,     "halt",     1,
```

```
    CMreboot,   "reboot",   0,
    CMpanic,    "panic",    0,
};
```

Uses CMhalt-846 626f, CMpanic-848 626f, and CMreboot-847 626f.

⟨syswrite() *locals* 627a⟩+≡ (623b) ◁623c

```
Cmdbuf *cb;
Cmdtab *ct;
```

⟨syswrite() *cases* 627b⟩+≡ (623b) ◁625h 659b▷

```
case Qreboot:
    if(!iseve())
        error(Eperm);
    cb = parsecmd(a, n);

    if(waserror()) {
        free(cb);
        nexterror();
    }
    ct = lookupcmd(cb, rebootmsg, nelem(rebootmsg));
    switch(ct->index) {
    case CMhalt:
        arch_reboot(nil, 0, 0);
        break;
    case CMreboot:
        rebootcmd(cb->nf-1, cb->f+1);
        break;
    case CMpanic:
        *(ulong*)0=0;
        panic("/dev/reboot");
    }
    poperror();
    free(cb);
    break;
```

Uses CMhalt-846 626f, CMpanic-848 626f, CMreboot-847 626f, Qreboot-843 626d, free() 104a, iseve() 591b, lookupcmd() 595b, nelem 853i, nexterror, parsecmd() 594c, poperror 666d, rebootmsg 626f, and waserror 666d.

⟨kbdputsc() *reboot if ctl-alt-del* 627c⟩≡ (378d)

```
if(kbscan->ctl)
    if(kbscan->alt && c == Del) // Ctl-Alt-Del
        arch_exit(0);
```

Uses Del-983 376a and arch_exit.

⟨ConsKbd *other fields* 627d⟩+≡ (382a) ◁396a

```
bool ctlpoff; // ^P will not reboot if true
```

⟨conswrite() *Qconsctl other ifs* 627e⟩≡ (397a)

```
else if(strncmp(a, "ctlpon", 6) == 0){
    kbd.ctlpoff = false;
} else if(strncmp(a, "ctlpoff", 7) == 0){
    kbd.ctlpoff = true;
}
```

Uses false, kbd-1778 382b, and true.

⟨echo() *special key C-p* 627f⟩≡ (642c)

```
case 0x10:  /* ^P */
    if(cpuserver && !kbd.ctlpoff){
        active.exiting = true;
        return;
    }
    break;
```

Uses active 65e, kbd-1778 382b, and true.

⟨Active *other fields* 628a⟩≡ (65d) 629a▷
```
bool ispanic;    /* shutdown in response to a panic */
```

The shutdown sequence calls each device driver's `shutdown()` method in reverse order of initialization. This gives devices a chance to flush buffers, complete pending I/O, and put hardware into a safe state before the system halts or reboots.

⟨*function* chandevshutdown 628b⟩≡ (808c)
```
void
chandevshutdown(void)
{
    int i;

    /* shutdown in reverse order */
    for(i=0; devtab[i] != nil; i++)
        ;
    for(i--; i >= 0; i--)
        devtab[i]->shutdown();
}
```

⟨Dev *methods* 628c⟩+≡ (71b) ◁433b
```
void  (*shutdown)(void);
```

arch_exit()(ARM)

⟨*function* main_arch_exit*(arm)* 628d⟩≡ (991c)
```
/*
 *  exit kernel either on a panic or user request
 */
void
main_arch_exit(int code)
{
    void (*f)(ulong, ulong, ulong);

    shutdown(code);
    splfhi();
    if(cpu->cpuno == 0)
        archreboot();
    else{
        f = (void*)REBOOTADDR;
        intrcpushutdown();
        cacheuwbinv();
        l2cacheuwbinv();
        (*f)(0, 0, 0);
        for(;;){}
    }
}
```
Uses archreboot() 1001b and intrcpushutdown() 631a.

⟨*constant* REBOOTADDR*(arm)* 628e⟩≡ (1005)
```
/* address at which to copy and execute rebootcode */
#define REBOOTADDR  (KZERO+0x1800)
```

⟨*function* shutdown*((init/arm/main.c)(arm))* 628f⟩≡ (991c)
```
static void
shutdown(int ispanic)
{
    int ms, once;
```

```
    lock(&active);
    if(ispanic)
        active.ispanic = ispanic;
    else if(cpu->cpuno == 0 && (active.cpus & (1 << cpu->cpuno)) == 0)
        active.ispanic = 0;
    once = active.cpus & (1 << cpu->cpuno);
    active.cpus &= ~(1 << cpu->cpuno);
    active.exiting = 1;
    unlock(&active);

    if(once)
        iprint("cpu%d: exiting\n", cpu->cpuno);
    arch_spllo();
    for(ms = 5*1000; ms > 0; ms -= TK2MS(2)){
        arch_delay(TK2MS(2));
        if(active.cpus == 0 && consactive() == 0)
            break;
    }
    arch_delay(100*cpu->cpuno);
}
```

Uses TK2MS 244d, active 65e, arch_delay, consactive() 653b, iprint, lock() 76d, and unlock() 78d.

⟨Active *other fields* 629a⟩+≡ (65d) ◁628a

```
bool rebooting;    /* just idle cpus > 0 */
```

The reboot function must carefully dismantle the running kernel: first it migrates to CPU 0, copies the reboot trampoline to a fixed physical address, shuts down all other CPUs, devices, and interrupts, then calls the trampoline which turns off the MMU and copies the new kernel image to its final location in physical memory before jumping to the entry point. The trampoline itself must be position-independent and work without virtual memory, since it disables the MMU before doing the copy.

⟨*function* arch_reboot*(arm)* 629b⟩≡ (991c)

```
/*
 * the new kernel is already loaded at address ‘code’
 * of size ‘size’ and entry point ‘entry’.
 */
void
arch_reboot(void *entry, void *code, ulong size)
{
    void (*f)(ulong, ulong, ulong);

    //writeconf();

    /*
     * the boot processor is cpu0.  execute this function on it
     * so that the new kernel has the same cpu0.
     */
    if (cpu->cpuno != 0) {
        procwired(up, 0);
        sched();
    }
    if (cpu->cpuno != 0)
        print("on cpu%d (not 0)!\n", cpu->cpuno);

    /* setup reboot trampoline function */
    f = (void*)REBOOTADDR;
    memmove(f, rebootcode, sizeof(rebootcode));
    cachedwbse(f, sizeof(rebootcode));

    shutdown(0);
```

```
    /*
     * should be the only processor running now
     */

    arch_delay(5000);
    print("active.machs = %x\n", active.cpus);
    print("reboot entry %#lux code %#lux size %ld\n",
        PADDR(entry), PADDR(code), size);
    arch_delay(100);

    /* turn off buffered serial console */
    serialoq = nil;
    kprintoq = nil;
    screenputs = nil;

    /* shutdown devices */
    if(!waserror()){
        chandevshutdown();
        poperror();
    }

    /* stop the clock (and watchdog if any) */
    clockshutdown();

    splfhi();
    intrshutdown();

    /* off we go - never to return */
    cacheuwbinv();
    l2cacheuwbinv();
    (*f)(PADDR(entry), PADDR(code), size);

    iprint("loaded kernel returned!\n");
    arch_delay(1000);
    archreboot();
}
```

Uses active 65e, arch_delay, archreboot() 1001b, chandevshutdown() 628b, clockshutdown() 630b, intrshutdown() 630a, iprint, kprintoq 651e, poperror 666d, print, procwired() 237g, sched, screenputs 384c, serialoq 653a, up, and waserror 666d.

⟨*function* intrshutdown*(arm)* 630a⟩≡ (1002c)

```
/* called from cpu0 after other cpus are shutdown */
void
intrshutdown(void)
{
    intrsoff();
    intrcpushutdown();
}
```

Uses intrcpushutdown() 631a and intrsoff() 279a.

⟨*function* clockshutdown*(arm)* 630b⟩≡ (1011b)

```
void
clockshutdown(void)
{
    Armtimer *tm;

    tm = (Armtimer*)ARMTIMER;
    tm->ctl = 0;
    //wdogoff();
```

```
}
```
Uses ARMTIMER-2648 43c.

⟨*function* intrcpushutdown*(arm)* 631a⟩≡ (1002c)
```
void
intrcpushutdown(void)
{
    u32int *enable;

    if(soc.armlocal == 0)
        return;
    enable = (u32int*)(LOCALREGS + Localtimerint) + cpu->cpuno;
    *enable = 0;
    if(cpu->cpuno){
        enable = (u32int*)(LOCALREGS + Localmboxint) + cpu->cpuno;
        *enable = 1;
    }
}
```
Uses LOCALREGS-2404 553g, Localmboxint-2409 553f, and Localtimerint-2408 553d.

rebootcode.s (ARM)

⟨*constant* PTEDRAM*(arm)* 631b⟩≡ (1000a)
```
#define PTEDRAM     (Dom0|L1AP(Krw)|Section)
```

⟨*instruction WFI(arm)* 631c⟩≡ (1000a)
```
#define WFI WORD    $0xe320f003 /* wait for interrupt */
```

⟨*instruction WFE(arm)* 631d⟩≡ (1000a)
```
#define WFE WORD    $0xe320f002 /* wait for event */
```

⟨*function main(rebootcode.s)(arm)* 631e⟩≡ (1000a)
```
/*
 * Turn off MMU, then copy the new kernel to its correct location
 * in physical memory.  Then jump to the start of the kernel.
 */

/* main(PADDR(entry), PADDR(code), size); */
TEXT    main(SB), 1, $-4
    MOVW    $setR12(SB), R12

    /* copy in arguments before stack gets unmapped */
    MOVW    R0, R8          /* entry point */
    MOVW    p2+4(FP), R9        /* source */
    MOVW    n+8(FP), R6     /* byte count */

    /* SVC mode, interrupts disabled */
    MOVW    $(PsrDirq|PsrDfiq|PsrMsvc), R1
    MOVW    R1, CPSR

    /* prepare to turn off mmu  */
    BL  cachesoff(SB)

    /* turn off mmu */
    MRC CpSC, 0, R1, C(CpCONTROL), C(0), CpMainctl
    BIC $CpCmmu, R1
    MCR CpSC, 0, R1, C(CpCONTROL), C(0), CpMainctl

    /* continue with reboot only on cpu0 */
```

```
    CPUID(R2)
    BEQ bootcpu

    /* other cpus wait for inter processor interrupt from cpu0 */
    /* turn icache back on */
    MRC CpSC, 0, R1, C(CpCONTROL), C(0), CpMainctl
    ORR $(CpCicache), R1
    MCR CpSC, 0, R1, C(CpCONTROL), C(0), CpMainctl
    BARRIERS
dowfi:
    WFI
    MOVW    $0x40000060, R1
    ADD     R2<<2, R1
    MOVW    0(R1), R0
    AND     $0x10, R0
    BEQ     dowfi
    MOVW    $0x8000, R1
    BL      (R1)
    B       dowfi

bootcpu:
    /* set up a tiny stack for local vars and memmove args */
    MOVW    R8, SP          /* stack top just before kernel dest */
    SUB $20, SP         /* allocate stack frame */

    /* copy the kernel to final destination */
    MOVW    R8, 16(SP)      /* save dest (entry point) */
    MOVW    R8, R0          /* first arg is dest */
    MOVW    R9, 8(SP)       /* push src */
    MOVW    R6, 12(SP)      /* push size */
    BL  memmove(SB)
    MOVW    16(SP), R8      /* restore entry point */

    /* jump to kernel physical entry point */
    ORR R8,R8
    B   (R8)
    B   0(PC)

/*
 * turn the caches off, double map PHYSDRAM & KZERO, invalidate TLBs, revert
 * to tiny addresses.  upon return, it will be safe to turn off the mmu.
 * clobbers R0-R2, and returns with SP invalid.
 */
TEXT cachesoff(SB), 1, $-4
    MOVM.DB.W [R14,R1-R10], (R13)       /* save regs on stack */

    /* turn caches off, invalidate icache */
    MRC CpSC, 0, R1, C(CpCONTROL), C(0), CpMainctl
    BIC $(CpCdcache|CpCicache|CpCpredict), R1
    MCR CpSC, 0, R1, C(CpCONTROL), C(0), CpMainctl
    MOVW    $0, R0
    MCR CpSC, 0, R0, C(CpCACHE), C(CpCACHEinvi), CpCACHEall

    /* invalidate stale TLBs before changing them */
    BARRIERS
    MOVW    $0, R0
    MCR CpSC, 0, R0, C(CpTLB), C(CpTLBinvu), CpTLBinv
    BARRIERS

    /* redo double map of first MiB PHYSDRAM = KZERO */
```

```
MOVW    12(R(CPU)), R2      /* m->mmul1 (virtual addr) */
MOVW    $PTEDRAM, R1            /* PTE bits */
MOVW    R1, (R2)

/* invalidate stale TLBs again */
BARRIERS
MOVW    $0, R0
MCR CpSC, 0, R0, C(CpTLB), C(CpTLBinvu), CpTLBinv
BARRIERS

/* relocate SB and return address to PHYSDRAM addressing */
MOVW    $KSEGM, R1      /* clear segment bits */
BIC R1, R12         /* adjust SB */
MOVM.IA.W (R13), [R14,R1-R10]       /* restore regs from stack */

MOVW    $KSEGM, R1      /* clear segment bits */
BIC R1, R14         /* adjust return address */

RET
```

23.6 /dev/

The /dev/ device (device letter #c, for "cons") is a grab-bag of per-process and miscellaneous files. Some files return per-process values (/dev/pid, /dev/ppid, /dev/user, /dev/cputime), while others provide system services (/dev/random, /dev/null, /dev/zero, /dev/time). The console (/dev/cons) was covered in the Devices chapter; here we see the remaining files.

⟨*devcons.c enum Qxxx cases* 633a⟩≡ (388a) 650f▷

```
Qbintime,
Qcputime,
Qnull,
Qpgrpid,
Qpid,
Qppid,
Qrandom,
Qswap,
Qtime,
Quser,
Qzero,
```

⟨consdir *fields* 633b⟩≡ (388b) 650g▷

```
"bintime",  {Qbintime}, 24,     0664,
"cputime",  {Qcputime}, 6*NUMSIZE,  0444,
"null",     {Qnull},    0,      0666,
"pgrpid",   {Qpgrpid},  NUMSIZE,    0444,
"pid",      {Qpid},     NUMSIZE,    0444,
"ppid",     {Qppid},    NUMSIZE,    0444,
"random",   {Qrandom},  0,      0444,
"swap",     {Qswap},    0,      0664,
"time",     {Qtime},    NUMSIZE+3*VLNUMSIZE,    0664,
"user",     {Quser},    0,      0666,
"zero",     {Qzero},    0,      0444,
```

Uses NUMSIZE 867a, Qbintime-1783 633a, Qcputime-1784 633a, Qnull-1785 633a, Qpgrpid-1786 633a, Qpid-1787 633a, Qppid-1788 633a, Qrandom-1789 633a, Qswap-1790 633a, Qtime-1791 633a, Quser-1792 633a, Qzero-1793 633a, and VLNUMSIZE-1796 260c.

⟨consinit() *initializing things* 634a⟩+≡ (393b) ◁383b
```
todinit();
randominit();
```
Uses randominit() 636a and todinit() 260d.

23.6.1 /dev/random

The random number generator uses two layers. The outer layer (nrand()) is a simple linear congruential generator seeded from the inner layer. The inner layer (/dev/random) collects entropy from timing jitter: a dedicated kernel process (kgenrandom) busy-loops incrementing a counter, and a periodic clock callback (randomclock) samples that counter to extract unpredictable bits. The idea is that the exact count depends on cache behavior, interrupts, and scheduling—all sources of genuine randomness. The entropy is accumulated in a circular buffer (rb) using a producer/consumer pattern.

⟨*global* randn 634b⟩≡ (696b)
```
static  ulong   randn;
```

⟨*function* seedrand 634c⟩≡ (696b)
```
static void
seedrand(void)
{
    if(!waserror()){
        randomread((void*)&randn, sizeof(randn));
        poperror();
    }
}
```
Uses poperror 666d, randn-1798 634b, randomread() 636b, and waserror 666d.

⟨*function* nrand 634d⟩≡ (696b)
```
int
nrand(int n)
{
    if(randn == 0)
        seedrand();
    randn = randn*1103515245 + 12345 + CPUS(0)->ticks;
    return (randn>>16) % n;
}
```
Uses CPUS 702, randn-1798 634b, and seedrand() 634c.

⟨*struct* Rb 634e⟩≡ (868b)
```
struct Rb
{
    QLock;
    Rendez  producer;
    Rendez  consumer;
    ulong   randomcount;
    byte    buf[1024];
    byte    *ep;
    byte    *rp;
    byte    *wp;
    byte    next;
    byte    wakeme;
    ushort  bits;
    ulong   randn;
};
```

⟨*global* rb 634f⟩≡ (868b)
```
struct Rb rb;
```
Uses Rb 634e.

⟨*struct* `rbnotfull` 635a⟩≡ (868b)
```
static int
rbnotfull(void*)
{
    int i;

    i = rb.rp - rb.wp;
    return i != 1 && i != (1 - sizeof(rb.buf));
}
```
Uses `rb` 634f.

⟨*struct* `rbnotempty` 635b⟩≡ (868b)
```
static int
rbnotempty(void*)
{
    return rb.wp != rb.rp;
}
```
Uses `rb` 634f.

⟨*function* `genrandom` 635c⟩≡ (868b)
```
static void
genrandom(void*)
{
    up->basepri = PriNormal;
    up->priority = up->basepri;

    for(;;){
        for(;;)
            if(++rb.randomcount > 100000)
                break;
        if(anyhigher())
            sched();
        if(!rbnotfull(0))
            sleep(&rb.producer, rbnotfull, 0);
    }
}
```
Uses `PriNormal` 219b, `anyhigher()` 236d, `rb` 634f, `rbnotfull()` 635a, `sched`, `sleep`, and `up`.

⟨*clock callback randomclock* 635d⟩≡ (868b)
```
/*
 *  produce random bits in a circular buffer
 */
static void
randomclock(void)
{
    if(rb.randomcount == 0 || !rbnotfull(0))
        return;

    rb.bits = (rb.bits<<2) ^ rb.randomcount;
    rb.randomcount = 0;

    rb.next++;
    if(rb.next != 8/2)
        return;
    rb.next = 0;

    *rb.wp ^= rb.bits;
    if(rb.wp+1 == rb.ep)
        rb.wp = rb.buf;
```

```
    else
        rb.wp = rb.wp+1;

    if(rb.wakeme)
        wakeup(&rb.consumer);
}
```

Uses rb 634f, rbnotfull() 635a, and wakeup.

⟨*function* randominit 636a⟩≡ (868b)

```
void
randominit(void)
{
    /* Frequency close but not equal to HZ */
    addclock0link(randomclock, 13);
    rb.ep = rb.buf + sizeof(rb.buf);
    rb.rp = rb.wp = rb.buf;
    kproc("kgenrandom", genrandom, 0);
}
```

Uses addclock0link() 253f, genrandom() 635c, kproc() 157c, randomclock() 635d, and rb 634f.

⟨*function* randomread 636b⟩≡ (868b)

```
/*
 *  consume random bytes from a circular buffer
 */
ulong
randomread(void *xp, ulong n)
{
    byte *e, *p;
    ulong x;

    p = xp;

    if(waserror()){
        qunlock(&rb);
        nexterror();
    }

    qlock(&rb);
    for(e = p + n; p < e; ){
        if(rb.wp == rb.rp){
            rb.wakeme = 1;
            wakeup(&rb.producer);
            sleep(&rb.consumer, rbnotempty, 0);
            rb.wakeme = 0;
            continue;
        }

        /*
         *  beating clocks will be predictable if
         *  they are synchronized.  Use a cheap pseudo-
         *  random number generator to obscure any cycles.
         */
        x = rb.randn*1103515245 ^ *rb.rp;
        *p++ = rb.randn = x;

        if(rb.rp+1 == rb.ep)
            rb.rp = rb.buf;
        else
            rb.rp = rb.rp+1;
    }
```

```
        qunlock(&rb);
        poperror();

        wakeup(&rb.producer);

        return n;
    }
```

Uses nexterror, poperror 666d, qlock() 81f, qunlock() 82c, rb 634f, rbnotempty() 635b, sleep, wakeup, and waserror 666d.

⟨consread() *cases* 637⟩+≡ (394c) ◁504b 651a▷

```
    case Qrandom:
        return randomread(buf, n);
```

Uses Qrandom-1789 633a and randomread() 636b.

Chapter 24

Conclusion

You now know how the Plan 9 kernel works, to the smallest details: from the first instruction at `_start` to the scheduler loop, from page faults to pipes, from system calls to the 9P protocol—and more generally how many operating system kernels work.

Despite being a complete operating system kernel—with virtual memory, preemptive scheduling, multiprocessor support, a file-oriented device framework, and a network-transparent namespace—the entire kernel fits in roughly 30 000 lines of C and a few hundred lines of assembly. This is remarkably small. The key to this compactness is the "everything is a file server" philosophy: instead of adding features to the kernel, Plan 9 pushes them to user-space file servers communicating over 9P. The kernel provides just enough mechanism (processes, memory, files, namespaces) and leaves policy to user space.

Along the way, you have seen many recurring patterns: reference counting for shared resources (`Ref`), sleep/wakeup for blocking (`Rendez`), two-level tables for sparse mappings (`Segment`/`Pagetable`, L1/L2 page tables), dispatch tables for polymorphism (`systab`, `devtab`), and the `waserror`/`poperror` mechanism for exception handling in C. You have also seen how a well-chosen set of abstractions —`Chan`, `Dev`, `Segment`, `Proc`, `Page`—can compose to build a complete system where each piece is understandable in isolation.

24.1 Patterns and techniques

These techniques apply far beyond operating systems:

- *Uniform interface*: collapsing many operations into `open`/`read`/`write`/`close` is the same strategy as REST (everything is a resource with GET/PUT/POST/DELETE) and SQL (everything is a table with SELECT/INSERT/UPDATE/DELETE). A single interface lets generic tools work without knowing what is behind it.

- *Dispatch tables for polymorphism*: `devtab` maps device characters to method tables, `systab` maps call numbers to handlers. This is the C equivalent of virtual method dispatch, and the same pattern appears in protocol handlers (HTTP method dispatch), plugin architectures (editor command tables), and interpreters (opcode dispatch).

- *Exception handling in C*: the `waserror`/`poperror` mechanism, built on `setjmp`/`longjmp`, gives C code a form of try/catch. The same technique is used in Lua's `lua_pcall()`, SQLite's error recovery, and the JPEG library's `setjmp`-based error handler. It solves the problem of unwinding deeply nested calls in languages that lack native exceptions.

- *Reference counting*: `Ref` manages shared ownership without garbage collection. Python uses it as its primary memory strategy, Rust has `Rc`/`Arc`, and file systems use hard link counts. It works wherever multiple owners share a resource and you need deterministic cleanup.

- *Condition variables*: `Rendez` is the same abstraction as POSIX condition variables and Java's `wait()`/`notify()`. It is the fundamental building block for producer/consumer synchronization in any concurrent system.
- *Sparse multi-level tables*: two-level indirection with on-demand allocation trades one pointer dereference for enormous memory savings. The same structure appears in Unicode character databases and IP routing tries.
- *Per-process namespaces*: giving each process its own view of the file hierarchy is the same idea behind Docker containers, chroot jails, and Python virtual environments—scoping a shared resource per context.

24.2 Connections to other books

The kernel is the foundation on which every other program in Principia Softwarica runs. The books are deeply interconnected:

- The SHELL book [Pad18] explains `rc`, the first program most users interact with. Every command you type in `rc` exercises `rfork()`, `exec()`, `wait()`, pipes, and file descriptors—all of which you now understand from the kernel side.
- The COMPILER book [Pad16b] and ASSEMBLER book [Pad15a] describe the tools that produce the kernel binary itself, including the `5c`/`5a` compilers whose calling conventions and `extern register` extension you have seen throughout the ARM-specific code.
- The LINKER book [Pad15c] explains how `5l` produces the a.out binary that `sysexec()` loads, and how the static base register `R12` is set up.
- The BUILDER book [Pad16a] explains `mk`, which orchestrates the kernel build and whose `mkfile` rules drive the compilation of all the source files discussed in this book.
- The GRAPHICS book [Pad16d] covers `devdraw` and the framebuffer infrastructure that this book only touched briefly (the screen output and `screenputs` hook).
- The LIBCORE book [Pad16c] explains the C library that sits between user programs and the kernel: the `pwrite()`/`brk()`/`rfork()` wrappers, the `Pool` allocator shared with the kernel's `malloc`, and the `print()` formatting engine.

24.3 What this book does not cover

This book focused on the core kernel mechanisms. Several important subsystems were left for other books or future work:

- *Networking:* The IP stack (`devip`), the Ethernet driver (`devether`), and the network file servers are substantial enough for their own treatment.
- *Graphics:* The draw device (`devdraw`), and the graphics libraries deserve a dedicated GRAPHICS book [Pad16d].
- *On-disk file systems:* The file servers `fossil`, `cwfs`, and `hjfs` implement the actual storage layer that the kernel's `devmnt` connects to over 9P.
- *Authentication:* The `factotum` authentication agent and the security architecture (capabilities, `srvauth`, tickets) are covered only superficially here.

Despite these omissions, the kernel described in this book is complete and self-contained: it can boot on a Raspberry Pi, run user programs, manage memory, handle interrupts, and serve files. Everything else is built on top of the mechanisms you now understand.

24.4 Beyond the Plan 9 kernel

The Plan 9 kernel is intentionally minimal. Modern production kernels like Linux, macOS/XNU, and Windows NT are far larger and more featureful. Some of that complexity is essential, some is accidental—but all of it is worth understanding in contrast to Plan 9's deliberate simplicity:

- *Scheduling*: Plan 9 uses a simple multi-level priority scheduler. Linux's EEVDF (formerly CFS) scheduler uses a virtual-runtime red-black tree with cgroups for hierarchical resource control, providing fair CPU sharing across thousands of tasks with fine-grained isolation between workloads. macOS uses a Mach-derived scheduler with separate "clutch" groups for efficiency and performance cores on Apple Silicon.

- *Virtual memory*: Plan 9 has straightforward demand-paged virtual memory with a simple `Page` free list. Linux adds huge pages (2 MB/1 GB), NUMA-aware allocation, memory cgroups with per-group limits, Kernel Same-page Merging (KSM) for deduplicating identical pages across processes, and memory compaction to reduce fragmentation. These are sophisticated mechanisms that make a real difference at datacenter scale.

- *Security*: Plan 9's security model—per-process namespaces and the `factotum` authentication agent—is conceptually elegant and was ahead of its time. Linux has since added mandatory access controls (SELinux, AppArmor), fine-grained POSIX capabilities, seccomp-BPF syscall filtering, and ASLR. macOS adds code signing and sandboxing. Notably, Linux's namespace mechanism—the foundation of containers—was directly inspired by Plan 9's `rfork(RFNOMNT)`.

- *Containers and virtualization*: Plan 9's `rfork` flags were doing namespace isolation before "containers" existed as a concept. Modern container runtimes (Docker, Kubernetes) build on Linux namespaces combined with cgroups, overlay file systems, and seccomp policies. On the virtualization side, Linux's KVM turns the kernel itself into a hypervisor, enabling efficient virtual machines without a separate hypervisor layer.

- *File systems*: Plan 9's approach of delegating file systems to user-space 9P servers is architecturally clean—crashes in a file server cannot panic the kernel, and new file systems need no kernel modifications. The trade-off is the lack of an in-kernel caching layer comparable to Linux's page cache, dentry cache, and inode cache. Linux also supports journaling (ext4, XFS) and copy-on-write snapshots (ZFS, btrfs) directly in the kernel for performance. Linux's FUSE, which allows user-space file systems, follows a similar idea to the 9P model.

- *Networking*: the Plan 9 IP stack is a clean, readable implementation. Linux adds zero-copy networking, TCP offload, eBPF for programmable in-kernel packet processing, and XDP for line-rate packet filtering—driven by datacenter demands where every microsecond matters and where Plan 9's simplicity would genuinely be insufficient.

- *Device model*: Plan 9's static `Dev` table is simple—adding a device means adding one entry. Linux's dynamic model with sysfs, udev, device tree, and power management (suspend/resume, runtime PM) handles hotplug, hardware discovery, and the vast diversity of modern peripherals.

The pattern is consistent: features that Plan 9 pioneered (per-process namespaces, user-space file servers, a unified file-based interface) have been partially adopted by mainstream systems, often with considerably more complexity. The Plan 9 kernel shows that a complete, working kernel can be built from a small set of well-chosen abstractions. Understanding it gives you the conceptual foundation on which the additional engineering of production kernels is built.

Appendix A

Debugging

This appendix collects the kernel's debugging infrastructure: special keyboard shortcuts that dump internal state, the `kmesg` and `kprint` log devices, the serial console (essential when the screen is not yet working), and the remote debugger interface. These are the tools a kernel developer reaches for when something goes wrong during boot or at runtime.

A.1 `sysnop()`

A.2 Special keys and dumpers

Pressing F11 enables keyboard debug mode, which prints every scan code as it arrives. F12 turns it off. Other special key combinations dump internal state (memory allocator, process table, etc.) to the console—invaluable when the system is hung and you cannot type commands:

⟨*global* `kdebug` 641a⟩≡ (708)
```
bool kdebug;
```

⟨`kbdputsc()` *special keyboard debug keys cases* 641b⟩≡ (378d)
```
case KF|11:
    print("kbd debug on, F12 turns it off\n");
    kdebug = true;
    kbdputc(c);
    break;
case KF|12:
    kdebug = false;
    kbdputc(c);
    break;
```
Uses `KF-959` 376a, `false`, `kbdputc()` 382c, `kdebug` 641a, `print`, and `true`.

⟨`kbdputsc()` *debugging* 641c⟩≡ (378d)
```
if(kdebug)
    print("sc %x (ms %d)\n", k, mouseshifted);
```
Uses `kdebug` 641a, `mouseshifted` 719, and `print`.

⟨`kbdputsc()` *debugging up shift* 641d⟩≡ (378d)
```
if(kdebug)
    print("shiftclr\n");
```
Uses `kdebug` 641a and `print`.

⟨`kbdputsc()` *debugging down shift* 641e⟩≡ (378d)
```
if(kdebug)
    print("shift\n");
```
Uses `kdebug` 641a and `print`.

⟨kbdputc() *debugging* 642a⟩≡ (382c)

```
extern bool kdebug;

if(kdebug)
    print("kbdputc(0x%x)\n", ch);
```

Uses kdebug 641a and print.

⟨echo() *locals* 642b⟩≡ (384a)

```
static int ctrlt;
void* tmp;
int x;
char *e, *p;
```

⟨echo() *special keys handler* 642c⟩≡ (384a)

```
    e = buf+n;
    for(p = buf; p < e; p++){
        switch(*p){

        ⟨echo() special key C-p 627f⟩

        case 0x14:  /* ^T */
            ctrlt++;
            if(ctrlt > 2)
                ctrlt = 2;
            continue;
        }
        if(ctrlt != 2)
            continue;

        /* ^T escapes */
        ctrlt = 0;
        switch(*p){
        case 's':
            arch_dumpstack();
            return;
        case 'S':
            x = arch_splhi();
            arch_dumpstack();
            procdump();
            arch_splx(x);
            return;
        case 'x':
            xsummary();
            tmp = xalloc(1000); // PAD: just to debug
            xalloc(1000);
            xfree(tmp);
            xsummary();
            return;
        case 'X':
            xsummary();
            ixsummary();
            mallocsummary();
            arch_memorysummary();
            pagersummary();
            return;
        case 'm':
            arch_memorysummary();
            return;
        case 'p':
            x = arch_spllo();
```

```
            procdump();
            arch_splx(x);
            return;
        case 'q':
            scheddump();
            return;
        case 'k':
            killbig("^t ^t k");
            return;
        case 'r':
            arch_exit(0);
            return;
        ⟨echo() C-t C-t special keys handler other cases 652g⟩
        }
    }
```

Uses arch_dumpstack, arch_exit, ixsummary() 644d, killbig() 876d, mallocsummary() 643b, pagersummary() 644c, procdump() 645a, scheddump() 644f, xalloc() 95d, xfree() 96a, and xsummary() 643a.

⟨*function* xsummary 643a⟩≡ (859c)

```
void
xsummary(void)
{
    int i;
    Hole *h;

    for(i = 0; i < Nhole && xlists.hole[i].top != 0; i++) {
        print("|i| = %d (0x%luX), addr 0x%luX, top = 0x%luX, size = %ld, link = 0x%luX\n",
              i, &xlists.hole[i],
              xlists.hole[i].addr, xlists.hole[i].top, xlists.hole[i].size,
              xlists.hole[i].next
              );
    }
    print("flists = 0x%luX, sorted_holes = 0x%luX\n", xlists.unused_slots, xlists.sorted_holes);
    i = 0;
    for(h = xlists.unused_slots; h; h = h->next)
        i++;

    print("%d holes free", i);
    i = 0;
    for(h = xlists.sorted_holes; h; h = h->next) {
        if (0) {
            print("addr %#.8lux top %#.8lux size %lud\n",
                h->addr, h->top, h->size);
            arch_delay(10);
        }
        i += h->size;
        if (h == h->next) {
            print("xsummary: infinite loop broken\n");
            break;
        }
    }
    print(" %d bytes free\n", i);
}
```

Uses Nhole 855, arch_delay, print, and xlists-1849 95a.

⟨*function* mallosummary 643b⟩≡ (860b)

```
void
mallocsummary(void)
{
    poolsummary(mainmem);
```

```
    poolsummary(imagmem);
}
```

Uses imagmem 102b, mainmem 102b, and poolsummary() 644a.

⟨*function* poolsummary 644a⟩≡ (860b)

```
void
poolsummary(Pool *p)
{
    print("%s max %lud cur %lud free %lud alloc %lud\n", p->name,
        p->maxsize, p->cursize, p->curfree, p->curalloc);
}
```

Uses print.

⟨*function* arch_memorysummary*(arm)* 644b⟩≡ (991c)

```
// called from devcons.c
void
arch_memorysummary(void) {
}
```

⟨*function* pagersummary 644c⟩≡ (866b)

```
void
pagersummary(void)
{
    print("%lud/%lud memory %lud/%lud swap %d iolist\n",
        palloc.user-palloc.freecount,
        palloc.user, conf.nswap-swapalloc.free, conf.nswap,
        ioptr);
}
```

Uses ioptr-1808 208c, palloc 109b, and print.

⟨*function* ixsummary 644d⟩≡ (813c)

```
void
ixsummary(void)
{
    debugging ^= 1;
    iallocsummary();
    print("pad %lud, concat %lud, pullup %lud, copy %lud\n",
        padblockcnt, concatblockcnt, pullupblockcnt, copyblockcnt);
    print("consume %lud, produce %lud, qcopy %lud\n",
        consumecnt, producecnt, qcopycnt);
}
```

Uses concatblockcnt-771 812d, consumecnt-774 812d, copyblockcnt-773 812d, debugging-778 813a, iallocsummary() 644e, padblockcnt-770 812d, print, producecnt-775 812d, pullupblockcnt-772 812d, and qcopycnt-776 812d.

⟨*function* iallocsummary 644e⟩≡ (811)

```
void
iallocsummary(void)
{
    print("ialloc %lud/%lud\n", ialloc.bytes, conf.ialloc);
}
```

Uses conf 62b, ialloc 437b, and print.

⟨*function* scheddump 644f⟩≡ (877c)

```
void
scheddump(void)
{
    Proc *p;
    Schedq *rq;
```

```
    for(rq = &runq[Nrq-1]; rq >= runq; rq--){
        if(rq->head == 0)
            continue;
        print("rq%ld:", rq-runq);
        for(p = rq->head; p; p = p->rnext)
            print(" %lud(%lud)", p->pid, cpu->ticks - p->readytime);
        print("\n");
        arch_delay(150);
    }
    print("nrdy %d\n", nrdy);
}
```

Uses Nrq 221a, arch_delay, nrdy 222a, print, and runq 221b.

⟨*function* procdump 645a⟩≡ (877c)

```
void
procdump(void)
{
    int i;
    Proc *p;

    if(up)
        print("up %lud\n", up->pid);
    else
        print("no current process\n");
    for(i=0; i<conf.nproc; i++) {
        p = &procalloc.arena[i];
        if(p->state == Dead)
            continue;

        dumpaproc(p);
    }
}
```

Uses Dead 67c, conf 62b, dumpaproc, print, procalloc-2458 128d, and up.

⟨*function* dumpaproc 645b⟩≡ (877c)

```
void
proc_dumpaproc(Proc *p)
{
    ulong bss;
    char *s;

    if(p == nil)
        return;

    bss = 0;
    if(p->seg[BSEG])
        bss = p->seg[BSEG]->top;

    s = p->psstate;
    if(s == nil)
        s = statename[p->state];
    print("%3lud:%10s pc %8lux dbgpc %8lux  %8s (%s) ut %ld st %ld bss %lux qpc %lux nl %lud nd %lud lpc %lux p
        p->pid, p->text, p->pc, arch_dbgpc(p),  s, statename[p->state],
        p->time[TUser], p->time[TSys], bss, p->qpc, p->nlocks.ref, p->delaysched, p->lastlock ? p->lastlock->pc
}
```

Uses BSEG 118c, TSys 503a, TUser 503a, print, and statename 67d.

⟨*function* arch_dbgpc*(arm)* 645c⟩≡ (1008b)

```
/*
```

```
 *  pc output by dumpaproc
 */
uintptr
arch_dbgpc(Proc* p)
{
    Ureg *ureg;

    ureg = p->dbgreg;
    if(ureg == 0)
        return 0;

    return ureg->pc;
}
```

⟨*function* `lockloop` 646a⟩≡ (694a)

```
void
lockloop(Lock *l, ulong pc)
{
    Proc *p;

    p = l->p;
    print("lock %#p loop key %#lux pc %#lux held by pc %#lux proc %lud\n",
        l, l->key, pc, l->pc, p ? p->pid : 0);
    dumpaproc(up);
    if(p != nil)
        dumpaproc(p);
}
```

Uses `dumpaproc`, `print`, and `up`.

⟨*function* `dumpregs` *(arm)* 646b⟩≡ (1002c)

```
void
dumpregs(Ureg* ureg)
{
    int s;

    if (ureg == nil) {
        iprint("trap: no user process\n");
        return;
    }
    s = arch_splhi();
    iprint("trap: %s", trapname(ureg->type));
    if(ureg != nil && (ureg->psr & PsrMask) != PsrMsvc)
        iprint(" in %s", trapname(ureg->psr));
    iprint("\n");
    iprint("psr %8.8lux type %2.2lux pc %8.8lux link %8.8lux\n",
        ureg->psr, ureg->type, ureg->pc, ureg->link);
    iprint("R14 %8.8lux R13 %8.8lux R12 %8.8lux R11 %8.8lux R10 %8.8lux\n",
        ureg->r14, ureg->r13, ureg->r12, ureg->r11, ureg->r10);
    iprint("R9  %8.8lux R8  %8.8lux R7  %8.8lux R6  %8.8lux R5  %8.8lux\n",
        ureg->r9, ureg->r8, ureg->r7, ureg->r6, ureg->r5);
    iprint("R4  %8.8lux R3  %8.8lux R2  %8.8lux R1  %8.8lux R0  %8.8lux\n",
        ureg->r4, ureg->r3, ureg->r2, ureg->r1, ureg->r0);
    iprint("stack is at %#p\n", ureg);
    iprint("pc %#lux link %#lux\n", ureg->pc, ureg->link);

    if(up)
        iprint("user stack: %#p-%#p\n", up->kstack, up->kstack+KSTACK-4);
    else
        iprint("kernel stack: %8.8lux-%8.8lux\n",
            (ulong)(cpu+1), (ulong)cpu+BY2PG-4);
```

```
    dumplongs("stack", (ulong *)(ureg + 1), 16);
    arch_delay(2000);
    arch_dumpstack();
    arch_splx(s);
}
```

Uses PsrMask 38, PsrMsvc 38, arch_delay, arch_dumpstack, dumplongs() 647a, iprint, trapname() 647c, and up.

⟨*function* dumplongs *(arm)* 647a⟩≡ (1002c)

```
static void
dumplongs(char *msg, ulong *v, int n)
{
    int i, l;

    l = 0;
    iprint("%s at %.8p: ", msg, v);
    for(i=0; i<n; i++){
        if(l >= 4){
            iprint("\n    %.8p: ", v);
            l = 0;
        }
        if(isvalidaddr(v)){
            iprint(" %.8lux", *v++);
            l++;
        }else{
            iprint(" invalid");
            break;
        }
    }
    iprint("\n");
}
```

Uses iprint and isvalidaddr() 647b.

⟨*function* isvalidaddr *(arm)* 647b⟩≡ (1002c)

```
int
isvalidaddr(void *v)
{
    return (uintptr)v >= KZERO;
}
```

⟨*function* trapname *(arm)* 647c⟩≡ (1002c)

```
static char *
trapname(int psr)
{
    char *s;

    s = trapnames[psr & PsrMask];
    if(s == nil)
        s = "unknown trap number in psr";
    return s;
}
```

Uses PsrMask 38 and trapnames-2414 161b.

⟨faultarm() *if debug* 647d⟩≡ (182b)

```
if (Debug)
    ckfaultstuck(va);
```

Uses Debug-2405 1002b and ckfaultstuck() 648a.

⟨*function* ckfaultstuck*(arm)* 648a⟩≡ (1002c)
```
/* this is quite helpful during mmu and cache debugging */
static void
ckfaultstuck(uintptr va)
{
    static int cnt, lastpid;
    static uintptr lastva;

    if (va == lastva && up->pid == lastpid) {
        ++cnt;
        if (cnt >= 2)
            /* fault() isn't fixing the underlying cause */
            panic("fault: %d consecutive faults for va %#p",
                cnt+1, va);
    } else {
        cnt = 0;
        lastva = va;
        lastpid = up->pid;
    }
}
```
Uses up.

A.3 /bin/ktrace (ARM)

⟨*function* trap_arch_dumpstack*(arm)* 648b⟩≡ (1002c)
```
void
trap_arch_dumpstack(void)
{
    arch_callwithureg(dumpstackwithureg);
}
```
Uses dumpstackwithureg() 649a.

⟨*function* arch_callwithureg*(arm)* 648c⟩≡ (1002c)
```
void
arch_callwithureg(void (*fn)(Ureg*))
{
    Ureg ureg;

    getpcsp((ulong*)&ureg.pc, (ulong*)&ureg.sp);
    ureg.r14 = getcallerpc(&fn);
    fn(&ureg);
}
```
Uses getpcsp() 648d.

⟨*function* getpcsp*(arm)* 648d⟩≡ (1002c)
```
/*
 * Fill in enough of Ureg to get a stack trace, and call a function.
 * Used by debugging interface rdb.
 */
static void
getpcsp(ulong *pc, ulong *sp)
{
    *pc = getcallerpc(&pc);
    *sp = (ulong)&pc-4;
}
```

⟨*function* dumpstackwithureg*(arm)* 649a⟩≡ (1002c)

```
static void
dumpstackwithureg(Ureg *ureg)
{
    uintptr l, i, v, estack;
    u32int *p;
    char *s;

    if((s = getconf("*nodumpstack")) != nil && strcmp(s, "0") != 0){
        iprint("dumpstack disabled\n");
        return;
    }
    iprint("ktrace /kernel/path %#.8lux %#.8lux %#.8lux # pc, sp, link\n",
        ureg->pc, ureg->sp, ureg->r14);
    arch_delay(2000);
    i = 0;
    if(up != nil && (uintptr)&l <= (uintptr)up->kstack+KSTACK)
        estack = (uintptr)up->kstack+KSTACK;
    else if((uintptr)&l >= (uintptr)cpu->stack
         && (uintptr)&l <= (uintptr)cpu + CPUSIZE)
        estack = (uintptr)cpu + CPUSIZE;
    else{
        if(up != nil)
            iprint("&up->kstack %#p &l %#p\n", up->kstack, &l);
        else
            iprint("&m %#p &l %#p\n", cpu, &l);
        return;
    }
    for(l = (uintptr)&l; l < estack; l += sizeof(uintptr)){
        v = *(uintptr*)l;
        if(KTZERO < v && v < (uintptr)etext && !(v & 3)){
            v -= sizeof(u32int);        /* back up an instr */
            p = (u32int*)v;
            if((*p & 0x0f000000) == 0x0b000000){    /* BL instr? */
                iprint("%#8.8lux=%#8.8lux ", l, v);
                i++;
            }
        }
        if(i == 4){
            i = 0;
            iprint("\n");
        }
    }
    if(i)
        iprint("\n");
}
```

Uses arch_delay, iprint, and up.

A.4 /dev/kmesg

⟨*struct* KMesg 649b⟩≡ (696b)

```
/*
 * Log console output so it can be retrieved via /dev/kmesg.
 * This is good for catching boot-time messages after the fact.
 */
struct KMesg {
    Lock lk;
    char buf[KMESGSIZE];
```

```
    uint n;
};
```

⟨*global* `kmesg` 650a⟩≡ (696b)
```
struct KMesg kmesg;
```
Uses `KMesg` 649b.

⟨*constant* `KMESGSIZE` *(arm)* 650b⟩≡ (973a)
```
#define KMESGSIZE (16*1024) // for /dev/kmesg
```

⟨`echo()` *hooks* 650c⟩+≡ (384a) ◁384b 653c▷
```
kmesgputs(buf, n);
```
Uses `kmesgputs()` 650e.

⟨`putstrn0()` *kmesg handling* 650d⟩≡ (390b)
```
/*
 *  how many different output devices do we need?
 */
kmesgputs(str, n);
```
Uses `kmesgputs()` 650e.

⟨*function* `kmesgputs` 650e⟩≡ (696b)
```
static void
kmesgputs(char *str, int n)
{
    uint nn, d;

    ilock(&kmesg.lk);
    /* take the tail of huge writes */
    if(n > sizeof kmesg.buf){
        d = n - sizeof kmesg.buf;
        str += d;
        n -= d;
    }

    /* slide the buffer down to make room */
    nn = kmesg.n;
    if(nn + n >= sizeof kmesg.buf){
        d = nn + n - sizeof kmesg.buf;
        if(d)
            memmove(kmesg.buf, kmesg.buf+d, sizeof kmesg.buf-d);
        nn -= d;
    }

    /* copy the data in */
    memmove(kmesg.buf+nn, str, n);
    nn += n;
    kmesg.n = nn;
    iunlock(&kmesg.lk);
}
```
Uses `ilock()` 79d, `iunlock()` 80b, and `kmesg` 650a.

⟨*devcons.c enum Qxxx cases* 650f⟩+≡ (388a) ◁633a 651b▷
```
Qkmesg,
```

⟨`consdir` *fields* 650g⟩+≡ (388b) ◁633b 651c▷
```
"kmesg",    {Qkmesg},   0,      0440,
```
Uses `Qkmesg-1794` 650f.

⟨consread() *cases* 651a⟩+≡ (394c) ◁637 652b▷
```
case Qkmesg:
    /*
     * This is unlocked to avoid tying up a process
     * that's writing to the buffer.  kmesg.n never
     * gets smaller, so worst case the reader will
     * see a slurred buffer.
     */
    if(off >= kmesg.n)
        n = 0;
    else{
        if(off+n > kmesg.n)
            n = kmesg.n - off;
        memmove(buf, kmesg.buf+off, n);
    }
    return n;
```
Uses Qkmesg-1794 650f and kmesg 650a.

A.5 /dev/kprint

⟨*devcons.c enum Qxxx cases* 651b⟩+≡ (388a) ◁650f
```
Qkprint,
```

⟨consdir *fields* 651c⟩+≡ (388b) ◁650g
```
"kprint",   {Qkprint, 0, QTEXCL},   0,  DMEXCL|0440,
```
Uses DMEXCL, QTEXCL, and Qkprint-1795 651b.

⟨Dirmode *cases* 651d⟩+≡ (337a) ◁432a
```
DMEXCL = 0x20000000,  /* mode bit for exclusive use files */
```

⟨*global* kprintoq 651e⟩≡ (696b)
```
Queue*  kprintoq;       /* console output, for /dev/kprint */
```

⟨*global* kprintinuse 651f⟩≡ (696b)
```
ulong   kprintinuse;        /* test and set whether /dev/kprint is open */
```

⟨consopen() *cases* 651g⟩+≡ (394a) ◁396d
```
case Qkprint:
    if(arch_tas(&kprintinuse) != 0){
        c->flag &= ~COPEN;
        error(Einuse);
    }
    if(kprintoq == nil){
        kprintoq = qopen(8*1024, Qcoalesce, 0, 0);
        if(kprintoq == nil){
            c->flag &= ~COPEN;
            error(Enomem);
        }
        qnoblock(kprintoq, true);
    }else
        qreopen(kprintoq);
    c->iounit = qiomaxatomic;
    break;
```
Uses COPEN 294b, Qcoalesce 399a, Qkprint-1795 651b, kprintinuse 651f, kprintoq 651e, qiomaxatomic 813b, qnoblock() 420c, qopen() 409a, qreopen() 419b, and true.

⟨consclose() *cases* 652a⟩+≡ (394b) ◁396e
```
/* close of kprint allows other opens */
case Qkprint:
    if(c->flag & COPEN){
        kprintinuse = 0;
        qhangup(kprintoq, nil);
    }
    break;
```
Uses COPEN 294b, Qkprint-1795 651b, kprintinuse 651f, kprintoq 651e, and qhangup() 418b.

⟨consread() *cases* 652b⟩+≡ (394c) ◁651a
```
case Qkprint:
    return qread(kprintoq, buf, n);
```
Uses Qkprint-1795 651b, kprintoq 651e, and qread() 413.

⟨putstrn0() *if kprint* 652c⟩≡ (390b)
```
/*
 *  if someone is reading /dev/kprint,
 *  put the message there.
 *  if not and there's an attached bit mapped display,
 *  put the message there.
 *
 *  if there's a serial line being used as a console,
 *  put the message there.
 */
if(kprintoq != nil && !qisclosed(kprintoq)){
    if(usewrite)
        qwrite(kprintoq, str, n);
    else
        qiwrite(kprintoq, str, n);
}
```
Uses kprintoq 651e, qisclosed() 419a, qiwrite() 416, and qwrite() 401b.

⟨panic() *reset kprintoq* 652d⟩≡ (663b)
```
kprintoq = nil; /* don't try to write to /dev/kprint */
```
Uses kprintoq 651e.

A.6 Remote debugger

⟨*hook consdebug* 652e⟩≡ (696b)
```
void    (*consdebug)(void) = nil; // for rdb
```
Uses consdebug 652e.

⟨panic() *run consdebug hook* 652f⟩≡ (663b)
```
if(consdebug)
    (*consdebug)();
```
Uses consdebug 652e.

⟨echo() *C-t C-t special keys handler other cases* 652g⟩≡ (642c)
```
case 'd':
    if(consdebug == nil)
        consdebug = rdb;
    else
        consdebug = nil;
    print("consdebug now %#p\n", consdebug);
    return;
case 'D':
```

```
    if(consdebug == nil)
        consdebug = rdb;
    consdebug();
    return;
```

Uses consdebug 652e, print, and rdb() 698b.

A.7 Serial line

On the Raspberry Pi, printing to the screen requires a working framebuffer, font rendering, and drawing primitives—none of which are available during early boot. The serial UART is the escape hatch: it works with a single register write, so `uartputs()` can output text from the very first instruction. This is how kernel developers debug boot failures.

A.7.1 serialoq

⟨*global* serialoq 653a⟩≡ (696b)

```
// option<Queue>
Queue*  serialoq;       /* serial console output */
```

⟨*function* consactive 653b⟩≡ (696b)

```
int
consactive(void)
{
    if(serialoq)
        return qlen(serialoq) > 0;
    return 0;
}
```

Uses qlen() 419c and serialoq 653a.

⟨echo() *hooks* 653c⟩+≡ (384a) ◁650c

```
if(serialoq)
    echoserialoq(buf, n);
```

Uses echoserialoq() 653d and serialoq 653a.

⟨*function* echoserialoq 653d⟩≡ (696b)

```
static void
echoserialoq(char *buf, int n)
{
    char *e, *p;
    char ebuf[128];
    int x;

    p = ebuf;
    e = ebuf + sizeof(ebuf) - 4;
    while(n-- > 0){
        if(p >= e){
            qiwrite(serialoq, ebuf, p - ebuf);
            p = ebuf;
        }
        x = *buf++;
        if(x == '\n'){
            *p++ = '\r';
            *p++ = '\n';
        } else if(x == 0x15){
            *p++ = '^';
            *p++ = 'U';
```

```
            *p++ = '\n';
        } else
            *p++ = x;
    }
    if(p != ebuf)
        qiwrite(serialoq, ebuf, p - ebuf);
}
```

Uses qiwrite() 416 and serialoq 653a.

⟨putstrn0() *serialoq handling* 654a⟩≡ (390b)

```
/*
 *   Convert \n to \r\n for serial
 *   line consoles.  Locking of the queues is left up to the screen
 *   or uart code.  Multi-line messages to serial consoles may get
 *   interspersed with other messages.
 */
⟨putstrn0() if serialoq nil 654b⟩
// else

while(n > 0) {
    t = memchr(str, '\n', n);
    if(t && !kbd.raw) {
        m = t-str;
        if(usewrite){
            qwrite(serialoq, str, m);
            qwrite(serialoq, "\r\n", 2);
        } else {
            qiwrite(serialoq, str, m);
            qiwrite(serialoq, "\r\n", 2);
        }
        n -= m+1;
        str = t+1;
    } else {
        if(usewrite)
            qwrite(serialoq, str, n);
        else
            qiwrite(serialoq, str, n);
        break;
    }
}
```

Uses kbd-1778 382b, qiwrite() 416, qwrite() 401b, and serialoq 653a.

A.7.2 uartconsinit()

A.7.3 uartputs()

⟨putstrn0() *if serialoq nil* 654b⟩≡ (654a)

```
if(serialoq == nil){
    uartputs(str, n);
    return;
}
```

Uses serialoq 653a and uartputs() 676d.

A.7.4 consuart

⟨*global* consuart 654c⟩≡ (676d)

```
Uart* consuart;
```

A.7.5 Uart, PhysUart

Appendix B

Profiling

This appendix collects the kernel's performance measurement code: per-CPU statistics (idle time, interrupt time), the `/dev/kprof` sampling profiler, and various counters for locks, qlocks, and scheduling. These are the "gauges on the dashboard" that let developers find bottlenecks.

B.1 `/sys/sysstat` (ARM)

⟨Cpu *stat fields* 656a⟩≡ (64a) 656f▷
```
  Perf  perf;     /* performance counters */
```

⟨*struct* Perf 656b⟩≡ (702)
```
  /*
   *  performance timers, all units in perfticks
   */
  struct Perf
  {
      // intr-ts? interrupt time stamp?
      ulong intrts;   /* time of last interrupt */

      ulong inintr;   /* time since last clock tick in interrupt handlers */
      ulong avg_inintr; /* avg time per clock tick in interrupt handlers */

      ulong inidle;   /* time since last clock tick in idle loop */
      ulong avg_inidle; /* avg time per clock tick in idle loop */

      ulong last;   /* value of arch_perfticks() at last clock tick */
      ulong period;   /* arch_perfticks() per clock tick */
  };
```

⟨runproc() *other locals* 656c⟩+≡ (225) ◁499f
```
  ulong start, now;
```

⟨runproc() *start* 656d⟩≡ (225)
```
  start = arch_perfticks();
```

⟨runproc() *after* arch_idlehands, *before restarting loop* 656e⟩≡ (225)
```
  /* remember how much time we're here */
  now = arch_perfticks();
  cpu->perf.inidle += now-start;
  start = now;
```

⟨Cpu *stat fields* 656f⟩+≡ (64a) ◁656a 658c▷
```
  int load;
```

⟨hzclock() *accounttime* 657a⟩≡ (236f)
```
accounttime();
```
Uses accounttime() 657b.

⟨*function* accounttime 657b⟩≡ (877c)
```
/*
 *  time accounting called by clock() splhi'd
 */
void
accounttime(void)
{
    Proc *p;
    ulong n, per;
    static ulong nrun;

    p = cpu->proc; // why not p = up?
    if(p) {
        nrun++;
        ⟨accountime() update time of current process 504a⟩
    }

    /* calculate decaying duty cycles */
    n = arch_perfticks();
    per = n - cpu->perf.last;
    cpu->perf.last = n;
    per = (cpu->perf.period*(Arch_HZ-1) + per)/Arch_HZ;
    if(per != 0)
        cpu->perf.period = per;

    cpu->perf.avg_inidle =
      (cpu->perf.avg_inidle*(Arch_HZ-1)+cpu->perf.inidle)/Arch_HZ;
    cpu->perf.inidle = 0;

    cpu->perf.avg_inintr =
      (cpu->perf.avg_inintr*(Arch_HZ-1)+cpu->perf.inintr)/Arch_HZ;
    cpu->perf.inintr = 0;

    /* only one processor gets to compute system load averages */
    if(cpu->cpuno != 0)
        return;

    /*
     * calculate decaying load average.
     * if we decay by (n-1)/n then it takes
     * n clock ticks to go from load L to .36 L once
     * things quiet down.  it takes about 5 n clock
     * ticks to go to zero.  so using HZ means this is
     * approximately the load over the last second,
     * with a tail lasting about 5 seconds.
     */
    n = nrun;
    nrun = 0;
    n = (nrdy+n)*1000;
    cpu->load = (cpu->load*(Arch_HZ-1)+n)/Arch_HZ;
}
```
Uses nrdy 222a.

⟨*function* intrtime*(arm)* 657c⟩≡ (1002c)
```
static void
intrtime(void)
```

```
{
    ulong diff;
    ulong x;

    x = arch_perfticks();
    diff = x - cpu->perf.intrts;
    cpu->perf.intrts = x;

    cpu->perf.inintr += diff;
    if(up == nil && cpu->perf.inidle > diff)
        cpu->perf.inidle -= diff;
}
```

Uses up.

⟨*function* arch_perfticks*(arm)* 658a⟩≡ (1011b)

```
ulong
arch_perfticks(void)
{
    Armtimer *tm;

    tm = (Armtimer*)ARMTIMER;
    return tm->count;
}
```

Uses ARMTIMER-2648 43c.

⟨*struct* Armtimer*(arm)* 658b⟩≡ (1011b)

```
struct Armtimer {
    u32int  load;
    u32int  val;
    u32int  ctl;
    u32int  irqack;
    u32int  irq;
    u32int  maskedirq;
    u32int  reload;
    u32int  predivider;
    u32int  count;
};
```

⟨Cpu *stat fields* 658c⟩+≡ (64a) ◁656f

```
int cs; // context switch, sched() and sleep() call
int intr;
int syscall;
int pfault;
int tlbfault;
int tlbpurge;
ulong spuriousintr; // not really used
```

⟨*devsys.c enum Qxxx cases* 658d⟩+≡ (622a) ◁626d

```
Qsysstat,
```

⟨sysdir *fields* 658e⟩+≡ (622b) ◁626e

```
"sysstat",  {Qsysstat}, 0,      0666,
```

Uses Qsysstat-844 658d.

⟨sysread() *cases* 659a⟩+≡ (622e) ◁626c

```
case Qsysstat:
    b = smalloc(conf.ncpu*(NUMSIZE*11+1) + 1); /* +1 for NUL */
    bp = b;
    for(id = 0; id < MAXCPUS; id++) {
        if(active.cpus & (1<<id)) {
            mp = CPUS(id);
            readnum(0, bp, NUMSIZE, id, NUMSIZE);
            bp += NUMSIZE;
            readnum(0, bp, NUMSIZE, mp->cs, NUMSIZE);
            bp += NUMSIZE;
            readnum(0, bp, NUMSIZE, mp->intr, NUMSIZE);
            bp += NUMSIZE;
            readnum(0, bp, NUMSIZE, mp->syscall, NUMSIZE);
            bp += NUMSIZE;
            readnum(0, bp, NUMSIZE, mp->pfault, NUMSIZE);
            bp += NUMSIZE;
            readnum(0, bp, NUMSIZE, mp->tlbfault, NUMSIZE);
            bp += NUMSIZE;
            readnum(0, bp, NUMSIZE, mp->tlbpurge, NUMSIZE);
            bp += NUMSIZE;
            readnum(0, bp, NUMSIZE, mp->load, NUMSIZE);
            bp += NUMSIZE;
            readnum(0, bp, NUMSIZE,
                (mp->perf.avg_inidle*100)/mp->perf.period,
                NUMSIZE);
            bp += NUMSIZE;
            readnum(0, bp, NUMSIZE,
                (mp->perf.avg_inintr*100)/mp->perf.period,
                NUMSIZE);
            bp += NUMSIZE;
            *bp++ = '\n';
        }
    }
    if(waserror()){
        free(b);
        nexterror();
    }
    n = readstr((ulong)offset, buf, n, b);
    free(b);
    poperror();
    return n;
```

Uses CPUS 702, NUMSIZE 867a, Qsysstat-844 658d, active 65e, conf 62b, free() 104a, nexterror, poperror 666d, readnum() 670c, readstr() 670d, smalloc() 104b, and waserror 666d.

⟨syswrite() *cases* 659b⟩+≡ (623b) ◁627b

```
case Qsysstat:
    for(id = 0; id < 32; id++) {
        if(active.cpus & (1<<id)) {
            mp = CPUS(id);
            mp->cs = 0;
            mp->intr = 0;
            mp->syscall = 0;
            mp->pfault = 0;
            mp->tlbfault = 0;
            mp->tlbpurge = 0;
        }
    }
    break;
```

Uses CPUS 702, Qsysstat-844 658d, and active 65e.

B.2 /dev/kprof

⟨hzclock() *if kproftimer* 660a⟩≡ (236f)
```
if(kproftimer != nil)
    kproftimer(ur->pc);
```
Uses kproftimer 660b.

⟨*hook kproftimer* 660b⟩≡ (873a)
```
void (*kproftimer)(ulong);
```

B.3 Statistics

B.3.1 Lock.lockcycles

⟨*lock ifdef LOCKCYCLES* 660c⟩≡ (79d 78f 76)
```
#ifdef LOCKCYCLES
        l->lockcycles = -arch_lcycles();
#endif
```

⟨Lock *debugging fields* 660d⟩+≡ (76c) ◁88a
```
//#ifdef LOCKCYCLES
long  lockcycles;
//#endif
```

⟨*unlock ifdef LOCKCYCLES* 660e⟩≡ (78d)
```
#ifdef LOCKCYCLES
    l->lockcycles += lcycles();
    cumlockcycles += l->lockcycles;
    if(l->lockcycles > maxlockcycles){
        maxlockcycles = l->lockcycles;
        maxlockpc = l->pc;
    }
#endif
```

⟨*iunlock ifdef LOCKCYCLES* 660f⟩≡ (80b)
```
#ifdef LOCKCYCLES
    l->lockcycles += lcycles();
    cumilockcycles += l->lockcycles;
    if(l->lockcycles > maxilockcycles){
        maxilockcycles = l->lockcycles;
        maxilockpc = l->pc;
    }
    if(l->lockcycles > 2400)
        ilockpcs[n++ & 0xff]  = l->pc;
#endif
```

⟨*globals lockcycles* 660g⟩≡ (694a)
```
#ifdef LOCKCYCLES
long maxlockcycles;
long maxilockcycles;
long cumlockcycles;
long cumilockcycles;
ulong maxlockpc;
ulong maxilockpc;

ulong ilockpcs[0x100] = { [0xff] = 1 };
static int n;
#endif
```

B.3.2 TaslockStats

⟨*struct* TaslockStats 661a⟩≡ (694a)
```
struct TaslockStats
{
    ulong   locks;
    ulong   glare;
    ulong   inglare;
};
```

⟨lock() *start of lock, stat* 661b⟩≡ (76d)
```
lockstats.locks++;
```
Uses lockstats 661e.

⟨lock() *when lock already held, stat* 661c⟩≡ (76d)
```
lockstats.glare++;
```
Uses lockstats 661e.

⟨lock() *in loop when lock already held, stat* 661d⟩≡ (76d)
```
lockstats.inglare++;
```
Uses lockstats 661e.

⟨*global* lockstats 661e⟩≡ (694a)
```
struct TaslockStats lockstats;
```
Uses TaslockStats 661a.

B.3.3 QlockStats

⟨*struct* QlockStats 661f⟩≡ (694b)
```
struct QlockStats {
    ulong rlock;
    ulong rlockq;
    ulong wlock;
    ulong wlockq;
    ulong qlock;
    ulong qlockq;
};
```

⟨*global* rwstats 661g⟩≡ (694b)
```
struct QlockStats rwstats;
```
Uses QlockStats 661f.

⟨qlock() *start of qlock, stat* 661h⟩≡ (81f)
```
rwstats.qlock++;
```
Uses rwstats 661g.

⟨qlock() *when qlock already held, stat* 661i⟩≡ (81f)
```
rwstats.qlockq++;
```
Uses rwstats 661g.

B.3.4 Kimage statistics

⟨*struct* Irstats 661j⟩≡ (864b)
```
struct Irstats {
    int calls;          /* times imagereclaim was called */
    int loops;          /* times the main loop was run */
    uvlong  ticks;          /* total time in the main loop */
    uvlong  maxt;           /* longest time in main loop */
};
```

⟨*segment.c global irstats* 662a⟩≡ (864b)

```
static struct Irstats  irstats;
```

Uses `Irstats` 661j.

B.3.5 Scheduling

⟨*proc.c statistics* 662b⟩≡ (877c)

```
long preempts;
ulong delayedscheds;    /* statistics */
long skipscheds;
```

B.3.6 Clock

⟨*clock.c statistics* 662c⟩≡ (874c)

```
ulong intrcount[MAXCPUS];
ulong fcallcount[MAXCPUS];
```

⟨`timerintr()` *start, stat* 662d⟩≡ (252)

```
intrcount[cpu->cpuno]++;
```

Uses `intrcount` 662c.

⟨`timerintr()` *in timers loop, stat* 662e⟩≡ (252)

```
fcallcount[cpu->cpuno]++;
```

Uses `fcallcount` 662c.

Appendix C

Error Management

This appendix covers the kernel's error handling mechanisms: `panic()` for unrecoverable errors that halt the system, `error()` for recoverable errors that unwind the call stack (using `setjmp`/`longjmp`-style labels), and `waserror()`/`poperror()` which together form a try/catch mechanism in C. The `errstr` syscall lets user programs retrieve error strings—Plan 9's replacement for UNIX's numeric `errno`.

C.1 `panic()`

⟨*global* `panicking` 663a⟩≡ (696b)
```
bool panicking;
```

⟨*function* `panic` 663b⟩≡ (696b)
```
void
devcons_panic(char *fmt, ...)
{
    int n, s;
    va_list arg;
    char buf[PRINTSIZE];

    ⟨panic() reset kprintoq 652d⟩

    if(panicking)
        for(;;);
    panicking = true;

    s = arch_splhi();
    strcpy(buf, "panic: ");
    va_start(arg, fmt);
    n = vseprint(buf+strlen(buf), buf+sizeof(buf), fmt, arg) - buf;
    va_end(arg);
    iprint("%s\n", buf);
    ⟨panic() run consdebug hook 652f⟩
    arch_splx(s);
    prflush();
    buf[n] = '\n';
    putstrn(buf, n+1);
    //TODO: put in comment for now because already got some panic with
    // lapic, but it seems to work still :)
    //arch_dumpstack();
    //exit(1);
}
```
Uses `PRINTSIZE` 391a, `iprint`, `panicking` 663a, `prflush()` 664a, `putstrn()` 391c, and `true`.

⟨*function* `prflush` 664a⟩≡ (696b)
```
void
prflush(void)
{
    ulong now;

    now = cpu->ticks;
    while(consactive())
        if(cpu->ticks - now >= Arch_HZ)
            break;
}
```
Uses `consactive()` 653b.

⟨*function* `sysfatal` 664b⟩≡ (696b)
```
/* libmp at least contains a few calls to sysfatal; simulate with panic */
// note that this is not a system call, even though it's prefixed with sys
//@Scheck: no dead, override also sysfatal from libc/9sys/sysfatal.c
void sysfatal(char *fmt, ...)
{
    char err[256];
    va_list arg;

    va_start(arg, fmt);
    vseprint(err, err + sizeof err, fmt, arg);
    va_end(arg);
    panic("sysfatal: %s", err);
}
```

⟨*function* `_assert` 664c⟩≡ (696b)
```
void
devcons__assert(char *fmt)
{
    panic("assert failed at %#p: %s", getcallerpc(&fmt), fmt);
}
```

C.2 `error()`

⟨*global Exxx errors* 664d⟩≡ (705a)
```
char Emount[] = "inconsistent mount";
char Eunmount[] = "not mounted";
char Eismtpt[] = "is a mount point";
char Eunion[] = "not in union";
char Emountrpc[] = "mount rpc error";
char Eshutdown[] = "device shut down";
char Enocreate[] = "mounted directory forbids creation";
char Enonexist[] = "file does not exist";
char Eexist[] = "file already exists";
char Ebadsharp[] = "unknown device in # filename";
char Enotdir[] = "not a directory";
char Eisdir[] = "file is a directory";
char Ebadchar[] = "bad character in file name";
char Efilename[] = "file name syntax";
char Eperm[] = "permission denied";
char Ebadusefd[] = "inappropriate use of fd";
char Ebadarg[] = "bad arg in system call";
char Einuse[] = "device or object already in use";
char Eio[] = "i/o error";
char Etoobig[] = "read or write too large";
```

```
char Etoosmall[] = "read or write too small";
char Ehungup[] = "i/o on hungup channel";
char Ebadctl[] = "bad process or channel control request";
char Enodev[] = "no free devices";
char Eprocdied[] = "process exited";
char Enochild[] = "no living children";
char Eioload[] = "i/o error in demand load";
char Enovmem[] = "virtual memory allocation failed";
char Ebadfd[] = "fd out of range or not open";
char Enofd[] = "no free file descriptors";
char Eisstream[] = "seek on a stream";
char Ebadexec[] = "exec header invalid";
char Etimedout[] = "connection timed out";
char Econrefused[] = "connection refused";
char Econinuse[] = "connection in use";
char Eintr[] = "interrupted";
char Enomem[] = "kernel allocate failed";
char Esoverlap[] = "segments overlap";
char Eshort[] = "i/o count too small";
char Egreg[] = "jmk added reentrancy for threads";
char Ebadspec[] = "bad attach specifier";
char Enoreg[] = "process has no saved registers";
char Enoattach[] = "mount/attach disallowed";
char Eshortstat[] = "stat buffer too small";
char Ebadstat[] = "malformed stat buffer";
char Enegoff[] = "negative i/o offset";
char Ecmdargs[] = "wrong #args in control message";
char Ebadip[] = "bad ip address syntax";
char Edirseek[] = "seek in directory";
char Echange[] = "media or partition has changed";
char Edetach[] = "device is detached";
char Enotconf[] = "endpoint not configured";
char Estalled[] = "endpoint stalled";
char Esbadstat[] = "invalid directory entry received from server";
char Enoversion[] = "version not established for mount channel";
```

Uses Ebadarg 664d, Ebadchar 664d, Ebadexec 664d, Ebadfd 664d, Ebadip 664d, Ebadsharp 664d, Ebadspec 664d, Ebadusefd 664d, Echange 664d, Ecmdargs 664d, Econinuse 664d, Econrefused 664d, Edetach 664d, Edirseek 664d, Efilename 664d, Egreg 664d, Ehungup 664d, Eioload 664d, Eismtpt 664d, Eisstream 664d, Emount 664d, Emountrpc 664d, Enegoff 664d, Enoattach 664d, Enochild 664d, Enocreate 664d, Enodev 664d, Enofd 664d, Enoreg 664d, Enotconf 664d, Enoversion 664d, Enovmem 664d, Eprocdied 664d, Esbadstat 664d, Eshort 664d, Eshortstat 664d, Eshutdown 664d, Esoverlap 664d, Estalled 664d, Etimedout 664d, Etoobig 664d, Eunion 664d, and Eunmount 664d.

⟨*global* Edoesnotexist 665a⟩≡ (808c)

```
static char Edoesnotexist[] = "does not exist";
```

Uses Edoesnotexist-768 665a.

⟨Proc *error managment fields* 665b⟩≡ (66) 667b ▷

```
// array<Label>, error labels, poor's man exceptions in C
Label errlab[NERR];
// length(errlab) used.
int nerrlab;

// ref<string> point to errbuf0 or to syserrstr (which points to errbuf1)
char  *errstr;  /* reason we're unwinding the error stack, errbuf1 or 0 */
char  errbuf0[ERRMAX];
char  errbuf1[ERRMAX];
```

⟨newproc() *set fields of* p 665c⟩+≡ (130a) ◁618b

```
p->nerrlab = 0;
p->errstr = p->errbuf0;
```

```
  p->syserrstr = p->errbuf1;
  p->errbuf0[0] = '\0';
  p->errbuf1[0] = '\0';
```

⟨*constant* NERR 666a⟩≡ (868c)
```
  NERR = 64,
```

⟨*function* error 666b⟩≡ (877c)
```
  void
  proc_error(char *err)
  {
      arch_spllo();

      assert(up->nerrlab < NERR);
      kstrcpy(up->errstr, err, ERRMAX);
      arch_setlabel(&up->errlab[NERR-1]);
      nexterror();
  }
```
Uses ERRMAX, NERR 666a, assert 853f, kstrcpy() 669a, nexterror, and up.

⟨*function* exhausted 666c⟩≡ (877c)
```
  void
  exhausted(char *resource)
  {
      char buf[ERRMAX];

      snprint(buf, sizeof buf, "no free %s", resource);
      iprint("%s\n", buf);
      error(buf);
  }
```
Uses ERRMAX and iprint.

C.3 waserror(), nexterror(), poperror()

This is Plan 9's exception mechanism in C, built on arch_setlabel/arch_gotolabel (the same primitives used by the scheduler). waserror() pushes a label onto a per-process stack and returns false. If error() or nexterror() is later called, it pops the stack and jumps back to the waserror() call site, which now returns true—entering the "catch" branch. poperror() simply decrements the stack when the protected code completes normally.

⟨*macro waserror poperror* 666d⟩≡ (868c)
```
  // poor's man exceptions in C
  //  - waserror() =~ try
  //     * if (!waserror()) { } else { } <=> try { } catch { }
  //     * if (waserror()) { }  <=> finally { }
  //  - poperror() = nothing
  //  - error() =~ raise
  //  - nexterror() =~ re raise from exn handler
  // note, arch_setlabel() return false, so the branch is never taken first
  // but nexterror() is using arch_gotolabel() which returns true, see l_switch.s
  #define waserror()  (up->nerrlab++, arch_setlabel(&up->errlab[up->nerrlab-1]))
  #define poperror()    up->nerrlab--
```

⟨*function* nexterror 667a⟩≡ (877c)
```
// raise an exception
void
proc_nexterror(void)
{
    arch_gotolabel(&up->errlab[--up->nerrlab]);
}
```
Uses up.

C.4 syserrstr()

⟨Proc *error managment fields* 667b⟩+≡ (66) ◁665b
```
char  *syserrstr; /* last error from a system call, errbuf0 or 1 */
```

⟨*syscall errstr* 667c⟩≡ (881b)
```
// int errstr(char *err, uint nerr);
long
syserrstr(ulong* arg)
{
    return generrstr((char*)arg[0], arg[1]);
}
```
Uses generrstr() 667d.

⟨*function* generrstr 667d⟩≡ (881b)
```
static long
generrstr(char *buf, uint nbuf)
{
    char tmp[ERRMAX];

    if(nbuf == 0)
        error(Ebadarg);
    validaddr((ulong)buf, nbuf, true);
    if(nbuf > sizeof tmp)
        nbuf = sizeof tmp;
    memmove(tmp, buf, nbuf);

    /* make sure it's NUL-terminated */
    tmp[nbuf-1] = '\0';
    memmove(buf, up->syserrstr, nbuf);
    buf[nbuf-1] = '\0';
    memmove(up->syserrstr, tmp, nbuf);
    return 0;
}
```
Uses ERRMAX, Ebadarg 664d, true, up, and validaddr() 120b.

⟨*function* werrstr 667e⟩≡ (881b)
```
//@Scheck: this is also defined in libc, so it's supposed to override it? TODO
void werrstr(char *fmt, ...)
{
    va_list va;

    if(up == nil)
        return;

    va_start(va, fmt);
    vseprint(up->syserrstr, up->syserrstr+ERRMAX, fmt, va);
    va_end(va);
}
```
Uses ERRMAX and up.

Appendix D

Utilities

This appendix collects low-level utility code used throughout the kernel: memory area operations (`memmove`, `memset`), string functions, and simple data structure helpers.

D.1 Memory areas operations

⟨*lib.h mem functions decl* 668a⟩≡ (853i)
```
extern  void* memccpy(void*, void*, int, ulong);
extern  void* memset(void*, int, ulong);
extern  int memcmp(void*, void*, ulong);
extern  void* memmove(void*, void*, ulong);
extern  void* memchr(void*, int, ulong);
```

⟨*function* `memrchr` 668b⟩≡ (808c)
```
void*
memrchr(void *va, int c, long n)
{
    byte *a, *e;

    a = va;
    for(e=a+n-1; e>a; e--)
        if(*e == c)
            return e;
    return nil;
}
```

D.2 String functions

⟨*lib.h string functions decl* 668c⟩≡ (853i)
```
extern  char* strchr(char*, int);
extern  char* strrchr(char*, int);
extern  int strcmp(char*, char*);
extern  char* strcpy(char*, char*);
extern  char* strecpy(char*, char*, char*);
extern  char* strncpy(char*, char*, long);
extern  int strncmp(char*, char*, long);
extern  long  strlen(char*);
extern  char* strstr(char*, char*);
extern  int atoi(char*);
extern  int fullrune(char*, int);
```

⟨*function* kstrcpy 669a⟩≡ (861b)

```
/*
 * Rather than strncpy, which zeros the rest of the buffer, kstrcpy
 * truncates if necessary, always zero terminates, does not zero fill,
 * and puts ... at the end of the string if it's too long.  Usually used to
 * save a string in up->genbuf;
 */
void
kstrcpy(char *s, char *t, int ns)
{
    int nt;

    nt = strlen(t);
    if(nt+1 <= ns){
        memmove(s, t, nt+1);
        return;
    }
    /* too long */
    if(ns < 4){
        /* but very short! */
        strncpy(s, t, ns);
        return;
    }
    /* truncate with ... at character boundary (very rare case) */
    memmove(s, t, ns-4);
    ns -= 4;
    s[ns] = '\0';
    /* look for first byte of UTF-8 sequence by skipping continuation bytes */
    while(ns>0 && (s[--ns]&0xC0)==0x80)
        ;
    strcpy(s+ns, "...");
}
```

⟨*function* kstrdup 669b⟩≡ (861b)

```
/*
 * Atomically replace *p with copy of s
 */
void
kstrdup(char **p, char *s)
{
    int n;
    char *t, *prev;

    n = strlen(s)+1;
    /* if it's a user, we can wait for memory; if not, something's very wrong */
    if(up){
        t = smalloc(n);
        setmalloctag(t, getcallerpc(&p));
    }else{
        t = malloc(n);
        if(t == nil)
            panic("kstrdup: no memory");
    }
    memmove(t, s, n);
    prev = *p;
    *p = t;
    free(prev);
}
```

Uses `free()` 104a, `malloc()` 103c, `setmalloctag()` 106a, `smalloc()` 104b, and `up`.

⟨*function* `emptystr` 670a⟩≡ (808c)
```
bool
emptystr(char *s)
{
    if(s == nil)
        return true;
    if(s[0] == '\0')
        return true;
    return false;
}
```
Uses `false` and `true`.

D.3 Conversion functions

⟨*lib.h strto functions decl* 670b⟩≡ (853i)
```
extern  long  strtol(char*, char**, int);
extern  ulong strtoul(char*, char**, int);
extern  vlong strtoll(char*, char**, int);
extern  uvlong  strtoull(char*, char**, int);
```

⟨*function* `readnum` 670c⟩≡ (705c)
```
int
readnum(ulong off, char *buf, ulong n, ulong val, int size)
{
    char tmp[64];

    snprint(tmp, sizeof(tmp), "%*lud", size-1, val);
    tmp[size-1] = ' ';
    if(off >= size)
        return 0;
    if(off+n > size)
        n = size-off;
    memmove(buf, tmp+off, n);
    return n;
}
```

⟨*function* `readstr` 670d⟩≡ (705c)
```
int
readstr(ulong off, char *buf, ulong n, char *str)
{
    int size;

    size = strlen(str);
    if(off >= size)
        return 0;
    if(off+n > size)
        n = size-off;
    memmove(buf, str+off, n);
    return n;
}
```

D.4 Pool allocation

D.5 Printf and `Fmt`

⟨*lib.h printf functions decl* 670e⟩≡ (853i)

```
extern  char* seprint(char*, char*, char*, ...);
extern  char* vseprint(char*, char*, char*, va_list);
extern  int snprint(char*, int, char*, ...);
extern  int sprint(char*, char*, ...);
```

⟨*struct* Fmt 671a⟩≡ (853i)

```
struct Fmt{
  byte runes;      /* output buffer is runes or chars? */
  void  *start;     /* of buffer */
  void  *to;      /* current place in the buffer */
  void  *stop;      /* end of the buffer; overwritten if flush fails */
  int (*flush)(Fmt *);  /* called when to == stop */
  void  *farg;      /* to make flush a closure */
  int nfmt;     /* num chars formatted so far */
  va_list args;     /* args passed to dofmt */
  int r;      /* % format Rune */
  int width;
  int prec;
  ulong flags;
};
```

⟨*lib.h fmt functions decl* 671b⟩≡ (853i)

```
extern  int fmtstrinit(Fmt*);
extern  int fmtinstall(int, int (*)(Fmt*));
extern  void  quotefmtinstall(void);
extern  int fmtprint(Fmt*, char*, ...);
extern  int fmtstrcpy(Fmt*, char*);
extern  char* fmtstrflush(Fmt*);
```

⟨*lib.h pragmas* 671c⟩≡ (853i)

```
#pragma varargck  argpos  fmtprint  2
#pragma varargck  argpos  print   1
#pragma varargck  argpos  seprint   3
#pragma varargck  argpos  snprint   3
#pragma varargck  argpos  sprint    2

#pragma varargck  type  "lld" vlong
#pragma varargck  type  "llx" vlong
#pragma varargck  type  "lld" uvlong
#pragma varargck  type  "llx" uvlong
#pragma varargck  type  "ld"  long
#pragma varargck  type  "lx"  long
#pragma varargck  type  "ld"  ulong
#pragma varargck  type  "lx"  ulong
#pragma varargck  type  "d" int
#pragma varargck  type  "x" int
#pragma varargck  type  "c" int
#pragma varargck  type  "C" int
#pragma varargck  type  "d" uint
#pragma varargck  type  "x" uint
#pragma varargck  type  "c" uint
#pragma varargck  type  "C" uint
#pragma varargck  type  "s" char*
#pragma varargck  type  "q" char*
#pragma varargck  type  "S" Rune*
#pragma varargck  type  "%" void
#pragma varargck  type  "p" uintptr
#pragma varargck  type  "p" void*
#pragma varargck  flag  ','
```

D.6 Runes, a.k.a unicode

⟨*enum* utf 672a⟩≡ (853i)
```
enum
{
  UTFmax    = 4,    /* maximum bytes per rune */
  Runeself  = 0x80,   /* rune and UTF sequences are the same (<) */
};
```

⟨*lib.h rune functions decl* 672b⟩≡ (853i)
```
extern  int runetochar(char*, Rune*);
extern  int chartorune(Rune*, char*);
extern  char* utfrune(char*, long);
extern  int utfnlen(char*, long);
```

⟨Kbscan *other fields* 672c⟩+≡ (378a) ◁397b
```
bool collecting;
int nk;
Rune    kc[5];
```

⟨kbdputsc() *if collecting* 672d⟩≡ (378d)
```
if(kbscan->collecting){
    int i;
    // pad's additional overflow checking, just to make sure
    if(kbscan->nk >= nelem(kbscan->kc)) {
      kbscan->nk = 0;
      kbscan->collecting = false;
      print("collecting overflow, possible bug in latin1()\n");
    }
    kbscan->kc[kbscan->nk++] = c;
    c = latin1(kbscan->kc, kbscan->nk);
    if(c < -1)  /* need more keystrokes */
        return;
    if(c != -1) /* valid sequence */
        kbdputc(c);
    else    /* dump characters */
        for(i=0; i<kbscan->nk; i++)
            kbdputc(kbscan->kc[i]);
    kbscan->nk = 0;
    kbscan->collecting = false;
}
```
Uses `false`, `kbdputc()` 382c, `latin1()` 709, `nelem` 853i, and `print`.

⟨kbdputsc() *start collecting* 672e⟩≡ (378d)
```
/*
 * VMware and Qemu use Ctl-Alt as the key combination
 * to make the VM give up keyboard and mouse focus.
 * This has the unfortunate side effect that when you
 * come back into focus, Plan 9 thinks you want to type
 * a compose sequence (you just typed alt).
 *
 * As a clumsy hack around this, we look for ctl-alt
 * and don't treat it as the start of a compose sequence.
 */
if(!kbscan->ctl){
    kbscan->collecting = true;
    kbscan->nk = 0;
}
```
Uses `true`.

⟨latin1.h 672f⟩≡

Appendix E

Extra Portable Code

E.1 ../include/

include/trace.h

⟨trace.h 673a⟩≡
```
⟨enum Tevent 499b⟩

⟨struct Traceevent 499c⟩

typedef enum Tevent Tevent;
typedef struct Traceevent Traceevent;
```

include/tos.h

⟨tos.h 673b⟩≡
```
typedef struct Plink Plink;
#pragma incomplete Plink

⟨struct Tos 507c⟩
typedef struct Tos Tos;

extern Tos *_tos;
```

E.2 buses/

buses/portdat_buses.h

⟨struct ISAConf 673c⟩≡ (674b)
```
struct ISAConf {
  char  *type;
  ulong port;
  int irq;
  ulong dma;
  ulong mem;
  ulong size;
  ulong freq;

  int nopt;
  char  *opt[NISAOPT];
};
```

⟨*constant* NISAOPT 674a⟩≡ (674b)

```
/*
 *  a parsed plan9.ini line
 */
#define NISAOPT   8
```

⟨buses/portdat_buses.h 674b⟩≡

```
⟨constant NISAOPT 674a⟩

⟨struct ISAConf 673c⟩

/*
 *  routines to access UART hardware
 */
struct PhysUart
{
  char* name;
  Uart* (*pnp)(void);
  void  (*enable)(Uart*, int);
  void  (*disable)(Uart*);
  void  (*kick)(Uart*);
  void  (*dobreak)(Uart*, int);
  int (*baud)(Uart*, int);
  int (*bits)(Uart*, int);
  int (*stop)(Uart*, int);
  int (*parity)(Uart*, int);
  void  (*modemctl)(Uart*, int);
  void  (*rts)(Uart*, int);
  void  (*dtr)(Uart*, int);
  long  (*status)(Uart*, void*, long, long);
  void  (*fifo)(Uart*, int);
  void  (*power)(Uart*, int);
  int (*getc)(Uart*); /* polling versions, for iprint, rdb */
  void  (*putc)(Uart*, int);
};

/*
 *  software UART
 */
struct Uart
{
  void* regs;     /* hardware stuff */
  void* saveregs;   /* place to put registers on power down */
  char* name;     /* internal name */
  ulong freq;     /* clock frequency */
  int bits;     /* bits per character */
  int stop;     /* stop bits */
  int parity;     /* even, odd or no parity */
  int baud;     /* baud rate */
  PhysUart*phys;
  bool console;    /* used as a serial console */
  int special;    /* internal kernel device */
  Uart* next;     /* list of allocated uarts */

  QLock;
  int type;     /* ?? */
  int dev;
  int opens;
```

```
  int enabled;
  Uart  *elist;     /* next enabled interface */

  int perr;     /* parity errors */
  int ferr;     /* framing errors */
  int oerr;     /* rcvr overruns */
  int berr;     /* no input buffers */
  int serr;     /* input queue overflow */

  /* buffers */
  int (*putc)(Queue*, int);
  Queue *iq;
  Queue *oq;

  Lock  rlock;
  uchar istage[STAGESIZE];
  uchar *iw;
  uchar *ir;
  uchar *ie;

  Lock  tlock;      /* transmit */
  uchar ostage[STAGESIZE];
  uchar *op;
  uchar *oe;
  int drain;

  int modem;      /* hardware flow control on */
  int xonoff;     /* software flow control on */
  int blocked;
  int cts, dsr, dcd;  /* keep track of modem status */
  int ctsbackoff;
  int hup_dsr, hup_dcd; /* send hangup upstream? */
  int dohup;

  Rendez  r;
};

extern  Uart* consuart;

extern Dev uartdevtab;
extern PhysUart* physuart[];

//void (*lprint)(char *, int);
```

buses/portfns_buses.h

⟨portfns_buses.h 675⟩≡

```
// devuart.c
int   uartgetc(void);
void  uartputs(char*, int);
void  uartrecv(Uart*, char);
int   uartctl(Uart*, char*);
int   uartstageoutput(Uart*);
void  uartkick(void*);

// TODO: move outside main.c?
// <arch>/main.c for now (called from port)
//int arch_isaconfig(char*, int, ISAConf*); // now in core/ for backward deps
```

buses/devuart.c

⟨*function* kbdcr2nl 676a⟩≡ (696b)
```
/*
 *  Called by a uart interrupt for console input.
 *
 *  turn '\r' into '\n' before putting it into the queue.
 */
int
kbdcr2nl(Queue*, int ch)
{
    char *next;

    ilock(&kbd.lockputc);       /* just a mutex */
    if(ch == '\r' && !kbd.raw)
        ch = '\n';
    next = kbd.iw+1;
    if(next >= kbd.ie)
        next = kbd.istage;
    if(next != kbd.ir){
        *kbd.iw = ch;
        kbd.iw = next;
    }
    iunlock(&kbd.lockputc);
    return 0;
}
```
Uses ilock() 79d, iunlock() 80b, and kbd-1778 382b.

⟨*devuart.c forward decl* 676b⟩≡ (676d)
```
static void uartclock(void);
static void uartflow(void*);
```

⟨*constant* STAGESIZE*(x86)* 676c⟩≡
```
// used by devuart.c
STAGESIZE = 2048, // default is 64
```

⟨buses/devuart.c 676d⟩≡
⟨*kernel basic includes* 49a⟩

```
#include    "io.h"
#include    "../port/netif.h"

enum
{
    /* soft flow control chars */
    CTLS= 023,
    CTLQ= 021,
};

static Uart* uartlist;
static Uart** uart;
static int uartnuart;
static Dirtab *uartdir;
static int uartndir;
static Timer *uarttimer;

struct Uartalloc {
```

```
    Lock;
    Uart *elist;    /* list of enabled interfaces */
} uartalloc;
```

⟨*devuart.c forward decl* 676b⟩

```
/*
 *  enable/disable uart and add/remove to list of enabled uarts
 */
Uart*
uartenable(Uart *p)
{
    Uart **l;

    if(p->enabled)
        return p;
    if(p->iq == nil){
        if((p->iq = qopen(8*1024, 0, uartflow, p)) == nil)
            return nil;
    }
    else
        qreopen(p->iq);
    if(p->oq == nil){
        if((p->oq = qopen(8*1024, 0, uartkick, p)) == nil){
            qfree(p->iq);
            p->iq = nil;
            return nil;
        }
    }
    else
        qreopen(p->oq);

    p->ir = p->istage;
    p->iw = p->istage;
    p->ie = &p->istage[STAGESIZE];
    p->op = p->ostage;
    p->oe = p->ostage;

    p->hup_dsr = p->hup_dcd = 0;
    p->dsr = p->dcd = 0;

    /* assume we can send */
    p->cts = 1;
    p->ctsbackoff = 0;

    if(p->bits == 0)
        uartctl(p, "l8");
    if(p->stop == 0)
        uartctl(p, "s1");
    if(p->parity == 0)
        uartctl(p, "pn");
    if(p->baud == 0)
        uartctl(p, "b9600");
    (*p->phys->enable)(p, 1);

    /*
     * use ilock because uartclock can otherwise interrupt here
     * and would hang on an attempt to lock uartalloc.
     */
    ilock(&uartalloc);
```

```
    for(l = &uartalloc.elist; *l; l = &(*l)->elist){
        if(*l == p)
            break;
    }
    if(*l == 0){
        p->elist = uartalloc.elist;
        uartalloc.elist = p;
    }
    p->enabled = 1;
    iunlock(&uartalloc);

    return p;
}

static void
uartdisable(Uart *p)
{
    Uart **l;

    if(!p->enabled)
        return;
    (*p->phys->disable)(p);

    ilock(&uartalloc);
    for(l = &uartalloc.elist; *l; l = &(*l)->elist){
        if(*l == p){
            *l = p->elist;
            break;
        }
    }
    p->enabled = 0;
    iunlock(&uartalloc);
}


static void
setlength(int i)
{
    Uart *p;

    if(i > 0){
        p = uart[i];
        if(p && p->opens && p->iq)
            uartdir[1+3*i].length = qlen(p->iq);
    } else for(i = 0; i < uartnuart; i++){
        p = uart[i];
        if(p && p->opens && p->iq)
            uartdir[1+3*i].length = qlen(p->iq);
    }
}

/*
 *  set up the '#t' directory
 */
static void
uartreset(void)
{
    int i;
    Dirtab *dp;
    Uart *p, *tail;
```

```
tail = nil;
for(i = 0; physuart[i] != nil; i++){
    if(physuart[i]->pnp == nil)
        continue;
    if((p = physuart[i]->pnp()) == nil)
        continue;
    if(uartlist != nil)
        tail->next = p;
    else
        uartlist = p;
    for(tail = p; tail->next != nil; tail = tail->next)
        uartnuart++;
    uartnuart++;
}

if(uartnuart) {
    uart = xalloc(uartnuart*sizeof(Uart*));
    if (uart == nil)
        panic("uartreset: no memory");
}

uartndir = 1 + 3*uartnuart;
uartdir = xalloc(uartndir * sizeof(Dirtab));
if (uartdir == nil)
    panic("uartreset: no memory");
dp = uartdir;
strcpy(dp->name, ".");
mkqid(&dp->qid, 0, 0, QTDIR);
dp->length = 0;
dp->perm = DMDIR|0555;
dp++;
p = uartlist;
for(i = 0; i < uartnuart; i++){
    /* 3 directory entries per port */
    snprint(dp->name, sizeof dp->name, "eia%d", i);
    dp->qid.path = NETQID(i, Ndataqid);
    dp->perm = 0660;
    dp++;
    snprint(dp->name, sizeof dp->name, "eia%dctl", i);
    dp->qid.path = NETQID(i, Nctlqid);
    dp->perm = 0660;
    dp++;
    snprint(dp->name, sizeof dp->name, "eia%dstatus", i);
    dp->qid.path = NETQID(i, Nstatqid);
    dp->perm = 0444;
    dp++;

    uart[i] = p;
    p->dev = i;
    if(p->console || p->special){
        if(uartenable(p) != nil){
            if(p->console){
                kbdq = p->iq;
                serialoq = p->oq;
                p->putc = kbdcr2nl;
            }
            p->opens++;
        }
    }
```

```
        p = p->next;
    }

    if(uartnuart){
        /*
         * at 115200 baud, the 1024 char buffer takes 56 ms to process,
         * processing it every 22 ms should be fine.
         */
        uarttimer = addclock0link(uartclock, 22);
    }
}


static Chan*
uartattach(char *spec)
{
    return devattach('t', spec);
}

static Walkqid*
uartwalk(Chan *c, Chan *nc, char **name, int nname)
{
    return devwalk(c, nc, name, nname, uartdir, uartndir, devgen);
}

static int
uartstat(Chan *c, uchar *dp, int n)
{
    if(NETTYPE(c->qid.path) == Ndataqid)
        setlength(NETID(c->qid.path));
    return devstat(c, dp, n, uartdir, uartndir, devgen);
}

static Chan*
uartopen(Chan *c, int omode)
{
    Uart *p;

    c = devopen(c, omode, uartdir, uartndir, devgen);

    switch(NETTYPE(c->qid.path)){
    case Nctlqid:
    case Ndataqid:
        p = uart[NETID(c->qid.path)];
        qlock(p);
        if(p->opens++ == 0 && uartenable(p) == nil){
            qunlock(p);
            c->flag &= ~COPEN;
            error(Enodev);
        }
        qunlock(p);
        break;
    }

    c->iounit = qiomaxatomic;
    return c;
}

static int
uartdrained(void* arg)
```

```
{
    Uart *p;

    p = arg;
    return qlen(p->oq) == 0 && p->op == p->oe;
}

static void
uartdrainoutput(Uart *p)
{
    if(!p->enabled)
        return;

    p->drain = 1;
    if(waserror()){
        p->drain = 0;
        nexterror();
    }
    sleep(&p->r, uartdrained, p);
    poperror();
}

static void
uartclose(Chan *c)
{
    Uart *p;

    if(c->qid.type & QTDIR)
        return;
    if((c->flag & COPEN) == 0)
        return;
    switch(NETTYPE(c->qid.path)){
    case Ndataqid:
    case Nctlqid:
        p = uart[NETID(c->qid.path)];
        qlock(p);
        if(--(p->opens) == 0){
            qclose(p->iq);
            ilock(&p->rlock);
            p->ir = p->iw = p->istage;
            iunlock(&p->rlock);

            /*
             */
            qhangup(p->oq, nil);
            if(!waserror()){
                uartdrainoutput(p);
                poperror();
            }
            qclose(p->oq);
            uartdisable(p);
            p->dcd = p->dsr = p->dohup = 0;
        }
        qunlock(p);
        break;
    }
}

static long
uartread(Chan *c, void *buf, long n, vlong off)
```

```
{
    Uart *p;
    ulong offset = off;

    if(c->qid.type & QTDIR){
        setlength(-1);
        return devdirread(c, buf, n, uartdir, uartndir, devgen);
    }

    p = uart[NETID(c->qid.path)];
    switch(NETTYPE(c->qid.path)){
    case Ndataqid:
        return qread(p->iq, buf, n);
    case Nctlqid:
        return readnum(offset, buf, n, NETID(c->qid.path), NUMSIZE);
    case Nstatqid:
        return (*p->phys->status)(p, buf, n, offset);
    }

    return 0;
}

int
uartctl(Uart *p, char *cmd)
{
    char *f[16];
    int i, n, nf;

    nf = tokenize(cmd, f, nelem(f));
    for(i = 0; i < nf; i++){
        if(strncmp(f[i], "break", 5) == 0){
            (*p->phys->dobreak)(p, 0);
            continue;
        }

        n = atoi(f[i]+1);
        switch(*f[i]){
        case 'B':
        case 'b':
            uartdrainoutput(p);
            if((*p->phys->baud)(p, n) < 0)
                return -1;
            break;
        case 'C':
        case 'c':
            p->hup_dcd = n;
            break;
        case 'D':
        case 'd':
            uartdrainoutput(p);
            (*p->phys->dtr)(p, n);
            break;
        case 'E':
        case 'e':
            p->hup_dsr = n;
            break;
        case 'f':
        case 'F':
            if(p->oq != nil)
                qflush(p->oq);
```

```
    break;
case 'H':
case 'h':
    if(p->iq != nil)
        qhangup(p->iq, 0);
    if(p->oq != nil)
        qhangup(p->oq, 0);
    break;
case 'i':
case 'I':
    uartdrainoutput(p);
    (*p->phys->fifo)(p, n);
    break;
case 'K':
case 'k':
    uartdrainoutput(p);
    (*p->phys->dobreak)(p, n);
    break;
case 'L':
case 'l':
    uartdrainoutput(p);
    if((*p->phys->bits)(p, n) < 0)
        return -1;
    break;
case 'm':
case 'M':
    uartdrainoutput(p);
    (*p->phys->modemctl)(p, n);
    break;
case 'n':
case 'N':
    if(p->oq != nil)
        qnoblock(p->oq, n);
    break;
case 'P':
case 'p':
    uartdrainoutput(p);
    if((*p->phys->parity)(p, *(f[i]+1)) < 0)
        return -1;
    break;
case 'Q':
case 'q':
    if(p->iq != nil)
        qsetlimit(p->iq, n);
    if(p->oq != nil)
        qsetlimit(p->oq, n);
    break;
case 'R':
case 'r':
    uartdrainoutput(p);
    (*p->phys->rts)(p, n);
    break;
case 'S':
case 's':
    uartdrainoutput(p);
    if((*p->phys->stop)(p, n) < 0)
        return -1;
    break;
case 'W':
case 'w':
```

```
            if(uarttimer == nil || n < 1)
                return -1;
            uarttimer->tns = (vlong)n * 100000LL;
            break;
        case 'X':
        case 'x':
            if(p->enabled){
                ilock(&p->tlock);
                p->xonoff = n;
                iunlock(&p->tlock);
            }
            break;
        }
    }
    return 0;
}

static long
uartwrite(Chan *c, void *buf, long n, vlong)
{
    Uart *p;
    char *cmd;

    if(c->qid.type & QTDIR)
        error(Eperm);

    p = uart[NETID(c->qid.path)];

    switch(NETTYPE(c->qid.path)){
    case Ndataqid:
        qlock(p);
        if(waserror()){
            qunlock(p);
            nexterror();
        }

        n = qwrite(p->oq, buf, n);

        qunlock(p);
        poperror();
        break;
    case Nctlqid:
        cmd = malloc(n+1);
        if(cmd == nil)
            error(Enomem);
        memmove(cmd, buf, n);
        cmd[n] = 0;
        qlock(p);
        if(waserror()){
            qunlock(p);
            free(cmd);
            nexterror();
        }

        /* let output drain */
        if(uartctl(p, cmd) < 0)
            error(Ebadarg);

        qunlock(p);
        poperror();
```

```
        free(cmd);
        break;
    }

    return n;
}

static int
uartwstat(Chan *c, uchar *dp, int n)
{
    DirEntry d;
    Dirtab *dt;

    if(!iseve())
        error(Eperm);
    if(QTDIR & c->qid.type)
        error(Eperm);
    if(NETTYPE(c->qid.path) == Nstatqid)
        error(Eperm);

    dt = &uartdir[1 + 3 * NETID(c->qid.path)];
    n = convM2D(dp, n, &d, nil);
    if(n == 0)
        error(Eshortstat);
    if(d.mode != ~0UL)
        dt[0].perm = dt[1].perm = d.mode;
    return n;
}

//void
//uartpower(int on)
//{
//    Uart *p;
//
//    for(p = uartlist; p != nil; p = p->next) {
//        if(p->phys->power)
//            (*p->phys->power)(p, on);
//    }
//}

Dev uartdevtab = {
    .dc       =    't',
    .name     =    "uart",

    .reset    =    uartreset,
    .init     =    devinit,
    .shutdown =    devshutdown,
    .attach   =    uartattach,
    .walk     =    uartwalk,
    .stat     =    uartstat,
    .open     =    uartopen,
    .create   =    devcreate,
    .close    =    uartclose,
    .read     =    uartread,
    .bread    =    devbread,
    .write    =    uartwrite,
    .bwrite   =    devbwrite,
    .remove   =    devremove,
    .wstat    =    uartwstat,
```

```
//    .power    =    uartpower,
};

/*
 *  restart input if it's off
 */
static void
uartflow(void *v)
{
    Uart *p;

    p = v;
    if(p->modem)
        (*p->phys->rts)(p, 1);
}

/*
 *  put some bytes into the local queue to avoid calling
 *  qconsume for every character
 */
int
uartstageoutput(Uart *p)
{
    int n;

    n = qconsume(p->oq, p->ostage, STAGESIZE);
    if(n <= 0)
        return 0;
    p->op = p->ostage;
    p->oe = p->ostage + n;
    return n;
}

/*
 *  restart output
 */
void
uartkick(void *v)
{
    Uart *p = v;

    if(p->blocked)
        return;

    ilock(&p->tlock);
    (*p->phys->kick)(p);
    iunlock(&p->tlock);

    if(p->drain && uartdrained(p)){
        p->drain = 0;
        wakeup(&p->r);
    }
}

/*
 * Move data from the interrupt staging area to
 * the input Queue.
 */
static void
uartstageinput(Uart *p)
```

```
{
    int n;
    uchar *ir, *iw;

    while(p->ir != p->iw){
        ir = p->ir;
        if(p->ir > p->iw){
            iw = p->ie;
            p->ir = p->istage;
        }
        else{
            iw = p->iw;
            p->ir = p->iw;
        }
        if((n = qproduce(p->iq, ir, iw - ir)) < 0){
            p->serr++;
            (*p->phys->rts)(p, 0);
        }
        else if(n == 0)
            p->berr++;
    }
}

/*
 *  receive a character at interrupt time
 */
void
uartrecv(Uart *p,  char ch)
{
    uchar *next;

    /* software flow control */
    if(p->xonoff){
        if(ch == CTLS){
            p->blocked = 1;
        }else if(ch == CTLQ){
            p->blocked = 0;
            p->ctsbackoff = 2; /* clock gets output going again */
        }
    }

    /* receive the character */
    if(p->putc)
        p->putc(p->iq, ch);
    else if (p->iw) {       /* maybe the line isn't enabled yet */
        ilock(&p->rlock);
        next = p->iw + 1;
        if(next == p->ie)
            next = p->istage;
        if(next == p->ir)
            uartstageinput(p);
        if(next != p->ir){
            *p->iw = ch;
            p->iw = next;
        }
        iunlock(&p->rlock);
    }
}

/*
```

```
 *  we save up input characters till clock time to reduce
 *  per character interrupt overhead.
 */
static void
uartclock(void)
{
    Uart *p;

    ilock(&uartalloc);
    for(p = uartalloc.elist; p; p = p->elist){

        /* this hopefully amortizes cost of qproduce to many chars */
        if(p->iw != p->ir){
            ilock(&p->rlock);
            uartstageinput(p);
            iunlock(&p->rlock);
        }

        /* hang up if requested */
        if(p->dohup){
            qhangup(p->iq, 0);
            qhangup(p->oq, 0);
            p->dohup = 0;
        }

        /* this adds hysteresis to hardware/software flow control */
        if(p->ctsbackoff){
            ilock(&p->tlock);
            if(p->ctsbackoff){
                if(--(p->ctsbackoff) == 0)
                    (*p->phys->kick)(p);
            }
            iunlock(&p->tlock);
        }
    }
    iunlock(&uartalloc);
}

/*
 * polling console input, output
 */
⟨global consuart 654c⟩

void (*lprint)(char *, int);

int
uartgetc(void)
{
    if(consuart == nil || consuart->phys->getc == nil)
        return -1;
    return consuart->phys->getc(consuart);
}

void
uartputs(char *s, int n)
{
    char *e;

    if(consuart == nil || consuart->phys->putc == nil) {
        if (lprint)
```

```
            (*lprint)(s, n);
        return;
    }
    e = s+n;
    for(; s<e; s++){
        if(*s == '\n')
            consuart->phys->putc(consuart, '\r');
        consuart->phys->putc(consuart, *s);
    }
}
```

Uses COPEN 294b, CTLQ-1852 676d, CTLS-1851 676d, DMDIR, Ebadarg 664d, Enodev 664d, Eshortstat 664d, NETID, NETQID, NETTYPE, NUMSIZE 867a, Nctlqid, Ndataqid, Nstatqid, QTDIR 70b, Uartalloc 676d, addclock0link() 253f, consuart 654c, devattach() 297b, devbread() 434a, devbwrite() 434b, devcreate() 810, devdirread() 432c, devgen() 428, devinit() 810, devremove() 810, devshutdown() 810, devstat() 431, devwalk() 429b, free() 104a, ilock() 79d, iseve() 591b, iunlock() 80b, kbdcr2nl() 676a, kbdq 388c, lprint 676d, malloc() 103c, nelem 853i, nexterror, poperror 666d, qclose() 418a, qconsume() 403, qflush() 420d, qfree(), qhangup() 418b, qiomaxatomic 813b, qlen() 419c, qlock() 81f, qnoblock() 420c, qopen() 409a, qproduce() 407, qread() 413, qreopen() 419b, qsetlimit() 420b, qunlock() 82c, qwrite() 401b, readnum() 670c, serialoq 653a, setlength() 676d, sleep, uart-1854 676d, uartalloc 676d, uartattach() 676d, uartclock() 676d, uartclose() 676d, uartctl() 676d, uartdir-1856 676d, uartdisable() 676d, uartdrained() 676d, uartdrainoutput() 676d, uartenable() 676d, uartflow() 676d, uartkick() 676d, uartlist-1853 676d, uartndir-1857 676d, uartnuart-1855 676d, uartopen() 676d, uartread() 676d, uartreset() 676d, uartstageinput() 676d, uartstat() 676d, uarttimer-1858 676d, uartwalk() 676d, uartwrite() 676d, uartwstat() 676d, wakeup, waserror 666d, and xalloc() 95d.

usb.h

⟨buses/usb.h 689⟩≡

```
/*
 * common USB definitions.
 */
#define dprint          if(debug)print
#define ddprint         if(debug>1)print
#define deprint         if(debug || ep->debug)print
#define ddeprint        if(debug>1 || ep->debug>1)print

#define GET2(p)         ((((p)[1]&0xFF)<<8)|((p)[0]&0xFF))
#define PUT2(p,v)       {((p)[0] = (v)); ((p)[1] = (v)>>8);}

typedef struct Udev Udev;       /* USB device */
typedef struct Ep Ep;           /* Endpoint */
typedef struct Hci Hci;         /* Host Controller Interface */
typedef struct Hciimpl Hciimpl; /* Link to the controller impl. */

enum
{
        /* fundamental constants */
        Ndeveps = 16,           /* max nb. of endpoints per device */

        /* tunable parameters */
        Nhcis   = 16,           /* max nb. of HCIs */
        Neps    = 128,          /* max nb. of endpoints */
        Maxctllen = 32*1024, /* max allowed sized for ctl. xfers; see Maxdevconf */
        Xfertmout = 2000,       /* default request time out (ms) */

        /* transfer types. keep this order */
        Tnone = 0,              /* no tranfer type configured */
        Tctl,                   /* wr req + rd/wr data + wr/rd sts */
        Tiso,                   /* stream rd or wr (real time) */
        Tbulk,                  /* stream rd or wr */
```

```
        Tintr,                  /* msg rd or wr */
        Nttypes,                /* number of transfer types */

        Epmax   = 0xF,          /* max ep. addr */
        Devmax  = 0x7F,         /* max dev. addr */

        /* Speeds */
        Fullspeed = 0,
        Lowspeed,
        Highspeed,
        Nospeed,

        /* request type */
        Rh2d = 0<<7,
        Rd2h = 1<<7,
        Rstd = 0<<5,
        Rclass =  1<<5,
        Rdev = 0,
        Rep = 2,
        Rother = 3,

        /* req offsets */
        Rtype   = 0,
        Rreq    = 1,
        Rvalue  = 2,
        Rindex  = 4,
        Rcount  = 6,
        Rsetuplen = 8,

        /* standard requests */
        Rgetstatus      = 0,
        Rclearfeature   = 1,
        Rsetfeature     = 3,
        Rsetaddr        = 5,
        Rgetdesc        = 6,

        /* device states */
        Dconfig  = 0,           /* configuration in progress */
        Denabled,               /* address assigned */
        Ddetach,                /* device is detached */
        Dreset,                 /* its port is being reset */

        /* (root) Hub reply to port status (reported to usbd) */
        HPpresent       = 0x1,
        HPenable        = 0x2,
        HPsuspend       = 0x4,
        HPovercurrent   = 0x8,
        HPreset         = 0x10,
        HPpower         = 0x100,
        HPslow          = 0x200,
        HPhigh          = 0x400,
        HPstatuschg     = 0x10000,
        HPchange        = 0x20000,
};

/*
 * Services provided by the driver.
 * epopen allocates hardware structures to prepare the endpoint
 * for I/O. This happens when the user opens the data file.
 * epclose releases them. This happens when the data file is closed.
```

```
 * epwrite tries to write the given bytes, waiting until all of them
 * have been written (or failed) before returning; but not for Iso.
 * epread does the same for reading.
 * It can be assumed that endpoints are DMEXCL but concurrent
 * read/writes may be issued and the controller must take care.
 * For control endpoints, device-to-host requests must be followed by
 * a read of the expected length if needed.
 * The port requests are called when usbd issues commands for root
 * hubs. Port status must return bits as a hub request would do.
 * Toggle handling and other details are left for the controller driver
 * to avoid mixing too much the controller and the comon device.
 * While an endpoint is closed, its toggles are saved in the Ep struct.
 */
struct Hciimpl
{
        void    *aux;                           /* for controller info */
        void    (*init)(Hci*);                  /* init. controller */
        void    (*dump)(Hci*);                  /* debug */
        void    (*interrupt)(Ureg*, void*);     /* service interrupt */
        void    (*epopen)(Ep*);                 /* prepare ep. for I/O */
        void    (*epclose)(Ep*);                /* terminate I/O on ep. */
        long    (*epread)(Ep*,void*,long);      /* transmit data for ep */
        long    (*epwrite)(Ep*,void*,long);     /* receive data for ep */
        char*   (*seprintep)(char*,char*,Ep*);  /* debug */
        int     (*portenable)(Hci*, int, int);  /* enable/disable port */
        int     (*portreset)(Hci*, int, int);   /* set/clear port reset */
        int     (*portstatus)(Hci*, int);       /* get port status */
        void    (*shutdown)(Hci*);              /* shutdown for reboot */
        void    (*debug)(Hci*, int);            /* set/clear debug flag */
};

struct Hci
{
        ISAConf;                                /* hardware info */
        int     tbdf;                           /* type+busno+devno+funcno */
        int     ctlrno;                         /* controller number */
        int     nports;                         /* number of ports in hub */
        int     highspeed;
        Hciimpl;                                        /* HCI driver  */
};

/*
 * USB endpoint.
 * All endpoints are kept in a global array. The first
 * block of fields is constant after endpoint creation.
 * The rest is configuration information given to all controllers.
 * The first endpoint for a device (known as ep0) represents the
 * device and is used to configure it and create other endpoints.
 * Its QLock also protects per-device data in dev.
 * See Hciimpl for clues regarding how this is used by controllers.
 */
struct Ep
{
        Ref;                    /* one per fid (and per dev ep for ep0s) */

        /* const once inited. */
        int     idx;            /* index in global eps array */
        int     nb;             /* endpoint number in device */
        Hci*    hp;             /* HCI it belongs to */
        Udev*   dev;            /* device for the endpoint */
```

```
        Ep*     ep0;            /* control endpoint for its device */

        QLock;                  /* protect fields below */
        char*   name;           /* for ep file names at #u/ */
        int     inuse;          /* endpoint is open */
        int     mode;           /* OREAD, OWRITE, or ORDWR */
        int     clrhalt;        /* true if halt was cleared on ep. */
        int     debug;          /* per endpoint debug flag */
        char*   info;           /* for humans to read */
        long    maxpkt;         /* maximum packet size */
        int     ttype;          /* tranfer type */
        ulong   load;           /* in µs, for a fransfer of maxpkt bytes */
        void*   aux;            /* for controller specific info */
        int     rhrepl;         /* fake root hub replies */
        int     toggle[2];      /* saved toggles (while ep is not in use) */
        long    pollival;               /* poll interval ([µ]frames; intr/iso) */
        long    hz;             /* poll frequency (iso) */
        long    samplesz;       /* sample size (iso) */
        int     ntds;           /* nb. of Tds per µframe */
        int     tmout;          /* 0 or timeout for transfers (ms) */
};

/*
 * Per-device configuration and cached list of endpoints.
 * eps[0]->QLock protects it.
 */
struct Udev
{
        int     nb;             /* USB device number */
        int     state;          /* state for the device */
        int     ishub;          /* hubs can allocate devices */
        int     isroot;         /* is a root hub */
        int     speed;          /* Full/Low/High/No -speed */
        int     hub;            /* dev number for the parent hub */
        int     port;           /* port number in the parent hub */
        Ep*     eps[Ndeveps];   /* end points for this device (cached) */
};

void    addhcitype(char *type, int (*reset)(Hci*));

extern char *usbmodename[];

extern char *seprintdata(char*,char*,uchar*,int);
```

E.3 concurrency/

concurrency/portdat_concurrency.h

⟨concurrency/portdat_concurrency.h 692⟩≡

```
//*****************************************************************************
// Mutual exclusion
//*****************************************************************************

// arch_tas() < Lock|ILock < QLock|RWLock

// This used to be in <arch>/, but its fields are used from port/, so the
```

```
// code must be portable!
⟨struct Lock 76c⟩

⟨struct ILock 79a⟩

⟨struct QLock 81a⟩

⟨struct RWlock 83b⟩

//*************************************************************************
// Atomicity
//*************************************************************************

⟨struct Ref 87a⟩

⟨struct Counter 87d⟩

//*************************************************************************
// Synchronization
//*************************************************************************

// defined in this directory but no functions are operating on it in this dir
⟨struct Rendez 233a⟩

⟨struct Sema 473⟩

// see also Waitq in portdat_processes.h?
```

concurrency/portfns_concurrency.h

⟨portfns_concurrency.h 693⟩≡

```
// taslock.c
int  lock(Lock*);
void unlock(Lock*);
void ilock(Lock*);
void iunlock(Lock*);
int  canlock(Lock*);

// ref.c
long incref(Ref*);
long decref(Ref*);

// qlock.c
void qlock(QLock*);
void qunlock(QLock*);
int  canqlock(QLock*);

void rlock(RWlock*);
void runlock(RWlock*);
int  canrlock(RWlock*);
void wlock(RWlock*);
void wunlock(RWlock*);

// in <arch>/concurrency.s (but used in port)
//@Scheck: Assembly
int  arch_splhi(void);
//@Scheck: Assembly
int  arch_spllo(void);
//@Scheck: Assembly
```

```
  void arch_splx(int);
  //@Scheck: Assembly
  bool arch_islo(void);

  //test-and-set
  //@Scheck: Assembly
  int  arch_tas(void*);
  //@Scheck: Assembly
  void arch_xinc(long*);
  //@Scheck: Assembly
  long arch_xdec(long*);
  // <arch>/??? (called from port but signature not portable across <arch>)
  //int   arch_cmpswap(long*, long, long);

  //void  arch_coherence(void); // now in core/ for backward deps
```

concurrency/taslock.c

⟨concurrency/taslock.c 694a⟩≡

```
  // TAS: Test And Set
```

⟨*kernel basic includes* 49a⟩

⟨*struct* TaslockStats 661a⟩
⟨*global* lockstats 661e⟩
⟨*globals lockcycles* 660g⟩

⟨*function* inccnt 88i⟩

⟨*function* deccnt 89a⟩

⟨*function* lockloop 646a⟩

⟨*function* lock 76d⟩

⟨*function* ilock 79d⟩

⟨*function* canlock 78f⟩

⟨*function* unlock 78d⟩

⟨*function* iunlock 80b⟩

concurrency/qlock.c

⟨concurrency/qlock.c 694b⟩≡

⟨*kernel basic includes* 49a⟩

⟨*struct* QlockStats 661f⟩
⟨*global* rwstats 661g⟩

⟨*function* qlock 81f⟩

⟨*function* canqlock 83a⟩

⟨*function* qunlock 82c⟩

⟨*function* rlock 83e⟩

⟨*function* runlock 84b⟩

⟨*function* wlock 85b⟩

⟨*function* wunlock 85d⟩

⟨*function* canrlock 86⟩

concurrency/ref.c

⟨concurrency/ref.c 695a⟩≡
⟨*kernel basic includes* 49a⟩

⟨*function* incref 87b⟩

⟨*function* decref 87c⟩

E.4 console/

console/portdat_console.h

⟨portdat_console.h 695b⟩≡
```
extern  Queue*  kbdq;
extern  Queue*  kprintoq;
extern  Queue*  serialoq;

// consdevtab
```

console/portfns_console.h

⟨portfns_console.h 695c⟩≡
```
// devcons.c
// (*print) is declared lib.h
// now in portfns_core.h, to remove some backward dependencies
//int  (*pprint)(char*, ...);
//void (*_assert)(char*);
//int  (*iprint)(char*, ...);
//void (*panic)(char*, ...);
//
void kbdqinit(void);
void lineqinit(void);
void kbdputc(Rune);
int  consactive(void);
int  kbdcr2nl(Queue*, int);
int  nrand(int);
void putstrn(char*, int);
// as a pointer in the original too
void  (*screenputs)(char*, int);
// overrides also sysfatal from libc/9sys/sysfatal.c, that are called
// from a few libc functions

// rdb.c
void    rdb(void);
```

```
// print.c
// overrides _fmtlock, from lib_core/libc/fmt/fmtlock.c that are used
// in fmt related functions
```

console/devcons.c

⟨*devcons.c forward decl* 696a⟩≡ (696b)

```
static void seedrand(void);
static int  readtime(ulong, char*, int);
static int  readbintime(char*, int);
static int  writetime(char*, int);
static int  writebintime(char*, int);
```

⟨console/devcons.c 696b⟩≡

⟨*kernel basic includes* 49a⟩

```
#include    <pool.h>

// used also by edf.c
```

⟨*global* panicking 663a⟩

⟨*hook screenputs* 384c⟩
⟨*hook consdebug* 652e⟩

⟨*global* kbdq 388c⟩
⟨*global* lineq 389a⟩
⟨*global* serialoq 653a⟩
⟨*global* kprintoq 651e⟩

⟨*global* kprintinuse 651f⟩
⟨*global* iprintscreenputs 392a⟩

⟨*struct* ConsKbd 382a⟩

⟨*global* kbd 382b⟩

⟨*global* fasthz 265a⟩

⟨*devcons.c forward decl* 696a⟩

⟨*function* kbdqinit 388d⟩

⟨*function* lineqinit 389b⟩

⟨*function* consactive 653b⟩

⟨*function* prflush 664a⟩

⟨*struct* KMesg 649b⟩

⟨*global* kmesg 650a⟩

⟨*function* kmesgputs 650e⟩

⟨*function* putstrn0 390b⟩

⟨*function* putstrn 391c⟩

⟨*function* `print` 391b⟩

⟨*global* `iprintlock` 392c⟩

⟨*function* `iprintcanlock` 392d⟩

⟨*function* `iprint` 392b⟩

⟨*function* `panic` 663b⟩

⟨*function* `sysfatal` 664b⟩

⟨*function* `_assert` 664c⟩

⟨*function* `pprint` 393a⟩

⟨*function* `echoscreen` 384d⟩

⟨*function* `echoserialoq` 653d⟩

⟨*function* `echo` 384a⟩

⟨*function* `kbdcr2nl` 676a⟩

⟨*function* `kbdputc` 382c⟩

⟨*clock callback kbdputcclock* 383a⟩

⟨*devcons.c enum Qxxx* 388a⟩

```
enum
{
    ⟨constant VLNUMSIZE 260c⟩
};
```

⟨*global* `consdir` 388b⟩

⟨*method consinit* 393b⟩

```
static Chan*
consattach(char *spec)
{
    return devattach('c', spec);
}

static Walkqid*
conswalk(Chan *c, Chan *nc, char **name, int nname)
{
    return devwalk(c, nc, name,nname, consdir, nelem(consdir), devgen);
}

static int
consstat(Chan *c, uchar *dp, int n)
{
    return devstat(c, dp, n, consdir, nelem(consdir), devgen);
}
```

⟨*method consopen* 394a⟩

⟨*method consclose* 394b⟩

⟨*method consread* 394c⟩

⟨*method conswrite* 395a⟩

⟨*global* `consdevtab` 395c⟩

⟨*global* `randn` 634b⟩

⟨*function* `seedrand` 634c⟩

⟨*function* `nrand` 634d⟩

⟨*global* `uvorder` 267b⟩

⟨*function* `le2vlong` 267c⟩

⟨*function* `vlong2le` 268a⟩

⟨*global* `order` 268b⟩

⟨*function* `le2long` 268c⟩

⟨*devcons.c Exxx errors* 260b⟩

⟨*function* `readtime` 265c⟩

⟨*function* `writetime` 265d⟩

⟨*function* `readbintime` 266c⟩

⟨*function* `writebintime` 267a⟩

Uses `consdir-1797` 388b, `devattach()` 297b, `devgen()` 428, `devstat()` 431, `devwalk()` 429b, and `nelem` 853i.

console/rdb.c

⟨*rdb.c debugging macro* 698a⟩≡ (698b)

```
#define DBG if(0)scrprint
#pragma varargck argpos scrprint 1
//static Ureg ureg;

static void
scrprint(char *fmt, ...)
{
    char buf[128];
    va_list va;
    int n;

    va_start(va, fmt);
    n = vseprint(buf, buf+sizeof buf, fmt, va)-buf;
    va_end(va);
    putstrn(buf, n);
}
```

Uses `buf` and `putstrn()` 391c.

⟨`console/rdb.c` 698b⟩≡

```
⟨kernel basic includes 49a⟩
#include "io.h"
```

```
#include <ureg.h>

⟨rdb.c debugging macro 698a⟩

static char*
getline(void)
{
    static char buf[128];
    int i, c;

    for(;;){
        for(i=0; i<nelem(buf) && (c=uartgetc()) != '\n'; i++){
            DBG("%c...", c);
            buf[i] = c;
        }

        if(i < nelem(buf)){
            buf[i] = 0;
            return buf;
        }
    }
}

static void*
addr(char *s, Ureg *ureg, char **p)
{
    ulong a;

    a = strtoul(s, p, 16);
    if(a < sizeof(Ureg))
        return ((uchar*)ureg)+a;
    return (void*)a;
}

static void
talkrdb(Ureg *ureg)
{
    uchar *a;
    char *p, *req;

    serialoq = nil;     /* turn off serial console */
    kprintoq = nil;     /* turn off /dev/kprint if active */
    DBG("Plan 9 debugger\n");
    iprint("Edebugger reset\n");
    for(;;){
        req = getline();
        switch(*req){
        case 'r':
            a = addr(req+1, ureg, nil);
            DBG("read %p\n", a);
            iprint("R%.8lux %.2ux %.2ux %.2ux %.2ux\n",
                strtoul(req+1, 0, 16), a[0], a[1], a[2], a[3]);
            break;

        case 'w':
            a = addr(req+1, ureg, &p);
            *(ulong*)a = strtoul(p, nil, 16);
            iprint("W\n");
            break;
/*
```

```
 *      case Tmput:
            n = min[4];
            if(n > 4){
                mesg(Rerr, Ecount);
                break;
            }
            a = addr(min+0);
            scrprint("mput %.8lux\n", a);
            memmove(a, min+5, n);
            mesg(Rmput, mout);
            break;
 *
 */
        default:
            DBG("unknown %c\n", *req);
            iprint("Eunknown message\n");
            break;
        }
    }
}

void
rdb(void)
{
    arch_splhi();
    iprint("rdb...");
    arch_callwithureg(talkrdb);
}
```

Uses DBG-1801 698a, addr() 698b, buf, c, iprint, kprintoq 651e, nelem 853i, p, req, serialoq 653a, talkrdb() 698b, and uartgetc() 676d.

console/print.c

⟨console/print.c 700⟩≡

```
⟨kernel basic includes 49a⟩
// overrides functions defined in lib_core/libc/fmt/fmtlock.c
// to behave in a special way when libc functions related to fmt
// are called in the kernel.

static Lock fmtl;

//@Scheck: no dead, called from libc
void _fmtlock(void)
{
    lock(&fmtl);
}

//@Scheck: no dead, called from libc
void _fmtunlock(void)
{
    unlock(&fmtl);
}

//@Scheck: no dead, called from libc
int _efgfmt(Fmt*)
{
    return -1;
}
```

Uses fmtl-1802 700, lock() 76d, and unlock() 78d.

E.5 core/

core/portdat_forward.h

⟨portdat_forward.h *pragmas* 701a⟩≡ (701b)

```
#pragma incomplete Edf
#pragma incomplete Mntcache
#pragma incomplete Mntrpc
#pragma incomplete Queue
#pragma incomplete Timers
```

⟨core/portdat_forward.h 701b⟩≡

```
typedef struct Alarms Alarms;
typedef struct Block  Block;
typedef struct Chan Chan;
typedef struct Cmdbuf Cmdbuf;
typedef struct Cmdtab Cmdtab;
typedef struct Dev  Dev;
typedef struct Dirtab Dirtab;
typedef struct Edf  Edf;
typedef struct Egrp Egrp;
typedef struct Evalue Evalue;
typedef struct Execvals Execvals;
typedef struct Fgrp Fgrp;
typedef struct KImage KImage;
typedef struct Log  Log;
typedef struct Logflag  Logflag;
typedef struct Mntcache Mntcache;
typedef struct Mount  Mount;
typedef struct Mntrpc Mntrpc;
typedef struct Mntwalk  Mntwalk;
typedef struct Mnt  Mnt;
typedef struct Mhead  Mhead;
typedef struct Note Note;
typedef struct Page Page;
typedef struct Path Path;
typedef struct Palloc Palloc;
typedef struct Pallocmem  Pallocmem;
typedef struct Perf Perf;
typedef struct PhysUart PhysUart;
typedef struct Pgrp Pgrp;
typedef struct Physseg  Physseg;
typedef struct Proc Proc;
typedef struct Pagetable  Pagetable;
typedef struct KQLock QLock;
typedef struct Queue  Queue;
typedef struct Ref  Ref;
typedef struct Rendez Rendez;
typedef struct Rgrp Rgrp;
typedef struct RWlock RWlock;
typedef struct Sargs  Sargs;
typedef struct Schedq Schedq;
typedef struct Segment  Segment;
typedef struct Sema Sema;
typedef struct Timer  Timer;
typedef struct Timers Timers;
typedef struct Uart Uart;
typedef struct Waitq  Waitq;
typedef struct Walkqid  Walkqid;
typedef struct Watchdog Watchdog;
```

```
typedef int    Devgen(Chan*, char*, Dirtab*, int, int, DirEntry*);

// was in dat_forward.h
typedef struct Conf Conf;
typedef struct Confmem  Confmem;
typedef struct Lock Lock;
typedef struct Cpu Cpu;
typedef struct Label  Label;
typedef struct DevConf  DevConf;
typedef struct Devport  Devport;
typedef struct ISAConf  ISAConf;

// was in cache.c
typedef struct Extent Extent;
// was in xallo.c
typedef struct Hole Hole;
typedef struct Xalloc Xalloc;
typedef struct Xhdr Xhdr;

// was in sd.h
typedef struct SDev SDev;
typedef struct SDifc SDifc;
typedef struct SDio SDio;
typedef struct SDpart SDpart;
typedef struct SDperm SDperm;
typedef struct SDreq SDreq;
typedef struct SDunit SDunit;

// was in kbd.c
typedef struct Kbscan Kbscan;
```

⟨portdat_forward.h *pragmas* 701a⟩

Uses Alarms 701b, Block 701b, Chan 701b, Cmdbuf 701b, Cmdtab 701b, Conf 62a, Confmem 62c, Cpu 64a, Dev 701b, DevConf 701b, Devport 701b, Dirtab 701b, Edf 701b, Egrp 701b, Evalue 701b, Execvals 701b, Extent 701b, Fgrp 701b, Hole 701b, ISAConf 701b, KImage 701b, KQLock 701b, Kbscan 701b, Label 226d, Lock 701b, Mhead 701b, Mnt 701b, Mntcache 701b, Mntrpc 701b, Mntwalk 701b, Mount 701b, Note 701b, Page 701b, Pagetable 701b, Pallocmem 701b, Path 701b, Perf 656b, Pgrp 701b, PhysUart 701b, Physseg 701b, Proc 701b, Queue 701b, RWlock 701b, Ref 701b, Rendez 701b, Rgrp 701b, SDev 701b, SDifc 701b, SDio 701b, SDpart 701b, SDperm 701b, SDreq 701b, SDunit 701b, Sargs 701b, Schedq 701b, Segment 701b, Sema 701b, Timer 701b, Timers 701b, Uart 701b, Waitq 701b, Walkqid 701b, Watchdog 701b, Xalloc 701b, and Xhdr 701b.

core/portdat_core.h

⟨core/portdat_core.h 702⟩≡

```
// All those structs used to be in <arch>/dat.h, but many of their fields
// were used from port/ so I've moved them here (and put the arch-specific
// fields in dat_core.h)

//*****************************************************************************
// Conf
//*****************************************************************************
```

⟨*struct* Confmem 62c⟩

⟨*struct* Conf 62a⟩

```
extern Conf conf;
```

```
⟨constant MAXCONF 63b⟩
// hash<string, string>
extern char *confname[];
extern char *confval[];
// Hashtbl.length(confname)
extern int nconf;

extern bool cpuserver; // defined in $CONF.c

char* getconf(char *name);

//*********************************************************************************
// Cpu
//*********************************************************************************

⟨struct Label 226d⟩

⟨struct Perf 656b⟩

⟨struct Cpu 64a⟩


// array<ref<Cpu>>, MAXCPUS is defined in <arch>/mem.h
extern Cpu* cpus[MAXCPUS];
⟨macro CPUS 65c⟩

// 'up' and 'cpu' can have different implement in different arch
// so they are not defined here.

//*********************************************************************************
// Other
//*********************************************************************************

extern char* eve;
int iseve(void);
// accessed by /dev/hostdomain, defined in auth.c
extern  char  hostdomain[];

// defined in ???
extern ulong    kerndate;
```

core/portfns_core.h

⟨portfns_core.h *pragmas* 703a⟩≡ (703b)

```
#pragma varargck argpos iprint  1
#pragma varargck argpos pprint  1
#pragma varargck argpos panic 1
```

⟨core/portfns_core.h 703b⟩≡

```
// exported in lib.h, defined in portfns.c
// (*print)

// exported in lib.h, defined in 386/fns.c (but used in port)
// int  cistrcmp(char*, char*);
// int  cistrncmp(char*, char*, int);

// could be in lib.h
```

⟨*portdat.h macros* 47b⟩

```
// portfns.c (mostly here just to remove some backward dependencies)
```
⟨portfns_core.h *backward deps breaker* **??**⟩

```
// portfns.c
bool returnfalse(void*);
int   readnum(ulong, char*, ulong, ulong, int);
int   readstr(ulong, char*, ulong, char*);
```

⟨portfns_core.h *pragmas* 703a⟩

core/portdat.h

⟨*portdat.h pragmas(x86)* 704a⟩≡
```
#pragma varargck  type  "I" uchar*
#pragma varargck  type  "V" uchar*
#pragma varargck  type  "E" uchar*
#pragma varargck  type  "M" uchar*
```

core/error.h

⟨error.h 704b⟩≡
```
extern char Emount[];   /* inconsistent mount */
extern char Eunmount[];   /* not mounted */
extern char Eismtpt[];    /* is a mount point */
extern char Eunion[];   /* not in union */
extern char Emountrpc[];  /* mount rpc error */
extern char Eshutdown[];  /* device shut down */
extern char Enocreate[];  /* mounted directory forbids creation */
extern char Enonexist[];  /* file does not exist */
extern char Eexist[];   /* file already exists */
extern char Ebadsharp[];  /* unknown device in # filename */
extern char Enotdir[];    /* not a directory */
extern char Eisdir[];   /* file is a directory */
extern char Ebadchar[];   /* bad character in file name */
extern char Efilename[];  /* file name syntax */
extern char Eperm[];    /* permission denied */
extern char Ebadusefd[];  /* inappropriate use of fd */
extern char Ebadarg[];    /* bad arg in system call */
extern char Einuse[];   /* device or object already in use */
extern char Eio[];    /* i/o error */
extern char Etoobig[];    /* read or write too large */
extern char Etoosmall[];  /* read or write too small */
extern char Ehungup[];    /* i/o on hungup channel */
extern char Ebadctl[];    /* bad process or channel control request */
extern char Enodev[];   /* no free devices */
extern char Eprocdied[];  /* process exited */
extern char Enochild[];   /* no living children */
extern char Eioload[];    /* i/o error in demand load */
extern char Enovmem[];    /* virtual memory allocation failed */
extern char Ebadfd[];   /* fd out of range or not open */
extern char Enofd[];    /* no free file descriptors */
extern char Eisstream[];  /* seek on a stream */
extern char Ebadexec[];   /* exec header invalid */
extern char Etimedout[];  /* connection timed out */
extern char Econrefused[];  /* connection refused */
extern char Econinuse[];  /* connection in use */
```

```
extern char Eintr[];    /* interrupted */
extern char Enomem[];   /* kernel allocate failed */
extern char Esoverlap[];  /* segments overlap */
extern char Eshort[];   /* i/o count too small */
extern char Egreg[];    /* jmk added reentrancy for threads */
extern char Ebadspec[];   /* bad attach specifier */
extern char Enoreg[];   /* process has no saved registers */
extern char Enoattach[];  /* mount/attach disallowed */
extern char Eshortstat[]; /* stat buffer too small */
extern char Ebadstat[];   /* malformed stat buffer */
extern char Enegoff[];    /* negative i/o offset */
extern char Ecmdargs[];   /* wrong #args in control message */
extern char Ebadip[];   /* bad ip address syntax */
extern char Edirseek[];   /* seek in directory */
extern char Echange[];    /* media or partition has changed */
extern char Edetach[];    /* device is detached */
extern char Enotconf[];   /* endpoint not configured */
extern char Estalled[];   /* endpoint stalled */
extern char Esbadstat[];  /* invalid directory entry received from server */
extern char Enoversion[]; /* version not established for mount channel */
```

core/error.c

⟨core/error.c 705a⟩≡
  ⟨*global Exxx errors* 664d⟩

core/portdat.c

⟨core/portdat.c 705b⟩≡
  ⟨*kernel basic includes* 49a⟩

```
// was in init/main.c
```
⟨*global* conf 62b⟩
```
// bool cpuserver; // defined in $CONF.c
```

⟨*globals confname and confvar* 63a⟩
⟨*global* nconf 63c⟩

```
//TODO: to factorize code between pc and pi
//< <function [[getconf]]> >

// was in dat.h
```
⟨*global* cpus 65a⟩
```
// cpu is in init/main.c because it can differ in different archi

// was in security/auth.c
```
⟨*global* eve 591a⟩
```
// should be in portfns.c, but then backward deps
/*
 *  return true if current user is eve
 */
```
⟨*function* iseve 591b⟩

core/portfns.c

⟨core/portfns.c 705c⟩≡

⟨*kernel basic includes* 49a⟩

⟨*portfns.c backward deps breaker* ??⟩

⟨*function* `returnfalse` 104c⟩

```
// was in devcons.c, could be in lib/misc.c
```
⟨*function* `readnum` 670c⟩

⟨*function* `readstr` 670d⟩

E.6 devices/

devices/portdat_devices.h

⟨*struct* `Devport` 706a⟩≡ (706c)
```
struct Devport {
    ulong port;
    int size;
};
```

⟨*struct* `DevConf` 706b⟩≡ (706c)
```
struct DevConf
{
    ulong intnum;     /* interrupt number */
    char  *type;      /* card type, malloced */
    int nports;     /* Number of ports */
    Devport *ports;     /* The ports themselves */
};
```

⟨`devices/portdat_devices.h` 706c⟩≡
```
/*
 *  hardware info about a device
 */
```
⟨*struct* `Devport` 706a⟩

⟨*struct* `DevConf` 706b⟩

```
// keyboard
```

⟨*enum* `kbscan` 378b⟩

⟨*struct* `Kbscan` 378a⟩

```
// keyboard/portkbd.c
extern Kbscan kbscans[KbNscans];

// mouse

/*
 *  mouse types
 */
enum MouseType
{
    Mouseother= 0,
    Mouseserial=    1,
    MousePS2=   2,
};
```

```
// mouse/devmouse.c (used in portmouse.c, portkbd.c, ...)
extern int mouseshifted;

// mouse/portmouse.c (used in <arch>/mouse.c)
extern int mousetype;
extern int packetsize;
extern bool mousehwaccel;
extern bool intellimouse;

/* devdraw.c */ // used in some <arch>/screen.c
extern QLock drawlock;
// for gscreen, gscreendata, see portscreen.h
```

devices/portfns_devices.h

⟨portfns_devices.h 707⟩≡

```
// sys/devroot.c
extern void  addbootfile(char*, uchar*, ulong);

// sys/devmnt.c
extern Chan* mntauth(Chan*, char*);
extern long  mntversion(Chan*, char*, int, int);

// sys/devsrv.c
extern char* srvname(Chan*);

// sys/devproc.c
extern int   procfdprint(Chan*, int, int, char*, int);

// sys/devenv.c
extern void  envcpy(Egrp*, Egrp*);
extern void  ksetenv(char*, char*, int);
extern char* getconfenv(void);



// screen/devdraw.c
extern void  drawactive(bool);
extern void  drawcmap(void);

// mouse/devmouse.c (called from portmouse.c or <arch>/mouse.c)
extern void  mouseresize(void);
extern void  mouseaccelerate(int x);
extern void  mousetrack(int dx, int dy, int b, int msec);
// to call to let know devmouse.c about some mouse related keyboard events
extern void  (*kbdmouse)(int);

// sys/devrtc.c
//long    rtctime(void);


// mouse/portmouse.c (called by devmouse.c for arch-specific mouse settings)
extern void  kmousectl(Cmdbuf*);

// keyboard/latin1.c (used by portkbd.c)
//@Scheck: def currently skipped, see skip_list.txt, because of unicode
```

```
extern long  latin1(Rune*, int);

// keyboard/portkbd.c
extern void  kbdputsc(byte, int);
extern int   kbdgetmap(uint, int*, int*, Rune*);
extern void  kbdputmap(ushort, ushort, Rune);


// mouse/<arch>/mouse.c (called from portmouse.c)
extern void  arch_ps2mouse(void);
extern void  arch_setaccelerated(void);
extern void  arch_setlinear(void);
extern void  arch_setres(int n);
extern void  arch_setintellimouse(void);
extern void  arch_resetmouse(void);

// keyboard/<arch>/kbd.c (called from portkbd.c)
extern void  arch_setleds(Kbscan *kbscan);

// screen/<arch>/screen.c (called from main)
extern void  arch__screeninit(void);

// sys/<arch>/devarch.c (called from arch specific)
extern Dirtab* addarchfile(char*, int, long(*)(Chan*,void*,long,vlong), long(*)(Chan*,void*,long,vlong));
```

E.7 devices/keyboard/

devices/keyboard/portkbd.c

⟨devices/keyboard/portkbd.c 708⟩≡

```
/*
 * keyboard input // the portable part
 */
```

⟨*kernel basic includes* 49a⟩

```
// The main entry point in this file is kbdputsc() called
// from devkbin.c when a process (e.g., usbd) writes into /dev/kbin.

// The functions kbdputmap/kbdgetmap are also portable functions called
// from devkbmap.c
```

⟨*global* kdebug 641a⟩

⟨*enum* specialkey 376a⟩

⟨*global* kbtab 376b⟩

⟨*global* kbtabshift 377a⟩

⟨*global* kbtabesc1 380c⟩

⟨*global* kbtabaltgr 377c⟩

```
// see kbtab.c for the ctrl one, put in another file because
// of issues with TeX with the special characters it contain
// can't LPize it.
```

⟨*global kbtabctrl decl* 377b⟩

⟨*global* kbscans 378c⟩

```
// called when write in /dev/kbin (see devkbin.c)
```
⟨*function* kbdputsc 378d⟩

```
// called when interact with /dev/kbmap (see devkbmap.c)
```
⟨*function* kbdputmap 385b⟩

⟨*function* kbdgetmap 385a⟩

devices/keyboard/latin1.c

⟨devices/keyboard/latin1.c 709⟩≡
⟨*kernel basic includes* 49a⟩

```
/*
 * The code makes two assumptions: strlen(ld) is 1 or 2; latintab[i].ld can be a
 * prefix of latintab[j].ld only when j<i.
 */
struct cvlist
{
    char    *ld;        /* must be seen before using this conversion */
    char    *si;        /* options for last input characters */

    Rune    *so;        /* the corresponding Rune for each si entry */
};

struct cvlist latintab[] = {
#include "../port/latin1.h"
    0,  0,      0
};

/*
 * Given n characters k[0]..k[n-1], find the rune or return -1 for failure.
 */
long
unicode(Rune *k, int n)
{
    long c;
    Rune *r;

    c = 0;
    for(r = &k[1]; r<&k[n]; r++){       /* +1 to skip [Xx] */
        c <<= 4;
        if('0'<=*r && *r<='9')
            c += *r-'0';
        else if('a'<=*r && *r<='f')
            c += 10 + *r-'a';
        else if('A'<=*r && *r<='F')
            c += 10 + *r-'A';
        else
            return -1;
    }
    return c;
}

/*
```

```
 * Given n characters k[0]..k[n-1], find the corresponding rune or return -1 for
 * failure, or something < -1 if n is too small.  In the latter case, the result
 * is minus the required n.
 */
long
latin1(Rune* k, int n)
{
    struct cvlist *l;
    char c;
    char* p;

//    print("latin1: %d, %d\n", n, k[n-1]);

    if(k[0] == 'X')
        if(n>=5)
            return unicode(k, 5);
        else
            return -5;
    if(k[0] == 'x')
        if(n>=UTFmax*2+1)
            return unicode(k, UTFmax*2+1);
        else
            return -(UTFmax+1);

    for(l=latintab; l->ld!=0; l++)
        if(k[0] == l->ld[0]){
//          print("found: %s, %s\n", l->ld, l->si);
            if(n == 1)
                return -2;
            if(l->ld[1] == 0)
                c = k[1];
            else if(l->ld[1] != k[1])
                continue;
            else if(n == 2)
                return -3;
            else
                c = k[2];
            for(p=l->si; *p!=0; p++)
                if(*p == c) {
                    print("found: 0x%x\n", l->so[p - l->si]);
                    return l->so[p - l->si];
                }
            return -1;
        }
    return -1;
}
```

Uses UTFmax, cvlist 709, and print.

devices/keyboard/latin1.h

TO LP SPLIT devices/keyboard/devkbin.c

⟨devices/keyboard/devkbin.c 710⟩≡

```
/*
 *  keyboard scan code input from outside the kernel.
 *  to avoid duplication of keyboard map processing for usb.
 */
```

⟨*kernel basic includes* 49a⟩

```
extern  void kbdputsc(byte, int);

enum {
    Qdir,
    Qkbd,
};

Dirtab kbintab[] = {
    ".",    {Qdir, 0, QTDIR},   0,  0555,
    "kbin", {Qkbd, 0},      0,  0200,
};

Lock    kbinlck;
int kbinbusy;

static Chan *
kbinattach(char *spec)
{
    return devattach(L'I', spec);
}

static Walkqid*
kbinwalk(Chan *c, Chan *nc, char **name, int nname)
{
    return devwalk(c, nc, name, nname, kbintab, nelem(kbintab), devgen);
}

static int
kbinstat(Chan *c, uchar *dp, int n)
{
    return devstat(c, dp, n, kbintab, nelem(kbintab), devgen);
}

static Chan*
kbinopen(Chan *c, int omode)
{
    if(!iseve())
        error(Eperm);
    if(c->qid.path == Qkbd){
        lock(&kbinlck);
        if(kbinbusy){
            unlock(&kbinlck);
            error(Einuse);
        }
        kbinbusy++;
        unlock(&kbinlck);
    }
    return devopen(c, omode, kbintab, nelem(kbintab), devgen);
}

static void
kbinclose(Chan *c)
{
    if(c->aux){
        free(c->aux);
        c->aux = nil;
    }
    if(c->qid.path == Qkbd)
        kbinbusy = 0;
```

```
}

static long
kbinread(Chan *c, void *a, long n, vlong )
{
    if(c->qid.type == QTDIR)
        return devdirread(c, a, n, kbintab, nelem(kbintab), devgen);
    return 0;
}

static long
kbinwrite(Chan *c, void *a, long n, vlong)
{
    int i;
    uchar *p = a;

    if(c->qid.type == QTDIR)
        error(Eisdir);
    switch((int)c->qid.path){
    case Qkbd:
        for(i = 0; i < n; i++)
            kbdputsc(*p++, 1);  /* external source */
        break;
    default:
        error(Egreg);
    }
    return n;
}

Dev kbindevtab = {
    .dc = L'I',
    .name = "kbin",

    .reset = devreset,
    .init = devinit,
    .shutdown = devshutdown,
    .attach = kbinattach,
    .walk = kbinwalk,
    .stat = kbinstat,
    .open = kbinopen,
    .create = devcreate,
    .close = kbinclose,
    .read = kbinread,
    .bread = devbread,
    .write = kbinwrite,
    .bwrite = devbwrite,
    .remove = devremove,
    .wstat = devwstat,
};
```

Uses Egreg 664d, QTDIR 70b, Qdir-992 710, Qkbd-993 710, devattach() 297b, devbread() 434a, devbwrite() 434b, devcreate() 810, devdirread() 432c, devgen() 428, devinit() 810, devremove() 810, devreset() 810, devshutdown() 810, devstat() 431, devwalk() 429b, devwstat() 810, free() 104a, iseve() 591b, kbdputsc() 378d, kbinattach() 710, kbinbusy 710, kbinclose() 710, kbinlck 710, kbinopen() 710, kbinread() 710, kbinstat() 710, kbintab 710, kbinwalk() 710, kbinwrite() 710, lock() 76d, nelem 853i, and unlock() 78d.

TO LPSPLIT devices/keyboard/devkbmap.c

⟨devices/keyboard/devkbmap.c 712⟩≡

```
/*
```

```
 *  keyboard map
 */
```
⟨*kernel basic includes* 49a⟩

```
enum{
    Qdir,
    Qdata,
};
Dirtab kbmaptab[]={
    ".",        {Qdir, 0, QTDIR},   0,  0555,
    "kbmap",    {Qdata, 0},     0,  0600,
};
#define NKBFILE sizeof(kbmaptab)/sizeof(kbmaptab[0])

#define KBLINELEN   (3*NUMSIZE+1)   /* t code val\n */

static Chan *
kbmapattach(char *spec)
{
    return devattach(L'K', spec);
}

static Walkqid*
kbmapwalk(Chan *c, Chan *nc, char **name, int nname)
{
    return devwalk(c, nc, name, nname, kbmaptab, NKBFILE, devgen);
}

static int
kbmapstat(Chan *c, uchar *dp, int n)
{
    return devstat(c, dp, n, kbmaptab, NKBFILE, devgen);
}

static Chan*
kbmapopen(Chan *c, int omode)
{
    if(!iseve())
        error(Eperm);
    return devopen(c, omode, kbmaptab, NKBFILE, devgen);
}

static void
kbmapclose(Chan *c)
{
    if(c->aux){
        free(c->aux);
        c->aux = nil;
    }
}

static long
kbmapread(Chan *c, void *a, long n, vlong offset)
{
    char *bp;
    char tmp[KBLINELEN+1];
    int t, sc;
    Rune r;

    if(c->qid.type == QTDIR)
```

```
        return devdirread(c, a, n, kbmaptab, NKBFILE, devgen);

    switch((int)(c->qid.path)){
    case Qdata:
        if(kbdgetmap(offset/KBLINELEN, &t, &sc, &r)) {
            bp = tmp;
            bp += readnum(0, bp, NUMSIZE, t, NUMSIZE);
            bp += readnum(0, bp, NUMSIZE, sc, NUMSIZE);
            bp += readnum(0, bp, NUMSIZE, r, NUMSIZE);
            *bp++ = '\n';
            *bp = 0;
            n = readstr(offset%KBLINELEN, a, n, tmp);
        } else
            n = 0;
        break;
    default:
        n=0;
        break;
    }
    return n;
}

static long
kbmapwrite(Chan *c, void *a, long n, vlong)
{
    char line[100], *lp, *b;
    int key, m, l;
    Rune r;

    if(c->qid.type == QTDIR)
        error(Eperm);

    switch((int)(c->qid.path)){
    case Qdata:
        b = a;
        l = n;
        lp = line;
        if(c->aux){
            strcpy(line, c->aux);
            lp = line+strlen(line);
            free(c->aux);
            c->aux = nil;
        }
        while(--l >= 0) {
            *lp++  = *b++;
            if(lp[-1] == '\n' || lp == &line[sizeof(line)-1]) {
                *lp = 0;
                if(*line == 0)
                    error(Ebadarg);
                if(*line == '\n' || *line == '#'){
                    lp = line;
                    continue;
                }
                lp = line;
                while(*lp == ' ' || *lp == '\t')
                    lp++;
                m = strtoul(line, &lp, 0);
                key = strtoul(lp, &lp, 0);
                while(*lp == ' ' || *lp == '\t')
                    lp++;
```

```
                r = 0;
                if(*lp == '\'' && lp[1])
                    chartorune(&r, lp+1);
                else if(*lp == '^' && lp[1]){
                    chartorune(&r, lp+1);
                    if(0x40 <= r && r < 0x60)
                        r -= 0x40;
                    else
                        error(Ebadarg);
                }else if(*lp == 'M' && ('1' <= lp[1] && lp[1] <= '5'))
                    r = 0xF900+lp[1]-'0';
                else if(*lp>='0' && *lp<='9') /* includes 0x... */
                    r = strtoul(lp, &lp, 0);
                else
                    error(Ebadarg);
                kbdputmap(m, key, r);
                lp = line;
            }
        }
        if(lp != line){
            l = lp-line;
            c->aux = lp = smalloc(l+1);
            memmove(lp, line, l);
            lp[l] = 0;
        }
        break;
    default:
        error(Ebadusefd);
    }
    return n;
}


Dev kbmapdevtab = {
    .dc = L'k',
    .name = "kbmap",

    .reset = devreset,
    .init = devinit,
    .shutdown = devshutdown,
    .attach = kbmapattach,
    .walk = kbmapwalk,
    .stat = kbmapstat,
    .open = kbmapopen,
    .create = devcreate,
    .close = kbmapclose,
    .read = kbmapread,
    .bread = devbread,
    .write = kbmapwrite,
    .bwrite = devbwrite,
    .remove = devremove,
    .wstat = devwstat,
};
```

Uses Ebadarg 664d, Ebadusefd 664d, KBLINELEN-997 712, NKBFILE-996 712, NUMSIZE 867a, QTDIR 70b, Qdata-995 712, Qdir-994 712, devattach() 297b, devbread() 434a, devbwrite() 434b, devcreate() 810, devdirread() 432c, devgen() 428, devinit() 810, devremove() 810, devreset() 810, devshutdown() 810, devstat() 431, devwalk() 429b, devwstat() 810, free() 104a, iseve() 591b, kbdgetmap() 385a, kbdputmap() 385b, kbmapattach() 712, kbmapclose() 712, kbmapopen() 712, kbmapread() 712, kbmapstat() 712, kbmaptab 712, kbmapwalk() 712, kbmapwrite() 712, readnum() 670c, readstr() 670d, and smalloc() 104b.

E.8 devices/mouse/

TO LPSPLIT devices/mouse/portmouse.c

⟨devices/mouse/portmouse.c 716⟩≡
⟨*kernel basic includes* 49a⟩

```
// enum<MouseType>
int mousetype;

int packetsize;
static int resolution;
static int accelerated;

bool mousehwaccel;
bool intellimouse;

static QLock mousectlqlock;

// for serial mouse
//static char mouseport[5];

enum
{
    CMaccelerated,
    CMhwaccel,
    CMintellimouse,
    CMlinear,
    CMps2,
    CMps2intellimouse,
    CMres,
    CMreset,
    CMserial,
};

static Cmdtab mousectlmsg[] =
{
    CMaccelerated     , "accelerated"     , 0 ,
    CMhwaccel         , "hwaccel"         , 2 ,
    CMintellimouse    , "intellimouse"    , 1 ,
    CMlinear          , "linear"          , 1 ,
    CMps2             , "ps2"             , 1 ,
    CMps2intellimouse , "ps2intellimouse" , 1 ,
    CMres             , "res"             , 0 ,
    CMreset           , "reset"           , 1 ,
    CMserial          , "serial"          , 0 ,
};


/*
 *  set up a ps2 mouse
 */
static void
ps2mouse(void)
{
    if(mousetype == MousePS2)
        return;
    arch_ps2mouse();

    mousetype = MousePS2;
```

```
    packetsize = 3;
    mousehwaccel = 1;
}

/*
 * The PS/2 Trackpoint multiplexor on the IBM Thinkpad T23 ignores
 * acceleration commands.  It is supposed to pass them on
 * to the attached device, but my Logitech mouse is simply
 * not behaving any differently.  For such devices, we allow
 * the user to use "hwaccel off" to tell us to back off to
 * software acceleration even if we're using the PS/2 port.
 * (Serial mice are always software accelerated.)
 * For more information on the Thinkpad multiplexor, see
 * http://wwwcssrv.almaden.ibm.com/trackpoint/
 */
static void
setaccelerated(int x)
{
    accelerated = x;
    arch_setaccelerated();
    // call in devmouse.c
    mouseaccelerate(x);
}

static void
setlinear(void)
{
    accelerated = 0;
    arch_setlinear();
    mouseaccelerate(0);
}

static void
setres(int n)
{
    resolution = n;
    arch_setres(n);
}

static void
setintellimouse(void)
{
    intellimouse = 1;
    packetsize = 4;
    arch_setintellimouse();
}

static void
resetmouse(void)
{
    packetsize = 3;
    arch_resetmouse();
}

void
kmousectl(Cmdbuf *cb)
{
    Cmdtab *ct;

    qlock(&mousectlqlock);
```

```
if(waserror()){
    qunlock(&mousectlqlock);
    nexterror();
}

ct = lookupcmd(cb, mousectlmsg, nelem(mousectlmsg));
switch(ct->index){
case CMaccelerated:
    setaccelerated(cb->nf == 1 ? 1: atoi(cb->f[1]));
    break;
case CMlinear:
    setlinear();
    break;
case CMps2:
    intellimouse = 0;
    ps2mouse();
    break;
case CMres:
    if(cb->nf >= 2)
        setres(atoi(cb->f[1]));
    else
        setres(1);
    break;
case CMreset:
    resetmouse();
    if(accelerated)
        setaccelerated(accelerated);
    if(resolution)
        setres(resolution);
    if(intellimouse)
        setintellimouse();
    break;
case CMhwaccel:
    if(strcmp(cb->f[1], "on")==0)
        mousehwaccel = 1;
    else if(strcmp(cb->f[1], "off")==0)
        mousehwaccel = 0;
    else
        cmderror(cb, "bad mouse control message");
    break;
case CMintellimouse:
    setintellimouse();
    break;
case CMps2intellimouse:
    ps2mouse();
    setintellimouse();
    break;
case CMserial:
    //if(mousetype == Mouseserial)
    //  error(Emouseset);
    //
    //if(cb->nf > 2){
    //  if(strcmp(cb->f[2], "M") == 0)
    //      i8250mouse(cb->f[1], m3mouseputc, 0);
    //  else if(strcmp(cb->f[2], "MI") == 0)
    //      i8250mouse(cb->f[1], m5mouseputc, 0);
    //  else
    //      i8250mouse(cb->f[1], mouseputc, cb->nf == 1);
    //} else
    //  i8250mouse(cb->f[1], mouseputc, cb->nf == 1);
```

```
        //
        //mousetype = Mouseserial;
        //strncpy(mouseport, cb->f[1], sizeof(mouseport)-1);
        //packetsize = 3;
        error("serial mice not supported anymore");
        break;
    }
    qunlock(&mousectlqlock);
    poperror();
}
```

Uses CMaccelerated-930 716, CMhwaccel-931 716, CMintellimouse-932 716, CMlinear-933 716, CMps2-934 716, CMps2intellimouse-935 716, CMres-936 716, CMreset-937 716, CMserial-938 716, MousePS2 706c, accelerated-928 716, cmderror() 595a, intellimouse 716, lookupcmd() 595b, mouseaccelerate() 719, mousectlmsg-939 716, mousectlqlock-929 716, mousehwaccel 716, mousetype 716, nelem 853i, nexterror, packetsize 716, poperror 666d, ps2mouse() 716, qlock() 81f, qunlock() 82c, resetmouse() 716, resolution-927 716, setaccelerated() 716, setintellimouse() 716, setlinear() 716, setres() 716, and waserror 666d.

TO LPSPLIT devices/mouse/devmouse.c

⟨devices/mouse/devmouse.c 719⟩≡
⟨*kernel basic includes* 49a⟩

```
#include    <draw.h>
#include    <draw_private.h> // for BPLONG
#include    <memdraw.h>
#include    <cursor.h>

#include    "../port/portscreen.h"

enum {
    ScrollUp = 0x08,
    ScrollDown = 0x10,
    ScrollLeft = 0x20,
    ScrollRight = 0x40,
};

typedef struct Mouseinfo    Mouseinfo;
typedef struct Mousestate   Mousestate;

struct Mousestate
{
    Point   xy;     /* mouse.xy */
    int buttons;    /* mouse.buttons */
    ulong   counter;    /* increments every update */
    ulong   msec;       /* time of last event */
};

struct Mouseinfo
{
    Lock;
    Mousestate;
    int dx;
    int dy;
    bool track;      /* dx & dy updated */
    bool redraw;     /* update cursor on screen */
    ulong   lastcounter;    /* value when /dev/mouse read */
    ulong   lastresize;
    ulong   resize;
```

```
    Rendez  r;
    Ref;
    QLock;
    bool open;
    int acceleration;
    int maxacc;
    Mousestate  queue[16];  /* circular buffer of click events */
    int ri;     /* read index into queue */
    int wi;     /* write index into queue */
    uchar   qfull;      /* queue is full */
};

Mouseinfo   mouse;


Cursorinfo  cursor;
Cursor      curs;


// set to mousefromkbd here in mouseinit(), but used outside devmouse.c
// to let know devmouse.c about some keyboard events related to the mouse
void    (*kbdmouse)(int);
int     kbdbuttons;

// not used in this file
bool     mouseshifted;


enum{
    Qdir,
    Qcursor,
    Qmouse,
    Qmousein,
    Qmousectl,
};

static Dirtab mousedir[]={
    ".",    {Qdir, 0, QTDIR},   0,          DMDIR|0555,
    "cursor",   {Qcursor},  0,          0666,
    "mouse",    {Qmouse},   0,          0666,
    "mousein",  {Qmousein}, 0,          0220,
    "mousectl", {Qmousectl},    0,          0220,
};

enum
{
    CMbuttonmap,
    CMscrollswap,
    CMswap,
    CMwildcard,
};

static Cmdtab mousectlmsg[] =
{
    CMbuttonmap,    "buttonmap",    0,
    CMscrollswap,   "scrollswap",   0,
    CMswap,     "swap",     1,
    CMwildcard, "*",        0,
};
// for CMwildcard, devmouse.c relies on arch defined kmousectl()
```

```
static uchar buttonmap[8] = {
    0, 1, 2, 3, 4, 5, 6, 7,
};
static bool mouseswap;
static bool scrollswap;
static ulong mousetime;



void Cursortocursor(Cursor*);
int mousechanged(void*);
static void mouseclock(void);




static void
mousereset(void)
{
    curs = arch_arrow;
    Cursortocursor(&arch_arrow);
    /* redraw cursor about 30 times per second */
    addclock0link(mouseclock, 33);
}

static void
mousefromkbd(int buttons)
{
    kbdbuttons = buttons;
    mousetrack(0, 0, 0, TK2MS(CPUS(0)->ticks));
}

static int
mousedevgen(Chan *c, char *name, Dirtab *tab, int ntab, int i, DirEntry *dp)
{
    int rc;

    rc = devgen(c, name, tab, ntab, i, dp);
    if(rc != -1)
        dp->atime = mousetime;
    return rc;
}

static void
mouseinit(void)
{
    curs = arch_arrow;
    Cursortocursor(&arch_arrow);
    arch_cursoron(true);
    kbdmouse = mousefromkbd;
    mousetime = seconds();
}

static Chan*
mouseattach(char *spec)
{
    return devattach('m', spec);
```

```
}

static Walkqid*
mousewalk(Chan *c, Chan *nc, char **name, int nname)
{
    Walkqid *wq;

    /*
     * We use devgen() and not mousedevgen() here
     * see "Ugly problem" in dev.c/devwalk()
     */
    wq = devwalk(c, nc, name, nname, mousedir, nelem(mousedir), devgen);
    if(wq != nil && wq->clone != c && wq->clone != nil && (wq->clone->qid.type&QTDIR)==0)
        incref(&mouse);
    return wq;
}

static int
mousestat(Chan *c, uchar *db, int n)
{
    return devstat(c, db, n, mousedir, nelem(mousedir), mousedevgen);
}

static Chan*
mouseopen(Chan *c, int omode)
{
    switch((ulong)c->qid.path){
    case Qdir:
        if(omode != OREAD)
            error(Eperm);
        break;
    case Qmouse:
        lock(&mouse);
        if(mouse.open){
            unlock(&mouse);
            error(Einuse);
        }
        mouse.open = true;
        mouse.ref++;
        mouse.lastresize = mouse.resize;
        unlock(&mouse);
        break;
    case Qmousein:
        if(!iseve())
            error(Eperm);
        break;
    default:
        incref(&mouse);
    }
    c->mode = openmode(omode);
    c->flag |= COPEN;
    c->offset = 0;
    return c;
}

static void
mousecreate(Chan*, char*, int, ulong)
{
    error(Eperm);
}
```

```
static void
mouseclose(Chan *c)
{
    if((c->qid.type&QTDIR)==0 && (c->flag&COPEN)){
        if(c->qid.path == Qmousein)
            return;
        lock(&mouse);
        if(c->qid.path == Qmouse)
            mouse.open = false;
        if(--mouse.ref == 0){
            arch_cursoroff(true);
            curs = arch_arrow;
            Cursortocursor(&arch_arrow);
            arch_cursoron(true);
        }
        unlock(&mouse);
    }
}


static long
mouseread(Chan *c, void *va, long n, vlong off)
{
    char buf[1+4*12+1];
    uchar *p;
    static int map[8] = {0, 4, 2, 6, 1, 5, 3, 7 };
    ulong offset = off;
    Mousestate m;
    int b;

    p = va;
    switch((ulong)c->qid.path){
    case Qdir:
        return devdirread(c, va, n, mousedir, nelem(mousedir), mousedevgen);

    case Qcursor:
        if(offset != 0)
            return 0;
        if(n < 2*4+2*2*16)
            error(Eshort);
        n = 2*4+2*2*16;
        lock(&cursor);
        BPLONG(p+0, curs.offset.x);
        BPLONG(p+4, curs.offset.y);
        memmove(p+8, curs.clr, 2*16);
        memmove(p+40, curs.set, 2*16);
        unlock(&cursor);
        return n;

    case Qmouse:
        while(mousechanged(nil) == 0)
            sleep(&mouse.r, mousechanged, nil);

        mouse.qfull = 0;
        mousetime = seconds();

        /*
         * No lock of the indices is necessary here, because ri is only
         * updated by us, and there is only one mouse reader
```

```
         * at a time.  I suppose that more than one process
         * could try to read the fd at one time, but such behavior
         * is degenerate and already violates the calling
         * conventions for sleep above.
         */
        if(mouse.ri != mouse.wi) {
            m = mouse.queue[mouse.ri];
            if(++mouse.ri == nelem(mouse.queue))
                mouse.ri = 0;
        } else {
            while(!canlock(&cursor))
                tsleep(&up->sleepr, returnfalse, 0, TK2MS(1));

            m = mouse.Mousestate;
            unlock(&cursor);
        }

        b = buttonmap[m.buttons&7];
        /* put buttons 4 and 5 back in */
        b |= m.buttons & (3<<3);
        if (scrollswap)
            if (b == 8)
                b = 16;
            else if (b == 16)
                b = 8;
        snprint(buf, sizeof buf, "m%11d %11d %11d %11lud ",
            m.xy.x, m.xy.y,
            b,
            m.msec);
        mouse.lastcounter = m.counter;
        if(n > 1+4*12)
            n = 1+4*12;
        if(mouse.lastresize != mouse.resize){
            mouse.lastresize = mouse.resize;
            buf[0] = 'r';
        }
        memmove(va, buf, n);
        return n;
    }
    return 0;
}

static void
setbuttonmap(char* map)
{
    int i, x, one, two, three;

    one = two = three = 0;
    for(i = 0; i < 3; i++){
        if(map[i] == 0)
            error(Ebadarg);
        if(map[i] == '1'){
            if(one)
                error(Ebadarg);
            one = 1<<i;
        }
        else if(map[i] == '2'){
            if(two)
                error(Ebadarg);
            two = 1<<i;
```

```
        }
        else if(map[i] == '3'){
            if(three)
                error(Ebadarg);
            three = 1<<i;
        }
        else
            error(Ebadarg);
    }
    if(map[i])
        error(Ebadarg);

    memset(buttonmap, 0, 8);
    for(i = 0; i < 8; i++){
        x = 0;
        if(i & 1)
            x |= one;
        if(i & 2)
            x |= two;
        if(i & 4)
            x |= three;
        buttonmap[x] = i;
    }
}

static long
mousewrite(Chan *c, void *va, long n, vlong)
{
    char *p;
    Point pt;
    Cmdbuf *cb;
    Cmdtab *ct;
    char buf[64];
    int b, msec;

    p = va;
    switch((ulong)c->qid.path){
    case Qdir:
        error(Eisdir);

    case Qcursor:
        arch_cursoroff(true);
        if(n < 2*4+2*2*16){
            curs = arch_arrow;
            Cursortocursor(&arch_arrow);
        }else{
            n = 2*4+2*2*16;
            curs.offset.x = BGLONG(p+0);
            curs.offset.y = BGLONG(p+4);
            memmove(curs.clr, p+8, 2*16);
            memmove(curs.set, p+40, 2*16);
            Cursortocursor(&curs);
        }
        qlock(&mouse);
        mouse.redraw = true;
        mouseclock();
        qunlock(&mouse);
        arch_cursoron(true);
        return n;
```

```
case Qmousectl:
    cb = parsecmd(va, n);
    if(waserror()){
        free(cb);
        nexterror();
    }

    ct = lookupcmd(cb, mousectlmsg, nelem(mousectlmsg));

    switch(ct->index){
    case CMswap:
        if(mouseswap)
            setbuttonmap("123");
        else
            setbuttonmap("321");
        mouseswap ^= true;
        break;

    case CMscrollswap:
        scrollswap ^= true;
        break;

    case CMbuttonmap:
        if(cb->nf == 1)
            setbuttonmap("123");
        else
            setbuttonmap(cb->f[1]);
        break;

    case CMwildcard:
        kmousectl(cb); // device specific hook
        break;
    }

    free(cb);
    poperror();
    return n;

case Qmousein:
    if(n > sizeof buf-1)
        n = sizeof buf -1;
    memmove(buf, va, n);
    buf[n] = 0;
    p = 0;
    pt.x = strtol(buf+1, &p, 0);
    if(p == 0)
        error(Eshort);
    pt.y = strtol(p, &p, 0);
    if(p == 0)
        error(Eshort);
    b = strtol(p, &p, 0);
    msec = strtol(p, &p, 0);
    if(msec == 0)
        msec = TK2MS(CPUS(0)->ticks);
    mousetrack(pt.x, pt.y, b, msec);
    return n;

case Qmouse:
    if(n > sizeof buf-1)
        n = sizeof buf -1;
```

```
        memmove(buf, va, n);
        buf[n] = 0;
        p = 0;
        pt.x = strtoul(buf+1, &p, 0);
        if(p == 0)
            error(Eshort);
        pt.y = strtoul(p, 0, 0);
        qlock(&mouse);
        if(ptinrect(pt, gscreen->r)){
            mouse.xy = pt;
            mouse.redraw = true;
            mouse.track = true;
            mouseclock();
        }
        qunlock(&mouse);
        return n;
    }

    error(Egreg);
    return -1;
}

Dev mousedevtab = {
    .dc       =    'm',
    .name     =    "mouse",

    .reset    =    mousereset,
    .init     =    mouseinit,
    .shutdown =    devshutdown,
    .attach   =    mouseattach,
    .walk     =    mousewalk,
    .stat     =    mousestat,
    .open     =    mouseopen,
    .create   =    mousecreate,
    .close    =    mouseclose,
    .read     =    mouseread,
    .bread    =    devbread,
    .write    =    mousewrite,
    .bwrite   =    devbwrite,
    .remove   =    devremove,
    .wstat    =    devwstat,
};

void
Cursortocursor(Cursor *c)
{
    lock(&cursor);
    memmove(&cursor.Cursor, c, sizeof(Cursor));
    arch_ksetcursor(c);
    unlock(&cursor);
}


/*
 *  called by the clock routine to redraw the cursor
 */
static void
mouseclock(void)
{
    if(mouse.track){
```

```
        mousetrack(mouse.dx, mouse.dy, mouse.buttons, TK2MS(CPUS(0)->ticks));
        mouse.track = false;
        mouse.dx = 0;
        mouse.dy = 0;
    }
    if(mouse.redraw && canlock(&cursor)){
        mouse.redraw = false;
        arch_cursoroff(false);
        mouse.redraw = arch_cursoron(false);
        unlock(&cursor);
    }
    drawactive(false);
}

static int
scale(int x)
{
    int sign = 1;

    if(x < 0){
        sign = -1;
        x = -x;
    }
    switch(x){
    case 0:
    case 1:
    case 2:
    case 3:
        break;
    case 4:
        x = 6 + (mouse.acceleration>>2);
        break;
    case 5:
        x = 9 + (mouse.acceleration>>1);
        break;
    default:
        x *= mouse.maxacc;
        break;
    }
    return sign*x;
}

/*
 *  called at interrupt level to update the structure and
 *  awaken any waiting procs.
 */
void
mousetrack(int dx, int dy, int b, int msec)
{
    int x, y, lastb;

    if(gscreen==nil)
        return;

    if(mouse.acceleration){
        dx = scale(dx);
        dy = scale(dy);
    }
    x = mouse.xy.x + dx;
    if(x < gscreen->clipr.min.x)
```

```
        x = gscreen->clipr.min.x;
    if(x >= gscreen->clipr.max.x)
        x = gscreen->clipr.max.x;
    y = mouse.xy.y + dy;
    if(y < gscreen->clipr.min.y)
        y = gscreen->clipr.min.y;
    if(y >= gscreen->clipr.max.y)
        y = gscreen->clipr.max.y;

    lastb = mouse.buttons;
    mouse.xy = Pt(x, y);
    mouse.buttons = b|kbdbuttons;
    mouse.redraw = true;
    mouse.counter++;
    mouse.msec = msec;

    /*
     * if the queue fills, we discard the entire queue and don't
     * queue any more events until a reader polls the mouse.
     */
    if(!mouse.qfull && lastb != b) {    /* add to ring */
        mouse.queue[mouse.wi] = mouse.Mousestate;
        if(++mouse.wi == nelem(mouse.queue))
            mouse.wi = 0;
        if(mouse.wi == mouse.ri)
            mouse.qfull = 1;
    }
    wakeup(&mouse.r);
    drawactive(true);
}

int
mousechanged(void*)
{
    return mouse.lastcounter != mouse.counter ||
        mouse.lastresize != mouse.resize;
}

// called from outside devmouse.c
Point
mousexy(void)
{
    return mouse.xy;
}

// called from outside devmouse.c to let know devmouse.c of changes
void
mouseaccelerate(int x)
{
    mouse.acceleration = x;
    if(mouse.acceleration < 3)
        mouse.maxacc = 2;
    else
        mouse.maxacc = mouse.acceleration;
}

// called from outside devmouse.c to let know devmouse.c of changes
/*
 * notify reader that screen has been resized
 */
```

```
void
mouseresize(void)
{
    mouse.resize++;
    wakeup(&mouse.r);
}
```

Uses CMbuttonmap-950 719, CMscrollswap-951 719, CMswap-952 719, CMwildcard-953 719, COPEN 294b, CPUS 702, Cursortocursor() 719, DMDIR, Ebadarg 664d, Egreg 664d, Eshort 664d, Mouseinfo 719, Mousestate 719, OREAD 853i, QTDIR 70b, Qcursor-945 719, Qdir-944 719, Qmouse-946 719, Qmousectl-948 719, Qmousein-947 719, TK2MS 244d, addclock0link() 253f, buttonmap-955 719, canlock() 78f, curs 719, cursor 719, devattach() 297b, devbread() 434a, devbwrite() 434b, devdirread() 432c, devgen() 428, devremove() 810, devshutdown() 810, devstat() 431, devwalk() 429b, devwstat() 810, drawactive(), false, free() 104a, gscreen, incref() 87b, iseve() 591b, kbdbuttons 719, kbdmouse 719, kmousectl() 716, lock() 76d, lookupcmd() 595b, mouse 719, mouseattach() 719, mousechanged() 719, mouseclock() 719, mouseclose() 719, mousecreate() 719, mousectlmsg-954 719, mousedevgen() 719, mousedir-949 719, mousefromkbd() 719, mouseinit() 719, mouseopen() 719, mouseread() 719, mousereset() 719, mousestat() 719, mouseswap-956 719, mousetime-958 719, mousetrack() 719, mousewalk() 719, mousewrite() 719, nelem 853i, nexterror, openmode() 302b, parsecmd() 594c, poperror 666d, qlock() 81f, qunlock() 82c, returnfalse() 104c, scrollswap-957 719, seconds() 263b, setbuttonmap() 719, sleep, true, tsleep, unlock() 78d, up, wakeup, and waserror 666d.

E.9 devices/sys/

devices/sys/devroot.c

⟨*devroot.c forward decl* 730a⟩≡ (730c)
```
typedef struct Dirlist Dirlist;
```
Uses Dirlist 423b.

⟨*method rootclose* 730b⟩≡ (730c)
```
/*
 * sysremove() knows this is a nop
 */
static void
rootclose(Chan*)
{
}
```

⟨devices/sys/devroot.c 730c⟩≡
⟨*kernel basic includes* 49a⟩

```
//*****************************************************************************
// Constants and types
//*****************************************************************************
```

⟨*devroot enum Qxxx* 423a⟩

⟨*devroot.c forward decl* 730a⟩
⟨*struct* Dirlist 423b⟩

```
//*****************************************************************************
// Root
//*****************************************************************************

// note: directories have by convention 0 length
```

⟨*globals rootdir, rootdata, rootlist* 423c⟩

```
//*****************************************************************************
```

```
// Boot
//*****************************************************************************
```

⟨*globals bootdir, bootdata, bootlist* 426⟩

```
//*****************************************************************************
// Functions
//*****************************************************************************
```

⟨*function* `addlist` 424a⟩

⟨*function* `addbootfile` 427a⟩

⟨*function* `addrootdir` 424b⟩

⟨*method rootreset* 424c⟩

```
static Chan*
rootattach(char *spec)
{
    return devattach('/', spec);
}
```

⟨*function* `rootgen` 425a⟩

```
static Walkqid*
rootwalk(Chan *c, Chan *nc, char **name, int nname)
{
    return devwalk(c,  nc, name, nname, nil, 0, rootgen);
}

static int
rootstat(Chan *c, uchar *dp, int n)
{
    return devstat(c, dp, n, nil, 0, rootgen);
}

static Chan*
rootopen(Chan *c, int omode)
{
    return devopen(c, omode, nil, 0, devgen);
}
```

⟨*method rootclose* 730b⟩

⟨*method rootread* 425b⟩

```
static long
rootwrite(Chan*, void*, long, vlong)
{
    error(Egreg);
    return 0;
}
```

⟨*global* `rootdevtab` 422b⟩

Uses `Egreg` 664d, `devattach()` 297b, `devgen()` 428, `devstat()` 431, `devwalk()` 429b, and `rootgen()` 425a.

devices/sys/devsys.c

⟨*method syswalk* 732a⟩≡ (732e)
```
static Walkqid*
syswalk(Chan *c, Chan *nc, char **name, int nname)
{
    return devwalk(c, nc, name, nname, sysdir, nelem(sysdir), devgen);
}
```
Uses `devgen()` 428, `devwalk()` 429b, `nelem` 853i, and `sysdir-845` 622b.

⟨*method sysstat* 732b⟩≡ (732e)
```
static int
sysstat(Chan *c, uchar *dp, int n)
{
    return devstat(c, dp, n, sysdir, nelem(sysdir), devgen);
}
```
Uses `devgen()` 428, `devstat()` 431, `nelem` 853i, and `sysdir-845` 622b.

⟨*method sysopen* 732c⟩≡ (732e)
```
static Chan*
sysopen(Chan *c, int omode)
{
    c->aux = nil;
    c = devopen(c, omode, sysdir, nelem(sysdir), devgen);
    switch((ulong)c->qid.path){
    }
    return c;
}
```
Uses `devgen()` 428, `nelem` 853i, and `sysdir-845` 622b.

⟨*method sysclose* 732d⟩≡ (732e)
```
static void
sysclose(Chan *c)
{
    switch((ulong)c->qid.path){
    }
}
```

⟨`devices/sys/devsys.c` 732e⟩≡
⟨*kernel basic includes* 49a⟩

```
#include    <authsrv.h>
```

⟨*devsys.c enum Qxxx* 622a⟩

⟨*global* `sysdir` 622b⟩

⟨*devsys.c decls and globals* 624d⟩
⟨*global* `sysname` 625d⟩

⟨*method sysinit* 622c⟩

⟨*method sysattach* 622d⟩

⟨*method syswalk* 732a⟩

⟨*method sysstat* 732b⟩

⟨*method sysopen* 732c⟩

⟨*method sysclose* 732d⟩

⟨*method sysread* 622e⟩

⟨*method syswrite* 623b⟩

⟨*global* `sysdevtab` 623d⟩

TO LPSPLIT `devices/sys/devenv.c`

⟨*devenv.c forward decl* 733a⟩≡ (734)
```
static Egrp *envgrp(Chan *c);
static bool envwriteable(Chan *c);
```

⟨*function* `envgrp` 733b⟩≡ (734)
```
static Egrp*
envgrp(Chan *c)
{
    if(c->aux == nil)
        return up->egrp;
    return c->aux;
}
```
Uses `up`.

⟨*function* `envwriteable` 733c⟩≡ (734)
```
static bool
envwriteable(Chan *c)
{
    return iseve() || c->aux == nil;
}
```
Uses `iseve()` 591b.

⟨*function* `getconfenv` 733d⟩≡ (734)
```
/*
 * Return a copy of configuration environment as a sequence of strings.
 * The strings alternate between name and value.  A zero length name string
 * indicates the end of the list
 */
char *
getconfenv(void)
{
    Egrp *eg = &confegrp;
    Evalue *e;
    char *p, *q;
    int i, n;

    rlock(eg);
    if(waserror()) {
        runlock(eg);
        nexterror();
    }

    /* determine size */
    n = 0;
    for(i=0; i<eg->nent; i++){
        e = eg->ent[i];
        n += strlen(e->name) + e->len + 2;
    }
```

```
    p = malloc(n + 1);
    if(p == nil)
        error(Enomem);
    q = p;
    for(i=0; i<eg->nent; i++){
        e = eg->ent[i];
        strcpy(q, e->name);
        q += strlen(q) + 1;
        memmove(q, e->value, e->len);
        q[e->len] = 0;
        /* move up to the first null */
        q += strlen(q) + 1;
    }
    *q = 0;

    poperror();
    runlock(eg);
    return p;
}
```

Uses confegrp-787 734, malloc() 103c, nexterror, poperror 666d, rlock() 83e, runlock() 84b, and waserror 666d.

⟨devices/sys/devenv.c 734⟩≡

```
⟨kernel basic includes 49a⟩

enum
{
    Maxenvsize = 16300,
};

⟨devenv.c forward decl 733a⟩

static Egrp confegrp;   /* global environment group containing the kernel configuration */

static Evalue*
envlookup(Egrp *eg, char *name, ulong qidpath)
{
    Evalue *e;
    int i;

    for(i=0; i<eg->nent; i++){
        e = eg->ent[i];
        if(e->qid.path == qidpath || (name && e->name[0]==name[0] && strcmp(e->name, name) == 0))
            return e;
    }
    return nil;
}

static int
envgen(Chan *c, char *name, Dirtab*, int, int s, DirEntry *dp)
{
    Egrp *eg;
    Evalue *e;

    if(s == DEVDOTDOT){
        devdir(c, c->qid, "#e", 0, eve, DMDIR|0775, dp);
        return 1;
    }

    eg = envgrp(c);
    rlock(eg);
```

```
    e = nil;
    if(name)
        e = envlookup(eg, name, -1);
    else if(s < eg->nent)
        e = eg->ent[s];

    if(e == nil) {
        runlock(eg);
        return -1;
    }

    /* make sure name string continues to exist after we release lock */
    kstrcpy(up->genbuf, e->name, sizeof up->genbuf);
    devdir(c, e->qid, up->genbuf, e->len, eve, 0666, dp);
    runlock(eg);
    return 1;
}

static Chan*
envattach(char *spec)
{
    Chan *c;
    Egrp *egrp = nil;

    if(spec && *spec) {
        if(strcmp(spec, "c") == 0)
            egrp = &confegrp;
        if(egrp == nil)
            error(Ebadarg);
    }

    c = devattach('e', spec);
    c->aux = egrp;
    return c;
}

static Walkqid*
envwalk(Chan *c, Chan *nc, char **name, int nname)
{
    return devwalk(c, nc, name, nname, 0, 0, envgen);
}

static int
envstat(Chan *c, uchar *db, int n)
{
    if(c->qid.type & QTDIR)
        c->qid.vers = envgrp(c)->vers;
    return devstat(c, db, n, 0, 0, envgen);
}

static Chan*
envopen(Chan *c, int omode)
{
    Egrp *eg;
    Evalue *e;
    bool trunc;

    eg = envgrp(c);
    if(c->qid.type & QTDIR) {
        if(omode != OREAD)
```

```
            error(Eperm);
    }
    else {
        trunc = omode & OTRUNC;
        if(omode != OREAD && !envwriteable(c))
            error(Eperm);
        if(trunc)
            wlock(eg);
        else
            rlock(eg);
        e = envlookup(eg, nil, c->qid.path);
        if(e == nil) {
            if(trunc)
                wunlock(eg);
            else
                runlock(eg);
            error(Enonexist);
        }
        if(trunc && e->value) {
            e->qid.vers++;
            free(e->value);
            e->value = nil;
            e->len = 0;
        }
        if(trunc)
            wunlock(eg);
        else
            runlock(eg);
    }
    c->mode = openmode(omode);
    c->flag |= COPEN;
    c->offset = 0;
    return c;
}

static void
envcreate(Chan *c, char *name, int omode, ulong)
{
    Egrp *eg;
    Evalue *e;
    Evalue **ent;

    if(c->qid.type != QTDIR)
        error(Eperm);
    if(strlen(name) >= sizeof up->genbuf)
        error("name too long");         /* protect envgen */

    omode = openmode(omode);
    eg = envgrp(c);

    wlock(eg);
    if(waserror()) {
        wunlock(eg);
        nexterror();
    }

    if(envlookup(eg, name, -1))
        error(Eexist);

    e = smalloc(sizeof(Evalue));
```

```
    e->name = smalloc(strlen(name)+1);
    strcpy(e->name, name);

    // realloc
    if(eg->nent == eg->ment){
        eg->ment += 32;
        ent = smalloc(sizeof(eg->ent[0])*eg->ment);
        if(eg->nent)
            memmove(ent, eg->ent, sizeof(eg->ent[0])*eg->nent);
        free(eg->ent);
        eg->ent = ent;
    }
    e->qid.path = ++eg->path;
    e->qid.vers = 0;
    eg->vers++;
    eg->ent[eg->nent++] = e;
    c->qid = e->qid;

    wunlock(eg);
    poperror();

    c->offset = 0;
    c->mode = omode;
    c->flag |= COPEN;
}

static void
envremove(Chan *c)
{
    int i;
    Egrp *eg;
    Evalue *e;

    if(c->qid.type & QTDIR)
        error(Eperm);

    eg = envgrp(c);
    wlock(eg);
    e = nil;
    for(i=0; i<eg->nent; i++){
        if(eg->ent[i]->qid.path == c->qid.path){
            e = eg->ent[i];
            eg->nent--;
            eg->ent[i] = eg->ent[eg->nent];
            eg->vers++;
            break;
        }
    }
    wunlock(eg);
    if(e == nil)
        error(Enonexist);
    free(e->name);
    if(e->value)
        free(e->value);
    free(e);
}

static void
envclose(Chan *c)
{
```

```
    /*
     * cclose can't fail, so errors from remove will be ignored.
     * since permissions aren't checked,
     * envremove can't not remove it if its there.
     */
    if(c->flag & CRCLOSE)
        envremove(c);
}

static long
envread(Chan *c, void *a, long n, vlong off)
{
    Egrp *eg;
    Evalue *e;
    ulong offset = off;

    if(c->qid.type & QTDIR)
        return devdirread(c, a, n, 0, 0, envgen);

    eg = envgrp(c);
    rlock(eg);
    e = envlookup(eg, nil, c->qid.path);
    if(e == nil) {
        runlock(eg);
        error(Enonexist);
    }

    if(offset > e->len) /* protects against overflow converting vlong to ulong */
        n = 0;
    else if(offset + n > e->len)
        n = e->len - offset;
    if(n <= 0)
        n = 0;
    else
        memmove(a, e->value+offset, n);
    runlock(eg);
    return n;
}

static long
envwrite(Chan *c, void *a, long n, vlong off)
{
    byte *s;
    ulong len;
    Egrp *eg;
    Evalue *e;
    ulong offset = off;

    if(n <= 0)
        return 0;
    if(offset > Maxenvsize || n > (Maxenvsize - offset))
        error(Etoobig);

    eg = envgrp(c);
    wlock(eg);
    e = envlookup(eg, nil, c->qid.path);
    if(e == nil) {
        wunlock(eg);
        error(Enonexist);
    }
```

```
    len = offset+n;
    if(len > e->len) {
        s = smalloc(len);
        if(e->value){
            memmove(s, e->value, e->len);
            free(e->value);
        }
        e->value = s;
        e->len = len;
    }
    memmove(e->value+offset, a, n);
    e->qid.vers++;
    eg->vers++;
    wunlock(eg);
    return n;
}
```

⟨*global* envdevtab 599⟩

⟨*function* envcpy 598a⟩

⟨*function* envgrp 733b⟩

⟨*function* envwriteable 733c⟩

⟨*function* ksetenv 598b⟩

⟨*function* getconfenv 733d⟩

Uses COPEN 294b, CRCLOSE 322b, DEVDOTDOT 805, DMDIR, Etoobig 664d, Maxenvsize-786 734, OREAD 853i, OTRUNC, QTDIR 70b, devdir() 427c, devdirread() 432c, devstat() 431, devwalk() 429b, envgen() 734, envgrp() 733b, envlookup() 734, envremove() 734, envwriteable() 733c, eve 591a, free() 104a, kstrcpy() 669a, nexterror, openmode() 302b, poperror 666d, rlock() 83e, runlock() 84b, smalloc() 104b, up, waserror 666d, wlock() 85b, and wunlock() 85d.

devices/sys/devpipe.c

⟨*devpipe.c forward decl* 739a⟩≡ (739b)

```
typedef struct Pipe Pipe;
```

Uses Pipe 458c.

⟨devices/sys/devpipe.c 739b⟩≡

⟨*kernel basic includes* 49a⟩

```
// need that??
#include    "netif.h"
```

⟨*devpipe.c forward decl* 739a⟩

⟨*struct* Pipe 458c⟩

⟨*struct* Pipealloc 459a⟩

⟨*global* pipealloc 459b⟩

⟨*devpipe.c enum Qxxx* 459c⟩

⟨*global* `pipedir` 459d⟩

```
#define NPIPEDIR 3
```

⟨*method pipeinit* 459f⟩

⟨*method pipeattach* 460a⟩

⟨*function* `pipegen` 460b⟩

⟨*method pipewalk* 461a⟩

⟨*method pipestat* 461b⟩

⟨*method pipewstat* 462⟩

⟨*method pipeopen* 463a⟩

⟨*method pipeclose* 463b⟩

⟨*method piperead* 464⟩

⟨*method pipebread* 465a⟩

⟨*method pipewrite* 465b⟩

⟨*method pipebwrite* 466a⟩

⟨*global* `pipedevtab` 466b⟩

devices/sys/devproc.c

⟨*devproc.c forward decl* 740a⟩≡ (740b)

```
void    procctlreq(Proc*, char*, int);
int procctlmemio(Proc*, ulong, int, void*, int);
Chan*   proctext(Chan*, Proc*);
Segment* text2data(Proc*, Segment*);
int procstopped(void*);
void    mntscan(Mntwalk*, Proc*);
```

⟨`devices/sys/devproc.c` 740b⟩≡

⟨*kernel basic includes* 49a⟩

```
#include    <trace.h>
#include    <tos.h>
#include    <ureg.h>
```

⟨*struct* `Mntwalk` 351a⟩

⟨*devproc.c enum Qxxx* 602a⟩

⟨*devproc enum CMxxx* 602b⟩

```
enum{
```

⟨*constant* `Nevents` 500e⟩

⟨*constant* `Emask` 500f⟩

```
};

#define STATSIZE    (2*KNAMELEN+12+9*12)
```

⟨*global* `procdir` 603a⟩

⟨*global* `proccmd` 603b⟩

```
/* Segment type from portdat.h */
```
⟨*global* `sname` 113a⟩

⟨*devproc QXXX macros* 604a⟩

⟨*devproc.c forward decl* 740a⟩

⟨*global trace txxx* 500b⟩

⟨*clock callback profclock* 506b⟩

⟨*function* `procgen` 614d⟩

⟨*function* `_proctrace` 500d⟩

⟨*method procinit* 604d⟩

```
static Chan*
procattach(char *spec)
{
    return devattach('p', spec);
}

static Walkqid*
procwalk(Chan *c, Chan *nc, char **name, int nname)
{
    return devwalk(c, nc, name, nname, 0, 0, procgen);
}

static int
procstat(Chan *c, uchar *db, int n)
{
    return devstat(c, db, n, 0, 0, procgen);
}
```

⟨*function* `nonone` 616a⟩

⟨*method procopen* 611⟩

⟨*method procwstat* 613⟩

⟨*function* `procoffset` 614a⟩

⟨*function* `procqidwidth` 614b⟩

⟨*function* `procfdprint` 614c⟩

⟨*function* `procfds` 616b⟩

⟨*method procclose* 617⟩

⟨*function* `int2flag` 353a⟩

⟨*function* `procargs` 618c⟩

⟨*function* `eventsavailable` 500c⟩

⟨*method procread* 605d⟩

⟨*function* `mntscan` 353b⟩

⟨*method procwrite* 608c⟩

⟨*global* `procdevtab` 605a⟩

⟨*function* `proctext` 618d⟩

⟨*function* `procstopwait` 486c⟩

⟨*function* `procctlcloseone` 619a⟩

⟨*function* `procctlclosefiles` 619b⟩

⟨*function* `parsetime` 620a⟩

⟨*function* `procctlreq` 620b⟩

⟨*function* `procstopped` 486d⟩

⟨*function* `procctlmemio` 488f⟩

⟨*function* `text2data` 490⟩

Uses `devattach()` 297b, `devstat()` 431, `devwalk()` 429b, and `procgen()` 614d.

devices/sys/devdup.c

⟨*global* `dupdevtab` 742a⟩≡ (742b)

```
Dev dupdevtab = {
    .dc       =    'd',
    .name     =    "dup",

    .reset    =    devreset,
    .init     =    devinit,
    .shutdown =    devshutdown,
    .attach   =    dupattach,
    .walk     =    dupwalk,
    .stat     =    dupstat,
    .open     =    dupopen,
    .create   =    devcreate,
    .close    =    dupclose,
    .read     =    dupread,
    .bread    =    devbread,
    .write    =    dupwrite,
    .bwrite   =    devbwrite,
    .remove   =    devremove,
    .wstat    =    devwstat,
};
```

Uses `devbread()` 434a, `devbwrite()` 434b, `devcreate()` 810, `devinit()` 810, `devremove()` 810, `devreset()` 810, `devshutdown()` 810, `devwstat()` 810, `dupattach()` 742b, `dupclose()` 742b, `dupopen()` 742b, `dupread()` 742b, `dupstat()` 742b, `dupwalk()` 742b, and `dupwrite()` 742b.

⟨`devices/sys/devdup.c` 742b⟩≡

⟨*kernel basic includes* 49a⟩

```
/* Qid is (2*fd + (file is ctl))+1 */

static int
dupgen(Chan *c, char *, Dirtab*, int, int s, DirEntry *dp)
{
    Fgrp *fgrp = up->fgrp;
    Chan *f;
    static int perm[] = { 0400, 0200, 0600, 0 };
    int p;
    Qid q;

    if(s == DEVDOTDOT){
        devdir(c, c->qid, ".", 0, eve, DMDIR|0555, dp);
        return 1;
    }
    if(s == 0)
        return 0;
    s--;
    if(s/2 > fgrp->maxfd)
        return -1;
    if((f=fgrp->fd[s/2]) == nil)
        return 0;
    if(s & 1){
        p = 0400;
        snprint(up->genbuf, sizeof up->genbuf, "%dctl", s/2);
    }else{
        p = perm[f->mode&3];
        snprint(up->genbuf, sizeof up->genbuf, "%d", s/2);
    }
    mkqid(&q, s+1, 0, QTFILE);
    devdir(c, q, up->genbuf, 0, eve, p, dp);
    return 1;
}

static Chan*
dupattach(char *spec)
{
    return devattach('d', spec);
}

static Walkqid*
dupwalk(Chan *c, Chan *nc, char **name, int nname)
{
    return devwalk(c, nc, name, nname, (Dirtab *)0, 0, dupgen);
}

static int
dupstat(Chan *c, uchar *db, int n)
{
    return devstat(c, db, n, (Dirtab *)0, 0L, dupgen);
}

static Chan*
dupopen(Chan *c, int omode)
{
    Chan *f;
    int fd, twicefd;
```

```
    if(c->qid.type & QTDIR){
        if(omode != 0)
            error(Eisdir);
        c->mode = 0;
        c->flag |= COPEN;
        c->offset = 0;
        return c;
    }
    if(c->qid.type & QTAUTH)
        error(Eperm);
    twicefd = c->qid.path - 1;
    fd = twicefd/2;
    if((twicefd & 1)){
        /* ctl file */
        f = c;
        f->mode = openmode(omode);
        f->flag |= COPEN;
        f->offset = 0;
    }else{
        /* fd file */
        f = fdtochan(fd, openmode(omode), false, true);
        cclose(c);
    }
    if(omode & OCEXEC)
        f->flag |= CCEXEC;
    return f;
}

static void
dupclose(Chan*)
{
}

static long
dupread(Chan *c, void *va, long n, vlong offset)
{
    char *a = va;
    char buf[256];
    int fd, twicefd;

    if(c->qid.type == QTDIR)
        return devdirread(c, a, n, (Dirtab *)0, 0L, dupgen);
    twicefd = c->qid.path - 1;
    fd = twicefd/2;
    if(twicefd & 1){
        c = fdtochan(fd, -1, false, true);
        procfdprint(c, fd, 0, buf, sizeof buf);
        cclose(c);
        return readstr((ulong)offset, va, n, buf);
    }
    panic("dupread");
    return 0;
}

static long
dupwrite(Chan*, void*, long, vlong)
{
    error(Eperm);
    return 0;       /* not reached */
}
```

⟨*global* dupdevtab 742a⟩

Uses CCEXEC 321b, COPEN 294b, DEVDOTDOT 805, DMDIR, OCEXEC, QTAUTH, QTDIR 70b, QTFILE, cclose, devattach() 297b, devdir() 427c, devdirread() 432c, devstat() 431, devwalk() 429b, dupgen() 742b, eve 591a, false, fdtochan() 302d, openmode() 302b, procfdprint() 614c, readstr() 670d, true, and up.

devices/sys/devmnt.c

⟨*devmnt.c forward decl* 745a⟩≡ (746b)

```
Mnt*    mntchk(Chan*);
void    mntdirfix(uchar*, Chan*);
Mntrpc* mntflushalloc(Mntrpc*, ulong);
void    mntflushfree(Mnt*, Mntrpc*);
void    mntgate(Mnt*);
void    mntqrm(Mnt*, Mntrpc*);
Mntrpc* mntralloc(Chan*, ulong);
long    mntrdwr(int, Chan*, void*, long, vlong);
int mntrpcread(Mnt*, Mntrpc*);
void    mountio(Mnt*, Mntrpc*);
void    mountmux(Mnt*, Mntrpc*);
void    mountrpc(Mnt*, Mntrpc*);
int rpcattn(void*);
Chan*   mntchan(void);
```

⟨Chan *other fields* 745b⟩+≡ (68b) ◁432b 816b▷

```
int fid;      /* for devmnt */
ulong iounit;     /* chunk size for i/o; 0==default */

Mnt*  mux;      /* Mnt for clients using me for messages */

Qid mqid;     /* qid of root of mount point */
```

⟨*global* mntdevtab 745c⟩≡ (746b)

```
Dev mntdevtab = {
    .dc       =    'M',
    .name     =    "mnt",

    .reset    =    mntreset,
    .init     =    devinit,
    .shutdown =    devshutdown,
    .attach   =    mntattach,
    .walk     =    mntwalk,
    .stat     =    mntstat,
    .open     =    mntopen,
    .create   =    mntcreate,
    .close    =    mntclose,
    .read     =    mntread,
    .bread    =    devbread,
    .write    =    mntwrite,
    .bwrite   =    devbwrite,
    .remove   =    mntremove,
    .wstat    =    mntwstat,
};
```

Uses devbread() 434a, devbwrite() 434b, devinit() 810, devshutdown() 810, mntattach() 588, mntclose() 746b, mntcreate() 746b, mntopen() 746b, mntread() 746b, mntremove() 746b, mntreset() 746b, mntstat() 746b, mntwalk() 589, mntwrite() 746b, and mntwstat() 746b.

⟨ChannelFlag *cases* 745d⟩+≡ (294b) ◁582c

```
CCACHE  = 0x0080,   /* client cache */
```

⟨namec() *set channel flag before open* 746a⟩≡ (305f)

```
if(omode == OEXEC)
    c->flag &= ~CCACHE;
```

Uses CCACHE 745d and OEXEC 301c.

⟨devices/sys/devmnt.c 746b⟩≡

⟨*kernel basic includes* 49a⟩

```
#define MAXRPC (IOHDRSZ+8192)
```

⟨*devmnt.c forward decl* 745a⟩

```
void (*mntstats)(int, Chan*, uvlong, ulong);

static void
mntreset(void)
{
    mntalloc.id = 1;
    mntalloc.tagmask[0] = 1;            /* don't allow 0 as a tag */
    mntalloc.tagmask[NMASK-1] = 0x80000000UL;   /* don't allow NOTAG */
    fmtinstall('F', fcallfmt);
    fmtinstall('D', dirfmt);
/* We can't install %M since eipfmt does and is used in the kernel [sape] */

    cinit();
}


Chan*
mntauth(Chan *c, char *spec)
{
    Mnt *m;
    Mntrpc *r;

    m = c->mux;

    if(m == nil){
        mntversion(c, VERSION9P, MAXRPC, 0);
        m = c->mux;
        if(m == nil)
            error(Enoversion);
    }

    c = mntchan();
    if(waserror()) {
        /* Close must not be called since it will
         * call mnt recursively
         */
        chanfree(c);
        nexterror();
    }

    r = mntralloc(0, m->msize);

    if(waserror()) {
        mntfree(r);
        nexterror();
    }
```

```
    r->request.type = Tauth;
    r->request.afid = c->fid;
    r->request.uname = up->user;
    r->request.aname = spec;
    mountrpc(m, r);

    c->qid = r->reply.aqid;
    c->mchan = m->c;
    incref(m->c);
    c->mqid = c->qid;
    c->mode = ORDWR;

    poperror(); /* r */
    mntfree(r);

    poperror(); /* c */

    return c;

}
```

⟨*function* mntattach 588⟩

```
Chan*
mntchan(void)
{
    Chan *c;

    c = devattach('M', 0);
    lock(&mntalloc);
    c->dev = mntalloc.id++;
    unlock(&mntalloc);

    if(c->mchan)
        panic("mntchan non-zero %p", c->mchan);
    return c;
}
```

⟨*function* mntwalk 589⟩

```
static int
mntstat(Chan *c, uchar *dp, int n)
{
    Mnt *m;
    Mntrpc *r;

    if(n < BIT16SZ)
        error(Eshortstat);
    m = mntchk(c);
    r = mntralloc(c, m->msize);
    if(waserror()) {
        mntfree(r);
        nexterror();
    }
    r->request.type = Tstat;
    r->request.fid = c->fid;
    mountrpc(m, r);

    if(r->reply.nstat > n){
        n = BIT16SZ;
```

```
        PBIT16((uchar*)dp, r->reply.nstat-2);
    }else{
        n = r->reply.nstat;
        memmove(dp, r->reply.stat, n);
        validstat(dp, n);
        mntdirfix(dp, c);
    }
    poperror();
    mntfree(r);
    return n;
}

static Chan*
mntopencreate(int type, Chan *c, char *name, int omode, ulong perm)
{
    Mnt *m;
    Mntrpc *r;

    m = mntchk(c);
    r = mntralloc(c, m->msize);
    if(waserror()) {
        mntfree(r);
        nexterror();
    }
    r->request.type = type;
    r->request.fid = c->fid;
    r->request.mode = omode;
    if(type == Tcreate){
        r->request.perm = perm;
        r->request.name = name;
    }
    mountrpc(m, r);

    c->qid = r->reply.qid;
    c->offset = 0;
    c->mode = openmode(omode);
    c->iounit = r->reply.iounit;
    if(c->iounit == 0 || c->iounit > m->msize-IOHDRSZ)
        c->iounit = m->msize-IOHDRSZ;
    c->flag |= COPEN;
    poperror();
    mntfree(r);

    if(c->flag & CCACHE)
        copen(c);

    return c;
}

static Chan*
mntopen(Chan *c, int omode)
{
    return mntopencreate(Topen, c, nil, omode, 0);
}

static void
mntcreate(Chan *c, char *name, int omode, ulong perm)
{
    mntopencreate(Tcreate, c, name, omode, perm);
}
```

```
static void
mntclunk(Chan *c, int t)
{
    Mnt *m;
    Mntrpc *r;

    m = mntchk(c);
    r = mntralloc(c, m->msize);
    if(waserror()){
        mntfree(r);
        nexterror();
    }

    r->request.type = t;
    r->request.fid = c->fid;
    mountrpc(m, r);
    mntfree(r);
    poperror();
}



static void
mntclose(Chan *c)
{
    mntclunk(c, Tclunk);
}

static void
mntremove(Chan *c)
{
    mntclunk(c, Tremove);
}

static int
mntwstat(Chan *c, uchar *dp, int n)
{
    Mnt *m;
    Mntrpc *r;

    m = mntchk(c);
    r = mntralloc(c, m->msize);
    if(waserror()) {
        mntfree(r);
        nexterror();
    }
    r->request.type = Twstat;
    r->request.fid = c->fid;
    r->request.nstat = n;
    r->request.stat = dp;
    mountrpc(m, r);
    poperror();
    mntfree(r);
    return n;
}

static long
mntread(Chan *c, void *buf, long n, vlong off)
{
```

```
    uchar *p, *e;
    int nc, cache, isdir, dirlen;

    isdir = 0;
    cache = c->flag & CCACHE;
    if(c->qid.type & QTDIR) {
        cache = 0;
        isdir = 1;
    }

    p = buf;
    if(cache) {
        nc = cread(c, buf, n, off);
        if(nc > 0) {
            n -= nc;
            if(n == 0)
                return nc;
            p += nc;
            off += nc;
        }
        n = mntrdwr(Tread, c, p, n, off);
        cupdate(c, p, n, off);
        return n + nc;
    }

    n = mntrdwr(Tread, c, buf, n, off);
    if(isdir) {
        for(e = &p[n]; p+BIT16SZ < e; p += dirlen){
            dirlen = BIT16SZ+GBIT16(p);
            if(p+dirlen > e)
                break;
            validstat(p, dirlen);
            mntdirfix(p, c);
        }
        if(p != e)
            error(Esbadstat);
    }
    return n;
}

static long
mntwrite(Chan *c, void *buf, long n, vlong off)
{
    return mntrdwr(Twrite, c, buf, n, off);
}

long
mntrdwr(int type, Chan *c, void *buf, long n, vlong off)
{
    Mnt *m;
    Mntrpc *r;
    char *uba;
    int cache;
    ulong cnt, nr, nreq;

    m = mntchk(c);
    uba = buf;
    cnt = 0;
    cache = c->flag & CCACHE;
    if(c->qid.type & QTDIR)
```

```
        cache = 0;
    for(;;) {
        r = mntralloc(c, m->msize);
        if(waserror()) {
            mntfree(r);
            nexterror();
        }
        r->request.type = type;
        r->request.fid = c->fid;
        r->request.offset = off;
        r->request.data = uba;
        nr = n;
        if(nr > m->msize-IOHDRSZ)
            nr = m->msize-IOHDRSZ;
        r->request.count = nr;
        mountrpc(m, r);
        nreq = r->request.count;
        nr = r->reply.count;
        if(nr > nreq)
            nr = nreq;

        if(type == Tread)
            r->b = bl2mem((uchar*)uba, r->b, nr);
        else if(cache)
            cwrite(c, (uchar*)uba, nr, off);

        poperror();
        mntfree(r);
        off += nr;
        uba += nr;
        cnt += nr;
        n -= nr;
        if(nr != nreq || n == 0 || up->nnote)
            break;
    }
    return cnt;
}
```

⟨*function* mountrpc 586⟩

⟨*function* mountio 587⟩

```
static int
doread(Mnt *m, int len)
{
    Block *b;

    while(qlen(m->q) < len){
        b = devtab[m->c->type]->bread(m->c, m->msize, 0);
        if(b == nil)
            return -1;
        if(blocklen(b) == 0){
            freeblist(b);
            return -1;
        }
        qaddlist(m->q, b);
    }
    return 0;
}
```

```
int
mntrpcread(Mnt *m, Mntrpc *r)
{
    int i, t, len, hlen;
    Block *b, **l, *nb;

    r->reply.type = 0;
    r->reply.tag = 0;

    /* read at least length, type, and tag and pullup to a single block */
    if(doread(m, BIT32SZ+BIT8SZ+BIT16SZ) < 0)
        return -1;
    nb = pullupqueue(m->q, BIT32SZ+BIT8SZ+BIT16SZ);

    /* read in the rest of the message, avoid ridiculous (for now) message sizes */
    len = GBIT32(nb->rp);
    if(len > m->msize){
        qdiscard(m->q, qlen(m->q));
        return -1;
    }
    if(doread(m, len) < 0)
        return -1;

    /* pullup the header (i.e. everything except data) */
    t = nb->rp[BIT32SZ];
    switch(t){
    case Rread:
        hlen = BIT32SZ+BIT8SZ+BIT16SZ+BIT32SZ;
        break;
    default:
        hlen = len;
        break;
    }
    nb = pullupqueue(m->q, hlen);

    if(convM2S(nb->rp, len, &r->reply) <= 0){
        /* bad message, dump it */
        print("mntrpcread: convM2S failed\n");
        qdiscard(m->q, len);
        return -1;
    }

    /* hang the data off of the fcall struct */
    l = &r->b;
    *l = nil;
    do {
        b = qremove(m->q);
        if(hlen > 0){
            b->rp += hlen;
            len -= hlen;
            hlen = 0;
        }
        i = BLEN(b);
        if(i <= len){
            len -= i;
            *l = b;
            l = &(b->next);
        } else {
            /* split block and put unused bit back */
```

```
            nb = allocb(i-len);
            memmove(nb->wp, b->rp+len, i-len);
            b->wp = b->rp+len;
            nb->wp += i-len;
            qputback(m->q, nb);
            *l = b;
            return 0;
        }
    }while(len > 0);

    return 0;
}

void
mntgate(Mnt *m)
{
    Mntrpc *q;

    lock(m);
    m->rip = 0;
    for(q = m->queue; q; q = q->list) {
        if(q->done == 0)
        if(wakeup(&q->r))
            break;
    }
    unlock(m);
}

void
mountmux(Mnt *m, Mntrpc *r)
{
    Mntrpc **l, *q;

    lock(m);
    l = &m->queue;
    for(q = *l; q; q = q->list) {
        /* look for a reply to a message */
        if(q->request.tag == r->reply.tag) {
            *l = q->list;
            if(q != r) {
                /*
                 * Completed someone else.
                 * Trade pointers to receive buffer.
                 */
                q->reply = r->reply;
                q->b = r->b;
                r->b = nil;
            }
            q->done = 1;
            unlock(m);
            if(mntstats != nil)
                (*mntstats)(q->request.type,
                    m->c, q->stime,
                    q->reqlen + r->replen);
            if(q != r)
                wakeup(&q->r);
            return;
        }
        l = &q->list;
    }
```

```
    unlock(m);
    print("unexpected reply tag %ud; type %d\n", r->reply.tag, r->reply.type);
}

/*
 * Create a new flush request and chain the previous
 * requests from it
 */
Mntrpc*
mntflushalloc(Mntrpc *r, ulong iounit)
{
    Mntrpc *fr;

    fr = mntralloc(0, iounit);

    fr->request.type = Tflush;
    if(r->request.type == Tflush)
        fr->request.oldtag = r->request.oldtag;
    else
        fr->request.oldtag = r->request.tag;
    fr->flushed = r;

    return fr;
}

/*
 *  Free a chain of flushes.  Remove each unanswered
 *  flush and the original message from the unanswered
 *  request queue.  Mark the original message as done
 *  and if it hasn't been answered set the reply to to
 *  Rflush.
 */
void
mntflushfree(Mnt *m, Mntrpc *r)
{
    Mntrpc *fr;

    while(r){
        fr = r->flushed;
        if(!r->done){
            r->reply.type = Rflush;
            mntqrm(m, r);
        }
        if(fr)
            mntfree(r);
        r = fr;
    }
}

int
alloctag(void)
{
    int i, j;
    ulong v;

    for(i = 0; i < NMASK; i++){
        v = mntalloc.tagmask[i];
        if(v == ~0UL)
            continue;
        for(j = 0; j < 1<<TAGSHIFT; j++)
```

```
            if((v & (1<<j)) == 0){
                mntalloc.tagmask[i] |= 1<<j;
                return (i<<TAGSHIFT) + j;
            }
    }
    panic("no friggin tags left");
    return NOTAG;
}


Mntrpc*
mntralloc(Chan *c, ulong msize)
{
    Mntrpc *new;

    lock(&mntalloc);
    new = mntalloc.rpcfree;
    if(new == nil){
        new = malloc(sizeof(Mntrpc));
        if(new == nil) {
            unlock(&mntalloc);
            exhausted("mount rpc header");
        }
        /*
         * The header is split from the data buffer as
         * mountmux may swap the buffer with another header.
         */
        new->rpc = mallocz(msize, 0);
        if(new->rpc == nil){
            free(new);
            unlock(&mntalloc);
            exhausted("mount rpc buffer");
        }
        new->rpclen = msize;
        new->request.tag = alloctag();
    }
    else {
        mntalloc.rpcfree = new->list;
        mntalloc.nrpcfree--;
        if(new->rpclen < msize){
            free(new->rpc);
            new->rpc = mallocz(msize, 0);
            if(new->rpc == nil){
                free(new);
                mntalloc.nrpcused--;
                unlock(&mntalloc);
                exhausted("mount rpc buffer");
            }
            new->rpclen = msize;
        }
    }
    mntalloc.nrpcused++;
    unlock(&mntalloc);
    new->c = c;
    new->done = 0;
    new->flushed = nil;
    new->b = nil;
    return new;
}
```

```
void
mntqrm(Mnt *m, Mntrpc *r)
{
    Mntrpc **l, *f;

    lock(m);
    r->done = 1;

    l = &m->queue;
    for(f = *l; f; f = f->list) {
        if(f == r) {
            *l = r->list;
            break;
        }
        l = &f->list;
    }
    unlock(m);
}

Mnt*
mntchk(Chan *c)
{
    Mnt *m;

    /* This routine is mostly vestiges of prior lives; now it's just sanity checking */

    if(c->mchan == nil)
        panic("mntchk 1: nil mchan c %s\n", chanpath(c));

    m = c->mchan->mux;

    if(m == nil)
        print("mntchk 2: nil mux c %s c->mchan %s \n", chanpath(c), chanpath(c->mchan));

    /*
     * Was it closed and reused (was error(Eshutdown); now, it cannot happen)
     */
    if(m->id == 0 || m->id >= c->dev)
        panic("mntchk 3: can't happen");

    return m;
}

/*
 * Rewrite channel type and dev for in-flight data to
 * reflect local values.  These entries are known to be
 * the first two in the Dir encoding after the count.
 */
void
mntdirfix(uchar *dirbuf, Chan *c)
{
    uint r;

    r = devtab[c->type]->dc;
    dirbuf += BIT16SZ;  /* skip count */
    PBIT16(dirbuf, r);
    dirbuf += BIT16SZ;
    PBIT32(dirbuf, c->dev);
}
```

```
int
rpcattn(void *v)
{
    Mntrpc *r;

    r = v;
    return r->done || r->m->rip == 0;
}
```

⟨*global* mntdevtab 745c⟩

Uses BLEN 435b, CCACHE 745d, COPEN 294b, Enoversion 664d, Esbadstat 664d, Eshortstat 664d, MAXRPC-825 746b, NMASK 580d, ORDWR 301c, QTDIR 70b, TAGSHIFT 580d, allocb() 436b, alloctag() 746b, bl2mem() 445, blocklen() 440a, chanfree() 296, chanpath() 318b, cinit() 817, copen() 817, cread() 817, cupdate() 817, cwrite() 817, devattach() 297b, doread() 746b, exhausted() 666c, free() 104a, freeblist() 439a, incref() 87b, lock() 76d, malloc() 103c, mallocz() 104d, mntalloc 580c, mntchan() 746b, mntchk() 746b, mntclunk() 746b, mntdirfix() 746b, mntfree() 581b, mntopencreate() 746b, mntqrm() 746b, mntralloc() 746b, mntrdwr() 746b, mntstats 746b, mntversion() 582d, mountrpc() 586, nexterror, openmode() 302b, poperror 666d, print, pullupqueue() 401a, qaddlist() 410b, qdiscard() 402b, qlen() 419c, qputback() 411a, qremove() 410c, unlock() 78d, up, validstat() 340b, wakeup, and waserror 666d.

devices/sys/devsrv.c

⟨*devsrv.c forward decl* 757a⟩≡ (763b)

```
typedef struct Srv Srv;
```

Uses Srv 757b.

⟨*struct* Srv 757b⟩≡ (763b)

```
struct Srv
{
    char    *name;
    char    *owner;
    ulong   perm;
    Chan    *chan;
    Srv *link;
    ulong   path;
};
```

⟨*global* srv 757c⟩≡ (763b)

```
static Srv  *srv;
```

⟨*global* srvlk 757d⟩≡ (763b)

```
static QLock    srvlk;
```

⟨*global* qidpath 757e⟩≡ (763b)

```
static int  qidpath;
```

⟨*method srvgen* 757f⟩≡ (763b)

```
static int
srvgen(Chan *c, char*, Dirtab*, int, int s, DirEntry *dp)
{
    Srv *sp;
    Qid q;

    if(s == DEVDOTDOT){
        devdir(c, c->qid, "#s", 0, eve, 0555, dp);
        return 1;
    }
```

```
    qlock(&srvlk);
    for(sp = srv; sp && s; sp = sp->link)
        s--;

    if(sp == 0) {
        qunlock(&srvlk);
        return -1;
    }

    mkqid(&q, sp->path, 0, QTFILE);
    /* make sure name string continues to exist after we release lock */
    kstrcpy(up->genbuf, sp->name, sizeof up->genbuf);
    devdir(c, q, up->genbuf, 0, sp->owner, sp->perm, dp);
    qunlock(&srvlk);
    return 1;
}
```

⟨*method srvinit* 758a⟩≡ (763b)

```
static void
srvinit(void)
{
    qidpath = 1;
}
```

Uses qidpath-785 757e.

⟨*function* srvlookup 758b⟩≡ (763b)

```
static Srv*
srvlookup(char *name, ulong qidpath)
{
    Srv *sp;
    for(sp = srv; sp; sp = sp->link)
        if(sp->path == qidpath || (name && strcmp(sp->name, name) == 0))
            return sp;
    return nil;
}
```

Uses srv-784 757c.

⟨*function* srvname 758c⟩≡ (763b)

```
char*
srvname(Chan *c)
{
    int size;
    Srv *sp;
    char *s;

    for(sp = srv; sp; sp = sp->link)
        if(sp->chan == c){
            size = 3+strlen(sp->name)+1;
            s = smalloc(size);
            snprint(s, size, "#s/%s", sp->name);
            return s;
        }
    return nil;
}
```

Uses smalloc() 104b and srv-784 757c.

⟨*method srvopen* 758d⟩≡ (763b)

```
static Chan*
srvopen(Chan *c, int omode)
```

```
{
    Srv *sp;

    if(c->qid.type == QTDIR){
        if(omode & ORCLOSE)
            error(Eperm);
        if(omode != OREAD)
            error(Eisdir);
        c->mode = omode;
        c->flag |= COPEN;
        c->offset = 0;
        return c;
    }
    qlock(&srvlk);
    if(waserror()){
        qunlock(&srvlk);
        nexterror();
    }

    sp = srvlookup(nil, c->qid.path);
    if(sp == 0 || sp->chan == 0)
        error(Eshutdown);

    if(omode&OTRUNC)
        error("srv file already exists");
    if(openmode(omode)!=sp->chan->mode && sp->chan->mode!=ORDWR)
        error(Eperm);
    devpermcheck(sp->owner, sp->perm, omode);

    cclose(c);
    incref(sp->chan);
    qunlock(&srvlk);
    poperror();
    return sp->chan;
}
```

Uses COPEN 294b, Eshutdown 664d, ORCLOSE, ORDWR 301c, OREAD 853i, OTRUNC, QTDIR 70b, cclose, devpermcheck() 432d, incref() 87b, nexterror, openmode() 302b, poperror 666d, qlock() 81f, qunlock() 82c, srvlk-783 757d, srvlookup() 758b, and waserror 666d.

⟨*method srvcreate* 759⟩≡ (763b)

```
static void
srvcreate(Chan *c, char *name, int omode, ulong perm)
{
    char *sname;
    Srv *sp;

    if(openmode(omode) != OWRITE)
        error(Eperm);

    if(omode & OCEXEC)  /* can't happen */
        panic("someone broke namec");

    sp = smalloc(sizeof *sp);
    sname = smalloc(strlen(name)+1);

    qlock(&srvlk);
    if(waserror()){
        free(sp);
        free(sname);
        qunlock(&srvlk);
```

```
        nexterror();
    }
    if(sp == nil || sname == nil)
        error(Enomem);
    if(srvlookup(name, -1))
        error(Eexist);

    sp->path = qidpath++;
    sp->link = srv;
    strcpy(sname, name);
    sp->name = sname;
    c->qid.type = QTFILE;
    c->qid.path = sp->path;
    srv = sp;
    qunlock(&srvlk);
    poperror();

    kstrdup(&sp->owner, up->user);
    sp->perm = perm&0777;

    c->flag |= COPEN;
    c->mode = OWRITE;
}
```

Uses COPEN 294b, OCEXEC, OWRITE, QTFILE, free() 104a, kstrdup() 669b, nexterror, openmode() 302b, poperror 666d, qidpath-785 757e, qlock() 81f, qunlock() 82c, smalloc() 104b, srv-784 757c, srvlk-783 757d, srvlookup() 758b, up, and waserror 666d.

⟨*method srvremove* 760⟩≡ (763b)

```
static void
srvremove(Chan *c)
{
    Srv *sp, **l;

    if(c->qid.type == QTDIR)
        error(Eperm);

    qlock(&srvlk);
    if(waserror()){
        qunlock(&srvlk);
        nexterror();
    }
    l = &srv;
    for(sp = *l; sp; sp = sp->link) {
        if(sp->path == c->qid.path)
            break;

        l = &sp->link;
    }
    if(sp == 0)
        error(Enonexist);

    /*
     * Only eve can remove system services.
     * No one can remove #s/boot.
     */
    if(strcmp(sp->owner, eve) == 0 && !iseve())
        error(Eperm);
    if(strcmp(sp->name, "boot") == 0)
        error(Eperm);
```

```
    /*
     * No removing personal services.
     */
    if((sp->perm&7) != 7 && strcmp(sp->owner, up->user) && !iseve())
        error(Eperm);

    *l = sp->link;
    qunlock(&srvlk);
    poperror();

    if(sp->chan)
        cclose(sp->chan);
    free(sp->owner);
    free(sp->name);
    free(sp);
}
```

Uses QTDIR 70b, cclose, eve 591a, free() 104a, iseve() 591b, nexterror, poperror 666d, qlock() 81f, qunlock() 82c, srv-784 757c, srvlk-783 757d, up, and waserror 666d.

⟨*method srvwstat* 761⟩≡ (763b)

```
static int
srvwstat(Chan *c, uchar *dp, int n)
{
    char *strs;
    DirEntry d;
    Srv *sp;

    if(c->qid.type & QTDIR)
        error(Eperm);

    strs = nil;
    qlock(&srvlk);
    if(waserror()){
        qunlock(&srvlk);
        free(strs);
        nexterror();
    }

    sp = srvlookup(nil, c->qid.path);
    if(sp == 0)
        error(Enonexist);

    if(strcmp(sp->owner, up->user) != 0 && !iseve())
        error(Eperm);

    strs = smalloc(n);
    n = convM2D(dp, n, &d, strs);
    if(n == 0)
        error(Eshortstat);
    if(d.mode != ~0UL)
        sp->perm = d.mode & 0777;
    if(d.uid && *d.uid)
        kstrdup(&sp->owner, d.uid);
    if(d.name && *d.name && strcmp(sp->name, d.name) != 0) {
        if(strchr(d.name, '/') != nil)
            error(Ebadchar);
        kstrdup(&sp->name, d.name);
    }
    qunlock(&srvlk);
    free(strs);
```

```
    poperror();
    return n;
}
```

⟨*method srvclose* 762a⟩≡ (763b)

```
static void
srvclose(Chan *c)
{
    /*
     * in theory we need to override any changes in removability
     * since open, but since all that's checked is the owner,
     * which is immutable, all is well.
     */
    if(c->flag & CRCLOSE){
        if(waserror())
            return;
        srvremove(c);
        poperror();
    }
}
```

Uses CRCLOSE 322b, poperror 666d, srvremove() 760, and waserror 666d.

⟨*method srvread* 762b⟩≡ (763b)

```
static long
srvread(Chan *c, void *va, long n, vlong)
{
    error_if_not_dir(c);
    return devdirread(c, va, n, 0, 0, srvgen);
}
```

Uses devdirread() 432c and error_if_not_dir() 323b.

⟨*method srvwrite* 762c⟩≡ (763b)

```
static long
srvwrite(Chan *c, void *va, long n, vlong)
{
    Srv *sp;
    Chan *c1;
    int fd;
    char buf[32];

    if(n >= sizeof buf)
        error(Egreg);
    memmove(buf, va, n);    /* so we can NUL-terminate */
    buf[n] = 0;
    fd = strtoul(buf, 0, 0);

    c1 = fdtochan(fd, -1, false, true);    /* error check and inc ref */

    qlock(&srvlk);
    if(waserror()) {
        qunlock(&srvlk);
        cclose(c1);
        nexterror();
    }
    if(c1->flag & (CCEXEC|CRCLOSE))
        error("posted fd has remove-on-close or close-on-exec");
    if(c1->qid.type & QTAUTH)
        error("cannot post auth file in srv");
    sp = srvlookup(nil, c->qid.path);
    if(sp == 0)
```

```
        error(Enonexist);

    if(sp->chan)
        error(Ebadusefd);

    sp->chan = c1;
    qunlock(&srvlk);
    poperror();
    return n;
}
```

Uses CCEXEC 321b, CRCLOSE 322b, Ebadusefd 664d, Egreg 664d, QTAUTH, cclose, false, fdtochan() 302d, nexterror, poperror 666d, qlock() 81f, qunlock() 82c, srvlk-783 757d, srvlookup() 758b, true, and waserror 666d.

⟨*global* srvdevtab 763a⟩≡ (763b)

```
Dev srvdevtab = {
    .dc       =    's',
    .name     =    "srv",

    .reset    =    devreset,
    .init     =    srvinit,
    .shutdown =    devshutdown,
    .attach   =    srvattach,
    .walk     =    srvwalk,
    .stat     =    srvstat,
    .open     =    srvopen,
    .create   =    srvcreate,
    .close    =    srvclose,
    .read     =    srvread,
    .bread    =    devbread,
    .write    =    srvwrite,
    .bwrite   =    devbwrite,
    .remove   =    srvremove,
    .wstat    =    srvwstat,
};
```

Uses devbread() 434a, devbwrite() 434b, devreset() 810, devshutdown() 810, srvattach() 763b, srvclose() 762a, srvcreate() 759, srvinit() 758a, srvread() 762b, srvremove() 760, srvstat() 763b, srvwalk() 763b, and srvwrite() 762c.

⟨devices/sys/devsrv.c 763b⟩≡

```
⟨kernel basic includes 49a⟩

⟨devsrv.c forward decl 757a⟩

⟨struct Srv 757b⟩

⟨global srvlk 757d⟩
⟨global srv 757c⟩
⟨global qidpath 757e⟩

⟨method srvgen 757f⟩

⟨method srvinit 758a⟩

static Chan*
srvattach(char *spec)
{
    return devattach('s', spec);
}

static Walkqid*
srvwalk(Chan *c, Chan *nc, char **name, int nname)
```

```
{
    return devwalk(c, nc, name, nname, 0, 0, srvgen);
}
```

⟨*function* `srvlookup` 758b⟩

```
static int
srvstat(Chan *c, uchar *db, int n)
{
    return devstat(c, db, n, 0, 0, srvgen);
}
```

⟨*function* `srvname` 758c⟩

⟨*method srvopen* 758d⟩

⟨*method srvcreate* 759⟩

⟨*method srvremove* 760⟩

⟨*method srvwstat* 761⟩

⟨*method srvclose* 762a⟩

⟨*method srvread* 762b⟩

⟨*method srvwrite* 762c⟩

⟨*global* `srvdevtab` 763a⟩

Uses `devattach()` 297b, `devstat()` 431, and `devwalk()` 429b.

E.10 devices/storage/

devices/storage/sd.h

⟨*macro* `sdfree`*(arm)* 764a⟩≡ (974)

```
#define sdfree(p) free(p)
```

⟨*macro* `sdmalloc`*(arm)* 764b⟩≡ (974)

```
#define sdmalloc(n) mallocalign(n, BLOCKALIGN, 0, 0)
```

⟨`devices/storage/sd.h` 764c⟩≡

```
/*
 * Storage Device.
 */

struct SDperm {
  char* name;
  char* user;
  ulong perm;
};

struct SDpart {
  uvlong  start;
  uvlong  end;
  SDperm;
  int valid;
  ulong vers;
```

```
};

struct SDunit {
  SDev* dev;
  int subno;
  uchar inquiry[255];   /* format follows SCSI spec */
  uchar sense[18];    /* format follows SCSI spec */
  SDperm;

  QLock ctl;
  uvlong  sectors;
  ulong secsize;
  SDpart* part;     /* nil or array of size npart */
  int npart;
  ulong vers;
  SDperm  ctlperm;

  QLock raw;      /* raw read or write in progress */
  ulong rawinuse;   /* really just a test-and-set */
  int state;
  SDreq*  req;
  SDperm  rawperm;
};

/*
 * Each controller is represented by a SDev.
 */
struct SDev {
  Ref r;      /* Number of callers using device */
  SDifc*  ifc;      /* pnp/legacy */
  void* ctlr;
  int idno;
  char  name[8];
  SDev* next;

  QLock;        /* enable/disable */
  int enabled;
  int nunit;      /* Number of units */
  QLock unitlock;   /* 'Loading' of units */
  int*  unitflg;    /* Unit flags */
  SDunit**unit;
};

struct SDifc {
  char* name;

  SDev* (*pnp)(void);
  SDev* (*legacy)(int, int);
  int (*enable)(SDev*);
  int (*disable)(SDev*);

  int (*verify)(SDunit*);
  int (*online)(SDunit*);
  int (*rio)(SDreq*);
  int (*rctl)(SDunit*, char*, int);
  int (*wctl)(SDunit*, Cmdbuf*);

  long  (*bio)(SDunit*, int, int, void*, long, uvlong);
  SDev* (*probe)(DevConf*);
  void  (*clear)(SDev*);
```

```
  char* (*rtopctl)(SDev*, char*, char*);
  int (*wtopctl)(SDev*, Cmdbuf*);
};

struct SDreq {
  SDunit* unit;
  int lun;
  int write;
  uchar cmd[16];
  int clen;
  void* data;
  int dlen;

  int flags;

  int status;
  long  rlen;
  uchar sense[256];
};

enum {
  SDnosense = 0x00000001,
  SDvalidsense  = 0x00010000,

  SDinq0periphqual= 0xe0,
  SDinq0periphtype= 0x1f,
  SDinq1removable = 0x80,

  /* periphtype values */
  SDperdisk = 0,  /* Direct access (disk) */
  SDpertape = 1,  /* Sequential eg, tape */
  SDperpr   = 2,  /* Printer */
  SDperworm = 4,  /* Worm */
  SDpercd   = 5,  /* CD-ROM */
  SDpermo   = 7,  /* rewriteable MO */
  SDperjuke = 8,  /* medium-changer */
};

enum {
  SDretry   = -5,   /* internal to controllers */
  SDmalloc  = -4,
  SDeio   = -3,
  SDtimeout = -2,
  SDnostatus  = -1,

  SDok    = 0,

  SDcheck   = 0x02,   /* check condition */
  SDbusy    = 0x08,   /* busy */

  SDmaxio   = 2048*1024,
  SDnpart   = 16,
};

/*
 * Allow the default #defines for sdmalloc & sdfree to be overridden by
 * system-specific versions.  This can be used to avoid extra copying
 * by making sure sd buffers are cache-aligned (some ARM systems) or
 * page-aligned (xen) for DMA.
 */
```

```
#ifdef sdmalloc
#else
#define sdmalloc(n) malloc(n)
#define sdfree(p) free(p)
#endif

/*
 * mmc/sd/sdio host controller interface
 */
struct SDio {
 char *name;
 int (*init)(void);
 void (*enable)(void);
 int (*inquiry)(char*, int);
 int (*cmd)(u32int, u32int, u32int*);
 void (*iosetup)(int, void*, int, int);
 void (*io)(int, uchar*, int);
};

extern SDio sdio;

/* sdscsi.c */
extern int scsiverify(SDunit*);
extern int scsionline(SDunit*);
extern long scsibio(SDunit*, int, int, void*, long, uvlong);
//extern SDev* scsiid(SDev*, SDifc*);
```

devices/storage/devsd.c

⟨*devsd.c forward decl* 767a⟩≡ (768)

```
extern void sdaddpart(SDunit*, char*, uvlong, uvlong);
extern void sdadddevs(SDev*);
```

⟨*global* sddevtab 767b⟩≡ (768)

```
Dev sddevtab = {
    .dc       =    'S',
    .name     =    "sd",

    .reset    =    sdreset,
    .init     =    devinit,
    .shutdown =    devshutdown,
    .attach   =    sdattach,
    .walk     =    sdwalk,
    .stat     =    sdstat,
    .open     =    sdopen,
    .create   =    devcreate,
    .close    =    sdclose,
    .read     =    sdread,
    .bread    =    devbread,
    .write    =    sdwrite,
    .bwrite   =    devbwrite,
    .remove   =    devremove,
    .wstat    =    sdwstat,

//    .power    = devpower,
//    .config   = sdconfig,   /* probe; only called for pcmcia-like devices */
};
```

Uses devbread() 434a, devbwrite() 434b, devcreate() 810, devinit() 810, devremove() 810, devshutdown() 810, sdattach() 768, sdclose() 768, sdopen() 768, sdread() 768, sdreset() 768, sdstat() 768, sdwalk() 768, sdwrite() 768, and sdwstat() 768.

⟨devices/storage/devsd.c 768⟩≡

```
/*
 * Storage Device.
 */
```

⟨*kernel basic includes* 49a⟩

```
#include "io.h"
#include "../port/sd.h"

#include <ureg.h>

extern Dev sddevtab;
extern SDifc* sdifc[];

static char devletters[] = "0123456789"
    "abcdefghijklmnopqrstuvwxyz"
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

static SDev *devs[sizeof devletters-1];
static QLock devslock;
```

⟨*devsd.c forward decl* 767a⟩

```
enum {
    Rawcmd,
    Rawdata,
    Rawstatus,
};

enum {
    Qtopdir     = 1,        /* top level directory */
    Qtopbase,
    Qtopctl      = Qtopbase,

    Qunitdir,           /* directory per unit */
    Qunitbase,
    Qctl        = Qunitbase,
    Qraw,
    Qpart,

    TypeLOG     = 4,
    NType       = (1<<TypeLOG),
    TypeMASK    = (NType-1),
    TypeSHIFT   = 0,

    PartLOG     = 8,
    NPart       = (1<<PartLOG),
    PartMASK    = (NPart-1),
    PartSHIFT   = TypeLOG,

    UnitLOG     = 8,
    NUnit       = (1<<UnitLOG),
    UnitMASK    = (NUnit-1),
    UnitSHIFT   = (PartLOG+TypeLOG),

    DevLOG      = 8,
```

```
    NDev        = (1 << DevLOG),
    DevMASK     = (NDev-1),
    DevSHIFT     = (UnitLOG+PartLOG+TypeLOG),

    Ncmd = 20,
};

#define TYPE(q)     ((((ulong)(q).path)>>TypeSHIFT) & TypeMASK)
#define PART(q)     ((((ulong)(q).path)>>PartSHIFT) & PartMASK)
#define UNIT(q)     ((((ulong)(q).path)>>UnitSHIFT) & UnitMASK)
#define DEV(q)      ((((ulong)(q).path)>>DevSHIFT) & DevMASK)
#define QID(d,u, p, t)  (((d)<<DevSHIFT)|((u)<<UnitSHIFT)|\
                     ((p)<<PartSHIFT)|((t)<<TypeSHIFT))


void
sdaddpart(SDunit* unit, char* name, uvlong start, uvlong end)
{
    SDpart *pp;
    int i, partno;

    /*
     * Check name not already used
     * and look for a free slot.
     */
    if(unit->part != nil){
        partno = -1;
        for(i = 0; i < unit->npart; i++){
            pp = &unit->part[i];
            if(!pp->valid){
                if(partno == -1)
                    partno = i;
                break;
            }
            if(strcmp(name, pp->name) == 0){
                if(pp->start == start && pp->end == end)
                    return;
                error(Ebadctl);
            }
        }
    }
    else{
        if((unit->part = malloc(sizeof(SDpart)*SDnpart)) == nil)
            error(Enomem);
        unit->npart = SDnpart;
        partno = 0;
    }

    /*
     * If no free slot found then increase the
     * array size (can't get here with unit->part == nil).
     */
    if(partno == -1){
        if(unit->npart >= NPart)
            error(Enomem);
        if((pp = malloc(sizeof(SDpart)*(unit->npart+SDnpart))) == nil)
            error(Enomem);
        memmove(pp, unit->part, sizeof(SDpart)*unit->npart);
        free(unit->part);
        unit->part = pp;
```

```
        partno = unit->npart;
        unit->npart += SDnpart;
    }

    /*
     * Check size and extent are valid.
     */
    if(start > end || end > unit->sectors)
        error(Eio);
    pp = &unit->part[partno];
    pp->start = start;
    pp->end = end;
    kstrdup(&pp->name, name);
    kstrdup(&pp->user, eve);
    pp->perm = 0640;
    pp->valid = 1;
}

static void
sddelpart(SDunit* unit, char* name)
{
    int i;
    SDpart *pp;

    /*
     * Look for the partition to delete.
     * Can't delete if someone still has it open.
     */
    pp = unit->part;
    for(i = 0; i < unit->npart; i++){
        if(strcmp(name, pp->name) == 0)
            break;
        pp++;
    }
    if(i >= unit->npart)
        error(Ebadctl);
    if(strcmp(up->user, pp->user) && !iseve())
        error(Eperm);
    pp->valid = 0;
    pp->vers++;
}

static void
sdincvers(SDunit *unit)
{
    int i;

    unit->vers++;
    if(unit->part){
        for(i = 0; i < unit->npart; i++){
            unit->part[i].valid = 0;
            unit->part[i].vers++;
        }
    }
}

static int
sdinitpart(SDunit* unit)
{
    int nf;
```

```
    uvlong start, end;
    char *f[4], *p, *q, buf[10];

    if(unit->sectors > 0){
        unit->sectors = unit->secsize = 0;
        sdincvers(unit);
    }

    /* device must be connected or not; other values are trouble */
    if(unit->inquiry[0] & 0xC0) /* see SDinq0periphqual */
        return 0;
    switch(unit->inquiry[0] & SDinq0periphtype){
    case SDperdisk:
    case SDperworm:
    case SDpercd:
    case SDpermo:
        break;
    default:
        return 0;
    }

    if(unit->dev->ifc->online)
        unit->dev->ifc->online(unit);
    if(unit->sectors){
        sdincvers(unit);
        sdaddpart(unit, "data", 0, unit->sectors);

        /*
         * Use partitions passed from boot program,
         * e.g.
         *  sdC0part=dos 63 123123/plan9 123123 456456
         * This happens before /boot sets hostname so the
         * partitions will have the null-string for user.
         * The gen functions patch it up.
         */
        snprint(buf, sizeof buf, "%spart", unit->name);
        for(p = getconf(buf); p != nil; p = q){
            if(q = strchr(p, '/'))
                *q++ = '\0';
            nf = tokenize(p, f, nelem(f));
            if(nf < 3)
                continue;

            start = strtoull(f[1], 0, 0);
            end = strtoull(f[2], 0, 0);
            if(!waserror()){
                sdaddpart(unit, f[0], start, end);
                poperror();
            }
        }
    }

    return 1;
}

static int
sdindex(int idno)
{
    char *p;
```

```
    p = strchr(devletters, idno);
    if(p == nil)
        return -1;
    return p-devletters;
}

static SDev*
sdgetdev(int idno)
{
    SDev *sdev;
    int i;

    if((i = sdindex(idno)) < 0)
        return nil;

    qlock(&devslock);
    if(sdev = devs[i])
        incref(&sdev->r);
    qunlock(&devslock);
    return sdev;
}

static SDunit*
sdgetunit(SDev* sdev, int subno)
{
    SDunit *unit;
    char buf[32];

    /*
     * Associate a unit with a given device and sub-unit
     * number on that device.
     * The device will be probed if it has not already been
     * successfully accessed.
     */
    qlock(&sdev->unitlock);
    if(subno > sdev->nunit){
        qunlock(&sdev->unitlock);
        return nil;
    }

    unit = sdev->unit[subno];
    if(unit == nil){
        /*
         * Probe the unit only once. This decision
         * may be a little severe and reviewed later.
         */
        if(sdev->unitflg[subno]){
            qunlock(&sdev->unitlock);
            return nil;
        }
        if((unit = malloc(sizeof(SDunit))) == nil){
            qunlock(&sdev->unitlock);
            return nil;
        }
        sdev->unitflg[subno] = 1;

        snprint(buf, sizeof(buf), "%s%d", sdev->name, subno);
        kstrdup(&unit->name, buf);
        kstrdup(&unit->user, eve);
        unit->perm = 0555;
```

```
        unit->subno = subno;
        unit->dev = sdev;

        if(sdev->enabled == 0 && sdev->ifc->enable)
            sdev->ifc->enable(sdev);
        sdev->enabled = 1;

        /*
         * No need to lock anything here as this is only
         * called before the unit is made available in the
         * sdunit[] array.
         */
        if(unit->dev->ifc->verify(unit) == 0){
            qunlock(&sdev->unitlock);
            free(unit);
            return nil;
        }
        sdev->unit[subno] = unit;
    }
    qunlock(&sdev->unitlock);
    return unit;
}

static void
sdreset(void)
{
    int i;
    SDev *sdev;

    /*
     * Probe all known controller types and register any devices found.
     */
    for(i = 0; sdifc[i] != nil; i++){
        if(sdifc[i]->pnp == nil || (sdev = sdifc[i]->pnp()) == nil)
            continue;
        sdadddevs(sdev);
    }
}

void
sdadddevs(SDev *sdev)
{
    int i, j, id;
    SDev *next;

    for(; sdev; sdev=next){
        next = sdev->next;

        sdev->unit = (SDunit**)malloc(sdev->nunit * sizeof(SDunit*));
        sdev->unitflg = (int*)malloc(sdev->nunit * sizeof(int));
        if(sdev->unit == nil || sdev->unitflg == nil){
            print("sdadddevs: out of memory\n");
        giveup:
            free(sdev->unit);
            free(sdev->unitflg);
            if(sdev->ifc->clear)
                sdev->ifc->clear(sdev);
            free(sdev);
            continue;
        }
```

```
        id = sdindex(sdev->idno);
        if(id == -1){
            print("sdadddevs: bad id number %d (%C)\n", id, id);
            goto giveup;
        }
        qlock(&devslock);
        for(i=0; i<nelem(devs); i++){
            if(devs[j = (id+i)%nelem(devs)] == nil){
                sdev->idno = devletters[j];
                devs[j] = sdev;
                snprint(sdev->name, sizeof sdev->name, "sd%c", devletters[j]);
                break;
            }
        }
        qunlock(&devslock);
        if(i == nelem(devs)){
            print("sdadddevs: out of device letters\n");
            goto giveup;
        }
    }
}

static int
sd2gen(Chan* c, int i, DirEntry* dp)
{
    Qid q;
    uvlong l;
    SDpart *pp;
    SDperm *perm;
    SDunit *unit;
    SDev *sdev;
    int rv;

    sdev = sdgetdev(DEV(c->qid));
    assert(sdev);
    unit = sdev->unit[UNIT(c->qid)];

    rv = -1;
    switch(i){
    case Qctl:
        mkqid(&q, QID(DEV(c->qid), UNIT(c->qid), PART(c->qid), Qctl),
            unit->vers, QTFILE);
        perm = &unit->ctlperm;
        if(emptystr(perm->user)){
            kstrdup(&perm->user, eve);
            perm->perm = 0644;  /* nothing secret in ctl */
        }
        devdir(c, q, "ctl", 0, perm->user, perm->perm, dp);
        rv = 1;
        break;

    case Qraw:
        mkqid(&q, QID(DEV(c->qid), UNIT(c->qid), PART(c->qid), Qraw),
            unit->vers, QTFILE);
        perm = &unit->rawperm;
        if(emptystr(perm->user)){
            kstrdup(&perm->user, eve);
            perm->perm = DMEXCL|0600;
        }
        devdir(c, q, "raw", 0, perm->user, perm->perm, dp);
```

```
        rv = 1;
        break;

    case Qpart:
        pp = &unit->part[PART(c->qid)];
        l = (pp->end - pp->start) * unit->secsize;
        mkqid(&q, QID(DEV(c->qid), UNIT(c->qid), PART(c->qid), Qpart),
            unit->vers+pp->vers, QTFILE);
        if(emptystr(pp->user))
            kstrdup(&pp->user, eve);
        devdir(c, q, pp->name, l, pp->user, pp->perm, dp);
        rv = 1;
        break;
    }

    decref(&sdev->r);
    return rv;
}

static int
sd1gen(Chan* c, int i, DirEntry* dp)
{
    Qid q;

    switch(i){
    case Qtopctl:
        mkqid(&q, QID(0, 0, 0, Qtopctl), 0, QTFILE);
        devdir(c, q, "sdctl", 0, eve, 0644, dp);    /* no secrets */
        return 1;
    }
    return -1;
}

static int
sdgen(Chan* c, char*, Dirtab*, int, int s, DirEntry* dp)
{
    Qid q;
    uvlong l;
    int i, r;
    SDpart *pp;
    SDunit *unit;
    SDev *sdev;

    switch(TYPE(c->qid)){
    case Qtopdir:
        if(s == DEVDOTDOT){
            mkqid(&q, QID(0, 0, 0, Qtopdir), 0, QTDIR);
            snprint(up->genbuf, sizeof up->genbuf, "#%C",
                sddevtab.dc);
            devdir(c, q, up->genbuf, 0, eve, 0555, dp);
            return 1;
        }

        if(s+Qtopbase < Qunitdir)
            return sd1gen(c, s+Qtopbase, dp);
        s -= (Qunitdir-Qtopbase);

        qlock(&devslock);
        for(i=0; i<nelem(devs); i++){
            if(devs[i]){
```

```
            if(s < devs[i]->nunit)
                break;
            s -= devs[i]->nunit;
        }
    }

    if(i == nelem(devs)){
        /* Run off the end of the list */
        qunlock(&devslock);
        return -1;
    }

    if((sdev = devs[i]) == nil){
        qunlock(&devslock);
        return 0;
    }

    incref(&sdev->r);
    qunlock(&devslock);

    if((unit = sdev->unit[s]) == nil)
        if((unit = sdgetunit(sdev, s)) == nil){
            decref(&sdev->r);
            return 0;
        }

    mkqid(&q, QID(sdev->idno, s, 0, Qunitdir), 0, QTDIR);
    if(emptystr(unit->user))
        kstrdup(&unit->user, eve);
    devdir(c, q, unit->name, 0, unit->user, unit->perm, dp);
    decref(&sdev->r);
    return 1;

case Qunitdir:
    if(s == DEVDOTDOT){
        mkqid(&q, QID(0, 0, 0, Qtopdir), 0, QTDIR);
        snprint(up->genbuf, sizeof up->genbuf, "#%C",
            sddevtab.dc);
        devdir(c, q, up->genbuf, 0, eve, 0555, dp);
        return 1;
    }

    if((sdev = sdgetdev(DEV(c->qid))) == nil){
        devdir(c, c->qid, "unavailable", 0, eve, 0, dp);
        return 1;
    }

    unit = sdev->unit[UNIT(c->qid)];
    qlock(&unit->ctl);

    /*
     * Check for media change.
     * If one has already been detected, sectors will be zero.
     * If there is one waiting to be detected, online
     * will return > 1.
     * Online is a bit of a large hammer but does the job.
     */
    if(unit->sectors == 0
    || (unit->dev->ifc->online && unit->dev->ifc->online(unit) > 1))
        sdinitpart(unit);
```

```
        i = s+Qunitbase;
        if(i < Qpart){
            r = sd2gen(c, i, dp);
            qunlock(&unit->ctl);
            decref(&sdev->r);
            return r;
        }
        i -= Qpart;
        if(unit->part == nil || i >= unit->npart){
            qunlock(&unit->ctl);
            decref(&sdev->r);
            break;
        }
        pp = &unit->part[i];
        if(!pp->valid){
            qunlock(&unit->ctl);
            decref(&sdev->r);
            return 0;
        }
        l = (pp->end - pp->start) * unit->secsize;
        mkqid(&q, QID(DEV(c->qid), UNIT(c->qid), i, Qpart),
            unit->vers+pp->vers, QTFILE);
        if(emptystr(pp->user))
            kstrdup(&pp->user, eve);
        devdir(c, q, pp->name, l, pp->user, pp->perm, dp);
        qunlock(&unit->ctl);
        decref(&sdev->r);
        return 1;
    case Qraw:
    case Qctl:
    case Qpart:
        if((sdev = sdgetdev(DEV(c->qid))) == nil){
            devdir(c, q, "unavailable", 0, eve, 0, dp);
            return 1;
        }
        unit = sdev->unit[UNIT(c->qid)];
        qlock(&unit->ctl);
        r = sd2gen(c, TYPE(c->qid), dp);
        qunlock(&unit->ctl);
        decref(&sdev->r);
        return r;
    case Qtopctl:
        return sd1gen(c, TYPE(c->qid), dp);
    default:
        break;
    }

    return -1;
}

static Chan*
sdattach(char* spec)
{
    Chan *c;
    char *p;
    SDev *sdev;
    int idno, subno;

    if(*spec == '\0'){
```

```
        c = devattach(sddevtab.dc, spec);
        mkqid(&c->qid, QID(0, 0, 0, Qtopdir), 0, QTDIR);
        return c;
    }

    if(spec[0] != 's' || spec[1] != 'd')
        error(Ebadspec);
    idno = spec[2];
    subno = strtol(&spec[3], &p, 0);
    if(p == &spec[3])
        error(Ebadspec);

    if((sdev=sdgetdev(idno)) == nil)
        error(Enonexist);
    if(sdgetunit(sdev, subno) == nil){
        decref(&sdev->r);
        error(Enonexist);
    }

    c = devattach(sddevtab.dc, spec);
    mkqid(&c->qid, QID(sdev->idno, subno, 0, Qunitdir), 0, QTDIR);
    c->dev = (sdev->idno << UnitLOG) + subno;
    decref(&sdev->r);
    return c;
}

static Walkqid*
sdwalk(Chan* c, Chan* nc, char** name, int nname)
{
    return devwalk(c, nc, name, nname, nil, 0, sdgen);
}

static int
sdstat(Chan* c, uchar* db, int n)
{
    return devstat(c, db, n, nil, 0, sdgen);
}

static Chan*
sdopen(Chan* c, int omode)
{
    SDpart *pp;
    SDunit *unit;
    SDev *sdev;
    uchar tp;

    c = devopen(c, omode, 0, 0, sdgen);
    if((tp = TYPE(c->qid)) != Qctl && tp != Qraw && tp != Qpart)
        return c;

    sdev = sdgetdev(DEV(c->qid));
    if(sdev == nil)
        error(Enonexist);

    unit = sdev->unit[UNIT(c->qid)];

    switch(TYPE(c->qid)){
    case Qctl:
        c->qid.vers = unit->vers;
        break;
```

```
    case Qraw:
        c->qid.vers = unit->vers;
        if(arch_tas(&unit->rawinuse) != 0){
            c->flag &= ~COPEN;
            decref(&sdev->r);
            error(Einuse);
        }
        unit->state = Rawcmd;
        break;
    case Qpart:
        qlock(&unit->ctl);
        if(waserror()){
            qunlock(&unit->ctl);
            c->flag &= ~COPEN;
            decref(&sdev->r);
            nexterror();
        }
        pp = &unit->part[PART(c->qid)];
        c->qid.vers = unit->vers+pp->vers;
        qunlock(&unit->ctl);
        poperror();
        break;
    }
    decref(&sdev->r);
    return c;
}

static void
sdclose(Chan* c)
{
    SDunit *unit;
    SDev *sdev;

    if(c->qid.type & QTDIR)
        return;
    if(!(c->flag & COPEN))
        return;

    switch(TYPE(c->qid)){
    default:
        break;
    case Qraw:
        sdev = sdgetdev(DEV(c->qid));
        if(sdev){
            unit = sdev->unit[UNIT(c->qid)];
            unit->rawinuse = 0;
            decref(&sdev->r);
        }
        break;
    }
}

static long
sdbio(Chan* c, int write, char* a, long len, uvlong off)
{
    int nchange;
    long l;
    uchar *b;
    SDpart *pp;
    SDunit *unit;
```

```
SDev *sdev;
ulong max, nb, offset;
uvlong bno;

sdev = sdgetdev(DEV(c->qid));
if(sdev == nil){
    decref(&sdev->r);
    error(Enonexist);
}
unit = sdev->unit[UNIT(c->qid)];
if(unit == nil)
    error(Enonexist);

nchange = 0;
qlock(&unit->ctl);
while(waserror()){
    /* notification of media change; go around again */
    if(strcmp(up->errstr, Eio) == 0 && unit->sectors == 0 && nchange++ == 0){
        sdinitpart(unit);
        continue;
    }

    /* other errors; give up */
    qunlock(&unit->ctl);
    decref(&sdev->r);
    nexterror();
}
pp = &unit->part[PART(c->qid)];
if(unit->vers+pp->vers != c->qid.vers)
    error(Echange);

/*
 * Check the request is within bounds.
 * Removeable drives are locked throughout the I/O
 * in case the media changes unexpectedly.
 * Non-removeable drives are not locked during the I/O
 * to allow the hardware to optimise if it can; this is
 * a little fast and loose.
 * It's assumed that non-removeable media parameters
 * (sectors, secsize) can't change once the drive has
 * been brought online.
 */
bno = (off/unit->secsize) + pp->start;
nb = ((off+len+unit->secsize-1)/unit->secsize) + pp->start - bno;
max = SDmaxio/unit->secsize;
if(nb > max)
    nb = max;
if(bno+nb > pp->end)
    nb = pp->end - bno;
if(bno >= pp->end || nb == 0){
    if(write)
        error(Eio);
    qunlock(&unit->ctl);
    decref(&sdev->r);
    poperror();
    return 0;
}
if(!(unit->inquiry[1] & SDinq1removable)){
    qunlock(&unit->ctl);
    poperror();
```

```
    }

    b = sdmalloc(nb*unit->secsize);
    if(b == nil)
        error(Enomem);
    if(waserror()){
        sdfree(b);
        if(!(unit->inquiry[1] & SDinq1removable))
            decref(&sdev->r);       /* gadverdamme! */
        nexterror();
    }

    offset = off%unit->secsize;
    if(offset+len > nb*unit->secsize)
        len = nb*unit->secsize - offset;
    if(write){
        if(offset || (len%unit->secsize)){
            l = unit->dev->ifc->bio(unit, 0, 0, b, nb, bno);
            if(l < 0)
                error(Eio);
            if(l < (nb*unit->secsize)){
                nb = l/unit->secsize;
                l = nb*unit->secsize - offset;
                if(len > l)
                    len = l;
            }
        }
        memmove(b+offset, a, len);
        l = unit->dev->ifc->bio(unit, 0, 1, b, nb, bno);
        if(l < 0)
            error(Eio);
        if(l < offset)
            len = 0;
        else if(len > l - offset)
            len = l - offset;
    }
    else{
        l = unit->dev->ifc->bio(unit, 0, 0, b, nb, bno);
        if(l < 0)
            error(Eio);
        if(l < offset)
            len = 0;
        else if(len > l - offset)
            len = l - offset;
        memmove(a, b+offset, len);
    }
    sdfree(b);
    poperror();

    if(unit->inquiry[1] & SDinq1removable){
        qunlock(&unit->ctl);
        poperror();
    }

    decref(&sdev->r);
    return len;
}

static long
sdrio(SDreq* r, void* a, long n)
```

```
{
    void *data;

    if(n >= SDmaxio || n < 0)
        error(Etoobig);

    data = nil;
    if(n){
        if((data = sdmalloc(n)) == nil)
            error(Enomem);
        if(r->write)
            memmove(data, a, n);
    }
    r->data = data;
    r->dlen = n;

    if(waserror()){
        sdfree(data);
        r->data = nil;
        nexterror();
    }

    if(r->unit->dev->ifc->rio(r) != SDok)
        error(Eio);

    if(!r->write && r->rlen > 0)
        memmove(a, data, r->rlen);
    sdfree(data);
    r->data = nil;
    poperror();

    return r->rlen;
}

static long
sdread(Chan *c, void *a, long n, vlong off)
{
    char *p, *e, *buf;
    SDpart *pp;
    SDunit *unit;
    SDev *sdev;
    ulong offset;
    int i, l, m, status;

    offset = off;
    switch(TYPE(c->qid)){
    default:
        error(Eperm);
    case Qtopctl:
        m = 64*1024;    /* room for register dumps */
        p = buf = malloc(m);
        if(p == nil)
            error(Enomem);
        e = p + m;
        qlock(&devslock);
        for(i = 0; i < nelem(devs); i++){
            sdev = devs[i];
            if(sdev && sdev->ifc->rtopctl)
                p = sdev->ifc->rtopctl(sdev, p, e);
        }
```

```
    qunlock(&devslock);
    n = readstr(off, a, n, buf);
    free(buf);
    return n;

case Qtopdir:
case Qunitdir:
    return devdirread(c, a, n, 0, 0, sdgen);

case Qctl:
    sdev = sdgetdev(DEV(c->qid));
    if(sdev == nil)
        error(Enonexist);

    unit = sdev->unit[UNIT(c->qid)];
    m = 16*1024;    /* room for register dumps */
    p = malloc(m);
    if(p == nil)
        error(Enomem);
    l = snprint(p, m, "inquiry %.48s\n",
        (char*)unit->inquiry+8);
    qlock(&unit->ctl);
    /*
     * If there's a device specific routine it must
     * provide all information pertaining to night geometry
     * and the garscadden trains.
     */
    if(unit->dev->ifc->rctl)
        l += unit->dev->ifc->rctl(unit, p+l, m-l);
    if(unit->sectors == 0)
        sdinitpart(unit);
    if(unit->sectors){
        if(unit->dev->ifc->rctl == nil)
            l += snprint(p+l, m-l,
                "geometry %llud %lud\n",
                unit->sectors, unit->secsize);
        pp = unit->part;
        for(i = 0; i < unit->npart; i++){
            if(pp->valid)
                l += snprint(p+l, m-l,
                    "part %s %llud %llud\n",
                    pp->name, pp->start, pp->end);
            pp++;
        }
    }
    qunlock(&unit->ctl);
    decref(&sdev->r);
    l = readstr(offset, a, n, p);
    free(p);
    return l;

case Qraw:
    sdev = sdgetdev(DEV(c->qid));
    if(sdev == nil)
        error(Enonexist);

    unit = sdev->unit[UNIT(c->qid)];
    qlock(&unit->raw);
    if(waserror()){
        qunlock(&unit->raw);
```

```
            decref(&sdev->r);
            nexterror();
        }
        if(unit->state == Rawdata){
            unit->state = Rawstatus;
            i = sdrio(unit->req, a, n);
        }
        else if(unit->state == Rawstatus){
            status = unit->req->status;
            unit->state = Rawcmd;
            free(unit->req);
            unit->req = nil;
            i = readnum(0, a, n, status, NUMSIZE);
        } else
            i = 0;
        qunlock(&unit->raw);
        decref(&sdev->r);
        poperror();
        return i;

    case Qpart:
        return sdbio(c, 0, a, n, off);
    }
}

static void legacytopctl(Cmdbuf*);

static long
sdwrite(Chan* c, void* a, long n, vlong off)
{
    char *f0;
    int i;
    uvlong end, start;
    Cmdbuf *cb;
    SDifc *ifc;
    SDreq *req;
    SDunit *unit;
    SDev *sdev;

    switch(TYPE(c->qid)){
    default:
        error(Eperm);
    case Qtopctl:
        cb = parsecmd(a, n);
        if(waserror()){
            free(cb);
            nexterror();
        }
        if(cb->nf == 0)
            error("empty control message");
        f0 = cb->f[0];
        cb->f++;
        cb->nf--;
        if(strcmp(f0, "config") == 0){
            /* wormhole into ugly legacy interface */
            legacytopctl(cb);
            poperror();
            free(cb);
            break;
        }
```

```
    /*
     * "ata arg..." invokes sdifc[i]->wtopctl(nil, cb),
     * where sdifc[i]->name=="ata" and cb contains the args.
     */
    ifc = nil;
    sdev = nil;
    for(i=0; sdifc[i]; i++){
        if(strcmp(sdifc[i]->name, f0) == 0){
            ifc = sdifc[i];
            sdev = nil;
            goto subtopctl;
        }
    }
    /*
     * "sd1 arg..." invokes sdifc[i]->wtopctl(sdev, cb),
     * where sdifc[i] and sdev match controller letter "1",
     * and cb contains the args.
     */
    if(f0[0]=='s' && f0[1]=='d' && f0[2] && f0[3] == 0){
        if((sdev = sdgetdev(f0[2])) != nil){
            ifc = sdev->ifc;
            goto subtopctl;
        }
    }
    error("unknown interface");

subtopctl:
    if(waserror()){
        if(sdev)
            decref(&sdev->r);
        nexterror();
    }
    if(ifc->wtopctl)
        ifc->wtopctl(sdev, cb);
    else
        error(Ebadctl);
    poperror();
    poperror();
    if (sdev)
        decref(&sdev->r);
    free(cb);
    break;

case Qctl:
    cb = parsecmd(a, n);
    sdev = sdgetdev(DEV(c->qid));
    if(sdev == nil)
        error(Enonexist);
    unit = sdev->unit[UNIT(c->qid)];

    qlock(&unit->ctl);
    if(waserror()){
        qunlock(&unit->ctl);
        decref(&sdev->r);
        free(cb);
        nexterror();
    }
    if(unit->vers != c->qid.vers)
        error(Echange);
```

```
    if(cb->nf < 1)
        error(Ebadctl);
    if(strcmp(cb->f[0], "part") == 0){
        if(cb->nf != 4)
            error(Ebadctl);
        if(unit->sectors == 0 && !sdinitpart(unit))
            error(Eio);
        start = strtoull(cb->f[2], 0, 0);
        end = strtoull(cb->f[3], 0, 0);
        sdaddpart(unit, cb->f[1], start, end);
    }
    else if(strcmp(cb->f[0], "delpart") == 0){
        if(cb->nf != 2 || unit->part == nil)
            error(Ebadctl);
        sddelpart(unit, cb->f[1]);
    }
    else if(unit->dev->ifc->wctl)
        unit->dev->ifc->wctl(unit, cb);
    else
        error(Ebadctl);
    qunlock(&unit->ctl);
    decref(&sdev->r);
    poperror();
    free(cb);
    break;

case Qraw:
    sdev = sdgetdev(DEV(c->qid));
    if(sdev == nil)
        error(Enonexist);
    unit = sdev->unit[UNIT(c->qid)];
    qlock(&unit->raw);
    if(waserror()){
        qunlock(&unit->raw);
        decref(&sdev->r);
        nexterror();
    }
    switch(unit->state){
    case Rawcmd:
        if(n < 6 || n > sizeof(req->cmd))
            error(Ebadarg);
        if((req = malloc(sizeof(SDreq))) == nil)
            error(Enomem);
        req->unit = unit;
        memmove(req->cmd, a, n);
        req->clen = n;
        req->flags = SDnosense;
        req->status = ~0;

        unit->req = req;
        unit->state = Rawdata;
        break;

    case Rawstatus:
        unit->state = Rawcmd;
        free(unit->req);
        unit->req = nil;
        error(Ebadusefd);

    case Rawdata:
```

```
            unit->state = Rawstatus;
            unit->req->write = 1;
            n = sdrio(unit->req, a, n);
        }
        qunlock(&unit->raw);
        decref(&sdev->r);
        poperror();
        break;
    case Qpart:
        return sdbio(c, 1, a, n, off);
    }

    return n;
}

static int
sdwstat(Chan* c, uchar* dp, int n)
{
    DirEntry *d;
    SDpart *pp;
    SDperm *perm;
    SDunit *unit;
    SDev *sdev;

    if(c->qid.type & QTDIR)
        error(Eperm);

    sdev = sdgetdev(DEV(c->qid));
    if(sdev == nil)
        error(Enonexist);
    unit = sdev->unit[UNIT(c->qid)];
    qlock(&unit->ctl);
    d = nil;
    if(waserror()){
        free(d);
        qunlock(&unit->ctl);
        decref(&sdev->r);
        nexterror();
    }

    switch(TYPE(c->qid)){
    default:
        error(Eperm);
    case Qctl:
        perm = &unit->ctlperm;
        break;
    case Qraw:
        perm = &unit->rawperm;
        break;
    case Qpart:
        pp = &unit->part[PART(c->qid)];
        if(unit->vers+pp->vers != c->qid.vers)
            error(Enonexist);
        perm = &pp->SDperm;
        break;
    }

    if(strcmp(up->user, perm->user) && !iseve())
        error(Eperm);
```

```
    d = smalloc(sizeof(DirEntry)+n);
    n = convM2D(dp, n, &d[0], (char*)&d[1]);
    if(n == 0)
        error(Eshortstat);
    if(!emptystr(d[0].uid))
        kstrdup(&perm->user, d[0].uid);
    if(d[0].mode != ~0UL)
        perm->perm = (perm->perm & ~0777) | (d[0].mode & 0777);

    free(d);
    qunlock(&unit->ctl);
    decref(&sdev->r);
    poperror();
    return n;
}

static int
configure(char* spec, DevConf* cf)
{
    SDev *s, *sdev;
    char *p;
    int i;

    if(sdindex(*spec) < 0)
        error("bad sd spec");

    if((p = strchr(cf->type, '/')) != nil)
        *p++ = '\0';

    for(i = 0; sdifc[i] != nil; i++)
        if(strcmp(sdifc[i]->name, cf->type) == 0)
            break;
    if(sdifc[i] == nil)
        error("sd type not found");
    if(p)
        *(p-1) = '/';

    if(sdifc[i]->probe == nil)
        error("sd type cannot probe");

    sdev = sdifc[i]->probe(cf);
    for(s=sdev; s; s=s->next)
        s->idno = *spec;
    sdadddevs(sdev);
    return 0;
}

static int
unconfigure(char* spec)
{
    int i;
    SDev *sdev;
    SDunit *unit;

    if((i = sdindex(*spec)) < 0)
        error(Enonexist);

    qlock(&devslock);
    if((sdev = devs[i]) == nil){
        qunlock(&devslock);
```

```
        error(Enonexist);
    }
    if(sdev->r.ref){
        qunlock(&devslock);
        error(Einuse);
    }
    devs[i] = nil;
    qunlock(&devslock);

    /* make sure no interrupts arrive anymore before removing resources */
    if(sdev->enabled && sdev->ifc->disable)
        sdev->ifc->disable(sdev);

    for(i = 0; i != sdev->nunit; i++){
        if(unit = sdev->unit[i]){
            free(unit->name);
            free(unit->user);
            free(unit);
        }
    }

    if(sdev->ifc->clear)
        sdev->ifc->clear(sdev);
    free(sdev);
    return 0;
}

static int
sdconfig(int on, char* spec, DevConf* cf)
{
    if(on)
        return configure(spec, cf);
    return unconfigure(spec);
}
```

⟨*global* sddevtab 767b⟩

```
/*
 * This is wrong for so many reasons.  This code must go.
 */
typedef struct Confdata Confdata;
struct Confdata {
    int on;
    char*   spec;
    DevConf cf;
};

static void
parseswitch(Confdata* cd, char* option)
{
    if(!strcmp("on", option))
        cd->on = 1;
    else if(!strcmp("off", option))
        cd->on = 0;
    else
        error(Ebadarg);
}

static void
parsespec(Confdata* cd, char* option)
```

```
{
    if(strlen(option) > 1)
        error(Ebadarg);
    cd->spec = option;
}

static Devport*
getnewport(DevConf* dc)
{
    Devport *p;

    p = (Devport *)malloc((dc->nports + 1) * sizeof(Devport));
    if(p == nil)
        error(Enomem);
    if(dc->nports > 0){
        memmove(p, dc->ports, dc->nports * sizeof(Devport));
        free(dc->ports);
    }
    dc->ports = p;
    p = &dc->ports[dc->nports++];
    p->size = -1;
    p->port = (ulong)-1;
    return p;
}

static void
parseport(Confdata* cd, char* option)
{
    char *e;
    Devport *p;

    if(cd->cf.nports == 0 || cd->cf.ports[cd->cf.nports-1].port != (ulong)-1)
        p = getnewport(&cd->cf);
    else
        p = &cd->cf.ports[cd->cf.nports-1];
    p->port = strtol(option, &e, 0);
    if(e == nil || *e != '\0')
        error(Ebadarg);
}

static void
parsesize(Confdata* cd, char* option)
{
    char *e;
    Devport *p;

    if(cd->cf.nports == 0 || cd->cf.ports[cd->cf.nports-1].size != -1)
        p = getnewport(&cd->cf);
    else
        p = &cd->cf.ports[cd->cf.nports-1];
    p->size = (int)strtol(option, &e, 0);
    if(e == nil || *e != '\0')
        error(Ebadarg);
}

static void
parseirq(Confdata* cd, char* option)
{
    char *e;
```

```
    cd->cf.intnum = strtoul(option, &e, 0);
    if(e == nil || *e != '\0')
        error(Ebadarg);
}

static void
parsetype(Confdata* cd, char* option)
{
    cd->cf.type = option;
}

static struct {
    char    *name;
    void    (*parse)(Confdata*, char*);
} options[] = {
    "switch",   parseswitch,
    "spec",     parsespec,
    "port",     parseport,
    "size",     parsesize,
    "irq",      parseirq,
    "type",     parsetype,
};

static void
legacytopctl(Cmdbuf *cb)
{
    char *opt;
    int i, j;
    Confdata cd;

    memset(&cd, 0, sizeof cd);
    cd.on = -1;
    for(i=0; i<cb->nf; i+=2){
        if(i+2 > cb->nf)
            error(Ebadarg);
        opt = cb->f[i];
        for(j=0; j<nelem(options); j++)
            if(strcmp(opt, options[j].name) == 0){
                options[j].parse(&cd, cb->f[i+1]);
                break;
            }
        if(j == nelem(options))
            error(Ebadarg);
    }
    /* this has been rewritten to accomodate sdaoe */
    if(cd.on < 0 || cd.spec == 0)
        error(Ebadarg);
    if(cd.on && cd.cf.type == nil)
        error(Ebadarg);
    sdconfig(cd.on, cd.spec, &cd.cf);
}
```

Uses COPEN 294b, Confdata 768, DEV-1612 768, DEVDOTDOT 805, DMEXCL, DevLOG-1604 768, Ebadarg 664d, Ebadspec 664d, Ebadusefd 664d, Echange 664d, Eshortstat 664d, Etoobig 664d, NDev-1605 768, NPart-1597 768, NType-1593 768, NUMSIZE 867a, NUnit-1601 768, PART-1610 768, PartLOG-1596 768, QID-1613 768, QTDIR 70b, QTFILE, Qctl-1589 768, Qpart-1591 768, Qraw-1590 768, Qtopbase-1585 768, Qtopctl-1586 768, Qtopdir-1584 768, Qunitbase-1588 768, Qunitdir-1587 768, Rawcmd-1581 768, Rawdata-1582 768, Rawstatus-1583 768, SDinq0periphtype 764c, SDinq1removable 764c, SDmaxio 764c, SDnosense 764c, SDnpart 764c, SDok 764c, SDpercd 764c, SDperdisk 764c, SDpermo 764c, SDperworm 764c, TYPE-1609 768, TypeLOG-1592 768, UNIT-1611 768, UnitLOG-1600 768, __anon_struct_197 768, assert 853f, configure() 768, decref() 87c, devattach() 297b, devdir() 427c, devdirread() 432c, devletters-1578 768, devs-1579 768, devslock-1580 768,

devstat() 431, devwalk() 429b, emptystr() 670a, eve 591a, free() 104a, getnewport() 768, incref() 87b, iseve() 591b, kstrdup() 669b, legacytopctl() 768, malloc() 103c, nelem 853i, nexterror, options-1614 768, parsecmd() 594c, parseirq() 768, parseport() 768, parsesize() 768, parsespec() 768, parseswitch() 768, parsetype() 768, poperror 666d, print, qlock() 81f, qunlock() 82c, readnum() 670c, readstr() 670d, sd1gen() 768, sd2gen() 768, sdadddevs() 768, sdaddpart() 768, sdbio() 768, sdconfig() 768, sddelpart() 768, sddevtab 767b, sdgen() 768, sdgetdev() 768, sdgetunit() 768, sdincvers() 768, sdindex() 768, sdinitpart() 768, sdrio() 768, smalloc() 104b, unconfigure() 768, up, and waserror 666d.

devices/storage/sdscsi.c

⟨devices/storage/sdscsi.c 792⟩≡
⟨*kernel basic includes* 49a⟩

```
#include "io.h"
#include "../port/sd.h"

#include <ureg.h>

static int
scsitest(SDreq* r)
{
    r->write = 0;
    memset(r->cmd, 0, sizeof(r->cmd));
    r->cmd[1] = r->lun<<5;
    r->clen = 6;
    r->data = nil;
    r->dlen = 0;
    r->flags = 0;

    r->status = ~0;

    return r->unit->dev->ifc->rio(r);
}

int
scsiverify(SDunit* unit)
{
    SDreq *r;
    int i, status;
    uchar *inquiry;

    if((r = malloc(sizeof(SDreq))) == nil)
        return 0;
    if((inquiry = sdmalloc(sizeof(unit->inquiry))) == nil){
        free(r);
        return 0;
    }
    r->unit = unit;
    r->lun = 0;     /* ??? */

    memset(unit->inquiry, 0, sizeof(unit->inquiry));
    r->write = 0;
    r->cmd[0] = 0x12;
    r->cmd[1] = r->lun<<5;
    r->cmd[4] = sizeof(unit->inquiry)-1;
    r->clen = 6;
    r->data = inquiry;
    r->dlen = sizeof(unit->inquiry)-1;
    r->flags = 0;
```

```
r->status = ~0;
if(unit->dev->ifc->rio(r) != SDok){
    free(r);
    return 0;
}
memmove(unit->inquiry, inquiry, r->dlen);
free(inquiry);

SET(status);
for(i = 0; i < 3; i++){
    while((status = scsitest(r)) == SDbusy)
        ;
    if(status == SDok || status != SDcheck)
        break;
    if(!(r->flags & SDvalidsense))
        break;
    if((r->sense[2] & 0x0F) != 0x02)
        continue;

    /*
     * Unit is 'not ready'.
     * If it is in the process of becoming ready or needs
     * an initialising command, set status so it will be spun-up
     * below.
     * If there's no medium, that's OK too, but don't
     * try to spin it up.
     */
    if(r->sense[12] == 0x04){
        if(r->sense[13] == 0x02 || r->sense[13] == 0x01){
            status = SDok;
            break;
        }
    }
    if(r->sense[12] == 0x3A)
        break;
}

if(status == SDok){
    /*
     * Try to ensure a direct-access device is spinning.
     * Don't wait for completion, ignore the result.
     */
    if((unit->inquiry[0] & SDinq0periphtype) == SDperdisk){
        memset(r->cmd, 0, sizeof(r->cmd));
        r->write = 0;
        r->cmd[0] = 0x1B;
        r->cmd[1] = (r->lun<<5)|0x01;
        r->cmd[4] = 1;
        r->clen = 6;
        r->data = nil;
        r->dlen = 0;
        r->flags = 0;

        r->status = ~0;
        unit->dev->ifc->rio(r);
    }
}
free(r);
```

```
    if(status == SDok || status == SDcheck)
        return 1;
    return 0;
}

static int
scsirio(SDreq* r)
{
    /*
     * Perform an I/O request, returning
     *  -1  failure
     *   0  ok
     *   1  no medium present
     *   2  retry
     * The contents of r may be altered so the
     * caller should re-initialise if necesary.
     */
    r->status = ~0;
    switch(r->unit->dev->ifc->rio(r)){
    default:
        break;
    case SDcheck:
        if(!(r->flags & SDvalidsense))
            break;
        switch(r->sense[2] & 0x0F){
        case 0x00:      /* no sense */
        case 0x01:      /* recovered error */
            return 2;
        case 0x06:      /* check condition */
            /*
             * 0x28 - not ready to ready transition,
             *    medium may have changed.
             * 0x29 - power on or some type of reset.
             */
            if(r->sense[12] == 0x28 && r->sense[13] == 0)
                return 2;
            if(r->sense[12] == 0x29)
                return 2;
            break;
        case 0x02:      /* not ready */
            /*
             * If no medium present, bail out.
             * If unit is becoming ready, rather than not
             * not ready, wait a little then poke it again.                    */
            if(r->sense[12] == 0x3A)
                break;
            if(r->sense[12] != 0x04 || r->sense[13] != 0x01)
                break;

            while(waserror())
                ;
            tsleep(&up->sleepr, returnfalse, 0, 500);
            poperror();
            scsitest(r);
            return 2;
        default:
            break;
        }
        break;
    case SDok:
```

```
        return 0;
    }
    return -1;
}

int
scsionline(SDunit* unit)
{
    SDreq *r;
    uchar *p;
    int ok, retries;

    if((r = malloc(sizeof(SDreq))) == nil)
        return 0;
    if((p = sdmalloc(8)) == nil){
        free(r);
        return 0;
    }

    ok = 0;

    r->unit = unit;
    r->lun = 0;             /* ??? */
    for(retries = 0; retries < 10; retries++){
        /*
         * Read-capacity is mandatory for DA, WORM, CD-ROM and
         * MO. It may return 'not ready' if type DA is not
         * spun up, type MO or type CD-ROM are not loaded or just
         * plain slow getting their act together after a reset.
         */
        r->write = 0;
        memset(r->cmd, 0, sizeof(r->cmd));
        r->cmd[0] = 0x25;
        r->cmd[1] = r->lun<<5;
        r->clen = 10;
        r->data = p;
        r->dlen = 8;
        r->flags = 0;

        r->status = ~0;
        switch(scsirio(r)){
        default:
            break;
        case 0:
            unit->sectors = (p[0]<<24)|(p[1]<<16)|(p[2]<<8)|p[3];
            unit->secsize = (p[4]<<24)|(p[5]<<16)|(p[6]<<8)|p[7];

            /*
             * Some ATAPI CD readers lie about the block size.
             * Since we don't read audio via this interface
             * it's okay to always fudge this.
             */
            if(unit->secsize == 2352)
                unit->secsize = 2048;
            /*
             * Devices with removable media may return 0 sectors
             * when they have empty media (e.g. sata dvd writers);
             * if so, keep the count zero.
             *
             * Read-capacity returns the LBA of the last sector,
```

```
             * therefore the number of sectors must be incremented.
             */
            if(unit->sectors != 0)
                unit->sectors++;
            ok = 1;
            break;
        case 1:
            ok = 1;
            break;
        case 2:
            continue;
        }
        break;
    }
    free(p);
    free(r);

    if(ok)
        return ok+retries;
    else
        return 0;
}

static void
scsifmt10(SDreq *r, int write, int lun, ulong nb, uvlong bno)
{
    uchar *c;

    c = r->cmd;
    if(write == 0)
        c[0] = 0x28;
    else
        c[0] = 0x2A;
    c[1] = lun<<5;
    c[2] = bno>>24;
    c[3] = bno>>16;
    c[4] = bno>>8;
    c[5] = bno;
    c[6] = 0;
    c[7] = nb>>8;
    c[8] = nb;
    c[9] = 0;

    r->clen = 10;
}

static void
scsifmt16(SDreq *r, int write, int lun, ulong nb, uvlong bno)
{
    uchar *c;

    c = r->cmd;
    if(write == 0)
        c[0] = 0x88;
    else
        c[0] = 0x8A;
    c[1] = lun<<5;      /* so wrong */
    c[2] = bno>>56;
    c[3] = bno>>48;
    c[4] = bno>>40;
```

```
    c[5] = bno>>32;
    c[6] = bno>>24;
    c[7] = bno>>16;
    c[8] = bno>>8;
    c[9] = bno;
    c[10] = nb>>24;
    c[11] = nb>>16;
    c[12] = nb>>8;
    c[13] = nb;
    c[14] = 0;
    c[15] = 0;

    r->clen = 16;
}

long
scsibio(SDunit* unit, int lun, int write, void* data, long nb, uvlong bno)
{
    SDreq *r;
    long rlen;

    if((r = malloc(sizeof(SDreq))) == nil)
        error(Enomem);
    r->unit = unit;
    r->lun = lun;
again:
    r->write = write;
    if(bno >= (1ULL<<32))
        scsifmt16(r, write, lun, nb, bno);
    else
        scsifmt10(r, write, lun, nb, bno);
    r->data = data;
    r->dlen = nb*unit->secsize;
    r->flags = 0;

    r->status = ~0;
    switch(scsirio(r)){
    default:
        rlen = -1;
        break;
    case 0:
        rlen = r->rlen;
        break;
    case 2:
        rlen = -1;
        if(!(r->flags & SDvalidsense))
            break;
        switch(r->sense[2] & 0x0F){
        default:
            break;
        case 0x01:      /* recovered error */
            print("%s: recovered error at sector %llud\n",
                unit->name, bno);
            rlen = r->rlen;
            break;
        case 0x06:      /* check condition */
            /*
             * Check for a removeable media change.
             * If so, mark it by zapping the geometry info
             * to force an online request.
```

```
             */
            if(r->sense[12] != 0x28 || r->sense[13] != 0)
                break;
            if(unit->inquiry[1] & SDinq1removable)
                unit->sectors = 0;
            break;
        case 0x02:      /* not ready */
            /*
             * If unit is becoming ready,
             * rather than not not ready, try again.
             */
            if(r->sense[12] == 0x04 && r->sense[13] == 0x01)
                goto again;
            break;
        }
        break;
    }
    free(r);

    return rlen;
}
```

Uses SDbusy 764c, SDcheck 764c, SDinq0periphtype 764c, SDinq1removable 764c, SDok 764c, SDperdisk 764c, SDvalidsense 764c, free() 104a, malloc() 103c, poperror 666d, print, returnfalse() 104c, scsifmt10() 792, scsifmt16() 792, scsirio() 792, scsitest() 792, tsleep, up, and waserror 666d.

devices/storage/sdmmc.c

⟨devices/storage/sdmmc.c 798⟩≡

```
/*
 * mmc / sd memory card
 *
 * Copyright © 2012 Richard Miller <r.miller@acm.org>
 *
 * Assumes only one card on the bus
 */

#include "u.h"
#include "../port/lib.h"
#include "../port/error.h"
#include "mem.h"
#include "dat.h"
#include "fns.h"
#include "io.h"

#include "../port/sd.h"

#define CSD(end, start) rbits(csd, start, (end)-(start)+1)

typedef struct Ctlr Ctlr;

enum {
        Inittimeout     = 15,
        Multiblock      = 1,

        /* Commands */
        GO_IDLE_STATE   = 0,
        ALL_SEND_CID    = 2,
        SEND_RELATIVE_ADDR= 3,
```

```
        SELECT_CARD     = 7,
        SD_SEND_IF_COND = 8,
        SEND_CSD        = 9,
        STOP_TRANSMISSION= 12,
        SEND_STATUS     = 13,
        SET_BLOCKLEN    = 16,
        READ_SINGLE_BLOCK= 17,
        READ_MULTIPLE_BLOCK= 18,
        WRITE_BLOCK     = 24,
        WRITE_MULTIPLE_BLOCK= 25,
        APP_CMD         = 55,   /* prefix for following app-specific commands */
        SET_BUS_WIDTH   = 6,
        SD_SEND_OP_COND = 41,

        /* Command arguments */
        /* SD_SEND_IF_COND */
        Voltage         = 1<<8,
        Checkpattern    = 0x42,

        /* SELECT_CARD */
        Rcashift        = 16,

        /* SD_SEND_OP_COND */
        Hcs     = 1<<30,        /* host supports SDHC & SDXC */
        Ccs     = 1<<30,        /* card is SDHC or SDXC */
        V3_3    = 3<<20,        /* 3.2-3.4 volts */

        /* SET_BUS_WIDTH */
        Width1  = 0<<0,
        Width4  = 2<<0,

        /* OCR (operating conditions register) */
        Powerup = 1<<31,
};

struct Ctlr {
        SDev    *dev;
        SDio    *io;
        /* SD card registers */
        u16int  rca;
        u32int  ocr;
        u32int  cid[4];
        u32int  csd[4];
};

extern SDifc sdmmcifc;
extern SDio sdio;

static uint
rbits(u32int *p, uint start, uint len)
{
        uint w, off, v;

        w   = start / 32;
        off = start % 32;
        if(off == 0)
                v = p[w];
        else
                v = p[w] >> off | p[w+1] << (32-off);
        if(len < 32)
```

```
                return v & ((1<<len) - 1);
        else
                return v;
}

static void
identify(SDunit *unit, u32int *csd)
{
        uint csize, mult;

        unit->secsize = 1 << CSD(83, 80);
        switch(CSD(127, 126)){
        case 0:                         /* CSD version 1 */
                csize = CSD(73, 62);
                mult = CSD(49, 47);
                unit->sectors = (csize+1) * (1<<(mult+2));
                break;
        case 1:                         /* CSD version 2 */
                csize = CSD(69, 48);
                unit->sectors = (csize+1) * 512LL*KiB / unit->secsize;
                break;
        }
        if(unit->secsize == 1024){
                unit->sectors <<= 1;
                unit->secsize = 512;
        }
}

static SDev*
mmcpnp(void)
{
        SDev *sdev;
        Ctlr *ctl;

        if(sdio.init() < 0)
                return nil;
        sdev = malloc(sizeof(SDev));
        if(sdev == nil)
                return nil;
        ctl = malloc(sizeof(Ctlr));
        if(ctl == nil){
                free(sdev);
                return nil;
        }
        sdev->idno = 'M';
        sdev->ifc = &sdmmcifc;
        sdev->nunit = 1;
        sdev->ctlr = ctl;
        ctl->dev = sdev;
        ctl->io = &sdio;
        return sdev;
}

static int
mmcverify(SDunit *unit)
{
        int n;
        Ctlr *ctl;

        ctl = unit->dev->ctlr;
```

```
        n = ctl->io->inquiry((char*)&unit->inquiry[8], sizeof(unit->inquiry)-8);
        if(n < 0)
                return 0;
        unit->inquiry[0] = SDperdisk;
        unit->inquiry[1] = SDinq1removable;
        unit->inquiry[4] = sizeof(unit->inquiry)-4;
        return 1;
}

static int
mmcenable(SDev* dev)
{
        Ctlr *ctl;

        ctl = dev->ctlr;
        ctl->io->enable();
        return 1;
}

static int
mmconline(SDunit *unit)
{
        int hcs, i;
        u32int r[4];
        Ctlr *ctl;
        SDio *io;

        ctl = unit->dev->ctlr;
        io = ctl->io;
        assert(unit->subno == 0);

        if(waserror()){
                unit->sectors = 0;
                return 0;
        }
        if(unit->sectors != 0){
                io->cmd(SEND_STATUS, ctl->rca<<Rcashift, r);
                poperror();
                return 1;
        }
        io->cmd(GO_IDLE_STATE, 0, r);
        hcs = 0;
        if(!waserror()){
                io->cmd(SD_SEND_IF_COND, Voltage|Checkpattern, r);
                if(r[0] == (Voltage|Checkpattern))      /* SD 2.0 or above */
                        hcs = Hcs;
                poperror();
        }
        for(i = 0; i < Inittimeout; i++){
                arch_delay(100);
                io->cmd(APP_CMD, 0, r);
                io->cmd(SD_SEND_OP_COND, hcs|V3_3, r);
                if(r[0] & Powerup)
                        break;
        }
        if(i == Inittimeout){
                print("sdmmc: card won't power up\n");
                poperror();
                return 2;
        }
```

```
        ctl->ocr = r[0];
        io->cmd(ALL_SEND_CID, 0, r);
        memmove(ctl->cid, r, sizeof ctl->cid);
        io->cmd(SEND_RELATIVE_ADDR, 0, r);
        ctl->rca = r[0]>>16;
        io->cmd(SEND_CSD, ctl->rca<<Rcashift, r);
        memmove(ctl->csd, r, sizeof ctl->csd);
        identify(unit, ctl->csd);
        io->cmd(SELECT_CARD, ctl->rca<<Rcashift, r);
        io->cmd(SET_BLOCKLEN, unit->secsize, r);
        io->cmd(APP_CMD, ctl->rca<<Rcashift, r);
        io->cmd(SET_BUS_WIDTH, Width4, r);
        poperror();
        return 1;
}

static int
mmcrctl(SDunit *unit, char *p, int l)
{
        Ctlr *ctl;
        int i, n;

        ctl = unit->dev->ctlr;
        assert(unit->subno == 0);
        if(unit->sectors == 0){
                mmconline(unit);
                if(unit->sectors == 0)
                        return 0;
        }
        n = snprint(p, l, "rca %4.4ux ocr %8.8ux\ncid ", ctl->rca, ctl->ocr);
        for(i = nelem(ctl->cid)-1; i >= 0; i--)
                n += snprint(p+n, l-n, "%8.8ux", ctl->cid[i]);
        n += snprint(p+n, l-n, " csd ");
        for(i = nelem(ctl->csd)-1; i >= 0; i--)
                n += snprint(p+n, l-n, "%8.8ux", ctl->csd[i]);
        n += snprint(p+n, l-n, "\ngeometry %llud %ld\n",
                unit->sectors, unit->secsize);
        return n;
}

static long
mmcbio(SDunit *unit, int lun, int write, void *data, long nb, uvlong bno)
{
        int len, tries;
        ulong b;
        u32int r[4];
        uchar *buf;
        Ctlr *ctl;
        SDio *io;

        USED(lun);
        ctl = unit->dev->ctlr;
        io = ctl->io;
        assert(unit->subno == 0);
        if(unit->sectors == 0)
                error("media change");
        buf = data;
        len = unit->secsize;
        if(Multiblock){
                b = bno;
```

```
                tries = 0;
                while(waserror())
                        if(++tries == 3)
                                nexterror();
                io->iosetup(write, buf, len, nb);
                if(waserror()){
                        io->cmd(STOP_TRANSMISSION, 0, r);
                        nexterror();
                }
                io->cmd(write? WRITE_MULTIPLE_BLOCK: READ_MULTIPLE_BLOCK,
                        ctl->ocr & Ccs? b: b * len, r);
                io->io(write, buf, nb * len);
                poperror();
                io->cmd(STOP_TRANSMISSION, 0, r);
                poperror();
                b += nb;
        }else{
                for(b = bno; b < bno + nb; b++){
                        io->iosetup(write, buf, len, 1);
                        io->cmd(write? WRITE_BLOCK : READ_SINGLE_BLOCK,
                                ctl->ocr & Ccs? b: b * len, r);
                        io->io(write, buf, len);
                        buf += len;
                }
        }
        return (b - bno) * len;
}

static int
mmcrio(SDreq*)
{
        return -1;
}

SDifc sdmmcifc = {
        .name   = "mmc",
        .pnp    = mmcpnp,
        .enable = mmcenable,
        .verify = mmcverify,
        .online = mmconline,
        .rctl   = mmcrctl,
        .bio    = mmcbio,
        .rio    = mmcrio,
};
```

E.11 files/

files/portdat_files.h

⟨*struct* Extent 803⟩≡ (804b)

```
struct Extent
{
  int bid;
  ulong start;
  int len;
  Page  *cache;
  Extent  *next;
```

```
};
```

⟨*struct* Mntcache 804a⟩≡ (804b)

```
struct Mntcache
{
  Qid qid;
  int dev;
  int type;
  QLock;
  Extent   *list;
  Mntcache *hash;
  Mntcache *prev;
  Mntcache *next;
};
```

⟨files/portdat_files.h 804b⟩≡

```
⟨enum blockflag 434c⟩

⟨struct Block 435a⟩

⟨function BLEN 435b⟩
⟨function BALLOC 435c⟩


⟨enum queuestate 399a⟩

// defined in qio.c
extern  uint  qiomaxatomic;

/*
 *  IO queues
 */
// was in qio.c
⟨struct Queue 399b⟩


// was in cache.c
⟨struct Extent 803⟩

// was in cache.c
⟨struct Mntcache 804a⟩


⟨struct Mount 345a⟩

⟨struct Mhead 344a⟩



#include <fcall.h>

// was in devmnt.c
⟨struct Mntrpc 582b⟩

⟨struct Mnt 580a⟩



⟨struct Path 324d⟩
```

⟨*enum* accessnamec 305b⟩

⟨*enum* channelflag 294b⟩

⟨*struct* Chan 68b⟩

⟨*struct* Evalue 597a⟩

⟨*struct* Egrp 597b⟩

```
// internals
```

⟨*struct* Walkqid 327c⟩

⟨*struct* Dev 71b⟩

```
// array<Dev>, it looks like an allocated array<ref<dev>> but
// it is really a static array put here to avoid backward deps on conf_devtab,
// and it is not really a <ref<dev>> because it's pointers to static
// structures (e.g. mousedevtab, vgadevtab, etc).
extern Dev** devtab;
```

⟨*struct* Dirtab 422a⟩

```
// used by with devmnt.c and sysfile.c
struct Bogus {
    Chan    *chan;
    Chan    *authchan;
    char    *spec;
    int flags;
};


//*********************************************************************
// Internal to files/
//*********************************************************************
```

⟨*constants tags* 580d⟩

```
// actually internal to devmnt.c and mnt.c
```
⟨*struct* Mntalloc 580b⟩
```
extern struct Mntalloc mntalloc;
```

files/portfns_files.h

⟨files/portfns_files.h 805⟩≡

```
// allocb.c
Block* allocb(int);
void   freeb(Block*);
Block* iallocb(int);
void   iallocsummary(void);
void   checkb(Block*, char*);

// cache.c
```

```
void   cinit(void);
void   copen(Chan*);
int    cread(Chan*, uchar*, int, vlong);
void   cwrite(Chan*, uchar*, int, vlong);
void   cupdate(Chan*, uchar*, int, vlong);

// qio.c
void   freeblist(Block*);
int    blocklen(Block*);
Queue* qopen(int, int, void (*)(void*), void*);
void   qhangup(Queue*, char*);
Block* qget(Queue*);
int    qdiscard(Queue*, int);
void   qreopen(Queue*);
Block* qremove(Queue*);
void   qputback(Queue*, Block*);
Block* packblock(Block*);
Queue* qbypass(void (*)(void*, Block*), void*);
int    pullblock(Block**, int);
Block* bl2mem(uchar*, Block*, int);
void   qnoblock(Queue*, bool);
int    qisclosed(Queue*);
int    qfull(Queue*);
int    qwindow(Queue*);
void   qsetlimit(Queue*, int);
int    qpass(Queue*, Block*);
void   qclose(Queue*);
int    qiwrite(Queue*, void*, int);
int    qproduce(Queue*, void*, int);
Block* padblock(Block*, int);
Block* pullupblock(Block*, int);
int    qpassnolim(Queue*, Block*);
Block* copyblock(Block*, int);
Block* qcopy(Queue*, int, ulong);
void   qflush(Queue*);
long   qbwrite(Queue*, Block*);
int    qconsume(Queue*, void*, int);
//Block*    mem2bl(uchar*, int);
Block* trimblock(Block*, int, int);
void   qaddlist(Queue*, Block*);
Block* concatblock(Block*);
void   ixsummary(void);
Block* pullupqueue(Queue*, int);
void   qfree(Queue*);
int    qwrite(Queue*, void*, int);
Block* adjustblock(Block*, int);
Block* qbread(Queue*, int);
long   qread(Queue*, void*, int);
int    qcanread(Queue*);
int    qlen(Queue*);

// chan.c
Chan*  cclone(Chan*);
void   putmhead(Mhead*);
int    eqchantdqid(Chan*, int, int, Qid, bool);
Mhead* newmhead(Chan*);
void   error_if_not_dir(Chan*);
char*  chanpath(Chan*);
int    emptystr(char*);
void   chandevinit(void);
```

```
void   chandevshutdown(void);
void   chandevreset(void);
int    eqqid(Qid, Qid);
int    cmount(Chan*, Chan*, int, char*);
void   cunmount(Chan*, Chan*);
bool   findmount(Chan**, Mhead**, int, int, Qid);
void   pathclose(Path*);
Path*  newpath(char*);
Chan*  newchan(void);
void   chanfree(Chan*);
//cchan_close => cclose
char*  validnamedup(char*, bool);
void   validname(char*, bool);
void   nameerror(char*, char*);
int    walk(Chan**, char**, int, int, int*);
void   ccloseq(Chan*);
Chan*  namec(char*, int, int, ulong);
//long    unionread(Chan*, void*, long);

// file.c
Chan*  fdtochan(int, int, bool, bool);
int    openmode(ulong);
void   fdclose(int, int);
void   validstat(uchar*, int);

// dev.c
void   mkqid(Qid*, vlong, ulong, int);
int    devno(Rune, bool);
void   devdir(Chan*, Qid, char*, vlong, char*, long, DirEntry*);
void   devinit(void);
void   devreset(void);
void   devshutdown(void);
Chan*  devclone(Chan*);
long   devdirread(Chan*, char*, long, Dirtab*, int, Devgen*);
void   devpermcheck(char*, ulong, int);
void   devcreate(Chan*, char*, int, ulong);
Block* devbread(Chan*, long, ulong);
long   devbwrite(Chan*, Block*, ulong);
void   devremove(Chan*);
int    devwstat(Chan*, uchar*, int);
#define DEVDOTDOT -1
Devgen    devgen;//TODO?
//void    devpower(int);
//int   devconfig(int, char *, DevConf *);//TODO? why fp no-deps?
int    devstat(Chan*, uchar*, int, Dirtab*, int, Devgen*);
Chan*  devopen(Chan*, int, Dirtab*, int, Devgen*);
Chan*  devattach(Rune, char*);
Walkqid* devwalk(Chan*, Chan*, char**, int, Dirtab*, int, Devgen*);

// env.c
void   closeegrp(Egrp*);

// mnt.c
void   muxclose(Mnt*);
void   mntfree(Mntrpc*);
//void    mntpntfree(Mnt*);

// sysfile.c
int    newfd(Chan*);
// many sysxxx functions (used in syscalls/ without requiring extern decl)
```

files/chan.c

⟨*chan.c forward decl* 808a⟩≡ (808c)
```
char*       skipslash(char*);
void closeproc(void*);
typedef struct Elemlist Elemlist;
```
Uses Elemlist 309b.

⟨*chan.c debugging macro* 808b⟩≡ (808c)
```
bool chandebug=false;        /* toggle it in sysnop if you want */
#define DBG if(chandebug)iprint
```
Uses chandebug 808b and false.

⟨files/chan.c 808c⟩≡
```
⟨kernel basic includes 49a⟩

⟨chan.c forward decl 808a⟩

⟨chan.c debugging macro 808b⟩

enum
{
⟨constant PATHSLOP 325a⟩
⟨constant PATHMSLOP 356b⟩
};

⟨struct Chanalloc 295b⟩
⟨global chanalloc 295d⟩

⟨struct Elemlist 309b⟩

⟨function chanpath 318b⟩

⟨function isdotdot 326b⟩

⟨function emptystr 670a⟩

⟨function chandevreset 286a⟩

⟨function chandevinit 289b⟩

⟨function chandevshutdown 628b⟩

⟨constructor newchan 295e⟩

⟨global npath 324e⟩

⟨constructor newpath 324f⟩

⟨function copypath 325b⟩

⟨destructor pathclose 325c⟩

⟨function fixdotdotname 327a⟩

⟨function uniquepath 325d⟩

⟨function addelem 326a⟩

⟨destructor chanfree 296⟩
```

⟨*function* `cclose` 304b⟩

⟨*struct* `Clunkq` 319c⟩
⟨*global* `clunkq` 319d⟩

⟨*function* `ccloseq` 320a⟩

⟨*function* `clunkwork` 320b⟩

⟨*function* `closeproc` 320c⟩

⟨*function* `uniquechan` 297a⟩

⟨*function* `eqqid` 70c⟩

⟨*function* `eqchan` 70e⟩

⟨*function* `eqchantdqid` 71a⟩

⟨*constructor* `newmhead` 344b⟩

⟨*function* `cmount` 350c⟩

⟨*function* `cunmount` 365d⟩

⟨*function* `cclone` 332a⟩

⟨*function* `findmount` 354d⟩

⟨*function* `domount` 355⟩

⟨*function* `undomount` 357b⟩

⟨*function* `ewalk` 330a⟩

⟨*global* `Edoesnotexist` 665a⟩

⟨*function* `walk` 328f⟩

⟨*function* `createdir` 361b⟩

⟨*function* `growparse` 310c⟩

⟨*function* `parsename` 310b⟩

⟨*function* `memrchr` 668b⟩

⟨*function* `namelenerror` 311d⟩

⟨*function* `nameerror` 312a⟩

⟨*function* `namec` 305f⟩

⟨*function* `skipslash` 311a⟩

⟨*global* `isfrog` 309a⟩

⟨*function* `validname0` 308b⟩

⟨*function* `validname` 308a⟩

⟨*function* `validnamedup` 307e⟩

⟨*function* `error_if_not_dir` 323b⟩

⟨*destructor* `putmhead` 344c⟩

files/dev.c

⟨`files/dev.c` 810⟩≡
⟨*kernel basic includes* 49a⟩

```
// was in $CONF.c
```
⟨*global* `devtab` 71c⟩

⟨*function* `mkqid` 70d⟩

⟨*function* `devno` 297c⟩

⟨*function* `devdir` 427c⟩

⟨*function* `devgen` 428⟩

```
void
devreset(void)
{
}

void
devinit(void)
{
}

void
devshutdown(void)
{
}
```

⟨*function* `devattach` 297b⟩

⟨*function* `devclone` 429a⟩

⟨*function* `devwalk` 429b⟩

⟨*function* `devstat` 431⟩

⟨*function* `devdirread` 432c⟩

⟨*function* `devpermcheck` 432d⟩

⟨*function* `devopen` 433a⟩

```
void
devcreate(Chan*, char*, int, ulong)
{
    error(Eperm);
}
```

⟨*function* devbread 434a⟩

⟨*function* devbrwrite 434b⟩

```
void
devremove(Chan*)
{
    error(Eperm);
}

int
devwstat(Chan*, uchar*, int)
{
    error(Eperm);
    return 0;
}

//void
//devpower(int)
//{
//    error(Eperm);
//}
//
//int
//devconfig(int, char *, DevConf *)
//{
//    error(Eperm);
//    return 0;
//}
```

files/allocb.c

⟨files/allocb.c 811⟩≡
⟨*kernel basic includes* 49a⟩

```
enum
{
    Hdrspc      = 64,       /* leave room for high-level headers */
    Bdead       = 0x51494F42,   /* "QIOB" */
};
```

⟨*struct* Ialloc 437a⟩
⟨*global* ialloc 437b⟩

⟨*function* _allocb 435e⟩

⟨*function* allocb 436b⟩

⟨*function* iallocb 437c⟩

⟨*function* freeb 438a⟩

⟨*function* checkb 438b⟩

⟨*function* iallocsummary 644e⟩

files/env.c

⟨files/env.c 812a⟩≡
  ⟨*kernel basic includes* 49a⟩

```
// this used to be in devenv.c, but to avoid backward deps I've splitted
// this file in 2
```

  ⟨*function* closeegrp 597d⟩

files/file.c

⟨files/file.c 812b⟩≡
  ⟨*kernel basic includes* 49a⟩

  ⟨*function* fdtochan 302d⟩

  ⟨*function* openmode 302b⟩

  ⟨*function* fdclose 303b⟩

  ⟨*function* validstat 340b⟩

files/mnt.c

⟨files/mnt.c 812c⟩≡
  ⟨*kernel basic includes* 49a⟩

```
// this used to be in devmnt.c, but to avoid backward deps I've splitted
// this file in 2 (which forced to put more stuff in portdat_files.h though).
```

  ⟨*global* mntalloc 580c⟩

```
#define MAXRPC (IOHDRSZ+8192)
```

  ⟨*function* freetag 581a⟩

  ⟨*function* mntfree 581b⟩

  ⟨*function* mntpntfree 581c⟩

  ⟨*function* muxclose 582a⟩

  ⟨*function* mntversion 582d⟩

files/qio.c

⟨*qio.c globals cnt* 812d⟩≡ (813c)

```
static ulong padblockcnt;
static ulong concatblockcnt;
static ulong pullupblockcnt;
static ulong copyblockcnt;
static ulong consumecnt;
static ulong producecnt;
static ulong qcopycnt;
static ulong noblockcnt;
```

⟨*qio.c global debugging* 813a⟩≡ (813c)
```
static int debugging;

//#define QDEBUG  if(0)
```

⟨*global* `qiomaxatomic` 813b⟩≡ (813c)
```
enum
{
    Maxatomic   = 64*1024,
};

uint    qiomaxatomic = Maxatomic;
```
Uses `Maxatomic-779` 813b and `qiomaxatomic` 813b.

⟨`files/qio.c` 813c⟩≡
⟨*kernel basic includes* 49a⟩

⟨*qio.c globals cnt* 812d⟩

⟨*qio.c global debugging* 813a⟩

⟨*global* `qiomaxatomic` 813b⟩

⟨*function* `ixsummary` 644d⟩

⟨*function* `freeblist` 439a⟩

⟨*function* `padblock` 439b⟩

⟨*function* `blocklen` 440a⟩

⟨*function* `blockalloclen` 440b⟩

⟨*function* `concatblock` 441a⟩

⟨*function* `pullupblock` 441b⟩

⟨*function* `pullupqueue` 401a⟩

⟨*function* `trimblock` 443a⟩

⟨*function* `copyblock` 443b⟩

⟨*function* `adjustblock` 444a⟩

⟨*function* `pullblock` 444b⟩

⟨*function* `qget` 402a⟩

⟨*function* `qdiscard` 402b⟩

⟨*function* `qconsume` 403⟩

⟨*function* `qpass` 405⟩

⟨*function* `qpassnolim` 406⟩

⟨*function* `packblock` 442⟩

⟨*function* `qproduce` 407⟩

⟨*function* `qcopy` 408⟩

⟨*function* `qopen` 409a⟩

⟨*function* `qbypass` 409b⟩

⟨*function* `notempty` 409c⟩

⟨*function* `qwait` 410a⟩

⟨*function* `qaddlist` 410b⟩

⟨*function* `qremove` 410c⟩

⟨*function* `bl2mem` 445⟩

⟨*function* `qputback` 411a⟩

⟨*function* `qwakeup_iunlock` 411b⟩

⟨*function* `qbread` 412⟩

⟨*function* `qread` 413⟩

⟨*function* `qnotfull` 414a⟩

⟨*function* `qbwrite` 414b⟩

⟨*function* `qwrite` 401b⟩

⟨*function* `qiwrite` 416⟩

⟨*function* `qfree` 417⟩

⟨*function* `qclose` 418a⟩

⟨*function* `qhangup` 418b⟩

⟨*function* `qiclosed` 419a⟩

⟨*function* `qreopen` 419b⟩

⟨*function* `qlen` 419c⟩

⟨*function* `qwindow` 419d⟩

⟨*function* `qcanread` 420a⟩

⟨*function* `qsetlimit` 420b⟩

⟨*function* `qnoblock` 420c⟩

⟨*function* `qflush` 420d⟩

⟨*function* `qfull` 421⟩

files/sysfile.c

⟨files/sysfile.c 815⟩≡
⟨*kernel basic includes* 49a⟩

```
/*
 * The sys*() routines needn't poperror() as they return directly to syscall().
 */
```

⟨*function* unlockfgrp 301b⟩

⟨*function* growfd 300c⟩

⟨*function* findfreefd 300b⟩

⟨*function* newfd 300a⟩

⟨*function* newfd2 458a⟩

⟨*syscall fd2path* 318a⟩

⟨*syscall pipe* 457⟩

⟨*syscall dup* 317⟩

⟨*syscall open* 302a⟩

⟨*syscall close* 302c⟩

⟨*function* unionread 359f⟩

⟨*function* unionrewind 360b⟩

⟨*function* dirfixed 372⟩

⟨*function* offsetof_name_direntry 338a⟩

⟨*function* direntry_setname 337b⟩

⟨*function* mountrock 369c⟩

⟨*function* mountrockread 370a⟩

⟨*function* mountrewind 370b⟩

⟨*function* mountfix 371⟩

⟨*function* read 313b⟩

⟨*syscall pread* 312b⟩

⟨*function* write 314c⟩

⟨*syscall pwrite* 314a⟩

⟨*function* sseek 316⟩

⟨*syscall seek* 315⟩

⟨*function* `pathlast` 339b⟩

⟨*syscall fstat* 339a⟩

⟨*syscall stat* 338c⟩

⟨*syscall chdir* 336a⟩

⟨*function* `bindmount` 349a⟩

⟨*syscall bind* 348d⟩

⟨*syscall mount* 363g⟩

⟨*syscall unmount* 365a⟩

⟨*syscall create* 332b⟩

⟨*syscall remove* 335d⟩

⟨*function* `wstat` 341⟩

⟨*syscall wstat* 339d⟩

⟨*syscall fwstat* 340a⟩

files/cache.c

⟨*cache.c forward decl* 816a⟩≡ (817)
```
typedef struct Cache Cache;
typedef struct Ecache Ecache;
```
Uses `Cache` 817 and `Ecache` 817.

⟨`Chan` *other fields* 816b⟩+≡ (68b) ◁745b
```
Mntcache* mcp;      /* Mount cache pointer */
```

⟨*function* `auxpage` 816c⟩≡ (863d)
```
Page*
auxpage(void)
{
    Page *p;

    lock(&palloc);
    p = palloc.head;
    if(palloc.freecount < swapalloc.highwater) {
        unlock(&palloc);
        return nil;
    }
    pageunchain(p);

    lock(p);
    if(p->ref != 0)
        panic("auxpage");
    p->ref++;
    uncachepage(p);
    unlock(p);
    unlock(&palloc);
```

```
    return p;
}
```

Uses lock() 76d, pageunchain() 110d, palloc 109b, swapalloc 204b, uncachepage() 192a, and unlock() 78d.

⟨files/cache.c 817⟩≡

```
⟨kernel basic includes 49a⟩

enum
{
    NHASH       = 128,
    MAXCACHE    = 1024*1024,
    NFILE       = 4096,
    NEXTENT     = 200,      /* extent allocation size */
};

⟨cache.c forward decl 816a⟩

struct Cache
{
    QLock;
    int     pgno;
    Mntcache    *head;
    Mntcache    *tail;
    Mntcache    *hash[NHASH];
};

struct Ecache
{
    Lock;
    int total;
    int free;
    Extent* head;
};

static KImage fscache;
static Cache cache;
static Ecache ecache;
static int maxcache = MAXCACHE;

static void
extentfree(Extent* e)
{
    lock(&ecache);
    e->next = ecache.head;
    ecache.head = e;
    ecache.free++;
    unlock(&ecache);
}

static Extent*
extentalloc(void)
{
    Extent *e;
    int i;

    lock(&ecache);
    if(ecache.head == nil){
        e = xalloc(NEXTENT*sizeof(Extent));
        if(e == nil){
            unlock(&ecache);
```

```
            return nil;
        }
        for(i = 0; i < NEXTENT; i++){
            e->next = ecache.head;
            ecache.head = e;
            e++;
        }
        ecache.free += NEXTENT;
        ecache.total += NEXTENT;
    }

    e = ecache.head;
    ecache.head = e->next;
    memset(e, 0, sizeof(Extent));
    ecache.free--;
    unlock(&ecache);

    return e;
}

void
cinit(void)
{
    int i;
    Mntcache *m;

    cache.head = xalloc(sizeof(Mntcache)*NFILE);
    m = cache.head;
    if (m == nil)
        panic("cinit: no memory");

    /* a better algorithm would be nice */
    if(conf.npage*BY2PG > 400*MB)
        maxcache = 20*MAXCACHE;
    else if(conf.npage*BY2PG > 200*MB)
        maxcache = 10*MAXCACHE;

    for(i = 0; i < NFILE-1; i++) {
        m->next = m+1;
        m->prev = m-1;
        m++;
    }

    cache.tail = m;
    cache.tail->next = 0;
    cache.head->prev = 0;

    fscache.notext = true;
}

Page*
cpage(Extent *e)
{
    /* Easy consistency check */
    if(e->cache->daddr != e->bid)
        return 0;

    return lookpage(&fscache, e->bid);
}
```

```
void
cnodata(Mntcache *m)
{
    Extent *e, *n;

    /*
     * Invalidate all extent data
     * Image lru will waste the pages
     */
    for(e = m->list; e; e = n) {
        n = e->next;
        extentfree(e);
    }
    m->list = 0;
}

void
ctail(Mntcache *m)
{
    /* Unlink and send to the tail */
    if(m->prev)
        m->prev->next = m->next;
    else
        cache.head = m->next;
    if(m->next)
        m->next->prev = m->prev;
    else
        cache.tail = m->prev;

    if(cache.tail) {
        m->prev = cache.tail;
        cache.tail->next = m;
        m->next = 0;
        cache.tail = m;
    }
    else {
        cache.head = m;
        cache.tail = m;
        m->prev = 0;
        m->next = 0;
    }
}

void
copen(Chan *c)
{
    int h;
    Extent *e, *next;
    Mntcache *m, *f, **l;

    /* directories aren't cacheable and append-only files confuse us */
    if(c->qid.type&(QTDIR|QTAPPEND))
        return;

    h = c->qid.path%NHASH;
    qlock(&cache);
    for(m = cache.hash[h]; m; m = m->hash) {
        if(m->qid.path == c->qid.path)
        if(m->qid.type == c->qid.type)
        if(m->dev == c->dev && m->type == c->type) {
```

```
            c->mcp = m;
            ctail(m);
            qunlock(&cache);

            /* File was updated, invalidate cache */
            if(m->qid.vers != c->qid.vers) {
                m->qid.vers = c->qid.vers;
                qlock(m);
                cnodata(m);
                qunlock(m);
            }
            return;
        }
    }

    /* LRU the cache headers */
    m = cache.head;
    l = &cache.hash[m->qid.path%NHASH];
    for(f = *l; f; f = f->hash) {
        if(f == m) {
            *l = m->hash;
            break;
        }
        l = &f->hash;
    }

    m->qid = c->qid;
    m->dev = c->dev;
    m->type = c->type;

    l = &cache.hash[h];
    m->hash = *l;
    *l = m;
    ctail(m);

    qlock(m);
    c->mcp = m;
    e = m->list;
    m->list = 0;
    qunlock(&cache);

    while(e) {
        next = e->next;
        extentfree(e);
        e = next;
    }
    qunlock(m);
}

static int
cdev(Mntcache *m, Chan *c)
{
    if(m->qid.path != c->qid.path)
        return 0;
    if(m->qid.type != c->qid.type)
        return 0;
    if(m->dev != c->dev)
        return 0;
    if(m->type != c->type)
        return 0;
```

```
    if(m->qid.vers != c->qid.vers)
        return 0;
    return 1;
}

int
cread(Chan *c, uchar *buf, int len, vlong off)
{
    Arch_KMap *k;
    Page *p;
    Mntcache *m;
    Extent *e, **t;
    int o, l, total;
    ulong offset;

    if(off+len > maxcache)
        return 0;

    m = c->mcp;
    if(m == 0)
        return 0;

    qlock(m);
    if(cdev(m, c) == 0) {
        qunlock(m);
        return 0;
    }

    offset = off;
    t = &m->list;
    for(e = *t; e; e = e->next) {
        if(offset >= e->start && offset < e->start+e->len)
            break;
        t = &e->next;
    }

    if(e == 0) {
        qunlock(m);
        return 0;
    }

    total = 0;
    while(len) {
        p = cpage(e);
        if(p == 0) {
            *t = e->next;
            extentfree(e);
            qunlock(m);
            return total;
        }

        o = offset - e->start;
        l = len;
        if(l > e->len-o)
            l = e->len-o;

        k = arch_kmap(p);
        if(waserror()) {
            arch_kunmap(k);
            putpage(p);
```

```
            qunlock(m);
            nexterror();
        }

        memmove(buf, (uchar*)VA(k) + o, l);

        poperror();
        arch_kunmap(k);

        putpage(p);

        buf += l;
        len -= l;
        offset += l;
        total += l;
        t = &e->next;
        e = e->next;
        if(e == 0 || e->start != offset)
            break;
    }

    qunlock(m);
    return total;
}

Extent*
cchain(uchar *buf, ulong offset, int len, Extent **tail)
{
    int l;
    Page *p;
    Arch_KMap *k;
    Extent *e, *start, **t;

    start = 0;
    *tail = 0;
    t = &start;
    while(len) {
        e = extentalloc();
        if(e == 0)
            break;

        p = auxpage();
        if(p == 0) {
            extentfree(e);
            break;
        }
        l = len;
        if(l > BY2PG)
            l = BY2PG;

        e->cache = p;
        e->start = offset;
        e->len = l;

        qlock(&cache);
        e->bid = cache.pgno;
        cache.pgno += BY2PG;
        /* wrap the counter; low bits are unused by pghash but checked by lookpage */
        if((cache.pgno & ~(BY2PG-1)) == 0){
            if(cache.pgno == BY2PG-1){
```

```
                print("cache wrapped\n");
                cache.pgno = 0;
            }else
                cache.pgno++;
        }
        qunlock(&cache);

        p->daddr = e->bid;
        k = arch_kmap(p);
        if(waserror()) {        /* buf may be virtual */
            arch_kunmap(k);
            nexterror();
        }
        memmove((void*)VA(k), buf, l);
        poperror();
        arch_kunmap(k);

        cachepage(p, &fscache);
        putpage(p);

        buf += l;
        offset += l;
        len -= l;

        *t = e;
        *tail = e;
        t = &e->next;
    }

    return start;
}

int
cpgmove(Extent *e, uchar *buf, int boff, int len)
{
    Page *p;
    Arch_KMap *k;

    p = cpage(e);
    if(p == 0)
        return 0;

    k = arch_kmap(p);
    if(waserror()) {        /* Since buf may be virtual */
        arch_kunmap(k);
        nexterror();
    }

    memmove((uchar*)VA(k)+boff, buf, len);

    poperror();
    arch_kunmap(k);
    putpage(p);

    return 1;
}

void
cupdate(Chan *c, uchar *buf, int len, vlong off)
{
```

```
Mntcache *m;
Extent *tail;
Extent *e, *f, *p;
int o, ee, eblock;
ulong offset;

if(off > maxcache || len == 0)
    return;

m = c->mcp;
if(m == 0)
    return;
qlock(m);
if(cdev(m, c) == 0) {
    qunlock(m);
    return;
}

/*
 * Find the insertion point
 */
offset = off;
p = 0;
for(f = m->list; f; f = f->next) {
    if(f->start > offset)
        break;
    p = f;
}

/* trim if there is a successor */
eblock = offset+len;
if(f != 0 && eblock > f->start) {
    len -= (eblock - f->start);
    if(len <= 0) {
        qunlock(m);
        return;
    }
}

if(p == 0) {        /* at the head */
    e = cchain(buf, offset, len, &tail);
    if(e != 0) {
        m->list = e;
        tail->next = f;
    }
    qunlock(m);
    return;
}

/* trim to the predecessor */
ee = p->start+p->len;
if(offset < ee) {
    o = ee - offset;
    len -= o;
    if(len <= 0) {
        qunlock(m);
        return;
    }
    buf += o;
    offset += o;
```

```
    }

    /* try and pack data into the predecessor */
    if(offset == ee && p->len < BY2PG) {
        o = len;
        if(o > BY2PG - p->len)
            o = BY2PG - p->len;
        if(cpgmove(p, buf, p->len, o)) {
            p->len += o;
            buf += o;
            len -= o;
            offset += o;
            if(len <= 0) {
if(f && p->start + p->len > f->start) print("CACHE: p->start=%uld p->len=%d f->start=%uld\n", p->start, p->len,
                qunlock(m);
                return;
            }
        }
    }

    e = cchain(buf, offset, len, &tail);
    if(e != 0) {
        p->next = e;
        tail->next = f;
    }
    qunlock(m);
}

void
cwrite(Chan* c, uchar *buf, int len, vlong off)
{
    int o, eo;
    Mntcache *m;
    ulong eblock, ee;
    Extent *p, *f, *e, *tail;
    ulong offset;

    if(off > maxcache || len == 0)
        return;

    m = c->mcp;
    if(m == 0)
        return;

    qlock(m);
    if(cdev(m, c) == 0) {
        qunlock(m);
        return;
    }

    offset = off;
    m->qid.vers++;
    c->qid.vers++;

    p = 0;
    for(f = m->list; f; f = f->next) {
        if(f->start >= offset)
            break;
        p = f;
    }
```

```
    if(p != 0) {
        ee = p->start+p->len;
        eo = offset - p->start;
        /* pack in predecessor if there is space */
        if(offset <= ee && eo < BY2PG) {
            o = len;
            if(o > BY2PG - eo)
                o = BY2PG - eo;
            if(cpgmove(p, buf, eo, o)) {
                if(eo+o > p->len)
                    p->len = eo+o;
                buf += o;
                len -= o;
                offset += o;
            }
        }
    }

    /* free the overlap -- it's a rare case */
    eblock = offset+len;
    while(f && f->start < eblock) {
        e = f->next;
        extentfree(f);
        f = e;
    }

    /* link the block (if any) into the middle */
    e = cchain(buf, offset, len, &tail);
    if(e != 0) {
        tail->next = f;
        f = e;
    }

    if(p == 0)
        m->list = f;
    else
        p->next = f;
    qunlock(m);
}
```

Uses MAXCACHE-758 817, MB 867a, NEXTENT-760 817, NFILE-759 817, NHASH-757 817, QTAPPEND 427b, QTDIR 70b, auxpage() 816c, cache-762 817, cachepage() 190e, cchain() 817, cdev() 817, cnodata() 817, conf 62b, cpage() 817, cpgmove() 817, ctail() 817, ecache-763 817, extentalloc() 817, extentfree() 817, fscache-761 817, lock() 76d, lookpage() 191a, maxcache-764 817, nexterror, poperror 666d, print, putpage() 111a, qlock() 81f, qunlock() 82c, true, unlock() 78d, waserror 666d, and xalloc() 95d.

E.12 filesystems/

filesystems/devfs.c

⟨*devfs.c forward decl* 826⟩≡ (827c)

```
typedef struct Inner Inner;
typedef struct Fsdev Fsdev;
typedef struct Tree Tree;

extern Dev fsdevtab;        /* forward */
```

Uses Fsdev 827c, Inner 827c, and Tree 827c.

⟨*devfs.c debugging macro* 827a⟩≡ (827c)
```
static int debug;
#define dprint if(debug)print
```

⟨*global* `fsdevtab` 827b⟩≡ (827c)
```
Dev fsdevtab = {
    .dc       =    'k',
    .name     =    "fs",

    .reset    =    devreset,
    .init     =    devinit,
    .shutdown =    devshutdown,
    .attach   =    mattach,
    .walk     =    mwalk,
    .stat     =    mstat,
    .open     =    mopen,
    .create   =    devcreate,
    .close    =    mclose,
    .read     =    mread,
    .bread    =    devbread,
    .write    =    mwrite,
    .bwrite   =    devbwrite,
    .remove   =    devremove,
    .wstat    =    devwstat,

//    .power    =    devpower,
//    .config   =    devconfig,
};
```
Uses `devbread()` 434a, `devbwrite()` 434b, `devcreate()` 810, `devinit()` 810, `devremove()` 810, `devreset()` 810, `devshutdown()` 810, `devwstat()` 810, `mattach()` 827c, `mclose()` 827c, `mopen()` 827c, `mread()` 827c, `mstat()` 827c, `mwalk()` 827c, and `mwrite()` 827c.

⟨`filesystems/devfs.c` 827c⟩≡
```
/*
 * File system devices.
 * Follows device config in Ken's file server.
 * Builds mirrors, concatenations, interleavings, and partitions
 * of devices out of other (inner) devices.
 * It is ok if inner devices are provided by this driver.
 *
 * Built files are grouped on different directories
 * (called trees, and used to represent disks).
 * The "#k/fs" tree is always available and never goes away.
 * Configuration changes happen only while no I/O is in progress.
 *
 * Default sector size is one byte unless changed by the "disk" ctl.
 */

⟨kernel basic includes 49a⟩
#include "io.h"
#include <ureg.h>

enum
{
    Fnone,
    Fmirror,        /* mirror of others */
    Fcat,           /* catenation of others */
    Finter,         /* interleaving of others */
    Fpart,          /* part of other */
    Fclear,         /* start over */
```

```
    Fdel,           /* delete a configure device */
    Fdisk,          /* set default tree and sector sz*/

    Sectorsz = 1,
    Blksize = 8*1024,   /* for Finter only */

    Incr = 5,       /* Increments for the dev array */

    /*
     * All qids are decorated with the tree number.
     * #k/fs is tree number 0, is automatically added and
     * its first qid is for the ctl file. It never goes away.
     */
    Qtop    = 0,        /* #k */
    Qdir,           /* directory (#k/fs) */
    Qctl,           /* ctl, only for #k/fs/ctl */
    Qfirst,         /* first qid assigned for device */

    Iswrite = 0,
    Isread,

    Optional = 0,
    Mustexist,

    /* tunable parameters */
    Maxconf = 4*1024,   /* max length for config */
    Ndevs   = 32,       /* max. inner devs per command */
    Ntrees  = 128,      /* max. number of trees */
    Maxretries = 3,     /* max. retries of i/o errors */
    Retrypause = 5000,  /* ms. to pause between retries */
};
```

⟨*devfs.c forward decl* 826⟩

⟨*devfs.c debugging macro* 827a⟩

```
struct Inner
{
    char    *iname;     /* inner device name */
    vlong   isize;      /* size of inner device */
    Chan    *idev;      /* inner device */
};

struct Fsdev
{
    Ref;            /* one per Chan doing I/O */
    int gone;       /* true if removed */
    int vers;       /* qid version for this device */
    int type;       /* Fnone, Fmirror, ... */
    char    *name;      /* name for this fsdev */
    Tree*   tree;       /* where the device is kept */
    vlong   size;       /* min(inner[X].isize) */
    vlong   start;      /* start address (for Fpart) */
    uint    ndevs;      /* number of inner devices */
    int perm;       /* minimum of inner device perms */
    Inner   *inner[Ndevs];  /* inner devices */
};

struct Tree
```

```
{
    char    *name;      /* name for #k/<name> */
    Fsdev   **devs;     /* devices in dir. */
    uint    ndevs;      /* number of devices */
    uint    nadevs;     /* number of allocated devices in devs */
};



static RWlock lck;      /* r: use devices; w: change config  */
static Tree fstree;     /* The main "fs" tree. Never goes away */
static Tree *trees[Ntrees]; /* internal representation of config */
static int ntrees;      /* max number of trees */
static int qidvers;

static char *disk;      /* default tree name used */
static char *source;        /* default inner device used */
static int sectorsz = Sectorsz; /* default sector size */

static char confstr[Maxconf];   /* textual configuration */


static char cfgstr[] = "fsdev:\n";

static Qid tqid = {Qtop, 0, QTDIR};
static Qid cqid = {Qctl, 0, 0};

static char* tnames[] = {
    [Fmirror]   "mirror",
    [Fcat]      "cat",
    [Finter]    "inter",
    [Fpart]     "part",
};

static Cmdtab configs[] = {
    Fmirror,"mirror",   0,
    Fcat,   "cat",      0,
    Finter, "inter",    0,
    Fpart,  "part",     0,
    Fclear, "clear",    1,
    Fdel,   "del",      2,
    Fdisk,  "disk",     0,
};

static char Egone[] = "device is gone";     /* file has been removed */

static char*
seprintdev(char *s, char *e, Fsdev *mp)
{
    int i;

    if(mp == nil)
        return seprint(s, e, "<null Fsdev>");
    if(mp->type < 0 || mp->type >= nelem(tnames) || tnames[mp->type] == nil)
        return seprint(s, e, "bad device type %d\n", mp->type);

    s = strecpy(s, e, tnames[mp->type]);
    if(mp->tree != &fstree)
        s = seprint(s, e, " %s/%s", mp->tree->name, mp->name);
    else
```

```
        s = seprint(s, e, " %s", mp->name);
    for(i = 0; i < mp->ndevs; i++)
        s = seprint(s, e, " %s", mp->inner[i]->iname);
    switch(mp->type){
    case Fmirror:
    case Fcat:
    case Finter:
        s = strecpy(s, e, "\n");
        break;
    case Fpart:
        s = seprint(s, e, " %ulld %ulld\n", mp->start, mp->size);
        break;
    default:
        panic("#k: seprintdev bug");
    }
    return s;
}

static vlong
mkpath(int tree, int devno)
{
    return (tree&0xFFFF)<<16 | devno&0xFFFF;
}

static int
path2treeno(int q)
{
    return q>>16 & 0xFFFF;
}

static int
path2devno(int q)
{
    return q & 0xFFFF;
}

static Tree*
gettree(int i, int mustexist)
{
    dprint("gettree %d\n", i);
    if(i < 0)
        panic("#k: bug: bad tree index %d in gettree", i);
    if(i >= ntrees || trees[i] == nil)
        if(mustexist)
            error(Enonexist);
        else
            return nil;
    return trees[i];
}

static Fsdev*
getdev(Tree *t, int i, int mustexist)
{
    dprint("getdev %d\n", i);
    if(i < 0)
        panic("#k: bug: bad dev index %d in getdev", i);
    if(i >= t->nadevs || t->devs[i] == nil)
        if(mustexist)
            error(Enonexist);
        else
```

```
            return nil;
    return t->devs[i];
}

static Fsdev*
path2dev(int q)
{
    Tree    *t;

    dprint("path2dev %ux\n", q);
    t = gettree(path2treeno(q), Mustexist);
    return getdev(t, path2devno(q) - Qfirst, Mustexist);
}

static Tree*
treealloc(char *name)
{
    int i;
    Tree    *t;

    dprint("treealloc %s\n", name);
    for(i = 0; i < nelem(trees); i++)
        if(trees[i] == nil)
            break;
    if(i == nelem(trees))
        return nil;
    t = trees[i] = mallocz(sizeof(Tree), 1);
    if(t == nil)
        return nil;
    if(i == ntrees)
        ntrees++;
    kstrdup(&t->name, name);
    return t;
}

static Tree*
lookuptree(char *name)
{
    int i;

    dprint("lookuptree %s\n", name);
    for(i = 0; i < ntrees; i++)
        if(trees[i] != nil && strcmp(trees[i]->name, name) == 0)
            return trees[i];
    return nil;
}

static Fsdev*
devalloc(Tree *t, char *name)
{
    int i, ndevs;
    Fsdev   *mp, **devs;

    dprint("devalloc %s %s\n", t->name, name);
    mp = mallocz(sizeof(Fsdev), 1);
    if(mp == nil)
        return nil;
    for(i = 0; i < t->nadevs; i++)
        if(t->devs[i] == nil)
            break;
```

```
    if(i >= t->nadevs){
        if(t->nadevs % Incr == 0){
            ndevs = t->nadevs + Incr;
            devs = realloc(t->devs, ndevs * sizeof(Fsdev*));
            if(devs == nil){
                free(mp);
                return nil;
            }
            t->devs = devs;
        }
        t->devs[t->nadevs] = nil;
        t->nadevs++;
    }
    kstrdup(&mp->name, name);
    mp->vers = ++qidvers;
    mp->tree = t;
    t->devs[i] = mp;
    t->ndevs++;
    return mp;
}

static void
deltree(Tree *t)
{
    int i;

    dprint("deltree %s\n", t->name);
    for(i = 0; i < ntrees; i++)
        if(trees[i] == t){
            if(i > 0){      /* "fs" never goes away */
                free(t->name);
                free(t->devs);
                free(t);
                trees[i] = nil;
            }
            return;
        }
    panic("#k: deltree: bug: tree not found");
}

/*
 * A device is gone and we know that all its users are gone.
 * A tree is gone when all its devices are gone ("fs" is never gone).
 * Must close devices outside locks, so we could nest our own devices.
 */
static void
mdeldev(Fsdev *mp)
{
    int i;
    Inner   *in;
    Tree    *t;

    dprint("deldev %s gone %d ref %uld\n", mp->name, mp->gone, mp->ref);

    mp->gone = 1;
    mp->vers = ++qidvers;

    wlock(&lck);
    t = mp->tree;
    for(i = 0; i < t->nadevs; i++)
```

```
        if(t->devs[i] == mp){
            t->devs[i] = nil;
            t->ndevs--;
            if(t->ndevs == 0)
                deltree(t);
            break;
        }
    wunlock(&lck);

    free(mp->name);
    for(i = 0; i < mp->ndevs; i++){
        in = mp->inner[i];
        if(in->idev != nil)
            cclose(in->idev);
        free(in->iname);
        free(in);
    }
    if(debug)
        memset(mp, 9, sizeof *mp);  /* poison */
    free(mp);
}

/*
 * Delete one or all devices in one or all trees.
 */
static void
mdelctl(char *tname, char *dname)
{
    int i, alldevs, alltrees, some;
    Fsdev *mp;
    Tree *t;

    dprint("delctl %s\n", dname);
    alldevs = strcmp(dname, "*") == 0;
    alltrees = strcmp(tname, "*") == 0;
    some = 0;
Again:
    wlock(&lck);
    for(i = 0; i < ntrees; i++){
        t = trees[i];
        if(t == nil)
            continue;
        if(alltrees == 0 && strcmp(t->name, tname) != 0)
            continue;
        for(i = 0; i < t->nadevs; i++){
            mp = t->devs[i];
            if(t->devs[i] == nil)
                continue;
            if(alldevs == 0 && strcmp(mp->name, dname) != 0)
                continue;
            /*
             * Careful: must close outside locks and that
             * may change the file tree we are looking at.
             */
            some++;
            mp->gone = 1;
            if(mp->ref == 0){
                incref(mp); /* keep it there */
                wunlock(&lck);
                mdeldev(mp);
```

```
                goto Again; /* tree can change */
            }
        }
    }
    wunlock(&lck);
    if(some == 0 && alltrees == 0)
        error(Enonexist);
}

static void
setdsize(Fsdev* mp, vlong *ilen)
{
    int i;
    vlong   inlen;
    Inner   *in;

    dprint("setdsize %s\n", mp->name);
    for (i = 0; i < mp->ndevs; i++){
        in = mp->inner[i];
        in->isize = ilen[i];
        inlen = in->isize;
        switch(mp->type){
        case Finter:
            /* truncate to multiple of Blksize */
            inlen &= ~(Blksize-1);
            in->isize = inlen;
            /* fall through */
        case Fmirror:
            /* use size of smallest inner device */
            if (mp->size == 0 || mp->size > inlen)
                mp->size = inlen;
            break;
        case Fcat:
            mp->size += inlen;
            break;
        case Fpart:
            if(mp->start > inlen)
                error("partition starts after device end");
            if(inlen < mp->start + mp->size){
                print("#k: %s: partition truncated from "
                    "%lld to %lld bytes\n", mp->name,
                    mp->size, inlen - mp->start);
                mp->size = inlen - mp->start;
            }
            break;
        }
    }
    if(mp->type == Finter)
        mp->size *= mp->ndevs;
}

static void
validdevname(Tree *t, char *dname)
{
    int i;

    for(i = 0; i < t->nadevs; i++)
        if(t->devs[i] != nil && strcmp(t->devs[i]->name, dname) == 0)
            error(Eexist);
}
```

```
static void
parseconfig(char *a, long n, Cmdbuf **cbp, Cmdtab **ctp)
{
    Cmdbuf  *cb;
    Cmdtab  *ct;

    *cbp = cb = parsecmd(a, n);
    *ctp = ct = lookupcmd(cb, configs, nelem(configs));

    cb->f++;            /* skip command */
    cb->nf--;
    switch(ct->index){
    case Fmirror:
    case Fcat:
    case Finter:
        if(cb->nf < 2)
            error("too few arguments for ctl");
        if(cb->nf - 1 > Ndevs)
            error("too many devices in ctl");
        break;
    case Fdisk:
        if(cb->nf < 1 || cb->nf > 3)
            error("ctl usage: disk name [sz dev]");
        break;
    case Fpart:
        if(cb->nf != 4 && (cb->nf != 3 || source == nil))
            error("ctl usage: part new [file] off len");
        break;
    }
}

static void
parsename(char *name, char *disk, char **tree, char **dev)
{
    char *slash;

    slash = strchr(name, '/');
    if(slash == nil){
        if(disk != nil)
            *tree = disk;
        else
            *tree = "fs";
        *dev = name;
    }else{
        *tree = name;
        *slash++ = 0;
        *dev = slash;
    }
    validname(*tree, false);
    validname(*dev, false);
}

static int
getattrs(Chan *c, vlong *lenp, int *permp)
{
    uchar   buf[128];   /* old DIRLEN plus a little should be plenty */
    DirEntry d;
    long    l;
```

```
    *lenp = 0;
    *permp = 0;
    l = devtab[c->type]->stat(c, buf, sizeof buf);
    if (l >= 0 && convM2D(buf, l, &d, nil) > 0) {
        *lenp = d.length;
        *permp = d.mode & 0777;
    }
    return l;
}

/*
 * Process a single line of configuration,
 * often of the form "cmd newname idev0 idev1".
 * locking is tricky, because we need a write lock to
 * add/remove devices yet adding/removing them may lead
 * to calls to this driver that require a read lock (when
 * inner devices are also provided by us).
 */
static void
mconfig(char* a, long n)
{
    int i;
    int *iperm;
    vlong   size, start;
    vlong   *ilen;
    char    *tname, *dname, *fakef[4];
    Chan    **idev;
    Cmdbuf  *cb;
    Cmdtab  *ct;
    Fsdev   *mp;
    Inner   *inprv;
    Tree    *t;

    /* ignore comments & empty lines */
    if (*a == '\0' || *a == '#' || *a == '\n')
        return;

    dprint("mconfig\n");
    size = 0;
    start = 0;
    mp = nil;
    cb = nil;
    idev = nil;
    ilen = nil;
    iperm = nil;

    if(waserror()){
        free(cb);
        nexterror();
    }

    parseconfig(a, n, &cb, &ct);
    switch (ct->index) {
    case Fdisk:
        kstrdup(&disk, cb->f[0]);
        if(cb->nf >= 2)
            sectorsz = strtoul(cb->f[1], 0, 0);
        else
            sectorsz = Sectorsz;
        if(cb->nf == 3)
```

```
            kstrdup(&source, cb->f[2]);
        else{
            free(source);
            source = nil;
        }
        poperror();
        free(cb);
        return;
    case Fclear:
        poperror();
        free(cb);
        mdelctl("*", "*");      /* del everything */
        return;
    case Fpart:
        if(cb->nf == 3){
            /*
             * got a request in the format of sd(3),
             * pretend we got one in our format.
             * later we change end to be len.
             */
            fakef[0] = cb->f[0];
            fakef[1] = source;
            fakef[2] = cb->f[1];
            fakef[3] = cb->f[2];
            cb->f = fakef;
            cb->nf = 4;
        }
        start = strtoll(cb->f[2], nil, 10);
        size =  strtoll(cb->f[3], nil, 10);
        if(cb->f == fakef)
            size -= start;      /* it was end */
        cb->nf -= 2;
        break;
    }
    parsename(cb->f[0], disk, &tname, &dname);
    for(i = 1; i < cb->nf; i++)
        validname(cb->f[i], true);

    if(ct->index == Fdel){
        mdelctl(tname, dname);
        poperror();
        free(cb);
        return;
    }

    /*
     * Open all inner devices while we have only a read lock.
     */
    poperror();
    rlock(&lck);
    if(waserror()){
        runlock(&lck);
Fail:
        for(i = 1; i < cb->nf; i++)
            if(idev != nil && idev[i-1] != nil)
                cclose(idev[i]);
        if(mp != nil)
            mdeldev(mp);
        free(idev);
        free(ilen);
```

```
    free(iperm);
    free(cb);
    nexterror();
}
/* record names, lengths and perms of all named files */
idev = smalloc(sizeof(Chan*) * Ndevs);
ilen = smalloc(sizeof(vlong) * Ndevs);
iperm = smalloc(sizeof(int) * Ndevs);
for(i = 1; i < cb->nf; i++){
    idev[i-1] = namec(cb->f[i], Aopen, ORDWR, 0);
    getattrs(idev[i-1], &ilen[i-1], &iperm[i-1]);
}
poperror();
runlock(&lck);

/*
 * Get a write lock and add the device if we can.
 */
wlock(&lck);
if(waserror()){
    wunlock(&lck);
    goto Fail;
}

t = lookuptree(tname);
if(t != nil)
    validdevname(t, dname);
else
    t = treealloc(tname);
if(t == nil)
    error("no more trees");
mp = devalloc(t, dname);
if(mp == nil){
    if(t->ndevs == 0)   /* it was created for us */
        deltree(t); /* but we will not mdeldev() */
    error(Enomem);
}

/* construct mp from iname, idev and iperm arrays */
mp->type = ct->index;
if(mp->type == Fpart){
    mp->start = start * sectorsz;
    mp->size = size * sectorsz;
}
mp->perm = 0666;
for(i = 1; i < cb->nf; i++){
    inprv = mp->inner[i-1] = mallocz(sizeof(Inner), 1);
    if(inprv == nil)
        error(Enomem);
    mp->ndevs++;
    kstrdup(&inprv->iname, cb->f[i]);
    inprv->idev = idev[i-1];
    idev[i-1] = nil;
    /* use the most restrictive of the inner permissions */
    mp->perm &= iperm[i-1];
}
setdsize(mp, ilen);

poperror();
wunlock(&lck);
```

```
    free(idev);
    free(ilen);
    free(iperm);
    free(cb);
}

static void
rdconf(void)
{
    int mustrd;
    char *c, *e, *p, *s;
    Chan *cc;
    static int configed;

    /* only read config file once */
    if (configed)
        return;
    configed = 1;

    dprint("rdconf\n");
    /* add the std "fs" tree */
    trees[0] = &fstree;
    ntrees++;
    fstree.name = "fs";

    /* identify the config file */
    s = getconf("fsconfig");
    if (s == nil){
        mustrd = 0;
        s = "/dev/sdC0/fscfg";
    } else
        mustrd = 1;

    /* read it */
    cc = nil;
    c = nil;
    if (waserror()){
        if (cc != nil)
            cclose(cc);
        if (c)
            free(c);
        if (!mustrd)
            return;
        nexterror();
    }
    cc = namec(s, Aopen, OREAD, 0);
    devtab[cc->type]->read(cc, confstr, sizeof confstr, 0);
    cclose(cc);
    cc = nil;

    /* validate, copy and erase config; mconfig will repopulate confstr */
    if (strncmp(confstr, cfgstr, sizeof cfgstr - 1) != 0)
        error("bad #k config, first line must be: 'fsdev:\\n'");
    kstrdup(&c, confstr + sizeof cfgstr - 1);
    memset(confstr, 0, sizeof confstr);

    /* process config copy one line at a time */
    for (p = c; p != nil && *p != '\0'; p = e){
        e = strchr(p, '\n');
        if (e == nil)
```

```
            e = p + strlen(p);
        else
            e++;
        mconfig(p, e - p);
    }
    USED(cc);       /* until now, can be used in waserror clause */
    poperror();
}

static int
mgen(Chan *c, char*, Dirtab*, int, int i, DirEntry *dp)
{
    int treeno;
    Fsdev   *mp;
    Qid qid;
    Tree    *t;

    dprint("mgen %#ullx %d\n", c->qid.path, i);
    qid.type = QTDIR;
    qid.vers = 0;
    if(c->qid.path == Qtop){
        if(i == DEVDOTDOT){
            devdir(c, tqid, "#k", 0, eve, DMDIR|0775, dp);
            return 1;
        }
        t = gettree(i, Optional);
        if(t == nil){
            dprint("no\n");
            return -1;
        }
        qid.path = mkpath(i, Qdir);
        devdir(c, qid, t->name, 0, eve, DMDIR|0775, dp);
        return 1;
    }

    treeno = path2treeno(c->qid.path);
    t = gettree(treeno, Optional);
    if(t == nil){
        dprint("no\n");
        return -1;
    }
    if((c->qid.type & QTDIR) != 0){
        if(i == DEVDOTDOT){
            devdir(c, tqid, "#k", 0, eve, DMDIR|0775, dp);
            return 1;
        }
        if(treeno == 0){
            /* take care of #k/fs/ctl */
            if(i == 0){
                devdir(c, cqid, "ctl", 0, eve, 0664, dp);
                return 1;
            }
            i--;
        }
        mp = getdev(t, i, Optional);
        if(mp == nil){
            dprint("no\n");
            return -1;
        }
        qid.type = QTFILE;
```

```
        qid.vers = mp->vers;
        qid.path = mkpath(treeno, Qfirst+i);
        devdir(c, qid, mp->name, mp->size, eve, mp->perm, dp);
        return 1;
    }

    if(i == DEVDOTDOT){
        qid.path = mkpath(treeno, Qdir);
        devdir(c, qid, t->name, 0, eve, DMDIR|0775, dp);
        return 1;
    }
    dprint("no\n");
    return -1;
}

static Chan*
mattach(char *spec)
{
    dprint("mattach\n");
    return devattach(fsdevtab.dc, spec);
}

static Walkqid*
mwalk(Chan *c, Chan *nc, char **name, int nname)
{
    Walkqid *wq;

    rdconf();

    dprint("mwalk %llux\n", c->qid.path);
    rlock(&lck);
    if(waserror()){
        runlock(&lck);
        nexterror();
    }
    wq = devwalk(c, nc, name, nname, 0, 0, mgen);
    poperror();
    runlock(&lck);
    return wq;
}

static int
mstat(Chan *c, uchar *db, int n)
{
    int p;
    DirEntry d;
    Fsdev   *mp;
    Qid q;
    Tree    *t;

    dprint("mstat %llux\n", c->qid.path);
    rlock(&lck);
    if(waserror()){
        runlock(&lck);
        nexterror();
    }
    p = c->qid.path;
    memset(&d, 0, sizeof d);
    switch(p){
    case Qtop:
```

```
        devdir(c, tqid, "#k", 0, eve, DMDIR|0775, &d);
        break;
    case Qctl:
        devdir(c, cqid, "ctl", 0, eve, 0664, &d);
        break;
    default:
        t = gettree(path2treeno(p), Mustexist);
        if(c->qid.type & QTDIR)
            devdir(c, c->qid, t->name, 0, eve, DMDIR|0775, &d);
        else{
            mp = getdev(t, path2devno(p) - Qfirst, Mustexist);
            q = c->qid;
            q.vers = mp->vers;
            devdir(c, q, mp->name, mp->size, eve, mp->perm, &d);
        }
    }
    n = convD2M(&d, db, n);
    if (n == 0)
        error(Ebadarg);
    poperror();
    runlock(&lck);
    return n;
}

static Chan*
mopen(Chan *c, int omode)
{
    int q;
    Fsdev   *mp;

    dprint("mopen %llux\n", c->qid.path);
    if((c->qid.type & QTDIR) && omode != OREAD)
        error(Eperm);
    if(c->qid.path != Qctl && (c->qid.type&QTDIR) == 0){
        rlock(&lck);
        if(waserror()){
            runlock(&lck);
            nexterror();
        }
        q = c->qid.path;
        mp = path2dev(q);
        if(mp->gone)
            error(Egone);
        devpermcheck(eve, mp->perm, omode);
        incref(mp);
        poperror();
        runlock(&lck);
    }
    /*
     * Our mgen does not return the info for the qid
     * but only for its children. Don't use devopen here.
     */
    c->offset = 0;
    c->mode = openmode(omode & ~OTRUNC);
    c->flag |= COPEN;
    return c;
}

static void
mclose(Chan *c)
```

```
{
    int mustdel, q;
    Fsdev   *mp;

    dprint("mclose %llux\n", c->qid.path);
    if(c->qid.type & QTDIR || !(c->flag & COPEN))
        return;
    rlock(&lck);
    if(waserror()){
        runlock(&lck);
        nexterror();
    }
    mustdel = 0;
    mp = nil;
    q = c->qid.path;
    if(q == Qctl){
        free(disk);
        disk = nil; /* restore defaults */
        free(source);
        source = nil;
        sectorsz = Sectorsz;
    }else{
        mp = path2dev(q);
        if(mp->gone != 0 && mp->ref == 1)
            mustdel = 1;
        else
            decref(mp);
    }
    poperror();
    runlock(&lck);
    if(mustdel)
        mdeldev(mp);
}

static long
io(Fsdev *mp, Inner *in, int isread, void *a, long l, vlong off)
{
    long wl;
    Chan    *mc;

    mc = in->idev;
    if(mc == nil)
        error(Egone);
    if (waserror()) {
        print("#k: %s: byte %,lld count %ld (of #k/%s): %s error: %s\n",
            in->iname, off, l, mp->name, (isread? "read": "write"),
            (up && up->errstr? up->errstr: ""));
        nexterror();
    }
    if (isread)
        wl = devtab[mc->type]->read(mc, a, l, off);
    else
        wl = devtab[mc->type]->write(mc, a, l, off);
    poperror();
    return wl;
}

/* NB: a transfer could span multiple inner devices */
static long
catio(Fsdev *mp, int isread, void *a, long n, vlong off)
```

```
{
    int i;
    long    l, res;
    Inner   *in;

    if(debug)
        print("catio %d %p %ld %lld\n", isread, a, n, off);
    res = n;
    for (i = 0; n > 0 && i < mp->ndevs; i++){
        in = mp->inner[i];
        if (off >= in->isize){
            off -= in->isize;
            continue;       /* not there yet */
        }
        if (off + n > in->isize)
            l = in->isize - off;
        else
            l = n;
        if(debug)
            print("\tdev %d %p %ld %lld\n", i, a, l, off);

        if (io(mp, in, isread, a, l, off) != l)
            error(Eio);

        a = (char*)a + l;
        off = 0;
        n -= l;
    }
    if(debug)
        print("\tres %ld\n", res - n);
    return res - n;
}

static long
interio(Fsdev *mp, int isread, void *a, long n, vlong off)
{
    int i;
    long    boff, res, l, wl, wsz;
    vlong   woff, blk, mblk;

    blk  = off / Blksize;
    boff = off % Blksize;
    wsz  = Blksize - boff;
    res = n;
    while(n > 0){
        mblk = blk / mp->ndevs;
        i    = blk % mp->ndevs;
        woff = mblk*Blksize + boff;
        if (n > wsz)
            l = wsz;
        else
            l = n;

        wl = io(mp, mp->inner[i], isread, a, l, woff);
        if (wl != l)
            error(Eio);

        blk++;
        boff = 0;
        wsz = Blksize;
```

```
        a = (char*)a + l;
        n -= l;
    }
    return res;
}

static char*
seprintconf(char *s, char *e)
{
    int i, j;
    Tree    *t;

    *s = 0;
    for(i = 0; i < ntrees; i++){
        t = trees[i];
        if(t != nil)
            for(j = 0; j < t->nadevs; j++)
                if(t->devs[j] != nil)
                    s = seprintdev(s, e, t->devs[j]);
    }
    return s;
}

static long
mread(Chan *c, void *a, long n, vlong off)
{
    int i, retry;
    long    l, res;
    Fsdev   *mp;
    Tree    *t;

    dprint("mread %llux\n", c->qid.path);
    rlock(&lck);
    if(waserror()){
        runlock(&lck);
        nexterror();
    }
    res = -1;
    if(c->qid.type & QTDIR){
        res = devdirread(c, a, n, 0, 0, mgen);
        goto Done;
    }
    if(c->qid.path == Qctl){
        seprintconf(confstr, confstr + sizeof(confstr));
        res = readstr((long)off, a, n, confstr);
        goto Done;
    }

    t = gettree(path2treeno(c->qid.path), Mustexist);
    mp = getdev(t, path2devno(c->qid.path) - Qfirst, Mustexist);

    if(off >= mp->size){
        res = 0;
        goto Done;
    }
    if(off + n > mp->size)
        n = mp->size - off;
    if(n == 0){
        res = 0;
        goto Done;
```

```
    }

    switch(mp->type){
    case Fcat:
        res = catio(mp, Isread, a, n, off);
        break;
    case Finter:
        res = interio(mp, Isread, a, n, off);
        break;
    case Fpart:
        res = io(mp, mp->inner[0], Isread, a, n, mp->start + off);
        break;
    case Fmirror:
        retry = 0;
        do {
            if (retry > 0) {
                print("#k/%s: retry %d read for byte %,lld "
                    "count %ld: %s\n", mp->name, retry, off,
                    n, (up && up->errstr? up->errstr: ""));
                /*
                 * pause before retrying in case it's due to
                 * a transient bus or controller problem.
                 */
                tsleep(&up->sleepr, returnfalse, 0, Retrypause);
            }
            for (i = 0; i < mp->ndevs; i++){
                if (waserror())
                    continue;
                l = io(mp, mp->inner[i], Isread, a, n, off);
                poperror();
                if (l >= 0){
                    res = l;
                    break;      /* read a good copy */
                }
            }
        } while (i == mp->ndevs && ++retry <= Maxretries);
        if (retry > Maxretries) {
            /* no mirror had a good copy of the block */
            print("#k/%s: byte %,lld count %ld: CAN'T READ "
                "from mirror: %s\n", mp->name, off, n,
                (up && up->errstr? up->errstr: ""));
            error(Eio);
        } else if (retry > 0)
            print("#k/%s: byte %,lld count %ld: retry read OK "
                "from mirror: %s\n", mp->name, off, n,
                (up && up->errstr? up->errstr: ""));
        break;
    }
Done:
    poperror();
    runlock(&lck);
    return res;
}

static long
mwrite(Chan *c, void *a, long n, vlong off)
{
    int i, allbad, anybad, retry;
    long    l, res;
    Fsdev   *mp;
```

```
Tree    *t;

dprint("mwrite %llux\n", c->qid.path);
if (c->qid.type & QTDIR)
    error(Eisdir);
if (c->qid.path == Qctl){
    mconfig(a, n);
    return n;
}

rlock(&lck);
if(waserror()){
    runlock(&lck);
    nexterror();
}

t = gettree(path2treeno(c->qid.path), Mustexist);
mp = getdev(t, path2devno(c->qid.path) - Qfirst, Mustexist);

if(off >= mp->size){
    res = 0;
    goto Done;
}
if(off + n > mp->size)
    n = mp->size - off;
if(n == 0){
    res = 0;
    goto Done;
}
res = n;
switch(mp->type){
case Fcat:
    res = catio(mp, Iswrite, a, n, off);
    break;
case Finter:
    res = interio(mp, Iswrite, a, n, off);
    break;
case Fpart:
    res = io(mp, mp->inner[0], Iswrite, a, n, mp->start + off);
    if (res != n)
        error(Eio);
    break;
case Fmirror:
    retry = 0;
    do {
        if (retry > 0) {
            print("#k/%s: retry %d write for byte %,lld "
                "count %ld: %s\n", mp->name, retry, off,
                n, (up && up->errstr? up->errstr: ""));
            /*
             * pause before retrying in case it's due to
             * a transient bus or controller problem.
             */
            tsleep(&up->sleepr, returnfalse, 0, Retrypause);
        }
        allbad = 1;
        anybad = 0;
        for (i = mp->ndevs - 1; i >= 0; i--){
            if (waserror()) {
                anybad = 1;
```

```
                continue;
            }
            l = io(mp, mp->inner[i], Iswrite, a, n, off);
            poperror();
            if (l == n)
                allbad = 0; /* wrote a good copy */
            else
                anybad = 1;
        }
    } while (anybad && ++retry <= Maxretries);
    if (allbad) {
        /* no mirror took a good copy of the block */
        print("#k/%s: byte %,lld count %ld: CAN'T WRITE "
            "to mirror: %s\n", mp->name, off, n,
            (up && up->errstr? up->errstr: ""));
        error(Eio);
    } else if (retry > 0)
        print("#k/%s: byte %,lld count %ld: retry wrote OK "
            "to mirror: %s\n", mp->name, off, n,
            (up && up->errstr? up->errstr: ""));

    break;
  }
Done:
  poperror();
  runlock(&lck);
  return res;
}
```

⟨*global* fsdevtab 827b⟩

Uses Aopen 305b, Blksize-1746 827c, COPEN 294b, DEVDOTDOT 805, DMDIR, Ebadarg 664d, Egone-1777 827c, Fcat-1739 827c, Fclear-1742 827c, Fdel-1743 827c, Fdisk-1744 827c, Finter-1740 827c, Fmirror-1738 827c, Fpart-1741 827c, Incr-1747 827c, Isread-1753 827c, Iswrite-1752 827c, Maxconf-1756 827c, Maxretries-1759 827c, Mustexist-1755 827c, Ndevs-1757 827c, Ntrees-1758 827c, ORDWR 301c, OREAD 853i, OTRUNC, Optional-1754 827c, QTDIR 70b, QTFILE, Qctl-1750 827c, Qdir-1749 827c, Qfirst-1751 827c, Qtop-1748 827c, Retrypause-1760 827c, Sectorsz-1745 827c, catio() 827c, cclose, cfgstr-1772 827c, configs-1776 827c, confstr-1771 827c, cqid-1774 827c, debug-1761 827a, decref() 87c, deltree() 827c, devalloc() 827c, devattach() 297b, devdir() 427c, devdirread() 432c, devpermcheck() 432d, devwalk() 429b, disk-1768 827c, dprint-1762 827a, eve 591a, false, free() 104a, fsdevtab 827b, fstree-1764 827c, getattrs() 827c, getdev() 827c, gettree() 827c, incref() 87b, interio() 827c, kstrdup() 669b, lck-1763 827c, lookupcmd() 595b, lookuptree() 827c, mallocz() 104d, mconfig() 827c, mdelctl() 827c, mdeldev() 827c, mgen() 827c, namec() 305f, nelem 853i, nexterror, ntrees-1766 827c, openmode() 302b, parsecmd() 594c, parseconfig() 827c, path2dev() 827c, path2devno() 827c, path2treeno() 827c, poperror 666d, print, qidvers-1767 827c, readstr() 670d, realloc() 105b, returnfalse() 104c, rlock() 83e, runlock() 84b, sectorsz-1770 827c, seprintdev() 827c, setdsize() 827c, smalloc() 104b, source-1769 827c, tnames-1775 827c, tqid-1773 827c, treealloc() 827c, trees-1765 827c, true, tsleep, up, validdevname() 827c, validname() 308a, waserror 666d, wlock() 85b, and wunlock() 85d.

E.13 init/

init/portfns_init.h

⟨portfns_init.h 848⟩≡

```
// rebootcmd.c
void    rebootcmd(int, char**);

//in <arch>/main.c (but called from syswrite)
void    arch_reboot(kern_addr3, kern_addr3, ulong);
```

```
  //TODO: should add prefix arch_ (or conf_ ?)
  // in <arch>/<conf>.c (called from main)
  void  links(void);
  // in <arch>/<conf>.root.c (called from main)
  void    bootlinks(void);

  // in <arch>/?? (called from main) but different prototypes
  //void  arch_touser(void*);
```

init/user/preboot/initcode.c

init/user/boot/aux.c

init/user/boot/boot.c

init/user/boot/boot.h

⟨boot.h 849a⟩≡

```
  /* libc equivalent */
  // aux.c:
  extern void fatal(char*);
  extern void warning(char*);
  extern void run(char *file, ...);
  extern void runv(char **argv);
  /* pad style */
  void bind_safe(char* old, char* new, int flag);
  int open_safe(char* path, int flag);
  void print_safe(int fd, char* str);
  void close_safe(int fd);

  /* methods */
  // local.c:
  extern void  connectlocal(void);
  //extern void configlocal(Method*);
  // other possible method connections: network.c, ...

  // <arch>/boot$CONF.c
  //extern Method method[];   /* defined in ../$arch/boot$CONF.c */
  //extern char*  bootdisk;   /* defined in ../$arch/boot$CONF.c */
  //void main(int argc, char **argv)

  /* main entry point */
  // boot.c:
  extern void boot(int, char **);
```

init/user/boot/local.c

init/rebootcmd.c

⟨init/rebootcmd.c 849b⟩≡

⟨*kernel basic includes* 49a⟩

```
  #include    <a.out.h>
  #include    <elf.h>
```

```
struct Execvals {
  uvlong  entry;
  ulong textsize;
  ulong datasize;
};
//TODO: who sets this? it's used by rebootcmd but who sets it? qemu?
int (*parseboothdr)(Chan *, ulong, Execvals *);


enum {
    Ehdr32sz    = 52,
    Phdr32sz    = 32,
    Shdr32sz    = 40,

    Ehdr64sz    = 64,
    Phdr64sz    = 56,
    Shdr64sz    = 64,
};

static uchar elfident[] = {
    '\177', 'E', 'L', 'F',
};

static void
readn(Chan *c, void *vp, long n)
{
    char *p = vp;
    long nn;

    while(n > 0) {
        nn = devtab[c->type]->read(c, p, n, c->offset);
        if(nn == 0)
            error(Eshort);
        c->offset += nn;
        p += nn;
        n -= nn;
    }
}

/* assume the elf header is in the byte order of this machine */
int
readelfhdr(Chan *c, ulong, Execvals *evp)
{
    Ehdr ehdr;
    Phdr phdrs[3];

    c->offset = 0;          /* back up */
    readn(c, &ehdr, sizeof ehdr);
    if(memcmp(&ehdr.ident[MAG0], elfident, sizeof elfident) != 0 ||
        ehdr.ident[CLASS] != ELFCLASS32)
        return -1;

    /* get textsize and datasize from Phdrs */
    readn(c, phdrs, sizeof phdrs);
    evp->entry = ehdr.elfentry;
    evp->textsize = phdrs[0].filesz;
    evp->datasize = phdrs[1].filesz;
    c->offset = ROUNDUP(Ehdr32sz + 3*Phdr32sz, 16); /* position for text */
    return 0;
}
```

```
//static int
//readelf64hdr(Chan *c, ulong, Execvals *evp)
//{
//    E64hdr ehdr;
//    P64hdr phdrs[3];
//
//    c->offset = 0;          /* back up */
//    readn(c, &ehdr, sizeof ehdr);
//    if(memcmp(&ehdr.ident[MAG0], elfident, sizeof elfident) != 0 ||
//        ehdr.ident[CLASS] != ELFCLASS64)
//        return -1;
//
//    /* get textsize and datasize from Phdrs */
//    readn(c, phdrs, sizeof phdrs);
//    evp->entry = ehdr.elfentry;
//    evp->textsize = phdrs[0].filesz;
//    evp->datasize = phdrs[1].filesz;
//    c->offset = ROUNDUP(Ehdr64sz + 3*Phdr64sz, 16); /* position for text */
//    return 0;
//}

static void
setbootcmd(int argc, char *argv[])
{
    char *buf, *p, *ep;
    int i;

    buf = malloc(1024);
    if(buf == nil)
        error(Enomem);
    p = buf;
    ep = buf + 1024;
    for(i=0; i<argc; i++)
        p = seprint(p, ep, "%q ", argv[i]);
    *p = 0;
    ksetenv("bootcmd", buf, 1);
    free(buf);
}

void
rebootcmd(int argc, char *argv[])
{
    Chan *c;
    Exec exec;
    Execvals ev;
    ulong magic, text, rtext, entry, data, size;
    uchar *p;

    if(argc == 0)
        arch_exit(0);

    c = namec(argv[0], Aopen, OEXEC, 0);
    if(waserror()){
        cclose(c);
        nexterror();
    }

    readn(c, &exec, sizeof(Exec));
    magic = l2be(exec.magic);
```

```
    /*
     * AOUT_MAGIC is sometimes defined like this:
     * #define AOUT_MAGIC   V_MAGIC || magic==M_MAGIC
     * so we can only use it in a fairly stylized manner.
     */
    if(magic == AOUT_MAGIC) {
        entry = l2be(exec.entry);
        text = l2be(exec.text);
        data = l2be(exec.data);
    } else if(parseboothdr && (*parseboothdr)(c, magic, &ev) >= 0 ||
        readelfhdr(c, magic, &ev) >= 0
        //|| readelf64hdr(c, magic, &ev) >= 0
        ){
        entry = ev.entry;
        text = ev.textsize;
        data = ev.datasize;
    } else {
        error(Ebadexec);
        return;             /* for the compiler */
    }

    /* round text out to page boundary */
    rtext = ROUNDUP(entry+text, BY2PG) - entry;
    size = rtext + data;
    p = malloc(size);
    if(p == nil)
        error(Enomem);

    if(waserror()){
        free(p);
        nexterror();
    }

    memset(p, 0, size);
    readn(c, p, text);
    readn(c, p + rtext, data);

    ksetenv("bootfile", argv[0], 1);
    setbootcmd(argc-1, argv+1);

    arch_reboot((void*)entry, p, size);

    panic("return from reboot!");
}
```

Uses Ehdr32sz-2416 849b, Eshort 664d, Phdr32sz-2417 849b, ROUNDUP 47f, elfident-2422 849b, free() 104a, ksetenv() 598b, malloc() 103c, and readn() 849b.

E.14 interrupts/

interrupts/portfns_interrupts.h

⟨portfns_interrupts.h 852⟩≡

```
// <arch>/trap.c (used in arch specific)
void  arch_intrenable(int, void (*)(Ureg*, void*), void*, int, char*);

// <arch>/trap.c (called from main)
void  arch__trapinit(void);
```

E.15 lib/

lib/lib.h

⟨*lib.h forward decl* 853a⟩≡ (853i)
```
typedef struct Qid  Qid;
typedef struct DirEntry  DirEntry;
typedef struct DirEntry  Dir; // for fcall.h
typedef struct Waitmsg  Waitmsg;
typedef struct Fmt  Fmt;
```
Uses Fmt 853a and Waitmsg 853a.

⟨*constant* ERRMAX 853b⟩≡ (853d)
```
ERRMAX = 128, /* max length of error string */
```

⟨*constant* KNAMELEN 853c⟩≡ (853d)
```
KNAMELEN = 28,  /* max length of name held in kernel */
```

⟨*enum* miscsize 853d⟩≡ (853i)
```
enum Miscsize {
    ⟨constant ERRMAX 853b⟩
    ⟨constant KNAMELEN 853c⟩
};
```

⟨*function* nelem 853e⟩≡ (853i)
```
#define nelem(x)  (sizeof(x)/sizeof((x)[0]))
```

⟨*function* offsetof 853f⟩≡ (853i)
```
//@Scheck: not dead, actually exanded in macros.h
#define offsetof(s, m)  (ulong)(&(((s*)0)->m))
```

⟨*function* assert 853g⟩≡ (853i)
```
#define assert(x) do { if(x) {}else _assert("x");}while(0)
```

⟨*lib.h print decl* 853h⟩≡ (853i)
```
// This used to be regular function, but to avoid backward deps in the kernel
// I made it into a pointer function (a bit ugly, and maybe unsafe)
extern  int (*print)(char*, ...);
```

⟨lib/lib.h 853i⟩≡
```
// coupling: libc.h
//
// This file mostly exports code from lib_core/libc/ (linked with the kernel).
// Those functions are also exported in include/libc.h but better to
// not include it and instead to reexport here a subset specific to the kernel.
//
// There are also the poolxxx() functions exported in include/pool.h
// (also part of libc) that are used in memory/alloc.c.
//
// Some functions in libc are also "overriden" (via linker abuse):
//  - print.c overrides _fmtlock(), _fmtunlock() that were in libc/fmt/
//  - devcons.s overrides the sysfatal() that was in libc/9sys/

// pad specific, could be in u.h
⟨pad basic types ??⟩
```

```
//#define nil (void*)0 in u.h
#define nilptr 0

⟨lib.h forward decl 853a⟩

/*
 * functions (mostly) linked in from libc.
 */
⟨function nelem 853e⟩
⟨function offsetof 853f⟩
⟨function assert 853g⟩

/*
 * mem routines
 */
⟨lib.h mem functions decl 668a⟩

/*
 * string routines
 */
⟨lib.h string functions decl 668c⟩

//redefined in the kernel
extern  int cistrcmp(char*, char*);
extern  int cistrncmp(char*, char*, int);

⟨enum utf 672a⟩

/*
 * rune routines
 */
⟨lib.h rune functions decl 672b⟩

extern  int abs(int);

/*
 * print routines
 */
typedef int (*Fmts)(Fmt*);
⟨struct Fmt 671a⟩

⟨lib.h print decl 853h⟩

⟨lib.h printf functions decl 670e⟩

⟨lib.h pragmas 671c⟩

⟨lib.h fmt functions decl 671b⟩

/*
 * one-of-a-kind
 */
⟨lib.h strto functions decl 670b⟩

//todo: should return a (void*)
extern  ulong getcallerpc(void*);
extern  char* cleanname(char*);
extern  int getfields(char*, char**, int, int, char*);
extern  int tokenize(char*, char**, int);
extern  void  qsort(void*, long, long, int (*)(void*, void*));
```

⟨*lib.h exxx decl* 274b⟩

```
/*
 * Syscall data structures
 */
```
⟨*enum* `mount` 343⟩

⟨*enum* `open` 301c⟩

⟨*enum* `note` 449b⟩

⟨*enum* `miscsize` 853d⟩

⟨*enum* `qidtype` 70b⟩

⟨*enum* `dirmode` 337a⟩

⟨*struct* `Qid` 70a⟩

⟨*struct* `DirEntry` 336c⟩

⟨*struct* `Waitmsg` 149e⟩

```
// used only by devtls.c
extern  int dec64(uchar*, int, char*, int);
extern  int encodefmt(Fmt*);
```
Uses `DirEntry` 853i and `Qid` 853i.

E.16 memory/

memory/portdat_memory.h

⟨`memory/portdat_memory.h` 855⟩≡

```
// see also Arch_KMap in <arch>/ (but used in port)

//*****************************************************************************
// Page < Pagetable (can be filled by KImage) < Segment
//*****************************************************************************

// All the ref<Page> here are references to Pages in the array<Page> of
// Palloc.pages (pool allocator)
// All the ref<Kimage> here are references to KImage in the ?? of
// Imagealloc.free?
```

⟨*enum* `modref` 108b⟩

⟨*enum* `Cachectl` 554f⟩

⟨*struct* `Page` 108a⟩

⟨*type* `PageOrSwap` 206b⟩

⟨*struct* Pagetable 114b⟩

```
// used to be arch-specific, but many constants were used in port/ code
```
⟨*type* PTExxx 184⟩

⟨*struct* KImage 193a⟩

⟨*enum* segtype 112b⟩

⟨*constant* PG_ONSWAP 206a⟩

⟨*function* onswap 206c⟩
⟨*function* pagedout 197a⟩
⟨*function* swapaddr 216a⟩

```
#define SEGMAXSIZE  (PAGEDIRSIZE*PAGETABMAPMEM)
```

⟨*struct* Physseg 556d⟩

```
enum
{
    ⟨constant LRESPROF 505f⟩
};
```

⟨*struct* Segment 113b⟩

```
//*********************************************************************************
// Internal to memory/
//*********************************************************************************

// See also RMap in 386/

// actually internal to xalloc.c, but important so here
```
⟨*constants holes* 93c⟩

⟨*struct* Hole 94⟩

⟨*struct* Xhdr 95b⟩

⟨*struct* Xalloc 93a⟩
```
//IMPORTANT: static Xalloc xlists; // private to xalloc.c


// from pool.h
//struct Pool {
//  char* name;
//  ulong maxsize;
//
//  ulong cursize;
//  ulong curfree;
//  ulong curalloc;
//
//  ulong minarena; /* smallest size of new arena */
//  ulong quantum;  /* allocated blocks should be multiple of */
//  ulong minblock; /* smallest newly allocated block */
//
//  void* freeroot; /* actually Free* */
//  void* arenalist;  /* actually Arena* */
//
//  void* (*alloc)(ulong);
```

```
//  int (*merge)(void*, void*);
//  void  (*move)(void* from, void* to);
//
//  int flags;
//  int nfree;
//  int lastcompact;
//
//  void  (*lock)(Pool*);
//  void  (*unlock)(Pool*);
//  void  (*print)(Pool*, char*, ...);
//  void  (*panic)(Pool*, char*, ...);
//  void  (*logstack)(Pool*);
//
//  void* private;
//};

// exported by libc include/pool.h, used by malloc, defined in pool.c in this dir
// memory pools for malloc()/free() (using xalloc pools)
//IMPORTANT: extern Pool*  mainmem;
// memory pools for ??
//IMPORTANT: extern Pool*  imagmem;
```

⟨*struct* Pallocmem 109d⟩

⟨*function* pghash 190b⟩

⟨*struct* Palloc 108c⟩

```
extern  Palloc  palloc;
```

⟨*constant* NFREECHAN 285b⟩

⟨*function* ihash 194a⟩

⟨*struct* Imagealloc 193b⟩

```
//IMPORTANT: static struct Imagealloc imagealloc; (segment.c)
// so have conf.nimage + 1 Kimages
extern  KImage  swapimage;
```

⟨*struct* Swapalloc 204a⟩

```
extern struct Swapalloc swapalloc;
```

memory/portfns_memory.h

⟨portfns_memory.h 857⟩≡

```
// xalloc.c
void* xalloc(ulong);
void  xfree(void*);
void  xsummary(void);
int   xmerge(void*, void*);
void  xinit(void);
void* xspanalloc(ulong, int, ulong);

// pool.c
// see include/pool.h: poollock(), ...
void  mallocsummary(void);

// alloc.c
```

```
void* malloc(ulong);
void  free(void*);
void* smalloc(ulong);
void* mallocz(ulong, int);
void* mallocalign(ulong, ulong, long, ulong);
void* realloc(void *v, ulong size);
ulong msize(void*);
void  setmalloctag(void*, ulong);
void  kstrcpy(char*, char*, int);
void  kstrdup(char**, char*);

// page.c
void  pageinit(void);
void  pagechainhead(Page*);
void  copypage(Page*, Page*); //TODO: why fp no-deps in codegraph?
void  cachepage(Page *p, KImage *i);
void  uncachepage(Page*);
void  cachedel(KImage*, ulong);
Pagetable*    ptalloc(void);
Pagetable*    ptcpy(Pagetable*);
void  freept(Segment*, Pagetable*);
ulong pagenumber(Page*);
Page* lookpage(KImage*, ulong);
void  putpage(Page*);
Page* auxpage(void);
Page* newpage(int, Segment **, ulong);
int   duppage(Page*);
void  checkpagerefs(void);

// swap.c
void  swapinit(void);
void  putswap(Page*);
void  dupswap(Page*);
int   swapcount(ulong);
void  pagersummary(void);
void  setswapchan(Chan*);
void  kickpager(void);

// fault.c
Segment*  seg(Proc*, ulong, int);
void  checkpages(void);
void  validaddr(ulong, ulong, bool);
int   okaddr(ulong, ulong, int);
int   fixfault(Segment*, ulong, int, int);
void* vmemchr(virt_addr3, int, int);
int   fault(ulong, int);

// segment.c
Segment*  newseg(int, ulong, ulong);
void  relocateseg(Segment*, ulong);
void  segpage(Segment*, Page*);
void  putimage(KImage*);
void  mfreeseg(Segment*, ulong, int);
void  segclock(ulong);
void  putseg(Segment*);
Segment*  dupseg(Segment**, int, int);
long  ibrk(ulong, int);
int   addphysseg(Physseg*);
ulong segattach(Proc*, ulong, char *, ulong, ulong);
void  imageinit(void);
```

```
KImage*   attachimage(int, Chan*, ulong, ulong);

// sysfile.c
// syssetflush (used in syscalls/ without requiring extern decl)

// defined in <arch>/mmu.c (used in port)
Arch_KMap* arch_kmap(Page*);
void  arch_kunmap(Arch_KMap*);
void  arch_flushmmu(void);
void  arch_checkmmu(ulong va, ulong pa);
void  arch_putmmu(ulong, ulong, Page*);
void  arch_mmurelease(Proc*);
void  arch_mmuswitch(Proc*);
ulong arch_cankaddr(ulong);
// actually KADDR is used in port, but it's expanding to kaddr
kern_addr3 arch_kaddr(phys_addr);
// actually PADDR is used in port, but it's expanding to paddr
phys_addr arch_paddr(kern_addr3);
void  arch_countpagerefs(ulong*, int);

// defined in <arch>/ (used in port)
void  arch_memorysummary(void);

// in <arch>/mmu.c (called from main and other other cpu init function)
void  arch_mmuinit(void);
```

memory/portdat_memory.c

⟨memory/portdat_memory.c 859a⟩≡
```
⟨kernel basic includes 49a⟩

⟨global palloc 109b⟩

⟨global swapalloc 204b⟩
⟨global swapimage 205b⟩
```

memory/xalloc.c

⟨xalloc.c forward decl 859b⟩≡ (859c)
```
void        xhole(ulong, ulong);
```

⟨memory/xalloc.c 859c⟩≡
```
⟨kernel basic includes 49a⟩

⟨xalloc.c forward decl 859b⟩

//*********************************************************************************
// The global
//*********************************************************************************

⟨global xlists 95a⟩

//*********************************************************************************
// Initialization
//*********************************************************************************
```

⟨*function* xinit 282⟩

```
//****************************************************************************
// Functions
//****************************************************************************
```

⟨*function* xspanalloc 98d⟩

⟨*function* xallocz 95e⟩

⟨*function* xalloc 95d⟩

⟨*function* xfree 96a⟩

⟨*function* xmerge 98a⟩

⟨*function* xhole 97⟩

```
//****************************************************************************
// Debugging
//****************************************************************************
```

⟨*function* xsummary 643a⟩

memory/pool.c

⟨*pool.c forward decl* 860a⟩≡ (860b)

```
typedef struct Private  Private;
```

Uses Private 100a.

⟨memory/pool.c 860b⟩≡

⟨*kernel basic includes* 49a⟩

```
#include    <pool.h>
```

⟨*pool.c forward decl* 860a⟩

```
//****************************************************************************
// Concurrency
//****************************************************************************

// See Pool.private, for mutual exclusion on memory pools
```

⟨*pool.c struct Private* 100a⟩

```
//****************************************************************************
// Pool methods
//****************************************************************************
```

⟨*function* poolprint 103a⟩

⟨*function* ppanic 103b⟩

⟨*function* plock 100b⟩

⟨*function* punlock 100c⟩

⟨*function* poolsummary 644a⟩

```
//****************************************************************************
```

```
// The globals
//****************************************************************************
```

⟨*global* pmainmem 101⟩

⟨*global* pimagmem 102a⟩

⟨*global mainmem and imagmem* 102b⟩

⟨*function* mallosummary 643b⟩

memory/alloc.c

⟨*alloc.c debugging macro* 861a⟩≡ (861b)

```
/* non tracing
 *
enum {
    Npadlong    = 0,
    MallocOffset = 0,
    ReallocOffset = 0,
};
 *
 */
```

⟨memory/alloc.c 861b⟩≡

⟨*kernel basic includes* 49a⟩

```
#include    <pool.h>

/* everything from here down should be the same in libc, libdebugmalloc, and the kernel */
/* - except the code for malloc(), which alternately doesn't clear or does. */
/* - except the code for smalloc(), which lives only in the kernel. */

//****************************************************************************
// Debugging support
//****************************************************************************

/*
 * Npadlong is the number of 32-bit longs to leave at the beginning of
 * each allocated buffer for our own bookkeeping.  We return to the callers
 * a pointer that points immediately after our bookkeeping area.  Incoming pointers
 * must be decremented by that much, and outgoing pointers incremented.
 * The malloc tag is stored at MallocOffset from the beginning of the block,
 * and the realloc tag at ReallocOffset.  The offsets are from the true beginning
 * of the block, not the beginning the caller sees.
 *
 * The extra if(Npadlong != 0) in various places is a hint for the compiler to
 * compile out function calls that would otherwise be no-ops.
 */

/* tracing */
enum {
    Npadlong    = 2,
    MallocOffset = 0,
    ReallocOffset = 1
};
```

⟨*alloc.c debugging macro* 861a⟩

⟨*function* setmalloctag 106a⟩

⟨*function* setrealloctag 106b⟩

```
//*****************************************************************************
// Pool wrappers
//*****************************************************************************
```

⟨*function* smalloc 104b⟩

⟨*function* malloc 103c⟩

⟨*function* mallocz 104d⟩

⟨*function* mallocalign 105a⟩

⟨*function* free 104a⟩

⟨*function* realloc 105b⟩

⟨*function* msize 436a⟩

```
//*****************************************************************************
// kstr functions
//*****************************************************************************

//pad: was in chan.c
```

⟨*function* kstrcpy 669a⟩

⟨*function* kstrdup 669b⟩

memory/fault.c

⟨*function* checkpages 862⟩≡ (863b)

```
void
checkpages(void)
{
    int checked;
    ulong addr, off;
    Pagetable *p;
    Page *pg;
    Segment **sp, **ep, *s;

    if(up == nil)
        return;

    checked = 0;
    // foreach(up->seg)
    for(sp=up->seg, ep=&up->seg[NSEG]; sp<ep; sp++){
        s = *sp;
        if(s == nil)
            continue;
        qlock(&s->lk);
        for(addr=s->base; addr<s->top; addr+=BY2PG){
            off = addr - s->base;
            p = s->pagedir[off/PAGETABMAPMEM];
```

```
            if(p == nil)
                continue;
            pg = p->pagetab[(off&(PAGETABMAPMEM-1))/BY2PG];
            if(pg == nil || pagedout(pg))
                continue;
            arch_checkmmu(addr, pg->pa);
            checked++;
        }
        qunlock(&s->lk);
    }
    print("%ld %s: checked %d page table entries\n", up->pid, up->text, checked);
}
```

Uses NSEG 118c, pagedout 855, print, qlock() 81f, qunlock() 82c, and up.

⟨*fault.c forward decl* 863a⟩≡ (863b)

```
void        pio(Segment *, ulong, ulong, Page **);
```

⟨memory/fault.c 863b⟩≡

⟨*kernel basic includes* 49a⟩

⟨*fault.c forward decl* 863a⟩

⟨*function* fault 183⟩

⟨*function* faulterror 199a⟩

```
// for debugging SG_PHYSICAL
void    (*checkaddr)(ulong, Segment *, Page *);
ulong   addr2check;
```

⟨*function* fixfault 185⟩

⟨*function* pio 197d⟩

⟨*function* okaddr 119c⟩

⟨*function* validaddr 120b⟩

⟨*function* vmemchr 120c⟩

⟨*function* seg 119b⟩

⟨*function* checkpages 862⟩

memory/page.c

⟨*page.c forward decl* 863c⟩≡ (863d)

```
int hasfreepages(void*);
void portcountpagerefs(ulong*, int);
```

⟨memory/page.c 863d⟩≡

⟨*kernel basic includes* 49a⟩

⟨*page.c forward decl* 863c⟩

```
//*********************************************************************
// Initialization
//*********************************************************************
```

⟨*function* `pageinit` 283c⟩

```
//*****************************************************************************
// Functions
//*****************************************************************************
```

⟨*function* `pageunchain` 110d⟩

⟨*function* `pagechaintail` 112a⟩

⟨*function* `pagechainhead` 111b⟩

⟨*constructor* `newpage` 110a⟩

⟨*function* `hasfreepages` 211b⟩

⟨*destructor* `putpage` 111a⟩

⟨*function* `auxpage` 816c⟩

⟨*global* `dupretries` 199c⟩

⟨*function* `duppage` 199d⟩

⟨*function* `copypage` 189c⟩

⟨*function* `uncachepage` 192a⟩

⟨*function* `cachepage` 190e⟩

⟨*function* `cachedel` 192b⟩

⟨*function* `lookpage` 191a⟩

⟨*function* `ptcpy` 115c⟩

⟨*constructor* `ptalloc` 115a⟩

⟨*destructor* `freept` 115b⟩

⟨*function* `pagenumber` 560a⟩

⟨*function* `checkpagerefs` 558c⟩

⟨*function* `portcountpagerefs` 559c⟩

memory/segment.c

⟨*segment.c forward decl* 864a⟩≡ (864b)

```
static void imagereclaim(void);
static void imagechanreclaim(void);
Segment* data2txt(Segment *s);
```

⟨`memory/segment.c` 864b⟩≡

```
⟨kernel basic includes 49a⟩
#include "io.h"

//*****************************************************************************
```

```
// Global
//****************************************************************************
```

⟨*global* `imagealloc` 194b⟩

```
//****************************************************************************
// Misc
//****************************************************************************
```

⟨*global* `physseg` 556f⟩
⟨*global* `physseglock` 556g⟩

⟨*hook* `_globalsegattach` 468b⟩

⟨*segment.c forward decl* 864a⟩

```
//****************************************************************************
// Initialization
//****************************************************************************
```

⟨*function* `imageinit` 285c⟩

```
//****************************************************************************
// Functions
//****************************************************************************
```

⟨*constructor* `newseg` 116a⟩

⟨*destructor* `putseg` 116c⟩

⟨*function* `relocateseg` 147b⟩

⟨*function* `dupseg` 121a⟩

⟨*function* `segpage` 118a⟩

⟨*constructor* `attachimage` 194e⟩

⟨*struct* `Irstats` 661j⟩
⟨*segment.c global irstats* 662a⟩

⟨*function* `imagereclaim` 203⟩

⟨*function* `imagechanreclaim` 202b⟩

⟨*destructor* `putimage` 201d⟩

⟨*function* `ibrk` 123b⟩

⟨*function* `mfreeseg` 125c⟩

⟨*function* `isoverlap` 469⟩

⟨*function* `addphysseg` 557a⟩

⟨*function* `segattach` 468c⟩

⟨*clock callback segclock* 506c⟩

```
// was in another file before
```

⟨*function* `data2txt` 196e⟩

memory/swap.c

⟨*swap.c forward decl* 866a⟩≡ (866b)

```
static int  canflush(Proc*, Segment*);
static void executeio(void);
static int  needpages(void*);
static void pageout(Proc*, Segment*);
static void pagepte(int, Page**);
static void pager(void*);
```

⟨`memory/swap.c` 866b⟩≡

⟨*kernel basic includes* 49a⟩

⟨*swap.c forward decl* 866a⟩

⟨*global* `iolist` 208b⟩
⟨*global* `ioptr` 208c⟩

⟨*global* `genxxx` 214c⟩

⟨*function* `gentick` 214b⟩

⟨*function* `swapinit` 284b⟩

⟨*function* `newswap` 205c⟩

⟨*function* `putswap` 206d⟩

⟨*function* `dupswap` 207a⟩

⟨*function* `swapcount` 207c⟩

⟨*function* `kickpager` 211c⟩

⟨*function* `pager` 211d⟩

⟨*function* `pageout` 208d⟩

⟨*function* `canflush` 213f⟩

⟨*function* `pagepte` 207d⟩

⟨*function* `pagersummary` 644c⟩

⟨*function* `pageiocomp` 215b⟩

⟨*function* `executeio` 215a⟩

⟨*function* `needpages` 213a⟩

⟨*function* `setswapchan` 216c⟩

memory/sysmemory.c

⟨`memory/sysmemory.c` 866c⟩≡

⟨*kernel basic includes* 49a⟩

```
// see also sysbrk in sysproc.c

// those functions used to be in segment.c

void
ptflush(Pagetable *pt, int s, int e)
{
 int i;
 Page *p;

 for(i = s; i < e; i++) {
  p = pt->pagetab[i];
  if(pagedout(p) == 0)
   memset(p->cachectl, PG_TXTFLUSH, sizeof(p->cachectl));
 }
}
```

⟨*syscall segflush* 472c⟩

Uses PG_TXTFLUSH 554f and pagedout 855.

E.17 misc/

misc/portdat_misc.h

⟨misc/portdat_misc.h 867a⟩≡

```
enum misc_constants_portable {
    /* READSTR was 1000, which is way too small for usb's ctl file */
    READSTR = 4000,   /* temporary buffer size for device reads */

    KB = 1024,
    MB = (1024*1024),

    NUMSIZE = 12,   /* size of formatted number */

    ⟨constant PRINTSIZE 391a⟩
};
```

⟨*struct* Cmdbuf 594b⟩

⟨*struct* Cmdtab 594a⟩

misc/portfns_misc.h

⟨portfns_misc.h 867b⟩≡

```
// parse.c
Cmdbuf*  parsecmd(char *a, int n);
Cmdtab*  lookupcmd(Cmdbuf*, Cmdtab*, int);
void     cmderror(Cmdbuf*, char*);

// random.c
void     randominit(void);
```

```
ulong    randomread(void*, ulong);
```

misc/parse.c

⟨misc/parse.c 868a⟩≡
⟨*kernel basic includes* 49a⟩

⟨*ncmdfield* 596⟩

⟨*function* parsecmd 594c⟩

⟨*function* cmderror 595a⟩

⟨*function* lookupcmd 595b⟩

misc/random.c

⟨misc/random.c 868b⟩≡
⟨*kernel basic includes* 49a⟩

⟨*struct* Rb 634e⟩

⟨*global* rb 634f⟩

⟨*struct* rbnotfull 635a⟩

⟨*struct* rbnotempty 635b⟩

⟨*function* genrandom 635c⟩

⟨*clock callback randomclock* 635d⟩

⟨*function* randominit 636a⟩

⟨*function* randomread 636b⟩

E.18 processes/

processes/portdat_processes.h

⟨processes/portdat_processes.h 868c⟩≡

```
// in lib.h: Waitmsg, ERRMAX

//*****************************************************************************
// Proc components
//*****************************************************************************
// All the ref<Proc> here are references to Proc in the array<Proc> of
// Procalloc.arena (pool allocator)

//-----------------------------------------------------------------------
// State
//-----------------------------------------------------------------------

// TODO: state transition diagram
```

⟨*enum* procstate 67c⟩

```
// hash<enum<procstate>, string>, to debug
extern  char* statename[];

//---------------------------------------------------------------------
// Memory
//---------------------------------------------------------------------
```
⟨*enum* procseg 118c⟩

```
//---------------------------------------------------------------------
// Files
//---------------------------------------------------------------------

enum
{
   
```
⟨*constant* DELTAFD 299a⟩
```
};
```

⟨*struct* Fgrp 298a⟩

⟨*function* MOUNTH 346d⟩

⟨*struct* Pgrp 346b⟩

```
//---------------------------------------------------------------------
// System call
//---------------------------------------------------------------------
```

⟨*constant* MAXSYSARG 163a⟩

⟨*struct* Sargs 163b⟩

```
//---------------------------------------------------------------------
// Notes
//---------------------------------------------------------------------

enum {
    
```
⟨*constant* NNOTE 450f⟩
```
};
```

⟨*enum* notekind 449a⟩

⟨*struct* Note 449d⟩
```
extern Counter  noteidalloc;

//---------------------------------------------------------------------
// Process children waiting
//---------------------------------------------------------------------
```

⟨*struct* Waitq 149f⟩

```
//---------------------------------------------------------------------
// Synchronization (Rendez vous)
//---------------------------------------------------------------------
```

⟨*function* REND 479a⟩

⟨*struct* Rgrp 479b⟩

```
//------------------------------------------------------------------
// Alarms, timers
//------------------------------------------------------------------
// now in portdat_time.h

//------------------------------------------------------------------
// Scheduling
//------------------------------------------------------------------

enum {
    ⟨constant Npriq 219a⟩
    ⟨constant Nrq 221a⟩
};

⟨enum priority 219b⟩

⟨enum edfflags 566g⟩

⟨struct Edf 567a⟩

//------------------------------------------------------------------
// Error management
//------------------------------------------------------------------
enum {
    ⟨constant NERR 666a⟩
};

//------------------------------------------------------------------
// Stats, profiling
//------------------------------------------------------------------
⟨enum proctimer 503a⟩

//------------------------------------------------------------------
// Debugger
//------------------------------------------------------------------

⟨enum procctl 484d⟩

//------------------------------------------------------------------
// Misc
//------------------------------------------------------------------

⟨enum fpsavestatus 513a⟩

//*********************************************************************************
// Proc, the big one
//*********************************************************************************

⟨struct Proc 66⟩

⟨macro waserror poperror 666d⟩


//*********************************************************************************
// Internal to process/
//*********************************************************************************

// Proc allocator (singleton), was actually in proc.c, but important so here
⟨struct Procalloc 128c⟩
```

```
//IMPORTANT: static struct Procalloc procalloc; (in proc.c)

⟨struct Schedq 221c⟩
// hash<enum<priority>, Schedq>, Nrq is the number of priority level (20+2)
//IMPORTANT: Schedq  runq[Nrq];  (in proc.c)

// was in alarm.c, but important so here
⟨struct Alarms 256f⟩
//IMPORTANT: static Alarms alarms; (in alarm.c)
//IMPORTANT: static Rendez alarmr; (in alarm.c)

⟨struct Active 65d⟩
extern struct Active active;
```

processes/portfns_processes.h

⟨portfns_processes.h 871⟩≡

```
// proc.c
// (now in portfns_core.h, to remove some backward dependencies)
//void  (*sched)(void);
//void  (*sleep)(Rendez*, int(*)(void*), void*);
//void  (*tsleep)(Rendez*, int (*)(void*), void*, ulong);
//Proc* (*wakeup)(Rendez*);
//int   (*postnote)(Proc*, int, char*, int);
//void  (*error)(char*);
//void  (*nexterror)(void);
//void  (*pexit)(char*, bool);
//void  (*ready)(Proc*);
//void  (*dumpaproc)(Proc*);
//
void    yield(void);
void    hzsched(void);
void    schedinit(void);
Proc*   newproc(void);
void    preempt(void);
void    procinit(void);
int     procindex(ulong);
void    kproc(char*, void(*)(void*), void*);
Proc*   dequeueproc(Schedq*, Proc*);
//
void    scheddump(void);
void    procdump(void);
//
void    exhausted(char*);
void    procctl(Proc*);
void    procwired(Proc*, int);
void    procpriority(Proc*, int, int);
void    accounttime(void);
int     canpage(Proc*);
int     anyhigher(void);
void    killbig(char*);
void    procflushseg(Segment*);
void    renameuser(char*, char*);
ulong   pwait(Waitmsg*);
void    unbreak(Proc*);
int     freebroken(void);
int     haswaitq(void*);
```

```
// pgrp.c
Rgrp*   newrgrp(void);
Pgrp*   newpgrp(void);
void    closergrp(Rgrp*);
void    closefgrp(Fgrp*);
Fgrp*   dupfgrp(Fgrp*);
void    closepgrp(Pgrp*);
void    pgrpcpy(Pgrp*, Pgrp*);
void    forceclosefgrp(void);
void    pgrpnote(ulong, char*, long, int);
void    mountfree(Mount*);
Mount*  newmount(Chan*, int, char*);
void    resrcwait(char*);


// edf.c (used to be declared in edf.h)
Edf*    edflock(Proc*);
void    edfunlock(void);
void    edfrun(Proc*, int);
void    edfinit(Proc*);
void    edfrecord(Proc*);
void    edfstop(Proc*);
void    edfyield(void);
int     edfready(Proc*);
char*   edfadmit(Proc*);

// sysproc.c
// not useless, used by misc/rebootcmd.c, weird not recognized by cg
ulong   l2be(long);
// many sysxxx functions (used in syscalls/ without requiring extern decl)


// <arch>/trap.c (but used in port)
int     arch_userureg(Ureg *);
void    arch_callwithureg(void(*)(Ureg*));
ulong   arch_dbgpc(Proc*);
ulong   arch_userpc(void);
long    arch_execregs(ulong, ulong, ulong);
void    arch_forkchild(Proc*, Ureg*);
void    arch_kprocchild(Proc*, void (*)(void*), void*);
void    arch_setregisters(Ureg*, char*, char*, int);
void    arch_setkernur(Ureg*, Proc*);
void    arch_validalign(uintptr, unsigned); // TODO: should be in memory/?

// in <arch>/forkret.s, called from arch_forkchild
//@Scheck: Assembly
void  arch__forkret(void);


// <arch>/main_processes.c (but used in port)
void  arch_procsetup(Proc*);
void  arch_procsave(Proc*);
void  arch_procrestore(Proc *);
void  arch_idlehands(void);


// in <arch>/l.s (but used in port)
//@Scheck: Assembly
void  arch_gotolabel(Label*);
```

```
  //@Scheck: Assembly
  int   arch_setlabel(Label*);
  //@Scheck: Assembly
  void  arch_mul64fract(uvlong*, uvlong, uvlong);

  // portdat_processes.c
  void (*proctrace)(Proc*, int, vlong); // was in devproc.c
  void (*kproftimer)(ulong); // was in portfns.h
```

processes/portdat_processes.c

⟨processes/portdat_processes.c 873a⟩≡
  ⟨*kernel basic includes* 49a⟩

  ⟨*hook proctrace* 499a⟩
  ⟨*hook kproftimer* 660b⟩

  ⟨*global* `active` 65e⟩

time/portfns_time.h

⟨time/portfns_time.h 873b⟩≡

```
  // portclock.c
  ulong   tk2ms(ulong);
  ⟨macro TK2MS 244d⟩
  ⟨macro MS2NS 244b⟩
  #define TK2SEC(t) ((t)/Arch_HZ) /* ticks to seconds */ // not used in port
  //#define MS2HZ  (1000/Arch_HZ)  /* millisec per clock tick */
  ulong   ms2tk(ulong);
  void    timerdel(Timer*);
  void    timeradd(Timer*);
  Timer*  addclock0link(void (*)(void), int);
  void    timerintr(Ureg*, Tval);
  void    timersinit(void);

  // alarm.c
  ulong   procalarm(ulong);
  void    alarmkproc(void*);
  void    checkalarms(void);

  // tod.c
  // initialize the hidden global tod.c#tod
  void    todinit(void);
  void    todsetfreq(vlong);
  void    todset(vlong, vlong, int);
  vlong   todget(vlong*);
  uvlong  fastticks2us(uvlong);
  uvlong  ns2fastticks(uvlong);
  long    seconds(void);

  // <arch>/clock.c (called from port)
  void   arch_timerset(Tval x); // called from portclock.c (e.g., addclock0link)
  ulong  arch_us(void);         // called from edf.c
  long   arch_lcycles(void); // called from edf.c, sysproc.c, taslock.c
  ulong  arch_perfticks(void);
  // now in core/ for backward deps
  //uvlong arch_fastticks(uvlong *hz); // called from portclock.c
```

```
//void   arch_microdelay(int);
//void   arch_delay(int);
// <arch>/clock.c (called from port but signature not portable across <arch>)
//void arch_cycles(uvlong*);
```

time/portdat_time.h

⟨time/portdat_time.h 874a⟩≡

```
⟨enum timermode 248a⟩

⟨type Txxx 244a⟩

⟨struct Timer 247⟩

// was in clock.c
⟨struct Timers 248b⟩

// used only in arm/ for now
enum {
 Mhz = 1000 * 1000,
};

// <arch>/dat_time.h should defined Arch_HZ

#pragma varargck  type  "t"   long
#pragma varargck  type  "U"   uvlong
```

time/alarm.c

⟨time/alarm.c 874b⟩≡

```
⟨kernel basic includes 49a⟩

⟨global alarms 257a⟩
⟨global alarmr 258d⟩

⟨kernel process alarmkproc 258e⟩

⟨function checkalarms 259b⟩

⟨function procalarm 257e⟩
```

time/portclock.c

⟨time/portclock.c 874c⟩≡

```
⟨kernel basic includes 49a⟩

#include "io.h"

#include <ureg.h>

enum {
   ⟨constant Maxtimerloops 253d⟩
};

⟨global timers 248e⟩
```

⟨*global* `timersinited` 281a⟩

⟨*clock.c statistics* 662c⟩

⟨*function* `tadd` 248f⟩

⟨*function* `tdel` 249d⟩

⟨*function* `timeradd` 250b⟩

⟨*function* `timerdel` 250e⟩

⟨*clock callback hzclock* 236f⟩

⟨*interrupt callback timerintr* 252⟩

⟨*function* `timersinit` 281b⟩

⟨*function* `addclock0link` 253f⟩

⟨*function* `tk2ms` 245a⟩

⟨*function* `ms2tk` 245b⟩

processes/pgrp.c

⟨`processes/pgrp.c` 875⟩≡

```
// Process group, and Namespace.
```

⟨*kernel basic includes* 49a⟩

```
enum {
    ⟨constant Whinesecs 132a⟩
};
```

⟨*global* `pgrpid` 347c⟩
⟨*global* `mountid` 367b⟩

⟨*function* `pgrpnote` 610⟩

⟨*constructor* `newpgrp` 347d⟩

⟨*constructor* `newrgrp` 479d⟩

⟨*destructor* `closergrp` 479e⟩

⟨*destructor* `closepgrp` 347e⟩

⟨*function* `pgrpinsert` 367e⟩

⟨*function* `pgrpcpy` 368a⟩

⟨*function* `dupfgrp` 298c⟩

⟨*function* `closefgrp` 299b⟩

⟨*function* `forceclosefgrp` 319b⟩

⟨*constructor* `newmount` 345b⟩

⟨*destructor* `mountfree` 346a⟩

⟨*function* `resrcwait` 131c⟩

processes/proc.c

⟨*proc.c forward decl* 876a⟩≡ (877c)
```
Proc* runproc(void);
void updatecpu(Proc*);
int reprioritize(Proc*);

static void pidhash(Proc*);
static void pidunhash(Proc*);
static void rebalance(void);
```

⟨`Procctl` *cases* 876b⟩+≡ (484d) ◁502a
```
Proc_exitbig,
```

⟨`procctl()` `Proc_exitbig` *case* 876c⟩≡ (485b)
```
case Proc_exitbig:
    arch_spllo();
    pexit("Killed: Insufficient physical memory", true);
    // Fallthrough
```
Uses `Proc_exitbig` 876b, `pexit`, and `true`.

⟨*function* `killbig` 876d⟩≡ (877c)
```
void
killbig(char *why)
{
    int i;
    Segment *s;
    ulong l, max;
    Proc *p, *ep, *kp;

    max = 0;
    kp = 0;
    ep = procalloc.arena+conf.nproc;
    for(p = procalloc.arena; p < ep; p++) {
        if(p->state == Dead || p->kp)
            continue;
        l = 0;
        for(i=1; i<NSEG; i++) {
            s = p->seg[i];
            if(s != 0)
                l += s->top - s->base;
        }
        if(l > max && ((p->procmode&0222) || strcmp(eve, p->user)!=0)) {
            kp = p;
            max = l;
        }
    }

    print("%lud: %s killed: %s\n", kp->pid, kp->text, why);
    for(p = procalloc.arena; p < ep; p++) {
        if(p->state == Dead || p->kp)
            continue;
        if(p != kp && p->seg[BSEG] && p->seg[BSEG] == kp->seg[BSEG])
            p->procctl = Proc_exitbig;
```

```
    }
    kp->procctl = Proc_exitbig;
    for(i = 0; i < NSEG; i++) {
        s = kp->seg[i];
        if(s != 0 && canqlock(&s->lk)) {
            mfreeseg(s, s->base, (s->top - s->base)/BY2PG);
            qunlock(&s->lk);
        }
    }
}
```

Uses BSEG 118c, Dead 67c, NSEG 118c, Proc_exitbig 876b, canqlock() 83a, conf 62b, eve 591a, mfreeseg() 125c, print, procalloc-2458 128d, and qunlock() 82c.

⟨Proc *scheduling fields* 877a⟩+≡ (66) ◁566d

```
bool preempted;  /* true if this process hasn't finished the interrupt
       *  that last preempted it
       */
```

⟨*function* preempt 877b⟩≡ (877c)

```
/*
 *  here at the end of non-clock interrupts to see if we should preempt the
 *  current process.
 */
void
preempt(void)
{
    if(up && up->state == Running)
      if(up->preempted == false)
        if(anyhigher())
          if(!active.exiting){
              cpu->readied = nil;   /* avoid cooperative scheduling */
              up->preempted = true;
              sched();
              arch_splhi(); // still in interrupt context
              up->preempted = false;
          }
    return;
}
```

Uses Running 67c, active 65e, anyhigher() 236d, false, sched, true, and up.

⟨processes/proc.c 877c⟩≡

```
⟨kernel basic includes 49a⟩
#include    <trace.h>

//*********************************************************************************
// Globals
//*********************************************************************************

⟨global runq 221b⟩
⟨global runveq 221e⟩

⟨global procalloc 128d⟩

⟨global nrdy 222a⟩
⟨global noteidalloc 449c⟩

⟨proc.c statistics 662b⟩

⟨global pidalloc 128b⟩
```

```
enum
{
    ⟨constant Schedagain 239f⟩
    ⟨constant Scaling 239e⟩
};

⟨global statename 67d⟩

⟨proc.c forward decl 876a⟩

//*****************************************************************************
// Error management
//*****************************************************************************

// see also waserror() and poperror() macro in portdat_processes.h

⟨function error 666b⟩

⟨function nexterror 667a⟩

⟨function exhausted 666c⟩

//*****************************************************************************
// Functions
//*****************************************************************************

⟨function schedinit 224f⟩

⟨function sched 223c⟩

⟨function anyready 221f⟩

⟨function anyhigher 236d⟩

⟨function hzsched 236c⟩

⟨function preempt 877b⟩

⟨function updatecpu 239h⟩

⟨function reprioritize 240⟩

⟨function queueproc 222b⟩

⟨function dequeueproc 222c⟩

⟨function ready 232a⟩

⟨function yield 231b⟩

⟨global balancetime 242a⟩

⟨function rebalance 241b⟩

⟨function runproc 225⟩

⟨function canpage 214a⟩

⟨function noprocpanic 131b⟩
```

⟨*constructor* `newproc` 130a⟩

⟨*function* `procwired` 237g⟩

⟨*function* `procpriority` 220a⟩

⟨*function* `procinit` 283a⟩

⟨*function* `sleep` 233e⟩

⟨*function* `tfn` 256d⟩

⟨*function* `twakeup` 256e⟩

⟨*function* `tsleep` 255c⟩

⟨*function* `wakeup` 235a⟩

⟨*function* `postnote` 455⟩

⟨*struct* `Broken` 483a⟩
⟨*global* `broken` 483b⟩

⟨*function* `addbroken` 483d⟩

⟨*function* `unbreak` 483e⟩

⟨*function* `freebroken` 484a⟩

⟨*function* `pexit` 152c⟩

⟨*function* `haswaitq` 151d⟩

⟨*function* `pwait` 150c⟩

⟨*function* `proctab` 604b⟩

⟨*function* `dumpaproc` 645b⟩

⟨*function* `procdump` 645a⟩

⟨*function* `procflushseg` 471d⟩

⟨*function* `scheddump` 644f⟩

⟨*function* `kproc` 157c⟩

⟨*function* `procctl` 485b⟩

⟨*function* `killbig` 876d⟩

⟨*function* `renameuser` 133a⟩

⟨*function* `accounttime` 657b⟩

⟨*function* `pidhash` 128g⟩

⟨*function* `pidunhash` 129a⟩

⟨*function* `procindex` 129b⟩

time/tod.c

⟨*tod.c forward decl* 880a⟩≡ (880b)
```
uvlong      mk64fract(uvlong, uvlong);
static void todfix(void);
```

⟨time/tod.c 880b⟩≡
```
// TOD: Time Of Day.

⟨kernel basic includes 49a⟩

⟨tod.c forward decl 880a⟩

/*
 * Compute nanosecond epoch time from the fastest ticking clock
 * on the system.  Converting the time to nanoseconds requires
 * the following formula
 *
 *  t = (((1000000000<<31)/f)*ticks)>>31
 *
 *  where
 *
 *  'f'     is the clock frequency
 *  'ticks'     are clock ticks
 *
 *  to avoid too much calculation in todget(), we calculate
 *
 *  mult = (1000000000<<32)/f
 *
 *  each time f is set.  f is normally set by a user level
 *  program writing to /dev/fastclock.  arch_mul64fract will then
 *  take that fractional multiplier and a 64 bit integer and
 *  return the resulting integer product.
 *
 *  We assume that the cpu's of a multiprocessor are synchronized.
 *  This assumption needs to be questioned with each new architecture.
 */

/* frequency of the tod clock */
#define TODFREQ     1000000000ULL
#define MicroFREQ   1000000ULL

⟨struct TOD 259c⟩
⟨global tod 260a⟩

⟨function todinit 260d⟩

⟨function todsetfreq 261a⟩

⟨function todset 261b⟩

⟨function todget 262⟩

⟨function todfix 263a⟩

⟨function seconds 263b⟩

⟨function fastticks2us 264a⟩

⟨function ns2fastticks 264b⟩
```

⟨*function* `mk64fract` 264c⟩

processes/sysproc.c

⟨*sysproc.c forward decl* 881a⟩≡ (881b)

```
int shargs(char*, int, char**);
```

⟨`processes/sysproc.c` 881b⟩≡

⟨*kernel basic includes* 49a⟩

```
#include    <tos.h>
#include    <a.out.h>
```

⟨*enum* `rfork` 133b⟩

⟨*sysproc.c forward decl* 881a⟩

⟨*syscall nop* 57d⟩

⟨*syscall rfork* 133c⟩

⟨*function* `l2be` 143b⟩

⟨*syscall exec* 141c⟩

⟨*function* `shargs` 511a⟩

⟨*syscall sleep* 255a⟩

⟨*syscall alarm* 257b⟩

⟨*syscall exits* 151e⟩

⟨*syscall await* 150a⟩

⟨*function* `werrstr` 667e⟩

⟨*function* `generrstr` 667d⟩

⟨*syscall errstr* 667c⟩

⟨*syscall notify* 450h⟩

⟨*syscall noted* 453a⟩

⟨*syscall segbrk* 467c⟩

⟨*syscall segattach* 468a⟩

⟨*syscall segdetach* 470⟩

⟨*syscall segfree* 471a⟩

⟨*syscall brk* 123a⟩

⟨*syscall rendezvous* 480b⟩

processes/syssema.c

⟨processes/syssema.c 882⟩≡
⟨*kernel basic includes* 49a⟩

```
/*
 * The implementation of semaphores is complicated by needing
 * to avoid rescheduling in syssemrelease, so that it is safe
 * to call from real-time processes.  This means syssemrelease
 * cannot acquire any qlocks, only spin locks.
 *
 * Semacquire and semrelease must both manipulate the semaphore
 * wait list.  Lock-free linked lists only exist in theory, not
 * in practice, so the wait list is protected by a spin lock.
 *
 * The semaphore value *addr is stored in user memory, so it
 * cannot be read or written while holding spin locks.
 *
 * Thus, we can access the list only when holding the lock, and
 * we can access the semaphore only when not holding the lock.
 * This makes things interesting.  Note that sleep's condition function
 * is called while holding two locks - r and up->rlock - so it cannot
 * access the semaphore value either.
 *
 * An acquirer announces its intention to try for the semaphore
 * by putting a Sema structure onto the wait list and then
 * setting Sema.waiting.  After one last check of semaphore,
 * the acquirer sleeps until Sema.waiting==0.  A releaser of n
 * must wake up n acquirers who have Sema.waiting set.  It does
 * this by clearing Sema.waiting and then calling wakeup.
 *
 * There are three interesting races here.

 * The first is that in this particular sleep/wakeup usage, a single
 * wakeup can rouse a process from two consecutive sleeps!
 * The ordering is:
 *
 *  (a) set Sema.waiting = 1
 *  (a) call sleep
 *  (b) set Sema.waiting = 0
 *  (a) check Sema.waiting inside sleep, return w/o sleeping
 *  (a) try for semaphore, fail
 *  (a) set Sema.waiting = 1
 *  (a) call sleep
 *  (b) call wakeup(a)
 *  (a) wake up again
 *
 * This is okay - semacquire will just go around the loop
 * again.  It does mean that at the top of the for(;;) loop in
 * semacquire, phore.waiting might already be set to 1.
 *
 * The second is that a releaser might wake an acquirer who is
 * interrupted before he can acquire the lock.  Since
 * release(n) issues only n wakeup calls -- only n can be used
 * anyway -- if the interrupted process is not going to use his
 * wakeup call he must pass it on to another acquirer.
 *
 * The third race is similar to the second but more subtle.  An
 * acquirer sets waiting=1 and then does a final canacquire()
 * before going to sleep.  The opposite order would result in
```

```
 * missing wakeups that happen between canacquire and
 * waiting=1.  (In fact, the whole point of Sema.waiting is to
 * avoid missing wakeups between canacquire() and sleep().) But
 * there can be spurious wakeups between a successful
 * canacquire() and the following semdequeue().  This wakeup is
 * not useful to the acquirer, since he has already acquired
 * the semaphore.  Like in the previous case, though, the
 * acquirer must pass the wakeup call along.
 *
 * This is all rather subtle.  The code below has been verified
 * with the spin model /sys/src/9/port/semaphore.p.  The
 * original code anticipated the second race but not the first
 * or third, which were caught only with spin.  The first race
 * is mentioned in /sys/doc/sleep.ps, but I'd forgotten about it.
 * It was lucky that my abstract model of sleep/wakeup still managed
 * to preserve that behavior.
 *
 * I remain slightly concerned about memory coherence
 * outside of locks.  The spin model does not take
 * queued processor writes into account so we have to
 * think hard.  The only variables accessed outside locks
 * are the semaphore value itself and the boolean flag
 * Sema.waiting.  The value is only accessed with cmpswap,
 * whose job description includes doing the right thing as
 * far as memory coherence across processors.  That leaves
 * Sema.waiting.  To handle it, we call coherence() before each
 * read and after each write.       - rsc
 */
```

⟨*function* semqueue 474e⟩

⟨*function* semdequeue 475a⟩

⟨*function* semwakeup 475b⟩

⟨*function* semrelease 475c⟩

⟨*function* canacquire 476a⟩

⟨*function* semawoke 476b⟩

⟨*function* semacquire 476c⟩

⟨*function* tsemacquire 477a⟩

⟨*syscall semacquire* 477b⟩

⟨*syscall tsemacquire* 478a⟩

⟨*syscall semrelease* 478b⟩

E.19 security/

security/portfns_security.h

⟨portfns_security.h 883⟩≡

```
// auth.c
long  userwrite(char*, int);
long  hostdomainwrite(char*, int);
long  hostownerwrite(char*, int);

// and some sysxxx functions
```

security/auth.c

⟨security/auth.c 884a⟩≡
⟨*kernel basic includes* 49a⟩

```
#include    <authsrv.h>

//now in core/
//char  *eve;
//int iseve(void) { }
```

⟨*global* hostdomain 591d⟩

⟨*syscall fversion* 591e⟩

⟨*syscall fauth* 591f⟩

⟨*function* userwrite 592a⟩

⟨*function* hostownerwrite 132d⟩

⟨*function* hostdomainwrite 592b⟩

E.20 syscalls/

syscalls/systab.h

syscalls/syscallfmt.c

⟨syscalls/syscallfmt.c 884b⟩≡

```
/*
 * Print functions for system call tracing.
 */
```

⟨*kernel basic includes* 49a⟩

```
#include "../port/systab.h"
```

⟨*function* fmtrwdata 497⟩

⟨*function* fmtuserstring 498a⟩

⟨*function* syscallfmt 492e⟩

⟨*function* sysretfmt 496⟩

syscalls/systab.c

⟨*systab.c forward decl* 884c⟩≡ (886a)

```
extern Syscall sysnop;
```

```
extern Syscall sysbind;
extern Syscall syschdir;
extern Syscall sysclose;
extern Syscall sysdup;
extern Syscall sysalarm;
extern Syscall sysexec;
extern Syscall sysexits;
extern Syscall sysfauth;
extern Syscall syssegbrk;
extern Syscall sysopen;
extern Syscall syssleep;
extern Syscall sysrfork;
extern Syscall syspipe;
extern Syscall syscreate;
extern Syscall sysfd2path;
extern Syscall sysbrk;
extern Syscall sysremove;
extern Syscall sysnotify;
extern Syscall sysnoted;
extern Syscall syssegattach;
extern Syscall syssegdetach;
extern Syscall syssegfree;
extern Syscall syssegflush;
extern Syscall sysrendezvous;
extern Syscall sysunmount;
extern Syscall syssemacquire;
extern Syscall syssemrelease;
extern Syscall sysseek;
extern Syscall sysfversion;
extern Syscall syserrstr;
extern Syscall sysstat;
extern Syscall sysfstat;
extern Syscall syswstat;
extern Syscall sysfwstat;
extern Syscall sysmount;
extern Syscall sysawait;
extern Syscall syspread;
extern Syscall syspwrite;
extern Syscall systsemacquire;
```

⟨*global* sysstab 885⟩≡ (886a)

```
char *sysctab[] = {
    [NOP]     "Nop",
    [BIND]      "Bind",
    [CHDIR]     "Chdir",
    [CLOSE]     "Close",
    [DUP]       "Dup",
    [ALARM]     "Alarm",
    [EXEC]      "Exec",
    [EXITS]     "Exits",
    [FAUTH]     "Fauth",
    [SEGBRK]    "Segbrk",
    [OPEN]      "Open",
    [SLEEP]     "Sleep",
    [RFORK]     "Rfork",
    [PIPE]      "Pipe",
    [CREATE]    "Create",
    [FD2PATH]   "Fd2path",
    [BRK]      "Brk",
    [REMOVE]    "Remove",
```

```
    [NOTIFY]    "Notify",
    [NOTED]     "Noted",
    [SEGATTACH] "Segattach",
    [SEGDETACH] "Segdetach",
    [SEGFREE]   "Segfree",
    [SEGFLUSH]  "Segflush",
    [RENDEZVOUS]    "Rendez",
    [UNMOUNT]   "Unmount",
    [SEMACQUIRE]    "Semacquire",
    [SEMRELEASE]    "Semrelease",
    [SEEK]      "Seek",
    [FVERSION]  "Fversion",
    [ERRSTR]    "Errstr",
    [STAT]      "Stat",
    [FSTAT]     "Fstat",
    [WSTAT]     "Wstat",
    [FWSTAT]    "Fwstat",
    [MOUNT]     "Mount",
    [AWAIT]     "Await",
    [PREAD]     "Pread",
    [PWRITE]    "Pwrite",
    [TSEMACQUIRE]   "Tsemacquire",
};
```

⟨syscalls/systab.c 886a⟩≡
```
⟨kernel basic includes 49a⟩

#include "../port/systab.h"

⟨systab.c forward decl 884c⟩

//coupling: debuggers/acid/conf/syscall
//coupling: debuggers/acid/conf/truss
⟨global systab 57b⟩
⟨global nsyscall 57c⟩

⟨global sysstab 885⟩
```
Uses nelem 853i, nsyscall 886a, and systab 57b.

E.21 time/

time/mul64fract.c

⟨time/mul64fract.c 886b⟩≡
```
#include <u.h>

// default implementation for mul64fract

/* arch_mul64fract(uvlong*r, uvlong a, uvlong b)
 *
 * Multiply two 64 numbers and return the middle 64 bits of the 128 bit result.
 *
 * The assumption is that one of the numbers is a
 * fixed point number with the integer portion in the
 * high word and the fraction in the low word.
 *
 * There should be an assembler version of this routine
 * for each architecture.  This one is intended to
```

```
 * make ports easier.
 *
 *  ignored     r0 = lo(a0*b0)
 *  lsw of result   r1 = hi(a0*b0) +lo(a0*b1) +lo(a1*b0)
 *  msw of result   r2 =        hi(a0*b1) +hi(a1*b0) +lo(a1*b1)
 *  ignored     r3 = hi(a1*b1)
 */
void
arch_mul64fract(uvlong *r, uvlong a, uvlong b)
{
    uvlong bh, bl;
    uvlong ah, al;
    uvlong res;

    bl = b & 0xffffffffULL;
    bh = b >> 32;
    al = a & 0xffffffffULL;
    ah = a >> 32;

    res = (al*bl)>>32;
    res += (al*bh);
    res += (ah*bl);
    res += (ah*bh)<<32;

    *r = res;
}
```

Appendix F

Extra ARM Code

F.1 ../include/arch/arm/

include/arch/arm/ureg.h

⟨arm/ureg.h 888a⟩≡
```
typedef struct Ureg Ureg;
```
⟨*struct* Ureg*(arm)* 37c⟩

F.2 arch/arm/

arch/arm/dat_arch.h

⟨arch/arm/dat_arch.h 888b⟩≡

⟨*struct* Soc*(arm)* 72a⟩

```
extern Soc soc;
```

arch/arm/arm.h

⟨arch/arm/arm.h 888c⟩≡
```
/*
 * ARM-specific definitions for armv6 (arm11), armv7 (cortex-a7 and -a8)
 * these are used in C and assembler
 */

/*
 * Program Status Registers
 */
```
⟨*type* PsrMode*(arm)* 38⟩

⟨*type* PsrDisable*(arm)* 39a⟩

```
/*
 * Coprocessors
 */
```
⟨*type* Coprocessors*(arm)* 39c⟩

```
/*
 * Primary (CRn) CpSC registers.
 */
```

⟨*type CpSC primary registers(arm)* 39d⟩

```
// CpSC secondary registers
```

⟨*type CpID secondary registers(arm)* 535b⟩
⟨*type CpID opcode2(arm)* 535c⟩
⟨*type CpID CpIDid op1(arm)* 548c⟩

⟨*type CpCONTROL opcode2(arm)* 40a⟩
⟨*type CpCONTROL CpMainctl(arm)* 40b⟩
⟨*type CpCONTROL CpAuxctl(arm)* 539e⟩

⟨*type* `CpTTB`*(arm)* 536a⟩

⟨*type* `CpFSR`*(arm)* 171e⟩

⟨*type CpCACHE secondary registers(arm)* 530e⟩
⟨*type CpCACHE opcode2(arm)* 530f⟩

⟨*type CpTLB secondary registers(arm)* 180h⟩
⟨*type CpTLB opcode2(arm)* 180i⟩

⟨*type CpCLD secondary registers(arm)* 537d⟩
⟨*type CpCLD opcode2(arm)* 537e⟩

⟨*type* `CpTIMER`*(arm)* 537b⟩

⟨*type CpSPM secondary registers(arm)* 508d⟩
⟨*type CpSPM opcode2(arm)* 508e⟩

```
// For MCRR
```

⟨*type CpCACHERANGE opcode2(arm)* 529f⟩

```
// Misc
```

⟨*type CpTTB cache control bits(arm)* 536b⟩

```
/*
 * MMU page table entries.
 */
```
⟨*constant* `Mbz`*(arm)* 178e⟩

⟨*constant* `Fault`*(arm)* 179c⟩

⟨*type* `PageDirGranularity`*(arm)* 178d⟩
⟨*type* `PageTableGranularity`*(arm)* 178f⟩

⟨*type* `PageTableEntryAttribute1`*(arm)* 529h⟩

⟨*constant* `Dom0`*(arm)* 179d⟩

⟨*type* `PageTableEntryAttribute2`*(arm)* 535e⟩

⟨*type* `PageTableEntryAttribute3`*(arm)* 179b⟩

⟨*type* `DAC`*(arm)* 179a⟩

⟨*macro* `F`*(arm)* 179g⟩

⟨*macro* AP*(arm)* 179f⟩
⟨*macro* L1AP*(arm)* 179e⟩
```
/* L2AP differs between armv6 and armv7 -- see l2ap in arch*.c */
```
⟨*macro* DAC*(arm)* 179h⟩

⟨*constant* HVECTORS*(arm)* 39b⟩

arch/arm/arminstr.ha

⟨*arch/arm/arminstr.ha* 890a⟩≡
```
/*
 * armv6/v7 machine assist, definitions
 *
 * loader uses R11 as scratch.
 */
```

⟨*macro PADDR bis(arm)* 56c⟩
⟨*macro L1X bis(arm)* 178c⟩

```
/*
 * new instructions
 */
```
⟨*instruction ISB(arm)* 530c⟩
⟨*instruction DSB(arm)* 530d⟩
⟨*instruction BARRIERS(arm)* 534h⟩

⟨*instruction MCRR(arm)* 529g⟩
⟨*instruction MRRC(arm)* 1012b⟩

⟨*instruction LDREX(arm)* 546b⟩
⟨*instruction STREX(arm)* 546c⟩
⟨*instruction CLREX(arm)* 546d⟩

⟨*instruction CPSID(arm)* 548b⟩

⟨*instruction CPUID(arm)* 535d⟩

arch/arm/io.h

⟨arch/arm/io.h 890b⟩≡
```
// could be in dat_interrupts.h?
```
⟨*enum* IRQ*(arm)* 174b⟩
⟨*macro* IRQDMA*(arm)* 955a⟩

⟨*enum* Dma*(arm)* 955b⟩

⟨*enum* Power*(arm)* 562c⟩

```
// could be in dat_time.h?
```
⟨*enum* Clock*(arm)* 896a⟩

arch/arm/coproc.c

⟨*enum* _anon_*(arm)* 890c⟩≡ (892c)
```
enum {
    /* alternates:
```

```
         *  0xe12fff1e  BX (R14); last e is R14
         *  0xe28ef000  B 0(R14); second e is R14 (ken)
         */
        RETinst = 0xe1a0f00e,       /* MOV R14, R15 */

        Opmask  = MASK(3),
        Regmask = MASK(4),
    };
```

Uses MASK 47c.

⟨*function* setupcpop*(arm)* 891a⟩≡ (892c)

```
    // setup coprocessor operation
    static void
    setupcpop(ulong instr[2], ulong opcode, int cp, int op1, int crn, int crm,
        int op2)
    {
        ulong instrsz[2];

        op1 &= Opmask;
        op2 &= Opmask;
        crn &= Regmask;
        crm &= Regmask;
        cp  &= Regmask;
        instr[0] = opcode | op1 << 21 | crn << 16 | cp << 8 | op2 << 5 | crm;
        instr[1] = RETinst;

        cachedwbse(instr, sizeof instrsz);
        cacheiinv();
    }
```

Uses Opmask-2384 890c, RETinst-2383 890c, and Regmask-2385 890c.

⟨*function* cprd*(arm)* 891b⟩≡ (892c)

```
    ulong
    cprd(int cp, int op1, int crn, int crm, int op2)
    {
        int s, r;
        volatile ulong instr[2];
        Pufv fp;

        s = arch_splhi();
        /*
         * MRC.  return value will be in R0, which is convenient.
         * Rt will be R0.
         */
        // MRC 0x e(Mxx) e(Always) 0 (MRC) .... 10 (part of Mxxx opcode)
        setupcpop(instr, 0xee100010, cp, op1, crn, crm, op2);
        fp = (Pufv)instr;
        r = fp();
        arch_splx(s);
        return r;
    }
```

Uses setupcpop() 891a.

⟨*function* cpwr*(arm)* 891c⟩≡ (892c)

```
    void
    cpwr(int cp, int op1, int crn, int crm, int op2, ulong val)
    {
        int s;
        volatile ulong instr[2];
```

```
    Pvfu fp;

    s = arch_splhi();
    // MCR 0x e(Mxx) e(Always) 0 (MCR) .... 10 (part of Mxxx opcode)
    setupcpop(instr, 0xee000010, cp, op1, crn, crm, op2); /* MCR, Rt is R0 */
    fp = (Pvfu)instr;
    fp(val);
    arch_coherence();
    arch_splx(s);
}
```

Uses arch_coherence and setupcpop() 891a.

⟨*function* cprdsc*(arm)* 892a⟩≡ (892c)

```
ulong
cprdsc(int op1, int crn, int crm, int op2)
{
    return cprd(CpSC, op1, crn, crm, op2);
}
```

Uses CpSC 39c and cprd() 891b.

⟨*function* cpwrsc*(arm)* 892b⟩≡ (892c)

```
void
cpwrsc(int op1, int crn, int crm, int op2, ulong val)
{
    cpwr(CpSC, op1, crn, crm, op2, val);
}
```

Uses CpSC 39c and cpwr() 891c.

⟨arch/arm/coproc.c 892c⟩≡

```
/*
 * ARM co-processors
 * mainly to cope with ARM hard-wiring register numbers into instructions.
 *
 * CP15 (system control) is the one that gets used the most in practice.
 * these routines must be callable from KZERO space or the 0 segment.
 */
```

⟨*kernel basic includes* 49a⟩

```
#include "io.h"
#include "arm.h"
```

⟨*enum* _anon_*(arm)* 890c⟩

```
// pointer to ulong function void
typedef ulong (*Pufv)(void);
// pointer to void function ulong
typedef void  (*Pvfu)(ulong);
```

⟨*function* setupcpop*(arm)* 891a⟩

⟨*function* cprd*(arm)* 891b⟩

⟨*function* cpwr*(arm)* 891c⟩

⟨*function* cprdsc*(arm)* 892a⟩

⟨*function* cpwrsc*(arm)* 892b⟩

```
/* floating point */
```

⟨*function* `setupfpctlop`*(arm)* 519a⟩

⟨*function* `fprd`*(arm)* 519b⟩

⟨*function* `fpwr`*(arm)* 519c⟩

⟨*function* `setupfpop`*(arm)* 520a⟩

⟨*function* `fpsavereg`*(arm)* 520c⟩

⟨*function* `fprestreg`*(arm)* 520b⟩

arch/arm/vcore.c

⟨*enum* `_anon_(`(arch/arm/vcore.c)(arm)`)` 893a⟩≡ (897b)

```
enum {
    Read        = 0x00>>2,
    Write       = 0x00>>2,
    Peek        = 0x10>>2,
    Sender      = 0x14>>2,
    Status      = 0x18>>2,
        Full        = 1<<31,
        Empty       = 1<<30,
    Config      = 0x1C>>2,

    NRegs       = 0x20>>2,
};
```

⟨*enum* `_anon_(`(arch/arm/vcore.c)2(arm)`)` 893b⟩≡ (897b)

```
enum {
    ChanFb      = 1,
    ChanProps   = 8,
    ChanMask    = 0xF,
};
```

⟨*enum* `_anon_(`(arch/arm/vcore.c)3(arm)`)` 893c⟩≡ (897b)

```
enum {
    Req         = 0x0,
    RspOk       = 0x80000000,
    TagResp     = 1<<31,
};
```

⟨*enum* `_anon_(`(arch/arm/vcore.c)4(arm)`)` 893d⟩≡ (897b)

```
enum {
    TagGetfwrev = 0x00000001,
    TagGetrev   = 0x00010002,

    TagGetmac   = 0x00010003,
    TagGetram   = 0x00010005,

    TagGetpower = 0x00020001,
    TagSetpower = 0x00028001,
        Powerwait   = 1<<1,

    TagGetclkspd= 0x00030002,
    TagGetclkmax= 0x00030004,
    TagSetclkspd= 0x00038002,
```

```
    TagGettemp  = 0x00030006,

    TagFballoc  = 0x00040001,
    TagFbfree   = 0x00048001,
    TagFbblank  = 0x00040002,

    TagGetres   = 0x00040003,
    TagSetres   = 0x00048003,
    TagGetvres  = 0x00040004,
    TagSetvres  = 0x00048004,
    TagGetdepth = 0x00040005,
    TagSetdepth = 0x00048005,
    TagGetrgb   = 0x00044006,
    TagSetrgb   = 0x00048006,
};
```

⟨*struct* Prophdr*(arm)* 894a⟩≡ (897b)

```
struct Prophdr {
    u32int  len;
    u32int  req;
    u32int  tag;
    u32int  tagbuflen;
    u32int  taglen;
    u32int  data[1];
};
```

⟨*function* vcwrite*(arm)* 894b⟩≡ (897b)

```
static void
vcwrite(uint chan, int val)
{
    u32int *r;

    r = (u32int*)MAILBOX + NRegs;
    val &= ~ChanMask;
    while(r[Status] & Full)
        ;
    arch_coherence();
    r[Write] = val | chan;
}
```

Uses ChanMask-2357 893b, Full-2351 893a, MAILBOX-2345 43h, NRegs-2354 893a, Status-2350 893a, Write-2347 893a, and arch_coherence.

⟨*function* vcread*(arm)* 894c⟩≡ (897b)

```
static int
vcread(uint chan)
{
    u32int *r;
    int x;

    r = (u32int*)MAILBOX;
    do{
        while(r[Status]&Empty)
            ;
        arch_coherence();
        x = r[Read];
    }while((x&ChanMask) != chan);
    return x & ~ChanMask;
}
```

Uses ChanMask-2357 893b, Empty-2352 893a, MAILBOX-2345 43h, Read-2346 893a, Status-2350 893a, and arch_coherence.

⟨*function* vcreq*(arm)* 895a⟩≡ (897b)

```
/*
 * Property interface
 */

static int
vcreq(int tag, void *buf, int vallen, int rsplen)
{
    uintptr r;
    int n;
    Prophdr *prop = (Prophdr*)(VCBUFFER);
    uintptr aprop;
    static bool busaddr = true;

    if(rsplen < vallen)
        rsplen = vallen;
    rsplen = (rsplen+3) & ~3;

    n = sizeof(Prophdr) + rsplen + 8;
    memset(prop, 0, n);
    prop->len = n;
    prop->req = Req;
    prop->tag = tag;
    prop->tagbuflen = rsplen;
    prop->taglen = vallen;
    if(vallen > 0)
        memmove(prop->data, buf, vallen);
    cachedwbinvse(prop, prop->len);
    for(;;){
        aprop = busaddr? dmaaddr(prop) : PTR2UINT(prop);
        vcwrite(ChanProps, aprop);
        r = vcread(ChanProps);
        if(r == aprop)
            break;
        if(!busaddr)
            return -1;
        busaddr = false;
    }
    if(prop->req == RspOk &&
       prop->tag == tag &&
       (prop->taglen&TagResp)) {
        if((n = prop->taglen & ~TagResp) < rsplen)
            rsplen = n;
        memmove(buf, prop->data, rsplen);
    }else
        rsplen = -1;

    return rsplen;
}
```

Uses ChanProps-2356 893b, PTR2UINT 56g, Req-2358 893c, RspOk-2359 893c, TagResp-2360 893c, VCBUFFER 386a, dmaaddr() 957h, false, true, vcread() 894c, and vcwrite() 894b.

⟨*function* getethermac*(arm)* 895b⟩≡ (897b)

```
/*
 * Get ethernet address (as hex string)
 *   [not reentrant]
 */
char *
getethermac(void)
{
```

```
    uchar ea[8];
    char *p;
    int i;
    static char buf[16];

    memset(ea, 0, sizeof ea);
    vcreq(TagGetmac, ea, 0, sizeof ea);
    p = buf;
    for(i = 0; i < 6; i++)
        p += sprint(p, "%.2x", ea[i]);
    return buf;
}
```

Uses TagGetmac-2363 893d and vcreq() 895a.

⟨*enum* Clock*(arm)* 896a⟩≡ (890b)

```
enum {
    ClkEmmc     = 1,
    ClkUart,
    ClkArm,
    ClkCore,
    ClkV3d,
    ClkH264,
    ClkIsp,
    ClkSdram,
    ClkPixel,
    ClkPwm,
};
```

⟨*function* getclkrate*(arm)* 896b⟩≡ (897b)

```
/*
 * Get clock rate
 */
ulong
getclkrate(int clkid)
{
    u32int buf[2];

    buf[0] = clkid;
    if(vcreq(TagGetclkspd, buf, sizeof(buf[0]), sizeof(buf)) != sizeof buf)
        return 0;
    return buf[1];
}
```

Uses TagGetclkspd-2368 893d and vcreq() 895a.

⟨*function* setclkrate*(arm)* 896c⟩≡ (897b)

```
/*
 * Set clock rate to hz (or max speed if hz == 0)
 */
void
setclkrate(int clkid, ulong hz)
{
    u32int buf[2];

    buf[0] = clkid;
    if(hz != 0)
        buf[1] = hz;
    else if(vcreq(TagGetclkmax, buf, sizeof(buf[0]), sizeof(buf)) != sizeof buf)
        return;
    vcreq(TagSetclkspd, buf, sizeof(buf), sizeof(buf));
}
```

Uses TagSetclkspd-2370 893d and vcreq() 895a.

⟨*function* `getcputemp`*(arm)* 897a⟩≡ (897b)

```
/*
 * Get cpu temperature
 */
uint
getcputemp(void)
{
    u32int buf[2];

    buf[0] = 0;
    if(vcreq(TagGettemp, buf, sizeof(buf[0]), sizeof buf) != sizeof buf)
        return 0;
    return buf[1];
}
```

Uses `TagGettemp-2371` 893d and `vcreq()` 895a.

⟨`arch/arm/vcore.c` 897b⟩≡

⟨*kernel basic includes* 49a⟩

```
/*
 * Mailbox interface with videocore gpu
 */
```

⟨*constant* `MAILBOX`*(arm)* 43h⟩

```
typedef struct Fbinfo Fbinfo;
typedef struct Prophdr Prophdr;
```

⟨*enum* `_anon_(`(arch/arm/vcore.c)(arm)`)` 893a⟩
⟨*enum* `_anon_(`(arch/arm/vcore.c)2(arm)`)` 893b⟩

⟨*enum* `_anon_(`(arch/arm/vcore.c)3(arm)`)` 893c⟩

⟨*enum* `_anon_(`(arch/arm/vcore.c)4(arm)`)` 893d⟩

⟨*struct* `Fbinfo`*(arm)* 43i⟩

⟨*struct* `Prophdr`*(arm)* 894a⟩

⟨*function* `vcwrite`*(arm)* 894b⟩

⟨*function* `vcread`*(arm)* 894c⟩

⟨*function* `vcreq`*(arm)* 895a⟩

⟨*function* `fbdefault`*(arm)* 387b⟩

⟨*function* `fbinit`*(arm)* 386b⟩

⟨*function* `fbblank`*(arm)* 387c⟩

⟨*function* `setpower`*(arm)* 563a⟩

⟨*function* `getpower`*(arm)* 563b⟩

⟨*function* `getethermac`*(arm)* 895b⟩

⟨*function* `getboardrev`*(arm)* 286g⟩

⟨*function* `getfirmware`*(arm)* 287a⟩

⟨*function* `getramsize`*(arm)* 275d⟩

⟨*function* `getclkrate`*(arm)* 896b⟩

⟨*function* `setclkrate`*(arm)* 896c⟩

⟨*function* `getcputemp`*(arm)* 897a⟩

Uses `Fbinfo` 43i and `Prophdr` 894a.

F.3 buses/arm/

buses/arm/dwcotg.h

⟨*enum* `_anon_`*(*`(buses/arm/dwcotg.h)(arm)`*)* 898a⟩≡ (899)

```
enum {
    Maxchans    = 16,   /* actual number of channels in ghwcfg2 */
};
```

⟨*struct* `Dwcregs`*(arm)* 898b⟩≡ (899)

```
struct Dwcregs {
    /* Core global registers 0x000-0x140 */
    Reg gotgctl;    /* OTG Control and Status */
    Reg gotgint;    /* OTG Interrupt */
    Reg gahbcfg;    /* Core AHB Configuration */
    Reg gusbcfg;    /* Core USB Configuration */
    Reg grstctl;    /* Core Reset */
    Reg gintsts;    /* Core Interrupt */
    Reg gintmsk;    /* Core Interrupt Mask */
    Reg grxstsr;    /* Receive Status Queue Read (RO) */
    Reg grxstsp;    /* Receive Status Queue Read & POP (RO) */
    Reg grxfsiz;    /* Receive FIFO Size */
    Reg gnptxfsiz;  /* Non Periodic Transmit FIFO Size */
    Reg gnptxsts;   /* Non Periodic Transmit FIFO/Queue Status (RO) */
    Reg gi2cctl;    /* I2C Access */
    Reg gpvndctl;   /* PHY Vendor Control */
    Reg ggpio;      /* General Purpose Input/Output */
    Reg guid;       /* User ID */
    Reg gsnpsid;    /* Synopsys ID (RO) */
    Reg ghwcfg1;    /* User HW Config1 (RO) (DEVICE) */
    Reg ghwcfg2;    /* User HW Config2 (RO) */
    Reg ghwcfg3;    /* User HW Config3 (RO) */
    Reg ghwcfg4;    /* User HW Config4 (RO)*/
    Reg glpmcfg;    /* Core LPM Configuration */
    Reg gpwrdn;     /* Global PowerDn */
    Reg gdfifocfg;  /* Global DFIFO SW Config (DEVICE?) */
    Reg adpctl;     /* ADP Control */
    Reg reserved0[39];
    Reg hptxfsiz;   /* Host Periodic Transmit FIFO Size */
    Reg dtxfsiz[15];    /* Device Periodic Transmit FIFOs (DEVICE) */
    char    pad0[0x400-0x140];

    /* Host global registers 0x400-0x420 */
    Reg hcfg;       /* Configuration */
    Reg hfir;       /* Frame Interval */
    Reg hfnum;      /* Frame Number / Frame Remaining (RO) */
    Reg reserved1;
    Reg hptxsts;    /* Periodic Transmit FIFO / Queue Status */
```

```
    Reg haint;      /* All Channels Interrupt */
    Reg haintmsk;   /* All Channels Interrupt Mask */
    Reg hflbaddr;   /* Frame List Base Address */
    char    pad1[0x440-0x420];

    /* Host port register 0x440 */
    Reg hport0;     /* Host Port 0 Control and Status */
    char    pad2[0x500-0x444];

    /* Host channel specific registers 0x500-0x700 */
    struct  Hostchan {
        Reg hcchar; /* Characteristic */
        Reg hcsplt; /* Split Control */
        Reg hcint;  /* Interrupt */
        Reg hcintmsk; /* Interrupt Mask */
        Reg hctsiz; /* Transfer Size */
        Reg hcdma;  /* DMA Address */
        Reg reserved;
        Reg hcdmab; /* DMA Buffer Address */
    } hchan[Maxchans];
    char    pad3[0xE00-0x700];

    /* Power & clock gating control register 0xE00 */
    Reg pcgcctl;
};
```

Uses Hostchan 898b and Maxchans 898a.

⟨buses/arm/dwcotg.h 899⟩≡

```
/*
 * USB host driver for BCM2835
 *  Synopsis DesignWare Core USB 2.0 OTG controller
 *
 * Device register definitions
 */

typedef unsigned int Reg;
typedef struct Dwcregs Dwcregs;
typedef struct Hostchan Hostchan;
```

⟨*enum* _anon_*(*(buses/arm/dwcotg.h)(arm)*)* 898a⟩

⟨*struct* Dwcregs*(arm)* 898b⟩

```
enum {
    /* gotgctl */
    Sesreqscs   = 1<<0,
    Sesreq      = 1<<1,
    Vbvalidoven = 1<<2,
    Vbvalidovval    = 1<<3,
    Avalidoven  = 1<<4,
    Avalidovval = 1<<5,
    Bvalidoven  = 1<<6,
    Bvalidovval = 1<<7,
    Hstnegscs   = 1<<8,
    Hnpreq      = 1<<9,
    Hstsethnpen = 1<<10,
    Devhnpen    = 1<<11,
    Conidsts    = 1<<16,
    Dbnctime    = 1<<17,
    Asesvld     = 1<<18,
```

```
Bsesvld     = 1<<19,
Otgver      = 1<<20,
Multvalidbc = 0x1F<<22,
Chirpen     = 1<<27,

/* gotgint */
Sesenddet   = 1<<2,
Sesreqsucstschng= 1<<8,
Hstnegsucstschng= 1<<9,
Hstnegdet   = 1<<17,
Adevtoutchng    = 1<<18,
Debdone     = 1<<19,
Mvic        = 1<<20,

/* gahbcfg */
Glblintrmsk = 1<<0,
/* bits 1:4 redefined for BCM2835 */
Axiburstlen = 0x3<<1,
    BURST1      = 3<<1,
    BURST2      = 2<<1,
    BURST3      = 1<<1,
    BURST4      = 0<<1,
Axiwaitwrites   = 1<<4,
Dmaenable   = 1<<5,
Nptxfemplvl = 1<<7,
    NPTX_HALFEMPTY  = 0<<7,
    NPTX_EMPTY  = 1<<7,
Ptxfemplvl  = 1<<8,
    PTX_HALFEMPTY   = 0<<8,
    PTX_EMPTY   = 1<<8,
Remmemsupp  = 1<<21,
Notialldmawrit  = 1<<22,
Ahbsingle   = 1<<23,

/* gusbcfg */
Toutcal     = 0x7<<0,
Phyif       = 1<<3,
Ulpi_utmi_sel   = 1<<4,
Fsintf      = 1<<5,
    FsUnidir    = 0<<5,
    FsBidir     = 1<<5,
Physel      = 1<<6,
    PhyHighspeed    = 0<<6,
    PhyFullspeed    = 1<<6,
Ddrsel      = 1<<7,
Srpcap      = 1<<8,
Hnpcap      = 1<<9,
Usbtrdtim   = 0xf<<10,
    OUsbtrdtim      = 10,
Phylpwrclksel   = 1<<15,
Otgutmifssel    = 1<<16,
Ulpi_fsls   = 1<<17,
Ulpi_auto_res   = 1<<18,
Ulpi_clk_sus_m  = 1<<19,
Ulpi_ext_vbus_drv= 1<<20,
Ulpi_int_vbus_indicator= 1<<21,
Term_sel_dl_pulse= 1<<22,
Indicator_complement= 1<<23,
Indicator_pass_through= 1<<24,
Ulpi_int_prot_dis= 1<<25,
```

```
Ic_usb_cap  = 1<<26,
Ic_traffic_pull_remove= 1<<27,
Tx_end_delay    = 1<<28,
Force_host_mode = 1<<29,
Force_dev_mode  = 1<<30,

/* grstctl */
Csftrst     = 1<<0,
Hsftrst     = 1<<1,
Hstfrm      = 1<<2,
Intknqflsh  = 1<<3,
Rxfflsh     = 1<<4,
Txfflsh     = 1<<5,
Txfnum      = 0x1f<<6,
    TXF_ALL     = 0x10<<6,
Dmareq      = 1<<30,
Ahbidle     = 1<<31,

/* gintsts, gintmsk */
Curmode     = 1<<0,
    HOSTMODE    = 1<<0,
    DEVMODE     = 0<<0,
Modemismatch    = 1<<1,
Otgintr     = 1<<2,
Sofintr     = 1<<3,
Rxstsqlvl   = 1<<4,
Nptxfempty  = 1<<5,
Ginnakeff   = 1<<6,
Goutnakeff  = 1<<7,
Ulpickint   = 1<<8,
I2cintr     = 1<<9,
Erlysuspend = 1<<10,
Usbsuspend  = 1<<11,
Usbreset    = 1<<12,
Enumdone    = 1<<13,
Isooutdrop  = 1<<14,
Eopframe    = 1<<15,
Restoredone = 1<<16,
Epmismatch  = 1<<17,
Inepintr    = 1<<18,
Outepintr   = 1<<19,
Incomplisoin    = 1<<20,
Incomplisoout   = 1<<21,
Fetsusp     = 1<<22,
Resetdet    = 1<<23,
Portintr    = 1<<24,
Hcintr      = 1<<25,
Ptxfempty   = 1<<26,
Lpmtranrcvd = 1<<27,
Conidstschng    = 1<<28,
Disconnect  = 1<<29,
Sessreqintr = 1<<30,
Wkupintr    = 1<<31,

/* grxsts[rp] */
Chnum       = 0xf<<0,
Bcnt        = 0x7ff<<4,
Dpid        = 0x3<<15,
Pktsts      = 0xf<<17,
    PKTSTS_IN       = 2<<17,
```

```
    PKTSTS_IN_XFER_COMP = 3<<17,
    PKTSTS_DATA_TOGGLE_ERR  = 5<<17,
    PKTSTS_CH_HALTED    = 7<<17,

/* hptxfsiz, gnptxfsiz */
Startaddr   = 0xffff<<0,
Depth       = 0xffff<<16,
    ODepth      = 16,

/* gnptxsts */
Nptxfspcavail   = 0xffff<<0,
Nptxqspcavail   = 0xff<<16,
Nptxqtop_terminate= 1<<24,
Nptxqtop_token  = 0x3<<25,
Nptxqtop_chnep  = 0xf<<27,

/* gpvndctl */
Regdata     = 0xff<<0,
Vctrl       = 0xff<<8,
Regaddr16_21    = 0x3f<<16,
Regwr       = 1<<22,
Newregreq   = 1<<25,
Vstsbsy     = 1<<26,
Vstsdone    = 1<<27,
Disulpidrvr = 1<<31,

/* ggpio */
Gpi     = 0xffff<<0,
Gpo     = 0xffff<<16,

/* ghwcfg2 */
Op_mode     = 0x7<<0,
    HNP_SRP_CAPABLE_OTG = 0<<0,
    SRP_ONLY_CAPABLE_OTG    = 1<<0,
    NO_HNP_SRP_CAPABLE  = 2<<0,
    SRP_CAPABLE_DEVICE  = 3<<0,
    NO_SRP_CAPABLE_DEVICE   = 4<<0,
    SRP_CAPABLE_HOST    = 5<<0,
    NO_SRP_CAPABLE_HOST = 6<<0,
Architecture    = 0x3<<3,
    SLAVE_ONLY      = 0<<3,
    EXT_DMA         = 1<<3,
    INT_DMA         = 2<<3,
Point2point = 1<<5,
Hs_phy_type = 0x3<<6,
    PHY_NOT_SUPPORTED   = 0<<6,
    PHY_UTMI        = 1<<6,
    PHY_ULPI        = 2<<6,
    PHY_UTMI_ULPI       = 3<<6,
Fs_phy_type = 0x3<<8,
Num_dev_ep  = 0xf<<10,
Num_host_chan   = 0xf<<14,
    ONum_host_chan      = 14,
Perio_ep_supported= 1<<18,
Dynamic_fifo    = 1<<19,
Nonperio_tx_q_depth= 0x3<<22,
Host_perio_tx_q_depth= 0x3<<24,
Dev_token_q_depth= 0x1f<<26,
Otg_enable_ic_usb= 1<<31,
```

```
/* ghwcfg3 */
Xfer_size_cntr_width    = 0xf<<0,
Packet_size_cntr_width  = 0x7<<4,
Otg_func        = 1<<7,
I2c         = 1<<8,
Vendor_ctrl_if      = 1<<9,
Optional_features   = 1<<10,
Synch_reset_type    = 1<<11,
Adp_supp        = 1<<12,
Otg_enable_hsic     = 1<<13,
Bc_support      = 1<<14,
Otg_lpm_en      = 1<<15,
Dfifo_depth     = 0xffff<<16,
    ODfifo_depth        = 16,

/* ghwcfg4 */
Num_dev_perio_in_ep = 0xf<<0,
Power_optimiz       = 1<<4,
Min_ahb_freq        = 1<<5,
Hiber           = 1<<6,
Xhiber          = 1<<7,
Utmi_phy_data_width = 0x3<<14,
Num_dev_mode_ctrl_ep    = 0xf<<16,
Iddig_filt_en       = 1<<20,
Vbus_valid_filt_en  = 1<<21,
A_valid_filt_en     = 1<<22,
B_valid_filt_en     = 1<<23,
Session_end_filt_en = 1<<24,
Ded_fifo_en     = 1<<25,
Num_in_eps      = 0xf<<26,
Desc_dma        = 1<<30,
Desc_dma_dyn        = 1<<31,

/* glpmcfg */
Lpm_cap_en  = 1<<0,
Appl_resp   = 1<<1,
Hird        = 0xf<<2,
Rem_wkup_en = 1<<6,
En_utmi_sleep   = 1<<7,
Hird_thres  = 0x1f<<8,
Lpm_resp    = 0x3<<13,
Prt_sleep_sts   = 1<<15,
Sleep_state_resumeok= 1<<16,
Lpm_chan_index  = 0xf<<17,
Retry_count = 0x7<<21,
Send_lpm    = 1<<24,
Retry_count_sts = 0x7<<25,
Hsic_connect    = 1<<30,
Inv_sel_hsic    = 1<<31,

/* gpwrdn */
Pmuintsel   = 1<<0,
Pmuactv     = 1<<1,
Restore     = 1<<2,
Pwrdnclmp   = 1<<3,
Pwrdnrstn   = 1<<4,
Pwrdnswtch  = 1<<5,
Dis_vbus    = 1<<6,
Lnstschng   = 1<<7,
Lnstchng_msk    = 1<<8,
```

```
Rst_det     = 1<<9,
Rst_det_msk = 1<<10,
Disconn_det = 1<<11,
Disconn_det_msk = 1<<12,
Connect_det = 1<<13,
Connect_det_msk = 1<<14,
Srp_det     = 1<<15,
Srp_det_msk = 1<<16,
Sts_chngint = 1<<17,
Sts_chngint_msk = 1<<18,
Linestate   = 0x3<<19,
Idsts       = 1<<21,
Bsessvld    = 1<<22,
Adp_int     = 1<<23,
Mult_val_id_bc  = 0x1f<<24,

/* gdfifocfg */
Gdfifocfg   = 0xffff<<0,
Epinfobase  = 0xffff<<16,

/* adpctl */
Prb_dschg   = 0x3<<0,
Prb_delta   = 0x3<<2,
Prb_per     = 0x3<<4,
Rtim        = 0x7ff<<6,
Enaprb      = 1<<17,
Enasns      = 1<<18,
Adpres      = 1<<19,
Adpen       = 1<<20,
Adp_prb_int = 1<<21,
Adp_sns_int = 1<<22,
Adp_tmout_int   = 1<<23,
Adp_prb_int_msk = 1<<24,
Adp_sns_int_msk = 1<<25,
Adp_tmout_int_msk= 1<<26,
Ar      = 0x3<<27,

/* hcfg */
Fslspclksel = 0x3<<0,
    HCFG_30_60_MHZ  = 0<<0,
    HCFG_48_MHZ = 1<<0,
    HCFG_6_MHZ  = 2<<0,
Fslssupp    = 1<<2,
Ena32khzs   = 1<<7,
Resvalid    = 0xff<<8,
Descdma     = 1<<23,
Frlisten    = 0x3<<24,
Modechtimen = 1<<31,

/* hfir */
Frint       = 0xffff<<0,
Hfirrldctrl = 1<<16,

/* hfnum */
Frnum       = 0xffff<<0,
    MAX_FRNUM   = 0x3FFF<<0,
Frrem       = 0xffff<<16,

/* hptxsts */
Ptxfspcavail    = 0xffff<<0,
```

```
    Ptxqspcavail    = 0xff<<16,
    Ptxqtop_terminate= 1<<24,
    Ptxqtop_token   = 0x3<<25,
    Ptxqtop_chnum   = 0xf<<27,
    Ptxqtop_odd = 1<<31,

    /* haint, haintmsk */
#define CHANINT(n)  (1<<(n))

    /* hport0 */
    Prtconnsts  = 1<<0,     /* connect status (RO) */
    Prtconndet  = 1<<1,     /* connect detected R/W1C) */
    Prtena      = 1<<2,     /* enable (R/W1C) */
    Prtenchng   = 1<<3,     /* enable/disable change (R/W1C) */
    Prtovrcurract   = 1<<4,     /* overcurrent active (RO) */
    Prtovrcurrchng  = 1<<5,     /* overcurrent change (R/W1C) */
    Prtres      = 1<<6,     /* resume */
    Prtsusp     = 1<<7,     /* suspend */
    Prtrst      = 1<<8,     /* reset */
    Prtlnsts    = 0x3<<10,  /* line state {D+,D-} (RO) */
    Prtpwr      = 1<<12,    /* power on */
    Prttstctl   = 0xf<<13,  /* test */
    Prtspd      = 0x3<<17,  /* speed (RO) */
        HIGHSPEED   = 0<<17,
        FULLSPEED   = 1<<17,
        LOWSPEED    = 2<<17,

    /* hcchar */
    Mps     = 0x7ff<<0, /* endpoint maximum packet size */
    Epnum       = 0xf<<11,  /* endpoint number */
        OEpnum      = 11,
    Epdir       = 1<<15,    /* endpoint direction */
        Epout       = 0<<15,
        Epin        = 1<<15,
    Lspddev     = 1<<17,    /* device is lowspeed */
    Eptype      = 0x3<<18,  /* endpoint type */
        Epctl       = 0<<18,
        Episo       = 1<<18,
        Epbulk      = 2<<18,
        Epintr      = 3<<18,
    Multicnt    = 0x3<<20,  /* transactions per μframe */
                    /* or retries per periodic split */
        OMulticnt   = 20,
    Devaddr     = 0x7f<<22, /* device address */
        ODevaddr    = 22,
    Oddfrm      = 1<<29,    /* xfer in odd frame (iso/interrupt) */
    Chdis       = 1<<30,    /* channel disable (write 1 only) */
    Chen        = 1<<31,    /* channel enable (write 1 only) */

    /* hcsplt */
    Prtaddr     = 0x7f<<0,  /* port address of recipient */
                    /* transaction translator */
    Hubaddr     = 0x7f<<7,  /* dev address of transaction */
                    /* translator's hub */
        OHubaddr    = 7,
    Xactpos     = 0x3<<14,  /* payload's position within transaction */
        POS_MID     = 0<<14,
        POS_END     = 1<<14,
        POS_BEGIN   = 2<<14,
        POS_ALL     = 3<<14, /* all of data (<= 188 bytes) */
```

```
    Compsplt    = 1<<16,    /* do complete split */
    Spltena     = 1<<31,    /* channel enabled to do splits */

    /* hcint, hcintmsk */
    Xfercomp    = 1<<0,     /* transfer completed without error */
    Chhltd      = 1<<1,     /* channel halted */
    Ahberr      = 1<<2,     /* AHB dma error */
    Stall       = 1<<3,
    Nak     = 1<<4,
    Ack     = 1<<5,
    Nyet        = 1<<6,
    Xacterr     = 1<<7, /* transaction error (crc, t/o, bit stuff, eop) */
    Bblerr      = 1<<8,
    Frmovrun    = 1<<9,
    Datatglerr  = 1<<10,
    Bna     = 1<<11,
    Xcs_xact    = 1<<12,
    Frm_list_roll   = 1<<13,

    /* hctsiz */
    Xfersize    = 0x7ffff<<0,   /* expected total bytes */
    Pktcnt      = 0x3ff<<19,    /* expected number of packets */
        OPktcnt     = 19,
    Pid     = 0x3<<29,  /* packet id for initial transaction */
        DATA0       = 0<<29,
        DATA1       = 2<<29,    /* sic */
        DATA2       = 1<<29,    /* sic */
        MDATA       = 3<<29,    /* (non-ctl ep) */
        SETUP       = 3<<29,    /* (ctl ep) */
    Dopng       = 1<<31,    /* do PING protocol */

    /* pcgcctl */
    Stoppclk        = 1<<0,
    Gatehclk        = 1<<1,
    Pwrclmp         = 1<<2,
    Rstpdwnmodule       = 1<<3,
    Enbl_sleep_gating   = 1<<5,
    Phy_in_sleep        = 1<<6,
    Deep_sleep      = 1<<7,
    Resetaftsusp        = 1<<8,
    Restoremode     = 1<<9,
    Enbl_extnd_hiber    = 1<<10,
    Extnd_hiber_pwrclmp = 1<<11,
    Extnd_hiber_switch  = 1<<12,
    Ess_reg_restored    = 1<<13,
    Prt_clk_sel     = 0x3<<14,
    Port_power      = 1<<16,
    Max_xcvrselect      = 0x3<<17,
    Max_termsel     = 1<<19,
    Mac_dev_addr        = 0x7f<<20,
    P2hd_dev_enum_spd   = 0x3<<27,
    P2hd_prt_spd        = 0x3<<29,
    If_dev_mode     = 1<<31,
};
```

Uses Dwcregs 898b and Hostchan 898b.

buses/arm/devusb.c

⟨*enum* _anon_*(*(buses/arm/devusb.c)(arm)*)* 906⟩≡ (933)

```
enum
{
    /* Qid numbers */
    Qdir = 0,       /* #u */
    Qusbdir,            /* #u/usb */
    Qctl,           /* #u/usb/ctl - control requests */

    Qep0dir,            /* #u/usb/ep0.0 - endpoint 0 dir */
    Qep0io,         /* #u/usb/ep0.0/data - endpoint 0 I/O */
    Qep0ctl,        /* #u/usb/ep0.0/ctl - endpoint 0 ctl. */
    Qep0dummy,      /* give 4 qids to each endpoint */

    Qepdir = 0,     /* (qid-qep0dir)&3 is one of these */
    Qepio,          /* to identify which file for the endpoint */
    Qepctl,

    /* ... */

    /* Usb ctls. */
    CMdebug = 0,        /* debug on|off */
    CMdump,         /* dump (data structures for debug) */

    /* Ep. ctls */
    CMnew = 0,      /* new nb ctl|bulk|intr|iso r|w|rw (endpoint) */
    CMnewdev,       /* newdev full|low|high portnb (allocate new devices) */
    CMhub,          /* hub (set the device as a hub) */
    CMspeed,        /* speed full|low|high|no */
    CMmaxpkt,       /* maxpkt size */
    CMntds,         /* ntds nb (max nb. of tds per µframe) */
    CMclrhalt,      /* clrhalt (halt was cleared on endpoint) */
    CMpollival,     /* pollival interval (interrupt/iso) */
    CMhz,           /* hz n (samples/sec; iso) */
    CMsamplesz,     /* samplesz n (sample size; iso) */
    CMinfo,         /* info infostr (ke.ep info for humans) */
    CMdetach,       /* detach (abort I/O forever on this ep). */
    CMaddress,      /* address (address is assigned) */
    CMdebugep,      /* debug n (set/clear debug for this ep) */
    CMname,         /* name str (show up as #u/name as well) */
    CMtmout,        /* timeout n (activate timeouts for ep) */
    CMpreset,       /* reset the port */

    /* Hub feature selectors */
    Rportenable = 1,
    Rportreset  = 4,

};
```

⟨*struct* Hcitype*(arm)* 907a⟩≡ (933)

```
struct Hcitype
{
    char*   type;
    int (*reset)(Hci*);
};
```

⟨*macro* QID*(arm)* 907b⟩≡ (933)

```
#define QID(q)  ((int)(q).path)
```

⟨*global* usbctls*(arm)* 907c⟩≡ (933)

```
static Cmdtab usbctls[] =
{
```

```
    {CMdebug,   "debug",    2},
    {CMdump,    "dump",     1},
};
```

Uses CMdebug-2250 906 and CMdump-2251 906.

⟨*global* epctls *(arm)* 908a⟩≡ (933)

```
static Cmdtab epctls[] =
{
    {CMnew,     "new",      4},
    {CMnewdev,  "newdev",   3},
    {CMhub,     "hub",      1},
    {CMspeed,   "speed",    2},
    {CMmaxpkt,  "maxpkt",   2},
    {CMntds,    "ntds",     2},
    {CMpollival,    "pollival", 2},
    {CMsamplesz,    "samplesz", 2},
    {CMhz,      "hz",       2},
    {CMinfo,    "info",     0},
    {CMdetach,  "detach",   1},
    {CMaddress, "address",  1},
    {CMdebugep, "debug",    2},
    {CMclrhalt, "clrhalt",  1},
    {CMname,    "name",     2},
    {CMtmout,   "timeout",  2},
    {CMpreset,  "reset",    1},
};
```

Uses CMaddress-2264 906, CMclrhalt-2258 906, CMdebugep-2265 906, CMdetach-2263 906, CMhub-2254 906, CMhz-2260 906, CMinfo-2262 906, CMmaxpkt-2256 906, CMname-2266 906, CMnew-2252 906, CMnewdev-2253 906, CMntds-2257 906, CMpollival-2259 906, CMpreset-2268 906, CMsamplesz-2261 906, CMspeed-2255 906, and CMtmout-2267 906.

⟨*global* usbdir *(arm)* 908b⟩≡ (933)

```
static Dirtab usbdir[] =
{
    "ctl",      {Qctl},     0,  0666,
};
```

Uses Qctl-2242 906.

⟨*global* usbmodename *(arm)* 908c⟩≡ (933)

```
char *usbmodename[] =
{
    [OREAD] "r",
    [OWRITE]    "w",
    [ORDWR] "rw",
};
```

⟨*global* ttname *(arm)* 908d⟩≡ (933)

```
static char *ttname[] =
{
    [Tnone] "none",
    [Tctl]  "control",
    [Tiso]  "iso",
    [Tintr] "interrupt",
    [Tbulk] "bulk",
};
```

⟨*global* spname*(arm)* 909a⟩≡ (933)
```
static char *spname[] =
{
    [Fullspeed] "full",
    [Lowspeed]  "low",
    [Highspeed] "high",
    [Nospeed]   "no",
};
```

⟨*global* debug*(arm)* 909b⟩≡ (933)
```
static int  debug;
```

⟨*global* hcitypes*(arm)* 909c⟩≡ (933)
```
static Hcitype  hcitypes[Nhcis];
```

⟨*global* hcis*(arm)* 909d⟩≡ (933)
```
static Hci* hcis[Nhcis];
```

⟨*global* epslck*(arm)* 909e⟩≡ (933)
```
static QLock    epslck;     /* add, del, lookup endpoints */
```

⟨*global* eps*(arm)* 909f⟩≡ (933)
```
static Ep*  eps[Neps];  /* all endpoints known */
```

⟨*global* epmax*(arm)* 909g⟩≡ (933)
```
static int  epmax;      /* 1 + last endpoint index used  */
```

⟨*global* usbidgen*(arm)* 909h⟩≡ (933)
```
static int  usbidgen;   /* device address generator */
```

⟨*function* seprintdata*(arm)* 909i⟩≡ (933)
```
/*
 * Is there something like this in a library? should it be?
 */
char*
seprintdata(char *s, char *se, uchar *d, int n)
{
    int i, l;

    s = seprint(s, se, " %#p[%d]: ", d, n);
    l = n;
    if(l > 10)
        l = 10;
    for(i=0; i<l; i++)
        s = seprint(s, se, " %2.2ux", d[i]);
    if(l < n)
        s = seprint(s, se, "...");
    return s;
}
```

⟨*function* name2speed*(arm)* 909j⟩≡ (933)
```
static int
name2speed(char *name)
{
    int i;

    for(i = 0; i < nelem(spname); i++)
        if(strcmp(name, spname[i]) == 0)
            return i;
    return Nospeed;
}
```
Uses nelem 853i and spname-2276 909a.

⟨*function* name2ttype*(arm)* 910a⟩≡ (933)

```
static int
name2ttype(char *name)
{
    int i;

    for(i = 0; i < nelem(ttname); i++)
        if(strcmp(name, ttname[i]) == 0)
            return i;
    /* may be a std. USB ep. type */
    i = strtol(name, nil, 0);
    switch(i+1){
    case Tctl:
    case Tiso:
    case Tbulk:
    case Tintr:
        return i+1;
    default:
        return Tnone;
    }
}
```

Uses nelem 853i and ttname-2275 908d.

⟨*function* name2mode*(arm)* 910b⟩≡ (933)

```
static int
name2mode(char *mode)
{
    int i;

    for(i = 0; i < nelem(usbmodename); i++)
        if(strcmp(mode, usbmodename[i]) == 0)
            return i;
    return -1;
}
```

Uses nelem 853i and usbmodename 908c.

⟨*function* qid2epidx*(arm)* 910c⟩≡ (933)

```
static int
qid2epidx(int q)
{
    q = (q-Qep0dir)/4;
    if(q < 0 || q >= epmax || eps[q] == nil)
        return -1;
    return q;
}
```

Uses Qep0dir-2243 906, epmax-2282 909g, and eps-2281 909f.

⟨*function* isqtype*(arm)* 910d⟩≡ (933)

```
static int
isqtype(int q, int type)
{
    if(q < Qep0dir)
        return 0;
    q -= Qep0dir;
    return (q & 3) == type;
}
```

Uses Qep0dir-2243 906.

⟨*function* addhcitype*(arm)* 911a⟩≡ (933)

```
void
addhcitype(char* t, int (*r)(Hci*))
{
    static int ntype;

    if(ntype == Nhcis)
        panic("too many USB host interface types");
    hcitypes[ntype].type = t;
    hcitypes[ntype].reset = r;
    ntype++;
}
```

Uses hcitypes-2278 909c.

⟨*function* seprintep*(arm)* 911b⟩≡ (933)

```
static char*
seprintep(char *s, char *se, Ep *ep, int all)
{
    static char* dsnames[] = { "config", "enabled", "detached", "reset" };
    Udev *d;
    int i;
    int di;

    d = ep->dev;

    qlock(ep);
    if(waserror()){
        qunlock(ep);
        nexterror();
    }
    di = ep->dev->nb;
    if(all)
        s = seprint(s, se, "dev %d ep %d ", di, ep->nb);
    s = seprint(s, se, "%s", dsnames[ep->dev->state]);
    s = seprint(s, se, " %s", ttname[ep->ttype]);
    assert(ep->mode == OREAD || ep->mode == OWRITE || ep->mode == ORDWR);
    s = seprint(s, se, " %s", usbmodename[ep->mode]);
    s = seprint(s, se, " speed %s", spname[d->speed]);
    s = seprint(s, se, " maxpkt %ld", ep->maxpkt);
    s = seprint(s, se, " pollival %ld", ep->pollival);
    s = seprint(s, se, " samplesz %ld", ep->samplesz);
    s = seprint(s, se, " hz %ld", ep->hz);
    s = seprint(s, se, " hub %d", ep->dev->hub);
    s = seprint(s, se, " port %d", ep->dev->port);
    if(ep->inuse)
        s = seprint(s, se, " busy");
    else
        s = seprint(s, se, " idle");
    if(all){
        s = seprint(s, se, " load %uld", ep->load);
        s = seprint(s, se, " ref %ld addr %#p", ep->ref, ep);
        s = seprint(s, se, " idx %d", ep->idx);
        if(ep->name != nil)
            s = seprint(s, se, " name '%s'", ep->name);
        if(ep->tmout != 0)
            s = seprint(s, se, " tmout");
        if(ep == ep->ep0){
            s = seprint(s, se, " ctlrno %#x", ep->hp->ctlrno);
            s = seprint(s, se, " eps:");
            for(i = 0; i < nelem(d->eps); i++)
```

```
                if(d->eps[i] != nil)
                    s = seprint(s, se, " ep%d.%d", di, i);
            }
        }
        if(ep->info != nil)
            s = seprint(s, se, "\n%s %s\n", ep->info, ep->hp->type);
        else
            s = seprint(s, se, "\n");
        qunlock(ep);
        poperror();
        return s;
    }
```

Uses ORDWR 301c, OREAD 853i, OWRITE, assert 853f, nelem 853i, nexterror, poperror 666d, qlock() 81f, qunlock() 82c, spname-2276 909a, ttname-2275 908d, usbmodename 908c, and waserror 666d.

⟨*function* epalloc*(arm)* 912a⟩≡ (933)

```
    static Ep*
    epalloc(Hci *hp)
    {
        Ep *ep;
        int i;

        ep = smalloc(sizeof(Ep));
        ep->ref = 1;
        qlock(&epslck);
        for(i = 0; i < Neps; i++)
            if(eps[i] == nil)
                break;
        if(i == Neps){
            qunlock(&epslck);
            free(ep);
            print("usb: bug: too few endpoints.\n");
            return nil;
        }
        ep->idx = i;
        if(epmax <= i)
            epmax = i+1;
        eps[i] = ep;
        ep->hp = hp;
        ep->maxpkt = 8;
        ep->ntds = 1;
        ep->samplesz = ep->pollival = ep->hz = 0; /* make them void */
        qunlock(&epslck);
        return ep;
    }
```

Uses epmax-2282 909g, eps-2281 909f, epslck-2280 909e, free() 104a, print, qlock() 81f, qunlock() 82c, and smalloc() 104b.

⟨*function* getep*(arm)* 912b⟩≡ (933)

```
    static Ep*
    getep(int i)
    {
        Ep *ep;

        if(i < 0 || i >= epmax || eps[i] == nil)
            return nil;
        qlock(&epslck);
        ep = eps[i];
        if(ep != nil)
            incref(ep);
        qunlock(&epslck);
```

```
    return ep;
}
```
Uses epmax-2282 909g, eps-2281 909f, epslck-2280 909e, incref() 87b, qlock() 81f, and qunlock() 82c.

⟨*function* putep*(arm)* 913a⟩≡ (933)
```
static void
putep(Ep *ep)
{
    Udev *d;

    if(ep != nil && decref(ep) == 0){
        d = ep->dev;
        deprint("usb: ep%d.%d %#p released\n", d->nb, ep->nb, ep);
        qlock(&epslck);
        eps[ep->idx] = nil;
        if(ep->idx == epmax-1)
            epmax--;
        if(ep == ep->ep0 && ep->dev != nil && ep->dev->nb == usbidgen)
            usbidgen--;
        qunlock(&epslck);
        if(d != nil){
            qlock(ep->ep0);
            d->eps[ep->nb] = nil;
            qunlock(ep->ep0);
        }
        if(ep->ep0 != ep){
            putep(ep->ep0);
            ep->ep0 = nil;
        }
        free(ep->info);
        free(ep->name);
        free(ep);
    }
}
```
Uses decref() 87c, deprint, epmax-2282 909g, eps-2281 909f, epslck-2280 909e, free() 104a, putep() 913a, qlock() 81f, qunlock() 82c, and usbidgen-2283 909h.

⟨*function* dumpeps*(arm)* 913b⟩≡ (933)
```
static void
dumpeps(void)
{
    int i;
    static char buf[512];
    char *s;
    char *e;
    Ep *ep;

    print("usb dump eps: epmax %d Neps %d (ref=1+ for dump):\n", epmax, Neps);
    for(i = 0; i < epmax; i++){
        s = buf;
        e = buf+sizeof(buf);
        ep = getep(i);
        if(ep != nil){
            if(waserror()){
                putep(ep);
                nexterror();
            }
            s = seprint(s, e, "ep%d.%d ", ep->dev->nb, ep->nb);
            seprintep(s, e, ep, 1);
            print("%s", buf);
```

```
            ep->hp->seprintep(buf, e, ep);
            print("%s", buf);
            poperror();
            putep(ep);
        }
    }
    print("usb dump hcis:\n");
    for(i = 0; i < Nhcis; i++)
        if(hcis[i] != nil)
            hcis[i]->dump(hcis[i]);
}
```

Uses epmax-2282 909g, getep() 912b, hcis-2279 909d, nexterror, poperror 666d, print, putep() 913a, and waserror 666d.

⟨*function* newusbid*(arm)* 914a⟩≡ (933)

```
static int
newusbid(Hci *)
{
    int id;

    qlock(&epslck);
    id = ++usbidgen;
    if(id >= 0x7F)
        print("#u: too many device addresses; reuse them more\n");
    qunlock(&epslck);
    return id;
}
```

Uses epslck-2280 909e, print, qlock() 81f, qunlock() 82c, and usbidgen-2283 909h.

⟨*function* newdev*(arm)* 914b⟩≡ (933)

```
/*
 * Create endpoint 0 for a new device
 */
static Ep*
newdev(Hci *hp, int ishub, int isroot)
{
    Ep *ep;
    Udev *d;

    ep = epalloc(hp);
    d = ep->dev = smalloc(sizeof(Udev));
    d->nb = newusbid(hp);
    d->eps[0] = ep;
    ep->nb = 0;
    ep->toggle[0] = ep->toggle[1] = 0;
    d->ishub = ishub;
    d->isroot = isroot;
    if(hp->highspeed != 0)
        d->speed = Highspeed;
    else
        d->speed = Fullspeed;
    d->state = Dconfig;     /* address not yet set */
    ep->dev = d;
    ep->ep0 = ep;           /* no ref counted here */
    ep->ttype = Tctl;
    ep->tmout = Xfertmout;
    ep->mode = ORDWR;
    dprint("newdev %#p ep%d.%d %#p\n", d, d->nb, ep->nb, ep);
    return ep;
}
```

Uses ORDWR 301c, dprint, epalloc() 912a, newusbid() 914a, and smalloc() 104b.

⟨*function* newdevep*(arm)* 915a⟩≡ (933)

```
/*
 * Create a new endpoint for the device
 * accessed via the given endpoint 0.
 */
static Ep*
newdevep(Ep *ep, int i, int tt, int mode)
{
    Ep *nep;
    Udev *d;

    d = ep->dev;
    if(d->eps[i] != nil)
        error("endpoint already in use");
    nep = epalloc(ep->hp);
    incref(ep);
    d->eps[i] = nep;
    nep->nb = i;
    nep->toggle[0] = nep->toggle[1] = 0;
    nep->ep0 = ep;
    nep->dev = ep->dev;
    nep->mode = mode;
    nep->ttype = tt;
    nep->debug = ep->debug;
    /* set defaults */
    switch(tt){
    case Tctl:
        nep->tmout = Xfertmout;
        break;
    case Tintr:
        nep->pollival = 10;
        break;
    case Tiso:
        nep->tmout = Xfertmout;
        nep->pollival = 10;
        nep->samplesz = 4;
        nep->hz = 44100;
        break;
    }
    deprint("newdevep ep%d.%d %#p\n", d->nb, nep->nb, nep);
    return ep;
}
```

Uses `deprint`, `epalloc()` 912a, and `incref()` 87b.

⟨*function* epdataperm*(arm)* 915b⟩≡ (933)

```
static int
epdataperm(int mode)
{

    switch(mode){
    case OREAD:
        return 0440|DMEXCL;
        break;
    case OWRITE:
        return 0220|DMEXCL;
        break;
    default:
        return 0660|DMEXCL;
    }
}
```

Uses DMEXCL, OREAD 853i, and OWRITE.

⟨*function* usbgen*(arm)* 916⟩≡ (933)

```
static int
usbgen(Chan *c, char *, Dirtab*, int, int s, Dir *dp)
{
    Qid q;
    Dirtab *dir;
    int perm;
    char *se;
    Ep *ep;
    int nb;
    int mode;

    if(0)ddprint("usbgen q %#x s %d...", QID(c->qid), s);
    if(s == DEVDOTDOT){
        if(QID(c->qid) <= Qusbdir){
            mkqid(&q, Qdir, 0, QTDIR);
            devdir(c, q, "#u", 0, eve, 0555, dp);
        }else{
            mkqid(&q, Qusbdir, 0, QTDIR);
            devdir(c, q, "usb", 0, eve, 0555, dp);
        }
        if(0)ddprint("ok\n");
        return 1;
    }

    switch(QID(c->qid)){
    case Qdir:              /* list #u */
        if(s == 0){
            mkqid(&q, Qusbdir, 0, QTDIR);
            devdir(c, q, "usb", 0, eve, 0555, dp);
            if(0)ddprint("ok\n");
            return 1;
        }
        s--;
        if(s < 0 || s >= epmax)
            goto Fail;
        ep = getep(s);
        if(ep == nil || ep->name == nil){
            if(ep != nil)
                putep(ep);
            if(0)ddprint("skip\n");
            return 0;
        }
        if(waserror()){
            putep(ep);
            nexterror();
        }
        mkqid(&q, Qep0io+s*4, 0, QTFILE);
        devdir(c, q, ep->name, 0, eve, epdataperm(ep->mode), dp);
        putep(ep);
        poperror();
        if(0)ddprint("ok\n");
        return 1;

    case Qusbdir:               /* list #u/usb */
    Usbdir:
        if(s < nelem(usbdir)){
            dir = &usbdir[s];
```

```
        mkqid(&q, dir->qid.path, 0, QTFILE);
        devdir(c, q, dir->name, dir->length, eve, dir->perm, dp);
        if(0)ddprint("ok\n");
        return 1;
    }
    s -= nelem(usbdir);
    if(s < 0 || s >= epmax)
        goto Fail;
    ep = getep(s);
    if(ep == nil){
        if(0)ddprint("skip\n");
        return 0;
    }
    if(waserror()){
        putep(ep);
        nexterror();
    }
    se = up->genbuf+sizeof(up->genbuf);
    seprint(up->genbuf, se, "ep%d.%d", ep->dev->nb, ep->nb);
    mkqid(&q, Qep0dir+4*s, 0, QTDIR);
    putep(ep);
    poperror();
    devdir(c, q, up->genbuf, 0, eve, 0755, dp);
    if(0)ddprint("ok\n");
    return 1;

case Qctl:
    s = 0;
    goto Usbdir;

default:                /* list #u/usb/epN.M */
    nb = qid2epidx(QID(c->qid));
    ep = getep(nb);
    if(ep == nil)
        goto Fail;
    mode = ep->mode;
    putep(ep);
    if(isqtype(QID(c->qid), Qepdir)){
    Epdir:
        switch(s){
        case 0:
            mkqid(&q, Qep0io+nb*4, 0, QTFILE);
            perm = epdataperm(mode);
            devdir(c, q, "data", 0, eve, perm, dp);
            break;
        case 1:
            mkqid(&q, Qep0ctl+nb*4, 0, QTFILE);
            devdir(c, q, "ctl", 0, eve, 0664, dp);
            break;
        default:
            goto Fail;
        }
    }else if(isqtype(QID(c->qid), Qepctl)){
        s = 1;
        goto Epdir;
    }else{
        s = 0;
        goto Epdir;
    }
    if(0)ddprint("ok\n");
```

```
        return 1;
    }
Fail:
    if(0)ddprint("fail\n");
    return -1;
}
```

Uses DEVDOTDOT 805, QID-2271 907b, QTDIR 70b, QTFILE, Qctl-2242 906, Qdir-2240 906, Qep0ctl-2245 906, Qep0dir-2243 906, Qep0io-2244 906, Qepctl-2249 906, Qepdir-2247 906, Qusbdir-2241 906, ddprint, devdir() 427c, epdataperm() 915b, epmax-2282 909g, eve 591a, getep() 912b, isqtype() 910d, nelem 853i, nexterror, poperror 666d, putep() 913a, qid2epidx() 910c, up, usbdir-2274 908b, and waserror 666d.

⟨*function* hciprobe*(arm)* 918a⟩≡ (933)

```
static Hci*
hciprobe(int cardno, int ctlrno)
{
    Hci *hp;
    char *type;
    char name[64];
    static int epnb = 1;    /* guess the endpoint nb. for the controller */

    ddprint("hciprobe %d %d\n", cardno, ctlrno);
    hp = smalloc(sizeof(Hci));
    hp->ctlrno = ctlrno;

    if(cardno < 0)
        for(cardno = 0; cardno < Nhcis; cardno++){
            if(hcitypes[cardno].type == nil)
                break;
            type = hp->type;
            if(type==nil || *type==0)
                type = "uhci";
            if(cistrcmp(hcitypes[cardno].type, type) == 0)
                break;
        }

    if(cardno >= Nhcis || hcitypes[cardno].type == nil){
        free(hp);
        return nil;
    }
    dprint("%s...", hcitypes[cardno].type);
    if(hcitypes[cardno].reset(hp) < 0){
        free(hp);
        return nil;
    }

    snprint(name, sizeof(name), "usb%s", hcitypes[cardno].type);
    arch_intrenable(hp->irq, hp->interrupt, hp, UNKNOWN, name);

    print("#u/usb/ep%d.0: %s: port %#luX irq %d\n",
        epnb, hcitypes[cardno].type, hp->port, hp->irq);
    epnb++;

    return hp;
}
```

Uses ddprint, dprint, free() 104a, hcitypes-2278 909c, print, and smalloc() 104b.

⟨*function* usbreset*(arm)* 918b⟩≡ (933)

```
static void
usbreset(void)
{
```

```
    int cardno, ctlrno;
    Hci *hp;

    dprint("usbreset\n");

    for(ctlrno = 0; ctlrno < Nhcis; ctlrno++)
        if((hp = hciprobe(-1, ctlrno)) != nil)
            hcis[ctlrno] = hp;
    cardno = ctlrno = 0;
    while(cardno < Nhcis && ctlrno < Nhcis && hcitypes[cardno].type != nil)
        if(hcis[ctlrno] != nil)
            ctlrno++;
        else{
            hp = hciprobe(cardno, ctlrno);
            if(hp == nil)
                cardno++;
            hcis[ctlrno++] = hp;
        }
    if(hcis[Nhcis-1] != nil)
        print("usbreset: bug: Nhcis too small\n");
}
```

Uses dprint, hciprobe() 918a, hcis-2279 909d, hcitypes-2278 909c, and print.

⟨*function* usbinit*(arm)* 919a⟩≡ (933)

```
static void
usbinit(void)
{
    Hci *hp;
    int ctlrno;
    Ep *d;
    char info[40];

    dprint("usbinit\n");
    for(ctlrno = 0; ctlrno < Nhcis; ctlrno++){
        hp = hcis[ctlrno];
        if(hp != nil){
            if(hp->init != nil)
                hp->init(hp);
            d = newdev(hp, 1, 1);       /* new root hub */
            d->dev->state = Denabled;   /* although addr == 0 */
            d->maxpkt = 64;
            snprint(info, sizeof(info), "ports %d", hp->nports);
            kstrdup(&d->info, info);
        }
    }
}
```

Uses dprint, hcis-2279 909d, kstrdup() 669b, and newdev() 914b.

⟨*function* usbattach*(arm)* 919b⟩≡ (933)

```
static Chan*
usbattach(char *spec)
{
    return devattach(L'u', spec);
}
```

Uses devattach() 297b.

⟨*function* usbwalk*(arm)* 919c⟩≡ (933)

```
static Walkqid*
usbwalk(Chan *c, Chan *nc, char **name, int nname)
```

```
{
    return devwalk(c, nc, name, nname, nil, 0, usbgen);
}
```
Uses devwalk() 429b and usbgen() 916.

⟨*function* usbstat*(arm)* 920a⟩≡ (933)
```
static int
usbstat(Chan *c, uchar *db, int n)
{
    return devstat(c, db, n, nil, 0, usbgen);
}
```
Uses devstat() 431 and usbgen() 916.

⟨*function* usbload*(arm)* 920b⟩≡ (933)
```
/*
 * µs for the given transfer, for bandwidth allocation.
 * This is a very rough worst case for what 5.11.3
 * of the usb 2.0 spec says.
 * Also, we are using maxpkt and not actual transfer sizes.
 * Only when we are sure we
 * are not exceeding b/w might we consider adjusting it.
 */
static ulong
usbload(int speed, int maxpkt)
{
    enum{ Hostns = 1000, Hubns = 333 };
    ulong l;
    ulong bs;

    l = 0;
    bs = 10UL * maxpkt;
    switch(speed){
    case Highspeed:
        l = 55*8*2 + 2 * (3 + bs) + Hostns;
        break;
    case Fullspeed:
        l = 9107 + 84 * (4 + bs) + Hostns;
        break;
    case Lowspeed:
        l = 64107 + 2 * Hubns + 667 * (3 + bs) + Hostns;
        break;
    default:
        print("usbload: bad speed %d\n", speed);
        /* let it run */
    }
    return l / 1000UL;  /* in µs */
}
```
Uses print.

⟨*function* usbopen*(arm)* 920c⟩≡ (933)
```
static Chan*
usbopen(Chan *c, int omode)
{
    int q;
    Ep *ep;
    int mode;

    mode = openmode(omode);
    q = QID(c->qid);
```

```
    if(q >= Qep0dir && qid2epidx(q) < 0)
        error(Eio);
    if(q < Qep0dir || isqtype(q, Qepctl) || isqtype(q, Qepdir))
        return devopen(c, omode, nil, 0, usbgen);

    ep = getep(qid2epidx(q));
    if(ep == nil)
        error(Eio);
    deprint("usbopen q %#x fid %d omode %d\n", q, c->fid, mode);
    if(waserror()){
        putep(ep);
        nexterror();
    }
    qlock(ep);
    if(ep->inuse){
        qunlock(ep);
        error(Einuse);
    }
    ep->inuse = 1;
    qunlock(ep);
    if(waserror()){
        ep->inuse = 0;
        nexterror();
    }
    if(mode != OREAD && ep->mode == OREAD)
        error(Eperm);
    if(mode != OWRITE && ep->mode == OWRITE)
        error(Eperm);
    if(ep->ttype == Tnone)
        error(Enotconf);
    ep->clrhalt = 0;
    ep->rhrepl = -1;
    if(ep->load == 0)
        ep->load = usbload(ep->dev->speed, ep->maxpkt);
    ep->hp->epopen(ep);

    poperror(); /* ep->inuse */
    poperror(); /* don't putep(): ref kept for fid using the ep. */

    c->mode = mode;
    c->flag |= COPEN;
    c->offset = 0;
    c->aux = nil;   /* paranoia */
    return c;
}
```

Uses COPEN 294b, Enotconf 664d, OREAD 853i, OWRITE, QID-2271 907b, Qep0dir-2243 906, Qepctl-2249 906, Qepdir-2247 906, deprint, getep() 912b, isqtype() 910d, nexterror, openmode() 302b, poperror 666d, putep() 913a, qid2epidx() 910c, qlock() 81f, qunlock() 82c, usbgen() 916, usbload() 920b, and waserror 666d.

⟨*function* epclose*(arm)* 921⟩≡ (933)

```
static void
epclose(Ep *ep)
{
    qlock(ep);
    if(waserror()){
        qunlock(ep);
        nexterror();
    }
    if(ep->inuse){
```

```
        ep->hp->epclose(ep);
        ep->inuse = 0;
    }
    qunlock(ep);
    poperror();
}
```

Uses nexterror, poperror 666d, qlock() 81f, qunlock() 82c, and waserror 666d.

⟨*function* usbclose*(arm)* 922a⟩≡ (933)

```
static void
usbclose(Chan *c)
{
    int q;
    Ep *ep;

    q = QID(c->qid);
    if(q < Qep0dir || isqtype(q, Qepctl) || isqtype(q, Qepdir))
        return;

    ep = getep(qid2epidx(q));
    if(ep == nil)
        return;
    deprint("usbclose q %#x fid %d ref %ld\n", q, c->fid, ep->ref);
    if(waserror()){
        putep(ep);
        nexterror();
    }
    if(c->flag & COPEN){
        free(c->aux);
        c->aux = nil;
        epclose(ep);
        putep(ep);  /* release ref kept since usbopen */
        c->flag &= ~COPEN;
    }
    poperror();
    putep(ep);
}
```

Uses COPEN 294b, QID-2271 907b, Qep0dir-2243 906, Qepctl-2249 906, Qepdir-2247 906, deprint, free() 104a, getep() 912b, isqtype() 910d, nexterror, poperror 666d, putep() 913a, qid2epidx() 910c, and waserror 666d.

⟨*function* ctlread*(arm)* 922b⟩≡ (933)

```
static long
ctlread(Chan *c, void *a, long n, vlong offset)
{
    int q;
    char *s;
    char *us;
    char *se;
    Ep *ep;
    int i;

    q = QID(c->qid);
    us = s = smalloc(READSTR);
    se = s + READSTR;
    if(waserror()){
        free(us);
        nexterror();
    }
    if(q == Qctl)
        for(i = 0; i < epmax; i++){
```

```
            ep = getep(i);
            if(ep != nil){
                if(waserror()){
                    putep(ep);
                    nexterror();
                }
                s = seprint(s, se, "ep%d.%d ", ep->dev->nb, ep->nb);
                s = seprintep(s, se, ep, 0);
                poperror();
            }
            putep(ep);
        }
    else{
        ep = getep(qid2epidx(q));
        if(ep == nil)
            error(Eio);
        if(waserror()){
            putep(ep);
            nexterror();
        }
        if(c->aux != nil){
            /* After a new endpoint request we read
             * the new endpoint name back.
             */
            strecpy(s, se, c->aux);
            free(c->aux);
            c->aux = nil;
        }else
            seprintep(s, se, ep, 0);
        poperror();
        putep(ep);
    }
    n = readstr(offset, a, n, us);
    poperror();
    free(us);
    return n;
}
```

Uses QID-2271 907b, Qctl-2242 906, READSTR 867a, epmax-2282 909g, free() 104a, getep() 912b, nexterror, poperror 666d, putep() 913a, qid2epidx() 910c, readstr() 670d, smalloc() 104b, and waserror 666d.

⟨*function* rhubread*(arm)* 923⟩≡ (933)

```
/*
 * Fake root hub emulation.
 */
static long
rhubread(Ep *ep, void *a, long n)
{
    char *b;

    if(ep->dev->isroot == 0 || ep->nb != 0 || n < 2)
        return -1;
    if(ep->rhrepl < 0)
        return -1;

    b = a;
    memset(b, 0, n);
    PUT2(b, ep->rhrepl);
    ep->rhrepl = -1;
    return n;
}
```

Uses PUT2.

⟨*function* rhubwrite*(arm)* 924a⟩≡ (933)
```
static long
rhubwrite(Ep *ep, void *a, long n)
{
    uchar *s;
    int cmd;
    int feature;
    int port;
    Hci *hp;

    if(ep->dev == nil || ep->dev->isroot == 0 || ep->nb != 0)
        return -1;
    if(n != Rsetuplen)
        error("root hub is a toy hub");
    ep->rhrepl = -1;
    s = a;
    if(s[Rtype] != (Rh2d|Rclass|Rother) && s[Rtype] != (Rd2h|Rclass|Rother))
        error("root hub is a toy hub");
    hp = ep->hp;
    cmd = s[Rreq];
    feature = GET2(s+Rvalue);
    port = GET2(s+Rindex);
    if(port < 1 || port > hp->nports)
        error("bad hub port number");
    switch(feature){
    case Rportenable:
        ep->rhrepl = hp->portenable(hp, port, cmd == Rsetfeature);
        break;
    case Rportreset:
        ep->rhrepl = hp->portreset(hp, port, cmd == Rsetfeature);
        break;
    case Rgetstatus:
        ep->rhrepl = hp->portstatus(hp, port);
        break;
    default:
        ep->rhrepl = 0;
    }
    return n;
}
```
Uses GET2, Rclass, Rd2h, Rgetstatus, Rh2d, Rother, Rportenable-2269 906, Rportreset-2270 906, and Rsetfeature.

⟨*function* usbread*(arm)* 924b⟩≡ (933)
```
static long
usbread(Chan *c, void *a, long n, vlong offset)
{
    int q;
    Ep *ep;
    int nr;

    q = QID(c->qid);

    if(c->qid.type == QTDIR)
        return devdirread(c, a, n, nil, 0, usbgen);

    if(q == Qctl || isqtype(q, Qepctl))
        return ctlread(c, a, n, offset);

    ep = getep(qid2epidx(q));
```

```
    if(ep == nil)
        error(Eio);
    if(waserror()){
        putep(ep);
        nexterror();
    }
    if(ep->dev->state == Ddetach)
        error(Edetach);
    if(ep->mode == OWRITE || ep->inuse == 0)
        error(Ebadusefd);
    switch(ep->ttype){
    case Tnone:
        error("endpoint not configured");
    case Tctl:
        nr = rhubread(ep, a, n);
        if(nr >= 0){
            n = nr;
            break;
        }
        /* else fall */
    default:
        ddeprint("\nusbread q %#x fid %d cnt %ld off %lld\n",q,c->fid,n,offset);
        n = ep->hp->epread(ep, a, n);
        break;
    }
    poperror();
    putep(ep);
    return n;
}
```

Uses Ebadusefd 664d, Edetach 664d, OWRITE, QID-2271 907b, QTDIR 70b, Qctl-2242 906, Qepctl-2249 906, ctlread() 922b, devdirread() 432c, getep() 912b, isqtype() 910d, nexterror, poperror 666d, putep() 913a, qid2epidx() 910c, rhubread() 923, usbgen() 916, and waserror 666d.

⟨*function* pow2*(arm)* 925a⟩≡ (933)

```
static long
pow2(int n)
{
    return 1 << n;
}
```

⟨*function* setmaxpkt*(arm)* 925b⟩≡ (933)

```
static void
setmaxpkt(Ep *ep, char* s)
{
    long spp;   /* samples per packet */

    if(ep->dev->speed == Highspeed)
        spp = (ep->hz * ep->pollival * ep->ntds + 7999) / 8000;
    else
        spp = (ep->hz * ep->pollival + 999) / 1000;
    ep->maxpkt = spp * ep->samplesz;
    deprint("usb: %s: setmaxpkt: hz %ld poll %ld"
        " ntds %d %s speed -> spp %ld maxpkt %ld\n", s,
        ep->hz, ep->pollival, ep->ntds, spname[ep->dev->speed],
        spp, ep->maxpkt);
    if(ep->maxpkt > 1024){
        print("usb: %s: maxpkt %ld > 1024. truncating\n", s, ep->maxpkt);
        ep->maxpkt = 1024;
    }
}
```

Uses deprint, print, and spname-2276 909a.

⟨*function* epctl*(arm)* 926⟩≡ (933)

```
/*
 * Many endpoint ctls. simply update the portable representation
 * of the endpoint. The actual controller driver will look
 * at them to setup the endpoints as dictated.
 */
static long
epctl(Ep *ep, Chan *c, void *a, long n)
{
    int i, l, mode, nb, tt;
    char *b, *s;
    Cmdbuf *cb;
    Cmdtab *ct;
    Ep *nep;
    Udev *d;
    static char *Info = "info ";

    d = ep->dev;

    cb = parsecmd(a, n);
    if(waserror()){
        free(cb);
        nexterror();
    }
    ct = lookupcmd(cb, epctls, nelem(epctls));
    if(ct == nil)
        error(Ebadctl);
    i = ct->index;
    if(i == CMnew || i == CMspeed || i == CMhub || i == CMpreset)
        if(ep != ep->ep0)
            error("allowed only on a setup endpoint");
    if(i != CMclrhalt && i != CMdetach && i != CMdebugep && i != CMname)
        if(ep != ep->ep0 && ep->inuse != 0)
            error("must configure before using");
    switch(i){
    case CMnew:
        deprint("usb epctl %s\n", cb->f[0]);
        nb = strtol(cb->f[1], nil, 0);
        if(nb < 0 || nb >= Ndeveps)
            error("bad endpoint number");
        tt = name2ttype(cb->f[2]);
        if(tt == Tnone)
            error("unknown endpoint type");
        mode = name2mode(cb->f[3]);
        if(mode < 0)
            error("unknown i/o mode");
        newdevep(ep, nb, tt, mode);
        break;
    case CMnewdev:
        deprint("usb epctl %s\n", cb->f[0]);
        if(ep != ep->ep0 || d->ishub == 0)
            error("not a hub setup endpoint");
        l = name2speed(cb->f[1]);
        if(l == Nospeed)
            error("speed must be full|low|high");
        nep = newdev(ep->hp, 0, 0);
        nep->dev->speed = l;
        if(nep->dev->speed  != Lowspeed)
```

```
        nep->maxpkt = 64;   /* assume full speed */
    nep->dev->hub = d->nb;
    nep->dev->port = atoi(cb->f[2]);
    /* next read request will read
     * the name for the new endpoint
     */
    l = sizeof(up->genbuf);
    snprint(up->genbuf, l, "ep%d.%d", nep->dev->nb, nep->nb);
    kstrdup(&c->aux, up->genbuf);
    break;
case CMhub:
    deprint("usb epctl %s\n", cb->f[0]);
    d->ishub = 1;
    break;
case CMspeed:
    l = name2speed(cb->f[1]);
    deprint("usb epctl %s %d\n", cb->f[0], l);
    if(l == Nospeed)
        error("speed must be full|low|high");
    qlock(ep->ep0);
    d->speed = l;
    qunlock(ep->ep0);
    break;
case CMmaxpkt:
    l = strtoul(cb->f[1], nil, 0);
    deprint("usb epctl %s %d\n", cb->f[0], l);
    if(l < 1 || l > 1024)
        error("maxpkt not in [1:1024]");
    qlock(ep);
    ep->maxpkt = l;
    qunlock(ep);
    break;
case CMntds:
    l = strtoul(cb->f[1], nil, 0);
    deprint("usb epctl %s %d\n", cb->f[0], l);
    if(l < 1 || l > 3)
        error("ntds not in [1:3]");
    qlock(ep);
    ep->ntds = l;
    qunlock(ep);
    break;
case CMpollival:
    if(ep->ttype != Tintr && ep->ttype != Tiso)
        error("not an intr or iso endpoint");
    l = strtoul(cb->f[1], nil, 0);
    deprint("usb epctl %s %d\n", cb->f[0], l);
    if(ep->ttype == Tiso ||
       (ep->ttype == Tintr && ep->dev->speed == Highspeed)){
        if(l < 1 || l > 16)
            error("pollival power not in [1:16]");
        l = pow2(l-1);
    }else
        if(l < 1 || l > 255)
            error("pollival not in [1:255]");
    qlock(ep);
    ep->pollival = l;
    if(ep->ttype == Tiso)
        setmaxpkt(ep, "pollival");
    qunlock(ep);
    break;
```

```
case CMsamplesz:
    if(ep->ttype != Tiso)
        error("not an iso endpoint");
    l = strtoul(cb->f[1], nil, 0);
    deprint("usb epctl %s %d\n", cb->f[0], l);
    if(l <= 0 || l > 8)
        error("samplesz not in [1:8]");
    qlock(ep);
    ep->samplesz = l;
    setmaxpkt(ep, "samplesz");
    qunlock(ep);
    break;
case CMhz:
    if(ep->ttype != Tiso)
        error("not an iso endpoint");
    l = strtoul(cb->f[1], nil, 0);
    deprint("usb epctl %s %d\n", cb->f[0], l);
    if(l <= 0 || l > 100000)
        error("hz not in [1:100000]");
    qlock(ep);
    ep->hz = l;
    setmaxpkt(ep, "hz");
    qunlock(ep);
    break;
case CMclrhalt:
    qlock(ep);
    deprint("usb epctl %s\n", cb->f[0]);
    ep->clrhalt = 1;
    qunlock(ep);
    break;
case CMinfo:
    deprint("usb epctl %s\n", cb->f[0]);
    l = strlen(Info);
    s = a;
    if(n < l+2 || strncmp(Info, s, l) != 0)
        error(Ebadctl);
    if(n > 1024)
        n = 1024;
    b = smalloc(n);
    memmove(b, s+l, n-l);
    b[n-l] = 0;
    if(b[n-l-1] == '\n')
        b[n-l-1] = 0;
    qlock(ep);
    free(ep->info);
    ep->info = b;
    qunlock(ep);
    break;
case CMaddress:
    deprint("usb epctl %s\n", cb->f[0]);
    ep->dev->state = Denabled;
    break;
case CMdetach:
    if(ep->dev->isroot != 0)
        error("can't detach a root hub");
    deprint("usb epctl %s ep%d.%d\n",
        cb->f[0], ep->dev->nb, ep->nb);
    ep->dev->state = Ddetach;
    /* Release file system ref. for its endpoints */
    for(i = 0; i < nelem(ep->dev->eps); i++)
```

```
            putep(ep->dev->eps[i]);
        break;
    case CMdebugep:
        if(strcmp(cb->f[1], "on") == 0)
            ep->debug = 1;
        else if(strcmp(cb->f[1], "off") == 0)
            ep->debug = 0;
        else
            ep->debug = strtoul(cb->f[1], nil, 0);
        print("usb: ep%d.%d debug %d\n",
            ep->dev->nb, ep->nb, ep->debug);
        break;
    case CMname:
        deprint("usb epctl %s %s\n", cb->f[0], cb->f[1]);
        validname(cb->f[1], 0);
        kstrdup(&ep->name, cb->f[1]);
        break;
    case CMtmout:
        deprint("usb epctl %s\n", cb->f[0]);
        if(ep->ttype == Tiso || ep->ttype == Tctl)
            error("ctl ignored for this endpoint type");
        ep->tmout = strtoul(cb->f[1], nil, 0);
        if(ep->tmout != 0 && ep->tmout < Xfertmout)
            ep->tmout = Xfertmout;
        break;
    case CMpreset:
        deprint("usb epctl %s\n", cb->f[0]);
        if(ep->ttype != Tctl)
            error("not a control endpoint");
        if(ep->dev->state != Denabled)
            error("forbidden on devices not enabled");
        ep->dev->state = Dreset;
        break;
    default:
        panic("usb: unknown epctl %d", ct->index);
    }
    free(cb);
    poperror();
    return n;
}
```

Uses CMaddress-2264 906, CMclrhalt-2258 906, CMdebugep-2265 906, CMdetach-2263 906, CMhub-2254 906, CMhz-2260 906, CMinfo-2262 906, CMmaxpkt-2256 906, CMname-2266 906, CMnew-2252 906, CMnewdev-2253 906, CMntds-2257 906, CMpollival-2259 906, CMpreset-2268 906, CMsamplesz-2261 906, CMspeed-2255 906, CMtmout-2267 906, deprint, epctls-2273 908a, free() 104a, kstrdup() 669b, lookupcmd() 595b, name2mode() 910b, name2speed() 909j, name2ttype() 910a, nelem 853i, newdev() 914b, newdevep() 915a, nexterror, parsecmd() 594c, poperror 666d, pow2() 925a, print, putep() 913a, qlock() 81f, qunlock() 82c, setmaxpkt() 925b, smalloc() 104b, up, validname() 308a, and waserror 666d.

⟨*function* usbctl*(arm)* 929⟩≡ (933)

```
static long
usbctl(void *a, long n)
{
    Cmdtab *ct;
    Cmdbuf *cb;
    Ep *ep;
    int i;

    cb = parsecmd(a, n);
    if(waserror()){
        free(cb);
```

```
        nexterror();
    }
    ct = lookupcmd(cb, usbctls, nelem(usbctls));
    dprint("usb ctl %s\n", cb->f[0]);
    switch(ct->index){
    case CMdebug:
        if(strcmp(cb->f[1], "on") == 0)
            debug = 1;
        else if(strcmp(cb->f[1], "off") == 0)
            debug = 0;
        else
            debug = strtol(cb->f[1], nil, 0);
        print("usb: debug %d\n", debug);
        for(i = 0; i < epmax; i++)
            if((ep = getep(i)) != nil){
                ep->hp->debug(ep->hp, debug);
                putep(ep);
            }
        break;
    case CMdump:
        dumpeps();
        break;
    }
    free(cb);
    poperror();
    return n;
}
```

Uses CMdebug-2250 906, CMdump-2251 906, debug-2277 909b, dprint, dumpeps() 913b, epmax-2282 909g, free() 104a, getep() 912b, lookupcmd() 595b, nelem 853i, nexterror, parsecmd() 594c, poperror 666d, print, putep() 913a, usbctls-2272 907c, and waserror 666d.

⟨*function* ctlwrite*(arm)* 930⟩≡ (933)

```
static long
ctlwrite(Chan *c, void *a, long n)
{
    int q;
    Ep *ep;

    q = QID(c->qid);
    if(q == Qctl)
        return usbctl(a, n);

    ep = getep(qid2epidx(q));
    if(ep == nil)
        error(Eio);
    if(waserror()){
        putep(ep);
        nexterror();
    }
    if(ep->dev->state == Ddetach)
        error(Edetach);
    if(isqtype(q, Qepctl) && c->aux != nil){
        /* Be sure we don't keep a cloned ep name */
        free(c->aux);
        c->aux = nil;
        error("read, not write, expected");
    }
    n = epctl(ep, c, a, n);
    putep(ep);
    poperror();
```

```
    return n;
}
```

Uses Edetach 664d, QID-2271 907b, Qctl-2242 906, Qepctl-2249 906, epctl() 926, free() 104a, getep() 912b, isqtype() 910d, nexterror, poperror 666d, putep() 913a, qid2epidx() 910c, usbctl() 929, and waserror 666d.

⟨*function* usbwrite*(arm)* 931a⟩≡ (933)

```
static long
usbwrite(Chan *c, void *a, long n, vlong off)
{
    int nr, q;
    Ep *ep;

    if(c->qid.type == QTDIR)
        error(Eisdir);

    q = QID(c->qid);

    if(q == Qctl || isqtype(q, Qepctl))
        return ctlwrite(c, a, n);

    ep = getep(qid2epidx(q));
    if(ep == nil)
        error(Eio);
    if(waserror()){
        putep(ep);
        nexterror();
    }
    if(ep->dev->state == Ddetach)
        error(Edetach);
    if(ep->mode == OREAD || ep->inuse == 0)
        error(Ebadusefd);

    switch(ep->ttype){
    case Tnone:
        error("endpoint not configured");
    case Tctl:
        nr = rhubwrite(ep, a, n);
        if(nr >= 0){
            n = nr;
            break;
        }
        /* else fall */
    default:
        ddeprint("\nusbwrite q %#x fid %d cnt %ld off %lld\n",q, c->fid, n, off);
        ep->hp->epwrite(ep, a, n);
    }
    putep(ep);
    poperror();
    return n;
}
```

Uses Ebadusefd 664d, Edetach 664d, OREAD 853i, QID-2271 907b, QTDIR 70b, Qctl-2242 906, Qepctl-2249 906, ctlwrite() 930, getep() 912b, isqtype() 910d, nexterror, poperror 666d, putep() 913a, qid2epidx() 910c, rhubwrite() 924a, and waserror 666d.

⟨*function* usbbread*(arm)* 931b⟩≡ (933)

```
Block*
usbbread(Chan *c, long n, ulong offset)
{
    Block *bp;
```

```
    bp = allocb(n);
    if(bp == 0)
        error(Enomem);
    if(waserror()) {
        freeb(bp);
        nexterror();
    }
    bp->wp += usbread(c, bp->wp, n, offset);
    poperror();
    return bp;
}
```

Uses `allocb()` 436b, `freeb()` 438a, `nexterror`, `poperror` 666d, `usbread()` 924b, and `waserror` 666d.

⟨*function* `usbbwrite`*(arm)* 932a⟩≡ (933)

```
long
usbbwrite(Chan *c, Block *bp, ulong offset)
{
    long n;

    if(waserror()) {
        freeb(bp);
        nexterror();
    }
    n = usbwrite(c, bp->rp, BLEN(bp), offset);
    poperror();
    freeb(bp);

    return n;
}
```

Uses `BLEN` 435b, `freeb()` 438a, `nexterror`, `poperror` 666d, `usbwrite()` 931a, and `waserror` 666d.

⟨*function* `usbshutdown`*(arm)* 932b⟩≡ (933)

```
void
usbshutdown(void)
{
    Hci *hp;
    int i;

    for(i = 0; i < Nhcis; i++){
        hp = hcis[i];
        if(hp == nil)
            continue;
        if(hp->shutdown == nil)
            print("#u: no shutdown function for %s\n", hp->type);
        else
            hp->shutdown(hp);
    }
}
```

Uses `hcis-2279` 909d and `print`.

⟨*global* `usbdevtab`*(arm)* 932c⟩≡ (933)

```
Dev usbdevtab = {
    .dc = L'u',
    .name = "usb",

    .reset = usbreset,
    .init = usbinit,
    .shutdown = usbshutdown,
    .attach = usbattach,
```

```
    .walk = usbwalk,
    .stat = usbstat,
    .open = usbopen,
    .create = devcreate,
    .close = usbclose,
    .read = usbread,
    .bread = usbbread,
    .write = usbwrite,
    .bwrite = usbbwrite,
    .remove = devremove,
    .wstat = devwstat,
};
```

Uses devcreate() 810, devremove() 810, devwstat() 810, usbattach() 919b, usbbread() 931b, usbbwrite() 932a, usbclose() 922a, usbinit() 919a, usbopen() 920c, usbread() 924b, usbreset() 918b, usbshutdown() 932b, usbstat() 920a, usbwalk() 919c, and usbwrite() 931a.

⟨buses/arm/devusb.c 933⟩≡

```
/*
 * USB device driver framework.
 *
 * This is in charge of providing access to actual HCIs
 * and providing I/O to the various endpoints of devices.
 * A separate user program (usbd) is in charge of
 * enumerating the bus, setting up endpoints and
 * starting devices (also user programs).
 *
 * The interface provided is a violation of the standard:
 * you're welcome.
 *
 * The interface consists of a root directory with several files
 * plus a directory (epN.M) with two files per endpoint.
 * A device is represented by its first endpoint, which
 * is a control endpoint automatically allocated for each device.
 * Device control endpoints may be used to create new endpoints.
 * Devices corresponding to hubs may also allocate new devices,
 * perhaps also hubs. Initially, a hub device is allocated for
 * each controller present, to represent its root hub. Those can
 * never be removed.
 *
 * All endpoints refer to the first endpoint (epN.0) of the device,
 * which keeps per-device information, and also to the HCI used
 * to reach them. Although all endpoints cache that information.
 *
 * epN.M/data files permit I/O and are considered DMEXCL.
 * epN.M/ctl files provide status info and accept control requests.
 *
 * Endpoints may be given file names to be listed also at #u,
 * for those drivers that have nothing to do after configuring the
 * device and its endpoints.
 *
 * Drivers for different controllers are kept at usb[oue]hci.c
 * It's likely we could factor out much from controllers into
 * a generic controller driver, the problem is that details
 * regarding how to handle toggles, tokens, Tds, etc. will
 * get in the way. Thus, code is probably easier the way it is.
 */
⟨kernel basic includes 49a⟩

#include    "io.h"
#include    "../port/usb.h"
```

```
typedef struct Hcitype Hcitype;
```

⟨*enum* _anon_*(*(buses/arm/devusb.c)(arm)*)* 906⟩

⟨*struct* Hcitype*(arm)* 907a⟩

⟨*macro* QID*(arm)* 907b⟩

⟨*global* usbctls*(arm)* 907c⟩

⟨*global* epctls*(arm)* 908a⟩

⟨*global* usbdir*(arm)* 908b⟩

⟨*global* usbmodename*(arm)* 908c⟩

⟨*global* ttname*(arm)* 908d⟩

⟨*global* spname*(arm)* 909a⟩

⟨*global* debug*(arm)* 909b⟩
⟨*global* hcitypes*(arm)* 909c⟩
⟨*global* hcis*(arm)* 909d⟩
⟨*global* epslck*(arm)* 909e⟩
⟨*global* eps*(arm)* 909f⟩
⟨*global* epmax*(arm)* 909g⟩
⟨*global* usbidgen*(arm)* 909h⟩

⟨*function* seprintdata*(arm)* 909i⟩

⟨*function* name2speed*(arm)* 909j⟩

⟨*function* name2ttype*(arm)* 910a⟩

⟨*function* name2mode*(arm)* 910b⟩

⟨*function* qid2epidx*(arm)* 910c⟩

⟨*function* isqtype*(arm)* 910d⟩

⟨*function* addhcitype*(arm)* 911a⟩

⟨*function* seprintep*(arm)* 911b⟩

⟨*function* epalloc*(arm)* 912a⟩

⟨*function* getep*(arm)* 912b⟩

⟨*function* putep*(arm)* 913a⟩

⟨*function* dumpeps*(arm)* 913b⟩

⟨*function* newusbid*(arm)* 914a⟩

⟨*function* newdev*(arm)* 914b⟩

⟨*function* newdevep*(arm)* 915a⟩

⟨*function* epdataperm*(arm)* 915b⟩

⟨*function* `usbgen`*(arm)* 916⟩

⟨*function* `hciprobe`*(arm)* 918a⟩

⟨*function* `usbreset`*(arm)* 918b⟩

⟨*function* `usbinit`*(arm)* 919a⟩

⟨*function* `usbattach`*(arm)* 919b⟩

⟨*function* `usbwalk`*(arm)* 919c⟩

⟨*function* `usbstat`*(arm)* 920a⟩

⟨*function* `usbload`*(arm)* 920b⟩

⟨*function* `usbopen`*(arm)* 920c⟩

⟨*function* `epclose`*(arm)* 921⟩

⟨*function* `usbclose`*(arm)* 922a⟩

⟨*function* `ctlread`*(arm)* 922b⟩

⟨*function* `rhubread`*(arm)* 923⟩

⟨*function* `rhubwrite`*(arm)* 924a⟩

⟨*function* `usbread`*(arm)* 924b⟩

⟨*function* `pow2`*(arm)* 925a⟩

⟨*function* `setmaxpkt`*(arm)* 925b⟩

⟨*function* `epctl`*(arm)* 926⟩

⟨*function* `usbctl`*(arm)* 929⟩

⟨*function* `ctlwrite`*(arm)* 930⟩

⟨*function* `usbwrite`*(arm)* 931a⟩

⟨*function* `usbbread`*(arm)* 931b⟩

⟨*function* `usbbwrite`*(arm)* 932a⟩

⟨*function* `usbshutdown`*(arm)* 932b⟩

⟨*global* `usbdevtab`*(arm)* 932c⟩

Uses `Hcitype` 907a.

buses/arm/usbdwc.c

⟨*enum* `_anon_(`(buses/arm/usbdwc.c)(arm)`)` 935⟩≡ (953c)

```
enum
{
    Enabledelay = 50,
    Resetdelay  = 10,
```

```
    ResetdelayHS    = 50,

    Read        = 0,
    Write       = 1,
};
```

⟨*struct* Ctlr*((buses/arm/usbdwc.c)(arm))* 936a⟩≡ (953c)
```
struct Ctlr {
    Lock;
    Dwcregs *regs;      /* controller registers */
    int nchan;      /* number of host channels */
    ulong   chanbusy;   /* bitmap of in-use channels */
    QLock   chanlock;   /* serialise access to chanbusy */
    QLock   split;      /* serialise split transactions */
    int splitretry; /* count retries of Nyet */
    int sofchan;    /* bitmap of channels waiting for sof */
    int wakechan;   /* bitmap of channels to wakeup after fiq */
    int debugchan;  /* bitmap of channels for interrupt debug */
    Rendez  *chanintr;  /* sleep till interrupt on channel N */
};
```

⟨*struct* Epio*(arm)* 936b⟩≡ (953c)
```
struct Epio {
    QLock;
    Block   *cb;
    ulong   lastpoll;
};
```

⟨*global* dwc*(arm)* 936c⟩≡ (953c)
```
static Ctlr dwc;
```

⟨*global* debug*((buses/arm/usbdwc.c)(arm))* 936d⟩≡ (953c)
```
static int debug;
```

⟨*global* Ebadlen*(arm)* 936e⟩≡ (953c)
```
static char Ebadlen[] = "bad usb request length";
```
Uses Ebadlen-2237 936e.

⟨*function* filock*(arm)* 936f⟩≡ (953c)
```
static void
filock(Lock *l)
{
    int x;

    x = splfhi();
    ilock(l);
    l->sr = x;
}
```
Uses ilock() 79d.

⟨*function* fiunlock*(arm)* 936g⟩≡ (953c)
```
static void
fiunlock(Lock *l)
{
    iunlock(l);
}
```
Uses iunlock() 80b.

⟨*function* chanalloc*(arm)* 937a⟩≡ (953c)

```
static Hostchan*
chanalloc(Ep *ep)
{
    Ctlr *ctlr;
    int bitmap, i;

    ctlr = ep->hp->aux;
    qlock(&ctlr->chanlock);
    bitmap = ctlr->chanbusy;
    for(i = 0; i < ctlr->nchan; i++)
        if((bitmap & (1<<i)) == 0){
            ctlr->chanbusy = bitmap | 1<<i;
            qunlock(&ctlr->chanlock);
            return &ctlr->regs->hchan[i];
        }
    qunlock(&ctlr->chanlock);
    panic("miller is a lazy git");
    return nil;
}
```

Uses qlock() 81f and qunlock() 82c.

⟨*function* chanrelease*(arm)* 937b⟩≡ (953c)

```
static void
chanrelease(Ep *ep, Hostchan *chan)
{
    Ctlr *ctlr;
    int i;

    ctlr = ep->hp->aux;
    i = chan - ctlr->regs->hchan;
    qlock(&ctlr->chanlock);
    ctlr->chanbusy &= ~(1<<i);
    qunlock(&ctlr->chanlock);
}
```

Uses qlock() 81f and qunlock() 82c.

⟨*function* chansetup*(arm)* 937c⟩≡ (953c)

```
static void
chansetup(Hostchan *hc, Ep *ep)
{
    int hcc;
    Ctlr *ctlr = ep->hp->aux;

    if(ep->debug)
        ctlr->debugchan |= 1 << (hc - ctlr->regs->hchan);
    else
        ctlr->debugchan &= ~(1 << (hc - ctlr->regs->hchan));
    switch(ep->dev->state){
    case Dconfig:
    case Dreset:
        hcc = 0;
        break;
    default:
        hcc = ep->dev->nb<<ODevaddr;
        break;
    }
    hcc |= ep->maxpkt | 1<<OMulticnt | ep->nb<<OEpnum;
    switch(ep->ttype){
    case Tctl:
```

```
            hcc |= Epctl;
            break;
        case Tiso:
            hcc |= Episo;
            break;
        case Tbulk:
            hcc |= Epbulk;
            break;
        case Tintr:
            hcc |= Epintr;
            break;
        }
        switch(ep->dev->speed){
        case Lowspeed:
            hcc |= Lspddev;
            /* fall through */
        case Fullspeed:
            if(ep->dev->hub > 1){
                hc->hcsplt = Spltena | POS_ALL | ep->dev->hub<<OHubaddr |
                    ep->dev->port;
                break;
            }
            /* fall through */
        default:
            hc->hcsplt = 0;
            break;
        }
        hc->hcchar = hcc;
        hc->hcint = ~0;
    }
```

⟨*function* `sofdone` *(arm)* 938a⟩≡ (953c)

```
    static int
    sofdone(void *a)
    {
        Dwcregs *r;

        r = a;
        return (r->gintmsk & Sofintr) == 0;
    }
```

⟨*function* `sofwait` *(arm)* 938b⟩≡ (953c)

```
    static void
    sofwait(Ctlr *ctlr, int n)
    {
        Dwcregs *r;

        r = ctlr->regs;
        do{
            filock(ctlr);
            r->gintsts = Sofintr;
            ctlr->sofchan |= 1<<n;
            r->gintmsk |= Sofintr;
            fiunlock(ctlr);
            sleep(&ctlr->chanintr[n], sofdone, r);
        }while((r->hfnum & 7) == 6);
    }
```

Uses `filock()` 936f, `fiunlock()` 936g, `sleep`, and `sofdone()` 938a.

⟨*function* chandone*(arm)* 939a⟩≡ (953c)

```
static int
chandone(void *a)
{
    Hostchan *hc;

    hc = a;
    if(hc->hcint == (Chhltd|Ack))
        return 0;
    return (hc->hcint & hc->hcintmsk) != 0;
}
```

⟨*function* chanwait*(arm)* 939b⟩≡ (953c)

```
static int
chanwait(Ep *ep, Ctlr *ctlr, Hostchan *hc, int mask)
{
    int intr, n, ointr;
    ulong start, now;
    Dwcregs *r;

    r = ctlr->regs;
    n = hc - r->hchan;
    for(;;){
restart:
        filock(ctlr);
        r->haintmsk |= 1<<n;
        hc->hcintmsk = mask;
        fiunlock(ctlr);
        sleep(&ctlr->chanintr[n], chandone, hc);
        hc->hcintmsk = 0;
        intr = hc->hcint;
        if(intr & Chhltd)
            return intr;
        start = arch_fastticks(0);
        ointr = intr;
        now = start;
        do{
            intr = hc->hcint;
            if(intr & Chhltd){
                if((ointr != Ack && ointr != (Ack|Xfercomp)) ||
                   intr != (Ack|Chhltd|Xfercomp) ||
                   (now - start) > 60)
                    dprint("await %x after %ld %x -> %x\n",
                        mask, now - start, ointr, intr);
                return intr;
            }
            if((intr & mask) == 0){
                dprint("ep%d.%d await %x intr %x -> %x\n",
                    ep->dev->nb, ep->nb, mask, ointr, intr);
                goto restart;
            }
            now = arch_fastticks(0);
        }while(now - start < 100);
        dprint("ep%d.%d halting channel %8.8ux hcchar %8.8ux "
            "grxstsr %8.8ux gnptxsts %8.8ux hptxsts %8.8ux\n",
            ep->dev->nb, ep->nb, intr, hc->hcchar, r->grxstsr,
            r->gnptxsts, r->hptxsts);
        mask = Chhltd;
        hc->hcchar |= Chdis;
        start = cpu->ticks;
```

```
        while(hc->hcchar & Chen){
            if(cpu->ticks - start >= 100){
                print("ep%d.%d channel won't halt hcchar %8.8ux\n",
                    ep->dev->nb, ep->nb, hc->hcchar);
                break;
            }
        }
        logdump(ep);
    }
}
```

Uses arch_fastticks, chandone() 939a, dprint, filock() 936f, fiunlock() 936g, logdump() 941b, print, and sleep.

⟨*function* chanintr*(arm)* 940a⟩≡ (953c)

```
static int
chanintr(Ctlr *ctlr, int n)
{
    Hostchan *hc;
    int i;

    hc = &ctlr->regs->hchan[n];
    if(ctlr->debugchan & (1<<n))
        clog(nil, hc);
    if((hc->hcsplt & Spltena) == 0)
        return 0;
    i = hc->hcint;
    if(i == (Chhltd|Ack)){
        hc->hcsplt |= Compsplt;
        ctlr->splitretry = 0;
    }else if(i == (Chhltd|Nyet)){
        if(++ctlr->splitretry >= 3)
            return 0;
    }else
        return 0;
    if(hc->hcchar & Chen){
        iprint("hcchar %8.8ux hcint %8.8ux", hc->hcchar, hc->hcint);
        hc->hcchar |= Chen | Chdis;
        while(hc->hcchar&Chen)
            ;
        iprint(" %8.8ux\n", hc->hcint);
    }
    hc->hcint = i;
    if(ctlr->regs->hfnum & 1)
        hc->hcchar &= ~Oddfrm;
    else
        hc->hcchar |= Oddfrm;
    hc->hcchar = (hc->hcchar &~ Chdis) | Chen;
    return 1;
}
```

Uses clog() 941a and iprint.

⟨*global* chanlog*(arm)* 940b⟩≡ (953c)

```
static Reg chanlog[32][5];
```

⟨*global* nchanlog*(arm)* 940c⟩≡ (953c)

```
static int nchanlog;
```

⟨*function* logstart*(arm)* 940d⟩≡ (953c)

```
static void
logstart(Ep *ep)
{
```

```
    if(ep->debug)
        nchanlog = 0;
}
```

Uses nchanlog-2239 940c.

⟨*function* clog*(arm)* 941a⟩≡ (953c)

```
static void
clog(Ep *ep, Hostchan *hc)
{
    Reg *p;

    if(ep != nil && !ep->debug)
        return;
    if(nchanlog == 32)
        nchanlog--;
    p = chanlog[nchanlog];
    p[0] = dwc.regs->hfnum;
    p[1] = hc->hcchar;
    p[2] = hc->hcint;
    p[3] = hc->hctsiz;
    p[4] = hc->hcdma;
    nchanlog++;
}
```

Uses chanlog-2238 940b, dwc-2235 936c, and nchanlog-2239 940c.

⟨*function* logdump*(arm)* 941b⟩≡ (953c)

```
static void
logdump(Ep *ep)
{
    Reg *p;
    int i;

    if(!ep->debug)
        return;
    p = chanlog[0];
    for(i = 0; i < nchanlog; i++){
        print("%5.5d.%5.5d %8.8ux %8.8ux %8.8ux %8.8ux\n",
            p[0]&0xFFFF, p[0]>>16, p[1], p[2], p[3], p[4]);
        p += 5;
    }
    nchanlog = 0;
}
```

Uses chanlog-2238 940b, nchanlog-2239 940c, and print.

⟨*function* chanio*(arm)* 941c⟩≡ (953c)

```
static int
chanio(Ep *ep, Hostchan *hc, int dir, int pid, void *a, int len)
{
    Ctlr *ctlr;
    int nleft, n, nt, i, maxpkt, npkt;
    uint hcdma, hctsiz;

    ctlr = ep->hp->aux;
    maxpkt = ep->maxpkt;
    npkt = HOWMANY(len, ep->maxpkt);
    if(npkt == 0)
        npkt = 1;

    hc->hcchar = (hc->hcchar & ~Epdir) | dir;
```

```
if(dir == Epin)
    n = ROUND(len, ep->maxpkt);
else
    n = len;
hc->hctsiz = n | npkt<<OPktcnt | pid;
hc->hcdma  = dmaaddr(a);

nleft = len;
logstart(ep);
for(;;){
    hcdma = hc->hcdma;
    hctsiz = hc->hctsiz;
    hc->hctsiz = hctsiz & ~Dopng;
    if(hc->hcchar&Chen){
        dprint("ep%d.%d before chanio hcchar=%8.8ux\n",
            ep->dev->nb, ep->nb, hc->hcchar);
        hc->hcchar |= Chen | Chdis;
        while(hc->hcchar&Chen)
            ;
        hc->hcint = Chhltd;
    }
    if((i = hc->hcint) != 0){
        dprint("ep%d.%d before chanio hcint=%8.8ux\n",
            ep->dev->nb, ep->nb, i);
        hc->hcint = i;
    }
    if(hc->hcsplt & Spltena){
        qlock(&ctlr->split);
        sofwait(ctlr, hc - ctlr->regs->hchan);
        if((dwc.regs->hfnum & 1) == 0)
            hc->hcchar &= ~Oddfrm;
        else
            hc->hcchar |= Oddfrm;
    }
    hc->hcchar = (hc->hcchar &~ Chdis) | Chen;
    clog(ep, hc);
    if(ep->ttype == Tbulk && dir == Epin)
        i = chanwait(ep, ctlr, hc, /* Ack| */ Chhltd);
    else if(ep->ttype == Tintr && (hc->hcsplt & Spltena))
        i = chanwait(ep, ctlr, hc, Chhltd);
    else
        i = chanwait(ep, ctlr, hc, Chhltd|Nak);
    clog(ep, hc);
    hc->hcint = i;

    if(hc->hcsplt & Spltena){
        hc->hcsplt &= ~Compsplt;
        qunlock(&ctlr->split);
    }

    if((i & Xfercomp) == 0 && i != (Chhltd|Ack) && i != Chhltd){
        if(i & Stall)
            error(Estalled);
        if(i & (Nyet|Frmovrun))
            continue;
        if(i & Nak){
            if(ep->ttype == Tintr)
                tsleep(&up->sleepr, returnfalse, 0, ep->pollival);
            else
                tsleep(&up->sleepr, returnfalse, 0, 1);
```

```
                continue;
            }
            logdump(ep);
            print("usbotg: ep%d.%d error intr %8.8ux\n",
                ep->dev->nb, ep->nb, i);
            if(i & ~(Chhltd|Ack))
                error(Eio);
            if(hc->hcdma != hcdma)
                print("usbotg: weird hcdma %x->%x intr %x->%x\n",
                    hcdma, hc->hcdma, i, hc->hcint);
        }
        n = hc->hcdma - hcdma;
        if(n == 0){
            if((hc->hctsiz & Pktcnt) != (hctsiz & Pktcnt))
                break;
            else
                continue;
        }
        if(dir == Epin && ep->ttype == Tbulk && n == nleft){
            nt = (hctsiz & Xfersize) - (hc->hctsiz & Xfersize);
            if(nt != n){
                if(n == ROUND(nt, 4))
                    n = nt;
                else
                    print("usbotg: intr %8.8ux "
                        "dma %8.8ux-%8.8ux "
                        "hctsiz %8.8ux-%8.ux\n",
                        i, hcdma, hc->hcdma, hctsiz,
                        hc->hctsiz);
            }
        }
        if(n > nleft){
            if(n != ROUND(nleft, 4))
                dprint("too much: wanted %d got %d\n",
                    len, len - nleft + n);
            n = nleft;
        }
        nleft -= n;
        if(nleft == 0 || (n % maxpkt) != 0)
            break;
        if((i & Xfercomp) && ep->ttype != Tctl)
            break;
        if(dir == Epout)
            dprint("too little: nleft %d hcdma %x->%x hctsiz %x->%x intr %x\n",
                nleft, hcdma, hc->hcdma, hctsiz, hc->hctsiz, i);
    }
    logdump(ep);
    return len - nleft;
}
```

Uses Estalled 664d, HOWMANY 47f, ROUND 47b, chanwait() 939b, clog() 941a, dmaaddr() 957h, dprint, dwc-2235 936c, logdump() 941b, logstart() 940d, print, qlock() 81f, qunlock() 82c, returnfalse() 104c, sofwait() 938b, tsleep, and up.

⟨*function* multitrans*(arm)* 943⟩≡ (953c)

```
static long
multitrans(Ep *ep, Hostchan *hc, int rw, void *a, long n)
{
    long sofar, m;

    sofar = 0;
    do{
```

```
        m = n - sofar;
        if(m > ep->maxpkt)
            m = ep->maxpkt;
        m = chanio(ep, hc, rw == Read? Epin : Epout, ep->toggle[rw],
            (char*)a + sofar, m);
        ep->toggle[rw] = hc->hctsiz & Pid;
        sofar += m;
    }while(sofar < n && m == ep->maxpkt);
    return sofar;
}
```

Uses Read-2233 935 and chanio() 941c.

⟨*function* eptrans*(arm)* 944a⟩≡ (953c)

```
static long
eptrans(Ep *ep, int rw, void *a, long n)
{
    Hostchan *hc;

    if(ep->clrhalt){
        ep->clrhalt = 0;
        if(ep->mode != OREAD)
            ep->toggle[Write] = DATA0;
        if(ep->mode != OWRITE)
            ep->toggle[Read] = DATA0;
    }
    hc = chanalloc(ep);
    if(waserror()){
        ep->toggle[rw] = hc->hctsiz & Pid;
        chanrelease(ep, hc);
        if(strcmp(up->errstr, Estalled) == 0)
            return 0;
        nexterror();
    }
    chansetup(hc, ep);
    if(rw == Read && ep->ttype == Tbulk)
        n = multitrans(ep, hc, rw, a, n);
    else{
        n = chanio(ep, hc, rw == Read? Epin : Epout, ep->toggle[rw],
            a, n);
        ep->toggle[rw] = hc->hctsiz & Pid;
    }
    chanrelease(ep, hc);
    poperror();
    return n;
}
```

Uses Estalled 664d, OREAD 853i, OWRITE, Read-2233 935, Write-2234 935, chanio() 941c, chanrelease() 937b, chansetup() 937c, multitrans() 943, nexterror, poperror 666d, up, and waserror 666d.

⟨*function* ctltrans*(arm)* 944b⟩≡ (953c)

```
static long
ctltrans(Ep *ep, uchar *req, long n)
{
    Hostchan *hc;
    Epio *epio;
    Block *b;
    uchar *data;
    int datalen;

    epio = ep->aux;
    if(epio->cb != nil){
```

```
        freeb(epio->cb);
        epio->cb = nil;
    }
    if(n < Rsetuplen)
        error(Ebadlen);
    if(req[Rtype] & Rd2h){
        datalen = GET2(req+Rcount);
        if(datalen <= 0 || datalen > Maxctllen)
            error(Ebadlen);
        /* XXX cache madness */
        epio->cb = b = allocb(ROUND(datalen, ep->maxpkt));
        assert(((uintptr)b->wp & (BLOCKALIGN-1)) == 0);
        memset(b->wp, 0x55, b->lim - b->wp);
        cachedwbinvse(b->wp, b->lim - b->wp);
        data = b->wp;
    }else{
        b = nil;
        datalen = n - Rsetuplen;
        data = req + Rsetuplen;
    }
    hc = chanalloc(ep);
    if(waserror()){
        chanrelease(ep, hc);
        if(strcmp(up->errstr, Estalled) == 0)
            return 0;
        nexterror();
    }
    chansetup(hc, ep);
    chanio(ep, hc, Epout, SETUP, req, Rsetuplen);
    if(req[Rtype] & Rd2h){
        if(ep->dev->hub <= 1){
            ep->toggle[Read] = DATA1;
            b->wp += multitrans(ep, hc, Read, data, datalen);
        }else
            b->wp += chanio(ep, hc, Epin, DATA1, data, datalen);
        chanio(ep, hc, Epout, DATA1, nil, 0);
        n = Rsetuplen;
    }else{
        if(datalen > 0)
            chanio(ep, hc, Epout, DATA1, data, datalen);
        chanio(ep, hc, Epin, DATA1, nil, 0);
        n = Rsetuplen + datalen;
    }
    chanrelease(ep, hc);
    poperror();
    return n;
}
```

Uses Ebadlen-2237 936e, Estalled 664d, GET2, ROUND 47b, Rd2h, Read-2233 935, allocb() 436b, assert 853f, chanio() 941c, chanrelease() 937b, chansetup() 937c, freeb() 438a, multitrans() 943, nexterror, poperror 666d, up, and waserror 666d.

⟨*function* ctldata*(arm)* 945⟩≡ (953c)

```
static long
ctldata(Ep *ep, void *a, long n)
{
    Epio *epio;
    Block *b;

    epio = ep->aux;
    b = epio->cb;
    if(b == nil)
```

```
        return 0;
    if(n > BLEN(b))
        n = BLEN(b);
    memmove(a, b->rp, n);
    b->rp += n;
    if(BLEN(b) == 0){
        freeb(b);
        epio->cb = nil;
    }
    return n;
}
```

Uses BLEN 435b and freeb() 438a.

⟨*function* greset*(arm)* 946a⟩≡ (953c)

```
static void
greset(Dwcregs *r, int bits)
{
    r->grstctl |= bits;
    while(r->grstctl & bits)
        ;
    arch_microdelay(10);
}
```

Uses arch_microdelay.

⟨*function* init*(arm)* 946b⟩≡ (953c)

```
static void
init(Hci *hp)
{
    Ctlr *ctlr;
    Dwcregs *r;
    uint n, rx, tx, ptx;

    ctlr = hp->aux;
    r = ctlr->regs;

    ctlr->nchan = 1 + ((r->ghwcfg2 & Num_host_chan) >> ONum_host_chan);
    ctlr->chanintr = malloc(ctlr->nchan * sizeof(Rendez));

    r->gahbcfg = 0;
    setpower(PowerUsb, 1);

    while((r->grstctl&Ahbidle) == 0)
        ;
    greset(r, Csftrst);

    r->gusbcfg |= Force_host_mode;
    tsleep(&up->sleepr, returnfalse, 0, 25);
    r->gahbcfg |= Dmaenable;

    n = (r->ghwcfg3 & Dfifo_depth) >> ODfifo_depth;
    rx = 0x306;
    tx = 0x100;
    ptx = 0x200;
    r->grxfsiz = rx;
    r->gnptxfsiz = rx | tx<<ODepth;
    tsleep(&up->sleepr, returnfalse, 0, 1);
    r->hptxfsiz = (rx + tx) | ptx << ODepth;
    greset(r, Rxfflsh);
    r->grstctl = TXF_ALL;
    greset(r, Txfflsh);
```

```
    dprint("usbotg: FIFO depth %d sizes rx/nptx/ptx %8.8ux %8.8ux %8.8ux\n",
        n, r->grxfsiz, r->gnptxfsiz, r->hptxfsiz);

    r->hport0 = Prtpwr|Prtconndet|Prtenchng|Prtovrcurrchng;
    r->gintsts = ~0;
    r->gintmsk = Hcintr;
    r->gahbcfg |= Glblintrmsk;
}
```

Uses PowerUsb 562c, dprint, greset() 946a, malloc() 103c, returnfalse() 104c, setpower() 563a, tsleep, and up.

⟨*function* dump*(*(buses/arm/usbdwc.c)(arm)*)* 947a⟩≡ (953c)

```
static void
dump(Hci*)
{
}
```

⟨*function* fiqintr*(arm)* 947b⟩≡ (953c)

```
static void
fiqintr(Ureg*, void *a)
{
    Hci *hp;
    Ctlr *ctlr;
    Dwcregs *r;
    uint intr, haint, wakechan;
    int i;

    hp = a;
    ctlr = hp->aux;
    r = ctlr->regs;
    wakechan = 0;
    filock(ctlr);
    intr = r->gintsts;
    if(intr & Hcintr){
        haint = r->haint & r->haintmsk;
        for(i = 0; haint; i++){
            if(haint & 1){
                if(chanintr(ctlr, i) == 0){
                    r->haintmsk &= ~(1<<i);
                    wakechan |= 1<<i;
                }
            }
            haint >>= 1;
        }
    }
    if(intr & Sofintr){
        r->gintsts = Sofintr;
        if((r->hfnum&7) != 6){
            r->gintmsk &= ~Sofintr;
            wakechan |= ctlr->sofchan;
            ctlr->sofchan = 0;
        }
    }
    if(wakechan){
        ctlr->wakechan |= wakechan;
        armtimerset(1);
    }
    fiunlock(ctlr);
}
```

Uses armtimerset() 948a, chanintr() 940a, filock() 936f, and fiunlock() 936g.

⟨*function* armtimerset*(arm)* 948a⟩≡ (1011b)

```
void
armtimerset(int n)
{
    Armtimer *tm;

    tm = (Armtimer*)ARMTIMER;
    if(n > 0){
        tm->ctl |= TmrEnable|TmrIntEnable;
        tm->load = n;
    }else{
        tm->load = 0;
        tm->ctl &= ~(TmrEnable|TmrIntEnable);
        tm->irq = 1;
    }
}
```

Uses ARMTIMER-2648 43c, TmrEnable-2659 1011a, and TmrIntEnable-2660 1011a.

⟨*function* irqintr*(arm)* 948b⟩≡ (953c)

```
static void
irqintr(Ureg*, void *a)
{
    Ctlr *ctlr;
    uint wakechan;
    int i;

    ctlr = a;
    filock(ctlr);
    armtimerset(0);
    wakechan = ctlr->wakechan;
    ctlr->wakechan = 0;
    fiunlock(ctlr);
    for(i = 0; wakechan; i++){
        if(wakechan & 1)
            wakeup(&ctlr->chanintr[i]);
        wakechan >>= 1;
    }
}
```

Uses armtimerset() 948a, filock() 936f, fiunlock() 936g, and wakeup.

⟨*function* epopen*(arm)* 948c⟩≡ (953c)

```
static void
epopen(Ep *ep)
{
    ddprint("usbotg: epopen ep%d.%d ttype %d\n",
        ep->dev->nb, ep->nb, ep->ttype);
    switch(ep->ttype){
    case Tnone:
        error(Enotconf);
    case Tintr:
        assert(ep->pollival > 0);
        /* fall through */
    case Tbulk:
        if(ep->toggle[Read] == 0)
            ep->toggle[Read] = DATA0;
        if(ep->toggle[Write] == 0)
            ep->toggle[Write] = DATA0;
        break;
    }
    ep->aux = malloc(sizeof(Epio));
```

```
    if(ep->aux == nil)
        error(Enomem);
}
```

Uses Enotconf 664d, Read-2233 935, Write-2234 935, assert 853f, ddprint, and malloc() 103c.

⟨*function* epclose*((buses/arm/usbdwc.c)(arm))* 949a⟩≡ (953c)

```
static void
epclose(Ep *ep)
{
    ddprint("usbotg: epclose ep%d.%d ttype %d\n",
        ep->dev->nb, ep->nb, ep->ttype);
    switch(ep->ttype){
    case Tctl:
        freeb(((Epio*)ep->aux)->cb);
        /* fall through */
    default:
        free(ep->aux);
        break;
    }
}
```

Uses ddprint, free() 104a, and freeb() 438a.

⟨*function* epread*(arm)* 949b⟩≡ (953c)

```
static long
epread(Ep *ep, void *a, long n)
{
    Epio *epio;
    Block *b;
    uchar *p;
    ulong elapsed;
    long nr;

    ddprint("epread ep%d.%d %ld\n", ep->dev->nb, ep->nb, n);
    epio = ep->aux;
    b = nil;
    qlock(epio);
    if(waserror()){
        qunlock(epio);
        if(b)
            freeb(b);
        nexterror();
    }
    switch(ep->ttype){
    default:
        error(Egreg);
    case Tctl:
        nr = ctldata(ep, a, n);
        qunlock(epio);
        poperror();
        return nr;
    case Tintr:
        elapsed = TK2MS(cpu->ticks) - epio->lastpoll;
        if(elapsed < ep->pollival)
            tsleep(&up->sleepr, returnfalse, 0, ep->pollival - elapsed);
        /* fall through */
    case Tbulk:
        /* XXX cache madness */
        b = allocb(ROUND(n, ep->maxpkt));
        p = b->rp;
        assert(((uintptr)p & (BLOCKALIGN-1)) == 0);
```

```
        cachedinvse(p, n);
        nr = eptrans(ep, Read, p, n);
        cachedinvse(p, nr);
        epio->lastpoll = TK2MS(cpu->ticks);
        memmove(a, p, nr);
        qunlock(epio);
        freeb(b);
        poperror();
        return nr;
    }
}
```

Uses Egreg 664d, ROUND 47b, Read-2233 935, TK2MS 244d, allocb() 436b, assert 853f, ctldata() 945, ddprint, eptrans() 944a, freeb() 438a, nexterror, poperror 666d, qlock() 81f, qunlock() 82c, returnfalse() 104c, tsleep, up, and waserror 666d.

⟨*function* epwrite*(arm)* 950⟩≡ (953c)

```
static long
epwrite(Ep *ep, void *a, long n)
{
    Epio *epio;
    Block *b;
    uchar *p;
    ulong elapsed;

    ddprint("epwrite ep%d.%d %ld\n", ep->dev->nb, ep->nb, n);
    epio = ep->aux;
    b = nil;
    qlock(epio);
    if(waserror()){
        qunlock(epio);
        if(b)
            freeb(b);
        nexterror();
    }
    switch(ep->ttype){
    default:
        error(Egreg);
    case Tintr:
        elapsed = TK2MS(cpu->ticks) - epio->lastpoll;
        if(elapsed < ep->pollival)
            tsleep(&up->sleepr, returnfalse, 0, ep->pollival - elapsed);
        /* fall through */
    case Tctl:
    case Tbulk:
        /* XXX cache madness */
        b = allocb(n);
        p = b->wp;
        assert(((uintptr)p & (BLOCKALIGN-1)) == 0);
        memmove(p, a, n);
        cachedwbse(p, n);
        if(ep->ttype == Tctl)
            n = ctltrans(ep, p, n);
        else{
            n = eptrans(ep, Write, p, n);
            epio->lastpoll = TK2MS(cpu->ticks);
        }
        qunlock(epio);
        freeb(b);
        poperror();
        return n;
    }
```

```
}
```

Uses Egreg 664d, TK2MS 244d, Write-2234 935, allocb() 436b, assert 853f, ctltrans() 944b, ddprint, eptrans() 944a, freeb() 438a, nexterror, poperror 666d, qlock() 81f, qunlock() 82c, returnfalse() 104c, tsleep, up, and waserror 666d.

⟨*function* seprintep*((buses/arm/usbdwc.c)(arm))* 951a⟩≡ (953c)

```
static char*
seprintep(char *s, char*, Ep*)
{
    return s;
}
```

⟨*function* portenable*(arm)* 951b⟩≡ (953c)

```
static int
portenable(Hci *hp, int port, int on)
{
    Ctlr *ctlr;
    Dwcregs *r;

    assert(port == 1);
    ctlr = hp->aux;
    r = ctlr->regs;
    dprint("usbotg enable=%d; sts %#x\n", on, r->hport0);
    if(!on)
        r->hport0 = Prtpwr | Prtena;
    tsleep(&up->sleepr, returnfalse, 0, Enabledelay);
    dprint("usbotg enable=%d; sts %#x\n", on, r->hport0);
    return 0;
}
```

Uses Enabledelay-2230 935, assert 853f, dprint, returnfalse() 104c, tsleep, and up.

⟨*function* portreset*(arm)* 951c⟩≡ (953c)

```
static int
portreset(Hci *hp, int port, int on)
{
    Ctlr *ctlr;
    Dwcregs *r;
    int b, s;

    assert(port == 1);
    ctlr = hp->aux;
    r = ctlr->regs;
    dprint("usbotg reset=%d; sts %#x\n", on, r->hport0);
    if(!on)
        return 0;
    r->hport0 = Prtpwr | Prtrst;
    tsleep(&up->sleepr, returnfalse, 0, ResetdelayHS);
    r->hport0 = Prtpwr;
    tsleep(&up->sleepr, returnfalse, 0, Enabledelay);
    s = r->hport0;
    b = s & (Prtconndet|Prtenchng|Prtovrcurrchng);
    if(b != 0)
        r->hport0 = Prtpwr | b;
    dprint("usbotg reset=%d; sts %#x\n", on, s);
    if((s & Prtena) == 0)
        print("usbotg: host port not enabled after reset");
    return 0;
}
```

Uses Enabledelay-2230 935, ResetdelayHS-2232 935, assert 853f, dprint, print, returnfalse() 104c, tsleep, and up.

⟨*function* portstatus*(arm)* 952a⟩≡ (953c)

```
static int
portstatus(Hci *hp, int port)
{
    Ctlr *ctlr;
    Dwcregs *r;
    int b, s;

    assert(port == 1);
    ctlr = hp->aux;
    r = ctlr->regs;
    s = r->hport0;
    b = s & (Prtconndet|Prtenchng|Prtovrcurrchng);
    if(b != 0)
        r->hport0 = Prtpwr | b;
    b = 0;
    if(s & Prtconnsts)
        b |= HPpresent;
    if(s & Prtconndet)
        b |= HPstatuschg;
    if(s & Prtena)
        b |= HPenable;
    if(s & Prtenchng)
        b |= HPchange;
    if(s & Prtovrcurract)
         b |= HPovercurrent;
    if(s & Prtsusp)
        b |= HPsuspend;
    if(s & Prtrst)
        b |= HPreset;
    if(s & Prtpwr)
        b |= HPpower;
    switch(s & Prtspd){
    case HIGHSPEED:
        b |= HPhigh;
        break;
    case LOWSPEED:
        b |= HPslow;
        break;
    }
    return b;
}
```

Uses assert 853f.

⟨*function* shutdown*(arm)* 952b⟩≡ (953c)

```
static void
shutdown(Hci*)
{
}
```

⟨*function* setdebug*(arm)* 952c⟩≡ (953c)

```
static void
setdebug(Hci*, int d)
{
    debug = d;
}
```

Uses debug-2236 936d.

⟨*function* reset*(arm)* 953a⟩≡ (953c)

```
static int
reset(Hci *hp)
{
    Ctlr *ctlr;
    uint id;

    ctlr = &dwc;
    if(ctlr->regs != nil)
        return -1;
    ctlr->regs = (Dwcregs*)USBREGS;
    id = ctlr->regs->gsnpsid;
    if((id>>16) != ('O'<<8 | 'T'))
        return -1;
    dprint("usbotg: rev %d.%3.3x\n", (id>>12)&0xF, id&0xFFF);

    arch_intrenable(IRQtimerArm, irqintr, ctlr, 0, "dwc");

    hp->aux = ctlr;
    hp->port = 0;
    hp->irq = IRQusb;
    hp->tbdf = 0;
    hp->nports = 1;
    hp->highspeed = 1;

    hp->init = init;
    hp->dump = dump;
    hp->interrupt = fiqintr;
    hp->epopen = epopen;
    hp->epclose = epclose;
    hp->epread = epread;
    hp->epwrite = epwrite;
    hp->seprintep = seprintep;
    hp->portenable = portenable;
    hp->portreset = portreset;
    hp->portstatus = portstatus;
    hp->shutdown = shutdown;
    hp->debug = setdebug;
    hp->type = "dwcotg";
    return 0;
}
```

Uses IRQusb 174b, USBREGS-2229 44b, dprint, dwc-2235 936c, epopen() 948c, epread() 949b, epwrite() 950, fiqintr() 947b, portenable() 951b, and setdebug() 952c.

⟨*function* usbdwclink*(arm)* 953b⟩≡ (953c)

```
void
usbdwclink(void)
{
    addhcitype("dwcotg", reset);
}
```

Uses addhcitype() 911a and reset() 953a.

⟨buses/arm/usbdwc.c 953c⟩≡

```
/*
 * USB host driver for BCM2835
 *  Synopsis DesignWare Core USB 2.0 OTG controller
 *
 * Copyright © 2012 Richard Miller <r.miller@acm.org>
 *
 * This is work in progress:
```

```
 * - no isochronous pipes
 * - no bandwidth budgeting
 * - frame scheduling is crude
 * - error handling is overly optimistic
 * It should be just about adequate for a Plan 9 terminal with
 * keyboard, mouse, ethernet adapter, and an external flash drive.
 */
```
⟨*kernel basic includes* 49a⟩

```
#include    "io.h"
#include    "../port/usb.h"

#include "dwcotg.h"
```

⟨*constant* USBREGS*(arm)* 44b⟩
⟨*enum* _anon_*(*(buses/arm/usbdwc.c)(arm)*)* 935⟩

```
typedef struct Ctlr Ctlr;
typedef struct Epio Epio;
```

⟨*struct* Ctlr*(*(buses/arm/usbdwc.c)(arm)*)* 936a⟩

⟨*struct* Epio*(arm)* 936b⟩

⟨*global* dwc*(arm)* 936c⟩
⟨*global* debug*(*(buses/arm/usbdwc.c)(arm)*)* 936d⟩

⟨*global* Ebadlen*(arm)* 936e⟩

```
static void clog(Ep *ep, Hostchan *hc);
static void logdump(Ep *ep);
```

⟨*function* filock*(arm)* 936f⟩

⟨*function* fiunlock*(arm)* 936g⟩

⟨*function* chanalloc*(arm)* 937a⟩

⟨*function* chanrelease*(arm)* 937b⟩

⟨*function* chansetup*(arm)* 937c⟩

⟨*function* sofdone*(arm)* 938a⟩

⟨*function* sofwait*(arm)* 938b⟩

⟨*function* chandone*(arm)* 939a⟩

⟨*function* chanwait*(arm)* 939b⟩

⟨*function* chanintr*(arm)* 940a⟩

⟨*global* chanlog*(arm)* 940b⟩
⟨*global* nchanlog*(arm)* 940c⟩

⟨*function* logstart*(arm)* 940d⟩

⟨*function* clog*(arm)* 941a⟩

⟨*function* logdump*(arm)* 941b⟩

⟨*function* `chanio`*(arm)* 941c⟩

⟨*function* `multitrans`*(arm)* 943⟩

⟨*function* `eptrans`*(arm)* 944a⟩

⟨*function* `ctltrans`*(arm)* 944b⟩

⟨*function* `ctldata`*(arm)* 945⟩

⟨*function* `greset`*(arm)* 946a⟩

⟨*function* `init`*(arm)* 946b⟩

⟨*function* `dump`*((buses/arm/usbdwc.c)(arm))* 947a⟩

⟨*function* `fiqintr`*(arm)* 947b⟩

⟨*function* `irqintr`*(arm)* 948b⟩

⟨*function* `epopen`*(arm)* 948c⟩

⟨*function* `epclose`*((buses/arm/usbdwc.c)(arm))* 949a⟩

⟨*function* `epread`*(arm)* 949b⟩

⟨*function* `epwrite`*(arm)* 950⟩

⟨*function* `seprintep`*((buses/arm/usbdwc.c)(arm))* 951a⟩

⟨*function* `portenable`*(arm)* 951b⟩

⟨*function* `portreset`*(arm)* 951c⟩

⟨*function* `portstatus`*(arm)* 952a⟩

⟨*function* `shutdown`*(arm)* 952b⟩

⟨*function* `setdebug`*(arm)* 952c⟩

⟨*function* `reset`*(arm)* 953a⟩

⟨*function* `usbdwclink`*(arm)* 953b⟩

Uses `Epio` 936b.

buses/arm/dma.c

⟨*macro* `IRQDMA`*(arm)* 955a⟩≡ (890b)

```
#define IRQDMA(chan)    (IRQdma0+(chan))
```

⟨*enum* `Dma`*(arm)* 955b⟩≡ (890b)

```
enum DmaFlags {
    DmaD2M      = 0,        /* device to memory */
    DmaM2D      = 1,        /* memory to device */
    DmaM2M      = 2,        /* memory to memory */

    DmaChanEmmc = 4,        /* can only use 2-5, maybe 0 */
    DmaChanSpiTx= 2,
```

```
    DmaChanSpiRx= 0,

    DmaDevSpiTx = 6,
    DmaDevSpiRx = 7,
    DmaDevEmmc  = 11,
};
```

⟨*enum* _anon_*(*(buses/arm/dma.c)(arm)*)* 956⟩≡ (960)

```
enum {
    Nchan       = 7,        /* number of dma channels */
    Regsize     = 0x100,    /* size of regs for each chan */
    Cbalign     = 32,       /* control block byte alignment */
    Dbg     = 0,

    /* registers for each dma controller */
    Cs      = 0x00>>2,
    Conblkad    = 0x04>>2,
    Ti      = 0x08>>2,
    Sourcead    = 0x0c>>2,
    Destad      = 0x10>>2,
    Txfrlen     = 0x14>>2,
    Stride      = 0x18>>2,
    Nextconbk   = 0x1c>>2,
    Debug       = 0x20>>2,

    /* collective registers */
    Intstatus   = 0xfe0>>2,
    Enable      = 0xff0>>2,

    /* Cs */
    Reset       = 1<<31,
    Abort       = 1<<30,
    Error       = 1<<8,
    Waitwrite   = 1<<6,
    Waitdreq    = 1<<5,
    Paused      = 1<<4,
    Dreq        = 1<<3,
    Int     = 1<<2,
    End     = 1<<1,
    Active      = 1<<0,

    /* Ti */
    Permapshift= 16,
    Srcignore   = 1<<11,
    Srcdreq     = 1<<10,
    Srcwidth128 = 1<<9,
    Srcinc      = 1<<8,
    Destignore  = 1<<7,
    Destdreq    = 1<<6,
    Destwidth128    = 1<<5,
    Destinc     = 1<<4,
    Waitresp    = 1<<3,
    Tdmode      = 1<<1,
    Inten       = 1<<0,

    /* Debug */
    Lite        = 1<<28,
    Clrerrors   = 7<<0,
};
```

⟨*struct* `Ctlr` *(arm)* 957a⟩≡ (960)
```
struct Ctlr {
    u32int  *regs;
    Cb  *cb;
    Rendez  r;
    int dmadone;
};
```

⟨*struct* `Cb` *(arm)* 957b⟩≡ (960)
```
struct Cb {
    u32int  ti;
    u32int  sourcead;
    u32int  destad;
    u32int  txfrlen;
    u32int  stride;
    u32int  nextconbk;
    u32int  reserved[2];
};
```

⟨*global* `dma` *(arm)* 957c⟩≡ (960)
```
static Ctlr dma[Nchan];
```
Uses `Nchan-2285` 956.

⟨*global* `dmaregs` *(arm)* 957d⟩≡ (960)
```
static u32int *dmaregs = (u32int*)DMAREGS;
```
Uses `DMAREGS-2284` 43e and `dmaregs-2325` 957d.

⟨`Soc` *bus fields* 957e⟩≡ (72a) 957i▷
```
uintptr busdram;
```

⟨`soc(raspberry pi1)` *other fields(arm)* 957f⟩+≡ (72b) ◁536g 957j▷
```
.busdram    = 0x40000000,
```

⟨`soc(raspberry pi2)` *other fields(arm)* 957g⟩+≡ (534i) ◁536h 957k▷
```
.busdram    = 0xC0000000,
```

⟨*function* `dmaaddr` *(arm)* 957h⟩≡ (960)
```
uintptr
dmaaddr(void *va)
{
    return soc.busdram | (PTR2UINT(va) & ~KSEGM); // PADDR?
}
```
Uses `KSEGM` 56e and `PTR2UINT` 56g.

⟨`Soc` *bus fields* 957i⟩+≡ (72a) ◁957e
```
uintptr busio;
```

⟨`soc(raspberry pi1)` *other fields(arm)* 957j⟩+≡ (72b) ◁957f
```
.busio      = 0x7E000000,
```

⟨`soc(raspberry pi2)` *other fields(arm)* 957k⟩+≡ (534i) ◁957g
```
.busio      = 0x7E000000,
```

⟨*function* `dmaioaddr` *(arm)* 957l⟩≡ (960)
```
static uintptr
dmaioaddr(void *va)
{
    return soc.busio | (PTR2UINT(va) & ~VIRTIO);
}
```
Uses `PTR2UINT` 56g and `VIRTIO` 43a.

⟨*function* dump*(arm)* 958a⟩≡ (960)
```
static void
dump(char *msg, uchar *p, int n)
{
    print("%s", msg);
    while(n-- > 0)
        print(" %2.2x", *p++);
    print("\n");
}
```
Uses print.

⟨*function* dumpdregs*(arm)* 958b⟩≡ (960)
```
static void
dumpdregs(char *msg, u32int *r)
{
    int i;

    print("%s: %#p =", msg, r);
    for(i = 0; i < 9; i++)
        print(" %8.8uX", r[i]);
    print("\n");
}
```
Uses print.

⟨*function* dmadone*(arm)* 958c⟩≡ (960)
```
static int
dmadone(void *a)
{
    return ((Ctlr*)a)->dmadone;
}
```

⟨*function* dmainterrupt*(arm)* 958d⟩≡ (960)
```
static void
dmainterrupt(Ureg*, void *a)
{
    Ctlr *ctlr;

    ctlr = a;
    ctlr->regs[Cs] = Int;
    ctlr->dmadone = 1;
    wakeup(&ctlr->r);
}
```
Uses Cs-2289 956, Int-2307 956, and wakeup.

⟨*function* dmastart*(arm)* 958e⟩≡ (960)
```
void
dmastart(int chan, int dev, int dir, void *src, void *dst, int len)
{
    Ctlr *ctlr;
    Cb *cb;
    int ti;

    ctlr = &dma[chan];
    if(ctlr->regs == nil){
        ctlr->regs = (u32int*)(DMAREGS + chan*Regsize);
        ctlr->cb = xspanalloc(sizeof(Cb), Cbalign, 0);
        assert(ctlr->cb != nil);
        dmaregs[Enable] |= 1<<chan;
        ctlr->regs[Cs] = Reset;
```

```
        while(ctlr->regs[Cs] & Reset)
            ;
        arch_intrenable(IRQDMA(chan), dmainterrupt, ctlr, 0, "dma");
    }
    cb = ctlr->cb;
    ti = 0;
    switch(dir){
    case DmaD2M:
        cachedinvse(dst, len);
        ti = Srcdreq | Destinc;
        cb->sourcead = dmaioaddr(src);
        cb->destad = dmaaddr(dst);
        break;
    case DmaM2D:
        cachedwbse(src, len);
        ti = Destdreq | Srcinc;
        cb->sourcead = dmaaddr(src);
        cb->destad = dmaioaddr(dst);
        break;
    case DmaM2M:
        cachedwbse(src, len);
        cachedinvse(dst, len);
        ti = Srcinc | Destinc;
        cb->sourcead = dmaaddr(src);
        cb->destad = dmaaddr(dst);
        break;
    }
    cb->ti = ti | dev<<Permapshift | Inten;
    cb->txfrlen = len;
    cb->stride = 0;
    cb->nextconbk = 0;
    cachedwbse(cb, sizeof(Cb));
    ctlr->regs[Cs] = 0;
    arch_microdelay(1);
    ctlr->regs[Conblkad] = dmaaddr(cb);
    DBG print("dma start: %ux %ux %ux %ux %ux %ux\n",
        cb->ti, cb->sourcead, cb->destad, cb->txfrlen,
        cb->stride, cb->nextconbk);
    DBG print("intstatus %ux\n", dmaregs[Intstatus]);
    dmaregs[Intstatus] = 0;
    ctlr->regs[Cs] = Int;
    arch_microdelay(1);
    arch_coherence();
    DBG dumpdregs("before Active", ctlr->regs);
    ctlr->regs[Cs] = Active;
    DBG dumpdregs("after Active", ctlr->regs);
}
```

⟨*function* dmawait*(arm)* 959⟩≡ (960)

```
int
dmawait(int chan)
{
    Ctlr *ctlr;
    u32int *r;
    int s;

    ctlr = &dma[chan];
    tsleep(&ctlr->r, dmadone, ctlr, 3000);
    ctlr->dmadone = 0;
    r = ctlr->regs;
```

```
    DBG dumpdregs("after sleep", r);
    s = r[Cs];
    if((s & (Active|End|Error)) != End){
        print("dma chan %d %s Cs %ux Debug %ux\n", chan,
            (s&End)? "error" : "timeout", s, r[Debug]);
        r[Cs] = Reset;
        r[Debug] = Clrerrors;
        return -1;
    }
    r[Cs] = Int|End;
    return 0;
}
```

⟨buses/arm/dma.c 960⟩≡

```
/*
 * bcm2835 dma controller
 *
 * simplest to use only channels 0-6
 *  channels 7-14 have reduced functionality
 *  channel 15 is at a weird address
 *  channels 0 and 15 have an "external 128 bit 8 word read FIFO"
 *    for memory to memory transfers
 *
 * Experiments show that only channels 2-5,11-12 work with mmc
 */
```

⟨*kernel basic includes* 49a⟩

```
#include "io.h"
```

⟨*constant* DMAREGS*(arm)* 43e⟩

```
#define DBG if(Dbg)
```

⟨*enum* _anon_*(*(buses/arm/dma.c)(arm)*)* 956⟩

```
typedef struct Ctlr Ctlr;
typedef struct Cb Cb;
```

⟨*struct* Ctlr*(arm)* 957a⟩

⟨*struct* Cb*(arm)* 957b⟩

⟨*global* dma*(arm)* 957c⟩
⟨*global* dmaregs*(arm)* 957d⟩

⟨*function* dmaaddr*(arm)* 957h⟩

⟨*function* dmaioaddr*(arm)* 957l⟩

⟨*function* dump*(arm)* 958a⟩

⟨*function* dumpdregs*(arm)* 958b⟩

⟨*function* dmadone*(arm)* 958c⟩

⟨*function* dmainterrupt*(arm)* 958d⟩

⟨*function* dmastart*(arm)* 958e⟩

⟨*function* dmawait*(arm)* 959⟩

buses/arm/uartmini.c

⟨*constant* OkLed*(arm)* 961a⟩≡ (969b)
```
#define OkLed       16
```

⟨*constant* TxPin*(arm)* 961b⟩≡ (969b)
```
#define TxPin       14
```

⟨*constant* RxPin*(arm)* 961c⟩≡ (969b)
```
#define RxPin       15
```

⟨*enum* _anon_*(*(buses/arm/uartmini.c)(arm)*)* 961d⟩≡ (969b)
```
/* GPIO regs */
enum {
    Fsel0   = 0x00>>2,
        FuncMask= 0x7,
        Input   = 0x0,
        Output  = 0x1,
        Alt0    = 0x4,
        Alt1    = 0x5,
        Alt2    = 0x6,
        Alt3    = 0x7,
        Alt4    = 0x3,
        Alt5    = 0x2,
    Set0    = 0x1c>>2,
    Clr0    = 0x28>>2,
    Lev0    = 0x34>>2,
    PUD = 0x94>>2,
        Off = 0x0,
        Pulldown= 0x1,
        Pullup  = 0x2,
    PUDclk0 = 0x98>>2,
    PUDclk1 = 0x9c>>2,
};
```

⟨*enum* _anon_*(*(buses/arm/uartmini.c)2(arm)*)* 961e⟩≡ (969b)
```
/* AUX regs */
enum {
    Irq = 0x00>>2,
        UartIrq = 1<<0,
    Enables = 0x04>>2,
        UartEn  = 1<<0,
    MuIo    = 0x40>>2,
    MuIer   = 0x44>>2,
        RxIen   = 1<<0,
        TxIen   = 1<<1,
    MuIir   = 0x48>>2,
    MuLcr   = 0x4c>>2,
        Bitsmask= 3<<0,
        Bits7   = 2<<0,
        Bits8   = 3<<0,
    MuMcr   = 0x50>>2,
        RtsN    = 1<<1,
    MuLsr   = 0x54>>2,
        TxDone  = 1<<6,
        TxRdy   = 1<<5,
        RxRdy   = 1<<0,
    MuCntl  = 0x60>>2,
        CtsFlow = 1<<3,
        TxEn    = 1<<1,
        RxEn    = 1<<0,
```

```
    MuBaud  = 0x68>>2,
};
```

⟨*global* `miniuart` *(arm)* 962a⟩≡ (969b)
```
static Uart miniuart = {
    .regs   = (u32int*)AUXREGS,
    .name   = "uart0",
    .freq   = 250000000,
    .phys   = &miniphysuart,
};
```
Uses `AUXREGS-2181` 43g and `miniphysuart` 968b.

⟨*function* `gpiosel` *(arm)* 962b⟩≡ (969b)
```
void
gpiosel(uint pin, int func)
{
    u32int *gp, *fsel;
    int off;

    gp = (u32int*)GPIOREGS;
    fsel = &gp[Fsel0 + pin/10];
    off = (pin % 10) * 3;
    *fsel = (*fsel & ~(FuncMask<<off)) | func<<off;
}
```
Uses `Fsel0-2185` 961d, `FuncMask-2186` 961d, and `GPIOREGS-2180` 44d.

⟨*function* `gpiopull` *(arm)* 962c⟩≡ (969b)
```
static void
gpiopull(uint pin, int func)
{
    u32int *gp, *reg;
    u32int mask;

    gp = (u32int*)GPIOREGS;
    reg = &gp[PUDclk0 + pin/32];
    mask = 1 << (pin % 32);
    gp[PUD] = func;
    arch_microdelay(1);
    *reg = mask;
    arch_microdelay(1);
    *reg = 0;
}
```
Uses `GPIOREGS-2180` 44d, `PUD-2198` 961d, `PUDclk0-2202` 961d, and `arch_microdelay`.

⟨*function* `gpiopulloff` *(arm)* 962d⟩≡ (969b)
```
void
gpiopulloff(uint pin)
{
    gpiopull(pin, Off);
}
```
Uses `Off-2199` 961d and `gpiopull()` 962c.

⟨*function* `gpiopullup` *(arm)* 962e⟩≡ (969b)
```
void
gpiopullup(uint pin)
{
    gpiopull(pin, Pullup);
}
```
Uses `Pullup-2201` 961d and `gpiopull()` 962c.

⟨*function* gpiopulldown*(arm)* 963a⟩≡ (969b)
```
void
gpiopulldown(uint pin)
{
    gpiopull(pin, Pulldown);
}
```
Uses Pulldown-2200 961d and gpiopull() 962c.

⟨*function* gpioout*(arm)* 963b⟩≡ (969b)
```
void
gpioout(uint pin, int set)
{
    u32int *gp;
    int v;

    gp = (u32int*)GPIOREGS;
    v = set? Set0 : Clr0;
    gp[v + pin/32] = 1 << (pin % 32);
}
```
Uses Clr0-2196 961d, GPIOREGS-2180 44d, and Set0-2195 961d.

⟨*function* gpioin*(arm)* 963c⟩≡ (969b)
```
int
gpioin(uint pin)
{
    u32int *gp;

    gp = (u32int*)GPIOREGS;
    return (gp[Lev0 + pin/32] & (1 << (pin % 32))) != 0;
}
```
Uses GPIOREGS-2180 44d and Lev0-2197 961d.

⟨*function* interrupt*(arm)* 963d⟩≡ (969b)
```
static void
interrupt(Ureg*, void *arg)
{
    Uart *uart;
    u32int *ap;

    uart = arg;
    ap = (u32int*)uart->regs;

    arch_coherence();
    if(0 && (ap[Irq] & UartIrq) == 0)
        return;
    if(ap[MuLsr] & TxRdy)
        uartkick(uart);
    if(ap[MuLsr] & RxRdy){
        if(uart->console){
            if(uart->opens == 1)
                uart->putc = kbdcr2nl;
            else
                uart->putc = nil;
        }
        do{
            uartrecv(uart, ap[MuIo] & 0xFF);
        }while(ap[MuLsr] & RxRdy);
    }
    arch_coherence();
```

```
}
```

Uses Irq-2204 961e, MuIo-2208 961e, MuLsr-2219 961e, RxRdy-2222 961e, TxRdy-2221 961e, UartIrq-2205 961e, arch_coherence, kbdcr2nl() 676a, uartkick() 676d, and uartrecv() 676d.

⟨*function* pnp*(arm)* 964a⟩≡ (969b)

```
static Uart*
pnp(void)
{
    Uart *uart;

    uart = &miniuart;
    if(uart->console == 0)
        kbdq = qopen(8*1024, 0, nil, nil);
    return uart;
}
```

Uses kbdq 388c, miniuart-2228 962a, and qopen() 409a.

⟨*function* enable*(arm)* 964b⟩≡ (969b)

```
static void
enable(Uart *uart, int ie)
{
    u32int *ap;

    ap = (u32int*)uart->regs;
    arch_delay(10);
    gpiosel(TxPin, Alt5);
    gpiosel(RxPin, Alt5);
    gpiopulloff(TxPin);
    gpiopulloff(RxPin);
    ap[Enables] |= UartEn;
    ap[MuIir] = 6;
    ap[MuLcr] = Bits8;
    ap[MuCntl] = TxEn|RxEn;
    ap[MuBaud] = uart->freq/(115200*8) - 1;
    if(ie){
        arch_intrenable(IRQaux, interrupt, uart, 0, "uart");
        ap[MuIer] = RxIen|TxIen;
    }else
        ap[MuIer] = 0;
}
```

Uses Alt5-2194 961d, Bits8-2216 961e, Enables-2206 961e, MuBaud-2227 961e, MuCntl-2223 961e, MuIer-2209 961e, MuIir-2212 961e, MuLcr-2213 961e, RxEn-2226 961e, RxIen-2210 961e, RxPin-2184 961c, TxEn-2225 961e, TxIen-2211 961e, TxPin-2183 961b, UartEn-2207 961e, arch_delay, gpiopulloff() 962d, and gpiosel() 962b.

⟨*function* disable*(arm)* 964c⟩≡ (969b)

```
static void
disable(Uart *uart)
{
    u32int *ap;

    ap = (u32int*)uart->regs;
    ap[MuCntl] = 0;
    ap[MuIer] = 0;
}
```

Uses MuCntl-2223 961e and MuIer-2209 961e.

⟨*function* kick*(arm)* 965a⟩≡ (969b)

```
static void
kick(Uart *uart)
{
    u32int *ap;

    ap = (u32int*)uart->regs;
    if(uart->blocked)
        return;
    arch_coherence();
    while(ap[MuLsr] & TxRdy){
        if(uart->op >= uart->oe && uartstageoutput(uart) == 0)
            break;
        ap[MuIo] = *(uart->op++);
    }
    if(ap[MuLsr] & TxDone)
        ap[MuIer] &= ~TxIen;
    else
        ap[MuIer] |= TxIen;
    arch_coherence();
}
```

Uses MuIer-2209 961e, MuIo-2208 961e, MuLsr-2219 961e, TxDone-2220 961e, TxIen-2211 961e, TxRdy-2221 961e, arch_coherence, and uartstageoutput() 676d.

⟨*function* dobreak*(arm)* 965b⟩≡ (969b)

```
/* TODO */
static void
dobreak(Uart *uart, int ms)
{
    USED(uart, ms);
}
```

⟨*function* baud*(arm)* 965c⟩≡ (969b)

```
static int
baud(Uart *uart, int n)
{
    u32int *ap;

    ap = (u32int*)uart->regs;
    if(uart->freq == 0 || n <= 0)
        return -1;
    ap[MuBaud] = (uart->freq + 4*n - 1) / (8 * n) - 1;
    uart->baud = n;
    return 0;
}
```

Uses MuBaud-2227 961e.

⟨*function* bits*(arm)* 965d⟩≡ (969b)

```
static int
bits(Uart *uart, int n)
{
    u32int *ap;
    int set;

    ap = (u32int*)uart->regs;
    switch(n){
    case 7:
        set = Bits7;
        break;
    case 8:
```

```
            set = Bits8;
            break;
        default:
            return -1;
        }
        ap[MuLcr] = (ap[MuLcr] & ~Bitsmask) | set;
        uart->bits = n;
        return 0;
    }
```

Uses Bits7-2215 961e, Bits8-2216 961e, Bitsmask-2214 961e, and MuLcr-2213 961e.

⟨*function* stop*(arm)* 966a⟩≡ (969b)

```
    static int
    stop(Uart *uart, int n)
    {
        if(n != 1)
            return -1;
        uart->stop = n;
        return 0;
    }
```

⟨*function* parity*(arm)* 966b⟩≡ (969b)

```
    static int
    parity(Uart *uart, int n)
    {
        if(n != 'n')
            return -1;
        uart->parity = n;
        return 0;
    }
```

⟨*function* modemctl*(arm)* 966c⟩≡ (969b)

```
    /*
     * cts/rts flow control
     *   need to bring signals to gpio pins before enabling this
     */

    static void
    modemctl(Uart *uart, int on)
    {
        u32int *ap;

        ap = (u32int*)uart->regs;
        if(on)
            ap[MuCntl] |= CtsFlow;
        else
            ap[MuCntl] &= ~CtsFlow;
        uart->modem = on;
    }
```

Uses CtsFlow-2224 961e and MuCntl-2223 961e.

⟨*function* rts*(arm)* 966d⟩≡ (969b)

```
    static void
    rts(Uart *uart, int on)
    {
        u32int *ap;

        ap = (u32int*)uart->regs;
        if(on)
            ap[MuMcr] &= ~RtsN;
```

```
        else
            ap[MuMcr] |= RtsN;
    }
```

Uses MuMcr-2217 961e and RtsN-2218 961e.

⟨*function* status*(arm)* 967a⟩≡ (969b)

```
    static long
    status(Uart *uart, void *buf, long n, long offset)
    {
        char *p;

        p = malloc(READSTR);
        if(p == nil)
            error(Enomem);
        snprint(p, READSTR,
            "b%d\n"
            "dev(%d) type(%d) framing(%d) overruns(%d) "
            "berr(%d) serr(%d)\n",

            uart->baud,
            uart->dev,
            uart->type,
            uart->ferr,
            uart->oerr,
            uart->berr,
            uart->serr
        );
        n = readstr(offset, buf, n, p);
        free(p);

        return n;
    }
```

Uses READSTR 867a, free() 104a, malloc() 103c, and readstr() 670d.

⟨*function* donothing*(arm)* 967b⟩≡ (969b)

```
    static void
    donothing(Uart*, int)
    {
    }
```

⟨*function* putc*(arm)* 967c⟩≡ (969b)

```
    void
    putc(Uart*, int c)
    {
        u32int *ap;

        ap = (u32int*)AUXREGS;
        while((ap[MuLsr] & TxRdy) == 0)
            ;
        ap[MuIo] = c;
        while((ap[MuLsr] & TxRdy) == 0)
            ;
    }
```

Uses AUXREGS-2181 43g, MuIo-2208 961e, MuLsr-2219 961e, and TxRdy-2221 961e.

⟨*function* getc*(arm)* 967d⟩≡ (969b)

```
    int
    getc(Uart*)
    {
```

```
    u32int *ap;

    ap = (u32int*)AUXREGS;
    while((ap[MuLsr] & RxRdy) == 0)
        ;
    return ap[MuIo] & 0xFF;
}
```

Uses AUXREGS-2181 43g, MuIo-2208 961e, MuLsr-2219 961e, and RxRdy-2222 961e.

⟨*function* uartconsinit*(arm)* 968a⟩≡ (969b)

```
void
uartconsinit(void)
{
    Uart *uart;
    int n;
    char *p, *cmd;

    if((p = getconf("console")) == nil)
        return;
    n = strtoul(p, &cmd, 0);
    if(p == cmd)
        return;
    switch(n){
    default:
        return;
    case 0:
        uart = &miniuart;
        break;
    }

    if(!uart->enabled)
        (*uart->phys->enable)(uart, 0);
    uartctl(uart, "b9600 l8 pn s1");
    if(*cmd != '\0')
        uartctl(uart, cmd);

    consuart = uart;
    uart->console = 1;
}
```

Uses consuart 654c, miniuart-2228 962a, and uartctl() 676d.

⟨*global* miniphysuart*(arm)* 968b⟩≡ (969b)

```
PhysUart miniphysuart = {
    .name       = "miniuart",
    .pnp        = pnp,
    .enable     = enable,
    .disable    = disable,
    .kick       = kick,
    .dobreak    = dobreak,
    .baud       = baud,
    .bits       = bits,
    .stop       = stop,
    .parity     = parity,
    .modemctl   = donothing,
    .rts        = rts,
    .dtr        = donothing,
    .status     = status,
    .fifo       = donothing,
    .getc       = getc,
    .putc       = putc,
```

```
};
```
Uses baud() 965c, bits() 965d, dobreak() 965b, donothing() 967b, enable() 964b, getc() 967d, kick() 965a, parity() 966b, pnp() 964a, putc() 967c, rts() 966d, status() 967a, and stop() 966a.

⟨*function* okay*(arm)* 969a⟩≡ (969b)
```
void
okay(int on)
{
    static int first;
    static int okled, polarity;
    char *p;

    if(!first++){
        p = getconf("bcm2709.disk_led_gpio");
        if(p == nil)
            p = getconf("bcm2708.disk_led_gpio");
        if(p != nil)
            okled = strtol(p, 0, 0);
        else
            okled = OkLed;
        p = getconf("bcm2709.disk_led_active_low");
        if(p == nil)
            p = getconf("bcm2708.disk_led_active_low");
        polarity = (p == nil || *p == '1');
        gpiosel(okled, Output);
    }
    gpioout(okled, on^polarity);
}
```
Uses OkLed-2182 961a, Output-2188 961d, gpioout() 963b, and gpiosel() 962b.

⟨buses/arm/uartmini.c 969b⟩≡
```
/*
 * bcm2835 mini uart (UART1)
 */
```
⟨*kernel basic includes* 49a⟩

```
#include "io.h"
```

⟨*constant* GPIOREGS*(arm)* 44d⟩
⟨*constant* AUXREGS*(arm)* 43g⟩
⟨*constant* OkLed*(arm)* 961a⟩
⟨*constant* TxPin*(arm)* 961b⟩
⟨*constant* RxPin*(arm)* 961c⟩

⟨*enum* _anon_*(*(buses/arm/uartmini.c)(arm)*)* 961d⟩

⟨*enum* _anon_*(*(buses/arm/uartmini.c)2(arm)*)* 961e⟩

```
extern PhysUart miniphysuart;
```

⟨*global* miniuart*(arm)* 962a⟩

⟨*function* gpiosel*(arm)* 962b⟩

⟨*function* gpiopull*(arm)* 962c⟩

⟨*function* gpiopulloff*(arm)* 962d⟩

⟨*function* gpiopullup*(arm)* 962e⟩

⟨*function* `gpiopulldown`*(arm)* 963a⟩

⟨*function* `gpioout`*(arm)* 963b⟩

⟨*function* `gpioin`*(arm)* 963c⟩

⟨*function* `interrupt`*(arm)* 963d⟩

⟨*function* `pnp`*(arm)* 964a⟩

⟨*function* `enable`*(arm)* 964b⟩

⟨*function* `disable`*(arm)* 964c⟩

⟨*function* `kick`*(arm)* 965a⟩

⟨*function* `dobreak`*(arm)* 965b⟩

⟨*function* `baud`*(arm)* 965c⟩

⟨*function* `bits`*(arm)* 965d⟩

⟨*function* `stop`*(arm)* 966a⟩

⟨*function* `parity`*(arm)* 966b⟩

⟨*function* `modemctl`*(arm)* 966c⟩

⟨*function* `rts`*(arm)* 966d⟩

⟨*function* `status`*(arm)* 967a⟩

⟨*function* `donothing`*(arm)* 967b⟩

⟨*function* `putc`*(arm)* 967c⟩

⟨*function* `getc`*(arm)* 967d⟩

⟨*function* `uartconsinit`*(arm)* 968a⟩

⟨*global* `miniphysuart`*(arm)* 968b⟩

⟨*function* `okay`*(arm)* 969a⟩

F.4 concurrency/arm/

concurrency/arm/spl.s

⟨`concurrency/arm/spl.s` 970⟩≡

```
#include "mem.h"
#include "arm.h"
#include "arminstr.ha"
```

⟨*function* `arch_splhi`*(arm)* 74a⟩

⟨*function* `arch_spllo`*(arm)* 74b⟩

⟨*function* `arch_splx`*(arm)* 74c⟩

```
TEXT spldone(SB), 1, $0             /* end marker for devkprof.c */
    RET
```

⟨*function* arch_islo*(arm)* 74d⟩

⟨*function* arm_arch_coherence*(arm)* 547h⟩

```
// FIQ
```
⟨*function* splfhi*(arm)* 562b⟩

concurrency/arm/tas_raspi1.s

⟨concurrency/arm/tas_raspi1.s 971a⟩≡
```
TEXT    _tas(SB), $-4
```
⟨*function* arch_tas*(raspberry pi1)(arm)* 75⟩

concurrency/arm/tas_raspi2.s

⟨concurrency/arm/tas_raspi2.s 971b⟩≡
```
#include "mem.h"
#include "arm.h"
#include "arminstr.ha"
```

⟨*instruction DMB(arm)* 546e⟩

```
/* tas/cas strex debugging limits; started at 10000 */
#define MAXSC 100000

TEXT    _tas(SB), $-4           /* _tas(ulong *) */
```
⟨*function* arch_tas*(raspberry pi2)(arm)* 546f⟩

concurrency/arm/concurrency_raspi1.c

⟨*function* ainc*(arm)* 971c⟩≡ (972a)
```
// overwrite libc!
int
ainc(int *p)
{
    int s, v;

    s = arch_splhi();
    v = ++*p;
    arch_splx(s);
    return v;
}
```

⟨*function* adec*(arm)* 971d⟩≡ (972a)
```
int
adec(int *p)
{
    int s, v;

    s = arch_splhi();
```

```
    v = --*p;
    arch_splx(s);
    return v;
}
```

⟨concurrency/arm/concurrency_raspi1.c 972a⟩≡

```
/*
 * atomic ops
 * make sure that we don't drag in the C library versions
 */
```

⟨*kernel basic includes* 49a⟩

⟨*function* arch_xdec*(raspberry pi1)(arm)* 76b⟩

⟨*function* arch_xinc*(raspberry pi1)(arm)* 76a⟩

⟨*function* ainc*(arm)* 971c⟩

⟨*function* adec*(arm)* 971d⟩

⟨*function* cas32*(arm)* 474d⟩

⟨*function* arch_cmpswap*(raspberry pi1)(arm)* 474c⟩

concurrency/arm/concurrency_raspi2.c

⟨concurrency/arm/concurrency_raspi2.c 972b⟩≡

⟨*kernel basic includes* 49a⟩

```
// in libc/arm/atom.s (use LDREX, STREX special instructions)
extern long _xdec(long *p);
extern long _xinc(long *p);

long
arch_xdec(long *p)
{
  return _xdec(p);
}

void
arch_xinc(long *p)
{
  _xinc(p);
}

int
arch_cmpswap(long *addr, long old, long new)
{
    return cas((ulong*)addr, old, new);
}
```

F.5 core/arm/

core/arm/dat.h

⟨*constant* STAGESIZE*(arm)* 972c⟩≡ (973a)

```
  // was a default in portdat.h
  #define STAGESIZE 64 // for struct Uart
```

⟨core/arm/dat.h 973a⟩≡
```
  // pad's memory pointers (used in Lock so must be early)
  ⟨pad memory pointer types(arm) 55a⟩
  //TODO: need to update port/ code to use the new pointer types
  typedef void* virt_addr3;
  typedef void* kern_addr3;

  #include "dat_forward.h"
  #include "../port/portdat_forward.h"
  ⟨constant STAGESIZE(arm) 972c⟩

  ⟨dat.h includes(arm) 49b⟩

  // DO NOT switch those declarations. 5c allocates R10
  // for the first 'extern register' declaration seen in a file
  // and R9 for the second one. See Compiler.nw.
  // Then mem.h relies on this to define the UP and CPU macros used
  // inside assembly code.
  ⟨global cpu(arm) 64f⟩
  ⟨global up(arm) 68a⟩

  // in main.c (used in mmu.c)
  extern ulong memsize;

  ⟨constant KMESGSIZE(arm) 650b⟩

  // TODO: put that in portdat.h or in netif.h?
  #pragma varargck  type  "I" uchar*
  #pragma varargck  type  "V" uchar*
  #pragma varargck  type  "E" uchar*
  #pragma varargck  type  "M" uchar*
```

core/arm/dat_core.h

⟨core/arm/dat_core.h 973b⟩≡
```
  ⟨type PTE(arm) 179l⟩

  ⟨struct Arch_Cpu(arm) 64e⟩
```

core/arm/dat_forward.h

⟨core/arm/dat_forward.h 973c⟩≡
```
  // could be in portdat_forward.h
  typedef struct Arch_FPsave  Arch_FPsave;
  typedef struct Ureg Ureg;
  #pragma incomplete Ureg

  typedef struct Soc Soc;
```
Uses Soc 973c.

core/arm/fns.h

⟨core/arm/fns.h 974⟩≡

⟨*fns.h includes(arm)* 49c⟩

```
// not classified yet
extern void arch_cpuidprint(void);

// different signatures in archs or macro so cant factorize in portdat_xxx.h
extern void  arch_touser(uintptr);
extern int   arch_cmpswap(long*, long, long);
```
⟨*macro* getpgcolor*(arm)* 554d⟩
⟨*macro* arch_cycles*(arm)* 508c⟩
⟨*macro* arch_countpagerefs*(arm)* 559a⟩
⟨*macro* arch_intrenable*(arm)* 175e⟩

```
extern int arch__notify(Ureg*);
extern void arch__kexit(Ureg*);

extern void dumpregs(Ureg*);


extern void mmuinit1(void*);
extern void mmuinvalidate(void);
extern void mmuinvalidateaddr(u32int);


extern void irqenable(int, void (*)(Ureg*, void*), void*);

extern void intrcpushutdown(void);
extern void intrshutdown(void);
extern void intrsoff(void);


extern void archreboot(void);
extern void archreset(void);

extern void clockinit(void);
extern void clockshutdown(void);


extern int cas32(void*, u32int, u32int);
extern int cas(ulong*, ulong, ulong);

extern char *cputype2name(char *buf, int size);

extern void armtimerset(int);

extern void cachedwb(void);
extern void cachedwbinv(void);
extern void cachedinvse(void*, int);
extern void cachedwbse(void*, int);
extern void cachedwbinvse(void*, int);
extern void cacheiinv(void);
extern void cacheuwbinv(void);


// coproc.c
extern ulong cprd(int cp, int op1, int crn, int crm, int op2);
```

```
extern ulong cprdsc(int op1, int crn, int crm, int op2);
extern void cpwr(int cp, int op1, int crn, int crm, int op2, ulong val);
extern void cpwrsc(int op1, int crn, int crm, int op2, ulong val);

// dma.c
extern uintptr dmaaddr(void *va);
extern void dmastart(int, int, int, void*, void*, int);
extern int dmawait(int);

// vcore.c
extern int fbblank(int);
extern void* fbinit(bool, int*, int*, int*);


extern void fpon(void);
extern ulong fprd(int fpreg);
extern void fprestreg(int fpreg, uvlong val);
extern void fpsave(Arch_FPsave *);
extern ulong fpsavereg(int fpreg, uvlong *fpp);
extern void fpwr(int fpreg, ulong val);

extern u32int fsrget(void);
extern u32int farget(void);
extern u32int ifsrget(void);

// vcore.c
extern uint getboardrev(void);
extern ulong getclkrate(int);
extern uint getcputemp(void);
extern char *getethermac(void);
extern uint getfirmware(void);
extern void getramsize(Confmem*);

extern int getncpus(void);
extern int getpower(int);


extern void gpiosel(uint, int);
extern void gpiopullup(uint);
extern void gpiopulloff(uint);
extern void gpiopulldown(uint);

extern void i2cread(uint, void*, int);
extern void i2cwrite(uint, void*, int);


extern void okay(int);

extern int l2ap(int);
extern void l2cacheuwbinv(void);
```

⟨*macro* sdfree*(arm)* 764a⟩
⟨*macro* sdmalloc*(arm)* 764b⟩

```
extern void setclkrate(int, ulong);
extern void setpower(int, int);
extern void setr13(int, u32int*);

extern void spirw(uint, void*, int);
extern int splfhi(void);
```

```
extern int splflo(void);

extern int startcpus(uint);
extern void stopcpu(uint);


extern void uartconsinit(void);
extern void vectors(void);
extern void vtable(void);

extern void wdogoff(void);

/*
 * floating point emulation
 */
extern int fpiarm(Ureg*);
extern int fpudevprocio(Proc*, void*, long, uintptr, int);
extern void fpuinit(void);
extern void fpunoted(void);
extern void fpunotify(Ureg*);
extern void fpuprocrestore(Proc*);
extern void fpuprocsave(Proc*);
extern void fpusysprocsetup(Proc*);
extern void fpusysrfork(Ureg*);
extern void fpusysrforkchild(Proc*, Ureg*, Proc*);
extern int fpuemu(Ureg*);

/*
 * Miscellaneous machine dependent stuff.
 */
extern char* getenv(char*, char*, int);

uintptr mmukmap(uintptr, uintptr, usize);
uintptr mmukunmap(uintptr, uintptr, usize);


⟨macro PTR2UINT(arm) 56g⟩
⟨macro UINT2PTR(arm) 56h⟩

⟨macro KADDR(arm) 56f⟩
⟨macro PADDR(arm) 56b⟩
```

F.6 devices/arm/

devices/keyboard/arm/kbd.c

⟨devices/keyboard/arm/kbd.c 976⟩≡

```
⟨kernel basic includes 49a⟩

// In the Raspberry Pi, the keyboard is handled by usbd
// with help from /dev/kbin (which then calls functions from portkbd
// which calls arch_setleds below).

⟨function arch_setleds(arm) 380a⟩
```

devices/mouse/arm/mouse.c

⟨devices/mouse/arm/mouse.c 977a⟩≡
  ⟨*kernel basic includes* 49a⟩

```
// In the Raspberry Pi, the mouse is handled by usbd
// with help from /dev/mousein.

// empty callbacks for portmouse.c

void arch_ps2mouse(void) { }
void arch_setaccelerated(void) { }
void arch_setlinear(void) { }
void arch_setres(int n) { USED(n); }
void arch_setintellimouse(void) { }
void arch_resetmouse(void) { }
```

F.7 devices/sys/arm/

devices/sys/arm/devarch.c

⟨*enum* _anon_*(*(devices/sys/arm/devarch.c)(arm)*)* 977b⟩≡ (981a)

```
enum {
    Qdir = 0,
    Qbase,

    Qmax = 16,
};
```

⟨*global* readfn*(arm)* 977c⟩≡ (981a)

```
static Rdwrfn *readfn[Qmax];
```

Uses Qmax-851 977b.

⟨*global* writefn*(arm)* 977d⟩≡ (981a)

```
static Rdwrfn *writefn[Qmax];
```

Uses Qmax-851 977b.

⟨*global* archdir*(arm)* 977e⟩≡ (981a)

```
static Dirtab archdir[Qmax] = {
    ".",        { Qdir, 0, QTDIR }, 0,  0555,
};
```

Uses QTDIR 70b, Qdir-849 977b, and Qmax-851 977b.

⟨*global* archwlock*(arm)* 977f⟩≡ (981a)

```
Lock archwlock; /* the lock is only for changing archdir */
```

⟨*global* narchdir*(arm)* 977g⟩≡ (981a)

```
int narchdir = Qbase;
```

Uses Qbase-850 977b.

⟨*function* addarchfile*(arm)* 978a⟩≡ (981a)

```
/*
 * Add a file to the #P listing.  Once added, you can't delete it.
 * You can't add a file with the same name as one already there,
 * and you get a pointer to the Dirtab entry so you can do things
 * like change the Qid version.  Changing the Qid path is disallowed.
 */
Dirtab*
addarchfile(char *name, int perm, Rdwrfn *rdfn, Rdwrfn *wrfn)
{
    int i;
    Dirtab d;
    Dirtab *dp;

    memset(&d, 0, sizeof d);
    strcpy(d.name, name);
    d.perm = perm;

    lock(&archwlock);
    if(narchdir >= Qmax){
        unlock(&archwlock);
        return nil;
    }

    for(i=0; i<narchdir; i++)
        if(strcmp(archdir[i].name, name) == 0){
            unlock(&archwlock);
            return nil;
        }

    d.qid.path = narchdir;
    archdir[narchdir] = d;
    readfn[narchdir] = rdfn;
    writefn[narchdir] = wrfn;
    dp = &archdir[narchdir++];
    unlock(&archwlock);

    return dp;
}
```

Uses Qmax-851 977b, archdir-854 977e, lock() 76d, readfn-852 977c, unlock() 78d, and writefn-853 977d.

⟨*function* archattach*(arm)* 978b⟩≡ (981a)

```
static Chan*
archattach(char* spec)
{
    return devattach('P', spec);
}
```

Uses devattach() 297b.

⟨*function* archwalk*(arm)* 978c⟩≡ (981a)

```
Walkqid*
archwalk(Chan* c, Chan *nc, char** name, int nname)
{
    return devwalk(c, nc, name, nname, archdir, narchdir, devgen);
}
```

Uses archdir-854 977e, devgen() 428, and devwalk() 429b.

⟨*function* archstat*(arm)* 978d⟩≡ (981a)

```
static int
```

```
archstat(Chan* c, uchar* dp, int n)
{
    return devstat(c, dp, n, archdir, narchdir, devgen);
}
```
Uses archdir-854 977e, devgen() 428, and devstat() 431.

⟨*function* archopen*(arm)* 979a⟩≡ (981a)
```
static Chan*
archopen(Chan* c, int omode)
{
    return devopen(c, omode, archdir, narchdir, devgen);
}
```
Uses archdir-854 977e and devgen() 428.

⟨*function* archclose*(arm)* 979b⟩≡ (981a)
```
static void
archclose(Chan*)
{
}
```

⟨*function* archread*(arm)* 979c⟩≡ (981a)
```
static long
archread(Chan *c, void *a, long n, vlong offset)
{
    Rdwrfn *fn;

    switch((ulong)c->qid.path){
    case Qdir:
        return devdirread(c, a, n, archdir, narchdir, devgen);

    default:
        if(c->qid.path < narchdir && (fn = readfn[c->qid.path]))
            return fn(c, a, n, offset);
        error(Eperm);
        break;
    }

    return 0;
}
```
Uses Qdir-849 977b, archdir-854 977e, devdirread() 432c, devgen() 428, and readfn-852 977c.

⟨*function* archwrite*(arm)* 979d⟩≡ (981a)
```
static long
archwrite(Chan *c, void *a, long n, vlong offset)
{
    Rdwrfn *fn;

    if(c->qid.path < narchdir && (fn = writefn[c->qid.path]))
        return fn(c, a, n, offset);
    error(Eperm);

    return 0;
}
```
Uses writefn-853 977d.

⟨*global* `archdevtab` *(arm)* 980a⟩≡ (981a)

```
Dev archdevtab = {
    .dc = 'P',
    .name = "arch",

    .reset = devreset,
    .init = archinit,
    .shutdown = devshutdown,
    .attach = archattach,
    .walk = archwalk,
    .stat = archstat,
    .open = archopen,
    .create = devcreate,
    .close = archclose,
    .read = archread,
    .bread = devbread,
    .write = archwrite,
    .bwrite = devbwrite,
    .remove = devremove,
    .wstat = devwstat,
};
```

Uses `devbread()` 434a, `devbwrite()` 434b, `devcreate()` 810, `devremove()` 810, `devreset()` 810, `devshutdown()` 810, and `devwstat()` 810.

⟨*function* `cputyperead` *(arm)* 980b⟩≡ (981a)

```
static long
cputyperead(Chan*, void *a, long n, vlong offset)
{
    char name[64], str[128];

    cputype2name(name, sizeof name);
    snprint(str, sizeof str, "ARM %s %d\n", name, cpu->cpumhz);
    return readstr(offset, a, n, str);
}
```

Uses `readstr()` 670d.

⟨*function* `cputempread` *(arm)* 980c⟩≡ (981a)

```
static long
cputempread(Chan*, void *a, long n, vlong offset)
{
    char str[16];

    snprint(str, sizeof str, "%ud\n", (getcputemp()+500)/1000);
    return readstr(offset, a, n, str);
}
```

Uses `getcputemp()` 897a and `readstr()` 670d.

⟨*function* `archinit` *(arm)* 980d⟩≡ (981a)

```
void
archinit(void)
{
    addarchfile("cputype", 0444, cputyperead, nil);
    addarchfile("cputemp", 0444, cputempread, nil);
}
```

Uses `cputempread()` 980c.

⟨devices/sys/arm/devarch.c 981a⟩≡
```
⟨kernel basic includes 49a⟩

#include "io.h"

⟨enum _anon_((devices/sys/arm/devarch.c)(arm)) 977b⟩

typedef long Rdwrfn(Chan*, void*, long, vlong);
void archinit(void);

⟨global readfn(arm) 977c⟩
⟨global writefn(arm) 977d⟩

⟨global archdir(arm) 977e⟩

⟨global archwlock(arm) 977f⟩
⟨global narchdir(arm) 977g⟩

⟨function addarchfile(arm) 978a⟩

⟨function archattach(arm) 978b⟩

⟨function archwalk(arm) 978c⟩

⟨function archstat(arm) 978d⟩

⟨function archopen(arm) 979a⟩

⟨function archclose(arm) 979b⟩

⟨function archread(arm) 979c⟩

⟨function archwrite(arm) 979d⟩

⟨global archdevtab(arm) 980a⟩

⟨function cputyperead(arm) 980b⟩

⟨function cputempread(arm) 980c⟩

⟨function archinit(arm) 980d⟩
```

F.8 devices/storage/arm/

devices/storage/arm/emmc.c

⟨enum _anon_((devices/storage/arm/emmc.c)(arm)) 981b⟩≡ (989b)
```
enum {
    Extfreq     = 100*Mhz,  /* guess external clock frequency if */
                    /* not available from vcore */
    Initfreq    = 400000,   /* initialisation frequency for MMC */
    SDfreq      = 25*Mhz,   /* standard SD frequency */
    DTO     = 14,       /* data timeout exponent (guesswork) */

    MMCSelect   = 7,        /* mmc/sd card select command */
    Setbuswidth = 6,        /* mmc/sd set bus width command */
};
```
Uses Mhz 874a.

⟨*enum* _anon_((devices/storage/arm/emmc.c)2(arm)) 982⟩≡ (989b)

```
enum {
    /* Controller registers */
    Arg2            = 0x00>>2,
    Blksizecnt      = 0x04>>2,
    Arg1            = 0x08>>2,
    Cmdtm           = 0x0c>>2,
    Resp0           = 0x10>>2,
    Resp1           = 0x14>>2,
    Resp2           = 0x18>>2,
    Resp3           = 0x1c>>2,
    Data            = 0x20>>2,
    Status          = 0x24>>2,
    Control0        = 0x28>>2,
    Control1        = 0x2c>>2,
    Interrupt       = 0x30>>2,
    Irptmask        = 0x34>>2,
    Irpten          = 0x38>>2,
    Control2        = 0x3c>>2,
    Forceirpt       = 0x50>>2,
    Boottimeout     = 0x70>>2,
    Dbgsel          = 0x74>>2,
    Exrdfifocfg     = 0x80>>2,
    Exrdfifoen      = 0x84>>2,
    Tunestep        = 0x88>>2,
    Tunestepsstd        = 0x8c>>2,
    Tunestepsddr        = 0x90>>2,
    Spiintspt       = 0xf0>>2,
    Slotisrver      = 0xfc>>2,

    /* Control0 */
    Dwidth4         = 1<<1,
    Dwidth1         = 0<<1,

    /* Control1 */
    Srstdata        = 1<<26,    /* reset data circuit */
    Srstcmd         = 1<<25,    /* reset command circuit */
    Srsthc          = 1<<24,    /* reset complete host controller */
    Datatoshift     = 16,       /* data timeout unit exponent */
    Datatomask      = 0xF0000,
    Clkfreq8shift       = 8,        /* SD clock base divider LSBs */
    Clkfreq8mask        = 0xFF00,
    Clkfreqms2shift     = 6,        /* SD clock base divider MSBs */
    Clkfreqms2mask      = 0xC0,
    Clkgendiv       = 0<<5,     /* SD clock divided */
    Clkgenprog      = 1<<5,     /* SD clock programmable */
    Clken           = 1<<2,     /* SD clock enable */
    Clkstable       = 1<<1,
    Clkintlen       = 1<<0,     /* enable internal EMMC clocks */

    /* Cmdtm */
    Indexshift      = 24,
    Suspend         = 1<<22,
    Resume          = 2<<22,
    Abort           = 3<<22,
    Isdata          = 1<<21,
    Ixchken         = 1<<20,
    Crcchken        = 1<<19,
    Respmask        = 3<<16,
    Respnone        = 0<<16,
```

```
    Resp136         = 1<<16,
    Resp48          = 2<<16,
    Resp48busy      = 3<<16,
    Multiblock      = 1<<5,
    Host2card       = 0<<4,
    Card2host       = 1<<4,
    Autocmd12       = 1<<2,
    Autocmd23       = 2<<2,
    Blkcnten        = 1<<1,

    /* Interrupt */
    Acmderr     = 1<<24,
    Denderr     = 1<<22,
    Dcrcerr     = 1<<21,
    Dtoerr      = 1<<20,
    Cbaderr     = 1<<19,
    Cenderr     = 1<<18,
    Ccrcerr     = 1<<17,
    Ctoerr      = 1<<16,
    Err     = 1<<15,
    Cardintr    = 1<<8,     /* not in Broadcom datasheet */
    Cardinsert  = 1<<6,     /* not in Broadcom datasheet */
    Readrdy     = 1<<5,
    Writerdy    = 1<<4,
    Datadone    = 1<<1,
    Cmddone     = 1<<0,

    /* Status */
    Bufread     = 1<<11,    /* not in Broadcom datasheet */
    Bufwrite    = 1<<10,    /* not in Broadcom datasheet */
    Readtrans   = 1<<9,
    Writetrans  = 1<<8,
    Datactive   = 1<<2,
    Datinhibit  = 1<<1,
    Cmdinhibit  = 1<<0,
  };
```

⟨*global* `cmdinfo` *(arm)* 983a⟩≡ (989b)

```
  int cmdinfo[64] = {
  [0]  Ixchken,
  [2]  Resp136,
  [3]  Resp48 | Ixchken | Crcchken,
  [6]  Resp48 | Ixchken | Crcchken,
  [7]  Resp48busy | Ixchken | Crcchken,
  [8]  Resp48 | Ixchken | Crcchken,
  [9]  Resp136,
  [12] Resp48busy | Ixchken | Crcchken,
  [13] Resp48 | Ixchken | Crcchken,
  [16] Resp48,
  [17] Resp48 | Isdata | Card2host | Ixchken | Crcchken,
  [18] Resp48 | Isdata | Card2host | Multiblock | Blkcnten | Ixchken | Crcchken,
  [24] Resp48 | Isdata | Host2card | Ixchken | Crcchken,
  [25] Resp48 | Isdata | Host2card | Multiblock | Blkcnten | Ixchken | Crcchken,
  [41] Resp48,
  [55] Resp48 | Ixchken | Crcchken,
  };
```

Uses `Blkcnten-1681` 982, `Card2host-1678` 982, `Crcchken-1670` 982, `Host2card-1677` 982, `Isdata-1668` 982, `Ixchken-1669` 982, `Multiblock-1676` 982, `Resp136-1673` 982, `Resp48-1674` 982, and `Resp48busy-1675` 982.

⟨*struct* `Ctlr` *(*`(devices/storage/arm/emmc.c)(arm)`*)* 983b⟩≡ (989b)

```
struct Ctlr {
    Rendez  r;
    int datadone;
    int fastclock;
    ulong   extclk;
};
```

⟨*global* emmc *(arm)* 984a⟩≡ (989b)
```
static Ctlr emmc;
```

⟨*function* WR *(arm)* 984b⟩≡ (989b)
```
static void
WR(int reg, u32int val)
{
    u32int *r = (u32int*)EMMCREGS;

    //if(0)print("WR %2.2ux %ux\n", reg<<2, val);
    arch_microdelay(emmc.fastclock? 2 : 20);
    r[reg] = val;
}
```
Uses EMMCREGS-1615 44c, arch_microdelay, and emmc-1704 984a.

⟨*function* clkdiv *(arm)* 984c⟩≡ (989b)
```
static uint
clkdiv(uint d)
{
    uint v;

    assert(d < 1<<10);
    v = (d << Clkfreq8shift) & Clkfreq8mask;
    v |= ((d >> 8) << Clkfreqms2shift) & Clkfreqms2mask;
    return v;
}
```
Uses Clkfreq8mask-1656 982, Clkfreq8shift-1655 982, Clkfreqms2mask-1658 982, Clkfreqms2shift-1657 982, and assert 853f.

⟨*function* datadone *(arm)* 984d⟩≡ (989b)
```
static int
datadone(void*)
{
    int i;

    u32int *r = (u32int*)EMMCREGS;
    i = r[Interrupt];
    return i & (Datadone|Err);
}
```
Uses Datadone-1695 982, EMMCREGS-1615 44c, Err-1690 982, and Interrupt-1634 982.

⟨*function* emmcinit *(arm)* 984e⟩≡ (989b)
```
static int
emmcinit(void)
{
    u32int *r;
    ulong clk;
    char *s;

    clk = getclkrate(ClkEmmc);
    s = "";
    if(clk == 0){
```

```
        s = "Assuming ";
        clk = Extfreq;
    }
    emmc.extclk = clk;
    print("%seMMC external clock %lud Mhz\n", s, clk / Mhz);
    r = (u32int*)EMMCREGS;
    //if(0)print("emmc control %8.8ux %8.8ux %8.8ux\n", r[Control0], r[Control1], r[Control2]);
    WR(Control1, Srsthc);
    arch_delay(10);
    while(r[Control1] & Srsthc)
        ;
    return 0;
}
```

Uses ClkEmmc 896a, Control1-1633 982, EMMCREGS-1615 44c, Extfreq-1616 981b, Mhz 874a, Srsthc-1652 982, WR() 984b, arch_delay, emmc-1704 984a, getclkrate() 896b, and print.

⟨*function* emmcinquiry*(arm)* 985a⟩≡ (989b)

```
static int
emmcinquiry(char *inquiry, int inqlen)
{
    u32int *r;
    uint ver;

    r = (u32int*)EMMCREGS;
    ver = r[Slotisrver] >> 16;
    return snprint(inquiry, inqlen,
        "Arasan eMMC SD Host Controller %2.2x Version %2.2x",
        ver&0xFF, ver>>8);
}
```

Uses EMMCREGS-1615 44c and Slotisrver-1647 982.

⟨*function* mmcinterrupt*(arm)* 985b⟩≡ (989b)

```
static void
mmcinterrupt(Ureg*, void*)
{
    u32int *r;
    r = (u32int*)EMMCREGS;
    if(r[Interrupt]&(Datadone|Err)){
        WR(Irpten, 0);
        wakeup(&emmc.r);
    }
}
```

Uses Datadone-1695 982, EMMCREGS-1615 44c, Err-1690 982, Interrupt-1634 982, Irpten-1636 982, WR() 984b, emmc-1704 984a, and wakeup.

⟨*function* emmcenable*(arm)* 985c⟩≡ (989b)

```
static void
emmcenable(void)
{
    u32int *r;
    int i;

    r = (u32int*)EMMCREGS;
    WR(Control1, clkdiv(emmc.extclk/Initfreq - 1) |
        DTO<<Datatoshift | Clkgendiv | Clken | Clkintlen);
    for(i = 0; i < 1000; i++){
        arch_delay(1);
        if(r[Control1] & Clkstable)
            break;
```

```
    }
    if(i == 1000)
        print("SD clock won't initialise!\n");
    WR(Irptmask, ~(Dtoerr|Cardintr));
    arch_intrenable(IRQmmc, mmcinterrupt, nil, 0, "mmc");
}
```

Uses Cardintr-1691 982, Clken-1661 982, Clkgendiv-1659 982, Clkintlen-1663 982, Clkstable-1662 982, Control1-1633 982, DTO-1619 981b, Datatoshift-1653 982, Dtoerr-1685 982, EMMCREGS-1615 44c, Initfreq-1617 981b, Irptmask-1635 982, WR() 984b, arch_delay, clkdiv() 984c, emmc-1704 984a, and print.

⟨*function* emmccmd*(arm)* 986⟩≡ (989b)

```
static int
emmccmd(u32int cmd, u32int arg, u32int *resp)
{
    u32int *r;
    u32int c;
    int i;
    ulong now;

    r = (u32int*)EMMCREGS;
    assert(cmd < nelem(cmdinfo) && cmdinfo[cmd] != 0);
    c = (cmd << Indexshift) | cmdinfo[cmd];
    if(r[Status] & Cmdinhibit){
        print("emmccmd: need to reset Cmdinhibit intr %ux stat %ux\n",
            r[Interrupt], r[Status]);
        WR(Control1, r[Control1] | Srstcmd);
        while(r[Control1] & Srstcmd)
            ;
        while(r[Status] & Cmdinhibit)
            ;
    }
    if((r[Status] & Datinhibit) &&
       ((c & Isdata) || (c & Respmask) == Resp48busy)){
        print("emmccmd: need to reset Datinhibit intr %ux stat %ux\n",
            r[Interrupt], r[Status]);
        WR(Control1, r[Control1] | Srstdata);
        while(r[Control1] & Srstdata)
            ;
        while(r[Status] & Datinhibit)
            ;
    }
    WR(Arg1, arg);
    if((i = r[Interrupt]) != 0){
        if(i != Cardinsert)
            print("emmc: before command, intr was %ux\n", i);
        WR(Interrupt, i);
    }
    WR(Cmdtm, c);
    now = cpu->ticks;
    while(((i=r[Interrupt])&(Cmddone|Err)) == 0)
        if(cpu->ticks - now > Arch_HZ)
            break;
    if((i&(Cmddone|Err)) != Cmddone){
        if((i&~Err) != Ctoerr)
            print("emmc: cmd %ux error intr %ux stat %ux\n", c, i, r[Status]);
        WR(Interrupt, i);
        if(r[Status]&Cmdinhibit){
            WR(Control1, r[Control1]|Srstcmd);
            while(r[Control1]&Srstcmd)
                ;
```

```
    }
    error(Eio);
}
WR(Interrupt, i & ~(Datadone|Readrdy|Writerdy));
switch(c & Respmask){
case Resp136:
    resp[0] = r[Resp0]<<8;
    resp[1] = r[Resp0]>>24 | r[Resp1]<<8;
    resp[2] = r[Resp1]>>24 | r[Resp2]<<8;
    resp[3] = r[Resp2]>>24 | r[Resp3]<<8;
    break;
case Resp48:
case Resp48busy:
    resp[0] = r[Resp0];
    break;
case Respnone:
    resp[0] = 0;
    break;
}
if((c & Respmask) == Resp48busy){
    WR(Irpten, Datadone|Err);
    tsleep(&emmc.r, datadone, 0, 3000);
    WR(Irpten, 0);
    emmc.datadone = 0;
    i = r[Interrupt];
    if((i & Datadone) == 0)
        print("emmcio: no Datadone after CMD%d\n", cmd);
    if(i & Err)
        print("emmcio: CMD%d error interrupt %ux\n",
            cmd, r[Interrupt]);
    WR(Interrupt, i);
}
/*
 * Once card is selected, use faster clock
 */
if(cmd == MMCSelect){
    arch_delay(10);
    WR(Control1, clkdiv(emmc.extclk/SDfreq - 1) |
        DTO<<Datatoshift | Clkgendiv | Clken | Clkintlen);
    for(i = 0; i < 1000; i++){
        arch_delay(1);
        if(r[Control1] & Clkstable)
            break;
    }
    arch_delay(10);
    emmc.fastclock = 1;
}
/*
 * If card bus width changes, change host bus width
 */
if(cmd == Setbuswidth){
    switch(arg){
    case 0:
        WR(Control0, r[Control0] & ~Dwidth4);
        break;
    case 2:
        WR(Control0, r[Control0] | Dwidth4);
        break;
    }
}
```

```
    return 0;
}
```

Uses Arg1-1624 982, Cardinsert-1692 982, Clken-1661 982, Clkgendiv-1659 982, Clkintlen-1663 982, Clkstable-1662 982, Cmddone-1696 982, Cmdinhibit-1703 982, Cmdtm-1625 982, Control0-1632 982, Control1-1633 982, Ctoerr-1689 982, DTO-1619 981b, Datadone-1695 982, Datatoshift-1653 982, Datinhibit-1702 982, Dwidth4-1648 982, EMMCREGS-1615 44c, Err-1690 982, Indexshift-1664 982, Interrupt-1634 982, Irpten-1636 982, Isdata-1668 982, MMCSelect-1620 981b, Readrdy-1693 982, Resp0-1626 982, Resp1-1627 982, Resp136-1673 982, Resp2-1628 982, Resp3-1629 982, Resp48-1674 982, Resp48busy-1675 982, Respmask-1671 982, Respnone-1672 982, SDfreq-1618 981b, Setbuswidth-1621 981b, Srstcmd-1651 982, Srstdata-1650 982, Status-1631 982, WR() 984b, Writerdy-1694 982, arch_delay, assert 853f, clkdiv() 984c, cmdinfo 983a, datadone() 984d, emmc-1704 984a, nelem 853i, print, and tsleep.

⟨*function* emmciosetup*(arm)* 988a⟩≡ (989b)

```
void
emmciosetup(int write, void *buf, int bsize, int bcount)
{
    USED(write);
    USED(buf);
    WR(Blksizecnt, bcount<<16 | bsize);
}
```

Uses Blksizecnt-1623 982 and WR() 984b.

⟨*function* emmcio*(arm)* 988b⟩≡ (989b)

```
static void
emmcio(int write, uchar *buf, int len)
{
    u32int *r;
    int i;

    r = (u32int*)EMMCREGS;
    assert((len&3) == 0);
    okay(1);
    if(waserror()){
        okay(0);
        nexterror();
    }
    if(write)
        dmastart(DmaChanEmmc, DmaDevEmmc, DmaM2D,
            buf, &r[Data], len);
    else
        dmastart(DmaChanEmmc, DmaDevEmmc, DmaD2M,
            &r[Data], buf, len);
    if(dmawait(DmaChanEmmc) < 0)
        error(Eio);
    if(!write)
        cachedinvse(buf, len);
    WR(Irpten, Datadone|Err);
    tsleep(&emmc.r, datadone, 0, 3000);
    WR(Irpten, 0);
    emmc.datadone = 0;
    i = r[Interrupt];
    if((i & Datadone) == 0){
        print("emmcio: %d timeout intr %ux stat %ux\n",
            write, i, r[Status]);
        WR(Interrupt, i);
        error(Eio);
    }
    if(i & Err){
        print("emmcio: %d error intr %ux stat %ux\n",
            write, r[Interrupt], r[Status]);
        WR(Interrupt, i);
```

```
            error(Eio);
        }
        if(i)
            WR(Interrupt, i);
        poperror();
        okay(0);
    }
```

Uses Data-1630 982, Datadone-1695 982, DmaChanEmmc 955b, DmaD2M 955b, DmaDevEmmc 955b, DmaM2D 955b, EMMCREGS-1615 44c, Err-1690 982, Interrupt-1634 982, Irpten-1636 982, Status-1631 982, WR() 984b, assert 853f, datadone() 984d, emmc-1704 984a, nexterror, okay() 969a, poperror 666d, print, tsleep, and waserror 666d.

⟨*global* sdio*(arm)* 989a⟩≡ (989b)

```
SDio sdio = {
    "emmc",
    emmcinit,
    emmcenable,
    emmcinquiry,
    emmccmd,
    emmciosetup,
    emmcio,
};
```

Uses emmccmd() 986, emmcenable() 985c, emmcinit() 984e, emmcinquiry() 985a, emmcio() 988b, and emmciosetup() 988a.

⟨devices/storage/arm/emmc.c 989b⟩≡

```
/*
 * bcm2835 external mass media controller (mmc / sd host interface)
 *
 * Copyright © 2012 Richard Miller <r.miller@acm.org>
 */
```

⟨*kernel basic includes* 49a⟩

```
#include "io.h"
#include "../port/sd.h"
```

⟨*constant* EMMCREGS*(arm)* 44c⟩

⟨*enum* _anon_*(*(devices/storage/arm/emmc.c)(arm)*)* 981b⟩

⟨*enum* _anon_*(*(devices/storage/arm/emmc.c)2(arm)*)* 982⟩

⟨*global* cmdinfo*(arm)* 983a⟩

```
typedef struct Ctlr Ctlr;
```

⟨*struct* Ctlr*(*(devices/storage/arm/emmc.c)(arm)*)* 983b⟩

⟨*global* emmc*(arm)* 984a⟩

⟨*function* WR*(arm)* 984b⟩

⟨*function* clkdiv*(arm)* 984c⟩

⟨*function* datadone*(arm)* 984d⟩

⟨*function* emmcinit*(arm)* 984e⟩

⟨*function* emmcinquiry*(arm)* 985a⟩

⟨*function* mmcinterrupt*(arm)* 985b⟩

⟨*function* emmcenable*(arm)* 985c⟩

⟨*function* `emmccmd`*(arm)* 986⟩

⟨*function* `emmciosetup`*(arm)* 988a⟩

⟨*function* `emmcio`*(arm)* 988b⟩

⟨*global* `sdio`*(arm)* 989a⟩

F.9 init/arm/

init/arm/start.s

⟨`init/arm/start.s` 990a⟩≡

```
/*
 * Common startup for armv6 and armv7
 * The rest of l.s has been moved to armv[67].s
 */
#include "mem.h"
#include "arm.h"
#include "arminstr.ha"
```

⟨*function* `_start`*(arm)* 269⟩

init/arm/startv6.s

⟨`init/arm/startv6.s` 990b⟩≡

```
/*
 * Broadcom bcm2835 SoC, as used in Raspberry Pi
 * arm1176jzf-s processor (armv6)
 */
#include "mem.h"
#include "arm.h"
#include "arminstr.ha"
```

⟨*function* `armstart`*(raspberry pi1)(arm)* 270⟩

⟨*function* `_startpg`*(raspberry pi1)(arm)* 272a⟩

⟨*function* `arch_idlehands`*(raspberry pi1)(arm)* 242c⟩

init/arm/startv7.s

⟨`init/arm/startv7.s` 990c⟩≡

```
/*
 * Broadcom bcm2836 SoC, as used in Raspberry Pi 2
 * 4 x Cortex-A7 processor (armv7)
 */
#include "mem.h"
#include "arm.h"
#include "arminstr.ha"
```

⟨*instruction* `WFI_EQ`*(arm)* 545c⟩

⟨*function* armstart*(raspberry pi2)(arm)* 537f⟩

⟨*function* _startpg*(raspberry pi2)(arm)* 539f⟩

⟨*function* cpureset*(raspberry pi2)(arm)* 542⟩

⟨*function* _startpg2*(raspberry pi2)(arm)* 543⟩

⟨*function* cpidget*(raspberry pi2)(arm)* 545a⟩

⟨*function* arch_idlehands*(raspberry pi2)(arm)* 545b⟩

init/arm/main.c

⟨*function* getconf*(arm)* 991a⟩≡ (991c)

```
char*
getconf(char *name)
{
    USED(name);
    //int i;
    //
    //i = findconf(name);
    //if(i >= 0)
    //  return confval[i];
    return nil;
}
```

⟨*function* main_arch_isaconfig*(arm)* 991b⟩≡ (991c)

```
/*
 * stub for ../omap/devether.c
 */
int
main_arch_isaconfig(char *class, int ctlrno, ISAConf *isa)
{
    char cc[32], *p;
    int i;

    if(strcmp(class, "ether") != 0)
        return 0;
    snprint(cc, sizeof cc, "%s%d", class, ctlrno);
    p = getconf(cc);
    if(p == nil)
        return (ctlrno == 0);
    isa->type = "";
    isa->nopt = tokenize(p, isa->opt, NISAOPT);
    for(i = 0; i < isa->nopt; i++){
        p = isa->opt[i];
        if(cistrncmp(p, "type=", 5) == 0)
            isa->type = p + 5;
    }
    return 1;
}
```

Uses NISAOPT 674a and cistrncmp().

⟨init/arm/main.c 991c⟩≡

⟨*kernel basic includes* 49a⟩

```
#include "io.h"
// for mainmem and imagmem
#include <pool.h>

// initcode binary
#include "init.h"

// rebootcode binary
#include "reboot.h"

// in portscreen.h
extern void swcursor_init(void);
```

⟨*constant* `Minfirmrev` *(arm)* 286e⟩
⟨*constant* `Minfirmdate` *(arm)* 286f⟩

```
// part of a trick to remove some backward dependencies
int devcons_print(char*, ...);
int devcons_iprint(char*, ...);
int devcons_pprint(char*, ...);
void devcons_panic(char*, ...);
void devcons__assert(char*);
void proc_dumpaproc(Proc *p);
void proc_error(char*);
void proc_nexterror(void);
void proc_sched(void);
void proc_ready(Proc*);
void proc_sleep(Rendez*, int(*)(void*), void*);
void proc_tsleep(Rendez *r, int (*fn)(void*), void *arg, ulong ms);
Proc* proc_wakeup(Rendez*);
void proc_pexit(char *exitstr, bool freemem);
Proc* proc_proctab(int i);
int proc_postnote(Proc *p, int dolock, char *n, int flag);
void chan_cclose(Chan *c);

uvlong clock_arch_fastticks(uvlong *hz);
void clock_arch_delay(int millisecs);
void clock_arch_microdelay(int microsecs);
void trap_arch_dumpstack(void);
void arm_arch_coherence(void);
void main_arch_exit(int ispanic);
int  main_arch_isaconfig(char *class, int ctlrno, ISAConf *isa);

//=========================================================================
// Configuration
//=========================================================================
// See globals in portdat_globals.h

// <conf>.c
extern  Dev*  conf_devtab[];
extern  char*   conffile;
```

⟨*function* `getconf` *(arm)* 991a⟩

```
//=========================================================================
// Boot parameters, see bootconf.c (not used by pad)
//=========================================================================

//=========================================================================
```

```
// Cpu init
//*****************************************************************************
```
⟨*function* `arch__cpuinit`*(arm)* 274d⟩

⟨*function* `arch__cpu0init`*(arm)* 274c⟩

⟨*function* `machon`*(arm)* 275a⟩

```
//*****************************************************************************
// Conf init
//*****************************************************************************
```

⟨*global* `memsize`*(arm)* 276c⟩

⟨*function* `arch__confinit`*(arm)* 276d⟩

```
//*****************************************************************************
// First process init
//*****************************************************************************
```

⟨*global* `sp`*(arm)* 287b⟩

```
// kernel space instructions executed by first process
```
⟨*function* `init0`*(arm)* 289a⟩

```
// user space instructions executed by first process
// see initcode in init.h (comes from ../port/initcode.c and init9.s

extern uintptr bootargs(uintptr base);
```
⟨*function* `arch__userinit`*(arm)* 287d⟩

```
//*****************************************************************************
// Shutdown/reboot
//*****************************************************************************
```

⟨*function* `shutdown`*(*`(init/arm/main.c)(arm)`*)* 628f⟩

⟨*function* `main_arch_exit`*(arm)* 628d⟩

⟨*function* `arch_reboot`*(arm)* 629b⟩

```
//*****************************************************************************
// Misc
//*****************************************************************************
```

⟨*function* `main_arch_isaconfig`*(arm)* 991b⟩

⟨*function* `arch_memorysummary`*(arm)* 644b⟩

```
//*****************************************************************************
// Main entry point!
//*****************************************************************************
```

⟨*function* `launchinit`*(arm)* 540a⟩

⟨*function* `main`*(arm)* 272b⟩

init/arm/raspi1.c

⟨*function* archether*(raspberry pi1)(arm)* 994a⟩≡ (994b)

```
int
archether(unsigned ctlrno, Ether *ether)
{
    ether->type = "usb";
    ether->ctlrno = ctlrno;
    ether->irq = -1;
    ether->nopt = 0;
    return 1;
}
```

⟨init/arm/raspi1.c 994b⟩≡

```
/*
 * bcm2835 (e.g. original raspberry pi) architecture-specific stuff
 */
```

⟨*kernel basic includes* 49a⟩

```
#include "io.h"
#include "arm.h"

#include "../port/netif.h"
#include "../port/etherif.h"
```

⟨*global* soc*(raspberry pi1)(arm)* 72b⟩

⟨*function* cputype2name*(raspberry pi1)(arm)* 275c⟩

⟨*function* arch_cpuidprint*(raspberry pi1)(arm)* 275b⟩

⟨*function* getncpus*(raspberry pi1)(arm)* 540e⟩

⟨*function* startcpus*(raspberry pi1)(arm)* 540d⟩

⟨*function* archether*(raspberry pi1)(arm)* 994a⟩

⟨*function* l2ap*(raspberry pi1)(arm)* 179j⟩

init/arm/raspi2.c

⟨init/arm/raspi2.c 994c⟩≡

```
/*
 * bcm2836 (e.g.raspberry pi 2) architecture-specific stuff
 */
```

⟨*kernel basic includes* 49a⟩

```
#include "io.h"
#include "arm.h"

#include "../port/netif.h"
#include "../port/etherif.h"

typedef struct Mbox Mbox;
typedef struct Mboxes Mboxes;
```

⟨*constant* ARMLOCAL*(arm)* 536e⟩

⟨*global* soc*(raspberry pi2)(arm)* 534i⟩

⟨*struct* Mbox*(arm)* 536c⟩
⟨*struct* Mboxes*(arm)* 536d⟩

```
enum {
    Mboxregs    = 0x80
};

static Lock startlock[MAXCPUS + 1];

// in startv7.s
extern u32int cpidget(void);
```

⟨*function* cputype2name*(raspberry pi2)(arm)* 544d⟩

⟨*function* arch_cpuidprint*(raspberry pi2)(arm)* 544c⟩

⟨*function* getncpus*(raspberry pi2)(arm)* 540f⟩

⟨*function* startcpu*(raspberry pi2)(arm)* 541c⟩

⟨*function* startcpus*(raspberry pi2)(arm)* 541b⟩

```
int
archether(unsigned ctlrno, Ether *ether)
{
    ether->type = "usb";
    ether->ctlrno = ctlrno;
    ether->irq = -1;
    ether->nopt = 0;
    return 1;
}
```

⟨*function* l2ap*(raspberry pi2)(arm)* 535g⟩

```
// in main.c
extern void arch__cpuinit(void);
extern void machon(uint);
```

⟨*function* cpustart*(raspberry pi2)(arm)* 544a⟩

Uses Mbox 536c and Mboxes 536d.

init/arm/bootconf.c

⟨*enum* _anon_*((init/arm/bootconf.c)(arm))* 995a⟩≡ (999b)

```
enum {
    /* space for syscall args, return PC, top-of-stack struct */
    Ustkheadroom    = sizeof(Sargs) + sizeof(uintptr) + sizeof(Tos),
};
```

⟨*constant* CONFADDR*(arm)* 995b⟩≡ (1005)

```
#define CONFADDR    (KZERO+0x100)       /* unparsed plan9.ini */
```

⟨*constant* BOOTARGS*(arm)* 996a⟩≡ (999b)
```
/*
 * Where configuration info is left for the loaded programme.
 */
#define BOOTARGS    ((char*)CONFADDR)
```

⟨*constant* BOOTARGSLEN*(arm)* 996b⟩≡ (999b)
```
#define BOOTARGSLEN (CPUADDR-CONFADDR)
```

⟨*global* oargc*(arm)* 996c⟩≡ (999b)
```
/*
 * Option arguments from the command line.
 * oargv[0] is the boot file.
 */
static int oargc;
```

⟨*global* oargv*(arm)* 996d⟩≡ (999b)
```
static char* oargv[20];
```

⟨*global* oargb*(arm)* 996e⟩≡ (999b)
```
static char oargb[128];
```

⟨*global* oargblen*(arm)* 996f⟩≡ (999b)
```
static int oargblen;
```

⟨*struct* Atag*(arm)* 996g⟩≡ (999b)
```
struct Atag {
    u32int  size;   /* size of atag in words, including this header */
    u32int  tag;    /* atag type */
    union {
        u32int  data[1];    /* actually [size-2] */
        /* AtagMem */
        struct {
            u32int  size;
            u32int  base;
        } mem;
        /* AtagCmdLine */
        char    cmdline[1]; /* actually [4*(size-2)] */
    };
};
```
Uses __anon_struct_336 996g and __anon_struct_337 996g.

⟨*enum* _anon_*(*(init/arm/bootconf.c)2(arm)*)* 996h⟩≡ (999b)
```
enum {
    AtagNone    = 0x00000000,
    AtagCore    = 0x54410001,
    AtagMem     = 0x54410002,
    AtagCmdline = 0x54410009,
};
```

⟨*function* findconf*(arm)* 996i⟩≡ (999b)
```
static int
findconf(char *name)
{
    int i;

    for(i = 0; i < nconf; i++)
        if(cistrcmp(confname[i], name) == 0)
            return i;
    return -1;
}
```
Uses confname 63a and nconf 63c.

⟨*function* addconf *(arm)* 997a⟩≡ (999b)

```
void
addconf(char *name, char *val)
{
    int i;

    i = findconf(name);
    if(i < 0){
        if(val == nil || nconf >= MAXCONF)
            return;
        i = nconf++;
        strecpy(confname[i], confname[i]+sizeof(confname[i]), name);
    }
    strecpy(confval[i], confval[i]+sizeof(confval[i]), val);
}
```

Uses MAXCONF 63b, confname 63a, confval 63a, findconf() 996i, and nconf 63c.

⟨*function* writeconf *(arm)* 997b⟩≡ (999b)

```
static void
writeconf(void)
{
    char *p, *q;
    int n;

    p = getconfenv();

    if(waserror()) {
        free(p);
        nexterror();
    }

    /* convert to name=value\n format */
    for(q=p; *q; q++) {
        q += strlen(q);
        *q = '=';
        q += strlen(q);
        *q = '\n';
    }
    n = q - p + 1;
    if(n >= BOOTARGSLEN)
        error("kernel configuration too large");
    memmove(BOOTARGS, p, n);
    memset(BOOTARGS + n, '\n', BOOTARGSLEN - n);
    poperror();
    free(p);
}
```

Uses BOOTARGS-2436 996a, BOOTARGSLEN-2437 996b, free() 104a, getconfenv() 733d, nexterror, poperror 666d, and waserror 666d.

⟨*function* plan9iniinit *(arm)* 997c⟩≡ (999b)

```
static void
plan9iniinit(char *s, int cmdline)
{
    char *toks[MAXCONF];
    int i, c, n;
    char *v;

    if((c = *s) < ' ' || c >= 0x80)
        return;
    if(cmdline)
```

```
        n = tokenize(s, toks, MAXCONF);
    else
        n = getfields(s, toks, MAXCONF, 1, "\n");
    for(i = 0; i < n; i++){
        if(toks[i][0] == '#')
            continue;
        v = strchr(toks[i], '=');
        if(v == nil)
            continue;
        *v++ = '\0';
        addconf(toks[i], v);
    }
}
```

Uses MAXCONF 63b and addconf() 997a.

⟨*function* ataginit*(arm)* 998a⟩≡ (999b)

```
static void
ataginit(Atag *a)
{
    int n;

    if(a->tag != AtagCore){
        plan9iniinit((char*)a, 0);
        return;
    }
    while(a->tag != AtagNone){
        switch(a->tag){
        case AtagMem:
            /* use only first bank */
            if(conf.mem[0].limit == 0 && a->mem.size != 0){
                memsize = a->mem.size;
                conf.mem[0].base = a->mem.base;
                conf.mem[0].limit = a->mem.base + memsize;
            }
            break;
        case AtagCmdline:
            n = (a->size * sizeof(u32int)) - offsetof(Atag, cmdline[0]);
            if(a->cmdline + n < BOOTARGS + BOOTARGSLEN)
                a->cmdline[n] = 0;
            else
                BOOTARGS[BOOTARGSLEN-1] = 0;
            plan9iniinit(a->cmdline, 1);
            break;
        }
        a = (Atag*)((u32int*)a + a->size);
    }
}
```

Uses AtagCmdline-2445 996h, AtagCore-2443 996h, AtagMem-2444 996h, AtagNone-2442 996h, BOOTARGS-2436 996a, BOOTARGSLEN-2437 996b, conf 62b, memsize 276c, offsetof, and plan9iniinit() 997c.

⟨*function* optionsinit*(arm)* 998b⟩≡ (999b)

```
static void
optionsinit(char* s)
{
    strecpy(oargb, oargb+sizeof(oargb), s);

    oargblen = strlen(oargb);
    oargc = tokenize(oargb, oargv, nelem(oargv)-1);
    oargv[oargc] = nil;
}
```

Uses nelem 853i, oargb-2440 996e, oargblen-2441 996f, oargc-2438 996c, and oargv-2439 996d.

⟨*function* bootargs*(arm)* 999a⟩≡ (999b)
```
uintptr
bootargs(uintptr base)
{
    int i;
    ulong ssize;
    char **av, *p;

    /*
     * Push the boot args onto the stack.
     * The initial value of the user stack must be such
     * that the total used is larger than the maximum size
     * of the argument list checked in syscall.
     */
    i = oargblen+1;
    p = UINT2PTR(STACKALIGN(base + BY2PG - Ustkheadroom - i));
    memmove(p, oargb, i);

    /*
     * Now push the argv pointers.
     * The code jumped to by touser in lproc.s expects arguments
     *  main(char* argv0, ...)
     * and calls
     *  startboot("/boot/boot", &argv0)
     * not the usual (int argc, char* argv[])
     */
    av = (char**)(p - (oargc+1)*sizeof(char*));
    ssize = base + BY2PG - PTR2UINT(av);
    for(i = 0; i < oargc; i++)
        *av++ = (oargv[i] - oargb) + (p - base) + (USTKTOP - BY2PG);
    *av = nil;
    //sp = USTKTOP - ssize;
    return USTKTOP - ssize;
}
```
Uses PTR2UINT 56g, STACKALIGN 287c, UINT2PTR 56h, Ustkheadroom-2435 995a, oargb-2440 996e, oargblen-2441 996f, oargc-2438 996c, and oargv-2439 996d.

⟨init/arm/bootconf.c 999b⟩≡
```
⟨kernel basic includes 49a⟩

#include "tos.h"

⟨enum _anon_(init/arm/bootconf.c)(arm) 995a⟩

⟨constant BOOTARGS(arm) 996a⟩
⟨constant BOOTARGSLEN(arm) 996b⟩
//#define MAXCONF       64
//#define MAXCONFLINE   160

⟨global oargc(arm) 996c⟩
⟨global oargv(arm) 996d⟩
⟨global oargb(arm) 996e⟩
⟨global oargblen(arm) 996f⟩

/* store plan9.ini contents here at least until we stash them in #ec */
//static char confname[MAXCONF][KNAMELEN];
//static char confval[MAXCONF][MAXCONFLINE];
//static int nconf;
```

```
typedef struct Atag Atag;
```
⟨*struct* Atag*(arm)* 996g⟩

⟨*enum* _anon_*(*(init/arm/bootconf.c)2(arm)*)* 996h⟩

⟨*function* findconf*(arm)* 996i⟩

⟨*function* addconf*(arm)* 997a⟩

⟨*function* writeconf*(arm)* 997b⟩

⟨*function* plan9iniinit*(arm)* 997c⟩

⟨*function* ataginit*(arm)* 998a⟩

⟨*function* optionsinit*(arm)* 998b⟩

⟨*function* bootargs*(arm)* 999a⟩

Uses Atag 996g.

init/user/preboot/arm/init9.s

init/arm/rebootcode.s

⟨init/arm/rebootcode.s 1000a⟩≡

```
/*
 * armv6/armv7 reboot code
 */
#include "mem.h"
#include "arm.h"
#include "arminstr.ha"
```

⟨*constant* PTEDRAM*(arm)* 631b⟩

⟨*instruction WFI(arm)* 631c⟩
⟨*instruction WFE(arm)* 631d⟩

⟨*function main(rebootcode.s)(arm)* 631e⟩

init/arm/watchdog.c

⟨*enum* _anon_*(*(init/arm/watchdog.c)(arm)*)* 1000b⟩≡ (1002a)

```
enum {
    Wdogfreq    = 65536,
    Wdogtime    = 5,    /* seconds, <= 15 */
};
```

⟨*enum* _anon_*(*(init/arm/watchdog.c)2(arm)*)* 1000c⟩≡ (1002a)

```
/*
 * Power management / watchdog registers
 */
enum {
    Rstc        = 0x1c>>2,
        Password    = 0x5A<<24,
        CfgMask     = 0x03<<4,
```

```
        CfgReset    = 0x02<<4,
    Rsts        = 0x20>>2,
    Wdog        = 0x24>>2,
};
```

⟨*function* `archreset` *(arm)* 1001a⟩≡ (1002a)
```
void
archreset(void)
{
    fpon();
}
```
Uses `fpon()` 524c.

⟨*function* `archreboot` *(arm)* 1001b⟩≡ (1002a)
```
void
archreboot(void)
{
    u32int *r;

    r = (u32int*)POWERREGS;
    r[Wdog] = Password | 1;
    r[Rstc] = Password | (r[Rstc] & ~CfgMask) | CfgReset;
    arch_coherence();
    for(;;)
        ;
}
```
Uses `CfgMask-2454` 1000c, `CfgReset-2455` 1000c, `POWERREGS-2449` 43f, `Password-2453` 1000c, `Rstc-2452` 1000c, `Wdog-2457` 1000c, and `arch_coherence`.

⟨*function* `wdogfeed` *(arm)* 1001c⟩≡ (1002a)
```
static void
wdogfeed(void)
{
    u32int *r;

    r = (u32int*)POWERREGS;
    r[Wdog] = Password | (Wdogtime * Wdogfreq);
    r[Rstc] = Password | (r[Rstc] & ~CfgMask) | CfgReset;
}
```
Uses `CfgMask-2454` 1000c, `CfgReset-2455` 1000c, `POWERREGS-2449` 43f, `Password-2453` 1000c, `Rstc-2452` 1000c, `Wdog-2457` 1000c, `Wdogfreq-2450` 1000b, and `Wdogtime-2451` 1000b.

⟨*function* `wdogoff` *(arm)* 1001d⟩≡ (1002a)
```
void
wdogoff(void)
{
    u32int *r;

    r = (u32int*)POWERREGS;
    r[Rstc] = Password | (r[Rstc] & ~CfgMask);
}
```
Uses `CfgMask-2454` 1000c, `POWERREGS-2449` 43f, `Password-2453` 1000c, and `Rstc-2452` 1000c.

⟨*function* `watchdoglink` *(arm)* 1001e⟩≡ (1002a)
```
void
watchdoglink(void)
{
    addclock0link(wdogfeed, Arch_HZ);
}
```
Uses `addclock0link()` 253f and `wdogfeed()` 1001c.

⟨init/arm/watchdog.c 1002a⟩≡
  ⟨kernel basic includes 49a⟩

  ⟨constant POWERREGS(arm) 43f⟩

  ⟨enum _anon_((init/arm/watchdog.c)(arm)) 1000b⟩

  ⟨enum _anon_((init/arm/watchdog.c)2(arm)) 1000c⟩

  ⟨function archreset(arm) 1001a⟩

  ⟨function archreboot(arm) 1001b⟩

  ⟨function wdogfeed(arm) 1001c⟩

  ⟨function wdogoff(arm) 1001d⟩

  ⟨function watchdoglink(arm) 1001e⟩

F.10 interrupts/arm/

interrupts/arm/trap.c

⟨enum _anon_((interrupts/arm/trap.c)(arm)) 1002b⟩≡ (1002c)
```
enum {
    Debug = 0,

    ⟨constant Nvec(arm) 278d⟩
    ⟨constant Fiqenable(arm) 561e⟩

    ⟨constant Localtimerint(arm) 553d⟩
    ⟨constant Localmboxint(arm) 553f⟩
    ⟨constant Localintpending(arm) 553e⟩
};
```

⟨interrupts/arm/trap.c 1002c⟩≡
```
/*
 * traps, exceptions, interrupts, system calls.
 */
```
  ⟨kernel basic includes 49a⟩
```
#include "io.h"
#include "ureg.h"
#include "arm.h"
```

  ⟨constant INTREGS(arm) 43d⟩
  ⟨constant LOCALREGS(arm) 553g⟩

```
typedef struct Intregs Intregs;
typedef struct Vctl Vctl;
typedef struct Vpage0 Vpage0;
```

  ⟨enum _anon_((interrupts/arm/trap.c)(arm)) 1002b⟩

  ⟨struct Vpage0(arm) 278c⟩

  ⟨struct Intregs(arm) 175d⟩

  ⟨struct Vctl(arm) 174c⟩

⟨*global* `vctllock`*(arm)* 175c⟩

⟨*global* `vctl`*(arm)* 175a⟩

⟨*global* `vfiq`*(arm)* 561d⟩

⟨*global* `trapnames`*(arm)* 161b⟩

⟨*function* `arch__trapinit`*(arm)* 278e⟩

⟨*function* `intrcpushutdown`*(arm)* 631a⟩

⟨*function* `intrsoff`*(arm)* 279a⟩

⟨*function* `intrshutdown`*(arm)* 630a⟩

⟨*function* `intrtime`*(arm)* 657c⟩

⟨*function* `irq`*(arm)* 177b⟩

⟨*function* `fiq`*(arm)* 562a⟩

⟨*function* `irqenable`*(arm)* 176a⟩

⟨*function* `trapname`*(arm)* 647c⟩

⟨*function* `ckfaultstuck`*(arm)* 648a⟩

⟨*function* `faultarm`*(arm)* 182b⟩

⟨*function* `writetomem`*(arm)* 172e⟩

⟨*function* `arch__trap`*(arm)* 168b⟩

⟨*function* `isvalidaddr`*(arm)* 647b⟩

⟨*function* `dumplongs`*(arm)* 647a⟩

⟨*function* `dumpstackwithureg`*(arm)* 649a⟩

⟨*function* `getpcsp`*(arm)* 648d⟩

⟨*function* `arch_callwithureg`*(arm)* 648c⟩

⟨*function* `trap_arch_dumpstack`*(arm)* 648b⟩

⟨*function* `dumpregs`*(arm)* 646b⟩

Uses `Intregs` 175d and `Vpage0` 278c.

interrupts/arm/fault_helpers.s

⟨`interrupts/arm/fault_helpers.s` 1003⟩≡

```
#include "mem.h"
#include "arm.h"
```

⟨*function* `fstget`*(arm)* 172c⟩

⟨*function* `ifsrget`*(arm)* 171d⟩

⟨*function* `farget`*(arm)* 172b⟩

interrupts/arm/lexception.s

⟨`interrupts/arm/lexception.s` 1004a⟩≡

```
/*
 * arm exception handlers
 */
#include "mem.h"
#include "arm.h"
#include "arminstr.ha"
```

⟨*global* `vectors`*(arm)* 161c⟩

⟨*global* `vtable`*(arm)* 161d⟩

⟨*function* `_vsvc`*(arm)* 162c⟩

⟨*function* `_vund`*(arm)* 166a⟩

⟨*function* `_vpabt`*(arm)* 166b⟩

⟨*function* `_vdabt`*(arm)* 166c⟩

⟨*function* `_virq`*(arm)* 166d⟩

⟨*label* `_vswitch`*(arm)* 167a⟩

⟨*function* `_vfiq`*(arm)* 561a⟩

⟨*function* `setr13`*(arm)* 279d⟩

F.11 memory/arm/

memory/arm/dat_memory.h

⟨`memory/arm/dat_memory.h` 1004b⟩≡

```
/*
 *  MMU stuff in proc
 */
```

⟨*constant* `NCOLOR`*(arm)* 554e⟩

⟨*struct* `Arch_Proc`*(arm)* 67a⟩

```
/*
 * Fake kmap.
 */
typedef void        Arch_KMap;
```

⟨*macro* `VA`*(arm)* 180e⟩

memory/arm/mem.h

⟨memory/arm/mem.h 1005⟩≡

```
/*
 * Memory and machine-specific definitions.  Used in C and assembler.
 */

⟨constant KiB(arm) 47g⟩
⟨constant MiB(arm) 48a⟩
⟨constant GiB(arm) 48b⟩

/*
 * Sizes
 */
⟨constant BY2PG(arm) 41a⟩
⟨constant PGSHIFT(arm) 41b⟩

⟨constant MAXCPUS(arm) 65b⟩
⟨constant CPUSIZE(arm) 64b⟩
⟨constant L1SIZE(arm) 540b⟩

⟨macro STACKALIGN(arm) 287c⟩

/*
 * Magic registers
 */
⟨constant UP(arm) 162b⟩
⟨constant CPU(arm) 162a⟩

/*
 * Address spaces.
 * KTZERO is used by kprof and dumpstack (if any).
 *
 * KZERO is mapped to physical 0 (start of ram).
 *
 * vectors are at 0, plan9.ini is at KZERO+256 and is limited to 16K by
 * devenv.
 */
⟨constant KSEG0(arm) 51b⟩
⟨constant KSEGM(arm) 56e⟩

⟨constant KZERO(arm) 51c⟩

⟨constant CONFADDR(arm) 995b⟩

// L1 must be 16KiB aligned so this dictates the organization below
⟨constant CPUADDR(arm) 56a⟩
⟨constant L2(arm) 271b⟩
⟨constant VCBUFFER(arm) 386a⟩
⟨constant FIQSTKTOP(arm) 560e⟩
⟨constant L1(arm) 271a⟩

⟨constant KTZERO(arm) 51a⟩

⟨constant VIRTIO(arm) 43a⟩
⟨constant FRAMEBUFFER(arm) 44a⟩

⟨constant UZERO(arm) 52a⟩
⟨constant UTZERO(arm) 52b⟩
⟨macro UTROUND(arm) 143c⟩
```

⟨*constant* USTKTOP*(arm)* 52c⟩
⟨*constant* USTKSIZE*(arm)* 52d⟩
⟨*constant* TSTKTOP*(arm)* 145b⟩
⟨*constant* TSTKSIZ*(arm)* 145c⟩

⟨*constant* REBOOTADDR*(arm)* 628e⟩

```
/*
 * Legacy...
 */
//old: was 64 in bcm-latest, but then get panic in _allocb
```
⟨*constant* BLOCKALIGN*(arm)* 435d⟩
⟨*constant* KSTACK*(arm)* 226g⟩

```
/*
 * Sizes
 */
```
⟨*constant* BY2WD*(arm)* 37a⟩
⟨*constant* BY2V*(arm)* 37b⟩

⟨*constant* PAGETABMAPMEM*(arm)* 114c⟩
⟨*constant* PAGETABSIZE*(arm)* 114d⟩
⟨*constant* PAGEDIRSIZE*(arm)* 114a⟩
⟨*constant* SMALLPAGEDIRSIZE*(arm)* 117c⟩
⟨*macro* PPN*(arm)* 180d⟩

```
/*
 * Physical machine information from here on.
 *  PHYS addresses as seen from the arm cpu.
 *  BUS  addresses as seen from the videocore gpu.
 */
```
⟨*constant* PHYSDRAM*(arm)* 56d⟩
⟨*constant* IOSIZE*(arm)* 45⟩

memory/arm/mmu.c

⟨*enum* _anon_*(*(memory/arm/mmu.c)(arm)*)* 1006a⟩≡ (1006b)
```
enum {
```
⟨*constant* L1lo*(arm)* 180a⟩
⟨*constant* L1hi*(arm)* 180b⟩
```
};
```

⟨memory/arm/mmu.c 1006b⟩≡
⟨*kernel basic includes* 49a⟩
```
#include "arm.h"
```

⟨*macro* L1X*(arm)* 178a⟩
⟨*macro* L2X*(arm)* 178b⟩
⟨*macro* L2AP*(arm)* 179i⟩

⟨*constant* L1ptedramattrs*(arm)* 530a⟩
⟨*constant* L2ptedramattrs*(arm)* 530b⟩

⟨*enum* _anon_*(*(memory/arm/mmu.c)(arm)*)* 1006a⟩

⟨*function* mmuinit*(arm)* 271c⟩

⟨*function* mmuinit1*(arm)* 281c⟩

⟨*function* `mmul2empty`*(arm)* 181a⟩

⟨*function* `mmul1empty`*(arm)* 181e⟩

⟨*function* `arch_mmuswitch`*(arm)* 230b⟩

⟨*function* `arch_flushmmu`*(arm)* 182a⟩

⟨*function* `arch_mmurelease`*(arm)* 180f⟩

⟨*function* `arch_putmmu`*(arm)* 186⟩

⟨*function* `arch_cankaddr`*(arm)* 56i⟩

⟨*function* `mmukmap`*(arm)* 387a⟩

⟨*function* `arch_checkmmu`*(arm)* 188c⟩

⟨*function* `arch_kmap`*(arm)* 188d⟩

⟨*function* `arch_kunmap`*(arm)* 188e⟩

memory/arm/mmuinvalidate.s

⟨`memory/arm/mmuinvalidate.s` 1007a⟩≡
```
#include "mem.h"
#include "arm.h"
#include "arminstr.ha"
```

⟨*function* `mmuinvalidate`*(arm)* 180g⟩

⟨*function* `mmuinvalidateaddr`*(arm)* 187b⟩

memory/arm/cache_raspi1.s

⟨`memory/arm/cache_raspi1.s` 1007b⟩≡
```
#include "mem.h"
#include "arm.h"
#include "arminstr.ha"
```

⟨*constant CACHELINESZ(raspberry pi1)(ram)* 530g⟩

⟨*function cachexxx(raspberry pi1)(ram)* 532e⟩

memory/arm/cache_raspi2.s

⟨`memory/arm/cache_raspi2.s` 1007c⟩≡
```
#include "mem.h"
#include "arm.h"
#include "arminstr.ha"
```

⟨*constant CACHELINESZ(raspberry pi2)(ram)* 548a⟩

⟨*function cachexxx(raspberry pi2)(ram)* 548d⟩

F.12 processes/arm/

processes/arm/dat_processes.h

⟨processes/arm/dat_processes.h 1008a⟩≡

⟨*constant* AOUT_MAGIC*(arm)* 140d⟩

⟨*enum* _anon_*(*(arch/arm/dat_processes.h)(arm)*)* 514i⟩

⟨*struct* Arch_FPsave*(arm)* 514g⟩

processes/arm/arch.c

⟨processes/arm/arch.c 1008b⟩≡
⟨*kernel basic includes* 49a⟩

```
#include <tos.h>
#include "ureg.h"

#include "arm.h"

/*
 * A lot of this stuff doesn't belong here
 * but this is a convenient dumping ground for
 * later sorting into the appropriate buckets.
 */
```

⟨*function* arch_userpc*(arm)* 165f⟩

⟨*function* arch_dbgpc*(arm)* 645c⟩

⟨*function* arch_userureg*(arm)* 165h⟩

⟨*function* arch_setregisters*(arm)* 482c⟩

⟨*function* arch_setkernur*(arm)* 608b⟩

⟨*function* arch_validalign*(arm)* 563c⟩

⟨*function* arch__kexit*(arm)* 169⟩

⟨*function* linkproc*(arm)* 158c⟩

⟨*function* arch_kprocchild*(arm)* 158b⟩

processes/arm/vfp3.c

⟨processes/arm/vfp3.c 1008c⟩≡

```
/*
 * VFPv2 or VFPv3 floating point unit
 */
```

⟨*kernel basic includes* 49a⟩

```
#include "ureg.h"
#include "arm.h"
```

⟨*enum* _anon_*(*(processes/arm/vfp3.c)(arm)*)* 518c⟩

⟨*enum* _anon_*(*(processes/arm/vfp3.c)2(arm)*)* 521a⟩
⟨*enum* _anon_*(*(processes/arm/vfp3.c)3(arm)*)* 521b⟩
⟨*enum* _anon_*(*(processes/arm/vfp3.c)4(arm)*)* 521c⟩

⟨*function* subarch*(arm)* 521d⟩

⟨*function* implement*(arm)* 522a⟩

⟨*function* havefp*(arm)* 522c⟩

⟨*function* fpoff*(arm)* 523a⟩

⟨*function* fpononly*(arm)* 523b⟩

⟨*function* fpcfg*(arm)* 523c⟩

⟨*function* fpinit*(arm)* 524b⟩

⟨*function* fpon*(arm)* 524c⟩

⟨*function* fpclear*(arm)* 524d⟩

⟨*function* fpunotify*(arm)* 517c⟩

⟨*function* fpunoted*(arm)* 518a⟩

⟨*function* fpusysrfork*(arm)* 517a⟩

⟨*function* fpusysrforkchild*(arm)* 515b⟩

⟨*function* fpsave*(arm)* 517b⟩

⟨*function* fprestore*(arm)* 528a⟩

⟨*function* fpuprocsave*(arm)* 516a⟩

⟨*function* fpuprocrestore*(arm)* 516c⟩

⟨*function* fpusysprocsetup*(arm)* 515d⟩

⟨*function* arch_procsetup*(arm)* 143e⟩

⟨*function* arch_procsave*(arm)* 230c⟩

⟨*function* arch_procrestore*(arm)* 231a⟩

⟨*function* mathnote*(arm)* 527b⟩

⟨*function* mathemu*(arm)* 527c⟩

⟨*function* fpstuck*(arm)* 524e⟩

⟨*enum* _anon_*(*(processes/arm/vfp3.c)5(arm)*)* 525b⟩

⟨*function* condok*(arm)* 525c⟩

```
/* instruction decoding */ // was in arm.h
⟨macro ISCPOP(arm) 526a⟩
⟨macro ISVFPOP(arm) 526b⟩

⟨function fpuemu(arm) 526c⟩
```

processes/arm/lproc.s

⟨processes/arm/lproc.s 1010a⟩≡

```
#include "mem.h"
#include "arm.h"

⟨function arch_touser(arm) 290b⟩

⟨function arch__forkret(arm) 136b⟩
```

processes/arm/switch.s

⟨processes/arm/switch.s 1010b⟩≡

```
⟨function arch_setlabel(arm) 227b⟩

⟨function arch_gotolabel(arm) 227a⟩

⟨function getcallerpc(arm) 47a⟩
```

F.13 time/arm/

time/arm/dat_time.h

⟨time/arm/dat_time.h 1010c⟩≡

```
/*
 * Time.
 *
 * HZ should divide 1000 evenly, ideally.
 * 100, 125, 200, 250 and 333 are okay.
 */
⟨constant Arch_HZ(arm) 235c⟩

// was vlong in x86
//typedef uvlong        Tval;
```

time/arm/clock.c

⟨enum _anon_(time/arm/clock.c)(arm) 1010d⟩≡ (1011b)

```
enum {
    ⟨constant Localctl(arm) 547a⟩
    ⟨constant Prescaler(arm) 547b⟩

    ⟨constant SystimerFreq(arm) 245d⟩

    ⟨constant MaxPeriod(arm) 251c⟩
    ⟨constant MinPeriod(arm) 251d⟩
};
```

⟨*enum* _anon_*(*(time/arm/clock.c)2(arm)*)* 1011a⟩≡ (1011b)

```
enum {
    CntPrescaleShift= 16,   /* freq is sys_clk/(prescale+1) */
    CntPrescaleMask = 0xFF,
    CntEnable   = 1<<9,
    TmrDbgHalt  = 1<<8,
    TmrEnable   = 1<<7,
    TmrIntEnable    = 1<<5,
    TmrPrescale1    = 0x00<<2,
    TmrPrescale16   = 0x01<<2,
    TmrPrescale256  = 0x02<<2,
    CntWidth16  = 0<<1,
    CntWidth32  = 1<<1,

    /* generic timer (cortex-a7) */
    Enable  = 1<<0,
    Imask   = 1<<1,
    Istatus = 1<<2,
};
```

⟨time/arm/clock.c 1011b⟩≡

```
/*
 * bcm283[56] timers
 *  System timers run at 1MHz (timers 1 and 2 are used by GPU)
 *  ARM timer usually runs at 250MHz (may be slower in low power modes)
 *  Cycle counter runs at 700MHz (unless overclocked)
 *    All are free-running up-counters
 *  Cortex-a7 has local generic timers per cpu (which we run at 1MHz)
 *
 * Use system timer 3 (64 bits) for hzclock interrupts and fastticks
 *   For smp on bcm2836, use local generic timer for interrupts on cpu1-3
 * Use ARM timer (32 bits) for perfticks
 * Use ARM timer to force immediate interrupt
 * Use cycle counter for cycles()
 */
```

⟨*kernel basic includes* 49a⟩

```
#include "io.h"
#include "ureg.h"
#include "arm.h"
```

⟨*constant* SYSTIMERS*(arm)* 43b⟩
⟨*constant* ARMTIMER*(arm)* 43c⟩

```
//TODO factorize in mem.h
```

⟨*constant* ARMLOCAL*(arm)* 536e⟩

⟨*enum* _anon_*(*(time/arm/clock.c)(arm)*)* 1010d⟩

```
typedef struct Systimers Systimers;
typedef struct Armtimer Armtimer;
```

⟨*struct* Systimers*(arm)* 246c⟩

⟨*struct* Armtimer*(arm)* 658b⟩

⟨*enum* _anon_*(*(time/arm/clock.c)2(arm)*)* 1011a⟩

⟨*function* clockintr*(arm)* 246d⟩

⟨*function* localclockintr*(arm)* 547e⟩

⟨*function* `clockshutdown`*(arm)* 630b⟩

⟨*function* `clockinit`*(arm)* 280⟩

⟨*function* `arch_timerset`*(arm)* 251a⟩

⟨*function* `clock_arch_fastticks`*(arm)* 246a⟩

⟨*function* `arch_perfticks`*(arm)* 658a⟩

⟨*function* `armtimerset`*(arm)* 948a⟩

⟨*function* `arch_us`*(arm)* 245f⟩

⟨*function* `clock_arch_microdelay`*(arm)* 254d⟩

⟨*function* `clock_arch_delay`*(arm)* 254e⟩

Uses `Armtimer` 658b and `Systimers` 246c.

time/arm/time_raspi1.s

⟨`time/arm/time_raspi1.s` 1012a⟩≡

```
#include "mem.h"
#include "arm.h"
```

⟨*function* `arch_lcycles`*(raspberry pi1)(arm)* 508g⟩

time/arm/time_raspi2.s

⟨*instruction MRRC(arm)* 1012b⟩≡ (890a)

```
#define MRRC(coproc, op, rd, rn, crm) \
    WORD $(0xec500000|(rn)<<16|(rd)<<12|(coproc)<<8|(op)<<4|(crm))
```

⟨`time/arm/time_raspi2.s` 1012c⟩≡

```
#include "mem.h"
#include "arm.h"
#include "arminstr.ha"
```

⟨*function* `arch_lcycles`*(raspberry pi2)(arm)* 547f⟩

F.14 syscalls/arm/

syscalls/arm/syscall.c

⟨`syscalls/arm/syscall.c` 1012d⟩≡

```
⟨kernel basic includes 49a⟩
#include "../port/systab.h"
#include <tos.h>
#include "ureg.h"
#include "arm.h"

typedef struct NFrame NFrame;
⟨struct NFrame(arm) 451b⟩
```

⟨*function* `arch__noted`*(arm)* 454⟩

⟨*function* `arch__notify`*(arm)* 452⟩

⟨*function* `arch__syscall`*(arm)* 163e⟩

⟨*function* `arch_execregs`*(arm)* 148a⟩

⟨*function* `arch_forkchild`*(arm)* 135e⟩

Uses `NFrame` 451b.

Glossary

```
UP   = User Process
PC   = Program Counter
SPL  = Set Priority Level
PD   = Page Directory
PT   = Page Table
PDB  = Page Directory Base
PDE  = Page Directory Entry
PTE  = Page Table Entry
PPN  = Physical Page Number?
EDF  = Earliest Deadline First
RPC  = Remote Procedure Call
EGRP = Environment GRouP
FGRP = File descriptor GRouP (channels)
PGRP = Process GRouP (namespace)
QID  = uniQue IDentifier (uid is taken for user id)
TOD  = Time Of Day
UART = Universal Asynchronous Receive/Transmitter

(ARM)
ARM = Acorn Risc Machines
CPSR = Current Program Status Register
SPSR = Saved Program Status Register
MCR = Move to Coprocessor from ARM Register
MRC = Move to ARM Register from Coprocessor
TTB = Translation Table Base
TLB = Translation Lookup Table
AP  = Access Permission
DAC = Domain Access Control
RFE = Return From Exception
IRQ  = Interrupt Request

(x86)
IRQ  = Interrupt Request
INTR = INTeRrupt (not INT, probably to avoid ambiguity with integer)
VNO  = Vector Number (interrupt)
ISR  = Interrupt Service Routine
NMI  = Non Maskable Interrupts (hardware error)
AP   = Application Processor
RMAP = RAM map
UMB  = Upper Memory Block
```

```
GDT  = Global Descriptor Table
VPT  = Virtual Page Table
APIC = Advanded Programmable Interrupt Controller (a next gen 8259)
ACPI = Advanced Configuration & Power Interface
TSC  = Time Stamp Counter
```

Indexes

Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. This index is generated automatically.

Aaccess: 305b, 305f, 338c, 339d
Abind: 349a, 349c, 349d
Abort-1667: 982
Abort-2301: 956
accelerated-928: 716, 716
AccessFlags: 305b
accounttime(): 657a, 657b
Acmderr-1682: 982
Acreate: 305b, 305f, 332b, 333a, 333d, 598b
Active: 65d, 65e
active: 65e, 131b, 237b, 237c, 274c, 275a, 437c, 568c, 627f, 628f, 629b, 659a, 659b, 877b
Active-2309: 956
Active.cpus: 65d
Active.exiting: 65d
Active.ispanic: 628a
Active.main_reached_sched: 65d
Active.rebooting: 629a
addbootfile(): 427a
addbroken(): 483c, 483d
addclock0link(): 253f, 260d, 383b, 506a, 636a, 676d, 719, 1001e
addconf(): 997a, 997c
addelem(): 326a, 328f, 333d
addhcitype(): 911a, 953b
addlist(): 424a, 424b, 427a
addphysseg(): 557a
addr(): 698b, 698b
addr2check: 557d, 863b
addrootdir(): 424b, 424c
adec(): 971d
adjustblock(): 444a
Admitted: 564g, 565b, 566f, 566g, 568c
ainc(): 971c
alarmkproc(): 258e, 289a
alarmr-2608: 258d, 258e, 259b
Alarms: 701b, 701b
alarms-2607: 257a, 257e, 258e, 259b
Alarms (typedef): 701b
Allcc-2582: 521b, 524c, 528a

Allexc-2581: 521b
allocb(): 401b, 408, 412, 434a, 436b, 439b, 441a, 441b, 442, 443b, 746b, 931b, 944b, 949b, 950
alloctag(): 746b, 746b
Alltraps-2579: 521b, 523c
Alt-966: 376a, 376b, 377a, 378d
Alt0-2189: 961d
Alt1-2190: 961d
Alt2-2191: 961d
Alt3-2192: 961d
Alt4-2193: 961d
Alt5-2194: 961d, 964b
Altgr-970: 376a, 377c, 378d, 380c
Amount: 349a, 349c, 350a, 365a
anyhigher(): 236c, 236d, 635c, 877b
anyready(): 221f, 231b, 236c
Aopen: 302a, 305b, 305f, 324c, 365a, 827c
AP: 179f, 179j, 535g
archdir-854: 977e, 978a, 978c, 978d, 979a, 979c
archether(): 994a
archreboot(): 628d, 629b, 1001b
archreset(): 1001a
Arch_Cpu: 64a
Arch_FPsave: 973c
Arch_Proc: 66
arch_cmpswap(): 972b
arch_coherence: 475b, 476b, 476c, 477a, 519c, 520b, 534a, 534b, 534c, 534d, 534e, 534f, 534g, 562a, 611, 891c, 894b, 894c, 963d, 965a, 1001b
arch_countpagerefs: 559a
arch_cycles: 508c
arch_delay: 131b, 170c, 275b, 436b, 544c, 628f, 629b, 643a, 644f, 646b, 649a, 964b, 984e, 985c, 986
arch_dumpstack: 170c, 234b, 438a, 519b, 642c, 646b
arch_exit: 237c, 437c, 627c, 642c
arch_fastticks: 245f, 249b, 251a, 252, 260d, 261b, 262, 263a, 263c, 265c, 266c, 587, 939b
arch_idlehands(): 225
arch_intrenable: 175e
arch_isaconfig:
arch_microdelay: 254e, 392d, 946a, 962c, 984b
arch_xdec(): 972b
arch_xinc(): 972b
arch__confinit(): 272b, 276d
arch__cpu0init(): 272b, 274c
arch__cpuinit(): 274c, 274d, 544a
arch__kexit(): 163e, 168b, 169
arch__notify(): 450g, 451a, 452
arch__syscall(): 163e
arch__trap(): 168b
arch__userinit(): 272b, 287d
Aremove: 305b, 305f, 335d
Arg1-1624: 982, 986

Arg2-1622: 982
ARMLOCAL-2446: 536e, 541c, 544a
ARMLOCAL-2649: 536e, 547c
Armtimer: 658b, 1011b
ARMTIMER-2648: 43c, 280, 630b, 658a, 948a
Armtimer.count: 658b
Armtimer.ctl: 658b
Armtimer.irq: 658b
Armtimer.irqack: 658b
Armtimer.load: 658b
Armtimer.maskedirq: 658b
Armtimer.predivider: 658b
Armtimer.reload: 658b
Armtimer.val: 658b
Armtimer (typedef): 1011b
armtimerset(): 947b, 948a, 948b
assert: 170a, 249a, 249b, 249d, 252, 272b, 283c, 289a, 328f, 387a, 517b, 528a, 544d, 561f, 568c, 666b, 768, 853f, 911b, 944b, 948c, 949b, 950, 951b, 951c, 952a, 984c, 986, 988b
Atag: 996g, 999b
Atag.size: 996g
Atag.tag: 996g
Atag (typedef): 999b
AtagCmdline-2445: 996h, 998a
AtagCore-2443: 996h, 998a
ataginit(): 998a
AtagMem-2444: 996h, 998a
AtagNone-2442: 996h, 998a
Atodir: 289a, 305b, 336a, 336b, 457
attachimage(): 194e
Autocmd12-1679: 982
Autocmd23-1680: 982
auxpage(): 816c, 817
AUXREGS-2181: 43g, 962a, 967c, 967d
balancetime: 242a, 242b
BALLOC: 402a, 402b, 403, 405, 406, 407, 410c, 411a, 414b, 416, 435c, 440b, 442
baud(): 965c, 968b
Bdead-781: 438a, 438b, 811
BFREE: 434c
bindmount(): 348d, 349a, 363g
BINTR: 434c, 437c, 438a
Bipck: 434c
bits(): 965d, 968b
Bits7-2215: 961e, 965d
Bits8-2216: 961e, 964b, 965d
Bitsmask-2214: 961e, 965d
bl2mem(): 413, 445, 746b
BLEN: 401a, 402a, 402b, 403, 405, 406, 407, 408, 410c, 411a, 412, 413, 414b, 434b, 435b, 439b, 440a, 441a, 441b, 442, 443a, 443b, 444a, 444b, 445, 746b, 932a, 945
Blkcnten-1681: 982, 983a

Blksize-1746: 827c, 827c
Blksizecnt-1623: 982, 988a
Block: 701b, 701b
Block (typedef): 701b
blockalloclen(): 410b, 440b
BlockFlags: 434c
blocklen(): 410b, 440a, 441a, 443a, 746b
Bogus: 364a, 588, 804b
Bogus.authchan: 804b
Bogus.chan: 804b
Bogus.flags: 804b
Bogus.spec: 804b
Bool: 853i
bool (typedef): 853i
bool_ushort (typedef):
BOOTARGS-2436: 996a, 997b, 998a
BOOTARGSLEN-2437: 996b, 997b, 998a
bootdata-834: 426, 426
bootdir-833: 426, 426
bootlist-835: 425a, 425b, 426, 427a
Boottimeout-1639: 982
Bpktck: 434c
Break-964: 376a, 377c, 380c
Broken: 154d, 482d, 483a, 483b, 483d, 484b, 486c
broken: 483b, 483d, 483e, 484a
Broken.n: 483a
Broken.p: 483a
BSEG: 118c, 123a, 124a, 645b, 876d
Btcpck: 434c
Budpck: 434c
buf: 698a, 698b
Buffered: 529h, 529j, 535f
Bufread-1697: 982
Bufwrite-1698: 982
buttonmap-955: 719, 719
byte (typedef):
c: 698b
C-2587: 525b, 525c
Cache: 816a, 817
cache-762: 817, 817
Cache.hash: 817
Cache.head: 817
Cache.pgno: 817
Cache.tail: 817
Cache (typedef): 816a
Cachectl: 554f
Cached: 529h, 529j, 535f
cachedel(): 192b, 207d
cachepage(): 190e, 197d, 199d, 207d, 817

canacquire(): 476a, 476c, 477a
canflush(): 208d, 213f
canlock(): 78f, 83a, 110d, 111b, 112a, 199d, 222c, 392d, 455, 719
canpage(): 213f, 214a
canqlock(): 83a, 202b, 208d, 211d, 258e, 600b, 601a, 876d
canrlock(): 86
Caps-967: 376a, 378d
Card2host-1678: 982, 983a
Cardinsert-1692: 982, 986
Cardintr-1691: 982, 985c
cas32(): 474c, 474d
catio(): 827c, 827c
Cb: 957b
Cb.destad: 957b
Cb.nextconbk: 957b
Cb.reserved: 957b
Cb.sourcead: 957b
Cb.stride: 957b
Cb.ti: 957b
Cb.txfrlen: 957b
Cb (typedef): 960
Cbaderr-1686: 982
Cbalign-2287: 956
CCACHE: 588, 589, 745d, 746a, 746b
CCEXEC: 321b, 321c, 321d, 591f, 742b, 762c
cchain(): 817, 817
cclone(): 289a, 297a, 332a, 359f, 361b, 457
cclose: 156c, 202b, 216c, 297a, 299b, 302a, 303a, 303b, 305f, 313b, 314c, 316, 317, 318a, 328f, 331a, 331b, 331c, 332b, 333d, 335d, 336a, 338c, 339a, 341, 346a, 347e, 349a, 354d, 355, 356e, 357a, 357b, 359e, 359f, 360b, 361b, 364b, 365a, 365d, 368b, 371, 429b, 457, 589, 591e, 591f, 598b, 611, 618d, 619a, 742b, 758d, 760, 762c, 827c
ccloseq(): 319b, 320a
Ccrcerr-1688: 982
cdev(): 817, 817
Cenderr-1687: 982
CfgMask-2454: 1000c, 1001b, 1001c, 1001d
CfgReset-2455: 1000c, 1001b, 1001c
cfgstr-1772: 827c, 827c
CFREE: 294b, 296, 304b, 320a
Chan: 701b, 701b
Chan (typedef): 701b
Chanalloc: 295b, 295d
chanalloc: 295d, 937a
chanalloc(): 295d, 937a
Chanalloc.fid: 295b
Chanalloc.free: 295b
Chanalloc.list: 295b
chandebug: 808b, 808b
chandevinit(): 289a, 289b

chandevreset(): 272b, 286a
chandevshutdown(): 628b, 629b
chandone(): 939a, 939b
ChanFb-2355: 386b, 893b
chanfree(): 296, 304b, 320c, 588, 746b
chanintr(): 940a, 947b
chanio(): 941c, 943, 944a, 944b
chanlog-2238: 940b, 941a, 941b
ChanMask-2357: 893b, 894b, 894c
ChannelFlag: 294b
chanpath(): 318a, 318b, 341, 746b
ChanProps-2356: 893b, 895a
chanrelease(): 937b, 944a, 944b
chansetup(): 937c, 944a, 944b
chanwait(): 939b, 941c
chan_cclose(): 304b
checkaddr: 557d, 863b
checkalarms(): 259a, 259b
checkb(): 438b
checkpagerefs(): 558c
checkpages(): 862
cinit(): 746b, 817
cistrncmp(): 991b
ckfaultstuck(): 647d, 648a
Client: 179a
ClkArm: 272b, 896a
ClkCore: 896a
clkdiv(): 984c, 985c, 986
ClkEmmc: 896a, 984e
Clken-1661: 982, 985c, 986
Clkfreq8mask-1656: 982, 984c
Clkfreq8shift-1655: 982, 984c
Clkfreqms2mask-1658: 982, 984c
Clkfreqms2shift-1657: 982, 984c
Clkgendiv-1659: 982, 985c, 986
Clkgenprog-1660: 982
ClkH264: 896a
Clkintlen-1663: 982, 985c, 986
ClkIsp: 896a
ClkPixel: 896a
ClkPwm: 896a
ClkSdram: 896a
Clkstable-1662: 982, 985c, 986
ClkUart: 896a
ClkV3d: 896a
clockinit(): 272b, 280, 544a
clockintr(): 246d
clockshutdown(): 629b, 630b
clock_arch_delay(): 254e

clock_arch_fastticks(): 246a
clock_arch_microdelay(): 254d
clog(): 940a, 941a, 941c
closeegrp(): 140a, 156c, 597d
closefgrp(): 139d, 156c, 299b, 619b
closepgrp(): 139e, 156c, 347e
closeproc(): 320a, 320c
closergrp(): 140b, 156c, 479e
Clr0-2196: 961d, 963b
Clrerrors-2323: 956
Clunkq: 319c, 319d
clunkq: 319d, 320a, 320b, 320c
Clunkq.head: 319c
Clunkq.l: 319c
Clunkq.nclosed: 319c
Clunkq.nqueued: 319c
Clunkq.q: 319c
Clunkq.r: 319c
Clunkq.tail: 319c
clunkwork(): 320b, 320c
CMaccelerated-930: 716, 716
CMaddress-2264: 906, 908a, 926
CMadmit-901: 565c, 602b, 603b
CMbuttonmap-950: 719, 719
CMclose-879: 602b, 603b, 620b
CMclosefiles-880: 602b, 603b, 620b
CMclrhalt-2258: 906, 908a, 926
CMcost-898: 565c, 602b, 603b
Cmdbuf: 701b, 701b
Cmdbuf (typedef): 701b
Cmddone-1696: 982, 986
CMdeadline-897: 565c, 602b, 603b
CMdeadlinenotes-900: 565c, 602b, 603b
CMdebug-2250: 906, 907c, 929
CMdebugep-2265: 906, 908a, 926
cmderror(): 595a, 595b, 716
CMdetach-2263: 906, 908a, 926
cmdinfo: 983a, 986
Cmdinhibit-1703: 982, 986
Cmdtab: 701b, 701b
Cmdtab (typedef): 701b
Cmdtm-1625: 982, 986
CMdump-2251: 906, 907c, 929
CMevent-904: 500a, 602b, 603b
CMexpel-903: 565c, 602b, 603b
CMextra-902: 565c, 602b, 603b
CMfixedpri-888: 220f, 602b, 603b
CMhalt-846: 626f, 626f, 627b
CMhang-881: 487c, 602b, 603b

CMhub-2254: 906, 908a, 926
CMhwaccel-931: 716, 716
CMhz-2260: 906, 908a, 926
CMinfo-2262: 906, 908a, 926
CMintellimouse-932: 716, 716
CMkill-882: 159c, 602b, 603b
CMlinear-933: 716, 716
CMmaxpkt-2256: 906, 908a, 926
CMname-2266: 906, 908a, 926
CMnew-2252: 906, 908a, 926
CMnewdev-2253: 906, 908a, 926
CMnohang-883: 487d, 602b, 603b
CMnoswap-884: 213e, 602b, 603b
CMntds-2257: 906, 908a, 926
cmount(): 349a, 350c
CMpanic-848: 626f, 626f, 627b
CMperiod-896: 565c, 602b, 603b
CMpollival-2259: 906, 908a, 926
CMpreset-2268: 906, 908a, 926
CMpri-887: 220e, 602b, 603b
CMprivate-885: 601i, 602b, 603b
CMprofile-886: 506d, 602b, 603b
CMps2-934: 716, 716
CMps2intellimouse-935: 716, 716
CMreboot-847: 626f, 626f, 627b
CMres-936: 716, 716
CMreset-937: 716, 716
CMsamplesz-2261: 906, 908a, 926
CMscrollswap-951: 719, 719
CMserial-938: 716, 716
CMSG: 302d, 427c, 431, 465b, 466a, 582c, 582d
CMspeed-2255: 906, 908a, 926
CMsporadic-899: 565c, 602b, 603b
CMstart-890: 485d, 602b, 603b
CMstartstop-891: 502c, 602b, 603b
CMstartsyscall-892: 491b, 602b, 603b
CMstop-893: 485c, 602b, 603b
CMswap-952: 719, 719
CMtmout-2267: 906, 908a, 926
CMtrace-895: 498c, 602b, 603b
CMwaitstop-894: 487g, 602b, 603b
CMwildcard-953: 719, 719
CMwired-889: 237f, 602b, 603b
cnodata(): 817, 817
CntEnable-2657: 280, 1011a
CntPrescaleMask-2656: 1011a
CntPrescaleShift-2655: 1011a
CntWidth16-2664: 1011a
CntWidth32-2665: 280, 1011a

Coarse: 178d, 186, 230b, 271c
Conblkad-2290: 956
concatblock(): 441a
concatblockcnt-771: 441a, 644d, 812d
condok(): 525c, 526c
Conf: 62a, 701b
conf: 62b, 133a, 189a, 205d, 208d, 211d, 213f, 215a, 216c, 237g, 240, 274c, 275a, 276d, 278a, 278b, 282, 283a, 283c, 284b, 285c, 437c, 459f, 460a, 471d, 488a, 559c, 566f, 568c, 604d, 608c, 610, 614d, 644e, 645a, 659a, 817, 876d, 998a
Conf.copymode: 188f
Conf.ialloc: 276b
Conf.mem: 62a
Conf.ncpu: 62a
Conf.nimage: 276b
Conf.npage: 276b
Conf.nproc: 62a
Conf.nswap: 205a
Conf.nswppo: 208a
Conf.pipeqsize: 459e
Conf.upages: 276b
Conf (typedef): 701b
Confdata: 768, 768
Confdata.cf: 768
Confdata.on: 768
Confdata.spec: 768
Confdata (typedef): 768
confegrp-787: 733d, 734
Config-2353: 893a
configs-1776: 827c, 827c
configure(): 768, 768
Confmem: 62c, 701b
Confmem.base: 62c
Confmem.kbase: 488c
Confmem.klimit: 488c
Confmem.limit: 62c
Confmem.npage: 62c
Confmem (typedef): 701b
confname: 63a, 289a, 996i, 997a
confstr-1771: 827c, 827c
confval: 63a, 289a, 997a
consactive(): 628f, 653b, 664a
consattach(): 395c, 696b
consclose(): 394b, 395c
consdebug: 652e, 652e, 652f, 652g
consdevtab: 395c
consdir-1797: 388b, 394a, 394c, 696b
consinit(): 393b, 395c
ConsKbd: 382a, 382b
ConsKbd.ctl: 396a

ConsKbd.ctlpoff: 627d
ConsKbd.ie: 382a
ConsKbd.ir: 382a
ConsKbd.istage: 382a
ConsKbd.iw: 382a
ConsKbd.line: 389c
ConsKbd.lockputc: 382a
ConsKbd.raw: 396a
ConsKbd.x: 389c
consopen(): 394a, 395c
consread(): 394c, 395c
consstat(): 395c, 696b
consuart: 654c, 676d, 968a
consumecnt-774: 403, 644d, 812d
conswalk(): 395c, 696b
conswrite(): 395a, 395c
Control0-1632: 982, 986
Control1-1633: 982, 984e, 985c, 986
Control2-1637: 982
COPEN: 294b, 396e, 429a, 433a, 461a, 463a, 463b, 500g, 618d, 651g, 652a, 676d, 719, 734, 742b, 746b, 758d, 759, 768, 827c, 920c, 922a
copen(): 746b, 817
copyblock(): 443b, 444a
copyblockcnt-773: 443b, 644d, 812d
copypage(): 189b, 189c, 199d
copypath(): 325b, 325d
Counter (typedef): 87d
Cpaccd16-2584: 521c, 522c
Cpaccnosimd-2583: 521c, 522c
CpACsmp: 539e
cpage(): 817, 817
CpAuxctl: 539d
CpCACHE: 529a
CpCACHEall: 530f
CpCACHEintr: 530e
CpCACHEinvd: 530e
CpCACHEinvi: 530e
CpCACHEinvu: 530e
CpCACHERANGEdwb: 529f
CpCACHERANGEdwbi: 529f
CpCACHERANGEinvd: 529f
CpCACHEse: 530f
CpCACHEsi: 530f
CpCACHEwait: 530f
CpCACHEwb: 530e
CpCACHEwbi: 530e
CpCdcache: 529b
CpChv: 40b
CpCicache: 529b

CpCLD: 537c
CpCLDcyc: 537d
CpCLDena: 537d
CpCLDenacyc: 537e
CpCLDenapmnc: 537e
CpCmmu: 40b
CpCONTROL: 39d, 522c
CpCPaccess: 522b, 522c
CpCpredict: 539a
CpCsbo: 539c
CpCsbz: 539c
CpCsw: 539a
CpCtre: 539b
CpCve: 539b
CpDAC: 39d
CpDFP: 518b, 520b, 522c
CpFAR: 39d
CpFP: 518b, 519a, 522c
CpFSR: 39d
CpFSRdata: 171e
CpFSRinst: 171e
cpgmove(): 817, 817
CpID: 535a, 547c
CpIDcsize: 548c
CpIDcssel: 548c
CpIDfeat: 535b, 547c
CpIDid: 535c
CpIDidct: 535b
CpIDmpid: 535c
CpMainctl: 40a
cprd(): 891b, 892a
cprdsc(): 522c, 547c, 892a
CpSC: 39c, 892a, 892b
CpSPM: 39d
CpSPMctl: 508e
CpSPMcyc: 508e
CpSPMperf: 508d
CpTIMER: 537a, 547c, 547e, 547g
CpTIMERphys: 537b, 547c, 547e, 547g
CpTIMERphysctl: 537b, 547c, 547e, 547g
CpTIMERphysval: 537b, 547g
CpTLB: 39d
CpTLBinv: 180i
CpTLBinvse: 180i
CpTLBinvu: 180h
CpTTB: 39d
CpTTB1: 536a
CpTTBs: 536b
CPU: 162a

Cpu: 64a, 701b
Cpu.cpuhz: 64d
Cpu.cpumhz: 64d
Cpu.cpuno: 64a
Cpu.cs: 658c
Cpu.cyclefreq: 508a
Cpu.flushmmu: 472b
Cpu.ilockdepth: 89c
Cpu.intr: 658c
Cpu.lastintr: 64a
Cpu.load: 656f
Cpu.perf: 656a
Cpu.pfault: 658c
Cpu.proc: 64a
Cpu.readied: 232c
Cpu.sched: 224e
Cpu.schedticks: 236a
Cpu.splpc: 74f
Cpu.spuriousintr: 658c
Cpu.syscall: 658c
Cpu.ticks: 64c
Cpu.tlbfault: 658c
Cpu.tlbpurge: 658c
Cpu (typedef): 701b
CPUS: 78c, 78g, 80a, 224a, 225, 232b, 232d, 237g, 239d, 239h, 240, 241b, 257e, 258a, 258e, 259b, 261b, 262, 274d, 392d, 471d, 503c, 503d, 504b, 504c, 504f, 634d, 659a, 659b, 702, 719
cpus: 65a, 274c, 540a
cputempread(): 980c, 980d
cputype2name(): 275c
cpwr(): 891c, 892b
cpwrsc(): 522c, 547c, 547e, 547g, 892b
cqid-1774: 827c, 827c
Crcchken-1670: 982, 983a
CRCLOSE: 322b, 322c, 322d, 734, 762a, 762c
cread(): 746b, 817
createdir(): 361a, 361b
Cs-2289: 956, 958d
ctail(): 817, 817
ctldata(): 945, 949b
CTLQ-1852: 676d, 676d
ctlread(): 922b, 924b
CTLS-1851: 676d, 676d
ctltrans(): 944b, 950
ctlwrite(): 930, 931a
Ctoerr-1689: 982, 986
Ctrl-965: 376a, 376b, 377a, 377c, 378d, 380c
CtsFlow-2224: 961e, 966c
cunmount(): 365a, 365d
cupdate(): 746b, 817

curs: 719, 719
cursor: 719, 719
Cursortocursor(): 719, 719
cvlist: 709, 709
cvlist.ld: 709
cvlist.si: 709
cvlist.so: 709
cwrite(): 746b, 817
DAC: 179h
Data-1630: 982, 988b
data2txt(): 196d, 196e
Datactive-1701: 982
datadone(): 984d, 986, 988b
Datadone-1695: 982, 984d, 985b, 986, 988b
Datatomask-1654: 982
Datatoshift-1653: 982, 985c, 986
Datinhibit-1702: 982, 986
DBG-1801: 698a, 698b
Dbg-2288: 956
DBG-765: 304b, 305f, 320a, 325b, 355, 356f, 357a, 357b, 808b
Dbgsel-1640: 982
Dcrcerr-1684: 982
ddprint: 916, 918a, 948c, 949a, 949b, 950
Dead: 67c, 154a, 211d, 213f, 471d, 568c, 608c, 610, 611, 618d, 645a, 876d
Deadline: 566g
deadlineintr(): 568c, 568c
debug-1761: 827a, 827c
debug-2236: 936d, 952c
debug-2277: 909b, 929
Debug-2297: 956
Debug-2405: 647d, 1002b
debugging-778: 644d, 813a
deccnt(): 88h, 89a, 229c
decref(): 87c, 299b, 304b, 320a, 325c, 344c, 347e, 396e, 479e, 597d, 768, 827c, 913a
Del-983: 376a, 377c, 380c, 627c
delayedscheds: 228c, 568c, 662b
DELTAFD: 298c, 299a, 300c
deltree(): 827c, 827c
Denderr-1683: 982
deprint: 913a, 915a, 920c, 922a, 925b, 926
dequeueproc(): 222c, 225, 241b, 568c
Destad-2293: 956
Destdreq-2316: 956
Destignore-2315: 956
Destinc-2318: 956
Destwidth128-2317: 956
Dev: 701b, 701b
DEV-1612: 768, 768
Dev (typedef): 701b

devalloc(): 827c, 827c
devattach(): 297b, 460a, 622d, 676d, 696b, 710, 712, 719, 730c, 740b, 742b, 746b, 763b, 768, 827c, 919b, 978b
devbread(): 395c, 422b, 434a, 465a, 599, 605a, 623d, 676d, 710, 712, 719, 742a, 745c, 763a, 767b, 827b, 980a
devbwrite(): 395c, 422b, 434b, 599, 605a, 623d, 676d, 710, 712, 719, 742a, 745c, 763a, 767b, 827b, 980a
devclone(): 429a, 429b, 589
DevConf: 701b, 701b
DevConf (typedef): 701b
devcons_iprint(): 392b
devcons_panic(): 663b
devcons_pprint(): 393a
devcons_print(): 391b
devcons__assert(): 664c
devcreate(): 395c, 422b, 466b, 605a, 623d, 676d, 710, 712, 742a, 767b, 810, 827b, 932c, 980a
devdir(): 425a, 427c, 428, 431, 460b, 461b, 614d, 734, 742b, 768, 827c, 916
devdirread(): 394c, 425b, 432c, 464, 605d, 622e, 676d, 710, 712, 719, 734, 742b, 762b, 768, 827c, 924b, 979c
DEVDOTDOT: 425a, 428, 429b, 460b, 614d, 734, 742b, 768, 805, 827c, 916
devgen(): 394a, 394c, 425a, 428, 429b, 622e, 676d, 696b, 710, 712, 719, 730c, 732a, 732b, 732c, 978c, 978d, 979a, 979c
Devgen (typedef): 701b
devinit(): 422b, 599, 676d, 710, 712, 742a, 745c, 767b, 810, 827b
devletters-1578: 768, 768
DevLOG-1604: 768, 768
DevMASK-1606: 768
devno(): 297b, 297c, 348b, 364b, 372, 457
devopen(): 433a
devpermcheck(): 432d, 433a, 758d, 827c
Devport: 701b, 701b
Devport (typedef): 701b
devremove(): 395c, 422b, 466b, 605a, 623d, 676d, 710, 712, 719, 742a, 767b, 810, 827b, 932c, 980a
devreset(): 395c, 466b, 599, 605a, 623d, 710, 712, 742a, 763a, 810, 827b, 980a
devs-1579: 768, 768
DevSHIFT-1607: 768
devshutdown(): 395c, 422b, 466b, 599, 605a, 623d, 676d, 710, 712, 719, 742a, 745c, 763a, 767b, 810, 827b, 980a
devslock-1580: 768, 768
devstat(): 431, 676d, 696b, 710, 712, 719, 730c, 732b, 734, 740b, 742b, 763b, 768, 920a, 978d
devtab: 71c
devwalk(): 429b, 461a, 676d, 696b, 710, 712, 719, 730c, 732a, 734, 740b, 742b, 763b, 768, 827c, 919c, 978c
devwstat(): 395c, 422b, 599, 613, 623d, 710, 712, 719, 742a, 810, 827b, 932c, 980a
DirEntry: 853i, 853i
DirEntry (typedef): 853i
direntry_setname(): 337b, 338c, 371
dirfixed(): 371, 372
Dirlist: 423b, 730a
Dirlist.base: 423b
Dirlist.data: 423b
Dirlist.dir: 423b
Dirlist.mdir: 423b

Dirlist.ndir: 423b
Dirlist (typedef): 730a
Dirmode: 853i
Dirtab: 701b, 701b
Dirtab (typedef): 701b
disk-1768: 827c, 827c
Dl-2597: 568c, 568c
dma-2324: 957c
dmaaddr(): 386b, 895a, 941c, 957h
DmaChanEmmc: 955b, 988b
DmaChanSpiRx: 955b
DmaChanSpiTx: 955b
DmaD2M: 955b, 988b
DmaDevEmmc: 955b, 988b
DmaDevSpiRx: 955b
DmaDevSpiTx: 955b
dmadone(): 958c
DmaFlags: 955b
dmainterrupt(): 958d
dmaioaddr(): 957l
DmaM2D: 955b, 988b
DmaM2M: 955b
DMAREGS-2284: 43e, 957d
dmaregs-2325: 957d, 957d
DMDIR: 333a, 388b, 423c, 424a, 424b, 426, 431, 459d, 460b, 461b, 614d, 622b, 676d, 719, 734, 742b, 827c
DMEXCL: 651c, 768, 915b
DMEXEC: 337a
DMMOUNT: 337a, 431
DMREAD: 337a
DMWRITE: 337a
Dn-2576: 521b, 523c
dobreak(): 965b, 968b
Dom0: 179d, 186, 230b, 271c, 387a, 536i
domount(): 328f, 349d, 354b, 355
donothing(): 967b, 968b
Dontprint-2592: 567b
doread(): 746b, 746b
Down-980: 376a, 377c, 380c
dprint: 914b, 918a, 918b, 919a, 929, 939b, 941c, 946b, 951b, 951c, 953a
dprint-1762: 827a, 827c
DPRINT-2593: 567b, 568c
drawactive(): 719
Dreq-2306: 956
DSEG: 118c, 124a
DTO-1619: 981b, 985c, 986
Dtoerr-1685: 982, 985c
dumpaproc: 645a, 646a
dumpdregs(): 958b
dumpeps(): 913b, 929

dumplongs(): 646b, 647a
dumpstackwithureg(): 648b, 649a
dupattach(): 742a, 742b
dupclose(): 742a, 742b
dupdevtab: 742a
dupfgrp(): 137b, 139d, 287d, 298c
dupgen(): 742b, 742b
dupopen(): 742a, 742b
duppage(): 199b, 199d
dupread(): 742a, 742b
dupretries-1850: 199c, 199c, 199d
dupseg(): 121a, 135d
dupstat(): 742a, 742b
dupswap(): 207a, 207b
dupwalk(): 742a, 742b
dupwrite(): 742a, 742b
dwc-2235: 936c, 941a, 941c, 953a
Dwcregs: 898b, 899
Dwcregs.adpctl: 898b
Dwcregs.dtxfsiz: 898b
Dwcregs.gahbcfg: 898b
Dwcregs.gdfifocfg: 898b
Dwcregs.ggpio: 898b
Dwcregs.ghwcfg1: 898b
Dwcregs.ghwcfg2: 898b
Dwcregs.ghwcfg3: 898b
Dwcregs.ghwcfg4: 898b
Dwcregs.gi2cctl: 898b
Dwcregs.gintmsk: 898b
Dwcregs.gintsts: 898b
Dwcregs.glpmcfg: 898b
Dwcregs.gnptxfsiz: 898b
Dwcregs.gnptxsts: 898b
Dwcregs.gotgctl: 898b
Dwcregs.gotgint: 898b
Dwcregs.gpvndctl: 898b
Dwcregs.gpwrdn: 898b
Dwcregs.grstctl: 898b
Dwcregs.grxfsiz: 898b
Dwcregs.grxstsp: 898b
Dwcregs.grxstsr: 898b
Dwcregs.gsnpsid: 898b
Dwcregs.guid: 898b
Dwcregs.gusbcfg: 898b
Dwcregs.haint: 898b
Dwcregs.haintmsk: 898b
Dwcregs.hcfg: 898b
Dwcregs.hchan: 898b
Dwcregs.hfir: 898b

Dwcregs.hflbaddr: 898b
Dwcregs.hfnum: 898b
Dwcregs.hport0: 898b
Dwcregs.hptxfsiz: 898b
Dwcregs.hptxsts: 898b
Dwcregs.pad0: 898b
Dwcregs.pad1: 898b
Dwcregs.pad2: 898b
Dwcregs.pad3: 898b
Dwcregs.pcgcctl: 898b
Dwcregs.reserved0: 898b
Dwcregs.reserved1: 898b
Dwcregs (typedef): 899
Dwidth1-1649: 982
Dwidth4-1648: 982, 986
Ebadarg: 120b, 123c, 132d, 137a, 137c, 138a, 139c, 165b, 216b, 302b, 316, 349a, 385a, 385b, 431, 463a, 467c, 468c, 470, 471a, 472c, 477b, 478a, 478b, 488a, 488f, 563c, 591e, 592b, 608c, 625h, 664d, 664d, 667d, 676d, 712, 719, 768, 827c
Ebadchar: 308b, 664d, 664d
Ebadctl: 664d
Ebadexec: 664d, 664d
Ebadfd: 302d, 317, 664d, 664d
Ebadip: 664d, 664d
Ebadlen-2237: 936e, 936e, 944b
Ebadsharp: 348b, 664d, 664d
Ebadspec: 664d, 664d, 768
Ebadstat: 664d
Ebadtimectl: 260b, 260b, 265d, 267a
Ebadusefd: 302d, 664d, 664d, 712, 762c, 768, 924b, 931a
Ecache: 816a, 817
ecache-763: 817, 817
Ecache.free: 817
Ecache.head: 817
Ecache.total: 817
Ecache (typedef): 816a
Echange: 664d, 664d, 768
echo(): 383a, 384a
echoscreen(): 384b, 384d
echoserialoq(): 653c, 653d
Ecmdargs: 595b, 664d, 664d
Econinuse: 664d, 664d
Econrefused: 664d, 664d
Edetach: 664d, 664d, 924b, 930, 931a
Edf: 701b, 701b
Edf (typedef): 701b
edfadmit(): 565c, 568c
EdfFlags: 566g
edfinit(): 565c, 568c
edfinited: 568c, 568c

edflock(): 564c, 568c, 568c
edfnrun: 568a, 568c
edfready(): 565a, 568c
edfrecord(): 564g, 568c
edfrun(): 564c, 568c, 568c
edfschedlock: 568c, 568c
edfstop(): 483d, 564h, 565c, 566b, 568c
edfunlock(): 564c, 568c, 568c
edfyield(): 565b, 568c
Edirseek: 359c, 664d, 664d
Edoesnotexist-768: 331c, 665a, 665a
Eexist: 664d
Efilename: 348b, 664d, 664d
Egone-1777: 827c, 827c
Egreg: 216b, 394c, 395a, 425b, 453a, 457, 605d, 608c, 611, 622e, 623b, 664d, 664d, 710, 719, 730c, 762c, 949b, 950
Egrp: 701b, 701b
Egrp (typedef): 701b
Ehdr32sz-2416: 849b, 849b
Ehdr64sz-2419: 849b
Ehungup: 410a, 418a, 418b, 664d, 664d
Eintr: 664d
Einuse: 664d
Eio: 664d
Eioload: 197d, 664d, 664d
Eisdir: 664d
Eismtpt: 335d, 341, 664d, 664d
Eisstream: 458b, 664d, 664d
Elemlist: 309b, 808a
Elemlist.aname: 309b
Elemlist.elems: 309b
Elemlist.mustbedir: 309b
Elemlist.name: 309b
Elemlist.nelems: 309b
Elemlist.nerror: 309b
Elemlist.off: 309b
Elemlist.prefix: 309b
Elemlist (typedef): 808a
elfident-2422: 849b, 849b
Emask-906: 500d, 500f, 501b
emmc-1704: 984a, 984b, 984e, 985b, 985c, 986, 988b
emmccmd(): 986, 989a
emmcenable(): 985c, 989a
emmcinit(): 984e, 989a
emmcinquiry(): 985a, 989a
emmcio(): 988b, 989a
emmciosetup(): 988a, 989a
EMMCREGS-1615: 44c, 984b, 984d, 984e, 985a, 985b, 985c, 986, 988b
Emount: 350c, 362b, 664d, 664d

Emountrpc: 586, 587, 664d, 664d
Empty-2352: 893a, 894c
emptystr(): 462, 613, 670a, 768
enable(): 964b, 968b
Enable-2299: 956
Enable-2666: 547e, 547g, 1011a
Enabledelay-2230: 935, 951b, 951c
Enables-2206: 961e, 964b
End-2308: 956
End-979: 376a, 377c, 380c
Enegoff: 313b, 314c, 316, 664d, 664d
Enoattach: 363a, 363b, 664d, 664d
Enochild: 151a, 600c, 664d, 664d
Enocreate: 361b, 664d, 664d
Enodev: 664d, 664d, 676d
Enofd: 302a, 317, 332b, 457, 591f, 664d, 664d
Enomem: 664d
Enonexist: 664d
Enoreg: 605d, 608c, 664d, 664d
Enotconf: 664d, 664d, 920c, 948c
Enotdir: 664d
Enoversion: 588, 664d, 664d, 746b
Enovmem: 116b, 124a, 125a, 194e, 468c, 664d, 664d
envclose(): 599, 734
envcpy(): 138b, 140a, 598a
envcreate(): 599, 734
envdevtab: 599
envgen(): 734, 734
envgrp(): 733b, 734
envlookup(): 734, 734
envopen(): 599, 734
envread(): 599, 734
envremove(): 599, 734, 734
envstat(): 599, 734
envwalk(): 599, 734
envwrite(): 599, 734
envwriteable(): 733c, 734
epalloc(): 912a, 914b, 915a
epclose(): 949a
epctl(): 926, 930
epctls-2273: 908a, 926
epdataperm(): 915b, 916
Eperm: 664d
Epio: 936b, 953c
Epio.cb: 936b
Epio.lastpoll: 936b
Epio (typedef): 953c
epmax-2282: 909g, 910c, 912a, 912b, 913a, 913b, 916, 922b, 929
epopen(): 948c, 953a

epread(): 949b, 953a
Eprocdied: 352c, 486c, 600b, 605d, 608c, 611, 613, 618d, 619b, 664d, 664d
eps-2281: 909f, 910c, 912a, 912b, 913a
epslck-2280: 909e, 912a, 912b, 913a, 914a
eptrans(): 944a, 949b, 950
epwrite(): 950, 953a
eqchan(): 70e, 350c, 365d
eqchantdqid(): 71a, 354d, 371
eqqid(): 70c
Err-1690: 982, 984d, 985b, 986, 988b
ERRMAX: 152a, 152b, 170d, 182b, 199a, 311b, 311d, 418b, 429b, 451b, 452, 527b, 595a, 608c, 610, 666b, 666c, 667d, 667e
error:
Error-2302: 956
error0 (typedef):
error1 (typedef):
errorneg1 (typedef):
ERROR_0:
ERROR_1:
ERROR_NEG1: 185, 387b, 387c, 563b
error_if_not_dir(): 323b, 429b, 762b
Esbadstat: 664d, 664d, 746b
ESEG: 118c, 468c
Eshort: 432c, 582d, 608c, 664d, 664d, 719, 849b
Eshortstat: 461b, 462, 613, 664d, 664d, 676d, 746b, 768
Eshutdown: 664d, 664d, 758d
Esoverlap: 124e, 468c, 664d, 664d
Estalled: 664d, 664d, 941c, 944a, 944b
Etimedout: 664d, 664d
Etoobig: 608c, 610, 664d, 664d, 734, 768
Etoosmall: 664d
Eunion: 365d, 664d, 664d
Eunmount: 365d, 664d, 664d
Evalue: 701b, 701b
Evalue (typedef): 701b
eve: 132d, 157c, 287d, 425a, 427c, 428, 431, 432d, 460b, 461b, 462, 591a, 591b, 613, 614d, 625b, 734, 742b, 760, 768, 827c, 876d, 916
eventsavailable(): 500c, 501b
ewalk(): 328f, 330a, 358b
executeio(): 211d, 215a
Execvals: 701b, 701b
Execvals (typedef): 701b
exhausted(): 122e, 460a, 582d, 666c, 746b
Exrdfifocfg-1641: 982
Exrdfifoen-1642: 982
Extent: 701b, 701b
Extent (typedef): 701b
extentalloc(): 817, 817
extentfree(): 817, 817

Extfreq-1616: 981b, 984e
Extratime: 565c, 566g, 568c
F: 179g
false: 70e, 71a, 78f, 80d, 82c, 83a, 85d, 86, 98a, 104c, 119c, 120c, 135e, 152b, 157b, 157c, 158c, 159d, 163e, 168b, 172e, 177b, 180f, 189a, 189b, 197d, 213c, 213f, 214a, 219d, 220e, 230b, 252, 274c, 295e, 297b, 302a, 302c, 308a, 310b, 314c, 317, 318a, 328f, 330b, 332b, 335d, 336a, 338c, 339a, 339d, 340a, 340b, 348d, 349a, 353b, 354d, 364b, 365a, 370a, 378d, 381c, 389d, 390b, 391c, 392d, 396e, 397a, 397c, 429b, 435e, 449f, 450a, 450h, 457, 468c, 471d, 472a, 475b, 476a, 476c, 477a, 482a, 487d, 488d, 488f, 491c, 492e, 497, 498a, 500d, 522c, 523a, 591e, 591f, 601f, 605d, 618b, 620b, 627e, 641b, 670a, 672d, 719, 742b, 762c, 808b, 827c, 877b, 895a
fasthz: 265a, 265c, 266c, 267a
fastticks2us(): 263c, 264a
Fault: 179c, 181a, 186, 387a
fault(): 182b, 183
faultarm(): 171c, 172d, 173a, 182b
faulterror(): 197d, 199a
fbblank(): 387c
fbdefault(): 386b, 387b
Fbinfo: 43i, 897b
Fbinfo.base: 43i
Fbinfo.bpp: 43i
Fbinfo.pitch: 43i
Fbinfo.screensize: 43i
Fbinfo.xoffset: 43i
Fbinfo.xres: 43i
Fbinfo.xresvirtual: 43i
Fbinfo.yoffset: 43i
Fbinfo.yres: 43i
Fbinfo.yresvirtual: 43i
Fbinfo (typedef): 897b
fbinit(): 386b
fcallcount: 662c, 662e
Fcat-1739: 827c, 827c
Fclear-1742: 827c, 827c
fdclose(): 302c, 303b, 364c
Fdel-1743: 827c, 827c
Fdisk-1744: 827c, 827c
fdt (typedef):
fdtochan(): 216b, 302c, 302d, 313b, 314c, 316, 317, 318a, 339a, 340a, 364b, 591e, 591f, 742b, 762c
FEXT: 47e
Fgrp: 701b, 701b
Fgrp (typedef): 701b
filock(): 936f, 938b, 939b, 947b, 948b
findconf(): 996i, 997a
findfreefd(): 300a, 300b, 458a
findmount(): 328f, 354d, 355, 361a, 371
Finter-1740: 827c, 827c
fiq(): 562a
Fiqenable-2407: 561c, 561e, 561f
fiqintr(): 947b, 953a

FIQSTKTOP: 560e, 560f
fiunlock(): 936g, 938b, 939b, 947b, 948b
fixdotdotname(): 326a, 327a
fixfault(): 183, 185, 488f
Fmirror-1738: 827c, 827c
Fmt: 853a, 853a
Fmt (typedef): 853a
fmtl-1802: 700, 700
fmtrwdata(): 492e, 496, 497
Fmts (typedef): 853i
fmtuserstring(): 492e, 496, 498a
Fnone-1737: 827c
forceclosefgrp(): 319a, 319b
Forceirpt-1638: 982
FPactive: 513a, 516a, 517a, 517c, 527c
FPAIDNRM-2580: 521b, 521b
Fpart-1741: 827c, 827c
fpcfg(): 523c, 524b, 524c
fpclear(): 516a, 524d, 527c
Fpdex-2572: 521b, 521b
Fpenabled-2571: 521b, 523b
Fpex-2570: 521b
Fpexc-2567: 521a, 523a, 523b, 524d
FPIDNRM-2578: 521b, 521b
FPillegal: 513a, 514c, 515b, 517c, 518a, 526c
FPinactive: 513a, 516a, 517a, 517c, 527c
FPinit: 513a, 514a, 527c
fpinit(): 524b, 527c
Fpinst-2568: 521a
Fpinst2-2569: 521a
Fpmbc-2573: 521b, 524d
fpoff(): 515d, 517b, 523a
fpon(): 517b, 524c, 524d, 528a, 1001a
fponly(): 523b, 524b, 524c
fprd(): 517b, 519b, 522c, 523c, 524c, 524d, 528a
fprestore(): 528a
fprestreg(): 520b, 528a
fpsave(): 517b
FPSaveStatus: 513a
Fpscr-2564: 517b, 521a, 523c, 524c, 528a
Fpsid-2563: 521a, 522c, 523c
fpstuck(): 524e, 526c
fpuemu(): 526c, 527a
fpunoted(): 454, 518a
fpunotify(): 452, 517c
fpuprocrestore(): 516b, 516c
fpuprocsave(): 515e, 516a
fpusysprocsetup(): 515c, 515d
fpusysrfork(): 516d, 517a

fpusysrforkchild(): 515a, 515b
fpwr(): 519c, 523a, 523b, 523c, 524c, 524d, 528a
FRAMEBUFFER: 44a, 386b
free(): 104a, 115b, 116c, 124c, 150c, 154d, 155c, 201d, 297b, 298c, 299b, 300c, 305f, 308b, 310a, 310c, 311c, 325c, 326a, 328f, 331c, 332a, 344c, 346a, 347e, 352b, 352c, 356e, 364b, 364c, 369a, 369c, 371, 417, 429b, 438a, 460a, 462, 463b, 479e, 491c, 491e, 492c, 492e, 496, 497, 498a, 505e, 506d, 557c, 564h, 581b, 582a, 582d, 589, 591f, 594c, 597d, 600b, 608c, 613, 620b, 626c, 627b, 659a, 669b, 676d, 710, 712, 719, 734, 746b, 759, 760, 768, 792, 827c, 849b, 912a, 913a, 918a, 922a, 922b, 926, 929, 930, 949a, 967a, 997b
freeb(): 401b, 402b, 413, 414b, 416, 434a, 434b, 438a, 439a, 439b, 441b, 442, 443a, 444a, 444b, 445, 931b, 932a, 944b, 945, 949a, 949b, 950
freeblist(): 403, 405, 406, 418a, 420d, 439a, 441a, 443a, 444a, 581b, 746b
freebroken(): 211d, 484a
freept(): 115b, 116c
freetag(): 581a, 581b
fscache-761: 817, 817
Fsdev: 826, 827c
Fsdev.gone: 827c
Fsdev.inner: 827c
Fsdev.name: 827c
Fsdev.ndevs: 827c
Fsdev.perm: 827c
Fsdev.size: 827c
Fsdev.start: 827c
Fsdev.tree: 827c
Fsdev.type: 827c
Fsdev.vers: 827c
Fsdev (typedef): 826
fsdevtab: 827b, 827c
Fsel0-2185: 961d, 962b
fstree-1764: 827c, 827c
Full-2351: 893a, 894b
FuncMask-2186: 961d, 962b
Fz-2577: 521b, 523c
genage-1809: 208d, 214b, 214c
genclock-1810: 208d, 214b, 214c
gencount-1811: 208d, 214b, 214c
generrstr(): 667c, 667d
genrandom(): 635c, 636a
gensum-1812: 208d, 214b, 214c
gentick(): 211d, 214b
GET2: 924a, 944b
getattrs(): 827c, 827c
getboardrev(): 286d, 286g
getc(): 967d, 968b
getclkrate(): 896b, 984e
getconfenv(): 733d, 997b
getcputemp(): 897a, 980c
getdev(): 827c, 827c
getep(): 912b, 913b, 916, 920c, 922a, 922b, 924b, 929, 930, 931a

getethermac(): 289a, 895b
getfirmware(): 286d, 287a
getncpus(): 540e
getnewport(): 768, 768
getpcsp(): 648c, 648d
getpgcolor: 554d
getpower(): 563b
getramsize(): 275d, 276d
gettree(): 827c, 827c
GiB: 48b
gpioin(): 963c
gpioout(): 963b, 969a
gpiopull(): 962c, 962d, 962e, 963a
gpiopulldown(): 963a
gpiopulloff(): 962d, 964b
gpiopullup(): 962e
GPIOREGS-2180: 44d, 962b, 962c, 963b, 963c
gpiosel(): 962b, 964b, 969a
greset(): 946a, 946b
growfd(): 300b, 300c, 317
growparse(): 310b, 310c
gscreen: 719
hasfreepages(): 211b
haswaitq(): 151c, 151d, 600b
havefp(): 522c, 523b, 524b, 524c
hciprobe(): 918a, 918b
hcis-2279: 909d, 913b, 918b, 919a, 932b
Hcitype: 907a, 933
Hcitype.reset: 907a
Hcitype.type: 907a
Hcitype (typedef): 933
hcitypes-2278: 909c, 911a, 918a, 918b
Hdrspc-780: 435e, 811
Hole: 701b, 701b
Hole (typedef): 701b
Home-973: 376a, 377c, 380c
Host2card-1677: 982, 983a
Hostchan: 898b, 898b, 899
Hostchan.hcchar: 898b
Hostchan.hcdma: 898b
Hostchan.hcdmab: 898b
Hostchan.hcint: 898b
Hostchan.hcintmsk: 898b
Hostchan.hcsplt: 898b
Hostchan.hctsiz: 898b
Hostchan.reserved: 898b
Hostchan (typedef): 899
hostdomain: 591d, 592b, 625b
hostdomainwrite(): 592b, 625c

hostownerwrite(): 132d, 625c
HOWMANY: 47f, 941c
HVECTORS: 39b, 271c, 278e
hzclock(): 236f, 252
hzsched(): 236c, 236f
Ialloc: 437a, 437b
ialloc: 437b, 437c, 438a, 644e
Ialloc.bytes: 437a
iallocb(): 407, 416, 437c
iallocsummary(): 644d, 644e
ibrk(): 123a, 123b, 467c
ihash: 193b, 194e, 201d
IHASHSIZE: 193b
ilock(): 79d, 95e, 97, 100b, 250b, 250c, 250e, 252, 253f, 260d, 261a, 261b, 262, 263a, 382c, 402a, 402b, 403, 405, 406, 407, 408, 410a, 412, 413, 414b, 416, 418a, 418b, 419b, 420d, 437c, 438a, 568c, 650e, 676a, 676d, 936f
ILock (typedef): 79a
Imagealloc: 193b, 194b
imagealloc-1803: 194b, 194e, 201d, 202b, 285c, 285d
Imagealloc.free: 202a
Imagealloc.freechan: 193b
Imagealloc.ireclaim: 202a
Imagealloc.nfreechan: 193b
Imagealloc.szfreechan: 193b
imagechanreclaim(): 194e, 202b
imageinit(): 272b, 285c
imagmem: 102b, 102b, 205d, 276d, 643b
Imask-2667: 547c, 547e, 1011a
implement(): 522a, 523c
inccnt(): 88g, 88i
Incr-1747: 827c, 827c
incref(): 87b, 121c, 130a, 135b, 137b, 137d, 138b, 138c, 140c, 157c, 190e, 194e, 196c, 196e, 298c, 302d, 305f, 324f, 325b, 328f, 332a, 333d, 344b, 345b, 347d, 354b, 354d, 355, 356d, 357a, 396d, 449e, 490, 588, 589, 618d, 719, 746b, 758d, 768, 827c, 912b, 915a
Indexshift-1664: 982, 986
initcode: 287d
Initfreq-1617: 981b, 985c
Inner: 826, 827c
Inner.idev: 827c
Inner.iname: 827c
Inner.isize: 827c
Inner (typedef): 826
Input-2187: 961d
Ins-982: 376a, 377c, 380c
Int-2307: 956, 958d
int2flag(): 352c, 353a
intellimouse: 716, 716
Inten-2321: 956
interio(): 827c, 827c
Interrupt-1634: 982, 984d, 985b, 986, 988b

intrcount: 662c, 662d
intrcpushutdown(): 628d, 630a, 631a
Intregs: 175d, 1002c
INTREGS-2403: 43d, 176a, 279a
Intregs.ARMdisable: 175d
Intregs.ARMenable: 175d
Intregs.ARMpending: 175d
Intregs.FIQctl: 175d
Intregs.GPUdisable: 175d
Intregs.GPUenable: 175d
Intregs.GPUpending: 175d
Intregs (typedef): 1002c
intrshutdown(): 629b, 630a
intrsoff(): 278e, 279a, 630a
Intstatus-2298: 956
iolist-1807: 207d, 208b, 215a, 284b, 285a
ioptr-1808: 207d, 208c, 208d, 211d, 215a, 644c
IOSIZE: 45, 271c
iprint: 100c, 170c, 170e, 171a, 235b, 253e, 261b, 263a, 437c, 482b, 526c, 558c, 559c, 568c, 628f, 629b, 646b, 647a, 649a, 663b, 666c, 698b, 940a
iprintcanlock(): 392b, 392d
iprintlock-1779: 392b, 392c
iprintscreenputs: 392a, 392a, 392b
Irpten-1636: 982, 985b, 986, 988b
Irptmask-1635: 982, 985c
IRQ: 174b
irq(): 177a, 177b
Irq-2204: 961e, 963d
IRQaux: 174b
IRQbasic: 176c, 176c, 176d
IRQclock: 174b, 177b
IRQcntpns: 177b, 552
IRQcntps: 177b, 552
IRQDMA: 955a
IRQdma0: 174b
irqenable(): 176a
IRQfiq: 561b, 561c
IRQi2c: 174b
irqintr(): 948b
IRQlocal: 552, 552, 553b, 553c
IRQmmc: 174b
IRQspi: 174b
IRQtimer0: 174b
IRQtimer1: 174b
IRQtimer2: 174b
IRQtimer3: 174b, 174b
IRQtimerArm: 176c
IRQusb: 174b, 561b, 953a
Irstats: 661j, 662a

irstats-1806: 662a
Irstats.calls: 661j
Irstats.loops: 661j
Irstats.maxt: 661j
Irstats.ticks: 661j
ISAConf: 701b, 701b
ISAConf (typedef): 701b
ISCPOP-2590: 526a
Isdata-1668: 982, 983a, 986
isdotdot(): 326a, 326b, 330b, 357a
iseve(): 132d, 216b, 220e, 220f, 265b, 266b, 488a, 591b, 592b, 627b, 676d, 710, 712, 719, 733c, 760, 768
isfrog: 309a
isoverlap(): 468c, 469
isqtype(): 910d, 916, 920c, 922a, 924b, 930, 931a
Isread-1753: 827c, 827c
Istatus-2668: 1011a
isvalidaddr(): 647a, 647b
ISVFPOP-2591: 526b, 526c
Iswrite-1752: 827c, 827c
iunlock(): 80b, 95e, 97, 100c, 103b, 250b, 250c, 250e, 252, 253f, 260d, 261a, 261b, 262, 263a, 382c, 402a, 402b, 403, 405, 406, 407, 408, 410a, 411b, 412, 413, 414b, 416, 418a, 418b, 419b, 420d, 437c, 438a, 568c, 650e, 676a, 676d, 936g
Ixchken-1669: 982, 983a
ixsummary(): 642c, 644d
KB: 101, 283c, 867a
kbd-1778: 382b, 382b, 382c, 383a, 389d, 396b, 396c, 396d, 396e, 397a, 627e, 627f, 654a, 676a
kbdbuttons: 719, 719
kbdcr2nl(): 676a, 676d, 963d
kbdgetmap(): 385a, 712
kbdmouse: 397e, 398, 719, 719
kbdputc(): 378d, 382c, 641b, 672d
kbdputcclock(): 383a, 383b
kbdputmap(): 385b, 712
kbdputsc(): 378d, 710
kbdq: 384a, 388c, 388d, 389d, 396b, 397a, 676d, 964a
kbdqinit(): 388d
KbExt: 378b, 378d
kbinattach(): 710, 710
kbinbusy: 710, 710
kbinclose(): 710, 710
kbindevtab: 710
kbinlck: 710, 710
kbinopen(): 710, 710
kbinread(): 710, 710
kbinstat(): 710, 710
KbInt: 378b, 378d
kbintab: 710, 710
kbinwalk(): 710, 710
kbinwrite(): 710, 710

KBLINELEN-997: 712, 712
kbmapattach(): 712, 712
kbmapclose(): 712, 712
kbmapdevtab: 712
kbmapopen(): 712, 712
kbmapread(): 712, 712
kbmapstat(): 712, 712
kbmaptab: 712, 712
kbmapwalk(): 712, 712
kbmapwrite(): 712, 712
KbNscans: 378b, 378c
Kbscan: 701b, 701b
Kbscan (typedef): 701b
KbscanKind: 378b
kbscans: 378c, 378d
kbtab: 376b, 378d, 380b, 385a, 385b
kbtabaltgr: 377c, 378d, 385a, 385b
kbtabctrl: 378d, 385a, 385b
kbtabesc1: 380c, 381c, 385a, 385b
kbtabshift: 377a, 378d, 385a, 385b
kdebug: 641a, 641b, 641c, 641d, 641e, 642a
KF-959: 376a, 376a, 376b, 377a, 378d, 641b
KiB: 47g
kick(): 965a, 968b
kickpager(): 211c, 216b
killbig(): 211d, 642c, 876d
KImage: 701b, 701b
KImage (typedef): 701b
KMesg: 649b, 650a
kmesg: 650a, 650e, 651a
KMesg.buf: 649b
KMesg.lk: 649b
KMesg.n: 649b
kmesgputs(): 392b, 650c, 650d, 650e
KMESGSIZE: 650b
Kmouse-971: 376a, 397e, 398
kmousectl(): 716, 719
KNAMELEN: 289a, 598b, 605d, 853b
kprintinuse: 651f, 651g, 652a
kprintoq: 629b, 651e, 651g, 652a, 652b, 652c, 652d, 698b
kproc(): 157c, 211c, 289a, 320a, 636a
kproftimer: 660a, 660b
KQLock: 701b, 701b
Krw: 179b, 271c, 387a, 536i
KSEG0: 51b
KSEGM: 56e, 957h
ksetenv(): 289a, 598b, 849b
kstrcpy(): 307b, 429b, 666b, 669a, 734
kstrdup(): 130e, 132c, 132d, 133a, 135a, 144d, 157c, 287d, 310b, 345b, 582d, 592a, 613, 625h, 669b, 759, 768,

827c, 919a, 926
L1: 271a, 272b
L1AP: 179e, 271c, 387a, 536i
L1hi-1848: 180b, 181e, 531b, 531c
L1lo-1847: 180a, 181e, 531b, 531c
L1ptedramattrs-1845: 271c, 530a
L1sharable: 535e, 535f
L1SIZE: 540a, 540b
L1wralloc: 535e, 535f
L1X-1842: 178a, 186, 271c, 281c, 387a, 531b, 531c, 531d, 536i
L2: 271b, 271c
l2ap(): 179j
L2AP-1844: 179i, 186, 271c
l2be(): 143b
L2ptedramattrs-1846: 186, 271c, 530b
L2sharable: 535e, 535f
L2wralloc: 535e, 535f
L2X-1843: 178b, 186, 271c, 531g
Label: 226d, 701b
Label.pc: 226d
Label.sp: 226d
Label (typedef): 701b
Large: 178f
latin1(): 672d, 709
launchinit(): 539g, 540a
lck-1763: 827c, 827c
le2long(): 267a, 268c
le2vlong(): 267a, 267c
Left-977: 376a, 377c, 380c
legacytopctl(): 768, 768
Len-2575: 521b
Lev0-2197: 961d, 963c
lineq: 389a, 389b, 389d
lineqinit(): 272b, 389b
Lite-2322: 956
localclockintr(): 547e
Localctl-2650: 547a, 547c
Localintpending-2410: 553b, 553e
Localmboxint-2409: 553f, 631a
LOCALREGS-2404: 553b, 553g, 631a
Localtimerint-2408: 553b, 553d, 631a
Lock: 701b, 701b
lock(): 76d, 81f, 82c, 83e, 84b, 85b, 85d, 86, 87b, 87c, 111a, 115c, 116c, 128g, 129a, 129b, 130a, 131a, 131b, 150c, 151a, 152c, 154d, 155a, 156d, 176a, 185, 190e, 191a, 192a, 192b, 194e, 199d, 201b, 201d, 202b, 205c, 206d, 207a, 207d, 213f, 214a, 215a, 222b, 233e, 235a, 275a, 295e, 296, 298c, 300a, 302d, 303b, 313b, 314c, 316, 317, 320a, 320c, 368a, 393a, 455, 458a, 460a, 474e, 475a, 475b, 480b, 500g, 501a, 541b, 557a, 557c, 558c, 568c, 581b, 581c, 582d, 587, 600b, 600c, 616b, 618d, 619a, 619b, 628f, 700, 710, 719, 746b, 816c, 817, 978a
Lock (typedef): 701b
lockloop(): 76d, 646a

lockstats: 79d, 661b, 661c, 661d, 661e
Log: 701b
Log (typedef): 701b
logdump(): 939b, 941b, 941c
Logflag: 701b
Logflag (typedef): 701b
logstart(): 940d, 941c
lookpage(): 191a, 197d, 817
lookupcmd(): 595b, 620b, 627b, 716, 719, 827c, 926, 929
lookuptree(): 827c, 827c
lprint: 676d, 676d
LRESPROF: 113b, 506c, 506d, 507a, 507b
machon(): 272b, 275a, 544a
MAFTER: 343, 349a, 350c, 353a, 362c
Magichole: 94, 95e, 96b, 98b, 98c
MAILBOX-2345: 43h, 894b, 894c
main-2434(): 272b
mainmem: 102b, 102b, 103c, 104a, 104b, 104d, 105a, 105b, 205d, 276d, 436a, 643b
main_arch_exit(): 628d
main_arch_isaconfig(): 991b
malloc(): 103c, 176a, 201d, 253f, 281b, 285c, 298c, 300c, 352a, 409a, 409b, 460a, 500g, 506d, 558c, 568c, 582d, 608c, 626c, 669b, 676d, 733d, 746b, 768, 792, 849b, 946b, 948c, 967a
mallocalign(): 105a, 540a
MallocOffset-1814: 106a, 861b
mallocsummary(): 436b, 642c, 643b
mallocz(): 104d, 435e, 746b, 827c
Manager: 179a
MASK: 47c, 521b, 522c, 523c, 526c, 544d, 890c
mattach(): 827b, 827c
Maxatomic-779: 401b, 416, 813b, 813b
MAXCACHE-758: 817, 817
maxcache-764: 817, 817
Maxchans: 898a, 898b
MAXCONF: 63a, 63b, 997a, 997c
Maxconf-1756: 827c, 827c
Maxenvsize-786: 734, 734
Maxfpregs: 514i
MaxPeriod-2653: 251b, 251c
Maxretries-1759: 827c, 827c
MAXRPC-782: 582d, 812c
MAXRPC-825: 746b, 746b
Maxsteps-2594: 568c, 568c
MAXSYSARG: 163a
Maxtimerloops-2644: 253c, 253d, 253e
MB: 101, 102a, 276d, 277, 817, 867a
MBEFORE: 343, 349a, 353a
Mbox: 536c, 994c
Mbox.doorbell: 536c
Mbox.mbox1: 536c

Mbox.mbox2: 536c
Mbox.startcpu: 536c
Mbox (typedef): 994c
Mboxes: 536d, 994c
Mboxes.clr: 536d
Mboxes.set: 536d
Mboxes (typedef): 994c
Mboxregs-2447: 541c, 544a, 994c
Mbz: 178e
MCACHE: 343, 353a, 364b, 588
mclose(): 827b, 827c
mconfig(): 827c, 827c
MCREATE: 343, 350c, 353a, 361b, 362b
mdelctl(): 827c, 827c
mdeldev(): 827c, 827c
memrchr(): 311d, 668b
memsize: 56i, 276c, 276c, 277, 278a, 278b, 998a
mfreeseg(): 125b, 125c, 471a, 876d
mgen(): 827c, 827c
Mhead: 701b, 701b
Mhead (typedef): 701b
Mhz: 245d, 263c, 280, 874a, 981b, 984e
MiB: 48a, 72b, 180a, 180b, 271c, 387a, 534i
MicroFREQ-2670: 261a, 880b
Middle-969: 376a
MIN: 47d, 492e, 496
Minfirmdate-2432: 286d, 286f
Minfirmrev-2431: 286d, 286e
miniphysuart: 962a, 968b
miniuart-2228: 962a, 964a, 968a
MinPeriod-2654: 251b, 251d
misc_constants_portable: 867a
mk64fract(): 261a, 264c
mkqid(): 70d
MMASK: 349a
mmcinterrupt(): 985b
MMCSelect-1620: 981b, 986
mmuinit(): 271c
mmuinit1(): 272b, 281c, 544a
mmukmap(): 386b, 387a
mmul1empty(): 180f, 181e, 230b
mmul2empty(): 180f, 181a, 230b
Mnt: 701b, 701b
Mnt (typedef): 701b
Mntalloc: 580b, 580c
mntalloc: 580c, 581a, 581b, 581c, 582d, 746b
Mntalloc.id: 580b
Mntalloc.list: 580b
Mntalloc.mntfree: 580b

Mntalloc.nrpcfree: 580b
Mntalloc.nrpcused: 580b
Mntalloc.rpcfree: 580b
Mntalloc.tagmask: 580b
mntattach(): 588, 745c
mntauth(): 591f, 746b
Mntcache: 701b, 701b
Mntcache (typedef): 701b
mntchan(): 588, 746b, 746b
mntchk(): 589, 746b, 746b
mntclose(): 745c, 746b
mntclunk(): 746b, 746b
mntcreate(): 745c, 746b
mntdevtab: 745c
mntdirfix(): 746b, 746b
mntflushalloc(): 587, 746b
mntflushfree(): 587, 746b
mntfree(): 581b, 582a, 588, 589, 746b
mntgate(): 587, 746b
MNTHASH: 346d, 347e, 353b, 368a
MNTLOG: 346d, 346d
mntopen(): 745c, 746b
mntopencreate(): 746b, 746b
mntpntfree(): 581c, 582a
mntqrm(): 746b, 746b
mntralloc(): 588, 589, 746b, 746b
mntrdwr(): 746b, 746b
mntread(): 745c, 746b
mntremove(): 745c, 746b
mntreset(): 745c, 746b
Mntrpc: 701b, 701b
Mntrpc (typedef): 701b
mntrpcread(): 587, 746b
mntscan(): 352c, 353b
mntstat(): 745c, 746b
mntstats: 746b, 746b
mntversion(): 582d, 588, 591e, 746b
Mntwalk: 701b, 701b
mntwalk(): 589, 745c
Mntwalk (typedef): 701b
mntwrite(): 745c, 746b
mntwstat(): 745c, 746b
modemctl(): 966c
Modref: 108b
mopen(): 827b, 827c
MORDERMASK: 349a, 350c, 364d
Moribund: 152c, 153a, 154a, 228c, 516a
Mount: 701b, 701b
Mount (typedef): 701b

mountfix(): 359c, 371
MountFlags: 853i
mountfree(): 346a, 347e, 350c, 365d
MOUNTH: 346d, 350c, 354d, 365d
mountid-2606: 345b, 367b, 368a
mountio(): 586, 587
mountmux(): 587, 746b
mountrewind(): 360a, 370b
mountrock(): 369c, 371
mountrockread(): 359c, 370a
mountrpc(): 586, 588, 589, 746b
mouse: 719, 719
mouseaccelerate(): 716, 719
mouseattach(): 719, 719
mousechanged(): 719, 719
mouseclock(): 719, 719
mouseclose(): 719, 719
mousecreate(): 719, 719
mousectlmsg-939: 716, 716
mousectlmsg-954: 719, 719
mousectlqlock-929: 716, 716
mousedevgen(): 719, 719
mousedevtab: 719
mousedir-949: 719, 719
mousefromkbd(): 719, 719
mousehwaccel: 716, 716
Mouseinfo: 719, 719
Mouseinfo.acceleration: 719
Mouseinfo.dx: 719
Mouseinfo.dy: 719
Mouseinfo.lastcounter: 719
Mouseinfo.lastresize: 719
Mouseinfo.maxacc: 719
Mouseinfo.open: 719
Mouseinfo.qfull: 719
Mouseinfo.queue: 719
Mouseinfo.r: 719
Mouseinfo.redraw: 719
Mouseinfo.resize: 719
Mouseinfo.ri: 719
Mouseinfo.track: 719
Mouseinfo.wi: 719
Mouseinfo (typedef): 719
mouseinit(): 719, 719
mouseopen(): 719, 719
Mouseother: 706c
MousePS2: 706c, 716
mouseread(): 719, 719
mousereset(): 719, 719

mouseresize(): 719
Mouseserial: 706c
mouseshifted: 397c, 397d, 641c, 719
mousestat(): 719, 719
Mousestate: 719, 719
Mousestate.buttons: 719
Mousestate.counter: 719
Mousestate.msec: 719
Mousestate.xy: 719
Mousestate (typedef): 719
mouseswap-956: 719, 719
mousetime-958: 719, 719
mousetrack(): 719, 719
MouseType: 706c
mousetype: 716, 716
mousewalk(): 719, 719
mousewrite(): 719, 719
mousexy(): 719
mread(): 827b, 827c
MREPL: 350c, 362c, 853i
MS2NS: 244b, 255c
ms2tk(): 245b, 257e
msize(): 435e, 436a
mstat(): 827b, 827c
MuBaud-2227: 961e, 964b, 965c
MuCntl-2223: 961e, 964b, 964c, 966c
MuIer-2209: 961e, 964b, 964c, 965a
MuIir-2212: 961e, 964b
MuIo-2208: 961e, 963d, 965a, 967c, 967d
MuLcr-2213: 961e, 964b, 965d
MuLsr-2219: 961e, 963d, 965a, 967c, 967d
Multiblock-1676: 982, 983a
multitrans(): 943, 944a, 944b
MuMcr-2217: 961e, 966d
Mustexist-1755: 827c, 827c
muxclose(): 368b, 582a
Mvfr0-2566: 521a
Mvfr1-2565: 521a
mwalk(): 827b, 827c
mwrite(): 827b, 827c
N-2585: 525b, 525c
name2mode(): 910b, 926
name2speed(): 909j, 926
name2ttype(): 910a, 926
namec(): 289a, 302a, 305f, 332b, 335d, 336a, 338c, 339d, 349a, 365a, 457, 598b, 827c
nameerror(): 312a, 341
namelenerror(): 311c, 311d, 312a
Nbootfiles-829: 423a, 426
NBROKEN-2462: 483a, 483a, 483d, 483e

Nchan-2285: 956, 957c
nchanlog-2239: 940c, 940d, 941a, 941b
Ncmd-1608: 768
ncmdfield(): 594c, 596
nconf: 63c, 289a, 996i, 997a
NCONT: 454, 853i
Nctlqid: 676d
Ndataqid: 676d
NDebug: 120b, 165b, 170e, 173b, 173d, 182b, 199a, 449a, 452, 454, 482a, 482b, 526c, 527b, 563c
NDev-1605: 768, 768
Ndevs-1757: 827c, 827c
NDFLT: 454
needpages(): 211d, 213a
nelem: 117d, 128g, 129a, 129b, 276d, 282, 283c, 394a, 394c, 488a, 521d, 557a, 614d, 620a, 620b, 622e, 627b, 672d, 676d, 696b, 698b, 710, 716, 719, 732a, 732b, 732c, 768, 827c, 853i, 886a, 909j, 910a, 910b, 911b, 916, 926, 929, 986, 998b
NERR: 165d, 666a, 666b
NETID: 460b, 676d
NETQID: 460a, 460b, 676d
NETTYPE: 461a, 461b, 462, 463a, 463b, 464, 465a, 465b, 466a, 676d
Nevents-905: 500d, 500e, 500f, 500g, 501b
newchan(): 295e, 297b, 429a
newdev(): 914b, 919a, 926
newdevep(): 915a, 926
newfd(): 300a, 302a, 317, 332b, 591f
newfd2(): 457, 458a
newmhead(): 344b, 350c, 368a
newmount(): 345b, 350c, 362c, 368a
newpath(): 289a, 297b, 324f
newpgrp(): 137d, 139e, 157c, 287d, 347d
newproc(): 130a, 134b, 157c, 287d
newrgrp(): 138c, 140b, 287d, 479d
newseg(): 116a, 121d, 122b, 122c, 194e, 196e, 287d, 468c, 490
newswap(): 205c, 207d
newusbid(): 914a, 914b
NExit: 159c, 449a
Nextconbk-2296: 956
NEXTENT-760: 817, 817
nexterror: 121a, 135d, 215a, 255c, 302a, 305f, 311c, 311d, 313b, 314c, 316, 317, 332b, 333d, 335d, 338c, 339a, 341, 349a, 350c, 352c, 361b, 364b, 365a, 389d, 401b, 412, 413, 414b, 434a, 434b, 457, 462, 465b, 466a, 470, 476c, 477a, 486c, 488f, 500g, 582d, 587, 588, 589, 591e, 591f, 594c, 600b, 601a, 605d, 608c, 611, 613, 616b, 618d, 620b, 626c, 627b, 636b, 659a, 666b, 676d, 716, 719, 733d, 734, 746b, 758d, 759, 760, 762c, 768, 817, 827c, 911b, 913b, 916, 920c, 921, 922a, 922b, 924b, 926, 929, 930, 931a, 931b, 932a, 944a, 944b, 949b, 950, 988b, 997b
NFILE-759: 817, 817
Nfpctlregs: 514i, 519a
NFrame: 451b, 1012d
NFrame.arg0: 451b
NFrame.arg1: 451b

NFrame.ip: 451b
NFrame.msg: 451b
NFrame.old: 451b
NFrame.ureg: 451b
NFrame (typedef): 1012d
NFREECHAN: 194a, 201d, 285c
NHASH-757: 817, 817
Nhole: 282, 643a, 855
nilptr: 82c, 99, 123d, 130c, 185
NISAOPT: 674a, 991b
NKBFILE-996: 712, 712
NMASK: 580b, 580d, 746b
NNOTE: 450f, 455
No-972: 376a, 376b, 377a, 377c, 380c
Noaccess: 179b
noblockcnt-777: 414b, 812d
nonone(): 611, 613, 616a
noprint: 391b, 391b
noprocpanic(): 131a, 131b
Note: 701b, 701b
Note (typedef): 701b
NOTEID-914: 604a, 608c
noteidalloc: 140c, 449c, 449e
NoteKind: 449a
notempty(): 409c, 410a
now-2595: 568c, 568c
Npadlong-1813: 103c, 104a, 104b, 104d, 105a, 105b, 105c, 106a, 106b, 106c, 106d, 106e, 106f, 106g, 436a, 861b
NPart-1597: 768, 768
npath: 324e, 324f, 325b, 325c
NPIPEDIR-858: 461a, 464, 739b
Npriq: 219a, 220d, 221a, 241b, 564a
nrand(): 634d
nrdy: 222a, 222b, 222c, 568c, 644f, 657b
NRegs-2354: 893a, 894b
Nrootfiles-828: 423a, 423c
Nrq: 221a, 221b, 225, 564b, 644f
NRSTR: 449b, 453a, 454
ns2fastticks(): 249b, 264b
NSAVE: 449b, 454
Nscan-985: 376a, 376b, 377a, 377c, 380c, 385a, 385b
NSEG: 118c, 119b, 122e, 124e, 135d, 156a, 211d, 213f, 467c, 468c, 469, 470, 471d, 490, 559c, 605c, 605d, 862, 876d
Nstatqid: 676d
nsyscall: 165b, 492e, 886a, 886a
Ntrees-1758: 827c, 827c
ntrees-1766: 827c, 827c
NType-1593: 768, 768
Num-968: 376a, 376b, 377a, 378d
NUMSIZE: 265c, 394c, 504b, 504c, 605d, 633b, 659a, 676d, 712, 768, 867a

NUnit-1601: 768, 768
NUser: 258e, 449a, 452, 455, 465b, 466a, 608c
Nvec-2406: 278c, 278d
oargb-2440: 996e, 998b, 999a
oargblen-2441: 996f, 998b, 999a
oargc-2438: 996c, 998b, 999a
oargv-2439: 996d, 998b, 999a
OCEXEC: 302b, 305f, 321c, 321d, 333d, 742b, 759
OEXCL: 332b, 333d
OEXEC: 301c, 302b, 324c, 746a
Off-2199: 961d, 962d
offsetof: 95e, 96a, 98a, 998a
offsetof_name_direntry(): 337b, 338a, 341, 371
okaddr(): 119c, 120b, 452, 454
okay(): 969a, 988b
OkLed-2182: 961a, 969a
OK_0: 124c, 125b, 185, 387b
OK_1:
Onemicrosecond-2600: 568c
Onemillisecond-2601: 568c, 568c
OneRound-2603: 568c, 568c
Onesecond-2602: 568c, 568c
onswap: 206e, 206f, 207b, 216a
openmode(): 302a, 302b, 332b, 433a, 463a, 500g, 611, 719, 734, 742b, 746b, 758d, 759, 827c, 920c
Opmask-2384: 890c, 891a
Optional-1754: 827c, 827c
options-1614: 768, 768
optionsinit(): 998b
ORCLOSE: 302b, 322c, 322d, 758d
order-1800: 268b, 268b, 268c
ORDWR: 301c, 302d, 364b, 393a, 457, 591e, 591f, 746b, 758d, 827c, 911b, 914b
OREAD: 302b, 302d, 313b, 352a, 359f, 365a, 433a, 463a, 500g, 611, 618d, 719, 734, 758d, 827c, 853i, 911b, 915b, 920c, 931a, 944a
OTRUNC: 302b, 302d, 333d, 734, 758d, 827c
Output-2188: 961d, 969a
OWRITE: 314c, 393a, 598b, 611, 759, 911b, 915b, 920c, 924b, 944a
p: 698b
packblock(): 442
packetsize: 716, 716
padblock(): 439b
padblockcnt-770: 439b, 644d, 812d
Page: 701b, 701b
Page (typedef): 701b
pagechainhead(): 111a, 111b, 181d
pagechaintail(): 112a, 191b, 199d
pagedout: 197b, 197c, 197d, 208d, 488f, 559c, 855, 862, 866c
pageinit(): 272b, 283c
pageiocomp(): 215a, 215b
pagenumber(): 559c, 560a

PageOrSwap (typedef): 206b
pageout(): 208d, 211d
pagepte(): 207d, 208d
pager(): 211c, 211d
pagersummary(): 642c, 644c
Pagetable: 701b, 701b
Pagetable (typedef): 701b
pageunchain(): 110d, 191a, 199d, 816c
Palloc: 109b, 701b
palloc: 109b, 110d, 111a, 111b, 112a, 152c, 154a, 181d, 190e, 191a, 192a, 192b, 199d, 205d, 211a, 211b, 211d, 213a, 228c, 282, 283c, 284a, 558c, 560a, 644c, 816c
Palloc (typedef): 701b
Pallocmem: 701b, 701b
Pallocmem (typedef): 701b
panic:
panicking: 568c, 663a, 663b
parity(): 966b, 968b
parseboothdr: 849b
parsecmd(): 594c, 620b, 627b, 719, 768, 827c, 926, 929
parseconfig(): 827c, 827c
parseirq(): 768, 768
parsename(): 310b
parseport(): 768, 768
parsesize(): 768, 768
parsespec(): 768, 768
parseswitch(): 768, 768
parsetime(): 565c, 620a
parsetype(): 768, 768
PART-1610: 768, 768
PartLOG-1596: 768, 768
PartMASK-1598: 768
PartSHIFT-1599: 768
Password-2453: 1000c, 1001b, 1001c, 1001d
Path: 701b, 701b
path2dev(): 827c, 827c
path2devno(): 827c, 827c
path2treeno(): 827c, 827c
Path (typedef): 701b
pathclose(): 289a, 296, 325c, 325d, 328f, 331a, 331b, 331c, 333d, 354b
pathlast(): 338c, 339b
PATHMSLOP-767: 356b, 356c, 357a
PATHSLOP-766: 324f, 325a, 326a
Paused-2305: 956
Peek-2348: 893a
Perf: 656b, 701b
Perf.avg_inidle: 656b
Perf.avg_inintr: 656b
Perf.inidle: 656b
Perf.inintr: 656b

Perf.intrts: 656b
Perf.last: 656b
Perf.period: 656b
Perf (typedef): 701b
Permapshift-2310: 956
pexit: 151e, 158c, 159b, 194e, 199a, 320c, 452, 454, 876c
PF-961: 376a
Pgdown-981: 376a, 377c, 380c
pghash: 108c, 190e, 191a, 192a, 192b
PGHLOG: 108c
PGHSIZE: 108c
PGROUND: 120c, 120d, 123b, 276d, 283c, 468c, 471a
Pgrp: 701b, 701b
Pgrp (typedef): 701b
pgrpcpy(): 137d, 139e, 368a
pgrpid-2605: 347c, 347d
pgrpinsert(): 367e, 368a
pgrpnote(): 608c, 610
Pgup-975: 376a, 377c, 380c
PG_MOD: 108b, 185, 557d
PG_NEWCOL: 554f
PG_NOFLUSH: 554f, 555c
PG_NOTHING: 108b
PG_ONSWAP: 207d
PG_REF: 108b, 185, 189a, 208d, 557d
PG_TXTFLUSH: 554f, 555c, 555d, 555l, 555m, 866c
Phdr32sz-2417: 849b, 849b
Phdr64sz-2420: 849b
PHYSDRAM: 56d
Physseg: 701b, 701b
physseg-1804: 468c, 556f, 557a
Physseg (typedef): 701b
physseglock-1805: 556g, 557a
PhysUart: 701b, 701b
PhysUart (typedef): 701b
PID-913: 352c, 604a, 605d, 608c, 611, 613, 618d
pidalloc-2459: 128b, 130a
pidhash(): 128g, 130a
pidunhash(): 129a, 155a
pimagmem-1841: 102a, 102b
pimagpriv-1840: 102a, 102a
pio(): 197b, 197c, 197d
Pipe: 458c, 739a
Pipe.next: 458c
Pipe.path: 458c
Pipe.perm: 458c
Pipe.q: 458c
Pipe.qref: 458c
Pipe.ref: 458c

Pipe (typedef): 739a
Pipealloc: 459a, 459b
pipealloc: 459b, 460a
Pipealloc.path: 459a
pipeattach(): 460a, 466b
pipebread(): 465a, 466b
pipebwrite(): 466a, 466b
pipeclose(): 463b, 466b
pipedevtab: 466b
pipedir: 459d, 460a, 461a, 464
pipegen(): 460b, 461a, 464
pipeinit(): 459f, 466b
pipeopen(): 463a, 466b
piperead(): 464, 466b
pipestat(): 461b, 466b
pipewalk(): 461a, 466b
pipewrite(): 465b, 466b
pipewstat(): 462, 466b
plan9iniinit(): 997c, 998a
plock(): 100b, 101, 102a
pmainmem-1839: 101, 102b
pmainpriv-1838: 101, 101
pnp(): 964a, 968b
poolprint(): 101, 102a, 103a
poolsummary(): 643b, 644a
poperror: 121a, 135d, 152b, 154c, 163e, 194e, 196d, 197d, 208d, 215a, 255c, 258e, 289a, 302a, 304b, 305f, 313b, 314c, 316, 317, 320c, 330a, 332b, 333d, 335d, 338c, 339a, 341, 349a, 350c, 352c, 359f, 361b, 364b, 364c, 365a, 371, 389d, 393a, 401b, 412, 413, 414b, 429b, 434a, 434b, 457, 462, 465b, 466a, 470, 476c, 477a, 486c, 488f, 500g, 526c, 582d, 587, 588, 589, 591e, 591f, 594c, 600b, 601b, 605d, 608c, 610, 611, 613, 616b, 618d, 620b, 626c, 627b, 629b, 634c, 636b, 659a, 666d, 676d, 716, 719, 733d, 734, 746b, 758d, 759, 760, 762a, 762c, 768, 792, 817, 827c, 911b, 913b, 916, 920c, 921, 922a, 922b, 924b, 926, 929, 930, 931a, 931b, 932a, 944a, 944b, 949b, 950, 988b, 997b
portcountpagerefs(): 558c, 559c
portenable(): 951b, 953a
postnote: 120b, 159c, 165b, 170e, 173b, 173d, 182b, 199a, 258e, 465b, 466a, 482a, 482b, 526c, 527b, 563c, 608c, 610
pow2(): 925a, 926
PowerCcp2tx: 562c
PowerI2c0: 562c
PowerI2c1: 562c
PowerI2c2: 562c
POWERREGS-2449: 43f, 1001b, 1001c, 1001d
PowerSd: 562c
PowerSpi: 562c
PowerUart0: 562c
PowerUart1: 562c
PowerUsb: 562c, 946b
Powerwait-2367: 563a, 893d
ppanic(): 101, 102a, 103b

pprint: 119c, 165b, 301b, 452, 454, 608c, 611
preempt(): 877b
preempts: 225, 662b
Prescaler-2651: 547b, 547c
prflush(): 663b, 664a
PriEdf: 564a, 564c, 568c
PriExtra: 564a, 568c
PriKproc: 157c, 219b
PriNormal: 132d, 219b, 219d, 220e, 220f, 592a, 635c
print: 57d, 78e, 80c, 80f, 82d, 88d, 89b, 89f, 97, 98c, 131c, 165d, 183, 199d, 211d, 223b, 234a, 234b, 237c, 256a, 272b, 275b, 282, 283c, 286a, 286d, 300c, 305f, 308b, 311c, 319b, 354d, 355, 356c, 357b, 358f, 358g, 359a, 365c, 380b, 394c, 395a, 429b, 438b, 441b, 461a, 465b, 522c, 523c, 544c, 558c, 566f, 568c, 586, 589, 604d, 622e, 623b, 629b, 641b, 641c, 641d, 641e, 642a, 643a, 644a, 644c, 644d, 644e, 644f, 645a, 645b, 646a, 652g, 672d, 709, 746b, 768, 792, 817, 827c, 862, 876d, 912a, 913b, 914a, 918a, 918b, 920b, 925b, 926, 929, 932b, 939b, 941b, 941c, 951c, 958a, 958b, 984e, 985c, 986, 988b
Print-976: 376a, 377c, 380c
PRINTSIZE: 391a, 391b, 392b, 393a, 663b
Priority: 219b
PriRelease: 564a
PriRoot: 220g
Private: 100a, 860a
Private.lk: 100a
Private.msg: 100a
Private (typedef): 860a
Proc: 701b, 701b
Proc (typedef): 701b
procalarm(): 257b, 257e
Procalloc: 128c, 128d
procalloc-2458: 128d, 128g, 129a, 129b, 130a, 131a, 133a, 152c, 154a, 228c, 283a, 283b, 471d, 604b, 645a, 876d
Procalloc.arena: 128c
Procalloc.free: 128c
Procalloc.ht: 128e
procargs(): 605d, 618c
procattach(): 605a, 740b
procclose(): 605a, 617
proccmd-908: 603b, 620b
Procctl: 484d
procctl(): 452, 485b, 491c, 491e
procctlclosefiles(): 619b, 620b
procctlcloseone(): 619a, 619b
procctlmemio(): 488a, 488d, 488f
procctlreq(): 608c, 620b
procdevtab: 605a
procdir: 603a, 614d
procdump(): 131b, 642c, 645a
procfdprint(): 614c, 616b, 742b
procfds(): 605d, 616b
procflushseg(): 471b, 471c, 471d

procgen(): 605d, 611, 614d, 740b
procindex(): 129b, 614d
procinit(): 272b, 283a
procoffset(): 614a, 616b
procopen(): 605a, 611
procpriority(): 157c, 219d, 220a, 220e, 220f
procqidwidth(): 614b, 614c, 616b
procread(): 605a, 605d
Procseg: 118c
procstat(): 605a, 740b
Procstate: 67c
procstopped(): 486c, 486d
procstopwait(): 485c, 486c, 487g, 491b, 502c
proctab: 211d, 213f, 237g, 559c, 568c, 605d, 608c, 610, 611, 613, 614d
proctext(): 611, 618d
Proctime: 503a
proctrace: 499a, 499a, 499e, 499g, 499i, 499k, 500a, 500g, 501a, 568c
procwalk(): 605a, 740b
procwired(): 237f, 237g, 238b, 629b
procwrite(): 605a, 608c
procwstat(): 605a, 613
Proc_exitbig: 876b, 876c, 876d
Proc_exitme: 159a, 159b, 159c, 319a, 319b
Proc_nothing: 159d, 484d, 485b, 486c, 487g
Proc_stopme: 484d, 485b, 485c, 491c, 491e
Proc_traceme: 502a, 502b, 502c
Proc_tracesyscall: 491a, 491b, 491c, 491e
proc_dumpaproc(): 645b
proc_error(): 666b
proc_init(): 604d, 605a
proc_nexterror(): 667a
proc_pexit(): 152c
proc_postnote(): 455
proc_proctab(): 604b
proc_ready(): 232a
proc_sched(): 223c
proc_sleep(): 233e
proc_tsleep(): 255c
proc_wakeup(): 235a
producecnt-775: 407, 644d, 812d
profclock(): 506a, 506b
Prophdr: 894a, 897b
Prophdr.data: 894a
Prophdr.len: 894a
Prophdr.req: 894a
Prophdr.tag: 894a
Prophdr.tagbuflen: 894a
Prophdr.taglen: 894a
Prophdr (typedef): 897b

ps2mouse(): 716, 716
PsrDfiq: 454, 560d
PsrDirq: 39a, 454
PsrMabt: 161a, 168b, 171c, 172a, 279b
PsrMask: 38, 161b, 165h, 168b, 454, 646b, 647c
PsrMfiq: 560b, 560f
PsrMirq: 161a, 177a, 279b
PsrMsvc: 38, 646b
PsrMund: 161a, 170e, 279b
PsrMusr: 38, 165h, 168b
ptalloc(): 115a, 115c, 118a, 185
ptcpy(): 115c, 122a
PTE (typedef): 179l
PTEGLOBAL: 193a
PTERONLY: 184, 185, 189a
PTESIZE: 193a
PTEUNCACHED: 557d, 558a, 558b
PTEUSER: 855
PTEVALID: 184, 185, 189a, 557d
PTEWRITE: 184, 185, 186, 557d
ptflush(): 472c, 866c
PTR2UINT: 56g, 158b, 276d, 287d, 452, 454, 608b, 895a, 957h, 957l, 999a
PUD-2198: 961d, 962c
PUDclk0-2202: 961d, 962c
PUDclk1-2203: 961d
Pufv (typedef): 892c
pullblock(): 444b
Pulldown-2200: 961d, 963a
Pullup-2201: 961d, 962e
pullupblock(): 401a, 441b
pullupblockcnt-772: 441b, 644d, 812d
pullupqueue(): 401a, 746b
punlock(): 100c, 101, 102a
PUT2: 923
putc(): 967c, 968b
putep(): 913a, 913a, 913b, 916, 920c, 922a, 922b, 924b, 926, 929, 930, 931a
putimage(): 192a, 192b, 201c, 201d
putmhead(): 328f, 331a, 331b, 331c, 333d, 344c, 347e, 354c, 354d, 358c, 358e, 358f, 358g, 361c, 365d, 371
putpage(): 111a, 115b, 125c, 189b, 197d, 207d, 215a, 817
putseg(): 116c, 156a, 208d, 470, 490
putstrn(): 391b, 391c, 663b, 698a
putstrn0(): 390a, 390b, 391c
putswap(): 197d, 206d, 206e, 206f
Pvfu (typedef): 892c
pwait(): 150a, 150c
qaddlist(): 410b, 746b
Qargs-861: 602a, 603a, 605d, 608c, 611
Qbase-850: 977b, 977g
Qbintime-1783: 266a, 266b, 633a, 633b

Qboot-827: 423a, 423c, 425a, 425b, 426
qbread(): 412, 465a
qbwrite(): 401b, 414b, 466a
qbypass(): 409b
qcanread(): 389d, 396b, 420a
qclose(): 417, 418a, 463b, 676d
Qclosed: 399a, 405, 406, 409c, 410a, 414a, 414b, 418a, 418b, 419a, 419b
Qcoalesce: 399a, 413, 651g
Qconfig-838: 624e, 624f, 624g
Qcons-1781: 388a, 388b, 389d, 390a
Qconsctl-1782: 388a, 388b, 396d, 396e, 397a
qconsume(): 403, 676d
qcopy(): 408
qcopycnt-776: 408, 644d, 812d
Qcputime-1784: 504b, 633a, 633b
Qctl-1589: 768, 768
Qctl-1750: 827c, 827c
Qctl-2242: 906, 908b, 916, 922b, 924b, 930, 931a
Qctl-862: 602a, 603a, 608c, 611
Qcursor-945: 719, 719
Qdata-995: 712, 712
Qdata0-856: 459c, 459d, 460a, 460b, 461a, 461b, 463a, 463b, 464, 465a, 465b, 466a
Qdata1-857: 459c, 459d, 460b, 461a, 461b, 463a, 463b, 464, 465a, 465b, 466a
Qdir-1749: 827c, 827c
Qdir-1780: 388a, 388b, 394c
Qdir-2240: 906, 916
Qdir-826: 423a, 423c, 425a, 425b
Qdir-836: 622a, 622b, 622e
Qdir-849: 977b, 977e, 979c
Qdir-855: 459c, 459d, 460a, 461b, 462, 464
Qdir-859: 602a, 614d
Qdir-944: 719, 719
Qdir-992: 710, 710
Qdir-994: 712, 712
qdiscard(): 402b, 746b
Qdrivers-842: 626a, 626b, 626c
Qep0ctl-2245: 906, 916
Qep0dir-2243: 906, 910c, 910d, 916, 920c, 922a
Qep0dummy-2246: 906
Qep0io-2244: 906, 916
Qepctl-2249: 906, 916, 920c, 922a, 924b, 930, 931a
Qepdir-2247: 906, 916, 920c, 922a
Qepio-2248: 906
Qfd-863: 602a, 603a, 605d, 611
Qfirst-1751: 827c, 827c
Qflow: 399a, 402a, 402b, 403, 405, 406, 407, 411b, 414b, 418a, 421
qflush(): 420d, 676d
Qfpregs-864: 528b, 528c, 602a, 603a, 611
qfree(): 581c, 676d

qfull(): 421
qget(): 402a
qhangup(): 418b, 463b, 652a, 676d
Qhostdomain-840: 624h, 625a, 625b, 625c
Qhostowner-839: 624h, 625a, 625b, 625c
Qid: 853i, 853i
QID-1613: 768, 768
QID-2271: 907b, 916, 920c, 922a, 922b, 924b, 930, 931a
QID-911: 352b, 500g, 501a, 501b, 604a, 605d, 608c, 611, 613, 614d
qid2epidx(): 910c, 916, 920c, 922a, 922b, 924b, 930, 931a
Qid (typedef): 853i
qidpath-785: 757e, 758a, 759
Qidtype: 853i
qidvers-1767: 827c, 827c
qiomaxatomic: 463a, 651g, 676d, 813b, 813b
qisclosed(): 419a, 652c
qiwrite(): 416, 652c, 653d, 654a
Qkbd-993: 710, 710
Qkick: 399a, 414b
Qkmesg-1794: 650f, 650g, 651a
Qkprint-1795: 651b, 651c, 651g, 652a, 652b
Qkregs-865: 602a, 603a, 608a, 611
qlen(): 419c, 460b, 461b, 653b, 676d, 746b
qlock(): 81f, 116c, 119b, 121a, 123b, 135d, 156a, 156c, 197d, 257e, 258e, 320c, 347e, 352c, 359f, 360b, 369c, 370a, 389d, 412, 413, 414b, 452, 454, 455, 461a, 463a, 463b, 470, 483d, 483e, 484a, 486a, 486c, 487a, 490, 568c, 582d, 605d, 608c, 610, 611, 613, 616b, 619a, 636b, 676d, 716, 719, 758d, 759, 760, 762c, 768, 817, 862, 911b, 912a, 912b, 913a, 914a, 920c, 921, 926, 937a, 937b, 941c, 949b, 950
QLock (typedef): 701b
QlockStats: 661f, 661g
QlockStats.qlock: 661f
QlockStats.qlockq: 661f
QlockStats.rlock: 661f
QlockStats.rlockq: 661f
QlockStats.wlock: 661f
QlockStats.wlockq: 661f
Qmax-851: 977b, 977c, 977d, 977e, 978a
Qmem-866: 488a, 488d, 602a, 603a, 611
Qmouse-946: 719, 719
Qmousectl-948: 719, 719
Qmousein-947: 719, 719
Qmsg: 399a, 401b, 403, 412, 413, 416
qnoblock(): 388d, 389b, 420c, 651g, 676d
Qnote-867: 602a, 603a, 605d, 608c, 611
Qnoteid-868: 602a, 603a, 605d, 608c, 611
Qnotepg-869: 602a, 603a, 608c, 611
qnotfull(): 414a, 414b
Qns-870: 352a, 352b, 352c, 602a, 603a
Qnull-1785: 394c, 395a, 633a, 633b
qopen(): 388d, 389b, 409a, 460a, 582d, 651g, 676d, 964a

Qosversion-837: 624a, 624b, 624c
Qpart-1591: 768, 768
qpass(): 405
qpassnolim(): 406
Qpgrpid-1786: 394c, 633a, 633b
Qpid-1787: 394c, 633a, 633b
Qppid-1788: 394c, 633a, 633b
Qproc-871: 602a, 603a, 605d, 611
qproduce(): 384a, 407, 676d
Qprofile-877: 507a, 507b, 602a, 603a, 611
qputback(): 411a, 412, 413, 746b
Qrandom-1789: 633a, 633b, 637
Qraw-1590: 768, 768
qread(): 389d, 413, 464, 652b, 676d
Qreboot-843: 626d, 626e, 627b
Qregs-872: 602a, 603a, 605d, 608c, 611
qremove(): 410c, 412, 413, 746b
qreopen(): 419b, 463b, 651g, 676d
qschedulability-2599: 568c, 568c
Qsegment-873: 122e, 602a, 603a, 611
qsetlimit(): 420b, 676d
QSHIFT-910: 604a, 604d, 614d
Qstarve: 399a, 402a, 403, 405, 406, 407, 409a, 410a, 414b, 416, 418a, 419b
Qstatus-874: 602a, 603a, 605d, 611
Qswap-1790: 205d, 216b, 633a, 633b
Qsyscall-878: 492d, 602a, 603a, 611
Qsysname-841: 625e, 625f, 625g, 625h
Qsysstat-844: 658d, 658e, 659a, 659b
QTAPPEND: 427b, 817
QTAUTH: 742b, 762c
QTDIR: 70b, 297b, 314c, 323b, 324a, 324b, 324c, 331a, 331c, 336b, 350c, 359c, 388b, 423c, 424a, 425a, 426, 429b, 431, 433a, 459d, 460a, 463a, 605d, 608c, 611, 613, 614d, 622b, 676d, 710, 712, 719, 734, 742b, 746b, 758d, 760, 768, 817, 827c, 916, 924b, 931a, 977e
QTEXCL: 651c
Qtext-875: 602a, 603a, 611
QTFILE: 201d, 460b, 614d, 742b, 759, 768, 827c, 916
Qtime-1791: 264d, 265b, 633a, 633b
QTMOUNT: 70b, 427c
Qtop-1748: 827c, 827c
Qtopbase-1585: 768, 768
Qtopctl-1586: 768, 768
Qtopdir-1584: 768, 768
Qtrace-860: 500g, 501a, 501b, 602a, 613, 614d
Queue: 701b, 701b
Queue (typedef): 701b
Queueing: 81e, 81f
QueueingR: 83d, 83e, 85d
QueueingW: 84b, 85a, 85b, 85d
queueproc(): 222b, 232a, 241b

QueueState: 399a
Qunitbase-1588: 768, 768
Qunitdir-1587: 768, 768
qunlock(): 82c, 116c, 119b, 121a, 124a, 124c, 124e, 125a, 125b, 135d, 156a, 156c, 183, 185, 196d, 197d, 202b, 208d, 211d, 257e, 258e, 320c, 347e, 352c, 359f, 360b, 369c, 370a, 389d, 412, 413, 414b, 452, 454, 455, 461a, 463a, 463b, 470, 471a, 472c, 483d, 483e, 484a, 486a, 486c, 487a, 490, 568c, 582d, 600b, 601a, 601b, 605d, 608c, 610, 611, 613, 616b, 619a, 636b, 676d, 716, 719, 758d, 759, 760, 762c, 768, 817, 862, 876d, 911b, 912a, 912b, 913a, 914a, 920c, 921, 926, 937a, 937b, 941c, 949b, 950
Qusbdir-2241: 906, 916
Quser-1792: 394c, 395a, 633a, 633b
qwait(): 410a, 412, 413
Qwait-876: 600b, 602a, 603a, 611, 614d
qwakeup_iunlock(): 411b, 412, 413
qwindow(): 419d
qwrite(): 389d, 397a, 401b, 465b, 652c, 654a, 676d
Qzero-1793: 394c, 633a, 633b
randn-1798: 634b, 634c, 634d
randomclock(): 635d, 636a
randominit(): 634a, 636a
randomread(): 634c, 636b, 637
Rawcmd-1581: 768, 768
Rawdata-1582: 768, 768
Rawstatus-1583: 768, 768
Rb: 634e, 634f
rb: 634f, 635a, 635b, 635c, 635d, 636a, 636b
Rb.bits: 634e
Rb.buf: 634e
Rb.consumer: 634e
Rb.ep: 634e
Rb.next: 634e
Rb.producer: 634e
Rb.randn: 634e
Rb.randomcount: 634e
Rb.rp: 634e
Rb.wakeme: 634e
Rb.wp: 634e
rbnotempty(): 635b, 636b
rbnotfull(): 635a, 635c, 635d
Rclass: 924a
Rd2h: 924a, 944b
rdb(): 652g, 698b
read(): 312b, 313b
Read-2233: 935, 943, 944a, 944b, 948c, 949b
Read-2346: 893a, 894c
readbintime(): 266a, 266c
readelfhdr(): 849b
readfn-852: 977c, 978a, 979c
readn(): 849b, 849b
readnum(): 394c, 504b, 504c, 605d, 659a, 670c, 676d, 712, 768

Readrdy-1693: 982, 986
READSTR: 626c, 867a, 922b, 967a
readstr(): 205d, 265c, 394c, 492d, 605d, 616b, 624c, 624g, 625b, 625g, 626c, 659a, 670d, 712, 742b, 768, 827c, 922b, 967a, 980b, 980c
readtime(): 264d, 265c
Readtrans-1699: 982
Ready: 223b, 231c, 232a, 232d, 568c
ready: 82c, 84b, 85d, 134b, 157c, 159c, 228b, 235a, 287d, 455, 480b, 483d, 483e, 484a, 485d, 491b, 502c, 568c
realloc(): 105b, 827c
ReallocOffset-1815: 106b, 861b
rebalance(): 241a, 241b
rebootmsg: 626f, 627b
Ref: 87d, 701b, 701b
Ref (typedef): 701b
Reg (typedef): 899
Regmask-2385: 890c, 891a
REGPC-2589: 525b
Regsize-2286: 956
release(): 568c, 568c
releaseintr(): 568c, 568c
relocateseg(): 147b
renameuser(): 132d, 133a
REND: 455, 479a, 480b
Rendez: 701b, 701b
Rendez (typedef): 701b
Rendezvous: 455, 480a, 480b
RENDHASH: 479a
RENDLOG: 479a, 479a
reprioritize(): 238c, 239a, 240, 241b
req: 698b
Req-2358: 893c, 895a
reset(): 953a, 953b
Reset-2300: 956
Resetdelay-2231: 935
ResetdelayHS-2232: 935, 951c
resetmouse(): 716, 716
resolution-927: 716, 716
Resp0-1626: 982, 986
Resp1-1627: 982, 986
Resp136-1673: 982, 983a, 986
Resp2-1628: 982, 986
Resp3-1629: 982, 986
Resp48-1674: 982, 983a, 986
Resp48busy-1675: 982, 983a, 986
Respmask-1671: 982, 986
Respnone-1672: 982, 986
resrcwait(): 131a, 131c
Resume-1666: 982
RETinst-2383: 519a, 520a, 890c, 891a

Retrypause-1760: 827c, 827c
returnfalse(): 104b, 104c, 131c, 211d, 255a, 258e, 719, 792, 827c, 941c, 946b, 949b, 950, 951b, 951c
RFCENVG-2556: 133b, 138a, 138b, 140a
RFCFDG-2557: 133b, 137a, 137b, 139d
RFCNAMEG-2555: 133b, 137c, 137d, 139e
RFENVG-2553: 133b, 138a, 138b, 140a
RFFDG-2554: 133b, 137a, 137b, 139d
RFMEM-2550: 133b, 135d, 139c
RFNAMEG-2552: 133b, 137c, 137d, 139e
RFNOMNT-2560: 133b, 363e, 363f
RFNOTEG-2558: 133b, 138d, 140c
RFNOWAIT-2551: 133b, 139c, 156d
Rfork: 133b
RFPROC-2549: 133b, 139b
RFREND-2559: 133b, 138c, 140b
Rgetstatus: 924a
Rgrp: 701b, 701b
Rgrp (typedef): 701b
Rh2d: 924a
rhubread(): 923, 924b
rhubwrite(): 924a, 931a
Right-978: 376a, 377c, 380c
Rl-2598: 568c, 568c
rlock(): 83e, 353b, 354d, 358b, 359f, 361b, 368a, 598a, 733d, 734, 827c
rootattach(): 422b, 730c
rootclose(): 422b, 730b
rootdata-831: 423c, 423c
rootdevtab: 422b
rootdir-830: 423c, 423c
rootgen(): 425a, 425b, 730c
rootlist-832: 423c, 424b, 425a, 425b
rootopen(): 422b, 730c
rootread(): 422b, 425b
rootreset(): 422b, 424c
rootstat(): 422b, 730c
rootwalk(): 422b, 730c
rootwrite(): 422b, 730c
Rother: 924a
ROUND: 47b, 116a, 124c, 369c, 435e, 472c, 941c, 944b, 949b
ROUNDUP: 47f, 849b
rpcattn(): 587, 746b
Rportenable-2269: 906, 924a
Rportreset-2270: 906, 924a
Rsetfeature: 924a
RspOk-2359: 893c, 895a
Rstc-2452: 1000c, 1001b, 1001c, 1001d
Rsts-2456: 1000c
rts(): 966d, 968b
RtsN-2218: 961e, 966d

Runeself: 308b, 672a
runlock(): 84b, 353b, 354d, 358b, 359f, 361b, 368a, 598a, 733d, 734, 827c
Running: 67c, 224a, 228b, 236f, 506b, 568c, 877b
runproc(): 224a, 225
runq: 214a, 221b, 222b, 222c, 223a, 225, 232a, 232d, 241b, 564b, 564c, 568c, 644f
runvec: 221e, 221f, 222b, 222c, 236d, 568c
RWlock: 701b, 701b
RWlock (typedef): 701b
rwstats: 83e, 85b, 86, 661g, 661h, 661i
RxEn-2226: 961e, 964b
RxIen-2210: 961e, 964b
RxPin-2184: 961c, 964b
RxRdy-2222: 961e, 963d, 967d
Sargs: 701b, 701b
Sargs (typedef): 701b
scale(): 719
Scaling-2461: 239d, 239e, 239g, 239h
sched: 81f, 83e, 85b, 134b, 152c, 194e, 199d, 224f, 229c, 229d, 229e, 231b, 414b, 455, 471d, 480b, 483d, 485b, 608b, 629b, 635c, 877b
scheddump(): 642c, 644f
schedgain-2460: 239f, 239h
Scheding: 129c, 130a, 225
schedinit(): 224f, 272b, 544a
Schedq: 701b, 701b
Schedq (typedef): 701b
screenputs: 384b, 384c, 384c, 384d, 390b, 392b, 629b
Scroll-984: 376a, 376b, 377a
ScrollDown-941: 719
ScrollLeft-942: 719
ScrollRight-943: 719
scrollswap-957: 719, 719
ScrollUp-940: 719
scrprint(): 698a
scsibio(): 792
scsifmt10(): 792, 792
scsifmt16(): 792, 792
scsionline(): 792
scsirio(): 792, 792
scsitest(): 792, 792
scsiverify(): 792
sd1gen(): 768, 768
sd2gen(): 768, 768
sdadddevs(): 768, 768
sdaddpart(): 768, 768
sdattach(): 767b, 768
sdbio(): 768, 768
SDbusy: 764c, 792
SDcheck: 764c, 792
sdclose(): 767b, 768

sdconfig(): 768, 768
sddelpart(): 768, 768
sddevtab: 767b, 768
SDeio: 764c
SDev: 701b, 701b
SDev (typedef): 701b
sdfree: 764a
SDfreq-1618: 981b, 986
sdgen(): 768, 768
sdgetdev(): 768, 768
sdgetunit(): 768, 768
SDifc: 701b, 701b
SDifc (typedef): 701b
sdincvers(): 768, 768
sdindex(): 768, 768
sdinitpart(): 768, 768
SDinq0periphqual: 764c
SDinq0periphtype: 764c, 768, 792
SDinq1removable: 764c, 768, 792
SDio: 701b, 701b
sdio: 989a
SDio (typedef): 701b
SDmalloc: 764c
sdmalloc: 764b
SDmaxio: 764c, 768
SDnosense: 764c, 768
SDnostatus: 764c
SDnpart: 764c, 768
SDok: 764c, 768, 792
sdopen(): 767b, 768
SDpart: 701b, 701b
SDpart (typedef): 701b
SDpercd: 764c, 768
SDperdisk: 764c, 768, 792
SDperjuke: 764c
SDperm: 701b, 701b
SDperm (typedef): 701b
SDpermo: 764c, 768
SDperpr: 764c
SDpertape: 764c
SDperworm: 764c, 768
sdread(): 767b, 768
SDreq: 701b, 701b
SDreq (typedef): 701b
sdreset(): 767b, 768
SDretry: 764c
sdrio(): 768, 768
sdstat(): 767b, 768
SDtimeout: 764c

SDunit: 701b, 701b
SDunit (typedef): 701b
SDvalidsense: 764c, 792
sdwalk(): 767b, 768
sdwrite(): 767b, 768
sdwstat(): 767b, 768
seconds(): 131c, 263b, 427c, 719
Section: 178d, 271c, 387a, 536i
Sectorsz-1745: 827c, 827c
sectorsz-1770: 827c, 827c
seedrand(): 634c, 634d
seg(): 119b, 119c, 183, 471a, 472c, 477b, 478a, 478b, 482a, 488f
segattach(): 468a, 468c
segclock(): 506b, 506c
SEGMAXSIZE: 556d, 556f
Segment: 701b, 701b
Segment (typedef): 701b
segpage(): 118a, 287d
Sema: 701b, 701b
Sema (typedef): 701b
semacquire(): 476c, 477b
semawoke(): 476b, 476c, 477a
semdequeue(): 475a, 476c, 477a
semqueue(): 474e, 476c, 477a
semrelease(): 475c, 478b
semwakeup(): 475b, 475c, 476c, 477a
Sender-2349: 893a
Sendnotes: 565c, 566g
seprintconf(): 827c
seprintdev(): 827c, 827c
seprintep(): 951a
serialoq: 629b, 653a, 653b, 653c, 653d, 654a, 654b, 676d, 698b
Set0-2195: 961d, 963b
setaccelerated(): 716, 716
setbootcmd(): 849b
Setbuswidth-1621: 981b, 986
setbuttonmap(): 719, 719
setclkrate(): 272b, 896c
setdebug(): 952c, 953a
setdsize(): 827c, 827c
setintellimouse(): 716, 716
setlength(): 676d, 676d
setlinear(): 716, 716
setmalloctag(): 105c, 106a, 106c, 106d, 106e, 106g, 308b, 401b, 436b, 437c, 669b
setmaxpkt(): 925b, 926
setpower(): 563a, 946b
setrealloctag(): 105c, 106b, 106d, 106e, 106g
setres(): 716, 716
setswapchan(): 216b, 216c

setupcpop(): 891a, 891b, 891c
setupfpctlop(): 519a, 519b, 519c
setupfpop(): 520a, 520b
SG_BSS: 122c, 185, 207d, 211d, 556f
SG_CEXEC: 206a
SG_DATA: 122b, 185, 207d, 211d, 467c, 490, 556b
SG_PHYSICAL: 556c, 557c, 557d
SG_RONLY: 119c, 122e, 183, 855
SG_SHARED: 185, 207d, 211d, 556c, 556f
SG_STACK: 112b, 121d, 185, 207d, 211d, 287d, 467c
SG_TEXT: 121b, 185, 196e, 207d, 211d, 287d, 467c, 488f, 506d
SG_TYPE: 115b, 121a, 122e, 185, 208d, 211d, 467c, 468c, 488f, 506d
Shdr32sz-2418: 849b
Shdr64sz-2421: 849b
Shift-963: 376a, 376b, 377a, 377c, 378d, 380c
skipscheds: 232d, 662b
skipslash(): 310b, 311a
sleep: 151c, 211d, 255c, 258e, 410a, 414b, 476c, 486c, 568c, 587, 600b, 635c, 636b, 676d, 719, 938b, 939b
SLOT-912: 604a, 605d, 608c, 611, 613, 614d
Slotisrver-1647: 982, 985a
Small: 178f, 186, 271c
smalloc(): 104b, 115a, 116a, 124c, 138b, 140a, 154c, 226f, 287d, 295e, 297b, 298c, 308b, 310b, 310c, 324f, 325b, 326a, 344b, 345b, 347d, 356c, 356d, 357a, 369c, 371, 429b, 462, 479d, 497, 498a, 557d, 589, 594c, 598a, 613, 659a, 669b, 712, 734, 758c, 759, 768, 827c, 912a, 914b, 918a, 922b, 926
sname-909: 113a, 122e
Soc: 973c, 973c
soc: 72b
Soc (typedef): 973c
sofdone(): 938a, 938b
sofwait(): 938b, 941c
source-1769: 827c, 827c
Sourcead-2292: 956
sp-2433: 287b, 287d, 289a
Spec-960: 376a, 376a, 378d
Spiintspt-1646: 982
spname-2276: 909a, 909j, 911b, 925b
Sporadic: 565c, 566g, 568c
Srcdreq-2312: 956
Srcignore-2311: 956
Srcinc-2314: 956
Srcwidth128-2313: 956
Srstcmd-1651: 982, 986
Srstdata-1650: 982, 986
Srsthc-1652: 982, 984e
Srv: 757a, 757b
srv-784: 757c, 758b, 758c, 759, 760
Srv.chan: 757b
Srv.link: 757b
Srv.name: 757b

Srv.owner: 757b
Srv.path: 757b
Srv.perm: 757b
Srv (typedef): 757a
srvattach(): 763a, 763b
srvclose(): 762a, 763a
srvcreate(): 759, 763a
srvdevtab: 763a
srvinit(): 758a, 763a
srvlk-783: 757d, 758d, 759, 760, 762c
srvlookup(): 758b, 758d, 759, 762c
srvname(): 352c, 758c
srvopen(): 758d
srvread(): 762b, 763a
srvremove(): 760, 762a, 763a
srvstat(): 763a, 763b
srvwalk(): 763a, 763b
srvwrite(): 762c, 763a
sseek(): 315, 316
SSEG: 118c, 287d, 468c, 470
STACKALIGN: 287c, 287d, 999a
STAGESIZE: 972c
startcpu(): 541b, 541c
startcpus(): 540d
startlock-2448: 541b, 544a, 994c
statename: 67d, 223b, 568c, 605d, 645b
STATSIZE-907: 603a, 605d, 740b
status(): 967a, 968b
Status-1631: 982, 986, 988b
Status-2350: 893a, 894b, 894c
stop(): 966a, 968b
Stopped: 159c, 484c, 485b, 485d, 486d, 488d, 491b, 502c
Stride-2295: 956
Stride-2574: 521b
subarch(): 521d, 523c
Suspend-1665: 982
swapaddr: 197d, 556d
Swapalloc: 204a, 204b
swapalloc: 204b, 205c, 205d, 206d, 207a, 207c, 211b, 211c, 211d, 213a, 216c, 283c, 284b, 285a, 816c
Swapalloc.alloc: 204a
Swapalloc.free: 204a
Swapalloc.highwater: 204a
Swapalloc.last: 204a
Swapalloc.swmap: 204a
Swapalloc.top: 204a
swapcount(): 185, 207c
swapimage: 185, 191b, 197d, 205b, 207d, 211d, 215a, 216c, 284b
swapinit(): 272b, 284b
swcursor_init(): 272b

sysalarm(): 243, 257b
sysattach(): 622d
sysawait(): 127, 150a
sysbind(): 348d
sysbrk(): 92, 123a
Syscall (typedef): 56j
syscallfmt(): 491c, 492e
syschdir(): 323a, 336a
sysclose(): 302c
syscreate(): 332b
sysctab: 163e, 492e, 885
sysdevtab: 623d
sysdir-845: 622b, 622e, 732a, 732b, 732c
sysdup(): 317
syserrstr(): 667c
sysexits(): 127, 151e
sysfauth(): 591f
sysfd2path(): 318a, 323a
sysfstat(): 323a, 339a
sysfversion(): 57b, 591e
sysfwstat(): 323a, 340a
sysinit(): 622c
sysmount(): 342, 363g
sysname: 625d, 625g, 625h
sysnop(): 57b, 57d
sysnoted(): 448a, 453a
sysnotify(): 450h
sysopen(): 302a
syspipe(): 448a, 457
syspread(): 294a, 312b
syspwrite(): 294a, 314a
sysread(): 622e
sysremove(): 335d
sysrendezvous(): 480b
sysretfmt(): 491e, 496
sysrfork(): 127, 133c
sysseek(): 294a, 315
syssegattach(): 448a, 468a
syssegbrk(): 448a, 467c
syssegdetach(): 448a, 470
syssegflush(): 448a, 472c
syssegfree(): 448a, 471a
syssemacquire(): 448b, 477b
syssemrelease(): 448b, 478b
syssleep(): 255a
sysstat(): 338c
systab: 57b, 163e, 886a
SystimerFreq-2652: 245d, 246b, 251c, 251d, 263c
Systimers: 246c, 1011b

SYSTIMERS-2647: 43b, 246a, 246d, 251a, 254d, 280
Systimers.c0: 246c
Systimers.c1: 246c
Systimers.c2: 246c
Systimers.c3: 246c
Systimers.chi: 246c
Systimers.clo: 246c
Systimers.cs: 246c
Systimers (typedef): 1011b
systsemacquire(): 448b, 478a
sysunmount(): 342, 365a
syswalk(): 732a
syswrite(): 623b
syswstat(): 323a, 339d
tadd(): 248f, 250b, 252, 253f
TagFballoc-2372: 893d
TagFbblank-2374: 387c, 893d
TagFbfree-2373: 893d
TagGetclkmax-2369: 893d
TagGetclkspd-2368: 893d, 896b
TagGetdepth-2379: 387b, 893d
TagGetfwrev-2361: 287a, 893d
TagGetmac-2363: 893d, 895b
TagGetpower-2365: 563b, 893d
TagGetram-2364: 275d, 893d
TagGetres-2375: 387b, 893d
TagGetrev-2362: 286g, 893d
TagGetrgb-2381: 893d
TagGettemp-2371: 893d, 897a
TagGetvres-2377: 893d
TAGMASK: 580d, 581a
TagResp-2360: 893c, 895a
TagSetclkspd-2370: 893d, 896c
TagSetdepth-2380: 893d
TagSetpower-2366: 563a, 893d
TagSetres-2376: 893d
TagSetrgb-2382: 893d
TagSetvres-2378: 893d
TAGSHIFT: 580d, 580d, 581a, 746b
talkrdb(): 698b, 698b
TaslockStats: 661a, 661e
TaslockStats.glare: 661a
TaslockStats.inglare: 661a
TaslockStats.locks: 661a
tconsumed-918: 500b, 500c, 500d, 500g, 501b
TCSys: 503a, 504f, 505a
TCUser: 503a, 504f, 505a
tdel(): 249d, 250c, 250e
Tdmode-2320: 956

testenq(): 568c, 568c
testschedulability(): 568c
tevents-915: 500b, 500d, 500g, 501b
text2data(): 488f, 490
Tfast (typedef): 245e
tfn(): 255c, 256d
thelock-2596: 568c, 568c
Ti-2291: 956
timeconv(): 568c, 568c
Timer: 701b, 701b
Timer (typedef): 701b
timeradd(): 250b, 255c, 281b, 568c
timerdel(): 250d, 250e, 255c, 255d, 256a, 568c
timerintr(): 246d, 252, 547e
Timermode: 248a
Timers: 701b, 701b
timers-2645: 248e, 250b, 250e, 252, 253f
Timers (typedef): 701b
timersinit(): 272b, 281b, 544a
timersinited-2646: 254a, 281a, 281b
TK2MS: 244d, 255b, 477a, 504b, 504c, 506c, 509c, 628f, 719, 949b, 950
tk2ms(): 245a, 258a, 504f
TK2SEC: 873b
tlock-919: 500b, 500g, 501a
Tmicro (typedef): 244a
TmrDbgHalt-2658: 1011a
TmrEnable-2659: 948a, 1011a
TmrIntEnable-2660: 948a, 1011a
TmrPrescale1-2661: 280, 1011a
TmrPrescale16-2662: 1011a
TmrPrescale256-2663: 1011a
Tms (typedef): 244a
tnames-1775: 827c, 827c
Tnano (typedef): 244a
TOD: 259c, 260a
tod: 260a, 260d, 261a, 261b, 262, 263a, 264a, 264b
TOD.cnt: 259c
TOD.delta: 259c
TOD.divider: 259c
TOD.hz: 259c
TOD.init: 259c
TOD.last: 259c
TOD.lasttime: 259c
TOD.multiplier: 259c
TOD.off: 259c
TOD.send: 259c
TOD.sstart: 259c
TOD.udivider: 259c
TOD.umultiplier: 259c

todfix(): 260d, 263a
TODFREQ-2669: 261a, 263b, 880b
todget(): 163e, 262, 263b, 265c, 266c, 491e, 500d, 566f, 568c
todinit(): 260d, 261b, 262, 264a, 264b, 281b, 286a, 634a
todset(): 261b, 265d, 267a
todsetfreq(): 260d, 261a, 267a
topens-916: 500b, 500d, 500g, 501a
Tperiodic: 248a, 249b, 249c, 252, 253f, 281b
tproduced-917: 500b, 500c, 500d, 500g, 501b
tqid-1773: 827c, 827c
trapname(): 646b, 647c
trapnames-2414: 161b, 647c
trap_arch_dumpstack(): 648b
TReal: 503a, 503c, 503d, 504b, 504c, 504f, 505b, 505c
Tree: 826, 827c
Tree.devs: 827c
Tree.nadevs: 827c
Tree.name: 827c
Tree.ndevs: 827c
Tree (typedef): 826
treealloc(): 827c, 827c
trees-1765: 827c, 827c
Trelative: 248a, 249b, 255c, 568c
trimblock(): 443a
true: 70e, 71a, 78f, 80a, 81f, 83a, 84b, 85b, 86, 95d, 98a, 119c, 131b, 150a, 151e, 157c, 159b, 163e, 171c, 172e, 177b, 182a, 182b, 183, 185, 186, 194e, 211c, 213e, 213f, 214a, 216b, 220f, 252, 260d, 281b, 284b, 287d, 305f, 307e, 310b, 313b, 314c, 315, 316, 317, 318a, 320c, 328f, 330b, 338c, 339a, 340a, 348b, 350c, 352c, 353b, 354d, 363e, 363f, 363g, 364b, 365d, 370a, 371, 372, 378d, 381b, 388d, 389b, 389d, 392a, 392d, 397a, 397d, 429b, 437c, 455, 457, 471d, 476a, 476c, 477a, 477b, 478a, 478b, 487c, 488a, 498c, 522c, 523b, 523c, 555g, 555h, 555i, 555k, 567b, 591e, 591f, 601i, 608c, 620b, 627e, 627f, 641b, 651g, 663b, 667d, 670a, 672e, 719, 742b, 762c, 817, 827c, 876c, 877b, 895a
Tsec (typedef): 244a
TSEG: 118c, 196d, 287d, 506c, 506d, 507a, 507b, 618d
tsemacquire(): 477a, 478a
tsleep: 104b, 131c, 211d, 255a, 320c, 477a, 719, 792, 827c, 941c, 946b, 949b, 950, 951b, 951c, 986, 988b
TSys: 503a, 504a, 504f, 505b, 505c, 645b
ttname-2275: 908d, 910a, 911b
Tufast (typedef): 245e
Tunestep-1643: 982
Tunestepsddr-1645: 982
Tunestepsstd-1644: 982
TUser: 503a, 504a, 504f, 505b, 505c, 645b
Tval (typedef): 244c
twakeup(): 255c, 256e
TxDone-2220: 961e, 965a
TxEn-2225: 961e, 964b
Txfrlen-2294: 956
TxIen-2211: 961e, 964b, 965a
TxPin-2183: 961b, 964b

TxRdy-2221: 961e, 963d, 965a, 967c
TYPE-1609: 768, 768
TypeLOG-1592: 768, 768
TypeMASK-1594: 768
TypeSHIFT-1595: 768
Uart: 701b, 701b
uart-1854: 676d, 676d
Uart (typedef): 701b
Uartalloc: 676d, 676d
uartalloc: 676d, 676d
Uartalloc.elist: 676d
uartattach(): 676d, 676d
uartclock(): 676d, 676d
uartclose(): 676d, 676d
uartconsinit(): 272b, 968a
uartctl(): 676d, 676d, 968a
uartdevtab: 676d
uartdir-1856: 676d, 676d
uartdisable(): 676d, 676d
uartdrained(): 676d, 676d
uartdrainoutput(): 676d, 676d
UartEn-2207: 961e, 964b
uartenable(): 676d, 676d
uartflow(): 676d, 676d
uartgetc(): 676d, 698b
UartIrq-2205: 961e, 963d
uartkick(): 676d, 676d, 963d
uartlist-1853: 676d, 676d
uartndir-1857: 676d, 676d
uartnuart-1855: 676d, 676d
uartopen(): 676d, 676d
uartputs(): 392b, 654b, 676d
uartread(): 676d, 676d
uartrecv(): 676d, 963d
uartreset(): 676d, 676d
uartstageinput(): 676d, 676d
uartstageoutput(): 676d, 965a
uartstat(): 676d, 676d
uarttimer-1858: 676d, 676d
uartwalk(): 676d, 676d
uartwrite(): 676d, 676d
uartwstat(): 676d, 676d
UINT2PTR: 56h, 181a, 186, 187a, 287d, 452, 999a
unbreak(): 483e, 484b
uncachepage(): 192a, 199d, 207d, 816c
unconfigure(): 768, 768
undomount(): 328f, 357b
unionread(): 359c, 359f
unionrewind(): 360a, 360b

uniquechan(): 297a, 328f, 333d, 354b
uniquepath(): 325d, 326a, 355
UNIT-1611: 768, 768
UnitLOG-1600: 768, 768
UnitMASK-1602: 768
UnitSHIFT-1603: 768
unlock(): 78d, 81f, 82c, 83a, 83e, 84b, 85b, 85d, 86, 87b, 87c, 111a, 115c, 116c, 128g, 129a, 129b, 130a, 131a, 131b, 150c, 151a, 154a, 154d, 155a, 156d, 176a, 185, 190e, 191a, 192a, 192b, 194e, 199d, 201b, 201d, 202b, 205c, 206d, 207a, 207d, 213f, 214a, 215a, 222b, 222c, 223a, 233e, 235a, 275a, 295e, 296, 298c, 301b, 302d, 303b, 313b, 314c, 316, 320a, 320c, 368a, 392b, 393a, 455, 460a, 474e, 475a, 475b, 480b, 500g, 501a, 541b, 544a, 557a, 557c, 558c, 568c, 581b, 581c, 582d, 587, 600b, 600c, 616b, 618d, 619a, 619b, 628f, 700, 710, 719, 746b, 816c, 817, 978a
unlockfgrp(): 300a, 301b, 317, 458a
UP: 162b
up: 78c, 78g, 80a, 80b, 81f, 82a, 83e, 84a, 85b, 85c, 88b, 88c, 88d, 88e, 88g, 88h, 89b, 104b, 119c, 120b, 123b, 124a, 124e, 131a, 131c, 132d, 134b, 135a, 135b, 135d, 137b, 137d, 138b, 138c, 138d, 139d, 139e, 140a, 140b, 140c, 148a, 150c, 151a, 151c, 152c, 154a, 154c, 154d, 155a, 155c, 156a, 156c, 156d, 157c, 158a, 158c, 163d, 163e, 165b, 165d, 165f, 168b, 170c, 170e, 173b, 173d, 182a, 182b, 183, 186, 187a, 197d, 199a, 199d, 211d, 213d, 220c, 224a, 226c, 227c, 228a, 228b, 228c, 229c, 229d, 229e, 232b, 233e, 234a, 234b, 236c, 236d, 236f, 238b, 239b, 239g, 239h, 250d, 255a, 255c, 255d, 256a, 256d, 257d, 257e, 258a, 258b, 274c, 287d, 289a, 299b, 300a, 302d, 303b, 305f, 307b, 308b, 311c, 311d, 317, 319a, 319b, 331a, 331c, 333d, 336a, 348b, 350c, 354d, 363a, 363b, 363c, 363d, 363e, 365d, 393a, 394c, 401b, 414b, 429b, 432d, 436b, 449g, 450a, 450d, 450g, 450h, 451a, 452, 453a, 454, 458a, 462, 465b, 466a, 467c, 468a, 470, 471a, 472a, 472c, 477b, 478a, 478b, 480b, 482a, 482b, 483c, 483d, 486a, 486c, 487e, 491c, 491e, 492c, 492e, 496, 500a, 504b, 504f, 506b, 506c, 509b, 514c, 514f, 515a, 517a, 517c, 518a, 524e, 526c, 527b, 527c, 544a, 563c, 564g, 564h, 565b, 566b, 566f, 568c, 586, 587, 588, 591b, 592a, 594c, 595a, 600c, 601a, 601b, 601d, 601g, 605d, 610, 613, 614d, 616a, 629b, 635c, 645a, 646a, 646b, 648a, 649a, 657c, 666b, 667a, 667d, 667e, 669b, 719, 733b, 734, 742b, 746b, 759, 760, 768, 792, 827c, 862, 877b, 916, 926, 941c, 944a, 944b, 946b, 949b, 950, 951b, 951c
Up-974: 376a, 377c, 380c
updatecpu(): 238c, 239a, 239b, 239h, 241b
Uro: 179b, 186
Urw: 179b, 186
usbattach(): 919b, 932c
usbbread(): 931b, 932c
usbbwrite(): 932a, 932c
usbclose(): 922a, 932c
usbctl(): 929, 930
usbctls-2272: 907c, 929
usbdevtab: 932c
usbdir-2274: 908b, 916
usbdwclink(): 953b
usbgen(): 916, 919c, 920a, 920c, 924b
usbidgen-2283: 909h, 913a, 914a
usbinit(): 919a, 932c
usbload(): 920b, 920c
usbmodename: 908c, 910b, 911b
usbopen(): 920c, 932c
usbread(): 924b, 931b, 932c
USBREGS-2229: 44b, 953a

usbreset(): 918b, 932c
usbshutdown(): 932b, 932c
usbstat(): 920a, 932c
usbwalk(): 919c, 932c
usbwrite(): 931a, 932a, 932c
userwrite(): 395a, 592a
Ustkheadroom-2435: 995a, 999a
UTFmax: 297b, 311d, 382c, 709
uvorder-1799: 267b, 267b, 267c, 268a
V-2588: 525b, 525c
validaddr(): 120b, 120c, 150a, 152b, 165c, 302a, 313b, 314c, 315, 318a, 332b, 335d, 336a, 338c, 339a, 339d, 340a, 349a, 364b, 365a, 450h, 457, 468c, 477b, 478a, 478b, 491c, 492e, 497, 498a, 526c, 591e, 591f, 667d
validdevname(): 827c, 827c
validname(): 308a, 340b, 827c, 926
validname0(): 307e, 308a, 308b
validnamedup(): 305f, 307e, 364b, 591f
validstat(): 339d, 340a, 340b, 746b
VCBUFFER: 386a, 386b, 895a
vcread(): 386b, 894c, 895a
vcreq(): 275d, 286g, 287a, 387b, 387c, 563a, 563b, 895a, 895b, 896b, 896c, 897a
vctl-2412: 175a, 176a, 177b
vctllock-2411: 175c, 176a
vcwrite(): 386b, 894b, 895a
vfiq-2413: 561c, 561d, 562a
VFPKind: 518c
VFPv2-2561: 518c, 522c
VFPv3-2562: 518c, 522c
View-962: 376a, 376a
VIRTIO: 43a, 271c, 957l
VLNUMSIZE-1796: 260c, 265c, 633b
vlong2le(): 266c, 268a
vmemchr(): 120c, 152b, 308b, 468c, 498a
Vpage0: 278c, 1002c
Vpage0.vectors: 278c
Vpage0.vtable: 278c
Vpage0 (typedef): 1002c
v_or_u2.u: 316
v_or_u2.v: 316
v_or_u2: 316, 316
Waitdreq-2304: 956
Waitmsg: 853a, 853a
Waitmsg (typedef): 853a
Waitq: 701b, 701b
Waitq (typedef): 701b
Waitrelease: 568b, 568c
Waitresp-2319: 956
Waitwrite-2303: 956
Wakeme: 233d, 233e, 235b, 455, 568c
wakeup: 154d, 155a, 181d, 211a, 211c, 211d, 256e, 259b, 320a, 402a, 402b, 403, 405, 406, 407, 411b, 414b, 416,

418a, 418b, 420d, 475b, 486a, 487a, 568c, 635d, 636b, 676d, 719, 746b, 948b, 958d, 985b
walk(): 305f, 328f, 333d, 457
Walkqid: 701b, 701b
Walkqid (typedef): 701b
waserror: 121a, 135d, 152b, 152c, 163e, 194e, 197d, 208d, 211d, 215a, 255c, 258e, 289a, 302a, 304b, 305f, 313b, 314c, 316, 317, 320c, 330a, 332b, 333d, 335d, 338c, 339a, 341, 349a, 350c, 352c, 359f, 361b, 364b, 365a, 371, 389d, 393a, 401b, 412, 413, 414b, 429b, 434a, 434b, 457, 462, 465b, 466a, 470, 476c, 477a, 486c, 488f, 500g, 526c, 582d, 587, 588, 589, 591e, 591f, 594c, 600b, 601a, 605d, 608c, 610, 611, 613, 616b, 618d, 620b, 626c, 627b, 629b, 634c, 636b, 659a, 666d, 676d, 716, 719, 733d, 734, 746b, 758d, 759, 760, 762a, 762c, 768, 792, 817, 827c, 911b, 913b, 916, 920c, 921, 922a, 922b, 924b, 926, 929, 930, 931a, 931b, 932a, 944a, 944b, 949b, 950, 988b, 997b
Watchdog: 701b, 701b
Watchdog (typedef): 701b
watchdoglink(): 1001e
Wdog-2457: 1000c, 1001b, 1001c
wdogfeed(): 1001c, 1001e
Wdogfreq-2450: 1000b, 1001c
wdogoff(): 1001d
Wdogtime-2451: 1000b, 1001c
werrstr(): 667e
Whinesecs-2604: 131c, 132a
wlock(): 85b, 347e, 350c, 365d, 368a, 734, 827c
WR(): 984b, 984e, 985b, 985c, 986, 988a, 988b
write(): 314a, 314c
Write-2234: 935, 944a, 948c, 950
Write-2347: 893a, 894b
writebintime(): 266b, 267a
writefn-853: 977d, 978a, 979d
Writerdy-1694: 982, 986
writetime(): 265b, 265d
writetomem(): 172d, 172e, 173a
Writetrans-1700: 982
wstat(): 339d, 340a, 341
wunlock(): 85d, 347e, 350c, 365d, 368a, 734, 827c
Xalloc: 701b, 701b
xalloc(): 95d, 98d, 101, 102a, 283a, 283c, 284b, 285c, 642c, 676d, 817
Xalloc (typedef): 701b
xallocz(): 95d, 95e
xfree(): 96a, 642c
Xhdr: 701b, 701b
Xhdr (typedef): 701b
xhole(): 96a, 97, 98d, 282
xinit(): 272b, 282
xlists-1849: 95a, 95e, 97, 282, 643a
xmerge(): 98a, 101, 102a
xspanalloc(): 98d
xsummary(): 96b, 98c, 283b, 436b, 642c, 643a
yfn(): 568c, 568c
Yield: 566g, 568c

yield(): 231b, 255a
Yieldonblock: 566g, 568c
Z-2586: 525b, 525c
_SEG0: 118c
_SEG1: 118c
_SEG2: 118c
_SEG3: 118c
_SEG4: 118c
_allocb(): 435e, 436b, 437c
_assert:
_efgfmt(): 700
_fmtlock(): 700
_fmtunlock(): 700
_globalsegattach: 468b, 468c
_proctrace(): 500d, 500g
__anon_enum_102: 423a
__anon_enum_103: 622a
__anon_enum_104: 626f
__anon_enum_105: 977b
__anon_enum_106: 459c
__anon_enum_107: 602a
__anon_enum_108: 602b
__anon_enum_109: 740b
__anon_enum_112: 716
__anon_enum_113: 719
__anon_enum_114: 719
__anon_enum_115: 719
__anon_enum_116: 376a
__anon_enum_118: 710
__anon_enum_119: 712
__anon_enum_193: 764c
__anon_enum_194: 764c
__anon_enum_195: 768
__anon_enum_196: 768
__anon_enum_198: 981b
__anon_enum_199: 982
__anon_enum_213: 827c
__anon_enum_214: 388a
__anon_enum_215: 696b
__anon_enum_216: 855
__anon_enum_217: 93c
__anon_enum_219: 861b
__anon_enum_221: 1006a
__anon_enum_222: 676d
__anon_enum_293: 961d
__anon_enum_294: 961e
__anon_enum_295: 935
__anon_enum_296: 906
__anon_enum_298: 898a

__anon_enum_299: 956
__anon_enum_321: 562c
__anon_enum_322: 896a
__anon_enum_323: 893a
__anon_enum_324: 893b
__anon_enum_325: 893c
__anon_enum_326: 893d
__anon_enum_327: 890c
__anon_enum_332: 1002b
__anon_enum_333: 849b
__anon_enum_335: 995a
__anon_enum_338: 996h
__anon_enum_339: 994c
__anon_enum_340: 1000b
__anon_enum_341: 1000c
__anon_enum_342: 868c
__anon_enum_343: 346d
__anon_enum_344: 868c
__anon_enum_345: 479a
__anon_enum_346: 868c
__anon_enum_347: 868c
__anon_enum_348: 877c
__anon_enum_356: 514i
__anon_enum_357: 521a
__anon_enum_358: 521b
__anon_enum_359: 521c
__anon_enum_360: 525b
__anon_enum_361: 567b
__anon_enum_362: 568c
__anon_enum_363: 568c
__anon_enum_364: 568c
__anon_enum_365: 875
__anon_enum_369: 874c
__anon_enum_370: 1010d
__anon_enum_371: 1011a
__anon_enum_88: 817
__anon_enum_89: 808c
__anon_enum_91: 813b
__anon_enum_92: 811
__anon_enum_94: 580d
__anon_enum_96: 734
__anon_struct_197.name: 768
__anon_struct_197.parse: 768
__anon_struct_197: 768, 768
__anon_struct_336.cmdline: 996g
__anon_struct_336.data: 996g
__anon_struct_336.mem: 996g
__anon_struct_336: 996g, 996g
__anon_struct_337.base: 996g

__anon_struct_337.size: 996g
__anon_struct_337: 996g, 996g
__anon_struct_93.aux: 68b
__anon_struct_93.chanpipe: 459g
__anon_struct_93.mid: 605b
__anon_struct_93.pgrpid: 604c
__anon_struct_93: 68b, 68b

Bibliography

[Bac86] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice Hall, 1986. cited page(s) 19

[BC05] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 3rd edition, 2005. cited page(s) 17, 19

[Com84] Douglas Comer. *Operating System Design: The XINU Approach*. Prentice Hall, 1984. cited page(s) 17

[Com87] Douglas Comer. *Operating System Design: Volume II Internetworking with XINU*. Prentice Hall, 1987. cited page(s) 17

[Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. cited page(s) 19

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. cited page(s) 19

[Lio77] John Lions. *Lions Commentary on UNIX 6th Edition, with Source Code*. Peer to Peer Communications, 1977. cited page(s) 16, 19

[Lov10] Robert Love. *Linux Kernel Development*. Addison-Wesley, 3rd edition, 2010. cited page(s) 17

[MNNW14] Marshall Kirk McKusick, George V. Neville-Neil, and Robert N. M. Watson. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley, 2nd edition, 2014. cited page(s) 19

[Pad09] Yoann Padioleau. Syncweb, literate programming meets unison, 2009. https://github.com/aryx/syncweb. cited page(s) 19

[Pad15a] Yoann Padioleau. *Principia Softwarica: The ARM Assembler 5a*. 2015. cited page(s) 19, 35, 42, 45, 639

[Pad15b] Yoann Padioleau. *Principia Softwarica: The ARM Emulator 5i*. 2015. cited page(s) 19

[Pad15c] Yoann Padioleau. *Principia Softwarica: The ARM Linker 5l*. 2015. cited page(s) 45, 59, 161, 639

[Pad16a] Yoann Padioleau. *Principia Softwarica: The Build System mk*. 2016. cited page(s) 639

[Pad16b] Yoann Padioleau. *Principia Softwarica: The Plan 9 C Compiler 5c*. 2016. cited page(s) 45, 64, 639

[Pad16c] Yoann Padioleau. *Principia Softwarica: The Plan 9 Core Libraries*. 2016. cited page(s) 27, 30, 31, 34, 99, 446, 639

[Pad16d] Yoann Padioleau. *Principia Softwarica: The Plan 9 Graphics System draw*. 2016. cited page(s) 16, 44, 48, 57, 101, 386, 639

[Pad16e] Yoann Padioleau. *Principia Softwarica: The Plan 9 Network Stack /net*. 2016. cited page(s) 16, 28, 48

[Pad16f] Yoann Padioleau. *Principia Softwarica: The Plan 9 Windowing System rio*. 2016. cited page(s) 33

[Pad18] Yoann Padioleau. *Principia Softwarica: The Plan 9 Shell rc*. 2018. cited page(s) 639

[SGG18] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, 10th edition, 2018. cited page(s) 19, 21

[Tan87] Andrew S. Tanenbaum. *Operating Systems Design and Implementation*. Prentice Hall, 1987. cited page(s) 17, 19

[TB14] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Pearson, 4th edition, 2014. cited page(s) 19, 21