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, Dave Presotto, and Richard Miller

March 24, 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 **17**
- 1.1 Motivations . . . 17
- 1.2 The Plan 9 kernel: `9` . . . 18
- 1.3 Other kernels . . . 18
- 1.4 Getting started . . . 18
- 1.5 Requirements . . . 19
- 1.6 About this document . . . 20
- 1.7 Copyright . . . 20
- 1.8 Acknowledgments . . . 20

2 Overview **21**
- 2.1 Operating system principles . . . 21
- 2.2 `9`'s syscall interface . . . 21
- 2.3 `helloraw.c` and `helloraw.s` . . . 24
- 2.4 `minishell.c` . . . 25
- 2.5 The Raspberry Pi machine (ARM) . . . 26
  - 2.5.1 ARM CPU . . . 26
  - 2.5.2 ARM coprocessors . . . 28
  - 2.5.3 ARM MMU . . . 29
  - 2.5.4 Raspberry Pi Input/Output devices . . . 31
- 2.6 C and assembly . . . 33
- 2.7 Code organization . . . 34
  - 2.7.1 Portable versus arch-specific code . . . 34
  - 2.7.2 Main headers . . . 35
  - 2.7.3 Top directories . . . 35
- 2.8 Software architecture . . . 35
  - 2.8.1 Booting overview (ARM) . . . 36
  - 2.8.2 Memory overview (ARM) . . . 36
  - 2.8.3 Syscall overview (ARM) . . . 41
  - 2.8.4 FAQ . . . 43
  - 2.8.5 NSFAQ . . . 44
- 2.9 Book structure . . . 45

3 Core Data Structures **46**
- 3.1 `Conf` and `conf` . . . 46
- 3.2 `Cpu` and `cpu` . . . 47
- 3.3 `Proc` and `up` . . . 49
- 3.4 `Chan` . . . 51
- 3.5 `Dev` and `devtab` . . . 53
- 3.6 `Soc` and `soc` (ARM) . . . 54

4 Concurrency **55**
4.1 Critical regions . . . 55
4.2 Interrupts disabling: `arch_splhi()`, `arch_spllo()` (ARM) . . . 55
4.3 Test and set: `arch_tas()` (ARM) . . . 57
4.4 Atomic count: `arch_xinc()` and `arch_xdec()` (ARM) . . . 58
4.5 Spin `Lock` . . . 58
4.6 Uninterruptable `ILock` . . . 61
4.7 Waiting `QLock` . . . 62
4.8 Readers/Writer `RWLock` . . . 65
4.9 Atomic `References` and `Counter` . . . 68
4.10 Safeguards . . . 69
4.10.1 `Lock.p` . . . 70
4.10.2 `Proc.nlocks` . . . 70
4.10.3 `Cpu.ilockdepth` . . . 71
4.11 Synchronization . . . 71
4.11.1 Rendez-vous . . . 71
4.11.2 Producer/Consumer Queue . . . 72
4.11.3 Semaphore . . . 72

I Virtualizing the Machine **73**

5 Memory **74**
5.1 Overview . . . 74
5.2 Long-lived structure allocator: `xalloc()` . . . 75
5.2.1 Data structures . . . 75
5.2.2 `xalloc()` . . . 77
5.2.3 `xfree()` . . . 78
5.2.4 `xmerge()` . . . 79
5.3 Memory pools . . . 80
5.3.1 `Pool` . . . 81
5.3.2 `plock()` . . . 82
5.3.3 `pmainmem` . . . 82
5.3.4 `poolprint()` . . . 84
5.4 Kernel `malloc()` . . . 85
5.4.1 `malloc()` . . . 85
5.4.2 Sleeping `smalloc()` . . . 85
5.4.3 `mallocxxx()` . . . 86
5.4.4 malloc tags . . . 87
5.5 User `malloc()` . . . 88
5.6 `Page` and `palloc` . . . 88
5.6.1 `Page` . . . 89
5.6.2 `Palloc` . . . 90
5.6.3 `newpage()` . . . 91
5.7 `Segment` and `Pagetable` . . . 94
5.7.1 `Segment` . . . 94
5.7.2 `Pagetable` . . . 95
5.7.3 `newseg()` . . . 97
5.7.4 `smallpagedir` . . . 99
5.7.5 Adding a page to a segment: `segpage()` . . . 99

5.7.6 Process segments . . . 100
5.7.7 /proc/x/segment . . . 104
5.8 sysbrk() . . . 104
5.8.1 Extending a segment . . . 105
5.8.2 Shrinking a segment . . . 106

6 Processes **109**
6.1 Overview . . . 109
6.2 Process allocation . . . 109
6.2.1 Process ID: Proc.pid . . . 109
6.2.2 Procalloc . . . 110
6.2.3 PID hashing . . . 110
6.2.4 newproc() . . . 111
6.2.5 No more processes . . . 112
6.3 The user . . . 114
6.4 sysrfork() . . . 115
6.4.1 Fork: RFPROC . . . 116
6.4.2 Memory sharing: RFMEM . . . 117
6.4.3 arch_forkchild() (ARM) . . . 117
6.4.4 File descriptors: RFFDG, RFCFDG . . . 119
6.4.5 Namespace: RFNAMEG, RFCNAMEG . . . 119
6.4.6 Environment variables: RFENVG, RFCENVG . . . 120
6.4.7 Rendezvous group: RFREND . . . 120
6.4.8 Note group: RFNOTEG . . . 120
6.4.9 Process sharing settings (no RFPROC) . . . 121
6.5 sysexec() . . . 122
6.5.1 Text segment . . . 125
6.5.2 Data segment . . . 126
6.5.3 Bss segment . . . 126
6.5.4 Stack segment . . . 126
6.5.5 arch_execregs() (ARM) . . . 130
6.6 sysawait() . . . 130
6.6.1 Parent and children . . . 130
6.6.2 Waitmsg and Waitq . . . 131
6.6.3 sysawait() . . . 132
6.6.4 Rendez-vous: Proc.waitr . . . 133
6.7 sysexits() . . . 133
6.7.1 pexit() . . . 134
6.7.2 pexit() and schedinit() . . . 136
6.7.3 Wait message . . . 136
6.7.4 Freeing resources . . . 137
6.7.5 No waiting for child: RFNOWAIT . . . 138
6.8 Kernel processes . . . 139
6.8.1 arch_kprocchild() (ARM) . . . 140
6.9 Process control /proc/#/ctl . . . 141

7 Interrupts **142**
7.1 Overview . . . 142
7.2 Exceptions (ARM) . . . 143
7.3 Exception `vectors` (ARM) . . . 143
7.4 System calls . . . 144
7.4.1 `_vsvc()` (ARM) . . . 144
7.4.2 `Sargs` . . . 145
7.4.3 `arch__syscall()` (ARM) . . . 145
7.5 Exception handlers . . . 148
7.5.1 `_vxxx()` (ARM) . . . 148
7.5.2 `_vswitch()` (ARM) . . . 148
7.5.3 `arch__trap()` (ARM) . . . 150
7.5.4 Undefined instruction fault (ARM) . . . 152
7.5.5 Prefetch fault (ARM) . . . 153
7.5.6 Data fault (ARM) . . . 153
7.6 IRQs (ARM) . . . 156
7.6.1 `Vctl` (ARM) . . . 156
7.6.2 `Intregs` (ARM) . . . 157
7.6.3 `arch_intrenable()` (ARM) . . . 157
7.6.4 Basic IRQs (ARM) . . . 158
7.6.5 `irq()` (ARM) . . . 159

8 Virtual Memory **160**
8.1 Overview . . . 160
8.2 The ARM MMU (ARM) . . . 160
8.3 Pagination and `mmupd`/`mmupt` (ARM) . . . 161
8.4 Page fault . . . 164
8.4.1 `faultarm()` (ARM) . . . 164
8.4.2 `fault()` . . . 165
8.4.3 `fixfault()` . . . 165
8.4.4 `arch_putmmu()` (ARM) . . . 167
8.5 Copy on write . . . 169
8.6 Page cache . . . 170
8.7 Demand loading . . . 173
8.7.1 `Kimage` and the image cache . . . 173
8.7.2 `Segment` and `KImage` . . . 174
8.7.3 Paging in . . . 177
8.7.4 `duppage()` . . . 180
8.7.5 Collecting images . . . 182
8.8 Swapping . . . 184
8.8.1 `Swapalloc` . . . 185
8.8.2 `/dev/swap` . . . 186
8.8.3 `PageOrSwap` . . . 186
8.8.4 Swapping plan . . . 188
8.8.5 The `pager` . . . 190
8.8.6 Paging out . . . 195
8.8.7 Paging in . . . 196
8.8.8 Setting up the swap file . . . 196

9 Scheduling **198**
9.1 Overview . . . 198
9.2 Process priority . . . 198
9.3 The runqueue . . . 200
9.4 The scheduler . . . 202
9.4.1 sched() . . . 202
9.4.2 schedinit() . . . 203
9.4.3 runproc() . . . 204
9.5 Context switch . . . 205
9.5.1 Label . . . 205
9.5.2 Kernel stacks . . . 205
9.5.3 arch_gotolabel() (ARM) . . . 206
9.5.4 sched() and schedinit() dance . . . 206
9.5.5 Delayed sched() . . . 207
9.5.6 arch_mmuswitch() (ARM) . . . 209
9.5.7 arch_procsave() (ARM) . . . 209
9.6 Cooperative scheduling . . . 210
9.6.1 yield() . . . 210
9.6.2 ready() . . . 210
9.6.3 Rendez . . . 211
9.6.4 sleep() . . . 212
9.6.5 wakeup() . . . 214
9.7 Preemptive scheduling . . . 214
9.7.1 hzsched() . . . 215
9.7.2 hzclock() . . . 215
9.8 Customized scheduling . . . 216
9.9 Real-time scheduling . . . 217
9.10 Adjusting priority . . . 217
9.10.1 reprioritize() . . . 217
9.10.2 rebalance() . . . 220
9.11 Halting (ARM) . . . 221

10 Time **222**
10.1 Overview (arm) . . . 222
10.2 Time units . . . 222
10.2.1 Seconds . . . 222
10.2.2 Ticks . . . 222
10.2.3 Fastticks (ARM) . . . 223
10.3 Clock interrupt (ARM) . . . 224
10.4 Timers . . . 225
10.4.1 Timer . . . 225
10.4.2 Timers and CPU timers . . . 225
10.4.3 tadd()/tdel() . . . 226
10.4.4 timeradd()//timerdel() . . . 227
10.4.5 arch_timerset() (arm) . . . 228
10.4.6 timerintr() . . . 229
10.4.7 Clock callbacks: addclock0link() . . . 231
10.4.8 Delay (ARM) . . . 231
10.5 syssleep() . . . 232
10.6 Alarms . . . 234

10.7 sysalarm() . . . 234
10.7.1 alarmkproc() . . . 236
10.7.2 checkalarms() . . . 236
10.8 Time of day . . . 237
10.9 sysnsec() . . . 242
10.10 /dev/time . . . 242
10.11 /dev/bintime . . . 243
10.12 Watchdog . . . 246

11 Initialization **247**
11.1 Booting the kernel . . . 247
11.1.1 Loading the kernel in physical memory (ARM) . . . 247
11.1.2 First instructions (ARM) . . . 247
11.1.3 Basic pagination (ARM) . . . 248
11.1.4 Jumping to C main() (ARM) . . . 250
11.2 Kernel initialization (ARM) . . . 250
11.2.1 cpu initialization . . . 252
11.2.2 Memory detection (ARM) . . . 253
11.2.3 conf initialization (ARM) . . . 254
11.2.4 Exception vector initialization (ARM) . . . 256
11.2.5 Clock initialization (ARM) . . . 258
11.2.6 Timer initialization . . . 259
11.2.7 Virtual memory initialisation (ARM) . . . 259
11.2.8 Memory initialisation . . . 260
11.2.9 Device initialisation . . . 263
11.2.10 Displaying hardware information (ARM) . . . 264
11.3 The first process (ARM) . . . 265
11.4 Booting the user . . . 270

II Everything is a File **271**

12 Files **272**
12.1 Overview . . . 272
12.2 Channels . . . 272
12.3 Devices table devtab . . . 275
12.4 File descriptors, Fgrp . . . 276
12.5 sysopen(), sysclose() . . . 279
12.6 namec() . . . 282
12.6.1 Checking valid filename string . . . 285
12.6.2 Parsing filename . . . 287
12.6.3 Error management . . . 289
12.7 syspread(), syspwrite() . . . 290
12.8 sysseek() . . . 293
12.9 Collecting unclosed files . . . 295
12.10 Special opening modes . . . 297
12.10.1 Close on exec . . . 297
12.10.2 Remove on close . . . 298

13 Directories **299**
13.1 Overview . . . 299
13.2 `Path` . . . 300
13.3 `..` . . . 302
13.4 `walk()` . . . 303
13.5 `syscreate()` . . . 307
13.6 `sysremove()` . . . 310
13.7 `syschdir()` . . . 311
13.8 `sysfd2path()` (a.k.a `pwd()`) . . . 312
13.9 `DirEntry` . . . 312
13.10 `sysstat()` . . . 313

14 Namespaces **317**
14.1 Overview . . . 317
14.2 Mount point . . . 317
14.3 Mount table . . . 319
14.4 Device path . . . 321
14.5 `sysbind()` . . . 322
14.6 `/proc/#/ns` . . . 325
14.7 `findmount()`, `domount()` . . . 327
14.8 Path mount points and `..` . . . 329
14.9 Union mount . . . 331
14.9.1 Union walk . . . 331
14.9.2 `->umh` . . . 331
14.9.3 `syspread()` for directories . . . 332
14.9.4 `MCREATE` . . . 334
14.9.5 Union of union . . . 335
14.10 Sandboxing . . . 336
14.11 `sysmount()` . . . 337
14.12 `sysumount()` . . . 338
14.13 XTODO . . . 340
14.14 XTODO . . . 342

15 Devices **347**
15.1 The keyboard . . . 347
15.1.1 Overview . . . 347
15.1.2 Scan code . . . 347
15.1.3 Esc . . . 352
15.1.4 `kbd` . . . 353
15.1.5 `echo()` . . . 355
15.1.6 PC Keyboard (x86) . . . 356
15.1.7 Raspberry Pi USB Keyboard (ARM) . . . 356
15.1.8 Changing keyboard: `/dev/kbmap` . . . 356
15.1.9 External keyboard input: `/dev/kbin` . . . 357
15.2 The screen . . . 357
15.2.1 Overview . . . 357
15.2.2 Text output on a PC (x86) . . . 358
15.2.3 Text output on a Raspberry Pi (ARM) . . . 358
15.3 The console device `/dev/cons` . . . 359
15.3.1 Overview . . . 359

15.3.2 Buffered input . . . 360
15.3.3 Output . . . 362
15.3.4 `print()`, `iprint()`, `pprint()` . . . 362
15.3.5 `/dev/cons` . . . 365
15.3.6 `/dev/consctl` and raw input . . . 368
15.4 The mouse . . . 369
15.4.1 Mouse and keyboard . . . 369
15.4.2 PS2 mouse (x86) . . . 369
15.4.3 USB mouse (ARM) . . . 370
15.5 Other devices . . . 370

16 `Queue` 371

17 Filesystems 393
17.1 The root filesystem . . . 393
17.2 The `/boot/` filesystem . . . 397
17.3 Helpers . . . 398
17.4 SimpleFS . . . 405
17.5 `Block` . . . 405
17.6 Fileservers . . . 416
17.7 `Fcall` . . . 416
17.8 XTODO . . . 416

III Beyond the Basics 417

18 IPC 418
18.1 Notes (signals) . . . 418
18.1.1 `sysnotify()` (arm) . . . 420
18.1.2 `sysnoted()` (arm) . . . 422
18.1.3 `postnote()` . . . 424
18.2 Pipes . . . 426
18.2.1 `syspipe()` . . . 426
18.2.2 `/dev/pipe` . . . 427
18.3 Shared segment and semaphore . . . 435
18.3.1 Segment . . . 435
18.3.2 Semaphore . . . 442
18.4 `sysrendezvous()` . . . 447
18.5 Network . . . 449

19 Debugging Support 450
19.1 Core dumps and broken processes . . . 451
19.2 Stopping a process: `/proc/x/ctl` . . . 452
19.3 Inspecting process memory: `/proc/x/mem` . . . 456
19.4 `/bin/strace` . . . 458
19.5 Scheduler events: `/proc/trace` and `/bin/trace` . . . 466
19.6 `/bin/db` . . . 470
19.7 Other . . . 470

20 Profiling Support **471**
20.1 `/dev/cputime` and `/proc/#/status` . . . 471
20.2 `/bin/time` . . . 472
20.3 `/bin/tprof` . . . 473
20.4 `/bin/prof` . . . 475

21 Userspace System Programs **477**
21.1 `/boot/boot` . . . 477
21.2 `/root/init` . . . 477
21.3 The shell . . . 477
21.3.1 `#!` . . . 477

22 Advanced Topics **480**
22.1 Floating point . . . 480
22.1.1 `FPSaveStatus` . . . 480
22.1.2 `Arch_FPsave` (ARM) . . . 481
22.1.3 Floating point hooks . . . 482
22.1.4 ARM floating point coprocessors (ARM) . . . 485
22.1.5 `coproc.c` (ARM) . . . 485
22.1.6 `vfp.c` (ARM) . . . 488
22.1.7 `mathxxx()` (ARM) . . . 494
22.1.8 `/proc/x/fpregs` . . . 495
22.2 CPU cache . . . 495
22.2.1 Instruction and data cache (ARM) . . . 496
22.2.2 Cache policy . . . 496
22.2.3 Paging and caching (ARM) . . . 496
22.2.4 Memory barriers (ARM) . . . 497
22.2.5 Cache operations (ARM) . . . 497
22.2.6 `cachexxx()` (ARM) . . . 498
22.3 Memory model . . . 500
22.3.1 `arch_coherence()` . . . 501
22.3.2 Memory barriers (ARM) . . . 501
22.4 The Raspberry Pi 2 machine . . . 501
22.4.1 CPU ID . . . 502
22.4.2 MMU . . . 502
22.4.3 TTB . . . 503
22.4.4 Local registers . . . 503
22.4.5 Timer . . . 504
22.4.6 Performance monitor . . . 504
22.5 SMP (ARM) . . . 504
22.5.1 Initialization adjustments . . . 504
22.5.2 Setting `cpu` . . . 512
22.5.3 Concurrency adjustments . . . 513
22.5.4 Timer adjustments . . . 514
22.5.5 `arch_coherence()` (ARM) . . . 514
22.5.6 Cache adjustments . . . 515
22.5.7 IRQ adjustments . . . 519
22.5.8 Cycle counter adjustments . . . 520
22.6 Advanced virtual memory . . . 520
22.6.1 Page coloring . . . 521

22.6.2 Extra segment types . . . 523
22.6.3 arch_kmap() . . . 525
22.6.4 Extra checks . . . 525
22.7 Fast interrupts (FIQ) (ARM) . . . 527
22.7.1 Exception PsrMfiq and associated processor mode . . . 527
22.7.2 IRQfiq and vfiq . . . 528
22.7.3 fiq() . . . 529
22.8 Power management (ARM) . . . 529
22.9 Alignements (ARM) . . . 530
22.10 Real-time scheduling . . . 531
22.11 Fileservers . . . 546
22.11.1 Mnt . . . 547
22.11.2 MntRpc . . . 549
22.11.3 Fcall . . . 549
22.11.4 mntversion() . . . 549
22.11.5 mountrpc() . . . 552
22.11.6 devmnt . . . 554
22.12 Network . . . 557
22.13 Security . . . 557
22.14 Graphics . . . 559

23 Special Fileservers **560**
23.1 Cmdtab . . . 560
23.2 /env/ . . . 562
23.2.1 Egrp . . . 563
23.2.2 /env/ . . . 564
23.3 /proc/ . . . 565
23.3.1 /proc/x/ctl . . . 565
23.3.2 /proc/x/mem . . . 565
23.3.3 /proc/x/status . . . 565
23.3.4 /proc/x/args . . . 566
23.3.5 /proc/x/wait . . . 566
23.3.6 Other /proc/x/xxx . . . 567
23.3.7 XXX . . . 567
23.4 /fd/ . . . 588
23.4.1 sysdup() . . . 588
23.4.2 /fd/ . . . 589
23.5 /sys/ . . . 589
23.5.1 /sys/osversion . . . 591
23.5.2 /sys/config . . . 591
23.5.3 /ys/hosthowner, /sys/hostdomain . . . 591
23.5.4 /sys/sysname . . . 592
23.5.5 /sys/drivers . . . 592
23.5.6 /sys/reboot . . . 593
23.6 /dev/ . . . 600
23.6.1 /dev/random . . . 601

24 Conclusion **605**
24.1 Patterns and techniques . . . 605
24.2 Connections to other books . . . 606
24.3 What this book does not cover . . . 606
24.4 Beyond the Plan 9 kernel . . . 607

A Debugging **608**
A.1 `sysnop()` . . . 608
A.2 Special keys and dumpers . . . 608
A.3 `/bin/ktrace` (ARM) . . . 615
A.4 `/dev/kmesg` . . . 616
A.5 `/dev/kprint` . . . 618
A.6 Remote debugger . . . 619
A.7 Serial line . . . 620
A.7.1 `serialoq` . . . 620
A.7.2 `uartconsinit()` . . . 621
A.7.3 `uartputs()` . . . 621
A.7.4 `consuart` . . . 621
A.7.5 `Uart`, `PhysUart` . . . 622

B Profiling **623**
B.1 `/sys/sysstat` (ARM) . . . 623
B.2 `/dev/kprof` . . . 627
B.3 Statistics . . . 627
B.3.1 `Lock.lockcycles` . . . 627
B.3.2 `TaslockStats` . . . 628
B.3.3 `QlockStats` . . . 628
B.3.4 `Kimage` statistics . . . 628
B.3.5 Scheduling . . . 629
B.3.6 Clock . . . 629

C Error Management **630**
C.1 `panic()` . . . 630
C.2 `error()` . . . 631
C.3 `waserror()`, `nexterror()`, `poperror()` . . . 633
C.4 `syserrstr()` . . . 634

D Utilities **635**
D.1 Memory areas operations . . . 635
D.2 String functions . . . 635
D.3 Conversion functions . . . 637
D.4 Pool allocation . . . 637
D.5 Printf and `Fmt` . . . 637
D.6 Runes, a.k.a unicode . . . 639

E Extra Code **640**
E.1 `arch/` . . . 640
E.2 `buses/` . . . 640
E.3 `concurrency/` . . . 640
E.3.1 `concurrency/portdat_concurrency.h` . . . 640

E.3.2 concurrency/portfns_concurrency.h . . . 641
E.3.3 concurrency/taslock.c . . . 641
E.3.4 concurrency/qlock.c . . . 642
E.3.5 concurrency/ref.c . . . 642
E.4 console/ . . . 643
E.4.1 console/portdat_console.h . . . 643
E.4.2 console/portfns_console.h . . . 643
E.4.3 console/devcons.c . . . 643
E.5 core/ . . . 645
E.5.1 core/portdat_forward.h . . . 645
E.5.2 core/portdat_core.h . . . 645
E.5.3 core/portfns_core.h . . . 646
E.5.4 core/portdat.h . . . 646
E.5.5 core/error.h . . . 646
E.5.6 core/error.c . . . 646
E.5.7 core/portdat.c . . . 646
E.5.8 core/portfns.c . . . 647
E.6 devices/ . . . 647
E.6.1 devices/portdat_devices.h . . . 647
E.6.2 devices/portfns_devices.h . . . 648
E.6.3 devices/keyboard/portkbd.c . . . 648
E.6.4 devices/keyboard/latin1.c . . . 649
E.6.5 devices/keyboard/latin1.h . . . 651
E.6.6 TO LP SPLIT devices/keyboard/devkbin.c . . . 651
E.6.7 TO LPSPLIT devices/keyboard/devkbmap.c . . . 653
E.6.8 TO LPSPLIT devices/mouse/portmouse.c . . . 656
E.6.9 TO LPSPLIT devices/mouse/devmouse.c . . . 659
E.6.10 devices/sys/devroot.c . . . 670
E.6.11 devices/sys/devsys.c . . . 672
E.6.12 TO LPSPLIT devices/sys/devenv.c . . . 673
E.6.13 devices/sys/devpipe.c . . . 679
E.6.14 devices/sys/devproc.c . . . 680
E.7 files/ . . . 682
E.7.1 files/portdat_files.h . . . 682
E.7.2 files/portfns_files.h . . . 684
E.7.3 files/chan.c . . . 684
E.7.4 files/dev.c . . . 686
E.7.5 files/allocb.c . . . 687
E.7.6 files/env.c . . . 688
E.7.7 files/file.c . . . 688
E.7.8 files/mnt.c . . . 688
E.7.9 files/qio.c . . . 689
E.7.10 files/sysfile.c . . . 690
E.8 filesystems/ . . . 692
E.9 init/ . . . 692
E.9.1 init/portfns_init.h . . . 692
E.9.2 init/user/preboot/initcode.c . . . 692
E.9.3 init/user/boot/aux.c . . . 692
E.9.4 init/user/boot/boot.c . . . 692

E.9.5 init/user/boot/boot.h . . . 692
E.9.6 init/user/boot/local.c . . . 692
E.10 lib/ . . . 692
E.10.1 lib/lib.h . . . 692
E.11 memory/ . . . 694
E.11.1 memory/portdat_memory.h . . . 694
E.11.2 memory/portfns_memory.h . . . 697
E.11.3 memory/portdat_memory.c . . . 697
E.11.4 memory/xalloc.c . . . 697
E.11.5 memory/pool.c . . . 697
E.11.6 memory/alloc.c . . . 698
E.11.7 memory/fault.c . . . 699
E.11.8 memory/page.c . . . 701
E.11.9 memory/segment.c . . . 702
E.11.10 memory/swap.c . . . 703
E.11.11 memory/sysmemory.c . . . 703
E.12 misc/ . . . 704
E.12.1 misc/portdat_misc.h . . . 704
E.12.2 misc/portfns_misc.h . . . 704
E.12.3 misc/parse.c . . . 704
E.12.4 misc/random.c . . . 705
E.13 interrupts/ . . . 705
E.13.1 interrupts/portfns_interrupts.h . . . 705
E.14 processes/ . . . 705
E.14.1 processes/portdat_processes.h . . . 705
E.14.2 processes/portfns_processes.h . . . 708
E.14.3 processes/portdat_processes.c . . . 708
E.14.4 time/portfns_time.h . . . 708
E.14.5 time/portdat_time.h . . . 709
E.14.6 time/alarm.c . . . 709
E.14.7 time/portclock.c . . . 709
E.14.8 processes/pgrp.c . . . 710
E.14.9 processes/proc.c . . . 711
E.14.10 time/tod.c . . . 715
E.14.11 processes/sysproc.c . . . 716
E.14.12 processes/syssema.c . . . 716
E.15 security/ . . . 718
E.15.1 security/portfns_security.h . . . 718
E.15.2 security/auth.c . . . 718
E.16 syscalls/ . . . 719
E.16.1 syscalls/systab.h . . . 719
E.16.2 syscalls/syscallfmt.c . . . 719
E.16.3 syscalls/systab.c . . . 719
E.17 ../include/arch/arm/ . . . 720
E.17.1 ../include/arch/arm/ureg.h . . . 720
E.18 arch/arm/ . . . 720
E.18.1 arch/arm/dat_arch.h . . . 720
E.18.2 arch/arm/arm.h . . . 720
E.18.3 arch/arm/arminstr.ha . . . 721

E.18.4 arch/arm/io.h . . . 722
E.18.5 arch/arm/coproc.c . . . 722
E.18.6 arch/arm/vcore.c . . . 724
E.19 concurrency/arm/ . . . 730
E.19.1 concurrency/arm/spl.s . . . 730
E.19.2 concurrency/arm/tas_raspi1.s . . . 730
E.19.3 concurrency/arm/tas_raspi2.s . . . 730
E.19.4 concurrency/arm/concurrency_raspi1.c . . . 730
E.19.5 concurrency/arm/concurrency_raspi2.c . . . 731
E.20 core/arm/ . . . 732
E.20.1 core/arm/dat.h . . . 732
E.20.2 core/arm/dat_core.h . . . 732
E.20.3 core/arm/dat_forward.h . . . 733
E.20.4 core/arm/fns.h . . . 733
E.21 buses/arm/ . . . 736
E.22 devices/ . . . 736
E.22.1 devices/keyboard/arm/kbd.c . . . 736
E.22.2 devices/mouse/arm/mouse.c . . . 736
E.22.3 devices/sys/arm/devarch.c . . . 736
E.23 init/arm/ . . . 740
E.23.1 init/arm/start.s . . . 740
E.23.2 init/arm/startv6.s . . . 741
E.23.3 init/arm/startv7.s . . . 741
E.23.4 init/arm/main.c . . . 741
E.23.5 init/arm/raspi1.c . . . 744
E.23.6 init/arm/raspi2.c . . . 745
E.23.7 init/user/preboot/arm/init9.s . . . 746
E.23.8 init/arm/rebootcode.s . . . 746
E.24 memory/arm/ . . . 746
E.24.1 memory/arm/dat_memory.h . . . 746
E.24.2 memory/arm/mem.h . . . 747
E.24.3 memory/arm/mmu.c . . . 748
E.24.4 memory/arm/mmuinvalidate.s . . . 749
E.24.5 memory/arm/cache_raspi1.s . . . 749
E.24.6 memory/arm/cache_raspi2.s . . . 749
E.25 interrupts/arm/ . . . 750
E.25.1 interrupts/arm/trap.c . . . 750
E.25.2 interrupts/arm/fault_helpers.s . . . 751
E.25.3 interrupts/arm/lexception.s . . . 751
E.26 time/arm/ . . . 752
E.26.1 time/arm/dat_time.h . . . 752
E.26.2 time/arm/clock.c . . . 752
E.26.3 time/arm/time_raspi1.s . . . 754
E.26.4 time/arm/time_raspi2.s . . . 754
E.27 processes/arm/ . . . 754
E.27.1 processes/arm/dat_processes.h . . . 754
E.27.2 processes/arm/arch.c . . . 754
E.27.3 processes/arm/vfp3.c . . . 755
E.27.4 processes/arm/lproc.s . . . 756

E.27.5 processes/arm/switch.s . . . 757
E.28 syscalls/arm/ . . . 757
E.28.1 syscalls/arm/syscall.c . . . 757
E.29 Extra files . . . 757
E.29.1 init/arm/bootconf.c . . . 757
E.29.2 devices/storage/arm/emmc.c . . . 762
E.29.3 init/arm/watchdog.c . . . 770
E.29.4 buses/arm/dwcotg.h . . . 772
E.29.5 buses/arm/devusb.c . . . 781
E.29.6 buses/arm/usbdwc.c . . . 810
E.29.7 buses/arm/dma.c . . . 829
E.29.8 buses/arm/uartmini.c . . . 835

Glossary **845**

Indexes **847**

References **910**

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 `hello` 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. 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, 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`.

The graphics stack and the networking stack 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.

That said, Plan 9's design has trade-offs. The "everything is a file" philosophy 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 trade-offs: 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 (or its modern incarnation xv6) is too simple; there is no graphics, no network. Even though this book also does not cover graphics or networking (those are in GRAPHICS book [Pad16d] and NETWORK book [Pad16e]), the kernel presented here is the same one that supports those subsystems.
- XINU has a network stack but the kernel itself is too simple with no virtual memory for instance
- Minix is fairly small but it does not provide a simple windowing system as it relies on X11 (which is far more complicated than Plan 9 Rio windowing system)
- Linux 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.

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. See https://www.principia-softwarica.org/getting-started.html. Then you need to compile using a regular C compiler (e.g. gcc) 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

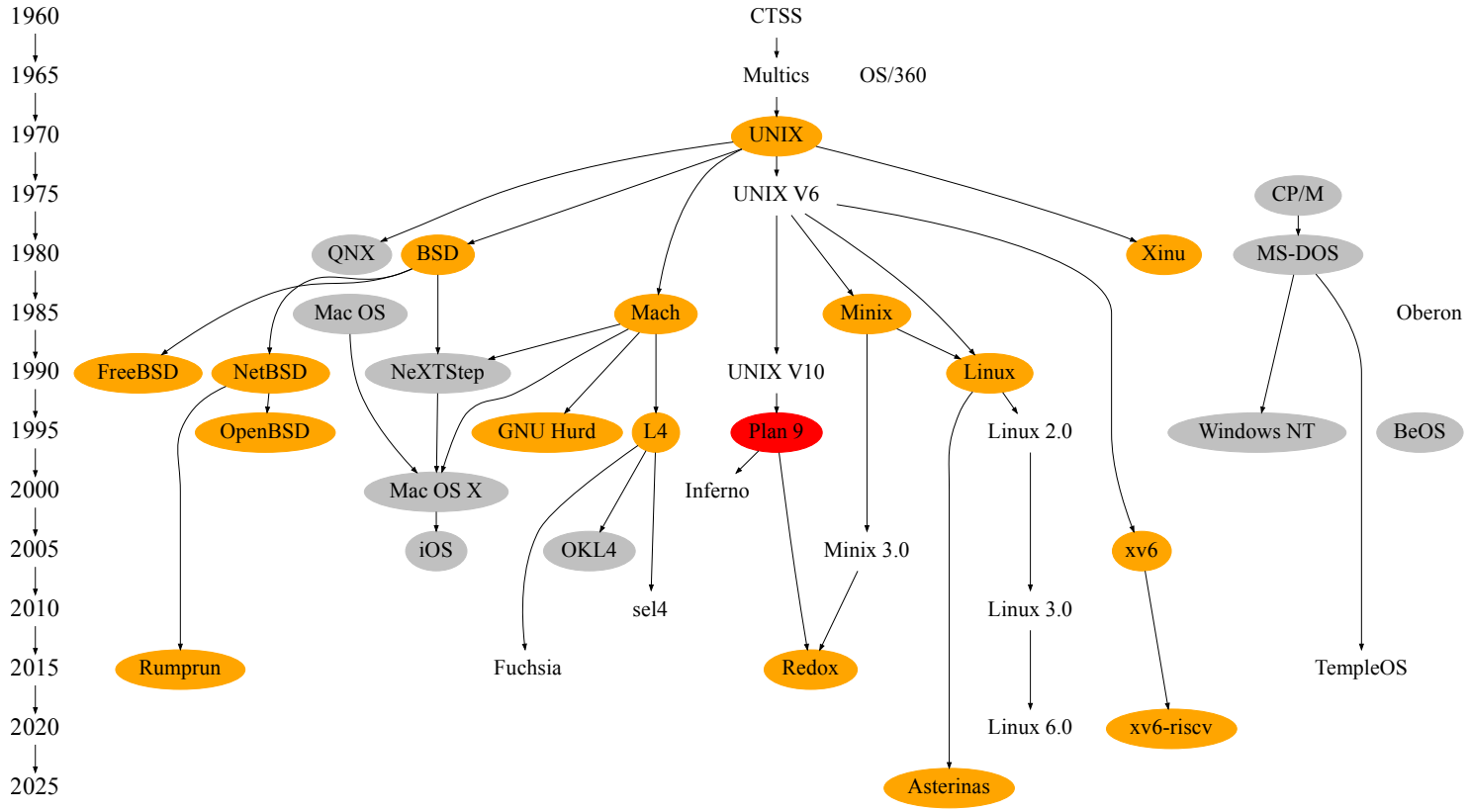


Figure 1.1: Kernels timeline

installed in the previous step. You can also 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.

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. The goal of this book is to explain with full details a kernel but we assume you already have a vague idea of how an operating system works and so that you are familiar with concepts such as virtual memory, critical regions, interrupts, system calls, etc. I assume you already know most of the theory; this book is here to cover the practice.

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.

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, the 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 those subjects. I assume you already know what is an operating system and we will now focus on the concrete services (API) a kernel provides to the 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 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`[41f] array shown later).

⟨`lib_core/libc/9syscall/sys.h` 22a⟩≡
```
#define NOP     0
⟨sys.h process syscalls 22b⟩
⟨sys.h memory syscalls 22c⟩
⟨sys.h file syscalls 22d⟩
⟨sys.h directory syscalls 23a⟩
⟨sys.h namespace syscalls 23b⟩
⟨sys.h time syscalls 23c⟩
⟨sys.h ipc syscalls 23d⟩
⟨sys.h concurrency syscalls 23e⟩
⟨sys.h special file syscalls 24a⟩
⟨sys.h security syscalls 24b⟩
#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.

The process syscalls mirror the UNIX model: `RFORK` creates a new process (a generalization of UNIX `fork`), `EXEC` replaces the current program, `EXITS` terminates, and `AWAIT` waits for a child (like UNIX `wait`).

⟨sys.h *process syscalls* 22b⟩≡ (22a)
```
#define RFORK       1
#define EXEC        2
#define EXITS       3
#define AWAIT       4
```

Plan 9 cleans up UNIX here too. 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.

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 "file not found"—the string says it directly.

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`. The library requests large chunks of memory from the kernel via `brk` and then subdivides them.

⟨sys.h *memory syscalls* 22c⟩≡ (22a)
```
#define BRK     5
```

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 five calls. 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. This is another cleanup: 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.

⟨sys.h *file syscalls* 22d⟩≡ (22a)
```
#define OPEN        6
```

```
#define CLOSE       7
#define PREAD       8
#define PWRITE      9
#define SEEK        10
```

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). `FD2PATH` is a Plan 9 addition that returns the file name associated with an open file descriptor—useful because in Plan 9, with per-process namespaces, a file descriptor is often more reliable than a path. Note `CREATE` with the full spelling—in UNIX, the syscall was famously called `creat` without the final 'e'. When asked what he would change about UNIX, Ken Thompson once answered: "I'd spell creat with an e." Plan 9 finally fixed this.

⟨`sys.h` *directory syscalls* 23a⟩≡ (22a)

```
#define CREATE      11
#define REMOVE      12
#define CHDIR       13
#define FD2PATH     14
#define STAT        15
#define FSTAT       16
#define WSTAT       17
#define FWSTAT      18
```

The namespace syscalls below 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. This is what makes `/dev/cons`, `/dev/draw`, etc. work cleanly: each process can have its own `/dev` populated by different servers.

⟨`sys.h` *namespace syscalls* 23b⟩≡ (22a)

```
#define BIND        19
#define MOUNT       20
#define UNMOUNT     21
```

⟨`sys.h` *time syscalls* 23c⟩≡ (22a)

```
#define SLEEP       22
#define ALARM       23
```

The IPC syscalls cover several mechanisms. `PIPE` creates a pipe—one of UNIX's greatest inventions. `NOTIFY`/`NOTED` handle notes, Plan 9's equivalent of UNIX signals. The `SEG` family (`SEGATTACH`, `SEGDETACH`, etc.) lets processes attach shared memory segments, similar to UNIX `mmap`.

⟨`sys.h` *ipc syscalls* 23d⟩≡ (22a)

```
#define PIPE        26

#define NOTIFY      24
#define NOTED       25

#define SEGATTACH   27
#define SEGDETACH   28
#define SEGFREE     29
#define SEGFLUSH    30
#define SEGBRK      31
```

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`) is implemented.

⟨`sys.h` *concurrency syscalls* 23e⟩≡ (22a)

```
#define RENDEZVOUS  32
#define SEMACQUIRE  33
#define SEMRELEASE  34
#define TSEMACQUIRE 35
```

⟨sys.h *special file syscalls* 24a⟩≡ (22a)
```
#define DUP         36
```

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* 24b⟩≡ (22a)
```
#define FVERSION    37
#define FAUTH       38
```

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. The file server abstraction absorbs what would otherwise be dozens of special-purpose syscalls.

2.3 `helloraw.c` and `helloraw.s`

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 24c⟩≡
```
#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 places the syscall number (`PWRITE`, which is `9` from `sys.h`) in register `R0` and executes a `SWI` (software interrupt) instruction to trap into the kernel. The assembly version makes this mechanism fully visible:

⟨helloraw.s 24d⟩≡
```
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
```

[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.

```
  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
```

This confirms that our small program generates exactly the two system calls we expected: `PWRITE` and `EXITS`. We will see in Chapters 7 and 6 how the kernel handles the `SWI` instruction, dispatches the syscall number through `systab`, 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 25⟩≡

```
  #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) {
        exec(buf, args);
        exits("exec failed");
      }
      await(buf, sizeof(buf));
    }
    exits(0);
  }
```

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)* 26a⟩≡ (747)
```
#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)* 26b⟩≡ (747)
```
#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)* 26c⟩≡ (720a)
```
//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
};
```

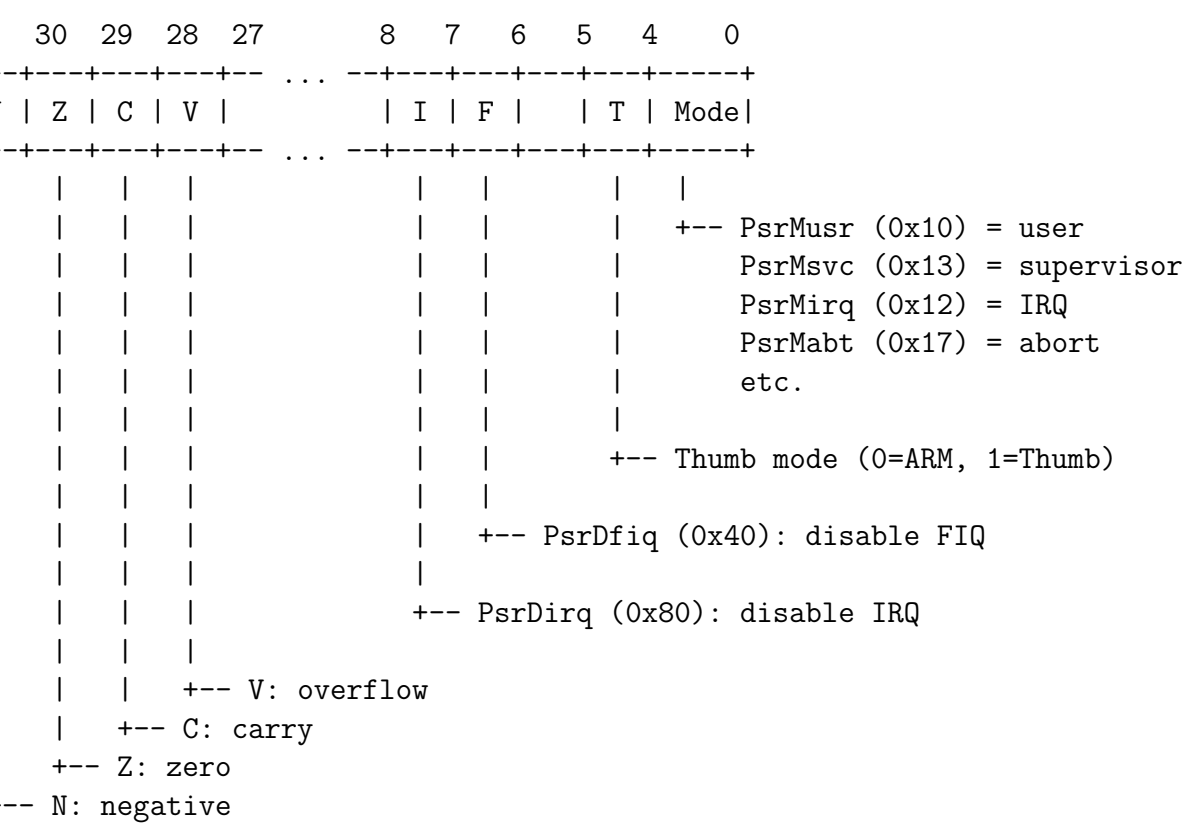
⟨*function* `getcallerpc`*(arm)* 27⟩≡ (757a)

```
TEXT getcallerpc(SB), 1, $-4
    MOVW    0(R13), R0
    RET
```

Program Status Register (`PSR`)

In addition to the general-purpose registers, the ARM has a Program Status Register (PSR) 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—this is the `Ureg.psr` field—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 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)* 28a⟩≡ (720c)
```
// Last 5 bits of PSR
#define PsrMusr     0x00000010  /* user mode */
#define PsrMsvc     0x00000013  /* 'protected mode for OS' */
⟨PsrMode other cases(arm) 143a⟩

#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)* 28b⟩≡ (720c)
```
#define PsrDirq     0x00000080      /* disable IRQ interrupts */
⟨PsrDisable other cases(arm) 527d⟩
```

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). Each exception type has its own entry in the vector table and its own processor mode.

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) to use "high vectors" at `0xFFFF0000`. This is more convenient both because 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).

Of course, `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)* 28c⟩≡ (720c)
```
#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)* 29a⟩≡ (720c)
```
#define CpSC        15          /* System Control */
⟨Coprocessors other cases(arm) 485b⟩
```

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)* 29b⟩≡ (720c)
```
#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) 496a⟩
```

⟨*type CpCONTROL opcode2(arm)* 29c⟩≡ (720c)
```
/*
 * CpCONTROL op2 codes, op1==0, Crm==0.
 */
#define CpMainctl   0
⟨CpCONTROL.opcode2 other cases(arm) 489b⟩
```

⟨*type CpCONTROL CpMainctl(arm)* 29d⟩≡ (720c)
```
/*
 * 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) 496b⟩
```

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.

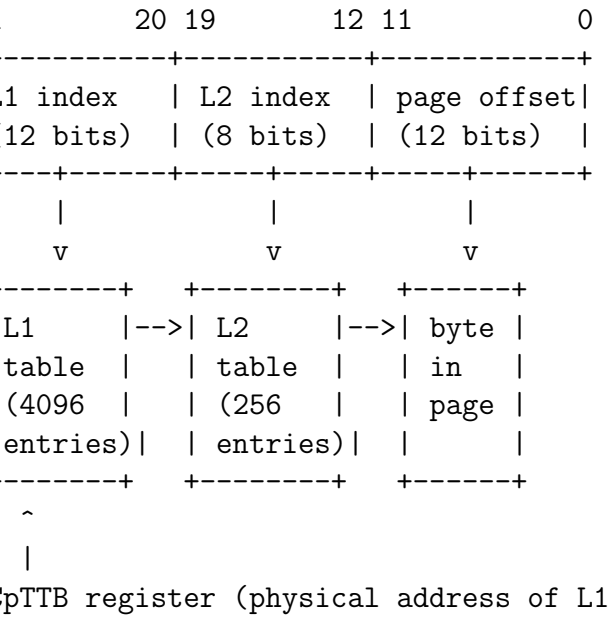
The ARM MMU supports two granularities of mapping. An L1 (first-level) table entry can either map a full 1 MiB section directly to a physical address, or point to an L2 (second-level) table that maps individual 4 KiB pages:

```
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`).

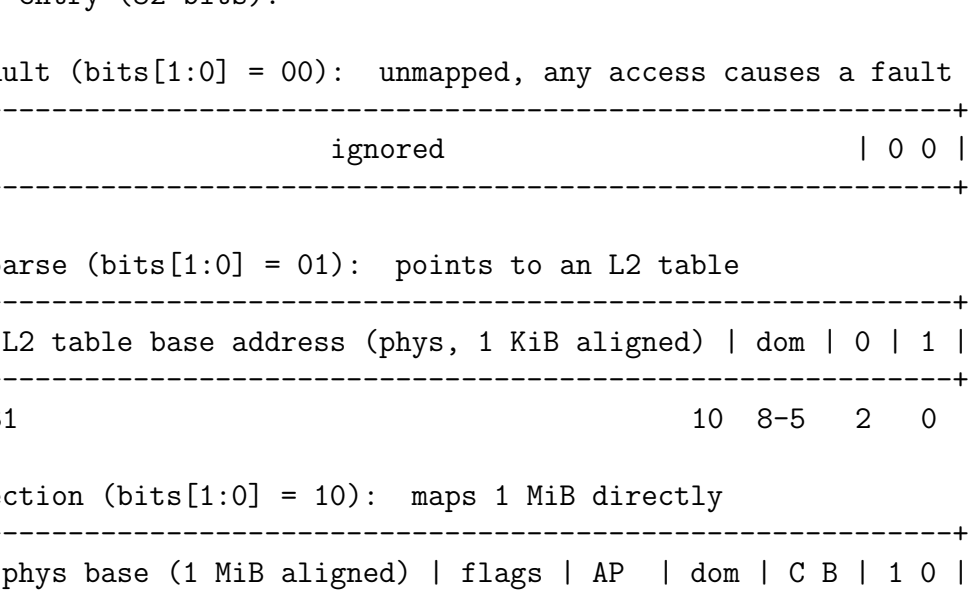
The basic unit of memory managed at the finest level is a page, which on ARM (and on x86) is 4 KiB:

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:



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:



```
+-------------------------------------------------------------+
 31                        20  19-12  11-10  8-5   3  2   1  0
```

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.

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) selects one of 16 domains; the domain register then says whether accesses to that domain are checked ("client"), unchecked ("manager"), or faulted ("no access"), providing a coarse-grained override. 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).

⟨*constant* `BY2PG` *(arm)* 31a⟩≡ (747)
```
#define BY2PG       (4*KiB)         /* bytes per page */
```

⟨*constant* `PGSHIFT` *(arm)* 31b⟩≡ (747)
```
#define PGSHIFT     12          /* log(BY2PG) */
```

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`, which is really `LDR`/`STR` under the hood). This is simpler than the x86 approach, which uses special `IN`/`OUT` instructions for I/O ports. 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)* 31c⟩≡ (747)
```
#define VIRTIO      0x7E000000      /* i/o registers */
```

⟨*constant* `IOSIZE` *(arm)* 31d⟩≡ (747)
```
#define IOSIZE      (16*MiB)
```

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

⟨*constant* `SYSTIMERS` *(arm)* 31e⟩≡ (753)
```
#define SYSTIMERS (VIRTIO+0x3000)
```

⟨*constant* `ARMTIMER` *(arm)* 31f⟩≡ (753)
```
#define ARMTIMER (VIRTIO+0xB400)
```

⟨*constant* `INTREGS` *(arm)* 31g⟩≡ (750b)
```
#define INTREGS     (VIRTIO+0xB200)
```

⟨*constant* `DMAREGS` *(arm)* 31h⟩≡ (834)
```
#define DMAREGS (VIRTIO+0x7000)
```

⟨*constant* `POWERREGS` *(arm)* 31i⟩≡ (772b)
```
#define POWERREGS   (VIRTIO+0x100000)
```

⟨*constant* AUXREGS*(arm)* 32a⟩≡ (843b)
```
#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. 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)* 32b⟩≡ (729)
```
#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)* 32c⟩≡ (729)
```
// 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;
    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 windowing system uses this framebuffer.

⟨*constant* FRAMEBUFFER*(arm)* 32d⟩≡ (747)
```
#define FRAMEBUFFER 0xC0000000      /* video framebuffer */
```

Input devices: USB keyboard and mouse

⟨*constant* USBREGS*(arm)* 32e⟩≡ (828)
```
#define  USBREGS (VIRTIO+0x980000)
```

Storage device: SD card

⟨*constant* EMMCREGS*(arm)* 32f⟩≡ (770a)
```
#define EMMCREGS    (VIRTIO+0x300000)
```

Other devices

⟨*constant* GPIOREGS*(arm)* 32g⟩≡ (843b)
```
#define GPIOREGS    (VIRTIO+0x200000)
```

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.

2.6 C and assembly

The kernel is written almost entirely in C, compiled by the Plan 9 C compiler `5c` (see the COMPILER book [Pad16b]). 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. This simplifies the kernel's context-switch code, because the `Label` structure (shown later) only needs to store `SP` and `PC`—the caller has already saved any registers it cares about. Note that this `Label` (just `SP`+`PC`) is used for voluntary context switches between kernel threads. 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.

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)
          +-------------+
```

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. This is why the `Syscall` typedef takes a `ulong*`—it points to the arguments on the user's stack.

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* 34a⟩≡ (? 0—1) 34b▷
```
#define ROUND(s, sz)  (((s)+(sz-1)) & ~(sz-1))
```

⟨*portdat.h macros* 34b⟩+≡ (? 0—1) ◁34a 34c▷
```
#define MASK(v) ((1UL << (v)) - 1)  /* mask 'v' bits wide */
```

⟨*portdat.h macros* 34c⟩+≡ (? 0—1) ◁34b 34d▷
```
#define MIN(a, b) ((a) < (b)? (a): (b))
```

⟨*portdat.h macros* 34d⟩+≡ (? 0—1) ◁34c 34e▷
```
// used in bcm/mmu.c
#define FEXT(d, o, w) (((d)>>(o)) & ((1<<(w))-1))
```

⟨*portdat.h macros* 34e⟩+≡ (? 0—1) ◁34d 102a▷
```
#define HOWMANY(x, y) (((x)+((y)-1))/(y))
#define ROUNDUP(x, y) (HOWMANY((x), (y))*(y)) /* ceiling */
```

⟨*constant* KiB*(arm)* 34f⟩≡ (747)
```
#define KiB     1024u           /* Kibi 0x0000000000000400 */
```

⟨*constant* MiB*(arm)* 34g⟩≡ (747)
```
#define MiB     1048576u        /* Mebi 0x0000000000100000 */
```

⟨*constant* GiB*(arm)* 34h⟩≡ (747)
```
#define GiB     1073741824u     /* Gibi 0000000040000000 */
```

2.7 Code organization

The kernel source is split across many files organized by subsystem: `core/`, `concurrency/`, `memory/`, `processes/`, `files/`, `filesystems/`, `time/`, `interrupts/`, `devices/`, `init/`, `console/`, etc. These directories map directly to the chapters in this book.

Each subsystem has both portable code (in `port/`) and architecture-specific code (in `bcm/` for ARM).

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* 35a⟩≡ (843b 834 828 807b 772b 770a 761b 757b 755 754e 753 750b 748b 745 744b 742b 740a 736 731 729 ? ? ? ? ?

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

2.7.3 Top directories

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)* 35b⟩≡ (732b)

```
#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)* 35c⟩≡ (733b)

```
#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.

The Plan 9 kernel is monolithic—device drivers, the file system switch, 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 file server `fossil`). This borrows a key idea from microkernel designs: keep the kernel small and move services to user space. In fact, the entire kernel is only about 30 000 lines of C—remarkably small for a complete operating system.

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` 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.

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.[2] The kernel binary is loaded at physical `0x8000` (virtual `KTZERO`).[3] 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`.

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)* 36a⟩≡ (747)
```
#define KSEG0       0x80000000      /* kernel segment */
```

⟨*constant* `KZERO` *(arm)* 36b⟩≡ (747)
```
#define KZERO       KSEG0           /* kernel address space */
```

⟨*constant* `KTZERO` *(arm)* 36c⟩≡ (747)
```
#define KTZERO      (KZERO+0x8000)      /* kernel text start */
```

[2] 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.

[3] 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 uses the memory below `0x8000` for its own data structures during boot. The kernel in turn uses some of this space (e.g., `CPUADDR` at `0x2000`) for early kernel structures like the first `Cpu`.

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 (one page above zero), and the user stack grows downward from USTKTOP:

⟨*constant* UZERO*(arm)* 37a⟩≡ (747)
```
#define UZERO       0           /* user segment */
```

⟨*constant* UTZERO*(arm)* 37b⟩≡ (747)
```
#define UTZERO      (UZERO+BY2PG)       /* user text start */
```

⟨*constant* USTKTOP*(arm)* 37c⟩≡ (747)
```
#define USTKTOP     0x20000000      /* user segment end +1 */
```

⟨*constant* USTKSIZE*(arm)* 37d⟩≡ (747)
```
#define USTKSIZE    (8*1024*1024)       /* user stack size */
```

The following diagram shows the full virtual address space. The kernel and user portions coexist in every process's page table (though the kernel portion is shared and identical across all processes):

```
                ...
            +---------------------+
            | exception vectors   |
0xFFFF0000  +---------------------+  <-- HVECTORS
                ...
            +---------------------+
            | GPU framebuffer     |
0xC0000000  +---------------------+  <-- FRAMEBUFFER
                ...
            +---------------------+
            | kernel heap         |  xalloc pool (large allocs:
            |  (grows up)         |   Proc[], Page[], etc.)
            |                     |  then malloc pool
            +---------------------+
            | kernel bss          |
            +---------------------+
            | kernel data         |  globals: conf, systab, devtab,
            |                     |  xlists, palloc, procalloc...
            +---------------------+
            | kernel text         |  main(), sched(), syscall
            |                     |  handlers, device drivers...
0x80008000  +---------------------+  <-- KTZERO
                ...
            +---------------------+
            | Cpu0 struct         |  CPU 0's state
            | + kernel boot stack |  (1 page = CPUSIZE)
0x80002000  +---------------------+  <-- CPUADDR
                ...
0x80000000  +=====================+  <-- KZERO
            |  KERNEL / USER      |
            |    BOUNDARY         |
            +=====================+
                ...
            +---------------------+
            | memory-mapped I/O   |  device registers
```

```
            | (16 MiB = IOSIZE)   |  (SYSTIMERS, INTREGS, ...)
0x7E000000  +---------------------+  <-- VIRTIO
                      ...
                (unmapped gap)
                      ...
0x20000000  +---------------------+  <-- USTKTOP
            | user stack          |  grows down
            |  (8 MiB max)        |  (USTKSIZE)
            +---------------------+
                      ...
            +---------------------+
            | user heap           |  managed by brk()
            |  (grows up)         |
            +---------------------+
            | user bss            |
            +---------------------+
            | user data           |
            +---------------------+
            | user text           |
0x00001000  +---------------------+  <-- UTZERO
            | (unmapped page)     |  null pointer trap
0x00000000  +---------------------+  <-- UZERO
```

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` and `malloc`) 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`[41f] (the syscall dispatch table), `devtab`[54b] (the device driver table), `conf`[46b] (hardware configuration), `palloc`[90d] (the page allocator state), and `procalloc`[110c] (the process table). Finally, the `Cpu`[48a] structure for CPU 0 occupies a single page at `CPUADDR`[40c], below the kernel text, in the small gap between `KZERO`[36b] and `KTZERO`[36c].

On multi-processor systems (e.g., Raspberry Pi 2), each additional CPU has its own `Cpu` struct allocated elsewhere; the `cpus`[49a] array and the dedicated register `R10` allow each processor to find its own.

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  +------------------+
```

The kernel's `mmuinit()`[249c] 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` together. 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()`[167] 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` 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)* 40a⟩≡ (732b) 40b▷
```
// 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)* 40b⟩+≡ (732b) ◁40a
```
// 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`:

⟨*constant* `CPUADDR` *(arm)* 40c⟩≡ (747)
```
#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)* 40d⟩≡ (733b)
```
#define PADDR(va)   PTR2UINT(PHYSDRAM | ((uintptr)(va) & ~KSEGM))
```

⟨*macro PADDR bis(arm)* 40e⟩≡ (721)
```
#define PADDR(va)   (((va) & ~KSEGM))
```

⟨*constant* `PHYSDRAM` *(arm)* 40f⟩≡ (747)
```
#define PHYSDRAM    0
```

⟨*constant* `KSEGM` *(arm)* 40g⟩≡ (747)
```
/* mask to check segment; good for 1GB dram */
#define KSEGM       0xC0000000
```

⟨*macro* `KADDR` *(arm)* 40h⟩≡ (733b)
```
#define KADDR(pa)   UINT2PTR(KZERO    | ((uintptr)(pa) & ~KSEGM))
```

⟨*macro* PTR2UINT*(arm)* 41a⟩≡ (733b)
```
#define PTR2UINT(p) ((uintptr)(p))
```

⟨*macro* UINT2PTR*(arm)* 41b⟩≡ (733b)
```
#define UINT2PTR(i) ((void*)(i))
```

⟨*function* arch_cankaddr*(arm)* 41c⟩≡ (748b)
```
/*
 * 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 254c.

2.8.3 Syscall overview (ARM)

We saw in the `helloraw.s` example how a user program invokes a syscall: 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`, the syscall dispatch table. Each entry in `systab` is a function pointer with a uniform signature:

⟨*typedef Syscall* 41d⟩≡ (41e)
```
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 casts and extracts what it needs from this generic array.

⟨syscalls/systab.h 41e⟩≡
```
// used by systab.c but also by trap.c for certain codes
#include "../../../../lib_core/libc/9syscall/sys.h"

⟨typedef Syscall 41d⟩

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` to the corresponding kernel function. This is the only point in the kernel where all syscalls are gathered together:

⟨*global* systab 41f⟩≡ (719b)
```
Syscall *systab[] = {
    [NOP]     sysnop,
    ⟨systab process syscalls 109a⟩
    ⟨systab memory syscalls 74⟩
    ⟨systab file syscalls 272a⟩
    ⟨systab directory syscalls 299a⟩
    ⟨systab namespace syscalls 317a⟩
    ⟨systab time syscalls 222a⟩
    ⟨systab ipc syscalls 418a⟩
    ⟨systab concurrency syscalls 418b⟩
```

```
    ⟨systab special file syscalls 588a⟩
    ⟨systab security syscalls 557c⟩
    [ERRSTR]    syserrstr,
};
```

Uses syserrstr() 634c and sysnop() 42b.

⟨*global* nsyscall 42a⟩≡ (719b)

```
int nsyscall = nelem(systab);
```

Uses nelem 692c, nsyscall 42a, and systab 41f.

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

⟨*syscall nop* 42b⟩≡ (716a)

```
// void nop(void);
long
sysnop(ulong*)
{
    print("Hello World\n");
    return 0;
}
```

Uses print.

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
```

Note the distinction between `Ureg` and `Label`: when entering the kernel from user space (step 4), `_vsvc` saves all user registers into a `Ureg` because the user program did not follow any calling convention—its registers could hold anything. Later, if the kernel decides to context switch to another process (a voluntary switch between kernel threads), it uses `Label`, which saves only `SP` and `PC`—because kernel code follows the caller-save convention and the caller has already saved any registers it needs.

The exception vector table at `HVECTORS` is set up by `arch__trapinit()`[256e] 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`[48a] 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`[??] structure (`up`[51d] 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

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.

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

Q: Why `SWI $0`? Does the argument to `SWI` matter? A: 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`.

Q: Why split the 4 GiB address space between kernel and user? Could each have its own full 4 GiB? A: Some systems do give user space the full address range and switch page tables entirely when entering the kernel. For example, xv6 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.

Q: Can a page fault occur inside the kernel? Can kernel memory be swapped out? A: The kernel's own pages are never swapped and are always mapped (via the L1 section entries set up by `mmuinit()`[249c]). 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.

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

Q: Why is `CPUADDR` at `0x2000`, below the kernel text at `0x8000`? A: 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 NSFAQ

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.

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.

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`, `Cpu`, `Label`, etc.) and concurrency primitives (locks, `sleep`/`wakeup`, `Rendez`). 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` and `Dev` 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`[26c], the processor modes (`PsrMode`), the address typedefs, and the syscall dispatch table (`systab`[41f]). Here I will cover the remaining structures that appear everywhere in the kernel: `Conf`[46a] (hardware configuration), `Cpu`[48a] (per-processor state), `Label`[205d] (context switch), and `Ref`[??] (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` 46a⟩≡ (645)
```
struct Conf
{
    ulong ncpu;    /* processors */
    Confmem mem[4];   /* physical memory */

    ulong nproc;    /* processes */
    ⟨Conf other fields 170a⟩
};
```

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` 46b⟩≡ (646b)
```
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` and `palloc`):

⟨*struct* `Confmem` 46c⟩≡ (645)
```
// memory "bank"
struct Confmem
{
    phys_addr base;
    uintptr limit; // used by bcm only. phys_addr too?
    ulong npage;
    ⟨Confmem other fields 456c⟩
};
```

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 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 bootloader passes configuration parameters to the 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`[77c]). 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* 47a⟩≡ (646b)
```
// conf (boot) parameters *e.g. { "*kernelpercent*" => "60" }
// hash<string, string>
char *confname[MAXCONF];
char *confval[MAXCONF];
```
Uses MAXCONF 47b.

⟨*constant* MAXCONF 47b⟩≡ (645)
```
#define MAXCONF         64
```

⟨*global* nconf 47c⟩≡ (646b)
```
// Hashtbl.length(confname)
int nconf;
```

⟨*function* getconf 47d⟩≡
```
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 48a⟩≡ (645)
```
//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 56f⟩
      // ref<Proc>, or None if halting?
    Proc* proc;     /* current process on this processor */

    struct Arch_Cpu;

    ⟨Cpu time fields 48c⟩
    ⟨Cpu stat fields 623a⟩
    ⟨Cpu other fields 71c⟩

    // 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)* 48b⟩≡ (747)
```
#define CPUSIZE BY2PG
```

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

⟨Cpu *time fields* 48c⟩≡ (48a) 48d▷
```
ulong ticks;      /* of the clock since boot time */
```

⟨Cpu *time fields* 48d⟩+≡ (48a) ◁48c
```
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 a Ken C compiler extension, 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)* 48e⟩≡ (732c)
```
//coupling: do not change the first field! Some assembly code assumes this order
struct Arch_Cpu {
    ⟨Cpu Arch mmu fields(arm) 161k⟩
    ⟨Cpu Arch float fields(arm) 485d⟩
    ⟨Cpu Arch save of first five registers fields(arm) 257c⟩
};
```

The cpu global must be per-processor: 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. 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 above). On x86, where registers are scarce, a different trick involving the page table is used instead.

⟨*global* cpu*(arm)* 48f⟩≡ (732b)
```
extern register Cpu* cpu;           /* R10 */
```

⟨*global* cpus 49a⟩≡ (646b)
```
/*
 * 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)* 49b⟩≡ (747)
```
#define MAXCPUS     4           /* max # cpus system can run */
```

⟨*macro* CPUS 49c⟩≡ (645)
```
#define CPUS(n)  (cpus[n])
```

The `Active` structure tracks global multiprocessor state that does not belong to any single `Cpu`. The `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` to serialize concurrent updates from multiple CPUs.

⟨*struct* Active 49d⟩≡ (705c)
```
struct Active
{
    // array<bool> (coupling: sizeof(int) must be >= MAXCPUS)
    int cpus;      /* bitmap of active CPUs */
    bool exiting;    /* shutdown */
    ⟨Active other fields 594f⟩
    // extra
    Lock;
};
```

⟨*global* active 49e⟩≡ (708a)
```
struct Active active;
```
Uses Active 49d.

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 49f⟩≡ (705c)
```
struct Proc
{
//------------------------------------------------------------------
//coupling: have to be first! Some assembly code assumes this order.
//------------------------------------------------------------------
    ⟨Proc assembly fields 205e⟩
//------------------------------------------------------------------
// State
//------------------------------------------------------------------
    ⟨Proc state fields 51a⟩
```

```
//------------------------------------------------------------------------
// Memory
//------------------------------------------------------------------------
    ⟨Proc memory fields 100b⟩

    struct Arch_Proc;
//------------------------------------------------------------------------
// Scheduling
//------------------------------------------------------------------------
    ⟨Proc scheduling fields 198c⟩
//------------------------------------------------------------------------
// Files
//------------------------------------------------------------------------
    ⟨Proc files fields 52a⟩
//------------------------------------------------------------------------
// Notes
//------------------------------------------------------------------------
    ⟨Proc notes fields 419h⟩
//------------------------------------------------------------------------
// Process hierarchy
//------------------------------------------------------------------------
    ⟨Proc hierarchy fields 130b⟩
//------------------------------------------------------------------------
// Synchronization
//------------------------------------------------------------------------
    ⟨Proc synchronization fields 70f⟩
//------------------------------------------------------------------------
// Error management
//------------------------------------------------------------------------
    ⟨Proc error managment fields 632b⟩
//------------------------------------------------------------------------
// Stats, profiling
//------------------------------------------------------------------------
    ⟨Proc stats and profiling fields 471b⟩
//------------------------------------------------------------------------
// Debugging (the kernel itself)
//------------------------------------------------------------------------
    ⟨Proc debugging fields 59b⟩
//------------------------------------------------------------------------
// For debugger, strace
//------------------------------------------------------------------------
    ⟨Proc debugger fields 147e⟩
//------------------------------------------------------------------------
// Other
//------------------------------------------------------------------------
    ⟨Proc other fields 139a⟩
//------------------------------------------------------------------------
// Extra
//------------------------------------------------------------------------
    ⟨Proc extra fields 63d⟩
};
```

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)* 50⟩≡ (746b)

```
struct Arch_Proc
{
    ⟨Proc Arch memory fields(arm) 162c⟩
};
```

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

⟨`Proc` *state fields* 51a⟩≡ (49f) 109b▷
```
// 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` 51b⟩≡ (705c)
```
/* Process states, Proc.state */
enum Procstate
{
    Dead = 0,
    Running,
    ⟨Procstate cases 63e⟩
};
```

⟨*global* `statename` 51c⟩≡ (712c)
```
// 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",
};
```

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)* 51d⟩≡ (732b)
```
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` 51e⟩≡ (682c)
```
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 428e⟩
    };
    ⟨Chan other fields 272c⟩

    // extra
    Ref; /* the Lock in this Ref is also Chan's lock */
    ⟨Chan extra fields 273b⟩
};
```

Uses __anon_struct_93 51e.

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* 52a⟩≡ (49f) 276b▷

```
// 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 52b⟩≡ (693c)

```
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 52c⟩≡ (693c)

```
/* bits in Qid.type */
enum Qidtype {
  QTFILE = 0x00,    /* plain file */
  QTDIR = 0x80,    /* type bit for directories */
  ⟨Qidtype cases 398b⟩
};
```

⟨*function* `eqqid` 53a⟩≡ (684)

```
bool
eqqid(Qid a, Qid b)
{
    return a.path==b.path && a.vers==b.vers;
}
```

⟨*function* `mkqid` 53b⟩≡ (686)

```
void
mkqid(Qid *q, vlong path, ulong vers, int type)
{
    q->type = type;
    q->vers = vers;
    q->path = path;
}
```

⟨*function* `eqchan` 53c⟩≡ (684)

```
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` 53d⟩≡ (684)

```
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 (`devcons`), the process table (`devproc`), the root file system (`devroot`), the SD card (`devsd`), etc. 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. 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.

⟨*struct* `Dev` 54a⟩≡ (682c)
```
struct Dev
{
    Rune dc; // dev character code, e.g. '/' (devroot), 'e' (devenv), ...
    char* name;

    ⟨Dev methods 264b⟩
};
```

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` 54b⟩≡ (686)
```
// array<ref_own<Dev>>
Dev** devtab = nil;
```

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)* 54c⟩≡ (720b)
```
struct Soc {            /* SoC dependent configuration */
    ulong   dramsize;
    phys_addr physio;
    ⟨Soc bus fields 831e⟩
    ⟨Soc mmu fields 496i⟩
    ⟨Soc other fields 503f⟩
};
```

⟨*global* `soc`*(raspberry pi1)(arm)* 54d⟩≡ (744b)
```
Soc soc = {
    .dramsize   = 512*MiB,    // 0 to 0x20000000
    .physio     = 0x20000000, // after RAM
    ⟨soc(raspberry pi1) other fields(arm) 496j⟩
};
```
Uses `MiB` 34g.

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`[??] (in `Active`[49d]) and `Ref`[??] (in `Chan`[??]) 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)* 56a⟩≡ (730a)
```
// bool   arch_splhi(void);
TEXT arch_splhi(SB), 1, $-4
    ⟨arch_splxxx() save caller information(arm) 56e⟩
    MOVW    CPSR, R0            /* turn off irqs (but not fiqs) */
    ORR $(PsrDirq), R0, R1
    MOVW    R1, CPSR
    RET
```

⟨*function* `arch_spllo` *(arm)* 56b⟩≡ (730a)
```
// 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)* 56c⟩≡ (730a)
```
// void    arch_splx(bool);
TEXT arch_splx(SB), 1, $-4
    ⟨arch_splxxx() save caller information(arm) 56e⟩
    MOVW    R0, R1              /* reset interrupt level */
    MOVW    CPSR, R0 //dead
    MOVW    R1, CPSR
    RET
```

⟨*function* `arch_islo` *(arm)* 56d⟩≡ (730a)
```
// 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)* 56e⟩≡ (56)
```
MOVW    $(CPUADDR+4), R2        /* save caller pc in Cpu */
MOVW    R14, 0(R2)
```

⟨`Cpu` *second field* 56f⟩≡ (48a)
```
// 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)* 57⟩≡ (730b)

```
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)* 58a⟩≡ (731b)
```
void
arch_xinc(long *p)
{
    int s;

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

⟨*function* `arch_xdec` *(raspberry pi1)(arm)* 58b⟩≡ (731b)
```
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` 58c⟩≡ (640)
```
// a.k.a Spinlock
struct Lock
{
    ulong key; // 0 when unset, 1 (or 0xDEADDEAD) when acquired, could be a bool

    ⟨Lock ilock fields 61b⟩
    ⟨Lock debugging fields 59a⟩
    ⟨Lock other fields 60a⟩
};
```

⟨*function* `lock` 58d⟩≡ (641b)
```
int
lock(Lock *l)
{
    int i;
```

```
    ⟨lock() other locals 60b⟩

    ⟨lock() start of lock, stat 628b⟩
    ⟨lock() increment nlocks 70g⟩
    if(arch_tas(&l->key) == 0){ // lock old value was 0, the lock was not held
        ⟨lock() got the lock, update other fields 60c⟩
        ⟨lock ifdef LOCKCYCLES 627c⟩
        return 0;
    }
    //else

    ⟨lock() decrement nlocks 70h⟩
    ⟨lock() when lock already held, stat 628c⟩

    // The lock was already held, need to spin
    for(;;){
    ⟨lock() in loop when lock already held, stat 628d⟩
        i = 0;
        while(l->key){
           ⟨lock() optional priority-inversion for real-time process 533f⟩
            if(i++ > 100000000){
                i = 0;
                lockloop(l, pc);
            }
        }
        // try again
        ⟨lock() increment nlocks 70g⟩
        if(arch_tas(&l->key) == 0){
            ⟨lock() got the lock, update other fields 60c⟩
            ⟨lock ifdef LOCKCYCLES 627c⟩
            return 1;
        }
        ⟨lock() decrement nlocks 70h⟩
    }
    return -1; // unreachable
}
```

Uses `lockloop()` 613a.

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* 59a⟩≡ (58c) 70a▷

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

⟨`Proc` *debugging fields* 59b⟩≡ (49f) 61c▷

```
Lock* lastlock;
```

⟨Lock *other fields* 60a⟩≡ (58c)
```
// option<ref<Cpu>>, None when key = 0?
Cpu  *cpu; // not that used, only in iprintcanlock apparently
```

⟨lock() *other locals* 60b⟩≡ (58d)
```
ulong pc = getcallerpc(&l);
```

⟨lock() *got the lock, update other fields* 60c⟩≡ (58) 62d ▷
```
if(up)
    up->lastlock = l;
l->pc = pc;
l->cpu = CPUS(cpu->cpuno); // pad's third bugfix
```
Uses CPUS 645 and up.

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

⟨*function* unlock 60d⟩≡ (641b)
```
void
unlock(Lock *l)
{
    ⟨unlock ifdef LOCKCYCLES 627e⟩

    ⟨unlock() sanity checks 60e⟩
    l->cpu = nil;
    l->key = 0;
    ⟨unlock() after modified key, call coherence 501a⟩

    ⟨unlock() if delaysched 208c⟩
}
```

⟨unlock() *sanity checks* 60e⟩≡ (60d) 62f ▷
```
if(l->key == 0)
    print("unlock: not locked: pc %#p\n", getcallerpc(&l));
```
Uses print.

⟨*function* canlock 60f⟩≡ (641b)
```
bool
canlock(Lock *l)
{
    ⟨lock() increment nlocks 70g⟩
    if(arch_tas(&l->key) != 0){ // lock old value != 0, lock was already held
       ⟨lock() decrement nlocks 70h⟩
        return false;
    }
    // else
    ⟨canlock() lock was not already held, update other fields 60g⟩
    ⟨lock ifdef LOCKCYCLES 627c⟩
    return true;
}
```
Uses false and true.

⟨canlock() *lock was not already held, update other fields* 60g⟩≡ (60f) 70c ▷
```
if(up)
    up->lastlock = l;
l->pc = getcallerpc(&l);
l->cpu = CPUS(cpu->cpuno);
```
Uses CPUS 645 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` 61a⟩≡ (640)
```
typedef Lock ILock;
```

⟨`Lock` *ilock fields* 61b⟩≡ (58c)
```
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* 61c⟩+≡ (49f) ◁59b 63c▷
```
Lock  *lastilock;
```

⟨*function* `ilock` 61d⟩≡ (641b)
```
// 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 62a⟩
    ⟨lock ifdef LOCKCYCLES 627c⟩
}
```

Uses `lockstats` 628e.

⟨ilock() *got the lock, update other fields* 62a⟩≡ (61d) 70e▷
```
if(up)
    up->lastilock = l;
l->isilock = true;
l->sr = x;
l->pc = pc;
l->cpu = CPUS(cpu->cpuno);
```
Uses CPUS 645, true, and up.

⟨*function* iunlock 62b⟩≡ (641b)
```
void
iunlock(Lock *l)
{
    ulong sr;

    ⟨iunlock ifdef LOCKCYCLES 627f⟩
    ⟨iunlock() sanity checks 62c⟩
    sr = l->sr;
    l->cpu = nil;
    l->key = 0;
    ⟨iunlock() after modified key, call coherence 501b⟩
    ⟨iunlock() decrement ilockdepth 71e⟩
    if(up)
        up->lastilock = nil;
    arch_splx(sr);
}
```
Uses up.

⟨iunlock() *sanity checks* 62c⟩≡ (62b)
```
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* 62d⟩+≡ (58) ◁60c 70b▷
```
l->isilock = false;
```
Uses false.

⟨canlock() *lock not held, update other fields* 62e⟩≡
```
l->isilock = false;
```

⟨unlock() *sanity checks* 62f⟩+≡ (60d) ◁60e 70d▷
```
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 63a⟩≡ (640)
```
// 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 63b⟩

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

⟨KQLock *debugging fields* 63b⟩≡ (63a)
```
kern_addr qpc;    /* pc of the holder */ // for debugging?
```

⟨Proc *debugging fields* 63c⟩+≡ (49f) ◁61c
```
kern_addr qpc;    /* pc calling last blocking qlock */
```

⟨Proc *extra fields* 63d⟩≡ (49f) 110e▷
```
// list<ref<Proc>> KQlock.head or RWLock.head (or Procalloc.free)
Proc  *qnext;   /* next process on queue for a QLock */
```

⟨Procstate *cases* 63e⟩≡ (51b) 65d▷
```
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 63f⟩≡ (642a)
```
void
qlock(QLock *q)
{
    Proc *p;

    ⟨qlock() sanity checks 64b⟩
    lock(&q->use);
    ⟨qlock() start of qlock, stat 628h⟩
    if(!q->locked) {
        q->locked = true;
        q->qpc = getcallerpc(&q);
        unlock(&q->use);
        return;
    }
    // else
    ⟨qlock() sanity check up 64a⟩
    ⟨qlock() when qlock already held, stat 628i⟩

    // 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 63e, lock() 58d, sched, true, unlock() 60d, and up.

⟨qlock() *sanity check up* 64a⟩≡ (63f)

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

Uses up.

⟨qlock() *sanity checks* 64b⟩≡ (63f) 71b ▷

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

⟨*function* qunlock 64c⟩≡ (642a)

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

        lock(&q->use);
        ⟨qunlock() sanity checks 64d⟩

        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() 58d, nilptr 693c, ready, and unlock() 60d.

⟨qunlock() *sanity checks* 64d⟩≡ (64c)

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

Uses print.

⟨*function* canqlock 65a⟩≡ (642a)
```
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() 60f, false, true, and unlock() 60d.

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[??]), process groups (Pgrp[??]), and environment groups (Egrp[??]). The implementation is subtle: it must avoid starving either readers or writers, and like QLock, waiters sleep rather than spin.

⟨*struct* RWlock 65b⟩≡ (640)
```
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 65c⟩
    // Extra
    Lock  use;
};
```

⟨RWlock *debugging fields* 65c⟩≡ (65b)
```
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* 65d⟩+≡ (51b) ◁63e 67a▷
```
QueueingR, // see rlock()
```

⟨*function* rlock 65e⟩≡ (642a)
```
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 66a⟩
    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 65d, lock() 58d, rwstats 628g, sched, unlock() 60d, and up.

⟨rlock() *sanity check up* 66a⟩≡ (65e)

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

Uses up.

⟨*function* runlock 66b⟩≡ (642a)

```
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 67a, lock() 58d, ready, true, and unlock() 60d.

Writers wait separately from readers:

⟨Procstate *cases* 67a⟩+≡ (51b) ◁65d 111c▷
```
QueueingW, // see wlock()
```

⟨*function* wlock 67b⟩≡ (642a)
```
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 67c⟩
    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 67a, lock() 58d, rwstats 628g, sched, true, unlock() 60d, and up.

⟨wlock() *sanity check up* 67c⟩≡ (67b)
```
if(up == nil)
    panic("wlock");
```

Uses up.

⟨*function* wunlock 67d⟩≡ (642a)
```
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 65d, QueueingW 67a, false, lock() 58d, ready, and unlock() 60d.

⟨*function* canrlock 68⟩≡ (642a)

```
/* 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() 58d, rwstats 628g, true, and unlock() 60d.

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` 69a⟩≡ (640)
```
// For reference counting shared things (e.g. a Page)
struct Ref
{
    long ref;

    Lock;
};
```

⟨*function* `incref` 69b⟩≡ (642b)
```
long
incref(Ref *r)
{
    long x;

    lock(r);
    x = ++r->ref;
    unlock(r);
    return x;
}
```
Uses `lock()` 58d and `unlock()` 60d.

⟨*function* `decref` 69c⟩≡ (642b)
```
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()` 58d and `unlock()` 60d.

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

⟨*struct* `Counter` 69d⟩≡ (640)
```
typedef struct Ref Counter;
```
Uses `Ref`.

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* 70a⟩+≡ (58c) ◁59a 627d▷
```
// 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* 70b⟩+≡ (58) ◁62d
```
l->p = up;
```
Uses up.

⟨canlock() *lock was not already held, update other fields* 70c⟩+≡ (60f) ◁60g
```
l->p = up;
```
Uses up.

⟨unlock() *sanity checks* 70d⟩+≡ (60d) ◁62f
```
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* 70e⟩+≡ (61d) ◁62a 71d▷
```
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* 70f⟩≡ (49f) 212b▷
```
// 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* 70g⟩≡ (60f 58)
```
if(up)
    inccnt(&up->nlocks);    /* prevent being scheded */
```
Uses inccnt() 70i and up.

⟨lock() *decrement nlocks* 70h⟩≡ (60f 58)
```
if(up)
    deccnt(&up->nlocks);
```
Uses deccnt() 71a and up.

⟨*function* inccnt 70i⟩≡ (641b)
```
// 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` 71a⟩≡ (641b)

```
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* 71b⟩+≡ (63f) ◁64b 71f▷

```
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* 71c⟩≡ (48a) 203e▷

```
int ilockdepth;
```

⟨`ilock()` *got the lock, update other fields* 71d⟩+≡ (61d) ◁70e

```
cpu->ilockdepth++;
```

⟨`iunlock()` *decrement ilockdepth* 71e⟩≡ (62b)

```
cpu->ilockdepth--;
```

⟨`qlock()` *sanity checks* 71f⟩+≡ (63f) ◁71b

```
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* 74⟩≡ (41f)

```
[BRK]       sysbrk,
```

Uses `sysbrk()` 104c.

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()`[77c] carves large structures from those banks. From here the hierarchy forks.

2a. On the kernel side, `malloc()`[85a] provides fine-grained allocation from a `Pool` backed by `xalloc`.

2b. On the user side, `palloc`[90d] 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`[??] 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 `Holes`.

⟨*struct* `Xalloc` 75a⟩≡ (694)

```
// 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* 75b⟩≡ (75d)

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

⟨*constants holes* 75c⟩≡ (694)

```
enum
{
    Nhole   = 128,
    ⟨constant Magichole 77b⟩
};
```

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` 75d⟩≡ (694)

```
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 75b⟩
};
```

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` 76⟩≡ (697b)

```
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` 77a⟩≡ (694)
```
// 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` 77b⟩≡ (75c)
```
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` 77c⟩≡ (697b)
```
kern_vp
xalloc(ulong size)
{
    return xallocz(size, true);
}
```
Uses `true` and `xallocz()` 77d.

⟨*function* `xallocz` 77d⟩≡ (697b)
```
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 75d, ilock() 61d, iunlock() 62b, offsetof 692d, and xlists-1849 76.

5.2.3 xfree()

xfree()[78a] 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 78a⟩≡ (697b)

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

    x = (Xhdr*)((ulong)p - offsetof(Xhdr, data[0]));
    ⟨xfree() sanity check magix mark 78b⟩
    xhole(PADDR((kern_addr)x), x->size);
}
```

Uses offsetof 692d and xhole() 78c.

⟨xfree() *sanity check magix mark* 78b⟩≡ (78a)

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

Uses Magichole 75d and xsummary() 610a.

hole() is the inverse of xalloc()[77c]: 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()[260] to seed the allocator with the initial Confmem banks.

⟨*function* xhole 78c⟩≡ (697b)

```
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()` 61d, `iunlock()` 62b, `print`, and `xlists-1849` 76.

5.2.4 `xmerge()`

`xmerge()`[79] 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` 79⟩≡ (697b)

```
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 80a⟩
    if((byte*)p+p->size == (byte*)q) {
        p->size += q->size;
        return true;
    }
    return false;
}
```

Uses false, offsetof 692d, and true.

⟨xmerge() *sanity check magix* 80a⟩≡ (79)

```
if(p->magix != Magichole || q->magix != Magichole) {
    ⟨xmerge() debug info when not magichole ??⟩
    panic("xmerge(%#p, %#p) bad magic %#lux, %#lux",
        vp, vq, p->magix, q->magix);
}
```

Uses Magichole 75d.

⟨*function* xspanalloc 80b⟩≡ (697b)

```
kern_vp
xspanalloc(ulong size, int align, ulong span)
{
    ulong a, v, t;
    a = (kern_addr)xalloc(size+align+span);
    ⟨xspanalloc() sanity check a 80c⟩

    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() 77c and xhole() 78c.

⟨xspanalloc() *sanity check a* 80c⟩≡ (80b)

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

Uses nilptr 693c.

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* 81⟩≡ (698a)

```
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 82a⟩≡ (698a)
```
static void
plock(Pool *p)
{
    Private *pv;

    pv = p->private;
    ilock(&pv->lk);
    pv->lk.pc = getcallerpc(&p);
    pv->msg[0] = 0;
}
```
Uses ilock() 61d.

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 82b⟩≡ (698a)
```
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() 62b.

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` 83a⟩≡ (698a)

```
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` 704a, `MB` 704a, `plock()` 82a, `pmainpriv-1838` 83a, `poolprint()` 84b, `ppanic()` 84c, `punlock()` 82b, `xalloc()` 77c, and `xmerge()` 79.

⟨*global* `pimagmem` 83b⟩≡ (698a)

```
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 704a, pimagpriv-1840 83b, plock() 82a, poolprint() 84b, ppanic() 84c, punlock() 82b, xalloc() 77c, and xmerge() 79.

⟨*global mainmem and imagmem* 84a⟩≡ (698a)

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

Uses imagmem 84a, mainmem 84a, pimagmem-1841 83b, and pmainmem-1839 83a.

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()`[254d] 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()`[84b] and `ppanic()`[84c] 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 84b⟩≡ (698a)

```
/*
 * 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 84c⟩≡ (698a)

```
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() 62b.

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 85a⟩≡ (698b)
```
kern_vp
malloc(ulong size)
{
    kern_vp v;

    v = poolalloc(mainmem, size + Npadlong*sizeof(ulong));
    if(v == nil)
        return nil;
    ⟨malloc() if Npadlong 87a⟩
    memset(v, 0, size);
    return v;
}
```
Uses Npadlong-1813 698b and mainmem 84a.

⟨*function* free 85b⟩≡ (698b)
```
void
free(kern_vp v)
{
    if(v != nil)
        poolfree(mainmem, (ulong*)v-Npadlong);
}
```
Uses Npadlong-1813 698b and mainmem 84a.

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 85c⟩≡ (698b)
```
// 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 87d⟩
        memset(v, 0, size); // clear
        return v;
    }
```

Uses Npadlong-1813 698b, mainmem 84a, returnfalse() 86a, tsleep, and up.

⟨*function* returnfalse 86a⟩≡ (647a)

```
    // usually used as default callback for sleep/tsleep
    bool
    returnfalse(void*)
    {
        return false;
    }
```

Uses false.

5.4.3 mallocxxx()

⟨*function* mallocz 86b⟩≡ (698b)

```
    kern_vp
    mallocz(ulong size, bool clr)
    {
        kern_vp v;

        v = poolalloc(mainmem, size+Npadlong*sizeof(ulong));
        ⟨mallocz() if Npadlong 88a⟩
        if(clr && v != nil)
            memset(v, 0, size);
        return v;
    }
```

Uses Npadlong-1813 698b and mainmem 84a.

⟨*function* mallocalign 86c⟩≡ (698b)

```
    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 88b⟩
        if(v)
            memset(v, 0, size);
        return v;
    }
```

Uses Npadlong-1813 698b and mainmem 84a.

⟨*function* realloc 86d⟩≡ (698b)

```
    kern_vp
    realloc(kern_vp v, ulong size)
    {
        kern_vp nv;

        if(v != nil)
            v = (ulong*)v - Npadlong;
        ⟨realloc() if Npadlong 88c⟩
```

```
    if(nv = poolrealloc(mainmem, v, size)){
        ⟨realloc() readjust nv if Npadlong 88d⟩
    }
    return nv;
}
```

Uses Npadlong-1813 698b and mainmem 84a.

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* 87a⟩≡ (85a)

```
if(Npadlong){
    v = (ulong*)v+Npadlong;
    setmalloctag(v, getcallerpc(&size));
    setrealloctag(v, 0);
}
```

Uses Npadlong-1813 698b, setmalloctag() 87b, and setrealloctag() 87c.

⟨*function* setmalloctag 87b⟩≡ (698b)

```
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 698b and Npadlong-1813 698b.

⟨*function* setrealloctag 87c⟩≡ (698b)

```
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 698b and ReallocOffset-1815 698b.

⟨smalloc() *if Npadlong* 87d⟩≡ (85c)

```
if(Npadlong){
    v = (ulong*)v+Npadlong;
    setmalloctag(v, getcallerpc(&size));
}
```

Uses Npadlong-1813 698b and setmalloctag() 87b.

⟨mallocz() *if Npadlong* 88a⟩≡ (86b)

```
if(Npadlong && v != nil){
    v = (ulong*)v+Npadlong;
    setmalloctag(v, getcallerpc(&size));
    setrealloctag(v, 0);
}
```

Uses Npadlong-1813 698b, setmalloctag() 87b, and setrealloctag() 87c.

⟨mallocalign() *if Npadlong* 88b⟩≡ (86c)

```
if(Npadlong && v != nil){
    v = (ulong*)v+Npadlong;
    setmalloctag(v, getcallerpc(&size));
    setrealloctag(v, 0);
}
```

Uses Npadlong-1813 698b, setmalloctag() 87b, and setrealloctag() 87c.

⟨realloc() *if Npadlong* 88c⟩≡ (86d)

```
if(Npadlong !=0 && size != 0)
    size += Npadlong*sizeof(ulong);
```

Uses Npadlong-1813 698b.

⟨realloc() *readjust nv if Npadlong* 88d⟩≡ (86d)

```
nv = (ulong*)nv+Npadlong;
setrealloctag(nv, getcallerpc(&v));
if(v == nil)
    setmalloctag(nv, getcallerpc(&v));
```

Uses Npadlong-1813 698b, setmalloctag() 87b, and setrealloctag() 87c.

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` 89⟩≡ (694)

```
// 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 171a⟩

    // extra
    Lock;
    ⟨Page extra fields 90c⟩
};
```

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()`[166b]); `PG_MOD` is set when a

page is written.[1]

⟨*enum* `modref` 90a⟩≡ (694)
```
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` 90b⟩≡ (694)
```
// Page Allocator (singleton)
struct Palloc
{
    // array<Page>, xalloc'ed in pageinit(), huge, cover physical user space
    Page  *pages; /* array of all pages */

    // 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 91a⟩
    ⟨Palloc other fields 171c⟩

    // extra
    Lock; // LOCK ORDERING: always do lock(&palloc); lock(&page)!!
    ⟨Palloc extra fields 191a⟩
};
```
Uses `PGHLOG`.

⟨`Page` *extra fields* 90c⟩≡ (89) 171d▷
```
// 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` 90d⟩≡ (697a)
```
// Page allocator
struct Palloc palloc;
```
Uses `Palloc`.

[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.

`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* 91a⟩≡ (90b)
```
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` 91b⟩≡ (694)
```
// 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` 91c⟩≡ (701)
```
Page*
newpage(bool clear, Segment **s, user_addr va)
{
    Page *p;
    ⟨newpage() other locals 92a⟩

    lock(&palloc);

    ⟨newpage() loop waiting freecount ¿ highwater 190b⟩

    //when no color: p = palloc.head;
    ⟨newpage() find free page p, set color and ct 521i⟩

    pageunchain(p);

    lock(p);
    if(p->ref != 0)
        panic("newpage: p->ref %d != 0", p->ref);

    ⟨newpage() uncachepage 172c⟩
    p->ref = 1;
    p->va = va;
    p->modref = PG_NOTHING;
    ⟨newpage() set cachectl using ct 522b⟩

    unlock(p);
    unlock(&palloc);
    ⟨newpage() if clear 92b⟩
```

```
    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* 92a⟩≡ (91c) 190a▷
```
Arch_KMap *k;
```

⟨`newpage()` *if clear* 92b⟩≡ (91c)
```
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` 92c⟩≡ (701)
```
// 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()` 60f and `palloc` 90d.

⟨*destructor* `putpage` 92d⟩≡ (701)
```
void
putpage(PageOrSwap *p)
{
    ⟨putpage if p is a swap address 187c⟩

    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 172b⟩
    else
        pagechainhead(p);

    ⟨putpage wakeup process waiting for freepage if any 191b⟩
    unlock(p);
    unlock(&palloc);
}
```

Uses `lock()` 58d, `pagechainhead()` 93a, `palloc` 90d, and `unlock()` 60d.

⟨*function* `pagechainhead` 93a⟩≡ (701)

```
// 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()` 60f and `palloc` 90d.

⟨*function* `pagechaintail` 93b⟩≡ (701)

```
// 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()` 60f and `palloc` 90d.

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` 94a⟩≡ (694)
```
/* Segment types */
enum Segtype
{
    SG_TEXT   = 00,
    SG_DATA   = 01,
    SG_BSS    = 02,
    SG_STACK  = 03,
    ⟨Segtype other cases 523a⟩
    SG_TYPE   = 07,   /* Mask type of segment */

    ⟨Segtype other flags 101b⟩
};
```

⟨*global* `sname` 94b⟩≡ (680)
```
// 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`[??] pointers:

⟨*struct* `Segment` 94c⟩≡ (694)
```
// 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 99a⟩

    // 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)* 95a⟩≡ (747)

```
#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 95b⟩≡ (694)

```
// 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)* 96a⟩≡ (747)
```
#define PAGETABMAPMEM   (1024*1024) // 1MB
```

⟨*constant* PAGETABSIZE*(arm)* 96b⟩≡ (747)
```
#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 96c⟩≡ (701)
```
Pagetable*
ptalloc(void)
{
    Pagetable *new;

    new = smalloc(sizeof(Pagetable));
    new->first = &new->pagetab[PAGETABSIZE];
    new->last = new->pagetab;
    return new;
}
```
Uses smalloc() 85c.

Note the use of smalloc()—a sleeping variant of malloc() that blocks if memory is not available instead of returning nil.

⟨*destructor* freept 96d⟩≡ (701)
```
void
freept(Segment *s, Pagetable *p)
{
    PageOrSwap **pte;
    ⟨freept() other locals 524b⟩

    switch(s->type&SG_TYPE) {
    ⟨freept() SG_PHYSICAL case 524c⟩
    default:
        for(pte = p->first; pte <= p->last; pte++)
            if(*pte) {
                putpage(*pte);
                *pte = nil;
            }
    }
    free(p);
}
```
Uses SG_TYPE, free() 85b, and putpage() 92d.

⟨*function* ptcpy 97a⟩≡ (701)

```
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 187f⟩
            else {
                lock(*src);
                (*src)->ref++;
                unlock(*src);
            }
            new->last = dst;
            *dst = *src;
        }
    return new;
}
```

Uses lock() 58d, ptalloc() 96c, and unlock() 60d.

5.7.3 newseg()

We have seen newpage()[91c] for allocating physical pages, and the Segment and Pagetable[??] data structures. Now we can see how a new segment is created:

newseg()[97b] 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()[104c] (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 97b⟩≡ (702)

```
Segment *
newseg(int type, user_addr base, ulong size)
{
    Segment *s;
    int pagedirsize;

    ⟨newseg() sanity check size 98a⟩
    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 442c⟩

    pagedirsize = ROUND(size, PAGETABSIZE)/PAGETABSIZE;

    ⟨newseg() if pagedirsize small 99c⟩
    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 34a and smalloc() 85c.

Again smalloc() is used, so if the kernel is low on memory the caller will sleep until some is freed.

⟨newseg() *sanity check size* 98a⟩≡ (97b)

```
if(size > (PAGEDIRSIZE*PAGETABSIZE))
    error(Enovmem);
```

Uses Enovmem 631d.

⟨*destructor* putseg 98b⟩≡ (702)

```
void
putseg(Segment *s)
{
    Pagetable **pde, **emap;
    ⟨putseg() other locals 182a⟩

    ⟨putseg() sanity check s 98c⟩
    ⟨putseg() if s has an image 182b⟩
    else
        lock(s);

    s->ref--;
    if(s->ref != 0) {
        unlock(s);
        return;
    }
    // else
    unlock(s);

    qlock(&s->lk);
    ⟨putseg() if s had an image 182c⟩

    emap = &s->pagedir[s->pagedirsize];
    for(pde = s->pagedir; pde < emap; pde++)
        if(*pde)
            freept(s, *pde);

    qunlock(&s->lk);

    ⟨putseg() if smallpagedir 99d⟩
    else
        free(s->pagedir);
    ⟨putseg() free profile 473e⟩
    free(s);
}
```

Uses free() 85b, freept() 96d, lock() 58d, qlock() 63f, qunlock() 64c, and unlock() 60d.

⟨putseg() *sanity check s* 98c⟩≡ (98b)

```
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* 99a⟩≡ (94c) 174f▷
```
// array<option<ref_own<Pagetable>>
Pagetable *smallpagedir[SMALLPAGEDIRSIZE];
```

⟨*constant* SMALLPAGEDIRSIZE*(arm)* 99b⟩≡ (747)
```
#define SMALLPAGEDIRSIZE    16
```

⟨newseg() *if pagedirsize small* 99c⟩≡ (97b)
```
if(pagedirsize <= nelem(s->smallpagedir)){
    s->pagedir = s->smallpagedir;
    s->pagedirsize = nelem(s->smallpagedir);
}
```
Uses nelem 692c.

⟨putseg() *if smallpagedir* 99d⟩≡ (98b)
```
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()[166b]), and segment growth (ibrk()[104d] for the bss).

⟨*function* segpage 99e⟩≡ (702)
```
void
segpage(Segment *s, Page *p)
{
    Pagetable **pt;
    ulong off;
    Page **pg;

    ⟨segpage() sanity check page in range of segment 99f⟩

    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() 96c.

⟨segpage() *sanity check page in range of segment* 99f⟩≡ (99e)
```
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` 100a⟩≡ (705c)

```
/*
 *  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* 100b⟩≡ (49f) 163f▷

```
// 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` 100c⟩≡ (700)

```
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 100a, qlock() 63f, and qunlock() 64c.

Addresses covering multiple segments, validaddr()

⟨*function* okaddr 101a⟩≡ (700)

```
/*
 * 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 694, false, pprint, seg() 100c, true, and up.

⟨Segtype *other flags* 101b⟩≡ (94a) 435b▷

```
SG_RONLY  = 0040,   /* Segment is read only */
```

⟨*function* validaddr 101c⟩≡ (700)

```
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 631d, NDebug 418c, okaddr() 101a, 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 101d⟩≡ (700)

```
/*
 * &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 102a, false, and validaddr() 101c.

⟨*portdat.h macros* 102a⟩+≡ (? 0—1) ◁34e

```
// 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 102b⟩≡ (702)

```
Segment*
dupseg(Segment **seg, int segno, bool share)
{
    Segment *s;
    Segment *n; // when allocate a new segment
    ⟨dupseg() other locals 103d⟩

    SET(n);

    s = seg[segno];

    qlock(&s->lk);
    if(waserror()){
        qunlock(&s->lk);
        nexterror();
    }
    switch(s->type&SG_TYPE) {
    ⟨dupseg() switch segment type cases 103a⟩
    }
    // not sameseg, we have allocated a new seg in n above
    ⟨dupseg() when not goto sameseg, when allocated a new segment 103e⟩

    poperror();
    qunlock(&s->lk);
    return n;
```

```
  sameseg:
      ⟨dupseg() in sameseg, when share a segment 103b⟩

      poperror();
      qunlock(&s->lk);
      return s;
  }
```

Uses SG_TYPE, nexterror, poperror 633d, qlock() 63f, qunlock() 64c, and waserror 633d.

⟨dupseg() *switch segment type cases* 103a⟩≡ (102b) 103c▷

```
  case SG_TEXT:       /* New segment shares pt set */
      goto sameseg;
```

Uses SG_TEXT.

⟨dupseg() *in sameseg, when share a segment* 103b⟩≡ (102b)

```
  incref(s);
```

Uses incref() 69b.

⟨dupseg() *switch segment type cases* 103c⟩+≡ (102b) ◁103a 103f▷

```
  case SG_STACK:
      n = newseg(s->type, s->base, s->size);
      break;
```

Uses SG_STACK 94a and newseg() 97b.

⟨dupseg() *other locals* 103d⟩≡ (102b)

```
  int i;
  int size; // pagedir
  Pagetable *pt;
```

⟨dupseg() *when not goto sameseg, when allocated a new segment* 103e⟩≡ (102b)

```
  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 522j⟩
  ⟨dupseg() if original segment was shared, flush it 440a⟩
```

Uses ptcpy() 97a.

⟨dupseg() *switch segment type cases* 103f⟩+≡ (102b) ◁103c 103g▷

```
  case SG_DATA:       /* Copy on write plus demand load info */
      ⟨dupseg() when SG_DATA, if Text segment 177b⟩
      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 177a⟩
      break;
```

Uses SG_DATA 523b and newseg() 97b.

⟨dupseg() *switch segment type cases* 103g⟩+≡ (102b) ◁103f 523c▷

```
  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() 97b.

5.7.7 /proc/x/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* 104a⟩≡ (571d) 325b▷
```
Segment *sg;
```

⟨procread() *cases* 104b⟩≡ (571d) 325e▷
```
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 100a, Qsegment-873 568a, SG_RONLY 694, SG_TYPE, exhausted() 633c, and sname-909 94b.

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* 104c⟩≡ (716a)
```
// int brk(void*);
long
sysbrk(ulong* arg)
{
    return ibrk(arg[0], BSEG); // BSS, the heap size is changed
}
```

Uses BSEG 100a and ibrk() 104d.

⟨*function* ibrk 104d⟩≡ (702)
```
long
ibrk(user_addr addr, int seg)
{
    Segment *s;
    user_addr newtop;
    ulong newsize;
    ⟨ibrk() other locals 105d⟩
```

```
    s = up->seg[seg];
    ⟨ibrk() sanity check s 105a⟩
    ⟨ibrk() if addr nil 105b⟩

    qlock(&s->lk);

    ⟨ibrk() if addr below base 105c⟩

    newtop = PGROUND(addr);
    newsize = (newtop - s->base)/BY2PG; // in nb pages

    ⟨ibrk() if newtop lower than oldtop, shrink the segment 106d⟩
    // else
    ⟨ibrk() if newtop more than oldtop, extend the segment 105e⟩

}
```

Uses PGROUND 102a, qlock() 63f, and up.

⟨ibrk() *sanity check s* 105a⟩≡ (104d)
```
if(s == nil)
    error(Ebadarg);
```

Uses Ebadarg 631d.

⟨ibrk() *if addr nil* 105b⟩≡ (104d)
```
if(addr == nilptr)
    return s->base;
```

Uses nilptr 693c.

⟨ibrk() *if addr below base* 105c⟩≡ (104d)
```
/* 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 100a, DSEG 100a, Enovmem 631d, qunlock() 64c, 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* 105d⟩≡ (104d) 106a▷
```
int mapsize;
Pagetable **map;
```

⟨ibrk() *if newtop more than oldtop, extend the segment* 105e⟩≡ (104d)
```
⟨ibrk() sanity check newtop does not overlap other segments 106b⟩
⟨ibrk() sanity check newsize 106c⟩

// 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 34a, free() 85b, qunlock() 64c, and smalloc() 85c.

⟨ibrk() *other locals* 106a⟩+≡ (104d) ◁105d

```
    Segment *ns;
    int i;
```

⟨ibrk() *sanity check newtop does not overlap other segments* 106b⟩≡ (105e)

```
    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 631d, NSEG 100a, qunlock() 64c, and up.

⟨ibrk() *sanity check newsize* 106c⟩≡ (105e)

```
    if(newsize > (PAGEDIRSIZE*PAGETABSIZE)) {
        qunlock(&s->lk);
        error(Enovmem);
    }
```

Uses Enovmem 631d and qunlock() 64c.

5.8.2 Shrinking a segment

⟨ibrk() *if newtop lower than oldtop, shrink the segment* 106d⟩≡ (104d)

```
    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() 107, and qunlock() 64c.

⟨*function* mfreeseg 107⟩≡ (702)

```
/*
 *  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 187d⟩
                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 440b⟩
    /* free the pages */
    for(pg = list; pg != nil; pg = list){
        list = list->next;
        putpage(pg);
    }
}
```

Uses `putpage()` 92d.

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* 109a⟩≡ (41f)
```
[RFORK]     sysrfork,
[EXEC]      sysexec,
[AWAIT]     sysawait,
[EXITS]     sysexits,
```
Uses `sysawait()` 132a, `sysexits()` 133e, and `sysrfork()` 115c.

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* 109b⟩+≡ (49f) ◁51a 112c▷
```
ulong pid;
```

⟨*global* `pidalloc` 110a⟩≡ (712c)
```
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` 110b⟩≡ (705c)
```
// 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 110d⟩

    // extra
    Lock;
};
```

⟨*global* `procalloc` 110c⟩≡ (712c)
```
static struct Procalloc procalloc;
```
Uses `Procalloc` 110b.

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* 110d⟩≡ (110b)
```
// hash<Proc.pid, ref<Proc>> (next = Proc.pidhash)>
Proc* ht[128];
```

⟨`Proc` *extra fields* 110e⟩+≡ (49f) ◁63d 200d▷
```
// hash<Proc.pid, ref<Proc>> Procalloc.ht
Proc  *pidhash; /* next proc in pid hash */
```

⟨*function* `pidhash` 110f⟩≡ (712c)
```
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()` 58d, `nelem` 692c, `procalloc-2458` 110c, and `unlock()` 60d.

⟨*function* `pidunhash` 111a⟩≡ (712c)

```
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()` 58d, `nelem` 692c, `procalloc-2458` 110c, and `unlock()` 60d.

`procindex()` translates a pid into an index in the `procalloc.arena` array. It is used by `devproc` to implement `/proc/\emphpid/...`:

⟨*function* `procindex` 111b⟩≡ (712c)

```
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()` 58d, `nelem` 692c, `procalloc-2458` 110c, and `unlock()` 60d.

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()`[115c] or `kproc()`[139c]) must finish the initialization before calling `ready()`[??].

⟨`Procstate` *cases* 111c⟩+≡ (51b) ◁67a 135a▷

```
Scheding,
```

⟨*constructor* `newproc` 111d⟩≡ (712c)

```
Proc*
newproc(void)
{
    Proc *p;
```

```
    ⟨newproc() other locals 112f⟩

    lock(&procalloc);
    // =~ p = procalloc.free
    ⟨newproc() set p to procalloc.free and wait if nil 113a⟩
    procalloc.free = p->qnext;
    unlock(&procalloc);
    p->qnext = nil;

    p->state = Scheding;
    p->pid = incref(&pidalloc);
    ⟨newproc() sanity check pid 112a⟩
    pidhash(p);

    ⟨newproc() set fields of p 112b⟩

    return p;
}
```

Uses Scheding 111c, incref() 69b, lock() 58d, pidalloc-2459 110a, pidhash() 110f, procalloc-2458 110c, and unlock() 60d.

```
⟨newproc() sanity check pid 112a⟩≡                                (111d)
  if(p->pid==0)
      panic("pidalloc");
```

```
⟨newproc() set fields of p 112b⟩≡                          (111d) 112d▷
  memset(p->seg, nilptr, sizeof p->seg);
```

Uses nilptr 693c.

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 112c⟩+≡                          (49f) ◁109b 114b▷
  // 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 112d⟩+≡                    (111d) ◁112b 112e▷
  p->psstate = "New";
  kstrdup(&p->text, "*notext");
```

Uses kstrdup() 636b.

```
⟨newproc() set fields of p 112e⟩+≡                    (111d) ◁112d 114c▷
  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()[111d] is called, the kernel sleeps and retries, hoping another process will exit soon. If the situation persists, it panics and reboots.

```
⟨newproc() other locals 112f⟩≡                                    (111d)
  char msg[64];
```

⟨newproc() *set* p *to* procalloc.free *and wait if nil* 113a⟩≡ (111d)

```
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() 58d, noprocpanic() 113b, procalloc-2458 110c, resrcwait() 113c, unlock() 60d, 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 113b⟩≡ (712c)

```
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 49e, arch_delay, lock() 58d, procdump() 612a, true, and unlock() 60d.

⟨*function* resrcwait 113c⟩≡ (710)

```
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 114a, print, returnfalse() 86a, seconds() 241a, tsleep, and up.

⟨*constant* Whinesecs 114a⟩≡ (710)

```
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* 114b⟩+≡ (49f) ◁112c 118a▷

```
// e.g.. "eve" (no uid/gid in plan9, because of its distributed nature?)
char  *user;
```

⟨newproc() *set fields of* p 114c⟩+≡ (111d) ◁112e 126d▷

```
kstrdup(&p->user, "*nouser");
```

Uses kstrdup() 636b.

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 114d⟩≡ (718)

```
/*
 *  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 631d, PriNormal 198b, eve 557a, iseve() 557b, kstrdup() 636b, renameuser() 115a, and up.

⟨*function* `renameuser` 115a⟩≡ (712c)
```
/*
 *  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` 46b, `kstrdup()` 636b, and `procalloc-2458` 110c.

6.4 `sysrfork()`

`sysrfork()`[115c] 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` 115b⟩≡ (716a)
```
//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* 115c⟩≡ (716a)
```
// int rfork(int flags);
long
sysrfork(ulong* arg)
{
    ulong flag;
    ⟨sysrfork() other locals 116a⟩

    flag = arg[0];
    /* Check flags before we commit */
    ⟨sysrfork() sanity check flags, clean or copy 119a⟩
```

```
    ⟨sysrfork() if no RFPROC, no new proc but property settings 121b⟩
    // else
    ⟨sysrfork() when RFPROC, fork a new process 116b⟩
}
```

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()[117e]), and makes it runnable. The parent returns the child's pid.

⟨sysrfork() *other locals* 116a⟩≡ (115c) 117c▷

```
Proc *p;
ulong pid;
```

⟨sysrfork() *when* RFPROC, *fork a new process* 116b⟩≡ (115c)

```
p = newproc();
pid = p->pid;

⟨sysrfork() when RFPROC propagate fields from up to p 117a⟩

/* Make a new set of memory segments */
⟨sysrfork() copy or share memory segments 117d⟩

/* File descriptors */
⟨sysrfork() copy, clean, or share file descriptors 119b⟩
/* Process groups */ // Namespace
⟨sysrfork() copy, clean, or share namespace 119d⟩
// Rendez vous group
⟨sysrfork() new or share rendezvous group 120c⟩
/* Environment group */
⟨sysrfork() copy, clean, or share environment 120b⟩
// Note group
⟨sysrfork() share or not note group 120d⟩

/* 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 138d⟩

⟨sysrfork() setting time field 471c⟩

/*
 *  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 217b⟩

ready(p);
sched(); // !!!
return pid;
```

Uses newproc() 111d, 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* 117a⟩≡ (116b) 117b▷
```
kstrdup(&p->text, up->text);
kstrdup(&p->user, up->user);
```
Uses `kstrdup()` 636b and `up`.

⟨`sysrfork()` *when* `RFPROC` *propagate fields from up to p* 117b⟩+≡ (116b) ◁117a 145d▷
```
p->slash = up->slash;
p->dot = up->dot;
incref(p->dot);
```
Uses `incref()` 69b 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()`[102b] makes copy-on-write copies. The text segment is always shared (it is read-only).

⟨`sysrfork()` *other locals* 117c⟩+≡ (115c) ◁116a 121a▷
```
bool share;
int i;
```

⟨`sysrfork()` *copy or share memory segments* 117d⟩≡ (116b)
```
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` 100a, `RFMEM-2550` 115b, `dupseg()` 102b, `nexterror`, `poperror` 633d, `qlock()` 63f, `qunlock()` 64c, `up`, and `waserror` 633d.

6.4.3 `arch_forkchild()` (ARM)

`arch_forkchild()`[117e] 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)* 117e⟩≡ (757b)
```
/*
 *  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) 482a⟩
}
```

Uses `false`.

⟨`Proc` *state fields* 118a⟩+≡ (49f) ◁114b 126c▷

```
bool insyscall; // true when process inside a syscall
```

⟨*function* `arch__forkret` *(arm)* 118b⟩≡ (756)

```
/*
 *  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* 119a⟩≡ (115c) 119c▷

```
if((flag & (RFFDG|RFCFDG)) == (RFFDG|RFCFDG))
    error(Ebadarg);
```

Uses Ebadarg 631d, RFCFDG-2557 115b, and RFFDG-2554 115b.

⟨sysrfork() *copy, clean, or share file descriptors* 119b⟩≡ (116b)

```
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 115b, RFFDG-2554 115b, dupfgrp() 276c, incref() 69b, 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* 119c⟩+≡ (115c) ◁119a 120a▷

```
if((flag & (RFNAMEG|RFCNAMEG)) == (RFNAMEG|RFCNAMEG))
    error(Ebadarg);
```

Uses Ebadarg 631d, RFCNAMEG-2555 115b, and RFNAMEG-2552 115b.

⟨sysrfork() *copy, clean, or share namespace* 119d⟩≡ (116b)

```
if(flag & (RFNAMEG|RFCNAMEG)) {
    p->pgrp = newpgrp();
    if(flag & RFNAMEG)
        pgrpcpy(p->pgrp, up->pgrp);
    ⟨sysrfork() inherit noattach, RFPROC==1 case 337a⟩
}
else {
    p->pgrp = up->pgrp;
```

```
        incref(p->pgrp);
    }
    ⟨sysrfork() set noattach to true when RFNOMNT, RFPROC==1 case 337c⟩
```

Uses RFCNAMEG-2555 115b, RFNAMEG-2552 115b, incref() 69b, newpgrp() 320f, pgrpcpy() 341c, and up.

6.4.6 Environment variables: RFENVG, RFCENVG

⟨sysrfork() *sanity check flags, clean or copy* 120a⟩+≡ (115c) ◁119c

```
    if((flag & (RFENVG|RFCENVG)) == (RFENVG|RFCENVG))
        error(Ebadarg);
```

Uses Ebadarg 631d, RFCENVG-2556 115b, and RFENVG-2553 115b.

⟨sysrfork() *copy, clean, or share environment* 120b⟩≡ (116b)

```
    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 115b, RFENVG-2553 115b, envcpy() 564a, incref() 69b, smalloc() 85c, 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* 120c⟩≡ (116b)

```
    if(flag & RFREND)
        p->rgrp = newrgrp();
    else {
        incref(up->rgrp);
        p->rgrp = up->rgrp;
    }
```

Uses RFREND-2559 115b, incref() 69b, newrgrp() 448c, 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* 120d⟩≡ (116b)

```
    if((flag&RFNOTEG) == 0)
        p->noteid = up->noteid; // rectify what was done in newproc
```

Uses RFNOTEG-2558 115b 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* 121a⟩+≡ (115c) ◁117c 217a▷
```
Fgrp *ofg;
Pgrp *opg;
Egrp *oeg;
Rgrp *org;
```

⟨sysrfork() *if no* RFPROC*, no new proc but property settings* 121b⟩≡ (115c)
```
if((flag&RFPROC) == 0) { // not a fork, just setting properties for up
    ⟨sysrfork() when no RFPROC, sanity check flag 121c⟩

    ⟨sysrfork() when no RFPROC, clean or copy file descriptors 121d⟩

    ⟨sysrfork() when no RFPROC, clean or copy namespace 121e⟩

    ⟨sysrfork() when no RFPROC, clean or copy environment 122a⟩

    ⟨sysrfork() when no RFPROC, new rendez vous group 122b⟩

    ⟨sysrfork() when no RFPROC, new note group 122c⟩

    return 0;
}
```
Uses RFPROC-2549 115b.

⟨sysrfork() *when no* RFPROC*, sanity check flag* 121c⟩≡ (121b)
```
if(flag & (RFMEM|RFNOWAIT))
    error(Ebadarg);
```
Uses Ebadarg 631d, RFMEM-2550 115b, and RFNOWAIT-2551 115b.

⟨sysrfork() *when no* RFPROC*, clean or copy file descriptors* 121d⟩≡ (121b)
```
if(flag & (RFFDG|RFCFDG)) {
    ofg = up->fgrp;
    if(flag & RFFDG)
        up->fgrp = dupfgrp(ofg);
    else
        up->fgrp = dupfgrp(nil);
    closefgrp(ofg);
}
```
Uses RFCFDG-2557 115b, RFFDG-2554 115b, closefgrp() 277b, dupfgrp() 276c, and up.

⟨sysrfork() *when no* RFPROC*, clean or copy namespace* 121e⟩≡ (121b)
```
if(flag & (RFNAMEG|RFCNAMEG)) {
    opg = up->pgrp;
    up->pgrp = newpgrp();
    if(flag & RFNAMEG)
        pgrpcpy(up->pgrp, opg);
    ⟨sysrfork() inherit noattach, RFPROC==0 case 336e⟩
    closepgrp(opg);
}
⟨sysrfork() set noattach to true when RFNOMNT, RFPROC==0 case 337b⟩
```
Uses RFCNAMEG-2555 115b, RFNAMEG-2552 115b, closepgrp() 321a, newpgrp() 320f, pgrpcpy() 341c, and up.

⟨sysrfork() *when no* RFPROC*, clean or copy environment* 122a⟩≡ (121b)
```
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 115b, RFENVG-2553 115b, closeegrp() 563d, envcpy() 564a, smalloc() 85c, and up.

⟨sysrfork() *when no* RFPROC*, new rendez vous group* 122b⟩≡ (121b)
```
if(flag & RFREND) {
    org = up->rgrp;
    up->rgrp = newrgrp();
    closergrp(org);
}
```
Uses RFREND-2559 115b, closergrp() 448d, newrgrp() 448c, and up.

⟨sysrfork() *when no* RFPROC*, new note group* 122c⟩≡ (121b)
```
if(flag & RFNOTEG)
    up->noteid = incref(&noteidalloc);
```
Uses RFNOTEG-2558 115b, incref() 69b, noteidalloc 419b, 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)* 122d⟩≡ (754d)
```
/*
 *  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* 122e⟩≡ (123b) 122f▷
```
char *file;
char *elem; // last element of binary, for up->text
char *file0;
```

⟨sysexec() *locals* 122f⟩+≡ (123b) ◁122e 122g▷
```
Chan *tc;
```

⟨sysexec() *locals* 122g⟩+≡ (123b) ◁122f 122h▷
```
Exec exec;
ulong magic;
user_addr text, entry, data, bss;
```

⟨sysexec() *locals* 122h⟩+≡ (123b) ◁122g 123a▷
```
user_addr t, d, b; // end of text/data/bss in address rounded to page
Segment *s, *ts;
```

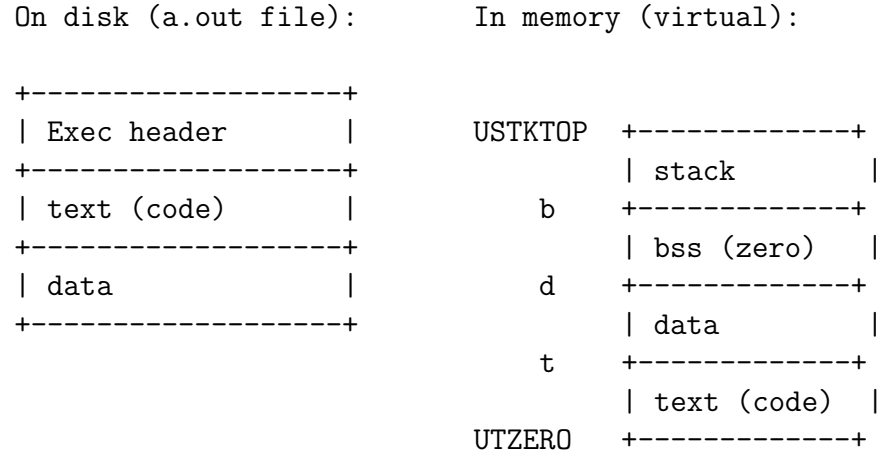
⟨sysexec() *locals* 123a⟩+≡ (123b) ◁122h ??▷
```
ulong ssize;
ulong nargs;
```

⟨*syscall exec* 123b⟩≡ (716a)
```
// void* exec(char *name, char* argv[]);
long
sysexec(ulong* arg)
{
    ⟨sysexec() locals 122e⟩

    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 282c⟩
        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 477b⟩
    }
    // from break above
    data = l2be(exec.data);
    bss = l2be(exec.bss);

    ⟨sysexec() compute end of text, data, bss segments 124⟩


    ⟨sysexec() build args count 127a⟩

    ⟨sysexec() build temporary stack segment and qlock seglock 127e⟩

    ⟨sysexec() build args 128b⟩
```

```
    ⟨sysexec() free old memory segments 125c⟩

    ⟨sysexec() close files marked as opened with close on exec 298b⟩


    ⟨sysexec() set Text segment 125e⟩

    ⟨sysexec() set Data segment 126a⟩

    ⟨sysexec() set Bss segment 126b⟩

    ⟨sysexec() relocate temporary stack segment, unlock seglock 129a⟩

    ⟨sysexec() set prioriry of root processes 199h⟩

    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 420c⟩
    arch_procsetup(up);
    qunlock(&up->debug);

    ⟨sysexec() if hang 455g⟩

    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()`[283d] (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 124⟩≡                (123b)
  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 125a⟩≡ (716a)

```
  ulong
  l2be(long l)
  {
      byte *cp;

      cp = (byte*)&l;
      return (cp[0]<<24) | (cp[1]<<16) | (cp[2]<<8) | cp[3];
  }
```

⟨*macro* UTROUND*(arm)* 125b⟩≡ (747)

```
  #define UTROUND(t)  ROUNDUP((t), BY2PG)
```

⟨sysexec() *free old memory segments* 125c⟩≡ (123b)

```
  /*
   * 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* 436a⟩

`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)* 125d⟩≡ (755)

```
  /*
   *  set mach dependent process state for a new process
   */
  void
  arch_procsetup(Proc* p)
  {
      ⟨arch_procsetup() floating point(arm) 482c⟩
  }
```

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* 125e⟩≡ (123b)

```
  /* Text.  Shared. Attaches to cache image if possible */
  ⟨sysexec() get text segment ts via demand loading on tc 176d⟩
  up->seg[TSEG] = ts;
```

6.5.2 Data segment

⟨sysexec() *set Data segment* 126a⟩≡ (123b)
```
/* Data. Shared. */
s = newseg(SG_DATA, t, (d-t)>>PGSHIFT);
up->seg[DSEG] = s;
⟨sysexec() adjust data segment s for demand loading on tc 176e⟩
```

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* 126b⟩≡ (123b)
```
/* 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* 126c⟩+≡ (49f) ◁118a 131b▷
```
char  *args;
int nargs;    /* number of bytes of args */
```

⟨newproc() *set fields of* p 126d⟩+≡ (111d) ◁114c 131a▷
```
kstrdup(&p->args, "");
p->nargs = 0;
```
Uses kstrdup() 636b.

⟨sysexec() *locals* 126e⟩+≡ (123b) ◁?? 127d▷
```
char **argp;
char *a, *args;
ulong nbytes;
```

⟨sysexec() *build args count* 127a⟩≡ (123b)
```
/*
 * Args: pass 1: count
 */
nargs = 0;
nbytes = 0;

⟨sysexec() nbytes tos adjustments 476j⟩
⟨sysexec() if indir arg adjustments 478b⟩

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)* 127b⟩≡ (747)
```
// end of new stack in sysexec
#define TSTKTOP     (USTKTOP-USTKSIZE)  /* sysexec temporary stack */
```

⟨*constant* TSTKSIZ*(arm)* 127c⟩≡ (747)
```
#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* 127d⟩+≡ (123b) ◁126e 128a▷
```
ulong spage;
```

⟨sysexec() *build temporary stack segment and qlock seglock* 127e⟩≡ (123b)
```
/*
 * 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* 128a⟩+≡ (123b) ◁127d 176c▷

```
char **argv;
char *charp;
```

⟨sysexec() *build args* 128b⟩≡ (123b)

```
/*
 * Args: pass 2: assemble; the pages will be faulted in
 */
⟨sysexec() tos settings 476c⟩

argv = (char**)(TSTKTOP - ssize);
charp = (char*)(TSTKTOP - nbytes);
args = charp;
⟨sysexec() if indir argp adjustments 478c⟩
else
    argp = (char**)arg[1];

for(i=0; i<nargs; i++){
    ⟨sysexec() if indir argp adjustments again 479⟩
    *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* 129a⟩≡ (123b)

```
/*
 * 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` 129b⟩≡ (702)

```
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)* 130a⟩≡ (757b)

```
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* 130b⟩≡ (49f) 131c▷

```
  // option<ref<Proc>> (nil for the boot process)
  Proc  *parent;
  int nchild;   /* Number of living children */
```

⟨newproc() *set fields of* p 131a⟩+≡ (111d) ◁126d 131d▷
```
  p->parent = nil;
  p->nchild = 0;
```

⟨Proc *state fields* 131b⟩+≡ (49f) ◁126c
```
  // 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* 131c⟩+≡ (49f) ◁130b 133b▷
```
  // 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 131d⟩+≡ (111d) ◁131a 139b▷
```
  p->waitq = nil;
  p->nwait = 0;
```

⟨*struct* Waitmsg 131e⟩≡ (693c)
```
  struct Waitmsg
  {
    int pid;    /* of loved one */ // pid of the child
    ⟨Waitmsg time field 472d⟩
    char  msg[ERRMAX];  /* actually variable-size in user mode */
  };
```

⟨*struct* Waitq 131f⟩≡ (705c)
```
  // 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* 132a⟩≡ (716a)
```
// 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 132b⟩
    i = snprint((char*)arg[0], n, "%d %lud %lud %lud %q",
        w.pid,
        ⟨sysawait() snprint time field arguments 473b⟩
        w.msg);

    return i;
}
```
Uses `pwait()` 132c, `true`, and `validaddr()` 101c.

⟨`sysawait()` *sanity check pid* 132b⟩≡ (132a)
```
if(pid < 0)
    return -1;
```

⟨*function* `pwait` 132c⟩≡ (712c)
```
ulong
pwait(Waitmsg *w)
{
    ulong cpid; // child pid
    Waitq *wq;

    ⟨pwait() lock qwaitr 567a⟩

    ⟨pwait() sanity check process has children 133a⟩

    // sleep!
    ⟨pwait() wait until has a waitq 133c⟩
    // 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 567b⟩

    if(w)
        memmove(w, &wq->w, sizeof(Waitmsg));
    cpid = wq->w.pid;
    free(wq);
```

```
    return cpid;
  }
```

Uses free() 85b, lock() 58d, unlock() 60d, and up.

⟨pwait() *sanity check process has children* 133a⟩≡ (132c)

```
  lock(&up->exl);
  if(up->nchild == 0 && up->waitq == nil) {
      unlock(&up->exl);
      error(Enochild);
  }
  unlock(&up->exl);
```

Uses Enochild 631d, lock() 58d, unlock() 60d, 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* 133b⟩+≡ (49f) ◁131c 566a▷

```
  Rendez  waitr;    /* Place to hang out in wait */
```

⟨pwait() *wait until has a waitq* 133c⟩≡ (132c)

```
  sleep(&up->waitr, haswaitq, up); // qwaitr is still locked
```

Uses haswaitq() 133d, sleep, and up.

⟨*function* haswaitq 133d⟩≡ (712c)

```
  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* 133e⟩≡ (716a)

```
  // void exits(char *msg);
  long
  sysexits(ulong* arg)
  {
      char *status;
      ⟨sysexits() other locals 134a⟩

      status = (char*)arg[0];
      ⟨sysexits() sanity check status, copy string in buf 134b⟩
      pexit(status, /*freemem*/true);

      panic("pexit: should never reach this point");
```

```
    return -1; // unreachable
}
```

Uses pexit and true.

⟨sysexits() *other locals* 134a⟩≡ (133e)

```
char *inval = "invalid exit string";
char buf[ERRMAX];
```

Uses ERRMAX 692a.

⟨sysexits() *sanity check status, copy string in buf* 134b⟩≡ (133e)

```
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 692a, false, poperror 633d, validaddr() 101c, vmemchr() 101d, and waserror 633d.

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()[203f]): 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 134c⟩≡ (712c)

```
void
proc_pexit(char *exitstr, bool freemem)
{
    Proc *p; // parent
```

```
    ⟨pexit() other locals 136b⟩

    // free 1
    ⟨pexit() at the start, free resources 228b⟩
    ⟨pexit() hook proctrace 467i⟩
    // free 2
    ⟨pexit() nil out resources under lock and free them 138c⟩

    ⟨pexit() if a kernel process, do nothing about parent and child 140a⟩
    else {
        /*
         * if not a kernel process and have a parent,
         * do some housekeeping.
         */
        p = up->parent;
        ⟨pexit() sanity check parent pointer p 135b⟩
        while(waserror())
            ; // ????

        // exit/await and your parent
        ⟨pexit() build waitq message wq 136c⟩
        ⟨pexit() add wq message in parent process p waitq 136d⟩
    }
    ⟨pexit() if not freemem, add process to broken processes list 451d⟩

    // free 3, must be after built waitq because exitstr in one of the segment
    ⟨pexit() free memory segments 138a⟩

    // exit/await and your (orphan) children
    ⟨pexit() wakeup process waiting on waitr and unhash pid 137a⟩

    // free 4
    ⟨pexit() free waitq 137c⟩
    // free 5
    ⟨pexit() release debuggers 454c⟩

    /* Sched must not loop for these locks */
    lock(&procalloc);
    lock(&palloc);

    ⟨pexit() optional edfstop() for real-time scheduling 533b⟩
    up->state = Moribund;
    // will arch_gotolabel() to schedinit() which has special code around Moribund
    sched();
    panic("pexit: should never reach this point");
}
```

Uses Moribund 135a, lock() 58d, palloc 90d, procalloc-2458 110c, sched, up, and waserror 633d.

```
⟨Procstate cases 135a⟩+≡                                    (51b) ◁111c 210c▷
  Moribund,
```

```
⟨pexit() sanity check parent pointer p 135b⟩≡                              (134c)
  // 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()`[203f] 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* 136a⟩≡ (207a) 207b▷

```
case Moribund:
    up->state = Dead;
    ⟨schedinit() optional real-time edfstop() 531h⟩
    /*
     * Holding locks from pexit:
     *  procalloc
     *  palloc
     */
    arch_mmurelease(up);

    up->qnext = procalloc.free;
    procalloc.free = up;

    unlock(&palloc);
    unlock(&procalloc);
    break;
```

Uses `Dead` 51b, `Moribund` 135a, `palloc` 90d, `procalloc-2458` 110c, `unlock()` 60d, 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* 136b⟩≡ (134c) 137b▷

```
Waitq *wq;
```

⟨`pexit()` *build waitq message* `wq` 136c⟩≡ (134c)

```
wq = smalloc(sizeof(Waitq));
poperror(); // of ????

wq->w.pid = up->pid;
⟨pexit() set wait msg time field 472f⟩
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` 633d, `smalloc()` 85c, and `up`.

Again `smalloc()`—the dying process may sleep here if memory is tight.

⟨`pexit()` *add* `wq` *message in parent process* `p` *waitq* 136d⟩≡ (134c)

```
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 473a⟩
```

```
    /*
     * 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` 451a 451b, `free()` 85b, `lock()` 58d, `unlock()` 60d, `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* 137a⟩≡ (134c)

```
  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()` 58d, `pidunhash()` 111a, `unlock()` 60d, `up`, and `wakeup`.

⟨`pexit()` *other locals* 137b⟩+≡ (134c) ◁136b 137d▷

```
  Waitq *f, *next;
```

⟨`pexit()` *free waitq* 137c⟩≡ (134c)

```
  for(f = up->waitq; f; f = next) {
      next = f->next;
      free(f);
  }
```

Uses `free()` 85b 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* 137d⟩+≡ (134c) ◁137b 138b▷

```
  Segment **s, **es;
```

⟨pexit() *free memory segments* 138a⟩≡ (134c)
```
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 100a, putseg() 98b, qlock() 63f, qunlock() 64c, and up.

⟨pexit() *other locals* 138b⟩+≡ (134c) ◁137d 467h▷
```
Fgrp *fgrp;
Egrp *egrp;
Rgrp *rgrp;
Pgrp *pgrp;
Chan *dot;
```

⟨pexit() *nil out resources under lock and free them* 138c⟩≡ (134c)
```
/* 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() 563d, closefgrp() 277b, closepgrp() 321a, closergrp() 448d, qlock() 63f, qunlock() 64c, 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* 138d⟩≡ (116b)
```
if(flag&RFNOWAIT)
    p->parentpid = 0;
```

```
else {
    lock(&up->exl);
    up->nchild++;
    unlock(&up->exl);
}
```

Uses RFNOWAIT-2551 115b, lock() 58d, unlock() 60d, 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()`[139c], which is simpler than `sysrfork()`[115c]: it calls `newproc()`[111d], sets `kp=true`, stores the entry function in `kpfun`, and makes the process runnable. When scheduled, the process enters `linkproc()`[140c], which enables interrupts and calls the function.

⟨Proc *other fields* 139a⟩≡ (49f) 145c▷

```
bool kp;   /* true if a kernel process */
void  (*kpfun)(void*);
void  *kparg;
```

⟨newproc() *set fields of* p 139b⟩+≡ (111d) ◁131d 141d▷

```
p->kp = false;
```

Uses false.

⟨*function* kproc 139c⟩≡ (712c)

```
// 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 471d⟩
    ready(p);
}
```

Uses PriKproc 198b, eve 557a, false, incref() 69b, kstrdup() 636b, newpgrp() 320f, newproc() 111d, procpriority() 199a, 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* 140a⟩≡ (134c)

```
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)* 140b⟩≡ (754e)

```
/*
 *  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 41a.

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)* 140c⟩≡ (754e)

```
/*
 *  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/#/ctl

Processes can be controlled from outside by writing commands to /proc/#/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* 141a⟩≡ (453b) 459a▷

```
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* 141b⟩≡ (453d)

```
case Proc_exitme:
    arch_spllo();        /* pexit has locks in it */
    pexit("Killed", true);
```

Uses Proc_exitme 141a, pexit, and true.

⟨procctlreq() *CMkill case* 141c⟩≡ (586b)

```
case CMkill:
    switch(p->state) {
    ⟨procctlreq() CMkill case, Broken case 452c⟩
    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 568b, NExit 418c, Proc_exitme 141a, Stopped 453a, postnote, and ready.

⟨newproc() *set fields of* p 141d⟩+≡ (111d) ◁139b 193c▷

```
p->pdbg = nil;
p->procctl = Proc_nothing;
p->syscalltrace = nil;
p->trace = false;
p->dbgreg = nil;
```

Uses Proc_nothing 453b 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)* 143a⟩≡ (28a) 527b▷

```
#define PsrMirq     0x00000012
#define PsrMabt     0x00000017
#define PsrMund     0x0000001B
```

⟨*global* `trapnames` *(arm)* 143b⟩≡ (750b)

```
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) 527c⟩
};
```

Uses `PsrMask` 28a.

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()`[256e] 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)* 143c⟩≡ (751b)

```
/*
 *  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)* 143d⟩≡ (751b)

```
//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)* 144a⟩≡ (747)
```
#define CPU     10      /* R10 is cpu-> */
```

⟨*constant* UP*(arm)* 144b⟩≡ (747)
```
#define UP      9       /* R9 is up-> */
```

⟨*function* _vsvc*(arm)* 144c⟩≡ (751b)
```
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) 512d⟩

    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` 145a⟩≡ (705c)
```
#define MAXSYSARG  5 /* for mount(fd, afd, mpt, flag, arg) */
```

⟨*struct* `Sargs` 145b⟩≡ (705c)
```
// syscall arguments copied from user stack
struct Sargs
{
    ulong args[MAXSYSARG];
};
```

⟨`Proc` *other fields* 145c⟩+≡ (49f) ◁139a 233d▷
```
Sargs sargs;    /* address of this is known by db */
```

⟨`sysrfork()` *when* `RFPROC` *propagate fields from up to p* 145d⟩+≡ (116b) ◁117b 193d▷
```
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)* 145e⟩≡ (757b)
```
void
arch__syscall(Ureg* ureg)
{
    int scallnr;
    long ret;
    user_addr sp;
    ⟨syscall() other locals(arm) 147a⟩

    ⟨syscall() sanity check call from userspace(arm) 147g⟩
    ⟨syscall() adjust kentry 476f⟩

    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) 483d⟩

    arch_spllo();
    sp = ureg->sp;

    ⟨syscall() Proc_tracesyscall if, syscall entry(arm) 459c⟩

    up->nerrlab = 0;
    ret = -1;

    startns = todget(nil);
    if(!waserror()){
        ⟨syscall() sanity check scallnr(arm) 147b⟩
        ⟨syscall() sanity check sp(arm) 147c⟩

        // 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) 147d⟩

    /*
     *  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) 459e⟩

    up->insyscall = false;
    up->psstate = nil;

    ⟨syscall() call noted() 423a⟩
    ⟨syscall() call notify(arm) 420g⟩

    ⟨syscall() if delaysched(arm) 208e⟩
    arch__kexit(ureg);
}
```

Uses arch__kexit() 151, false, poperror 633d, sysctab, systab 41f, todget() 239b, true, up, and waserror 633d.

⟨syscall() *other locals(arm)* 147a⟩≡ (145e) 459d ▷
```
char *e;
u32int s;
int i;
```

⟨syscall() *sanity check scallnr(arm)* 147b⟩≡ (145e)
```
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 631d, NDebug 418c, nsyscall 42a, postnote, pprint, and up.

⟨syscall() *sanity check sp(arm)* 147c⟩≡ (145e)
```
if(sp < (USTKTOP-BY2PG) || sp > (USTKTOP-sizeof(Sargs)-BY2WD))
    validaddr(sp, sizeof(Sargs)+BY2WD, 0);
```
Uses validaddr() 101c.

⟨syscall() *if nerrlab(arm)* 147d⟩≡ (145e)
```
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 633a, print, and up.

⟨Proc *debugger fields* 147e⟩≡ (49f) 453c ▷
```
void  *dbgreg;  /* User registers for devproc */
```

⟨*function* arch_userpc*(arm)* 147f⟩≡ (754e)
```
/*
 *  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)* 147g⟩≡ (145e)
```
if(!arch_userureg(ureg))
    panic("syscall: from kernel: pc %#lux r14 %#lux psr %#lux",
        ureg->pc, ureg->r14, ureg->psr);
```

⟨*function* arch_userureg*(arm)* 147h⟩≡ (754e)
```
bool
arch_userureg(Ureg* ureg)
{
    return (ureg->psr & PsrMask) == PsrMusr;
}
```
Uses PsrMask 28a and PsrMusr 28a.

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)* 148a⟩≡ (751b)

```
TEXT _vund(SB), 1, $-4          /* undefined */
    MOVM.IA [R0-R4], (R13)      /* free some working space */
    MOVW    $PsrMund, R0
    B   _vswitch
```

⟨*function* `_vpabt` *(arm)* 148b⟩≡ (751b)

```
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)* 148c⟩≡ (751b)

```
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)* 148d⟩≡ (751b)

```
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)* 149a⟩≡ (751b)

```
      /*
       *  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 150a⟩
      /* here for trap from USER mode */
      ⟨_vswitch trap from USER mode 149b⟩
```

⟨`_vswitch` *trap from USER mode* 149b⟩≡ (149a)

```
  _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) 513a⟩

      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* 150a⟩≡ (149a)

```
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()`[145e] 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)* 150b⟩≡ (750b)

```
/*
 *  here on all exceptions other than syscall (SWI) and fiq
 */
void
arch__trap(Ureg *ureg)
{
    bool user;
    bool clockintr;
    ⟨trap() other locals(arm) 152b⟩

    ⟨trap() sanity check interrupts are disabled(arm) 152a⟩
    ⟨trap() sanity check enough space in kernel stack(arm) 152c⟩

    user = (ureg->psr & PsrMask) == PsrMusr; // TODO: arch_userureg
    if(user){
        up->dbgreg = ureg;
        ⟨trap() adjust kentry when interrupt user 476g⟩
    }

    /*
     * 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) 152e⟩
    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 208d⟩

    if(user){
        ⟨trap() call possibly notify(arm) 420e⟩
        arch__kexit(ureg);
    }
}
```

Uses PsrMabt 143a, PsrMask 28a, PsrMusr 28a, arch__kexit() 151, 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)* 151⟩≡ (754e)

```
/* go to user space */
void
arch__kexit(Ureg*)
{
    ⟨arch__kexit() tos adjustments 476d⟩
    tos->cyclefreq = cpu->cpuhz;
    ⟨arch__kexit() write back cache(arm) 498m⟩
```

```
}
```

⟨trap() *sanity check interrupts are disabled(arm)* 152a⟩≡ (150b)
```
assert(!arch_islo());
```
Uses assert 693a.

⟨trap() *other locals(arm)* 152b⟩≡ (150b) 152d▷
```
int rem;
```

⟨trap() *sanity check enough space in kernel stack(arm)* 152c⟩≡ (150b)
```
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)* 152d⟩+≡ (150b) ◁152b 153b▷
```
char buf[ERRMAX];
```
Uses ERRMAX 692a.

⟨trap() *switch exception type cases(arm)* 152e⟩≡ (150b) 153c▷
```
case PsrMund:           /* undefined instruction */
    if(user){
        ⟨trap() when undefined instruction, if breakpoint(arm) 450a⟩
        else{
            ⟨trap() when undefined instruction, if float instr(arm) 494a⟩
            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) 153a⟩
        iprint("undefined instruction: pc %#lux inst %#ux\n",
            ureg->pc, *(u32int*)ureg->pc);
        panic("undefined instruction");
    }
    break;
```
Uses NDebug 418c, PsrMund 143a, iprint, postnote, and up.

⟨trap() *when undefined instruction, round pc(arm)* 153a⟩≡ (152e)
```
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()`[164b], 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)* 153b⟩+≡ (150b) ◁152d 153f▷
```
int x;
ulong fsr;
```

⟨trap() *switch exception type cases(arm)* 153c⟩+≡ (150b) ◁152e 154a▷
```
case PsrMabt:           /* prefetch fault */
    x = ifsrget();
    fsr = (x>>7) & 0x8 | x & 0x7;
    switch(fsr){
    ⟨trap() when prefetch fault, switch fst cases(arm) 450b⟩
    default:
        // Page fault on code!!
        faultarm(ureg, ureg->pc, user, true);
        break;
    }
    break;
```
Uses PsrMabt 143a, faultarm() 164b, and true.

⟨*function* ifsrget*(arm)* 153d⟩≡ (751a)
```
TEXT ifsrget(SB), 1, $-4            /* instruction fault status */
    MRC CpSC, 0, R0, C(CpFSR), C(0), CpFSRinst
    RET
```

⟨*type* CpFSR*(arm)* 153e⟩≡ (720c)
```
/*
 * 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()`[164b]) 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)* 153f⟩+≡ (150b) ◁153b 493d▷
```
virt_addr va; // can be a kern_addr?
ulong inst;
```

⟨trap() *switch exception type cases(arm)* 154a⟩+≡ (150b) ◁153c 159a▷
```
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) 154d⟩
    default:
    case 0xa:       /* ? was under external abort */
        panic("unknown data fault, 6b fsr %#lux", fsr);
        break;
    }
    break;
```
Uses PsrMabt 143a.

⟨*function* farget*(arm)* 154b⟩≡ (751a)
```
TEXT farget(SB), 1, $-4             /* fault address */
    MRC CpSC, 0, R0, C(CpFAR), C(0x0)
    RET
```

⟨*function* fstget*(arm)* 154c⟩≡ (751a)
```
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)* 154d⟩≡ (154a) 155a▷
```
case 0xd:
case 0xf:
    /* permission error, copy on write or real permission error */
    faultarm(ureg, va, user, !writetomem(inst));
    break;
```
Uses faultarm() 164b and writetomem() 154e.

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)* 154e⟩≡ (750b)
```
/*
 *  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)* 155a⟩+≡ (154a) ◁154d 155b▷
```
  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()` 164b and `writetomem()` 154e.

⟨trap() *when data fault, switch fst cases(arm)* 155b⟩+≡ (154a) ◁155a 155c▷
```
  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` 418c, `postnote`, and `up`.

⟨trap() *when data fault, switch fst cases(arm)* 155c⟩+≡ (154a) ◁155b 155d▷
```
  case 0x0:
      panic("vector exception at %#lux", ureg->pc);
      break;
```

⟨trap() *when data fault, switch fst cases(arm)* 155d⟩+≡ (154a) ◁155c 155e▷
```
  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` 418c, `postnote`, and `up`.

⟨trap() *when data fault, switch fst cases(arm)* 155e⟩+≡ (154a) ◁155d 155f▷
```
  case 0x2:
      panic("terminal exception at %#lux", ureg->pc);
      break;
```

⟨trap() *when data fault, switch fst cases(arm)* 155f⟩+≡ (154a) ◁155e 156a▷
```
  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)* 156a⟩+≡ (154a) ◁155f
```
  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()`[157e], and when an IRQ fires, `irq()`[159b] 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)* 156b⟩≡ (722a)
```
  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) 158c⟩
      ⟨IRQ local cases(arm) 519⟩
      ⟨IRQ other cases(arm) 528b⟩
  };
```
Uses `IRQtimer3` 156b.

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)* 156c⟩≡ (750b)
```
  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) 520a⟩

    // Extra
    ⟨Vctl extra fields(arm) 157b⟩
};
```

⟨*global* `vctl` *(arm)* 157a⟩≡ (750b)
```
// list<ref_own<Vctl>>> (next = Vctl.next)
static Vctl *vctl;
```

⟨`Vctl` *extra fields(arm)* 157b⟩≡ (156c)
```
Vctl    *next;
```

⟨*global* `vctllock` *(arm)* 157c⟩≡ (750b)
```
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)* 157d⟩≡ (750b)
```
/*
 * 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()`[157e] 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)* 157e⟩≡ (733b)
```
#define arch_intrenable(i, f, a, b, n) irqenable((i), (f), (a))
```

⟨*function* `irqenable` *(arm)* 158a⟩≡ (750b)
```
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) 158b⟩
    v->irq = irq;
    v->cpu = 0;
    ⟨irqenable() if IRQlocal(arm) 520b⟩
    ⟨irqenable() if IRQbasic(arm) 158d⟩
    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) 528c⟩
    else{
        v->next = vctl;
        vctl = v;
        ⟨irqenable() if IRQlocal adjust enable(arm) 520c⟩
        else
            *enable = v->mask;
    }
    unlock(&vctllock);
}
```
Uses `INTREGS-2403` 31g, `lock()` 58d, `malloc()` 85a, `unlock()` 60d, `vctl-2412` 157a, and `vctllock-2411` 157c.

⟨`irqenable()` *sanity check v(arm)* 158b⟩≡ (158a)
```
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)* 158c⟩≡ (156b)
```
IRQbasic    = 64,
IRQtimerArm = IRQbasic + 0,
```
Uses `IRQbasic` 158c.

⟨`irqenable()` *if IRQbasic(arm)* 158d⟩≡ (158a)
```
else if(irq >= IRQbasic){
    enable = &ip->ARMenable;
    v->reg = &ip->ARMpending;
    v->mask = 1 << (irq - IRQbasic);
}
```
Uses `IRQbasic` 158c.

7.6.5 irq() (ARM)

`irq()`[159b] 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)* 159a⟩+≡ (150b) ◁154a

```
case PsrMirq:
    clockintr = irq(ureg);
    cpu->intr++;
    break;
```

Uses `PsrMirq` 143a and `irq()` 159b.

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)* 159b⟩≡ (750b)

```
/*
 *  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) 501c⟩
            // Dispatch
            v->f(ureg, v->a);
            ⟨irq() after dispatch(arm) 501d⟩
            if(v->irq == IRQclock || v->irq == IRQcntps || v->irq == IRQcntpns)
                clockintr = true;
        }
    if(!found)
        cpu->spuriousintr++;
    intrtime();
    return clockintr;
}
```

Uses `IRQclock` 156b, `IRQcntpns` 519, `IRQcntps` 519, `false`, `true`, and `vctl-2412` 157a.

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()`[164b]. 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)* 160a⟩≡ (748b)
```
#define L1X(va)     FEXT((va), 20, 12)
```

⟨*macro* `L2X`*(arm)* 160b⟩≡ (748b)
```
#define L2X(va)     FEXT((va), 12, 8)
```

⟨*macro L1X bis(arm)* 160c⟩≡ (721)
```
#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)* 160d⟩≡ (720c)
```
#define Coarse      (Mbz|1)         /* L1 */
#define Section     (Mbz|2)         /* L1 1MB */
```

⟨*constant* `Mbz`*(arm)* 160e⟩≡ (720c)
```
/* Mbz (0x10) bit is implementation-defined and must be 0 on the cortex. */
#define Mbz     (0<<4)
```

⟨*type* `PageTableGranularity`*(arm)* 160f⟩≡ (720c)
```
#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)* 161a⟩≡ (720c)
```
#define Client      1           /* DAC */
#define Manager     3           /* DAC */
```

⟨*type* PageTableEntryAttribute3*(arm)* 161b⟩≡ (720c)
```
#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)* 161c⟩≡ (720c)
```
#define Fault       0x00000000      /* L[12] pte: unmapped */
```

⟨*constant* Dom0*(arm)* 161d⟩≡ (720c)
```
#define Dom0        0
```

⟨*macro* L1AP*(arm)* 161e⟩≡ (720c)
```
#define L1AP(ap)    (AP(3, (ap)))
```

⟨*macro* AP*(arm)* 161f⟩≡ (720c)
```
#define AP(n, v)    F((v), ((n)*2)+4, 2)
```

⟨*macro* F*(arm)* 161g⟩≡ (720c)
```
#define F(v, o, w)  (((v) & ((1<<(w))-1))<<(o))
```

⟨*macro* DAC*(arm)* 161h⟩≡ (720c)
```
#define DAC(n, v)   F((v), (n)*2, 2)
```

⟨*macro* L2AP*(arm)* 161i⟩≡ (748b)
```
#define L2AP(ap)    l2ap(ap)
```

⟨*function* l2ap*(raspberry pi1)(arm)* 161j⟩≡ (744b)
```
int
l2ap(int ap)
{
    return (AP(3, (ap))|AP(2, (ap))|AP(1, (ap))|AP(0, (ap)));
}
```
Uses AP 161f.

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)* 161k⟩≡ (48e) 169b▷
```
PTE*    mmul1;      /* l1 for this processor */
```

⟨*type* PTE*(arm)* 161l⟩≡ (732c)
```
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)* 162a⟩≡ (748a)
```
L1lo        = UZERO/MiB,        /* L1X(UZERO)? */
```
Uses `MiB` 34g.

⟨*constant* `L1hi` *(arm)* 162b⟩≡ (748a)
```
L1hi        = (USTKTOP+MiB-1)/MiB,  /* L1X(USTKTOP+MiB-1)? */
```
Uses `MiB` 34g.

⟨`Proc Arch` *memory fields(arm)* 162c⟩≡ (50) 163b▷
```
// list<ref<Page> > pages used to store page tables
Page*   mmul2;
```

⟨*macro* `PPN` *(arm)* 162d⟩≡ (747)
```
#define PPN(x)      ((x)&~(BY2PG-1))
```

⟨*macro* `VA` *(arm)* 162e⟩≡ (746b)
```
#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)* 162f⟩≡ (748b)
```
void
arch_mmurelease(Proc* proc)
{
    Page *page, *next;

    ⟨arch_mmurelease() invalidate cache before adjusting page tables(arm) 498i⟩
    mmul2empty(proc, false);
    ⟨arch_mmurelease() free l2 page cache(arm) 163d⟩
    mmul1empty();
    ⟨arch_mmurelease() write back cache after adjusting page tables(arm) 498b⟩
    /* lose any possible stale tlb entries */
    mmuinvalidate();
}
```
Uses `false`, `mmul1empty()` 163e, and `mmul2empty()` 163a.

⟨*function* `mmuinvalidate` *(arm)* 162g⟩≡ (749a)
```
/*
 * 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)* 162h⟩≡ (720c)
```
/*
 * CpTLB Secondary (CRm) registers and opcode2 fields.
 */
#define CpTLBinvu   7           /* unified */
```

⟨*type CpTLB opcode2(arm)* 162i⟩≡ (720c)
```
#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)* 163a⟩≡ (748b)
```
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) 163c⟩
}
```
Uses `Fault` 161c and `UINT2PTR` 41b.

⟨`Proc Arch` *memory fields(arm)* 163b⟩+≡ (50) ◁162c
```
// list<ref<Page> > (next = Page.next)
Page*   mmul2cache; /* free mmu pages */
```

⟨`mmul2empty()` *remember free pages(arm)* 163c⟩≡ (163a)
```
*l2 = proc->mmul2cache;
proc->mmul2cache = proc->mmul2;
proc->mmul2 = nil;
```

⟨`arch_mmurelease()` *free l2 page cache(arm)* 163d⟩≡ (162f)
```
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()` 93a, `palloc` 90d, and `wakeup`.

⟨*function* `mmul1empty`*(arm)* 163e⟩≡ (748b)
```
static void
mmul1empty(void)
{
    memset(&cpu->mmul1[L1lo], 0, (L1hi - L1lo)*sizeof(PTE));
}
```
Uses `L1hi-1848` 162b and `L1lo-1847` 162a.

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* 163f⟩+≡ (49f) ◁100b 193b▷
```
bool newtlb;    /* Pager has changed my pte's, I must flush */
```

⟨*function* `arch_flushmmu`*(arm)* 164a⟩≡ (748b)

```
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)* 164b⟩≡ (750b)

```
/*
 *  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 614d⟩
    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 692a, NDebug 418c, fault() 165, postnote, true, and up.

8.4.2 fault()

The portable fault() function finds which segment contains the faulting address (via seg()[100c]) 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 165⟩≡ (700)

```
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 694, fixfault() 166b, print, qunlock() 64c, seg() 100c, 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` 166a⟩≡ (694)

```
#define PTEVALID    (1<<0)
#define PTERONLY    0
#define PTEWRITE    (1<<1)
⟨PTExxx other cases 525a⟩
```

⟨*function* `fixfault` 166b⟩≡ (700)

```
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 177e⟩

        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 177f⟩

        ⟨fixfault() if read and copy on write, adjust mmupte and break 170b⟩
        // 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 180a⟩
        unlock(lkp);

        ⟨fixfault() if write and more than one ref then copy page 170c⟩

        mmupte = PPN((*pte)->pa) | PTEWRITE | PTEVALID;
        (*pte)->modref = PG_MOD|PG_REF;
        break;

    ⟨fixfault() SG_PHYSICAL case 524d⟩

    default:
        panic("fault");
        break;
    }
    qunlock(&s->lk);

    if(doputmmu)
        arch_putmmu(addr, mmupte, *pte);

    return OK_0;
}
```

Uses ERROR_NEG1, OK_0, PG_MOD 90a, PG_REF 90a, PTERONLY 166a, PTEVALID 166a, PTEWRITE 166a, SG_BSS, SG_DATA 523b, SG_SHARED, SG_STACK 94a, SG_TEXT, SG_TYPE, lock() 58d, nilptr 693c, ptalloc() 96c, qunlock() 64c, swapcount() 187g, swapimage 185d, true, and unlock() 60d.

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)* 167⟩≡ (748b)

```
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) 168⟩
        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) 498h⟩

        *l1 = PPN(pg->pa)|Dom0|Coarse;
        ⟨arch_putmmu() write back cache after changing pde(arm) 498f⟩

        ⟨arch_putmmu() maintain mmul1lo and mmul1hi(arm) 169c⟩
    }
    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) 525b⟩
    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) 498g⟩

    /* 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) 522c⟩
    arch_checkmmu(va, PPN(pa));
}
```

Uses Coarse 160d, Dom0 161d, Fault 161c, L1X-1842 160a, L2AP-1844 161i, L2X-1843 160b, L2ptedramattrs-1846 497b, PTEWRITE 166a, Small 160f, UINT2PTR 41b, Uro 161b, Urw 161b, true, and up.

⟨arch_putmmu() *if mmul2cache not empty(arm)* 168⟩≡ (167)

```
if(up->mmul2cache != nil){
    pg = up->mmul2cache;
    up->mmul2cache = pg->next;
    memset(UINT2PTR(pg->va), 0, BY2PG);
}
```

Uses UINT2PTR 41b and up.

⟨*function* mmuinvalidateaddr*(arm)* 169a⟩≡ (749a)

```
/*
 * 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)* 169b⟩+≡ (48e) ◁161k 209a▷

```
// index in mmul1
int mmul1lo;
int mmul1hi;
```

⟨arch_putmmu() *maintain mmul1lo and mmul1hi(arm)* 169c⟩≡ (167)

```
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)* 169d⟩≡ (748b)

```
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)* 169e⟩≡ (748b)

```
Arch_KMap*
arch_kmap(Page *p) {
  return (Arch_KMap*)((p)->pa|KZERO);
}
```

⟨*function* arch_kunmap*(arm)* 169f⟩≡ (748b)

```
void
arch_kunmap(Arch_KMap *k)
{
    ⟨arch_kunmap() write back cache for page(arm) 498j⟩
}
```

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* 170a⟩≡ (46a) 185c ▷
```
int copymode; /* 0 is copy on write, 1 is copy on reference */
```

⟨`fixfault()` *if read and copy on write, adjust mmupte and break* 170b⟩≡ (166b)
```
/*
 *  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` 90a, `PTERONLY` 166a, `PTEVALID` 166a, `conf` 46b, and `false`.

⟨`fixfault()` *if write and more than one ref then copy page* 170c⟩≡ (166b)
```
if(ref > 1){
    new = newpage(false, &s, addr);
    if(s == nil)
        return -1;
    *pte = new;
    copypage(lkp, *pte);
    putpage(lkp); //why??
}
```
Uses `copypage()` 170d, `false`, and `putpage()` 92d.

`copypage()` performs the actual page copy via temporary kernel mappings (`arch_kmap`):

⟨*function* `copypage` 170d⟩≡ (701)
```
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).

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* 171a⟩≡ (89) 195c▷
```
// option<ref<Kimage>>
KImage  *image;     /* Associated binary image or swap image */
ulong daddr;      /* Disc address on image */
```

⟨*function* `pghash` 171b⟩≡ (694)
```
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* 171c⟩≡ (90b)
```
// hash<Page.daddr, ref<Page>> (next = Page.hash)>
Page  *hash[PGHSIZE];
Lock  hashlock;
```

⟨`Page` *extra fields* 171d⟩+≡ (89) ◁90c
```
// 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` 171e⟩≡ (701)
```
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()` 69b, `lock()` 58d, `palloc` 90d, `pghash` 90b, and `unlock()` 60d.

`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` 172a⟩≡ (701)

```
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()` 58d, `pageunchain()` 92c, `palloc` 90d, `pghash` 90b, and `unlock()` 60d.

⟨`putpage` *if p has an image* 172b⟩≡ (92d)

```
if(p->image && p->image != &swapimage)
    pagechaintail(p);
```

Uses `pagechaintail()` 93b and `swapimage` 185d.

⟨`newpage()` *uncachepage* 172c⟩≡ (91c)

```
uncachepage(p);
```

⟨*function* `uncachepage` 172d⟩≡ (701)

```
/* 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()` 58d, `palloc` 90d, `pghash` 90b, `putimage()` 182d, and `unlock()` 60d.

`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` 173⟩≡ (701)

```
    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()` 58d, `palloc` 90d, `pghash` 90b, `putimage()` 182d, and `unlock()` 60d.

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 174a⟩≡ (694)
```
// 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 176a⟩

    // extra
    Ref;
    ⟨Kimage extra fields 174c⟩
};
```

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 174b⟩≡ (694)
```
// 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 183a⟩

    // extra
    Lock;
};
```

⟨Kimage *extra fields* 174c⟩≡ (174a) 175a▷
```
// list<ref<Kimage>> of Imagealloc.free
KImage  *next; /* Free list */
// hash<qid.path, ref<Kimage>> Imagealloc.hash
KImage  *hash; /* Qid hash chains */
```

⟨*function* ihash 174d⟩≡ (694)
```
#define IHASHSIZE 64
// actually internal to page.c, but important so here
#define ihash(qidpath)  imagealloc.hash[qidpath%IHASHSIZE]
```

⟨*global* imagealloc 174e⟩≡ (702)
```
static struct Imagealloc imagealloc;
```
Uses Imagealloc 174b.

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* 174f⟩+≡ (94c) ◁99a 176b▷
```
KImage  *image;   /* text in file attached to this segment */
```

⟨Kimage *extra fields* 175a⟩+≡ (174a) ◁174c
```
// option<ref<Segment>>?
Segment *s;     /* TEXT segment for image if running */
```

⟨*constructor* attachimage 175b⟩≡ (702)
```
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 631d, ihash 174b, imagealloc-1803 174e, imagechanreclaim() 183b, incref() 69b, lock() 58d, newseg() 97b, pexit, poperror 633d, sched, true, unlock() 60d, and waserror 633d.

⟨Kimage *other fields* 176a⟩≡ (174a)

```
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* 176b⟩+≡ (94c) ◁174f 442b▷

```
ulong fstart;   /* start address in file for demand load */
ulong flen;   /* length of segment in file */
```

⟨sysexec() *locals* 176c⟩+≡ (123b) ◁128a 298a▷

```
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* 176d⟩≡ (125e)

```
/* 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 522f⟩
unlock(img);
```

⟨sysexec() *adjust data segment s for demand loading on tc* 176e⟩≡ (126a)

```
/* 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* 177a⟩≡ (103f)
```
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()` 69b.

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* 177b⟩≡ (103f)
```
if(segno == TSEG){ // why not SG_TEXT then?
    poperror();
    qunlock(&s->lk);
    return data2txt(s);// ????
}
```
Uses `TSEG` 100a, `data2txt()` 177c, `poperror` 633d, and `qunlock()` 64c.

⟨*function* `data2txt` 177c⟩≡ (702)
```
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 522h⟩

    return ps;
}
```
Uses `SG_TEXT`, `incref()` 69b, and `newseg()` 97b.

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` 177d⟩≡ (694)
```
#define pagedout(s) (((ulong)s)==0 || onswap(s))
```

⟨`fixfault()` *page in for* `SG_TEXT` *pte if pagedout* 177e⟩≡ (166b)
```
if(pagedout(*pte))
    pio(s, addr, soff, pte);
```
Uses `pagedout` 694 and `pio()` 178.

⟨`fixfault()` *page in for* `SG_DATA` *or swapin (*`SG_BSS`*, etc) pte if pagedout* 177f⟩≡ (166b)
```
if(pagedout(*pte))
    pio(s, addr, soff, pte);
```
Uses `pagedout` 694 and `pio()` 178.

`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` 178⟩≡ (700)

```
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 522l⟩
  }
```

Uses Eioload 631d, cachepage() 171e, false, faulterror() 179, lookpage() 172a, pagedout 694, poperror 633d, putpage() 92d, putswap() 187b, qlock() 63f, qunlock() 64c, swapaddr 523d, swapimage 185d, up, and waserror 633d.

⟨*function* faulterror 179⟩≡ (700)

```
  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 692a, NDebug 418c, 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* 180a⟩≡ (166b)

```
if(ref == 1 && lkp->image){
    /* save a copy of the original for the image cache */
    duppage(lkp);
    ref = lkp->ref;
}
```

Uses duppage() 180c.

⟨*global* dupretries 180b⟩≡ (701)

```
static int dupretries = 15000;
```

Uses dupretries-1850 180b.

⟨*function* duppage 180c⟩≡ (701)

```
/* 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() 171e, canlock() 60f, copypage() 170d, dupretries-1850 180b, lock() 58d, pagechaintail() 93b, pageunchain() 92c, palloc 90d, print, sched, uncachepage() 172d, unlock() 60d, 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* 182a⟩≡ (98b)
```
KImage *img;
```

⟨`putseg()` *if s has an image* 182b⟩≡ (98b)
```
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()` 58d and `unlock()` 60d.

⟨`putseg()` *if s had an image* 182c⟩≡ (98b)
```
if(img)
    putimage(img);
```
Uses `putimage()` 182d.

`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` 182d⟩≡ (702)
```
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 174d, QTFILE 52c, free() 85b, ihash 174b, imagealloc-1803 174e, lock() 58d, malloc() 85a, and unlock() 60d.

⟨Imagealloc *other fields* 183a⟩≡ (174b)

```
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 183b⟩≡ (702)

```
/*
 *  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()` 65a, `cclose`, `imagealloc-1803` 174e, `lock()` 58d, `qunlock()` 64c, and `unlock()` 60d.

`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` 184⟩≡ (702)

```
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` 185a⟩≡ (694)
```
// 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` 185b⟩≡ (697a)
```
// Swap allocator
struct Swapalloc swapalloc;
```
Uses `Swapalloc` 185a.

⟨`Conf` *other fields* 185c⟩+≡ (46a) ◁170a 189a▷
```
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` 185d⟩≡ (697a)
```
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` 185e⟩≡ (703a)
```
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()` 58d, `swapalloc` 185b, and `unlock()` 60d.

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* 186a⟩≡ (366a) 242b▷

```
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` 600a, `conf` 46b, `imagmem` 84a, `mainmem` 84a, `palloc` 90d, `readstr()` 637d, and `swapalloc` 185b.

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` 186b⟩≡ (694)

```
#define PG_ONSWAP 1
```

⟨*type* `PageOrSwap` 186c⟩≡ (694)

```
typedef Page PageOrSwap;
```

⟨*function* onswap 187a⟩≡ (694)
```
#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 187b⟩≡ (703a)
```
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()` 58d, `swapalloc` 185b, and `unlock()` 60d.

⟨`putpage` *if p is a swap address* 187c⟩≡ (92d)
```
    if(onswap(p)) {
        putswap(p);
        return;
    }
```
Uses `onswap` 196b and `putswap()` 187b.

⟨`mfreeseg` *if pg is a swap address* 187d⟩≡ (107)
```
if(onswap(pg))
    putswap(pg);
```
Uses `onswap` 196b and `putswap()` 187b.

⟨*function* dupswap 187e⟩≡ (703a)
```
void
dupswap(Page *p)
{
    lock(&swapalloc);
    if(++swapalloc.swmap[((ulong)p)/BY2PG] == 0)
        panic("dupswap");
    unlock(&swapalloc);
}
```
Uses `lock()` 58d, `swapalloc` 185b, and `unlock()` 60d.

⟨`ptcpy()` *if src is a swap page* 187f⟩≡ (97a)
```
if(onswap(*src))
    dupswap(*src);
```
Uses `dupswap()` 187e and `onswap` 196b.

⟨*function* swapcount 187g⟩≡ (703a)
```
int
swapcount(ulong daddr)
{
    return swapalloc.swmap[daddr/BY2PG];
}
```
Uses `swapalloc` 185b.

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 188⟩≡ (703a)

```
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 523b, SG_SHARED, SG_STACK 94a, SG_TEXT, cachedel() 173, cachepage() 171e, iolist-1807 189b, ioptr-1808 189c, lock() 58d, newswap() 185e, putpage() 92d, swapimage 185d, uncachepage() 172d, and unlock() 60d.

⟨Conf *other fields* 189a⟩+≡ (46a) ◁185c 254b▷

```
int nswppo;   /* max # of pageouts per segment pass */
```

⟨*global* iolist 189b⟩≡ (703a)

```
// array<option<ref<Page>>>, xalloc'ed in swapinit(), size = Conf.nswppo
static  Page    **iolist;
```

⟨*global* ioptr 189c⟩≡ (703a)

```
// 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 189d⟩≡ (703a)

```
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 90a, SG_TYPE, canflush() 194a, canqlock() 65a, conf 46b, genage-1809 195b, genclock-1810 195b, gencount-1811 195b, gensum-1812 195b, ioptr-1808 189c, pagedout 694, pagepte() 188, poperror 633d, putseg() 98b, qunlock() 64c, and waserror 633d.

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* 190a⟩+≡ (91c) ◁92a 521h▷

```
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* 190b⟩≡ (91c)

```
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* 191a⟩≡ (90b)
```
Rendez  freememr; /* Sleep for free mem */ // hasfreepages()
QLock pwait; /* Queue of procs waiting for memory */
```

⟨putpage *wakeup process waiting for freepage if any* 191b⟩≡ (92d)
```
if(palloc.freememr.p != nil)
    wakeup(&palloc.freememr);
```
Uses palloc 90d 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 191c⟩≡ (701)
```
int
hasfreepages(void*)
{
    return palloc.freecount >= swapalloc.highwater;
}
```
Uses palloc 90d and swapalloc 185b.

`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 191d⟩≡ (703a)
```
void
kickpager(void)
{
    static bool started;

    if(started)
```

```
        wakeup(&swapalloc.r);
    else {
        kproc("kpager", pager, nil);
        started = true;
    }
}
```

Uses `kproc()` 139c, `pager()` 192, `swapalloc` 185b, `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` 192⟩≡ (703a)

```
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 51b, NSEG 100a, SG_BSS, SG_DATA 523b, SG_SHARED, SG_STACK 94a, SG_TEXT, SG_TYPE, canqlock() 65a, conf 46b, executeio() 195d, freebroken() 452b, gentick() 195a, ioptr-1808 189c, killbig() 711c, needpages() 193a, pageout() 189d, palloc 90d, print, proctab, qunlock() 64c, returnfalse() 86a, sleep, swapalloc 185b, swapimage 185d, tsleep, up, wakeup, and waserror 633d.

⟨*function* needpages 193a⟩≡ (703a)

```
static bool
needpages(void*)
{
    return palloc.freecount < swapalloc.headroom;
}
```

Uses palloc 90d and swapalloc 185b.

⟨Proc *memory fields* 193b⟩+≡ (49f) ◁163f

```
bool noswap;   /* process is not swappable */
```

⟨newproc() *set fields of* p 193c⟩+≡ (111d) ◁141d 198d▷

```
p->noswap = false;
```

Uses false.

⟨sysrfork() *when* RFPROC *propagate fields from up to p* 193d⟩+≡ (116b) ◁145d 199c▷

```
p->noswap = up->noswap;
```

Uses up.

⟨procctlreq() *CMnoswap case* 193e⟩≡ (586b)

```
case CMnoswap:
    p->noswap = true;
    break;
```

Uses CMnoswap-884 568b 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` 194a⟩≡ (703a)

```
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` 51b, `NSEG` 100a, `canpage()` 194b, `conf` 46b, `false`, `lock()` 58d, `proctab`, `true`, and `unlock()` 60d.

⟨*function* `canpage` 194b⟩≡ (712c)

```
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()` 58d, `runq` 200b, `true`, and `unlock()` 60d.

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` 195a⟩≡ (703a)
```
static void
gentick(void)
{
    genclock++;
    if(gencount)
        genage = gensum / gencount;
    else
        genage = 0;
    gensum = gencount = 0;
}
```
Uses `genage-1809` 195b, `genclock-1810` 195b, `gencount-1811` 195b, and `gensum-1812` 195b.

⟨*global* `genxxx` 195b⟩≡ (703a)
```
static  ulong   genage, genclock, gencount;
static  uvlong  gensum;
```

⟨`Page` *other fields* 195c⟩+≡ (89) ◁171a 521a▷
```
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` 195d⟩≡ (703a)
```
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 46b, iolist-1807 189b, ioptr-1808 189c, lock() 58d, nexterror, pageiocomp() 196a, poperror 633d, putpage() 92d, swapimage 185d, unlock() 60d, and waserror 633d.

⟨*function* pageiocomp 196a⟩≡ (703a)

```
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 196b⟩≡ (694)

```
#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* 196c⟩≡ (366b) 242d▷

```
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 631d, Egreg 631d, Qswap-1790 600a, fdtochan() 280c, iseve() 557b, kickpager() 191d, setswapchan() 197, 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 197⟩≡ (703a)

```
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 46b, swapalloc 185b, and swapimage 185d.

Chapter 9

Scheduling

9.1 Overview

The scheduler decides which process runs next. Its central function is `sched()` (and its twin `schedinit()`). The chapter has two halves: cooperative scheduling, where kernel code voluntarily calls `sched()` or `sleep()`/`wakeup()` to yield the processor; and preemptive scheduling, where the clock interrupt forces a context switch after a time quantum (100 ms by default). Both halves ultimately funnel through the same mechanism: save the current process's context, pick the next one from the run queue, and restore its context. Processes are organized into priority queues. The scheduler always picks the highest-priority ready process, but priorities are dynamically adjusted to prevent starvation.

9.2 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` 198a⟩≡ (705c)
```
Npriq   = 20,   /* number of scheduler priority levels */
```

⟨*enum* `priority` 198b⟩≡ (705c)
```
enum Priority
{
    PriNormal = 10,   /* base priority for normal processes */
    PriKproc  = 13,   /* base priority for kernel processes */
    ⟨Priority cases 199g⟩
    ⟨Priority cases for real-time priority 531a⟩
};
```

⟨`Proc` *scheduling fields* 198c⟩≡ (49f) 199b▷
```
// enum<Priority>
ulong priority; /* priority level */
```

⟨`newproc()` *set fields of* `p` 198d⟩+≡ (111d) ◁193c 203c▷
```
/* sched params */
procpriority(p, PriNormal, false);
```
Uses `PriNormal` 198b, `false`, and `procpriority()` 199a.

⟨*function* `procpriority` 199a⟩≡ (712c)
```
void
procpriority(Proc *p, int pri, bool fixed)
{
    ⟨procpriority() sanity check pri 199d⟩
    p->priority = pri;

    p->basepri = pri;
    p->fixedpri = fixed;
}
```

⟨`Proc` *scheduling fields* 199b⟩+≡ (49f) ◁198c 203b▷
```
// enum<Priority>
ulong basepri;  /* base priority level */
bool fixedpri; /* priority level doesn't change */
```

⟨`sysrfork()` *when* `RFPROC` *propagate fields from up to p* 199c⟩+≡ (116b) ◁193d 205c▷
```
p->priority = up->basepri;
p->basepri = up->basepri;
p->fixedpri = up->fixedpri;
```
Uses `up`.

⟨`procpriority()` *sanity check pri* 199d⟩≡ (199a)
```
if(pri >= Npriq)
    pri = Npriq - 1;
else if(pri < 0)
    pri = 0;
```
Uses `Npriq` 198a.

⟨`procctlreq()` *CMpri case* 199e⟩≡ (586b)
```
case CMpri:
    pri = atoi(cb->f[1]);
    if(pri > PriNormal && !iseve())
        error(Eperm);
    procpriority(p, pri, false);
    break;
```
Uses `CMpri-887` 568b, `PriNormal` 198b, `false`, `iseve()` 557b, and `procpriority()` 199a.

⟨`procctlreq()` *CMfixedpri case* 199f⟩≡ (586b)
```
case CMfixedpri:
    pri = atoi(cb->f[1]);
    if(pri > PriNormal && !iseve())
        error(Eperm);
    procpriority(p, pri, true);
    break;
```
Uses `CMfixedpri-888` 568b, `PriNormal` 198b, `iseve()` 557b, `procpriority()` 199a, and `true`.

⟨`Priority` *cases* 199g⟩≡ (198b)
```
PriRoot   = 13,   /* base priority for root processes */
```

⟨`sysexec()` *set prioriry of root processes* 199h⟩≡ (123b)
```
/*
 *  '/' processes are higher priority (hack to make /ip more responsive).
 */
if(devtab[tc->type]->dc == L'/')
    up->basepri = PriRoot;
up->priority = up->basepri;
```

9.3 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` 200a⟩≡ (705c)
```
Nrq   = Npriq+2,  /* number of priority levels including real time */
```
Uses `Npriq` 198a.

⟨*global* `runq` 200b⟩≡ (712c)
```
// The run queue!!
// hash<enum<priority>, queue<ref<Proc>>>
Schedq  runq[Nrq];
```
Uses `Nrq` 200a.

⟨*struct* `Schedq` 200c⟩≡ (705c)
```
// 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* 200d⟩+≡ (49f) ◁110e 234c▷
```
// list<ref<Proc>> of Schedq.head
Proc  *rnext;   /* next process in run queue */
```

⟨*global* `runveq` 200e⟩≡ (712c)
```
// array<bool>, each bit i represents whether the runq at pri i has processes
ulong   runvec; // coupling: sizeof(ulong) must be >= Nrq
```

⟨*function* `anyready` 200f⟩≡ (712c)
```
bool
anyready(void)
{
    return runvec;
}
```
Uses `runvec` 200e.

⟨*global* `nrdy` 201a⟩≡ (712c)
```
int nrdy;
```

⟨*function* `queueproc` 201b⟩≡ (712c)
```
/*
 * 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()` 58d, `nrdy` 201a, `runq` 200b, `runvec` 200e, and `unlock()` 60d.

⟨*function* `dequeueproc` 201c⟩≡ (712c)
```
/*
 *  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 202a⟩

    // 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 202b⟩
    unlock(runq);
    return p;
}
```

Uses canlock() 60f, nrdy 201a, runq 200b, runvec 200e, and unlock() 60d.

⟨dequeueproc() *return if process not found or already associated* 202a⟩≡ (201c)

```
/*
 *  p->cpu==nil only when process state is saved
 */
if(p == nil || p->cpu){
    unlock(runq);
    return nil;
}
```

Uses runq 200b and unlock() 60d.

⟨dequeueproc() *sanity check p* 202b⟩≡ (201c)

```
if(p->state != Ready)
    print("dequeueproc %s %lud %s\n", p->text, p->pid, statename[p->state]);
```

Uses Ready 210c, print, and statename 51c.

9.4 The scheduler

9.4.1 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 202c⟩≡ (712c)

```
/*
 *  If changing this routine, look also at sleep().  It
 *  contains a copy of the guts of sched().
 */
void
proc_sched(void)
{
    ⟨sched() locals 202d⟩

    ⟨sched() sanity check ilockdepth 203d⟩
    ⟨sched() if up 206c⟩
    // else
    ⟨sched() if no up 203a⟩
}
```

⟨sched() *locals* 202d⟩≡ (202c)

```
Proc *p;
```

⟨sched() *if no up* 203a⟩≡ (202c)
```
// 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 217c⟩
⟨sched() if readied process, do not adjust schedticks 215e⟩
else {
    ⟨sched() after found a new proc, adjust schedticks 215b⟩
}
⟨sched() reset readied 211e⟩

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 645, Running 51b, runproc() 204, and up.

⟨Proc *scheduling fields* 203b⟩+≡ (49f) ◁199b 205a▷
```
// option<ref<Cpu>>, null when not associated to a processor
Cpu  *cpu;    /* processor running this proc */
```

⟨newproc() *set fields of* p 203c⟩+≡ (111d) ◁198d 205b▷
```
p->cpu = nil;
```

⟨sched() *sanity check ilockdepth* 203d⟩≡ (202c)
```
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));
```

9.4.2 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* 203e⟩+≡ (48a) ◁71c 211c▷
```
Label sched;      /* scheduler wakeup */ // address of schedinit()
```

⟨*function* schedinit 203f⟩≡ (712c)
```
/*
 * Always arch_splhi()'ed.
 */
void
schedinit(void)     /* never returns */
{
    ⟨schedinit() locals 531f⟩

    arch_setlabel(&cpu->sched);
    ⟨schedinit() after setlabel, if up 207a⟩
    // ok at this point up is nil
    sched();
}
```
Uses sched.

9.4.3 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 204⟩≡ (712c)

```
/*
 *  pick a process to run
 */
Proc*
runproc(void)
{
    Proc *p;
    Schedq *rq;
    int i;
    ⟨runproc() other locals 467f⟩

    ⟨runproc() start 623d⟩
    ⟨runproc() cooperative scheduling if process readied 211d⟩
    //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 623e⟩
    }

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 531c⟩
    ⟨runproc() hook proctrace 467g⟩
    return p;
}
```

Uses CPUS 645, Nrq 200a, Scheding 111c, arch_idlehands(), dequeueproc() 201c, preempts 629b, and runq 200b.

```
⟨Proc scheduling fields 205a⟩+≡                                    (49f) ◁203b 208a▷
  Cpu *lastcpu;    /* processor this process last ran on */
```

```
⟨newproc() set fields of p 205b⟩+≡                                 (111d) ◁203c 205f▷
  p->lastcpu = nil;
```

```
⟨sysrfork() when RFPROC propagate fields from up to p 205c⟩+≡      (116b) ◁199c 419f▷
  p->lastcpu = up->lastcpu;
```

Uses up.

9.5 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.

9.5.1 Label

```
⟨struct Label 205d⟩≡                                               (645)
  // =~ 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 205e⟩≡                                       (49f)
  Label sched;    /* known to l.s */
  char  *kstack;  /* known to l.s */
```

9.5.2 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 205f⟩+≡                                 (111d) ◁205b 208b▷
  if(p->kstack == nil)
      p->kstack = smalloc(KSTACK);
```

Uses smalloc() 85c.

```
⟨constant KSTACK(arm) 205g⟩≡                                       (747)
  #define KSTACK      (8*KiB)
```

9.5.3 `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)* 206a⟩≡ (757a)
```
TEXT arch_gotolabel(SB), 1, $-4
    MOVW    0(R0), R13      /* sp */
    MOVW    4(R0), R14      /* pc */
    MOVW    $1, R0
    RET
```

⟨*function* `arch_setlabel` *(arm)* 206b⟩≡ (757a)
```
TEXT arch_setlabel(SB), 1, $-4
    MOVW    R13, 0(R0)      /* sp */
    MOVW    R14, 4(R0)      /* pc */
    MOVW    $0, R0
    RET
```

9.5.4 `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* 206c⟩≡ (202c)
```
if(up){
    ⟨sched() when up, possibly increment delaysched and return 207c⟩
    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* 207a⟩≡ (203f)

```
    if(up) {
        ⟨schedinit() optional real-time edfrecord() 531g⟩
        //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 136a⟩
        }
        up->cpu = nil;
        ⟨schedinit() updatecpu before reseting up 218b⟩
        cpu->proc = nil;
        up = nil; // same instruction than previous line on some archi (e.g. PC)
    }
```

Uses up.

⟨schedinit() *switch process state cases* 207b⟩+≡ (207a) ◁136a

```
    case Running:
        ready(up);
        break;
```

Uses Running 51b, ready, and up.

9.5.5 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* 207c⟩≡ (206c)

```
    /*
     * 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 135a, delayedscheds 629b, palloc 90d, procalloc-2458 110c, and up.

⟨Proc *scheduling fields* 208a⟩+≡ (49f) ◁205a 216d▷

```
  ulong delaysched;
```

⟨newproc() *set fields of* p 208b⟩+≡ (111d) ◁205f 216e▷

```
  p->nlocks.ref = 0;
  p->delaysched = 0;
```

⟨unlock() *if delaysched* 208c⟩≡ (60d)

```
  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() 71a, sched, and up.

⟨trap() *if delaysched and clockintr* 208d⟩≡ (150b)

```
  /* 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)* 208e⟩≡ (145e)

```
  /* if we delayed sched because we held a lock, sched now */
  if(up->delaysched){
      sched();
      arch_splhi();
  }
```

Uses sched and up.

9.5.6 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)* 209a⟩+≡ (48e) ◁169b
```
int mmupid;
```

⟨*function* arch_mmuswitch*(arm)* 209b⟩≡ (748b)
```
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) 498a⟩

    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) 498c⟩

    /* lose any possible stale tlb entries */
    mmuinvalidate();
}
```
Uses `Coarse` 160d, `Dom0` 161d, `false`, `mmul1empty()` 163e, and `mmul2empty()` 163a.

9.5.7 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)* 209c⟩≡ (755)
```
/*
```

```
 *  Save the mach dependent part of the process state.
 */
void
arch_procsave(Proc* p)
{
    ⟨arch_procsave() cycles adjustments 476h⟩
    ⟨arch_procsave() floating point(arm) 482e⟩
}
```

⟨*function* `arch_procrestore` *(arm)* 210a⟩≡ (755)

```
void
arch_procrestore(Proc* p)
{
    uvlong t;

    if(p->kp)
        return;
    ⟨arch_procrestore cycles adjustments 476i⟩
    ⟨arch_procrestore() floating point(arm) 483b⟩
}
```

9.6 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.

9.6.1 `yield()`

⟨*function* `yield` 210b⟩≡ (712c)

```
/*
 *  yield the processor and drop our priority
 */
void
yield(void)
{
    if(anyready()){
        ⟨yield() adjust lastupdate before scheduling another process 218g⟩
        sched();
    }
}
```

Uses `anyready()` 200f and `sched`.

9.6.2 `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* 210c⟩+≡ (51b) ◁135a 212d▷

```
Ready,
```

⟨*function* `ready` 211a⟩≡ (712c)

```
/*
 *  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 467d⟩

    s = arch_splhi();
    ⟨ready() optional edfready() for real-time scheduling 531i⟩

    ⟨ready() possibly set cpu->readied 211b⟩
    ⟨ready() update priority 218a⟩

    rq = &runq[pri];
    p->state = Ready;
    queueproc(rq, p);
    ⟨ready() hook proctrace 467e⟩
    arch_splx(s);
}
```

Uses `Ready` 210c, `queueproc()` 201b, and `runq` 200b.

⟨`ready()` *possibly set* `cpu->readied` 211b⟩≡ (211a)

```
if(up != p && (p->wired == nil || p->wired == CPUS(cpu->cpuno))) // pad fix
    cpu->readied = p; /* group scheduling */
```

Uses `CPUS` 645 and `up`.

⟨`Cpu` *other fields* 211c⟩+≡ (48a) ◁203e 215a▷

```
Proc* readied;    /* for runproc */
```

⟨`runproc()` *cooperative scheduling if process readied* 211d⟩≡ (204)

```
/* 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 531b⟩
){
    skipscheds++;
    rq = &runq[p->priority];
    goto found;
}
```

Uses `CPUS` 645, `Ready` 210c, `runq` 200b, and `skipscheds` 629b.

⟨`sched()` *reset readied* 211e⟩≡ (203a)

```
cpu->readied = nil;
```

9.6.3 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 212a⟩≡ (640)
```
struct Rendez
{
    // option<ref<Proc>>
    Proc  *p; // sleeping process

    Lock; // LOCK ORDERING: always do lock(r); lock(p->rlock);
};
```

⟨Proc *synchronization fields* 212b⟩+≡ (49f) ◁70f 212c▷
```
// option<ref<Rendez>>, can point to waitr, freememr, sleepr, etc
Rendez  *r;   /* rendezvous point slept on */
```

⟨Proc *synchronization fields* 212c⟩+≡ (49f) ◁212b 232b▷
```
Lock  rlock;    /* sync sleep/wakeup with postnote */
```

9.6.4 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* 212d⟩+≡ (51b) ◁210c 448e▷
```
Wakeme,
```

⟨*function* sleep 212e⟩≡ (712c)
```
/*
 *  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 468a⟩

    s = arch_splhi();
    ⟨sleep() sanity check nlocks 213a⟩
    lock(r);
    lock(&up->rlock);
    ⟨sleep() sanity check double sleep on this rendez vous 213b⟩

    /*
     *  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 468b⟩
        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 419g⟩
    arch_splx(s);
}
```

Uses Wakeme 212d, lock() 58d, unlock() 60d, and up.

⟨sleep() *sanity check nlocks* 213a⟩≡ (212e)

```
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* 213b⟩≡ (212e)

```
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.

9.6.5 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` 214a⟩≡ (712c)

```
/*
 *  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 214b⟩
        r->p = nil;
        p->r = nil;
        ready(p);
        unlock(&p->rlock);
    }
    unlock(r);
    arch_splx(s);
    return p;
}
```

Uses `lock()` 58d, `ready`, and `unlock()` 60d.

⟨`wakeup()` *sanity check process state* 214b⟩≡ (214a)

```
if(p->state != Wakeme || p->r != r){
    iprint("%p %p %d\n", p->r, r, p->state);
    panic("wakeup: state");
}
```

Uses `Wakeme` 212d and `iprint`.

9.7 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)* 214c⟩≡ (752a)

```
#define Arch_HZ     100         /* clock frequency */
```

9.7.1 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* 215a⟩+≡ (48a) ◁211c 441b▷
```
ulong schedticks;   /* next forced context switch */
```

⟨sched() *after found a new proc, adjust schedticks* 215b⟩≡ (203a)
```
cpu->schedticks = cpu->ticks + Arch_HZ/10; // 100ms of allocated time
```

⟨*function* hzsched 215c⟩≡ (712c)
```
/*
 *  here once per clock tick to see if we should resched
 */
void
hzsched(void)
{
    ⟨hzsched() rebalance ready procs 220a⟩
    /* 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() 215d, anyready() 200f, and up.

⟨*function* anyhigher 215d⟩≡ (712c)
```
int
anyhigher(void)
{
    return runvec & ~((1<<(up->priority+1))-1);
}
```
Uses runvec 200e and up.

⟨sched() *if readied process, do not adjust schedticks* 215e⟩≡ (203a)
```
if(p == cpu->readied) {
}
```

9.7.2 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* 215f⟩≡ (709c)
```
void
hzclock(Ureg *ur)
{
    cpu->ticks++;
    ⟨hzclock() adjust pc of proc 216a⟩
    ⟨hzclock() if flushmmu 441a⟩
    ⟨hzclock() accounttime 624a⟩
    ⟨hzclock() if kproftimer 627a⟩
```

```
    ⟨hzclock() if not active cpu, return 216b⟩
    // else
    ⟨hzclock() if exiting, exit 216c⟩
    // else

    ⟨hzclock() check the alarms 237a⟩
    if(up && up->state == Running)
        hzsched();  /* in proc.c */
}
```

Uses Running 51b, hzsched() 215c, and up.

⟨hzclock() *adjust pc of proc* 216a⟩≡ (215f)
```
if(cpu->proc) // why not using up here? why cpu->proc?
    cpu->proc->pc = ur->pc;
```

⟨hzclock() *if not active cpu, return* 216b⟩≡ (215f)
```
if((active.cpus & (1<<cpu->cpuno)) == 0)
    return;
```

Uses active 49e.

⟨hzclock() *if exiting, exit* 216c⟩≡ (215f)
```
if(active.exiting) {
    print("someone's exiting\n");
    arch_exit(0);
}
```

Uses active 49e, arch_exit, and print.

9.8 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* 216d⟩+≡ (49f) ◁208a 218c▷
```
Cpu  *wired;
```

⟨newproc() *set fields of* p 216e⟩+≡ (111d) ◁208b 218d▷
```
p->wired = nil;
```

⟨procctlreq() *CMwired case* 216f⟩≡ (586b)
```
case CMwired:
    procwired(p, atoi(cb->f[1]));
    break;
```

Uses CMwired-889 568b and procwired() 216g.

⟨*function* procwired 216g⟩≡ (712c)
```
/*
 * 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 645, conf 46b, and proctab.

⟨sysrfork() *other locals* 217a⟩+≡ (115c) ◁121a

```
Cpu *wm;
```

⟨sysrfork() *if parent is a wired proc* 217b⟩≡ (116b)

```
wm = up->wired;
if(wm)
    procwired(p, wm->cpuno);
```

Uses procwired() 216g and up.

9.9 Real-time scheduling

9.10 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.

9.10.1 reprioritize()

⟨sched() *after found process* p *to schedule, possibly update priority* 217c⟩≡ (203a)

```
⟨sched() if real-time process then do nothing, do not reprioritize 533a⟩
else {
    updatecpu(p);
    p->priority = reprioritize(p);
}
```

Uses reprioritize() 219 and updatecpu() 218h.

⟨ready() *update priority* 218a⟩≡ (211a)
```
updatecpu(p);
pri = reprioritize(p);
p->priority = pri;
```
Uses reprioritize() 219 and updatecpu() 218h.

⟨schedinit() *updatecpu before reseting up* 218b⟩≡ (207a)
```
updatecpu(up);
```
Uses up and updatecpu() 218h.

⟨Proc *scheduling fields* 218c⟩+≡ (49f) ◁216d 533c▷
```
ulong lastupdate; // dimension?? ticks * Scaling;
ulong cpuavg;    /* cpu average */
```

⟨newproc() *set fields of* p 218d⟩+≡ (111d) ◁216e 419d▷
```
p->cpuavg = 0;
p->lastupdate = CPUS(0)->ticks * Scaling;
```
Uses CPUS 645 and Scaling-2461 218e.

⟨*constant* Scaling 218e⟩≡ (712c)
```
Scaling=2,
```

⟨*constant* Schedagain 218f⟩≡ (712c)
```
schedgain = 30, /* units in seconds */
```

⟨yield() *adjust lastupdate before scheduling another process* 218g⟩≡ (210b)
```
/* pretend we just used 1/2 tick */
up->lastupdate -= Scaling/2;
```
Uses Scaling-2461 218e and up.

⟨*function* updatecpu 218h⟩≡ (712c)
```
/*
 * 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 531e⟩

    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 645, Scaling-2461 218e, schedgain-2460 218f, and up.

⟨*function* reprioritize 219⟩≡ (712c)

```
/*
 * 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 645 and conf 46b.

9.10.2 rebalance()

⟨hzsched() *rebalance ready procs* 220a⟩≡ (215c)

```
/* once a second, rebalance will reprioritize ready procs */
if(cpu->cpuno == 0)
    rebalance();
```

Uses rebalance() 220b.

⟨*function* rebalance 220b⟩≡ (712c)

```
static void
rebalance(void)
{
    int pri, npri, t, x;
    Schedq *rq;
    Proc *p;

    ⟨rebalance() return if less than one second since last rebalance 221b⟩

    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 645, Npriq 198a, dequeueproc() 201c, queueproc() 201b, reprioritize() 219, runq 200b, and updatecpu() 218h.

⟨*global* `balancetime` 221a⟩≡ (712c)

```
/*
 *  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* 221b⟩≡ (220b)

```
t = cpu->ticks;
if(t - balancetime < Arch_HZ)
    return;
balancetime = t;
```

Uses `balancetime` 221a.

9.11 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)* 221c⟩≡ (741a)

```
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* 222a⟩≡ (41f)
```
  [SLEEP]     syssleep,
  [ALARM]     sysalarm,
```
Uses `sysalarm()` 234e and `syssleep()` 232c.

10.1 Overview (arm)

This chapter covers the kernel's time infrastructure: how hardware timers generate interrupts, how the kernel multiplexes a single hardware timer into many software timers, and the time-related syscalls (`sleep` and `alarm`). We saw `hzclock()` in the Scheduling chapter; here we see how it gets called 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.

10.2 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:

10.2.1 Seconds

⟨*type* `Txxx` 222b⟩≡ (709a) 222d▷
```
  typedef vlong   Tsec; // seconds
  typedef int     Tms; // milliseconds
  typedef vlong   Tmicro; // microseconds
  typedef vlong   Tnano; // nanoseconds
```

⟨*macro* `MS2NS` 222c⟩≡ (708b)
```
  #define MS2NS(n) (((vlong)(n))*1000000LL)
```

10.2.2 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` 222d⟩+≡ (709a) ◁222b 223f▷
```
  typedef vlong   Tval; // ticks
```

⟨*macro* TK2MS 223a⟩≡ (708b)
```
#define TK2MS(x) ((x)*(1000/Arch_HZ))
```

⟨*function* tk2ms 223b⟩≡ (709c)
```
/*
 *  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 223c⟩≡ (709c)
```
ulong
ms2tk(ulong ms)
{
    ⟨ms2tk() overflow approximation 223d⟩
    return (ms * Arch_HZ + 500) / 1000;
}
```

⟨ms2tk() *overflow approximation* 223d⟩≡ (223c)
```
/* avoid overflows at the cost of precision */
if(ms >= 1000000000 / Arch_HZ)
    return (ms / 1000) * Arch_HZ;
```

10.2.3 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)* 223e⟩≡ (752b)
```
SystimerFreq    = 1*Mhz,
```
Uses Mhz 709a.

⟨*type* Txxx 223f⟩+≡ (709a) ◁222d
```
typedef vlong   Tfast; // fastticks
typedef uvlong   Tufast; // fastticks
```

⟨*function* arch_us*(arm)* 223g⟩≡ (753)
```
ulong // Tmicro
arch_us(void)
{
    ⟨arch_us() if non-standard systimer frequency 241b⟩
    return arch_fastticks(nil);
}
```
Uses arch_fastticks.

⟨*function* `clock_arch_fastticks`*(arm)* 224a⟩≡ (753)

```
Tufast
clock_arch_fastticks(uvlong *hz)
{
    Systimers *tn;
    ulong lo, hi;
    Tufast now;
    int s;

    ⟨arch_fastticks() if hz pointer given 224b⟩

    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` 31e.

⟨`arch_fastticks()` *if hz pointer given* 224b⟩≡ (224a)

```
if(hz)
    *hz = SystimerFreq;
```

Uses `SystimerFreq-2652` 223e.

10.3 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)* 224c⟩≡ (753)

```
// 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)* 224d⟩≡ (753)

```
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 31e and timerintr() 229d.

10.4 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.

10.4.1 `Timer`

⟨*struct* Timer 225a⟩≡ (709a)

```
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 226a⟩
    };
```

⟨*enum* timermode 225b⟩≡ (709a)

```
/*
 * fasttick timer interrupts
 */
enum Timermode
{
    Trelative,  /* timer programmed in ns from now */
    Tperiodic,  /* periodic timer, period in ns */
};
```

10.4.2 `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 225c⟩≡ (709a)

```
struct Timers
{
    // list<Timer> (next = Timer.tnext)
    Timer *head;
    // extra
    ILock;
};
```

⟨Timer *extra fields* 226a⟩≡ (225a) 226b▷
```
// list<Timer> of Timers.head
Timer *tnext;
```

⟨Timer *extra fields* 226b⟩+≡ (225a) ◁226a
```
// option<ref<list<Timer>> Timers.head (of one of global timers[MAXCPUS])
Timers  *tt;    /* Timers queue this timer runs on */
```

⟨*global* timers 226c⟩≡ (709c)
```
static Timers timers[MAXCPUS];
```

10.4.3 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 226d⟩≡ (709c)
```
static Tfast
tadd(Timers *tt, Timer *nt)
{
    Timer *t, **last;

    /* Called with tt locked */
    ⟨tadd() sanity check nt 226e⟩
    ⟨tadd() set nt->twhen 226f⟩

    // 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* 226e⟩≡ (226d)
```
assert(nt->tt == nil);
```
Uses assert 693a.

⟨tadd() *set* nt->twhen 226f⟩≡ (226d)
```
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 227a⟩
        else
            nt->twhen = arch_fastticks(nil);
    }
    nt->twhen += ns2fastticks(nt->tns);
```

```
        break;
    default:
        panic("timer: impossible");
        break;
    }
```

Uses Tperiodic 225b, Trelative 225b, arch_fastticks, assert 693a, and ns2fastticks() 241d.

⟨tadd() *look if can combine with identical timer* 227a⟩≡ (226f)

```
    /* 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 225b.

⟨*function* tdel 227b⟩≡ (709c)

```
    static Tfast
    tdel(Timer *dt)
    {
        Timer *t, **last;
        Timers *tt;

        tt = dt->tt;
        ⟨tdel() sanity check tt 227c⟩
        // 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 693a.

⟨tdel() *sanity check tt* 227c⟩≡ (227b)

```
    if (tt == nil)
        return 0; //todo: panic instead
```

10.4.4 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 227d⟩≡ (709c)

```
    /* 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 228a⟩
    tt = &timers[cpu->cpuno];
    ilock(tt);
    when = tadd(tt, nt);
    if(when)
        arch_timerset(when);
    iunlock(tt);
    iunlock(nt);
}
```

Uses ilock() 61d, iunlock() 62b, tadd() 226d, and timers-2645 226c.

⟨timeradd() *if timer was already in a list* 228a⟩≡ (227d)

```
if(tt = nt->tt){
    ilock(tt);
    tdel(nt);
    iunlock(tt);
}
```

Uses ilock() 61d, iunlock() 62b, and tdel() 227b.

⟨pexit() *at the start, free resources* 228b⟩≡ (134c) 234g▷

```
if (up->tt)
    timerdel(up);
```

Uses timerdel() 228c and up.

⟨*function* timerdel 228c⟩≡ (709c)

```
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() 61d, iunlock() 62b, tdel() 227b, and timers-2645 226c.

10.4.5 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)* 228d⟩≡ (753)

```
void
arch_timerset(Tfast next)
{
    Systimers *tn;
```

```
        Tufast now;
        long period;

        now = arch_fastticks(nil);
        period = next - now;
        ⟨arch_timerset() sanitize period(arm) 229a⟩
        ⟨arch_timerset() if not cpu0(arm) 514g⟩
        else{
            tn = (Systimers*)SYSTIMERS;
            tn->c3 = (ulong)(now + period);
        }
    }
```

Uses SYSTIMERS-2647 31e and arch_fastticks.

⟨arch_timerset() *sanitize period(arm)* 229a⟩≡ (228d)

```
    if(period < MinPeriod)
        period = MinPeriod;
    else if(period > MaxPeriod)
        period = MaxPeriod;
```

Uses MaxPeriod-2653 229b and MinPeriod-2654 229c.

⟨*constant* MaxPeriod*(arm)* 229b⟩≡ (752b)

```
    MaxPeriod   = SystimerFreq / Arch_HZ,
```

Uses SystimerFreq-2652 223e.

⟨*constant* MinPeriod*(arm)* 229c⟩≡ (752b)

```
    MinPeriod   = SystimerFreq / (100*Arch_HZ),
```

Uses SystimerFreq-2652 223e.

10.4.6 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* 229d⟩≡ (709c)

```
    void
    timerintr(Ureg *u, Tval)
    {
        Timer *t;
        Timers *tt;
        Tufast when, now;
        bool callhzclock = false;
        ⟨timerintr() other locals 230b⟩

        ⟨timerintr() start, stat 629d⟩
        tt = &timers[cpu->cpuno];
        now = arch_fastticks(nil);
        ⟨timerintr() sanity check now 230a⟩
        ilock(tt);
        ⟨timerintr() before loop, initialize count 230c⟩
        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 629e⟩
            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 230e⟩
        }
        iunlock(tt); // unreachable?
    }
```

Uses Tperiodic 225b, arch_fastticks, assert 693a, false, hzclock() 215f, ilock() 61d, iunlock() 62b, tadd() 226d, timers-2645 226c, and true.

⟨timerintr() *sanity check now* 230a⟩≡ (229d)

```
    if(now == 0)
        panic("timerintr: zero arch_fastticks()");
```

⟨timerintr() *other locals* 230b⟩≡ (229d)

```
    int count;
```

⟨timerintr() *before loop, initialize count* 230c⟩≡ (229d)

```
    count = Maxtimerloops;
```

Uses Maxtimerloops-2644 230d.

⟨*constant* Maxtimerloops 230d⟩≡ (709c)

```
    Maxtimerloops = 20*1000,
```

⟨timerintr() *end of timers loop, sanity check count* 230e⟩≡ (229d)

```
    if (--count <= 0) {
        count = Maxtimerloops;
        iprint("timerintr: probably stuck in while loop; "
            "scrutinise clock.c or use faster cycle "
            "counter\n");
    }
```

Uses Maxtimerloops-2644 230d and iprint.

10.4.7 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` 231a⟩≡ (709c)
```
Timer*
addclock0link(void (*f)(void), Tms ms)
{
    Timer *nt;
    Tfast when;

    ⟨addclock0link() sanity check timersinit 231b⟩

    nt = malloc(sizeof(Timer));
    ⟨addclock0link() sanity check nt 231c⟩
    ⟨addclock0link() sanitize ms 231d⟩

    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` 225b, `ilock()` 61d, `iunlock()` 62b, `malloc()` 85a, `tadd()` 226d, and `timers-2645` 226c.

⟨`addclock0link()` *sanity check timersinit* 231b⟩≡ (231a)
```
if(!timersinited)
    panic("addclock0link: timersinit not called yet");
```
Uses `timersinited-2646` 259a.

⟨`addclock0link()` *sanity check nt* 231c⟩≡ (231a)
```
if(nt == nil)
    error(Enomem);
```

⟨`addclock0link()` *sanitize ms* 231d⟩≡ (231a)
```
/* Synchronize to hztimer if ms is 0 */
if(ms == 0)
    ms = 1000 / Arch_HZ; // tk2ms(1)
```

10.4.8 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)* 231e⟩≡ (753)
```
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 31e.

⟨*function* clock_arch_delay*(arm)* 232a⟩≡ (753)

```
void
clock_arch_delay(Tms n)
{
    while(--n >= 0)
        arch_microdelay(1000);
}
```

Uses arch_microdelay.

10.5 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* 232b⟩+≡ (49f) ◁212c 448b▷

```
Rendez  sleepr;    /* place for syssleep/debug/tsleep */
```

⟨*syscall sleep* 232c⟩≡ (716a)

```
// int sleep(long millisecs);
long
syssleep(ulong* arg)
{
    int n;
    n = arg[0];
    if(n <= 0) {
        ⟨syssleep() optional edfyield() for real-time scheduling 532a⟩
        yield();
        return 0;
    }
    ⟨syssleep() sanitize n 232d⟩
    tsleep(&up->sleepr, returnfalse, 0, n);
    return 0;
}
```

Uses returnfalse() 86a, tsleep, up, and yield() 210b.

⟨syssleep() *sanitize n* 232d⟩≡ (232c)

```
if(n < TK2MS(1))
    n = TK2MS(1);
```

Uses TK2MS 223a.

⟨*function* `tsleep` 233a⟩≡ (712c)

```
void
proc_tsleep(Rendez *r, int (*fn)(void*), void *arg, ulong ms)
{
    ⟨tsleep() sanity check up timer 233c⟩
    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 233b⟩
    poperror();
}
```

Uses `MS2NS` 222c, `Trelative` 225b, `nexterror`, `poperror` 633d, `sleep`, `tfn()` 233f, `timeradd()` 227d, `timerdel()` 228c, `twakeup()` 234a, `up`, and `waserror` 633d.

⟨`tsleep()` *reset timer after sleep* 233b⟩≡ (233a)

```
if(up->tt)
    timerdel(up);
up->twhen = 0;
```

Uses `timerdel()` 228c and `up`.

⟨`tsleep()` *sanity check up timer* 233c⟩≡ (233a)

```
if (up->tt){
    print("tsleep: timer active: mode %d, tf %#p\n", up->tmode, up->tf);
    timerdel(up);
}
```

Uses `print`, `timerdel()` 228c, and `up`.

⟨`Proc` *other fields* 233d⟩+≡ (49f) ◁145c 233e▷

```
Timer;      /* For tsleep and real-time */
```

⟨`Proc` *other fields* 233e⟩+≡ (49f) ◁233d 234f▷

```
Rendez  *trend;
int (*tfn)(void*);
```

⟨*function* `tfn` 233f⟩≡ (712c)

```
static int
tfn(void *arg)
{
    return up->trend == nil || up->tfn(arg);
}
```

Uses `up`.

⟨*function* `twakeup` 234a⟩≡ (712c)
```
void
twakeup(Ureg*, Timer *t)
{
    Proc *p;
    Rendez *trend;

    p = t->ta;
    trend = p->trend;
    p->trend = nil;
    if(trend)
        wakeup(trend);
}
```
Uses `wakeup`.

10.6 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` 234b⟩≡ (705c)
```
struct Alarms
{
    // sorted_list<ref<Proc> (next = Proc.palarm, sort_key = Proc.alarm)
    Proc  *head;

    // Extra
    QLock;
};
```

⟨`Proc` *extra fields* 234c⟩+≡ (49f) ◁200d
```
// Alarms.head chain?
Proc  *palarm;  /* Next alarm time */
```

⟨*global* `alarms` 234d⟩≡ (709b)
```
static Alarms   alarms;
```

10.7 `sysalarm()`

⟨*syscall alarm* 234e⟩≡ (716a)
```
// long alarm(unsigned long millisecs);
long
sysalarm(ulong* arg)
{
    return procalarm(arg[0]);
}
```
Uses `procalarm()` 235a.

⟨`Proc` *other fields* 234f⟩+≡ (49f) ◁233e 285c▷
```
// option<ticks>
ulong alarm;    /* Time of call */ // Tval ticks
```

⟨`pexit()` *at the start, free resources* 234g⟩+≡ (134c) ◁228b 460c▷
```
up->alarm = 0;
```
Uses `up`.

⟨*function* procalarm 235a⟩≡ (709b)

```
ulong
procalarm(ulong time) // Tms
{
    Proc **l, *f;
    ulong when, old; // Tval ticks

    ⟨procalarm() compute remaining time in old alarm 235b⟩
    ⟨procalarm() if time is zero, reset alarm 235c⟩
    when = ms2tk(time) + CPUS(0)->ticks;
    ⟨procalarm() sanitize when 236a⟩

    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 645, alarms-2607 234d, ms2tk() 223c, qlock() 63f, qunlock() 64c, and up.

⟨procalarm() *compute remaining time in old alarm* 235b⟩≡ (235a)

```
if(up->alarm)
    old = tk2ms(up->alarm - CPUS(0)->ticks);
else
    old = 0;
```

Uses CPUS 645, tk2ms() 223b, and up.

⟨procalarm() *if time is zero, reset alarm* 235c⟩≡ (235a)

```
if(time == 0) {
    up->alarm = 0;
    return old;
}
```

Uses up.

⟨procalarm() *sanitize when* 236a⟩≡ (235a)
```
if(when == 0)       /* ticks have wrapped to 0? */
    when = 1;   /* distinguish a wrapped alarm from no alarm */
```

10.7.1 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 236b⟩≡ (709b)
```
static Rendez alarmr;
```

⟨*kernel process alarmkproc* 236c⟩≡ (709b)
```
// 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 645, NUser 418c, alarmr-2608 236b, alarms-2607 234d, canqlock() 65a, poperror 633d, postnote, qlock() 63f, qunlock() 64c, returnfalse() 86a, sleep, and waserror 633d.

10.7.2 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* 237a⟩≡ (215f)
```
checkalarms();
```
Uses checkalarms() 237b.

⟨*function* checkalarms 237b⟩≡ (709b)
```
/*
 *  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 645, alarmr-2608 236b, alarms-2607 234d, and wakeup.

10.8 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 237c⟩≡ (715)
```
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 237d⟩≡ (715)
```
struct TOD tod;
```
Uses TOD 237c.

⟨*devcons.c Exxx errors* 238a⟩≡ (643)
```
char *Ebadtimectl = "bad time control";
```
Uses Ebadtimectl 238a.

⟨*constant* VLNUMSIZE 238b⟩≡ (643)
```
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 238c⟩≡ (715)
```
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() 231a, arch_fastticks, ilock() 61d, iunlock() 62b, tod 237d, todfix() 240, todsetfreq() 238d, 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 238d⟩≡ (715)
```
/*
 *  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 715, TODFREQ-2669 715, ilock() 61d, iunlock() 62b, mk64fract() 242a, and tod 237d.

`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` 239a⟩≡ (715)

```
/*
 *  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` 645, `arch_fastticks`, `ilock()` 61d, `iprint`, `iunlock()` 62b, `tod` 237d, and `todinit()` 238c.

`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` 239b⟩≡ (715)

```
/*
 *  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 645, arch_fastticks, ilock() 61d, iunlock() 62b, tod 237d, and todinit() 238c.

`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 240⟩≡ (715)

```
/*
 *  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 > 300000000000ULL) iprint("todfix %llud\n", x);
        x += tod.off;

        /* protect against overflows */
        tod.last = ticks;
        tod.off = x;

        iunlock(&tod);
    }
  }
```

Uses arch_fastticks, ilock() 61d, iprint, iunlock() 62b, and tod 237d.

⟨*function* seconds 241a⟩≡ (715)

```
  long
  seconds(void)
  {
      return (vlong)todget(nil) / TODFREQ;
  }
```

Uses TODFREQ-2669 715 and todget() 239b.

⟨arch_us() *if non-standard systimer frequency* 241b⟩≡ (223g)

```
  if(SystimerFreq != 1*Mhz)
      return fastticks2us(arch_fastticks(nil));
```

Uses Mhz 709a, SystimerFreq-2652 223e, arch_fastticks, and fastticks2us() 241c.

⟨*function* fastticks2us 241c⟩≡ (715)

```
  uvlong
  fastticks2us(uvlong ticks)
  {
      uvlong res;

      if(!tod.init)
          todinit();
      arch_mul64fract(&res, ticks, tod.umultiplier);
      return res;
  }
```

Uses tod 237d and todinit() 238c.

⟨*function* ns2fastticks 241d⟩≡ (715)

```
  /*
   *  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 237d and todinit() 238c.

`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` 242a⟩≡ (715)
```
/*
 * 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;
}
```

10.9 sysnsec()

10.10 /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* 242b⟩+≡ (366a) ◁186a 243b▷
```
case Qtime:
    return readtime((ulong)offset, buf, n);
```
Uses `Qtime-1791` 600a and `readtime()` 242e.

⟨*global* `fasthz` 242c⟩≡ (643)
```
vlong   fasthz;
```

⟨`conswrite()` *cases* 242d⟩+≡ (366b) ◁196c 243c▷
```
case Qtime:
    if(!iseve())
        error(Eperm);
    return writetime(a, n);
```
Uses `Qtime-1791` 600a, `iseve()` 557b, and `writetime()` 243a.

⟨*function* `readtime` 242e⟩≡ (643)
```
/*
 *  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 704a, VLNUMSIZE-1796 238b, arch_fastticks, fasthz 242c, readstr() 637d, and todget() 239b.

⟨*function* writetime 243a⟩≡ (643)

```
/*
 *  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 238a and todset() 239a.

10.11 /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* 243b⟩+≡ (366a) ◁242b 472b▷

```
case Qbintime:
    return readbintime(buf, n);
```

Uses Qbintime-1783 600a and readbintime() 244a.

⟨conswrite() *cases* 243c⟩+≡ (366b) ◁242d 368f▷

```
case Qbintime:
    if(!iseve())
        error(Eperm);
    return writebintime(a, n);
```

Uses Qbintime-1783 600a, iseve() 557b, and writebintime() 244b.

⟨*function* `readbintime` 244a⟩≡ (643)

```
/*
 *  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` 242c, `todget()` 239b, and `vlong2le()` 245c.

⟨*function* `writebintime` 244b⟩≡ (643)

```
/*
 *  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 238a, fasthz 242c, le2long() 245e, le2vlong() 245b, todset() 239a, and todsetfreq() 238d.

⟨*global* uvorder 245a⟩≡ (643)

```
    static uvlong uvorder = 0x0001020304050607ULL;
```

Uses uvorder-1799 245a.

⟨*function* le2vlong 245b⟩≡ (643)

```
    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 245a.

⟨*function* vlong2le 245c⟩≡ (643)

```
    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 245a.

⟨*global* order 245d⟩≡ (643)

```
    static long order = 0x00010203;
```

Uses order-1800 245d.

⟨*function* le2long 245e⟩≡ (643)

```
    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 245d.

10.12 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

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)* 247⟩≡ (740b)

```
/*
 * 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)* 248⟩≡ (741a)

```
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) 496c⟩
    MCR CpSC, 0, R1, C(CpCONTROL), C(0), CpMainctl

    ⟨armstart() invalidate caches(arm) 496d⟩
    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) 496e⟩
    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) 249a⟩≡ (747)
```
#define L1          (KZERO+0x4000)      /* ttb ptes: 16KiB aligned */
```

⟨constant L2(arm) 249b⟩≡ (747)
```
#define L2          (KZERO+0x3000)      /* L2 ptes for vectors etc */
```

⟨function mmuinit(arm) 249c⟩≡ (748b)
```
/*
 * 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) 503i⟩

    /*
     * 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 160d, Dom0 161d, HVECTORS 28c, IOSIZE 31d, Krw 161b, L1AP 161e, L1X-1842 160a, L1ptedramattrs-1845 497a, L2 249b, L2AP-1844 161i, L2X-1843 160b, L2ptedramattrs-1846 497b, MiB 34g, Section 160d, Small 160f, and VIRTIO 31c.

11.1.4 Jumping to C `main()` (ARM)

⟨*function* `_startpg` *(raspberry pi1)(arm)* 250a⟩≡ (741a)

```
TEXT _startpg(SB), 1, $-4

    ⟨_startpg() enable cycle counter(raspberry pi1)(arm) 475g⟩
    /*
     * 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 (ARM)

`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)* 250b⟩≡ (742b)

```
void
main(void)
```

```
{
    ⟨main() locals(arm) 264c⟩

    ⟨main() initial assignments for backward deps(arm) ??⟩

    ⟨main() clear bss(arm) 252a⟩

    // 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) 254a⟩
    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) 264d⟩
    /* 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) 506g⟩

    // schedule the only ready user process (the one created by userinit)
    schedinit();
```

```
    assert(0);          /* shouldn't have returned */
}
```

Uses ClkArm 727b, L1 249a, arch__confinit() 254d, arch__cpu0init() 252c, arch__userinit() 265d, assert 693a, chandevreset() 264a, clockinit() 258, imageinit() 263c, lineqinit() 360e, machon() 253a, mmuinit1() 259c, pageinit() 261c, print, procinit() 261a, schedinit() 203f, setclkrate() 728b, swapinit() 262b, swcursor_init(), timersinit() 259b, uartconsinit() 842b, and xinit() 260.

⟨main() *clear bss(arm)* 252a⟩≡ (250b)

```
memset(edata, 0, end - edata);  /* clear bss */
```

⟨*lib.h exxx decl* 252b⟩≡ (693c)

```
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

`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)* 252c⟩≡ (742b)

```
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 49e, arch__cpuinit() 252d, conf 46b, cpus 49a, false, and up.

⟨*function* arch__cpuinit*(arm)* 252d⟩≡ (742b)

```
void
arch__cpuinit(void)
{
    Cpu *cpu0;

    cpu->ticks = 1;
    cpu->perf.period = 1;

    cpu0 = CPUS(0);
    ⟨arch__cpuinit() it not cpu0 511b⟩
}
```

Uses CPUS 645.

`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)* 253a⟩≡ (742b)
```
/* 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` 49e, `conf` 46b, `lock()` 58d, and `unlock()` 60d.

⟨*function* `arch_cpuidprint` *(raspberry pi1)(arm)* 253b⟩≡ (744b)
```
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)* 253c⟩≡ (744b)
```
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)* 253d⟩≡ (729)
```
/*
 * 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 725c and vcreq() 726d.

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)* 254a⟩≡ (250b)
```
//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* 254b⟩+≡ (46a) ◁189a 428c▷
```
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)* 254c⟩≡ (742b)
```
// used also in mmu.c
ulong   memsize = 128*1024*1024;
```
Uses memsize 254c.

⟨*function* arch__confinit*(arm)* 254d⟩≡ (742b)
```
void
arch__confinit(void)
{
    int i;
    char *p;
    phys_addr pa;
    ulong kpages;
    ulong kmem;

    ⟨arch__confinit() if maxmem boot parameter(arm) 255⟩
    // simpler than for x86 :)
    getramsize(&conf.mem[0]);
    ⟨arch__confinit() sanity check conf.mem[0](arm) 256a⟩

    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 704a, PGROUND 102a, PTR2UINT 41a, conf 46b, getramsize() 253d, imagmem 84a, mainmem 84a, and nelem 692c.

⟨arch__confinit() *if maxmem boot parameter(arm)* 255⟩≡ (254d)

```
if((p = getconf("*maxmem")) != nil){
```

```
        memsize = strtoul(p, 0, 0);
        if (memsize < 16*MB)        /* sanity */
            memsize = 16*MB;
    }
```

Uses MB 704a and memsize 254c.

⟨arch__confinit() *sanity check* conf.mem[0]*(arm)* 256a⟩≡ (254d)

```
    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) 256b⟩
```

Uses conf 46b and memsize 254c.

⟨arch__confinit() *if maxmem parameter and limit mem is zero(arm)* 256b⟩≡ (256a)

```
    else if(p != nil)
        conf.mem[0].limit = conf.mem[0].base + memsize;
```

Uses conf 46b and memsize 254c.

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)* 256c⟩≡ (750b)

```
    /*
     *   Layout at virtual address KZERO (double mapped at HVECTORS).
     */
    struct Vpage0 {
        void    (*vectors[Nvec])(void);
        u32int  vtable[Nvec];
    };
```

Uses Nvec-2406 256d.

⟨*constant* Nvec*(arm)* 256d⟩≡ (750a)

```
    Nvec = 8,       /* # of vectors at start of lexception.s */
```

⟨*function* arch__trapinit*(arm)* 256e⟩≡ (750b)

```
    /*
     *  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) 498k⟩
    }

    /* set up the stacks for the interrupt modes */
    ⟨arch__trapinit() set stack for other exception/processor-modes 257b⟩
    ⟨arch__trapinit() coherence(arm) 501e⟩
}
```

Uses HVECTORS 28c and intrsoff() 257a.

⟨*function* intrsoff*(arm)* 257a⟩≡ (750b)

```
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 31g.

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* 257b⟩≡ (256e) 527f▷

```
setr13(PsrMirq, cpu->sirq);
setr13(PsrMabt, cpu->sabt);
setr13(PsrMund, cpu->sund);
```

Uses PsrMabt 143a, PsrMirq 143a, and PsrMund 143a.

⟨Cpu Arch *save of first five registers fields(arm)* 257c⟩≡ (48e)

```
/* save areas for exceptions, hold R0-R4 */
u32int  sirq[5];
u32int  sund[5];
u32int  sabt[5];
```

⟨*function* setr13*(arm)* 257d⟩≡ (751b)

```
/*
 *  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)* 258⟩≡ (753)

```
void
clockinit(void)
{
    Systimers *tn;
    Armtimer *tm;
    u32int t0, t1, tstart, tend;

    ⟨clockinit() if many processors 514c⟩

    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 514d⟩
```

```
}
```

Uses ARMTIMER-2648 31f, CntEnable-2657 752c, CntWidth32-2665 752c, Mhz 709a, SYSTIMERS-2647 31e, and TmrPrescale1-2661 752c.

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` 259a⟩≡ (709c)

```
static bool timersinited;
```

⟨*function* `timersinit` 259b⟩≡ (709c)

```
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` 225b, `malloc()` 85a, `timeradd()` 227d, `timersinited-2646` 259a, `todinit()` 238c, 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)* 259c⟩≡ (748b)

```
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) 498d⟩
```

```
    mmuinvalidateaddr(0);
}
```

Uses L1X-1842 160a.

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` 260⟩≡ (697b)

```
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 456b⟩
        }
```

```
        /* 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 694, conf 46b, nelem 692c, palloc 90d, print, xhole() 78c, and xlists-1849 76.

procinit()

⟨*function* procinit 261a⟩≡ (712c)

```
void
procinit(void)
{
    Proc *p;
    int i;

    procalloc.free = xalloc(conf.nproc * sizeof(Proc));
    ⟨procinit() sanity check procalloc.free 261b⟩
    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 46b, procalloc-2458 110c, and xalloc() 77c.

⟨procinit() *sanity check* procalloc.free 261b⟩≡ (261a)

```
if(procalloc.free == nil){
    xsummary();
    panic("cannot allocate %lud procs (%ludMB)\n",
                conf.nproc, conf.nproc*sizeof(Proc)/MB);
}
```

Uses procalloc-2458 110c and xsummary() 610a.

pageinit()

⟨*function* pageinit 261c⟩≡ (701)

```
void
pageinit(void)
{
    int i, j;
    Page *p;
    Pallocmem *pm;
    ulong m, np, k, vkb, pkb;
    ⟨pageinit() other locals 521b⟩

    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 262a⟩

    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 521c⟩
            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 704a, PGROUND 102a, assert 693a, conf 46b, nelem 692c, palloc 90d, print, swapalloc 185b, and xalloc() 77c.

⟨pageinit() *sanity check* palloc.pages 262a⟩≡ (261c)

```
if(palloc.pages == nil)
    panic("pageinit");
```

Uses palloc 90d.

swapinit()

⟨*function* swapinit 262b⟩≡ (703a)

```
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 263a⟩
    swapimage.notext = true;
}
```

Uses conf 46b, iolist-1807 189b, swapalloc 185b, swapimage 185d, true, and xalloc() 77c.

⟨swapinit() *sanity check xalloc return values* 263a⟩≡ (262b)

```
if(swapalloc.swmap == nil || iolist == nil)
    panic("swapinit: not enough memory");
```

Uses iolist-1807 189b and swapalloc 185b.

imageinit()

⟨*constant* NFREECHAN 263b⟩≡ (694)

```
#define NFREECHAN 64
```

⟨*function* imageinit 263c⟩≡ (702)

```
void
imageinit(void)
{
    KImage *i, *ie;

    imagealloc.free = xalloc(conf.nimage * sizeof(KImage));
    ⟨imageinit() sanity check imagealloc.free 263d⟩
    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 174d, conf 46b, imagealloc-1803 174e, malloc() 85a, and xalloc() 77c.

⟨imageinit() *sanity check* imagealloc.free 263d⟩≡ (263c)

```
if (imagealloc.free == nil)
    panic("imageinit: no memory");
```

Uses imagealloc-1803 174e.

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` 264a⟩≡ (684)
```
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()` 238c.

⟨`Dev` *methods* 264b⟩≡ (54a) 268a▷
```
void  (*reset)(void); // done once at boot time
```

11.2.10 Displaying hardware information (ARM)

⟨`main()` *locals(arm)* 264c⟩≡ (250b)
```
uint firmware, board;
```

⟨`main()` *print board and firmware information(arm)* 264d⟩≡ (250b)
```
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` 264f, `Minfirmrev-2431` 264e, `getboardrev()` 264g, `getfirmware()` 265a, and `print`.

⟨*constant* `Minfirmrev`*(arm)* 264e⟩≡ (742b)
```
/* Firmware compatibility */
#define Minfirmrev  326770
```

⟨*constant* `Minfirmdate`*(arm)* 264f⟩≡ (742b)
```
#define Minfirmdate "19 Aug 2013"
```

⟨*function* `getboardrev`*(arm)* 264g⟩≡ (729)
```
/*
 * 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` 725c and `vcreq()` 726d.

⟨*function* `getfirmware` *(arm)* 265a⟩≡ (729)

```
/*
 * 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` 725c and `vcreq()` 726d.

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)* 265b⟩≡ (742b)

```
static user_addr sp;      /* XXX - must go - user stack of init proc */
```

⟨*macro* `STACKALIGN` *(arm)* 265c⟩≡ (747)

```
#define STACKALIGN(sp)  ((sp) & ~3)     /* bug: assure with alloc */
```

⟨*function* `arch__userinit` *(arm)* 265d⟩≡ (742b)

```
/*
 *  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 522g⟩
    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) 522d⟩
    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 41a, SG_STACK 94a, SG_TEXT, SSEG 100a, STACKALIGN 265c, TSEG 100a, UINT2PTR 41b, dupfgrp() 276c, eve 557a, initcode, kstrdup() 636b, newpgrp() 320f, newproc() 111d, newrgrp() 448c, newseg() 97b, ready, segpage() 99e, smalloc() 85c, sp-2433 265b, 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)` 267a⟩≡ (742b)

```
/*
 *  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) 501f⟩
    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` 282e, `KNAMELEN` 692b, `alarmkproc()` 236c, `assert` 693a, `cclone()` 307b, `chandevinit()` 267b, `confname` 47a, `confval` 47a, `getethermac()` 727a, `kproc()` 139c, `ksetenv()` 564b, `namec()` 283d, `nconf` 47c, `newpath()` 300c, `pathclose()` 301b, `poperror` 633d, `sp-2433` 265b, `up`, and `waserror` 633d.

⟨*function* `chandevinit` 267b⟩≡ (684)

```
void
chandevinit(void)
{
    int i;
```

```
    for(i=0; devtab[i] != nil; i++) {
        devtab[i]->init();
    }
}
```

⟨Dev *methods* 268a⟩+≡ (54a) ◁264b 282a▷
```
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)* 268b⟩≡ (756)
```
/*
 *  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 268c⟩≡
```
//@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 269a⟩≡

```
/*
 * 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 269b⟩≡

```
/*
 * 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* 272a⟩≡ (41f)
```
[OPEN]      sysopen,
[CLOSE]     sysclose,
[PREAD]     syspread,
[PWRITE]    syspwrite,
[SEEK]      sysseek,
```
Uses `syspread()` 290b, `syspwrite()` 292a, and `sysseek()` 293a.

12.1 Overview

In this chapter, I present the file-related system calls: `sysopen()`, `sysclose()`, `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` 272b⟩≡ (682c)
```
/*
 * channel flags
 */
enum ChannelFlag
{
  COPEN = 0x0001,   /* for i/o */
  CFREE = 0x0010,   /* not in use */
  ⟨ChannelFlag cases 297b⟩
};
```

⟨`Chan` *other fields* 272c⟩≡ (51e) 331d▷
```
// 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` 273a⟩≡ (684)
```
// 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* 273b⟩≡ (51e)
```
// list<ref<Chan> of Chanalloc.free
Chan* next;     /* allocation */
// list<ref<Chan> of Chanalloc.list
Chan* link;
```

⟨*global* `chanalloc` 273c⟩≡ (684)
```
struct Chanalloc chanalloc;
```
Uses `Chanalloc` 273a.

`\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` 273d⟩≡ (684)
```
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()` 58d, `smalloc()` 85c, and `unlock()` 60d.

⟨*destructor* `chanfree` 274a⟩≡ (684)

```
void
chanfree(Chan *c)
{
    c->flag = CFREE;

    ⟨chanfree() optional free 332a⟩

    pathclose(c->path);
    c->path = nil;

    lock(&chanalloc);
    c->next = chanalloc.free;
    chanalloc.free = c;
    unlock(&chanalloc);
}
```

Uses `CFREE` 272b, `lock()` 58d, `pathclose()` 301b, and `unlock()` 60d.

`\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` 274b⟩≡ (684)

```
/*
 * 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()` 307b 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` 275a⟩≡ (686)

```
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` 52c, `UTFmax` 639a, `devno()` 275b, `false`, `free()` 85b, `newchan()` 273d, `newpath()` 300c, and `smalloc()` 85c.

⟨*function* `devno` 275b⟩≡ (686)

```
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 276a⟩≡ (705c)
```
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 279a⟩
    // extra
    Ref;
};
```

⟨Proc *files fields* 276b⟩+≡ (49f) ◁52a 295a▷
```
// 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 276c⟩≡ (710)
```
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 277a, free() 85b, incref() 69b, lock() 58d, malloc() 85a, smalloc() 85c, and unlock() 60d.

⟨*constant* DELTAFD 277a⟩≡ (705c)

```
DELTAFD = 20    /* incremental increase in Fgrp.fd's */
```

⟨*function* closefgrp 277b⟩≡ (710)

```
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() 69c, free() 85b, 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 277c⟩≡ (690)

```
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()` 278a, `lock()` 58d, `unlockfgrp()` 279b, and `up`.

⟨*function* `findfreefd` 278a⟩≡ (690)

```
/*
 *  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()` 278b.

⟨*function* `growfd` 278b⟩≡ (690)

```
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 277a, free() 85b, malloc() 85a, and print.

⟨Fgrp *other fields* 279a⟩≡ (276a)

```
int exceed;     /* debugging */
```

⟨*function* unlockfgrp 279b⟩≡ (690)

```
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() 60d.

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 279c⟩≡ (693c)

```
enum OpenFlags {
    OREAD = 0, /* open for read */
    OWRITE = 1, /* write */
    ORDWR = 2, /* read and write */
    ⟨OpenFlags cases 282d⟩
};
```

⟨*syscall open* 279d⟩≡ (690)

```
// 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 282e, Enofd 631d, cclose, false, namec() 283d, newfd() 277c, nexterror, openmode() 280a, poperror 633d, validaddr() 101c, and waserror 633d.

⟨*function* openmode 280a⟩≡ (688b)

```
int
openmode(ulong o)
{
    o &= ~(OTRUNC|OCEXEC|ORCLOSE);
    if(o > OEXEC)
        error(Ebadarg);
    if(o == OEXEC)
        return OREAD;
    return o;
}
```

Uses Ebadarg 631d, OCEXEC 297a, OEXEC 282d, ORCLOSE 298c, OREAD 279c, and OTRUNC 310a.

⟨*syscall close* 280b⟩≡ (690)

```
// 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() 281b, and fdtochan() 280c.

\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 280c⟩≡ (688b)

```
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 281a⟩
        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 281a⟩
        error(Ebadusefd);
    }
    // the access mode must match the opening mode
    if((mode&~OTRUNC) != c->mode) {
        ⟨fdtochan() undo iref incref 281a⟩
        error(Ebadusefd);
    }

    return c;
}
```

Uses CMSG 549b, Ebadfd 631d, Ebadusefd 631d, ORDWR 279c, OREAD 279c, OTRUNC 310a, incref() 69b, lock() 58d, unlock() 60d, and up.

⟨fdtochan() *undo iref incref* 281a⟩≡ (280)

```
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 281b⟩≡ (688b)

```
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() 58d, unlock() 60d, and up.

⟨Dev *methods* 282a⟩+≡ (54a) ◁268a 285a▷
```
  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 282b⟩≡ (684)
```
  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 272b, DBG-765, chanfree() 274a, decref() 69c, poperror 633d, and waserror 633d.

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* 282c⟩≡ (123b)
```
  tc = namec(file, Aopen, OEXEC, 0);
```

⟨OpenFlags *cases* 282d⟩≡ (279c) 297a▷
```
  OEXEC = 3, /* execute, == read but check execute permission */
```

⟨*enum* accessnamec 282e⟩≡ (682c)
```
  /*
   * 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 323a⟩
  };
```

⟨namec() *locals* 283a⟩≡ (283d) 283b▷
```
char *name;
```

⟨namec() *locals* 283b⟩+≡ (283d) ◁283a 283c▷
```
Elemlist e;
```

⟨namec() *locals* 283c⟩+≡ (283d) ◁283b ??▷
```
Chan *c;
```

⟨*function* namec 283d⟩≡ (684)
```
/*
 * 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 283a⟩

    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 321c⟩
    case '/':
        c = up->slash;
        incref(c);
        break;

    default:
        c = up->dot;
        incref(c);
```

```
    break;
}

⟨namec() initializes Elemlist e 287b⟩

if(waserror()){
    cclose(c);
    ⟨namec() if waserror free Elemlist e and prepare nice error 289b⟩
}

/*
 * Build a list of elements in the name.
 */
parsename(name, &e);

/*
 * On create, ....
 */
⟨namec() adjust Elemlist e if Acreate 308a⟩

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 300a⟩

switch(amode){
case Aopen:
case Aremove:
case Aaccess:
Open:
    ⟨namec() case Aopen, Acreate, Aremove, Aaccess, handle mountpoint part1 327b⟩

    switch(amode){
    case Aopen:
    case Acreate:
        ⟨namec() print error if c-¿umh != nil 332b⟩
        ⟨namec() case Aopen, Acreate, handle mountpoint part2 331c⟩

        ⟨namec() set channel flag before open ??⟩
        c = devtab[c->type]->open(c, omode&~OCEXEC);
        ⟨namec() set channel flag after open 297c⟩
        break;

    case Aaccess:
    case Aremove:
       ⟨namec() case Aremove, Aaccess, handle mountpoint part2 328a⟩
        break;

    }
    break;
⟨namec() other cases 308d⟩
default:
    panic("unknown namec access %d\n", amode);
}
```

```
    ⟨namec() set genbuf from Elemlist e 285b⟩

    ⟨namec() free Elemlist e 287c⟩
    poperror(); /* e c */
    free(aname);
    poperror(); /* aname */

    return c;
}
```

Uses Aaccess 282e, Acreate 282e, Aopen 282e, Aremove 282e, DBG-765, OCEXEC 297a, cclose, free() 85b, incref() 69b, nexterror, poperror 633d, print, true, up, validnamedup() 285d, walk() 304b, and waserror 633d.

```
⟨Dev methods 285a⟩+≡                                    (54a) ◁282a 290c▷
  Chan* (*open)(Chan*, int);
```

```
⟨namec() set genbuf from Elemlist e 285b⟩≡                                (283d)
  /* 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() 636a and up.

```
⟨Proc other fields 285c⟩+≡                                (49f) ◁234f 480b▷
  char  genbuf[128];  /* buffer used e.g. for last name element from namec */
```

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 285d⟩≡                                            (684)
  char*
  validnamedup(char *aname, bool slashok)
  {
      return validname0(aname, slashok, true, getcallerpc(&aname));
  }
```

Uses true and validname0() 285f.

```
⟨function validname 285e⟩≡                                               (684)
  void
  validname(char *aname, bool slashok)
  {
      validname0(aname, slashok, false, getcallerpc(&aname));
  }
```

Uses false and validname0() 285f.

```
⟨function validname0 285f⟩≡                                              (684)
  /*
   * 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 631d, Runeself 639a, free() 85b, print, setmalloctag() 87b, smalloc() 85c, up, and vmemchr() 101d.

⟨*global* isfrog 286⟩≡ (684)

```
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` 287a⟩≡ (684)

```
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* 287b⟩≡ (283d)

```
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* 287c⟩≡ (283d)

```
free(e.name);
free(e.elems);
free(e.off);
```

Uses `free()` 85b.

⟨*function* `parsename` 287d⟩≡ (684)

```
/*
 * 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()` 288a, `kstrdup()` 636b, `skipslash()` 288b, `smalloc()` 85c, and `true`.

⟨*function* `growparse` 288a⟩≡ (684)

```
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()` 85b and `smalloc()` 85c.

⟨*function* `skipslash` 288b⟩≡ (684)

```
/*
 * 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* 289a⟩+≡ (283d) ◁?? 304a▷

```
char tmperrbuf[ERRMAX];
```

Uses `ERRMAX` 692a.

⟨`namec()` *if waserror free Elemlist e and prepare nice error* 289b⟩≡ (283d)

```
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()` 85b, `namelenerror()` 289c, `nexterror`, `print`, and `up`.

⟨*function* `namelenerror` 289c⟩≡ (684)

```
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 692a, UTFmax 639a, memrchr() 635b, nexterror, and up.

⟨*function* nameerror 290a⟩≡ (684)

```
void
nameerror(char *name, char *err)
{
    namelenerror(name, strlen(name), err);
}
```

Uses namelenerror() 289c.

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* 290b⟩≡ (690)

```
// 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() 291.

⟨Dev *methods* 290c⟩+≡ (54a) ◁285a 292b▷

```
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` 291⟩≡ (690)

```
// 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 334a⟩
    }

    ⟨read() if c is a QTDIR 332f⟩
    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` 631d, `OREAD` 279c, `cclose`, `fdtochan()` 280c, `lock()` 58d, `nexterror`, `poperror` 633d, `true`, `unlock()` 60d,

validaddr() 101c, and waserror 633d.

⟨*syscall pwrite* 292a⟩≡ (690)
```
// 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() 292c.

⟨Dev *methods* 292b⟩+≡ (54a) ◁290c 303a▷
```
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 292c⟩≡ (690)
```
// 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 631d, OWRITE 279c, QTDIR 52c, cclose, false, fdtochan() 280c, lock() 58d, nexterror, poperror 633d, true, unlock() 60d, validaddr() 101c, and waserror 633d.

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* 293a⟩≡ (690)

```
// 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() 293b, true, and validaddr() 101c.

⟨*function* sseek 293b⟩≡ (690)

```
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 427a⟩

    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 299c⟩
        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 299d⟩
        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 299d⟩
        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 631d, Enegoff 631d, cclose, fdtochan() 280c, lock() 58d, nexterror, poperror 633d, true, unlock() 60d, v_or_u2 293b, and waserror 633d.

12.9 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* 295a⟩+≡ (49f) ◁276b 320a▷
```
Fgrp  *closingfgrp; /* used during teardown */
```

⟨sleep() *forceclosefgrp* 295b⟩≡ (419g)
```
if(up->procctl == Proc_exitme && up->closingfgrp)
    forceclosefgrp();
```
Uses Proc_exitme 141a, forceclosefgrp() 295c, and up.

⟨*function* forceclosefgrp 295c⟩≡ (710)
```
/*
 * 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 141a, ccloseq() 296b, 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 295d⟩≡ (684)
```
/*
 * 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` 296a⟩≡ (684)
```
struct Clunkq clunkq;
```
Uses `Clunkq` 295d.

`\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` 296b⟩≡ (684)
```
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` 272b, `DBG-765`, `closeproc()` 296d, `clunkq` 296a, `decref()` 69c, `kproc()` 139c, `lock()` 58d, `unlock()` 60d, and `wakeup`.

⟨*function* `clunkwork` 296c⟩≡ (684)
```
static int
clunkwork(void*)
{
    return clunkq.head != nil;
}
```
Uses `clunkq` 296a.

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` 296d⟩≡ (684)
```
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() 274a, clunkq 296a, clunkwork() 296c, lock() 58d, pexit, poperror 633d, qlock() 63f, qunlock() 64c, true, tsleep, unlock() 60d, and waserror 633d.

12.10 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.

12.10.1 Close on exec

⟨OpenFlags *cases* 297a⟩+≡ (279c) ◁282d 298c▷

```
OCEXEC = 32,  /* or'ed in, close on exec */
```

⟨ChannelFlag *cases* 297b⟩≡ (272b) 298d▷

```
CCEXEC  = 0x0008,   /* close on exec */
```

⟨namec() *set channel flag after open* 297c⟩≡ (283d) 298e▷

```
if(omode & OCEXEC)
    c->flag |= CCEXEC;
```

Uses CCEXEC 297b and OCEXEC 297a.

⟨namec() *set channel flag after create* 297d⟩≡ (308d) 298f▷

```
if(omode & OCEXEC)
    cnew->flag |= CCEXEC;
```

Uses CCEXEC 297b and OCEXEC 297a.

⟨sysexec() *locals* 298a⟩+≡ (123b) ◁176c 476b▷

```
Fgrp *f;
```

⟨sysexec() *close files marked as opened with close on exec* 298b⟩≡ (123b)

```
/*
 * Close on exec
 */
f = up->fgrp;
for(i=0; i<=f->maxfd; i++)
    fdclose(i, CCEXEC);
```

12.10.2 Remove on close

⟨OpenFlags *cases* 298c⟩+≡ (279c) ◁297a 310a▷

```
ORCLOSE = 64,  /* or'ed in, remove on close */
```

⟨ChannelFlag *cases* 298d⟩+≡ (272b) ◁297b 549b▷

```
CRCLOSE = 0x0020,   /* remove on close */
```

⟨namec() *set channel flag after open* 298e⟩+≡ (283d) ◁297c

```
if(omode & ORCLOSE)
    c->flag |= CRCLOSE;
```

Uses CRCLOSE 298d and ORCLOSE 298c.

⟨namec() *set channel flag after create* 298f⟩+≡ (308d) ◁297d

```
if(omode & ORCLOSE)
    cnew->flag |= CRCLOSE;
```

Uses CRCLOSE 298d and ORCLOSE 298c.

Chapter 13

Directories

⟨systab *directory syscalls* 299a⟩≡ (41f)
```
[CREATE]    syscreate,
[REMOVE]    sysremove,

[CHDIR]     syschdir,
[FD2PATH]   sysfd2path, // =~ pwd

[STAT]      sysstat,
[FSTAT]     sysfstat,
[WSTAT]     syswstat,
[FWSTAT]    sysfwstat,
```
Uses syschdir() 311a, syscreate() 307c, sysfd2path() 312a, sysfstat() 314c, sysfwstat() 315c, sysremove() 310d, and syswstat() 315b.

13.1 Overview

This chapter covers the directory-related syscalls: `create`, `remove`, `chdir`, `fd2path`, `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 299b⟩≡ (684)
```
void
error_if_not_dir(Chan *c)
{
    if(c->qid.type & QTDIR)
        return;
    error(Enotdir);
}
```
Uses QTDIR 52c.

⟨sseek() *ensures off is 0 for directories* 299c⟩≡ (293b)
```
if((c->qid.type & QTDIR) && off != 0)
    error(Eisdir);
```
Uses QTDIR 52c.

⟨sseek() *disallows seek type 1 or 2 for directories* 299d⟩≡ (293)
```
if(c->qid.type & QTDIR)
    error(Eisdir);
```
Uses QTDIR 52c.

⟨namec() *error if not a directory or cannot exec directory* 300a⟩≡ (283d)
```
if(e.mustbedir && !(c->qid.type & QTDIR))
    error(Enotdir);

if((amode == Aopen) && (omode & OEXEC) && (c->qid.type & QTDIR))
    error("cannot exec directory");
```
Uses Aopen 282e, OEXEC 282d, and QTDIR 52c.

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 300b⟩≡ (682c)
```
struct Path
{
    char  *s;
    int len;      /* strlen(s) */
    int alen;     /* allocated length of s */

    ⟨Path mtpt fields 329a⟩

    // extra
    Ref;
};
```

⟨*constructor* newpath 300c⟩≡ (684)
```
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 329c⟩
    return p;
}
```
Uses PATHSLOP-766 300d, incref() 69b, npath, and smalloc() 85c.

⟨*constant* PATHSLOP 300d⟩≡ (684)
```
PATHSLOP    = 20,
```

⟨*function* `copypath` 301a⟩≡ (684)

```
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 330a⟩
    return pp;
}
```

Uses DBG-765, `incref()` 69b, `npath`, and `smalloc()` 85c.

⟨*destructor* `pathclose` 301b⟩≡ (684)

```
void
pathclose(Path *p)
{
    int i;

    if(p == nil)
        return;
    ⟨pathclose() debugging ??⟩
    if(decref(p))
        return;

    decref(&npath);
    free(p->s);

    ⟨pathclose() mtpt handling 330b⟩
    free(p);
}
```

Uses `decref()` 69c, `free()` 85b, and `npath`.

⟨*function* `uniquepath` 301c⟩≡ (684)

```
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()` 301a and `pathclose()` 301b.

`\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` 302a⟩≡ (684)

```
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 330c⟩

    return p;
}
```

Uses `PATHSLOP-766` 300d, `fixdotdotname()` 302c, `free()` 85b, `isdotdot()` 302b, `smalloc()` 85c, and `uniquepath()` 301c.

13.3 . .

⟨*function* `isdotdot` 302b⟩≡ (684)

```
int
isdotdot(char *p)
{
    return p[0]=='.' && p[1]=='.' && p[2]=='\0';
}
```

⟨*function* `fixdotdotname` 302c⟩≡ (684)

```
/*
 * 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 Qids for each element successfully walked. Taking multiple names in a single call avoids a 9P round-trip per path element.

⟨Dev *methods* 303a⟩+≡ (54a) ◁292b 308c▷
```
Walkqid*(*walk)(Chan*, Chan*, char**, int);
```

⟨*struct* Walkqid 303b⟩≡ (682c)
```
struct Walkqid
{
  Chan  *clone;
  int nqid;
  // variable array length, size = nqid
  Qid qid[1];
};
```

⟨walk() *locals* 303c⟩≡ (304b) 303d▷
```
Chan *c;
Path *path;
Walkqid *wq;
Chan *nc;
```

⟨walk() *locals* 303d⟩+≡ (304b) ◁303c 303e▷
```
Mhead *mh;
Mhead *nmh;
int type, dev;
```

⟨walk() *locals* 303e⟩+≡ (304b) ◁303d 303f▷
```
Chan *mtpt;
bool didmount;
```

⟨walk() *locals* 303f⟩+≡ (304b) ◁303e 303g▷
```
bool dotdot;
```

⟨walk() *locals* 303g⟩+≡ (304b) ◁303f ??▷
```
int nhave, ntry;
```

⟨namec() *locals* 304a⟩+≡ (283d) ◁289a 308b▷
```
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.

⟨*function* walk 304b⟩≡ (684)
```
/*
 * 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 303c⟩

    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 306c⟩

        ⟨walk() set dotdot and ntry to index of next '..' or end 306b⟩

        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 331b⟩
            ⟨walk() if wq is still nil, close and return 306d⟩
        }

        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 307a⟩
            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 332c⟩

pathclose(c->path);
c->path = path;

cclose(*cp);
*cp = c;
if(nerror)
    *nerror = nhave;
return 0;
```

```
}
```

Uses addelem() 302a, assert 693a, cclose, domount() 328c, ewalk() 306a, false, findmount() 328b, free() 85b, incref() 69b, pathclose() 301b, putmhead() 318b, true, undomount() 330d, and uniquechan() 274b.

⟨*function* ewalk 306a⟩≡ (684)

```
/*
 * 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 633d and waserror 633d.

⟨walk() *set dotdot and ntry to index of next '..' or end* 306b⟩≡ (304b)

```
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() 302b, and true.

⟨walk() *return error if channel c is not a dir* 306c⟩≡ (304b)

```
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 52c, cclose, pathclose() 301b, putmhead() 318b, and up.

⟨walk() *if wq is still nil, close and return* 306d⟩≡ (304b)

```
if(wq == nil){
    pathclose(path);
    cclose(c);
    if(nerror)
```

```
            *nerror = nhave+1;
        if(mh != nil)
            putmhead(mh);
        return -1;
    }
```

Uses cclose, pathclose() 301b, and putmhead() 318b.

⟨walk() *if nc == nil and wq-¿clone == nil return error* 307a⟩≡ (304b)

```
    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 632a, QTDIR 52c, cclose, free() 85b, pathclose() 301b, putmhead() 318b, and up.

⟨*function* cclone 307b⟩≡ (684)

```
    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() 85b and incref() 69b.

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* 307c⟩≡ (690)

```
    // 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 282e, Enofd 631d, OEXCL 310b, cclose, false, namec() 283d, newfd() 277c, nexterror, openmode() 280a, poperror 633d, validaddr() 101c, and waserror 633d.

⟨namec() *adjust Elemlist e if Acreate* 308a⟩≡ (283d)

```
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 282e and DMDIR 313a.

⟨namec() *locals* 308b⟩+≡ (283d) ◁304a 321b▷

```
Chan *cnew;
char *createerr;
```

⟨Dev *methods* 308c⟩+≡ (54a) ◁303a 310c▷

```
void  (*create)(Chan*, char*, int, ulong);
```

⟨namec() *other cases* 308d⟩≡ (283d) 311b▷

```
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 334c⟩
    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 297d⟩
        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 282e, OCEXEC 297a, OEXCL 310b, OTRUNC 310a, addelem() 302a, cclose, incref() 69b, nexterror, pathclose() 301b, poperror 633d, putmhead() 318b, uniquechan() 274b, up, walk() 304b, and waserror 633d.

⟨OpenFlags *cases* 310a⟩+≡ (279c) ◁298c 310b▷

```
OTRUNC = 16,  /* or'ed in (except for exec), truncate file first */
```

⟨OpenFlags *cases* 310b⟩+≡ (279c) ◁310a

```
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* 310c⟩+≡ (54a) ◁308c 314a▷

```
void  (*remove)(Chan*);
```

⟨*syscall remove* 310d⟩≡ (690)

```
// 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 282e, Eismtpt 631d, cclose, false, namec() 283d, nexterror, poperror 633d, validaddr() 101c, and waserror 633d.

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* 311a⟩≡ (690)

```
// 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 282e, cclose, false, namec() 283d, up, and validaddr() 101c.

⟨namec() *other cases* 311b⟩+≡ (283d) ◁308d 323b▷

```
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 282e and QTDIR 52c.

13.8 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* 312a⟩≡ (690)
```
// 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() 312b, false, fdtochan() 280c, true, and validaddr() 101c.

⟨*function* chanpath 312b⟩≡ (684)
```
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;
}
```

13.9 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 312c⟩≡ (693c)
```
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 313a⟩≡ (693c)
```
  /* 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 403a⟩
  };
```

⟨*function* direntry_setname 313b⟩≡ (690)
```
  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() 313c.

⟨*function* offsetof_name_direntry 313c⟩≡ (690)
```
  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;
  }
```

13.10 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* 314a⟩+≡ (54a) ◁310c 315a▷
```
int (*stat)(Chan*, byte*, int);
```

⟨*syscall stat* 314b⟩≡ (690)
```
// 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 282e, cclose, direntry_setname() 313b, false, namec() 283d, nexterror, pathlast() 314d, poperror 633d, true, validaddr() 101c, and waserror 633d.

⟨*syscall fstat* 314c⟩≡ (690)
```
// 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() 280c, nexterror, poperror 633d, true, validaddr() 101c, and waserror 633d.

⟨*function* pathlast 314d⟩≡ (690)
```
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* 315a⟩+≡ (54a) ◁314a 322a▷
```
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* 315b⟩≡ (690)
```
// 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` 282e, `false`, `namec()` 283d, `validaddr()` 101c, `validstat()` 315d, and `wstat()` 316.

⟨*syscall fwstat* 315c⟩≡ (690)
```
// 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()` 280c, `true`, `validaddr()` 101c, `validstat()` 315d, and `wstat()` 316.

⟨*function* `validstat` 315d⟩≡ (688b)
```
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()` 285e.

⟨*function* `wstat` 316⟩≡ (690)

```
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` 631d, `cclose`, `chanpath()` 312b, `nameerror()` 290a, `nexterror`, `offsetof_name_direntry()` 313c, `poperror` 633d, and `waserror` 633d.

Chapter 14

Namespaces

⟨systab *namespace syscalls* 317a⟩≡ (41f)
```
[BIND]      sysbind,
[MOUNT]     sysmount,
[UNMOUNT]   sysunmount,
```
Uses `sysbind()` 322b, `sysmount()` 337d, and `sysunmount()` 338c.

14.1 Overview

Namespaces are Plan 9's most distinctive kernel feature. Each process has its own namespace — a private view of the file hierarchy, defined by a mount table (`\hlinkPgrp`). The `bind` and `mount` syscalls modify this table, and `\hlinknamec` consults it when resolving paths. A mount point pairs a "from" channel (`\hlinkMhead`) with one or more "to" channels (`\hlinkMount` list). Union mounts allow multiple directories to appear merged at a single path, ordered by `MBEFORE`, `MAFTER`, or replaced entirely with `MREPL`.

⟨*enum* `mount` 317b⟩≡ (693c)
```
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 338b⟩

    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.

14.2 Mount point

⟨*struct* `Mhead` 317c⟩≡ (682c)
```
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 320c⟩
};
```

⟨*constructor* `newmhead` 318a⟩≡ (684)

```
Mhead*
newmhead(Chan *from)
{
    Mhead *mh;

    mh = smalloc(sizeof(Mhead));
    mh->ref = 1;
    mh->from = from;
    incref(from);
    return mh;
}
```

Uses `incref()` 69b and `smalloc()` 85c.

⟨*destructor* `putmhead` 318b⟩≡ (684)

```
/*
 * 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()` 69c and `free()` 85b.

A `\hlinkMount` holds the "to" channel and flags. The `MBEFORE`/`MAFTER` ordering is implicit in list position.

⟨*struct* `Mount` 318c⟩≡ (682c)

```
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 340a⟩

    ⟨Mount extra fields 319b⟩
};
```

⟨*constructor* `newmount` 319a⟩≡ (710)

```
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()` 69b, `kstrdup()` 636b, `mountid-2606` 340b, and `smalloc()` 85c.

⟨`Mount` *extra fields* 319b⟩≡ (318c) 341a▷

```
Mount*  next;
```

⟨*destructor* `mountfree` 319c⟩≡ (710)

```
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()` 85b.

14.3 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` 319d⟩≡ (705c)

```
// Mount table, aka Namespace, aka process group
struct Pgrp
{
```

```
    // hash<qid.path, list<ref<Mhead> (next = Mhead.next)>
    Mhead *mnthash[MNTHASH];

    ⟨Pgrp other fields 320d⟩

    // 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* 320a⟩+≡ (49f) ◁295a

```
// 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 320b⟩≡ (705c)

```
enum
{
    MNTLOG  = 5,
    MNTHASH = 1<<MNTLOG,  /* Hash to walk mount table */
};
#define MOUNTH(p,qid) ((p)->mnthash[(qid).path&((1<<MNTLOG)-1)])
```

Uses MNTLOG 320b.

⟨Mhead *extra fields* 320c⟩≡ (317c)

```
// hash<qid.path, list<ref<Mhead>> of Pgrp.mnthash
Mhead*  hash;     /* Hash chain */
```

⟨Pgrp *other fields* 320d⟩≡ (319d) 336b▷

```
ulong pgrpid;
```

⟨*global* pgrpid 320e⟩≡ (710)

```
static Counter pgrpid;
```

⟨*constructor* newpgrp 320f⟩≡ (710)

```
Pgrp*
newpgrp(void)
{
    Pgrp *p;

    p = smalloc(sizeof(Pgrp));
    p->ref = 1;
    p->pgrpid = incref(&pgrpid);
    return p;
}
```

Uses incref() 69b, pgrpid-2605 320e, and smalloc() 85c.

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` 321a⟩≡ (710)
```
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` 320b, `cclose`, `decref()` 69c, `free()` 85b, `mountfree()` 319c, `putmhead()` 318b, `qlock()` 63f, `qunlock()` 64c, `wlock()` 67b, and `wunlock()` 67d.

14.4 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* 321b⟩+≡ (283d) ◁308b 327a▷
```
Rune r;
```

⟨`namec()` *if name[0] is a sharp* 321c⟩≡ (283d)
```
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* 336d⟩

```
t = devno(r, true);
if(t == -1)
    error(Ebadsharp);

c = devtab[t]->attach(up->genbuf+n);
break;
```

Uses Ebadsharp 631d, Efilename 631d, devno() 275b, true, and up.

⟨Dev *methods* 322a⟩+≡ (54a) ◁315a 404b▷

```
Chan* (*attach)(char*);
```

14.5 sysbind()

Both bind and mount go through \hlinkbindmount, which resolves two paths and calls \hlinkcmount to insert a mount point.

⟨*syscall bind* 322b⟩≡ (690)

```
// 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() 322d and false.

⟨bindmount() *locals* 322c⟩≡ (322d) ??▷

```
Chan *c0;
Chan *c1;
```

⟨*function* bindmount 322d⟩≡ (690)

```
long
bindmount(bool ismount, int fd, int afd, char* arg0, char* arg1, ulong flag, char* spec)
{
    ⟨bindmount() locals 322c⟩

    if((flag&~MMASK) || (flag&MORDERMASK)==(MBEFORE|MAFTER))
        error(Ebadarg);

    ⟨bindmount() if ismount 337f⟩
    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 338a⟩
    return ret;
}
```

Uses Abind 323a, Amount 323a, Ebadarg 631d, MAFTER 317b, MBEFORE 317b, MMASK 317b, MORDERMASK 317b, cclose, cmount() 323e, false, namec() 283d, nexterror, poperror 633d, validaddr() 101c, and waserror 633d.

⟨AccessFlags cases 323a⟩≡ (282e)
```
Abind,      /* for left-hand-side of bind */
Amount,     /* to be mounted or mounted upon */
```

⟨namec() other cases 323b⟩+≡ (283d) ◁311b 323c▷
```
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 335a⟩
    break;
```

Uses Abind 323a and domount() 328c.

⟨namec() other cases 323c⟩+≡ (283d) ◁323b
```
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 323a.

⟨cmount() locals 323d⟩≡ (323e) 335b▷
```
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 323e⟩≡ (684)
```
int
cmount(Chan *new, Chan *old, int flag, char *spec)
{
    ⟨cmount() locals 323d⟩

    if((old->qid.type ^ new->qid.type) & QTDIR)
        error(Emount);
    ⟨cmount() print error if old-¿umh != nil 332d⟩
```

```
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* 335c⟩

```
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* 336a⟩

```
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 631d, MAFTER 317b, MCREATE 317b, MORDERMASK 317b, MOUNTH 320b, MREPL 317b, QTDIR 52c, eqchan() 53c, mountfree() 319c, newmhead() 318a, newmount() 319a, nexterror, poperror 633d, true, up, waserror 633d, wlock() 67b, and wunlock() 67d.

14.6 /proc/#/ns

A process's namespace can be inspected by reading `/proc/\#/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 325a⟩≡ (680)

```
struct Mntwalk              /* state for /proc/#/ns */
{
    bool cddone;
    Mhead*  mh;
    Mount*  cm;
};
```

⟨procread() *locals* 325b⟩+≡ (571d) ◁104a 571c▷

```
Mntwalk *mw;
char flag[10];
```

⟨procopen() *cases* 325c⟩≡ (577)

```
case Qns:
    if(omode != OREAD)
        error(Eperm);
    c->aux = malloc(sizeof(Mntwalk));
    break;
```

Uses OREAD 279c, Qns-870 568a, and malloc() 85a.

⟨procclose() *hooks* 325d⟩≡ (583)

```
if(QID(c->qid) == Qns && c->aux != nil)
    free(c->aux);
```

Uses QID-911 570a, Qns-870 568a, and free() 85b.

⟨procread() *cases* 325e⟩+≡ (571d) ◁104b 456a▷

```
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 631d, PID-913 570a, Qns-870 568a, free() 85b, int2flag() 326a, mntscan() 326b, nexterror, poperror 633d, qlock() 63f, qunlock() 64c, srvname(), true, and waserror 633d.

⟨*function* int2flag 326a⟩≡ (680)

```
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 317b, MBEFORE 317b, MCACHE 338b, and MCREATE 317b.

\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 326b⟩≡ (680)

```
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 320b, false, rlock() 65e, runlock() 66b, and true.

14.7 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* 327a⟩+≡ (283d) ◁321b

```
Mhead *m;
Path *path;
```

⟨namec() *case Aopen, Acreate, Aremove, Aaccess, handle mountpoint part1* 327b⟩≡ (283d)

```
/* 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() 328c, incref() 69b, pathclose() 301b, and uniquechan() 274b.

⟨namec() *case Aremove, Aaccess, handle mountpoint part2* 328a⟩≡ (283d)

```
  putmhead(m);
```

Uses putmhead() 318b.

⟨*function* findmount 328b⟩≡ (684)

```
  /* 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 320b, cclose, eqchantdqid() 53d, false, incref() 69b, print, putmhead() 318b, rlock() 65e, runlock() 66b, true, and up.

⟨*function* domount 328c⟩≡ (684)

```
  /*
   * 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, cclose, findmount() 328b, incref() 69b, print, and uniquepath() 301c.

14.8 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* 329a⟩≡ (300b)

```
// 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 329b⟩≡ (684)

```
PATHMSLOP   = 20,
```

⟨newpath() *mtpt handling* 329c⟩≡ (300c)

```
/*
 * 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 329b, print, and smalloc() 85c.

⟨copypath() *mtpt handling* 330a⟩≡ (301a)
```
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() 69b and smalloc() 85c.

⟨pathclose() *mtpt handling* 330b⟩≡ (301b)
```
for(i=0; i<p->mlen; i++)
    if(p->mtpt[i])
        cclose(p->mtpt[i]);
free(p->mtpt);
```
Uses cclose and free() 85b.

⟨addelem() *mtpt handling* 330c⟩≡ (302a)
```
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, PATHMSLOP-767 329b, cclose, incref() 69b, isdotdot() 302b, and smalloc() 85c.

⟨*function* undomount 330d⟩≡ (684)
```
/*
 * 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, cclose, and print.

14.9 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.

14.9.1 Union walk

⟨walk() *locals* 331a⟩+≡ (304b) ◁??

```
Mount *f;
```

⟨walk() *if c is a union mount, find and adjust type and dev* 331b⟩≡ (304b)

```
/* 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() 306a, rlock() 65e, and runlock() 66b.

14.9.2 ->umh

⟨namec() *case Aopen, Acreate, handle mountpoint part2* 331c⟩≡ (283d)

```
/* 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() 318b.

⟨Chan *other fields* 331d⟩+≡ (51e) ◁272c 332e▷

```
Mhead*  umh;      /* mount point that derived Chan; used in unionread */
```

⟨chanfree() *optional free* 332a⟩≡ (274a) 333a▷
```
if(c->umh != nil){
    putmhead(c->umh);
    c->umh = nil;
}
```
Uses putmhead() 318b.

⟨namec() *print error if c-¿umh != nil* 332b⟩≡ (283d)
```
if(c->umh != nil){
    print("uniquechan umh Open\n");
    putmhead(c->umh);
    c->umh = nil;
}
```
Uses print and putmhead() 318b.

⟨walk() *print error if c-¿umh != nil* 332c⟩≡ (304b)
```
if(c->umh != nil){  //BUG
    print("walk umh\n");
    putmhead(c->umh);
    c->umh = nil;
}
```
Uses print and putmhead() 318b.

⟨cmount() *print error if old-¿umh != nil* 332d⟩≡ (323e)
```
if(old->umh)
    print("cmount: unexpected umh, caller %#p\n", getcallerpc(&new));
```
Uses print.

14.9.3 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* 332e⟩+≡ (51e) ◁331d 332g▷
```
vlong devoffset;    /* in underlying device; see read */
```

⟨read() *if c is a QTDIR* 332f⟩≡ (291)
```
    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 631d, QTDIR 52c, mountfix() 344b, mountrockread() 343, and unionread() 333b.

⟨Chan *other fields* 332g⟩+≡ (51e) ◁332e 339a▷
```
Chan* umc;      /* channel in union; held for union read */
QLock umqlock;    /* serialize unionreads */
int uri;      /* union read index */
```

⟨chanfree() *optional free* 333a⟩+≡ (274a) ◁332a 342a▷
```
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 333b⟩≡ (690)
```
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 279c, cclone() 307b, cclose, poperror 633d, qlock() 63f, qunlock() 64c, rlock() 65e, runlock() 66b, and waserror 633d.

⟨read() *rewind when off == 0* 334a⟩≡ (291)
```
/*
 * 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() 344a and unionrewind() 334b.

⟨*function* unionrewind 334b⟩≡ (690)
```
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() 63f, and qunlock() 64c.

14.9.4 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* 334c⟩≡ (308d)
```
if(!sharppath && findmount(&cnew, &m, c->type, c->dev, c->qid))
    cnew = createdir(cnew, m);
```
Uses createdir() 334d and findmount() 328b.

⟨*function* createdir 334d⟩≡ (684)
```
/*
 * 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 631d, MCREATE 317b, cclone() 307b, cclose, nexterror, poperror 633d, rlock() 65e, runlock() 66b, and waserror 633d.

14.9.5 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* 335a⟩≡ (323b)

```
    if(c->umh != nil)
        putmhead(c->umh);
    c->umh = m;
```

Uses putmhead() 318b.

⟨cmount() *locals* 335b⟩+≡ (323e) ◁323d

```
int flg;
Mhead *umh, **l;
Mount *f, *um, **h;
```

⟨cmount() *if new is itself a mount point, error if cant create there* 335c⟩≡ (323e)

```
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 631d and MCREATE 317b.

⟨cmount() *if new is itself a mount point, copy mounts* 336a⟩≡ (323e)
```
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 317b, MREPL 317b, and newmount() 319a.

14.10 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* 336b⟩+≡ (319d) ◁320d
```
bool noattach;
```

⟨bindmount() *error if noattach* 336c⟩≡ (337f)
```
if(up->pgrp->noattach)
    error(Enoattach);
```
Uses Enoattach 631d and up.

⟨namec() *noattach checking* 336d⟩≡ (321c)
```
/*
 *  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 631d and up.

⟨sysrfork() *inherit noattach, RFPROC==0 case* 336e⟩≡ (121e)
```
/* inherit noattach */
up->pgrp->noattach = opg->noattach;
```
Uses up.

⟨sysrfork() *inherit noattach, RFPROC==1 case* 337a⟩≡ (119d)
```
/* inherit noattach */
p->pgrp->noattach = up->pgrp->noattach;
```
Uses up.

⟨sysrfork() *set noattach to true when RFNOMNT, RFPROC==0 case* 337b⟩≡ (121e)
```
if(flag & RFNOMNT)
    up->pgrp->noattach = true;
```
Uses RFNOMNT-2560 115b, true, and up.

⟨sysrfork() *set noattach to true when RFNOMNT, RFPROC==1 case* 337c⟩≡ (119d)
```
if(flag & RFNOMNT)
    p->pgrp->noattach = true;
```
Uses RFNOMNT-2560 115b and true.

14.11 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* 337d⟩≡ (690)
```
// 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() 322d and true.

⟨bindmount() *locals* 337e⟩+≡ (322d) ◁??
```
Chan *ac, *bc;

struct Bogus bogus;
```
Uses Bogus 682c.

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* 337f⟩≡ (322d)
```
if(ismount){
    validaddr((ulong)spec, 1, false);
    spec = validnamedup(spec, true);
    if(waserror()){
        free(spec);
        nexterror();
    }
    ⟨bindmount() error if noattach 336c⟩

    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 338b, ORDWR 279c, cclose, devno() 275b, false, fdtochan() 280c, free() 85b, nexterror, poperror 633d, true, validaddr() 101c, validnamedup() 285d, and waserror 633d.

⟨bindmount() *if ismount free* 338a⟩≡ (322d)

```
if(ismount){
    fdclose(fd, 0);
    poperror();
    free(spec);
}
```

Uses fdclose() 281b, free() 85b, and poperror 633d.

⟨MountFlags *cases* 338b⟩≡ (317b)

```
MCACHE = 0x0010,  /* cache some data */
```

14.12 sysumount()

\hlinkcunmount searches the \hlinkPgrp hash and removes either the entire mount point or a specific entry from the union list.

⟨*syscall unmount* 338c⟩≡ (690)

```
// 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 323a, Aopen 282e, OREAD 279c, cclose, cunmount() 339c, false, namec() 283d, nexterror, poperror 633d, validaddr() 101c, and waserror 633d.

⟨Chan *other fields* 339a⟩+≡ (51e) ◁332g 342c▷

```
    Chan* mchan;      /* channel to mounted server */
```

⟨cunmount() *print error if mnt-¿umh* 339b⟩≡ (339c)

```
    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 339c⟩≡ (684)

```
    void
    cunmount(Chan *mnt, Chan *mounted)
    {
        Pgrp *pg;
        Mhead *m, **l;
        Mount *f, **p;

        ⟨cunmount() print error if mnt-¿umh 339b⟩

        /*
         * 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` 631d, `Eunmount` 631d, `MOUNTH` 320b, `cclose`, `eqchan()` 53c, `mountfree()` 319c, `putmhead()` 318b, `true`, `up`, `wlock()` 67b, and `wunlock()` 67d.

14.13 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* 340a⟩≡ (318c) 340c▷

```
ulong mountid;
```

⟨*global* `mountid` 340b⟩≡ (710)

```
static Counter mountid;
```

⟨`Mount` *other fields* 340c⟩+≡ (318c) ◁340a

```
// list<ref<Mount>> sorted list by mount->mountid
Mount*  order;
```

⟨Mount *extra fields* 341a⟩+≡ (318c) ◁319b
```
Mount*  copy;
```

⟨*function* pgrpinsert 341b⟩≡ (710)
```
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 341c⟩≡ (710)
```
/*
 * 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 320b, lock() 58d, mountid-2606 340b, newmhead() 318a, newmount() 319a, pgrpinsert() 341b, rlock() 65e, runlock() 66b, unlock() 60d, wlock() 67b, and wunlock() 67d.

14.14 XTODO

⟨chanfree() *optional free* 342a⟩+≡ (274a) ◁333a 342b▷

```
if(c->mux != nil){
    muxclose(c->mux);
    c->mux = nil;
}
if(c->mchan != nil){
    cclose(c->mchan);
    c->mchan = nil;
}
```

Uses cclose and muxclose() 548d.

⟨chanfree() *optional free* 342b⟩+≡ (274a) ◁342a

```
if(c->dirrock != nil){
    free(c->dirrock);
    c->dirrock = nil;
    c->nrock = 0;
    c->mrock = 0;
}
```

Uses free() 85b.

⟨Chan *other fields* 342c⟩+≡ (51e) ◁339a 403b▷

```
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 342d⟩≡ (690)

```
/*
 * 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 34a, free() 85b, qlock() 63f, qunlock() 64c, and smalloc() 85c.

⟨*function* mountrockread 343⟩≡ (690)

```
/*
 * 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()` 63f, `qunlock()` 64c, and `true`.

⟨*function* `mountrewind` 344a⟩≡ (690)

```
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` 344b⟩≡ (690)

```
/*
 * 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() 313b, dirfixed() 346, eqchantdqid() 53d, findmount() 328b, free() 85b, mountrock() 342d, offsetof_name_direntry() 313c, poperror 633d, putmhead() 318b, smalloc() 85c, true, and waserror 633d.

⟨*function* dirfixed 346⟩≡ (690)

```
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() 275b 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 The keyboard

15.1.1 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:

```
hardware    interrupt          clock tick         process reads
  key  --> i8042intr --> kbdputsc --> kbdputc --> kbdputcclock --> echo
           scan code     Rune         UTF-8       |               |
           (1 byte)      (32-bit)     (1-4 bytes) |               |
                                      into        v               v
                                      istage[]    echoscreen()    qproduce(kbdq)
```

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.

15.1.2 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` 348a⟩≡ (648)

```
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` 348a, `Spec-960` 348a, and `View-962` 348a.

⟨*global* `kbtab` 348b⟩≡ (648)

```
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 348a, Ctrl-965 348a, KF-959 348a, No-972 348a, Nscan-985 348a, Num-968 348a, Scroll-984 348a, and Shift-963 348a.

⟨*global* kbtabshift 349a⟩≡ (648)

```
  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 348a, Ctrl-965 348a, KF-959 348a, No-972 348a, Nscan-985 348a, Num-968 348a, Scroll-984 348a, and Shift-963 348a.

⟨*global kbtabctrl decl* 349b⟩≡ (648)

```
  extern Rune kbtabctrl[];
```

⟨*global* kbtabaltgr 349c⟩≡ (648)

```
  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 348a, Break-964 348a, Ctrl-965 348a, Del-983 348a, Down-980 348a, End-979 348a, Home-973 348a, Ins-982 348a, Left-977 348a, No-972 348a, Nscan-985 348a, Pgdown-981 348a, Pgup-975 348a, Print-976 348a, Right-978 348a, Shift-963 348a, and Up-974 348a.

⟨*struct* Kbscan 350a⟩≡ (647d)
```
struct Kbscan {
    bool ctl;
    bool shift;
    bool caps;
    bool alt;
    bool altgr;
    bool num;
    ⟨Kbscan other fields 353a⟩
};
```

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 350b⟩≡ (647d)
```
/* kbscans indices */
enum KbscanKind {
    KbInt=    0,
    KbExt,

    KbNscans,
};
```

⟨*global* kbscans 350c⟩≡ (648)
```
// hash<enum<KbscanKind>, Kbscan>
Kbscan kbscans[KbNscans]; /* kernel and external scan code state */
```
Uses KbNscans 350b.

[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 350d⟩≡ (648)
```
/*
 * 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 608c⟩

    ⟨kbdputsc() esc key handling part1 and possible return 353b⟩
```

```
keyup = c & 0x80; // key released (1 bit)
c &= 0x7f; // 128

⟨kbdputsc() ensures c is in boundary ??⟩

⟨kbdputsc() esc key handling part2 and possible return 353c⟩
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 369b⟩
        ⟨kbdputsc() debugging up shift 608d⟩
        break;
    ⟨kbdputsc() mouse keyup cases 369d⟩
    }
    return;
}

/*
 *  normal character
 */
if(!(c & (Spec|KF))){
    ⟨kbdputsc() reboot if ctl-alt-del 594b⟩
    ⟨kbdputsc() if collecting 639d⟩
    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 639e⟩
        break;
```

```
        case Altgr:
            kbscan->altgr = true;
            break;
        case Shift:
            kbscan->shift = true;
            ⟨kbdputsc() set mouseshift 369c⟩
            ⟨kbdputsc() debugging down shift 608e⟩
            break;
        case Caps:
            kbscan->caps ^= true;
            break;
        case Num:
            kbscan->num ^= true;
            if(!external)
                arch_setleds(kbscan);
            break;
        ⟨kbdputsc() mouse keydown cases 369e⟩
        ⟨kbdputsc() special keyboard debug keys cases 608b⟩
        // e.g. Left, Right or other special key
        default:
            kbdputc(c);
            break;
        }
    }
}
```

Uses Alt-966 348a, Altgr-970 348a, Caps-967 348a, Ctrl-965 348a, KF-959 348a, KbExt 350b, KbInt 350b, Num-968 348a, Shift-963 348a, Spec-960 348a, false, kbdputc() 354b, kbscans 350c, kbtab 348b, kbtabaltgr 349c, kbtabctrl, kbtabshift 349a, and true.

⟨*function* arch_setleds*(arm)* 352a⟩≡ (736a)

```
// empty callback for portkbd.c
void arch_setleds(Kbscan *kbscan) { USED(kbscan); }
```

15.1.3 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 352b⟩≡ (648)

```
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 348a, Break-964 348a, Ctrl-965 348a, Del-983 348a, Down-980 348a, End-979 348a, Home-973 348a, Ins-982 348a, Left-977 348a, No-972 348a, Nscan-985 348a, Pgdown-981 348a, Pgup-975 348a, Print-976 348a, Right-978 348a, Shift-963 348a, and Up-974 348a.

⟨Kbscan *other fields* 353a⟩≡ (350a) 369a▷

```
  bool esc1;
  int esc2;
```

⟨kbdputsc() *esc key handling part1 and possible return* 353b⟩≡ (350d)

```
  /*
   *  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* 353c⟩≡ (350d)

```
  if(kbscan->esc1){
      c = kbtabesc1[c];
      kbscan->esc1 = false;
  } else if(kbscan->esc2){
      kbscan->esc2--;
      return;
  }
```

Uses false and kbtabesc1 352b.

15.1.4 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 353d⟩≡ (643)

```
  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 360f⟩
      // extra
      QLock;
  };
```

⟨*global* `kbd` 354a⟩≡ (643)

```
static struct ConsKbd kbd =
{
    .iw = kbd.istage,
    .ir = kbd.istage,
    .ie = kbd.istage + sizeof(kbd.istage),
};
```

Uses `ConsKbd` 353d and `kbd-1778` 354a.

[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` 354b⟩≡ (643)

```
/*
 *  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 609a⟩

    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` 639a, `ilock()` 61d, `iunlock()` 62b, and `kbd-1778` 354a.

[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* 354c⟩≡ (643)

```
/*
 *  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()` 355b and `kbd-1778` 354a.

⟨`consinit()` *initializing things* 355a⟩≡ (365a) 600c▷

```
/*
 * 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()` 231a and `kbdputcclock()` 354c.

15.1.5 `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` 355b⟩≡ (643)

```
static void
echo(char *buf, int n)
{
    ⟨echo() locals 609b⟩

    if(n == 0)
        return;

    ⟨echo() special keys handler 609c⟩

    qproduce(kbdq, buf, n); //!! add in kbd queue to be read from /dev/cons

    ⟨echo() return before any echoscreen if raw mode 368c⟩

    ⟨echo() hooks 356a⟩
}
```

Uses kbdq 360b and qproduce() 378.

⟨echo() *hooks* 356a⟩≡ (355b) 617c ▷
```
if(screenputs != nil)
    echoscreen(buf, n);
```
Uses echoscreen() 356c and screenputs 356b.

⟨*hook screenputs* 356b⟩≡ (643)
```
void    (*screenputs)(char*, int) = nil;
```
Uses screenputs 356b.

⟨*function* echoscreen 356c⟩≡ (643)
```
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 356b.

15.1.6 PC Keyboard (x86)

15.1.7 Raspberry Pi USB Keyboard (ARM)

15.1.8 Changing keyboard: /dev/kbmap

⟨*function* kbdgetmap 356d⟩≡ (648)
```
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 631d, Nscan-985 348a, kbtab 348b, kbtabaltgr 349c, kbtabctrl, kbtabesc1 352b, and kbtabshift 349a.

⟨*function* kbdputmap 357⟩≡ (648)

```
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 631d, Nscan-985 348a, kbtab 348b, kbtabaltgr 349c, kbtabctrl, kbtabesc1 352b, and kbtabshift 349a.

15.1.9 External keyboard input: /dev/kbin

15.2 The screen

15.2.1 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]).

15.2.2 Text output on a PC (x86)

15.2.3 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)* 358a⟩≡ (747)
```
#define VCBUFFER    (KZERO+0x3400)      /* videocore mailbox buffer */
```

⟨*function* `fbinit` *(arm)* 358b⟩≡ (729)
```
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) 498l⟩
    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` 725a, `FRAMEBUFFER` 32d, `VCBUFFER` 358a, `dmaaddr()` 831h, `fbdefault()` 359a, `mmukmap()` 358c, `vcread()` 726c, and `vcwrite()` 726b.

⟨*function* `mmukmap` *(arm)* 358c⟩≡ (748b)
```
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) 498e⟩
    return va + o;
}
```

Uses Dom0 161d, Fault 161c, Krw 161b, L1AP 161e, L1X-1842 160a, MiB 34g, Section 160d, and assert 693a.

⟨*function* fbdefault*(arm)* 359a⟩≡ (729)

```
/*
 * 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 725c, TagGetres-2375 725c, and vcreq() 726d.

⟨*function* fbblank*(arm)* 359b⟩≡ (729)

```
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 725c, and vcreq() 726d.

15.3 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.

15.3.1 Overview

⟨*devcons.c enum Qxxx* 359c⟩≡ (643)

```
enum{
```

```
    Qdir,

    Qcons,
    Qconsctl,

    ⟨devcons.c enum Qxxx cases 600a⟩
};
```

⟨*global* `consdir` 360a⟩≡ (643)
```
static Dirtab consdir[]={
    ".",    {Qdir, 0, QTDIR},   0,      DMDIR|0555,

    "cons",     {Qcons},    0,      0660,
    "consctl",  {Qconsctl}, 0,      0220,

    ⟨consdir fields 600b⟩
};
```
Uses `DMDIR` 313a, `QTDIR` 52c, `Qcons-1781` 359c, `Qconsctl-1782` 359c, and `Qdir-1780` 359c.

15.3.2 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` 360b⟩≡ (643)
```
Queue*  kbdq;           /* unprocessed console input */
```

⟨*function* `kbdqinit` 360c⟩≡ (643)
```
void
kbdqinit(void)
{
    kbdq = qopen(4*1024, 0, 0, 0);
    if(kbdq == nil)
        panic("kbdinit");
    qnoblock(kbdq, true);
}
```
Uses `kbdq` 360b, `qnoblock()` 391c, `qopen()` 380a, and `true`.

⟨*global* `lineq` 360d⟩≡ (643)
```
Queue*  lineq;          /* processed console input */
```

⟨*function* `lineqinit` 360e⟩≡ (643)
```
void
lineqinit(void)
{

    lineq = qopen(2*1024, 0, nil, nil);
    if(lineq == nil)
        panic("lineqinit");
    qnoblock(lineq, true);
}
```
Uses `lineq` 360d, `qnoblock()` 391c, `qopen()` 380a, and `true`.

⟨`ConsKbd` *other fields* 360f⟩≡ (353d) 368a▷
```
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* 361⟩≡ (366a)

```
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 368b⟩
        }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` 359c, `false`, `kbd-1778` 354a, `kbdq` 360b, `lineq` 360d, `nexterror`, `poperror` 633d, `qcanread()` 391a, `qlock()` 63f, `qread()` 384, `qunlock()` 64c, `qwrite()` 372b, `true`, and `waserror` 633d.

15.3.3 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* 362a⟩≡ (366b)

```
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` 359c and `putstrn0()` 362b.

⟨*function* `putstrn0` 362b⟩≡ (643)

```
/*
 *   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 617d⟩
    ⟨putstrn0() if kprint 619c⟩
    else if(screenputs != nil)
        screenputs(str, n);

    ⟨putstrn0() serialoq handling 621a⟩
}
```

Uses `false` and `screenputs` 356b.

15.3.4 `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 363a⟩≡ (704a)
```
PRINTSIZE = 256,
```

⟨*function* print 363b⟩≡ (643)
```
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 363a, noprint 363b, and putstrn() 363c.

⟨*function* putstrn 363c⟩≡ (643)
```
void
putstrn(char *str, int n)
{
    putstrn0(str, n, false);
}
```
Uses false and putstrn0() 362b.

iprint()

⟨*global* iprintscreenputs 363d⟩≡ (643)
```
bool iprintscreenputs = true;
```
Uses iprintscreenputs 363d and true.

⟨*function* iprint 363e⟩≡ (643)
```
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 363a, iprintcanlock() 364b, iprintlock-1779 364a, iprintscreenputs 363d, kmesgputs() 617e, screenputs 356b, uartputs(), and unlock() 60d.

⟨*global* iprintlock 364a⟩≡ (643)

```
/*
 * 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 364b⟩≡ (643)

```
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 645, arch_microdelay, canlock() 60f, false, and true.

pprint()

⟨*function* pprint 364c⟩≡ (643)

```
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 279c, OWRITE 279c, PRINTSIZE 363a, lock() 58d, poperror 633d, unlock() 60d, up, and waserror 633d.

15.3.5 /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* 365a⟩≡ (643)

```
static void
consinit(void)
{
    ⟨consinit() initializing things 355a⟩
}
```

⟨*method consopen* 365b⟩≡ (643)

```
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 368d⟩
    }
    return c;
}
```

Uses consdir-1797 360a, devgen() 399, and nelem 692c.

⟨*method consclose* 365c⟩≡ (643)

```
static void
consclose(Chan *c)
{
    switch((ulong)c->qid.path){
    ⟨consclose() cases 368e⟩
    }
}
```

⟨*method consread* 366a⟩≡ (643)

```
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 361⟩

    ⟨consread() cases 186a⟩


    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 631d, NUMSIZE 704a, Qdir-1780 359c, Qnull-1785 600a, Qpgrpid-1786 600a, Qpid-1787 600a, Qppid-1788 600a, Quser-1792 600a, Qzero-1793 600a, consdir-1797 360a, devdirread() 403c, devgen() 399, nelem 692c, print, readnum() 637c, readstr() 637d, and up.

⟨*method conswrite* 366b⟩≡ (643)

```
static long
conswrite(Chan *c, void *va, long n, vlong off)
{
    char *a;

    ⟨conswrite locals 367a⟩
```

```
        a = va;
        USED(off);

        switch((ulong)c->qid.path){

        ⟨conswrite() Qcons case 362a⟩

        ⟨conswrite() cases 196c⟩

        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 631d, Qnull-1785 600a, Quser-1792 600a, print, and userwrite() 558b.

⟨conswrite *locals* 367a⟩≡ (366b)

```
        char buf[256], ch;
        long l, bp;
        int fd;
        Chan *swc;
```

⟨*global* consdevtab 367b⟩≡ (643)

```
    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() 643, consclose() 365c, consinit() 365a, consopen() 365b, consread() 366a, consstat() 643, conswalk() 643, conswrite() 366b, devbread() 405a, devbwrite() 405b, devcreate() 686, devremove() 686, devreset() 686, devshutdown() 686, and devwstat() 686.

15.3.6 /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* 368a⟩+≡ (353d) ◁360f 594c▷
```
bool raw;        /* true if we shouldn't process input */
Ref ctl;        /* number of opens to the control file */
```

⟨consread() *else if raw mode* 368b⟩≡ (361)
```
}else if(kbd.raw){
    kbd.line[kbd.x++] = ch;
    send = !qcanread(kbdq);
```
Uses kbd-1778 354a, kbdq 360b, and qcanread() 391a.

⟨echo() *return before any echoscreen if raw mode* 368c⟩≡ (355b)
```
if(kbd.raw)
    return;
```
Uses kbd-1778 354a.

⟨consopen() *cases* 368d⟩≡ (365b) 618g▷
```
case Qconsctl:
    incref(&kbd.ctl);
    break;
```
Uses Qconsctl-1782 359c, incref() 69b, and kbd-1778 354a.

⟨consclose() *cases* 368e⟩≡ (365c) 619a▷
```
/* 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 272b, Qconsctl-1782 359c, decref() 69c, false, and kbd-1778 354a.

⟨conswrite() *cases* 368f⟩+≡ (366b) ◁243c
```
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 594d⟩
        if(a = strchr(a, ' '))
            a++;
    }
    break;
```
Uses Qconsctl-1782 359c, false, kbd-1778 354a, kbdq 360b, qwrite() 372b, and true.

15.4 The mouse

15.4.1 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* 369a⟩+≡ (350a) ◁353a 639c▷
```
int buttons;
```

⟨`kbdputsc()` *reset mouseshift* 369b⟩≡ (350d)
```
mouseshifted = false;
```
Uses `false` and `mouseshifted` 659.

⟨`kbdputsc()` *set mouseshift* 369c⟩≡ (350d)
```
mouseshifted = true;
```
Uses `mouseshifted` 659 and `true`.

⟨`kbdputsc()` *mouse keyup cases* 369d⟩≡ (350d)
```
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` 348a and `kbdmouse` 659.

⟨`kbdputsc()` *mouse keydown cases* 369e⟩≡ (350d)
```
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` 348a and `kbdmouse` 659.

15.4.2 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.

15.4.3 USB mouse (ARM)

15.5 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` 371a⟩≡ (682c)
```
/* 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` 371b⟩≡ (682c)
```
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];
  };
```

⟨*function* `pullupqueue` 372a⟩≡ (689b)

```
  /*
   *  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` 406b and `pullupblock()` 412.

[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` 372b⟩≡ (689b)

```
  /*
   *  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 689a, Qmsg 371a, allocb() 407b, freeb() 408b, nexterror, poperror 633d, qbwrite() 385b, setmalloctag() 87b, up, and waserror 633d.

\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 373⟩≡ (689b)

```
/*
 *  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 406c, BLEN 406b, Qflow 371a, Qstarve 371a, ilock() 61d, iunlock() 62b, and wakeup.

⟨*function* qdiscard 374⟩≡ (689b)

```
/*
 *  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 406c, BLEN 406b, Qflow 371a, freeb() 408b, ilock() 61d, iunlock() 62b, 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` 375⟩≡ (689b)

```
/*
 *  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 406c, BLEN 406b, Qflow 371a, Qmsg 371a, Qstarve 371a, consumecnt-774, freeblist() 409b, ilock() 61d, iunlock() 62b, 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 376⟩≡ (689b)

```
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 406c, BLEN 406b, Qclosed 371a, Qflow 371a, Qstarve 371a, freeblist() 409b, ilock() 61d, iunlock() 62b, and wakeup.

⟨*function* qpassnolim 377⟩≡ (689b)

```
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 406c, BLEN 406b, Qclosed 371a, Qflow 371a, Qstarve 371a, freeblist() 409b, ilock() 61d, iunlock() 62b, 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 378⟩≡ (689b)

```
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 406c, BLEN 406b, Qflow 371a, Qstarve 371a, iallocb() 408a, ilock() 61d, iunlock() 62b, producecnt-775, and wakeup.

⟨*function* qcopy 379⟩≡ (689b)

```
/*
 *  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 406b, allocb() 407b, ilock() 61d, iunlock() 62b, and qcopycnt-776.

[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 380a⟩≡ (689b)

```
/*
 *  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 371a and malloc() 85a.

⟨*function* qbypass 380b⟩≡ (689b)

```
/* 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() 85a.

⟨*function* notempty 380c⟩≡ (689b)

```
static int
notempty(void *a)
```

```
{
    Queue *q = a;

    return (q->state & Qclosed) || q->bfirst != 0;
}
```

Uses Qclosed 371a.

[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 381a⟩≡ (689b)

```
/*
 *  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 631d, Qclosed 371a, Qstarve 371a, ilock() 61d, iunlock() 62b, notempty() 380c, and sleep.

⟨*function* qaddlist 381b⟩≡ (689b)

```
/*
 * 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()` 411b and `blocklen()` 411a.

⟨*function* `qremove` 382a⟩≡ (689b)

```
/*
 *  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` 406c and `BLEN` 406b.

⟨*function* `qputback` 382b⟩≡ (689b)

```
/*
 *  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` 406c and `BLEN` 406b.

⟨*function* `qwakeup_iunlock` 382c⟩≡ (689b)

```
/*
 *  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` 371a, `iunlock()` 62b, 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` 383⟩≡ (689b)

```
/*
 *  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 406b, Qmsg 371a, allocb() 407b, ilock() 61d, iunlock() 62b, nexterror, poperror 633d, qlock() 63f, qputback() 382b, qremove() 382a, qunlock() 64c, qwait() 381a, qwakeup_iunlock() 382c, and waserror 633d.

⟨*function* qread 384⟩≡ (689b)

```
/*
 *  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 406b, Qcoalesce 371a, Qmsg 371a, bl2mem() 416a, freeb() 408b, ilock() 61d, iunlock() 62b, nexterror, poperror 633d, qlock() 63f, qputback() 382b, qremove() 382a, qunlock() 64c, qwait() 381a, qwakeup_iunlock() 382c, and waserror 633d.

⟨*function* qnotfull 385a⟩≡ (689b)

```
    static int
    qnotfull(void *a)
    {
        Queue *q = a;

        return q->len < q->limit || (q->state & Qclosed);
    }
```

Uses Qclosed 371a.

[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 385b⟩≡ (689b)

```
    /*
```

```
 *  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 406c, BLEN 406b, Qclosed 371a, Qflow 371a, Qkick 371a, Qstarve 371a, freeb() 408b, ilock() 61d, iunlock() 62b, nexterror, noblockcnt-777, poperror 633d, qlock() 63f, qnotfull() 385a, qunlock() 64c, sched, sleep, up, wakeup, and waserror 633d.

[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 387⟩≡ (689b)

```
/*
 *  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 406c, Maxatomic-779 689a, Qmsg 371a, Qstarve 371a, freeb() 408b, iallocb() 408a, ilock() 61d, iunlock() 62b, and wakeup.

⟨*function* qfree 389a⟩≡ (689b)
```
/*
 *  be extremely careful when calling this,
 *  as there is no reference accounting
 */
void
qfree(Queue *q)
{
    qclose(q);
    free(q);
}
```
Uses free() 85b and qclose() 389b.

[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 389b⟩≡ (689b)
```
/*
 *  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 631d, Qclosed 371a, Qflow 371a, Qstarve 371a, freeblist() 409b, ilock() 61d, iunlock() 62b, and wakeup.

⟨*function* qhangup 389c⟩≡ (689b)
```
/*
 *  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 692a, Ehungup 631d, Qclosed 371a, ilock() 61d, iunlock() 62b, and wakeup.

⟨*function* qiclosed 390a⟩≡ (689b)

```
/*
 *  return non-zero if the q is hungup
 */
int
qisclosed(Queue *q)
{
    return q->state & Qclosed;
}
```

Uses Qclosed 371a.

⟨*function* qreopen 390b⟩≡ (689b)

```
/*
 *  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 371a, Qstarve 371a, ilock() 61d, and iunlock() 62b.

⟨*function* qlen 390c⟩≡ (689b)

```
/*
 *  return bytes queued
 */
int
qlen(Queue *q)
{
    return q->dlen;
}
```

⟨*function* qwindow 390d⟩≡ (689b)

```
/*
 * 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 391a⟩≡ (689b)

```
/*
 *  return true if we can read without blocking
 */
int
qcanread(Queue *q)
{
    return q->bfirst!=0;
}
```

⟨*function* qsetlimit 391b⟩≡ (689b)

```
/*
 *  change queue limit
 */
void
qsetlimit(Queue *q, int limit)
{
    q->limit = limit;
}
```

⟨*function* qnoblock 391c⟩≡ (689b)

```
/*
 *  set blocking/nonblocking
 */
void
qnoblock(Queue *q, bool onoff)
{
    q->noblock = onoff;
}
```

⟨*function* qflush 391d⟩≡ (689b)

```
/*
 *  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()` 409b, `ilock()` 61d, `iunlock()` 62b, and `wakeup`.

⟨*function* `qfull` 392⟩≡ (689b)

```
int
qfull(Queue *q)
{
    return q->state & Qflow;
}
```

Uses `Qflow` 371a.

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` 393a⟩≡ (682c)
```
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` 393b⟩≡ (670b)
```
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() 405a, devbwrite() 405b, devcreate() 686, devinit() 686, devremove() 686, devshutdown() 686, devwstat() 686, rootattach() 670b, rootclose() 670a, rootopen() 670b, rootread() 396b, rootreset() 395c, rootstat() 670b, rootwalk() 670b, and rootwrite() 670b.

⟨*devroot enum Qxxx* 394a⟩≡ (670b)

```
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` 394b⟩≡ (670b)

```
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* 394c⟩≡ (670b)

```
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 313a, Nrootfiles-828 394a, QTDIR 52c, Qboot-827 394a, Qdir-826 394a, rootdata-831 394c, and rootdir-830 394c.

⟨*function* addlist 395a⟩≡ (670b)

```
/*
 *  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 313a and QTDIR 52c.

⟨*function* addrootdir 395b⟩≡ (670b)

```
/*
 *  add a root directory
 */
static void
addrootdir(char *name)
{
    addlist(&rootlist, name, nil, 0, DMDIR|0555);
}
```

Uses DMDIR 313a, addlist() 395a, and rootlist-832 394c.

[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* 395c⟩≡ (670b)

```
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() 395b.

[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` 396a⟩≡ (670b)

```
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`, `QTDIR` 52c, `Qboot-827` 394a, `Qdir-826` 394a, `bootlist-835` 397, `devdir()` 398c, `devgen()` 399, `eve` 557a, and `rootlist-832` 394c.

[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* 396b⟩≡ (670b)

```
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 631d, Qboot-827 394a, Qdir-826 394a, bootlist-835 397, devdirread() 403c, rootgen() 396a, and rootlist-832 394c.

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* 397⟩≡ (670b)

```
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 313a, Nbootfiles-829 394a, QTDIR 52c, Qboot-827 394a, bootdata-834 397, and bootdir-833 397.

⟨*function* addbootfile 398a⟩≡ (670b)

```
/*
 *  add a boot file
 */
void
addbootfile(char *name, byte *contents, ulong len)
{
    addlist(&bootlist, name, contents, len, 0555);
}
```

Uses addlist() 395a and bootlist-835 397.

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* 398b⟩≡ (52c) 416b ▷

```
QTMOUNT = 0x10,    /* type bit for mounted channel */
```

⟨*function* devdir 398c⟩≡ (686)

```
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 549b, QTMOUNT 398b, eve 557a, and seconds() 241a.

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` 399⟩≡ (686)

```
/*
 * (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, devdir() 398c, and eve 557a.

[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 400a⟩≡ (686)

```
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 272b and newchan() 273d.

[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 400b⟩≡ (686)

```
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, ERRMAX 692a, QTDIR 52c, cclose, devclone() 400a, devgen() 399, error_if_not_dir() 299b, false, free() 85b, kstrcpy() 636a, poperror 633d, print, smalloc() 85c, true, up, and waserror 633d.

[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 402⟩≡ (686)

```
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 549b, DMDIR 313a, DMMOUNT 403a, Ebadarg 631d, QTDIR 52c, devdir() 398c, and eve 557a.

⟨Dirmode *cases* 403a⟩≡ (313a) 618d▷

```
DMMOUNT = 0x10000000,  /* mode bit for mounted channel */
```

⟨Chan *other fields* 403b⟩+≡ (51e) ◁342c ??▷

```
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 403c⟩≡ (686)

```
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 631d.

[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 403d⟩≡ (686)

```
/*
 * 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 557a 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 404a⟩≡ (686)

```
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 272b, OREAD 279c, QTDIR 52c, devpermcheck() 403d, and openmode() 280a.

⟨Dev *methods* 404b⟩+≡ (54a) ◁322a 595b▷

```
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` 405a⟩≡ (686)
```
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()` 407b, `freeb()` 408b, `nexterror`, `poperror` 633d, and `waserror` 633d.

⟨*function* `devbrwrite` 405b⟩≡ (686)
```
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` 406b, `freeb()` 408b, `nexterror`, `poperror` 633d, and `waserror` 633d.

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` 405c⟩≡ (682c)
```
/* 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 406a⟩≡ (682c)
```
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: The region before `rp` is header space (`Hdrspc` bytes). Protocol layers prepend headers by decrementing `rp`, avoiding copies. `BLEN` gives the number of data bytes; `BALLOC` gives the total allocated size.

⟨*function* BLEN 406b⟩≡ (682c)
```
#define BLEN(s) ((s)->wp - (s)->rp)
```

⟨*function* BALLOC 406c⟩≡ (682c)
```
#define BALLOC(s) ((s)->lim - (s)->base)
```

⟨*constant* BLOCKALIGN*(arm)* 406d⟩≡ (747)
```
// 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 406e⟩≡ (687)
```
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-780 687, ROUND 34a, false, mallocz() 86b, and msize() 407a.

⟨*function* msize 407a⟩≡ (698b)

```
ulong
msize(void *v)
{
    return poolmsize(mainmem, (ulong*)v-Npadlong)-Npadlong*sizeof(ulong);
}
```

Uses Npadlong-1813 698b and mainmem 84a.

[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 407b⟩≡ (687)

```
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() 406e, arch_delay, mallocsummary() 610b, setmalloctag() 87b, up, and xsummary() 610a.

⟨*struct* Ialloc 407c⟩≡ (687)

```
struct Ialloc
{
    Lock;
    ulong   bytes;
};
```

⟨*global* ialloc 407d⟩≡ (687)

```
struct Ialloc ialloc;
```

Uses Ialloc 407c.

⟨*function* iallocb 408a⟩≡ (687)

```
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` 405c, `_allocb()` 406e, `active` 49e, `arch_exit`, `conf` 46b, `ialloc` 407d, `ilock()` 61d, `iprint`, `iunlock()` 62b, `setmalloctag()` 87b, 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 408b⟩≡ (687)

```
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 405c, Bdead-781 687, arch_dumpstack, free() 85b, ialloc 407d, ilock() 61d, and iunlock() 62b.

⟨*function* checkb 409a⟩≡ (687)

```
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-781 687 and print.

⟨*function* freeblist 409b⟩≡ (689b)

```
/*
 *  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()` 408b.

[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` 410⟩≡ (689b)

```
/*
 *  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 406b, allocb() 407b, freeb() 408b, and padblockcnt-770.

⟨*function* blocklen 411a⟩≡ (689b)

```
    /*
     *  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 406b.

⟨*function* blockalloclen 411b⟩≡ (689b)

```
    /*
     * 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 406c.

⟨*function* concatblock 411c⟩≡ (689b)

```
    /*
     *  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 406b, allocb() 407b, blocklen() 411a, concatblockcnt-771, and freeblist() 409b.

[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 412⟩≡ (689b)

```
/*
 *  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 406b, allocb() 407b, freeb() 408b, print, and pullupblockcnt-772.

⟨*function* packblock 413a⟩≡ (689b)

```
/*
 *  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 406c, BLEN 406b, allocb() 407b, and freeb() 408b.

⟨*function* trimblock 413b⟩≡ (689b)

```
/*
 *  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 406b, blocklen() 411a, freeb() 408b, and freeblist() 409b.

⟨*function* copyblock 414⟩≡ (689b)

```
/*
 *  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 406b, allocb() 407b, and copyblockcnt-773.

⟨*function* `adjustblock` 415a⟩≡ (689b)

```
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` 406b, `copyblock()` 414, `freeb()` 408b, and `freeblist()` 409b.

⟨*function* `pullblock` 415b⟩≡ (689b)

```
/*
 *  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 406b and freeb() 408b.

⟨*function* bl2mem 416a⟩≡ (689b)
```
/*
 *  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 406b and freeb() 408b.

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* 416b⟩+≡ (52c) ◁398b
```
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* 418a⟩≡ (41f)
```
[NOTIFY]    sysnotify,
[NOTED]     sysnoted,

[PIPE]      syspipe,

[SEGATTACH] syssegattach,
[SEGDETACH] syssegdetach,
[SEGFREE]   syssegfree,
[SEGFLUSH]  syssegflush,
[SEGBRK]    syssegbrk,
```
Uses `sysnoted()` 422, `sysnotify()` 420f, `syspipe()` 426a, `syssegattach()` 436c, `syssegbrk()` 436b, `syssegdetach()` 438b, `syssegflush()` 441c, and `syssegfree()` 439.

⟨`systab` *concurrency syscalls* 418b⟩≡ (41f)
```
[RENDEZVOUS]    sysrendezvous,

[SEMACQUIRE]    syssemacquire,
[SEMRELEASE]    syssemrelease,
[TSEMACQUIRE]   systsemacquire,
```
Uses `sysrendezvous()` 448f, `syssemacquire()` 446a, `syssemrelease()` 447a, and `systsemacquire()` 446b.

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` 418c⟩≡ (705c)
```
enum NoteKind
{
    NUser,      /* note provided externally */
    NExit,      /* deliver note quietly */
    NDebug,     /* print debug message */
};
```

⟨*enum* `note` 419a⟩≡ (693c)
```
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` 419b⟩≡ (712c)
```
// 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` 419c⟩≡ (705c)
```
// a kind of Unix signal
struct Note
{
    char  msg[ERRMAX];
    // enum<NoteKind>
    int flag;     /* whether system posted it */
};
```

⟨`newproc()` *set fields of* `p` 419d⟩+≡ (111d) ◁218d 419e▷
```
p->noteid = incref(&noteidalloc);
if(p->noteid==0)
    panic("pidalloc");
```
Uses `incref()` 69b and `noteidalloc` 419b.

⟨`newproc()` *set fields of* `p` 419e⟩+≡ (111d) ◁419d 420a▷
```
p->notepending = false;
p->notified = false;
```
Uses `false`.

⟨`sysrfork()` *when* `RFPROC` *propagate fields from up to p* 419f⟩+≡ (116b) ◁205c 420b▷
```
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* 419g⟩≡ (212e)
```
if(up->notepending) {
    up->notepending = false;
    arch_splx(s);
    ⟨sleep() forceclosefgrp 295b⟩
    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* 419h⟩≡ (49f)
```
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 420a⟩+≡ (111d) ◁419e 480c▷
```
  p->ureg = nil;
```

⟨sysrfork() *when* RFPROC *propagate fields from up to p* 420b⟩+≡ (116b) ◁419f 455f▷
```
  p->ureg = up->ureg; // will be adjusted for R0??
```
Uses up.

⟨sysexec() *when hold debug lock, reset some fields* 420c⟩≡ (123b) 567h▷
```
  up->nnote = 0;
  up->notify = nil;
  up->notified = false;
```

⟨*constant* NNOTE 420d⟩≡ (705c)
```
  NNOTE = 5,
```

⟨trap() *call possibly notify(arm)* 420e⟩≡ (150b)
```
  if(up->procctl || up->nnote)
      arch__notify(ureg);
```
Uses arch__notify() 421b and up.

18.1.1 sysnotify() (arm)

⟨*syscall notify* 420f⟩≡ (716a)
```
  // 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() 101c.

⟨syscall() *call notify(arm)* 420g⟩≡ (145e)
```
  arch_splhi();
  if(scallnr != RFORK && (up->procctl || up->nnote))
      arch__notify(ureg);
```
Uses arch__notify() 421b 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()`.

⟨*struct* `NFrame` *(arm)* 421a⟩≡ (757b)
```
struct NFrame {
    uintptr ip;
    Ureg*   arg0;
    char*   arg1;
    char    msg[ERRMAX];
    Ureg*   old;
    Ureg    ureg;
};
```
Uses `ERRMAX` 692a.

`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)* 421b⟩≡ (757b)
```
/*
 *  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 692a, NDebug 418c, NUser 418c, PTR2UINT 41a, UINT2PTR 41b, fpunotify() 484c, okaddr() 101a, pexit, pprint, procctl() 453d, qlock() 63f, qunlock() 64c, and up.

18.1.2 sysnoted() (arm)

⟨*syscall noted* 422⟩≡ (716a)

```
// int noted(int v);
long
sysnoted(ulong* arg)
{
    if(arg[0]!=NRSTR && !up->notified)
        error(Egreg);
    return 0;
}
```

Uses Egreg 631d, NRSTR 419a, and up.

⟨syscall() *call noted()* 423a⟩≡ (145e)

```
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)* 423b⟩≡ (757b)

```
/*
 *   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 419a, NDFLT 419a, NDebug 418c, NRSTR 419a, NSAVE 419a, PTR2UINT 41a, PsrDfiq 527d, PsrDirq 28b, PsrMask 28a, fpunoted() 485a, okaddr() 101a, pexit, pprint, qlock() 63f, qunlock() 64c, 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 424⟩≡ (712c)

```
/*
 *  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 420d, NUser 418c, REND 447b, Rendezvous 448e, Wakeme 212d, canlock() 60f, lock() 58d, qlock() 63f, qunlock() 64c, ready, sched, true, and unlock() 60d.

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. [syspipe] creates a pipe by attaching to the pipe device (#|), cloning the channel, then walking each clone to data and data1.

⟨*syscall pipe* 426a⟩≡ (690)

```
// 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 282e, Egreg 631d, Enofd 631d, ORDWR 279c, cclone() 307b, cclose, devno() 275b, false, namec() 283d, newfd2() 426b, nexterror, poperror 633d, true, validaddr() 101c, walk() 304b, and waserror 633d.

⟨*function* newfd2 426b⟩≡ (690)

```
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()` 278a, `lock()` 58d, `unlockfgrp()` 279b, and `up`.

⟨`sseek()` *and pipes* 427a⟩≡ (293b)

```
    if(devtab[c->type]->dc == '|')
        error(Eisstream);
```

Uses `Eisstream` 631d.

18.2.2 /dev/pipe

The `Pipe` structure holds two `Queue`s 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` 427b⟩≡ (679)

```
    struct Pipe
    {
        int ref;
        ulong   path;
        long    perm;
        Queue   *q[2];
        int qref[2];

        // extra
        QLock;
        Pipe    *next;
    };
```

⟨*struct* `Pipealloc` 427c⟩≡ (679)

```
    struct Pipealloc
    {
        ulong   path;

        // extra
        Lock;
    };
```

⟨*global* `pipealloc` 427d⟩≡ (679)

```
    struct Pipealloc pipealloc;
```

Uses `Pipealloc` 427c.

⟨*devpipe.c enum Qxxx* 428a⟩≡ (679)
```
enum
{
    Qdir,
    Qdata0,
    Qdata1,
};
```

⟨*global* `pipedir` 428b⟩≡ (679)
```
Dirtab pipedir[] =
{
    ".",        {Qdir,0,QTDIR}, 0,      DMDIR|0500,
    "data",     {Qdata0},   0,      0600,
    "data1",    {Qdata1},   0,      0600,
};
```
Uses `DMDIR` 313a, `QTDIR` 52c, `Qdata0-856` 428a, `Qdata1-857` 428a, and `Qdir-855` 428a.

⟨`Conf` *other fields* 428c⟩+≡ (46a) ◁254b
```
ulong pipeqsize;  /* size in bytes of pipe queues */
```

⟨*method pipeinit* 428d⟩≡ (679)
```
static void
pipeinit(void)
{
    if(conf.pipeqsize == 0){
        if(conf.ncpu > 1)
            conf.pipeqsize = 256*1024;
        else
            conf.pipeqsize = 32*1024;
    }
}
```
Uses `conf` 46b.

⟨`Chan` *union other fields* 428e⟩≡ (51e) 570c▷
```
/*Pipe*/void* chanpipe; // for pipes
```

⟨*method pipeattach* 428f⟩≡ (679)
```
/*
 *  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 52c, Qdata0-856 428a, Qdir-855 428a, conf 46b, devattach() 275a, exhausted() 633c, free() 85b, lock() 58d, malloc() 85a, pipealloc 427d, pipedir 428b, qopen() 380a, and unlock() 60d.

[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 429⟩≡ (679)

```
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, DMDIR 313a, NETID, NETQID, QTFILE 52c, Qdata0-856 428a, Qdata1-857 428a, devdir() 398c, eve 557a, and qlen() 390c.

[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* 430a⟩≡ (679)

```
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 272b, NETTYPE, NPIPEDIR-858 679, Qdata0-856 428a, Qdata1-857 428a, devwalk() 400b, pipedir 428b, pipegen() 429, print, qlock() 63f, and qunlock() 64c.

⟨*method pipestat* 430b⟩≡ (679)

```
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 313a, Eshortstat 631d, NETTYPE, Qdata0-856 428a, Qdata1-857 428a, Qdir-855 428a, devdir() 398c, eve 557a, and qlen() 390c.

[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* 431a⟩≡ (679)

```
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` 631d, `NETTYPE`, `Qdir-855` 428a, `emptystr()` 637a, `eve` 557a, `free()` 85b, `nexterror`, `poperror` 633d, `smalloc()` 85c, `up`, and `waserror` 633d.

[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* 431b⟩≡ (679)

```
/*
 *  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 272b, Ebadarg 631d, NETTYPE, OREAD 279c, QTDIR 52c, Qdata0-856 428a, Qdata1-857 428a, openmode() 280a, qiomaxatomic 689a, qlock() 63f, and qunlock() 64c.

[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* 432⟩≡ (679)

```
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 272b, NETTYPE, Qdata0-856 428a, Qdata1-857 428a, free() 85b, qclose() 389b, qhangup() 389c, qlock() 63f, qreopen() 390b, and qunlock() 64c.

⟨*method piperead* 433a⟩≡ (679)

```
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-858 679, Qdata0-856 428a, Qdata1-857 428a, Qdir-855 428a, devdirread() 403c, pipedir 428b, pipegen() 429, and qread() 384.

⟨*method pipebread* 433b⟩≡ (679)

```
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 428a, Qdata1-857 428a, devbread() 405a, and qbread() 383.

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* 434a⟩≡ (679)
```
/*
 *  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` 549b, `NETTYPE`, `NUser` 418c, `Qdata0-856` 428a, `Qdata1-857` 428a, `nexterror`, `poperror` 633d, `postnote`, `print`, `qwrite()` 372b, `up`, and `waserror` 633d.

⟨*method pipebwrite* 434b⟩≡ (679)
```
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 549b, NETTYPE, NUser 418c, Qdata0-856 428a, Qdata1-857 428a, nexterror, poperror 633d, postnote, qbwrite() 385b, up, and waserror 633d.

⟨*global* pipedevtab 435a⟩≡ (679)

```
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() 686, devremove() 686, devreset() 686, devshutdown() 686, pipeattach() 428f, pipebread() 433b, pipebwrite() 434b, pipeclose() 432, pipeinit() 428d, pipeopen() 431b, piperead() 433a, pipestat() 430b, pipewalk() 430a, pipewrite() 434a, and pipewstat() 431a.

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* 435b⟩+≡ (94a) ◁101b

```
SG_CEXEC  = 0100,   /* Detach at exec */
```

⟨sysexec() *free special segments if close on exec* 436a⟩≡ (125c)
```
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* 436b⟩≡ (716a)
```
// 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 631d, NSEG 100a, SG_DATA 523b, SG_STACK 94a, SG_TEXT, SG_TYPE, ibrk() 104d, and up.

⟨*syscall segattach* 436c⟩≡ (716a)
```
// 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() 437 and up.

⟨*hook* _globalsegattach 436d⟩≡ (702)
```
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 437⟩≡ (702)

```
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 100a, Ebadarg 631d, Enovmem 631d, Esoverlap 631d, NSEG 100a, PGROUND 102a, SG_TYPE, SSEG 100a, _globalsegattach 436d, false, isoverlap() 438a, newseg() 97b, physseg-1804 523f, validaddr() 101c, and vmemchr() 101d.

⟨*function* isoverlap 438a⟩≡ (702)

```
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 100a.

[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* 438b⟩≡ (716a)

```
// 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 631d, NSEG 100a, SSEG 100a, nexterror, poperror 633d, putseg() 98b, qlock() 63f, qunlock() 64c, up, and waserror 633d.

`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* 439⟩≡ (716a)

```
// 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 631d, PGROUND 102a, mfreeseg() 107, qunlock() 64c, seg() 100c, and up.

⟨dupseg() *if original segment was shared, flush it* 440a⟩≡ (103e)

```
if(s->ref > 1)
    procflushseg(s); // ??
```

Uses procflushseg() 440c.

⟨mfreeseg() *if segment was shared, flush it* 440b⟩≡ (107)

```
/* flush this seg in all other processes */
if(s->ref > 1)
    procflushseg(s);
```

Uses procflushseg() 440c.

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 440c⟩≡ (712c)

```
/*
 *  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 645, Dead 51b, NSEG 100a, conf 46b, false, procalloc-2458 110c, sched, and true.

⟨hzclock() *if flushmmu* 441a⟩≡ (215f)

```
    if(cpu->flushmmu){
        if(up)
            arch_flushmmu();
        cpu->flushmmu = false;
    }
```

Uses false and up.

⟨Cpu *other fields* 441b⟩+≡ (48a) ◁215a 475b▷

```
    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* 441c⟩≡ (703b)

```
    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 522k⟩
        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 631d, ROUND 34a, ptflush() 703b, qunlock() 64c, seg() 100c, 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 442a⟩≡ (640)

```
// 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* 442b⟩+≡ (94c) ◁176b 456e▷

```
Sema sema;
```

⟨newseg() *sema initialization* 442c⟩≡ (97b)

```
    // no list, just one sema
    s->sema.prev = &s->sema;
    s->sema.next = &s->sema;
```

cmpswap() (arm)

⟨*function* arch_cmpswap*(raspberry pi1)(arm)* 443a⟩≡ (731b)
```
bool
arch_cmpswap(long *addr, long old, long new)
{
    return cas32(addr, old, new);
}
```
Uses cas32() 443b.

⟨*function* cas32*(arm)* 443b⟩≡ (731b)
```
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) 501g⟩
    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 443c⟩≡ (716b)
```
/* 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() 58d and unlock() 60d.

⟨*function* semdequeue 443d⟩≡ (716b)
```
/* 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() 58d and unlock() 60d.

⟨*function* semwakeup 444a⟩≡ (716b)

```
/* 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() 58d, unlock() 60d, and wakeup.

⟨*function* semrelease 444b⟩≡ (716b)

```
/* 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() 444a.

⟨*function* canacquire 444c⟩≡ (716b)

```
/* 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 444d⟩≡ (716b)

```
/* 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` 445a⟩≡ (716b)

```
/* 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()` 444c, `false`, `nexterror`, `poperror` 633d, `semawoke()` 444d, `semdequeue()` 443d, `semqueue()` 443c, `semwakeup()` 444a, `sleep`, `true`, and `waserror` 633d.

[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` 445b⟩≡ (716b)

```
/* 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 223a, arch_coherence, canacquire() 444c, false, nexterror, poperror 633d, semawoke() 444d, semdequeue() 443d, semqueue() 443c, semwakeup() 444a, true, tsleep, and waserror 633d.

⟨*syscall semacquire* 446a⟩≡ (716b)

```
// 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 631d, seg() 100c, semacquire() 445a, true, up, and validaddr() 101c.

⟨*syscall tsemacquire* 446b⟩≡ (716b)

```
// 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 631d, seg() 100c, true, tsemacquire() 445b, up, and validaddr() 101c.

⟨*syscall semrelease* 447a⟩≡ (716b)

```
// 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 631d, seg() 100c, semrelease() 444b, true, up, and validaddr() 101c.

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 447b⟩≡ (705c)

```
enum
{
    RENDLOG = 5,
    RENDHASH =  1<<RENDLOG, /* Hash to lookup rendezvous tags */
};
#define REND(p,s) ((p)->rendhash[(s)&((1<<RENDLOG)-1)])
```

Uses RENDLOG 447b.

⟨*struct* `Rgrp` 448a⟩≡ (705c)
```
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* 448b⟩+≡ (49f) ◁232b
```
Rgrp  *rgrp;    /* Rendez group */

uintptr rendtag;  /* Tag for rendezvous */
uintptr rendval;  /* Value for rendezvous */
//??
Proc  *rendhash;  /* Hash list for tag values */
```

⟨*constructor* `newrgrp` 448c⟩≡ (710)
```
Rgrp*
newrgrp(void)
{
    Rgrp *r;

    r = smalloc(sizeof(Rgrp));
    r->ref = 1;
    return r;
}
```
Uses `smalloc()` 85c.

⟨*destructor* `closergrp` 448d⟩≡ (710)
```
void
closergrp(Rgrp *r)
{
    if(decref(r) == 0)
        free(r);
}
```
Uses `decref()` 69c and `free()` 85b.

⟨`Procstate` *cases* 448e⟩+≡ (51b) ◁212d 451a▷
```
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* 448f⟩≡ (716a)
```
// 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 447b, Rendezvous 448e, lock() 58d, ready, sched, unlock() 60d, 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/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`.

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)* 450a⟩≡ (152e)
```
if(seg(up, ureg->pc, false) != nil &&
   *(u32int*)ureg->pc == 0xD1200070)
    postnote(up, 1, "sys: breakpoint", NDebug);
```
Uses `NDebug` 418c, `false`, `postnote`, `seg()` 100c, and `up`.

⟨`trap()` *when prefetch fault, switch fst cases(arm)* 450b⟩≡ (153c)
```
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` 418c, `iprint`, `postnote`, and `up`.

⟨*function* `arch_setregisters`*(arm)* 450c⟩≡ (754e)
```
/* 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* 451a⟩+≡ (51b) ◁448e 453a▷
```
Broken,
```

⟨*struct* `Broken` 451b⟩≡ (712c)
```
/*
 * 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` 451b.

⟨*global* `broken` 451c⟩≡ (712c)
```
struct Broken broken;
```
Uses `Broken` 451a 451b.

⟨`pexit()` *if not freemem, add process to broken processes list* 451d⟩≡ (134c)
```
// instead of a core dump we just keep the process around
if(!freemem)
    addbroken(up); // will call sched()
```
Uses `addbroken()` 451e 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` 451e⟩≡ (712c)
```
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 451a 451b, NBROKEN-2462 451b, broken 451c, edfstop() 535b, qlock() 63f, qunlock() 64c, ready, sched, and up.

⟨*function* unbreak 452a⟩≡ (712c)

```
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 451b, broken 451c, qlock() 63f, qunlock() 64c, and ready.

⟨*function* freebroken 452b⟩≡ (712c)

```
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 451c, qlock() 63f, qunlock() 64c, and ready.

⟨procctlreq() *CMkill case, Broken case* 452c⟩≡ (141c)

```
        case Broken:
            // will resume the broken process to finally die and free its mem
            unbreak(p);
            break;
```

Uses Broken 451a 451b and unbreak() 452a.

19.2 Stopping a process: /proc/x/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* 453a⟩+≡ (51b) ◁451a 535a▷
```
Stopped,
```

⟨*enum* `procctl` 453b⟩≡ (705c)
```
enum Procctl
{
    Proc_nothing = 0,
    Proc_stopme,
    ⟨Procctl cases 141a⟩
};
```

⟨`Proc` *debugger fields* 453c⟩+≡ (49f) ◁147e 454b▷
```
// 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` 453d⟩≡ (712c)
```
/*
 *  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) 470b⟩
    case Proc_stopme:
        p->procctl = Proc_nothing;
        oldstate = p->psstate;
        p->psstate = "Stopped";
        s = arch_spllo();
        ⟨procctl() wakeup waiting debugger 455b⟩
        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 711b⟩
    ⟨procctl() Proc_exitme case 141b⟩
    }
}
```
Uses `Proc_nothing` 453b, `Proc_stopme` 453b, `Stopped` 453a, and `sched`.

⟨`procctlreq()` *CMstop case* 453e⟩≡ (586b)
```
case CMstop:
    procstopwait(p, Proc_stopme);
    break;
```
Uses `CMstop-893` 568b, `Proc_stopme` 453b, and `procstopwait()` 454e.

⟨procctlreq() *CMstart case* 454a⟩≡ (586b)
```
case CMstart:
    if(p->state != Stopped)
        error(Ebadctl);
    ready(p);
    break;
```
Uses CMstart-890 568b, Stopped 453a, and ready.

⟨Proc *debugger fields* 454b⟩+≡ (49f) ◁453c 454d▷
```
Proc  *pdbg;    /* the debugging process */
```

⟨pexit() *release debuggers* 454c⟩≡ (134c)
```
/* release debuggers */
qlock(&up->debug);
if(up->pdbg) {
    wakeup(&up->pdbg->sleepr);
    up->pdbg = nil;
}
qunlock(&up->debug);
```
Uses qlock() 63f, qunlock() 64c, up, and wakeup.

⟨Proc *debugger fields* 454d⟩+≡ (49f) ◁454b 455c▷
```
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 454e⟩≡ (680)
```
// 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 451a 451b, Eprocdied 631d, Proc_nothing 453b, nexterror, poperror 633d, procstopped() 455a, qlock() 63f, qunlock() 64c, sleep, up, and waserror 633d.

⟨*function* procstopped 455a⟩≡ (680)

```
bool
procstopped(void *a)
{
    Proc *p = a;
    return p->state == Stopped;
}
```

Uses Stopped 453a.

⟨procctl() *wakeup waiting debugger* 455b⟩≡ (453d)

```
/* free a waiting debugger */
qlock(&p->debug);
if(p->pdbg) {
    wakeup(&p->pdbg->sleepr);
    p->pdbg = nil;
}
qunlock(&p->debug);
```

Uses qlock() 63f, qunlock() 64c, 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* 455c⟩+≡ (49f) ◁454d 460b▷

```
bool hang;   /* hang at next exec for debug */
```

⟨procctlreq() *CMhang case* 455d⟩≡ (586b)

```
case CMhang:
    p->hang = true;
    break;
```

Uses CMhang-881 568b and true.

⟨procctlreq() *CMnohang case* 455e⟩≡ (586b)

```
case CMnohang:
    p->hang = false;
    break;
```

Uses CMnohang-883 568b and false.

⟨sysrfork() *when* RFPROC *propagate fields from up to p* 455f⟩+≡ (116b) ◁420b 481b▷

```
p->hang = up->hang;
```

Uses up.

⟨sysexec() *if hang* 455g⟩≡ (123b)

```
if(up->hang)
    up->procctl = Proc_stopme;
```

⟨procctlreq() *CMwaitstop case* 455h⟩≡ (586b)

```
case CMwaitstop:
    procstopwait(p, Proc_nothing);
    break;
```

Uses CMwaitstop-894 568b, Proc_nothing 453b, and procstopwait() 454e.

19.3 Inspecting process memory: /proc/x/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* 456a⟩+≡ (571d) ◁325e 460d▷
```
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 631d, Qmem-866 568a, conf 46b, iseve() 557b, nelem 692c, procctlmemio() 457, and true.

⟨xinit() *nkpages kernel memory in m* 456b⟩≡ (260)
```
m->kbase = (ulong)KADDR(m->base);
m->klimit = (ulong)KADDR(m->base+nkpages*BY2PG);
```

⟨Confmem *other fields* 456c⟩≡ (46c)
```
kern_addr kbase; // KADDR(Confmem.base)
kern_addr klimit; // KADDR(base+ x*BY2PG) where x <= Confmem.npage
```

⟨procwrite() *cases* 456d⟩≡ (574c)
```
case Qmem:
    if(p->state != Stopped)
        error(Ebadctl);

    n = procctlmemio(p, offset, n, va, false);
    break;
```
Uses Qmem-866 568a, Stopped 453a, false, and procctlmemio() 457.

⟨Segment *other fields* 456e⟩+≡ (94c) ◁442b 473d▷
```
ushort  steal;    /* Page stealer lock */
```

⟨*function* procctlmemio 457⟩≡ (680)

```
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 522m⟩

    s->steal--;

    if(read == 0)
        p->newtlb = false;

    return n;
}
```

Uses Ebadarg 631d, SG_TEXT, SG_TYPE, false, fixfault() 166b, nexterror, pagedout 694, poperror 633d, seg() 100c, text2data() 458, and waserror 633d.

⟨*function* text2data 458⟩≡ (680)

```
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 522i⟩

    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 100a, SG_DATA 523b, incref() 69b, newseg() 97b, putseg() 98b, qlock() 63f, and qunlock() 64c.

19.4 /bin/strace

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* 459a⟩+≡ (453b) ◁141a 470a▷

```
Proc_tracesyscall,
```

⟨procctlreq() *CMstartsyscall case* 459b⟩≡ (586b)

```
case CMstartsyscall:
    if(p->state != Stopped)
        error(Ebadctl);
    p->procctl = Proc_tracesyscall;
    ready(p);
    procstopwait(p, Proc_tracesyscall); // will sleep
    break;
```

Uses CMstartsyscall-892 568b, Proc_tracesyscall 459a, Stopped 453a, procstopwait() 454e, and ready.

⟨syscall() Proc_tracesyscall *if, syscall entry(arm)* 459c⟩≡ (145e)

```
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 453b, Proc_tracesyscall 459a, false, free() 85b, procctl() 453d, syscallfmt() 460e, up, and validaddr() 101c.

⟨syscall() *other locals(arm)* 459d⟩+≡ (145e) ◁147a

```
vlong startns, stopns;
```

⟨syscall() Proc_tracesyscall *if, syscall exit(arm)* 459e⟩≡ (145e)

```
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 453b, Proc_tracesyscall 459a, free() 85b, procctl() 453d, sysretfmt() 464, todget() 239b, and up.

⟨sysrfork() *inherit* Proc_tracesyscall 460a⟩≡

```
        if(up && up->procctl == Proc_tracesyscall)
            p->procctl = Proc_tracesyscall;
```

⟨Proc *debugger fields* 460b⟩+≡ (49f) ◁455c 466b▷

```
    char  *syscalltrace;  /* syscall trace */
```

⟨pexit() *at the start, free resources* 460c⟩+≡ (134c) ◁234g

```
    if(up->syscalltrace)
        free(up->syscalltrace);
```

Uses free() 85b and up.

⟨procread() *cases* 460d⟩+≡ (571d) ◁456a 474c▷

```
    case Qsyscall:
        if(!p->syscalltrace)
            return 0;
        n = readstr(offset, a, n, p->syscalltrace);
        return n;
```

Uses Qsyscall-878 568a and readstr() 637d.

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 460e⟩≡ (719a)

```
    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 34c, false, fmtrwdata() 465, fmtuserstring() 466a, free() 85b, nsyscall 42a, sysctab, up, and validaddr() 101c.

⟨*function* sysretfmt 464⟩≡ (719a)

```
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 34c, fmtrwdata() 465, fmtuserstring() 466a, free() 85b, and up.

⟨*function* fmtrwdata 465⟩≡ (719a)

```
// 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()` 85b, `smalloc()` 85c, and `validaddr()` 101c.

⟨*function* `fmtuserstring` 466a⟩≡ (719a)

```
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()` 85b, `smalloc()` 85c, `validaddr()` 101c, and `vmemchr()` 101d.

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* 466b⟩+≡ (49f) ◁460b 584a▷

```
bool trace;    /* process being traced? */
```

⟨`procctlreq()` *CMtrace case* 466c⟩≡ (586b)

```
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 568b and true.

⟨*hook proctrace* 467a⟩≡ (708a)
```
  void (*proctrace)(Proc*, /*enum<tevent>*/int, vlong) = 0; // was in devproc.c
```

Uses proctrace 467a.

⟨*enum* Tevent 467b⟩≡ (? 0—1)
```
  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 546⟩

      Nevent, // must be last
  };
```

⟨*struct* Traceevent 467c⟩≡ (? 0—1)
```
  struct Traceevent {
      ulong pid;
      // enum<tevent>
      ulong etype; /* Event type */
      vlong time; /* time stamp  */ // dimension?
  };
```

⟨ready() *other locals* 467d⟩≡ (211a)
```
  void (*pt)(Proc*, int, vlong);
```

⟨ready() *hook proctrace* 467e⟩≡ (211a)
```
  pt = proctrace;
  if(pt)
      pt(p, SReady, 0);
```

Uses proctrace 467a.

⟨runproc() *other locals* 467f⟩≡ (204) 623c▷
```
  void (*pt)(Proc*, int, vlong);
```

⟨runproc() *hook proctrace* 467g⟩≡ (204)
```
  pt = proctrace;
  if(pt)
      pt(p, SRun, 0);
```

Uses proctrace 467a.

⟨pexit() *other locals* 467h⟩+≡ (134c) ◁138b 472e▷
```
  void (*pt)(Proc*, int, vlong);
```

⟨pexit() *hook proctrace* 467i⟩≡ (134c)
```
  pt = proctrace;
  if(pt)
      pt(up, SDead, 0);
```

Uses proctrace 467a.

⟨sleep() *other locals* 468a⟩≡ (212e)
```
void (*pt)(Proc*, int, vlong);
```

⟨sleep() *hook proctrace* 468b⟩≡ (212e)
```
pt = proctrace;
if(pt)
    pt(up, SSleep, 0);
```
Uses proctrace 467a.

⟨procctlreq() *CMevent case* 468c⟩≡ (586b)
```
case CMevent:
    pt = proctrace;
    if(up->trace && pt)
        pt(up, SUser, 0);
    break;
```
Uses CMevent-904 568b, proctrace 467a, and up.

⟨*global trace txxx* 468d⟩≡ (680)
```
// array<Traceevent>
static Traceevent *tevents;
static int topens;
static int tproduced, tconsumed;
static Lock tlock;
```

⟨*function* eventsavailable 468e⟩≡ (680)
```
static int
eventsavailable(void *)
{
    return tproduced > tconsumed;
}
```
Uses tconsumed-918 468d and tproduced-917 468d.

⟨*function* _proctrace 468f⟩≡ (680)
```
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 468h, Nevents-905 468g, false, tconsumed-918 468d, tevents-915 468d, todget() 239b, topens-916 468d, and tproduced-917 468d.

⟨*constant* Nevents 468g⟩≡ (680)
```
Nevents = 0x4000,
```

⟨*constant* Emask 468h⟩≡ (680)
```
Emask = Nevents - 1,
```
Uses Nevents-905 468g.

⟨procopen() *Qtrace if* 469a⟩≡ (577)

```
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 272b, Nevents-905 468g, OREAD 279c, QID-911 570a, Qtrace-860 568a, _proctrace() 468f, lock() 58d, malloc() 85a, nexterror, openmode() 280a, poperror 633d, proctrace 467a, tconsumed-918 468d, tevents-915 468d, tlock-919 468d, topens-916 468d, tproduced-917 468d, unlock() 60d, and waserror 633d.

⟨procclose() *Qtrace if* 469b⟩≡ (583)

```
if(QID(c->qid) == Qtrace){
    lock(&tlock);
    if(topens > 0)
        topens--;
    if(topens == 0)
        proctrace = nil;
    unlock(&tlock);
}
```

Uses QID-911 570a, Qtrace-860 568a, lock() 58d, proctrace 467a, tlock-919 468d, topens-916 468d, and unlock() 60d.

⟨procread() *Qtrace if* 469c⟩≡ (571d)

```
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 468h, Nevents-905 468g, QID-911 570a, Qtrace-860 568a, eventsavailable() 468e, tconsumed-918 468d, tevents-915 468d, and tproduced-917 468d.

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* 470a⟩+≡ (453b) ◁459a 711a▷

```
Proc_traceme,
```

⟨procctl() Proc_traceme *case (and fallthrough* Proc_stopme*)* 470b⟩≡ (453d)

```
case Proc_traceme:
    if(p->nnote == 0)
        return;
    /* No break */
```

Uses Proc_traceme 470a.

⟨procctlreq() *CMstartstop case* 470c⟩≡ (586b)

```
case CMstartstop:
    if(p->state != Stopped)
        error(Ebadctl);
    p->procctl = Proc_traceme;
    ready(p);
    procstopwait(p, Proc_traceme);
    break;
```

Uses CMstartstop-891 568b, Proc_traceme 470a, Stopped 453a, procstopwait() 454e, 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/#/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` 471a⟩≡ (705c)
```
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* 471b⟩≡ (49f) 475c ▷
```
// hash<enum<proctime>, ulong>
ulong time[5];  /* User, Sys, Real; child U, S */
```

⟨`sysrfork()` *setting time field* 471c⟩≡ (116b)
```
memset(p->time, 0, sizeof(p->time));
p->time[TReal] = CPUS(0)->ticks;
```
Uses `CPUS` 645 and `TReal` 471a.

⟨`kproc()` *setting time field* 471d⟩≡ (139c)
```
memset(p->time, 0, sizeof(p->time));
p->time[TReal] = CPUS(0)->ticks;
```
Uses `CPUS` 645 and `TReal` 471a.

⟨accountime() *update time of current process* 472a⟩≡ (624b)
```
p->time[p->insyscall ? TSys : TUser]++;
```
Uses TSys 471a and TUser 471a.

⟨consread() *cases* 472b⟩+≡ (366a) ◁243b 604▷
```
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 645, NUMSIZE 704a, Qcputime-1784 600a, TK2MS 223a, TReal 471a, readnum() 637c, and up.

⟨procread() *Qstatus case, time part* 472c⟩≡ (571d)
```
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 645, NUMSIZE 704a, TK2MS 223a, TReal 471a, and readnum() 637c.

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* 472d⟩≡ (131e)
```
ulong time[3];  /* of loved one and descendants */
```

⟨pexit() *other locals* 472e⟩+≡ (134c) ◁467h
```
long utime, stime;
```

⟨pexit() *set wait msg time field* 472f⟩≡ (136c)
```
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 645, TCSys 471a, TCUser 471a, TReal 471a, TSys 471a, TUser 471a, tk2ms() 223b, and up.

⟨pexit() *update TC time of parent* 473a⟩≡ (136d)
```
p->time[TCUser] += utime;
p->time[TCSys] += stime;
```
Uses TCSys 471a and TCUser 471a.

⟨sysawait() *snprint time field arguments* 473b⟩≡ (132a)
```
w.time[TUser], w.time[TSys], w.time[TReal],
```
Uses TReal 471a, TSys 471a, and TUser 471a.

⟨procread() *Qwait case, snprint time field arguments* 473c⟩≡ (566b)
```
wq->w.time[TUser], wq->w.time[TSys], wq->w.time[TReal],
```
Uses TReal 471a, TSys 471a, and TUser 471a.

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* 473d⟩+≡ (94c) ◁456e 522e▷
```
kern_wp  profile;  /* Tick profile area */ // for TSEG only
```

⟨putseg() *free profile* 473e⟩≡ (98b)
```
if(s->profile != nil)
    free(s->profile);
```
Uses free() 85b.

⟨*constant* LRESPROF 473f⟩≡ (694)
```
LRESPROF  = 3,
```

⟨proc_init() *other init* 473g⟩≡ (570d)
```
addclock0link((void (*)(void))profclock, 113);  /* Relative prime to HZ */
```
Uses addclock0link() 231a and profclock() 473h.

⟨*clock callback profclock* 473h⟩≡ (680)
```
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 476e⟩
        segclock(ur->pc);
    }
}
```
Uses Running 51b, segclock() 474a, and up.

⟨*clock callback segclock* 474a⟩≡ (702)

```
// 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 94c, TK2MS 223a, TSEG 100a, and up.

⟨procctlreq() *CMprofile case* 474b⟩≡ (586b)

```
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 568b, LRESPROF 94c, SG_TEXT, SG_TYPE, TSEG 100a, free() 85b, and malloc() 85a.

⟨procread() *cases* 474c⟩+≡ (571d) ◁460d 566b▷

```
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 94c, Qprofile-877 568a, and TSEG 100a.

⟨procgen() *Qprofile case* 474d⟩≡ (580d)

```
case Qprofile:
    q = p->seg[TSEG];
    if(q && q->profile) {
        len = (q->top-q->base)>>LRESPROF;
        len *= sizeof(*q->profile);
    }
    break;
```

Uses LRESPROF 94c, Qprofile-877 568a, and TSEG 100a.

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` 475a⟩≡ (? 0—1)
```
struct Tos {
    ⟨Tos prof field ??⟩
    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 ??⟩
    /* top of stack is here */
};
```

⟨`Cpu` *other fields* 475b⟩+≡ (48a) ◁441b
```
// cyclefreq == cpuhz if havetsc, 0 otherwise
uvlong  cyclefreq;    /* Frequency of user readable cycle counter */
```

⟨`Proc` *stats and profiling fields* 475c⟩+≡ (49f) ◁471b
```
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)* 475d⟩≡ (733b)
```
#define arch_cycles(ip) *(ip) = arch_lcycles()
```

⟨*type CpSPM secondary registers(arm)* 475e⟩≡ (720c)
```
/*
 * CpSPM Secondary (CRm) registers and opcode2 fields (armv6)
 */
#define CpSPMperf   12          /* various counters */
```

⟨*type CpSPM opcode2(arm)* 475f⟩≡ (720c)
```
#define CpSPMctl    0           /* performance monitor control */
#define CpSPMcyc    1           /* cycle counter register */
```

⟨`_startpg()` *enable cycle counter(raspberry pi1)(arm)* 475g⟩≡ (250a)
```
/*
 * enable cycle counter
 */
MOVW    $1, R1
MCR CpSC, 0, R1, C(CpSPM), C(CpSPMperf), CpSPMctl
```

⟨*function* `arch_lcycles` *(raspberry pi1)(arm)* 476a⟩≡ (754a)
```
TEXT arch_lcycles(SB), 1, $-4
    MRC CpSC, 0, R0, C(CpSPM), C(CpSPMperf), CpSPMcyc
    RET
```

⟨`sysexec()` *locals* 476b⟩+≡ (123b) ◁298a 477a▷
```
Tos *tos;
```

⟨`sysexec()` *tos settings* 476c⟩≡ (128b)
```
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* 476d⟩≡ (151)
```
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* 476e⟩≡ (473h)
```
Tos *tos;
tos = (Tos*)(USTKTOP-sizeof(Tos));
tos->clock += TK2MS(1);
```
Uses `TK2MS` 223a.

⟨`syscall()` *adjust kentry* 476f⟩≡ (145e)
```
arch_cycles(&up->kentry);
```

⟨`trap()` *adjust kentry when interrupt user* 476g⟩≡ (150b)
```
arch_cycles(&up->kentry);
```

⟨`arch_procsave()` *cycles adjustments* 476h⟩≡ (209c)
```
uvlong t;

arch_cycles(&t);
p->pcycles += t;
```

⟨`arch_procrestore` *cycles adjustments* 476i⟩≡ (210a)
```
arch_cycles(&t);
p->pcycles -= t;
```

⟨`sysexec()` *nbytes tos adjustments* 476j⟩≡ (127a)
```
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* 477a⟩+≡ (123b) ◁476b
```
char line[sizeof(Exec)];
char *progarg[sizeof(Exec)/2+1];
char progelem[64];

bool indir = false;
```

⟨sysexec() *process sharpbang* 477b⟩≡ (123b)
```
/*
 * 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 478a⟩≡ (716a)

```
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* 478b⟩≡ (127a)

```
if(indir){
    argp = progarg;
    while(*argp){
        a = *argp++;
        nbytes += strlen(a) + 1;
        nargs++;
    }
}
```

⟨sysexec() *if indir argp adjustments* 478c⟩≡ (128b)

```
if(indir)
    argp = progarg;
```

⟨sysexec() *if indir argp adjustments again* 479⟩≡ (128b)

```
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` 480a⟩≡ (705c)

```
/*
 * 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* 480b⟩+≡ (49f) ◁285c 481d▷

```
// enum<FPSaveStatus>
int fpstate;
```

⟨`newproc()` *set fields of* `p` 480c⟩+≡ (111d) ◁420a 531d▷

⟨`newproc()` *fp init* 481a⟩

⟨newproc() *fp init* 481a⟩≡ (480c)
```
p->fpstate = FPinit;
```
Uses FPinit 480a.

⟨sysrfork() *when* RFPROC *propagate fields from up to p* 481b⟩+≡ (116b) ◁455f 481e▷
⟨sysrfork() *propagate fpstate* 481c⟩

⟨sysrfork() *propagate fpstate* 481c⟩≡ (481b)
```
/* don't penalize the child, it hasn't done FP in a note handler. */
p->fpstate = up->fpstate & ~FPillegal;
```
Uses FPillegal 480a and up.

22.1.2 Arch_FPsave (ARM)

⟨Proc *other fields* 481d⟩+≡ (49f) ◁480b 563c▷
```
Arch_FPsave  fpsave;   /* address of this is known by db */
```

⟨sysrfork() *when* RFPROC *propagate fields from up to p* 481e⟩+≡ (116b) ◁481b 567d▷
⟨sysrfork() *propagate fpsave* 481f⟩

⟨sysrfork() *propagate fpsave* 481f⟩≡ (481e)
```
p->fpsave = up->fpsave;
```
Uses up.

⟨*struct* Arch_FPsave*(arm)* 481g⟩≡ (754d)
```
/*
 * 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 481h⟩
};
```

⟨Proc Arch FPsave *other fields* 481h⟩≡ (481g)
```
int fpstate; // vs Proc.fpstate??

ulong   status;
ulong   control;

uintptr pc;     /* of failed fp instr. */
```

⟨*enum* _anon_*(*(arch/arm/dat_processes.h)(arm)*)* 481i⟩≡ (754d)
```
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)* 482a⟩≡ (117e)
```
fpusysrforkchild(p, cureg, up);
```
Uses `fpusysrforkchild()` 482b and `up`.

⟨*function* `fpusysrforkchild`*(arm)* 482b⟩≡ (755)
```
/*
 * 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` 480a.

⟨`arch_procsetup()` *floating point(arm)* 482c⟩≡ (125d)
```
fpusysprocsetup(p);
```
Uses `fpusysprocsetup()` 482d.

⟨*function* `fpusysprocsetup`*(arm)* 482d⟩≡ (755)
```
/*
 * 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()` 490a.

⟨`arch_procsave()` *floating point(arm)* 482e⟩≡ (209c)
```
/* TODO: save and restore VFPv3 FP state once 5[cal] know the new registers.*/
fpuprocsave(p);
```
Uses `fpuprocsave()` 483a.

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)* 483a⟩≡ (755)
```
/*
 * 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` 480a, `FPinactive` 480a, `Moribund` 135a, and `fpclear()` 491d.

⟨`arch_procrestore()` *floating point(arm)* 483b⟩≡ (210a)
```
fpuprocrestore(p);
```
Uses `fpuprocrestore()` 483c.

⟨*function* `fpuprocrestore`*(arm)* 483c⟩≡ (755)
```
/*
 * 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)* 483d⟩≡ (145e)
```
if(scallnr == RFORK)
    fpusysrfork(ureg);
```
Uses `fpusysrfork()` 484a.

⟨*function* `fpusysrfork`*(arm)* 484a⟩≡ (755)

```
/*
 * 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` 480a, `FPinactive` 480a, 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)* 484b⟩≡ (755)

```
/* 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` 488a, `assert` 693a, `fpoff()` 490a, `fpon()` 491c, and `fprd()` 486b.

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)* 484c⟩≡ (755)

```
/*
 * 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 480a, FPillegal 480a, FPinactive 480a, and up.

⟨*function* fpunoted*(arm)* 485a⟩≡ (755)

```
/*
 * Called from sysnoted() via the machine-dependent
 * noted() routine.
 * Clear the flag set above in fpunotify().
 */
void
fpunoted(void)
{
    up->fpstate &= ~FPillegal;
}
```

Uses FPillegal 480a 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)* 485b⟩≡ (29a)

```
#define CpFP        10          /* float FP, VFP cfg. */
#define CpDFP       11          /* double FP */
```

⟨*enum* _anon_*(*(processes/arm/vfp3.c)(arm)*)* 485c⟩≡ (755)

```
/* subarchitecture code in cpu->havefp */
enum VFPKind {
    VFPv2   = 2,
    VFPv3   = 3,
};
```

⟨Cpu Arch *float fields(arm)* 485d⟩≡ (48e) 485e▷

```
// 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)* 485e⟩+≡ (48e) ◁485d 485f▷

```
bool fpon;
```

⟨Cpu Arch *float fields(arm)* 485f⟩+≡ (48e) ◁485e 491a▷

```
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)* 486a⟩≡ (724b)
```
/* 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) 499a⟩
    ⟨setupfpctlop() invalidate instruction cache(arm) 499c⟩
}
```
Uses `CpFP` 485b, `Nfpctlregs` 481i, and `RETinst-2383` 722b.

⟨*function* `fprd`*(arm)* 486b⟩≡ (724b)
```
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()` 486a.

⟨*function* `fpwr`*(arm)* 486c⟩≡ (724b)
```
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()` 486a.

⟨*function* `setupfpop`*(arm)* 487a⟩≡ (724b)

```
/* 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) 499b⟩
    ⟨setupfpop() invalidate instruction cache(arm) 499d⟩
}
```

Uses `RETinst-2383` 722b.

⟨*function* `fprestreg`*(arm)* 487b⟩≡ (724b)

```
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` 485b, `arch_coherence`, and `setupfpop()` 487a.

⟨*function* `fpsavereg`*(arm)* 487c⟩≡ (724b)

```
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)*)* 488a⟩≡ (755)
```
/* 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)*)* 488b⟩≡ (755)
```
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 488b, FPIDNRM-2578 488b, Fpdex-2572 488b, and MASK 34b.

⟨*enum* _anon_*(*(processes/arm/vfp3.c)4(arm)*)* 488c⟩≡ (755)
```
enum {
    /* CpCPaccess bits */
    Cpaccnosimd =   1u << 31,
    Cpaccd16 =  1 << 30,
};
```

⟨*function* subarch*(arm)* 488d⟩≡ (755)
```
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 692c.

⟨*function* implement*(arm)* 489a⟩≡ (755)

```
static char *
implement(uchar impl)
{
    if (impl == 'A')
        return "arm";
    else
        return "unknown";
}
```

⟨CpCONTROL.opcode2 *other cases(arm)* 489b⟩≡ (29c) 506d ▷

```
#define CpCPaccess  2
```

⟨*function* havefp*(arm)* 489c⟩≡ (755)

```
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 29b, CpCPaccess 489b, CpDFP 485b, CpFP 485b, Cpaccd16-2584 488c, Cpaccnosimd-2583 488c, Fpsid-2563 488a, MASK 34b, VFPv2-2561 485c, VFPv3-2562 485c, cprdsc() 723c, cpwrsc() 724a, false, fprd() 486b, print, and true.

⟨*function* fpoff*(arm)* 490a⟩≡ (755)

```
    /*
     * 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 488a, false, and fpwr() 486c.

⟨*function* fponly*(arm)* 490b⟩≡ (755)

```
    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 488b, Fpexc-2567 488a, fpwr() 486c, havefp() 489c, and true.

⟨*function* fpcfg*(arm)* 490c⟩≡ (755)

```
    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 488b, Dn-2576 488b, Fpscr-2564 488a, Fpsid-2563 488a, Fz-2577 488b, MASK 34b, fprd() 486b, fpwr() 486c, implement() 489a, print, subarch() 488d, and true.

⟨Cpu Arch *float fields(arm)* 491a⟩+≡ (48e) ◁485f 492a▷

```
    bool fpconfiged;
```

⟨*function* fpinit*(arm)* 491b⟩≡ (755)

```
    void
    fpinit(void)
    {
        if (havefp()) {
            fpononly();
            fpcfg();
        }
    }
```

Uses fpcfg() 490c, fpononly() 490b, and havefp() 489c.

⟨*function* fpon*(arm)* 491c⟩≡ (755)

```
    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 488b, Fpscr-2564 488a, fpcfg() 490c, fpononly() 490b, fprd() 486b, fpwr() 486c, and havefp() 489c.

⟨*function* fpclear*(arm)* 491d⟩≡ (755)

```
    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 488a, Fpmbc-2573 488b, fpon() 491c, fprd() 486b, and fpwr() 486c.

⟨*function* fpstuck*(arm)* 491e⟩≡ (755)

```
    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)* 492a⟩+≡ (48e) ◁491a

```
    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)*)* 492b⟩≡ (755)

```
    enum {
        N = 1<<31,
        Z = 1<<30,
        C = 1<<29,
        V = 1<<28,
        REGPC = 15,
    };
```

⟨*function* condok*(arm)* 492c⟩≡ (755)

```
    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 492b, N-2585 492b, V-2588 492b, and Z-2586 492b.

⟨*macro* ISCPOP*(arm)* 493a⟩≡ (755)

```
#define ISCPOP(op)  ((op) == 0xE || ((op) & ~1) == 0xC)
```

⟨*macro* ISVFPOP*(arm)* 493b⟩≡ (755)

```
#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)* 493c⟩≡ (755)

```
/* 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 480a, ISVFPOP-2591 493b, MASK 34b, NDebug 418c, condok() 492c, fpstuck() 491e, iprint, poperror 633d, postnote, up, validaddr() 101c, and waserror 633d.

⟨trap() *other locals(arm)* 493d⟩+≡ (150b) ◁153f

```
int rv;
```

⟨`trap()` *when undefined instruction, if float instr(arm)* 494a⟩≡ (152e)

```
/* look for floating point instructions to interpret */
rv = fpuemu(ureg);
```

Uses `fpuemu()` 493c.

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)* 494b⟩≡ (755)

```
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` 692a, `NDebug` 418c, `postnote`, and `up`.

⟨*function* `mathemu`*(arm)* 494c⟩≡ (755)

```
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 480a, FPinactive 480a, FPinit 480a, fpclear() 491d, fpinit() 491b, and up.

⟨*function* fprestore*(arm)* 495a⟩≡ (755)

```
    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 488b, Fpscr-2564 488a, assert 693a, fpon() 491c, fprd() 486b, fprestreg() 487b, and fpwr() 486c.

22.1.8 /proc/x/fpregs

⟨procread() *Qfpregs case* 495b⟩≡ (571d)

```
    case Qfpregs:
        rptr = (uchar*)&p->fpsave;
        rsize = sizeof(Arch_FPsave);
        goto regread;
```

Uses Qfpregs-864 568a.

⟨procwrite *Qfpregs case* 495c⟩≡ (574c)

```
    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 568a.

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)* 496a⟩≡ (29b) 502a ▷
```
#define CpCACHE     7           /* cache/write buffer control */
```

⟨CpCONTROL.CpMainctl *other cases(arm)* 496b⟩≡ (29d) 506a ▷
```
#define CpCdcache   0x00000004  /* C: data cache on */
#define CpCicache   0x00001000  /* I: instruction cache on */
```

⟨armstart() *disable features(arm)* 496c⟩≡ (248)
```
BIC $(CpCdcache|CpCicache), R1
```

⟨armstart() *invalidate caches(arm)* 496d⟩≡ (248)
```
MCR CpSC, 0, R0, C(CpCACHE), C(CpCACHEinvu), CpCACHEall
```

⟨armstart() *reenable features(arm)* 496e⟩≡ (248)
```
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)* 496f⟩≡ (720c)
```
/*
 * CpCACHERANGE opcode2 fields for MCRR instruction (armv6)
 */
#define CpCACHERANGEinvd    6       /* invalidate data */
#define CpCACHERANGEdwb     12      /* writeback */
#define CpCACHERANGEdwbi    14      /* writeback+invalidate */
```

⟨*instruction MCRR(arm)* 496g⟩≡ (721)
```
#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)* 496h⟩≡ (720c)
```
#define Buffered    0x00000004      /* L[12]: write-back not -thru */
#define Cached      0x00000008      /* L[12] */
```

⟨Soc *mmu fields* 496i⟩≡ (54c)
```
u32int  l1ptedramattrs;
u32int  l2ptedramattrs;
```

⟨soc(raspberry pi1) *other fields(arm)* 496j⟩≡ (54d) 503g ▷
```
.l1ptedramattrs = Cached | Buffered,
.l2ptedramattrs = Cached | Buffered,
```
Uses `Buffered` 496h and `Cached` 496h.

⟨*constant* `L1ptedramattrs`*(arm)* 497a⟩≡ (748b)
```
#define L1ptedramattrs  soc.l1ptedramattrs
```

⟨*constant* `L2ptedramattrs`*(arm)* 497b⟩≡ (748b)
```
#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)* 497c⟩≡ (721)
```
#define ISB \
    MOVW    $0, R0; \
    MCR CpSC, 0, R0, C(CpCACHE), C(CpCACHEinvi), CpCACHEwait
```

⟨*instruction DSB(arm)* 497d⟩≡ (721)
```
#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)* 497e⟩≡ (720c)
```
/*
 * 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)* 497f⟩≡ (720c)
```
#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)* 497g⟩≡ (749b)
```
#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)* 498a⟩≡ (209b)
```
/* write back dirty and invalidate l1 caches */
cacheuwbinv();
```

⟨`arch_mmurelease()` *write back cache after adjusting page tables(arm)* 498b⟩≡ (162f)
```
/* make sure map is in memory */
/* could be smarter about how much? */
cachedwbse(&cpu->mmul1[L1X(UZERO)], (L1hi - L1lo)*sizeof(PTE));
```
Uses `L1X-1842` 160a, `L1hi-1848` 162b, and `L1lo-1847` 162a.

⟨`arch_mmuswitch()` *write back cache after adjusting page tables(arm)* 498c⟩≡ (209b)
```
/* make sure map is in memory */
/* could be smarter about how much? */
cachedwbse(&l1[L1X(UZERO)], (L1hi - L1lo)*sizeof(PTE));
```
Uses `L1X-1842` 160a, `L1hi-1848` 162b, and `L1lo-1847` 162a.

⟨`mmuinit1()` *write back cache after adjusting page tables(arm)* 498d⟩≡ (259c)
```
cachedwbse(&l1[L1X(0)], sizeof(PTE));
```
Uses `L1X-1842` 160a.

⟨`mmukmap()` *write back cache after changing page tables(arm)* 498e⟩≡ (358c)
```
cachedwbse(pte0, (uintptr)pte - (uintptr)pte0);
```

⟨`arch_putmmu()` *write back cache after changing pde(arm)* 498f⟩≡ (167)
```
cachedwbse(l1, sizeof *l1);
```

⟨`arch_putmmu()` *write back cache after changing pte(arm)* 498g⟩≡ (167)
```
cachedwbse(&pte[L2X(va)], sizeof pte[0]);
```
Uses `L2X-1843` 160b.

⟨`arch_putmmu()` *write back cache before fixing va fault(arm)* 498h⟩≡ (167)
```
/* force l2 page to memory */
cachedwbse((void *)pg->va, BY2PG);
```

⟨`arch_mmurelease()` *invalidate cache before adjusting page tables(arm)* 498i⟩≡ (162f)
```
/* write back dirty and invalidate l1 caches */
cacheuwbinv();
```

⟨`arch_kunmap()` *write back cache for page(arm)* 498j⟩≡ (169f)
```
cachedwbinvse(k, BY2PG);
```

⟨`arch__trapinit()` *invalidate cache after adjust vectors(arm)* 498k⟩≡ (256e)
```
cacheuwbinv();
l2cacheuwbinv();
```

⟨`fbinit()` *write back cache after modified fb info(arm)* 498l⟩≡ (358b)
```
cachedwbinvse(fi, sizeof(Fbinfo));
```

⟨`arch__kexit()` *write back cache(arm)* 498m⟩≡ (151)
```
/* make visible immediately to user proc */
cachedwbinvse(tos, sizeof *tos);
```

⟨setupfpctlop() *write back cache for modified instr(arm)* 499a⟩≡ (486a)
```
cachedwbse(instr, sizeof instrsz);
```

⟨setupfpop() *write back cache for modified instr(arm)* 499b⟩≡ (487a)
```
cachedwbse(instr, sizeof instrsz);
```

⟨setupfpctlop() *invalidate instruction cache(arm)* 499c⟩≡ (486a)
```
cacheiinv();
```

⟨setupfpop() *invalidate instruction cache(arm)* 499d⟩≡ (487a)
```
cacheiinv();
```

⟨*function cachexxx(raspberry pi1)(ram)* 499e⟩≡ (749b)
```
/*
 * 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* 501a⟩≡ (60d)
```
// 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* 501b⟩≡ (62b)
```
arch_coherence();
```
Uses arch_coherence.

⟨irq() *before dispatch(arm)* 501c⟩≡ (159b)
```
arch_coherence();
```
Uses arch_coherence.

⟨irq() *after dispatch(arm)* 501d⟩≡ (159b)
```
arch_coherence();
```
Uses arch_coherence.

⟨arch__trapinit() *coherence(arm)* 501e⟩≡ (256e)
```
arch_coherence();
```
Uses arch_coherence.

⟨init0() *coherence(arm)* 501f⟩≡ (267a)
```
arch_coherence();
```
Uses arch_coherence.

⟨cas32() *if address contained old value, call coherence(arm)* 501g⟩≡ (443b)
```
if (r)
    arch_coherence();
```
Uses arch_coherence.

22.3.2 Memory barriers (ARM)

⟨*instruction BARRIERS(arm)* 501h⟩≡ (721)
```
#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)* 501i⟩≡ (745)
```
Soc soc = {
    .dramsize   = 1024*MiB,
    .physio     = 0x3F000000,
     ⟨soc(raspberry pi2) other fields(arm) 502f⟩
};
```
Uses MiB 34g.

22.4.1 CPU ID

⟨CpSC primary registers *other cases(arm)* 502a⟩+≡ (29b) ◁496a 504a▷
```
#define CpID        0           /* ID and cache type */
```

⟨*type CpID secondary registers(arm)* 502b⟩≡ (720c)
```
/*
 * CpID Secondary (CRm) registers.
 */
#define CpIDidct    0
#define CpIDfeat    1
```

⟨*type CpID opcode2(arm)* 502c⟩≡ (720c)
```
/*
 * 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)* 502d⟩≡ (721)
```
/*
 * 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)* 502e⟩≡ (720c)
```
#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)* 502f⟩≡ (501i) 503h▷
```
.l1ptedramattrs = Cached | Buffered | L1wralloc | L1sharable,
.l2ptedramattrs = Cached | Buffered | L2wralloc | L2sharable,
```
Uses Buffered 496h, Cached 496h, L1sharable 502e, L1wralloc 502e, L2sharable 502e, and L2wralloc 502e.

⟨*function* l2ap*(raspberry pi2)(arm)* 502g⟩≡ (745)
```
int
l2ap(int ap)
{
    return (AP(0, (ap)));
}
```
Uses AP 161f.

22.4.3 TTB

⟨*type* CpTTB*(arm)* 503a⟩≡ (720c)
```
/*
 * CpTTB op1==0, Crm==0 opcode2 values.
 */
#define CpTTB1      1           /* cortex */
```

⟨*type CpTTB cache control bits(arm)* 503b⟩≡ (720c)
```
/*
 * 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)* 503c⟩≡ (745)
```
/*
 * Arm local regs for SMP
 */
struct Mbox {
    u32int  doorbell;
    u32int  mbox1;
    u32int  mbox2;
    u32int  startcpu;
};
```

⟨*struct* Mboxes*(arm)* 503d⟩≡ (745)
```
struct Mboxes {
    Mbox    set[4];
    Mbox    clr[4];
};
```

⟨*constant* ARMLOCAL*(arm)* 503e⟩≡ (753 745)
```
#define ARMLOCAL    (VIRTIO+IOSIZE)
```

⟨Soc *other fields* 503f⟩≡ (54c)
```
uintptr armlocal;
```

⟨soc(raspberry pi1) *other fields(arm)* 503g⟩+≡ (54d) ◁496j 831f▷
```
.armlocal   = 0,
```

⟨soc(raspberry pi2) *other fields(arm)* 503h⟩+≡ (501i) ◁502f 831g▷
```
.armlocal   = 0x40000000,
```

⟨mmuinit() *set armlocal virtual memory mapping(arm)* 503i⟩≡ (249c)
```
// for raspi2
pa = soc.armlocal;
if(pa)
    l1[L1X(va)] = pa|Dom0|L1AP(Krw)|Section;
```
Uses Dom0 161d, Krw 161b, L1AP 161e, L1X-1842 160a, and Section 160d.

22.4.5 Timer

⟨CpSC primary registers *other cases(arm)* 504a⟩+≡ (29b) ◁502a 504c▷
```
#define CpTIMER     14          /* Generic timer (cortex-a7) */
```

⟨*type* CpTIMER*(arm)* 504b⟩≡ (720c)
```
/*
 * 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)* 504c⟩+≡ (29b) ◁504a
```
#define CpCLD       9       // Performance monitor
```

⟨*type CpCLD secondary registers(arm)* 504d⟩≡ (720c)
```
/*
 * CpCLD Secondary (CRm) registers and opcode2 fields for op1==0. (cortex)
 */
#define CpCLDena    12          /* enables */
#define CpCLDcyc    13          /* cycle counter */
```

⟨*type CpCLD opcode2(arm)* 504e⟩≡ (720c)
```
#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)* 504f⟩≡ (741b)
```
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)* 506a⟩+≡ (29d) ◁496b 506b▷

```
#define CpCsw       (1<<10)     /* SW: SWP(B) enable (deprecated in v7) */
#define CpCpredict  0x00000800  /* Z: branch prediction (armv7) */
```

⟨CpCONTROL.CpMainctl *other cases(arm)* 506b⟩+≡ (29d) ◁506a 506c▷

```
#define CpCve       (1<<24)     /* VE: intr vectors enable */
#define CpCtre      (1<<28)     /* TRE: TEX remap enable */
```

⟨CpCONTROL.CpMainctl *other cases(arm)* 506c⟩+≡ (29d) ◁506b

```
#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)* 506d⟩+≡ (29c) ◁489b

```
#define CpAuxctl    1
```

⟨*type CpCONTROL CpAuxctl(arm)* 506e⟩≡ (720c)

```
/*
 * 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)* 506f⟩≡ (741b)

```
TEXT _startpg(SB), 1, $-4

    ⟨_startpg() enable cycle counter(raspberry pi2)(arm) 520h⟩
    /*
     * 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)* 506g⟩≡ (250b)

```
launchinit(getncpus());
```

Uses `launchinit()` 507a.

⟨*function* launchinit*(arm)* 507a⟩≡ (742b)
```
void
launchinit(int ncpus)
{
    int xcpu;
    Cpu *mm;
    PTE *l1;

    ⟨launchinit() sanity check ncpus(arm) 507c⟩
    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 507b, cpus 49a, and mallocalign() 86c.

⟨*constant* L1SIZE*(arm)* 507b⟩≡ (747)
```
#define L1SIZE      (4 * BY2PG)
```

⟨launchinit() *sanity check ncpus(arm)* 507c⟩≡ (507a)
```
if(ncpus > MAXCPUS)
    ncpus = MAXCPUS;
```

⟨*function* startcpus*(raspberry pi1)(arm)* 507d⟩≡ (744b)
```
int
startcpus(uint)
{
    return 1;
}
```

⟨*function* getncpus*(raspberry pi1)(arm)* 507e⟩≡ (744b)
```
int
getncpus(void)
{
    return 1;
}
```

⟨*function* getncpus*(raspberry pi2)(arm)* 507f⟩≡ (745)
```
int
getncpus(void)
{
    int n, max;
    char *p;

    n = 4;
    if(n > MAXCPUS)
```

```
        n = MAXCPUS;
    ⟨getncpus() adjust n if ncpu config parameter(arm) 508a⟩
    return n;
}
```

⟨getncpus() *adjust n if ncpu config parameter(arm)* 508a⟩≡ (507f)

```
p = getconf("*ncpu");
if(p && (max = atoi(p)) > 0 && n > max)
    n = max;
```

⟨*function* startcpus*(raspberry pi2)(arm)* 508b⟩≡ (745)

```
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() 58d, startcpu() 508c, startlock-2448 745, and unlock() 60d.

⟨*function* startcpu*(raspberry pi2)(arm)* 508c⟩≡ (745)

```
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 503e and Mboxregs-2447 745.

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)* 509⟩≡ (741b)

```
/*
 * 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)* 510⟩≡ (741b)

```
TEXT _startpg2(SB), 1, $-4

    ⟨_startpg2() enable cycle counter(raspberry pi2)(arm) 520i⟩
    /*
     * 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)* 511a⟩≡ (745)
```
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` 503e, `Mboxregs-2447` 745, `arch__cpuinit()` 252d, `clockinit()` 258, `machon()` 253a, `mmuinit1()` 259c, `schedinit()` 203f, `startlock-2448` 745, `timersinit()` 259b, `unlock()` 60d, and `up`.

⟨`arch__cpuinit()` *it not cpu0* 511b⟩≡ (252d)
```
if (cpu->cpuno != 0) {
    /* synchronise with cpu 0 */
    cpu->ticks = cpu0->ticks;
}
```

Displaying processor information

⟨*function* `arch_cpuidprint` *(raspberry pi2)(arm)* 511c⟩≡ (745)
```
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)* 511d⟩≡ (745)
```
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 34b and assert 693a.

⟨*function* cpidget *(raspberry pi2)(arm)* 512a⟩≡ (741b)

```
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)* 512b⟩≡ (741b)

```
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)* 512c⟩≡ (741b)

```
#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)* 512d⟩≡ (144c)

```
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)* 513a⟩≡ (149b)
```
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)* 513b⟩≡ (721)
```
#define LDREX(fp,t)   WORD $(0xe<<28|0x01900f9f | (fp)<<16 | (t)<<12)
```

⟨*instruction STREX(arm)* 513c⟩≡ (721)
```
/* '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)* 513d⟩≡ (721)
```
#define CLREX   WORD    $0xf57ff01f
```

⟨*instruction DMB(arm)* 513e⟩≡ (730c)
```
#define DMB WORD    $0xf57ff05f /* data mem. barrier; last f = SY */
```

⟨*function* `arch_tas` *(raspberry pi2)(arm)* 513f⟩≡ (730c)
```
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)* 514a⟩≡ (752b)
```
Localctl    = 0x00,
```

⟨*constant* `Prescaler` *(arm)* 514b⟩≡ (752b)
```
Prescaler   = 0x08,
```

⟨`clockinit()` *if many processors* 514c⟩≡ (258)
```
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` 503e, `CpID` 502a, `CpIDfeat` 502b, `CpTIMER` 504a, `CpTIMERphys` 504b, `CpTIMERphysctl` 504b, `Imask-2667` 752c, `Localctl-2650` 514a, `Prescaler-2651` 514b, `cprdsc()` 723c, and `cpwrsc()` 724a.

⟨`clockinit()` *if not cpu0* 514d⟩≡ (258)
```
else
    arch_intrenable(IRQcntpns, localclockintr, nil, 0, "clock");
```

⟨*function* `localclockintr` *(arm)* 514e⟩≡ (753)
```
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` 504a, `CpTIMERphys` 504b, `CpTIMERphysctl` 504b, `Enable-2666` 752c, `Imask-2667` 752c, `cpwrsc()` 724a, and `timerintr()` 229d.

⟨*function* `arch_lcycles` *(raspberry pi2)(arm)* 514f⟩≡ (754c)
```
TEXT arch_lcycles(SB), 1, $-4
    MRC CpSC, 0, R0, C(CpCLD), C(CpCLDcyc), 0
    RET
```

⟨`arch_timerset()` *if not cpu0(arm)* 514g⟩≡ (228d)
```
if(cpu->cpuno > 0){
    cpwrsc(0, CpTIMER, CpTIMERphys, CpTIMERphysval, period);
    cpwrsc(0, CpTIMER, CpTIMERphys, CpTIMERphysctl, Enable);
}
```
Uses `CpTIMER` 504a, `CpTIMERphys` 504b, `CpTIMERphysctl` 504b, `CpTIMERphysval` 504b, `Enable-2666` 752c, and `cpwrsc()` 724a.

22.5.5 `arch_coherence()` (ARM)

⟨*function* `arm_arch_coherence` *(arm)* 514h⟩≡ (730a)
```
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)* 515a⟩≡ (749c)
```
#define CACHELINESZ 64
```

⟨*instruction CPSID(arm)* 515b⟩≡ (721)
```
#define CPSID   WORD    $0xf10c0080 /* intr disable: sets I bit */
```

⟨*type CpID CpIDid op1(arm)* 515c⟩≡ (720c)
```
/* CpIDid op1 values */
#define CpIDcsize   1           /* cache size (cortex) */
#define CpIDcssel   2           /* cache size select (cortex) */
```

⟨*function cachexxx(raspberry pi2)(ram)* 515d⟩≡ (749c)
```
/*
 * '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)* 519⟩≡ (156b)

```
  IRQlocal    = 96,
  IRQcntps    = IRQlocal + 0,
  IRQcntpns   = IRQlocal + 1,
```

Uses IRQlocal 519.

⟨Vctl *other fields(arm)* 520a⟩≡ (156c)
```
int cpu;
```

⟨irqenable() *if IRQlocal(arm)* 520b⟩≡ (158a)
```
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 519, LOCALREGS-2404 520g, Localintpending-2410 520e, and Localtimerint-2408 520d.

⟨irqenable() *if IRQlocal adjust enable(arm)* 520c⟩≡ (158a)
```
if(irq >= IRQlocal)
    *enable |= 1 << (irq - IRQlocal);
```
Uses IRQlocal 519.

⟨*constant* Localtimerint*(arm)* 520d⟩≡ (750a)
```
Localtimerint   = 0x40,
```

⟨*constant* Localintpending*(arm)* 520e⟩≡ (750a)
```
Localintpending = 0x60,
```

⟨*constant* Localmboxint*(arm)* 520f⟩≡ (750a)
```
Localmboxint    = 0x50,
```

⟨*constant* LOCALREGS*(arm)* 520g⟩≡ (750b)
```
#define LOCALREGS   (VIRTIO+IOSIZE)
```

22.5.8 Cycle counter adjustments

⟨_startpg() *enable cycle counter(raspberry pi2)(arm)* 520h⟩≡ (506f)
```
/*
 * 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)* 520i⟩≡ (510)
```
/*
 * 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* 521a⟩+≡ (89) ◁195c 521g▷
```
// int [0..NCOLOR[
char color;   /* Cache coloring */
```

⟨`pageinit()` *other locals* 521b⟩≡ (261c)
```
int color = 0;
```

⟨`pageinit()` *set page color* 521c⟩≡ (261c)
```
p->color = color;
color = (color+1)%NCOLOR;
```

⟨*macro* `getpgcolor`*(arm)* 521d⟩≡ (733b)
```
#define getpgcolor(a)   0
```

⟨*constant* `NCOLOR`*(arm)* 521e⟩≡ (746b)
```
#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` 521f⟩≡ (694)
```
enum Cachectl {
    PG_NOFLUSH = 0,
    PG_TXTFLUSH = 1,  /* flush dcache and invalidate icache */

    PG_NEWCOL = 3,  /* page has been recolored */
};
```

⟨`Page` *other fields* 521g⟩+≡ (89) ◁521a
```
// array<enum<Cachectl>>
char cachectl[MAXCPUS]; /* Cache flushing control for putmmu */
```

⟨`newpage()` *other locals* 521h⟩+≡ (91c) ◁190a 522a▷
```
int color;
// enum<Cachectl>
uchar ct;
```

⟨`newpage()` *find free page p, set color and ct* 521i⟩≡ (91c)
```
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* 522a⟩+≡ (91c) ◁521h
```
int i;
```

⟨newpage() *set cachectl using ct* 522b⟩≡ (91c)
```
for(i = 0; i < MAXCPUS; i++)
    p->cachectl[i] = ct;
```

⟨arch_putmmu() *if* PG_TXTFLUSH, *invalidate cache(arm)* 522c⟩≡ (167)
```
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 521f and PG_TXTFLUSH 521f.

⟨arch__userinit() *set cachectl(arm)* 522d⟩≡ (265d)
```
memset(pg->cachectl, PG_TXTFLUSH, sizeof(pg->cachectl));
```
Uses PG_TXTFLUSH 521f.

⟨Segment *other fields* 522e⟩+≡ (94c) ◁473d 523e▷
```
bool flushme; /* maintain icache for this segment */
```

⟨sysexec() *initialize other fields* 522f⟩≡ (176d)
```
ts->flushme = true;
```

⟨arch__userinit() *set flushme for stack segment* 522g⟩≡ (265d)
```
s->flushme = true;
```
Uses true.

⟨data2txt() *initialize other segment fields* 522h⟩≡ (177c)
```
ps->flushme = true;
```
Uses true.

⟨text2data() *initialize other fields* 522i⟩≡ (458)
```
ps->flushme = true;
```
Uses true.

⟨dupseg() *copy other fields* 522j⟩≡ (103e)
```
n->flushme = s->flushme;
```

⟨syssegflush() *set flushme* 522k⟩≡ (441c)
```
s->flushme = true;
```
Uses true.

⟨pio() *set cachectl if flushme segment* 522l⟩≡ (178)
```
if(s->flushme)
    memset((*p)->cachectl, PG_TXTFLUSH, sizeof((*p)->cachectl));
```
Uses PG_TXTFLUSH 521f.

⟨procctlmemio() *set cachectl if flushme segment* 522m⟩≡ (457)
```
/* Ensure the process sees text page changes */
if(s->flushme)
    memset(pg->cachectl, PG_TXTFLUSH, sizeof(pg->cachectl));
```
Uses PG_TXTFLUSH 521f.

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* 523a⟩≡ (94a) 523b▷
```
SG_SHARED = 04,
```

⟨`Segtype` *other cases* 523b⟩+≡ (94a) ◁523a
```
SG_PHYSICAL = 05,
```

⟨`dupseg()` *switch segment type cases* 523c⟩+≡ (102b) ◁103g
```
case SG_SHARED:
case SG_PHYSICAL:
    goto sameseg;
```

Uses `SG_PHYSICAL` and `SG_SHARED`.

⟨*struct* `Physseg` 523d⟩≡ (694)
```
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* 523e⟩+≡ (94c) ◁522e 526b▷
```
Physseg *pseg;
```

⟨*global* `physseg` 523f⟩≡ (702)
```
/*
 * 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` 523d, `SG_BSS`, and `SG_SHARED`.

⟨*global* `physseglock` 523g⟩≡ (702)
```
static Lock physseglock;
```

⟨*function* addphysseg 524a⟩≡ (702)
```
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()` 58d, `nelem` 692c, `physseg-1804` 523f, `physseglock-1805` 523g, and `unlock()` 60d.

⟨`freept()` *other locals* 524b⟩≡ (96d)
```
int ref;
Page *pt, **ptop;
```

⟨`freept()` `SG_PHYSICAL` *case* 524c⟩≡ (96d)
```
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()` 85b, `lock()` 58d, and `unlock()` 60d.

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* 524d⟩≡ (166b)
```
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 90a, PG_REF 90a, PTEUNCACHED 525a, PTEVALID 166a, PTEWRITE 166a, SG_PHYSICAL, addr2check 700, checkaddr 700, and smalloc() 85c.

⟨PTExxx *other cases* 525a⟩≡ (166a)

```
#define PTEUNCACHED (1<<4)
```

⟨arch_putmmu() *if PTEUNCACHED, do not change x(arm)* 525b⟩≡ (167)

```
if(pa & PTEUNCACHED) {
}
```

Uses PTEUNCACHED 525a.

22.6.3 arch_kmap()

22.6.4 Extra checks

⟨*function* checkpagerefs 525c⟩≡ (701)

```
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() 58d, malloc() 85a, palloc 90d, portcountpagerefs() 526c, print, and unlock() 60d.

⟨*macro* arch_countpagerefs*(arm)* 526a⟩≡ (733b)

```
// same signatures but optimized away
#define arch_countpagerefs(a, b)
```

⟨Segment *other fields* 526b⟩+≡ (94c) ◁523e

```
ulong mark;   /* portcountrefs */
```

⟨*function* portcountpagerefs 526c⟩≡ (701)

```
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 100a, conf 46b, iprint, pagedout 694, pagenumber() 527a, and proctab.

⟨*function* pagenumber 527a⟩≡ (701)

```
ulong
pagenumber(Page *p)
{
    return p - palloc.pages;
}
```

Uses palloc 90d.

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)* 527b⟩+≡ (28a) ◁143a

```
#define PsrMfiq     0x00000011
```

⟨trapnames() *other entries(arm)* 527c⟩≡ (143b)

```
[ PsrMfiq ] "fiq interrupt",
```

⟨PsrDisable *other cases(arm)* 527d⟩≡ (28b)

```
#define PsrDfiq     0x00000040      /* disable FIQ interrupts */
```

⟨*constant* FIQSTKTOP*(arm)* 527e⟩≡ (747)

```
#define FIQSTKTOP   (KZERO+0x4000)      /* FIQ stack */
```

⟨arch__trapinit() *set stack for other exception/processor-modes* 527f⟩+≡ (256e) ◁257b

```
setr13(PsrMfiq, (u32int*)(FIQSTKTOP));
```

Uses FIQSTKTOP 527e and PsrMfiq 527b.

⟨*function* _vfiq*(arm)* 528a⟩≡ (751b)
```
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)* 528b⟩≡ (156b)
```
IRQfiq      = IRQusb,   /* only one source can be FIQ */
```
Uses IRQusb 156b.

⟨irqenable() *if IRQfiq(arm)* 528c⟩≡ (158a)
```
if(irq == IRQfiq){
    ⟨irqenable() sanity check no previous FIQ(arm) 528f⟩
    vfiq = v;
    ip->FIQctl = Fiqenable | irq;
}
```
Uses Fiqenable-2407 528e, IRQfiq 528b, and vfiq-2413 528d.

⟨*global* vfiq*(arm)* 528d⟩≡ (750b)
```
static Vctl *vfiq;
```

⟨*constant* Fiqenable*(arm)* 528e⟩≡ (750a)
```
Fiqenable = 1<<7,
```

⟨irqenable() *sanity check no previous FIQ(arm)* 528f⟩≡ (528c)
```
assert((ip->FIQctl & Fiqenable) == 0);
assert((*enable & v->mask) == 0);
```
Uses Fiqenable-2407 528e and assert 693a.

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)* 529a⟩≡ (750b)

```
/*
 * 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` 528d.

⟨*function* `splfhi`*(arm)* 529b⟩≡ (730a)

```
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)* 529c⟩≡ (722a)

```
enum {
    PowerSd     = 0,
    PowerUart0,
    PowerUart1,
    PowerUsb,
    PowerI2c0,
    PowerI2c1,
    PowerI2c2,
    PowerSpi,
    PowerCcp2tx,
};
```

⟨*function* `setpower`*(arm)* 530a⟩≡ (729)

```
/*
 * 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` 725c, `TagSetpower-2366` 725c, and `vcreq()` 726d.

⟨*function* `getpower`*(arm)* 530b⟩≡ (729)

```
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` 725c, and `vcreq()` 726d.

22.9 Alignements (ARM)

⟨*function* `arch_validalign`*(arm)* 530c⟩≡ (754e)

```
/*
 * 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 631d, NDebug 418c, postnote, and up.

22.10 Real-time scheduling

Plan 9 includes an Earliest Deadline First (EDF) real-time scheduler, configurable through /proc/x/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* 531a⟩≡ (198b)

```
  PriRelease  = Npriq,  /* released edf processes */
  PriEdf    = Npriq+1,  /* active edf processes */
  PriExtra  = Npriq-1,  /* edf processes at high best-effort pri */
```

Uses Npriq 198a.

⟨runproc() *test for empty real-time scheduling queue* 531b⟩≡ (211d)

```
  runq[Nrq-1].head == nil && runq[Nrq-2].head == nil
```

Uses Nrq 200a and runq 200b.

⟨runproc() *test if p is a real-time process* 531c⟩≡ (204)

```
  if(edflock(p)){
      edfrun(p, rq == &runq[PriEdf]); /* start deadline timer and do admin */
      edfunlock();
  }
```

Uses PriEdf 531a, edflock() 535b, edfrun() 535b, edfunlock() 535b, and runq 200b.

⟨newproc() *set fields of* p 531d⟩+≡ (111d) ◁480c 567f▷

```
  p->edf = nil;
```

⟨updatecpu() *return if real-time process* 531e⟩≡ (218h)

```
  if(p->edf)
      return;
```

⟨schedinit() *locals* 531f⟩≡ (203f)

```
  Edf *e;
```

⟨schedinit() *optional real-time* edfrecord() 531g⟩≡ (207a)

```
  if((e = up->edf) && (e->flags & Admitted))
      edfrecord(up);
```

Uses Admitted 533g, edfrecord() 535b, and up.

⟨schedinit() *optional real-time* edfstop() 531h⟩≡ (136a)

```
  edfstop(up);
  if (up->edf)
      free(up->edf);
  up->edf = nil;
```

Uses edfstop() 535b, free() 85b, and up.

⟨ready() *optional* edfready() *for real-time scheduling* 531i⟩≡ (211a)

```
  if(edfready(p)){
      arch_splx(s);
      return;
  }
```

Uses edfready() 535b.

⟨syssleep() *optional* edfyield() *for real-time scheduling* 532a⟩≡ (232c)

```
if (up->edf && (up->edf->flags & Admitted))
    edfyield();
else
```

Uses Admitted 533g, edfyield() 535b, and up.

EDF parameters are set by writing commands to /proc/x/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* 532b⟩≡ (586b)

```
/* 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 568b, CMcost-898 568b, CMdeadline-897 568b, CMdeadlinenotes-900 568b, CMexpel-903 568b, CMextra-902 568b, CMperiod-896 568b, CMsporadic-899 568b, Extratime 533g, Sendnotes 533g, Sporadic 533g, edfadmit() 535b, edfinit() 535b, edfstop() 535b, and parsetime() 586a.

⟨sched() *if real-time process then do nothing, do not reprioritize* 533a⟩≡ (217c)
```
if(p->edf) {
}
```

⟨pexit() *optional* edfstop() *for real-time scheduling* 533b⟩≡ (134c)
```
edfstop(up);
```
Uses edfstop() 535b and up.

⟨Proc *scheduling fields* 533c⟩+≡ (49f) ◁218c 533d▷
⟨Proc *optional* edf *field for real-time scheduling* 533e⟩

⟨Proc *scheduling fields* 533d⟩+≡ (49f) ◁533c 712a▷
```
ulong readytime;  /* time process came ready */
```

⟨Proc *optional* edf *field for real-time scheduling* 533e⟩≡ (533c)
```
// 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* 533f⟩≡ (58d)
```
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 533g, conf 46b, print, todget() 239b, and up.

⟨*enum* edfflags 533g⟩≡ (705c)
```
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` 534a⟩≡ (705c)

```
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 statistics* 534b⟩≡ (535b)

```
/* Statistics stuff */
ulong       edfnrun;
//ulong     nilcount;
//ulong     scheds;
//int       misseddeadlines;
```

⟨Procstate *cases* 535a⟩+≡ (51b) ◁453a
```
Waitrelease, // for real-time scheduling
```

⟨processes/edf.c 535b⟩≡
```
/* EDF scheduling */
⟨kernel basic includes 35a⟩

#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 ??⟩

enum
{
    Maxsteps = 200 * 100 * 2, /* 100 periods of 200 procs */
};


static long now;    /* Low order 32 bits of time in µs */


⟨edf.c statistics 534b⟩

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 533g, DPRINT-2593, Dead 51b, Dl-2597 535b, Extratime 533g, Maxsteps-2594 535b, OneRound-2603 535b, Onemillisecond-2601 535b, Onesecond-2602 535b, PriEdf 531a, PriExtra 531a, Ready 210c, Rl-2598 535b, Running 51b, Sporadic 533g, Trelative 225b, Waitrelease 535a, Wakeme 212d, Yield 533g, Yieldonblock 533g, active 49e, assert 693a, conf 46b, deadlineintr() 535b, delayedscheds 629b, dequeueproc() 201c, edfinited 535b, edflock() 535b, edfnrun 534b, edfrun() 535b, edfschedlock 535b, edfunlock() 535b, ilock() 61d, iprint, iunlock() 62b, lock() 58d, malloc() 85a, now-2595 535b, nrdy 201a, panicking 630a, print, proctab, proctrace 467a, qlock() 63f, qschedulability-2599 535b, qunlock() 64c, ready, release() 535b, releaseintr() 535b, runq 200b, runvec 200e, sleep, statename 51c, testenq() 535b, thelock-2596 535b, timeconv() 535b, timeradd() 227d, timerdel() 228c, todget() 239b, unlock() 60d, up, wakeup, and yfn() 535b.

Tracing scheduling events

⟨enum Tevent *real-time scheduling events* 546⟩≡ (467b)

```
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 547a⟩≡ (682c)
```
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 547b⟩≡ (682c)
```
/*
 * 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 547d.

⟨*global* mntalloc 547c⟩≡ (688c)
```
struct Mntalloc mntalloc;
```
Uses Mntalloc 547b.

⟨*constants tags* 547d⟩≡ (682c)
```
enum
{
    TAGSHIFT = 5,           /* ulong has to be 32 bits */
    TAGMASK = (1<<TAGSHIFT)-1,
    NMASK = (64*1024)>>TAGSHIFT,
};
```
Uses TAGSHIFT 547d.

⟨*function* `freetag` 548a⟩≡ (688c)

```
void
freetag(int t)
{
    mntalloc.tagmask[t>>TAGSHIFT] &= ~(1<<(t&TAGMASK));
}
```

Uses `TAGMASK` 547d, `TAGSHIFT` 547d, and `mntalloc` 547c.

⟨*function* `mntfree` 548b⟩≡ (688c)

```
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()` 85b, `freeblist()` 409b, `freetag()` 548a, `lock()` 58d, `mntalloc` 547c, and `unlock()` 60d.

⟨*function* `mntpntfree` 548c⟩≡ (688c)

```
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()` 58d, `mntalloc` 547c, `qfree()`, and `unlock()` 60d.

⟨*function* `muxclose` 548d⟩≡ (688c)

```
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()` 85b, `mntfree()` 548b, and `mntpntfree()` 548c.

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` 549a⟩≡ (682c)

```
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* 549b⟩+≡ (272b) ◁298d ??▷

```
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` 549c⟩≡ (688c)

```
/*
 * 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 549b, Eshort 631d, MAXRPC-782 688c, exhausted() 633c, free() 85b, kstrdup() 636b, lock() 58d, malloc() 85a, mntalloc 547c, nexterror, poperror 633d, qlock() 63f, qopen() 380a, qunlock() 64c, unlock() 60d, and waserror 633d.

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.

⟨*function* mountrpc 552⟩≡ (? 0—1)

```
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 631d, mountio() 553, print, and up.

⟨*function* mountio 553⟩≡ (? 0—1)

```
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 631d, arch_fastticks, lock() 58d, mntflushalloc(), mntflushfree(), mntgate(), mntrpcread(), mountmux(), nexterror, poperror 633d, rpcattn(), sleep, unlock() 60d, up, and waserror 633d.

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 554⟩≡ (? 0—1)

```
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 682c, CCACHE, Enoversion 631d, MCACHE 338b, chanfree() 274a, incref() 69b, mntchan(), mntfree() 548b, mntralloc(), mntversion() 549c, mountrpc() 552, nexterror, poperror 633d, up, and waserror 633d.

mntwalk()

⟨*function* mntwalk 555⟩≡ (? 0—1)

```
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, cclose, devclone() 400a, free() 85b, incref() 69b, mntchk(), mntfree() 548b, mntralloc(), mountrpc() 552, nexterror, poperror 633d, print, smalloc() 85c, and waserror 633d.

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` 557a⟩≡ (646b)
```
char    *eve;
```

⟨*function* `iseve` 557b⟩≡ (646b)
```
bool iseve(void) {
  return strcmp(eve, up->user) == 0;
}
```
Uses `eve` 557a and `up`.

⟨`systab` *security syscalls* 557c⟩≡ (41f)
```
[FAUTH]     sysfauth,
[FVERSION]  sysfversion,
```
Uses `sysfauth()` 558a and `sysfversion()` 557e.

⟨*global* `hostdomain` 557d⟩≡ (718)
```
char    hostdomain[DOMLEN];
```

⟨*syscall fversion* 557e⟩≡ (718)
```
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` 631d, `ORDWR` 279c, `cclose`, `false`, `fdtochan()` 280c, `mntversion()` 549c, `nexterror`, `poperror` 633d, `true`, `validaddr()` 101c, and `waserror` 633d.

⟨*syscall fauth* 558a⟩≡ (718)

```
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 297b, Enofd 631d, ORDWR 279c, cclose, false, fdtochan() 280c, free() 85b, mntauth(), newfd() 277c, nexterror, poperror 633d, true, validaddr() 101c, validnamedup() 285d, and waserror 633d.

⟨*function* userwrite 558b⟩≡ (718)

```
/*
 *  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 198b, kstrdup() 636b, and up.

⟨*function* hostdomainwrite 559⟩≡ (718)

```
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 631d, hostdomain 557d, and iseve() 557b.

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` 560a⟩≡ (704a)
```
struct Cmdtab
{
  int index;  /* used by client to switch on result */
  char  *cmd; /* command name */
  int narg; /* expected #args; 0 ==> variadic */
};
```

⟨*struct* `Cmdbuf` 560b⟩≡ (704a)
```
struct Cmdbuf
{
  char  *buf;
  char  **f;
  int nf;
};
```

The `parsecmd`[560c] 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` 560c⟩≡ (704b)
```
/*
 *  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() 85b, ncmdfield() 562, nexterror, poperror 633d, smalloc() 85c, up, and waserror 633d.

⟨*function* cmderror 561a⟩≡ (704b)

```
/*
 * 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 692a and up.

The lookupcmd[561b] 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 561b⟩≡ (704b)

```
/*
 * 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` 631d and `cmderror()` 561a.

⟨*ncmdfield* 562⟩≡ (704b)

```
/*
 * 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` 563a⟩≡ (682c)
```
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` 563b⟩≡ (682c)
```
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* 563c⟩+≡ (49f) ◁481d 567c▷
```
// ref_counted<Egrp>
Egrp  *egrp;    /* Environment group */
```

⟨*function* `closeegrp` 563d⟩≡ (688a)
```
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() 69c and free() 85b.

⟨*function* envcpy 564a⟩≡ (674)

```
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() 65e, runlock() 66b, and smalloc() 85c.

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 564b⟩≡ (674)

```
/*
 *  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 282e, KNAMELEN 692b, OWRITE 279c, cclose, and namec() 283d.

⟨*global* envdevtab 565⟩≡ (674)

```
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() 405a, devbwrite() 405b, devinit() 686, devreset() 686, devshutdown() 686, devwstat() 686, envclose() 674, envcreate() 674, envopen() 674, envread() 674, envremove() 674, envstat() 674, envwalk() 674, and envwrite() 674.

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/x/ctl

Writing text commands to /proc/\emphpid/ctl controls a process: "kill", "stop", "start", "hang", "private", and others. The commands are parsed by procctlreq[586b] 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/x/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/x/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/x/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/x/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* 566a⟩+≡ (49f) ◁133b
```
QLock qwaitr;
```

⟨procread() *cases* 566b⟩+≡ (571d) ◁474c 574a▷
```
case Qwait:
    if(!canqlock(&p->qwaitr))
        error(Einuse);

    if(waserror()) {
        qunlock(&p->qwaitr);
        nexterror();
    }

    ⟨procread() when Qwait case, sanity check process has children 566c⟩
    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 473c⟩
        wq->w.msg);
    free(wq);
    return n;
```
Uses `Eprocdied` 631d, `Qwait-876` 568a, `canqlock()` 65a, `free()` 85b, `haswaitq()` 133d, `lock()` 58d, `nexterror`, `poperror` 633d, `qunlock()` 64c, `sleep`, `unlock()` 60d, and `waserror` 633d.

⟨procread() *when Qwait case, sanity check process has children* 566c⟩≡ (566b)
```
// same in sysawait
lock(&p->exl);
if(up == p && p->nchild == 0 && p->waitq == 0) {
    unlock(&p->exl);
    error(Enochild);
}
```
Uses `Enochild` 631d, `lock()` 58d, `unlock()` 60d, and `up`.

⟨pwait() *lock qwaitr* 567a⟩≡ (132c)
```
if(!canqlock(&up->qwaitr))
    error(Einuse); // someone is reading /proc/pid/wait?
if(waserror()) {
    qunlock(&up->qwaitr);
    nexterror();
}
```
Uses canqlock() 65a, nexterror, qunlock() 64c, up, and waserror 633d.

⟨pwait() *unlock qwaitr* 567b⟩≡ (132c)
```
qunlock(&up->qwaitr);
poperror();
```
Uses poperror 633d, qunlock() 64c, and up.

23.3.6 Other /proc/x/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* 567c⟩+≡ (49f) ◁563c 567e▷
```
ulong procmode; /* proc device default file mode */
```

⟨sysrfork() *when* RFPROC *propagate fields from up to p* 567d⟩+≡ (116b) ◁481e 567g▷
```
p->procmode = up->procmode;
```
Uses up.

⟨Proc *other fields* 567e⟩+≡ (49f) ◁567c
```
bool privatemem; /* proc does not let anyone read mem */
```

⟨newproc() *set fields of* p 567f⟩+≡ (111d) ◁531d 584b▷
```
p->privatemem = false;
```
Uses false.

⟨sysrfork() *when* RFPROC *propagate fields from up to p* 567g⟩+≡ (116b) ◁567d
```
p->privatemem = up->privatemem;
```
Uses up.

⟨sysexec() *when hold debug lock, reset some fields* 567h⟩+≡ (123b) ◁420c
```
up->privatemem = false;
```

⟨procctlreq() *CMprivate case* 567i⟩≡ (586b)
```
case CMprivate:
    p->privatemem = true;
    break;
```
Uses CMprivate-885 568b and true.

⟨*devproc.c enum Qxxx* 568a⟩≡ (680)

```
enum
{
    Qdir,
    Qtrace,

    Qargs,
    Qctl,
    Qfd,
    Qfpregs,
    Qkregs,
    Qmem,
    Qnote,
    Qnoteid,
    Qnotepg,
    Qns,
    Qproc,
    Qregs,
    Qsegment,
    Qstatus,
    Qtext,
    Qwait,
    Qprofile,
    Qsyscall,
};
```

⟨*devproc enum CMxxx* 568b⟩≡ (680)

```
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` 569a⟩≡ (680)

```
/*
 * 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` 568a, `Qctl-862` 568a, `Qfd-863` 568a, `Qfpregs-864` 568a, `Qkregs-865` 568a, `Qmem-866` 568a, `Qnote-867` 568a, `Qnoteid-868` 568a, `Qnotepg-869` 568a, `Qns-870` 568a, `Qproc-871` 568a, `Qprofile-877` 568a, `Qregs-872` 568a, `Qsegment-873` 568a, `Qstatus-874` 568a, `Qsyscall-878` 568a, `Qtext-875` 568a, `Qwait-876` 568a, and `STATSIZE-907` 680.

⟨*global* `proccmd` 569b⟩≡ (680)

```
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 568b, CMclose-879 568b, CMclosefiles-880 568b, CMcost-898 568b, CMdeadline-897 568b, CMdeadlinenotes-900 568b, CMevent-904 568b, CMexpel-903 568b, CMextra-902 568b, CMfixedpri-888 568b, CMhang-881 568b, CMkill-882 568b, CMnohang-883 568b, CMnoswap-884 568b, CMperiod-896 568b, CMpri-887 568b, CMprivate-885 568b, CMprofile-886 568b, CMsporadic-899 568b, CMstart-890 568b, CMstartstop-891 568b, CMstartsyscall-892 568b, CMstop-893 568b, CMtrace-895 568b, CMwaitstop-894 568b, and CMwired-889 568b.

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* 570a⟩≡ (680)

```
/*
 * 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` 570b⟩≡ (712c)

```
Proc*
proc_proctab(int i)
{
    return &procalloc.arena[i];
}
```

Uses `procalloc-2458` 110c.

⟨`Chan` *union other fields* 570c⟩+≡ (51e) ◁428e 571b▷

```
Qid pgrpid;   /* for #p/notepg */
```

⟨*method procinit* 570d⟩≡ (680)

```
static void
proc_init(void)
{
    if(conf.nproc >= (1<<(31-QSHIFT))-1)
        print("warning: too many procs for devproc\n");

    ⟨proc_init() other init 473g⟩
}
```

Uses `QSHIFT-910` 570a, `conf` 46b, and `print`.

⟨*global* `procdevtab` 571a⟩≡ (680)

```
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()` 405a, `devbwrite()` 405b, `devcreate()` 686, `devremove()` 686, `devreset()` 686, `devshutdown()` 686, `proc_init()` 570d, `procattach()` 680, `procclose()` 583, `procopen()` 577, `procread()` 571d, `procstat()` 680, `procwalk()` 680, `procwrite()` 574c, and `procwstat()` 579.

⟨`Chan` *union other fields* 571b⟩+≡ (51e) ◁570c

```
ulong mid;    /* for ns in devproc */
```

⟨`procread()` *locals* 571c⟩+≡ (571d) ◁325b

```
/* 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` 100a.

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* 571d⟩≡ (680)

```
static long
procread(Chan *c, void *va, long n, vlong off)
{
    ⟨procread() locals 104a⟩

    a = va;
    offset = off;

    if(c->qid.type & QTDIR)
        return devdirread(c, a, n, 0, 0, procgen);
```

⟨procread() *Qtrace if* 469c⟩

```
p = proctab(SLOT(c->qid));
if(p->pid != PID(c->qid))
    error(Eprocdied);

switch(QID(c->qid)){
```

⟨procread() *cases* 104b⟩

```
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 495b⟩

    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 472c⟩

        /* 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 631d, Enoreg 631d, Eprocdied 631d, KNAMELEN 692b, NSEG 100a, NUMSIZE 704a, PID-913 570a, QID-911 570a, QTDIR 52c, Qargs-861 568a, Qfd-863 568a, Qnote-867 568a, Qnoteid-868 568a, Qproc-871 568a, Qregs-872 568a, Qstatus-874 568a, SLOT-912 570a, STATSIZE-907 680, devdirread() 403c, false, nexterror, poperror 633d, procargs() 584c, procfds() 582b, procgen() 580d, proctab, qlock() 63f, qunlock() 64c, readnum() 637c, readstr() 637d, statename 51c, up, and waserror 633d.

⟨procread() *cases* 574a⟩+≡ (571d) ◁566b

```
    case Qkregs:
        memset(&kur, 0, sizeof(Ureg));
        arch_setkernur(&kur, p);
        rptr = (byte*)&kur;
        rsize = sizeof(Ureg);
        goto regread;
```

Uses Qkregs-865 568a.

⟨*function* arch_setkernur*(arm)* 574b⟩≡ (754e)

```
    /* 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 41a 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* 574c⟩≡ (680)

```
    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* 456d⟩

```
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* 495c⟩

```
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 51b, ERRMAX 692a, Ebadarg 631d, Egreg 631d, Enoreg 631d, Eprocdied 631d, Eshort 631d, Etoobig 631d, NOTEID-914 570a, NUser 418c, PID-913 570a, QID-911 570a, QTDIR 52c, Qargs-861 568a, Qctl-862 568a, Qnote-867 568a, Qnoteid-868 568a, Qnotepg-869 568a, Qregs-872 568a, SLOT-912 570a, conf 46b, free() 85b, malloc() 85a, nexterror, pgrpnote() 576, poperror 633d, postnote, pprint, procctlreq() 586b, proctab, qlock() 63f, qunlock() 64c, true, and waserror 633d.

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 576⟩≡ (710)

```
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 51b, ERRMAX 692a, Etoobig 631d, conf 46b, poperror 633d, postnote, proctab, qlock() 63f, qunlock() 64c, up, and waserror 633d.

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* 577⟩≡ (680)

```
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 469a⟩

    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 325c⟩
    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 51b, Egreg 631d, Eprocdied 631d, OREAD 279c, OWRITE 279c, PID-913 570a, QID-911 570a, QTDIR 52c, Qargs-861 568a, Qctl-862 568a, Qfd-863 568a, Qfpregs-864 568a, Qkregs-865 568a, Qmem-866 568a, Qnote-867 568a, Qnoteid-868 568a, Qnotepg-869 568a, Qproc-871 568a, Qprofile-877 568a, Qregs-872 568a, Qsegment-873 568a, Qstatus-874 568a, Qsyscall-878 568a, Qtext-875 568a, Qwait-876 568a, SLOT-912 570a, arch_coherence, cclose, nexterror, nonone() 582a, openmode() 280a, poperror 633d, pprint, procgen() 580d, proctab, proctext() 584d, qlock() 63f, qunlock() 64c, and waserror 633d.

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* 579⟩≡ (680)

```
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 631d, Eshortstat 631d, PID-913 570a, QID-911 570a, QTDIR 52c, Qtrace-860 568a, SLOT-912 570a, devwstat() 686, emptystr() 637a, eve 557a, free() 85b, kstrdup() 636b, nexterror, nonone() 582a, poperror 633d, proctab, qlock() 63f, qunlock() 64c, smalloc() 85c, up, and waserror 633d.

⟨*function* procoffset 580a⟩≡ (680)

```
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 580b⟩≡ (680)

```
static int
procqidwidth(Chan *c)
{
    char buf[32];

    return snprint(buf, sizeof buf, "%lud", c->qid.vers);
}
```

⟨*function* procfdprint 580c⟩≡ (680)

```
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() 580b.

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 580d⟩≡ (680)

```
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 474d⟩
    }

    mkqid(&qid, path|tab->qid.path, c->qid.vers, QTFILE);
    devdir(c, qid, tab->name, len, p->user, perm, dp);
    return 1;
}
```

Uses DEVDOTDOT, DMDIR 313a, QID-911 570a, QSHIFT-910 570a, QTDIR 52c, QTFILE 52c, Qdir-859 568a, Qtrace-860 568a, Qwait-876 568a, SLOT-912 570a, conf 46b, devdir() 398c, eve 557a, nelem 692c, procdir 569a, procindex() 111b, 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 582a⟩≡ (680)

```
/*
 *  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[582b] 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 582b⟩≡ (680)

```
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() 58d, nexterror, poperror 633d, procfdprint() 580c, procoffset() 580a, procqidwidth() 580b, qlock() 63f, qunlock() 64c, readstr() 637d, unlock() 60d, and waserror 633d.

⟨*method procclose* 583⟩≡ (680)

```
static void
procclose(Chan* c)
{
    ⟨procclose() Qtrace if 469b⟩
    ⟨procclose() hooks 325d⟩
```

```
}
```

⟨Proc *debugger fields* 584a⟩+≡ (49f) ◁466b
```
bool setargs;
```

⟨newproc() *set fields of* p 584b⟩+≡ (111d) ◁567f 632c▷
```
p->setargs = false;
```
Uses false.

⟨*function* procargs 584c⟩≡ (680)
```
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[584d] 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 584d⟩≡ (680)
```
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 272b, Dead 51b, Eprocdied 631d, OREAD 279c, PID-913 570a, TSEG 100a, cclose, incref() 69b, lock() 58d, nexterror, poperror 633d, unlock() 60d, and waserror 633d.

⟨*function* procctlcloseone 585a⟩≡ (680)

```
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() 58d, qlock() 63f, qunlock() 64c, and unlock() 60d.

⟨*function* procctlclosefiles 585b⟩≡ (680)

```
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 631d, closefgrp() 277b, lock() 58d, procctlcloseone() 585a, and unlock() 60d.

⟨*function* parsetime 586a⟩≡ (680)

```
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 692c.

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 586b⟩≡ (680)

```
// 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 199e⟩
    ⟨procctlreq() CMfixedpri case 199f⟩
    ⟨procctlreq() CMwired case 216f⟩
    ⟨procctlreq() CMnoswap case 193e⟩

    ⟨procctlreq() CMprivate case 567i⟩

    ⟨procctlreq() CMhang case 455d⟩
    ⟨procctlreq() CMnohang case 455e⟩

    ⟨procctlreq() CMstop case 453e⟩
    ⟨procctlreq() CMstart case 454a⟩
    ⟨procctlreq() CMwaitstop case 455h⟩

    ⟨procctlreq() CMstartsyscall case 459b⟩
    ⟨procctlreq() CMstartstop case 470c⟩

    ⟨procctlreq() CMtrace case 466c⟩
    ⟨procctlreq() CMevent case 468c⟩

    ⟨procctlreq() CMprofile case 474b⟩


    ⟨procctlreq() CMkill case 141c⟩

    case CMclose:
        procctlclosefiles(p, false, atoi(cb->f[1]));
        break;
    case CMclosefiles:
        procctlclosefiles(p, true, 0);
        break;


    ⟨procctlreq() optional real-time commands 532b⟩

    }

    poperror();
    free(cb);
}
```

Uses CMclose-879 568b, CMclosefiles-880 568b, false, free() 85b, lookupcmd() 561b, nelem 692c, nexterror, parsecmd() 560c, poperror 633d, proccmd-908 569b, procctlclosefiles() 585b, true, and waserror 633d.

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 dup() system call below provides programmatic access to the same functionality.

23.4.1 sysdup()

⟨systab *special file syscalls* 588a⟩≡ (41f)
```
[DUP]       sysdup,
```
Uses sysdup() 588b.

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* 588b⟩≡ (690)
```
// 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 631d, Enofd 631d, cclose, false, fdtochan() 280c, growfd() 278b, lock() 58d, newfd() 277c, nexterror, poperror 633d, true, unlockfgrp() 279b, up, and waserror 633d.

23.4.2 /fd/

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* 589a⟩≡ (672e)
```
enum{
    Qdir,

    ⟨devsys.c enum Qxxx cases 591a⟩
};
```

⟨*global* sysdir 589b⟩≡ (672e)
```
static Dirtab sysdir[]={
    ".",    {Qdir, 0, QTDIR},   0,      DMDIR|0555,

    ⟨sysdir fields 591b⟩
};
```
Uses DMDIR 313a, QTDIR 52c, and Qdir-836 589a.

⟨*method sysinit* 589c⟩≡ (672e)
```
static void
sysinit(void)
{
}
```

⟨*method sysattach* 589d⟩≡ (672e)
```
static Chan*
sysattach(char *spec)
{
    return devattach('k', spec);
}
```
Uses devattach() 275a.

⟨*method sysread* 589e⟩≡ (672e)
```
static long
sysread(Chan *c, void *buf, long n, vlong off)
{
    vlong offset = off;
    char tmp[256];
    ⟨sysread() locals 590a⟩

    if(n <= 0)
        return n;

    switch((ulong)c->qid.path){
    case Qdir:
        return devdirread(c, buf, n, sysdir, nelem(sysdir), devgen);
```

```
    ⟨sysread() cases 591c⟩

    default:
        print("sysread %#llux\n", c->qid.path);
        error(Egreg);
    }
    return -1;      /* never reached */
}
```

Uses Egreg 631d, Qdir-836 589a, devdirread() 403c, devgen() 399, nelem 692c, print, and sysdir-845 589b.

⟨sysread() *locals* 590a⟩≡ (589e)

```
    char *b, *bp;
    int i, k;
    Cpu *mp;
    int id;
```

⟨*method syswrite* 590b⟩≡ (672e)

```
static long
syswrite(Chan *c, void *va, long n, vlong off)
{
    ulong offset;
    char *a;
    ⟨syswrite() locals 590c⟩

    offset = off;
    a = va;

    switch((ulong)c->qid.path){

    ⟨syswrite() cases 592b⟩

    default:
        print("syswrite: %#llux\n", c->qid.path);
        error(Egreg);
    }
    return n;
}
```

Uses Egreg 631d and print.

⟨syswrite() *locals* 590c⟩≡ (590b) 593f▷

```
    char buf[256];
    Cpu *mp;
    int id;
```

⟨*global* sysdevtab 590d⟩≡ (672e)

```
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() 405a, devbwrite() 405b, devcreate() 686, devremove() 686, devreset() 686, devshutdown() 686, and devwstat() 686.

23.5.1 /sys/osversion

⟨*devsys.c enum Qxxx cases* 591a⟩≡ (589a) 591e▷

```
Qosversion,
```

⟨sysdir *fields* 591b⟩≡ (589b) 591f▷

```
"osversion",    {Qosversion},    0,      0444,
```

Uses Qosversion-837 591a.

⟨sysread() *cases* 591c⟩≡ (589e) 591g▷

```
case Qosversion:
    snprint(tmp, sizeof tmp, "pad's version");
    n = readstr((ulong)offset, buf, n, tmp);
    return n;
```

Uses Qosversion-837 591a and readstr() 637d.

23.5.2 /sys/config

⟨*devsys.c decls and globals* 591d⟩≡ (672e) 593e▷

```
extern byte configfile[]; // in $CONF.c
```

⟨*devsys.c enum Qxxx cases* 591e⟩+≡ (589a) ◁591a 591h▷

```
Qconfig,
```

⟨sysdir *fields* 591f⟩+≡ (589b) ◁591b 591i▷

```
"config",    {Qconfig},    0,      0444,
```

Uses Qconfig-838 591e.

⟨sysread() *cases* 591g⟩+≡ (589e) ◁591c 592a▷

```
case Qconfig:
    return readstr((ulong)offset, buf, n, (char*) configfile);
```

Uses Qconfig-838 591e and readstr() 637d.

23.5.3 /ys/hosthowner, /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* 591h⟩+≡ (589a) ◁591e 592d▷

```
Qhostowner,
Qhostdomain,
```

⟨sysdir *fields* 591i⟩+≡ (589b) ◁591f 592e▷

```
"hostowner",    {Qhostowner},    0,      0664,
"hostdomain",   {Qhostdomain},   DOMLEN,     0664,
```

Uses Qhostdomain-840 591h and Qhostowner-839 591h.

⟨sysread() *cases* 592a⟩+≡ (589e) ◁591g 592f▷
```
  case Qhostowner:
      return readstr((ulong)offset, buf, n, eve);

  case Qhostdomain:
      return readstr((ulong)offset, buf, n, hostdomain);
```
Uses Qhostdomain-840 591h, Qhostowner-839 591h, eve 557a, hostdomain 557d, and readstr() 637d.

⟨syswrite() *cases* 592b⟩≡ (590b) 592g▷
```
  case Qhostowner:
      return hostownerwrite(a, n);

  case Qhostdomain:
      return hostdomainwrite(a, n);
```
Uses Qhostdomain-840 591h, Qhostowner-839 591h, hostdomainwrite() 559, and hostownerwrite() 114d.

23.5.4 /sys/sysname

⟨*global* sysname 592c⟩≡ (672e)
```
  char    *sysname;
```

⟨*devsys.c enum Qxxx cases* 592d⟩+≡ (589a) ◁591h 592h▷
```
  Qsysname,
```

⟨sysdir *fields* 592e⟩+≡ (589b) ◁591i 593a▷
```
  "sysname",  {Qsysname}, 0,      0664,
```
Uses Qsysname-841 592d.

⟨sysread() *cases* 592f⟩+≡ (589e) ◁592a 593b▷
```
  case Qsysname:
      if(sysname == nil)
          return 0;
      return readstr((ulong)offset, buf, n, sysname);
```
Uses Qsysname-841 592d, readstr() 637d, and sysname 592c.

⟨syswrite() *cases* 592g⟩+≡ (590b) ◁592b 594a▷
```
  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 631d, Qsysname-841 592d, kstrdup() 636b, and sysname 592c.

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* 592h⟩+≡ (589a) ◁592d 593c▷
```
  Qdrivers,
```

⟨sysdir *fields* 593a⟩+≡ (589b) ◁592e 593d▷
```
  "drivers",  {Qdrivers}, 0,      0444,
```
Uses Qdrivers-842 592h.

⟨sysread() *cases* 593b⟩+≡ (589e) ◁592f 626a▷
```
  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 592h, READSTR 704a, free() 85b, malloc() 85a, nexterror, poperror 633d, readstr() 637d, and waserror 633d.

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* 593c⟩+≡ (589a) ◁592h 625d▷
```
  Qreboot,
```

⟨sysdir *fields* 593d⟩+≡ (589b) ◁593a 625e▷
```
  "reboot",   {Qreboot},  0,      0660,
```
Uses Qreboot-843 593c.

⟨*devsys.c decls and globals* 593e⟩+≡ (672e) ◁591d
```
  enum
  {
      CMhalt,
      CMreboot,
      CMpanic,
  };

  Cmdtab rebootmsg[] =
  {
      CMhalt,     "halt",     1,
      CMreboot,   "reboot",   0,
      CMpanic,    "panic",    0,
  };
```
Uses CMhalt-846 593e, CMpanic-848 593e, and CMreboot-847 593e.

⟨syswrite() *locals* 593f⟩+≡ (590b) ◁590c
```
  Cmdbuf *cb;
  Cmdtab *ct;
```

⟨syswrite() *cases* 594a⟩+≡ (590b) ◁592g 626b▷

```
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 593e, CMpanic-848 593e, CMreboot-847 593e, Qreboot-843 593c, free() 85b, iseve() 557b, lookupcmd() 561b, nelem 692c, nexterror, parsecmd() 560c, poperror 633d, rebootmsg 593e, and waserror 633d.

⟨kbdputsc() *reboot if ctl-alt-del* 594b⟩≡ (350d)

```
if(kbscan->ctl)
    if(kbscan->alt && c == Del) // Ctl-Alt-Del
        arch_exit(0);
```

Uses Del-983 348a and arch_exit.

⟨ConsKbd *other fields* 594c⟩+≡ (353d) ◁368a

```
bool ctlpoff; // ^P will not reboot if true
```

⟨conswrite() *Qconsctl other ifs* 594d⟩≡ (368f)

```
else if(strncmp(a, "ctlpon", 6) == 0){
    kbd.ctlpoff = false;
} else if(strncmp(a, "ctlpoff", 7) == 0){
    kbd.ctlpoff = true;
}
```

Uses false, kbd-1778 354a, and true.

⟨echo() *special key C-p* 594e⟩≡ (609c)

```
case 0x10:  /* ^P */
    if(cpuserver && !kbd.ctlpoff){
        active.exiting = true;
        return;
    }
    break;
```

Uses active 49e, kbd-1778 354a, and true.

⟨Active *other fields* 594f⟩≡ (49d) 596a▷

```
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` 595a⟩≡ (684)
```
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* 595b⟩+≡ (54a) ◁404b
```
void  (*shutdown)(void);
```

`arch_exit()`(ARM)

⟨*function* `main_arch_exit`*(arm)* 595c⟩≡ (742b)
```
/*
 *  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()` 771b and `intrcpushutdown()` 598a.

⟨*constant* `REBOOTADDR`*(arm)* 595d⟩≡ (747)
```
/* address at which to copy and execute rebootcode */
#define REBOOTADDR  (KZERO+0x1800)
```

⟨*function* `shutdown`*((init/arm/main.c)(arm))* 595e⟩≡ (742b)
```
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 223a, active 49e, arch_delay, consactive() 620b, iprint, lock() 58d, and unlock() 60d.

⟨Active *other fields* 596a⟩+≡ (49d) ◁594f

```
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)* 596b⟩≡ (742b)

```
/*
 * 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 49e, arch_delay, archreboot() 771b, chandevshutdown() 595a, clockshutdown() 597b, intrshutdown() 597a, iprint, kprintoq 618e, poperror 633d, print, procwired() 216g, sched, screenputs 356b, serialoq 620a, up, and waserror 633d.

⟨*function* intrshutdown*(arm)* 597a⟩≡ (750b)

```
/* called from cpu0 after other cpus are shutdown */
void
intrshutdown(void)
{
    intrsoff();
    intrcpushutdown();
}
```

Uses intrcpushutdown() 598a and intrsoff() 257a.

⟨*function* clockshutdown*(arm)* 597b⟩≡ (753)

```
void
clockshutdown(void)
{
    Armtimer *tm;

    tm = (Armtimer*)ARMTIMER;
    tm->ctl = 0;
    //wdogoff();
}
```

Uses ARMTIMER-2648 31f.

⟨*function* intrcpushutdown*(arm)* 598a⟩≡ (750b)
```
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 520g, Localmboxint-2409 520f, and Localtimerint-2408 520d.

rebootcode.s (arm)

⟨*constant* PTEDRAM*(arm)* 598b⟩≡ (746a)
```
#define PTEDRAM     (Dom0|L1AP(Krw)|Section)
```

⟨*instruction WFI(arm)* 598c⟩≡ (746a)
```
#define WFI WORD    $0xe320f003 /* wait for interrupt */
```

⟨*instruction WFE(arm)* 598d⟩≡ (746a)
```
#define WFE WORD    $0xe320f002 /* wait for event */
```

⟨*function main(rebootcode.s)(arm)* 598e⟩≡ (746a)
```
/*
 * 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* 600a⟩≡ (359c) 617f▷

```
Qbintime,
Qcputime,
Qnull,
Qpgrpid,
Qpid,
Qppid,
Qrandom,
Qswap,
Qtime,
Quser,
Qzero,
```

⟨consdir *fields* 600b⟩≡ (360a) 617g▷

```
"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 704a, Qbintime-1783 600a, Qcputime-1784 600a, Qnull-1785 600a, Qpgrpid-1786 600a, Qpid-1787 600a, Qppid-1788 600a, Qrandom-1789 600a, Qswap-1790 600a, Qtime-1791 600a, Quser-1792 600a, Qzero-1793 600a, and VLNUMSIZE-1796 238b.

⟨consinit() *initializing things* 600c⟩+≡ (365a) ◁355a

```
todinit();
randominit();
```

Uses randominit() 603a and todinit() 238c.

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 601a⟩≡ (643)
```
static  ulong   randn;
```

⟨*function* seedrand 601b⟩≡ (643)
```
static void
seedrand(void)
{
    if(!waserror()){
        randomread((void*)&randn, sizeof(randn));
        poperror();
    }
}
```
Uses poperror 633d, randn-1798 601a, randomread() 603b, and waserror 633d.

⟨*function* nrand 601c⟩≡ (643)
```
int
nrand(int n)
{
    if(randn == 0)
        seedrand();
    randn = randn*1103515245 + 12345 + CPUS(0)->ticks;
    return (randn>>16) % n;
}
```
Uses CPUS 645, randn-1798 601a, and seedrand() 601b.

⟨*struct* Rb 601d⟩≡ (705a)
```
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 601e⟩≡ (705a)
```
struct Rb rb;
```
Uses Rb 601d.

⟨*struct* `rbnotfull` 602a⟩≡ (705a)

```
static int
rbnotfull(void*)
{
    int i;

    i = rb.rp - rb.wp;
    return i != 1 && i != (1 - sizeof(rb.buf));
}
```

Uses `rb` 601e.

⟨*struct* `rbnotempty` 602b⟩≡ (705a)

```
static int
rbnotempty(void*)
{
    return rb.wp != rb.rp;
}
```

Uses `rb` 601e.

⟨*function* `genrandom` 602c⟩≡ (705a)

```
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` 198b, `anyhigher()` 215d, `rb` 601e, `rbnotfull()` 602a, `sched`, `sleep`, and `up`.

⟨*clock callback randomclock* 602d⟩≡ (705a)

```
/*
 *  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 601e, rbnotfull() 602a, and wakeup.

⟨*function* randominit 603a⟩≡ (705a)

```
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() 231a, genrandom() 602c, kproc() 139c, randomclock() 602d, and rb 601e.

⟨*function* randomread 603b⟩≡ (705a)

```
/*
 *  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 633d, qlock() 63f, qunlock() 64c, rb 601e, rbnotempty() 602b, sleep, wakeup, and waserror 633d.

⟨consread() *cases* 604⟩+≡ (366a) ◁472b 618a▷

```
    case Qrandom:
        return randomread(buf, n);
```

Uses Qrandom-1789 600a and randomread() 603b.

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` 608a⟩≡ (648)
```
bool kdebug;
```

⟨`kbdputsc()` *special keyboard debug keys cases* 608b⟩≡ (350d)
```
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` 348a, `false`, `kbdputc()` 354b, `kdebug` 608a, `print`, and `true`.

⟨`kbdputsc()` *debugging* 608c⟩≡ (350d)
```
if(kdebug)
    print("sc %x (ms %d)\n", k, mouseshifted);
```
Uses `kdebug` 608a, `mouseshifted` 659, and `print`.

⟨`kbdputsc()` *debugging up shift* 608d⟩≡ (350d)
```
if(kdebug)
    print("shiftclr\n");
```
Uses `kdebug` 608a and `print`.

⟨`kbdputsc()` *debugging down shift* 608e⟩≡ (350d)
```
if(kdebug)
    print("shift\n");
```
Uses `kdebug` 608a and `print`.

⟨kbdputc() *debugging* 609a⟩≡ (354b)
```
extern bool kdebug;

if(kdebug)
    print("kbdputc(0x%x)\n", ch);
```
Uses kdebug 608a and print.

⟨echo() *locals* 609b⟩≡ (355b)
```
static int ctrlt;
void* tmp;
int x;
char *e, *p;
```

⟨echo() *special keys handler* 609c⟩≡ (355b)
```
    e = buf+n;
    for(p = buf; p < e; p++){
        switch(*p){

        ⟨echo() special key C-p 594e⟩

        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 619g⟩
        }
    }
```

Uses arch_dumpstack, arch_exit, ixsummary() 611d, killbig() 711c, mallocsummary() 610b, pagersummary() 611c, procdump() 612a, scheddump() 611f, xalloc() 77c, xfree() 78a, and xsummary() 610a.

⟨*function* xsummary 610a⟩≡ (697b)

```
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 694, arch_delay, print, and xlists-1849 76.

⟨*function* mallosummary 610b⟩≡ (698a)

```
void
mallocsummary(void)
{
    poolsummary(mainmem);
```

```
    poolsummary(imagmem);
}
```

Uses imagmem 84a, mainmem 84a, and poolsummary() 611a.

⟨*function* poolsummary 611a⟩≡ (698a)

```
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)* 611b⟩≡ (742b)

```
// called from devcons.c
void
arch_memorysummary(void) {
}
```

⟨*function* pagersummary 611c⟩≡ (703a)

```
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 189c, palloc 90d, and print.

⟨*function* ixsummary 611d⟩≡ (689b)

```
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, consumecnt-774, copyblockcnt-773, debugging-778, iallocsummary() 611e, padblockcnt-770, print, producecnt-775, pullupblockcnt-772, and qcopycnt-776.

⟨*function* iallocsummary 611e⟩≡ (687)

```
void
iallocsummary(void)
{
    print("ialloc %lud/%lud\n", ialloc.bytes, conf.ialloc);
}
```

Uses conf 46b, ialloc 407d, and print.

⟨*function* scheddump 611f⟩≡ (712c)

```
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 200a, arch_delay, nrdy 201a, print, and runq 200b.

⟨*function* procdump 612a⟩≡ (712c)

```
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 51b, conf 46b, dumpaproc, print, procalloc-2458 110c, and up.

⟨*function* dumpaproc 612b⟩≡ (712c)

```
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 100a, TSys 471a, TUser 471a, print, and statename 51c.

⟨*function* arch_dbgpc*(arm)* 612c⟩≡ (754e)

```
/*
```

```
 *  pc output by dumpaproc
 */
uintptr
arch_dbgpc(Proc* p)
{
    Ureg *ureg;

    ureg = p->dbgreg;
    if(ureg == 0)
        return 0;

    return ureg->pc;
}
```

⟨*function* lockloop 613a⟩≡ (641b)

```
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)* 613b⟩≡ (750b)

```
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 28a, PsrMsvc 28a, arch_delay, arch_dumpstack, dumplongs() 614a, iprint, trapname() 614c, and up.

⟨*function* dumplongs *(arm)* 614a⟩≡ (750b)

```
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() 614b.

⟨*function* isvalidaddr *(arm)* 614b⟩≡ (750b)

```
int
isvalidaddr(void *v)
{
    return (uintptr)v >= KZERO;
}
```

⟨*function* trapname *(arm)* 614c⟩≡ (750b)

```
static char *
trapname(int psr)
{
    char *s;

    s = trapnames[psr & PsrMask];
    if(s == nil)
        s = "unknown trap number in psr";
    return s;
}
```

Uses PsrMask 28a and trapnames-2414 143b.

⟨faultarm() *if debug* 614d⟩≡ (164b)

```
if (Debug)
    ckfaultstuck(va);
```

Uses Debug-2405 750a and ckfaultstuck() 615a.

⟨*function* ckfaultstuck*(arm)* 615a⟩≡ (750b)
```
/* 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)* 615b⟩≡ (750b)
```
void
trap_arch_dumpstack(void)
{
    arch_callwithureg(dumpstackwithureg);
}
```
Uses dumpstackwithureg() 616a.

⟨*function* arch_callwithureg*(arm)* 615c⟩≡ (750b)
```
void
arch_callwithureg(void (*fn)(Ureg*))
{
    Ureg ureg;

    getpcsp((ulong*)&ureg.pc, (ulong*)&ureg.sp);
    ureg.r14 = getcallerpc(&fn);
    fn(&ureg);
}
```
Uses getpcsp() 615d.

⟨*function* getpcsp*(arm)* 615d⟩≡ (750b)
```
/*
 * 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)* 616a⟩≡ (750b)

```
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 616b⟩≡ (643)

```
/*
 * 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` 617a⟩≡ (643)
```
struct KMesg kmesg;
```
Uses `KMesg` 616b.

⟨*constant* `KMESGSIZE` *(arm)* 617b⟩≡ (732b)
```
#define KMESGSIZE (16*1024) // for /dev/kmesg
```

⟨`echo()` *hooks* 617c⟩+≡ (355b) ◁356a 620c▷
```
kmesgputs(buf, n);
```
Uses `kmesgputs()` 617e.

⟨`putstrn0()` *kmesg handling* 617d⟩≡ (362b)
```
/*
 *  how many different output devices do we need?
 */
kmesgputs(str, n);
```
Uses `kmesgputs()` 617e.

⟨*function* `kmesgputs` 617e⟩≡ (643)
```
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()` 61d, `iunlock()` 62b, and `kmesg` 617a.

⟨*devcons.c enum Qxxx cases* 617f⟩+≡ (359c) ◁600a 618b▷
```
Qkmesg,
```

⟨`consdir` *fields* 617g⟩+≡ (360a) ◁600b 618c▷
```
"kmesg",    {Qkmesg},    0,      0440,
```
Uses `Qkmesg-1794` 617f.

⟨consread() *cases* 618a⟩+≡ (366a) ◁604 619b▷
```
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 617f and kmesg 617a.

A.5 /dev/kprint

⟨*devcons.c enum Qxxx cases* 618b⟩+≡ (359c) ◁617f
```
Qkprint,
```

⟨consdir *fields* 618c⟩+≡ (360a) ◁617g
```
"kprint",   {Qkprint, 0, QTEXCL},   0,  DMEXCL|0440,
```
Uses DMEXCL 618d, QTEXCL 416b, and Qkprint-1795 618b.

⟨Dirmode *cases* 618d⟩+≡ (313a) ◁403a
```
DMEXCL = 0x20000000,  /* mode bit for exclusive use files */
```

⟨*global* kprintoq 618e⟩≡ (643)
```
Queue*  kprintoq;       /* console output, for /dev/kprint */
```

⟨*global* kprintinuse 618f⟩≡ (643)
```
ulong   kprintinuse;        /* test and set whether /dev/kprint is open */
```

⟨consopen() *cases* 618g⟩+≡ (365b) ◁368d
```
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 272b, Qcoalesce 371a, Qkprint-1795 618b, kprintinuse 618f, kprintoq 618e, qiomaxatomic 689a, qnoblock() 391c, qopen() 380a, qreopen() 390b, and true.

⟨consclose() *cases* 619a⟩+≡ (365c) ◁368e
```
  /* close of kprint allows other opens */
  case Qkprint:
      if(c->flag & COPEN){
          kprintinuse = 0;
          qhangup(kprintoq, nil);
      }
      break;
```
Uses COPEN 272b, Qkprint-1795 618b, kprintinuse 618f, kprintoq 618e, and qhangup() 389c.

⟨consread() *cases* 619b⟩+≡ (366a) ◁618a
```
  case Qkprint:
      return qread(kprintoq, buf, n);
```
Uses Qkprint-1795 618b, kprintoq 618e, and qread() 384.

⟨putstrn0() *if kprint* 619c⟩≡ (362b)
```
  /*
   *  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 618e, qisclosed() 390a, qiwrite() 387, and qwrite() 372b.

⟨panic() *reset kprintoq* 619d⟩≡ (630b)
```
  kprintoq = nil; /* don't try to write to /dev/kprint */
```
Uses kprintoq 618e.

A.6 Remote debugger

⟨*hook consdebug* 619e⟩≡ (643)
```
  void    (*consdebug)(void) = nil; // for rdb
```
Uses consdebug 619e.

⟨panic() *run consdebug hook* 619f⟩≡ (630b)
```
  if(consdebug)
      (*consdebug)();
```
Uses consdebug 619e.

⟨echo() *C-t C-t special keys handler other cases* 619g⟩≡ (609c)
```
  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 619e, print, and rdb().

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 620a⟩≡ (643)

```
// option<Queue>
Queue*  serialoq;       /* serial console output */
```

⟨*function* consactive 620b⟩≡ (643)

```
int
consactive(void)
{
    if(serialoq)
        return qlen(serialoq) > 0;
    return 0;
}
```

Uses qlen() 390c and serialoq 620a.

⟨echo() *hooks* 620c⟩+≡ (355b) ◁617c

```
if(serialoq)
    echoserialoq(buf, n);
```

Uses echoserialoq() 620d and serialoq 620a.

⟨*function* echoserialoq 620d⟩≡ (643)

```
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() 387 and serialoq 620a.

⟨putstrn0() *serialoq handling* 621a⟩≡ (362b)

```
/*
 *   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 621b⟩
// 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 354a, qiwrite() 387, qwrite() 372b, and serialoq 620a.

A.7.2 uartconsinit()

A.7.3 uartputs()

⟨putstrn0() *if serialoq nil* 621b⟩≡ (621a)

```
if(serialoq == nil){
    uartputs(str, n);
    return;
}
```

Uses serialoq 620a and uartputs().

A.7.4 consuart

⟨*global* consuart 621c⟩≡ (? 0—1)

```
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* 623a⟩≡ (48a) 623f▷
```
  Perf  perf;     /* performance counters */
```

⟨*struct* Perf 623b⟩≡ (645)
```
  /*
   *  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* 623c⟩+≡ (204) ◁467f
```
  ulong start, now;
```

⟨runproc() *start* 623d⟩≡ (204)
```
  start = arch_perfticks();
```

⟨runproc() *after* arch_idlehands, *before restarting loop* 623e⟩≡ (204)
```
  /* remember how much time we're here */
  now = arch_perfticks();
  cpu->perf.inidle += now-start;
  start = now;
```

⟨Cpu *stat fields* 623f⟩+≡ (48a) ◁623a 625c▷
```
  int load;
```

⟨hzclock() *accounttime* 624a⟩≡ (215f)
```
accounttime();
```
Uses accounttime() 624b.

⟨*function* accounttime 624b⟩≡ (712c)
```
/*
 *  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 472a⟩
    }

    /* 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 201a.

⟨*function* intrtime*(arm)* 624c⟩≡ (750b)
```
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)* 625a⟩≡ (753)

```
ulong
arch_perfticks(void)
{
    Armtimer *tm;

    tm = (Armtimer*)ARMTIMER;
    return tm->count;
}
```

Uses ARMTIMER-2648 31f.

⟨*struct* Armtimer*(arm)* 625b⟩≡ (753)

```
struct Armtimer {
    u32int  load;
    u32int  val;
    u32int  ctl;
    u32int  irqack;
    u32int  irq;
    u32int  maskedirq;
    u32int  reload;
    u32int  predivider;
    u32int  count;
};
```

⟨Cpu *stat fields* 625c⟩+≡ (48a) ◁623f

```
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* 625d⟩+≡ (589a) ◁593c

```
Qsysstat,
```

⟨sysdir *fields* 625e⟩+≡ (589b) ◁593d

```
"sysstat",  {Qsysstat}, 0,      0666,
```

Uses Qsysstat-844 625d.

⟨sysread() *cases* 626a⟩+≡ (589e) ◁593b

```
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 645, NUMSIZE 704a, Qsysstat-844 625d, active 49e, conf 46b, free() 85b, nexterror, poperror 633d, readnum() 637c, readstr() 637d, smalloc() 85c, and waserror 633d.

⟨syswrite() *cases* 626b⟩+≡ (590b) ◁594a

```
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 645, Qsysstat-844 625d, and active 49e.

B.2 /dev/kprof

⟨hzclock() *if kproftimer* 627a⟩≡ (215f)
```
if(kproftimer != nil)
    kproftimer(ur->pc);
```
Uses kproftimer 627b.

⟨*hook kproftimer* 627b⟩≡ (708a)
```
void (*kproftimer)(ulong);
```

B.3 Statistics

B.3.1 Lock.lockcycles

⟨*lock ifdef LOCKCYCLES* 627c⟩≡ (61d 60f 58)
```
#ifdef LOCKCYCLES
        l->lockcycles = -arch_lcycles();
#endif
```

⟨Lock *debugging fields* 627d⟩+≡ (58c) ◁70a
```
//#ifdef LOCKCYCLES
long  lockcycles;
//#endif
```

⟨*unlock ifdef LOCKCYCLES* 627e⟩≡ (60d)
```
#ifdef LOCKCYCLES
    l->lockcycles += lcycles();
    cumlockcycles += l->lockcycles;
    if(l->lockcycles > maxlockcycles){
        maxlockcycles = l->lockcycles;
        maxlockpc = l->pc;
    }
#endif
```

⟨*iunlock ifdef LOCKCYCLES* 627f⟩≡ (62b)
```
#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* 627g⟩≡ (641b)
```
#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 628a⟩≡ (641b)
```
struct TaslockStats
{
    ulong   locks;
    ulong   glare;
    ulong   inglare;
};
```

⟨lock() *start of lock, stat* 628b⟩≡ (58d)
```
lockstats.locks++;
```
Uses lockstats 628e.

⟨lock() *when lock already held, stat* 628c⟩≡ (58d)
```
lockstats.glare++;
```
Uses lockstats 628e.

⟨lock() *in loop when lock already held, stat* 628d⟩≡ (58d)
```
lockstats.inglare++;
```
Uses lockstats 628e.

⟨*global* lockstats 628e⟩≡ (641b)
```
struct TaslockStats lockstats;
```
Uses TaslockStats 628a.

B.3.3 QlockStats

⟨*struct* QlockStats 628f⟩≡ (642a)
```
struct QlockStats {
    ulong rlock;
    ulong rlockq;
    ulong wlock;
    ulong wlockq;
    ulong qlock;
    ulong qlockq;
};
```

⟨*global* rwstats 628g⟩≡ (642a)
```
struct QlockStats rwstats;
```
Uses QlockStats 628f.

⟨qlock() *start of qlock, stat* 628h⟩≡ (63f)
```
rwstats.qlock++;
```
Uses rwstats 628g.

⟨qlock() *when qlock already held, stat* 628i⟩≡ (63f)
```
rwstats.qlockq++;
```
Uses rwstats 628g.

B.3.4 Kimage statistics

⟨*struct* Irstats 628j⟩≡ (702)
```
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* 629a⟩≡ (702)

```
static struct Irstats  irstats;
```

Uses `Irstats` 628j.

B.3.5 Scheduling

⟨*proc.c statistics* 629b⟩≡ (712c)

```
long preempts;
ulong delayedscheds;    /* statistics */
long skipscheds;
```

B.3.6 Clock

⟨*clock.c statistics* 629c⟩≡ (709c)

```
ulong intrcount[MAXCPUS];
ulong fcallcount[MAXCPUS];
```

⟨`timerintr()` *start, stat* 629d⟩≡ (229d)

```
intrcount[cpu->cpuno]++;
```

Uses `intrcount` 629c.

⟨`timerintr()` *in timers loop, stat* 629e⟩≡ (229d)

```
fcallcount[cpu->cpuno]++;
```

Uses `fcallcount` 629c.

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` 630a⟩≡ (643)
```
bool panicking;
```

⟨*function* `panic` 630b⟩≡ (643)
```
void
devcons_panic(char *fmt, ...)
{
    int n, s;
    va_list arg;
    char buf[PRINTSIZE];

    ⟨panic() reset kprintoq 619d⟩

    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 619f⟩
    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` 363a, `iprint`, `panicking` 630a, `prflush()` 631a, `putstrn()` 363c, and `true`.

⟨*function* `prflush` 631a⟩≡ (643)
```
void
prflush(void)
{
    ulong now;

    now = cpu->ticks;
    while(consactive())
        if(cpu->ticks - now >= Arch_HZ)
            break;
}
```
Uses `consactive()` 620b.

⟨*function* `sysfatal` 631b⟩≡ (643)
```
/* 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` 631c⟩≡ (643)
```
void
devcons__assert(char *fmt)
{
    panic("assert failed at %#p: %s", getcallerpc(&fmt), fmt);
}
```

C.2 `error()`

⟨*global Exxx errors* 631d⟩≡ (646a)
```
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 631d, Ebadchar 631d, Ebadexec 631d, Ebadfd 631d, Ebadip 631d, Ebadsharp 631d, Ebadspec 631d, Ebadusefd 631d, Echange 631d, Ecmdargs 631d, Econinuse 631d, Econrefused 631d, Edetach 631d, Edirseek 631d, Efilename 631d, Egreg 631d, Ehungup 631d, Eioload 631d, Eismtpt 631d, Eisstream 631d, Emount 631d, Emountrpc 631d, Enegoff 631d, Enoattach 631d, Enochild 631d, Enocreate 631d, Enodev 631d, Enofd 631d, Enoreg 631d, Enotconf 631d, Enoversion 631d, Enovmem 631d, Eprocdied 631d, Esbadstat 631d, Eshort 631d, Eshortstat 631d, Eshutdown 631d, Esoverlap 631d, Estalled 631d, Etimedout 631d, Etoobig 631d, Eunion 631d, and Eunmount 631d.

⟨*global* Edoesnotexist 632a⟩≡ (684)

```
static char Edoesnotexist[] = "does not exist";
```

Uses Edoesnotexist-768 632a.

⟨Proc *error managment fields* 632b⟩≡ (49f) 634b▷

```
// 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 632c⟩+≡ (111d) ◁584b

```
p->nerrlab = 0;
p->errstr = p->errbuf0;
```

```
  p->syserrstr = p->errbuf1;
  p->errbuf0[0] = '\0';
  p->errbuf1[0] = '\0';
```

⟨*constant* NERR 633a⟩≡ (705c)
```
  NERR = 64,
```

⟨*function* error 633b⟩≡ (712c)
```
  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 692a, NERR 633a, assert 693a, kstrcpy() 636a, nexterror, and up.

⟨*function* exhausted 633c⟩≡ (712c)
```
  void
  exhausted(char *resource)
  {
      char buf[ERRMAX];

      snprint(buf, sizeof buf, "no free %s", resource);
      iprint("%s\n", buf);
      error(buf);
  }
```
Uses ERRMAX 692a 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* 633d⟩≡ (705c)
```
  // 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` 634a⟩≡ (712c)
```
// raise an exception
void
proc_nexterror(void)
{
    arch_gotolabel(&up->errlab[--up->nerrlab]);
}
```
Uses `up`.

C.4 syserrstr()

⟨`Proc` *error managment fields* 634b⟩+≡ (49f) ◁632b
```
char  *syserrstr; /* last error from a system call, errbuf0 or 1 */
```

⟨*syscall errstr* 634c⟩≡ (716a)
```
// int errstr(char *err, uint nerr);
long
syserrstr(ulong* arg)
{
    return generrstr((char*)arg[0], arg[1]);
}
```
Uses `generrstr()` 634d.

⟨*function* `generrstr` 634d⟩≡ (716a)
```
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` 692a, `Ebadarg` 631d, `true`, `up`, and `validaddr()` 101c.

⟨*function* `werrstr` 634e⟩≡ (716a)
```
//@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` 692a 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* 635a⟩≡ (693c)
```
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` 635b⟩≡ (684)
```
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* 635c⟩≡ (693c)
```
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 636a⟩≡ (698b)

```
/*
 * 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 636b⟩≡ (698b)

```
/*
 * 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()` 85b, `malloc()` 85a, `setmalloctag()` 87b, `smalloc()` 85c, and `up`.

⟨*function* `emptystr` 637a⟩≡ (684)
```
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* 637b⟩≡ (693c)
```
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` 637c⟩≡ (647a)
```
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` 637d⟩≡ (647a)
```
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* 637e⟩≡ (693c)

```
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 638a⟩≡ (693c)

```
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* 638b⟩≡ (693c)

```
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* 638c⟩≡ (693c)

```
#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 639a⟩≡ (693c)
```
enum
{
  UTFmax    = 4,    /* maximum bytes per rune */
  Runeself  = 0x80,   /* rune and UTF sequences are the same (<) */
};
```

⟨*lib.h rune functions decl* 639b⟩≡ (693c)
```
extern  int runetochar(char*, Rune*);
extern  int chartorune(Rune*, char*);
extern  char* utfrune(char*, long);
extern  int utfnlen(char*, long);
```

⟨Kbscan *other fields* 639c⟩+≡ (350a) ◁369a
```
bool collecting;
int nk;
Rune    kc[5];
```

⟨kbdputsc() *if collecting* 639d⟩≡ (350d)
```
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() 354b, latin1() 649, nelem 692c, and print.

⟨kbdputsc() *start collecting* 639e⟩≡ (350d)
```
/*
 * 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 639f⟩≡

Appendix E

Extra Code

E.1 arch/

E.2 buses/

E.3 concurrency/

E.3.1 concurrency/portdat_concurrency.h

⟨concurrency/portdat_concurrency.h 640⟩≡

```
//*****************************************************************************
// 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 58c⟩

⟨struct ILock 61a⟩

⟨struct QLock 63a⟩

⟨struct RWlock 65b⟩

//*****************************************************************************
// Atomicity
//*****************************************************************************

⟨struct Ref 69a⟩

⟨struct Counter 69d⟩

//*****************************************************************************
// Synchronization
//*****************************************************************************

// defined in this directory but no functions are operating on it in this dir
⟨struct Rendez 212a⟩

⟨struct Sema 442a⟩
```

```
  // see also Waitq in portdat_processes.h?
```

E.3.2 concurrency/portfns_concurrency.h

⟨portfns_concurrency.h 641a⟩≡

```
  // 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
```

E.3.3 concurrency/taslock.c

⟨concurrency/taslock.c 641b⟩≡

```
  // TAS: Test And Set
  ⟨kernel basic includes 35a⟩

  ⟨struct TaslockStats 628a⟩
```

⟨*global* `lockstats` 628e⟩
⟨*globals lockcycles* 627g⟩

⟨*function* `inccnt` 70i⟩

⟨*function* `deccnt` 71a⟩

⟨*function* `lockloop` 613a⟩

⟨*function* `lock` 58d⟩

⟨*function* `ilock` 61d⟩

⟨*function* `canlock` 60f⟩

⟨*function* `unlock` 60d⟩

⟨*function* `iunlock` 62b⟩

E.3.4 `concurrency/qlock.c`

⟨`concurrency/qlock.c` 642a⟩≡
⟨*kernel basic includes* 35a⟩

⟨*struct* `QlockStats` 628f⟩
⟨*global* `rwstats` 628g⟩

⟨*function* `qlock` 63f⟩

⟨*function* `canqlock` 65a⟩

⟨*function* `qunlock` 64c⟩

⟨*function* `rlock` 65e⟩

⟨*function* `runlock` 66b⟩

⟨*function* `wlock` 67b⟩

⟨*function* `wunlock` 67d⟩

⟨*function* `canrlock` 68⟩

E.3.5 `concurrency/ref.c`

⟨`concurrency/ref.c` 642b⟩≡
⟨*kernel basic includes* 35a⟩

⟨*function* `incref` 69b⟩

⟨*function* `decref` 69c⟩

E.4 console/

E.4.1 console/portdat_console.h

E.4.2 console/portfns_console.h

E.4.3 console/devcons.c

⟨console/devcons.c 643⟩≡
⟨*kernel basic includes* 35a⟩

```
#include    <pool.h>

// used also by edf.c
```
⟨*global* panicking 630a⟩

⟨*hook screenputs* 356b⟩
⟨*hook consdebug* 619e⟩

⟨*global* kbdq 360b⟩
⟨*global* lineq 360d⟩
⟨*global* serialoq 620a⟩
⟨*global* kprintoq 618e⟩

⟨*global* kprintinuse 618f⟩
⟨*global* iprintscreenputs 363d⟩

⟨*struct* ConsKbd 353d⟩

⟨*global* kbd 354a⟩

⟨*global* fasthz 242c⟩

⟨*devcons.c forward decl* ??⟩

⟨*function* kbdqinit 360c⟩

⟨*function* lineqinit 360e⟩

⟨*function* consactive 620b⟩

⟨*function* prflush 631a⟩

⟨*struct* KMesg 616b⟩

⟨*global* kmesg 617a⟩

⟨*function* kmesgputs 617e⟩

⟨*function* putstrn0 362b⟩

⟨*function* putstrn 363c⟩

⟨*function* print 363b⟩

⟨*global* iprintlock 364a⟩

⟨*function* iprintcanlock 364b⟩

⟨*function* `iprint` 363e⟩

⟨*function* `panic` 630b⟩

⟨*function* `sysfatal` 631b⟩

⟨*function* `_assert` 631c⟩

⟨*function* `pprint` 364c⟩

⟨*function* `echoscreen` 356c⟩

⟨*function* `echoserialoq` 620d⟩

⟨*function* `echo` 355b⟩

⟨*function* `kbdcr2nl` **??**⟩

⟨*function* `kbdputc` 354b⟩

⟨*clock callback kbdputcclock* 354c⟩

⟨*devcons.c enum Qxxx* 359c⟩

```
enum
{
    ⟨constant VLNUMSIZE 238b⟩
};
```

⟨*global* `consdir` 360a⟩

⟨*method consinit* 365a⟩

```
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* 365b⟩

⟨*method consclose* 365c⟩

⟨*method consread* 366a⟩

⟨*method conswrite* 366b⟩

⟨*global* `consdevtab` 367b⟩

⟨*global* randn 601a⟩

⟨*function* seedrand 601b⟩

⟨*function* nrand 601c⟩

⟨*global* uvorder 245a⟩

⟨*function* le2vlong 245b⟩

⟨*function* vlong2le 245c⟩

⟨*global* order 245d⟩

⟨*function* le2long 245e⟩

⟨*devcons.c Exxx errors* 238a⟩

⟨*function* readtime 242e⟩

⟨*function* writetime 243a⟩

⟨*function* readbintime 244a⟩

⟨*function* writebintime 244b⟩

Uses consdir-1797 360a, devattach() 275a, devgen() 399, devstat() 402, devwalk() 400b, and nelem 692c.

E.5 core/

E.5.1 core/portdat_forward.h

E.5.2 core/portdat_core.h

⟨core/portdat_core.h 645⟩≡

```
// 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 46c⟩

⟨*struct* Conf 46a⟩

```
extern Conf conf;
```

⟨*constant* MAXCONF 47b⟩

```
// 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 205d⟩

⟨*struct* Perf 623b⟩

⟨*struct* Cpu 48a⟩

```
// array<ref<Cpu>>, MAXCPUS is defined in <arch>/mem.h
extern Cpu* cpus[MAXCPUS];
```
⟨*macro* CPUS 49c⟩

```
// '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;
```

E.5.3 core/portfns_core.h

E.5.4 core/portdat.h

E.5.5 core/error.h

E.5.6 core/error.c

⟨core/error.c 646a⟩≡
  ⟨*global Exxx errors* 631d⟩

E.5.7 core/portdat.c

⟨core/portdat.c 646b⟩≡
  ⟨*kernel basic includes* 35a⟩

```
  // was in init/main.c
```
  ⟨*global* conf 46b⟩
```
  // bool cpuserver; // defined in $CONF.c
```

  ⟨*globals confname and confvar* 47a⟩
  ⟨*global* nconf 47c⟩

```
//TODO: to factorize code between pc and pi
//< <function [[getconf]]> >

// was in dat.h
⟨global cpus 49a⟩
// cpu is in init/main.c because it can differ in different archi

// was in security/auth.c
⟨global eve 557a⟩
// should be in portfns.c, but then backward deps
/*
 *  return true if current user is eve
 */
⟨function iseve 557b⟩
```

E.5.8 core/portfns.c

```
⟨core/portfns.c 647a⟩≡
  ⟨kernel basic includes 35a⟩

  ⟨portfns.c backward deps breaker ??⟩

  ⟨function returnfalse 86a⟩

  // was in devcons.c, could be in lib/misc.c
  ⟨function readnum 637c⟩

  ⟨function readstr 637d⟩
```

E.6 devices/

E.6.1 devices/portdat_devices.h

```
⟨struct Devport 647b⟩≡                                    (647d)
  struct Devport {
      ulong port;
      int size;
  };

⟨struct DevConf 647c⟩≡                                    (647d)
  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 647d⟩≡
  /*
   *  hardware info about a device
   */
  ⟨struct Devport 647b⟩

  ⟨struct DevConf 647c⟩
```

```
// keyboard

⟨enum kbscan 350b⟩

⟨struct Kbscan 350a⟩

// 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
```

E.6.2 devices/portfns_devices.h

E.6.3 devices/keyboard/portkbd.c

⟨devices/keyboard/portkbd.c 648⟩≡

```
/*
 * keyboard input // the portable part
 */
⟨kernel basic includes 35a⟩

// 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 608a⟩

⟨enum specialkey 348a⟩

⟨global kbtab 348b⟩

⟨global kbtabshift 349a⟩
```

⟨*global* kbtabesc1 352b⟩

⟨*global* kbtabaltgr 349c⟩

```
// 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* 349b⟩

⟨*global* kbscans 350c⟩

```
// called when write in /dev/kbin (see devkbin.c)
```
⟨*function* kbdputsc 350d⟩

```
// called when interact with /dev/kbmap (see devkbmap.c)
```
⟨*function* kbdputmap 357⟩

⟨*function* kbdgetmap 356d⟩

E.6.4 devices/keyboard/latin1.c

⟨devices/keyboard/latin1.c 649⟩≡
⟨*kernel basic includes* 35a⟩

```
/*
 * 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 639a, cvlist 649, and print.

E.6.5 devices/keyboard/latin1.h

E.6.6 TO LP SPLIT devices/keyboard/devkbin.c

⟨devices/keyboard/devkbin.c 651⟩≡

```
/*
 *  keyboard scan code input from outside the kernel.
 *  to avoid duplication of keyboard map processing for usb.
 */
⟨kernel basic includes 35a⟩

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 631d, QTDIR 52c, Qdir-992 651, Qkbd-993 651, devattach() 275a, devbread() 405a, devbwrite() 405b, devcreate() 686, devdirread() 403c, devgen() 399, devinit() 686, devremove() 686, devreset() 686, devshutdown() 686, devstat() 402, devwalk() 400b, devwstat() 686, free() 85b, iseve() 557b, kbdputsc() 350d, kbinattach() 651, kbinbusy 651, kbinclose() 651, kbinlck 651, kbinopen() 651, kbinread() 651, kbinstat() 651, kbintab 651, kbinwalk() 651, kbinwrite() 651, lock() 58d, nelem 692c, and unlock() 60d.

E.6.7 TO LPSPLIT devices/keyboard/devkbmap.c

⟨devices/keyboard/devkbmap.c 653⟩≡

```
/*
 *  keyboard map
 */
⟨kernel basic includes 35a⟩

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 631d, Ebadusefd 631d, KBLINELEN-997 653, NKBFILE-996 653, NUMSIZE 704a, QTDIR 52c, Qdata-995 653, Qdir-994 653, devattach() 275a, devbread() 405a, devbwrite() 405b, devcreate() 686, devdirread() 403c, devgen() 399, devinit() 686, devremove() 686, devreset() 686, devshutdown() 686, devstat() 402, devwalk() 400b, devwstat() 686, free() 85b, iseve() 557b, kbdgetmap() 356d, kbdputmap() 357, kbmapattach() 653, kbmapclose() 653, kbmapopen() 653, kbmapread() 653, kbmapstat() 653, kbmaptab 653, kbmapwalk() 653, kbmapwrite() 653, readnum() 637c, readstr() 637d, and smalloc() 85c.

E.6.8 TO LPSPLIT devices/mouse/portmouse.c

⟨devices/mouse/portmouse.c 656⟩≡

⟨*kernel basic includes* 35a⟩

```
// 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-930 656, CMhwaccel-931 656, CMintellimouse-932 656, CMlinear-933 656, CMps2-934 656, CMps2intellimouse-935 656, CMres-936 656, CMreset-937 656, CMserial-938 656, MousePS2 647d, accelerated-928 656, cmderror() 561a, intellimouse 656, lookupcmd() 561b, mouseaccelerate() 659, mousectlmsg-939 656, mousectlqlock-929 656, mousehwaccel 656, mousetype 656, nelem 692c, nexterror, packetsize 656, poperror 633d, ps2mouse() 656, qlock() 63f, qunlock() 64c, resetmouse() 656, resolution-927 656, setaccelerated() 656, setintellimouse() 656, setlinear() 656, setres() 656, and waserror 633d.

E.6.9 TO LPSPLIT devices/mouse/devmouse.c

⟨devices/mouse/devmouse.c 659⟩≡
⟨*kernel basic includes* 35a⟩

```
#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-950 659, CMscrollswap-951 659, CMswap-952 659, CMwildcard-953 659, COPEN 272b, CPUS 645, Cursortocursor() 659, DMDIR 313a, Ebadarg 631d, Egreg 631d, Eshort 631d, Mouseinfo 659, Mousestate 659, OREAD 279c, QTDIR 52c, Qcursor-945 659, Qdir-944 659, Qmouse-946 659, Qmousectl-948 659, Qmousein-947 659, TK2MS 223a, addclock0link() 231a, buttonmap-955 659, canlock() 60f, curs 659, cursor 659, devattach() 275a, devbread() 405a, devbwrite() 405b, devdirread() 403c, devgen() 399, devremove() 686, devshutdown() 686, devstat() 402, devwalk() 400b, devwstat() 686, drawactive(), false, free() 85b, gscreen, incref() 69b, iseve() 557b, kbdbuttons 659, kbdmouse 659, kmousectl() 656, lock() 58d, lookupcmd() 561b, mouse 659, mouseattach() 659, mousechanged() 659, mouseclock() 659, mouseclose() 659, mousecreate() 659, mousectlmsg-954 659, mousedevgen() 659, mousedir-949 659, mousefromkbd() 659, mouseinit() 659, mouseopen() 659, mouseread() 659, mousereset() 659, mousestat() 659, mouseswap-956 659, mousetime-958 659, mousetrack() 659, mousewalk() 659, mousewrite() 659, nelem 692c, nexterror, openmode() 280a, parsecmd() 560c, poperror 633d, qlock() 63f, qunlock() 64c, returnfalse() 86a, scrollswap-957 659, seconds() 241a, setbuttonmap() 659, sleep, true, tsleep, unlock() 60d, up, wakeup, and waserror 633d.

E.6.10 devices/sys/devroot.c

⟨*method rootclose* 670a⟩≡ (670b)

```
/*
 * sysremove() knows this is a nop
 */
static void
rootclose(Chan*)
{
}
```

⟨devices/sys/devroot.c 670b⟩≡

```
⟨kernel basic includes 35a⟩

//*****************************************************************************
// Constants and types
//*****************************************************************************

⟨devroot enum Qxxx 394a⟩

⟨devroot.c forward decl ??⟩
⟨struct Dirlist 394b⟩

//*****************************************************************************
// Root
//*****************************************************************************

// note: directories have by convention 0 length
```

⟨*globals rootdir, rootdata, rootlist* 394c⟩

```
//*****************************************************************************
// Boot
//*****************************************************************************
```

⟨*globals bootdir, bootdata, bootlist* 397⟩

```
//*****************************************************************************
// Functions
//*****************************************************************************
```

⟨*function* `addlist` 395a⟩

⟨*function* `addbootfile` 398a⟩

⟨*function* `addrootdir` 395b⟩

⟨*method rootreset* 395c⟩

```
static Chan*
rootattach(char *spec)
{
    return devattach('/', spec);
}
```

⟨*function* `rootgen` 396a⟩

```
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* 670a⟩

⟨*method rootread* 396b⟩

```
static long
rootwrite(Chan*, void*, long, vlong)
{
    error(Egreg);
    return 0;
}
```

⟨*global* `rootdevtab` 393b⟩

Uses Egreg 631d, devattach() 275a, devgen() 399, devstat() 402, devwalk() 400b, and rootgen() 396a.

E.6.11 devices/sys/devsys.c

⟨*method syswalk* 672a⟩≡ (672e)
```
static Walkqid*
syswalk(Chan *c, Chan *nc, char **name, int nname)
{
    return devwalk(c, nc, name, nname, sysdir, nelem(sysdir), devgen);
}
```
Uses devgen() 399, devwalk() 400b, nelem 692c, and sysdir-845 589b.

⟨*method sysstat* 672b⟩≡ (672e)
```
static int
sysstat(Chan *c, uchar *dp, int n)
{
    return devstat(c, dp, n, sysdir, nelem(sysdir), devgen);
}
```
Uses devgen() 399, devstat() 402, nelem 692c, and sysdir-845 589b.

⟨*method sysopen* 672c⟩≡ (672e)
```
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() 399, nelem 692c, and sysdir-845 589b.

⟨*method sysclose* 672d⟩≡ (672e)
```
static void
sysclose(Chan *c)
{
    switch((ulong)c->qid.path){
    }
}
```

⟨devices/sys/devsys.c 672e⟩≡
```
⟨kernel basic includes 35a⟩

#include    <authsrv.h>

⟨devsys.c enum Qxxx 589a⟩

⟨global sysdir 589b⟩

⟨devsys.c decls and globals 591d⟩
⟨global sysname 592c⟩

⟨method sysinit 589c⟩

⟨method sysattach 589d⟩

⟨method syswalk 672a⟩
```

⟨*method sysstat* 672b⟩

⟨*method sysopen* 672c⟩

⟨*method sysclose* 672d⟩

⟨*method sysread* 589e⟩

⟨*method syswrite* 590b⟩

⟨*global* `sysdevtab` 590d⟩

E.6.12 TO LPSPLIT `devices/sys/devenv.c`

⟨*function* `envgrp` 673a⟩≡ (674)

```
static Egrp*
envgrp(Chan *c)
{
    if(c->aux == nil)
        return up->egrp;
    return c->aux;
}
```

Uses `up`.

⟨*function* `envwriteable` 673b⟩≡ (674)

```
static bool
envwriteable(Chan *c)
{
    return iseve() || c->aux == nil;
}
```

Uses `iseve()` 557b.

⟨*function* `getconfenv` 673c⟩≡ (674)

```
/*
 * 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-787 674, malloc() 85a, nexterror, poperror 633d, rlock() 65e, runlock() 66b, and waserror 633d.

⟨devices/sys/devenv.c 674⟩≡

```
⟨kernel basic includes 35a⟩

enum
{
    Maxenvsize = 16300,
};

⟨devenv.c forward decl ??⟩

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` 565⟩

⟨*function* `envcpy` 564a⟩

⟨*function* `envgrp` 673a⟩

⟨*function* `envwriteable` 673b⟩

⟨*function* `ksetenv` 564b⟩

⟨*function* `getconfenv` 673c⟩

Uses `COPEN` 272b, `CRCLOSE` 298d, `DEVDOTDOT`, `DMDIR` 313a, `Etoobig` 631d, `Maxenvsize-786` 674, `OREAD` 279c, `OTRUNC` 310a, `QTDIR` 52c, `devdir()` 398c, `devdirread()` 403c, `devstat()` 402, `devwalk()` 400b, `envgen()` 674, `envgrp()` 673a, `envlookup()` 674, `envremove()` 674, `envwriteable()` 673b, `eve` 557a, `free()` 85b, `kstrcpy()` 636a, `nexterror`, `openmode()` 280a, `poperror` 633d, `rlock()` 65e, `runlock()` 66b, `smalloc()` 85c, `up`, `waserror` 633d, `wlock()` 67b, and `wunlock()` 67d.

E.6.13 devices/sys/devpipe.c

⟨`devices/sys/devpipe.c` 679⟩≡

⟨*kernel basic includes* 35a⟩

```
// need that??
#include    "netif.h"
```

⟨*devpipe.c forward decl* **??**⟩

⟨*struct* `Pipe` 427b⟩

⟨*struct* `Pipealloc` 427c⟩

⟨*global* `pipealloc` 427d⟩

⟨*devpipe.c enum Qxxx* 428a⟩

⟨*global* `pipedir` 428b⟩

```
#define NPIPEDIR 3
```

⟨*method pipeinit* 428d⟩

⟨*method pipeattach* 428f⟩

⟨*function* `pipegen` 429⟩

⟨*method pipewalk* 430a⟩

⟨*method pipestat* 430b⟩

⟨*method pipewstat* 431a⟩

⟨*method pipeopen* 431b⟩

⟨*method pipeclose* 432⟩

⟨*method piperead* 433a⟩

⟨*method pipebread* 433b⟩

⟨*method pipewrite* 434a⟩

⟨*method pipebwrite* 434b⟩

⟨*global* `pipedevtab` 435a⟩

E.6.14 devices/sys/devproc.c

⟨`devices/sys/devproc.c` 680⟩≡

```
⟨kernel basic includes 35a⟩

#include    <trace.h>
#include    <tos.h>
#include    <ureg.h>

⟨struct Mntwalk 325a⟩

⟨devproc.c enum Qxxx 568a⟩

⟨devproc enum CMxxx 568b⟩

enum{
⟨constant Nevents 468g⟩
⟨constant Emask 468h⟩
};

#define STATSIZE    (2*KNAMELEN+12+9*12)
⟨global procdir 569a⟩

⟨global proccmd 569b⟩

/* Segment type from portdat.h */
⟨global sname 94b⟩

⟨devproc QXXX macros 570a⟩

⟨devproc.c forward decl ??⟩

⟨global trace txxx 468d⟩
```

⟨*clock callback profclock* 473h⟩

⟨*function* `procgen` 580d⟩

⟨*function* `_proctrace` 468f⟩

⟨*method procinit* 570d⟩

```
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` 582a⟩

⟨*method procopen* 577⟩

⟨*method procwstat* 579⟩

⟨*function* `procoffset` 580a⟩

⟨*function* `procqidwidth` 580b⟩

⟨*function* `procfdprint` 580c⟩

⟨*function* `procfds` 582b⟩

⟨*method procclose* 583⟩

⟨*function* `int2flag` 326a⟩

⟨*function* `procargs` 584c⟩

⟨*function* `eventsavailable` 468e⟩

⟨*method procread* 571d⟩

⟨*function* `mntscan` 326b⟩

⟨*method procwrite* 574c⟩

⟨*global* `procdevtab` 571a⟩

⟨*function* `proctext` 584d⟩

⟨*function* procstopwait 454e⟩

⟨*function* procctlcloseone 585a⟩

⟨*function* procctlclosefiles 585b⟩

⟨*function* parsetime 586a⟩

⟨*function* procctlreq 586b⟩

⟨*function* procstopped 455a⟩

⟨*function* procctlmemio 457⟩

⟨*function* text2data 458⟩

Uses devattach() 275a, devstat() 402, devwalk() 400b, and procgen() 580d.

E.7 files/

E.7.1 files/portdat_files.h

⟨*struct* Extent 682a⟩≡ (682c)

```
struct Extent
{
  int bid;
  ulong start;
  int len;
  Page  *cache;
  Extent  *next;
};
```

⟨*struct* Mntcache 682b⟩≡ (682c)

```
struct Mntcache
{
  Qid qid;
  int dev;
  int type;
  QLock;
  Extent   *list;
  Mntcache *hash;
  Mntcache *prev;
  Mntcache *next;
};
```

⟨files/portdat_files.h 682c⟩≡

⟨*enum* blockflag 405c⟩

⟨*struct* Block 406a⟩

⟨*function* BLEN 406b⟩
⟨*function* BALLOC 406c⟩

⟨*enum* queuestate 371a⟩

```
// defined in qio.c
```

```
extern  uint  qiomaxatomic;

/*
 *  IO queues
 */
// was in qio.c
⟨struct Queue 371b⟩


// was in cache.c
⟨struct Extent 682a⟩

// was in cache.c
⟨struct Mntcache 682b⟩


⟨struct Mount 318c⟩

⟨struct Mhead 317c⟩



#include <fcall.h>

// was in devmnt.c
⟨struct Mntrpc 549a⟩

⟨struct Mnt 547a⟩



⟨struct Path 300b⟩

⟨enum accessnamec 282e⟩

⟨enum channelflag 272b⟩

⟨struct Chan 51e⟩


⟨struct Evalue 563a⟩

⟨struct Egrp 563b⟩


// internals

⟨struct Walkqid 303b⟩

⟨struct Dev 54a⟩


// 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 393a⟩
```

```
// used by with devmnt.c and sysfile.c
struct Bogus {
    Chan    *chan;
    Chan    *authchan;
    char    *spec;
    int flags;
};


//*****************************************************************************
// Internal to files/
//*****************************************************************************
```

⟨*constants tags* 547d⟩

```
// actually internal to devmnt.c and mnt.c
```
⟨*struct* Mntalloc 547b⟩
```
extern struct Mntalloc mntalloc;
```

E.7.2 files/portfns_files.h

E.7.3 files/chan.c

⟨files/chan.c 684⟩≡

⟨*kernel basic includes* 35a⟩

⟨*chan.c forward decl* ??⟩

⟨*chan.c debugging macro* ??⟩

```
enum
{
```
⟨*constant* PATHSLOP 300d⟩
⟨*constant* PATHMSLOP 329b⟩
```
};
```

⟨*struct* Chanalloc 273a⟩
⟨*global* chanalloc 273c⟩

⟨*struct* Elemlist 287a⟩

⟨*function* chanpath 312b⟩

⟨*function* isdotdot 302b⟩

⟨*function* emptystr 637a⟩

⟨*function* chandevreset 264a⟩

⟨*function* chandevinit 267b⟩

⟨*function* chandevshutdown 595a⟩

⟨*constructor* newchan 273d⟩

⟨*global* npath ??⟩

⟨*constructor* newpath 300c⟩

⟨*function* `copypath` 301a⟩

⟨*destructor* `pathclose` 301b⟩

⟨*function* `fixdotdotname` 302c⟩

⟨*function* `uniquepath` 301c⟩

⟨*function* `addelem` 302a⟩

⟨*destructor* `chanfree` 274a⟩

⟨*function* `cclose` 282b⟩

⟨*struct* `Clunkq` 295d⟩
⟨*global* `clunkq` 296a⟩

⟨*function* `ccloseq` 296b⟩

⟨*function* `clunkwork` 296c⟩

⟨*function* `closeproc` 296d⟩

⟨*function* `uniquechan` 274b⟩

⟨*function* `eqqid` 53a⟩

⟨*function* `eqchan` 53c⟩

⟨*function* `eqchantdqid` 53d⟩

⟨*constructor* `newmhead` 318a⟩

⟨*function* `cmount` 323e⟩

⟨*function* `cunmount` 339c⟩

⟨*function* `cclone` 307b⟩

⟨*function* `findmount` 328b⟩

⟨*function* `domount` 328c⟩

⟨*function* `undomount` 330d⟩

⟨*function* `ewalk` 306a⟩

⟨*global* `Edoesnotexist` 632a⟩

⟨*function* `walk` 304b⟩

⟨*function* `createdir` 334d⟩

⟨*function* `growparse` 288a⟩

⟨*function* `parsename` 287d⟩

⟨*function* `memrchr` 635b⟩

⟨*function* `namelenerror` 289c⟩

⟨*function* `nameerror` 290a⟩

⟨*function* `namec` 283d⟩

⟨*function* `skipslash` 288b⟩

⟨*global* `isfrog` 286⟩

⟨*function* `validname0` 285f⟩

⟨*function* `validname` 285e⟩

⟨*function* `validnamedup` 285d⟩

⟨*function* `error_if_not_dir` 299b⟩

⟨*destructor* `putmhead` 318b⟩

E.7.4 `files/dev.c`

⟨`files/dev.c` 686⟩≡
⟨*kernel basic includes* 35a⟩

```
// was in $CONF.c
```
⟨*global* `devtab` 54b⟩

⟨*function* `mkqid` 53b⟩

⟨*function* `devno` 275b⟩

⟨*function* `devdir` 398c⟩

⟨*function* `devgen` 399⟩

```
void
devreset(void)
{
}

void
devinit(void)
{
}

void
devshutdown(void)
{
}
```

⟨*function* `devattach` 275a⟩

⟨*function* `devclone` 400a⟩

⟨*function* `devwalk` 400b⟩

⟨*function* `devstat` 402⟩

⟨*function* devdirread 403c⟩

⟨*function* devpermcheck 403d⟩

⟨*function* devopen 404a⟩

```
void
devcreate(Chan*, char*, int, ulong)
{
    error(Eperm);
}
```

⟨*function* devbread 405a⟩

⟨*function* devbrwrite 405b⟩

```
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;
//}
```

E.7.5 files/allocb.c

⟨files/allocb.c 687⟩≡

⟨*kernel basic includes* 35a⟩

```
enum
{
    Hdrspc      = 64,       /* leave room for high-level headers */
    Bdead       = 0x51494F42,   /* "QIOB" */
};
```

⟨*struct* Ialloc 407c⟩

⟨*global* ialloc 407d⟩

⟨*function* _allocb 406e⟩

⟨*function* allocb 407b⟩

⟨*function* `iallocb` 408a⟩

⟨*function* `freeb` 408b⟩

⟨*function* `checkb` 409a⟩

⟨*function* `iallocsummary` 611e⟩

E.7.6 `files/env.c`

⟨`files/env.c` 688a⟩≡
⟨*kernel basic includes* 35a⟩

```
// this used to be in devenv.c, but to avoid backward deps I've splitted
// this file in 2
```
⟨*function* `closeegrp` 563d⟩

E.7.7 `files/file.c`

⟨`files/file.c` 688b⟩≡
⟨*kernel basic includes* 35a⟩

⟨*function* `fdtochan` 280c⟩

⟨*function* `openmode` 280a⟩

⟨*function* `fdclose` 281b⟩

⟨*function* `validstat` 315d⟩

E.7.8 `files/mnt.c`

⟨`files/mnt.c` 688c⟩≡
⟨*kernel basic includes* 35a⟩

```
// 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` 547c⟩

```
#define MAXRPC (IOHDRSZ+8192)
```

⟨*function* `freetag` 548a⟩

⟨*function* `mntfree` 548b⟩

⟨*function* `mntpntfree` 548c⟩

⟨*function* `muxclose` 548d⟩

⟨*function* `mntversion` 549c⟩

E.7.9 files/qio.c

⟨*global* qiomaxatomic 689a⟩≡ (689b)

```
enum
{
    Maxatomic   = 64*1024,
};

uint    qiomaxatomic = Maxatomic;
```

Uses Maxatomic-779 689a and qiomaxatomic 689a.

⟨files/qio.c 689b⟩≡

⟨*kernel basic includes* 35a⟩

⟨*qio.c globals cnt* ??⟩

⟨*qio.c global debugging* ??⟩

⟨*global* qiomaxatomic 689a⟩

⟨*function* ixsummary 611d⟩

⟨*function* freeblist 409b⟩

⟨*function* padblock 410⟩

⟨*function* blocklen 411a⟩

⟨*function* blockalloclen 411b⟩

⟨*function* concatblock 411c⟩

⟨*function* pullupblock 412⟩

⟨*function* pullupqueue 372a⟩

⟨*function* trimblock 413b⟩

⟨*function* copyblock 414⟩

⟨*function* adjustblock 415a⟩

⟨*function* pullblock 415b⟩

⟨*function* qget 373⟩

⟨*function* qdiscard 374⟩

⟨*function* qconsume 375⟩

⟨*function* qpass 376⟩

⟨*function* qpassnolim 377⟩

⟨*function* packblock 413a⟩

⟨*function* qproduce 378⟩

⟨*function* qcopy 379⟩

⟨*function* `qopen` 380a⟩

⟨*function* `qbypass` 380b⟩

⟨*function* `notempty` 380c⟩

⟨*function* `qwait` 381a⟩

⟨*function* `qaddlist` 381b⟩

⟨*function* `qremove` 382a⟩

⟨*function* `bl2mem` 416a⟩

⟨*function* `qputback` 382b⟩

⟨*function* `qwakeup_iunlock` 382c⟩

⟨*function* `qbread` 383⟩

⟨*function* `qread` 384⟩

⟨*function* `qnotfull` 385a⟩

⟨*function* `qbwrite` 385b⟩

⟨*function* `qwrite` 372b⟩

⟨*function* `qiwrite` 387⟩

⟨*function* `qfree` 389a⟩

⟨*function* `qclose` 389b⟩

⟨*function* `qhangup` 389c⟩

⟨*function* `qiclosed` 390a⟩

⟨*function* `qreopen` 390b⟩

⟨*function* `qlen` 390c⟩

⟨*function* `qwindow` 390d⟩

⟨*function* `qcanread` 391a⟩

⟨*function* `qsetlimit` 391b⟩

⟨*function* `qnoblock` 391c⟩

⟨*function* `qflush` 391d⟩

⟨*function* `qfull` 392⟩

E.7.10 `files/sysfile.c`

⟨`files/sysfile.c` 690⟩≡
⟨*kernel basic includes* 35a⟩

```
/*
 * The sys*() routines needn't poperror() as they return directly to syscall().
 */
```

⟨*function* `unlockfgrp` 279b⟩

⟨*function* `growfd` 278b⟩

⟨*function* `findfreefd` 278a⟩

⟨*function* `newfd` 277c⟩

⟨*function* `newfd2` 426b⟩

⟨*syscall fd2path* 312a⟩

⟨*syscall pipe* 426a⟩

⟨*syscall dup* 588b⟩

⟨*syscall open* 279d⟩

⟨*syscall close* 280b⟩

⟨*function* `unionread` 333b⟩

⟨*function* `unionrewind` 334b⟩

⟨*function* `dirfixed` 346⟩

⟨*function* `offsetof_name_direntry` 313c⟩

⟨*function* `direntry_setname` 313b⟩

⟨*function* `mountrock` 342d⟩

⟨*function* `mountrockread` 343⟩

⟨*function* `mountrewind` 344a⟩

⟨*function* `mountfix` 344b⟩

⟨*function* `read` 291⟩

⟨*syscall pread* 290b⟩

⟨*function* `write` 292c⟩

⟨*syscall pwrite* 292a⟩

⟨*function* `sseek` 293b⟩

⟨*syscall seek* 293a⟩

⟨*function* `pathlast` 314d⟩

⟨*syscall fstat* 314c⟩

⟨*syscall stat* 314b⟩

⟨*syscall chdir* 311a⟩

⟨*function* `bindmount` 322d⟩

⟨*syscall bind* 322b⟩

⟨*syscall mount* 337d⟩

⟨*syscall unmount* 338c⟩

⟨*syscall create* 307c⟩

⟨*syscall remove* 310d⟩

⟨*function* `wstat` 316⟩

⟨*syscall wstat* 315b⟩

⟨*syscall fwstat* 315c⟩

E.8 filesystems/

E.9 init/

E.9.1 init/portfns_init.h

E.9.2 init/user/preboot/initcode.c

E.9.3 init/user/boot/aux.c

E.9.4 init/user/boot/boot.c

E.9.5 init/user/boot/boot.h

E.9.6 init/user/boot/local.c

E.10 lib/

E.10.1 lib/lib.h

⟨*constant* `ERRMAX` 692a⟩≡ (? 0—1)
```
ERRMAX = 128, /* max length of error string */
```

⟨*constant* `KNAMELEN` 692b⟩≡ (? 0—1)
```
KNAMELEN = 28,  /* max length of name held in kernel */
```

⟨*function* `nelem` 692c⟩≡ (693c)
```
#define nelem(x)  (sizeof(x)/sizeof((x)[0]))
```

⟨*function* `offsetof` 692d⟩≡ (693c)
```
//@Scheck: not dead, actually exanded in macros.h
#define offsetof(s, m)  (ulong)(&(((s*)0)->m))
```

⟨*function* assert 693a⟩≡ (693c)
```
#define assert(x) do { if(x) {}else _assert("x");}while(0)
```

⟨*lib.h print decl* 693b⟩≡ (693c)
```
// 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 693c⟩≡
```
// 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 they prefered 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 ??⟩

/*
 * functions (mostly) linked in from libc.
 */
⟨function nelem 692c⟩
⟨function offsetof 692d⟩
⟨function assert 693a⟩

/*
 * mem routines
 */
⟨lib.h mem functions decl 635a⟩

/*
 * string routines
 */
⟨lib.h string functions decl 635c⟩

//redefined in the kernel
extern  int cistrcmp(char*, char*);
extern  int cistrncmp(char*, char*, int);

⟨enum utf 639a⟩

/*
 * rune routines
 */
⟨lib.h rune functions decl 639b⟩

extern  int abs(int);

/*
 * print routines
 */
```

```
typedef int (*Fmts)(Fmt*);
```
⟨*struct* `Fmt` 638a⟩

⟨*lib.h print decl* 693b⟩

⟨*lib.h printf functions decl* 637e⟩

⟨*lib.h pragmas* 638c⟩

⟨*lib.h fmt functions decl* 638b⟩

```
/*
 * one-of-a-kind
 */
```
⟨*lib.h strto functions decl* 637b⟩

```
//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* 252b⟩

```
/*
 * Syscall data structures
 */
```
⟨*enum* `mount` 317b⟩

⟨*enum* `open` 279c⟩

⟨*enum* `note` 419a⟩

⟨*enum* `miscsize` ??⟩

⟨*enum* `qidtype` 52c⟩

⟨*enum* `dirmode` 313a⟩

⟨*struct* `Qid` 52b⟩

⟨*struct* `DirEntry` 312c⟩

⟨*struct* `Waitmsg` 131e⟩

E.11 memory/

E.11.1 memory/portdat_memory.h

⟨`memory/portdat_memory.h` 694⟩≡

```
// 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 90a⟩

⟨enum Cachectl 521f⟩

⟨struct Page 89⟩

⟨type PageOrSwap 186c⟩

⟨struct Pagetable 95b⟩

// used to be arch-specific, but many constants were used in port/ code
⟨type PTExxx 166a⟩

⟨struct KImage 174a⟩

⟨enum segtype 94a⟩

⟨constant PG_ONSWAP 186b⟩

⟨function onswap 187a⟩
⟨function pagedout 177d⟩
⟨function swapaddr 196b⟩

#define SEGMAXSIZE  (PAGEDIRSIZE*PAGETABMAPMEM)

⟨struct Physseg 523d⟩

enum
{
    ⟨constant LRESPROF 473f⟩
};

⟨struct Segment 94c⟩

//*************************************************************************
// Internal to memory/
//*************************************************************************

// See also RMap in 386/

// actually internal to xalloc.c, but important so here
⟨constants holes 75c⟩

⟨struct Hole 75d⟩

⟨struct Xhdr 77a⟩

⟨struct Xalloc 75a⟩
//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 91b⟩

⟨*function* pghash 171b⟩

⟨*struct* Palloc 90b⟩

```
extern  Palloc  palloc;
```

⟨*constant* NFREECHAN 263b⟩

⟨*function* ihash 174d⟩

⟨*struct* Imagealloc 174b⟩

```
//IMPORTANT: static struct Imagealloc imagealloc; (segment.c)
// so have conf.nimage + 1 Kimages
extern  KImage  swapimage;
```

⟨*struct* Swapalloc 185a⟩

```
extern struct Swapalloc swapalloc;
```

E.11.2 memory/portfns_memory.h

E.11.3 memory/portdat_memory.c

⟨memory/portdat_memory.c 697a⟩≡
⟨*kernel basic includes* 35a⟩

⟨*global* palloc 90d⟩

⟨*global* swapalloc 185b⟩
⟨*global* swapimage 185d⟩

E.11.4 memory/xalloc.c

⟨memory/xalloc.c 697b⟩≡
⟨*kernel basic includes* 35a⟩

⟨*xalloc.c forward decl* ??⟩

```
//*********************************************************************************
// The global
//*********************************************************************************
```

⟨*global* xlists 76⟩

```
//*********************************************************************************
// Initialization
//*********************************************************************************
```

⟨*function* xinit 260⟩

```
//*********************************************************************************
// Functions
//*********************************************************************************
```

⟨*function* xspanalloc 80b⟩

⟨*function* xallocz 77d⟩

⟨*function* xalloc 77c⟩

⟨*function* xfree 78a⟩

⟨*function* xmerge 79⟩

⟨*function* xhole 78c⟩

```
//*********************************************************************************
// Debugging
//*********************************************************************************
```

⟨*function* xsummary 610a⟩

E.11.5 memory/pool.c

⟨*pool.c forward decl* 697c⟩≡ (698a)

```
typedef struct Private  Private;
```

Uses Private 81.

⟨memory/pool.c 698a⟩≡
⟨*kernel basic includes* 35a⟩

```
#include    <pool.h>
```

⟨*pool.c forward decl* 697c⟩

```
//*****************************************************************************
// Concurrency
//*****************************************************************************

// See Pool.private, for mutual exclusion on memory pools
```
⟨*pool.c struct Private* 81⟩

```
//*****************************************************************************
// Pool methods
//*****************************************************************************
```

⟨*function* poolprint 84b⟩

⟨*function* ppanic 84c⟩

⟨*function* plock 82a⟩

⟨*function* punlock 82b⟩

⟨*function* poolsummary 611a⟩

```
//*****************************************************************************
// The globals
//*****************************************************************************
```

⟨*global* pmainmem 83a⟩

⟨*global* pimagmem 83b⟩

⟨*global mainmem and imagmem* 84a⟩

⟨*function* mallosummary 610b⟩

E.11.6 memory/alloc.c

⟨memory/alloc.c 698b⟩≡
⟨*kernel basic includes* 35a⟩

```
#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* ??⟩

⟨*function* setmalloctag 87b⟩

⟨*function* setrealloctag 87c⟩

```
//*****************************************************************************
// Pool wrappers
//*****************************************************************************
```

⟨*function* smalloc 85c⟩

⟨*function* malloc 85a⟩

⟨*function* mallocz 86b⟩

⟨*function* mallocalign 86c⟩

⟨*function* free 85b⟩

⟨*function* realloc 86d⟩

⟨*function* msize 407a⟩

```
//*****************************************************************************
// kstr functions
//*****************************************************************************

//pad: was in chan.c
```

⟨*function* kstrcpy 636a⟩

⟨*function* kstrdup 636b⟩

E.11.7 memory/fault.c

⟨*function* checkpages 699⟩≡ (700)

```
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 100a, pagedout 694, print, qlock() 63f, qunlock() 64c, and up.

⟨memory/fault.c 700⟩≡

```
⟨kernel basic includes 35a⟩

⟨fault.c forward decl ??⟩

⟨function fault 165⟩

⟨function faulterror 179⟩

// for debugging SG_PHYSICAL
void    (*checkaddr)(ulong, Segment *, Page *);
ulong   addr2check;

⟨function fixfault 166b⟩

⟨function pio 178⟩

⟨function okaddr 101a⟩

⟨function validaddr 101c⟩

⟨function vmemchr 101d⟩

⟨function seg 100c⟩

⟨function checkpages 699⟩
```

E.11.8 memory/page.c

⟨memory/page.c 701⟩≡
⟨*kernel basic includes* 35a⟩

⟨*page.c forward decl* ??⟩

```
//*****************************************************************************
// Initialization
//*****************************************************************************
```

⟨*function* pageinit 261c⟩

```
//*****************************************************************************
// Functions
//*****************************************************************************
```

⟨*function* pageunchain 92c⟩

⟨*function* pagechaintail 93b⟩

⟨*function* pagechainhead 93a⟩

⟨*constructor* newpage 91c⟩

⟨*function* hasfreepages 191c⟩

⟨*destructor* putpage 92d⟩

⟨*function* auxpage ??⟩

⟨*global* dupretries 180b⟩

⟨*function* duppage 180c⟩

⟨*function* copypage 170d⟩

⟨*function* uncachepage 172d⟩

⟨*function* cachepage 171e⟩

⟨*function* cachedel 173⟩

⟨*function* lookpage 172a⟩

⟨*function* ptcpy 97a⟩

⟨*constructor* ptalloc 96c⟩

⟨*destructor* freept 96d⟩

⟨*function* pagenumber 527a⟩

⟨*function* checkpagerefs 525c⟩

⟨*function* portcountpagerefs 526c⟩

E.11.9 memory/segment.c

⟨memory/segment.c 702⟩≡
```
⟨kernel basic includes 35a⟩
#include "io.h"

//****************************************************************************
// Global
//****************************************************************************

⟨global imagealloc 174e⟩

//****************************************************************************
// Misc
//****************************************************************************

⟨global physseg 523f⟩
⟨global physseglock 523g⟩

⟨hook _globalsegattach 436d⟩

⟨segment.c forward decl ??⟩

//****************************************************************************
// Initialization
//****************************************************************************

⟨function imageinit 263c⟩

//****************************************************************************
// Functions
//****************************************************************************

⟨constructor newseg 97b⟩

⟨destructor putseg 98b⟩

⟨function relocateseg 129b⟩

⟨function dupseg 102b⟩

⟨function segpage 99e⟩

⟨constructor attachimage 175b⟩

⟨struct Irstats 628j⟩
⟨segment.c global irstats 629a⟩

⟨function imagereclaim 184⟩

⟨function imagechanreclaim 183b⟩

⟨destructor putimage 182d⟩

⟨function ibrk 104d⟩

⟨function mfreeseg 107⟩

⟨function isoverlap 438a⟩
```

⟨*function* `addphysseg` 524a⟩

⟨*function* `segattach` 437⟩

⟨*clock callback segclock* 474a⟩

```
// was in another file before
```
⟨*function* `data2txt` 177c⟩

E.11.10 `memory/swap.c`

⟨`memory/swap.c` 703a⟩≡
⟨*kernel basic includes* 35a⟩

⟨*swap.c forward decl* **??**⟩

⟨*global* `iolist` 189b⟩
⟨*global* `ioptr` 189c⟩

⟨*global* `genxxx` 195b⟩

⟨*function* `gentick` 195a⟩

⟨*function* `swapinit` 262b⟩

⟨*function* `newswap` 185e⟩

⟨*function* `putswap` 187b⟩

⟨*function* `dupswap` 187e⟩

⟨*function* `swapcount` 187g⟩

⟨*function* `kickpager` 191d⟩

⟨*function* `pager` 192⟩

⟨*function* `pageout` 189d⟩

⟨*function* `canflush` 194a⟩

⟨*function* `pagepte` 188⟩

⟨*function* `pagersummary` 611c⟩

⟨*function* `pageiocomp` 196a⟩

⟨*function* `executeio` 195d⟩

⟨*function* `needpages` 193a⟩

⟨*function* `setswapchan` 197⟩

E.11.11 `memory/sysmemory.c`

⟨`memory/sysmemory.c` 703b⟩≡
⟨*kernel basic includes* 35a⟩

```
// 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* 441c⟩

Uses PG_TXTFLUSH 521f and pagedout 694.

E.12 misc/

E.12.1 misc/portdat_misc.h

⟨misc/portdat_misc.h 704a⟩≡

```
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 363a⟩
};
```

⟨*struct* Cmdbuf 560b⟩

⟨*struct* Cmdtab 560a⟩

E.12.2 misc/portfns_misc.h

E.12.3 misc/parse.c

⟨misc/parse.c 704b⟩≡

⟨*kernel basic includes* 35a⟩

⟨*ncmdfield* 562⟩

⟨*function* parsecmd 560c⟩

⟨*function* `cmderror` 561a⟩

⟨*function* `lookupcmd` 561b⟩

E.12.4 misc/random.c

⟨`misc/random.c` 705a⟩≡
⟨*kernel basic includes* 35a⟩

⟨*struct* `Rb` 601d⟩

⟨*global* `rb` 601e⟩

⟨*struct* `rbnotfull` 602a⟩

⟨*struct* `rbnotempty` 602b⟩

⟨*function* `genrandom` 602c⟩

⟨*clock callback randomclock* 602d⟩

⟨*function* `randominit` 603a⟩

⟨*function* `randomread` 603b⟩

E.13 interrupts/

E.13.1 interrupts/portfns_interrupts.h

⟨`portfns_interrupts.h` 705b⟩≡

```
// <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.14 processes/

E.14.1 processes/portdat_processes.h

⟨`processes/portdat_processes.h` 705c⟩≡

```
// 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 51b⟩

// hash<enum<procstate>, string>, to debug
extern  char* statename[];

//---------------------------------------------------------------------
// Memory
//---------------------------------------------------------------------
⟨enum procseg 100a⟩

//---------------------------------------------------------------------
// Files
//---------------------------------------------------------------------

enum
{
   ⟨constant DELTAFD 277a⟩
};

⟨struct Fgrp 276a⟩


⟨function MOUNTH 320b⟩

⟨struct Pgrp 319d⟩

//---------------------------------------------------------------------
// System call
//---------------------------------------------------------------------

⟨constant MAXSYSARG 145a⟩

⟨struct Sargs 145b⟩

//---------------------------------------------------------------------
// Notes
//---------------------------------------------------------------------

enum {
    ⟨constant NNOTE 420d⟩
};

⟨enum notekind 418c⟩

⟨struct Note 419c⟩
extern Counter  noteidalloc;

//---------------------------------------------------------------------
// Process children waiting
//---------------------------------------------------------------------

⟨struct Waitq 131f⟩

//---------------------------------------------------------------------
// Synchronization (Rendez vous)
//---------------------------------------------------------------------

⟨function REND 447b⟩
```

⟨*struct* Rgrp 448a⟩

```
//------------------------------------------------------------------
// Alarms, timers
//------------------------------------------------------------------
// now in portdat_time.h

//------------------------------------------------------------------
// Scheduling
//------------------------------------------------------------------

enum {
    ⟨constant Npriq 198a⟩
    ⟨constant Nrq 200a⟩
};

⟨enum priority 198b⟩

⟨enum edfflags 533g⟩

⟨struct Edf 534a⟩

//------------------------------------------------------------------
// Error management
//------------------------------------------------------------------
enum {
    ⟨constant NERR 633a⟩
};

//------------------------------------------------------------------
// Stats, profiling
//------------------------------------------------------------------
⟨enum proctimer 471a⟩

//------------------------------------------------------------------
// Debugger
//------------------------------------------------------------------

⟨enum procctl 453b⟩

//------------------------------------------------------------------
// Misc
//------------------------------------------------------------------

⟨enum fpsavestatus 480a⟩

//*********************************************************************************
// Proc, the big one
//*********************************************************************************

⟨struct Proc 49f⟩

⟨macro waserror poperror 633d⟩


//*********************************************************************************
// Internal to process/
//*********************************************************************************
```

```
// Proc allocator (singleton), was actually in proc.c, but important so here
⟨struct Procalloc 110b⟩
//IMPORTANT: static struct Procalloc procalloc; (in proc.c)

⟨struct Schedq 200c⟩
// 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 234b⟩
//IMPORTANT: static Alarms alarms; (in alarm.c)
//IMPORTANT: static Rendez alarmr; (in alarm.c)

⟨struct Active 49d⟩
extern struct Active active;
```

E.14.2 processes/portfns_processes.h

E.14.3 processes/portdat_processes.c

⟨processes/portdat_processes.c 708a⟩≡

```
⟨kernel basic includes 35a⟩

⟨hook proctrace 467a⟩
⟨hook kproftimer 627b⟩

⟨global active 49e⟩
```

E.14.4 time/portfns_time.h

⟨time/portfns_time.h 708b⟩≡

```
// portclock.c
ulong   tk2ms(ulong);
⟨macro TK2MS 223a⟩
⟨macro MS2NS 222c⟩
#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*);
```

E.14.5 time/portdat_time.h

⟨time/portdat_time.h 709a⟩≡

```
⟨enum timermode 225b⟩

⟨type Txxx 222b⟩

⟨struct Timer 225a⟩

// was in clock.c
⟨struct Timers 225c⟩

// 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
```

E.14.6 time/alarm.c

⟨time/alarm.c 709b⟩≡

```
⟨kernel basic includes 35a⟩

⟨global alarms 234d⟩
⟨global alarmr 236b⟩

⟨kernel process alarmkproc 236c⟩

⟨function checkalarms 237b⟩

⟨function procalarm 235a⟩
```

E.14.7 time/portclock.c

⟨time/portclock.c 709c⟩≡

```
⟨kernel basic includes 35a⟩
```

```
#include "io.h"

#include <ureg.h>

enum {
    ⟨constant Maxtimerloops 230d⟩
};

⟨global timers 226c⟩
⟨global timersinited 259a⟩

⟨clock.c statistics 629c⟩

⟨function tadd 226d⟩

⟨function tdel 227b⟩

⟨function timeradd 227d⟩

⟨function timerdel 228c⟩

⟨clock callback hzclock 215f⟩

⟨interrupt callback timerintr 229d⟩

⟨function timersinit 259b⟩

⟨function addclock0link 231a⟩

⟨function tk2ms 223b⟩

⟨function ms2tk 223c⟩
```

E.14.8 processes/pgrp.c

⟨processes/pgrp.c 710⟩≡

```
// Process group, and Namespace.

⟨kernel basic includes 35a⟩

enum {
    ⟨constant Whinesecs 114a⟩
};

⟨global pgrpid 320e⟩
⟨global mountid 340b⟩

⟨function pgrpnote 576⟩

⟨constructor newpgrp 320f⟩

⟨constructor newrgrp 448c⟩

⟨destructor closergrp 448d⟩

⟨destructor closepgrp 321a⟩

⟨function pgrpinsert 341b⟩
```

⟨*function* pgrpcpy 341c⟩

⟨*function* dupfgrp 276c⟩

⟨*function* closefgrp 277b⟩

⟨*function* forceclosefgrp 295c⟩

⟨*constructor* newmount 319a⟩

⟨*destructor* mountfree 319c⟩

⟨*function* resrcwait 113c⟩

E.14.9 processes/proc.c

⟨Procctl *cases* 711a⟩+≡ (453b) ◁470a
```
Proc_exitbig,
```

⟨procctl() Proc_exitbig *case* 711b⟩≡ (453d)
```
case Proc_exitbig:
    arch_spllo();
    pexit("Killed: Insufficient physical memory", true);
    // Fallthrough
```
Uses Proc_exitbig 711a, pexit, and true.

⟨*function* killbig 711c⟩≡ (712c)
```
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 100a, Dead 51b, NSEG 100a, Proc_exitbig 711a, canqlock() 65a, conf 46b, eve 557a, mfreeseg() 107, print, procalloc-2458 110c, and qunlock() 64c.

⟨Proc *scheduling fields* 712a⟩+≡ (49f) ◁533d

```
    bool preempted;  /* true if this process hasn't finished the interrupt
            *  that last preempted it
            */
```

⟨*function* preempt 712b⟩≡ (712c)

```
    /*
     *  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 51b, active 49e, anyhigher() 215d, false, sched, true, and up.

⟨processes/proc.c 712c⟩≡

```
    ⟨kernel basic includes 35a⟩
    #include    <trace.h>

    //*********************************************************************************
    // Globals
    //*********************************************************************************

    ⟨global runq 200b⟩
    ⟨global runveq 200e⟩

    ⟨global procalloc 110c⟩

    ⟨global nrdy 201a⟩
    ⟨global noteidalloc 419b⟩

    ⟨proc.c statistics 629b⟩

    ⟨global pidalloc 110a⟩
```

```
enum
{
    ⟨constant Schedagain 218f⟩
    ⟨constant Scaling 218e⟩
};

⟨global statename 51c⟩

⟨proc.c forward decl ??⟩

//*****************************************************************************
// Error management
//*****************************************************************************

// see also waserror() and poperror() macro in portdat_processes.h

⟨function error 633b⟩

⟨function nexterror 634a⟩

⟨function exhausted 633c⟩

//*****************************************************************************
// Functions
//*****************************************************************************

⟨function schedinit 203f⟩

⟨function sched 202c⟩

⟨function anyready 200f⟩

⟨function anyhigher 215d⟩

⟨function hzsched 215c⟩

⟨function preempt 712b⟩

⟨function updatecpu 218h⟩

⟨function reprioritize 219⟩

⟨function queueproc 201b⟩

⟨function dequeueproc 201c⟩

⟨function ready 211a⟩

⟨function yield 210b⟩

⟨global balancetime 221a⟩

⟨function rebalance 220b⟩

⟨function runproc 204⟩

⟨function canpage 194b⟩

⟨function noprocpanic 113b⟩
```

⟨*constructor* `newproc` 111d⟩

⟨*function* `procwired` 216g⟩

⟨*function* `procpriority` 199a⟩

⟨*function* `procinit` 261a⟩

⟨*function* `sleep` 212e⟩

⟨*function* `tfn` 233f⟩

⟨*function* `twakeup` 234a⟩

⟨*function* `tsleep` 233a⟩

⟨*function* `wakeup` 214a⟩

⟨*function* `postnote` 424⟩

⟨*struct* `Broken` 451b⟩
⟨*global* `broken` 451c⟩

⟨*function* `addbroken` 451e⟩

⟨*function* `unbreak` 452a⟩

⟨*function* `freebroken` 452b⟩

⟨*function* `pexit` 134c⟩

⟨*function* `haswaitq` 133d⟩

⟨*function* `pwait` 132c⟩

⟨*function* `proctab` 570b⟩

⟨*function* `dumpaproc` 612b⟩

⟨*function* `procdump` 612a⟩

⟨*function* `procflushseg` 440c⟩

⟨*function* `scheddump` 611f⟩

⟨*function* `kproc` 139c⟩

⟨*function* `procctl` 453d⟩

⟨*function* `killbig` 711c⟩

⟨*function* `renameuser` 115a⟩

⟨*function* `accounttime` 624b⟩

⟨*function* `pidhash` 110f⟩

⟨*function* `pidunhash` 111a⟩

⟨*function* procindex 111b⟩

E.14.10 time/tod.c

⟨time/tod.c 715⟩≡

```
// TOD: Time Of Day.

⟨kernel basic includes 35a⟩

⟨tod.c forward decl ??⟩

/*
 * 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 237c⟩
⟨global tod 237d⟩

⟨function todinit 238c⟩

⟨function todsetfreq 238d⟩

⟨function todset 239a⟩

⟨function todget 239b⟩

⟨function todfix 240⟩

⟨function seconds 241a⟩

⟨function fastticks2us 241c⟩

⟨function ns2fastticks 241d⟩
```

⟨*function* `mk64fract` 242a⟩

E.14.11 processes/sysproc.c

⟨`processes/sysproc.c` 716a⟩≡
⟨*kernel basic includes* 35a⟩

```
#include    <tos.h>
#include    <a.out.h>
```

⟨*enum* `rfork` 115b⟩

⟨*sysproc.c forward decl* **??**⟩

⟨*syscall nop* 42b⟩

⟨*syscall rfork* 115c⟩

⟨*function* `l2be` 125a⟩

⟨*syscall exec* 123b⟩

⟨*function* `shargs` 478a⟩

⟨*syscall sleep* 232c⟩

⟨*syscall alarm* 234e⟩

⟨*syscall exits* 133e⟩

⟨*syscall await* 132a⟩

⟨*function* `werrstr` 634e⟩

⟨*function* `generrstr` 634d⟩

⟨*syscall errstr* 634c⟩

⟨*syscall notify* 420f⟩

⟨*syscall noted* 422⟩

⟨*syscall segbrk* 436b⟩

⟨*syscall segattach* 436c⟩

⟨*syscall segdetach* 438b⟩

⟨*syscall segfree* 439⟩

⟨*syscall brk* 104c⟩

⟨*syscall rendezvous* 448f⟩

E.14.12 processes/syssema.c

⟨`processes/syssema.c` 716b⟩≡

⟨*kernel basic includes* 35a⟩

```
/*
 * 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 443c⟩

⟨*function* semdequeue 443d⟩

⟨*function* semwakeup 444a⟩

⟨*function* semrelease 444b⟩

⟨*function* canacquire 444c⟩

⟨*function* semawoke 444d⟩

⟨*function* semacquire 445a⟩

⟨*function* tsemacquire 445b⟩

⟨*syscall semacquire* 446a⟩

⟨*syscall tsemacquire* 446b⟩

⟨*syscall semrelease* 447a⟩

E.15 security/

E.15.1 security/portfns_security.h

E.15.2 security/auth.c

⟨security/auth.c 718⟩≡
  ⟨*kernel basic includes* 35a⟩

```
#include    <authsrv.h>

//now in core/
//char  *eve;
//int iseve(void) { }
```

⟨*global* `hostdomain` 557d⟩

⟨*syscall fversion* 557e⟩

⟨*syscall fauth* 558a⟩

⟨*function* `userwrite` 558b⟩

⟨*function* `hostownerwrite` 114d⟩

⟨*function* `hostdomainwrite` 559⟩

E.16 syscalls/

E.16.1 syscalls/systab.h

E.16.2 syscalls/syscallfmt.c

⟨`syscalls/syscallfmt.c` 719a⟩≡

```
/*
 * Print functions for system call tracing.
 */
```

⟨*kernel basic includes* 35a⟩

```
#include "../port/systab.h"
```

⟨*function* `fmtrwdata` 465⟩

⟨*function* `fmtuserstring` 466a⟩

⟨*function* `syscallfmt` 460e⟩

⟨*function* `sysretfmt` 464⟩

E.16.3 syscalls/systab.c

⟨`syscalls/systab.c` 719b⟩≡

⟨*kernel basic includes* 35a⟩

```
#include "../port/systab.h"
```

⟨*systab.c forward decl* ??⟩

```
//coupling: debuggers/acid/conf/syscall
//coupling: debuggers/acid/conf/truss
```

⟨*global* `systab` 41f⟩

⟨*global* `nsyscall` 42a⟩

⟨*global* `sysstab` ??⟩

E.17 ../include/arch/arm/

E.17.1 ../include/arch/arm/ureg.h

⟨arm/ureg.h 720a⟩≡
```
typedef struct Ureg Ureg;
```
⟨*struct* Ureg*(arm)* 26c⟩

E.18 arch/arm/

E.18.1 arch/arm/dat_arch.h

⟨arch/arm/dat_arch.h 720b⟩≡

⟨*struct* Soc*(arm)* 54c⟩

```
extern Soc soc;
```

E.18.2 arch/arm/arm.h

⟨arch/arm/arm.h 720c⟩≡
```
/*
 * ARM-specific definitions for armv6 (arm11), armv7 (cortex-a7 and -a8)
 * these are used in C and assembler
 */

/*
 * Program Status Registers
 */
```
⟨*type* PsrMode*(arm)* 28a⟩

⟨*type* PsrDisable*(arm)* 28b⟩

```
/*
 * Coprocessors
 */
```
⟨*type* Coprocessors*(arm)* 29a⟩

```
/*
 * Primary (CRn) CpSC registers.
 */
```
⟨*type CpSC primary registers(arm)* 29b⟩

```
// CpSC secondary registers
```

⟨*type CpID secondary registers(arm)* 502b⟩
⟨*type CpID opcode2(arm)* 502c⟩
⟨*type CpID CpIDid op1(arm)* 515c⟩

⟨*type CpCONTROL opcode2(arm)* 29c⟩
⟨*type CpCONTROL CpMainctl(arm)* 29d⟩
⟨*type CpCONTROL CpAuxctl(arm)* 506e⟩

⟨*type* CpTTB*(arm)* 503a⟩

⟨*type* CpFSR*(arm)* 153e⟩

⟨*type CpCACHE secondary registers(arm)* 497e⟩
⟨*type CpCACHE opcode2(arm)* 497f⟩

⟨*type CpTLB secondary registers(arm)* 162h⟩
⟨*type CpTLB opcode2(arm)* 162i⟩

⟨*type CpCLD secondary registers(arm)* 504d⟩
⟨*type CpCLD opcode2(arm)* 504e⟩

⟨*type* `CpTIMER`*(arm)* 504b⟩

⟨*type CpSPM secondary registers(arm)* 475e⟩
⟨*type CpSPM opcode2(arm)* 475f⟩

```
// For MCRR
```

⟨*type CpCACHERANGE opcode2(arm)* 496f⟩

```
// Misc
```

⟨*type CpTTB cache control bits(arm)* 503b⟩

```
/*
 * MMU page table entries.
 */
```
⟨*constant* `Mbz`*(arm)* 160e⟩

⟨*constant* `Fault`*(arm)* 161c⟩

⟨*type* `PageDirGranularity`*(arm)* 160d⟩
⟨*type* `PageTableGranularity`*(arm)* 160f⟩

⟨*type* `PageTableEntryAttribute1`*(arm)* 496h⟩

⟨*constant* `Dom0`*(arm)* 161d⟩

⟨*type* `PageTableEntryAttribute2`*(arm)* 502e⟩

⟨*type* `PageTableEntryAttribute3`*(arm)* 161b⟩

⟨*type* `DAC`*(arm)* 161a⟩

⟨*macro* `F`*(arm)* 161g⟩

⟨*macro* `AP`*(arm)* 161f⟩
⟨*macro* `L1AP`*(arm)* 161e⟩
```
/* L2AP differs between armv6 and armv7 -- see l2ap in arch*.c */
```
⟨*macro* `DAC`*(arm)* 161h⟩

⟨*constant* `HVECTORS`*(arm)* 28c⟩

E.18.3 arch/arm/arminstr.ha

⟨*arch/arm/arminstr.ha* 721⟩≡
```
/*
 * armv6/v7 machine assist, definitions
 *
 * loader uses R11 as scratch.
```

```
 */
```

⟨*macro PADDR bis(arm)* 40e⟩
⟨*macro L1X bis(arm)* 160c⟩

```
/*
 * new instructions
 */
```
⟨*instruction ISB(arm)* 497c⟩
⟨*instruction DSB(arm)* 497d⟩
⟨*instruction BARRIERS(arm)* 501h⟩

⟨*instruction MCRR(arm)* 496g⟩
⟨*instruction MRRC(arm)* 754b⟩

⟨*instruction LDREX(arm)* 513b⟩
⟨*instruction STREX(arm)* 513c⟩
⟨*instruction CLREX(arm)* 513d⟩

⟨*instruction CPSID(arm)* 515b⟩

⟨*instruction CPUID(arm)* 502d⟩

E.18.4 arch/arm/io.h

⟨arch/arm/io.h 722a⟩≡
```
// could be in dat_interrupts.h?
```
⟨*enum* IRQ*(arm)* 156b⟩
⟨*macro* IRQDMA*(arm)* 829⟩

⟨*enum* Dma*(arm)* 830a⟩

⟨*enum* Power*(arm)* 529c⟩

```
// could be in dat_time.h?
```
⟨*enum* Clock*(arm)* 727b⟩

E.18.5 arch/arm/coproc.c

⟨*enum* _anon_*(arm)* 722b⟩≡ (724b)
```
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 34b.

⟨*function* setupcpop*(arm)* 722c⟩≡ (724b)
```
// 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 722b, RETinst-2383 722b, and Regmask-2385 722b.

⟨*function* cprd*(arm)* 723a⟩≡ (724b)

```
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() 722c.

⟨*function* cpwr*(arm)* 723b⟩≡ (724b)

```
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() 722c.

⟨*function* cprdsc*(arm)* 723c⟩≡ (724b)

```
ulong
cprdsc(int op1, int crn, int crm, int op2)
{
```

```
    return cprd(CpSC, op1, crn, crm, op2);
}
```
Uses CpSC 29a and cprd() 723a.

⟨*function* cpwrsc*(arm)* 724a⟩≡ (724b)
```
void
cpwrsc(int op1, int crn, int crm, int op2, ulong val)
{
    cpwr(CpSC, op1, crn, crm, op2, val);
}
```
Uses CpSC 29a and cpwr() 723b.

⟨arch/arm/coproc.c 724b⟩≡
```
/*
 * 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* 35a⟩

```
#include "io.h"
#include "arm.h"
```

⟨*enum* _anon_*(arm)* 722b⟩

```
// pointer to ulong function void
typedef ulong (*Pufv)(void);
// pointer to void function ulong
typedef void  (*Pvfu)(ulong);
```

⟨*function* setupcpop*(arm)* 722c⟩

⟨*function* cprd*(arm)* 723a⟩

⟨*function* cpwr*(arm)* 723b⟩

⟨*function* cprdsc*(arm)* 723c⟩

⟨*function* cpwrsc*(arm)* 724a⟩

```
/* floating point */
```

⟨*function* setupfpctlop*(arm)* 486a⟩

⟨*function* fprd*(arm)* 486b⟩

⟨*function* fpwr*(arm)* 486c⟩

⟨*function* setupfpop*(arm)* 487a⟩

⟨*function* fpsavereg*(arm)* 487c⟩

⟨*function* fprestreg*(arm)* 487b⟩

E.18.6 arch/arm/vcore.c

⟨*enum* _anon_*(*(arch/arm/vcore.c)(arm)*)* 724c⟩≡ (729)

```
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)*)* 725a⟩≡ (729)

```
enum {
    ChanFb      = 1,
    ChanProps   = 8,
    ChanMask    = 0xF,
};
```

⟨*enum* _anon_*(*(arch/arm/vcore.c)3(arm)*)* 725b⟩≡ (729)

```
enum {
    Req         = 0x0,
    RspOk       = 0x80000000,
    TagResp     = 1<<31,
};
```

⟨*enum* _anon_*(*(arch/arm/vcore.c)4(arm)*)* 725c⟩≡ (729)

```
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)* 726a⟩≡ (729)
```
struct Prophdr {
    u32int  len;
    u32int  req;
    u32int  tag;
    u32int  tagbuflen;
    u32int  taglen;
    u32int  data[1];
};
```

⟨*function* vcwrite*(arm)* 726b⟩≡ (729)
```
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 725a, Full-2351 724c, MAILBOX-2345 32b, NRegs-2354 724c, Status-2350 724c, Write-2347 724c, and arch_coherence.

⟨*function* vcread*(arm)* 726c⟩≡ (729)
```
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 725a, Empty-2352 724c, MAILBOX-2345 32b, Read-2346 724c, Status-2350 724c, and arch_coherence.

⟨*function* vcreq*(arm)* 726d⟩≡ (729)
```
/*
 * 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 725a, PTR2UINT 41a, Req-2358 725b, RspOk-2359 725b, TagResp-2360 725b, VCBUFFER 358a, dmaaddr() 831h, false, true, vcread() 726c, and vcwrite() 726b.

⟨*function* getethermac*(arm)* 727a⟩≡ (729)

```
/*
 * 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 725c and vcreq() 726d.

⟨*enum* Clock*(arm)* 727b⟩≡ (722a)

```
enum {
```

```
    ClkEmmc     = 1,
    ClkUart,
    ClkArm,
    ClkCore,
    ClkV3d,
    ClkH264,
    ClkIsp,
    ClkSdram,
    ClkPixel,
    ClkPwm,
};
```

⟨*function* `getclkrate`*(arm)* 728a⟩≡ (729)

```
/*
 * 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` 725c and `vcreq()` 726d.

⟨*function* `setclkrate`*(arm)* 728b⟩≡ (729)

```
/*
 * 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` 725c and `vcreq()` 726d.

⟨*function* `getcputemp`*(arm)* 728c⟩≡ (729)

```
/*
 * 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` 725c and `vcreq()` 726d.

⟨arch/arm/vcore.c 729⟩≡
⟨*kernel basic includes* 35a⟩

```
/*
 * Mailbox interface with videocore gpu
 */
```

⟨*constant* MAILBOX*(arm)* 32b⟩

```
typedef struct Fbinfo Fbinfo;
typedef struct Prophdr Prophdr;
```

⟨*enum* _anon_*(*(arch/arm/vcore.c)(arm)*)* 724c⟩
⟨*enum* _anon_*(*(arch/arm/vcore.c)2(arm)*)* 725a⟩

⟨*enum* _anon_*(*(arch/arm/vcore.c)3(arm)*)* 725b⟩

⟨*enum* _anon_*(*(arch/arm/vcore.c)4(arm)*)* 725c⟩

⟨*struct* Fbinfo*(arm)* 32c⟩

⟨*struct* Prophdr*(arm)* 726a⟩

⟨*function* vcwrite*(arm)* 726b⟩

⟨*function* vcread*(arm)* 726c⟩

⟨*function* vcreq*(arm)* 726d⟩

⟨*function* fbdefault*(arm)* 359a⟩

⟨*function* fbinit*(arm)* 358b⟩

⟨*function* fbblank*(arm)* 359b⟩

⟨*function* setpower*(arm)* 530a⟩

⟨*function* getpower*(arm)* 530b⟩

⟨*function* getethermac*(arm)* 727a⟩

⟨*function* getboardrev*(arm)* 264g⟩

⟨*function* getfirmware*(arm)* 265a⟩

⟨*function* getramsize*(arm)* 253d⟩

⟨*function* getclkrate*(arm)* 728a⟩

⟨*function* setclkrate*(arm)* 728b⟩

⟨*function* getcputemp*(arm)* 728c⟩

Uses Fbinfo 32c and Prophdr 726a.

E.19 concurrency/arm/

E.19.1 concurrency/arm/spl.s

⟨concurrency/arm/spl.s 730a⟩≡
```
#include "mem.h"
#include "arm.h"
#include "arminstr.ha"

⟨function arch_splhi(arm) 56a⟩

⟨function arch_spllo(arm) 56b⟩

⟨function arch_splx(arm) 56c⟩

TEXT spldone(SB), 1, $0             /* end marker for devkprof.c */
    RET

⟨function arch_islo(arm) 56d⟩


⟨function arm_arch_coherence(arm) 514h⟩


// FIQ
⟨function splfhi(arm) 529b⟩
```

E.19.2 concurrency/arm/tas_raspi1.s

⟨concurrency/arm/tas_raspi1.s 730b⟩≡
```
TEXT    _tas(SB), $-4
⟨function arch_tas(raspberry pi1)(arm) 57⟩
```

E.19.3 concurrency/arm/tas_raspi2.s

⟨concurrency/arm/tas_raspi2.s 730c⟩≡
```
#include "mem.h"
#include "arm.h"
#include "arminstr.ha"

⟨instruction DMB(arm) 513e⟩

/* tas/cas strex debugging limits; started at 10000 */
#define MAXSC 100000

TEXT    _tas(SB), $-4           /* _tas(ulong *) */
⟨function arch_tas(raspberry pi2)(arm) 513f⟩
```

E.19.4 concurrency/arm/concurrency_raspi1.c

⟨function ainc(arm) 730d⟩≡ (731b)
```
// overwrite libc!
int
ainc(int *p)
{
    int s, v;
```

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

⟨*function* adec*(arm)* 731a⟩≡ (731b)

```
int
adec(int *p)
{
    int s, v;

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

⟨concurrency/arm/concurrency_raspi1.c 731b⟩≡

```
/*
 * atomic ops
 * make sure that we don't drag in the C library versions
 */
```

⟨*kernel basic includes* 35a⟩

⟨*function* arch_xdec*(raspberry pi1)(arm)* 58b⟩

⟨*function* arch_xinc*(raspberry pi1)(arm)* 58a⟩

⟨*function* ainc*(arm)* 730d⟩

⟨*function* adec*(arm)* 731a⟩

⟨*function* cas32*(arm)* 443b⟩

⟨*function* arch_cmpswap*(raspberry pi1)(arm)* 443a⟩

E.19.5 concurrency/arm/concurrency_raspi2.c

⟨concurrency/arm/concurrency_raspi2.c 731c⟩≡

⟨*kernel basic includes* 35a⟩

```
// 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);
}
```

E.20 core/arm/

E.20.1 core/arm/dat.h

⟨*constant* STAGESIZE*(arm)* 732a⟩≡ (732b)

```
// was a default in portdat.h
#define STAGESIZE 64 // for struct Uart
```

⟨core/arm/dat.h 732b⟩≡

```
// pad's memory pointers (used in Lock so must be early)
⟨pad memory pointer types(arm) 40a⟩
//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) 732a⟩

⟨dat.h includes(arm) 35b⟩

// 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) 48f⟩
⟨global up(arm) 51d⟩

// in main.c (used in mmu.c)
extern ulong memsize;

⟨constant KMESGSIZE(arm) 617b⟩

// 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*
```

E.20.2 core/arm/dat_core.h

⟨core/arm/dat_core.h 732c⟩≡

```
⟨type PTE(arm) 161l⟩

⟨struct Arch_Cpu(arm) 48e⟩
```

E.20.3 core/arm/dat_forward.h

⟨core/arm/dat_forward.h 733a⟩≡

```
// 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 733a.

E.20.4 core/arm/fns.h

⟨core/arm/fns.h 733b⟩≡

⟨*fns.h includes(arm)* 35c⟩

```
// 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)* 521d⟩
⟨*macro* arch_cycles*(arm)* 475d⟩
⟨*macro* arch_countpagerefs*(arm)* 526a⟩
⟨*macro* arch_intrenable*(arm)* 157e⟩

```
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)* ??⟩
⟨*macro* `sdmalloc`*(arm)* ??⟩

```
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)* 41a⟩
⟨*macro* `UINT2PTR`*(arm)* 41b⟩

⟨*macro* `KADDR`*(arm)* 40h⟩
⟨*macro* `PADDR`*(arm)* 40d⟩

E.21 buses/arm/

E.22 devices/

E.22.1 devices/keyboard/arm/kbd.c

⟨devices/keyboard/arm/kbd.c 736a⟩≡
  ⟨*kernel basic includes* 35a⟩

```
// 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)* 352a⟩

E.22.2 devices/mouse/arm/mouse.c

⟨devices/mouse/arm/mouse.c 736b⟩≡
  ⟨*kernel basic includes* 35a⟩

```
// 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) { }
```

E.22.3 devices/sys/arm/devarch.c

⟨*enum* _anon_*(*(devices/sys/arm/devarch.c)(arm)*)* 736c⟩≡ (740a)

```
enum {
    Qdir = 0,
    Qbase,

    Qmax = 16,
};
```

⟨*global* readfn*(arm)* 736d⟩≡ (740a)

```
static Rdwrfn *readfn[Qmax];
```

Uses Qmax-851 736c.

⟨*global* writefn*(arm)* 736e⟩≡ (740a)

```
static Rdwrfn *writefn[Qmax];
```

Uses Qmax-851 736c.

⟨*global* archdir*(arm)* 736f⟩≡ (740a)

```
static Dirtab archdir[Qmax] = {
    ".",        { Qdir, 0, QTDIR }, 0,  0555,
};
```

Uses QTDIR 52c, Qdir-849 736c, and Qmax-851 736c.

⟨*global* `archwlock`*(arm)* 737a⟩≡ (740a)
```
Lock archwlock; /* the lock is only for changing archdir */
```

⟨*global* `narchdir`*(arm)* 737b⟩≡ (740a)
```
int narchdir = Qbase;
```
Uses `Qbase-850` 736c.

⟨*function* `addarchfile`*(arm)* 737c⟩≡ (740a)
```
/*
 * 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` 736c, `archdir-854` 736f, `lock()` 58d, `readfn-852` 736d, `unlock()` 60d, and `writefn-853` 736e.

⟨*function* `archattach`*(arm)* 737d⟩≡ (740a)
```
static Chan*
archattach(char* spec)
{
    return devattach('P', spec);
}
```
Uses `devattach()` 275a.

⟨*function* `archwalk`*(arm)* 737e⟩≡ (740a)
```
Walkqid*
archwalk(Chan* c, Chan *nc, char** name, int nname)
```

```
{
    return devwalk(c, nc, name, nname, archdir, narchdir, devgen);
}
```
Uses archdir-854 736f, devgen() 399, and devwalk() 400b.

⟨*function* archstat*(arm)* 738a⟩≡ (740a)
```
static int
archstat(Chan* c, uchar* dp, int n)
{
    return devstat(c, dp, n, archdir, narchdir, devgen);
}
```
Uses archdir-854 736f, devgen() 399, and devstat() 402.

⟨*function* archopen*(arm)* 738b⟩≡ (740a)
```
static Chan*
archopen(Chan* c, int omode)
{
    return devopen(c, omode, archdir, narchdir, devgen);
}
```
Uses archdir-854 736f and devgen() 399.

⟨*function* archclose*(arm)* 738c⟩≡ (740a)
```
static void
archclose(Chan*)
{
}
```

⟨*function* archread*(arm)* 738d⟩≡ (740a)
```
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 736c, archdir-854 736f, devdirread() 403c, devgen() 399, and readfn-852 736d.

⟨*function* archwrite*(arm)* 738e⟩≡ (740a)
```
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 736e.

⟨*global* `archdevtab`*(arm)* 739a⟩≡ (740a)

```
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()` 405a, `devbwrite()` 405b, `devcreate()` 686, `devremove()` 686, `devreset()` 686, `devshutdown()` 686, and `devwstat()` 686.

⟨*function* `cputyperead`*(arm)* 739b⟩≡ (740a)

```
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()` 637d.

⟨*function* `cputempread`*(arm)* 739c⟩≡ (740a)

```
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()` 728c and `readstr()` 637d.

⟨*function* `archinit`*(arm)* 739d⟩≡ (740a)

```
void
archinit(void)
{
    addarchfile("cputype", 0444, cputyperead, nil);
    addarchfile("cputemp", 0444, cputempread, nil);
}
```

Uses `cputempread()` 739c.

⟨devices/sys/arm/devarch.c 740a⟩≡
```
⟨kernel basic includes 35a⟩

#include "io.h"

⟨enum _anon_((devices/sys/arm/devarch.c)(arm)) 736c⟩

typedef long Rdwrfn(Chan*, void*, long, vlong);
void archinit(void);

⟨global readfn(arm) 736d⟩
⟨global writefn(arm) 736e⟩

⟨global archdir(arm) 736f⟩

⟨global archwlock(arm) 737a⟩
⟨global narchdir(arm) 737b⟩

⟨function addarchfile(arm) 737c⟩

⟨function archattach(arm) 737d⟩

⟨function archwalk(arm) 737e⟩

⟨function archstat(arm) 738a⟩

⟨function archopen(arm) 738b⟩

⟨function archclose(arm) 738c⟩

⟨function archread(arm) 738d⟩

⟨function archwrite(arm) 738e⟩

⟨global archdevtab(arm) 739a⟩

⟨function cputyperead(arm) 739b⟩

⟨function cputempread(arm) 739c⟩

⟨function archinit(arm) 739d⟩
```

E.23 init/arm/

E.23.1 init/arm/start.s

⟨init/arm/start.s 740b⟩≡
```
/*
 * 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) 247⟩
```

E.23.2 init/arm/startv6.s

⟨init/arm/startv6.s 741a⟩≡
```
/*
 * 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)* 248⟩

⟨*function* _startpg*(raspberry pi1)(arm)* 250a⟩

⟨*function* arch_idlehands*(raspberry pi1)(arm)* 221c⟩

E.23.3 init/arm/startv7.s

⟨init/arm/startv7.s 741b⟩≡
```
/*
 * 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)* 512c⟩

⟨*function* armstart*(raspberry pi2)(arm)* 504f⟩

⟨*function* _startpg*(raspberry pi2)(arm)* 506f⟩

⟨*function* cpureset*(raspberry pi2)(arm)* 509⟩

⟨*function* _startpg2*(raspberry pi2)(arm)* 510⟩

⟨*function* cpidget*(raspberry pi2)(arm)* 512a⟩

⟨*function* arch_idlehands*(raspberry pi2)(arm)* 512b⟩

E.23.4 init/arm/main.c

⟨*function* getconf*(arm)* 741c⟩≡ (742b)
```
char*
getconf(char *name)
{
    USED(name);
    //int i;
    //
    //i = findconf(name);
    //if(i >= 0)
    //  return confval[i];
```

```
    return nil;
}
```

⟨*function* `main_arch_isaconfig`*(arm)* 742a⟩≡ (742b)

```
/*
 * 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` and `cistrncmp()`.

⟨`init/arm/main.c` 742b⟩≡

⟨*kernel basic includes* 35a⟩

```
#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)* 264e⟩
⟨*constant* `Minfirmdate`*(arm)* 264f⟩

```
// 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)* 741c⟩

```
//=========================================================================
// Boot parameters, see bootconf.c (not used by pad)
//=========================================================================

//=========================================================================
// Cpu init
//=========================================================================
```
⟨*function* `arch__cpuinit` *(arm)* 252d⟩

⟨*function* `arch__cpu0init` *(arm)* 252c⟩

⟨*function* `machon` *(arm)* 253a⟩

```
//=========================================================================
// Conf init
//=========================================================================
```

⟨*global* `memsize` *(arm)* 254c⟩

⟨*function* `arch__confinit` *(arm)* 254d⟩

```
//=========================================================================
// First process init
//=========================================================================
```

⟨*global* `sp` *(arm)* 265b⟩

```
// kernel space instructions executed by first process
```
⟨*function* `init0` *(arm)* 267a⟩

```
// 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) 265d⟩


//****************************************************************************
// Shutdown/reboot
//****************************************************************************

⟨function shutdown((init/arm/main.c)(arm)) 595e⟩

⟨function main_arch_exit(arm) 595c⟩

⟨function arch_reboot(arm) 596b⟩

//****************************************************************************
// Misc
//****************************************************************************

⟨function main_arch_isaconfig(arm) 742a⟩

⟨function arch_memorysummary(arm) 611b⟩

//****************************************************************************
// Main entry point!
//****************************************************************************

⟨function launchinit(arm) 507a⟩

⟨function main(arm) 250b⟩
```

E.23.5 init/arm/raspi1.c

⟨function archether(raspberry pi1)(arm) 744a⟩≡ (744b)

```
int
archether(unsigned ctlrno, Ether *ether)
{
    ether->type = "usb";
    ether->ctlrno = ctlrno;
    ether->irq = -1;
    ether->nopt = 0;
    return 1;
}
```

⟨init/arm/raspi1.c 744b⟩≡

```
/*
 * bcm2835 (e.g. original raspberry pi) architecture-specific stuff
 */
⟨kernel basic includes 35a⟩

#include "io.h"
#include "arm.h"

#include "../port/netif.h"
#include "../port/etherif.h"

⟨global soc(raspberry pi1)(arm) 54d⟩
```

⟨*function* cputype2name*(raspberry pi1)(arm)* 253c⟩

⟨*function* arch_cpuidprint*(raspberry pi1)(arm)* 253b⟩

⟨*function* getncpus*(raspberry pi1)(arm)* 507e⟩

⟨*function* startcpus*(raspberry pi1)(arm)* 507d⟩

⟨*function* archether*(raspberry pi1)(arm)* 744a⟩

⟨*function* l2ap*(raspberry pi1)(arm)* 161j⟩

E.23.6 init/arm/raspi2.c

⟨init/arm/raspi2.c 745⟩≡

```
/*
 * bcm2836 (e.g.raspberry pi 2) architecture-specific stuff
 */
```

⟨*kernel basic includes* 35a⟩

```
#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)* 503e⟩

⟨*global* soc*(raspberry pi2)(arm)* 501i⟩

⟨*struct* Mbox*(arm)* 503c⟩
⟨*struct* Mboxes*(arm)* 503d⟩

```
enum {
    Mboxregs    = 0x80
};

static Lock startlock[MAXCPUS + 1];

// in startv7.s
extern u32int cpidget(void);
```

⟨*function* cputype2name*(raspberry pi2)(arm)* 511d⟩

⟨*function* arch_cpuidprint*(raspberry pi2)(arm)* 511c⟩

⟨*function* getncpus*(raspberry pi2)(arm)* 507f⟩

⟨*function* startcpu*(raspberry pi2)(arm)* 508c⟩

⟨*function* startcpus*(raspberry pi2)(arm)* 508b⟩

```
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)* 502g⟩

```
// in main.c
extern void arch__cpuinit(void);
extern void machon(uint);
```

⟨*function* cpustart*(raspberry pi2)(arm)* 511a⟩

Uses Mbox 503c and Mboxes 503d.

E.23.7 init/user/preboot/arm/init9.s

E.23.8 init/arm/rebootcode.s

⟨init/arm/rebootcode.s 746a⟩≡

```
/*
 * armv6/armv7 reboot code
 */
#include "mem.h"
#include "arm.h"
#include "arminstr.ha"
```

⟨*constant* PTEDRAM*(arm)* 598b⟩

⟨*instruction WFI(arm)* 598c⟩
⟨*instruction WFE(arm)* 598d⟩

⟨*function main(rebootcode.s)(arm)* 598e⟩

E.24 memory/arm/

E.24.1 memory/arm/dat_memory.h

⟨memory/arm/dat_memory.h 746b⟩≡

```
/*
 *  MMU stuff in proc
 */
```

⟨*constant* NCOLOR*(arm)* 521e⟩

⟨*struct* Arch_Proc*(arm)* 50⟩

```
/*
 * Fake kmap.
 */
typedef void        Arch_KMap;
```

⟨*macro* VA*(arm)* 162e⟩

E.24.2 memory/arm/mem.h

⟨memory/arm/mem.h 747⟩≡

```
/*
 * Memory and machine-specific definitions.  Used in C and assembler.
 */
```

⟨*constant* KiB*(arm)* 34f⟩
⟨*constant* MiB*(arm)* 34g⟩
⟨*constant* GiB*(arm)* 34h⟩

```
/*
 * Sizes
 */
```

⟨*constant* BY2PG*(arm)* 31a⟩
⟨*constant* PGSHIFT*(arm)* 31b⟩

⟨*constant* MAXCPUS*(arm)* 49b⟩
⟨*constant* CPUSIZE*(arm)* 48b⟩
⟨*constant* L1SIZE*(arm)* 507b⟩

⟨*macro* STACKALIGN*(arm)* 265c⟩

```
/*
 * Magic registers
 */
```

⟨*constant* UP*(arm)* 144b⟩
⟨*constant* CPU*(arm)* 144a⟩

```
/*
 * 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)* 36a⟩
⟨*constant* KSEGM*(arm)* 40g⟩

⟨*constant* KZERO*(arm)* 36b⟩

⟨*constant* CONFADDR*(arm)* 757d⟩

```
// L1 must be 16KiB aligned so this dictates the organization below
```

⟨*constant* CPUADDR*(arm)* 40c⟩
⟨*constant* L2*(arm)* 249b⟩
⟨*constant* VCBUFFER*(arm)* 358a⟩
⟨*constant* FIQSTKTOP*(arm)* 527e⟩
⟨*constant* L1*(arm)* 249a⟩

⟨*constant* KTZERO*(arm)* 36c⟩

⟨*constant* VIRTIO*(arm)* 31c⟩
⟨*constant* FRAMEBUFFER*(arm)* 32d⟩

⟨*constant* UZERO*(arm)* 37a⟩
⟨*constant* UTZERO*(arm)* 37b⟩
⟨*macro* UTROUND*(arm)* 125b⟩
⟨*constant* USTKTOP*(arm)* 37c⟩
⟨*constant* USTKSIZE*(arm)* 37d⟩
⟨*constant* TSTKTOP*(arm)* 127b⟩
⟨*constant* TSTKSIZ*(arm)* 127c⟩

⟨*constant* REBOOTADDR*(arm)* 595d⟩

```
/*
 * Legacy...
 */
//old: was 64 in bcm-latest, but then get panic in _allocb
```
⟨*constant* BLOCKALIGN*(arm)* 406d⟩
⟨*constant* KSTACK*(arm)* 205g⟩

```
/*
 * Sizes
 */
```
⟨*constant* BY2WD*(arm)* 26a⟩
⟨*constant* BY2V*(arm)* 26b⟩

⟨*constant* PAGETABMAPMEM*(arm)* 96a⟩
⟨*constant* PAGETABSIZE*(arm)* 96b⟩
⟨*constant* PAGEDIRSIZE*(arm)* 95a⟩
⟨*constant* SMALLPAGEDIRSIZE*(arm)* 99b⟩
⟨*macro* PPN*(arm)* 162d⟩

```
/*
 * 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)* 40f⟩
⟨*constant* IOSIZE*(arm)* 31d⟩

E.24.3 memory/arm/mmu.c

⟨*enum* _anon_*(*(memory/arm/mmu.c)(arm)*)* 748a⟩≡ (748b)
```
enum {
```
⟨*constant* L1lo*(arm)* 162a⟩
⟨*constant* L1hi*(arm)* 162b⟩
```
};
```

⟨memory/arm/mmu.c 748b⟩≡
⟨*kernel basic includes* 35a⟩
```
#include "arm.h"
```

⟨*macro* L1X*(arm)* 160a⟩
⟨*macro* L2X*(arm)* 160b⟩
⟨*macro* L2AP*(arm)* 161i⟩

⟨*constant* L1ptedramattrs*(arm)* 497a⟩
⟨*constant* L2ptedramattrs*(arm)* 497b⟩

⟨*enum* _anon_*(*(memory/arm/mmu.c)(arm)*)* 748a⟩

⟨*function* `mmuinit`*(arm)* 249c⟩

⟨*function* `mmuinit1`*(arm)* 259c⟩

⟨*function* `mmul2empty`*(arm)* 163a⟩

⟨*function* `mmul1empty`*(arm)* 163e⟩

⟨*function* `arch_mmuswitch`*(arm)* 209b⟩

⟨*function* `arch_flushmmu`*(arm)* 164a⟩

⟨*function* `arch_mmurelease`*(arm)* 162f⟩

⟨*function* `arch_putmmu`*(arm)* 167⟩

⟨*function* `arch_cankaddr`*(arm)* 41c⟩

⟨*function* `mmukmap`*(arm)* 358c⟩

⟨*function* `arch_checkmmu`*(arm)* 169d⟩

⟨*function* `arch_kmap`*(arm)* 169e⟩

⟨*function* `arch_kunmap`*(arm)* 169f⟩

E.24.4 memory/arm/mmuinvalidate.s

⟨`memory/arm/mmuinvalidate.s` 749a⟩≡

```
#include "mem.h"
#include "arm.h"
#include "arminstr.ha"
```

⟨*function* `mmuinvalidate`*(arm)* 162g⟩

⟨*function* `mmuinvalidateaddr`*(arm)* 169a⟩

E.24.5 memory/arm/cache_raspi1.s

⟨`memory/arm/cache_raspi1.s` 749b⟩≡

```
#include "mem.h"
#include "arm.h"
#include "arminstr.ha"
```

⟨*constant CACHELINESZ(raspberry pi1)(ram)* 497g⟩

⟨*function cachexxx(raspberry pi1)(ram)* 499e⟩

E.24.6 memory/arm/cache_raspi2.s

⟨`memory/arm/cache_raspi2.s` 749c⟩≡

```
#include "mem.h"
#include "arm.h"
#include "arminstr.ha"
```

⟨*constant CACHELINESZ(raspberry pi2)(ram)* 515a⟩

⟨*function cachexxx(raspberry pi2)(ram)* 515d⟩

E.25 interrupts/arm/

E.25.1 interrupts/arm/trap.c

⟨*enum* _anon_*(*(interrupts/arm/trap.c)(arm)*)* 750a⟩≡ (750b)
```
enum {
    Debug = 0,

    ⟨constant Nvec(arm) 256d⟩
    ⟨constant Fiqenable(arm) 528e⟩

    ⟨constant Localtimerint(arm) 520d⟩
    ⟨constant Localmboxint(arm) 520f⟩
    ⟨constant Localintpending(arm) 520e⟩
};
```

⟨interrupts/arm/trap.c 750b⟩≡
```
/*
 * traps, exceptions, interrupts, system calls.
 */
⟨kernel basic includes 35a⟩
#include "io.h"
#include "ureg.h"
#include "arm.h"

⟨constant INTREGS(arm) 31g⟩
⟨constant LOCALREGS(arm) 520g⟩

typedef struct Intregs Intregs;
typedef struct Vctl Vctl;
typedef struct Vpage0 Vpage0;

⟨enum _anon_(*(interrupts/arm/trap.c)(arm)*) 750a⟩

⟨struct Vpage0(arm) 256c⟩

⟨struct Intregs(arm) 157d⟩

⟨struct Vctl(arm) 156c⟩

⟨global vctllock(arm) 157c⟩

⟨global vctl(arm) 157a⟩

⟨global vfiq(arm) 528d⟩

⟨global trapnames(arm) 143b⟩

⟨function arch__trapinit(arm) 256e⟩

⟨function intrcpushutdown(arm) 598a⟩

⟨function intrsoff(arm) 257a⟩
```

⟨*function* `intrshutdown`*(arm)* 597a⟩

⟨*function* `intrtime`*(arm)* 624c⟩

⟨*function* `irq`*(arm)* 159b⟩

⟨*function* `fiq`*(arm)* 529a⟩

⟨*function* `irqenable`*(arm)* 158a⟩

⟨*function* `trapname`*(arm)* 614c⟩

⟨*function* `ckfaultstuck`*(arm)* 615a⟩

⟨*function* `faultarm`*(arm)* 164b⟩

⟨*function* `writetomem`*(arm)* 154e⟩

⟨*function* `arch__trap`*(arm)* 150b⟩

⟨*function* `isvalidaddr`*(arm)* 614b⟩

⟨*function* `dumplongs`*(arm)* 614a⟩

⟨*function* `dumpstackwithureg`*(arm)* 616a⟩

⟨*function* `getpcsp`*(arm)* 615d⟩

⟨*function* `arch_callwithureg`*(arm)* 615c⟩

⟨*function* `trap_arch_dumpstack`*(arm)* 615b⟩

⟨*function* `dumpregs`*(arm)* 613b⟩

Uses `Intregs` 157d and `Vpage0` 256c.

E.25.2 interrupts/arm/fault_helpers.s

⟨`interrupts/arm/fault_helpers.s` 751a⟩≡

```
#include "mem.h"
#include "arm.h"
```

⟨*function* `fstget`*(arm)* 154c⟩

⟨*function* `ifsrget`*(arm)* 153d⟩

⟨*function* `farget`*(arm)* 154b⟩

E.25.3 interrupts/arm/lexception.s

⟨`interrupts/arm/lexception.s` 751b⟩≡

```
/*
 * arm exception handlers
 */
#include "mem.h"
#include "arm.h"
```

```
#include "arminstr.ha"
```

⟨*global* vectors*(arm)* 143c⟩

⟨*global* vtable*(arm)* 143d⟩

⟨*function* _vsvc*(arm)* 144c⟩

⟨*function* _vund*(arm)* 148a⟩

⟨*function* _vpabt*(arm)* 148b⟩

⟨*function* _vdabt*(arm)* 148c⟩

⟨*function* _virq*(arm)* 148d⟩

⟨*label* _vswitch*(arm)* 149a⟩

⟨*function* _vfiq*(arm)* 528a⟩

⟨*function* setr13*(arm)* 257d⟩

E.26 time/arm/

E.26.1 time/arm/dat_time.h

⟨time/arm/dat_time.h 752a⟩≡
```
/*
 * Time.
 *
 * HZ should divide 1000 evenly, ideally.
 * 100, 125, 200, 250 and 333 are okay.
 */
```
⟨*constant* Arch_HZ*(arm)* 214c⟩

```
// was vlong in x86
//typedef uvlong        Tval;
```

E.26.2 time/arm/clock.c

⟨*enum* _anon_*(*(time/arm/clock.c)(arm)*)* 752b⟩≡ (753)
```
enum {
```
⟨*constant* Localctl*(arm)* 514a⟩
⟨*constant* Prescaler*(arm)* 514b⟩

⟨*constant* SystimerFreq*(arm)* 223e⟩

⟨*constant* MaxPeriod*(arm)* 229b⟩
⟨*constant* MinPeriod*(arm)* 229c⟩
```
};
```

⟨*enum* _anon_*(*(time/arm/clock.c)2(arm)*)* 752c⟩≡ (753)
```
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 753⟩≡

```
/*
 * 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* 35a⟩

```
#include "io.h"
#include "ureg.h"
#include "arm.h"
```

⟨*constant* SYSTIMERS*(arm)* 31e⟩

⟨*constant* ARMTIMER*(arm)* 31f⟩

```
//TODO factorize in mem.h
```

⟨*constant* ARMLOCAL*(arm)* 503e⟩

⟨*enum* _anon_*(*(time/arm/clock.c)(arm)*)* 752b⟩

```
typedef struct Systimers Systimers;
typedef struct Armtimer Armtimer;
```

⟨*struct* Systimers*(arm)* 224c⟩

⟨*struct* Armtimer*(arm)* 625b⟩

⟨*enum* _anon_*(*(time/arm/clock.c)2(arm)*)* 752c⟩

⟨*function* clockintr*(arm)* 224d⟩

⟨*function* localclockintr*(arm)* 514e⟩

⟨*function* clockshutdown*(arm)* 597b⟩

⟨*function* clockinit*(arm)* 258⟩

⟨*function* arch_timerset*(arm)* 228d⟩

⟨*function* `clock_arch_fastticks`*(arm)* 224a⟩

⟨*function* `arch_perfticks`*(arm)* 625a⟩

⟨*function* `armtimerset`*(arm)* 822a⟩

⟨*function* `arch_us`*(arm)* 223g⟩

⟨*function* `clock_arch_microdelay`*(arm)* 231e⟩

⟨*function* `clock_arch_delay`*(arm)* 232a⟩

Uses `Armtimer` 625b and `Systimers` 224c.

E.26.3 time/arm/time_raspi1.s

⟨`time/arm/time_raspi1.s` 754a⟩≡

```
#include "mem.h"
#include "arm.h"
```

⟨*function* `arch_lcycles`*(raspberry pi1)(arm)* 476a⟩

E.26.4 time/arm/time_raspi2.s

⟨*instruction MRRC(arm)* 754b⟩≡ (721)

```
#define MRRC(coproc, op, rd, rn, crm) \
    WORD $(0xec500000|(rn)<<16|(rd)<<12|(coproc)<<8|(op)<<4|(crm))
```

⟨`time/arm/time_raspi2.s` 754c⟩≡

```
#include "mem.h"
#include "arm.h"
#include "arminstr.ha"
```

⟨*function* `arch_lcycles`*(raspberry pi2)(arm)* 514f⟩

E.27 processes/arm/

E.27.1 processes/arm/dat_processes.h

⟨`processes/arm/dat_processes.h` 754d⟩≡

⟨*constant* `AOUT_MAGIC`*(arm)* 122d⟩

⟨*enum* `_anon_`*(*`(arch/arm/dat_processes.h)(arm)`*)* 481i⟩

⟨*struct* `Arch_FPsave`*(arm)* 481g⟩

E.27.2 processes/arm/arch.c

⟨`processes/arm/arch.c` 754e⟩≡

⟨*kernel basic includes* 35a⟩

```
#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)* 147f⟩

⟨*function* `arch_dbgpc`*(arm)* 612c⟩

⟨*function* `arch_userureg`*(arm)* 147h⟩

⟨*function* `arch_setregisters`*(arm)* 450c⟩

⟨*function* `arch_setkernur`*(arm)* 574b⟩

⟨*function* `arch_validalign`*(arm)* 530c⟩

⟨*function* `arch__kexit`*(arm)* 151⟩

⟨*function* `linkproc`*(arm)* 140c⟩

⟨*function* `arch_kprocchild`*(arm)* 140b⟩

E.27.3 processes/arm/vfp3.c

⟨`processes/arm/vfp3.c` 755⟩≡

```
/*
 * VFPv2 or VFPv3 floating point unit
 */
```

⟨*kernel basic includes* 35a⟩

```
#include "ureg.h"
#include "arm.h"
```

⟨*enum* `_anon_(`(processes/arm/vfp3.c)(arm)`)` 485c⟩

⟨*enum* `_anon_(`(processes/arm/vfp3.c)2(arm)`)` 488a⟩
⟨*enum* `_anon_(`(processes/arm/vfp3.c)3(arm)`)` 488b⟩
⟨*enum* `_anon_(`(processes/arm/vfp3.c)4(arm)`)` 488c⟩

⟨*function* `subarch`*(arm)* 488d⟩

⟨*function* `implement`*(arm)* 489a⟩

⟨*function* `havefp`*(arm)* 489c⟩

⟨*function* `fpoff`*(arm)* 490a⟩

⟨*function* `fponly`*(arm)* 490b⟩

⟨*function* `fpcfg`*(arm)* 490c⟩

⟨*function* `fpinit`*(arm)* 491b⟩

⟨*function* `fpon`*(arm)* 491c⟩

⟨*function* `fpclear`*(arm)* 491d⟩

⟨*function* `fpunotify`*(arm)* 484c⟩

⟨*function* `fpunoted`*(arm)* 485a⟩

⟨*function* `fpusysrfork`*(arm)* 484a⟩

⟨*function* `fpusysrforkchild`*(arm)* 482b⟩

⟨*function* `fpsave`*(arm)* 484b⟩

⟨*function* `fprestore`*(arm)* 495a⟩

⟨*function* `fpuprocsave`*(arm)* 483a⟩

⟨*function* `fpuprocrestore`*(arm)* 483c⟩

⟨*function* `fpusysprocsetup`*(arm)* 482d⟩

⟨*function* `arch_procsetup`*(arm)* 125d⟩

⟨*function* `arch_procsave`*(arm)* 209c⟩

⟨*function* `arch_procrestore`*(arm)* 210a⟩

⟨*function* `mathnote`*(arm)* 494b⟩

⟨*function* `mathemu`*(arm)* 494c⟩

⟨*function* `fpstuck`*(arm)* 491e⟩

⟨*enum* `_anon_`*(*`(processes/arm/vfp3.c)5(arm)`*)* 492b⟩

⟨*function* `condok`*(arm)* 492c⟩

```
/* instruction decoding */ // was in arm.h
```
⟨*macro* `ISCPOP`*(arm)* 493a⟩
⟨*macro* `ISVFPOP`*(arm)* 493b⟩

⟨*function* `fpuemu`*(arm)* 493c⟩

E.27.4 processes/arm/lproc.s

⟨`processes/arm/lproc.s` 756⟩≡
```
#include "mem.h"
#include "arm.h"
```

⟨*function* `arch_touser`*(arm)* 268b⟩

⟨*function* `arch__forkret`*(arm)* 118b⟩

E.27.5 processes/arm/switch.s

⟨processes/arm/switch.s 757a⟩≡
  ⟨*function* arch_setlabel*(arm)* 206b⟩

  ⟨*function* arch_gotolabel*(arm)* 206a⟩

  ⟨*function* getcallerpc*(arm)* 27⟩

E.28 syscalls/arm/

E.28.1 syscalls/arm/syscall.c

⟨syscalls/arm/syscall.c 757b⟩≡
```
  ⟨kernel basic includes 35a⟩
  #include "../port/systab.h"
  #include <tos.h>
  #include "ureg.h"
  #include "arm.h"

  typedef struct NFrame NFrame;
  ⟨struct NFrame(arm) 421a⟩

  ⟨function arch__noted(arm) 423b⟩

  ⟨function arch__notify(arm) 421b⟩

  ⟨function arch__syscall(arm) 145e⟩

  ⟨function arch_execregs(arm) 130a⟩

  ⟨function arch_forkchild(arm) 117e⟩
```
Uses NFrame 421a.

E.29 Extra files

E.29.1 init/arm/bootconf.c

⟨*enum* _anon_*(*(init/arm/bootconf.c)(arm)*)* 757c⟩≡ (761b)
```
  enum {
      /* space for syscall args, return PC, top-of-stack struct */
      Ustkheadroom    = sizeof(Sargs) + sizeof(uintptr) + sizeof(Tos),
  };
```

⟨*constant* CONFADDR*(arm)* 757d⟩≡ (747)
```
  #define CONFADDR    (KZERO+0x100)       /* unparsed plan9.ini */
```

⟨*constant* BOOTARGS*(arm)* 757e⟩≡ (761b)
```
  /*
   * Where configuration info is left for the loaded programme.
   */
  #define BOOTARGS    ((char*)CONFADDR)
```

⟨*constant* BOOTARGSLEN*(arm)* 757f⟩≡ (761b)
```
  #define BOOTARGSLEN (CPUADDR-CONFADDR)
```

⟨*global* `oargc` *(arm)* 758a⟩≡ (761b)
```
/*
 * Option arguments from the command line.
 * oargv[0] is the boot file.
 */
static int oargc;
```

⟨*global* `oargv` *(arm)* 758b⟩≡ (761b)
```
static char* oargv[20];
```

⟨*global* `oargb` *(arm)* 758c⟩≡ (761b)
```
static char oargb[128];
```

⟨*global* `oargblen` *(arm)* 758d⟩≡ (761b)
```
static int oargblen;
```

⟨*struct* `Atag` *(arm)* 758e⟩≡ (761b)
```
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` 758e and `__anon_struct_337` 758e.

⟨*enum* `_anon_` *(*`(init/arm/bootconf.c)2(arm)`*)* 758f⟩≡ (761b)
```
enum {
    AtagNone    = 0x00000000,
    AtagCore    = 0x54410001,
    AtagMem     = 0x54410002,
    AtagCmdline = 0x54410009,
};
```

⟨*function* `findconf` *(arm)* 758g⟩≡ (761b)
```
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` 47a and `nconf` 47c.

⟨*function* addconf *(arm)* 759a⟩≡ (761b)

```
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 47b, confname 47a, confval 47a, findconf() 758g, and nconf 47c.

⟨*function* writeconf *(arm)* 759b⟩≡ (761b)

```
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 757e, BOOTARGSLEN-2437 757f, free() 85b, getconfenv() 673c, nexterror, poperror 633d, and waserror 633d.

⟨*function* plan9iniinit *(arm)* 759c⟩≡ (761b)

```
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 47b and addconf() 759a.

⟨*function* ataginit*(arm)* 760a⟩≡ (761b)

```
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 758f, AtagCore-2443 758f, AtagMem-2444 758f, AtagNone-2442 758f, BOOTARGS-2436 757e, BOOTARGSLEN-2437 757f, conf 46b, memsize 254c, offsetof 692d, and plan9iniinit() 759c.

⟨*function* optionsinit*(arm)* 760b⟩≡ (761b)

```
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 692c, oargb-2440 758c, oargblen-2441 758d, oargc-2438 758a, and oargv-2439 758b.

⟨*function* bootargs*(arm)* 761a⟩≡ (761b)
```
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 41a, STACKALIGN 265c, UINT2PTR 41b, Ustkheadroom-2435 757c, oargb-2440 758c, oargblen-2441 758d, oargc-2438 758a, and oargv-2439 758b.

⟨init/arm/bootconf.c 761b⟩≡
```
⟨kernel basic includes 35a⟩

#include "tos.h"

⟨enum _anon_(init/arm/bootconf.c)(arm) 757c⟩

⟨constant BOOTARGS(arm) 757e⟩
⟨constant BOOTARGSLEN(arm) 757f⟩
//#define MAXCONF       64
//#define MAXCONFLINE   160

⟨global oargc(arm) 758a⟩
⟨global oargv(arm) 758b⟩
⟨global oargb(arm) 758c⟩
⟨global oargblen(arm) 758d⟩

/* 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)* 758e⟩

⟨*enum* _anon_*(*(init/arm/bootconf.c)2(arm)*)* 758f⟩

⟨*function* findconf*(arm)* 758g⟩

⟨*function* addconf*(arm)* 759a⟩

⟨*function* writeconf*(arm)* 759b⟩

⟨*function* plan9iniinit*(arm)* 759c⟩

⟨*function* ataginit*(arm)* 760a⟩

⟨*function* optionsinit*(arm)* 760b⟩

⟨*function* bootargs*(arm)* 761a⟩

Uses Atag 758e.

E.29.2 devices/storage/arm/emmc.c

⟨*enum* _anon_*(*(devices/storage/arm/emmc.c)(arm)*)* 762a⟩≡ (770a)

```
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 709a.

⟨*enum* _anon_*(*(devices/storage/arm/emmc.c)2(arm)*)* 762b⟩≡ (770a)

```
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)* 764a⟩≡ (770a)

```
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 762b, Card2host-1678 762b, Crcchken-1670 762b, Host2card-1677 762b, Isdata-1668 762b, Ixchken-1669 762b, Multiblock-1676 762b, Resp136-1673 762b, Resp48-1674 762b, and Resp48busy-1675 762b.

⟨*struct* Ctlr*(*(devices/storage/arm/emmc.c)(arm)*)* 764b⟩≡ (770a)

```
struct Ctlr {
    Rendez  r;
    int datadone;
    int fastclock;
    ulong   extclk;
};
```

⟨*global* emmc*(arm)* 764c⟩≡ (770a)

```
static Ctlr emmc;
```

⟨*function* WR*(arm)* 764d⟩≡ (770a)

```
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 32f, arch_microdelay, and emmc-1704 764c.

⟨*function* clkdiv*(arm)* 765a⟩≡ (770a)
```
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 762b, Clkfreq8shift-1655 762b, Clkfreqms2mask-1658 762b, Clkfreqms2shift-1657 762b, and assert 693a.

⟨*function* datadone*(arm)* 765b⟩≡ (770a)
```
static int
datadone(void*)
{
    int i;

    u32int *r = (u32int*)EMMCREGS;
    i = r[Interrupt];
    return i & (Datadone|Err);
}
```
Uses Datadone-1695 762b, EMMCREGS-1615 32f, Err-1690 762b, and Interrupt-1634 762b.

⟨*function* emmcinit*(arm)* 765c⟩≡ (770a)
```
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 727b, Control1-1633 762b, EMMCREGS-1615 32f, Extfreq-1616 762a, Mhz 709a, Srsthc-1652 762b, WR() 764d, arch_delay, emmc-1704 764c, getclkrate() 728a, and print.

⟨*function* emmcinquiry*(arm)* 765d⟩≡ (770a)
```
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 32f and Slotisrver-1647 762b.

⟨*function* mmcinterrupt*(arm)* 766a⟩≡ (770a)

```
static void
mmcinterrupt(Ureg*, void*)
{
    u32int *r;
    r = (u32int*)EMMCREGS;
    if(r[Interrupt]&(Datadone|Err)){
        WR(Irpten, 0);
        wakeup(&emmc.r);
    }
}
```

Uses Datadone-1695 762b, EMMCREGS-1615 32f, Err-1690 762b, Interrupt-1634 762b, Irpten-1636 762b, WR() 764d, emmc-1704 764c, and wakeup.

⟨*function* emmcenable*(arm)* 766b⟩≡ (770a)

```
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 762b, Clken-1661 762b, Clkgendiv-1659 762b, Clkintlen-1663 762b, Clkstable-1662 762b, Control1-1633 762b, DTO-1619 762a, Datatoshift-1653 762b, Dtoerr-1685 762b, EMMCREGS-1615 32f, Initfreq-1617 762a, Irptmask-1635 762b, WR() 764d, arch_delay, clkdiv() 765a, emmc-1704 764c, and print.

⟨*function* emmccmd*(arm)* 766c⟩≡ (770a)

```
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 762b, Cardinsert-1692 762b, Clken-1661 762b, Clkgendiv-1659 762b, Clkintlen-1663 762b, Clkstable-1662 762b, Cmddone-1696 762b, Cmdinhibit-1703 762b, Cmdtm-1625 762b, Control0-1632 762b, Control1-1633 762b, Ctoerr-1689 762b, DTO-1619 762a, Datadone-1695 762b, Datatoshift-1653 762b, Datinhibit-1702 762b, Dwidth4-1648 762b, EMMCREGS-1615 32f, Err-1690 762b, Indexshift-1664 762b, Interrupt-1634 762b, Irpten-1636 762b, Isdata-1668 762b, MMCSelect-1620 762a, Readrdy-1693 762b, Resp0-1626 762b, Resp1-1627 762b, Resp136-1673 762b, Resp2-1628 762b, Resp3-1629 762b, Resp48-1674 762b, Resp48busy-1675 762b, Respmask-1671 762b, Respnone-1672 762b, SDfreq-1618 762a, Setbuswidth-1621 762a, Srstcmd-1651 762b, Srstdata-1650 762b, Status-1631 762b, WR() 764d, Writerdy-1694 762b, arch_delay, assert 693a, clkdiv() 765a, cmdinfo 764a, datadone() 765b, emmc-1704 764c, nelem 692c, print, and tsleep.

⟨*function* emmciosetup*(arm)* 768⟩≡ (770a)

```
void
emmciosetup(int write, void *buf, int bsize, int bcount)
{
    USED(write);
    USED(buf);
    WR(Blksizecnt, bcount<<16 | bsize);
}
```

Uses Blksizecnt-1623 762b and WR() 764d.

⟨*function* emmcio*(arm)* 769a⟩≡ (770a)

```
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 762b, Datadone-1695 762b, DmaChanEmmc 830a, DmaD2M 830a, DmaDevEmmc 830a, DmaM2D 830a, EMMCREGS-1615 32f, Err-1690 762b, Interrupt-1634 762b, Irpten-1636 762b, Status-1631 762b, WR() 764d, assert 693a, datadone() 765b, emmc-1704 764c, nexterror, okay() 843a, poperror 633d, print, tsleep, and waserror 633d.

⟨*global* sdio*(arm)* 769b⟩≡ (770a)

```
SDio sdio = {
    "emmc",
    emmcinit,
    emmcenable,
    emmcinquiry,
    emmccmd,
    emmciosetup,
    emmcio,
};
```

Uses emmccmd() 766c, emmcenable() 766b, emmcinit() 765c, emmcinquiry() 765d, emmcio() 769a, and emmciosetup() 768.

⟨devices/storage/arm/emmc.c 770a⟩≡
```
/*
 * bcm2835 external mass media controller (mmc / sd host interface)
 *
 * Copyright © 2012 Richard Miller <r.miller@acm.org>
 */
⟨kernel basic includes 35a⟩
#include "io.h"
#include "../port/sd.h"

⟨constant EMMCREGS(arm) 32f⟩

⟨enum _anon_((devices/storage/arm/emmc.c)(arm)) 762a⟩

⟨enum _anon_((devices/storage/arm/emmc.c)2(arm)) 762b⟩

⟨global cmdinfo(arm) 764a⟩

typedef struct Ctlr Ctlr;
⟨struct Ctlr((devices/storage/arm/emmc.c)(arm)) 764b⟩

⟨global emmc(arm) 764c⟩

⟨function WR(arm) 764d⟩

⟨function clkdiv(arm) 765a⟩

⟨function datadone(arm) 765b⟩

⟨function emmcinit(arm) 765c⟩

⟨function emmcinquiry(arm) 765d⟩

⟨function mmcinterrupt(arm) 766a⟩

⟨function emmcenable(arm) 766b⟩

⟨function emmccmd(arm) 766c⟩

⟨function emmciosetup(arm) 768⟩

⟨function emmcio(arm) 769a⟩


⟨global sdio(arm) 769b⟩
```

E.29.3 init/arm/watchdog.c

⟨enum _anon_((init/arm/watchdog.c)(arm)) 770b⟩≡ (772b)
```
enum {
    Wdogfreq    = 65536,
    Wdogtime    = 5,    /* seconds, <= 15 */
};
```

⟨enum _anon_((init/arm/watchdog.c)2(arm)) 770c⟩≡ (772b)
```
/*
 * 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)* 771a⟩≡ (772b)

```
void
archreset(void)
{
    fpon();
}
```

Uses `fpon()` 491c.

⟨*function* `archreboot` *(arm)* 771b⟩≡ (772b)

```
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` 770c, `CfgReset-2455` 770c, `POWERREGS-2449` 31i, `Password-2453` 770c, `Rstc-2452` 770c, `Wdog-2457` 770c, and `arch_coherence`.

⟨*function* `wdogfeed` *(arm)* 771c⟩≡ (772b)

```
static void
wdogfeed(void)
{
    u32int *r;

    r = (u32int*)POWERREGS;
    r[Wdog] = Password | (Wdogtime * Wdogfreq);
    r[Rstc] = Password | (r[Rstc] & ~CfgMask) | CfgReset;
}
```

Uses `CfgMask-2454` 770c, `CfgReset-2455` 770c, `POWERREGS-2449` 31i, `Password-2453` 770c, `Rstc-2452` 770c, `Wdog-2457` 770c, `Wdogfreq-2450` 770b, and `Wdogtime-2451` 770b.

⟨*function* `wdogoff` *(arm)* 771d⟩≡ (772b)

```
void
wdogoff(void)
{
    u32int *r;

    r = (u32int*)POWERREGS;
    r[Rstc] = Password | (r[Rstc] & ~CfgMask);
}
```

Uses `CfgMask-2454` 770c, `POWERREGS-2449` 31i, `Password-2453` 770c, and `Rstc-2452` 770c.

⟨*function* `watchdoglink`*(arm)* 772a⟩≡ (772b)

```
void
watchdoglink(void)
{
    addclock0link(wdogfeed, Arch_HZ);
}
```

Uses `addclock0link()` 231a and `wdogfeed()` 771c.

⟨`init/arm/watchdog.c` 772b⟩≡

⟨*kernel basic includes* 35a⟩

⟨*constant* `POWERREGS`*(arm)* 31i⟩

⟨*enum* `_anon_(`(`init/arm/watchdog.c`)(`arm`)`)` 770b⟩

⟨*enum* `_anon_(`(`init/arm/watchdog.c`)2(`arm`)`)` 770c⟩

⟨*function* `archreset`*(arm)* 771a⟩

⟨*function* `archreboot`*(arm)* 771b⟩

⟨*function* `wdogfeed`*(arm)* 771c⟩

⟨*function* `wdogoff`*(arm)* 771d⟩

⟨*function* `watchdoglink`*(arm)* 772a⟩

E.29.4 buses/arm/dwcotg.h

⟨*enum* `_anon_(`(`buses/arm/dwcotg.h`)(`arm`)`)` 772c⟩≡ (773)

```
enum {
    Maxchans    = 16,   /* actual number of channels in ghwcfg2 */
};
```

⟨*struct* `Dwcregs`*(arm)* 772d⟩≡ (773)

```
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 772d and Maxchans 772c.

⟨buses/arm/dwcotg.h 773⟩≡

```
/*
 * 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) 772c⟩

⟨struct Dwcregs(arm) 772d⟩

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 772d and Hostchan 772d.

E.29.5 buses/arm/devusb.c

⟨*enum* _anon_*(*(buses/arm/devusb.c)(arm)*)* 781⟩≡ (807b)

```
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)* 782a⟩≡ (807b)
```
struct Hcitype
{
    char*   type;
    int (*reset)(Hci*);
};
```

⟨*macro* `QID` *(arm)* 782b⟩≡ (807b)
```
#define QID(q)  ((int)(q).path)
```

⟨*global* `usbctls` *(arm)* 782c⟩≡ (807b)
```
static Cmdtab usbctls[] =
{
    {CMdebug,   "debug",    2},
    {CMdump,    "dump",     1},
};
```
Uses `CMdebug-2250` 781 and `CMdump-2251` 781.

⟨*global* `epctls` *(arm)* 782d⟩≡ (807b)
```
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` 781, `CMclrhalt-2258` 781, `CMdebugep-2265` 781, `CMdetach-2263` 781, `CMhub-2254` 781, `CMhz-2260` 781, `CMinfo-2262` 781, `CMmaxpkt-2256` 781, `CMname-2266` 781, `CMnew-2252` 781, `CMnewdev-2253` 781, `CMntds-2257` 781, `CMpollival-2259` 781, `CMpreset-2268` 781, `CMsamplesz-2261` 781, `CMspeed-2255` 781, and `CMtmout-2267` 781.

⟨*global* `usbdir` *(arm)* 782e⟩≡ (807b)
```
static Dirtab usbdir[] =
{
    "ctl",      {Qctl},     0,  0666,
};
```
Uses `Qctl-2242` 781.

⟨*global* `usbmodename` *(arm)* 782f⟩≡ (807b)
```
char *usbmodename[] =
{
    [OREAD] "r",
    [OWRITE]    "w",
    [ORDWR] "rw",
};
```

⟨*global* ttname*(arm)* 783a⟩≡ (807b)
```
static char *ttname[] =
{
    [Tnone] "none",
    [Tctl]  "control",
    [Tiso]  "iso",
    [Tintr] "interrupt",
    [Tbulk] "bulk",
};
```

⟨*global* spname*(arm)* 783b⟩≡ (807b)
```
static char *spname[] =
{
    [Fullspeed] "full",
    [Lowspeed]  "low",
    [Highspeed] "high",
    [Nospeed]   "no",
};
```

⟨*global* debug*(arm)* 783c⟩≡ (807b)
```
static int  debug;
```

⟨*global* hcitypes*(arm)* 783d⟩≡ (807b)
```
static Hcitype  hcitypes[Nhcis];
```

⟨*global* hcis*(arm)* 783e⟩≡ (807b)
```
static Hci* hcis[Nhcis];
```

⟨*global* epslck*(arm)* 783f⟩≡ (807b)
```
static QLock    epslck;     /* add, del, lookup endpoints */
```

⟨*global* eps*(arm)* 783g⟩≡ (807b)
```
static Ep*  eps[Neps];  /* all endpoints known */
```

⟨*global* epmax*(arm)* 783h⟩≡ (807b)
```
static int  epmax;      /* 1 + last endpoint index used  */
```

⟨*global* usbidgen*(arm)* 783i⟩≡ (807b)
```
static int  usbidgen;   /* device address generator */
```

⟨*function* seprintdata*(arm)* 783j⟩≡ (807b)
```
/*
 * 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)* 784a⟩≡ (807b)
```
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 692c and spname-2276 783b.

⟨*function* name2ttype*(arm)* 784b⟩≡ (807b)
```
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 692c and ttname-2275 783a.

⟨*function* name2mode*(arm)* 784c⟩≡ (807b)
```
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 692c and usbmodename 782f.

⟨*function* qid2epidx*(arm)* 784d⟩≡ (807b)
```
static int
qid2epidx(int q)
{
    q = (q-Qep0dir)/4;
    if(q < 0 || q >= epmax || eps[q] == nil)
        return -1;
    return q;
}
```
Uses Qep0dir-2243 781, epmax-2282 783h, and eps-2281 783g.

⟨*function* isqtype*(arm)* 785a⟩≡ (807b)

```
static int
isqtype(int q, int type)
{
    if(q < Qep0dir)
        return 0;
    q -= Qep0dir;
    return (q & 3) == type;
}
```

Uses Qep0dir-2243 781.

⟨*function* addhcitype*(arm)* 785b⟩≡ (807b)

```
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 783d.

⟨*function* seprintep*(arm)* 785c⟩≡ (807b)

```
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 279c, OREAD 279c, OWRITE 279c, assert 693a, nelem 692c, nexterror, poperror 633d, qlock() 63f, qunlock() 64c, spname-2276 783b, ttname-2275 783a, usbmodename 782f, and waserror 633d.

⟨*function* epalloc*(arm)* 786a⟩≡ (807b)

```
    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 783h, eps-2281 783g, epslck-2280 783f, free() 85b, print, qlock() 63f, qunlock() 64c, and smalloc() 85c.

⟨*function* getep*(arm)* 786b⟩≡ (807b)

```
    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 783h, eps-2281 783g, epslck-2280 783f, incref() 69b, qlock() 63f, and qunlock() 64c.

⟨*function* putep*(arm)* 787a⟩≡ (807b)

```
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() 69c, deprint, epmax-2282 783h, eps-2281 783g, epslck-2280 783f, free() 85b, putep() 787a, qlock() 63f, qunlock() 64c, and usbidgen-2283 783i.

⟨*function* dumpeps*(arm)* 787b⟩≡ (807b)

```
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 783h, getep() 786b, hcis-2279 783e, nexterror, poperror 633d, print, putep() 787a, and waserror 633d.

⟨*function* newusbid*(arm)* 788a⟩≡ (807b)

```
    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 783f, print, qlock() 63f, qunlock() 64c, and usbidgen-2283 783i.

⟨*function* newdev*(arm)* 788b⟩≡ (807b)

```
    /*
     * 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 279c, dprint, epalloc() 786a, newusbid() 788a, and smalloc() 85c.

⟨*function* newdevep*(arm)* 789a⟩≡ (807b)

```
/*
 * 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() 786a, and incref() 69b.

⟨*function* epdataperm*(arm)* 789b⟩≡ (807b)

```
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 618d, OREAD 279c, and OWRITE 279c.

⟨*function* usbgen*(arm)* 790⟩≡ (807b)

```
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, QID-2271 782b, QTDIR 52c, QTFILE 52c, Qctl-2242 781, Qdir-2240 781, Qep0ctl-2245 781, Qep0dir-2243 781, Qep0io-2244 781, Qepctl-2249 781, Qepdir-2247 781, Qusbdir-2241 781, ddprint, devdir() 398c, epdataperm() 789b, epmax-2282 783h, eve 557a, getep() 786b, isqtype() 785a, nelem 692c, nexterror, poperror 633d, putep() 787a, qid2epidx() 784d, up, usbdir-2274 782e, and waserror 633d.

⟨*function* hciprobe*(arm)* 792⟩≡ (807b)

```
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() 85b, hcitypes-2278 783d, print, and smalloc() 85c.

⟨*function* usbreset*(arm)* 793a⟩≡ (807b)

```
    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() 792, hcis-2279 783e, hcitypes-2278 783d, and print.

⟨*function* usbinit*(arm)* 793b⟩≡ (807b)

```
    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 783e, kstrdup() 636b, and newdev() 788b.

⟨*function* `usbattach`*(arm)* 794a⟩≡ (807b)

```
static Chan*
usbattach(char *spec)
{
    return devattach(L'u', spec);
}
```

Uses `devattach()` 275a.

⟨*function* `usbwalk`*(arm)* 794b⟩≡ (807b)

```
static Walkqid*
usbwalk(Chan *c, Chan *nc, char **name, int nname)
{
    return devwalk(c, nc, name, nname, nil, 0, usbgen);
}
```

Uses `devwalk()` 400b and `usbgen()` 790.

⟨*function* `usbstat`*(arm)* 794c⟩≡ (807b)

```
static int
usbstat(Chan *c, uchar *db, int n)
{
    return devstat(c, db, n, nil, 0, usbgen);
}
```

Uses `devstat()` 402 and `usbgen()` 790.

⟨*function* `usbload`*(arm)* 794d⟩≡ (807b)

```
/*
 * 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)* 795⟩≡ (807b)

```
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 272b, Enotconf 631d, OREAD 279c, OWRITE 279c, QID-2271 782b, Qep0dir-2243 781, Qepctl-2249 781, Qepdir-2247 781, deprint, getep() 786b, isqtype() 785a, nexterror, openmode() 280a, poperror 633d, putep() 787a, qid2epidx() 784d, qlock() 63f, qunlock() 64c, usbgen() 790, usbload() 794d, and waserror 633d.

⟨*function* epclose*(arm)* 796a⟩≡ (807b)
```
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 633d, qlock() 63f, qunlock() 64c, and waserror 633d.

⟨*function* usbclose*(arm)* 796b⟩≡ (807b)
```
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 272b, QID-2271 782b, Qep0dir-2243 781, Qepctl-2249 781, Qepdir-2247 781, deprint, free() 85b, getep() 786b, isqtype() 785a, nexterror, poperror 633d, putep() 787a, qid2epidx() 784d, and waserror 633d.

⟨*function* ctlread*(arm)* 796c⟩≡ (807b)
```
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 782b, Qctl-2242 781, READSTR 704a, epmax-2282 783h, free() 85b, getep() 786b, nexterror, poperror 633d, putep() 787a, qid2epidx() 784d, readstr() 637d, smalloc() 85c, and waserror 633d.

⟨*function* rhubread*(arm)* 797⟩≡ (807b)

```
/*
 * 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)* 798a⟩≡ (807b)

```
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 781, Rportreset-2270 781, and Rsetfeature.

⟨*function* usbread*(arm)* 798b⟩≡ (807b)

```
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 631d, Edetach 631d, OWRITE 279c, QID-2271 782b, QTDIR 52c, Qctl-2242 781, Qepctl-2249 781, ctlread() 796c, devdirread() 403c, getep() 786b, isqtype() 785a, nexterror, poperror 633d, putep() 787a, qid2epidx() 784d, rhubread() 797, usbgen() 790, and waserror 633d.

⟨*function* pow2*(arm)* 799a⟩≡ (807b)

```
static long
pow2(int n)
{
    return 1 << n;
}
```

⟨*function* setmaxpkt*(arm)* 799b⟩≡ (807b)

```
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 783b.

⟨*function* epctl*(arm)* 800⟩≡ (807b)

```
/*
 * 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 781, CMclrhalt-2258 781, CMdebugep-2265 781, CMdetach-2263 781, CMhub-2254 781, CMhz-2260 781, CMinfo-2262 781, CMmaxpkt-2256 781, CMname-2266 781, CMnew-2252 781, CMnewdev-2253 781, CMntds-2257 781, CMpollival-2259 781, CMpreset-2268 781, CMsamplesz-2261 781, CMspeed-2255 781, CMtmout-2267 781, deprint, epctls-2273 782d, free() 85b, kstrdup() 636b, lookupcmd() 561b, name2mode() 784c, name2speed() 784a, name2ttype() 784b, nelem 692c, newdev() 788b, newdevep() 789a, nexterror, parsecmd() 560c, poperror 633d, pow2() 799a, print, putep() 787a, qlock() 63f, qunlock() 64c, setmaxpkt() 799b, smalloc() 85c, up, validname() 285e, and waserror 633d.

⟨*function* usbctl*(arm)* 803⟩≡ (807b)

```
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 781, CMdump-2251 781, debug-2277 783c, dprint, dumpeps() 787b, epmax-2282 783h, free() 85b, getep() 786b, lookupcmd() 561b, nelem 692c, nexterror, parsecmd() 560c, poperror 633d, print, putep() 787a, usbctls-2272 782c, and waserror 633d.

⟨*function* ctlwrite*(arm)* 804⟩≡ (807b)

```
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 631d, QID-2271 782b, Qctl-2242 781, Qepctl-2249 781, epctl() 800, free() 85b, getep() 786b, isqtype() 785a, nexterror, poperror 633d, putep() 787a, qid2epidx() 784d, usbctl() 803, and waserror 633d.

⟨*function* usbwrite*(arm)* 805⟩≡ (807b)

```
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 631d, Edetach 631d, OREAD 279c, QID-2271 782b, QTDIR 52c, Qctl-2242 781, Qepctl-2249 781, ctlwrite() 804, getep() 786b, isqtype() 785a, nexterror, poperror 633d, putep() 787a, qid2epidx() 784d, rhubwrite() 798a, and waserror 633d.

⟨*function* usbbread*(arm)* 806a⟩≡ (807b)

```
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() 407b, freeb() 408b, nexterror, poperror 633d, usbread() 798b, and waserror 633d.

⟨*function* usbbwrite*(arm)* 806b⟩≡ (807b)

```
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 406b, freeb() 408b, nexterror, poperror 633d, usbwrite() 805, and waserror 633d.

⟨*function* usbshutdown*(arm)* 806c⟩≡ (807b)

```
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 783e and print.

⟨*global* usbdevtab*(arm)* 807a⟩≡ (807b)

```
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() 686, devremove() 686, devwstat() 686, usbattach() 794a, usbbread() 806a, usbbwrite() 806b, usbclose() 796b, usbinit() 793b, usbopen() 795, usbread() 798b, usbreset() 793a, usbshutdown() 806c, usbstat() 794c, usbwalk() 794b, and usbwrite() 805.

⟨buses/arm/devusb.c 807b⟩≡

```
/*
 * 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* 35a⟩

```
#include    "io.h"
#include    "../port/usb.h"

typedef struct Hcitype Hcitype;
```

⟨*enum* `_anon_(`(buses/arm/devusb.c)(arm)*)* 781⟩

⟨*struct* `Hcitype`*(arm)* 782a⟩

⟨*macro* `QID`*(arm)* 782b⟩

⟨*global* `usbctls`*(arm)* 782c⟩

⟨*global* `epctls`*(arm)* 782d⟩

⟨*global* `usbdir`*(arm)* 782e⟩

⟨*global* `usbmodename`*(arm)* 782f⟩

⟨*global* `ttname`*(arm)* 783a⟩

⟨*global* `spname`*(arm)* 783b⟩

⟨*global* `debug`*(arm)* 783c⟩
⟨*global* `hcitypes`*(arm)* 783d⟩
⟨*global* `hcis`*(arm)* 783e⟩
⟨*global* `epslck`*(arm)* 783f⟩
⟨*global* `eps`*(arm)* 783g⟩
⟨*global* `epmax`*(arm)* 783h⟩
⟨*global* `usbidgen`*(arm)* 783i⟩

⟨*function* `seprintdata`*(arm)* 783j⟩

⟨*function* `name2speed`*(arm)* 784a⟩

⟨*function* `name2ttype`*(arm)* 784b⟩

⟨*function* `name2mode`*(arm)* 784c⟩

⟨*function* `qid2epidx`*(arm)* 784d⟩

⟨*function* `isqtype`*(arm)* 785a⟩

⟨*function* `addhcitype`*(arm)* 785b⟩

⟨*function* `seprintep`*(arm)* 785c⟩

⟨*function* `epalloc`*(arm)* 786a⟩

⟨*function* `getep`*(arm)* 786b⟩

⟨*function* `putep`*(arm)* 787a⟩

⟨*function* `dumpeps`*(arm)* 787b⟩

⟨*function* `newusbid`*(arm)* 788a⟩

⟨*function* `newdev`*(arm)* 788b⟩

⟨*function* `newdevep`*(arm)* 789a⟩

⟨*function* `epdataperm`*(arm)* 789b⟩

⟨*function* `usbgen`*(arm)* 790⟩

⟨*function* `hciprobe`*(arm)* 792⟩

⟨*function* `usbreset`*(arm)* 793a⟩

⟨*function* `usbinit`*(arm)* 793b⟩

⟨*function* `usbattach`*(arm)* 794a⟩

⟨*function* `usbwalk`*(arm)* 794b⟩

⟨*function* `usbstat`*(arm)* 794c⟩

⟨*function* `usbload`*(arm)* 794d⟩

⟨*function* `usbopen`*(arm)* 795⟩

⟨*function* `epclose`*(arm)* 796a⟩

⟨*function* `usbclose`*(arm)* 796b⟩

⟨*function* `ctlread`*(arm)* 796c⟩

⟨*function* `rhubread`*(arm)* 797⟩

⟨*function* `rhubwrite`*(arm)* 798a⟩

⟨*function* `usbread`*(arm)* 798b⟩

⟨*function* `pow2`*(arm)* 799a⟩

⟨*function* `setmaxpkt`*(arm)* 799b⟩

⟨*function* `epctl`*(arm)* 800⟩

⟨*function* `usbctl`*(arm)* 803⟩

⟨*function* `ctlwrite`*(arm)* 804⟩

⟨*function* `usbwrite`*(arm)* 805⟩

⟨*function* `usbbread`*(arm)* 806a⟩

⟨*function* `usbbwrite`*(arm)* 806b⟩

⟨*function* `usbshutdown`*(arm)* 806c⟩

⟨*global* `usbdevtab`*(arm)* 807a⟩

Uses Hcitype 782a.

E.29.6 buses/arm/usbdwc.c

⟨*enum* _anon_*(*(buses/arm/usbdwc.c)(arm)*)* 810a⟩≡ (828)
```
enum
{
    Enabledelay = 50,
    Resetdelay  = 10,
    ResetdelayHS    = 50,

    Read        = 0,
    Write       = 1,
};
```

⟨*struct* Ctlr*(*(buses/arm/usbdwc.c)(arm)*)* 810b⟩≡ (828)
```
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)* 810c⟩≡ (828)
```
struct Epio {
    QLock;
    Block   *cb;
    ulong   lastpoll;
};
```

⟨*global* dwc*(arm)* 810d⟩≡ (828)
```
static Ctlr dwc;
```

⟨*global* debug*(*(buses/arm/usbdwc.c)(arm)*)* 810e⟩≡ (828)
```
static int debug;
```

⟨*global* Ebadlen*(arm)* 810f⟩≡ (828)
```
static char Ebadlen[] = "bad usb request length";
```
Uses Ebadlen-2237 810f.

⟨*function* filock*(arm)* 810g⟩≡ (828)
```
static void
filock(Lock *l)
{
    int x;

    x = splfhi();
    ilock(l);
    l->sr = x;
}
```
Uses ilock() 61d.

⟨*function* `fiunlock`*(arm)* 811a⟩≡ (828)

```
static void
fiunlock(Lock *l)
{
    iunlock(l);
}
```

Uses `iunlock()` 62b.

⟨*function* `chanalloc`*(arm)* 811b⟩≡ (828)

```
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()` 63f and `qunlock()` 64c.

⟨*function* `chanrelease`*(arm)* 811c⟩≡ (828)

```
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()` 63f and `qunlock()` 64c.

⟨*function* `chansetup`*(arm)* 811d⟩≡ (828)

```
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)* 812a⟩≡ (828)

```
    static int
    sofdone(void *a)
    {
        Dwcregs *r;

        r = a;
        return (r->gintmsk & Sofintr) == 0;
    }
```

⟨*function* `sofwait` *(arm)* 812b⟩≡ (828)

```
    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() 810g, fiunlock() 811a, sleep, and sofdone() 812a.

⟨*function* chandone*(arm)* 813a⟩≡ (828)

```
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)* 813b⟩≡ (828)

```
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() 813a, dprint, filock() 810g, fiunlock() 811a, logdump() 815d, print, and sleep.

⟨*function* chanintr*(arm)* 814a⟩≡ (828)

```
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() 815c and iprint.

⟨*global* chanlog*(arm)* 814b⟩≡ (828)

```
static Reg chanlog[32][5];
```

⟨*global* `nchanlog` *(arm)* 815a⟩≡ (828)
```
static int nchanlog;
```

⟨*function* `logstart` *(arm)* 815b⟩≡ (828)
```
static void
logstart(Ep *ep)
{
    if(ep->debug)
        nchanlog = 0;
}
```
Uses `nchanlog-2239` 815a.

⟨*function* `clog` *(arm)* 815c⟩≡ (828)
```
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` 814b, `dwc-2235` 810d, and `nchanlog-2239` 815a.

⟨*function* `logdump` *(arm)* 815d⟩≡ (828)
```
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` 814b, `nchanlog-2239` 815a, and `print`.

⟨*function* `chanio` *(arm)* 815e⟩≡ (828)
```
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 631d, HOWMANY 34e, ROUND 34a, chanwait() 813b, clog() 815c, dmaaddr() 831h, dprint, dwc-2235 810d, logdump() 815d, logstart() 815b, print, qlock() 63f, qunlock() 64c, returnfalse() 86a, sofwait() 812b, tsleep, and up.

⟨*function* multitrans*(arm)* 817⟩≡ (828)

```
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 810a and chanio() 815e.

⟨*function* eptrans*(arm)* 818a⟩≡ (828)

```
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 631d, OREAD 279c, OWRITE 279c, Read-2233 810a, Write-2234 810a, chanio() 815e, chanrelease() 811c, chansetup() 811d, multitrans() 817, nexterror, poperror 633d, up, and waserror 633d.

⟨*function* ctltrans*(arm)* 818b⟩≡ (828)

```
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 810f, Estalled 631d, GET2, ROUND 34a, Rd2h, Read-2233 810a, allocb() 407b, assert 693a, chanio() 815e, chanrelease() 811c, chansetup() 811d, freeb() 408b, multitrans() 817, nexterror, poperror 633d, up, and waserror 633d.

⟨*function* ctldata*(arm)* 819⟩≡ (828)

```
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 406b and freeb() 408b.

⟨*function* greset *(arm)* 820a⟩≡ (828)

```
static void
greset(Dwcregs *r, int bits)
{
    r->grstctl |= bits;
    while(r->grstctl & bits)
        ;
    arch_microdelay(10);
}
```

Uses arch_microdelay.

⟨*function* init *(arm)* 820b⟩≡ (828)

```
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 529c, dprint, greset() 820a, malloc() 85a, returnfalse() 86a, setpower() 530a, tsleep, and up.

⟨*function* dump*(*(buses/arm/usbdwc.c)(arm)*)* 821a⟩≡ (828)

```
static void
dump(Hci*)
{
}
```

⟨*function* fiqintr*(arm)* 821b⟩≡ (828)

```
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() 822a, chanintr() 814a, filock() 810g, and fiunlock() 811a.

⟨*function* armtimerset*(arm)* 822a⟩≡ (753)

```
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 31f, TmrEnable-2659 752c, and TmrIntEnable-2660 752c.

⟨*function* irqintr*(arm)* 822b⟩≡ (828)

```
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() 822a, filock() 810g, fiunlock() 811a, and wakeup.

⟨*function* epopen*(arm)* 822c⟩≡ (828)

```
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 631d, Read-2233 810a, Write-2234 810a, assert 693a, ddprint, and malloc() 85a.

⟨*function* epclose*((buses/arm/usbdwc.c)(arm))* 823a⟩≡ (828)

```
    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() 85b, and freeb() 408b.

⟨*function* epread*(arm)* 823b⟩≡ (828)

```
    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 631d, ROUND 34a, Read-2233 810a, TK2MS 223a, allocb() 407b, assert 693a, ctldata() 819, ddprint, eptrans() 818a, freeb() 408b, nexterror, poperror 633d, qlock() 63f, qunlock() 64c, returnfalse() 86a, tsleep, up, and waserror 633d.

⟨*function* epwrite*(arm)* 824⟩≡ (828)

```
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 631d, TK2MS 223a, Write-2234 810a, allocb() 407b, assert 693a, ctltrans() 818b, ddprint, eptrans() 818a, freeb() 408b, nexterror, poperror 633d, qlock() 63f, qunlock() 64c, returnfalse() 86a, tsleep, up, and waserror 633d.

⟨*function* seprintep*((buses/arm/usbdwc.c)(arm))* 825a⟩≡ (828)

```
static char*
seprintep(char *s, char*, Ep*)
{
    return s;
}
```

⟨*function* portenable*(arm)* 825b⟩≡ (828)

```
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 810a, assert 693a, dprint, returnfalse() 86a, tsleep, and up.

⟨*function* portreset*(arm)* 825c⟩≡ (828)

```
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 810a, ResetdelayHS-2232 810a, assert 693a, dprint, print, returnfalse() 86a, tsleep, and up.

⟨*function* portstatus*(arm)* 826a⟩≡ (828)

```
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 693a.

⟨*function* shutdown*(arm)* 826b⟩≡ (828)

```
static void
shutdown(Hci*)
{
}
```

⟨*function* `setdebug` *(arm)* 827a⟩≡ (828)

```
static void
setdebug(Hci*, int d)
{
    debug = d;
}
```

Uses `debug-2236` 810e.

⟨*function* `reset` *(arm)* 827b⟩≡ (828)

```
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` 156b, `USBREGS-2229` 32e, `dprint`, `dwc-2235` 810d, `epopen()` 822c, `epread()` 823b, `epwrite()` 824, `fiqintr()` 821b, `portenable()` 825b, and `setdebug()` 827a.

⟨*function* `usbdwclink` *(arm)* 827c⟩≡ (828)

```
void
usbdwclink(void)
{
    addhcitype("dwcotg", reset);
}
```

Uses `addhcitype()` 785b and `reset()` 827b.

⟨buses/arm/usbdwc.c 828⟩≡

```
/*
 * 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* 35a⟩

```
#include    "io.h"
#include    "../port/usb.h"

#include "dwcotg.h"
```

⟨*constant* USBREGS*(arm)* 32e⟩
⟨*enum* _anon_*(*(buses/arm/usbdwc.c)(arm)*)* 810a⟩

```
typedef struct Ctlr Ctlr;
typedef struct Epio Epio;
```

⟨*struct* Ctlr*(*(buses/arm/usbdwc.c)(arm)*)* 810b⟩

⟨*struct* Epio*(arm)* 810c⟩

⟨*global* dwc*(arm)* 810d⟩
⟨*global* debug*(*(buses/arm/usbdwc.c)(arm)*)* 810e⟩

⟨*global* Ebadlen*(arm)* 810f⟩

```
static void clog(Ep *ep, Hostchan *hc);
static void logdump(Ep *ep);
```

⟨*function* filock*(arm)* 810g⟩

⟨*function* fiunlock*(arm)* 811a⟩

⟨*function* chanalloc*(arm)* 811b⟩

⟨*function* chanrelease*(arm)* 811c⟩

⟨*function* chansetup*(arm)* 811d⟩

⟨*function* sofdone*(arm)* 812a⟩

⟨*function* sofwait*(arm)* 812b⟩

⟨*function* chandone*(arm)* 813a⟩

⟨*function* chanwait*(arm)* 813b⟩

⟨*function* chanintr*(arm)* 814a⟩

⟨*global* chanlog*(arm)* 814b⟩
⟨*global* nchanlog*(arm)* 815a⟩

⟨*function* logstart*(arm)* 815b⟩

⟨*function* clog*(arm)* 815c⟩

⟨*function* logdump*(arm)* 815d⟩

⟨*function* chanio*(arm)* 815e⟩

⟨*function* multitrans*(arm)* 817⟩

⟨*function* eptrans*(arm)* 818a⟩

⟨*function* ctltrans*(arm)* 818b⟩

⟨*function* ctldata*(arm)* 819⟩

⟨*function* greset*(arm)* 820a⟩

⟨*function* init*(arm)* 820b⟩

⟨*function* dump*((buses/arm/usbdwc.c)(arm))* 821a⟩

⟨*function* fiqintr*(arm)* 821b⟩

⟨*function* irqintr*(arm)* 822b⟩

⟨*function* epopen*(arm)* 822c⟩

⟨*function* epclose*((buses/arm/usbdwc.c)(arm))* 823a⟩

⟨*function* epread*(arm)* 823b⟩

⟨*function* epwrite*(arm)* 824⟩

⟨*function* seprintep*((buses/arm/usbdwc.c)(arm))* 825a⟩

⟨*function* portenable*(arm)* 825b⟩

⟨*function* portreset*(arm)* 825c⟩

⟨*function* portstatus*(arm)* 826a⟩

⟨*function* shutdown*(arm)* 826b⟩

⟨*function* setdebug*(arm)* 827a⟩

⟨*function* reset*(arm)* 827b⟩

⟨*function* usbdwclink*(arm)* 827c⟩

Uses Epio 810c.

E.29.7 buses/arm/dma.c

⟨*macro* IRQDMA*(arm)* 829⟩≡ (722a)

```
#define IRQDMA(chan)    (IRQdma0+(chan))
```

⟨*enum* Dma*(arm)* 830a⟩≡ (722a)

```
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)*)* 830b⟩≡ (834)

```
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)* 831a⟩≡ (834)
```
struct Ctlr {
    u32int  *regs;
    Cb  *cb;
    Rendez  r;
    int dmadone;
};
```

⟨*struct* Cb*(arm)* 831b⟩≡ (834)
```
struct Cb {
    u32int  ti;
    u32int  sourcead;
    u32int  destad;
    u32int  txfrlen;
    u32int  stride;
    u32int  nextconbk;
    u32int  reserved[2];
};
```

⟨*global* dma*(arm)* 831c⟩≡ (834)
```
static Ctlr dma[Nchan];
```
Uses Nchan-2285 830b.

⟨*global* dmaregs*(arm)* 831d⟩≡ (834)
```
static u32int *dmaregs = (u32int*)DMAREGS;
```
Uses DMAREGS-2284 31h and dmaregs-2325 831d.

⟨Soc *bus fields* 831e⟩≡ (54c) 831i▷
```
uintptr busdram;
```

⟨soc(raspberry pi1) *other fields(arm)* 831f⟩+≡ (54d) ◁503g 831j▷
```
.busdram    = 0x40000000,
```

⟨soc(raspberry pi2) *other fields(arm)* 831g⟩+≡ (501i) ◁503h 831k▷
```
.busdram    = 0xC0000000,
```

⟨*function* dmaaddr*(arm)* 831h⟩≡ (834)
```
uintptr
dmaaddr(void *va)
{
    return soc.busdram | (PTR2UINT(va) & ~KSEGM); // PADDR?
}
```
Uses KSEGM 40g and PTR2UINT 41a.

⟨Soc *bus fields* 831i⟩+≡ (54c) ◁831e
```
uintptr busio;
```

⟨soc(raspberry pi1) *other fields(arm)* 831j⟩+≡ (54d) ◁831f
```
.busio      = 0x7E000000,
```

⟨soc(raspberry pi2) *other fields(arm)* 831k⟩+≡ (501i) ◁831g
```
.busio      = 0x7E000000,
```

⟨*function* `dmaioaddr` *(arm)* 832a⟩≡ (834)
```
static uintptr
dmaioaddr(void *va)
{
    return soc.busio | (PTR2UINT(va) & ~VIRTIO);
}
```
Uses `PTR2UINT` 41a and `VIRTIO` 31c.

⟨*function* `dump` *(arm)* 832b⟩≡ (834)
```
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)* 832c⟩≡ (834)
```
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)* 832d⟩≡ (834)
```
static int
dmadone(void *a)
{
    return ((Ctlr*)a)->dmadone;
}
```

⟨*function* `dmainterrupt` *(arm)* 832e⟩≡ (834)
```
static void
dmainterrupt(Ureg*, void *a)
{
    Ctlr *ctlr;

    ctlr = a;
    ctlr->regs[Cs] = Int;
    ctlr->dmadone = 1;
    wakeup(&ctlr->r);
}
```
Uses `Cs-2289` 830b, `Int-2307` 830b, and `wakeup`.

⟨*function* `dmastart` *(arm)* 832f⟩≡ (834)
```
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)* 833⟩≡ (834)

```
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 834⟩≡

```
/*
 * 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* 35a⟩

```
#include "io.h"
```

⟨*constant* DMAREGS*(arm)* 31h⟩

```
#define DBG if(Dbg)
```

⟨*enum* _anon_*(*(buses/arm/dma.c)(arm)*)* 830b⟩

```
typedef struct Ctlr Ctlr;
typedef struct Cb Cb;
```

⟨*struct* Ctlr*(arm)* 831a⟩

⟨*struct* Cb*(arm)* 831b⟩

⟨*global* dma*(arm)* 831c⟩
⟨*global* dmaregs*(arm)* 831d⟩

⟨*function* dmaaddr*(arm)* 831h⟩

⟨*function* dmaioaddr*(arm)* 832a⟩

⟨*function* dump*(arm)* 832b⟩

⟨*function* dumpdregs*(arm)* 832c⟩

⟨*function* `dmadone`*(arm)* 832d⟩

⟨*function* `dmainterrupt`*(arm)* 832e⟩

⟨*function* `dmastart`*(arm)* 832f⟩

⟨*function* `dmawait`*(arm)* 833⟩

E.29.8 buses/arm/uartmini.c

⟨*constant* `OkLed`*(arm)* 835a⟩≡ (843b)
```
#define OkLed       16
```

⟨*constant* `TxPin`*(arm)* 835b⟩≡ (843b)
```
#define TxPin       14
```

⟨*constant* `RxPin`*(arm)* 835c⟩≡ (843b)
```
#define RxPin       15
```

⟨*enum* `_anon_(`(buses/arm/uartmini.c)(arm)`)` 835d⟩≡ (843b)
```
/* 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)`)` 835e⟩≡ (843b)
```
/* 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)* 836a⟩≡ (843b)

```
static Uart miniuart = {
    .regs   = (u32int*)AUXREGS,
    .name   = "uart0",
    .freq   = 250000000,
    .phys   = &miniphysuart,
};
```

Uses `AUXREGS-2181` 32a and `miniphysuart` 842c.

⟨*function* `gpiosel` *(arm)* 836b⟩≡ (843b)

```
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` 835d, `FuncMask-2186` 835d, and `GPIOREGS-2180` 32g.

⟨*function* `gpiopull` *(arm)* 836c⟩≡ (843b)

```
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` 32g, `PUD-2198` 835d, `PUDclk0-2202` 835d, and `arch_microdelay`.

⟨*function* `gpiopulloff` *(arm)* 836d⟩≡ (843b)

```
void
gpiopulloff(uint pin)
{
```

```
    gpiopull(pin, Off);
}
```

Uses Off-2199 835d and gpiopull() 836c.

⟨*function* gpiopullup*(arm)* 837a⟩≡ (843b)

```
void
gpiopullup(uint pin)
{
    gpiopull(pin, Pullup);
}
```

Uses Pullup-2201 835d and gpiopull() 836c.

⟨*function* gpiopulldown*(arm)* 837b⟩≡ (843b)

```
void
gpiopulldown(uint pin)
{
    gpiopull(pin, Pulldown);
}
```

Uses Pulldown-2200 835d and gpiopull() 836c.

⟨*function* gpioout*(arm)* 837c⟩≡ (843b)

```
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 835d, GPIOREGS-2180 32g, and Set0-2195 835d.

⟨*function* gpioin*(arm)* 837d⟩≡ (843b)

```
int
gpioin(uint pin)
{
    u32int *gp;

    gp = (u32int*)GPIOREGS;
    return (gp[Lev0 + pin/32] & (1 << (pin % 32))) != 0;
}
```

Uses GPIOREGS-2180 32g and Lev0-2197 835d.

⟨*function* interrupt*(arm)* 837e⟩≡ (843b)

```
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 835e, MuIo-2208 835e, MuLsr-2219 835e, RxRdy-2222 835e, TxRdy-2221 835e, UartIrq-2205 835e, arch_coherence, kbdcr2nl(), uartkick(), and uartrecv().

⟨*function* pnp*(arm)* 838a⟩≡ (843b)

```
static Uart*
pnp(void)
{
    Uart *uart;

    uart = &miniuart;
    if(uart->console == 0)
        kbdq = qopen(8*1024, 0, nil, nil);
    return uart;
}
```

Uses kbdq 360b, miniuart-2228 836a, and qopen() 380a.

⟨*function* enable*(arm)* 838b⟩≡ (843b)

```
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 835d, Bits8-2216 835e, Enables-2206 835e, MuBaud-2227 835e, MuCntl-2223 835e, MuIer-2209 835e, MuIir-2212 835e, MuLcr-2213 835e, RxEn-2226 835e, RxIen-2210 835e, RxPin-2184 835c, TxEn-2225 835e, TxIen-2211 835e, TxPin-2183 835b, UartEn-2207 835e, arch_delay, gpiopulloff() 836d, and gpiosel() 836b.

⟨*function* disable*(arm)* 838c⟩≡ (843b)

```
static void
disable(Uart *uart)
```

```
{
    u32int *ap;

    ap = (u32int*)uart->regs;
    ap[MuCntl] = 0;
    ap[MuIer] = 0;
}
```

Uses MuCntl-2223 835e and MuIer-2209 835e.

⟨*function* kick*(arm)* 839a⟩≡ (843b)

```
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 835e, MuIo-2208 835e, MuLsr-2219 835e, TxDone-2220 835e, TxIen-2211 835e, TxRdy-2221 835e, arch_coherence, and uartstageoutput().

⟨*function* dobreak*(arm)* 839b⟩≡ (843b)

```
/* TODO */
static void
dobreak(Uart *uart, int ms)
{
    USED(uart, ms);
}
```

⟨*function* baud*(arm)* 839c⟩≡ (843b)

```
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 835e.

⟨*function* bits*(arm)* 839d⟩≡ (843b)

```
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 835e, Bits8-2216 835e, Bitsmask-2214 835e, and MuLcr-2213 835e.

⟨*function* stop*(arm)* 840a⟩≡ (843b)

```
static int
stop(Uart *uart, int n)
{
    if(n != 1)
        return -1;
    uart->stop = n;
    return 0;
}
```

⟨*function* parity*(arm)* 840b⟩≡ (843b)

```
static int
parity(Uart *uart, int n)
{
    if(n != 'n')
        return -1;
    uart->parity = n;
    return 0;
}
```

⟨*function* modemctl*(arm)* 840c⟩≡ (843b)

```
/*
 * 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 835e and MuCntl-2223 835e.

⟨*function* rts*(arm)* 841a⟩≡ (843b)
```
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 835e and RtsN-2218 835e.

⟨*function* status*(arm)* 841b⟩≡ (843b)
```
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 704a, free() 85b, malloc() 85a, and readstr() 637d.

⟨*function* donothing*(arm)* 841c⟩≡ (843b)
```
static void
donothing(Uart*, int)
{
}
```

⟨*function* putc*(arm)* 841d⟩≡ (843b)
```
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 32a, MuIo-2208 835e, MuLsr-2219 835e, and TxRdy-2221 835e.

⟨*function* getc*(arm)* 842a⟩≡ (843b)

```
int
getc(Uart*)
{
    u32int *ap;

    ap = (u32int*)AUXREGS;
    while((ap[MuLsr] & RxRdy) == 0)
        ;
    return ap[MuIo] & 0xFF;
}
```

Uses AUXREGS-2181 32a, MuIo-2208 835e, MuLsr-2219 835e, and RxRdy-2222 835e.

⟨*function* uartconsinit*(arm)* 842b⟩≡ (843b)

```
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 621c, miniuart-2228 836a, and uartctl().

⟨*global* miniphysuart*(arm)* 842c⟩≡ (843b)

```
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() 839c, bits() 839d, dobreak() 839b, donothing() 841c, enable() 838b, getc() 842a, kick() 839a, parity() 840b, pnp() 838a, putc() 841d, rts() 841a, status() 841b, and stop() 840a.

⟨*function* okay*(arm)* 843a⟩≡ (843b)

```
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 835a, Output-2188 835d, gpioout() 837c, and gpiosel() 836b.

⟨buses/arm/uartmini.c 843b⟩≡

```
/*
 * bcm2835 mini uart (UART1)
 */
⟨kernel basic includes 35a⟩

#include "io.h"

⟨constant GPIOREGS(arm) 32g⟩
⟨constant AUXREGS(arm) 32a⟩
⟨constant OkLed(arm) 835a⟩
⟨constant TxPin(arm) 835b⟩
⟨constant RxPin(arm) 835c⟩

⟨enum _anon_((buses/arm/uartmini.c)(arm)) 835d⟩

⟨enum _anon_((buses/arm/uartmini.c)2(arm)) 835e⟩

extern PhysUart miniphysuart;

⟨global miniuart(arm) 836a⟩
```

⟨*function* `gpiosel`*(arm)* 836b⟩

⟨*function* `gpiopull`*(arm)* 836c⟩

⟨*function* `gpiopulloff`*(arm)* 836d⟩

⟨*function* `gpiopullup`*(arm)* 837a⟩

⟨*function* `gpiopulldown`*(arm)* 837b⟩

⟨*function* `gpioout`*(arm)* 837c⟩

⟨*function* `gpioin`*(arm)* 837d⟩

⟨*function* `interrupt`*(arm)* 837e⟩

⟨*function* `pnp`*(arm)* 838a⟩

⟨*function* `enable`*(arm)* 838b⟩

⟨*function* `disable`*(arm)* 838c⟩

⟨*function* `kick`*(arm)* 839a⟩

⟨*function* `dobreak`*(arm)* 839b⟩

⟨*function* `baud`*(arm)* 839c⟩

⟨*function* `bits`*(arm)* 839d⟩

⟨*function* `stop`*(arm)* 840a⟩

⟨*function* `parity`*(arm)* 840b⟩

⟨*function* `modemctl`*(arm)* 840c⟩

⟨*function* `rts`*(arm)* 841a⟩

⟨*function* `status`*(arm)* 841b⟩

⟨*function* `donothing`*(arm)* 841c⟩

⟨*function* `putc`*(arm)* 841d⟩

⟨*function* `getc`*(arm)* 842a⟩

⟨*function* `uartconsinit`*(arm)* 842b⟩

⟨*global* `miniphysuart`*(arm)* 842c⟩

⟨*function* `okay`*(arm)* 843a⟩

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: 282e, 283d, 314b, 315b
Abind: 322d, 323a, 323b
Abort-1667: 762b
Abort-2301: 830b
accelerated-928: 656, 656
AccessFlags: 282e
accounttime(): 624a, 624b
Acmderr-1682: 762b
Acreate: 282e, 283d, 307c, 308a, 308d, 564b
Active: 49d, 49e
active: 49e, 113b, 216b, 216c, 252c, 253a, 408a, 535b, 594e, 595e, 596b, 626a, 626b, 712b
Active-2309: 830b
Active.cpus: 49d
Active.exiting: 49d
Active.ispanic: 594f
Active.main_reached_sched: 49d
Active.rebooting: 596a
addbootfile(): 398a
addbroken(): 451d, 451e
addclock0link(): 231a, 238c, 355a, 473g, 603a, 659, 772a
addconf(): 759a, 759c
addelem(): 302a, 304b, 308d
addhcitype(): 785b, 827c
addlist(): 395a, 395b, 398a
addphysseg(): 524a
addr():
addr2check: 524d, 700
addrootdir(): 395b, 395c
adec(): 731a
adjustblock(): 415a
Admitted: 531g, 532a, 533f, 533g, 535b
ainc(): 730d
alarmkproc(): 236c, 267a
alarmr-2608: 236b, 236c, 237b
Alarms:
alarms-2607: 234d, 235a, 236c, 237b
Alarms (typedef):
Allcc-2582: 488b, 491c, 495a

Allexc-2581: 488b
allocb(): 372b, 379, 383, 405a, 407b, 410, 411c, 412, 413a, 414, 806a, 818b, 823b, 824
alloctag():
Alltraps-2579: 488b, 490c
Alt-966: 348a, 348b, 349a, 350d
Alt0-2189: 835d
Alt1-2190: 835d
Alt2-2191: 835d
Alt3-2192: 835d
Alt4-2193: 835d
Alt5-2194: 835d, 838b
Altgr-970: 348a, 349c, 350d, 352b
Amount: 322d, 323a, 323c, 338c
anyhigher(): 215c, 215d, 602c, 712b
anyready(): 200f, 210b, 215c
Aopen: 279d, 282e, 283d, 300a, 338c
AP: 161f, 161j, 502g
archdir-854: 736f, 737c, 737e, 738a, 738b, 738d
archether(): 744a
archreboot(): 595c, 596b, 771b
archreset(): 771a
Arch_Cpu: 48a
Arch_FPsave: 733a
Arch_Proc: 49f
arch_cmpswap(): 731c
arch_coherence: 444a, 444d, 445a, 445b, 486c, 487b, 501a, 501b, 501c, 501d, 501e, 501f, 501g, 529a, 577, 723b, 726b, 726c, 771b, 837e, 839a
arch_countpagerefs: 526a
arch_cycles: 475d
arch_delay: 113b, 152c, 253b, 407b, 511c, 595e, 596b, 610a, 611f, 613b, 616a, 765c, 766b, 766c, 838b
arch_dumpstack: 152c, 213b, 408b, 486b, 609c, 613b
arch_exit: 216c, 408a, 594b, 609c
arch_fastticks: 223g, 226f, 228d, 229d, 238c, 239a, 239b, 240, 241b, 242e, 244a, 553, 813b
arch_idlehands(): 204
arch_intrenable: 157e
arch_isaconfig:
arch_microdelay: 232a, 364b, 764d, 820a, 836c
arch_xdec(): 731c
arch_xinc(): 731c
arch__confinit(): 250b, 254d
arch__cpu0init(): 250b, 252c
arch__cpuinit(): 252c, 252d, 511a
arch__kexit(): 145e, 150b, 151
arch__notify(): 420e, 420g, 421b
arch__syscall(): 145e
arch__trap(): 150b
arch__userinit(): 250b, 265d
Aremove: 282e, 283d, 310d
Arg1-1624: 762b, 766c

Arg2-1622: 762b
ARMLOCAL-2446: 503e, 508c, 511a
ARMLOCAL-2649: 503e, 514c
Armtimer: 625b, 753
ARMTIMER-2648: 31f, 258, 597b, 625a, 822a
Armtimer.count: 625b
Armtimer.ctl: 625b
Armtimer.irq: 625b
Armtimer.irqack: 625b
Armtimer.load: 625b
Armtimer.maskedirq: 625b
Armtimer.predivider: 625b
Armtimer.reload: 625b
Armtimer.val: 625b
Armtimer (typedef): 753
armtimerset(): 821b, 822a, 822b
assert: 152a, 226e, 226f, 227b, 229d, 250b, 261c, 267a, 304b, 358c, 484b, 495a, 511d, 528f, 535b, 633b, 693a, 765a, 766c, 769a, 785c, 818b, 822c, 823b, 824, 825b, 825c, 826a
Atag: 758e, 761b
Atag.size: 758e
Atag.tag: 758e
Atag (typedef): 761b
AtagCmdline-2445: 758f, 760a
AtagCore-2443: 758f, 760a
ataginit(): 760a
AtagMem-2444: 758f, 760a
AtagNone-2442: 758f, 760a
Atodir: 267a, 282e, 311a, 311b, 426a
attachimage(): 175b
Autocmd12-1679: 762b
Autocmd23-1680: 762b
auxpage():
AUXREGS-2181: 32a, 836a, 841d, 842a
balancetime: 221a, 221b
BALLOC: 373, 374, 375, 376, 377, 378, 382a, 382b, 385b, 387, 406c, 411b, 413a
baud(): 839c, 842c
Bdead-781: 408b, 409a, 687
BFREE: 405c
bindmount(): 322b, 322d, 337d
BINTR: 405c, 408a, 408b
Bipck: 405c
bits(): 839d, 842c
Bits7-2215: 835e, 839d
Bits8-2216: 835e, 838b, 839d
Bitsmask-2214: 835e, 839d
bl2mem(): 384, 416a
BLEN: 372a, 373, 374, 375, 376, 377, 378, 379, 382a, 382b, 383, 384, 385b, 405b, 406b, 410, 411a, 411c, 412, 413a, 413b, 414, 415a, 415b, 416a, 806b, 819
Blkcnten-1681: 762b, 764a

Blksize-1746:
Blksizecnt-1623: 762b, 768
Block:
Block (typedef):
blockalloclen(): 381b, 411b
BlockFlags: 405c
blocklen(): 381b, 411a, 411c, 413b
Bogus: 337e, 554, 682c
Bogus.authchan: 682c
Bogus.chan: 682c
Bogus.flags: 682c
Bogus.spec: 682c
Bool:
bool (typedef):
bool_ushort (typedef):
BOOTARGS-2436: 757e, 759b, 760a
BOOTARGSLEN-2437: 757f, 759b, 760a
bootdata-834: 397, 397
bootdir-833: 397, 397
bootlist-835: 396a, 396b, 397, 398a
Boottimeout-1639: 762b
Bpktck: 405c
Break-964: 348a, 349c, 352b
Broken: 136d, 451a, 451b, 451c, 451e, 452c, 454e
broken: 451c, 451e, 452a, 452b
Broken.n: 451b
Broken.p: 451b
BSEG: 100a, 104c, 105c, 612b, 711c
Btcpck: 405c
Budpck: 405c
buf:
Buffered: 496h, 496j, 502f
Bufread-1697: 762b
Bufwrite-1698: 762b
buttonmap-955: 659, 659
byte (typedef):
c:
C-2587: 492b, 492c
Cache:
cache-762:
Cache.hash:
Cache.head:
Cache.pgno:
Cache.tail:
Cache (typedef):
Cachectl: 521f
Cached: 496h, 496j, 502f
cachedel(): 173, 188
cachepage(): 171e, 178, 180c, 188

canacquire(): 444c, 445a, 445b
canflush(): 189d, 194a
canlock(): 60f, 65a, 92c, 93a, 93b, 180c, 201c, 364b, 424, 659
canpage(): 194a, 194b
canqlock(): 65a, 183b, 189d, 192, 236c, 566b, 567a, 711c
canrlock(): 68
Caps-967: 348a, 350d
Card2host-1678: 762b, 764a
Cardinsert-1692: 762b, 766c
Cardintr-1691: 762b, 766b
cas32(): 443a, 443b
catio():
Cb: 831b
Cb.destad: 831b
Cb.nextconbk: 831b
Cb.reserved: 831b
Cb.sourcead: 831b
Cb.stride: 831b
Cb.ti: 831b
Cb.txfrlen: 831b
Cb (typedef): 834
Cbaderr-1686: 762b
Cbalign-2287: 830b
CCACHE: 554, 555
CCEXEC: 297b, 297c, 297d, 558a
cchain():
cclone(): 267a, 274b, 307b, 333b, 334d, 426a
cclose: 138c, 183b, 197, 274b, 277b, 279d, 281a, 281b, 283d, 291, 292c, 293b, 304b, 306c, 306d, 307a, 307c, 308d, 310d, 311a, 312a, 314b, 314c, 316, 319c, 321a, 322d, 328b, 328c, 330b, 330c, 330d, 333a, 333b, 334b, 334d, 337f, 338c, 339c, 342a, 344b, 400b, 426a, 555, 557e, 558a, 564b, 577, 584d, 585a, 588b
ccloseq(): 295c, 296b
Ccrcerr-1688: 762b
cdev():
Cenderr-1687: 762b
CfgMask-2454: 770c, 771b, 771c, 771d
CfgReset-2455: 770c, 771b, 771c
cfgstr-1772:
CFREE: 272b, 274a, 282b, 296b
Chan:
Chan (typedef):
Chanalloc: 273a, 273c
chanalloc: 273c, 811b
chanalloc(): 273c, 811b
Chanalloc.fid: 273a
Chanalloc.free: 273a
Chanalloc.list: 273a
chandebug:
chandevinit(): 267a, 267b
chandevreset(): 250b, 264a

chandevshutdown(): 595a, 596b
chandone(): 813a, 813b
ChanFb-2355: 358b, 725a
chanfree(): 274a, 282b, 296d, 554
chanintr(): 814a, 821b
chanio(): 815e, 817, 818a, 818b
chanlog-2238: 814b, 815c, 815d
ChanMask-2357: 725a, 726b, 726c
ChannelFlag: 272b
chanpath(): 312a, 312b, 316
ChanProps-2356: 725a, 726d
chanrelease(): 811c, 818a, 818b
chansetup(): 811d, 818a, 818b
chanwait(): 813b, 815e
chan_cclose(): 282b
checkaddr: 524d, 700
checkalarms(): 237a, 237b
checkb(): 409a
checkpagerefs(): 525c
checkpages(): 699
cinit():
cistrncmp(): 742a
ckfaultstuck(): 614d, 615a
Client: 161a
ClkArm: 250b, 727b
ClkCore: 727b
clkdiv(): 765a, 766b, 766c
ClkEmmc: 727b, 765c
Clken-1661: 762b, 766b, 766c
Clkfreq8mask-1656: 762b, 765a
Clkfreq8shift-1655: 762b, 765a
Clkfreqms2mask-1658: 762b, 765a
Clkfreqms2shift-1657: 762b, 765a
Clkgendiv-1659: 762b, 766b, 766c
Clkgenprog-1660: 762b
ClkH264: 727b
Clkintlen-1663: 762b, 766b, 766c
ClkIsp: 727b
ClkPixel: 727b
ClkPwm: 727b
ClkSdram: 727b
Clkstable-1662: 762b, 766b, 766c
ClkUart: 727b
ClkV3d: 727b
clockinit(): 250b, 258, 511a
clockintr(): 224d
clockshutdown(): 596b, 597b
clock_arch_delay(): 232a
clock_arch_fastticks(): 224a

clock_arch_microdelay(): 231e
clog(): 814a, 815c, 815e
closeegrp(): 122a, 138c, 563d
closefgrp(): 121d, 138c, 277b, 585b
closepgrp(): 121e, 138c, 321a
closeproc(): 296b, 296d
closergrp(): 122b, 138c, 448d
Clr0-2196: 835d, 837c
Clrerrors-2323: 830b
Clunkq: 295d, 296a
clunkq: 296a, 296b, 296c, 296d
Clunkq.head: 295d
Clunkq.l: 295d
Clunkq.nclosed: 295d
Clunkq.nqueued: 295d
Clunkq.q: 295d
Clunkq.r: 295d
Clunkq.tail: 295d
clunkwork(): 296c, 296d
CMaccelerated-930: 656, 656
CMaddress-2264: 781, 782d, 800
CMadmit-901: 532b, 568b, 569b
CMbuttonmap-950: 659, 659
CMclose-879: 568b, 569b, 586b
CMclosefiles-880: 568b, 569b, 586b
CMclrhalt-2258: 781, 782d, 800
CMcost-898: 532b, 568b, 569b
Cmdbuf:
Cmdbuf (typedef):
Cmddone-1696: 762b, 766c
CMdeadline-897: 532b, 568b, 569b
CMdeadlinenotes-900: 532b, 568b, 569b
CMdebug-2250: 781, 782c, 803
CMdebugep-2265: 781, 782d, 800
cmderror(): 561a, 561b, 656
CMdetach-2263: 781, 782d, 800
cmdinfo: 764a, 766c
Cmdinhibit-1703: 762b, 766c
Cmdtab:
Cmdtab (typedef):
Cmdtm-1625: 762b, 766c
CMdump-2251: 781, 782c, 803
CMevent-904: 468c, 568b, 569b
CMexpel-903: 532b, 568b, 569b
CMextra-902: 532b, 568b, 569b
CMfixedpri-888: 199f, 568b, 569b
CMhalt-846: 593e, 593e, 594a
CMhang-881: 455d, 568b, 569b
CMhub-2254: 781, 782d, 800

CMhwaccel-931: 656, 656
CMhz-2260: 781, 782d, 800
CMinfo-2262: 781, 782d, 800
CMintellimouse-932: 656, 656
CMkill-882: 141c, 568b, 569b
CMlinear-933: 656, 656
CMmaxpkt-2256: 781, 782d, 800
CMname-2266: 781, 782d, 800
CMnew-2252: 781, 782d, 800
CMnewdev-2253: 781, 782d, 800
CMnohang-883: 455e, 568b, 569b
CMnoswap-884: 193e, 568b, 569b
CMntds-2257: 781, 782d, 800
cmount(): 322d, 323e
CMpanic-848: 593e, 593e, 594a
CMperiod-896: 532b, 568b, 569b
CMpollival-2259: 781, 782d, 800
CMpreset-2268: 781, 782d, 800
CMpri-887: 199e, 568b, 569b
CMprivate-885: 567i, 568b, 569b
CMprofile-886: 474b, 568b, 569b
CMps2-934: 656, 656
CMps2intellimouse-935: 656, 656
CMreboot-847: 593e, 593e, 594a
CMres-936: 656, 656
CMreset-937: 656, 656
CMsamplesz-2261: 781, 782d, 800
CMscrollswap-951: 659, 659
CMserial-938: 656, 656
CMSG: 280c, 398c, 402, 434a, 434b, 549b, 549c
CMspeed-2255: 781, 782d, 800
CMsporadic-899: 532b, 568b, 569b
CMstart-890: 454a, 568b, 569b
CMstartstop-891: 470c, 568b, 569b
CMstartsyscall-892: 459b, 568b, 569b
CMstop-893: 453e, 568b, 569b
CMswap-952: 659, 659
CMtmout-2267: 781, 782d, 800
CMtrace-895: 466c, 568b, 569b
CMwaitstop-894: 455h, 568b, 569b
CMwildcard-953: 659, 659
CMwired-889: 216f, 568b, 569b
cnodata():
CntEnable-2657: 258, 752c
CntPrescaleMask-2656: 752c
CntPrescaleShift-2655: 752c
CntWidth16-2664: 752c
CntWidth32-2665: 258, 752c
Coarse: 160d, 167, 209b, 249c

Conblkad-2290: 830b
concatblock(): 411c
concatblockcnt-771: 411c, 611d
condok(): 492c, 493c
Conf: 46a
conf: 46b, 115a, 170b, 186a, 189d, 192, 194a, 195d, 197, 216g, 219, 252c, 253a, 254d, 256a, 256b, 260, 261a, 261c, 262b, 263c, 408a, 428d, 428f, 440c, 456a, 526c, 533f, 535b, 570d, 574c, 576, 580d, 611e, 612a, 626a, 711c, 760a
Conf.copymode: 170a
Conf.ialloc: 254b
Conf.mem: 46a
Conf.ncpu: 46a
Conf.nimage: 254b
Conf.npage: 254b
Conf.nproc: 46a
Conf.nswap: 185c
Conf.nswppo: 189a
Conf.pipeqsize: 428c
Conf.upages: 254b
Conf (typedef):
Confdata:
Confdata.cf:
Confdata.on:
Confdata.spec:
Confdata (typedef):
confegrp-787: 673c, 674
Config-2353: 724c
configs-1776:
configure():
Confmem: 46c
Confmem.base: 46c
Confmem.kbase: 456c
Confmem.klimit: 456c
Confmem.limit: 46c
Confmem.npage: 46c
Confmem (typedef):
confname: 47a, 267a, 758g, 759a
confstr-1771:
confval: 47a, 267a, 759a
consactive(): 595e, 620b, 631a
consattach(): 367b, 643
consclose(): 365c, 367b
consdebug: 619e, 619e, 619f, 619g
consdevtab: 367b
consdir-1797: 360a, 365b, 366a, 643
consinit(): 365a, 367b
ConsKbd: 353d, 354a
ConsKbd.ctl: 368a
ConsKbd.ctlpoff: 594c

ConsKbd.ie: 353d
ConsKbd.ir: 353d
ConsKbd.istage: 353d
ConsKbd.iw: 353d
ConsKbd.line: 360f
ConsKbd.lockputc: 353d
ConsKbd.raw: 368a
ConsKbd.x: 360f
consopen(): 365b, 367b
consread(): 366a, 367b
consstat(): 367b, 643
consuart: 621c, 842b
consumecnt-774: 375, 611d
conswalk(): 367b, 643
conswrite(): 366b, 367b
Control0-1632: 762b, 766c
Control1-1633: 762b, 765c, 766b, 766c
Control2-1637: 762b
COPEN: 272b, 368e, 400a, 404a, 430a, 431b, 432, 469a, 584d, 618g, 619a, 659, 674, 795, 796b
copen():
copyblock(): 414, 415a
copyblockcnt-773: 414, 611d
copypage(): 170c, 170d, 180c
copypath(): 301a, 301c
Counter (typedef): 69d
Cpaccd16-2584: 488c, 489c
Cpaccnosimd-2583: 488c, 489c
CpACsmp: 506e
cpage():
CpAuxctl: 506d
CpCACHE: 496a
CpCACHEall: 497f
CpCACHEintr: 497e
CpCACHEinvd: 497e
CpCACHEinvi: 497e
CpCACHEinvu: 497e
CpCACHERANGEdwb: 496f
CpCACHERANGEdwbi: 496f
CpCACHERANGEinvd: 496f
CpCACHEse: 497f
CpCACHEsi: 497f
CpCACHEwait: 497f
CpCACHEwb: 497e
CpCACHEwbi: 497e
CpCdcache: 496b
CpChv: 29d
CpCicache: 496b
CpCLD: 504c
CpCLDcyc: 504d

CpCLDena: 504d
CpCLDenacyc: 504e
CpCLDenapmnc: 504e
CpCmmu: 29d
CpCONTROL: 29b, 489c
CpCPaccess: 489b, 489c
CpCpredict: 506a
CpCsbo: 506c
CpCsbz: 506c
CpCsw: 506a
CpCtre: 506b
CpCve: 506b
CpDAC: 29b
CpDFP: 485b, 487b, 489c
CpFAR: 29b
CpFP: 485b, 486a, 489c
CpFSR: 29b
CpFSRdata: 153e
CpFSRinst: 153e
cpgmove():
CpID: 502a, 514c
CpIDcsize: 515c
CpIDcssel: 515c
CpIDfeat: 502b, 514c
CpIDid: 502c
CpIDidct: 502b
CpIDmpid: 502c
CpMainctl: 29c
cprd(): 723a, 723c
cprdsc(): 489c, 514c, 723c
CpSC: 29a, 723c, 724a
CpSPM: 29b
CpSPMctl: 475f
CpSPMcyc: 475f
CpSPMperf: 475e
CpTIMER: 504a, 514c, 514e, 514g
CpTIMERphys: 504b, 514c, 514e, 514g
CpTIMERphysctl: 504b, 514c, 514e, 514g
CpTIMERphysval: 504b, 514g
CpTLB: 29b
CpTLBinv: 162i
CpTLBinvse: 162i
CpTLBinvu: 162h
CpTTB: 29b
CpTTB1: 503a
CpTTBs: 503b
CPU: 144a
Cpu: 48a
Cpu.cpuhz: 48d

Cpu.cpumhz: 48d
Cpu.cpuno: 48a
Cpu.cs: 625c
Cpu.cyclefreq: 475b
Cpu.flushmmu: 441b
Cpu.ilockdepth: 71c
Cpu.intr: 625c
Cpu.lastintr: 48a
Cpu.load: 623f
Cpu.perf: 623a
Cpu.pfault: 625c
Cpu.proc: 48a
Cpu.readied: 211c
Cpu.sched: 203e
Cpu.schedticks: 215a
Cpu.splpc: 56f
Cpu.spuriousintr: 625c
Cpu.syscall: 625c
Cpu.ticks: 48c
Cpu.tlbfault: 625c
Cpu.tlbpurge: 625c
Cpu (typedef):
CPUS: 60c, 60g, 62a, 203a, 204, 211b, 211d, 216g, 218d, 218h, 219, 220b, 235a, 235b, 236c, 237b, 239a, 239b, 252d, 364b, 440c, 471c, 471d, 472b, 472c, 472f, 601c, 626a, 626b, 645, 659
cpus: 49a, 252c, 507a
cputempread(): 739c, 739d
cputype2name(): 253c
cpwr(): 723b, 724a
cpwrsc(): 489c, 514c, 514e, 514g, 724a
cqid-1774:
Crcchken-1670: 762b, 764a
CRCLOSE: 298d, 298e, 298f, 674
cread():
createdir(): 334c, 334d
Cs-2289: 830b, 832e
ctail():
ctldata(): 819, 823b
CTLQ-1852:
ctlread(): 796c, 798b
CTLS-1851:
ctltrans(): 818b, 824
ctlwrite(): 804, 805
Ctoerr-1689: 762b, 766c
Ctrl-965: 348a, 348b, 349a, 349c, 350d, 352b
CtsFlow-2224: 835e, 840c
cunmount(): 338c, 339c
cupdate():
curs: 659, 659
cursor: 659, 659

Cursortocursor(): 659, 659
cvlist: 649, 649
cvlist.ld: 649
cvlist.si: 649
cvlist.so: 649
cwrite():
DAC: 161h
Data-1630: 762b, 769a
data2txt(): 177b, 177c
Datactive-1701: 762b
datadone(): 765b, 766c, 769a
Datadone-1695: 762b, 765b, 766a, 766c, 769a
Datatomask-1654: 762b
Datatoshift-1653: 762b, 766b, 766c
Datinhibit-1702: 762b, 766c
DBG-1801:
Dbg-2288: 830b
DBG-765: 282b, 283d, 296b, 301a, 328c, 330c, 330d
Dbgsel-1640: 762b
Dcrcerr-1684: 762b
ddprint: 790, 792, 822c, 823a, 823b, 824
Dead: 51b, 136a, 192, 194a, 440c, 535b, 574c, 576, 577, 584d, 612a, 711c
Deadline: 533g
deadlineintr(): 535b, 535b
debug-1761:
debug-2236: 810e, 827a
debug-2277: 783c, 803
Debug-2297: 830b
Debug-2405: 614d, 750a
debugging-778: 611d
deccnt(): 70h, 71a, 208c
decref(): 69c, 277b, 282b, 296b, 301b, 318b, 321a, 368e, 448d, 563d, 787a
Del-983: 348a, 349c, 352b, 594b
delayedscheds: 207c, 535b, 629b
DELTAFD: 276c, 277a, 278b
deltree():
Denderr-1683: 762b
deprint: 787a, 789a, 795, 796b, 799b, 800
dequeueproc(): 201c, 204, 220b, 535b
Destad-2293: 830b
Destdreq-2316: 830b
Destignore-2315: 830b
Destinc-2318: 830b
Destwidth128-2317: 830b
Dev:
DEV-1612:
Dev (typedef):
devalloc():
devattach(): 275a, 428f, 589d, 643, 651, 653, 659, 670b, 680, 737d, 794a

devbread(): 367b, 393b, 405a, 433b, 565, 571a, 590d, 651, 653, 659, 739a
devbwrite(): 367b, 393b, 405b, 565, 571a, 590d, 651, 653, 659, 739a
devclone(): 400a, 400b, 555
DevConf:
DevConf (typedef):
devcons_iprint(): 363e
devcons_panic(): 630b
devcons_pprint(): 364c
devcons_print(): 363b
devcons__assert(): 631c
devcreate(): 367b, 393b, 435a, 571a, 590d, 651, 653, 686, 739a, 807a
devdir(): 396a, 398c, 399, 402, 429, 430b, 580d, 674, 790
devdirread(): 366a, 396b, 403c, 433a, 571d, 589e, 651, 653, 659, 674, 738d, 798b
DEVDOTDOT: 396a, 399, 400b, 429, 580d, 674, 790
devgen(): 365b, 366a, 396a, 399, 400b, 589e, 643, 651, 653, 659, 670b, 672a, 672b, 672c, 737e, 738a, 738b, 738d
Devgen (typedef):
devinit(): 393b, 565, 651, 653, 686
devletters-1578:
DevLOG-1604:
DevMASK-1606:
devno(): 275a, 275b, 321c, 337f, 346, 426a
devopen(): 404a
devpermcheck(): 403d, 404a
Devport:
Devport (typedef):
devremove(): 367b, 393b, 435a, 571a, 590d, 651, 653, 659, 686, 739a, 807a
devreset(): 367b, 435a, 565, 571a, 590d, 651, 653, 686, 739a
devs-1579:
DevSHIFT-1607:
devshutdown(): 367b, 393b, 435a, 565, 571a, 590d, 651, 653, 659, 686, 739a
devslock-1580:
devstat(): 402, 643, 651, 653, 659, 670b, 672b, 674, 680, 738a, 794c
devtab: 54b
devwalk(): 400b, 430a, 643, 651, 653, 659, 670b, 672a, 674, 680, 737e, 794b
devwstat(): 367b, 393b, 565, 579, 590d, 651, 653, 659, 686, 739a, 807a
Dir (typedef):
DirEntry:
DirEntry (typedef):
direntry_setname(): 313b, 314b, 344b
dirfixed(): 344b, 346
Dirlist: 394b
Dirlist.base: 394b
Dirlist.data: 394b
Dirlist.dir: 394b
Dirlist.mdir: 394b
Dirlist.ndir: 394b
Dirlist (typedef):
Dirmode: 313a

Dirtab:
Dirtab (typedef):
disk-1768:
Dl-2597: 535b, 535b
dma-2324: 831c
dmaaddr(): 358b, 726d, 815e, 831h
DmaChanEmmc: 769a, 830a
DmaChanSpiRx: 830a
DmaChanSpiTx: 830a
DmaD2M: 769a, 830a
DmaDevEmmc: 769a, 830a
DmaDevSpiRx: 830a
DmaDevSpiTx: 830a
dmadone(): 832d
DmaFlags: 830a
dmainterrupt(): 832e
dmaioaddr(): 832a
DmaM2D: 769a, 830a
DmaM2M: 830a
DMAREGS-2284: 31h, 831d
dmaregs-2325: 831d, 831d
DMDIR: 308a, 313a, 360a, 394c, 395a, 395b, 397, 402, 428b, 429, 430b, 580d, 589b, 659, 674
DMEXCL: 618c, 618d, 789b
DMEXEC: 313a
DMMOUNT: 402, 403a
DMREAD: 313a
DMWRITE: 313a
Dn-2576: 488b, 490c
dobreak(): 839b, 842c
Dom0: 161d, 167, 209b, 249c, 358c, 503i
domount(): 304b, 323b, 327b, 328c
donothing(): 841c, 842c
Dontprint-2592:
doread():
Down-980: 348a, 349c, 352b
dprint: 788b, 792, 793a, 793b, 803, 813b, 815e, 820b, 825b, 825c, 827b
dprint-1762:
DPRINT-2593: 535b
drawactive(): 659
Dreq-2306: 830b
DSEG: 100a, 105c
DTO-1619: 762a, 766b, 766c
Dtoerr-1685: 762b, 766b
dumpaproc: 612a, 613a
dumpdregs(): 832c
dumpeps(): 787b, 803
dumplongs(): 613b, 614a
dumpstackwithureg(): 615b, 616a
dupattach():

dupclose():
dupdevtab:
dupfgrp(): 119b, 121d, 265d, 276c
dupgen():
dupopen():
duppage(): 180a, 180c
dupread():
dupretries-1850: 180b, 180b, 180c
dupseg(): 102b, 117d
dupstat():
dupswap(): 187e, 187f
dupwalk():
dupwrite():
dwc-2235: 810d, 815c, 815e, 827b
Dwcregs: 772d, 773
Dwcregs.adpctl: 772d
Dwcregs.dtxfsiz: 772d
Dwcregs.gahbcfg: 772d
Dwcregs.gdfifocfg: 772d
Dwcregs.ggpio: 772d
Dwcregs.ghwcfg1: 772d
Dwcregs.ghwcfg2: 772d
Dwcregs.ghwcfg3: 772d
Dwcregs.ghwcfg4: 772d
Dwcregs.gi2cctl: 772d
Dwcregs.gintmsk: 772d
Dwcregs.gintsts: 772d
Dwcregs.glpmcfg: 772d
Dwcregs.gnptxfsiz: 772d
Dwcregs.gnptxsts: 772d
Dwcregs.gotgctl: 772d
Dwcregs.gotgint: 772d
Dwcregs.gpvndctl: 772d
Dwcregs.gpwrdn: 772d
Dwcregs.grstctl: 772d
Dwcregs.grxfsiz: 772d
Dwcregs.grxstsp: 772d
Dwcregs.grxstsr: 772d
Dwcregs.gsnpsid: 772d
Dwcregs.guid: 772d
Dwcregs.gusbcfg: 772d
Dwcregs.haint: 772d
Dwcregs.haintmsk: 772d
Dwcregs.hcfg: 772d
Dwcregs.hchan: 772d
Dwcregs.hfir: 772d
Dwcregs.hflbaddr: 772d
Dwcregs.hfnum: 772d
Dwcregs.hport0: 772d

Dwcregs.hptxfsiz: 772d
Dwcregs.hptxsts: 772d
Dwcregs.pad0: 772d
Dwcregs.pad1: 772d
Dwcregs.pad2: 772d
Dwcregs.pad3: 772d
Dwcregs.pcgcctl: 772d
Dwcregs.reserved0: 772d
Dwcregs.reserved1: 772d
Dwcregs (typedef): 773
Dwidth1-1649: 762b
Dwidth4-1648: 762b, 766c
Ebadarg: 101c, 105a, 114d, 119a, 119c, 120a, 121c, 147b, 196c, 280a, 293b, 322d, 356d, 357, 402, 431b, 436b, 437, 438b, 439, 441c, 446a, 446b, 447a, 456a, 457, 530c, 557e, 559, 574c, 592g, 631d, 631d, 634d, 653, 659
Ebadchar: 285f, 631d, 631d
Ebadctl: 631d
Ebadexec: 631d, 631d
Ebadfd: 280c, 588b, 631d, 631d
Ebadip: 631d, 631d
Ebadlen-2237: 810f, 810f, 818b
Ebadsharp: 321c, 631d, 631d
Ebadspec: 631d, 631d
Ebadstat: 631d
Ebadtimectl: 238a, 238a, 243a, 244b
Ebadusefd: 280c, 631d, 631d, 653, 798b, 805
Ecache:
ecache-763:
Ecache.free:
Ecache.head:
Ecache.total:
Ecache (typedef):
Echange: 631d, 631d
echo(): 354c, 355b
echoscreen(): 356a, 356c
echoserialoq(): 620c, 620d
Ecmdargs: 561b, 631d, 631d
Econinuse: 631d, 631d
Econrefused: 631d, 631d
Edetach: 631d, 631d, 798b, 804, 805
Edf:
Edf (typedef):
edfadmit(): 532b, 535b
EdfFlags: 533g
edfinit(): 532b, 535b
edfinited: 535b, 535b
edflock(): 531c, 535b, 535b
edfnrun: 534b, 535b
edfready(): 531i, 535b
edfrecord(): 531g, 535b

edfrun(): 531c, 535b, 535b
edfschedlock: 535b, 535b
edfstop(): 451e, 531h, 532b, 533b, 535b
edfunlock(): 531c, 535b, 535b
edfyield(): 532a, 535b
Edirseek: 332f, 631d, 631d
Edoesnotexist-768: 307a, 632a, 632a
Eexist: 631d
Efilename: 321c, 631d, 631d
Egone-1777:
Egreg: 196c, 366a, 366b, 396b, 422, 426a, 571d, 574c, 577, 589e, 590b, 631d, 631d, 651, 659, 670b, 823b, 824
Egrp:
Egrp (typedef):
Ehdr32sz-2416:
Ehdr64sz-2419:
Ehungup: 381a, 389b, 389c, 631d, 631d
Eintr: 631d
Einuse: 631d
Eio: 631d
Eioload: 178, 631d, 631d
Eisdir: 631d
Eismtpt: 310d, 316, 631d, 631d
Eisstream: 427a, 631d, 631d
Elemlist: 287a
Elemlist.aname: 287a
Elemlist.elems: 287a
Elemlist.mustbedir: 287a
Elemlist.name: 287a
Elemlist.nelems: 287a
Elemlist.nerror: 287a
Elemlist.off: 287a
Elemlist.prefix: 287a
Elemlist (typedef):
elfident-2422:
Emask-906: 468f, 468h, 469c
emmc-1704: 764c, 764d, 765c, 766a, 766b, 766c, 769a
emmccmd(): 766c, 769b
emmcenable(): 766b, 769b
emmcinit(): 765c, 769b
emmcinquiry(): 765d, 769b
emmcio(): 769a, 769b
emmciosetup(): 768, 769b
EMMCREGS-1615: 32f, 764d, 765b, 765c, 765d, 766a, 766b, 766c, 769a
Emount: 323e, 335c, 631d, 631d
Emountrpc: 552, 553, 631d, 631d
Empty-2352: 724c, 726c
emptystr(): 431a, 579, 637a
enable(): 838b, 842c
Enable-2299: 830b

Enable-2666: 514e, 514g, 752c
Enabledelay-2230: 810a, 825b, 825c
Enables-2206: 835e, 838b
End-2308: 830b
End-979: 348a, 349c, 352b
Enegoff: 291, 292c, 293b, 631d, 631d
Enoattach: 336c, 336d, 631d, 631d
Enochild: 133a, 566c, 631d, 631d
Enocreate: 334d, 631d, 631d
Enodev: 631d, 631d
Enofd: 279d, 307c, 426a, 558a, 588b, 631d, 631d
Enomem: 631d
Enonexist: 631d
Enoreg: 571d, 574c, 631d, 631d
Enotconf: 631d, 631d, 795, 822c
Enotdir: 631d
Enoversion: 554, 631d, 631d
Enovmem: 98a, 105c, 106c, 175b, 437, 631d, 631d
envclose(): 565, 674
envcpy(): 120b, 122a, 564a
envcreate(): 565, 674
envdevtab: 565
envgen(): 674, 674
envgrp(): 673a, 674
envlookup(): 674, 674
envopen(): 565, 674
envread(): 565, 674
envremove(): 565, 674, 674
envstat(): 565, 674
envwalk(): 565, 674
envwrite(): 565, 674
envwriteable(): 673b, 674
epalloc(): 786a, 788b, 789a
epclose(): 823a
epctl(): 800, 804
epctls-2273: 782d, 800
epdataperm(): 789b, 790
Eperm: 631d
Epio: 810c, 828
Epio.cb: 810c
Epio.lastpoll: 810c
Epio (typedef): 828
epmax-2282: 783h, 784d, 786a, 786b, 787a, 787b, 790, 796c, 803
epopen(): 822c, 827b
epread(): 823b, 827b
Eprocdied: 325e, 454e, 566b, 571d, 574c, 577, 579, 584d, 585b, 631d, 631d
eps-2281: 783g, 784d, 786a, 786b, 787a
epslck-2280: 783f, 786a, 786b, 787a, 788a
eptrans(): 818a, 823b, 824

epwrite(): 824, 827b
eqchan(): 53c, 323e, 339c
eqchantdqid(): 53d, 328b, 344b
eqqid(): 53a
Err-1690: 762b, 765b, 766a, 766c, 769a
ERRMAX: 134a, 134b, 152d, 164b, 179, 289a, 289c, 389c, 400b, 421a, 421b, 494b, 561a, 574c, 576, 633b, 633c, 634d, 634e, 692a
error:
Error-2302: 830b
error0 (typedef):
error1 (typedef):
errorneg1 (typedef):
ERROR_0:
ERROR_1:
ERROR_NEG1: 166b, 359a, 359b, 530b
error_if_not_dir(): 299b, 400b
Esbadstat: 631d, 631d
ESEG: 100a, 437
Eshort: 403c, 549c, 574c, 631d, 631d, 659
Eshortstat: 430b, 431a, 579, 631d, 631d
Eshutdown: 631d, 631d
Esoverlap: 106b, 437, 631d, 631d
Estalled: 631d, 631d, 815e, 818a, 818b
Etimedout: 631d, 631d
Etoobig: 574c, 576, 631d, 631d, 674
Etoosmall: 631d
Eunion: 339c, 631d, 631d
Eunmount: 339c, 631d, 631d
Evalue:
Evalue (typedef):
eve: 114d, 139c, 265d, 396a, 398c, 399, 402, 403d, 429, 430b, 431a, 557a, 557b, 579, 580d, 592a, 674, 711c, 790
eventsavailable(): 468e, 469c
ewalk(): 304b, 306a, 331b
executeio(): 192, 195d
Execvals:
Execvals (typedef):
exhausted(): 104b, 428f, 549c, 633c
Exrdfifocfg-1641: 762b
Exrdfifoen-1642: 762b
Extent:
Extent (typedef):
extentalloc():
extentfree():
Extfreq-1616: 762a, 765c
Extratime: 532b, 533g, 535b
F: 161g
false: 53c, 53d, 60f, 62d, 64c, 65a, 67d, 68, 79, 86a, 101a, 101d, 117e, 134b, 139b, 139c, 140c, 141d, 145e, 150b, 154e, 159b, 162f, 170b, 170c, 178, 193c, 194a, 194b, 198d, 199e, 209b, 229d, 252c, 273d, 275a, 279d, 280b, 285e, 287d, 292c, 304b, 306b, 307c, 310d, 311a, 312a, 314b, 314c, 315b, 315c, 315d, 322b, 322d, 326b,

328b, 337f, 338c, 343, 350d, 353c, 361, 362b, 363c, 364b, 368e, 368f, 369b, 400b, 406e, 419e, 419g, 420f, 426a, 437, 440c, 441a, 444a, 444c, 445a, 445b, 450a, 455e, 456d, 457, 459c, 460e, 465, 466a, 468f, 489c, 490a, 557e, 558a, 567f, 571d, 584b, 586b, 588b, 594d, 608b, 637a, 639d, 659, 712b, 726d
fasthz: 242c, 242e, 244a, 244b
fastticks2us(): 241b, 241c
Fault: 161c, 163a, 167, 358c
fault(): 164b, 165
faultarm(): 153c, 154d, 155a, 164b
faulterror(): 178, 179
fbblank(): 359b
fbdefault(): 358b, 359a
Fbinfo: 32c, 729
Fbinfo.base: 32c
Fbinfo.bpp: 32c
Fbinfo.pitch: 32c
Fbinfo.screensize: 32c
Fbinfo.xoffset: 32c
Fbinfo.xres: 32c
Fbinfo.xresvirtual: 32c
Fbinfo.yoffset: 32c
Fbinfo.yres: 32c
Fbinfo.yresvirtual: 32c
Fbinfo (typedef): 729
fbinit(): 358b
fcallcount: 629c, 629e
Fcat-1739:
Fclear-1742:
fdclose(): 280b, 281b, 338a
Fdel-1743:
Fdisk-1744:
fdt (typedef):
fdtochan(): 196c, 280b, 280c, 291, 292c, 293b, 312a, 314c, 315c, 337f, 557e, 558a, 588b
FEXT: 34d
Fgrp:
Fgrp (typedef):
filock(): 810g, 812b, 813b, 821b, 822b
findconf(): 758g, 759a
findfreefd(): 277c, 278a, 426b
findmount(): 304b, 328b, 328c, 334c, 344b
Finter-1740:
fiq(): 529a
Fiqenable-2407: 528c, 528e, 528f
fiqintr(): 821b, 827b
FIQSTKTOP: 527e, 527f
fiunlock(): 811a, 812b, 813b, 821b, 822b
fixdotdotname(): 302a, 302c
fixfault(): 165, 166b, 457
Fmirror-1738:
Fmt:

Fmt (typedef):
fmtl-1802:
fmtrwdata(): 460e, 464, 465
Fmts (typedef): 693c
fmtuserstring(): 460e, 464, 466a
Fnone-1737:
forceclosefgrp(): 295b, 295c
Forceirpt-1638: 762b
FPactive: 480a, 483a, 484a, 484c, 494c
FPAIDNRM-2580: 488b, 488b
Fpart-1741:
fpcfg(): 490c, 491b, 491c
fpclear(): 483a, 491d, 494c
Fpdex-2572: 488b, 488b
Fpenabled-2571: 488b, 490b
Fpex-2570: 488b
Fpexc-2567: 488a, 490a, 490b, 491d
FPIDNRM-2578: 488b, 488b
FPillegal: 480a, 481c, 482b, 484c, 485a, 493c
FPinactive: 480a, 483a, 484a, 484c, 494c
FPinit: 480a, 481a, 494c
fpinit(): 491b, 494c
Fpinst-2568: 488a
Fpinst2-2569: 488a
Fpmbc-2573: 488b, 491d
fpoff(): 482d, 484b, 490a
fpon(): 484b, 491c, 491d, 495a, 771a
fponly(): 490b, 491b, 491c
fprd(): 484b, 486b, 489c, 490c, 491c, 491d, 495a
fprestore(): 495a
fprestreg(): 487b, 495a
fpsave(): 484b
FPSaveStatus: 480a
Fpscr-2564: 484b, 488a, 490c, 491c, 495a
Fpsid-2563: 488a, 489c, 490c
fpstuck(): 491e, 493c
fpuemu(): 493c, 494a
fpunoted(): 423b, 485a
fpunotify(): 421b, 484c
fpuprocrestore(): 483b, 483c
fpuprocsave(): 482e, 483a
fpusysprocsetup(): 482c, 482d
fpusysrfork(): 483d, 484a
fpusysrforkchild(): 482a, 482b
fpwr(): 486c, 490a, 490b, 490c, 491c, 491d, 495a
FRAMEBUFFER: 32d, 358b
free(): 85b, 96d, 98b, 105e, 132c, 136d, 137c, 182d, 275a, 276c, 277b, 278b, 283d, 285f, 287c, 288a, 289b, 301b, 302a, 304b, 307a, 307b, 318b, 319c, 321a, 325d, 325e, 330b, 337f, 338a, 342b, 342d, 344b, 389a, 400b, 408b, 428f, 431a, 432, 448d, 459c, 459e, 460c, 460e, 464, 465, 466a, 473e, 474b, 524c, 531h, 548b, 548d, 549c,

555, 558a, 560c, 563d, 566b, 574c, 579, 586b, 593b, 594a, 626a, 636b, 651, 653, 659, 674, 759b, 786a, 787a, 792, 796b, 796c, 800, 803, 804, 823a, 841b
freeb(): 372b, 374, 384, 385b, 387, 405a, 405b, 408b, 409b, 410, 412, 413a, 413b, 415a, 415b, 416a, 806a, 806b, 818b, 819, 823a, 823b, 824
freeblist(): 375, 376, 377, 389b, 391d, 409b, 411c, 413b, 415a, 548b
freebroken(): 192, 452b
freept(): 96d, 98b
freetag(): 548a, 548b
fscache-761:
Fsdev:
Fsdev.gone:
Fsdev.inner:
Fsdev.name:
Fsdev.ndevs:
Fsdev.perm:
Fsdev.size:
Fsdev.start:
Fsdev.tree:
Fsdev.type:
Fsdev.vers:
Fsdev (typedef):
fsdevtab:
Fsel0-2185: 835d, 836b
fstree-1764:
Full-2351: 724c, 726b
FuncMask-2186: 835d, 836b
Fz-2577: 488b, 490c
genage-1809: 189d, 195a, 195b
genclock-1810: 189d, 195a, 195b
gencount-1811: 189d, 195a, 195b
generrstr(): 634c, 634d
genrandom(): 602c, 603a
gensum-1812: 189d, 195a, 195b
gentick(): 192, 195a
GET2: 798a, 818b
getattrs():
getboardrev(): 264d, 264g
getc(): 842a, 842c
getclkrate(): 728a, 765c
getconfenv(): 673c, 759b
getcputemp(): 728c, 739c
getdev():
getep(): 786b, 787b, 790, 795, 796b, 796c, 798b, 803, 804, 805
getethermac(): 267a, 727a
getfirmware(): 264d, 265a
getncpus(): 507e
getnewport():
getpcsp(): 615c, 615d
getpgcolor: 521d

getpower(): 530b
getramsize(): 253d, 254d
gettree():
GiB: 34h
gpioin(): 837d
gpioout(): 837c, 843a
gpiopull(): 836c, 836d, 837a, 837b
gpiopulldown(): 837b
gpiopulloff(): 836d, 838b
gpiopullup(): 837a
GPIOREGS-2180: 32g, 836b, 836c, 837c, 837d
gpiosel(): 836b, 838b, 843a
greset(): 820a, 820b
growfd(): 278a, 278b, 588b
growparse(): 287d, 288a
gscreen: 659
hasfreepages(): 191c
haswaitq(): 133c, 133d, 566b
havefp(): 489c, 490b, 491b, 491c
hciprobe(): 792, 793a
hcis-2279: 783e, 787b, 793a, 793b, 806c
Hcitype: 782a, 807b
Hcitype.reset: 782a
Hcitype.type: 782a
Hcitype (typedef): 807b
hcitypes-2278: 783d, 785b, 792, 793a
Hdrspc-780: 406e, 687
Hole:
Hole (typedef):
Home-973: 348a, 349c, 352b
Host2card-1677: 762b, 764a
Hostchan: 772d, 772d, 773
Hostchan.hcchar: 772d
Hostchan.hcdma: 772d
Hostchan.hcdmab: 772d
Hostchan.hcint: 772d
Hostchan.hcintmsk: 772d
Hostchan.hcsplt: 772d
Hostchan.hctsiz: 772d
Hostchan.reserved: 772d
Hostchan (typedef): 773
hostdomain: 557d, 559, 592a
hostdomainwrite(): 559, 592b
hostownerwrite(): 114d, 592b
HOWMANY: 34e, 815e
HVECTORS: 28c, 249c, 256e
hzclock(): 215f, 229d
hzsched(): 215c, 215f
Ialloc: 407c, 407d

ialloc: 407d, 408a, 408b, 611e
Ialloc.bytes: 407c
iallocb(): 378, 387, 408a
iallocsummary(): 611d, 611e
ibrk(): 104c, 104d, 436b
ihash: 174b, 175b, 182d
IHASHSIZE: 174b
ilock(): 61d, 77d, 78c, 82a, 227d, 228a, 228c, 229d, 231a, 238c, 238d, 239a, 239b, 240, 354b, 373, 374, 375, 376, 377, 378, 379, 381a, 383, 384, 385b, 387, 389b, 389c, 390b, 391d, 408a, 408b, 535b, 617e, 810g
ILock (typedef): 61a
Imagealloc: 174b, 174e
imagealloc-1803: 174e, 175b, 182d, 183b, 263c, 263d
Imagealloc.free: 183a
Imagealloc.freechan: 174b
Imagealloc.ireclaim: 183a
Imagealloc.nfreechan: 174b
Imagealloc.szfreechan: 174b
imagechanreclaim(): 175b, 183b
imageinit(): 250b, 263c
imagmem: 84a, 84a, 186a, 254d, 610b
Imask-2667: 514c, 514e, 752c
implement(): 489a, 490c
inccnt(): 70g, 70i
Incr-1747:
incref(): 69b, 103b, 111d, 117b, 119b, 119d, 120b, 120c, 122c, 139c, 171e, 175b, 177a, 177c, 276c, 280c, 283d, 300c, 301a, 304b, 307b, 308d, 318a, 319a, 320f, 327b, 328b, 328c, 330a, 330c, 368d, 419d, 458, 554, 555, 584d, 659, 786b, 789a
Indexshift-1664: 762b, 766c
initcode: 265d
Initfreq-1617: 762a, 766b
Inner:
Inner.idev:
Inner.iname:
Inner.isize:
Inner (typedef):
Input-2187: 835d
Ins-982: 348a, 349c, 352b
Int-2307: 830b, 832e
int2flag(): 325e, 326a
intellimouse: 656, 656
Inten-2321: 830b
interio():
Interrupt-1634: 762b, 765b, 766a, 766c, 769a
intrcount: 629c, 629d
intrcpushutdown(): 595c, 597a, 598a
Intregs: 157d, 750b
INTREGS-2403: 31g, 158a, 257a
Intregs.ARMdisable: 157d
Intregs.ARMenable: 157d

Intregs.ARMpending: 157d
Intregs.FIQctl: 157d
Intregs.GPUdisable: 157d
Intregs.GPUenable: 157d
Intregs.GPUpending: 157d
Intregs (typedef): 750b
intrshutdown(): 596b, 597a
intrsoff(): 256e, 257a, 597a
Intstatus-2298: 830b
iolist-1807: 188, 189b, 195d, 262b, 263a
ioptr-1808: 188, 189c, 189d, 192, 195d, 611c
IOSIZE: 31d, 249c
iprint: 82b, 152c, 152e, 153a, 214b, 230e, 239a, 240, 408a, 450b, 493c, 525c, 526c, 535b, 595e, 596b, 613b, 614a, 616a, 630b, 633c, 814a
iprintcanlock(): 363e, 364b
iprintlock-1779: 363e, 364a
iprintscreenputs: 363d, 363d, 363e
Irpten-1636: 762b, 766a, 766c, 769a
Irptmask-1635: 762b, 766b
IRQ: 156b
irq(): 159a, 159b
Irq-2204: 835e, 837e
IRQaux: 156b
IRQbasic: 158c, 158c, 158d
IRQclock: 156b, 159b
IRQcntpns: 159b, 519
IRQcntps: 159b, 519
IRQDMA: 829
IRQdma0: 156b
irqenable(): 158a
IRQfiq: 528b, 528c
IRQi2c: 156b
irqintr(): 822b
IRQlocal: 519, 519, 520b, 520c
IRQmmc: 156b
IRQspi: 156b
IRQtimer0: 156b
IRQtimer1: 156b
IRQtimer2: 156b
IRQtimer3: 156b, 156b
IRQtimerArm: 158c
IRQusb: 156b, 528b, 827b
Irstats: 628j, 629a
irstats-1806: 629a
Irstats.calls: 628j
Irstats.loops: 628j
Irstats.maxt: 628j
Irstats.ticks: 628j
ISAConf:

ISAConf (typedef):
ISCPOP-2590: 493a
Isdata-1668: 762b, 764a, 766c
isdotdot(): 302a, 302b, 306b, 330c
iseve(): 114d, 196c, 199e, 199f, 242d, 243c, 456a, 557b, 559, 594a, 651, 653, 659, 673b
isfrog: 286
isoverlap(): 437, 438a
isqtype(): 785a, 790, 795, 796b, 798b, 804, 805
Isread-1753:
Istatus-2668: 752c
isvalidaddr(): 614a, 614b
ISVFPOP-2591: 493b, 493c
Iswrite-1752:
iunlock(): 62b, 77d, 78c, 82b, 84c, 227d, 228a, 228c, 229d, 231a, 238c, 238d, 239a, 239b, 240, 354b, 373, 374, 375, 376, 377, 378, 379, 381a, 382c, 383, 384, 385b, 387, 389b, 389c, 390b, 391d, 408a, 408b, 535b, 617e, 811a
Ixchken-1669: 762b, 764a
ixsummary(): 609c, 611d
KB: 83a, 261c, 704a
kbd-1778: 354a, 354a, 354b, 354c, 361, 368b, 368c, 368d, 368e, 368f, 594d, 594e, 621a
kbdbuttons: 659, 659
kbdcr2nl(): 837e
kbdgetmap(): 356d, 653
kbdmouse: 369d, 369e, 659, 659
kbdputc(): 350d, 354b, 608b, 639d
kbdputcclock(): 354c, 355a
kbdputmap(): 357, 653
kbdputsc(): 350d, 651
kbdq: 355b, 360b, 360c, 361, 368b, 368f, 838a
kbdqinit(): 360c
KbExt: 350b, 350d
kbinattach(): 651, 651
kbinbusy: 651, 651
kbinclose(): 651, 651
kbindevtab: 651
kbinlck: 651, 651
kbinopen(): 651, 651
kbinread(): 651, 651
kbinstat(): 651, 651
KbInt: 350b, 350d
kbintab: 651, 651
kbinwalk(): 651, 651
kbinwrite(): 651, 651
KBLINELEN-997: 653, 653
kbmapattach(): 653, 653
kbmapclose(): 653, 653
kbmapdevtab: 653
kbmapopen(): 653, 653
kbmapread(): 653, 653
kbmapstat(): 653, 653

kbmaptab: 653, 653
kbmapwalk(): 653, 653
kbmapwrite(): 653, 653
KbNscans: 350b, 350c
Kbscan:
Kbscan (typedef):
KbscanKind: 350b
kbscans: 350c, 350d
kbtab: 348b, 350d, 356d, 357
kbtabaltgr: 349c, 350d, 356d, 357
kbtabctrl: 350d, 356d, 357
kbtabesc1: 352b, 353c, 356d, 357
kbtabshift: 349a, 350d, 356d, 357
kdebug: 608a, 608b, 608c, 608d, 608e, 609a
KF-959: 348a, 348a, 348b, 349a, 350d, 608b
KiB: 34f
kick(): 839a, 842c
kickpager(): 191d, 196c
killbig(): 192, 609c, 711c
KImage:
KImage (typedef):
KMesg: 616b, 617a
kmesg: 617a, 617e, 618a
KMesg.buf: 616b
KMesg.lk: 616b
KMesg.n: 616b
kmesgputs(): 363e, 617c, 617d, 617e
KMESGSIZE: 617b
Kmouse-971: 348a, 369d, 369e
kmousectl(): 656, 659
KNAMELEN: 267a, 564b, 571d, 692b
kprintinuse: 618f, 618g, 619a
kprintoq: 596b, 618e, 618g, 619a, 619b, 619c, 619d
kproc(): 139c, 191d, 267a, 296b, 603a
kproftimer: 627a, 627b
KQLock:
Krw: 161b, 249c, 358c, 503i
KSEG0: 36a
KSEGM: 40g, 831h
ksetenv(): 267a, 564b
kstrcpy(): 285b, 400b, 633b, 636a, 674
kstrdup(): 112d, 114c, 114d, 115a, 117a, 126d, 139c, 265d, 287d, 319a, 549c, 558b, 579, 592g, 636b, 793b, 800
L1: 249a, 250b
L1AP: 161e, 249c, 358c, 503i
L1hi-1848: 162b, 163e, 498b, 498c
L1lo-1847: 162a, 163e, 498b, 498c
L1ptedramattrs-1845: 249c, 497a
L1sharable: 502e, 502f
L1SIZE: 507a, 507b

L1wralloc: 502e, 502f
L1X-1842: 160a, 167, 249c, 259c, 358c, 498b, 498c, 498d, 503i
L2: 249b, 249c
l2ap(): 161j
L2AP-1844: 161i, 167, 249c
l2be(): 125a
L2ptedramattrs-1846: 167, 249c, 497b
L2sharable: 502e, 502f
L2wralloc: 502e, 502f
L2X-1843: 160b, 167, 249c, 498g
Label: 205d
Label.pc: 205d
Label.sp: 205d
Label (typedef):
Large: 160f
latin1(): 639d, 649
launchinit(): 506g, 507a
lck-1763:
le2long(): 244b, 245e
le2vlong(): 244b, 245b
Left-977: 348a, 349c, 352b
legacytopctl():
Len-2575: 488b
Lev0-2197: 835d, 837d
lineq: 360d, 360e, 361
lineqinit(): 250b, 360e
Lite-2322: 830b
localclockintr(): 514e
Localctl-2650: 514a, 514c
Localintpending-2410: 520b, 520e
Localmboxint-2409: 520f, 598a
LOCALREGS-2404: 520b, 520g, 598a
Localtimerint-2408: 520b, 520d, 598a
Lock:
lock(): 58d, 63f, 64c, 65e, 66b, 67b, 67d, 68, 69b, 69c, 92d, 97a, 98b, 110f, 111a, 111b, 111d, 113a, 113b, 132c, 133a, 134c, 136d, 137a, 138d, 158a, 166b, 171e, 172a, 172d, 173, 175b, 180c, 182b, 182d, 183b, 185e, 187b, 187e, 188, 194a, 194b, 195d, 201b, 212e, 214a, 253a, 273d, 274a, 276c, 277c, 280c, 281b, 291, 292c, 293b, 296b, 296d, 341c, 364c, 424, 426b, 428f, 443c, 443d, 444a, 448f, 469a, 469b, 508b, 524a, 524c, 525c, 535b, 548b, 548c, 549c, 553, 566b, 566c, 582b, 584d, 585a, 585b, 588b, 595e, 651, 659, 737c
Lock (typedef):
lockloop(): 58d, 613a
lockstats: 61d, 628b, 628c, 628d, 628e
Log:
Log (typedef):
logdump(): 813b, 815d, 815e
Logflag:
Logflag (typedef):
logstart(): 815b, 815e
lookpage(): 172a, 178

lookupcmd(): 561b, 586b, 594a, 656, 659, 800, 803
lookuptree():
lprint:
LRESPROF: 94c, 474a, 474b, 474c, 474d
machon(): 250b, 253a, 511a
MAFTER: 317b, 322d, 323e, 326a, 336a
Magichole: 75d, 77d, 78b, 80a
MAILBOX-2345: 32b, 726b, 726c
main-2434(): 250b
mainmem: 84a, 84a, 85a, 85b, 85c, 86b, 86c, 86d, 186a, 254d, 407a, 610b
main_arch_exit(): 595c
main_arch_isaconfig(): 742a
malloc(): 85a, 158a, 182d, 231a, 259b, 263c, 276c, 278b, 325c, 380a, 380b, 428f, 469a, 474b, 525c, 535b, 549c, 574c, 593b, 636b, 673c, 820b, 822c, 841b
mallocalign(): 86c, 507a
MallocOffset-1814: 87b, 698b
mallocsummary(): 407b, 609c, 610b
mallocz(): 86b, 406e
Manager: 161a
MASK: 34b, 488b, 489c, 490c, 493c, 511d, 722b
mattach():
Maxatomic-779: 372b, 387, 689a, 689a
MAXCACHE-758:
maxcache-764:
Maxchans: 772c, 772d
MAXCONF: 47a, 47b, 759a, 759c
Maxconf-1756:
Maxenvsize-786: 674, 674
Maxfpregs: 481i
MaxPeriod-2653: 229a, 229b
Maxretries-1759:
MAXRPC-782: 549c, 688c
MAXRPC-825:
Maxsteps-2594: 535b, 535b
MAXSYSARG: 145a
Maxtimerloops-2644: 230c, 230d, 230e
MB: 83a, 83b, 254d, 255, 704a
MBEFORE: 317b, 322d, 326a
Mbox: 503c, 745
Mbox.doorbell: 503c
Mbox.mbox1: 503c
Mbox.mbox2: 503c
Mbox.startcpu: 503c
Mbox (typedef): 745
Mboxes: 503d, 745
Mboxes.clr: 503d
Mboxes.set: 503d
Mboxes (typedef): 745
Mboxregs-2447: 508c, 511a, 745

Mbz: 160e
MCACHE: 326a, 337f, 338b, 554
mclose():
mconfig():
MCREATE: 317b, 323e, 326a, 334d, 335c
mdelctl():
mdeldev():
memrchr(): 289c, 635b
memsize: 41c, 254c, 254c, 255, 256a, 256b, 760a
mfreeseg(): 106d, 107, 439, 711c
mgen():
Mhead:
Mhead (typedef):
Mhz: 223e, 241b, 258, 709a, 762a, 765c
MiB: 34g, 54d, 162a, 162b, 249c, 358c, 501i
MicroFREQ-2670: 238d, 715
Middle-969: 348a
MIN: 34c, 460e, 464
Minfirmdate-2432: 264d, 264f
Minfirmrev-2431: 264d, 264e
miniphysuart: 836a, 842c
miniuart-2228: 836a, 838a, 842b
MinPeriod-2654: 229a, 229c
Miscsize:
misc_constants_portable: 704a
mk64fract(): 238d, 242a
mkqid(): 53b
MMASK: 317b, 322d
mmcinterrupt(): 766a
MMCSelect-1620: 762a, 766c
mmuinit(): 249c
mmuinit1(): 250b, 259c, 511a
mmukmap(): 358b, 358c
mmul1empty(): 162f, 163e, 209b
mmul2empty(): 162f, 163a, 209b
Mnt:
Mnt (typedef):
Mntalloc: 547b, 547c
mntalloc: 547c, 548a, 548b, 548c, 549c
Mntalloc.id: 547b
Mntalloc.list: 547b
Mntalloc.mntfree: 547b
Mntalloc.nrpcfree: 547b
Mntalloc.nrpcused: 547b
Mntalloc.rpcfree: 547b
Mntalloc.tagmask: 547b
mntattach(): 554
mntauth(): 558a
Mntcache:

Mntcache (typedef):
mntchan(): 554
mntchk(): 555
mntclose():
mntclunk():
mntcreate():
mntdevtab:
mntdirfix():
mntflushalloc(): 553
mntflushfree(): 553
mntfree(): 548b, 548d, 554, 555
mntgate(): 553
MNTHASH: 320b, 321a, 326b, 341c
MNTLOG: 320b, 320b
mntopen():
mntopencreate():
mntpntfree(): 548c, 548d
mntqrm():
mntralloc(): 554, 555
mntrdwr():
mntread():
mntremove():
mntreset():
Mntrpc:
Mntrpc (typedef):
mntrpcread(): 553
mntscan(): 325e, 326b
mntstat():
mntstats:
mntversion(): 549c, 554, 557e
Mntwalk:
mntwalk(): 555
Mntwalk (typedef):
mntwrite():
mntwstat():
modemctl(): 840c
Modref: 90a
mopen():
MORDERMASK: 317b, 322d, 323e
Moribund: 134c, 135a, 136a, 207c, 483a
Mount:
Mount (typedef):
mountfix(): 332f, 344b
MountFlags: 317b
mountfree(): 319c, 321a, 323e, 339c
MOUNTH: 320b, 323e, 328b, 339c
mountid-2606: 319a, 340b, 341c
mountio(): 552, 553
mountmux(): 553

mountrewind(): 334a, 344a
mountrock(): 342d, 344b
mountrockread(): 332f, 343
mountrpc(): 552, 554, 555
mouse: 659, 659
mouseaccelerate(): 656, 659
mouseattach(): 659, 659
mousechanged(): 659, 659
mouseclock(): 659, 659
mouseclose(): 659, 659
mousecreate(): 659, 659
mousectlmsg-939: 656, 656
mousectlmsg-954: 659, 659
mousectlqlock-929: 656, 656
mousedevgen(): 659, 659
mousedevtab: 659
mousedir-949: 659, 659
mousefromkbd(): 659, 659
mousehwaccel: 656, 656
Mouseinfo: 659, 659
Mouseinfo.acceleration: 659
Mouseinfo.dx: 659
Mouseinfo.dy: 659
Mouseinfo.lastcounter: 659
Mouseinfo.lastresize: 659
Mouseinfo.maxacc: 659
Mouseinfo.open: 659
Mouseinfo.qfull: 659
Mouseinfo.queue: 659
Mouseinfo.r: 659
Mouseinfo.redraw: 659
Mouseinfo.resize: 659
Mouseinfo.ri: 659
Mouseinfo.track: 659
Mouseinfo.wi: 659
Mouseinfo (typedef): 659
mouseinit(): 659, 659
mouseopen(): 659, 659
Mouseother: 647d
MousePS2: 647d, 656
mouseread(): 659, 659
mousereset(): 659, 659
mouseresize(): 659
Mouseserial: 647d
mouseshifted: 369b, 369c, 608c, 659
mousestat(): 659, 659
Mousestate: 659, 659
Mousestate.buttons: 659
Mousestate.counter: 659

Mousestate.msec: 659
Mousestate.xy: 659
Mousestate (typedef): 659
mouseswap-956: 659, 659
mousetime-958: 659, 659
mousetrack(): 659, 659
MouseType: 647d
mousetype: 656, 656
mousewalk(): 659, 659
mousewrite(): 659, 659
mousexy(): 659
mread():
MREPL: 317b, 323e, 336a
MS2NS: 222c, 233a
ms2tk(): 223c, 235a
msize(): 406e, 407a
mstat():
MuBaud-2227: 835e, 838b, 839c
MuCntl-2223: 835e, 838b, 838c, 840c
MuIer-2209: 835e, 838b, 838c, 839a
MuIir-2212: 835e, 838b
MuIo-2208: 835e, 837e, 839a, 841d, 842a
MuLcr-2213: 835e, 838b, 839d
MuLsr-2219: 835e, 837e, 839a, 841d, 842a
Multiblock-1676: 762b, 764a
multitrans(): 817, 818a, 818b
MuMcr-2217: 835e, 841a
Mustexist-1755:
muxclose(): 342a, 548d
Mvfr0-2566: 488a
Mvfr1-2565: 488a
mwalk():
mwrite():
N-2585: 492b, 492c
name2mode(): 784c, 800
name2speed(): 784a, 800
name2ttype(): 784b, 800
namec(): 267a, 279d, 283d, 307c, 310d, 311a, 314b, 315b, 322d, 338c, 426a, 564b
nameerror(): 290a, 316
namelenerror(): 289b, 289c, 290a
Nbootfiles-829: 394a, 397
NBROKEN-2462: 451b, 451b, 451e, 452a
Nchan-2285: 830b, 831c
nchanlog-2239: 815a, 815b, 815c, 815d
Ncmd-1608:
ncmdfield(): 560c, 562
nconf: 47c, 267a, 758g, 759a
NCONT: 419a, 423b
Nctlqid:

Ndataqid:
NDebug: 101c, 147b, 152e, 155b, 155d, 164b, 179, 418c, 421b, 423b, 450a, 450b, 493c, 494b, 530c
NDev-1605:
Ndevs-1757:
NDFLT: 419a, 423b
needpages(): 192, 193a
nelem: 42a, 99c, 110f, 111a, 111b, 254d, 260, 261c, 365b, 366a, 456a, 488d, 524a, 580d, 586a, 586b, 589e, 594a, 639d, 643, 651, 656, 659, 672a, 672b, 672c, 692c, 760b, 766c, 784a, 784b, 784c, 785c, 790, 800, 803
NERR: 147d, 633a, 633b
NETID: 429
NETQID: 428f, 429
NETTYPE: 430a, 430b, 431a, 431b, 432, 433a, 433b, 434a, 434b
Nevents-905: 468f, 468g, 468h, 469a, 469c
newchan(): 273d, 275a, 400a
newdev(): 788b, 793b, 800
newdevep(): 789a, 800
newfd(): 277c, 279d, 307c, 558a, 588b
newfd2(): 426a, 426b
newmhead(): 318a, 323e, 341c
newmount(): 319a, 323e, 336a, 341c
newpath(): 267a, 275a, 300c
newpgrp(): 119d, 121e, 139c, 265d, 320f
newproc(): 111d, 116b, 139c, 265d
newrgrp(): 120c, 122b, 265d, 448c
newseg(): 97b, 103c, 103f, 103g, 175b, 177c, 265d, 437, 458
newswap(): 185e, 188
newusbid(): 788a, 788b
NExit: 141c, 418c
Nextconbk-2296: 830b
NEXTENT-760:
nexterror: 102b, 117d, 195d, 233a, 279d, 283d, 289b, 289c, 291, 292c, 293b, 307c, 308d, 310d, 314b, 314c, 316, 322d, 323e, 325e, 334d, 337f, 338c, 361, 372b, 383, 384, 385b, 405a, 405b, 426a, 431a, 434a, 434b, 438b, 445a, 445b, 454e, 457, 469a, 549c, 553, 554, 555, 557e, 558a, 560c, 566b, 567a, 571d, 574c, 577, 579, 582b, 584d, 586b, 588b, 593b, 594a, 603b, 626a, 633b, 656, 659, 673c, 674, 759b, 769a, 785c, 787b, 790, 795, 796a, 796b, 796c, 798b, 800, 803, 804, 805, 806a, 806b, 818a, 818b, 823b, 824
NFILE-759:
Nfpctlregs: 481i, 486a
NFrame: 421a, 757b
NFrame.arg0: 421a
NFrame.arg1: 421a
NFrame.ip: 421a
NFrame.msg: 421a
NFrame.old: 421a
NFrame.ureg: 421a
NFrame (typedef): 757b
NFREECHAN: 174d, 182d, 263c
NHASH-757:
Nhole: 260, 610a, 694
nilptr: 64c, 80c, 105b, 112b, 166b, 693c

NISAOPT: 742a
NKBFILE-996: 653, 653
NMASK: 547b, 547d
NNOTE: 420d, 424
No-972: 348a, 348b, 349a, 349c, 352b
Noaccess: 161b
noblockcnt-777: 385b
nonone(): 577, 579, 582a
noprint: 363b, 363b
noprocpanic(): 113a, 113b
Note:
Note (typedef):
NoteHook: 419a
NOTEID-914: 570a, 574c
noteidalloc: 122c, 419b, 419d
NoteKind: 418c
notempty(): 380c, 381a
now-2595: 535b, 535b
Npadlong-1813: 85a, 85b, 85c, 86b, 86c, 86d, 87a, 87b, 87c, 87d, 88a, 88b, 88c, 88d, 407a, 698b
NPart-1597:
npath: 300c, 301a, 301b
NPIPEDIR-858: 430a, 433a, 679
Npriq: 198a, 199d, 200a, 220b, 531a
nrand(): 601c
nrdy: 201a, 201b, 201c, 535b, 611f, 624b
NRegs-2354: 724c, 726b
Nrootfiles-828: 394a, 394c
Nrq: 200a, 200b, 204, 531b, 611f
NRSTR: 419a, 422, 423b
ns2fastticks(): 226f, 241d
NSAVE: 419a, 423b
Nscan-985: 348a, 348b, 349a, 349c, 352b, 356d, 357
NSEG: 100a, 100c, 104b, 106b, 117d, 138a, 192, 194a, 436b, 437, 438a, 438b, 440c, 458, 526c, 571c, 571d, 699, 711c
Nstatqid:
nsyscall: 42a, 42a, 147b, 460e
Ntrees-1758:
ntrees-1766:
NType-1593:
Num-968: 348a, 348b, 349a, 350d
NUMSIZE: 242e, 366a, 472b, 472c, 571d, 600b, 626a, 653, 704a
NUnit-1601:
NUser: 236c, 418c, 421b, 424, 434a, 434b, 574c
Nvec-2406: 256c, 256d
oargb-2440: 758c, 760b, 761a
oargblen-2441: 758d, 760b, 761a
oargc-2438: 758a, 760b, 761a
oargv-2439: 758b, 760b, 761a
OCEXEC: 280a, 283d, 297a, 297c, 297d, 308d

OEXCL: 307c, 308d, 310b
OEXEC: 280a, 282d, 300a
Off-2199: 835d, 836d
offsetof: 77d, 78a, 79, 692d, 760a
offsetof_name_direntry(): 313b, 313c, 316, 344b
okaddr(): 101a, 101c, 421b, 423b
okay(): 769a, 843a
OkLed-2182: 835a, 843a
OK_0: 105e, 106d, 166b, 359a
OK_1:
Onemicrosecond-2600: 535b
Onemillisecond-2601: 535b, 535b
OneRound-2603: 535b, 535b
Onesecond-2602: 535b, 535b
onswap: 187c, 187d, 187f, 196b
OpenFlags: 279c
openmode(): 279d, 280a, 307c, 404a, 431b, 469a, 577, 659, 674, 795
Opmask-2384: 722b, 722c
Optional-1754:
options-1614:
optionsinit(): 760b
ORCLOSE: 280a, 298c, 298e, 298f
order-1800: 245d, 245d, 245e
ORDWR: 279c, 280c, 337f, 364c, 426a, 557e, 558a, 785c, 788b
OREAD: 279c, 280a, 280c, 291, 325c, 333b, 338c, 404a, 431b, 469a, 577, 584d, 659, 674, 785c, 789b, 795, 805, 818a
OTRUNC: 280a, 280c, 308d, 310a, 674
Output-2188: 835d, 843a
OWRITE: 279c, 292c, 364c, 564b, 577, 785c, 789b, 795, 798b, 818a
p:
packblock(): 413a
packetsize: 656, 656
padblock(): 410
padblockcnt-770: 410, 611d
Page:
Page (typedef):
pagechainhead(): 92d, 93a, 163d
pagechaintail(): 93b, 172b, 180c
pagedout: 177e, 177f, 178, 189d, 457, 526c, 694, 699, 703b
pageinit(): 250b, 261c
pageiocomp(): 195d, 196a
pagenumber(): 526c, 527a
PageOrSwap (typedef): 186c
pageout(): 189d, 192
pagepte(): 188, 189d
pager(): 191d, 192
pagersummary(): 609c, 611c
Pagetable:
Pagetable (typedef):

pageunchain(): 92c, 172a, 180c
Palloc: 90d
palloc: 90d, 92c, 92d, 93a, 93b, 134c, 136a, 163d, 171e, 172a, 172d, 173, 180c, 186a, 191b, 191c, 192, 193a, 207c, 260, 261c, 262a, 525c, 527a, 611c
Palloc (typedef):
Pallocmem:
Pallocmem (typedef):
panic:
panicking: 535b, 630a, 630b
parity(): 840b, 842c
parseboothdr:
parsecmd(): 560c, 586b, 594a, 659, 800, 803
parseconfig():
parseirq():
parsename(): 287d
parseport():
parsesize():
parsespec():
parseswitch():
parsetime(): 532b, 586a
parsetype():
PART-1610:
PartLOG-1596:
PartMASK-1598:
PartSHIFT-1599:
Password-2453: 770c, 771b, 771c, 771d
Path:
path2dev():
path2devno():
path2treeno():
Path (typedef):
pathclose(): 267a, 274a, 301b, 301c, 304b, 306c, 306d, 307a, 308d, 327b
pathlast(): 314b, 314d
PATHMSLOP-767: 329b, 329c, 330c
PATHSLOP-766: 300c, 300d, 302a
Paused-2305: 830b
Peek-2348: 724c
Perf: 623b
Perf.avg_inidle: 623b
Perf.avg_inintr: 623b
Perf.inidle: 623b
Perf.inintr: 623b
Perf.intrts: 623b
Perf.last: 623b
Perf.period: 623b
Perf (typedef):
Permapshift-2310: 830b
pexit: 133e, 140c, 141b, 175b, 179, 296d, 421b, 423b, 711b
PF-961: 348a

Pgdown-981: 348a, 349c, 352b
pghash: 90b, 171e, 172a, 172d, 173
PGHLOG: 90b
PGHSIZE: 90b
PGROUND: 101d, 102a, 104d, 254d, 261c, 437, 439
Pgrp:
Pgrp (typedef):
pgrpcpy(): 119d, 121e, 341c
pgrpid-2605: 320e, 320f
pgrpinsert(): 341b, 341c
pgrpnote(): 574c, 576
Pgup-975: 348a, 349c, 352b
PG_MOD: 90a, 166b, 524d
PG_NEWCOL: 521f
PG_NOFLUSH: 521f, 522c
PG_NOTHING: 90a
PG_ONSWAP: 188
PG_REF: 90a, 166b, 170b, 189d, 524d
PG_TXTFLUSH: 521f, 522c, 522d, 522l, 522m, 703b
Phdr32sz-2417:
Phdr64sz-2420:
PHYSDRAM: 40f
Physseg:
physseg-1804: 437, 523f, 524a
Physseg (typedef):
physseglock-1805: 523g, 524a
PhysUart:
PhysUart (typedef):
PID-913: 325e, 570a, 571d, 574c, 577, 579, 584d
pidalloc-2459: 110a, 111d
pidhash(): 110f, 111d
pidunhash(): 111a, 137a
pimagmem-1841: 83b, 84a
pimagpriv-1840: 83b, 83b
pio(): 177e, 177f, 178
Pipe: 427b
Pipe.next: 427b
Pipe.path: 427b
Pipe.perm: 427b
Pipe.q: 427b
Pipe.qref: 427b
Pipe.ref: 427b
Pipe (typedef):
Pipealloc: 427c, 427d
pipealloc: 427d, 428f
Pipealloc.path: 427c
pipeattach(): 428f, 435a
pipebread(): 433b, 435a
pipebwrite(): 434b, 435a

pipeclose(): 432, 435a
pipedevtab: 435a
pipedir: 428b, 428f, 430a, 433a
pipegen(): 429, 430a, 433a
pipeinit(): 428d, 435a
pipeopen(): 431b, 435a
piperead(): 433a, 435a
pipestat(): 430b, 435a
pipewalk(): 430a, 435a
pipewrite(): 434a, 435a
pipewstat(): 431a, 435a
plan9iniinit(): 759c, 760a
plock(): 82a, 83a, 83b
pmainmem-1839: 83a, 84a
pmainpriv-1838: 83a, 83a
pnp(): 838a, 842c
poolprint(): 83a, 83b, 84b
poolsummary(): 610b, 611a
poperror: 102b, 117d, 134b, 136c, 145e, 175b, 177b, 178, 189d, 195d, 233a, 236c, 267a, 279d, 282b, 283d, 291, 292c, 293b, 296d, 306a, 307c, 308d, 310d, 314b, 314c, 316, 322d, 323e, 325e, 333b, 334d, 337f, 338a, 338c, 344b, 361, 364c, 372b, 383, 384, 385b, 400b, 405a, 405b, 426a, 431a, 434a, 434b, 438b, 445a, 445b, 454e, 457, 469a, 493c, 549c, 553, 554, 555, 557e, 558a, 560c, 566b, 567b, 571d, 574c, 576, 577, 579, 582b, 584d, 586b, 588b, 593b, 594a, 596b, 601b, 603b, 626a, 633d, 656, 659, 673c, 674, 759b, 769a, 785c, 787b, 790, 795, 796a, 796b, 796c, 798b, 800, 803, 804, 805, 806a, 806b, 818a, 818b, 823b, 824
portcountpagerefs(): 525c, 526c
portenable(): 825b, 827b
postnote: 101c, 141c, 147b, 152e, 155b, 155d, 164b, 179, 236c, 434a, 434b, 450a, 450b, 493c, 494b, 530c, 574c, 576
pow2(): 799a, 800
PowerCcp2tx: 529c
PowerI2c0: 529c
PowerI2c1: 529c
PowerI2c2: 529c
POWERREGS-2449: 31i, 771b, 771c, 771d
PowerSd: 529c
PowerSpi: 529c
PowerUart0: 529c
PowerUart1: 529c
PowerUsb: 529c, 820b
Powerwait-2367: 530a, 725c
ppanic(): 83a, 83b, 84c
pprint: 101a, 147b, 279b, 421b, 423b, 574c, 577
preempt(): 712b
preempts: 204, 629b
Prescaler-2651: 514b, 514c
prflush(): 630b, 631a
PriEdf: 531a, 531c, 535b
PriExtra: 531a, 535b
PriKproc: 139c, 198b

PriNormal: 114d, 198b, 198d, 199e, 199f, 558b, 602c
print: 42b, 60e, 62c, 62f, 64d, 70d, 71b, 71f, 78c, 113c, 147d, 165, 180c, 192, 202b, 213a, 213b, 216c, 233c, 250b, 253b, 260, 261c, 264a, 264d, 278b, 283d, 285f, 289b, 295c, 328b, 328c, 329c, 330d, 332b, 332c, 332d, 339b, 366a, 366b, 400b, 409a, 412, 430a, 434a, 489c, 490c, 511c, 525c, 533f, 535b, 552, 555, 570d, 589e, 590b, 596b, 608b, 608c, 608d, 608e, 609a, 610a, 611a, 611c, 611d, 611e, 611f, 612a, 612b, 613a, 619g, 639d, 649, 699, 711c, 765c, 766b, 766c, 769a, 786a, 787b, 788a, 792, 793a, 794d, 799b, 800, 803, 806c, 813b, 815d, 815e, 825c, 832b, 832c
Print-976: 348a, 349c, 352b
PRINTSIZE: 363a, 363b, 363e, 364c, 630b
Priority: 198b
PriRelease: 531a
PriRoot: 199g
Private: 81, 697c
Private.lk: 81
Private.msg: 81
Private (typedef): 697c
Proc:
Proc (typedef):
procalarm(): 234e, 235a
Procalloc: 110b, 110c
procalloc-2458: 110c, 110f, 111a, 111b, 111d, 113a, 115a, 134c, 136a, 207c, 261a, 261b, 440c, 570b, 612a, 711c
Procalloc.arena: 110b
Procalloc.free: 110b
Procalloc.ht: 110d
procargs(): 571d, 584c
procattach(): 571a, 680
procclose(): 571a, 583
proccmd-908: 569b, 586b
Procctl: 453b
procctl(): 421b, 453d, 459c, 459e
procctlclosefiles(): 585b, 586b
procctlcloseone(): 585a, 585b
procctlmemio(): 456a, 456d, 457
procctlreq(): 574c, 586b
procdevtab: 571a
procdir: 569a, 580d
procdump(): 113b, 609c, 612a
procfdprint(): 580c, 582b
procfds(): 571d, 582b
procflushseg(): 440a, 440b, 440c
procgen(): 571d, 577, 580d, 680
procindex(): 111b, 580d
procinit(): 250b, 261a
procoffset(): 580a, 582b
procopen(): 571a, 577
procpriority(): 139c, 198d, 199a, 199e, 199f
procqidwidth(): 580b, 580c, 582b
procread(): 571a, 571d

Procseg: 100a
procstat(): 571a, 680
Procstate: 51b
procstopped(): 454e, 455a
procstopwait(): 453e, 454e, 455h, 459b, 470c
proctab: 192, 194a, 216g, 526c, 535b, 571d, 574c, 576, 577, 579, 580d
proctext(): 577, 584d
Proctime: 471a
proctrace: 467a, 467a, 467e, 467g, 467i, 468b, 468c, 469a, 469b, 535b
procwalk(): 571a, 680
procwired(): 216f, 216g, 217b, 596b
procwrite(): 571a, 574c
procwstat(): 571a, 579
Proc_exitbig: 711a, 711b, 711c
Proc_exitme: 141a, 141b, 141c, 295b, 295c
Proc_nothing: 141d, 453b, 453d, 454e, 455h
Proc_stopme: 453b, 453d, 453e, 459c, 459e
Proc_traceme: 470a, 470b, 470c
Proc_tracesyscall: 459a, 459b, 459c, 459e
proc_dumpaproc(): 612b
proc_error(): 633b
proc_init(): 570d, 571a
proc_nexterror(): 634a
proc_pexit(): 134c
proc_postnote(): 424
proc_proctab(): 570b
proc_ready(): 211a
proc_sched(): 202c
proc_sleep(): 212e
proc_tsleep(): 233a
proc_wakeup(): 214a
producecnt-775: 378, 611d
profclock(): 473g, 473h
Prophdr: 726a, 729
Prophdr.data: 726a
Prophdr.len: 726a
Prophdr.req: 726a
Prophdr.tag: 726a
Prophdr.tagbuflen: 726a
Prophdr.taglen: 726a
Prophdr (typedef): 729
ps2mouse(): 656, 656
PsrDfiq: 423b, 527d
PsrDirq: 28b, 423b
PsrMabt: 143a, 150b, 153c, 154a, 257b
PsrMask: 28a, 143b, 147h, 150b, 423b, 613b, 614c
PsrMfiq: 527b, 527f
PsrMirq: 143a, 159a, 257b
PsrMsvc: 28a, 613b

PsrMund: 143a, 152e, 257b
PsrMusr: 28a, 147h, 150b
ptalloc(): 96c, 97a, 99e, 166b
ptcpy(): 97a, 103e
PTE (typedef): 161l
PTEGLOBAL: 174a
PTERONLY: 166a, 166b, 170b
PTESIZE: 174a
PTEUNCACHED: 524d, 525a, 525b
PTEUSER: 694
PTEVALID: 166a, 166b, 170b, 524d
PTEWRITE: 166a, 166b, 167, 524d
ptflush(): 441c, 703b
PTR2UINT: 41a, 140b, 254d, 265d, 421b, 423b, 574b, 726d, 761a, 831h, 832a
PUD-2198: 835d, 836c
PUDclk0-2202: 835d, 836c
PUDclk1-2203: 835d
Pufv (typedef): 724b
pullblock(): 415b
Pulldown-2200: 835d, 837b
Pullup-2201: 835d, 837a
pullupblock(): 372a, 412
pullupblockcnt-772: 412, 611d
pullupqueue(): 372a
punlock(): 82b, 83a, 83b
PUT2: 797
putc(): 841d, 842c
putep(): 787a, 787a, 787b, 790, 795, 796b, 796c, 798b, 800, 803, 804, 805
putimage(): 172d, 173, 182c, 182d
putmhead(): 304b, 306c, 306d, 307a, 308d, 318b, 321a, 328a, 328b, 331c, 332a, 332b, 332c, 335a, 339c, 344b
putpage(): 92d, 96d, 107, 170c, 178, 188, 195d
putseg(): 98b, 138a, 189d, 438b, 458
putstrn(): 363b, 363c, 630b
putstrn0(): 362a, 362b, 363c
putswap(): 178, 187b, 187c, 187d
Pvfu (typedef): 724b
pwait(): 132a, 132c
qaddlist(): 381b
Qargs-861: 568a, 569a, 571d, 574c, 577
Qbase-850: 736c, 737b
Qbintime-1783: 243b, 243c, 600a, 600b
Qboot-827: 394a, 394c, 396a, 396b, 397
qbread(): 383, 433b
qbwrite(): 372b, 385b, 434b
qbypass(): 380b
qcanread(): 361, 368b, 391a
qclose(): 389a, 389b, 432
Qclosed: 371a, 376, 377, 380c, 381a, 385a, 385b, 389b, 389c, 390a, 390b
Qcoalesce: 371a, 384, 618g

Qconfig-838: 591e, 591f, 591g
Qcons-1781: 359c, 360a, 361, 362a
Qconsctl-1782: 359c, 360a, 368d, 368e, 368f
qconsume(): 375
qcopy(): 379
qcopycnt-776: 379, 611d
Qcputime-1784: 472b, 600a, 600b
Qctl-1589:
Qctl-1750:
Qctl-2242: 781, 782e, 790, 796c, 798b, 804, 805
Qctl-862: 568a, 569a, 574c, 577
Qcursor-945: 659, 659
Qdata-995: 653, 653
Qdata0-856: 428a, 428b, 428f, 429, 430a, 430b, 431b, 432, 433a, 433b, 434a, 434b
Qdata1-857: 428a, 428b, 429, 430a, 430b, 431b, 432, 433a, 433b, 434a, 434b
Qdir-1749:
Qdir-1780: 359c, 360a, 366a
Qdir-2240: 781, 790
Qdir-826: 394a, 394c, 396a, 396b
Qdir-836: 589a, 589b, 589e
Qdir-849: 736c, 736f, 738d
Qdir-855: 428a, 428b, 428f, 430b, 431a, 433a
Qdir-859: 568a, 580d
Qdir-944: 659, 659
Qdir-992: 651, 651
Qdir-994: 653, 653
qdiscard(): 374
Qdrivers-842: 592h, 593a, 593b
Qep0ctl-2245: 781, 790
Qep0dir-2243: 781, 784d, 785a, 790, 795, 796b
Qep0dummy-2246: 781
Qep0io-2244: 781, 790
Qepctl-2249: 781, 790, 795, 796b, 798b, 804, 805
Qepdir-2247: 781, 790, 795, 796b
Qepio-2248: 781
Qfd-863: 568a, 569a, 571d, 577
Qfirst-1751:
Qflow: 371a, 373, 374, 375, 376, 377, 378, 382c, 385b, 389b, 392
qflush(): 391d
Qfpregs-864: 495b, 495c, 568a, 569a, 577
qfree(): 548c
qfull(): 392
qget(): 373
qhangup(): 389c, 432, 619a
Qhostdomain-840: 591h, 591i, 592a, 592b
Qhostowner-839: 591h, 591i, 592a, 592b
Qid:
QID-1613:
QID-2271: 782b, 790, 795, 796b, 796c, 798b, 804, 805

QID-911: 325d, 469a, 469b, 469c, 570a, 571d, 574c, 577, 579, 580d
qid2epidx(): 784d, 790, 795, 796b, 796c, 798b, 804, 805
Qid (typedef):
qidpath-785:
Qidtype: 52c
qidvers-1767:
qiomaxatomic: 431b, 618g, 689a, 689a
qisclosed(): 390a, 619c
qiwrite(): 387, 619c, 620d, 621a
Qkbd-993: 651, 651
Qkick: 371a, 385b
Qkmesg-1794: 617f, 617g, 618a
Qkprint-1795: 618b, 618c, 618g, 619a, 619b
Qkregs-865: 568a, 569a, 574a, 577
qlen(): 390c, 429, 430b, 620b
qlock(): 63f, 98b, 100c, 102b, 104d, 117d, 138a, 138c, 178, 235a, 236c, 296d, 321a, 325e, 333b, 334b, 342d, 343, 361, 383, 384, 385b, 421b, 423b, 424, 430a, 431b, 432, 438b, 451e, 452a, 452b, 454c, 454e, 455b, 458, 535b, 549c, 571d, 574c, 576, 577, 579, 582b, 585a, 603b, 656, 659, 699, 785c, 786a, 786b, 787a, 788a, 795, 796a, 800, 811b, 811c, 815e, 823b, 824
QLock (typedef):
QlockStats: 628f, 628g
QlockStats.qlock: 628f
QlockStats.qlockq: 628f
QlockStats.rlock: 628f
QlockStats.rlockq: 628f
QlockStats.wlock: 628f
QlockStats.wlockq: 628f
Qmax-851: 736c, 736d, 736e, 736f, 737c
Qmem-866: 456a, 456d, 568a, 569a, 577
Qmouse-946: 659, 659
Qmousectl-948: 659, 659
Qmousein-947: 659, 659
Qmsg: 371a, 372b, 375, 383, 384, 387
qnoblock(): 360c, 360e, 391c, 618g
Qnote-867: 568a, 569a, 571d, 574c, 577
Qnoteid-868: 568a, 569a, 571d, 574c, 577
Qnotepg-869: 568a, 569a, 574c, 577
qnotfull(): 385a, 385b
Qns-870: 325c, 325d, 325e, 568a, 569a
Qnull-1785: 366a, 366b, 600a, 600b
qopen(): 360c, 360e, 380a, 428f, 549c, 618g, 838a
Qosversion-837: 591a, 591b, 591c
Qpart-1591:
qpass(): 376
qpassnolim(): 377
Qpgrpid-1786: 366a, 600a, 600b
Qpid-1787: 366a, 600a, 600b
Qppid-1788: 366a, 600a, 600b
Qproc-871: 568a, 569a, 571d, 577

qproduce(): 355b, 378
Qprofile-877: 474c, 474d, 568a, 569a, 577
qputback(): 382b, 383, 384
Qrandom-1789: 600a, 600b, 604
Qraw-1590:
qread(): 361, 384, 433a, 619b
Qreboot-843: 593c, 593d, 594a
Qregs-872: 568a, 569a, 571d, 574c, 577
qremove(): 382a, 383, 384
qreopen(): 390b, 432, 618g
qschedulability-2599: 535b, 535b
Qsegment-873: 104b, 568a, 569a, 577
qsetlimit(): 391b
QSHIFT-910: 570a, 570d, 580d
Qstarve: 371a, 373, 375, 376, 377, 378, 380a, 381a, 385b, 387, 389b, 390b
Qstatus-874: 568a, 569a, 571d, 577
Qswap-1790: 186a, 196c, 600a, 600b
Qsyscall-878: 460d, 568a, 569a, 577
Qsysname-841: 592d, 592e, 592f, 592g
Qsysstat-844: 625d, 625e, 626a, 626b
QTAPPEND: 416b
QTAUTH: 416b
QTDIR: 52c, 275a, 292c, 299b, 299c, 299d, 300a, 306c, 307a, 311b, 323e, 332f, 360a, 394c, 395a, 396a, 397, 400b, 402, 404a, 428b, 428f, 431b, 571d, 574c, 577, 579, 580d, 589b, 651, 653, 659, 674, 736f, 790, 798b, 805
QTEXCL: 416b, 618c
Qtext-875: 568a, 569a, 577
QTFILE: 52c, 182d, 429, 580d, 790
Qtime-1791: 242b, 242d, 600a, 600b
QTMOUNT: 398b, 398c
Qtop-1748:
Qtopbase-1585:
Qtopctl-1586:
Qtopdir-1584:
Qtrace-860: 469a, 469b, 469c, 568a, 579, 580d
Queue:
Queue (typedef):
Queueing: 63e, 63f
QueueingR: 65d, 65e, 67d
QueueingW: 66b, 67a, 67b, 67d
queueproc(): 201b, 211a, 220b
QueueState: 371a
Qunitbase-1588:
Qunitdir-1587:
qunlock(): 64c, 98b, 100c, 102b, 105c, 105e, 106b, 106c, 106d, 117d, 138a, 138c, 165, 166b, 177b, 178, 183b, 189d, 192, 235a, 236c, 296d, 321a, 325e, 333b, 334b, 342d, 343, 361, 383, 384, 385b, 421b, 423b, 424, 430a, 431b, 432, 438b, 439, 441c, 451e, 452a, 452b, 454c, 454e, 455b, 458, 535b, 549c, 566b, 567a, 567b, 571d, 574c, 576, 577, 579, 582b, 585a, 603b, 656, 659, 699, 711c, 785c, 786a, 786b, 787a, 788a, 795, 796a, 800, 811b, 811c, 815e, 823b, 824
Qusbdir-2241: 781, 790

Quser-1792: 366a, 366b, 600a, 600b
qwait(): 381a, 383, 384
Qwait-876: 566b, 568a, 569a, 577, 580d
qwakeup_iunlock(): 382c, 383, 384
qwindow(): 390d
qwrite(): 361, 368f, 372b, 434a, 619c, 621a
Qzero-1793: 366a, 600a, 600b
randn-1798: 601a, 601b, 601c
randomclock(): 602d, 603a
randominit(): 600c, 603a
randomread(): 601b, 603b, 604
Rawcmd-1581:
Rawdata-1582:
Rawstatus-1583:
Rb: 601d, 601e
rb: 601e, 602a, 602b, 602c, 602d, 603a, 603b
Rb.bits: 601d
Rb.buf: 601d
Rb.consumer: 601d
Rb.ep: 601d
Rb.next: 601d
Rb.producer: 601d
Rb.randn: 601d
Rb.randomcount: 601d
Rb.rp: 601d
Rb.wakeme: 601d
Rb.wp: 601d
rbnotempty(): 602b, 603b
rbnotfull(): 602a, 602c, 602d
Rclass: 798a
Rd2h: 798a, 818b
rdb(): 619g
read(): 290b, 291
Read-2233: 810a, 817, 818a, 818b, 822c, 823b
Read-2346: 724c, 726c
readbintime(): 243b, 244a
readelfhdr():
readfn-852: 736d, 737c, 738d
readn():
readnum(): 366a, 472b, 472c, 571d, 626a, 637c, 653
Readrdy-1693: 762b, 766c
READSTR: 593b, 704a, 796c, 841b
readstr(): 186a, 242e, 366a, 460d, 571d, 582b, 591c, 591g, 592a, 592f, 593b, 626a, 637d, 653, 739b, 739c, 796c, 841b
readtime(): 242b, 242e
Readtrans-1699: 762b
Ready: 202b, 210c, 211a, 211d, 535b
ready: 64c, 66b, 67d, 116b, 139c, 141c, 207b, 214a, 265d, 424, 448f, 451e, 452a, 452b, 454a, 459b, 470c, 535b
realloc(): 86d

ReallocOffset-1815: 87c, 698b
rebalance(): 220a, 220b
rebootmsg: 593e, 594a
Ref: 69d
Ref (typedef):
Reg (typedef): 773
Regmask-2385: 722b, 722c
REGPC-2589: 492b
Regsize-2286: 830b
release(): 535b, 535b
releaseintr(): 535b, 535b
relocateseg(): 129b
renameuser(): 114d, 115a
REND: 424, 447b, 448f
Rendez:
Rendez (typedef):
Rendezvous: 424, 448e, 448f
RENDHASH: 447b
RENDLOG: 447b, 447b
reprioritize(): 217c, 218a, 219, 220b
req:
Req-2358: 725b, 726d
reset(): 827b, 827c
Reset-2300: 830b
Resetdelay-2231: 810a
ResetdelayHS-2232: 810a, 825c
resetmouse(): 656, 656
resolution-927: 656, 656
Resp0-1626: 762b, 766c
Resp1-1627: 762b, 766c
Resp136-1673: 762b, 764a, 766c
Resp2-1628: 762b, 766c
Resp3-1629: 762b, 766c
Resp48-1674: 762b, 764a, 766c
Resp48busy-1675: 762b, 764a, 766c
Respmask-1671: 762b, 766c
Respnone-1672: 762b, 766c
resrcwait(): 113a, 113c
Resume-1666: 762b
RETinst-2383: 486a, 487a, 722b, 722c
Retrypause-1760:
returnfalse(): 85c, 86a, 113c, 192, 232c, 236c, 659, 815e, 820b, 823b, 824, 825b, 825c
RFCENVG-2556: 115b, 120a, 120b, 122a
RFCFDG-2557: 115b, 119a, 119b, 121d
RFCNAMEG-2555: 115b, 119c, 119d, 121e
RFENVG-2553: 115b, 120a, 120b, 122a
RFFDG-2554: 115b, 119a, 119b, 121d
RFMEM-2550: 115b, 117d, 121c
RFNAMEG-2552: 115b, 119c, 119d, 121e

RFNOMNT-2560: 115b, 337b, 337c
RFNOTEG-2558: 115b, 120d, 122c
RFNOWAIT-2551: 115b, 121c, 138d
Rfork: 115b
RFPROC-2549: 115b, 121b
RFREND-2559: 115b, 120c, 122b
Rgetstatus: 798a
Rgrp:
Rgrp (typedef):
Rh2d: 798a
rhubread(): 797, 798b
rhubwrite(): 798a, 805
Right-978: 348a, 349c, 352b
Rl-2598: 535b, 535b
rlock(): 65e, 326b, 328b, 331b, 333b, 334d, 341c, 564a, 673c, 674
rootattach(): 393b, 670b
rootclose(): 393b, 670a
rootdata-831: 394c, 394c
rootdevtab: 393b
rootdir-830: 394c, 394c
rootgen(): 396a, 396b, 670b
rootlist-832: 394c, 395b, 396a, 396b
rootopen(): 393b, 670b
rootread(): 393b, 396b
rootreset(): 393b, 395c
rootstat(): 393b, 670b
rootwalk(): 393b, 670b
rootwrite(): 393b, 670b
Rother: 798a
ROUND: 34a, 97b, 105e, 342d, 406e, 441c, 815e, 818b, 823b
ROUNDUP: 34e
rpcattn(): 553
Rportenable-2269: 781, 798a
Rportreset-2270: 781, 798a
Rsetfeature: 798a
RspOk-2359: 725b, 726d
Rstc-2452: 770c, 771b, 771c, 771d
Rsts-2456: 770c
rts(): 841a, 842c
RtsN-2218: 835e, 841a
Runeself: 285f, 639a
runlock(): 66b, 326b, 328b, 331b, 333b, 334d, 341c, 564a, 673c, 674
Running: 51b, 203a, 207b, 215f, 473h, 535b, 712b
runproc(): 203a, 204
runq: 194b, 200b, 201b, 201c, 202a, 204, 211a, 211d, 220b, 531b, 531c, 535b, 611f
runvec: 200e, 200f, 201b, 201c, 215d, 535b
RWlock:
RWlock (typedef):
rwstats: 65e, 67b, 68, 628g, 628h, 628i

RxEn-2226: 835e, 838b
RxIen-2210: 835e, 838b
RxPin-2184: 835c, 838b
RxRdy-2222: 835e, 837e, 842a
Sargs:
Sargs (typedef):
scale(): 659
Scaling-2461: 218d, 218e, 218g, 218h
sched: 63f, 65e, 67b, 116b, 134c, 175b, 180c, 203f, 208c, 208d, 208e, 210b, 385b, 424, 440c, 448f, 451e, 453d, 574b, 596b, 602c, 712b
scheddump(): 609c, 611f
schedgain-2460: 218f, 218h
Scheding: 111c, 111d, 204
schedinit(): 203f, 250b, 511a
Schedq:
Schedq (typedef):
screenputs: 356a, 356b, 356b, 356c, 362b, 363e, 596b
Scroll-984: 348a, 348b, 349a
ScrollDown-941: 659
ScrollLeft-942: 659
ScrollRight-943: 659
scrollswap-957: 659, 659
ScrollUp-940: 659
scrprint():
scsibio():
scsifmt10():
scsifmt16():
scsionline():
scsirio():
scsitest():
scsiverify():
sd1gen():
sd2gen():
sdadddevs():
sdaddpart():
sdattach():
sdbio():
SDbusy:
SDcheck:
sdclose():
sdconfig():
sddelpart():
sddevtab:
SDeio:
SDev:
SDev (typedef):
sdfree:
SDfreq-1618: 762a, 766c
sdgen():

sdgetdev():
sdgetunit():
SDifc:
SDifc (typedef):
sdincvers():
sdindex():
sdinitpart():
SDinq0periphqual:
SDinq0periphtype:
SDinq1removable:
SDio:
sdio: 769b
SDio (typedef):
SDmalloc:
sdmalloc:
SDmaxio:
SDnosense:
SDnostatus:
SDnpart:
SDok:
sdopen():
SDpart:
SDpart (typedef):
SDpercd:
SDperdisk:
SDperjuke:
SDperm:
SDperm (typedef):
SDpermo:
SDperpr:
SDpertape:
SDperworm:
sdread():
SDreq:
SDreq (typedef):
sdreset():
SDretry:
sdrio():
sdstat():
SDtimeout:
SDunit:
SDunit (typedef):
SDvalidsense:
sdwalk():
sdwrite():
sdwstat():
seconds(): 113c, 241a, 398c, 659
Section: 160d, 249c, 358c, 503i
Sectorsz-1745:

sectorsz-1770:
seedrand(): 601b, 601c
seg(): 100c, 101a, 165, 439, 441c, 446a, 446b, 447a, 450a, 457
segattach(): 436c, 437
segclock(): 473h, 474a
SEGMAXSIZE: 523d, 523f
Segment:
Segment (typedef):
segpage(): 99e, 265d
Sema:
Sema (typedef):
semacquire(): 445a, 446a
semawoke(): 444d, 445a, 445b
semdequeue(): 443d, 445a, 445b
semqueue(): 443c, 445a, 445b
semrelease(): 444b, 447a
semwakeup(): 444a, 444b, 445a, 445b
Sender-2349: 724c
Sendnotes: 532b, 533g
seprintconf():
seprintdev():
seprintep(): 825a
serialoq: 596b, 620a, 620b, 620c, 620d, 621a, 621b
Set0-2195: 835d, 837c
setaccelerated(): 656, 656
setbootcmd():
Setbuswidth-1621: 762a, 766c
setbuttonmap(): 659, 659
setclkrate(): 250b, 728b
setdebug(): 827a, 827b
setdsize():
setintellimouse(): 656, 656
setlength():
setlinear(): 656, 656
setmalloctag(): 87a, 87b, 87d, 88a, 88b, 88d, 285f, 372b, 407b, 408a, 636b
setmaxpkt(): 799b, 800
setpower(): 530a, 820b
setrealloctag(): 87a, 87c, 88a, 88b, 88d
setres(): 656, 656
setswapchan(): 196c, 197
setupcpop(): 722c, 723a, 723b
setupfpctlop(): 486a, 486b, 486c
setupfpop(): 487a, 487b
SG_BSS: 103g, 166b, 188, 192, 523f
SG_CEXEC: 186b
SG_DATA: 103f, 166b, 188, 192, 436b, 458, 523b
SG_PHYSICAL: 523c, 524c, 524d
SG_RONLY: 101a, 104b, 165, 694
SG_SHARED: 166b, 188, 192, 523c, 523f

SG_STACK: 94a, 103c, 166b, 188, 192, 265d, 436b
SG_TEXT: 103a, 166b, 177c, 188, 192, 265d, 436b, 457, 474b
SG_TYPE: 96d, 102b, 104b, 166b, 189d, 192, 436b, 437, 457, 474b
Shdr32sz-2418:
Shdr64sz-2421:
Shift-963: 348a, 348b, 349a, 349c, 350d, 352b
skipscheds: 211d, 629b
skipslash(): 287d, 288b
sleep: 133c, 192, 233a, 236c, 381a, 385b, 445a, 454e, 535b, 553, 566b, 602c, 603b, 659, 812b, 813b
SLOT-912: 570a, 571d, 574c, 577, 579, 580d
Slotisrver-1647: 762b, 765d
Small: 160f, 167, 249c
smalloc(): 85c, 96c, 97b, 105e, 120b, 122a, 136c, 205f, 265d, 273d, 275a, 276c, 285f, 287d, 288a, 300c, 301a, 302a, 318a, 319a, 320f, 329c, 330a, 330c, 342d, 344b, 400b, 431a, 448c, 465, 466a, 524d, 555, 560c, 564a, 579, 626a, 636b, 653, 674, 786a, 788b, 792, 796c, 800
sname-909: 94b, 104b
Soc: 733a, 733a
soc: 54d
Soc (typedef): 733a
sofdone(): 812a, 812b
sofwait(): 812b, 815e
source-1769:
Sourcead-2292: 830b
sp-2433: 265b, 265d, 267a
Spec-960: 348a, 348a, 350d
Spiintspt-1646: 762b
spname-2276: 783b, 784a, 785c, 799b
Sporadic: 532b, 533g, 535b
Srcdreq-2312: 830b
Srcignore-2311: 830b
Srcinc-2314: 830b
Srcwidth128-2313: 830b
Srstcmd-1651: 762b, 766c
Srstdata-1650: 762b, 766c
Srsthc-1652: 762b, 765c
Srv:
srv-784:
Srv.chan:
Srv.link:
Srv.name:
Srv.owner:
Srv.path:
Srv.perm:
Srv (typedef):
srvattach():
srvclose():
srvcreate():
srvdevtab:
srvinit():

srvlk-783:
srvlookup():
srvname(): 325e
srvopen():
srvread():
srvremove():
srvstat():
srvwalk():
srvwrite():
sseek(): 293a, 293b
SSEG: 100a, 265d, 437, 438b
STACKALIGN: 265c, 265d, 761a
STAGESIZE: 732a
startcpu(): 508b, 508c
startcpus(): 507d
startlock-2448: 508b, 511a, 745
statename: 51c, 202b, 535b, 571d, 612b
STATSIZE-907: 569a, 571d, 680
status(): 841b, 842c
Status-1631: 762b, 766c, 769a
Status-2350: 724c, 726b, 726c
stop(): 840a, 842c
Stopped: 141c, 453a, 453d, 454a, 455a, 456d, 459b, 470c
Stride-2295: 830b
Stride-2574: 488b
subarch(): 488d, 490c
Suspend-1665: 762b
swapaddr: 178, 523d
Swapalloc: 185a, 185b
swapalloc: 185b, 185e, 186a, 187b, 187e, 187g, 191c, 191d, 192, 193a, 197, 261c, 262b, 263a
Swapalloc.alloc: 185a
Swapalloc.free: 185a
Swapalloc.highwater: 185a
Swapalloc.last: 185a
Swapalloc.swmap: 185a
Swapalloc.top: 185a
swapcount(): 166b, 187g
swapimage: 166b, 172b, 178, 185d, 188, 192, 195d, 197, 262b
swapinit(): 250b, 262b
swcursor_init(): 250b
sysalarm(): 222a, 234e
sysattach(): 589d
sysawait(): 109a, 132a
sysbind(): 317a, 322b
sysbrk(): 74, 104c
Syscall (typedef): 41d
syscallfmt(): 459c, 460e
syschdir(): 299a, 311a
sysclose(): 280b

syscreate(): 299a, 307c
sysctab: 145e, 460e
sysdevtab: 590d
sysdir-845: 589b, 589e, 672a, 672b, 672c
sysdup(): 588a, 588b
syserrstr(): 41f, 634c
sysexits(): 109a, 133e
sysfauth(): 557c, 558a
sysfd2path(): 299a, 312a
sysfstat(): 299a, 314c
sysfversion(): 557c, 557e
sysfwstat(): 299a, 315c
sysinit(): 589c
sysmount(): 317a, 337d
sysname: 592c, 592f, 592g
sysnop(): 41f, 42b
sysnoted(): 418a, 422
sysnotify(): 418a, 420f
sysopen(): 279d
syspipe(): 418a, 426a
syspread(): 272a, 290b
syspwrite(): 272a, 292a
sysread(): 589e
sysremove(): 299a, 310d
sysrendezvous(): 418b, 448f
sysretfmt(): 459e, 464
sysrfork(): 109a, 115c
sysseek(): 272a, 293a
syssegattach(): 418a, 436c
syssegbrk(): 418a, 436b
syssegdetach(): 418a, 438b
syssegflush(): 418a, 441c
syssegfree(): 418a, 439
syssemacquire(): 418b, 446a
syssemrelease(): 418b, 447a
syssleep(): 222a, 232c
sysstat(): 314b
systab: 41f, 42a, 145e
SystimerFreq-2652: 223e, 224b, 229b, 229c, 241b
Systimers: 224c, 753
SYSTIMERS-2647: 31e, 224a, 224d, 228d, 231e, 258
Systimers.c0: 224c
Systimers.c1: 224c
Systimers.c2: 224c
Systimers.c3: 224c
Systimers.chi: 224c
Systimers.clo: 224c
Systimers.cs: 224c
Systimers (typedef): 753

systsemacquire(): 418b, 446b
sysunmount(): 317a, 338c
syswalk(): 672a
syswrite(): 590b
syswstat(): 299a, 315b
tadd(): 226d, 227d, 229d, 231a
TagFballoc-2372: 725c
TagFbblank-2374: 359b, 725c
TagFbfree-2373: 725c
TagGetclkmax-2369: 725c
TagGetclkspd-2368: 725c, 728a
TagGetdepth-2379: 359a, 725c
TagGetfwrev-2361: 265a, 725c
TagGetmac-2363: 725c, 727a
TagGetpower-2365: 530b, 725c
TagGetram-2364: 253d, 725c
TagGetres-2375: 359a, 725c
TagGetrev-2362: 264g, 725c
TagGetrgb-2381: 725c
TagGettemp-2371: 725c, 728c
TagGetvres-2377: 725c
TAGMASK: 547d, 548a
TagResp-2360: 725b, 726d
TagSetclkspd-2370: 725c, 728b
TagSetdepth-2380: 725c
TagSetpower-2366: 530a, 725c
TagSetres-2376: 725c
TagSetrgb-2382: 725c
TagSetvres-2378: 725c
TAGSHIFT: 547d, 547d, 548a
talkrdb():
TaslockStats: 628a, 628e
TaslockStats.glare: 628a
TaslockStats.inglare: 628a
TaslockStats.locks: 628a
tconsumed-918: 468d, 468e, 468f, 469a, 469c
TCSys: 471a, 472f, 473a
TCUser: 471a, 472f, 473a
tdel(): 227b, 228a, 228c
Tdmode-2320: 830b
testenq(): 535b, 535b
testschedulability(): 535b
tevents-915: 468d, 468f, 469a, 469c
text2data(): 457, 458
Tfast (typedef): 223f
tfn(): 233a, 233f
thelock-2596: 535b, 535b
Ti-2291: 830b
timeconv(): 535b, 535b

Timer:
Timer (typedef):
timeradd(): 227d, 233a, 259b, 535b
timerdel(): 228b, 228c, 233a, 233b, 233c, 535b
timerintr(): 224d, 229d, 514e
Timermode: 225b
Timers:
timers-2645: 226c, 227d, 228c, 229d, 231a
Timers (typedef):
timersinit(): 250b, 259b, 511a
timersinited-2646: 231b, 259a, 259b
TK2MS: 223a, 232d, 445b, 472b, 472c, 474a, 476e, 595e, 659, 823b, 824
tk2ms(): 223b, 235b, 472f
TK2SEC: 708b
tlock-919: 468d, 469a, 469b
Tmicro (typedef): 222b
TmrDbgHalt-2658: 752c
TmrEnable-2659: 752c, 822a
TmrIntEnable-2660: 752c, 822a
TmrPrescale1-2661: 258, 752c
TmrPrescale16-2662: 752c
TmrPrescale256-2663: 752c
Tms (typedef): 222b
tnames-1775:
Tnano (typedef): 222b
TOD: 237c, 237d
tod: 237d, 238c, 238d, 239a, 239b, 240, 241c, 241d
TOD.cnt: 237c
TOD.delta: 237c
TOD.divider: 237c
TOD.hz: 237c
TOD.init: 237c
TOD.last: 237c
TOD.lasttime: 237c
TOD.multiplier: 237c
TOD.off: 237c
TOD.send: 237c
TOD.sstart: 237c
TOD.udivider: 237c
TOD.umultiplier: 237c
todfix(): 238c, 240
TODFREQ-2669: 238d, 241a, 715
todget(): 145e, 239b, 241a, 242e, 244a, 459e, 468f, 533f, 535b
todinit(): 238c, 239a, 239b, 241c, 241d, 259b, 264a, 600c
todset(): 239a, 243a, 244b
todsetfreq(): 238c, 238d, 244b
topens-916: 468d, 468f, 469a, 469b
Tperiodic: 225b, 226f, 227a, 229d, 231a, 259b
tproduced-917: 468d, 468e, 468f, 469a, 469c

tqid-1773:
trapname(): 613b, 614c
trapnames-2414: 143b, 614c
trap_arch_dumpstack(): 615b
TReal: 471a, 471c, 471d, 472b, 472c, 472f, 473b, 473c
Tree:
Tree.devs:
Tree.nadevs:
Tree.name:
Tree.ndevs:
Tree (typedef):
treealloc():
trees-1765:
Trelative: 225b, 226f, 233a, 535b
trimblock(): 413b
true: 53c, 53d, 60f, 62a, 63f, 65a, 66b, 67b, 68, 77c, 79, 101a, 113b, 132a, 133e, 139c, 141b, 145e, 153c, 154e, 159b, 164a, 164b, 165, 166b, 167, 175b, 191d, 193e, 194a, 194b, 196c, 199f, 229d, 238c, 259b, 262b, 265d, 283d, 285d, 287d, 291, 292c, 293a, 293b, 296d, 304b, 306b, 312a, 314b, 314c, 315c, 321c, 323e, 325e, 326b, 328b, 337b, 337c, 337d, 337f, 339c, 343, 344b, 346, 350d, 353b, 360c, 360e, 361, 363d, 364b, 368f, 369c, 400b, 408a, 424, 426a, 440c, 444c, 445a, 445b, 446a, 446b, 447a, 455d, 456a, 466c, 489c, 490b, 490c, 522g, 522h, 522i, 522k, 557e, 558a, 567i, 574c, 586b, 588b, 594d, 594e, 608b, 618g, 630b, 634d, 637a, 639e, 659, 711b, 712b, 726d
Tsec (typedef): 222b
TSEG: 100a, 177b, 265d, 474a, 474b, 474c, 474d, 584d
tsemacquire(): 445b, 446b
tsleep: 85c, 113c, 192, 232c, 296d, 445b, 659, 766c, 769a, 815e, 820b, 823b, 824, 825b, 825c
TSys: 471a, 472a, 472f, 473b, 473c, 612b
ttname-2275: 783a, 784b, 785c
Tufast (typedef): 223f
Tunestep-1643: 762b
Tunestepsddr-1645: 762b
Tunestepsstd-1644: 762b
TUser: 471a, 472a, 472f, 473b, 473c, 612b
Tval (typedef): 222d
twakeup(): 233a, 234a
TxDone-2220: 835e, 839a
TxEn-2225: 835e, 838b
Txfrlen-2294: 830b
TxIen-2211: 835e, 838b, 839a
TxPin-2183: 835b, 838b
TxRdy-2221: 835e, 837e, 839a, 841d
TYPE-1609:
TypeLOG-1592:
TypeMASK-1594:
TypeSHIFT-1595:
Uart:
uart-1854:
Uart (typedef):
Uartalloc:

uartalloc:
Uartalloc.elist:
uartattach():
uartclock():
uartclose():
uartconsinit(): 250b, 842b
uartctl(): 842b
uartdevtab:
uartdir-1856:
uartdisable():
uartdrained():
uartdrainoutput():
UartEn-2207: 835e, 838b
uartenable():
uartflow():
uartgetc():
UartIrq-2205: 835e, 837e
uartkick(): 837e
uartlist-1853:
uartndir-1857:
uartnuart-1855:
uartopen():
uartputs(): 363e, 621b
uartread():
uartrecv(): 837e
uartreset():
uartstageinput():
uartstageoutput(): 839a
uartstat():
uarttimer-1858:
uartwalk():
uartwrite():
uartwstat():
UINT2PTR: 41b, 163a, 167, 168, 265d, 421b, 761a
unbreak(): 452a, 452c
uncachepage(): 172d, 180c, 188
unconfigure():
undomount(): 304b, 330d
unionread(): 332f, 333b
unionrewind(): 334a, 334b
uniquechan(): 274b, 304b, 308d, 327b
uniquepath(): 301c, 302a, 328c
UNIT-1611:
UnitLOG-1600:
UnitMASK-1602:
UnitSHIFT-1603:
unlock(): 60d, 63f, 64c, 65a, 65e, 66b, 67b, 67d, 68, 69b, 69c, 92d, 97a, 98b, 110f, 111a, 111b, 111d, 113a, 113b, 132c, 133a, 136a, 136d, 137a, 138d, 158a, 166b, 171e, 172a, 172d, 173, 175b, 180c, 182b, 182d, 183b, 185e, 187b, 187e, 188, 194a, 194b, 195d, 201b, 201c, 202a, 212e, 214a, 253a, 273d, 274a, 276c, 279b, 280c,

281b, 291, 292c, 293b, 296b, 296d, 341c, 363e, 364c, 424, 428f, 443c, 443d, 444a, 448f, 469a, 469b, 508b, 511a, 524a, 524c, 525c, 535b, 548b, 548c, 549c, 553, 566b, 566c, 582b, 584d, 585a, 585b, 595e, 651, 659, 737c
unlockfgrp(): 277c, 279b, 426b, 588b
UP: 144b
up: 60c, 60g, 62a, 62b, 63f, 64a, 65e, 66a, 67b, 67c, 70b, 70c, 70d, 70e, 70g, 70h, 71b, 85c, 101a, 101c, 104d, 105c, 106b, 113a, 113c, 114d, 116b, 117a, 117b, 117d, 119b, 119d, 120b, 120c, 120d, 121d, 121e, 122a, 122b, 122c, 130a, 132c, 133a, 133c, 134c, 136a, 136c, 136d, 137a, 137c, 138a, 138c, 138d, 139c, 140a, 140c, 145d, 145e, 147b, 147d, 147f, 150b, 152c, 152e, 155b, 155d, 164a, 164b, 165, 167, 168, 178, 179, 180c, 192, 193d, 199c, 203a, 205c, 206c, 207a, 207b, 207c, 208c, 208d, 208e, 211b, 212e, 213a, 213b, 215c, 215d, 215f, 217b, 218b, 218g, 218h, 228b, 232c, 233a, 233b, 233c, 233f, 234g, 235a, 235b, 235c, 252c, 265d, 267a, 277b, 277c, 280c, 281b, 283d, 285b, 285f, 289b, 289c, 295b, 295c, 306c, 307a, 308d, 311a, 321c, 323e, 328b, 336c, 336d, 336e, 337a, 337b, 339c, 364c, 366a, 372b, 385b, 400b, 403d, 407b, 419f, 419g, 420b, 420e, 420f, 420g, 421b, 422, 423b, 426b, 431a, 434a, 434b, 436b, 436c, 438b, 439, 441a, 441c, 446a, 446b, 447a, 448f, 450a, 450b, 451d, 451e, 454c, 454e, 455f, 459c, 459e, 460c, 460e, 464, 468c, 472b, 472f, 473h, 474a, 476d, 481c, 481f, 482a, 484a, 484c, 485a, 491e, 493c, 494b, 494c, 511a, 530c, 531g, 531h, 532a, 533b, 533f, 535b, 552, 553, 554, 557b, 558b, 560c, 561a, 566c, 567a, 567b, 567d, 567g, 571d, 576, 579, 580d, 582a, 588b, 596b, 602c, 612a, 613a, 613b, 615a, 616a, 624c, 633b, 634a, 634d, 634e, 636b, 659, 673a, 674, 699, 712b, 790, 800, 815e, 818a, 818b, 820b, 823b, 824, 825b, 825c
Up-974: 348a, 349c, 352b
updatecpu(): 217c, 218a, 218b, 218h, 220b
Uro: 161b, 167
Urw: 161b, 167
usbattach(): 794a, 807a
usbbread(): 806a, 807a
usbbwrite(): 806b, 807a
usbclose(): 796b, 807a
usbctl(): 803, 804
usbctls-2272: 782c, 803
usbdevtab: 807a
usbdir-2274: 782e, 790
usbdwclink(): 827c
usbgen(): 790, 794b, 794c, 795, 798b
usbidgen-2283: 783i, 787a, 788a
usbinit(): 793b, 807a
usbload(): 794d, 795
usbmodename: 782f, 784c, 785c
usbopen(): 795, 807a
usbread(): 798b, 806a, 807a
USBREGS-2229: 32e, 827b
usbreset(): 793a, 807a
usbshutdown(): 806c, 807a
usbstat(): 794c, 807a
usbwalk(): 794b, 807a
usbwrite(): 805, 806b, 807a
userwrite(): 366b, 558b
Ustkheadroom-2435: 757c, 761a
UTFmax: 275a, 289c, 354b, 639a, 649
uvorder-1799: 245a, 245a, 245b, 245c
V-2588: 492b, 492c

validaddr(): 101c, 101d, 132a, 134b, 147c, 279d, 291, 292c, 293a, 307c, 310d, 311a, 312a, 314b, 314c, 315b, 315c, 322d, 337f, 338c, 420f, 426a, 437, 446a, 446b, 447a, 459c, 460e, 465, 466a, 493c, 557e, 558a, 634d
validdevname():
validname(): 285e, 315d, 800
validname0(): 285d, 285e, 285f
validnamedup(): 283d, 285d, 337f, 558a
validstat(): 315b, 315c, 315d
VCBUFFER: 358a, 358b, 726d
vcread(): 358b, 726c, 726d
vcreq(): 253d, 264g, 265a, 359a, 359b, 530a, 530b, 726d, 727a, 728a, 728b, 728c
vctl-2412: 157a, 158a, 159b
vctllock-2411: 157c, 158a
vcwrite(): 358b, 726b, 726d
vfiq-2413: 528c, 528d, 529a
VFPKind: 485c
VFPv2-2561: 485c, 489c
VFPv3-2562: 485c, 489c
View-962: 348a, 348a
VIRTIO: 31c, 249c, 832a
VLNUMSIZE-1796: 238b, 242e, 600b
vlong2le(): 244a, 245c
vmemchr(): 101d, 134b, 285f, 437, 466a
Vpage0: 256c, 750b
Vpage0.vectors: 256c
Vpage0.vtable: 256c
Vpage0 (typedef): 750b
v_or_u2.u: 293b
v_or_u2.v: 293b
v_or_u2: 293b, 293b
Waitdreq-2304: 830b
Waitmsg:
Waitmsg (typedef):
Waitq:
Waitq (typedef):
Waitrelease: 535a, 535b
Waitresp-2319: 830b
Waitwrite-2303: 830b
Wakeme: 212d, 212e, 214b, 424, 535b
wakeup: 136d, 137a, 163d, 191b, 191d, 192, 234a, 237b, 296b, 373, 374, 375, 376, 377, 378, 382c, 385b, 387, 389b, 389c, 391d, 444a, 454c, 455b, 535b, 602d, 603b, 659, 766a, 822b, 832e
walk(): 283d, 304b, 308d, 426a
Walkqid:
Walkqid (typedef):
waserror: 102b, 117d, 134b, 134c, 145e, 175b, 178, 189d, 192, 195d, 233a, 236c, 267a, 279d, 282b, 283d, 291, 292c, 293b, 296d, 306a, 307c, 308d, 310d, 314b, 314c, 316, 322d, 323e, 325e, 333b, 334d, 337f, 338c, 344b, 361, 364c, 372b, 383, 384, 385b, 400b, 405a, 405b, 426a, 431a, 434a, 434b, 438b, 445a, 445b, 454e, 457, 469a, 493c, 549c, 553, 554, 555, 557e, 558a, 560c, 566b, 567a, 571d, 574c, 576, 577, 579, 582b, 584d, 586b, 588b, 593b, 594a, 596b, 601b, 603b, 626a, 633d, 656, 659, 673c, 674, 759b, 769a, 785c, 787b, 790, 795, 796a, 796b, 796c, 798b, 800, 803, 804, 805, 806a, 806b, 818a, 818b, 823b, 824

Watchdog:
Watchdog (typedef):
watchdoglink(): 772a
Wdog-2457: 770c, 771b, 771c
wdogfeed(): 771c, 772a
Wdogfreq-2450: 770b, 771c
wdogoff(): 771d
Wdogtime-2451: 770b, 771c
werrstr(): 634e
Whinesecs-2604: 113c, 114a
wlock(): 67b, 321a, 323e, 339c, 341c, 674
WR(): 764d, 765c, 766a, 766b, 766c, 768, 769a
write(): 292a, 292c
Write-2234: 810a, 818a, 822c, 824
Write-2347: 724c, 726b
writebintime(): 243c, 244b
writefn-853: 736e, 737c, 738e
Writerdy-1694: 762b, 766c
writetime(): 242d, 243a
writetomem(): 154d, 154e, 155a
Writetrans-1700: 762b
wstat(): 315b, 315c, 316
wunlock(): 67d, 321a, 323e, 339c, 341c, 674
Xalloc:
xalloc(): 77c, 80b, 83a, 83b, 261a, 261c, 262b, 263c, 609c
Xalloc (typedef):
xallocz(): 77c, 77d
xfree(): 78a, 609c
Xhdr:
Xhdr (typedef):
xhole(): 78a, 78c, 80b, 260
xinit(): 250b, 260
xlists-1849: 76, 77d, 78c, 260, 610a
xmerge(): 79, 83a, 83b
xspanalloc(): 80b
xsummary(): 78b, 261b, 407b, 609c, 610a
yfn(): 535b, 535b
Yield: 533g, 535b
yield(): 210b, 232c
Yieldonblock: 533g, 535b
Z-2586: 492b, 492c
_SEG0: 100a
_SEG1: 100a
_SEG2: 100a
_SEG3: 100a
_SEG4: 100a
_allocb(): 406e, 407b, 408a
_assert:
_efgfmt():

_fmtlock():
_fmtunlock():
_globalsegattach: 436d, 437
_proctrace(): 468f, 469a
__anon_enum_102: 394a
__anon_enum_103: 589a
__anon_enum_104: 593e
__anon_enum_105: 736c
__anon_enum_106: 428a
__anon_enum_107: 568a
__anon_enum_108: 568b
__anon_enum_109: 680
__anon_enum_112: 656
__anon_enum_113: 659
__anon_enum_114: 659
__anon_enum_115: 659
__anon_enum_116: 348a
__anon_enum_118: 651
__anon_enum_119: 653
__anon_enum_193:
__anon_enum_194:
__anon_enum_195:
__anon_enum_196:
__anon_enum_198: 762a
__anon_enum_199: 762b
__anon_enum_213:
__anon_enum_214: 359c
__anon_enum_215: 643
__anon_enum_216: 694
__anon_enum_217: 75c
__anon_enum_219: 698b
__anon_enum_221: 748a
__anon_enum_222:
__anon_enum_293: 835d
__anon_enum_294: 835e
__anon_enum_295: 810a
__anon_enum_296: 781
__anon_enum_298: 772c
__anon_enum_299: 830b
__anon_enum_321: 529c
__anon_enum_322: 727b
__anon_enum_323: 724c
__anon_enum_324: 725a
__anon_enum_325: 725b
__anon_enum_326: 725c
__anon_enum_327: 722b
__anon_enum_332: 750a
__anon_enum_333:
__anon_enum_335: 757c

__anon_enum_338: 758f
__anon_enum_339: 745
__anon_enum_340: 770b
__anon_enum_341: 770c
__anon_enum_342: 705c
__anon_enum_343: 320b
__anon_enum_344: 705c
__anon_enum_345: 447b
__anon_enum_346: 705c
__anon_enum_347: 705c
__anon_enum_348: 712c
__anon_enum_356: 481i
__anon_enum_357: 488a
__anon_enum_358: 488b
__anon_enum_359: 488c
__anon_enum_360: 492b
__anon_enum_361:
__anon_enum_362: 535b
__anon_enum_363: 535b
__anon_enum_364: 535b
__anon_enum_365: 710
__anon_enum_369: 709c
__anon_enum_370: 752b
__anon_enum_371: 752c
__anon_enum_88:
__anon_enum_89: 684
__anon_enum_91: 689a
__anon_enum_92: 687
__anon_enum_94: 547d
__anon_enum_96: 674
__anon_struct_197.name:
__anon_struct_197.parse:
__anon_struct_197:
__anon_struct_336.cmdline: 758e
__anon_struct_336.data: 758e
__anon_struct_336.mem: 758e
__anon_struct_336: 758e, 758e
__anon_struct_337.base: 758e
__anon_struct_337.size: 758e
__anon_struct_337: 758e, 758e
__anon_struct_93.aux: 51e
__anon_struct_93.chanpipe: 428e
__anon_struct_93.mid: 571b
__anon_struct_93.pgrpid: 570c
__anon_struct_93: 51e, 51e

Bibliography

[Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. cited page(s) 20

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. cited page(s) 19

[Pad09] Yoann Padioleau. Syncweb, literate programming meets unison, 2009. https://github.com/aryx/syncweb. cited page(s) 20

[Pad15a] Yoann Padioleau. *Principia Softwarica: The ARM Assembler 5a*. 2015. cited page(s) 19, 20, 606

[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) 43, 143, 606

[Pad16a] Yoann Padioleau. *Principia Softwarica: The Build System mk*. 2016. cited page(s) 606

[Pad16b] Yoann Padioleau. *Principia Softwarica: The Plan 9 C Compiler 5c*. 2016. cited page(s) 33, 606

[Pad16c] Yoann Padioleau. *Principia Softwarica: The Plan 9 Core Libraries*. 2016. cited page(s) 21, 22, 81, 416, 606

[Pad16d] Yoann Padioleau. *Principia Softwarica: The Plan 9 Graphics System draw*. 2016. cited page(s) 18, 32, 82, 357, 606

[Pad16e] Yoann Padioleau. *Principia Softwarica: The Plan 9 Network Stack /net*. 2016. cited page(s) 18

[Pad18] Yoann Padioleau. *Principia Softwarica: The Plan 9 Shell rc*. 2018. cited page(s) 606