Principia Softwarica: The Plan 9 Core Libraries

version 0.1

Yoann Padioleau
yoann.padioleau@gmail.com

April 28, 2026

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

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

Contents

1 Introduction **10**
- 1.1 Motivations . . . 10
- 1.2 The Plan 9 core libraries: `lib{c,bio,regexp,flate,thread}` . . . 11
- 1.3 Other core libraries . . . 11
- 1.4 Getting started . . . 12
- 1.5 Requirements . . . 13
- 1.6 About this document . . . 13
- 1.7 Copyright . . . 13
- 1.8 Acknowledgments . . . 14

2 Overview **15**
- 2.1 Principles . . . 15
  - 2.1.1 Core versus non-core . . . 15
  - 2.1.2 User/kernel bridge, system calls . . . 15
  - 2.1.3 Memory allocator . . . 16
  - 2.1.4 Zero-cost abstractions . . . 16
  - 2.1.5 Thread safety and reentrancy . . . 16
  - 2.1.6 The library that runs inside everything . . . 17
- 2.2 `libc.h` interface . . . 17
- 2.3 Crash course: Plan 9 ARM assembly . . . 17
  - 2.3.1 Registers . . . 17
  - 2.3.2 Pseudo-registers . . . 18
  - 2.3.3 Key directives and notations . . . 19
  - 2.3.4 The static base (`SB`) and `setR12` . . . 20
  - 2.3.5 Calling conventions . . . 21
- 2.4 Code organization . . . 21
- 2.5 Software architecture . . . 21
- 2.6 Book structure . . . 22

I Foundations **25**

3 Core Types and Data Structures **26**
- 3.1 Basic types . . . 26
  - 3.1.1 Booleans . . . 26
  - 3.1.2 Integers . . . 26
  - 3.1.3 Floats . . . 27
  - 3.1.4 Strings and characters . . . 28
- 3.2 Complex types . . . 31
  - 3.2.1 Pointers . . . 31

3.2.2 Arrays (and qsort()) . . . 31
3.2.3 Structures . . . 34
3.2.4 Functions . . . 34
3.3 Regular expressions . . . 35
3.4 Files . . . 35
3.5 Time . . . 36
3.6 Concurrency . . . 36
3.7 Collections . . . 36

4 User/Kernel Bridge **37**
4.1 The syscall table . . . 37
4.2 The assembly stubs (ARM) . . . 38
4.3 The calling conventions (ARM) . . . 38
4.4 Comparison with UNIX . . . 38

5 _main() and _exits() **40**
5.1 _main() (ARM) . . . 40
5.2 Command-line arguments: ARGBEGIN(), ARGEND() . . . 43
5.3 Syscall: _exits() . . . 44
5.4 exits() and atexit() . . . 44

6 Memory Area Operations **46**
6.1 Filling: memset() . . . 46
6.2 memset() (ARM) . . . 47
6.3 Copying: memcpy() and memmove() . . . 48
6.4 memmove() (ARM) . . . 49
6.5 Comparing: memcmp() . . . 53
6.6 Searching: memchr() . . . 54

7 Memory Management **55**
7.1 Memory allocator principles . . . 55
7.1.1 Garbage collection versus manual memory . . . 56
7.2 Syscall: brk() . . . 56
7.3 sbrk() . . . 57
7.4 Data structures . . . 58
7.4.1 Pool . . . 58
7.4.2 sbrkmem and mainmem . . . 59
7.4.3 Block headers and free list . . . 60
7.5 malloc() and free() . . . 63
7.6 poolalloc() . . . 64
7.7 poolfree() . . . 65
7.8 Other functions . . . 66
7.9 Debugging support . . . 69
7.10 Advanced topics . . . 70

8 Formatted Output **71**
8.1 Data structures: Fmt . . . 71
8.2 print() and vfprint() . . . 72
8.3 dofmt() . . . 73
8.4 fprint() . . . 75

8.5 sprint() and variants . . . 75

II Data Processing 77

9 Strings 78
9.1 Strings and UTF-8 principles . . . 78
9.2 Plain C strings . . . 79
9.3 Runes . . . 83
9.3.1 Conversion . . . 83
9.3.2 Basics . . . 86
9.4 UTF8 . . . 88
9.5 Quoted strings . . . 90
9.6 Extensible strings . . . 91
9.6.1 Data structure: String . . . 91
9.6.2 Allocation: s_new(), s_copy(), s_free() . . . 92
9.6.3 Growing: s_putc(), s_grow(), s_terminate() . . . 94
9.6.4 Appending: s_append(), s_nappend(), s_memappend() . . . 95
9.6.5 Resetting: s_reset(), s_restart() . . . 96
9.6.6 I/O: s_read_line(), s_read(), s_getline() . . . 96
9.6.7 Parsing: s_parse() . . . 99
9.6.8 Miscellaneous: s_tolower(), s_rdinstack() . . . 99

10 Regular Expressions 104
10.1 Regular expression principles . . . 104
10.2 Data structures: Reprog, Reinst, Reclass, Resub . . . 105
10.3 Compilation: regcomp() . . . 107
10.4 Execution: regexec() . . . 109
10.5 Substitution: regsub() . . . 113
10.6 Rune variants: rregexec(), rregsub() . . . 114
10.7 Error handling: regerror() . . . 119

11 Mathematics 120
11.1 Floating-point principles . . . 121
11.2 Basics . . . 122
11.3 Integers . . . 123
11.3.1 Parsing . . . 123
11.4 Floats . . . 126
11.4.1 Special numbers . . . 126
11.4.2 Parsing . . . 127
11.4.3 sqrt() . . . 136
11.4.4 modf() . . . 137
11.4.5 fmod() . . . 138
11.4.6 pow() . . . 139
11.5 Trigonometry . . . 141
11.5.1 Direct functions . . . 141
11.5.2 Inverse functions . . . 144
11.5.3 Hyperbolic functions . . . 147
11.6 Logarithms and exponentials . . . 149
11.6.1 exp() . . . 149

11.6.2 ldexp() . . . 150
11.6.3 frexp() . . . 151
11.6.4 log() . . . 151
11.7 Random numbers . . . 153
11.8 muldiv . . . 155
11.9 Advanced topics . . . 156
11.9.1 Arbitrary precision arithmetic . . . 156

12 Compression **157**
12.1 Compression principles . . . 158
12.2 Data structures and interface . . . 159
12.3 Overview of the DEFLATE format . . . 159
12.4 Inflation (decompression) . . . 160
12.4.1 Initialization: inflateinit() . . . 160
12.4.2 The decompression loop: inflate(), decode() . . . 161
12.4.3 Huffman tree decoding: hufftab(), hdecsym() . . . 165
12.4.4 Bit stream: sregfill(), sregunget() . . . 167
12.5 Deflation (compression) . . . 168
12.5.1 Initialization: deflateinit() . . . 168
12.5.2 LZ77 matching: lzmatch(), lzcomp() . . . 170
12.5.3 Huffman tree construction: mkprecode(), huffcodes(), leafsort() . . . 174
12.5.4 Bit output: lzput(), lzflushbits(), wrblock() . . . 179
12.5.5 The compression loop: deflate(), deflateb() . . . 180
12.6 Block interface: deflateblock(), inflateblock() . . . 184
12.7 Checksums: adler32(), blockcrc() . . . 185
12.8 Zlib wrappers: deflatezlib(), inflatezlib() . . . 187
12.9 Error handling: flateerr() . . . 188

III System Services **189**

13 File IO **190**
13.1 Data structures . . . 190
13.2 Syscalls: open(), pread(), etc . . . 191
13.3 read(), write() . . . 192
13.4 readn() . . . 192
13.5 Advanced topics . . . 192
13.5.1 Scatter/gather I/O . . . 192

14 Buffered IO **194**
14.1 Data structures . . . 195
14.2 Initialization: Binit(), Bopen(), Bterm() . . . 196
14.3 Reading . . . 199
14.3.1 Byte reading: Bgetc(), Bungetc() . . . 200
14.3.2 Rune reading: Bgetrune(), Bungetrune() . . . 201
14.3.3 Line reading: Brdline() . . . 202
14.3.4 Line reading (allocating): Brdstr() . . . 204
14.3.5 Bulk reading: Bread() . . . 206
14.4 Writing . . . 207
14.4.1 Byte writing: Bputc() . . . 207

14.4.2 Rune writing: Bputrune() . . . 207
14.4.3 Bulk writing: Bwrite() . . . 208
14.4.4 Formatted writing: Bprint(), Bvprint() . . . 208
14.4.5 Flushing: Bflush() . . . 210
14.5 Seeking and position: Bseek(), Boffset() . . . 210
14.6 Helpers: Bbuffered(), Bfildes(), Blinelen() . . . 212

15 Directories and File Paths **213**
15.1 Data structures: Dir . . . 213
15.2 Syscalls: create(), chdir(), etc . . . 214
15.3 xxxfstat() . . . 215
15.4 xxxstat() . . . 216
15.5 getwd() . . . 217
15.6 The deprecated mktemp() . . . 217
15.7 The safe mkstemp() . . . 218
15.8 Normalizing a filename: cleanname() . . . 218
15.9 access() . . . 220
15.10Path manipulation . . . 221

16 Namespace **222**
16.1 Syscalls: bind(), mount(), etc . . . 222

17 Time **224**
17.1 Syscalls: sleep(), alarm() . . . 224
17.2 Time and day . . . 224
17.3 Time zones and localtime() . . . 224
17.4 time() . . . 226
17.5 Conversions . . . 228

18 Fork, Exec, and Wait **230**
18.1 Syscalls: rfork(), exec(), await(), etc . . . 230
18.2 fork() . . . 231
18.3 execl() . . . 231
18.4 wait() . . . 232
18.5 getpid() . . . 232

19 Error Management **233**
19.1 Syscall: errstr() . . . 233
19.2 rerrstr(), werrstr() . . . 234
19.3 perror() . . . 234
19.4 Error return values . . . 234
19.5 assert() . . . 235
19.6 sysfatal() . . . 235

20 Security **237**
20.1 getuser() . . . 237
20.2 Cryptography . . . 237
20.3 SSL/TLS . . . 239

IV Concurrency and Communication 242

21 Concurrency 243
21.1 Syscalls: `rendezvous()`, etc . . . 244
21.2 Locks . . . 245
21.3 Atomic operations: `ainc()` (ARM) . . . 246
21.4 Waiting queues . . . 246
21.5 Read-write locks . . . 247
21.6 Rendez-vous . . . 251
21.7 Coroutines . . . 253

22 IPC 254
22.1 Syscalls: `pipe()`, `notify()`, `await()`, `segattach()` . . . 254
22.2 Helpers . . . 255
22.3 Parent-child IPC . . . 255
22.3.1 Wait message . . . 255
22.3.2 Environment . . . 257
22.4 Notes . . . 258
22.4.1 Posting . . . 258
22.4.2 Notified . . . 259
22.5 Misc . . . 260
22.6 Pipes . . . 260
22.7 Shared segments . . . 261
22.8 Fileservers . . . 261
22.8.1 `Fcall` . . . 261
22.8.2 Dumping . . . 263
22.8.3 Marshalling macros . . . 266
22.8.4 `readp9msg()` . . . 266
22.8.5 Parsing . . . 267

23 Channels and Cooperative Threads 283
23.1 Concurrency model principles . . . 283
23.1.1 The return of coroutines . . . 284
23.1.2 Events, threads, and the colored-function problem . . . 285
23.2 Overview . . . 285
23.2.1 Code organization . . . 285
23.2.2 Software architecture . . . 285
23.3 A toy example . . . 286
23.4 Core data structures . . . 287
23.4.1 Concurrency buiding blocks . . . 287
23.4.2 `Channel` . . . 288
23.4.3 `Proc` and `Pqueue` . . . 292
23.4.4 `Thread` and `Tqueue` . . . 294
23.4.5 `Alt` . . . 299
23.5 `main()` and `threadmain()` . . . 300
23.6 Threads scheduler . . . 301
23.7 `alt()` . . . 306
23.8 Send/Receive . . . 312
23.8.1 `runop` . . . 312
23.8.2 `send()`/`recv()` . . . 313

23.8.3 Non blocking operations: nbxxx() . . . 313
23.8.4 Typed send and receive . . . 313
23.9 IO processus . . . 315
23.9.1 Ioproc . . . 315
23.9.2 xiproc() . . . 316
23.9.3 iocall() . . . 317
23.9.4 IO wrappers . . . 318
23.10 Thread-aware libc . . . 320
23.10.1 Memory . . . 320
23.10.2 Exit . . . 321
23.10.3 Fork . . . 322
23.10.4 Exec . . . 323
23.10.5 Interruptions . . . 325
23.10.6 Notes . . . 325
23.10.7 Dialing . . . 328
23.11 Error management . . . 333
23.12 Debugging . . . 333
23.13 Advanced topics . . . 334
23.13.1 Per thread private data . . . 335
23.13.2 Thread groups . . . 337
23.13.3 Thread kills . . . 337
23.13.4 Rendez vous . . . 339
23.13.5 chanprint() . . . 341

24 Network **342**
24.1 Example: an echo server . . . 342
24.2 netmkaddr() . . . 343
24.3 announce(), accept(), listen(), reject() . . . 344
24.4 dial() . . . 347

V Debugging and Profiling **353**

25 Debugging Support **354**
25.1 Logging . . . 354
25.2 abort() . . . 356

26 Profiling Support **357**
26.1 Data structures . . . 357
26.2 prof() . . . 358
26.3 _profin() . . . 358
26.4 _profout() . . . 359
26.5 _profinit() . . . 360
26.6 _profmain() . . . 360
26.7 _profdump() . . . 361
26.8 cputime() . . . 363

27 Conclusion **365**
27.1 Patterns and techniques . . . 365
27.2 Connections to other books . . . 365
27.3 Beyond the Plan 9 core libraries . . . 366

A Extra Code **368**
A.1 `include/` . . . 368
A.1.1 `include/arch/arm/u.h` . . . 368
A.1.2 `include/core/libc.h` . . . 368
A.1.3 `include/core/ctype.h` . . . 379
A.1.4 `include/core/syscall.h` . . . 379
A.1.5 `include/core/internals/pool.h` . . . 381
A.1.6 `include/io/bio.h` . . . 381
A.1.7 `include/strings/regexp.h` . . . 383
A.1.8 `include/strings/str.h` . . . 383
A.1.9 `include/compress/flate.h` . . . 384
A.1.10 `include/ipc/thread.h` . . . 384
A.1.11 `include/ipc/fcall.h` . . . 386
A.2 `libc/port` . . . 387
A.3 `libc/9sys/` . . . 462
A.4 `libc/fmt/` . . . 500
A.5 `libc/arm/` . . . 538
A.6 `libbio/` . . . 553
A.7 `libthread/arm` . . . 557
A.8 `libthread/x86` . . . 557
A.9 `libthread/` . . . 558
A.10 `libregexp/` . . . 569
A.11 `libstring/` . . . 584
A.12 `libflate/` . . . 586

Glossary **608**

Index **609**

References **639**

Chapter 1

Introduction

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

1.1 Motivations

Why core libraries? Because I think you are a better programmer if you fully understand how things work under the hood. Core libraries are linked into every program. They are the code that runs before `main()`, the code behind `malloc()` and `free()`, the code that turns `printf()` into system calls. Understanding them means understanding what *every* program does, whether it knows it or not. Some components are surprisingly deep. The memory allocator is performance-critical enough that companies like Google and Facebook employ engineers to design custom allocators (tcmalloc, jemalloc). Unicode string handling, which most programmers take for granted, involves non-trivial encoding and decoding logic. A cooperative thread scheduler is the same kind of mechanism used inside language runtimes for Erlang, Go, and many others. Regular expression matching is a beautiful application of automata theory. Buffered I/O shows how a thin layer of user-space buffering turns expensive system calls into cheap function calls. And data compression is one of those algorithms that seems like magic until you see the implementation: Huffman coding plus dictionary-based matching, all done in a streaming fashion. In this book, we will see all of these from the inside.

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

- Is there any code executed before `main()` in a C program?
- Which program does memory allocation? The kernel? The C library? How is `malloc()` implemented? How does it relate to the `sbrk()` system call?
- How does a cooperative thread scheduler work?
- How is UTF-8 encoded and decoded? How do you search for a character in a UTF-8 string without accidentally matching a continuation byte?
- How does `printf()`-style formatted output work? How can you add your own format verbs?
- How are `sin()`, `cos()`, `exp()`, and `sqrt()` computed when the hardware has no floating-point unit?
- How does a regular expression engine work?
- Why do we need buffered I/O on top of `read()` and `write()`? How does reading one character at a time remain efficient?
- How does data compression work? How can a compressor work in a streaming fashion without knowing the entire input in advance?

- What does it mean for a library to be thread-safe? What is the difference between a thread-safe function and a reentrant function? When processes share memory, what happens if two of them call `malloc()` at the same time?

1.2 The Plan 9 core libraries: `lib{c,bio,regexp,flate,thread}`

To study all these topics concretely, I need a specific implementation. Like for most books in Principia Softwarica, I chose Plan 9 because its libraries are simple, small, elegant, open source, and they form together a coherent set. In Plan 9, the capabilities described above are spread across five libraries: `libc` (the C standard library: memory, strings, Unicode, formatted I/O, system calls, math, networking), `libbio` (buffered I/O), `libregexp` (regular expressions), `libflate` (compression), and `libthread` (cooperative threads with channels). Together, they total roughly 15 000 lines of C and a few hundred lines of assembly—small enough to explain in full detail in a single book.

Most of the code in this book is portable C—the same source is shared between the ARM and x86 ports. Some routines are written in assembly, either because there is no choice (the C runtime entry point, context switching, atomic operations) or for performance (memory and string operations). For those I focus on the ARM versions, since ARM is the architecture used throughout Principia Softwarica. Sections covering ARM-specific code are marked with "(ARM)" in their title.

1.3 Other core libraries

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

- GNU Libc[1] (also called `glibc`) is the standard C library on most Linux distributions. It is a full POSIX-compliant implementation, supporting dozens of architectures and locales. However, its codebase is enormous: over 1 000 000 LOC, which is almost two orders of magnitude more code than Plan 9's `libc`. It also cannot be linked statically in practice: `glibc` internally uses `dlopen` (dynamic loading) for NSS (Name Service Switch), a pluggable system that lets administrators choose how hostnames, users, and passwords are resolved at runtime. Because NSS loads shared libraries, any program linked against `glibc`—even statically—may still need dynamic linking at runtime, making it impossible to produce a truly self-contained binary.

- musl[2] is a lightweight, POSIX-compliant C library designed for static linking on Linux. It is much smaller than `glibc` (around 90 000 LOC) and is known for clean, readable code. It would have been a reasonable alternative to the Plan 9 libraries, but it is still significantly larger and is tied to the Linux kernel's system call interface.

- dietlibc[3] is a minimal C library by Felix von Leitner, optimized for producing the smallest possible statically-linked executables. The project appears to be largely unmaintained.

- uClibc[4] (originally "microcontroller C library") was designed for embedded Linux systems with limited memory. It is much smaller than `glibc` and supports many architectures, but targets Linux specifically. The project was largely dormant for years before being revived as uClibc-ng. For embedded use it has been mostly superseded by musl.

[1] https://www.gnu.org/software/libc/
[2] https://musl.libc.org/
[3] https://www.fefe.de/dietlibc/
[4] https://uclibc-ng.org/

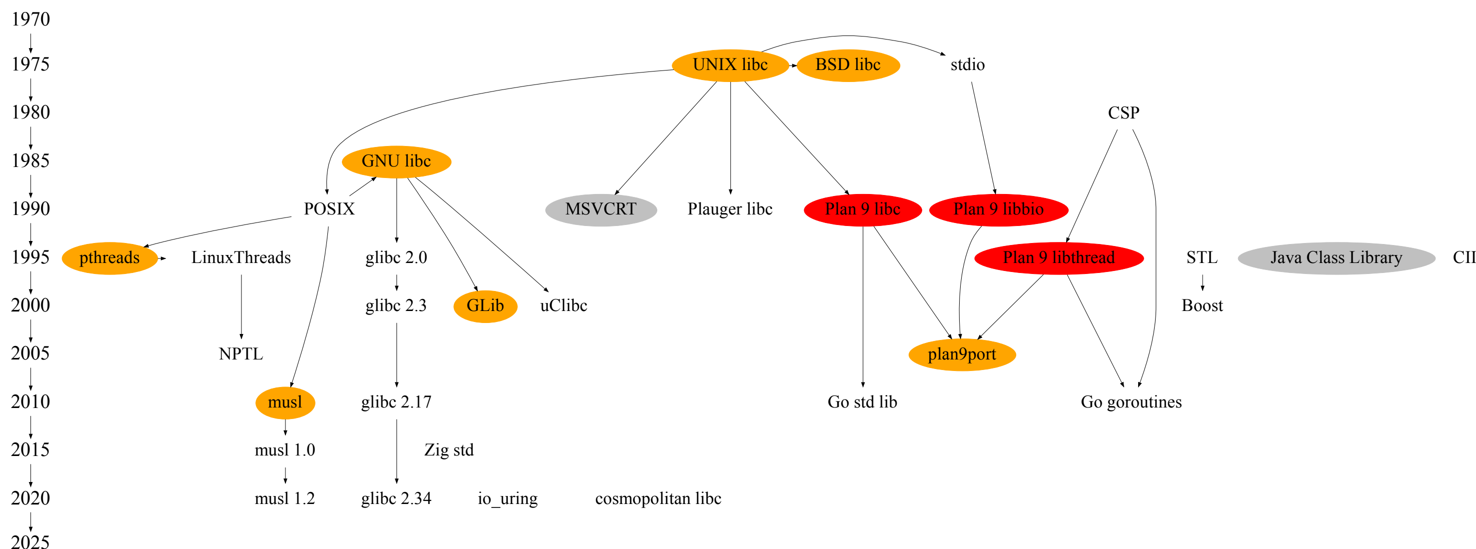


Figure 1.1: C standard libraries timeline

- Plauger's *The Standard C Library* [Pla92] is the closest predecessor to what this book attempts: a line-by-line explanation of every function in a C standard library. The complete source code is printed in the book (but is not open source). Plauger's implementation targets the C89 standard and is portable across platforms. However, it covers only the ANSI subset—no threading, no networking, no Unicode—and predates UTF-8 entirely.

- The C library from Hanson's *C Interfaces and Implementations* [Han96] provides a well-designed set of abstract data types (arenas, lists, tables, rings, sequences) that are notably absent from Plan 9's `libc`. However, it is a pedagogical library, not a system library—it builds on top of an existing `libc` rather than replacing it.

Figure 1.1 presents a timeline of major C standard library implementations. The Plan 9 libraries are highlighted in red: `libc`, `libbio`, and `libthread`. They descend from the original UNIX libc (1973) but diverge significantly—Plan 9's `libc` was designed from scratch for UTF-8, uses `errstr` instead of `errno`, and is far smaller than POSIX-compliant alternatives like GNU libc or musl.

The threading lineage is particularly interesting: Plan 9's `libthread` drew on Hoare's CSP (1978) to provide cooperative threads with channels, a model that later inspired Go's goroutines directly. The buffered I/O lineage runs from Lesk's `stdio` (1975) through AT&T's `sfio` to Plan 9's `libbio`, each iteration simplifying the API.

I think the Plan 9 core libraries represent the best compromise for this book: they cover the essential components—memory allocation, string manipulation, buffered I/O, Unicode, and threading—while remaining small enough to explain in full detail.

1.4 Getting started

The core libraries are normally compiled as part of the full Plan 9 build (see https://www.principia-softwarica.org/getting-started.html). To cross-compile the ARM libraries from a Linux or macOS host:

```
$ cd lib_core/libc
$ objtype=arm mk all
```

The linker `5l` automatically links `libc` into every program, so you do not need to specify it explicitly. To use `libthread` or `libregexp`, pass them on the `5l` command line:

```
$ 5c myprogram.c
$ 5l -lthread myprogram.5
```

Most of the libraries described in this book are also available through plan9port[5] and Goken9cc[6], where they are compiled natively on Linux, macOS, or Windows using gcc or clang. In particular, both provide `libc` (with the UTF-8 string functions, `print()`, `malloc()`), `libbio`, `libregexp`, and `libthread`—so programs using these libraries can run outside Plan 9. Some functions that depend on Plan 9 kernel features (e.g., `rfork()`, `/dev/cons`) are emulated or adapted for the host OS.

1.5 Requirements

Because this book is made of C source code, you will need a good knowledge of the C programming language. Kernighan and Ritchie's *The C Programming Language* [KR88] and Harbison and Steele's *C: A Reference Manual* [HS02] are both excellent references; they also describe the standard library's API (function signatures, semantics, edge cases), which complements this book's focus on the implementation. Some library code makes heavy use of pointer arithmetic and bit manipulation (especially the memory allocator), so comfort with low-level C is important. Reading Plauger's *The Standard C Library* [Pla92]—the only other book that explains a complete C library implementation line by line—is useful background but not required. Hanson's *C Interfaces and Implementations* [Han96] is also relevant for its treatment of arenas, assertions, and abstract data types.

Reading the KERNEL book [Pad14] is helpful but not required: the kernel implements the system calls that the libraries wrap, and understanding both sides gives a fuller picture. The ASSEMBLER book [Pad15a] is useful for the few routines written in ARM assembly (e.g., `memset`, `getcallerpc`).

If, while reading this book, you have specific questions on the API of a particular library function, I suggest you to consult the manual pages in my Plan 9 repository. Section 2 (`docs/man/2/`) documents the C libraries described here; `docs/man/2/0intro` is a useful overview page (the analog of `docs/man/1/0intro` for the utilities). Representative pages for the topics covered in this book include `2/malloc` and `2/pool` (memory allocator), `2/print` and `2/fmtinstall` (formatted I/O), `2/bio` (buffered I/O), `2/rune` and `2/isalpharune` (UTF-8), `2/regexp` (regular expressions), `2/flate` (compression), `2/cleanname` (path canonicalization), `2/rendezvous` and `2/lock` (synchronization), `2/thread` (the thread and channel library), and `2/dial` (network connections). The top-level `docs/articles/` directory contains two papers that are essential background for specific chapters: `docs/articles/utf.ps` ("Hello World", rendered in Greek and Japanese) for the chapter on UTF-8, and `docs/articles/lexnames.ps` ("Lexical File Names in Plan 9") for `cleanname` and the path-handling functions.

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.

[5] https://9fans.github.io/plan9port/
[6] https://github.com/aryx/goken9cc

1.8 Acknowledgments

I would like to acknowledge the authors of the Plan 9 core libraries: Rob Pike (UTF-8, `libbio`, the `fmt` library), Ken Thompson (UTF-8, much of `libc`), Russ Cox (`libthread`, `dial`, many `libc` improvements), Phil Winterbottom (`libthread`, the pool allocator), and the other Bell Labs and Plan 9 contributors.

Chapter 2

Overview

Before diving into the source code, this chapter provides the background needed to read the rest of the book: design principles, the library's interface and architecture, a crash course on Plan 9 ARM assembly, and a roadmap of the chapters.

2.1 Principles

Every C program, no matter how small, depends on a standard library. But what should that library contain? How should it be organized? And how does it relate to the operating system underneath? In this section I discuss the general principles behind the design of a C standard library.

2.1.1 Core versus non-core

What counts as a "core" library has evolved over time. A traditional Unix `libc` provides only the bare essentials: memory allocation, string manipulation, I/O, and system call wrappers. But modern languages have raised expectations: Java's standard library includes collections, concurrency, and networking; Go ships with HTTP servers, JSON, cryptography, and compression; Python's "batteries included" philosophy is similar. By "core" in this book I mean the non-graphical foundation that all Plan 9 programs share: `libc` itself, plus `libbio` (buffered I/O), `libthread` (cooperative threads and channels), `libregexp` (regular expressions), and `libflate` (compression). Together they form the common runtime that programs are built on. The graphical libraries (`libdraw`, `libmemdraw`) are covered in Graphics book [Pad16c] and the widget toolkit (`libpanel`) in Widgets book [Pad26a].

The notion of a shared C library dates back to Dennis Ritchie's original UNIX`libc` of 1973, catalogued in the 1978 Kernighan & Ritchie book, and each decade since has raised the floor of what "standard library" means. At the minimalist end sit libcs like `musl` (used by Alpine Linux and many static builds), `dietlibc`, and newlib for embedded systems, all stripping everything down to what POSIX demands and nothing more. Plan 9's `libc` sits in the middle—small enough to read end-to-end (a few thousand lines), but rich enough to support the whole Plan 9 user space.

2.1.2 User/kernel bridge, system calls

A recurring pattern in this book is the two-layer structure of each facility: a thin system call wrapper at the bottom, and higher-level library functions on top. For example, the memory chapter starts with `sbrk()` (the syscall that extends the process's data segment) and then builds `malloc()/free()` on top. The I/O chapter starts with `open()/read()/write()` and then builds `Biobuf` buffered I/O on top. Understanding both layers—what the kernel provides and what the library adds—is key to understanding how C programs really work.

Every UNIX`libc` follows this two-layer design—glibc, musl, FreeBSD `libc`, Apple's `libSystem`, and Windows' UCRT all sit on top of whatever system-call interface the kernel exposes. The separation itself dates to Thompson's first UNIX`libc` on the PDP-11 in 1972, which added `printf`-style formatting and buffered I/O on top of the bare `read`/`write` syscalls—the pattern every later `libc` has copied. What varies today is the *thickness* of the library layer: glibc is roughly 1.4 million lines of C (with features going far beyond POSIX), musl is about 100 000 (deliberately minimal), and Plan 9's `libc` is a few thousand—small enough to read the syscall wrapper and the library function side by side in the same chapter.

2.1.3 Memory allocator

The memory allocator deserves special attention. It is one of the few library components that is also used *inside the kernel*—the Plan 9 kernel shares the same `Pool` allocator with user-space programs. This means the allocator code must be careful: it cannot call `malloc()` itself (that would be circular), and certain kernel paths must avoid allocation entirely because they run with interrupts disabled.

Most other systems keep the user-space allocator and the kernel allocator completely separate: glibc's `ptmalloc2` (descended from Doug Lea's 1987 `dlmalloc`), jemalloc, and tcmalloc live only in user space, and the Linux kernel has its own dedicated SLAB/SLUB/SLOB allocators (descended from Jeff Bonwick's 1994 SunOS paper) that share no code with user-space `malloc`. Plan 9's unified `Pool` is the outlier. The full cross-allocator family tree—boundary-tag, buddy, slab, and friends—lives in Chapter 7's principles section.

2.1.4 Zero-cost abstractions

Core library code must be fast. Every program pays the cost of `malloc()`, `memcpy()`, and `print()`, so any inefficiency is multiplied across the entire system. This is why some routines (like `memset` and `memmove`) are implemented in hand-written assembly for the critical path, even though C versions exist. The principle is: the abstractions that the library provides should cost nothing (or very close to nothing) compared to doing the same work by hand.

The phrase *zero-cost abstractions* itself is Bjarne Stroustrup's famous motto for C++ from the 1980s ("what you don't use, you don't pay for; what you do use, you couldn't hand-code any better"), later picked up as an explicit design principle by the Rust project in 2014. Plan 9's `libc` is more aggressive than most about falling back to hand-written assembly when the C version is not fast enough.

2.1.5 Thread safety and reentrancy

When multiple processes share memory, the library must be careful about global state. A function is thread-safe if it can be called simultaneously from multiple processes (or threads) without corrupting shared data—typically by protecting global state with locks. A function is reentrant if it can be safely interrupted mid-execution and called again (e.g., from a signal or note handler)—a stronger property that requires using no global state at all, only stack-local variables and arguments. The distinction matters in practice. `malloc()` is thread-safe (it uses a lock internally), so two processes sharing memory can allocate concurrently. But `malloc()` is not reentrant: if a note handler calls `malloc()` while `malloc()` already holds its lock, the program deadlocks. Pure functions like `strlen()` or `sin()` are both thread-safe and reentrant, since they touch no global state. Functions like `atexit()` and `syslog()` are thread-safe (they use locks) but not reentrant. This distinction explains a recurring pattern in the code: locks appear throughout the library to ensure thread safety, and note delivery is deferred while critical locks are held to avoid reentrancy deadlocks.

This distinction was formalized in POSIX.1c (1995), which added threads to the UNIX standard and had to specify exactly which functions were safe to call from a signal handler. That work produced the published list of *async-signal-safe* functions and the `_r` reentrant variants for many non-safe ones (`strtok_r`, `asctime_r`, `getpwnam_r`). The canonical example of a function that violates both properties is `errno`, which was a plain global in V7 UNIX (1979) but is now a per-thread storage slot in glibc, musl, and Plan 9's `libc` alike.

2.1.6 The library that runs inside everything

One last perspective on why this book devotes hundreds of pages to what might seem like basic infrastructure. `libc` is not just "a library"—it is the most widely executed code on the machine. The kernel runs once at boot; each application runs sometimes; but `libc`'s `malloc`, `memcpy`, `printf`, and `read` are called billions of times a day by *every* process. A bug in `libc` is a bug in every program. An optimization in `libc` speeds up every program. A security flaw in `libc` is a security flaw in every program. This is why `libc` code is written with a level of care—and a level of paranoia about edge cases, thread safety, and signal safety—that would be overkill in any single application but is exactly right for code that runs inside everything.

2.2 `libc.h` interface

In Plan 9, there are no dozens of header files (`stdio.h`, `unistd.h`, `stdlib.h`, `string.h`, etc.) as in POSIX. Instead, almost everything is declared in a single header: `libc.h`. This, together with `u.h` (which defines basic types), is all that most Plan 9 programs include. Separate headers exist only for separate libraries: `thread.h` for `libthread`, `regexp.h` for `libregexp`, `draw.h` for `libdraw`, etc. This simplicity is deliberate: one header means one interface to learn, one place to look things up, and fast compilation (no redundant header processing).

2.3 Crash course: Plan 9 ARM assembly

Several chapters in this book include ARM assembly code. This section summarizes the key conventions of the Plan 9 assembler (see Assembler book [Pad15a] and Linker book [Pad15b] for the full details).

2.3.1 Registers

ARM has 16 registers, `R0` through `R15`. Some have conventional roles:

```
General purpose (conventions set by Plan 9, not hardware):
R0         first argument / return value
R1-R7      scratch (compiler register allocation)
R8-R10     may be used for extern registers (see below)
R11        reserved by linker (REGTMP, for instruction expansion)
R12        SB (static base) | reserved for global data access

Special (hardware-defined roles):
R13        SP (hardware stack pointer)
R14        LR (link register | return address, set by BL)
R15        PC (program counter)
```

Assembly files often define aliases for readability: `arg=0`, `sp=13`, `sb=12`, so that `R(sp)` means R13 and `R(arg)` means R0. The Plan 9 C compiler supports "extern register" variables that permanently occupy a high register (R8–R10). The kernel uses this for the `up` pointer (the current process), avoiding a memory load on every access. When extern registers are in use, those registers are unavailable for the compiler's register allocator.

A few registers are fixed by the ARM hardware: `R13` is the stack pointer, (the `PUSH`/`POP` and addressing modes that reference "SP" are hardwired to R13 by the CPU), `R14` is the link register (loaded by every `BL` instruction), and `R15` is the program counter (every branch rewrites R15). The other assignments—`R0` for the first argument and return value, `R12` as the static-data base (`SB`)—are calling conventions chosen by the Plan 9 compiler team. They are not universal: Linux/ARM uses a different static base strategy, and other compilers may allocate registers differently.

2.3.2 Pseudo-registers

The Plan 9 assembler introduces three pseudo-registers. They are called "pseudo" because the programmer uses them as abstract names, and the assembler/linker resolves them to offsets from real hardware registers (`SB` maps to `R12`, `FP` and `SP` map to `R13`):

- `SB` (static base): used to reference global variables. The name follows the ARM addressing notation `offset(base)`: just as `4(R13)` means "the word at R13 + 4," `_tos(SB)` means "the word at SB + the offset of `_tos`." It is "static" because global data does not move at runtime (unlike the stack), and "base" because it is the base address from which all globals are reached. The linker resolves `SB` to `R12`.

- `FP` (frame pointer): used to reference the *caller's* arguments. `4(FP)` means "the second argument passed to this function." Offsets go by 4 because ARM is a 32-bit architecture: the natural unit (a "word") is 4 bytes, and each argument occupies one word. So the first argument is at `0(FP)`, the second at `4(FP)`, the third at `8(FP)`. The assembler resolves `FP` references to offsets from `R13` based on the frame size.

- `SP` (pseudo stack pointer): used to reference the current function's *local* storage. Not to be confused with `R(sp)` (or equivalently `R13`) which is the actual hardware stack pointer. In practice they point to the same place, but `4(SP)` is resolved by the assembler while `4(R(sp))` is a raw hardware addressing mode.

A typical stack frame for a function with two arguments and two local variables looks like:

```
High addresses
+-------------------+
| arg 2             |  4(FP)   -- caller's arguments
+-------------------+
| arg 1 (or empty)  |  0(FP)   -- may be in R0 instead
+-------------------+
| return address    |           -- saved by BL
+-------------------+
| local 2           |  4(SP)   -- current function's locals
+-------------------+
| local 1           |  0(SP)
+-------------------+  <-- R13 (hardware stack pointer)
Low addresses
```

For a concrete example, consider:

```
int bar(int a, int b) {
    int tmp = a + b;
    return tmp;
}
```

When `foo` calls `bar(10, 20)`, the stack looks like:

```
+----------+
| 20       |  b+4(FP)        second arg, on stack
+----------+
|          |  a+0(FP)        empty (reserved for first arg)
+----------+
| ret addr |                  saved by BL instruction
+----------+
| tmp      |  tmp+0(SP)      local variable
```

```
+----------+  <-- R13

R0 = 10    (first argument, passed in register)
```

Note that the names before the offsets (`a`, `b`, `tmp`) are for documentation and the debugger's symbol table only—the assembler ignores them. What matters is the numeric offset. The linker converts pseudo-register references to concrete `R13` offsets when generating the final machine code. For `bar`, with a frame size of 4 (one local), the conversion is:

```
Pseudo-register       Concrete (R13-relative)
---------------       -----------------------
tmp+0(SP)       -->   0(R13)            local
(return addr)         4(R13)
a+0(FP)         -->   8(R13)            first arg
b+4(FP)         -->   12(R13)           second arg
```

2.3.3 Key directives and notations

In Plan 9 assembly, the `$` prefix turns a memory reference into a literal operand. Without `$`, the value is loaded *from* memory; with `$`, the address or constant itself is used. For example: `MOVW foo(SB), R1` loads the *contents* of global `foo` into `R1`, while `MOVW $foo(SB), R1` loads the *address* of `foo` into `R1`. Similarly, `MOVW $42, R1` loads the constant 42.

Key assembler directives:

- `TEXT name(SB), flags, $framesize` — declares a function. The `framesize` is how much stack space to reserve for locals *and* for any arguments passed to functions this function calls (the assembler adjusts R13 on entry).
- `DATA sym+offset(SB)/size, $value` — defines initialized global data.
- `GLOBL sym+0(SB), $size` — declares the total size of a global symbol.
- `BL` — branch and link (function call; saves return address in R14).
- `RET` — return (branches to R14).
- `SWI $0` — software interrupt (system call trap).
- `MOVM.IA.W` — move multiple registers (increment after, write back); used for fast block copies.

Note that the Plan 9 assembler uses *virtual instructions* that do not correspond one-to-one with ARM machine instructions. The linker translates them into real ARM code. For example:

- `MOVW` is a generic "move word"—the linker emits `LDR` (load from memory), `STR` (store to memory), `MOV` (register-to-register), or `ADD` depending on the operand types.
- `RET` is not a real ARM instruction—it translates to `MOV R14, R15` (copy the link register to the program counter). See Assembler book [Pad15a] and Linker book [Pad15b] for the full details.
- `CMP` with a large constant may be expanded into a load from a literal pool plus a compare.

This abstraction means the assembly code in this book reads more like pseudocode than raw ARM machine code. See Linker book [Pad15b] for how the translation works.

ARM instructions can have suffixes that modify their behavior:

- *Conditional suffixes*: `.EQ`, `.LT`, `.GT`, `.HS` (higher or same), etc. The instruction executes only if the condition flags match. For example, `BEQ label` branches only if the previous comparison was equal. This avoids short branches and is used extensively in the assembly code.

- *Post-increment*: `.P` (or `.W` for write-back). `MOVBU.P 1(R1), R2` loads a byte from `R1` and then increments `R1` by 1. This combines a load and a pointer advance in a single instruction, making byte-at-a-time loops more compact.

- *Byte/halfword*: `.B` or `.BU` for byte access (`U` = unsigned). `MOVBU` loads a single unsigned byte.

- *Set flags*: `.S` makes an arithmetic instruction set the condition flags. `AND.S` performs a bitwise AND and sets the zero flag.

2.3.4 The static base (SB) and setR12

Global variables are accessed as offsets from `R12` (the static base register). To understand why, consider how a Plan 9 program is laid out in memory:

```
+-------------------+
| BSS (zeroed)      |  uninitialized globals
+-------------------+
| Data              |  initialized globals ("hello", tables)
+-------------------+  <-- SB (= R12) points here = INITDAT
| Text              |  machine code
+-------------------+  <-- entry point
```

The data section comes *after* the text section. The reason for using a dedicated register is an ARM instruction encoding constraint: load/store instructions only support a 12-bit offset (~4096 bytes). Without a base register, accessing a global variable would require a multi-instruction sequence to build a full 32-bit address. With `R12` holding `INITDAT`, each global access is written as `MOVW foo(SB), R1` in the assembly source, and the linker translates it to a single ARM `LDR` instruction using `R12` as the base register. This also helps the linker compute instruction sizes during layout, since all global references have a predictable encoding (see Linker book [Pad15b]).

But how does the running program learn the value of `INITDAT`? The linker knows it, but there is no general mechanism to pass a constant from the linker to the program. The trick is a form of introspection: the linker creates a symbol called `setR12` at offset 0 in the data section, so the *address* of `setR12` equals `INITDAT`. The instruction `MOVW $setR12(SB), R12` means "load the address of `setR12` into `R12`" (recall that `$` turns a memory reference into a literal)—and that address *is* `INITDAT`. This is similar to how the linker defines `etext`, `edata`, and `end`—symbols whose addresses mark the boundaries between text, data, and BSS. C programs can reference these as `extern char end[]` to find where the heap starts (used by `sbrk()`[58a] in Chapter 7).

There is still a bootstrap problem: the `MOVW $setR12(SB), R12` instruction uses an `SB` reference, but `R12` is not yet set. The linker recognizes this specific instruction and emits an absolute load (e.g., `MOV $0x7000, R12`) with `INITDAT` hard-coded, bypassing the normal `R12`-relative encoding. Why not use absolute loads for *all* globals, eliminating the need for `R12` entirely? Because an absolute 32-bit load on ARM requires multiple instructions (ARM instructions are 32 bits wide, so a 32-bit constant cannot fit alongside the opcode). The `R12`-relative form fits in a single instruction as long as the offset is small enough, which is the common case. After this bootstrap (which we will see in `_main` in Chapter 5), all `xxx(SB)` references work. See Linker book [Pad15b] for the full details.

2.3.5 Calling conventions

The Plan 9 C calling convention on ARM passes the first argument in `R0` and the remaining arguments on the stack. The return value comes back in `R0`. All registers are caller-saved: a called function may freely overwrite any register, so the caller must save to the stack any register value it needs after the call. There are no callee-saved registers (unlike the standard ARM AAPCS convention which preserves R4–R11). The caller also reserves space on the stack for *all* arguments, including the first one—even though it is passed in `R0`. This leaves an empty slot at `0(FP)` that many functions use to save `R0` early on (freeing it for other uses):

```
Caller's stack frame for f(a, b, c):

+-------------------+
| c                 |  8(FP)   -- third arg (on stack)
+-------------------+
| b                 |  4(FP)   -- second arg (on stack)
+-------------------+
| a (empty slot)    |  0(FP)   -- reserved but not filled
+-------------------+                (a is in R0 instead)
| return address    |
+-------------------+
```

The callee typically starts with `MOVW R0, 0(FP)` to save the first argument into its reserved slot, so that all arguments are on the stack in a uniform layout. This is why the syscall stubs (Chapter 4) begin with this instruction.

System calls use a different convention: the syscall number replaces the first argument in `R0`, and `SWI $0` traps into the kernel. The kernel reads all arguments from the user's stack frame (including the first, which the stub saved to `0(FP)` before overwriting `R0` with the syscall number). See Chapter 4 for details.

2.4 Code organization

Table 2.1 presents short descriptions of the source files of the Plan 9 core libraries, together with the main entities (e.g., structures, functions) the file defines, and the corresponding chapters in this document in which the code contained in the file is primarily discussed. The code is split into several libraries and subdirectories: `libc/port/` for portable C implementations, `libc/9sys/` for system call wrappers and OS-specific code, `libc/fmt/` for formatted output, `libc/arm/` (or `libc/386/`) for architecture-specific assembly, `libbio/` for buffered I/O, and `libthread/` for cooperative threads and channels.

One header not shown above is `lib_core/libc/9syscall/sys.h`, which defines the numeric syscall codes (`EXITS=3`, `OPEN=6`, `PREAD=8`, etc.) used by the assembly stubs in `9syscall/`. It is presented in Kernel book [Pad14] (since the kernel and the stubs must agree on these numbers). The C-level declarations of all syscalls (`open`, `exits`, `rfork`, etc.) live in `syscall.h`, which is included by `libc.h`: this allows the C compiler to type-check every call site against the correct prototype, even though the actual implementations are assembly stubs.

2.5 Software architecture

Unlike a compiler or a build system, a library is not a pipeline. It is a collection of largely independent components that programs link against and call as needed. The "architecture" of a C library is therefore better understood as a layering:

1. *System call wrappers* (in `9sys/` and `9syscall/`) provide thin assembly stubs that trap into the kernel for `open`, `read`, `write`, `rfork`, `exec`, `brk`, `rendezvous`, etc.

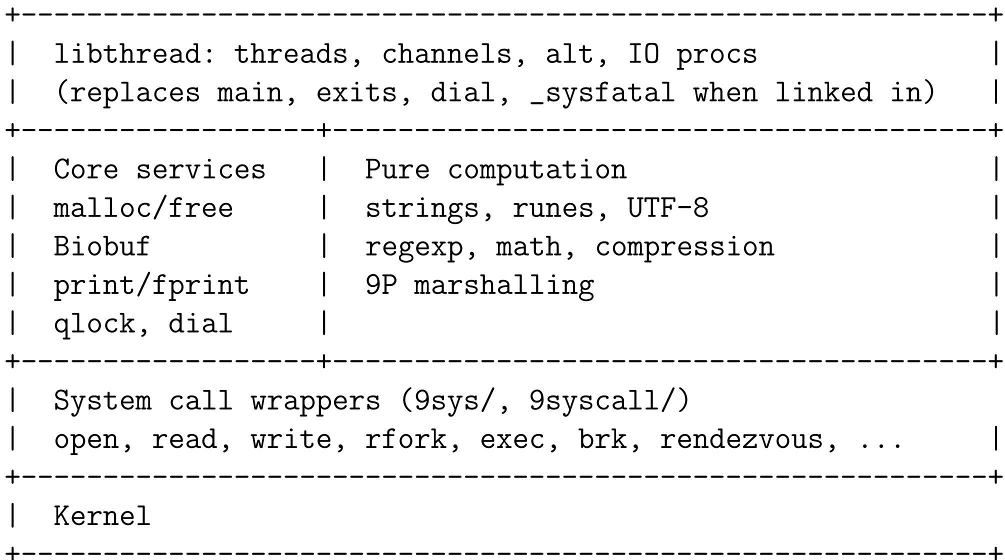
2. *Core services* built on those system calls: the memory allocator (`malloc`/`free` on top of `sbrk`), buffered I/O (`Biobuf` on top of `read`/`write`), formatted output (`print` on top of `write`), locking (`qlock` on top of `rendezvous`), and networking (`dial`/`announce` on top of `open`/`read`/`write` to `/net/`).

3. *Pure computation*: string manipulation, math functions, regular expressions, and 9P marshalling. These do not use system calls at all—though they may call `malloc`, which itself may call the `sbrk` system call to grow the heap.

4. `libthread`: a layer on top of everything else, providing cooperative threads, channels, and IO procs. When linked in, `libthread` replaces `main`, `exits`, `dial`, `_assert`, and `_sysfatal` with thread-aware versions.

The key dependency is that nearly everything depends on the memory allocator, and the allocator itself depends on locks. In Plan 9, `rfork(RFMEM)` creates a new process that shares its parent's memory—effectively a system-level thread. Since two such processes can call `malloc` concurrently, the allocator must be thread-safe (see the discussion of thread safety versus reentrancy in the Principles section). More generally, `libc` functions that use global state (e.g., `syslog`, `atexit`) protect it with locks, making Plan 9's `libc` safe for use with shared-memory processes. `libthread`'s channels add another layer, depending on locks, atomic operations, and the `rendezvous` system call.

The following diagram summarizes this layering:



```
+-------------------------------------------------------------+
|  libthread: threads, channels, alt, IO procs                |
|  (replaces main, exits, dial, _sysfatal when linked in)     |
+------------------+------------------------------------------+
|  Core services   |  Pure computation                        |
|  malloc/free     |  strings, runes, UTF-8                   |
|  Biobuf          |  regexp, math, compression               |
|  print/fprint    |  9P marshalling                          |
|  qlock, dial     |                                          |
+------------------+------------------------------------------+
|  System call wrappers (9sys/, 9syscall/)                      |
|  open, read, write, rfork, exec, brk, rendezvous, ...       |
+-------------------------------------------------------------+
|  Kernel                                                       |
+-------------------------------------------------------------+
```

2.6 Book structure

You now have enough background to understand the source code of the Plan 9 core libraries. The rest of the book is organized in five parts. Part I (*Foundations*) covers the bedrock: the core types and data structures in Chapter 3, the system call interface in Chapter 4, the C runtime entry point `_main()` in Chapter 5, memory management (Chapter 6 for low-level operations, Chapter 7 for the `malloc` allocator), and formatted output in Chapter 8 (the extensible `print` family used everywhere, including the kernel). Part II (*Data Processing*) covers pure computation: strings in Chapter 9 (C strings, runes, and UTF-8), regular expressions in Chapter 10, mathematics in Chapter 11, and compression in Chapter 12. Part III (*System Services*) covers the library functions that wrap system calls, roughly one family per chapter: file I/O in Chapter 13, buffered I/O in Chapter 14, directories in Chapter 15, namespace operations in Chapter 16, time in Chapter 17, process management in Chapter 18, error management in Chapter 19, and security in Chapter 20. Part IV (*Concurrency and Communication*) covers synchronization and IPC: locks and queuing locks in Chapter 21, IPC mechanisms (notes, pipes, 9P) in Chapter 22, the cooperative thread scheduler and channels in Chapter 23 (the largest chapter in the book), and networking in Chapter 24. Part V (*Debugging and Profiling*) covers the library support for the

Plan 9 debugging and profiling tools in Chapters 25 and 26. Finally, Chapter 27 concludes and gives pointers to other books in the Principia Softwarica series.

Function	Ch.	File	Entities	LOC
basic types	3	arm/u.h	Rune[30a] FPdbleword[27d]	80
libc interface	3	libc.h	Fmt[71b] Tm[36] Lock[245a] QLock[246a]	500
character classification	3	port/ctype.c	isalphaX toupper[29b]	30
sorting	3	port/qsort.c	qsort[32a]	130
syscall interface	4	syscall.h	DirX Qid[191a] Rfork_flagX	100
memory operations	6	port/memset.c etc.	memset[46b] memmove[48]	90
pool interface	7	pool.h	Pool[58c] Pool_flagX	100
heap allocator	7	port/pool.c	Bhdr[60] Free[62a] Arena[62e]	800
malloc wrappers	7	port/malloc.c etc.	malloc[63g] free[63i] sbrk[58a]	100
formatted output	8	fmt/dofmt.c etc.	fprint[75a] sprint[75b] Fmt	1300
C strings	9	port/strlen.c etc.	strlen[80a] strcmp[81a] strcpy[81c]	210
rune strings	9	port/rune.c etc.	chartorune[84d] runetochar[85]	330
UTF-8 strings	9	port/utflen.c etc.	utflen[89a] utfrune[89b]	90
regexp interface	10	regexp.h	Reprog[105b] Resub[106c]	40
regexp compilation	10	libregexp/regcomp.c	regcomp[107a] Reinst[106a]	660
regexp execution	10	libregexp/regexec.c etc.	regexec[109] regsub[113]	540
trigonometry	11	port/sin.c etc.	sin[141b] exp[149a] sqrt[137a]	700
parsing	11	port/atol.c etc.	atol[123c] strtol[124c] strtod[128]	500
random numbers	11	port/lrand.c	lrand[153c] nrand[407b]	80
deflate	12	libflate/deflate.c	deflate[180] LZstate	1360
inflate	12	libflate/inflate.c etc.	inflateX crcX	1050
file I/O	13	9sys/read.c etc.	read[192a] write[192b] readn[192c]	40
buffered I/O interface	14	bio.h	Biobuf[195a]	50
buffered I/O init	14	libbio/binit.c	Binit[197a]	136
buffered I/O reading	14	libbio/brdline.c etc.	Brdline[202] Bgetc[200a]	310
buffered I/O writing	14	libbio/bwrite.c etc.	Bwrite[208] Bprint[209a]	150
directory operations	15	9sys/dirstat.c etc.	dirstat[216b] nulldir[214d]	150
time	17	9sys/time.c etc.	time[226a] localtime[225a]	300
process management	18	9sys/fork.c etc.	fork[231a] execl[231b] wait[256a]	80
error management	19	9sys/rerrstr.c etc.	rerrstr[234a] werrstr[234b]	80
security	20	port/crypt.c etc.	encrypt[238a] getuser[237c]	180
locks	21	port/lock.c	lock[245b] qlock[246c] rsleep[251b]	380
9P marshalling	22	9sys/convM2S.c etc.	convM2S[267] convS2M[273]	700
IPC helpers	22	9sys/fcallfmt.c etc.	fcallfmt[263c] getenv[257a]	330
thread interface	23	thread.h	Channel[288e] Alt[300a] Ref[287a]	100
thread scheduler	23	libthread/sched.c	_schedX _threadreadyX	250
thread creation	23	libthread/create.c	threadcreate[296b] proccreate[293g]	200
channels	23	libthread/channel.c	alt[306] send[313a] recv[313b]	600
IO procs	23	libthread/ioproc.c	ioproc[316b] iocall[317]	300
thread-aware libc	23	libthread/main.c etc.	threadexits[321c] procexec[324b]	700
networking	24	9sys/announce.c etc.	announce[344] listen[346a] dial[347b]	500
logging	25	9sys/syslog.c	syslog[354c]	100
profiling	26	port/profile.c	_profinX _profoutX _profdumpX	280
Total				~15000

Table 2.1: Chapters and associated core library source files.

Part I

Foundations

This part covers the bedrock that everything else is built on: the type system (`u.h` and `libc.h`), the C runtime entry point (`_main`), the system call interface to the kernel, memory management—from the raw `sbrk` system call through the `Pool` allocator to `malloc`/`free`—and formatted output (the extensible `print` family).

Chapter 3

Core Types and Data Structures

This chapter presents the fundamental type definitions from `u.h` and `libc.h` that every Plan 9 C program relies on. These headers establish the vocabulary of types—from basic integers and booleans to Unicode runes and file descriptors—that the rest of the library builds upon.

3.1 Basic types

C provides a few primitive types (`int`, `char`, `double`, pointers), but a standard library typically extends these with aliases for portability, convenience, and clarity. This section covers the type definitions in `u.h` and `libc.h`: booleans, fixed-width integers, Unicode characters (runes), and floating-point bit manipulation.

3.1.1 Booleans

Traditional C has no boolean type—programmers use `int` with 0 for false and non-zero for true. C99 introduced `_Bool` (with `<stdbool.h>`), and C23 made `bool` a keyword. The original Plan 9 libraries did not define a boolean type, but the author added this `typedef` to make function signatures more readable. Go, Plan 9's spiritual successor, has a proper `bool` from the start.

⟨*type* `bool` 26a⟩≡ (368b)

```
typedef int bool;
enum _bool {
  false = 0,
  true = 1
};
```

3.1.2 Integers

Plan 9 defines shorthand aliases for unsigned types and fixed-width integers. The aliases exist because C's standard types have platform-dependent sizes: `long` is 32 bits on 32-bit architectures but 64 bits on 64-bit architectures under gcc/clang (the LP64 model), which is a perennial source of porting bugs. The `vlong` type ("very long") provides a guaranteed 64-bit integer regardless of architecture. The `u8int`, `u16int`, `u32int`, `u64int` types provide explicit widths, similar to C99's `<stdint.h>` types but predating them.

⟨*type* `uxxx` 26b⟩≡ (368a)

```
typedef unsigned short  ushort;
typedef unsigned char   uchar;
typedef unsigned long   ulong;
typedef unsigned int    uint;
```

⟨*type* `byte` 26c⟩≡ (368b)

```
typedef uchar byte;
```

⟨*type* `uxxxint` 27a⟩≡ (368a)

```
typedef unsigned char u8int;
typedef unsigned short u16int;
typedef unsigned int    u32int;
typedef unsigned long long u64int;
```

These typedefs live in `arm/u.h` (not `libc.h`) because they are architecture-specific. The Plan 9 C compiler keeps `long` at 32 bits on all architectures, but when porting to gcc/clang (as in plan9port or Goken), `long` becomes 64 bits on 64-bit hosts (the LP64 model). The explicit-width types like `u32int` guarantee exactly 32 bits regardless of compiler, avoiding subtle bugs when the same code is compiled for different targets.

⟨*type* `xxxvlong` 27b⟩≡ (368a)

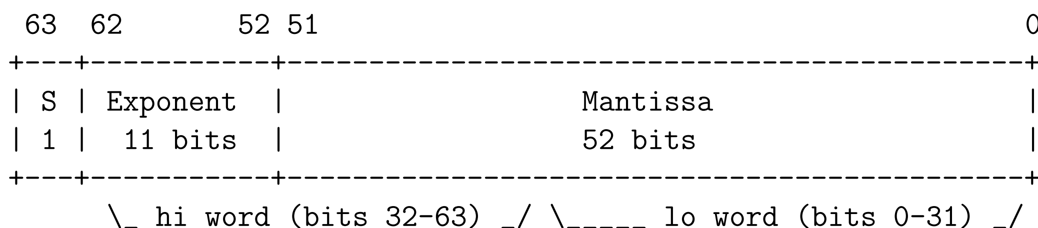
```
typedef long long   vlong;
typedef unsigned long long uvlong;
```

⟨*type* `usize` 27c⟩≡ (368a)

```
typedef unsigned long   usize;
```

3.1.3 Floats

`FPdbleword` is a union that overlays a `double` with its raw bit representation as two 32-bit words. This is essential for the math library: functions like `NaN()`[126a], `isNaN()`[126d], and `isInf()`[127a] need to inspect or construct the IEEE 754 bit pattern directly (exponent, mantissa, sign) rather than treating the value as a number. An IEEE 754 double-precision float is laid out as follows:



The `hi` word of `FPdbleword` contains the sign bit, the 11-bit exponent, and the upper 20 bits of the mantissa. The `lo` word contains the lower 32 bits of the mantissa.

⟨*type* `FPdbleword` 27d⟩≡ (368a)

```
union FPdbleword
{
    double  x;

    struct {    /* little endian */
        ulong lo;
        ulong hi;
    };
};
```

Note the `/* little endian */` comment: `lo` comes before `hi` in the struct because ARM stores the least significant word at the lower address. On a big-endian architecture, the fields would be reversed. This is why `FPdbleword` lives in `arm/u.h` rather than the portable headers.

Recall that a double encodes the value $(-1)^S \times 1.M \times 2^{E-1023}$, where S is the sign bit, E is the 11-bit exponent, and M is the 52-bit mantissa (the fractional part after an implicit leading 1). The exponent is stored with a bias of 1023: the actual exponent is $E - 1023$. The bias makes the stored exponent always non-negative (an unsigned integer), which simplifies hardware comparison—you can compare two floats by comparing their bit patterns as integers.

For example:

```
Value    S  Exponent (E)  Mantissa (M)   Meaning
-----    -  ------------  ------------   -------
 1.0     0  01111111111   000...000      1.0 x 2^(1023-1023) = 1
-1.0     1  01111111111   000...000      -(1.0 x 2^0) = -1
 0.5     0  01111111110   000...000      1.0 x 2^(1022-1023) = 0.5
 3.14    0  10000000000   100100011...   1.100100011... x 2^1 = 3.14
 NaN     0  11111111111   000...001      exp all 1s, mantissa != 0
+Inf     0  11111111111   000...000      exp all 1s, mantissa = 0
```

For 3.14: in binary $3.14 \approx 11.001000111\ldots$, which normalizes to $1.1001000111\ldots \times 2^1$. The exponent is $1 + 1023 = 1024$ (binary `10000000000`), and the mantissa stores `1001000111...` (the part after the implicit leading 1).

3.1.4 Strings and characters

Text in C is represented at three levels: bytes (ASCII characters), code points (Unicode characters), and encoded byte sequences (UTF-8). Most C libraries blur these distinctions, but Plan 9 keeps them explicit: a `char` is always a byte, a `Rune` is always a code point, and the `utf*` functions handle the encoding between them.

ASCII characters

Character classification uses a 256-entry lookup table (`_ctype`[28b]) where each byte is a bitmask of properties: upper, lower, digit, space, punctuation, control. The `isalpha`, `isdigit`, etc. macros index into this table, making classification a single array lookup with no branches.

⟨*type* `Ctype_flag` 28a⟩≡ (379a)

```
#define _U 01 // upper
#define _L 02 // lower
#define _N 04 // number
#define _S 010 // space
#define _P 020 // punctuation
#define _C 040 // ctlr
#define _B 0100 // ??
#define _X 0200 // valid hex digit
```

⟨*global* `_ctype` 28b⟩≡ (393d)

```
uchar   _ctype[256] =
{
/*   0   1   2   3   4   5   6   7  */

/*  0*/ _C, _C, _C, _C, _C, _C, _C, _C,
/* 10*/ _C, _S|_C,  _S|_C,  _S|_C,  _S|_C,  _S|_C,  _C, _C,
/* 20*/ _C, _C, _C, _C, _C, _C, _C, _C,
/* 30*/ _C, _C, _C, _C, _C, _C, _C, _C,
/* 40*/ _S|_B,  _P, _P, _P, _P, _P, _P, _P,
/* 50*/ _P, _P, _P, _P, _P, _P, _P, _P,
/* 60*/ _N|_X,  _N|_X,  _N|_X,  _N|_X,  _N|_X,  _N|_X,  _N|_X,  _N|_X,
/* 70*/ _N|_X,  _N|_X,  _P, _P, _P, _P, _P, _P,
/*100*/ _P, _U|_X,  _U|_X,  _U|_X,  _U|_X,  _U|_X,  _U|_X,  _U,
/*110*/ _U, _U, _U, _U, _U, _U, _U, _U,
/*120*/ _U, _U, _U, _U, _U, _U, _U, _U,
/*130*/ _U, _U, _U, _P, _P, _P, _P, _P,
/*140*/ _P, _L|_X,  _L|_X,  _L|_X,  _L|_X,  _L|_X,  _L|_X,  _L,
/*150*/ _L, _L, _L, _L, _L, _L, _L, _L,
/*160*/ _L, _L, _L, _L, _L, _L, _L, _L,
/*170*/ _L, _L, _L, _P, _P, _P, _P, _C,
};
```

For example, the character `'A'` has ASCII value 65. To find it in the table above, note that the row/column indices are in *octal*: 65 decimal is 101 octal, so row `/*100*/` (octal 100 = decimal 64), column 1. That entry is `_U|_X` (uppercase and hex digit). This means `isalpha('A')` tests `(_U|_X) & (_U|_L)`, which is true because the `_U` bit is set. `isdigit('A')` tests `(_U|_X) & _N`, which is false.

⟨*macros isxxx* 29a⟩≡ (379a)

```
#define isalpha(c) (_ctype[(unsigned char)(c)]&(_U|_L))
#define isupper(c) (_ctype[(unsigned char)(c)]&_U)
#define islower(c) (_ctype[(unsigned char)(c)]&_L)
#define isdigit(c) (_ctype[(unsigned char)(c)]&_N)
#define isxdigit(c) (_ctype[(unsigned char)(c)]&_X)
#define isspace(c) (_ctype[(unsigned char)(c)]&_S)
#define ispunct(c) (_ctype[(unsigned char)(c)]&_P)
#define isalnum(c) (_ctype[(unsigned char)(c)]&(_U|_L|_N))
#define isprint(c) (_ctype[(unsigned char)(c)]&(_P|_U|_L|_N|_B))
#define isgraph(c) (_ctype[(unsigned char)(c)]&(_P|_U|_L|_N))
#define iscntrl(c) (_ctype[(unsigned char)(c)]&_C)
#define isascii(c) ((unsigned char)(c)<=0177)
```

The `toupper`/`tolower` functions take `int` rather than `char`. In C, `char` is automatically promoted to `int` when passed to a function, so these work transparently with both `char` and `Rune` arguments. For ASCII-range characters (below 128), the conversion is a simple offset between the uppercase and lowercase ranges.

⟨*function* `toupper` 29b⟩≡ (455a)

```
int
toupper(int c)
{

    if(c < 'a' || c > 'z')
        return c;
    return _toupper(c);
}
```

⟨*macro* `_toupper` 29c⟩≡ (379a)

```
#define _toupper(c) ((c)-'a'+'A')
```

⟨*function* `tolower` 29d⟩≡ (455a)

```
int
tolower(int c)
{

    if(c < 'A' || c > 'Z')
        return c;
    return _tolower(c);
}
```

⟨*macro* `_tolower` 29e⟩≡ (379a)

```
#define _tolower(c) ((c)-'A'+'a')
```

`toascii()` strips the high bit, forcing any byte into the 7-bit ASCII range (0–127). This is a blunt tool for sanitizing input that might contain 8-bit characters, but it destroys UTF-8 multi-byte sequences and is rarely useful in Plan 9 where UTF-8 is the native encoding.

⟨*macro* `toascii` 29f⟩≡ (379a)

```
#define toascii(c) ((c)&0177)
```

Runes

A `Rune` is Plan 9's name for a Unicode code point—a 21-bit integer (stored in a 32-bit `uint`) that can represent any character in the Unicode standard (up to U+10FFFF). The term "rune" was coined by Rob Pike and Ken Thompson when they designed UTF-8 at Bell Labs. The name evokes the ancient Germanic runes—symbols that carry meaning in a single glyph, just as a `Rune` carries a single character regardless of how many bytes it takes to encode.

In Plan 9, `char` means a byte and `Rune` means a character; the UTF-8 functions convert between the two representations:

```
Char        Rune (code point)   UTF-8 bytes       char[]
----        -----------------   -----------       ------
A              U+0041 (65)         0x41              1 byte
e-acute        U+00E9 (233)        0xC3 0xA9         2 bytes
Greek beta     U+03B2 (946)        0xCE 0xB2         2 bytes
```

The first example is plain ASCII—a single byte in all representations. The second is the French letter é (e with acute accent)—it looks similar to `e` but is a distinct Unicode character (code point 233, above the ASCII range of 0–127), requiring 2 bytes in UTF-8. The third is the Greek letter β, also 2 bytes. Characters from other scripts (Chinese, Japanese, Arabic) require 3 bytes, and rare characters (emoji, historic scripts) require 4.

⟨*type* `Rune` 30a⟩≡ (368a)
```
typedef uint        Rune;
```

⟨*constant* `Runemax` 30b⟩≡ (368b)
```
Runemax     = 0x10FFFF, /* 21-bit rune */
```

⟨*constant* `Runemask` 30c⟩≡ (368b)
```
Runemask    = 0x1FFFFF, /* bits used by runes (see grep) */
```

UTF8

UTF-8 is the variable-length encoding of runes into byte sequences (1 to 4 bytes), designed by Ken Thompson and Rob Pike. Plan 9 was the first operating system with native UTF-8 support throughout the entire system. The encoding was designed under strict constraints: backward compatibility with ASCII (so existing C programs work unchanged), no null bytes inside multi-byte sequences (so `strlen`, `strcpy`, etc. still work), self-synchronization (from any byte position, you can find the start of the current character by scanning for a byte that does not begin with `10` in binary), and byte-order matching code-point order (so `strcmp` on UTF-8 strings gives the same ordering as comparing code points directly). The key constants are: `UTFmax` (4 bytes, the maximum encoding length), `Runeself` (0x80, below which UTF-8 is identical to ASCII), and `Runeerror` (the replacement character for invalid sequences).

⟨*constant* `UTFmax` 30d⟩≡ (368b)
```
UTFmax      = 4,        /* maximum bytes per rune */
```

⟨*constant* `Runesync` 30e⟩≡ (368b)
```
Runesync    = 0x80,     /* cannot represent part of a UTF sequence (<) */
```

⟨*constant* `Runeself` 30f⟩≡ (368b)
```
Runeself    = 0x80,     /* rune and UTF sequences are the same (<) */
```

⟨*constant* `Runeerror` 30g⟩≡ (368b)
```
Runeerror   = 0xFFFD,   /* decoding error in UTF */
```

3.2 Complex types

Beyond the scalar types above, the library defines a few macros and typedefs related to pointers, arrays, structures, and function calling conventions.

3.2.1 Pointers

Plan 9 uses `nil` instead of C's `NULL` because it is shorter and, unlike `NULL`, is always defined as `((void*)0)`—some C implementations define `NULL` as plain `0`, which can cause subtle bugs when passed to variadic functions (where the compiler cannot infer that an integer 0 should be a pointer).

⟨*constant* `nil` 31a⟩≡ (368a)
```
#define nil     ((void*)0)
```

The `uintptr` type is an integer wide enough to hold a pointer—critical for code that needs to do arithmetic on addresses. You cannot do arbitrary arithmetic on C pointers directly (e.g., masking the low bits for alignment, or XOR-ing two pointers for a hash); you must cast to `uintptr` first. For example, `qsort` checks alignment with `((uintptr)va + es) % sizeof(long)` to decide whether to use word-sized or byte-sized swaps.

⟨*type* `uintptr` 31b⟩≡ (368a)
```
typedef unsigned long   uintptr;
```

3.2.2 Arrays (and `qsort()`)

`nelem()` computes the number of elements in a statically allocated array at compile time—a safer alternative to hardcoding array sizes.

⟨*function* `nelem` 31c⟩≡ (368b)
```
#define nelem(x)    (sizeof(x)/sizeof((x)[0]))
```

For example, given `int table[16]`, writing `for(i=0; i<nelem(table); i+)+` is safer than hardcoding `i<16`—if the array size changes later, `nelem` adapts automatically. Note that `nelem` only works on actual arrays, not on pointers: if `table` is passed to a function as `int *table`, `nelem` would give the wrong answer (the size of a pointer divided by the size of an int).

The `qsort` implementation uses quicksort. The naive version picks the middle element as pivot, but that degrades to $O(n^2)$ on certain inputs (e.g., already-sorted arrays). This implementation uses a *median-of-three* pivot, sampling at positions 1/6, 1/2, and 5/6 of the array and choosing the median—a good approximation of the true median that avoids worst-case behavior on common inputs. After partitioning, it recurses on the smaller partition and loops on the larger, which limits stack depth to $O(\log n)$ even in the worst case. Note also the two swap functions: `swapi()`[32c] swaps in `long`-sized chunks (fast, for aligned data) and `swapb()`[32b] swaps byte-by-byte (for unaligned data). `qsort` checks alignment to choose between them.

⟨*type* `Sort` 31d⟩≡ (432c)
```
typedef
struct
{
    int (*cmp)(void*, void*);
    void    (*swap)(char*, char*, long);
    long    es;
} Sort;
```
Uses `__anon_struct_14` 31d.

⟨*function* qsort 32a⟩≡ (432c)
```
void
qsort(void *va, long n, long es, int (*cmp)(void*, void*))
{
    Sort s;

    s.cmp = cmp;
    s.es = es;
    s.swap = swapi;
    if(((uintptr)va | es) % sizeof(long))
        s.swap = swapb;
    qsorts((char*)va, n, &s);
}
```
Uses `qsorts()` 33, `swapb()` 32b, and `swapi()` 32c.

⟨*function* swapb 32b⟩≡ (432c)
```
static  void
swapb(char *i, char *j, long es)
{
    char c;

    do {
        c = *i;
        *i++ = *j;
        *j++ = c;
        es--;
    } while(es != 0);

}
```

⟨*function* swapi 32c⟩≡ (432c)
```
static  void
swapi(char *ii, char *ij, long es)
{
    long *i, *j, c;

    i = (long*)ii;
    j = (long*)ij;
    do {
        c = *i;
        *i++ = *j;
        *j++ = c;
        es -= sizeof(long);
    } while(es != 0);
}
```

⟨*function* pivot 32d⟩≡ (432c)
```
static  char*
pivot(char *a, long n, Sort *p)
{
    long j;
    char *pi, *pj, *pk;

    j = n/6 * p->es;
    pi = a + j; /* 1/6 */
    j += j;
    pj = pi + j;    /* 1/2 */
    pk = pj + j;    /* 5/6 */
    if(p->cmp(pi, pj) < 0) {
        if(p->cmp(pi, pk) < 0) {
```

```
            if(p->cmp(pj, pk) < 0)
                return pj;
            return pk;
        }
        return pi;
    }
    if(p->cmp(pj, pk) < 0) {
        if(p->cmp(pi, pk) < 0)
            return pi;
        return pk;
    }
    return pj;
}
```

⟨*function* qsorts 33⟩≡ (432c)

```
static  void
qsorts(char *a, long n, Sort *p)
{
    long j, es;
    char *pi, *pj, *pn;

    es = p->es;
    while(n > 1) {
        if(n > 10) {
            pi = pivot(a, n, p);
        } else
            pi = a + (n>>1)*es;

        p->swap(a, pi, es);
        pi = a;
        pn = a + n*es;
        pj = pn;
        for(;;) {
            do
                pi += es;
            while(pi < pn && p->cmp(pi, a) < 0);
            do
                pj -= es;
            while(pj > a && p->cmp(pj, a) > 0);
            if(pj < pi)
                break;
            p->swap(pi, pj, es);
        }
        p->swap(a, pj, es);
        j = (pj - a) / es;

        n = n-j-1;
        if(j >= n) {
            qsorts(a, j, p);
            a += (j+1)*es;
        } else {
            qsorts(a + (j+1)*es, n, p);
            n = j;
        }
    }
}
```

Uses `qsorts()` 33.

3.2.3 Structures

`offsetof()` computes the byte offset of a field within a structure at compile time. It works by casting `nil` to a pointer to the structure and taking the address of the field—the resulting address is the offset from zero. This macro is used by container-of idioms: given a pointer `p` to a `next` field inside a `Free` block, you can recover the enclosing struct with `(Free*)((char*)p - offsetof(Free, next))`. The Linux kernel uses the same pattern extensively via its `container_of` macro.

⟨*function* `offsetof` 34a⟩≡ (368b)

```
#define offsetof(s, m)  (ulong)(&(((s*)nil)->m))
```

3.2.4 Functions

C functions like `print(char *fmt, ...)` accept a variable number of arguments. The problem is: how does the function body retrieve those arguments? For a fixed-signature function, the compiler generates code to read each parameter at a known offset in the stack frame. But for variadic functions, the compiler cannot generate this code because it does not know how many arguments were passed. The `va_list`/`va_start`/`va_arg`/`va_end` macros let the function walk the stack frame at runtime, extracting arguments one at a time.

The `va_start` macro works by taking the address of the last named parameter (`&(start)`) and pointing just past it—this is where the first variadic argument lives on the stack. So `va_list` is simply a `char*` pointer into the stack:

```
High addresses
+----------+
| arg N    |  <-- va_arg advances here
+----------+
| ...      |
+----------+
| arg 1    |  <-- va_start points here (after 'start')
+----------+
| start    |  <-- last named parameter
+----------+
Low addresses
```

`va_start` sets the pointer just past the last named parameter. Each call to `va_arg` reads the value at the current position and advances the pointer by the argument's size. Small types (1 or 2 bytes) are promoted to 4 bytes on the stack, which the macros account for.

⟨*type* `va_list` 34b⟩≡ (368a)

```
typedef char*   va_list;
```

⟨*macro* `va_start` 34c⟩≡ (368a)

```
#define va_start(list, start) list =\
    (sizeof(start) < 4?\
        (char*)((int*)&(start)+1):\
        (char*)(&(start)+1))
```

⟨*macro* `va_end` 34d⟩≡ (368a)

```
#define va_end(list)\
    USED(list)
```

⟨*macro* `va_arg` 34e⟩≡ (368a)

```
#define va_arg(list, mode)\
    ((sizeof(mode) == 1)?\
        ((list += 4), (mode*)list)[-4]:\
    (sizeof(mode) == 2)?\
        ((list += 4), (mode*)list)[-2]:\
        ((list += sizeof(mode)), (mode*)list)[-1])
```

Here is a complete example showing how these macros are used:

```
int sum(int n, ...) {
    va_list ap;
    int i, total = 0;
    va_start(ap, n);       // ap points past n
    for(i = 0; i < n; i++)
        total += va_arg(ap, int);  // read and advance
    va_end(ap);
    return total;
}
// call: sum(3, 10, 20, 30) returns 60
```

The call `sum(3, 10, 20, 30)` produces this stack:

```
+------+
| 30   |  <-- va_arg(ap, int) third time
+------+
| 20   |  <-- va_arg(ap, int) second time
+------+
| 10   |  <-- va_start(ap, n) points here
+------+
| 3    |  <-- n (last named parameter)
+------+
```

Each `va_arg(ap, int)` advances `ap` by 4 bytes (the size of `int`) and returns the value at the previous position. For a 1-byte `char`, the macro still advances by 4 (because of stack alignment) but reads from the correct offset within the 4-byte slot. In practice, passing an explicit count is unusual. Real variadic functions use other conventions to know when to stop: `print` uses the format string (each `%` verb implies one argument and its type), `execl` uses a `nil` sentinel to mark the end of the argument list, and `EARGF` in the argument parsing macros uses the structure of command-line flags.

3.3 Regular expressions

The regular expression types—`Reprog`[105b] (a compiled pattern) and `Resub`[106c] (matched subexpressions)—are declared in `regexp.h` but their internal structure is opaque at this point. They will be fully defined and explained in Chapter 10.

3.4 Files

File descriptors are small integers returned by `open` and `create`. The `fdt` typedef makes function signatures more readable than bare `int`.

⟨*type* `fdt` 35a⟩≡ (368b)

```
typedef int fdt; // file descriptor type
```

The original Plan 9 code uses bare integers (0, 1, 2) for the standard file descriptors. The author added these symbolic names to improve readability—writing `write(STDERR, ...)` makes the intent clearer than `write(2, ...)`. The same motivation applies to the `fdt` typedef above.

⟨*constant* `STDxxx` 35b⟩≡ (368b)

```
#define STDIN 0
#define STDOUT 1
#define STDERR 2
```

Other file-related types—`Qid`[191a] (unique file identifier), `DirX` (directory entry with metadata), `Open_flag`[190c], `Access_flag`[190d]—are defined in `syscall.h` and presented in Chapter 13 and Chapter 15.

3.5 Time

The `Tm` structure holds a broken-down time: year, month, day, hour, minute, second, plus the timezone abbreviation and offset from UTC. This is the same data that UNIX's `struct tm` holds, but with a better design: Plan 9 puts the timezone name (`zone`) and UTC offset (`tzoff`) directly inside the structure, whereas UNIX splits them into separate globals (`tzname`, `timezone`) that are not attached to any particular `struct tm`. The Plan 9 design means a `Tm` is self-contained—you can pass it to another function without losing the timezone context.

For example, March 25, 2026 at 14:30 in Rome would be represented with `hour`=14, `mon`=2 (0-indexed), `year`=126 (years since 1900), `zone`="CET", and `tzoff`=3600 (one hour east of UTC).

⟨*type* `Tm` 36⟩≡ (368b)
```
struct Tm {
    int sec;
    int min;
    int hour;

    int mday;
    int mon;
    int year;
    int wday;
    int yday;

    char    zone[4];
    int     tzoff;
};
```

3.6 Concurrency

The concurrency data structures—`Lock`[245a], `QLock`[246a], `RWLock`[248a], `Rendez`[251a]—are declared in `libc.h` but will be presented in Chapter 21. The threading structures (`Channel`[288e], `Alt`[300a], `Thread`[295a], `Proc`[292]) are declared in `thread.h` and covered in Chapter 23.

3.7 Collections

Notably absent from Plan 9's `libc` is any standard collection type—no hash tables, linked lists, or dynamic arrays. This is partly a consequence of C itself: without generics or templates, a reusable list library must either use `void*` (losing type safety and adding indirection) or rely on macros. In practice, C systems programmers prefer to embed the `next` pointer directly inside their structures, which gives tight control over memory layout and avoids the overhead of a generic container. The cost is that each program reimplements the same patterns. In this book, you will see embedded `next` pointers in `QLp` (lock wait queues), `Free` (allocator free list), `Proc` and `Thread` (thread scheduler queues). The same idiom appears throughout Principia Softwarica—in the Assembler book [Pad15a], Compiler book [Pad16a], and Kernel book [Pad14]—for linked lists, hash tables, and queues.

Hanson's *C Interfaces and Implementations* [Han96] provides exactly the kind of reusable collections (hash tables, sequences, rings, arenas) that Plan 9's `libc` lacks, and could in principle be ported to Plan 9.

Chapter 4

User/Kernel Bridge

With the core types defined, we now turn to how the library communicates with the kernel. System calls are the boundary between user space and the kernel. In this chapter I present the Plan 9 syscall table, the assembly stubs that invoke them, and how the C and kernel calling conventions differ. The kernel side of this bridge is covered in the KERNEL book [Pad14].

4.1 The syscall table

The entire Plan 9 kernel interface is defined in a single header, `libc/9syscall/sys.h`, shared between the kernel and the C library. Each syscall is a small integer:

```
#define NOP         0
/* process */
#define RFORK       1
#define EXEC        2
#define EXITS       3
#define AWAIT       4
/* memory */
#define BRK         5
/* files */
#define OPEN        6
#define CLOSE       7
#define PREAD       8
#define PWRITE      9
#define SEEK       10
/* directories */
#define CREATE     11
#define REMOVE     12
...
/* concurrency */
#define RENDEZVOUS 32
#define SEMACQUIRE 33
...
#define ERRSTR     39
```

Only 39 syscalls. Modern Linux has over 400. Plan 9 can get away with so few because many operations that require dedicated syscalls on UNIX are done through the file system instead: reading `/dev/user` instead of `getuid()`, reading `/env/PATH` instead of `getenv("PATH")`, writing to `/proc/pid/note` instead of `kill()`. The

grouping matches the structure of this book: each chapter in Parts III and IV corresponds to one group of syscalls.

4.2 The assembly stubs (ARM)

Each syscall has a tiny assembly stub in `9syscall/`. For example, here is what `open.s` looks like on ARM:

```
TEXT open(SB), 1, $0
MOVW R0, 0(FP)       /* save first arg */
MOVW $6, R0          /* OPEN = 6 */
SWI $0               /* trap into kernel */
RET
```

Every stub follows the same pattern (see Section 2.3): save `R0` to its reserved slot at `0(FP)`, load the syscall number into `R0`, execute `SWI $0`, and return. These stubs are not written by hand—they are *generated* by the `mkfile` in `9syscall/`, which reads `sys.h`, extracts each syscall name and number, and emits the corresponding assembly file. This is a nice example of code generation: 40 files produced from a 40-line header.

4.3 The calling conventions (ARM)

The C calling convention (described in Section 2.3) passes the first argument in `R0` and the rest on the stack. But the kernel expects a *different* convention: the syscall number in `R0`, and *all* arguments on the stack (including the first). The assembly stub bridges these two conventions:

```
C convention:              Stub transforms:        Kernel sees:
+-----------+              +-----------+            +-----------+
| arg2      | 4(FP)        | arg2      | 4(FP)      | arg2      |
+-----------+              +-----------+            +-----------+
| (empty)   | 0(FP)  -->   | arg1      | 0(FP)  --> | arg1      |
+-----------+              +-----------+            +-----------+
R0 = arg1                  R0 = syscall#            R0 = syscall#
```

The full path of a system call then looks like this:

```
User C code           Library               Kernel
-----------           -------               ------
fd = open("f", 0) --> open.s: SWI $0  --> sysopen()
                   <-- return fd        <-- return fd
```

The kernel's trap handler saves the user registers, reads the syscall number from `R0`, indexes into the `systab` dispatch table (see KERNEL book [Pad14]), and calls the corresponding `sysxxx()` function. On return, the result is placed in `R0` and the kernel restores the user registers. Errors are signaled by returning −1 and setting the per-process error string (readable via `errstr()X`). This is different from UNIX, where `errno` is a global variable set by the C library—in Plan 9, the kernel itself sets the error string, and the `%r` format verb in `print` reads it directly.

4.4 Comparison with UNIX

The Plan 9 syscall interface differs from UNIX in several ways:

- *Fewer syscalls*: 39 vs. 400+ on Linux, because file operations replace specialized syscalls.

- *No* `ioctl`: device control is done by reading and writing control files (e.g., `/net/tcp/0/ctl`).
- `pread`/`pwrite`: the primary I/O syscalls take an explicit offset, making concurrent access safe. `read`/`write` are library wrappers, not syscalls.
- `rfork` *instead of* `fork`+`clone`: a single syscall with flags for fine-grained resource sharing.
- *String errors*: `errstr` instead of `errno`; `exits(char*)` instead of `exit(int)`.

Chapter 5

_main() and _exits()

Every Plan 9 program actually starts in `_main()` (not `main()`), the C runtime entry point. On UNIX systems, this role is played by `crt0` ("C runtime, file 0"—the first object file linked into every program). Some UNIX systems also have `crt1` and `crti/crtn` for additional initialization (e.g., constructors, profiling setup), but Plan 9 keeps it simple with a single `_main` that sets up `argc/argv` and then calls `main()`. The name may be confusing: `_main` is the true entry point of the program (the linker sets it as the default entry point—see LINKER book [Pad15b]), while `main()` is the user's entry point, called by `_main` after initialization is complete.

This chapter also covers the argument parsing macros and the `exits()`[45b] function that handles cleanup callbacks before termination.

⟨*signatures process syscall wrapper* 40⟩≡ (379b) 230b ▷

```
extern  void    exits(char*);
extern  int     atexit(void(*)(void));
```

5.1 _main() (ARM)

The C runtime entry point must be written in assembly because it sets up things that C code takes for granted: the static base register (needed to access any global variable), the stack frame (needed for local variables and function calls), and the `argc/argv` arguments (which the kernel places in a raw format that must be reformatted for C). On UNIX, the equivalent code (`crt0`) also initializes the dynamic linker, thread-local storage, and C++ constructors—Plan 9 is simpler but the same principle applies.

When the kernel starts a new process, it sets `R0` to point to the `Tos` structure (shared between user and kernel, used for profiling and the pid) and places `argc` and `argv` on the stack. The layout before `_main` runs:

```
Top of stack segment
+-------------------+
| Tos               |  pid, clock, kcycles, pcycles, Prof
+-------------------+  <-- R0 points here
|   (gap)           |
+-------------------+
| argv strings      |  "arg0\0arg1\0..."
+-------------------+
| argv[N] = nil     |
| ...               |
| argv[0]           |  pointers to the strings above
+-------------------+
| argc              |  number of arguments
+-------------------+  <-- SP (stack pointer)
Low addresses
```

The Tos ("top of stack") structure sits at the very top of the user stack segment, *above* the stack itself. It is shared between user and kernel and contains fields like **pid** (process ID), **clock** (cycle counter for profiling), **kcycles/pcycles** (kernel/user cycle counts), and the **Prof** structure for profiling data. The kernel passes its address in R0. See KERNEL book [Pad14] for the full definition.

Notice that the kernel's layout has no slot for a return address—that is not a mistake. In ARM, the return address is held in R14 (the link register), never pushed to the stack. However, the C calling convention still reserves a specific layout for how a callee sees its arguments via FP. The kernel cannot simply match that layout because it is not a C caller: it just pushes **argc** and **argv** at the initial SP without any calling-convention framing. It is **_main**'s job to bridge this gap.

With the kernel's entry layout understood, we are ready to read **_main** itself. The assembly sets up the SB register, saves Tos, carves the private storage area from the frame, and copies **argc/argv** into the positions that the C calling convention requires before branching to main().

```
⟨libc/arm/main9.s 41a⟩≡
  ⟨constant NPRIVATES(arm) 41d⟩
  ⟨constant NOPROF(arm) 43a⟩
  arg=0
  sp=13
  sb=12

  TEXT    _main(SB), NOPROF, $(16 + NPRIVATES*4)
          MOVW    $setR12(SB), R(sb)
          MOVW    R(arg), _tos(SB)

          ⟨_main()(arm) setup private storage 42a⟩
          ⟨_main()(arm) adjust argc argv 42c⟩

          // user main()
          BL      main(SB)

  loop:
          ⟨_main()(arm) exit (if not done by main) 42d⟩

  ⟨_main()(arm) _exitstr defintion 42e⟩
```

The first instruction initializes R12 via the setR12 bootstrap described in Section 2.3. Until this executes, no global variable can be accessed. The second instruction saves the Tos pointer (passed by the kernel in R0, which is the same than R(arg) given the definition of arg) to the global _tos (declared in libc.h as extern Tos *_tos). This works because R12 was just initialized.

```
⟨libc/arm/argv0.s 41b⟩≡
  ⟨globals argv0.s 41c⟩
```

```
⟨globals argv0.s 41c⟩≡                                    (41b) 42b ▷
  GLOBL _tos(SB), $4
```

The private storage area is carved from the stack frame (the $(16 + NPRIVATES*4) in the TEXT directive reserves space). _privates and _nprivates are globals used by privalloc() to implement per-process private data (see Chapter 23). The 16 bytes (4 words) are the argument-passing slots that _main needs for the functions it calls: the Plan 9 ARM convention requires the caller to reserve stack space for arguments, so _main's frame has room for the argc/argv it will place at 4(R(sp))/8(R(sp)) when calling main() (and similarly for the call to exits()).

```
⟨constant NPRIVATES(arm) 41d⟩≡                                    (41a)
  #define NPRIVATES       16
```

⟨_main() *(arm) setup private storage* 42a⟩≡ (41a)

```
  MOVW    $p-64(SP), R1
  MOVW    R1, _privates(SB)
  MOVW    $NPRIVATES, R1
  MOVW    R1, _nprivates(SB)
```

⟨*globals* argv0.s 42b⟩+≡ (41b) ◁41c 43c▷

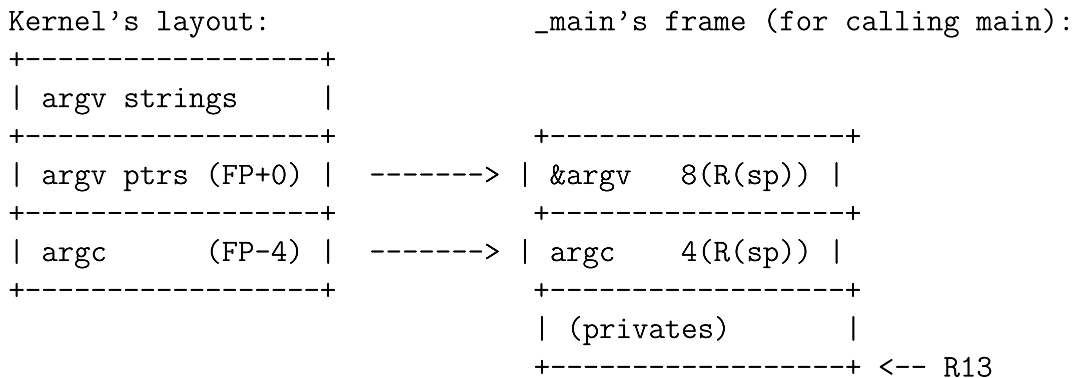
```
  GLOBL _privates(SB), $4
  GLOBL _nprivates(SB), $4
```

Next (using the register aliases and pseudo-registers described in Section 2.3), `_main` copies `argc` and `argv` from the kernel's stack layout (accessed via `FP`) into the positions expected by the C calling convention for calling `main(int argc, char **argv)`:

⟨_main() *(arm) adjust argc argv* 42c⟩≡ (41a)

```
  MOVW    $inargv+0(FP), R(arg)
  MOVW    R(arg), 8(R(sp))
  MOVW    inargc-4(FP), R(arg)
  MOVW    R(arg), 4(R(sp))
```

High addresses



Low addresses

The left side shows the kernel's original layout. In the initial diagram above, `argc` was at `SP`. But `_main`'s `TEXT` directive has since allocated a stack frame (for the private storage shown on the right), pushing `R13` down. The kernel's data is now above the frame and is reached via `FP`: `argc` at `FP-4` and `argv` at `FP0+` point to the same memory locations as before, just accessed with a different base. `_main` copies these values into `4(R(sp))` and `8(R(sp))`, which are where `main(int argc, char **argv)` expects its arguments.

If `main` returns (instead of calling `exits`), `_main` calls `exits("main")` in a loop. The string `"main"` signals that the program ended by falling off the end of `main` rather than exiting explicitly.

⟨_main() *(arm) exit (if not done by main)* 42d⟩≡ (41a)

```
  MOVW    $_exitstr<>(SB), R(arg)
  MOVW    R(arg), 4(R(sp))
  BL      exits(SB)
  // unreachable
  // force loading of div?
  BL      _div(SB)
  B       loop
```

The `DATA`/`GLOBL` directives define the exit string `"main"` as a global data symbol. The `BL _div` at the end is a linker trick: it forces the integer division routine to be linked in, since many programs need it but the linker might otherwise omit it.

⟨_main() *(arm)* _exitstr *defintion* 42e⟩≡ (41a)

```
  DATA    _exitstr<>+0(SB)/4, $"main"
  GLOBL   _exitstr<>+0(SB), $5
```

⟨*constant NOPROF(arm)* 43a⟩≡ (41a)

```
#define NOPROF 1 // see 5.out.h
```

When a program is linked with profiling enabled (`5l -p`), the linker uses `_main9p.s` instead, which calls `_profmain()` before `main()` to initialize the profiling infrastructure (see Chapter 26).

5.2 Command-line arguments: `ARGBEGIN()`, `ARGEND()`

`ARGBEGIN`/`ARGEND` are the standard Plan 9 idiom for parsing command-line flags. The macro expands to a `for` loop that scans each argument starting with `-`, decoding flag characters one at a time with `chartorune` (so flags can be Unicode). The special argument `--` terminates flag processing. `ARGF` and `EARGF` extract the next argument for flags that take a value.

⟨*signature global argv0* 43b⟩≡ (368b)

```
extern char *argv0;
```

⟨*globals* `argv0.s` 43c⟩+≡ (41b) ◁42b

```
GLOBL argv0(SB), $4
```

⟨*macro* `ARGBEGIN` 43d⟩≡ (368b)

```
#define ARGBEGIN    for((argv0||(argv0=*argv)),argv++,argc--;\
                argv[0] && argv[0][0]=='-' && argv[0][1];\
                argc--, argv++) {\
                char *_args, *_argt;\
                Rune _argc;\
                _args = &argv[0][1];\
                if(_args[0]=='-' && _args[1]==0){\
                    argc--; argv++; break;\
                }\
                _argc = 0;\
                while(*_args && (_args += chartorune(&_argc, _args)))\
                switch(_argc)
```

⟨*macro* `ARGEND` 43e⟩≡ (368b)

```
#define ARGEND      SET(_argt);USED(_argt,_argc,_args);}USED(argv, argc);
```

⟨*macro* `ARGF` 43f⟩≡ (368b)

```
#define ARGF()      (_argt=_args, _args="",\
                (*_argt? _argt: argv[1]? (argc--, *++argv): 0))
```

⟨*macro* `EARGF` 43g⟩≡ (368b)

```
#define EARGF(x)    (_argt=_args, _args="",\
                (*_argt? _argt: argv[1]? (argc--, *++argv): ((x), abort(), (char*)0)))
```

⟨*macro* `ARGC` 43h⟩≡ (368b)

```
#define ARGC()      _argc
```

A typical use looks like:

```
ARGBEGIN{
case 'd':  debug = 1;       break;
case 'o':  ofile = ARGF();  break;
default:   usage();
}ARGEND
```

For the command line `prog -d -o foo bar`, the macro iterates over `-d` and `-o`. On each iteration, `_args` points to the flag characters after the `-`, `_argc` holds the current flag rune (decoded via `chartorune`), and `_argt` is used by `ARGF` to extract the flag's argument (`foo` for `-o`). After `ARGEND`, `argv` points to the remaining arguments (`bar`) and `argc` is updated accordingly.

5.3 Syscall: _exits()

Each syscall has two definitions. The numeric code (EXITS=3, OPEN=6, etc.) lives in 9syscall/sys.h and is used only by the assembly stubs. The C-level signature (void _exits(char*)) is declared in syscall.h which is included from libc.h—so any .c file that includes libc.h gets type-checked against the correct prototype. The implementation itself is the assembly stub generated by 9syscall/mkfile.

⟨*signatures process syscall* 44a⟩≡ (379b) 230d ▷
```
extern void _exits(char*);
```

The raw system call is called _exits (with a leading underscore) rather than exits so that the library can wrap it with cleanup logic (running atexit()[45d] callbacks) in the exits()[45b] function below.

This is actually the only Plan 9 system call that uses this underscore convention—most other system calls (open, pread, rfork, etc.) are used directly without a wrapper. exits is special because it needs to run cleanup callbacks before terminating.

The assembly stub for _exits follows the pattern described in Chapter 4: it saves the argument (the exit string pointer, which arrived in R0) to the frame at 0(FP), loads the syscall number EXITS=3 into R0, and executes SWI $0. This is all the user-space code there is—the kernel's sysexits() does the actual process teardown.

Here is the actual ARM stub for _exits, following the same pattern as open.s in Chapter 4:

```
TEXT _exits(SB), 1, $0
MOVW R0, 0(FP)        /* save exit string pointer */
MOVW $3, R0            /* EXITS = 3 */
SWI $0                 /* trap into kernel */
RET                    /* never reached */
```

The RET is never reached because the kernel destroys the process. It is there because the assembler requires every TEXT block to end with a RET.

_exits(s) terminates the calling process immediately, passing the string s as its exit status (readable by the parent via await()). The empty string "" (or nil) conventionally means success; any other string signals failure. Internally, the kernel's sysexits() copies the string from user space and calls pexit(), which frees file descriptors, namespace, and segments, posts a wait message to the parent, and then—after switching to the CPU's main stack—releases the MMU state and returns the Proc slot to the free pool. See Kernel book [Pad14] for the full details.

In UNIX, exit() takes an integer status code (0 for success, non-zero for failure), and the parent decodes it via macros like WIFEXITED and WEXITSTATUS. The mapping from number to meaning is program-specific and often undocumented—you need to read the man page to know that exit code 2 means ENOENT ("no such file"). Plan 9's string-based exit is self-documenting: the parent sees "file not found" directly.

5.4 exits() and atexit()

exits()[45b] runs registered cleanup callbacks before calling the _exits() system call. The onex[44b] table stores up to 33 callback/pid pairs (see the Onex[388c] struct definition). The pid field exists because after rfork(RFMEM), parent and child share memory (including onex) but should only run their own callbacks—each process checks getpid() before invoking a handler.

⟨*global* onex 44b⟩≡ (388c)
```
Onex onex[NEXIT];
```
Uses NEXIT-54 44c.

⟨*constant* NEXIT 44c⟩≡ (388c)
```
#define NEXIT   33
```

⟨*struct* Onex 45a⟩≡ (388c)
```
struct Onex{
    void    (*f)(void); // callback
    int pid;
};
```

⟨*function* exits 45b⟩≡ (388c)
```
void
exits(char *s)
{
    int i, pid;
    void (*f)(void);

    pid = getpid();
    for(i = NEXIT-1; i >= 0; i--)
        if((f = onex[i].f) && pid == onex[i].pid) {
            onex[i].f = nil;
            (*f)();
        }
    _exits(s);
}
```
Uses NEXIT-54 44c, _exits(), getpid() 232a, and onex 44b.

The loop runs in reverse (from NEXIT-1 down to 0) so that callbacks are invoked in LIFO order—the last one registered runs first, mirroring how constructors and destructors are paired in C++. The name Onex is short for "on exit" (as in atexit—"at exit"), and onex is the array of registered callbacks. A subtle detail: onex[i].f = nil is set *before* calling the callback. This prevents infinite recursion if the callback itself calls exits. The pid == onex[i].pid check is important: after rfork(RFMEM), parent and child share the same onex array. Without the check, both processes would run all callbacks on exit—flushing the same buffers twice, closing the same file descriptors twice, or freeing the same memory twice. Most cleanup callbacks are not idempotent, so this would crash or corrupt data. The pid check ensures each process only runs the callbacks *it* registered. A child process that needs its own cleanup can call atexit after the fork.

⟨*global* onexlock 45c⟩≡ (388c)
```
static Lock onexlock;
```

The onexlock protects the onex array from concurrent access by shared-memory processes (see Chapter 21 for how Lock works).

atexit() registers a callback to be called when the program exits. It scans the onex array for an empty slot, records the current pid (so that after rfork(RFMEM), only the process that registered the callback will run it), and stores the function pointer. It returns 1 on success, 0 if the table is full.

⟨*function* atexit 45d⟩≡ (388c)
```
int
atexit(void (*f)(void))
{
    int i;

    lock(&onexlock);
    for(i=0; i<NEXIT; i++)
        if(onex[i].f == nil) {
            onex[i].pid = getpid();
            onex[i].f = f;
            unlock(&onexlock);
            return 1;
        }
    unlock(&onexlock);
    return 0;
}
```
Uses NEXIT-54 44c, getpid() 232a, lock() 245b, onex 44b, onexlock 45c, and unlock() 245c.

Chapter 6

Memory Area Operations

These functions operate on raw memory regions: filling, copying, comparing, and searching byte arrays. They are among the most frequently called functions in any C program—every `malloc`, every string copy, every structure initialization goes through them. Some functions (`memset`, `memmove`) have both a portable C implementation in `port/` and an optimized ARM assembly version in `arm/`; the others (`memcmp`, `memchr`) exist only in C. The build system automatically selects which version to link: the `mkfile` in `port/` uses a `reduce` script that filters out any C file for which a `.s` file exists in the architecture directory. So when building for ARM, `arm/memset.s` replaces `port/memset.c`, but `port/memcmp.c` is used as-is.

⟨*signatures of* `memxxx` *functions* 46a⟩≡ (368b)

```
extern  void*   memset(void*, int, ulong);
extern  void*   memcpy(void*, void*, ulong);
extern  void*   memmove(void*, void*, ulong);
extern  int     memcmp(void*, void*, ulong);
extern  void*   memchr(void*, int, ulong);
```

6.1 Filling: `memset()`

`memset` fills `n` bytes starting at `ap` with the value `c`. It is used to zero-initialize structures (`memset(&s, 0, sizeof s)`) clear buffers before use, and fill memory with a sentinel value for debugging (e.g., `0xFE` to detect uninitialized stack).

```
Before:  [  ?  |  ?  |  ?  |  ?  |  ?  ]
                                          memset(buf, 0, 5)
After:   [ 00  | 00  | 00  | 00  | 00  ]
```

⟨*function* `memset` 46b⟩≡ (405e)

```
void*
memset(void *ap, int c, ulong n)
{
    char *p;

    p = ap;
    while(n > 0) {
        *p++ = c;
        n--;
    }
    return ap;
}
```

Note that `memset` takes an `int`, not a `char`—it truncates to a byte when storing. This means `memset(buf, 255, n)` and `memset(buf, -1, n)` produce the same result: every byte set to `0xFF`. The parameter is `int` because passing 32-bit values is more efficient on most architectures than passing single bytes.

This byte-at-a-time loop is correct but slow. The ARM assembly version in `libc/arm/memset.s` fills 4 bytes at a time using word-sized stores, which is roughly 4× faster for large buffers.

6.2 memset() (ARM)

The ARM version of `memset` is dramatically faster than the C version for large buffers. The key optimizations are:

- The fill byte is replicated into all four bytes of a register (`ORR R4<<8` / `ORR R4<<16`), so a single 32-bit store writes 4 bytes at once.
- The destination is aligned to a 4-byte boundary first (the `_4align` loop handles the leading 1–3 bytes).
- The inner loop (`_f16loop`) uses `MOVM.IA.W` ("move multiple, increment after, write back") to store 16 bytes per iteration—four registers at once.
- The tail handles the remaining 1–3 bytes individually.

The register aliases make the code self-documenting: `TO` ("to", register 1) is the write pointer that advances through the destination; `TOE` ("to end", register 2) is the end address (`dest + n`), used as a loop terminator. `N` (register 3) holds the byte count, and `TMP` aliases the same register since `N` and `TMP` are never live at the same time.

⟨libc/arm/memset.s 47⟩≡

```
TO = 1
TOE = 2
N = 3
TMP = 3                                 /* N and TMP don't overlap */

TEXT memset(SB), $0
        MOVW    R0, R(TO)
        MOVW    data+4(FP), R(4)
        MOVW    n+8(FP), R(N)

        ADD     R(N), R(TO), R(TOE)     /* to end pointer */

        CMP     $4, R(N)                /* need at least 4 bytes to copy */
        BLT     _1tail

        AND     $0xFF, R(4)
        ORR     R(4)<<8, R(4)
        ORR     R(4)<<16, R(4)          /* replicate to word */

_4align:                                /* align on 4 */
        AND.S   $3, R(TO), R(TMP)
        BEQ     _4aligned

        MOVBU.P R(4), 1(R(TO))          /* implicit write back */
        B       _4align

_4aligned:
        SUB     $15, R(TOE), R(TMP)     /* do 16-byte chunks if possible */
        CMP     R(TMP), R(TO)
        BHS     _4tail
```

```
        MOVW    R4, R5                  /* replicate */
        MOVW    R4, R6
        MOVW    R4, R7

_f16loop:
        CMP     R(TMP), R(TO)
        BHS     _4tail

        MOVM.IA.W [R4-R7], (R(TO))
        B       _f16loop

_4tail:
        SUB     $3, R(TOE), R(TMP)      /* do remaining words if possible */
_4loop:
        CMP     R(TMP), R(TO)
        BHS     _1tail

        MOVW.P  R(4), 4(R(TO))          /* implicit write back */
        B       _4loop

_1tail:
        CMP     R(TO), R(TOE)
        BEQ     _return

        MOVBU.P R(4), 1(R(TO))          /* implicit write back */
        B       _1tail

_return:
        RET
```

6.3 Copying: `memcpy()` and `memmove()`

`memmove()` copies `n` bytes from `a2` to `a1`. It is used whenever raw data must be relocated: copying structures, shifting buffer contents, building packets. The key subtlety is handling overlapping regions:

```
Non-overlapping (forward copy is safe):
  src:  [A B C D E]
  dst:                [. . . . .]   -->  [A B C D E]

Overlapping (forward copy would clobber):
  [A B C D E . . .]
      ^src    ^dst
Forward: dst[0]=src[0]=A, dst[1]=src[1]=B, dst[2]=src[2]= oops,
         src[2] was already overwritten!
Solution: copy backwards from the end.
```

⟨*function* memmove 48⟩≡ (405d)

```
void*
memmove(void *a1, void *a2, ulong n)
{
    char *s1, *s2;

    if((long)n < 0)
        abort();
    s1 = a1;
```

```
    s2 = a2;
    if((s2 < s1) && (s2+n > s1))
        goto back;
    while(n > 0) {
        *s1++ = *s2++;
        n--;
    }
    return a1;

back:
    s1 += n;
    s2 += n;
    while(n > 0) {
        *--s1 = *--s2;
        n--;
    }
    return a1;
}
```

Uses `abort()` 356d.

The overlap test `(s2 < s1) && (s2n` ¿ s1)+ checks whether the source extends into the destination—if so, it copies backwards to avoid clobbering. In all other cases (non-overlapping, or source after destination), forward copy is safe.

`memcpy` is simply an alias for `memmove`—both handle overlapping regions. In standard C, `memcpy` has undefined behavior on overlapping memory, but Plan 9 makes the safe choice: there is only one implementation, and it always works correctly.

⟨*function* `memcpy` 49⟩≡ (405d)

```
void*
memcpy(void *a1, void *a2, ulong n)
{
    return memmove(a1, a2, n);
}
```

Uses `memmove()` 48.

6.4 `memmove()` (ARM)

The ARM `memmove` is the most complex assembly routine in the library. It handles four cases:

- *Forward, aligned*: both source and destination are word-aligned. The inner loop (`_f32loop`) copies 32 bytes per iteration using two `MOVM.IA.W` instructions (8 registers × 4 bytes).
- *Backward, aligned*: same, but using `MOVM.DB.W` ("decrement before") to copy from the end.
- *Forward, unaligned*: the source is not word-aligned. The code aligns the source, then uses shifts and ORs to assemble aligned words from two consecutive source words (`_fu8loop`).
- *Backward, unaligned*: same strategy in reverse (`_bu8loop`).

The direction (forward vs. backward) is chosen by comparing the source and destination pointers, exactly as in the C version. The unaligned cases are the most intricate: they read aligned words and use barrel-shifter operations (`<<R(LSHIFT)`, `>>R(RSHIFT)`) to extract the bytes that straddle word boundaries. Note that `memcmp`

and **memchr** have no ARM assembly versions—only the write-heavy functions (**memset**, **memmove**) were deemed worth optimizing.

```
⟨libc/arm/memmove.s 50⟩≡
  TS = 0
  TE = 1
  FROM = 2
  N = 3
  TMP = 3                                 /* N and TMP don't overlap */
  TMP1 = 4

  TEXT memcpy(SB), $0
          B       _memmove
  TEXT memmove(SB), $0
  _memmove:
          MOVW    R(TS), to+0(FP)         /* need to save for return value */
          MOVW    from+4(FP), R(FROM)
          MOVW    n+8(FP), R(N)

          ADD     R(N), R(TS), R(TE)      /* to end pointer */

          CMP     R(FROM), R(TS)
          BLS     _forward

  _back:
          ADD     R(N), R(FROM)           /* from end pointer */
          CMP     $4, R(N)                /* need at least 4 bytes to copy */
          BLT     _b1tail

  _b4align:                               /* align destination on 4 */
          AND.S   $3, R(TE), R(TMP)
          BEQ     _b4aligned

          MOVBU.W -1(R(FROM)), R(TMP)     /* pre-indexed */
          MOVBU.W R(TMP), -1(R(TE))       /* pre-indexed */
          B       _b4align

  _b4aligned:                             /* is source now aligned? */
          AND.S   $3, R(FROM), R(TMP)
          BNE     _bunaligned

          ADD     $31, R(TS), R(TMP)      /* do 32-byte chunks if possible */
  _b32loop:
          CMP     R(TMP), R(TE)
          BLS     _b4tail

          MOVM.DB.W (R(FROM)), [R4-R7]
          MOVM.DB.W [R4-R7], (R(TE))
          MOVM.DB.W (R(FROM)), [R4-R7]
          MOVM.DB.W [R4-R7], (R(TE))
          B       _b32loop

  _b4tail:                                /* do remaining words if possible */
          ADD     $3, R(TS), R(TMP)
  _b4loop:
          CMP     R(TMP), R(TE)
          BLS     _b1tail

          MOVW.W  -4(R(FROM)), R(TMP1)    /* pre-indexed */
          MOVW.W  R(TMP1), -4(R(TE))      /* pre-indexed */
```

```
	B	_b4loop

_b1tail:				/* remaining bytes */
	CMP	R(TE), R(TS)
	BEQ	_return

	MOVBU.W -1(R(FROM)), R(TMP)	/* pre-indexed */
	MOVBU.W R(TMP), -1(R(TE))	/* pre-indexed */
	B	_b1tail

_forward:
	CMP	$4, R(N)		/* need at least 4 bytes to copy */
	BLT	_f1tail

_f4align:				/* align destination on 4 */
	AND.S	$3, R(TS), R(TMP)
	BEQ	_f4aligned

	MOVBU.P 1(R(FROM)), R(TMP)	/* implicit write back */
	MOVBU.P R(TMP), 1(R(TS))	/* implicit write back */
	B	_f4align

_f4aligned:				/* is source now aligned? */
	AND.S	$3, R(FROM), R(TMP)
	BNE	_funaligned

	SUB	$31, R(TE), R(TMP)	/* do 32-byte chunks if possible */
_f32loop:
	CMP	R(TMP), R(TS)
	BHS	_f4tail

	MOVM.IA.W (R(FROM)), [R4-R7]
	MOVM.IA.W [R4-R7], (R(TS))
	MOVM.IA.W (R(FROM)), [R4-R7]
	MOVM.IA.W [R4-R7], (R(TS))
	B	_f32loop

_f4tail:
	SUB	$3, R(TE), R(TMP)	/* do remaining words if possible */
_f4loop:
	CMP	R(TMP), R(TS)
	BHS	_f1tail

	MOVW.P	4(R(FROM)), R(TMP1)	/* implicit write back */
	MOVW.P	R4, 4(R(TS))		/* implicit write back */
	B	_f4loop

_f1tail:
	CMP	R(TS), R(TE)
	BEQ	_return

	MOVBU.P 1(R(FROM)), R(TMP)	/* implicit write back */
	MOVBU.P R(TMP), 1(R(TS))	/* implicit write back */
	B	_f1tail

_return:
	MOVW	to+0(FP), R0
	RET

RSHIFT = 4
```

```
LSHIFT = 5
OFFSET = 11

BR0 = 6
BW0 = 7
BR1 = 7
BW1 = 8

_bunaligned:
        CMP     $2, R(TMP)              /* is R(TMP) < 2 ? */

        MOVW.LT $8, R(RSHIFT)           /* (R(n)<<24)|(R(n-1)>>8) */
        MOVW.LT $24, R(LSHIFT)
        MOVW.LT $1, R(OFFSET)

        MOVW.EQ $16, R(RSHIFT)          /* (R(n)<<16)|(R(n-1)>>16) */
        MOVW.EQ $16, R(LSHIFT)
        MOVW.EQ $2, R(OFFSET)

        MOVW.GT $24, R(RSHIFT)          /* (R(n)<<8)|(R(n-1)>>24) */
        MOVW.GT $8, R(LSHIFT)
        MOVW.GT $3, R(OFFSET)

        ADD     $8, R(TS), R(TMP)       /* do 8-byte chunks if possible */
        CMP     R(TMP), R(TE)
        BLS     _b1tail

        BIC     $3, R(FROM)             /* align source */
        MOVW    (R(FROM)), R(BR0)       /* prime first block register */

_bu8loop:
        CMP     R(TMP), R(TE)
        BLS     _bu1tail

        MOVW    R(BR0)<<R(LSHIFT), R(BW1)
        MOVM.DB.W (R(FROM)), [R(BR0)-R(BR1)]
        ORR     R(BR1)>>R(RSHIFT), R(BW1)

        MOVW    R(BR1)<<R(LSHIFT), R(BW0)
        ORR     R(BR0)>>R(RSHIFT), R(BW0)

        MOVM.DB.W [R(BW0)-R(BW1)], (R(TE))
        B       _bu8loop

_bu1tail:
        ADD     R(OFFSET), R(FROM)
        B       _b1tail

RSHIFT = 4
LSHIFT = 5
OFFSET = 11

FW0 = 6
FR0 = 7
FW1 = 7
FR1 = 8

_funaligned:
        CMP     $2, R(TMP)
```

```
	MOVW.LT $8, R(RSHIFT)		/* (R(n+1)<<24)|(R(n)>>8) */
	MOVW.LT $24, R(LSHIFT)
	MOVW.LT $3, R(OFFSET)

	MOVW.EQ $16, R(RSHIFT)		/* (R(n+1)<<16)|(R(n)>>16) */
	MOVW.EQ $16, R(LSHIFT)
	MOVW.EQ $2, R(OFFSET)

	MOVW.GT $24, R(RSHIFT)		/* (R(n+1)<<8)|(R(n)>>24) */
	MOVW.GT $8, R(LSHIFT)
	MOVW.GT $1, R(OFFSET)

	SUB	$8, R(TE), R(TMP)	/* do 8-byte chunks if possible */
	CMP	R(TMP), R(TS)
	BHS	_f1tail

	BIC	$3, R(FROM)		/* align source */
	MOVW.P	4(R(FROM)), R(FR1)	/* prime last block register, implicit write back */

_fu8loop:
	CMP	R(TMP), R(TS)
	BHS	_fu1tail

	MOVW	R(FR1)>>R(RSHIFT), R(FW0)
	MOVM.IA.W (R(FROM)), [R(FR0)-R(FR1)]
	ORR	R(FR0)<<R(LSHIFT), R(FW0)

	MOVW	R(FR0)>>R(RSHIFT), R(FW1)
	ORR	R(FR1)<<R(LSHIFT), R(FW1)

	MOVM.IA.W [R(FW0)-R(FW1)], (R(TS))
	B	_fu8loop

_fu1tail:
	SUB	R(OFFSET), R(FROM)
	B	_f1tail
```

6.5 Comparing: memcmp()

memcmp() compares two memory regions byte by byte, returning 0 if they are equal, 1 if the first differing byte in a1 is greater, or −1 if it is smaller. It is used to compare structures, binary data, and keys in hash tables. Unlike strcmp()[81a], it does not stop at null bytes—it always compares exactly n bytes.

⟨*function* memcmp 53⟩≡ (405c)

```
int
memcmp(void *a1, void *a2, ulong n)
{
    uchar *s1, *s2;
    uint c1, c2;

    s1 = a1;
    s2 = a2;
    while(n > 0) {
        c1 = *s1++;
        c2 = *s2++;
        if(c1 != c2) {
            if(c1 > c2)
                return 1;
```

```
            return -1;
        }
        n--;
    }
    return 0;
}
```

6.6 Searching: memchr()

memchr() scans n bytes starting at ap for the first occurrence of byte c, returning a pointer to it or nil if not found. It is used by strchr()[80c] (which searches for a character in a string) and by parsers that scan binary data for a delimiter. The c &= 0xFF ensures the comparison works correctly even if c was passed as a negative int.

⟨*function* memchr 54⟩≡ (405b)

```
void*
memchr(void *ap, int c, ulong n)
{
    uchar *sp;

    sp = ap;
    c &= 0xFF;
    while(n > 0) {
        if(*sp++ == c)
            return sp-1;
        n--;
    }
    return nil;
}
```

Chapter 7

Memory Management

This chapter covers the memory allocator: from the `brk()` system call that extends the heap, through the pool-based allocator that manages free blocks, to the `malloc`/`free` interface that programs use.

⟨*signatures memory management functions* 55⟩≡ (368b)

```
extern  void*   malloc(ulong);
extern  void    free(void*);
extern  void*   realloc(void*, ulong);

extern  void*   mallocz(ulong, bool);
extern  ulong   msize(void*);
```

Beyond the standard `malloc`/`free`/`realloc`, Plan 9 adds `mallocz` (allocate and optionally zero—the `bool` parameter controls zeroing) and `msize` (return the usable size of an allocated block, useful when the allocator rounds up).

7.1 Memory allocator principles

An allocator's job sounds simple: give the caller N contiguous bytes now, take them back on a later `free` call, and do both fast. The hard parts hide in three places. First, *fragmentation*: a long run of allocations and frees leaves the heap dotted with holes of wrong sizes. *External fragmentation* is having enough *total* free memory but no contiguous run big enough to satisfy a request; *internal fragmentation* is handing out a 40-byte block for a 33-byte request because 40 is the nearest size the allocator tracks. Every real allocator trades some of one for some of the other. Second, *speed*: `malloc` and `free` sit on the hottest path of every C program, so even a few nanoseconds per call show up in the profile. Third, *concurrency*: modern allocators are shared by many threads and must let several of them allocate simultaneously without tripping over each other.

Behind the huge literature on allocators, every design picks a point on four essentially orthogonal axes. *Where are the free blocks kept?* A sorted free list (simplest), a tree keyed by size (faster lookup), a bitmap, or an array of per-size "bucket" free lists (fastest for common sizes). *Which free block do you pick? First-fit* scans the list and takes the first block large enough (fast, tends to fragment near the head); *best-fit* takes the smallest block that still fits (tighter packing, slower search); *next-fit* remembers where the last allocation landed and starts from there; the *buddy* scheme rounds up to a power of two and splits. *When do you coalesce adjacent free blocks?* On every `free` (simple but slower), or lazily from a background scan. *Where does the metadata live? In-band headers* put a size-plus-free-flag word immediately before each block—which makes coalescing $O(1)$ and is the Knuth *boundary-tag* trick from 1973; *out-of-band metadata* keeps a separate bookkeeping table so user blocks are exactly their requested size, at the cost of an extra lookup on every call.

Most real allocators fall into one of three families. *Boundary-tag free-list* allocators—Knuth's original design—put a header before each block holding its size and a free-or-used bit, chain the free blocks together, coalesce neighbors on `free` by walking backward and forward through the headers, and pick a free block by first-fit, best-fit, or a tree lookup. K&R's 50-line allocator, Doug Lea's `dlmalloc`, glibc's `ptmalloc2` (`dlmalloc` plus

per-arena locks), and Plan 9's `Pool` all belong here. *Buddy allocators*, invented by Kenneth Knowlton in 1965, round every request up to a power of two and split the heap recursively in halves until the smallest-enough block is reached; freed blocks are merged with their "buddy" if that buddy is also free. This makes split and merge $O(\log N)$ but wastes up to half the memory to internal fragmentation, which is why it is used mainly for the Linux kernel's *page* allocator, not for bytes. *Slab* or *size-class* allocators, popularized by Jeff Bonwick's 1994 USENIX paper on SunOS, pre-compute a set of size classes (8, 16, 24, 32, ..., 4096 bytes) and maintain a separate free list per class. Allocation and free are $O(1)$ because no search is involved, and no coalescing is needed because all blocks on a list have the same size. This is the design of the Linux kernel's `slab/slub/slob` caches, FreeBSD's UMA zones, Google's `tcmalloc`, Facebook's `jemalloc`, Microsoft's `mimalloc`, and Go's runtime allocator. All three of them typically add a *per-thread cache* on top—a small private freelist per thread that absorbs the common case without taking any global lock—which is the main reason modern allocators scale on many-core machines.

Plan 9's allocator, called `Pool`, sits in the boundary-tag free-list family: each block has a header holding its size and a free flag, adjacent headers let `free` coalesce neighbors in $O(1)$, and the free blocks are indexed in a tree keyed by size so that `malloc` can find a best-fit block in $O(\log N)$. What makes `Pool` unusual is not its data structure but its code sharing: the same `Pool` implementation runs in user-space libc *and* inside the Plan 9 kernel, with different backing-store providers (`sbrk` in user space, `xalloc` in the kernel). It has no per-thread cache and no lock-free fast path—`Pool` protects itself with a single lock—so on highly-parallel workloads it is slower than `jemalloc` or `tcmalloc`. In exchange it is small enough to read end to end, which is exactly what the rest of this chapter does: starting with `brk` (the system call that asks the kernel for more heap), building `Pool` on top of that, and finally wiring `malloc/free/realloc/mallocz` onto the `Pool` interface.

7.1.1 Garbage collection versus manual memory

The entire memory-allocator chapter that follows assumes the programmer calls `malloc()` and `free()` explicitly—the *manual memory management* model that C, C++, and Plan 9 use. The alternative, *garbage collection* (GC), was invented by John McCarthy for Lisp in 1960. McCarthy's insight was radical for its era: the programmer's time is worth more than the machine's time, so the runtime should track which memory is still reachable and reclaim the rest automatically. When machines cost millions of dollars and programmer time was cheap, this seemed extravagant. When machines became cheap and programmer time became expensive, it seemed obvious.

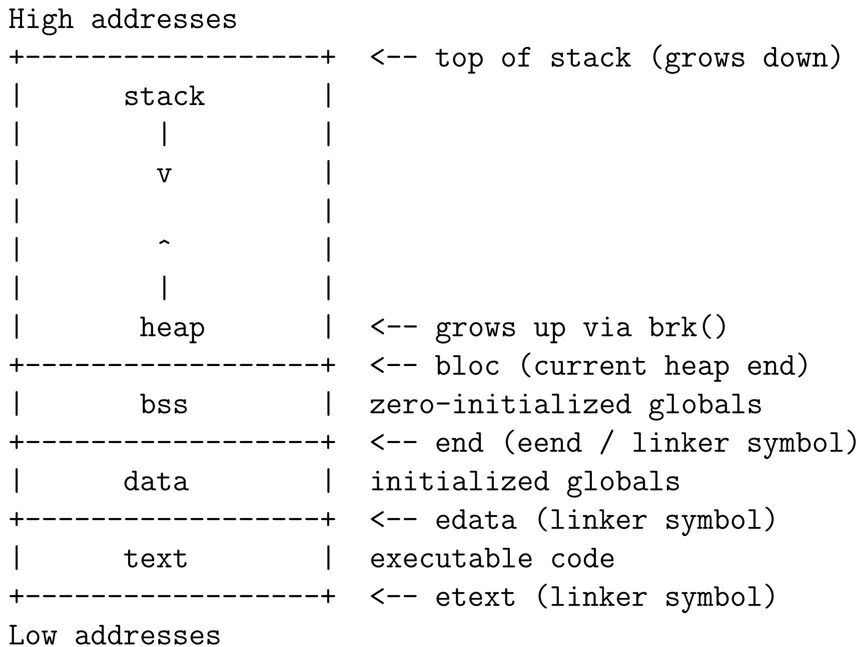
Java (1995) brought garbage collection to mainstream systems programming, and Go, Python, JavaScript, C#, Ruby, and Kotlin all followed. The trade-off is well understood: GC eliminates use-after-free bugs, double frees, and memory leaks at the cost of unpredictable *pause times* (the collector must occasionally stop the world to trace live objects) and higher memory overhead (the runtime needs headroom to defer collection). Then Rust (2015) found a *third path*: the compiler's *ownership and borrowing* rules track at compile time which function owns each allocation and when it can be freed, achieving memory safety *without* a runtime garbage collector and *without* manual `free()` calls. The price is a steep learning curve, but the result is the first language to offer both "no GC pauses" and "no use-after-free" simultaneously.

Plan 9's `Pool` allocator is firmly in the manual camp: `malloc()` and `free()`, with the programmer responsible for getting the pairing right. The code in this chapter is what that choice looks like at the implementation level. The advantage is predictability (no GC pauses, no collector thread, no memory overhead for the runtime) and simplicity (the allocator is a few hundred lines). The disadvantage is the bugs that manual management is famous for—use-after-free, double-free, leaks—which Plan 9 mitigates through careful coding discipline and `acid`'s post-mortem heap analysis rather than through language-level guarantees.

7.2 Syscall: `brk()`

⟨*signatures memory syscall* 56⟩≡ (379b)
```
extern  int brk(void*);
```

The `brk()` assembly stub is not reproduced here—it is generated by `9syscall/mkfile` using the same template as `_exits` above. Semantically, `brk(addr)` asks the kernel to set the top of the process's data segment to `addr`; the kernel maps or unmaps pages as needed.



The linker exports `etext`, `edata`, and `end` as symbols marking the boundaries between segments. `sbrk()` starts its heap at `end` (the first address after BSS) and advances it with each allocation.

In UNIX, both `brk` and `sbrk` are syscalls; Plan 9 has only `brk` as a syscall and implements `sbrk` in the library (shown below). On Linux, `glibc`'s `malloc` also uses `mmap` for large allocations: unlike `brk`, which can only grow or shrink the heap from the top, `mmap`/`munmap` can allocate and release memory anywhere in the address space, so large blocks can be returned to the kernel immediately when freed. Plan 9's simpler `brk`-only model works well for its typical workload of short-lived programs. For mapping shared memory, Plan 9 provides `segattach` instead of `mmap` (see Section 22.7).

7.3 sbrk()

⟨*signatures memory syscall wrapper* 57a⟩≡ (379b)
```
extern void* sbrk(ulong);
```

`sbrk()` extends the process's data segment by `n` bytes. The `bloc` pointer tracks the current end of the heap (initialized to `end`, the linker-defined symbol marking the end of BSS). The classic round-up idiom—add `Round` then mask low bits with bitwise AND—aligns the returned pointer to an 8-byte boundary—necessary because `malloc` must return memory usable for any type, including 64-bit values (`double`, `vlong`) which fault on ARM if misaligned.

⟨*global extern declaration* `end` 57b⟩≡ (493b)
```
extern  char    end[];
```

⟨*global* `bloc` 57c⟩≡ (493b)
```
// starting point for the heap, after the text, data, and bss.
static  char    *bloc = { end };
```

⟨*function* sbrk 58a⟩≡ (493b)
```
void*
sbrk(ulong n)
{
    uintptr bl;

    bl = ((uintptr)bloc + Round) & ~Round;
    if(brk((void*)(bl+n)) < 0)
        return (void*)-1;
    bloc = (char*)bl + n;
    return (void*)bl;
}
```
Uses `Round-258` 58b, `bloc-257` 57c, and `brk()`.

⟨*enum* _anon_ (9sys/sbrk.c) 58b⟩≡ (493b)
```
enum
{
    Round   = 7
};
```

7.4 Data structures

`sbrk()` can only grow the heap—it has no way to reclaim freed memory. Building a real allocator on top of it requires tracking which regions have been freed and reusing them. The Plan 9 allocator does this with a set of non-trivial data structures: a binary tree of free blocks (`Pool.freeroot`), a linked list of arenas, and per-block headers with boundary tags. We look at those structures next.

7.4.1 Pool

The `Pool` structure is the allocator's central object. It tracks the total and current memory usage, holds the free-block tree (`freeroot`) and the arena list (`arenalist`), and provides function pointers for customization: `alloc` obtains new memory from the OS, `lock`/`unlock` provide thread safety, and `panic` handles out-of-memory conditions. The flag bits enable debugging features like paranoid checking and allocation logging.

⟨*type* Pool 58c⟩≡ (381a)
```
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;
 };
```

⟨*type* Pool_flag 59a⟩≡ (381a)

```
 enum Pool_flag { /* flags */
  POOL_ANTAGONISM = 1<<0,
  POOL_PARANOIA = 1<<1,
  POOL_VERBOSITY = 1<<2,
  POOL_DEBUGGING = 1<<3,
  POOL_LOGGING = 1<<4,
  POOL_TOLERANCE = 1<<5,
  POOL_NOREUSE = 1<<6,
 };
```

⟨*struct* Private 59b⟩≡ (403d)

```
 struct Private {
     Lock    lk;
     int     pid;
     int     printfd;    /* gets debugging output if set */
 };
```

⟨*global* sbrkmempriv 59c⟩≡ (403d)

```
 Private sbrkmempriv;
```

7.4.2 sbrkmem and mainmem

⟨*global* sbrkmem 59d⟩≡ (403d)

```
 static Pool sbrkmem = {
     .name=      "sbrkmem",
     .maxsize=   (3840UL-1)*1024*1024,   /* up to ~0xf0000000 */
     .minarena=  4*1024,
     .quantum=   32,
     .alloc=     sbrkalloc,
     .merge=     sbrkmerge,
     .flags=     0,

     .lock=      plock,
     .unlock=        punlock,
     .print=     pprint,
     .panic=     ppanic,
     .private=       &sbrkmempriv,
 };
```

Uses `plock()` 401a, `ppanic()` 402c, `pprint()` 402a, `punlock()` 401b, `sbrkalloc()` 400b, `sbrkmempriv` 59c, and `sbrkmerge()` 400c.

⟨*global* mainmem 59e⟩≡ (403d)

```
 Pool *mainmem = &sbrkmem;
```

Uses `mainmem` 59e and `sbrkmem-196` 59d.

⟨*global* imagmem 59f⟩≡ (403d)

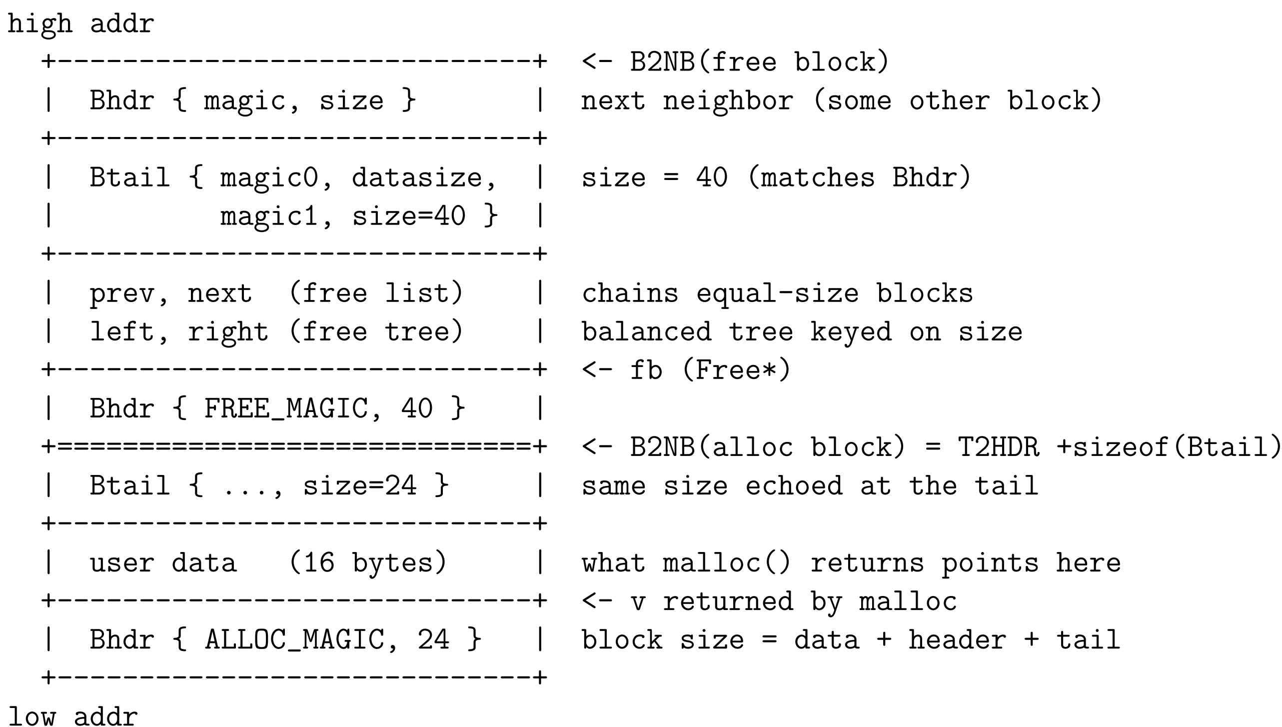
```
 Pool *imagmem = &sbrkmem;
```

Uses `imagmem` 59f and `sbrkmem-196` 59d.

7.4.3 Block headers and free list

Every allocated block is preceded by a `Bhdr` (block header) containing a magic number and the block's total size. Free blocks extend this with left/right/next/prev pointers to form a balanced binary tree of free blocks. The magic numbers (`ALLOC_MAGIC`, `FREE_MAGIC`, `NOT_MAGIC`, etc.) detect heap corruption—writing past the end of an allocation will overwrite the next block's magic, which the allocator checks on every operation. The `Btail` structure at the end of each block provides bidirectional traversal.

Here is what a contiguous slice of an arena looks like with one allocated block followed by a free block. The arena grows from low addresses at the bottom of the diagram to high addresses at the top, the same convention we use throughout this book:



The interesting design decision is `Btail`. A header-only scheme (`Bhdr` at the front of each block) is enough to walk the arena forward—`B2NB` just adds `size` to the current header—so why duplicate `size` at the tail? The reason is coalescing on `free()`. When a block is freed, the allocator wants to merge it with its neighbor on the *left* (lower address) if that neighbor is also free, to avoid fragmentation. Without a tail, finding the left neighbor would require a linear walk from the start of the arena, making `free()` O(n). With the tail, `B2PT` subtracts `sizeof(Btail)` from the current header to land on the previous block's `Btail`, and `T2HDR` then jumps back by `size` to reach its `Bhdr`. The whole operation is O(1). This is the concrete mechanism behind the Knuth boundary-tag trick introduced in Section 7.1—`Bhdr` plus `Btail` is what lets `Pool` find the left neighbor without walking the arena.

The `Free` struct uses the Plan 9 anonymous-field idiom (`Bhdr;` with no name), which lets a `Free*` be used wherever a `Bhdr*` is expected without an explicit cast or member access. That is why `fb->size` works even though `size` is declared in `Bhdr`, not in `Free`. The same idiom is used for `Alloc` and `Arena`, which are all `Bhdr`-prefixed. The `/* notused */` comment on `Alloc` hints at a subtle 8c constraint: if `Alloc` were a bare `typedef` for `Bhdr`, 8c would not distinguish the two types in its type checking, so the wrapper struct exists purely to give the compiler a distinct name to complain about when the wrong one is passed.

⟨*struct* `Bhdr` 60⟩≡ (426e)

```
struct Bhdr {
    ulong   magic;
    ulong   size;
};
```

⟨*enum* _anon_ (port/pool.c) 61a⟩≡ (426e)
```
enum {
    NOT_MAGIC = 0xdeadfa11,
    DEAD_MAGIC = 0xdeaddead,
};
```

⟨*macro* B2NB 61b⟩≡ (426e)
```
#define B2NB(b) ((Bhdr*)((uchar*)(b)+(b)->size))
```

⟨*macro* SHORT 61c⟩≡ (426e)
```
#define SHORT(x) (((x)[0] << 8) | (x)[1])
```

⟨*macro* PSHORT 61d⟩≡ (426e)
```
#define PSHORT(p, x) \
    (((uchar*)(p))[0] = ((x)>>8)&0xFF, \
    ((uchar*)(p))[1] = (x)&0xFF)
```

⟨*enum* _anon_ (port/pool.c)*2* 61e⟩≡ (426e)
```
enum {
    TAIL_MAGIC0 = 0xBE,
    TAIL_MAGIC1 = 0xEF
};
```

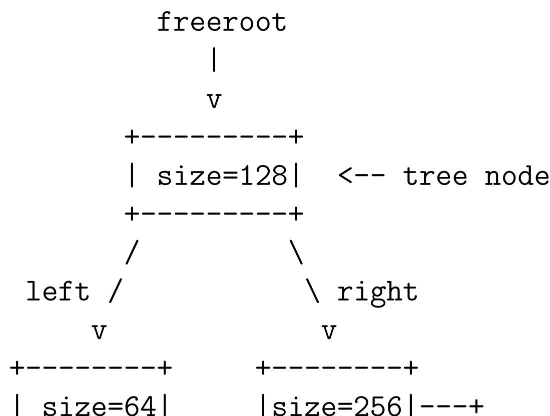
⟨*struct* Btail 61f⟩≡ (426e)
```
struct Btail {
    uchar   magic0;
    uchar   datasize[2];
    uchar   magic1;
    ulong   size;   /* same as Bhdr->size */
};
```

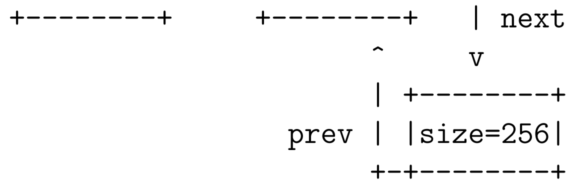
⟨*macro* B2T 61g⟩≡ (426e)
```
#define B2T(b)  ((Btail*)((uchar*)(b)+(b)->size-sizeof(Btail)))
```

⟨*macro* B2PT 61h⟩≡ (426e)
```
#define B2PT(b) ((Btail*)((uchar*)(b)-sizeof(Btail)))
```

⟨*macro* T2HDR 61i⟩≡ (426e)
```
#define T2HDR(t) ((Bhdr*)((uchar*)(t)+sizeof(Btail)-(t)->size))
```

The two pointer pairs are not redundant: `left`/`right` form a balanced binary search tree keyed on block *size*, while `next`/`prev` form a doubly-linked list of blocks that share the same size. Only one block per size sits in the tree—the "representative"—and the others hang off it via `next`/`prev`. A request for `n` bytes walks the tree once in $O(\log k)$ (where k is the number of distinct sizes), then pops the first element from the sibling list in $O(1)$. Here is a snapshot of `freeroot` with free blocks of sizes 64, 128, and 256 (two of them):






```
low addr
```

The sibling list is a micro-optimization: without it, a run of `malloc(256)` calls would keep rebalancing the tree. With it, the tree changes only when a size class becomes empty or appears for the first time.

⟨*struct* `Free` 62a⟩≡ (426e)
```
struct Free {
            Bhdr;
    Free*   left;
    Free*   right;
    Free*   next;
    Free*   prev;
};
```

⟨*enum* `_anon_ (port/pool.c)` *3* 62b⟩≡ (426e)
```
enum {
    FREE_MAGIC = 0xBA5EBA11,
};
```

⟨*struct* `Alloc` 62c⟩≡ (426e)
```
/*
 * the point of the notused fields is to make 8c differentiate
 * between Bhdr and Allocblk, and between Kempt and Unkempt.
 */
struct Alloc {
            Bhdr;
};
```

⟨*enum* `_anon_ (port/pool.c)` *4* 62d⟩≡ (426e)
```
enum {
    ALLOC_MAGIC = 0x0A110C09,
    UNALLOC_MAGIC = 0xCAB00D1E+1,
};
```

⟨*struct* `Arena` 62e⟩≡ (426e)
```
struct Arena {
            Bhdr;
    Arena*  aup;
    Arena*  down;
    ulong   asize;
    ulong   pad;    /* to a multiple of 8 bytes */
};
```

⟨*enum* `_anon_ (port/pool.c)` *5* 62f⟩≡ (426e)
```
enum {
    ARENA_MAGIC = 0xC0A1E5CE+1,
    ARENATAIL_MAGIC = 0xEC5E1A0C+1,
};
```

⟨*macro* `A2TB` 62g⟩≡ (426e)
```
#define A2TB(a) ((Bhdr*)((uchar*)(a)+(a)->asize-sizeof(Bhdr)))
```

⟨*macro* `A2B` 62h⟩≡ (426e)
```
#define A2B(a)  B2NB(a)
```

⟨*enum* _anon_ (port/pool.c) *6* 63a⟩≡ (426e)
```
enum {
    ALIGN_MAGIC = 0xA1F1D1C1,
};
```

⟨*enum* _anon_ (port/pool.c) *7* 63b⟩≡ (426e)
```
enum {
    MINBLOCKSIZE = sizeof(Free)+sizeof(Btail)
};
```

⟨*global* datamagic 63c⟩≡ (426e)
```
static uchar datamagic[] = { 0xFE, 0xF1, 0xF0, 0xFA };
```

⟨*constant* Poison 63d⟩≡ (426e)
```
#define Poison  (void*)0xCafeBabe
```

⟨*macro* _B2D 63e⟩≡ (426e)
```
#define _B2D(a) ((void*)((uchar*)a+sizeof(Bhdr)))
```

⟨*macro* _D2B 63f⟩≡ (426e)
```
#define _D2B(v) ((Alloc*)((uchar*)v-sizeof(Bhdr)))
```

7.5 malloc() and free()

malloc() is a thin wrapper around poolalloc()[64a]. When Npadlong is non-zero (debugging mode), extra space is allocated before the returned pointer to store the caller's PC—getcallerpc records who called malloc, which is invaluable for tracking down memory leaks.

⟨*function* malloc 63g⟩≡ (403d)
```
void*
malloc(ulong size)
{
    void *v;

    v = poolalloc(mainmem, size+Npadlong*sizeof(ulong));
    if(Npadlong && v != nil) {
        v = (ulong*)v+Npadlong;
        setmalloctag(v, getcallerpc(&size));
        setrealloctag(v, 0);
    }
    return v;
}
```
Uses Npadlong-198 63h, getcallerpc() 396f, mainmem 59e, setmalloctag() 69a, and setrealloctag() 69b.

⟨*constant* Npadlong 63h⟩≡ (402d)
```
Npadlong    = 2,
```

⟨*function* free 63i⟩≡ (403d)
```
void
free(void *v)
{
    if(v != nil)
        poolfree(mainmem, (ulong*)v-Npadlong);
}
```
Uses Npadlong-198 63h and mainmem 59e.

7.6 poolalloc()

With the block-header layout understood, the allocator's main loop is short. `poolallocl()` converts the user's requested `dsize` (data size) into a `bsize` (block size—data plus header plus tail, rounded up to the pool's `quantum`), searches the free tree for the smallest free block that is at least `bsize` bytes long (`treelookupgt`), and returns it. If no free block is large enough, it grows the arena with `poolnewarena` and tries again. The `trim` call handles the common case where the matching free block is strictly larger than needed: it splits the block, keeping only the requested size for the caller and putting the remainder back on the free tree. The returned pointer is converted from block address to data address with `B2D` (which adds `sizeof(Bhdr)` to step over the header).

The outer `poolalloc()` wrapper exists to enforce thread safety and debugging instrumentation around the hot inner loop. The `lock`/`unlock` pair serializes concurrent callers (remember that after `rfork(RFMEM)`, several processes share `mainmem`). The `paranoia` and `verbosity` macros are compile-time no-ops in a release build; under `POOL_PARANOIA` they walk every block in every arena to verify magic numbers and size fields, and under `POOL_VERBOSITY` they dump the allocator state after every operation. These are expensive but invaluable when debugging heap corruption—the allocator can report the exact call at which a block was damaged.

⟨*function* poolalloc 64a⟩≡ (426e)

```
void*
poolalloc(Pool *p, ulong n)
{
    void *v;

    p->lock(p);
    paranoia {
        poolcheckl(p);
    }
    verbosity {
        pooldumpl(p);
    }

    v = poolallocl(p, n);


    paranoia {
        poolcheckl(p);
    }
    verbosity {
        pooldumpl(p);
    }
    if(p->logstack && (p->flags & POOL_LOGGING)) p->logstack(p);
    LOG(p, "poolalloc %p %lud = %p\n", p, n, v);
    p->unlock(p);
    return v;
}
```

⟨*function* poolallocl 64b⟩≡ (426e)

```
/* poolallocl: attempt to allocate block to hold dsize user bytes; assumes lock held */
static void*
poolallocl(Pool *p, ulong dsize)
{
    ulong bsize;
    Free *fb;
    Alloc *ab;

    if(dsize >= 0x80000000UL){  /* for sanity, overflow */
        werrstr("invalid allocation size");
        return nil;
```

```
    }

    bsize = dsize2bsize(p, dsize);

    fb = treelookupgt(p->freeroot, bsize);
    if(fb == nil) {
        poolnewarena(p, bsize2asize(p, bsize));
        if((fb = treelookupgt(p->freeroot, bsize)) == nil) {
            /* assume poolnewarena failed and set %r */
            return nil;
        }
    }

    ab = trim(p, pooldel(p, fb), dsize);
    p->curalloc += ab->size;
    antagonism {
        memset(B2D(p, ab), 0xDF, dsize);
    }
    return B2D(p, ab);
}
```

7.7 poolfree()

`poolfreel()` is where the boundary-tag trick described in Section 7's "Block headers and free list" pays off. After converting the user pointer `v` back to an `Alloc*` with `D2B`, the function looks at its two neighbors:

- `back = T2HDR(B2PT(ab))`: walk back through the previous block's `Btail` to reach its `Bhdr`. If that `Bhdr`'s magic is `FREE_MAGIC`, merge the two blocks into one.
- `fwd = B2NB(ab)`: walk forward by adding the block's own `size` to get the next block's `Bhdr`. If that one is also free, merge again.

After up to two merges, the resulting (possibly larger) free block is inserted into the free tree via `pooladd`. This is the constant-time coalescing that distinguishes a real allocator from a toy one: without it, a program that allocates and frees in a pattern would gradually fragment the heap until no contiguous block was large enough to satisfy a request.

The `POOL_NOREUSE` branch is a debugging mode: instead of coalescing, it leaves the block marked `DEAD_MAGIC` and fills it with the poison byte `0xDA`. Subsequent use-after-free will see the poison and (because the magic is wrong) blow up on the next `blockcheck`. This is like `MALLOC_PROTECT` on Linux or the guard pages Valgrind inserts—with the difference that Plan 9's version is cheap enough to run in production if desired.

⟨*function* `poolfree` 65⟩≡ (426e)

```
void
poolfree(Pool *p, void *v)
{
    p->lock(p);
    paranoia {
        poolcheckl(p);
    }
    verbosity {
        pooldumpl(p);
    }

    poolfreel(p, v);

    paranoia {
```

```
        poolcheckl(p);
    }
    verbosity {
        pooldumpl(p);
    }
    if(p->logstack && (p->flags & POOL_LOGGING)) p->logstack(p);
    LOG(p, "poolfree %p %p\n", p, v);
    p->unlock(p);
}
```

⟨*function* poolfreel 66⟩≡ (426e)

```
/* poolfree: free block obtained from poolalloc; assumes lock held */
static void
poolfreel(Pool *p, void *v)
{
    Alloc *ab;
    Bhdr *back, *fwd;

    if(v == nil)    /* for ANSI */
        return;

    ab = D2B(p, v);
    blockcheck(p, ab);

    if(p->flags&POOL_NOREUSE){
        int n;

        ab->magic = DEAD_MAGIC;
        n = getdsize(ab)-8;
        if(n > 0)
            memset((uchar*)v+8, 0xDA, n);
        return;
    }

    p->nfree++;
    p->curalloc -= ab->size;
    back = T2HDR(B2PT(ab));
    if(back->magic == FREE_MAGIC)
        ab = blockmerge(p, back, ab);

    fwd = B2NB(ab);
    if(fwd->magic == FREE_MAGIC)
        ab = blockmerge(p, ab, fwd);

    pooladd(p, ab);
}
```

Uses B2NB-115 61b, B2PT-121 61h, D2B() 420c, DEAD_MAGIC-114 61a, FREE_MAGIC-123 62b, T2HDR-122 61i, blockmerge() 412c, getdsize() 413b, and memset() 46b.

7.8 Other functions

`mallocz` is "malloc zero"—it allocates and optionally zeros the memory. When the `clr` argument is true, the result is guaranteed to be all zeros, exactly like `calloc` in POSIX. The separate flag exists because many call sites know they are about to overwrite every byte of the allocation (e.g., filling in a struct field by field), in which case the extra `memset` is wasted work. Passing `false` gives them an uninitialized block and saves a memory-bandwidth-sized chunk of work on every allocation.

realloc has to unwind the Npadlong debugging shim before it can reach the underlying block. Recall that when debugging is enabled, malloc hands the caller a pointer that is Npadlong words *past* the real allocation (so there is room to record the caller's PC in front of the data). To pass the block to poolrealloc, realloc subtracts Npadlong from v to find the real start, adjusts size to include the padding, and then adds Npadlong back to the result. The setrealloctag call then records which caller triggered the reallocation—this is the hook that leak(8) and other Plan 9 memory debugging tools use to trace the origin of every live block.

⟨*function* mallocz 67a⟩≡ (403d)

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

    v = poolalloc(mainmem, size+Npadlong*sizeof(ulong));
    if(Npadlong && v != nil){
        v = (ulong*)v+Npadlong;
        setmalloctag(v, getcallerpc(&size));
        setrealloctag(v, 0);
    }
    if(clr && v != nil)
        memset(v, 0, size);
    return v;
}
```

Uses Npadlong-198 63h, getcallerpc() 396f, mainmem 59e, memset() 46b, setmalloctag() 69a, and setrealloctag() 69b.

⟨*function* realloc 67b⟩≡ (403d)

```
void*
realloc(void *v, ulong size)
{
    void *nv;

    if(size == 0){
        free(v);
        return nil;
    }

    if(v)
        v = (ulong*)v-Npadlong;
    size += Npadlong*sizeof(ulong);

    if(nv = poolrealloc(mainmem, v, size)){
        nv = (ulong*)nv+Npadlong;
        setrealloctag(nv, getcallerpc(&v));
        if(v == nil)
            setmalloctag(nv, getcallerpc(&v));
    }
    return nv;
}
```

Uses Npadlong-198 63h, free() 63i, getcallerpc() 396f, mainmem 59e, setmalloctag() 69a, and setrealloctag() 69b.

⟨*function* poolrealloc 67c⟩≡ (426e)

```
void*
poolrealloc(Pool *p, void *v, ulong n)
{
    void *nv;

    p->lock(p);
    paranoia {
        poolcheckl(p);
```

```
    }
    verbosity {
        pooldumpl(p);
    }
    nv = poolreallocl(p, v, n);
    paranoia {
        poolcheckl(p);
    }
    verbosity {
        pooldumpl(p);
    }
    if(p->logstack && (p->flags & POOL_LOGGING)) p->logstack(p);
    LOG(p, "poolrealloc %p %p %ld = %p\n", p, v, n, nv);
    p->unlock(p);
    return nv;
}
```

⟨*function* `msize` 68a⟩≡ (403d)

```
ulong
msize(void *v)
{
    return poolmsize(mainmem, (ulong*)v-Npadlong)-Npadlong*sizeof(ulong);
}
```

Uses `Npadlong-198` 63h and `mainmem` 59e.

⟨*function* `poolmsize` 68b⟩≡ (426e)

```
/*
 * Return the real size of a block, and let the user use it.
 */
ulong
poolmsize(Pool *p, void *v)
{
    Alloc *b;
    ulong dsize;

    p->lock(p);
    paranoia {
        poolcheckl(p);
    }
    verbosity {
        pooldumpl(p);
    }
    if(v == nil)    /* consistency with other braindead ANSI-ness */
        dsize = 0;
    else {
        b = D2B(p, v);
        dsize = (b->size&~(p->quantum-1)) - sizeof(Bhdr) - sizeof(Btail);
        assert(dsize >= getdsize(b));
        blocksetdsize(p, b, dsize);
    }
    paranoia {
        poolcheckl(p);
    }
    verbosity {
        pooldumpl(p);
    }
    if(p->logstack && (p->flags & POOL_LOGGING)) p->logstack(p);
    LOG(p, "poolmsize %p %p = %ld\n", p, v, dsize);
    p->unlock(p);
    return dsize;
```

```
}
```

7.9 Debugging support

The tag functions store the program counter of the caller that allocated or reallocated each block, directly in the `Npadlong` words reserved in front of the data. The layout, with `Npadlong=2`, `MallocOffset=0`, and `ReallocOffset=1`:

```
Block header area:
+-----------------+
|  Bhdr           |
+-----------------+
|  getcallerpc() | <- u[-2]  MallocOffset     (set by malloc)
+-----------------+
|  getcallerpc() | <- u[-1]  ReallocOffset    (set by realloc)
+-----------------+
|  user data      | <- v      what the caller sees
+-----------------+
```

When a heap debugging tool like `leak(8)` walks the live blocks later, it reads these PC values to identify exactly which line of C code allocated the block. The checks on `Npadlong <= MallocOffset` at the top of each function allow the whole tagging scheme to compile out cleanly when `Npadlong` is zero (release build): the tag functions become no-ops, and `malloc` returns the bare pool pointer with no slowdown.

⟨*function* `setmalloctag` 69a⟩≡ (403d)

```
void
setmalloctag(void *v, ulong pc)
{
    ulong *u;
    USED(v, pc);
    if(Npadlong <= MallocOffset || v == nil)
        return;
    u = v;
    u[-Npadlong+MallocOffset] = pc;
}
```

Uses `MallocOffset-199` 402d and `Npadlong-198` 63h.

⟨*function* `setrealloctag` 69b⟩≡ (403d)

```
void
setrealloctag(void *v, ulong pc)
{
    ulong *u;
    USED(v, pc);
    if(Npadlong <= ReallocOffset || v == nil)
        return;
    u = v;
    u[-Npadlong+ReallocOffset] = pc;
}
```

Uses `Npadlong-198` 63h and `ReallocOffset-200` 402d.

⟨*function* `getmalloctag` 69c⟩≡ (403d)

```
ulong
getmalloctag(void *v)
{
    USED(v);
```

```
    if(Npadlong <= MallocOffset)
        return ~0;
    return ((ulong*)v)[-Npadlong+MallocOffset];
}
```

Uses MallocOffset-199 402d and Npadlong-198 63h.

⟨*function* getrealloctag 70⟩≡ (403d)

```
ulong
getrealloctag(void *v)
{
    USED(v);
    if(Npadlong <= ReallocOffset)
        return ((ulong*)v)[-Npadlong+ReallocOffset];
    return ~0;
}
```

Uses Npadlong-198 63h and ReallocOffset-200 402d.

7.10 Advanced topics

Chapter 8

Formatted Output

Formatted output is a foundation: it is used everywhere in Plan 9, including inside the kernel itself. Plan 9's `print` family is more powerful than UNIX's `printf`: it supports user-defined format verbs via a callback mechanism (`fmtinstall()`), outputs to any destination (not just file descriptors or strings) through a `Fmt` closure, and handles Unicode natively. This chapter covers the `Fmt` data structure, the format string parser (`dofmt()`), the built-in format verbs, and the mechanism for registering custom ones.

Here is a preview of the public API. Note that `print` is a function pointer (not a plain function), so the kernel can redirect it to the console or serial port. `seprint(buf, ebuf, fmt, ...)` is the preferred safe variant: it takes both the start and end of the buffer, returning a pointer to the next position, so calls can be chained. The `#pragma varargck` lines are a compiler extension that enables type-checking of format arguments. `fmtinstall()` registers a custom format verb: pass a rune (e.g., `'H'`) and a callback, and `%H` becomes available in all print functions.

⟨*signatures* `xxxprint` *functions* 71a⟩≡ (368b)

```
extern  int     (*print)(char*, ...);
extern  int     fprint(fdt, char*, ...);

#pragma varargck    argpos  print       1
#pragma varargck    argpos  fprint      2

extern  int     sprint(char*, char*, ...);
extern  char*   seprint(char*, char*, char*, ...);
extern  int     snprint(char*, int, char*, ...);

#pragma varargck    argpos  sprint      2
#pragma varargck    argpos  seprint     3
#pragma varargck    argpos  snprint     3

extern  int fmtinstall(int, int (*)(Fmt*));
```

8.1 Data structures: `Fmt`

Plan 9's `Fmt` structure is more powerful than UNIX's `printf`: it supports user-defined format verbs via a callback mechanism, buffered output to any destination (not just file descriptors), and rune-aware formatting. The `flush` function pointer makes `Fmt` work like a closure: `fprint` flushes to a file descriptor, `sprint` flushes to a string, and `Bprint` flushes through a `Biobuf`. The `%r` verb (print the current error string) is a Plan 9 extension that replaces the separate `perror` pattern.

⟨*type* `Fmt` 71b⟩≡ (368b)

```
struct Fmt {
    uchar   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;
};
```

⟨*type* Fmt_flag 72a⟩≡ (368b)

```
enum Fmt_flag {
    FmtWidth    = 1,
    FmtLeft     = FmtWidth << 1,
    FmtPrec     = FmtLeft << 1,
    FmtSharp    = FmtPrec << 1,
    FmtSpace    = FmtSharp << 1,
    FmtSign     = FmtSpace << 1,
    FmtZero     = FmtSign << 1,
    FmtUnsigned = FmtZero << 1,
    FmtShort    = FmtUnsigned << 1,
    FmtLong     = FmtShort << 1,
    FmtVLong    = FmtLong << 1,
    FmtComma    = FmtVLong << 1,
    FmtByte     = FmtComma << 1,

    FmtFlag     = FmtByte << 1
};
```

8.2 print() and vfprint()

`print` is declared as a *function pointer*, not a function. This is unusual: almost every libc in existence defines `printf` as a plain extern function. The reason for the indirection is that Plan 9 reuses the same source code for the user-space library and for the kernel. In user space, `print` points to `libc_print`, which formats into a buffer and writes to file descriptor 1. In the kernel, the kernel's startup code reassigns `print` to its own implementation, which writes to the console or serial port instead. Both worlds can then share `dofmt`, `fmtinstall`, and all the format verbs without any conditional compilation. This is why `print` is initialized with an explicit `&libc_print`: the global symbol always exists, but its target can be overridden at link time.

⟨*global* print 72b⟩≡ (530c)

```
int (*print)(char *fmt, ...) = &libc_print;
```

Uses libc_print() 72c and print 72b.

⟨*function* libc_print 72c⟩≡ (530c)

```
int
libc_print(char *fmt, ...)
{
    int n;
    va_list args;

    va_start(args, fmt);
```

```
    n = vfprint(STDOUT, fmt, args);
    va_end(args);
    return n;
}
```

Uses vfprint() 73.

⟨*function* vfprint 73⟩≡ (536a)

```
int
vfprint(int fd, char *fmt, va_list args)
{
    Fmt f;
    char buf[256];
    int n;

    fmtfdinit(&f, fd, buf, sizeof(buf));
    f.args = args;
    n = dofmt(&f, fmt);
    if(n > 0 && _fmtFdFlush(&f) == 0)
        return -1;
    return n;
}
```

Uses _fmtFdFlush() 535e, dofmt() 74, and fmtfdinit() 522b.

8.3 dofmt()

`dofmt` is the core of the whole formatted-output system. It is a tiny state machine with exactly two states: "copying literal characters" and "interpreting a format verb." The outer `for(;;)` loop alternates between them:

```
+----------------------+
|  copy literal chars  |
|  into f->to buffer   |
+----------+-----------+
           |  hit '%'
           v
+----------------------+
|  _fmtdispatch()      |
|  parses width/prec,  |
|  calls verb handler  |
+----------+-----------+
           |  verb done
           +----------->  back to literal copy
```

The literal-copy branch is duplicated: one copy for byte output (`f->runes == 0`) and one for rune output (`f->runes != 0`), because `Biobuf`'s rune mode needs to write 32-bit runes into a `Rune*` buffer instead of bytes into a `char*` buffer. The duplication avoids a type-test inside the hot inner loop. When the destination buffer fills up (`t + n > s`), `_fmtflush` is called: for `fprint` it writes the buffer to the file descriptor; for `sprint`/`snprint` it reports that the buffer is full; for `Bprint` it forwards to the Biobuf's own flush. This is the closure-like use of `f->flush` mentioned in the `Fmt` section above.

Note how `dofmt` never calls `malloc` on the hot path. The `Fmt` struct lives on the caller's stack and writes into a caller-supplied buffer (`buf[256]` for `fprint`, the user's buffer for `sprint`). This matters because the memory allocator itself uses `print` for debugging output: if `dofmt` allocated, then a paranoid `poolcheck` that printed

the current arena state would re-enter the allocator and deadlock on the pool lock. The same reasoning is why the kernel can safely use `print` from interrupt context.

⟨*function* `dofmt` 74⟩≡ (510b)

```
/* format the output into f->to and return the number of characters fmted  */
int
dofmt(Fmt *f, char *fmt)
{
    Rune rune, *rt, *rs;
    int r;
    char *t, *s;
    int n, nfmt;

    nfmt = f->nfmt;
    for(;;){
        if(f->runes){
            rt = f->to;
            rs = f->stop;
            while((r = *(uchar*)fmt) && r != '%'){
                if(r < Runeself)
                    fmt++;
                else{
                    fmt += chartorune(&rune, fmt);
                    r = rune;
                }
                FMTRCHAR(f, rt, rs, r);
            }
            fmt++;
            f->nfmt += rt - (Rune *)f->to;
            f->to = rt;
            if(!r)
                return f->nfmt - nfmt;
            f->stop = rs;
        }else{
            t = f->to;
            s = f->stop;
            while((r = *(uchar*)fmt) && r != '%'){
                if(r < Runeself){
                    FMTCHAR(f, t, s, r);
                    fmt++;
                }else{
                    n = chartorune(&rune, fmt);
                    if(t + n > s){
                        t = _fmtflush(f, t, n);
                        if(t != nil)
                            s = f->stop;
                        else
                            return -1;
                    }
                    while(n--)
                        *t++ = *fmt++;
                }
            }
            fmt++;
            f->nfmt += t - (char *)f->to;
            f->to = t;
            if(!r)
                return f->nfmt - nfmt;
            f->stop = s;
        }
```

```
        fmt = _fmtdispatch(f, fmt, 0);
        if(fmt == nil)
            return -1;
    }
}
```

Uses FMTCHAR 500d, FMTRCHAR 500d, _fmtdispatch() 520, _fmtflush() 502a, and chartorune() 84d.

8.4 fprint()

⟨*function* fprint 75a⟩≡ (530b)

```
int
fprint(int fd, char *fmt, ...)
{
    int n;
    va_list args;

    va_start(args, fmt);
    n = vfprint(fd, fmt, args);
    va_end(args);
    return n;
}
```

Uses vfprint() 73.

8.5 sprint() and variants

⟨*function* sprint 75b⟩≡ (535d)

```
int
sprint(char *buf, char *fmt, ...)
{
    int n;
    va_list args;

    va_start(args, fmt);
    n = vsnprint(buf, 65536, fmt, args);    /* big number, but sprint is deprecated anyway */
    va_end(args);
    return n;
}
```

Uses vsnprint() 75c.

⟨*function* vsnprint 75c⟩≡ (538a)

```
int
vsnprint(char *buf, int len, char *fmt, va_list args)
{
    Fmt f;

    if(len <= 0)
        return -1;
    f.runes = 0;
    f.start = buf;
    f.to = buf;
    f.stop = buf + len - 1;
    f.flush = nil;
    f.farg = nil;
    f.nfmt = 0;
    f.args = args;
```

```
        dofmt(&f, fmt);
        *(char*)f.to = '\0';
        return (char*)f.to - buf;
    }
```

Uses `dofmt()` 74.

Part II

Data Processing

This part covers the libraries that transform data without making system calls: string manipulation at three levels (C strings, runes, UTF-8), regular expression matching, mathematical functions (trigonometry, logarithms, square roots), and data compression. These are pure computation—they depend only on the memory allocator from Part I.

Chapter 9

Strings

This chapter covers string manipulation at two levels: plain C strings (null-terminated byte arrays for ASCII) and rune strings (for Unicode). The plain C functions are the familiar `strlen()`, `strcmp()`, `strcpy()`, etc. The rune functions provide the same operations on arrays of `Rune`[30a] values, plus the UTF-8 encoding/decoding functions that convert between the two.

⟨*signatures of* `strxxx` *functions* 78a⟩≡ (368b)

```
extern  long    strlen(char*);
extern  int     strcmp(char*, char*);
extern  char*   strcpy(char*, char*);
extern  char*   strchr(char*, int);
extern  char*   strdup(char*);
extern  char*   strstr(char*, char*);
extern  char*   strcat(char*, char*);
```

⟨*signatures of* `runexxx` *functions* 78b⟩≡ (368b)

```
extern  long    runestrlen(Rune*);
extern  int     runestrcmp(Rune*, Rune*);
extern  Rune*   runestrcpy(Rune*, Rune*);
extern  Rune*   runestrchr(Rune*, Rune);
extern  Rune*   runestrdup(Rune*);
extern  Rune*   runestrstr(Rune*, Rune*);
extern  Rune*   runestrcat(Rune*, Rune*);
```

⟨*signatures rune conversion functions* 78c⟩≡ (368b)

```
extern  int chartorune(Rune*, char*);
extern  int runetochar(char*, Rune*);
```

⟨*signatures* `utfxxx` *functions* 78d⟩≡ (368b)

```
extern  int     utflen(char*);
extern  char*   utfrune(char*, long);
extern  char*   utfrrune(char*, long);
extern  char*   utfutf(char*, char*);
```

The conversion functions bridge the byte and rune worlds. `chartorune()`[84d] decodes one UTF-8 sequence into a `Rune`, returning the number of bytes consumed; `runetochar()`[85] encodes one `Rune` into UTF-8, returning the number of bytes produced. The `utfxxx` functions operate on UTF-8 strings directly: `utflen` returns the number of runes (not bytes), `utfrune`/`utfrrune` search forward/backward for a rune, and `utfutf` finds a UTF-8 substring.

9.1 Strings and UTF-8 principles

Before diving into the code, it helps to separate three concepts that get casually conflated in everyday conversation: the *abstract character*, the *code point*, and the *encoded byte sequence*. An abstract character is a

user-perceived unit: the Latin letter "A", the Greek letter β, the euro sign, a Han ideograph. A code point is the Unicode Consortium's numeric ID for an abstract character—a non-negative integer up to `0x10FFFF`, conventionally written `U+03B2` for the Greek beta. A code point is a pure number; it has no intrinsic width in memory. To actually *store* it you need an *encoding*: a rule that turns the number into some specific sequence of bytes. Different encodings give different byte sequences for the same code point, and picking one requires a set of trade-offs that the rest of this section is about.

The obvious encodings fall into two camps. *Fixed-width* encodings pick a single size—8 bits (ASCII, ISO-8859-1), 16 bits (UCS-2, Windows "Unicode", Java `char`), or 32 bits (UCS-4, the `wchar_t` on most Unix systems)—and store every code point at that size. Indexing and random access are trivial and `strlen` is $O(1)$ (divide bytes by width). The catches are that 8 bits is too small for anything beyond Western European text, 16 bits is too small for the full Unicode range (Unicode outgrew the Basic Multilingual Plane in 2001, so UCS-2 became technically invalid and Java, JavaScript, and the Windows API quietly redefined their "wide" char as a UTF-16 *code unit* with *surrogate pairs* for code points above `U+FFFF`), and 32 bits wastes four bytes on every ASCII character. *Variable-width* encodings use one to four bytes depending on the code point, trading constant-time indexing for compactness. Shift-JIS, EUC-JP, Big5, and UTF-8 are all variable-width; the first three are region-specific, the last is universal. An orthogonal issue—*byte order*—turns every 16-bit and 32-bit encoding into a cross-machine headache, which is why they need a byte-order mark (BOM) or explicit `UTF-16LE`/`UTF-16BE` variants.

UTF-8, the encoding this chapter is built around, was designed by Ken Thompson and Rob Pike on the back of a New Jersey diner placemat in September 1992 as a last-minute alternative to IBM's FSS-UTF proposal at an X/Open meeting; it was implemented in Plan 9's `libc` within days (`chartorune` and `runetochar` below are direct descendants) and is now the universal text encoding on every operating system except Microsoft Windows, which kept UTF-16 for historical compatibility. Four properties explain why it won. *ASCII-compatible*: every byte in the range `0x00`–`0x7F` is already a valid one-byte UTF-8 sequence, so every ASCII file is already a UTF-8 file and no embedded null bytes ever appear—C strings continue to work unchanged. *Self-synchronizing*: given any pointer into the middle of a UTF-8 byte stream, the start of the next rune is at most three bytes away (you walk forward past bytes whose top two bits are `10`), which is essential for `grep`/`sed`/regex and for error recovery after a truncated read. *Byte-order-independent and order-preserving*: the unit is a byte, so there is no big-endian/little-endian issue and no byte-order mark is needed, and sorting UTF-8 strings as byte strings gives the same answer as sorting by code point. *Variable width 1–4 bytes*: the common case (ASCII) costs exactly one byte, and the first byte of a multi-byte sequence encodes the total length via a prefix pattern so that decoders can read the length without scanning ahead. UTF-8 is today the W3C's mandatory encoding for HTML 5, the default for Python 3, Go, and Rust string types, and over 98% of public web pages are served as UTF-8.

Plan 9 was the birthplace of UTF-8 and uses a deliberately plain three-level vocabulary that maps cleanly onto the hierarchy above. A `char` is a single byte, just as in standard C—but because UTF-8 code points can be longer than one byte, iterating a Plan 9 string "character by character" in the colloquial sense does *not* mean iterating by `char`. For that you use a `Rune`, a 32-bit integer holding one Unicode code point. The conversion between the two is the job of `chartorune` (read the next UTF-8 sequence at a `char*` pointer, return the decoded `Rune` and the number of bytes consumed) and `runetochar` (encode one `Rune` into a `char*` buffer, return the number of bytes written). The rest of this chapter is organized along exactly that split: first the plain C-string functions (`strlen`, `strcmp`, `strcpy`, ...) that treat a string as a null-terminated byte array without regard for UTF-8; then `chartorune` and `runetochar` themselves; then the `runestr*` functions, which are the rune-array mirror of the C-string functions; and finally the `utf*` functions that operate on UTF-8 byte strings directly without decoding to a `Rune` array first.

9.2 Plain C strings

In C, a string is just a pointer to a sequence of bytes terminated by a null character (`\0`). There is no length field—`strlen` must scan to find the end. This simplicity has trade-offs: strings are cheap to create (just a pointer), but computing the length is $O(n)$, and the caller must ensure the destination buffer is large enough

for any copy or concatenation.

The Plan 9 implementations are notable for how they layer on each other. `strlen()`[80a] is implemented in terms of `strchr()`[80c]: it searches for the null terminator and returns the distance. `strcat()`[80b] uses `strchr()` to find the end of the destination string, then calls `strcpy()`[81c]. This layering keeps each function minimal and avoids duplicating loop logic.

⟨*function* `strlen` 80a⟩≡ (442e)
```
long
strlen(char *s)
{

    return strchr(s, '\0') - s;
}
```
Uses `strchr()` 80c.

⟨*function* `strcat` 80b⟩≡ (441b)
```
char*
strcat(char *s1, char *s2)
{

    strcpy(strchr(s1, '\0'), s2);
    return s1;
}
```
Uses `strchr()` 80c and `strcpy()` 81c.

`strchr()` is the fundamental building block: both `strlen()` and `strcat()` are built on it. It handles two cases separately: searching for `\0` (scan to the end) and searching for any other byte. `strrchr()`[80d] (reverse search) simply calls `strchr()` repeatedly, remembering the last match.

⟨*function* `strchr` 80c⟩≡ (441c)
```
char*
strchr(char *s, int c)
{
    char c0 = c;
    char c1;

    if(c == '\0') {
        while(*s++)
            ;
        return s-1;
    }

    while(c1 = *s++)
        if(c1 == c0)
            return s-1;
    return nil;
}
```

⟨*function* `strrchr` 80d⟩≡ (444e)
```
char*
strrchr(char *s, int c)
{
    char *r;

    if(c == '\0')
        return strchr(s, '\0');

    r = 0;
    while(s = strchr(s, c))
        r = s++;
```

```
    return r;
}
```
Uses strchr() 80c.

strcmp returns 1, -1, or 0 rather than an arbitrary positive/negative value. The comparison is done on unsigned characters to handle bytes above 0x7F correctly (important for UTF-8 strings, where high bytes are common).

⟨*function* strcmp 81a⟩≡ (441d)
```
int
strcmp(char *s1, char *s2)
{
    unsigned c1, c2;

    for(;;) {
        c1 = *s1++;
        c2 = *s2++;
        if(c1 != c2) {
            if(c1 > c2)
                return 1;
            return -1;
        }
        if(c1 == 0)
            return 0;
    }
}
```

⟨*constant* N 81b⟩≡ (441e)
```
#define N   10000
```

⟨*function* strcpy 81c⟩≡ (441e)
```
char*
strcpy(char *s1, char *s2)
{
    char *os1;

    os1 = s1;
    while(!memccpy(s1, s2, 0, N)) {
        s1 += N;
        s2 += N;
    }
    return os1;
}
```
Uses N-55 81b and memccpy() 81d.

strcpy is built on memccpy, which copies bytes until it finds the stop character (here, the null terminator) or reaches the limit N. If the string is longer than N bytes, memccpy returns nil and the loop advances both pointers to try the next chunk—effectively copying in 10 000-byte segments until the terminator is found.

⟨*function* memccpy 81d⟩≡ (405a)
```
void*
memccpy(void *a1, void *a2, int c, ulong n)
{
    uchar *s1, *s2;

    s1 = a1;
    s2 = a2;
    c &= 0xFF;
    while(n > 0) {
        if((*s1++ = *s2++) == c)
```

```
            return s1;
        n--;
    }
    return nil;
}
```

strdup allocates memory for a copy of the string. The **setmalloctag** call records the caller's PC so that memory debugging tools attribute the allocation to **strdup**'s caller, not to **strdup** itself—otherwise every leaked string would be blamed on **strdup** rather than the code that forgot to free it.

⟨*function* `strdup` 82a⟩≡ (442b)

```
char*
strdup(char *s)
{
    char *ns;

    ns = malloc(strlen(s) + 1);
    if(ns == nil)
        return nil;
    setmalloctag(ns, getcallerpc(&s));

    return strcpy(ns, s);
}
```

Uses `getcallerpc()` 396f, `malloc()` 63g, `setmalloctag()` 69a, `strcpy()` 81c, and `strlen()` 80a.

strstr()[82b] is the classic substring search: for each occurrence of the first character of **s2** in **s1**, it checks whether the rest of **s2** follows. This is the naive $O(nm)$ algorithm—good enough for the short strings typical in system programming, without the complexity of Boyer-Moore or KMP.

⟨*function* `strstr` 82b⟩≡ (445c)

```
/*
 * Return pointer to first occurrence of s2 in s1,
 * 0 if none
 */
char*
strstr(char *s1, char *s2)
{
    char *p, *pa, *pb;
    int c0, c;

    c0 = *s2;
    if(c0 == 0)
        return s1;
    s2++;
    for(p=strchr(s1, c0); p; p=strchr(p+1, c0)) {
        pa = p;
        for(pb=s2;; pb++) {
            c = *pb;
            if(c == 0)
                return p;
            if(c != *++pa)
                break;
        }
    }
    return 0;
}
```

Uses `strchr()` 80c.

9.3 Runes

While C strings are byte-oriented, rune strings operate on decoded Unicode code points—arrays of `Rune`[30a] (32-bit integers) terminated by a zero rune. Every plain C string function has a rune counterpart: `runestrlen()`[86a], `runestrcmp()`[87a], `runestrcpy()`[86b], etc. The implementations are nearly identical to the byte versions, just with `Rune` instead of `char`. Programs that need to manipulate characters by code point (editors, for instance) work with rune strings internally and convert to/from UTF-8 only at I/O boundaries.

The `_runebsearch()`[83c] function performs binary search on the Unicode character property tables (stored as sorted arrays of rune ranges), used by `isalpharune`, `istitlerune`, and similar classification functions.

⟨*function* `tobaserune` 83a⟩≡

```
Rune
tobaserune(Rune c)
{
    Rune *p;

    p = _runebsearch(c, _base2, nelem(_base2)/2, 2);
    if(p && c == p[0])
        c = p[1];
    return c;
}
```

⟨*function* `isbaserune` 83b⟩≡

```
int
isbaserune(Rune c)
{
    return tobaserune(c) == c;
}
```

⟨*function* `_runebsearch` 83c⟩≡ (437b)

```
Rune*
_runebsearch(Rune c, Rune *t, int n, int ne)
{
    Rune *p;
    int m;

    while(n > 1) {
        m = n/2;
        p = t + m*ne;
        if(c >= p[0]) {
            t = p;
            n = n-m;
        } else
            n = m;
    }
    if(n && c >= t[0])
        return t;
    return 0;
}
```

9.3.1 Conversion

`chartorune()`[84d] and `runetochar()`[85] are the core of Plan 9's UTF-8 support—they convert between UTF-8 byte sequences and runes. The encoding uses prefix bits to indicate the sequence length: a leading byte starting with 0 is ASCII (1 byte), 110 means 2 bytes, 1110 means 3 bytes, and 11110 means 4 bytes. Continuation bytes always start with 10. The decoder validates that sequences are not overlong (e.g., encoding ASCII as a 2-byte sequence) and rejects surrogate code points, returning `Runeerror` for invalid input.

The implementation uses helper macros to build and test the prefix bits. `Bit(i)` converts a bit position, `T(i)` builds a prefix mask (e.g., `T(2)` = `0xC0` for 2-byte sequences), and `RuneX(i)` computes the maximum code point that fits in an `i`-byte encoding. The decoder loop in `chartorune()` XORs each continuation byte with `Tx` (`0x80`) to strip the prefix, then shifts and accumulates the payload bits.

Here is the full UTF-8 layout for reference, with the free bits available for the rune payload marked `x`:

```
Range          Byte 1     Byte 2     Byte 3     Byte 4
-----          ------     ------     ------     ------
U+0000-007F    0xxxxxxx                                   (7  bits)
U+0080-07FF    110xxxxx   10xxxxxx                        (11 bits)
U+0800-FFFF    1110xxxx   10xxxxxx   10xxxxxx             (16 bits)
U+10000-10FFFF 11110xxx   10xxxxxx   10xxxxxx   10xxxxxx (21 bits)
```

A continuation byte is always `10xxxxxx`, so the test `c[i] ^= Tx; if(c[i] & Testx) goto bad;` checks that the top two bits were exactly `10`. The payload bits are then shifted into `l` left-to-right. For a worked example, take the Greek letter β (U+03B2 = 0x03B2 = binary `0000 0011 1011 0010`). This needs 11 bits, so it falls in the 2-byte range:

```
Rune bits:         000 1110 11 0010
Split for UTF-8:   000 1110         (top 5 bits)
                           11 0010  (bottom 6 bits)
Add prefixes:      110 01110         (T2 = 0xC0)
                           10 110010 (Tx = 0x80)
Final bytes:       0xCE     0xB2
```

Decoding reverses this. `chartorune` reads `0xCE`, sees that it is $\geq T(2)$ but $< T(3)$, so it knows the sequence is exactly 2 bytes. It strips the `110` prefix, shifts left by 6, ORs in the bottom 6 bits of `0xB2` (after stripping `10`), and returns U+03B2 with a length of 2.

Two checks at the end of the loop reject malformed input that would otherwise round-trip cleanly. First, "overlong" encodings: the test `l <= RuneX(i)` rejects sequences that use more bytes than the rune actually needs. For example, encoding U+002F (the ASCII slash) as the 2-byte sequence `0xC0 0xAF` is bit-legal but forbidden, because otherwise an attacker could smuggle a `/` past a path sanitizer that scanned for the literal byte. Second, surrogate code points (U+D800–U+DFFF) are reserved by the Unicode standard for UTF-16 and have no business appearing in UTF-8; the `SurrogateMin <= l && l <= SurrogateMax` check rejects them.

⟨*macro* `Bit` 84a⟩≡ (437a)

```
#define Bit(i) (7-(i))
```

⟨*macro* `T` 84b⟩≡ (437a)

```
/* N 0's preceded by i 1's, T(Bit(2)) is 1100 0000 */
#define T(i) (((1 << (Bit(i)+1))-1) ^ 0xFF)
```

⟨*macro* `RuneX` 84c⟩≡ (437a)

```
/* 0000 0000 0000 0111 1111 1111 */
#define RuneX(i) ((1 << (Bit(i) + ((i)-1)*Bitx))-1)
```

⟨*function* `chartorune` 84d⟩≡ (437a)

```
int
chartorune(Rune *rune, char *str)
{
    int c[UTFmax], i;
    Rune l;

    /*
     * N character sequence
```

```
	 *  00000-0007F => T1
	 *  00080-007FF => T2 Tx
	 *  00800-0FFFF => T3 Tx Tx
	 *  10000-10FFFF => T4 Tx Tx Tx
	 */

	c[0] = *(uchar*)(str);
	if(c[0] < Tx){
	    *rune = c[0];
	    return 1;
	}
	l = c[0];

	for(i = 1; i < UTFmax; i++) {
	    c[i] = *(uchar*)(str+i);
	    c[i] ^= Tx;
	    if(c[i] & Testx)
	        goto bad;
	    l = (l << Bitx) | c[i];
	    if(c[0] < T(i + 2)) {
	        l &= RuneX(i + 1);
	        if(i == 1) {
	            if(c[0] < T(2) || l <= Rune1)
	                goto bad;
	        } else if(l <= RuneX(i) || l > Runemax)
	            goto bad;
	        if (i == 2 && SurrogateMin <= l && l <= SurrogateMax)
	            goto bad;
	        *rune = l;
	        return i + 1;
	    }
	}

	/*
	 * bad decoding
	 */
  bad:
	*rune = Bad;
	return 1;
  }
```

Uses `Bad-112` 435d, `Bitx-105` 435d, `Rune1-107` 435d, `RuneX-104` 84c, `SurrogateMax-111` 435d, `SurrogateMin-110` 435d, `T-103` 84b, `Testx-109` 435d, and `Tx-106` 435d.

⟨*function* `runetochar` 85⟩≡ (437a)

```
  int
  runetochar(char *str, Rune *rune)
  {
	int i, j;
	Rune c;

	c = *rune;
	if(c <= Rune1) {
	    str[0] = c;
	    return 1;
	}

	/*
	 * one character sequence
	 *  00000-0007F => 00-7F
	 * two character sequence
```

```
     *  0080-07FF => T2 Tx
     * three character sequence
     *  0800-FFFF => T3 Tx Tx
     * four character sequence (21-bit value)
     *     10000-1FFFFF => T4 Tx Tx Tx
     * If the Rune is out of range or a surrogate half,
     * convert it to the error rune.
     * Do this test when i==3 because the error rune encodes to three bytes.
     * Doing it earlier would duplicate work, since an out of range
     * Rune wouldn't have fit in one or two bytes.
     */
    for(i = 2; i < UTFmax + 1; i++){
        if(i == 3){
            if(c > Runemax)
                c = Runeerror;
            if(SurrogateMin <= c && c <= SurrogateMax)
                c = Runeerror;
        }
        if (c <= RuneX(i) || i == UTFmax ) {
            str[0] = T(i) |  (c >> (i - 1)*Bitx);
            for(j = 1; j < i; j++)
                str[j] = Tx | ((c >> (i - j - 1)*Bitx) & Maskx);
            return i;
        }
    }
    return UTFmax;
}
```

Uses `Bitx-105` 435d, `Maskx-108` 435d, `Rune1-107` 435d, `RuneX-104` 84c, `SurrogateMax-111` 435d, `SurrogateMin-110` 435d, `T-103` 84b, and `Tx-106` 435d.

9.3.2 Basics

These are the rune-string equivalents of the plain C string functions. They follow the same layering pattern: `runestrlen()`[86a] calls `runestrchr()`[87b], `runestrcat()`[88a] calls `runestrcpy()`[86b], and so on. Since runes are fixed-width (unlike UTF-8 bytes), these functions are simpler—`runestrcpy()` is just a straightforward copy loop with no need for `memccpy()`[81d] tricks.

⟨*function* `runestrlen` 86a⟩≡ (438d)

```
long
runestrlen(Rune *s)
{

    return runestrchr(s, 0) - s;
}
```

Uses `runestrchr()` 87b.

⟨*function* `runestrcpy` 86b⟩≡ (437f)

```
Rune*
runestrcpy(Rune *s1, Rune *s2)
{
    Rune *os1;

    os1 = s1;
    while(*s1++ = *s2++)
        ;
    return os1;
}
```

⟨*function* runestrcmp 87a⟩≡ (437e)

```
int
runestrcmp(Rune *s1, Rune *s2)
{
    Rune c1, c2;

    for(;;) {
        c1 = *s1++;
        c2 = *s2++;
        if(c1 != c2) {
            if(c1 > c2)
                return 1;
            return -1;
        }
        if(c1 == 0)
            return 0;
    }
}
```

⟨*function* runestrchr 87b⟩≡ (437d)

```
Rune*
runestrchr(Rune *s, Rune c)
{
    Rune c0 = c;
    Rune c1;

    if(c == 0) {
        while(*s++)
            ;
        return s-1;
    }

    while(c1 = *s++)
        if(c1 == c0)
            return s-1;
    return nil;
}
```

⟨*function* runestrrchr 87c⟩≡ (439e)

```
Rune*
runestrrchr(Rune *s, Rune c)
{
    Rune *r;

    if(c == 0)
        return runestrchr(s, 0);
    r = 0;
    while(s = runestrchr(s, c))
        r = s++;
    return r;
}
```

Uses `runestrchr()` 87b.

⟨*function* runestrdup 87d⟩≡ (438a)

```
Rune*
runestrdup(Rune *s)
{
    Rune *ns;

    ns = malloc(sizeof(Rune)*(runestrlen(s) + 1));
```

```
    if(ns == nil)
        return nil;
    setmalloctag(ns, getcallerpc(&s));

    return runestrcpy(ns, s);
}
```

Uses `getcallerpc()` 396f, `malloc()` 63g, `runestrcpy()` 86b, `runestrlen()` 86a, and `setmalloctag()` 69a.

⟨*function* `runestrcat` 88a⟩≡ (437c)

```
Rune*
runestrcat(Rune *s1, Rune *s2)
{

    runestrcpy(runestrchr(s1, 0), s2);
    return s1;
}
```

Uses `runestrchr()` 87b and `runestrcpy()` 86b.

⟨*function* `runestrstr` 88b⟩≡ (440a)

```
/*
 * Return pointer to first occurrence of s2 in s1,
 * 0 if none
 */
Rune*
runestrstr(Rune *s1, Rune *s2)
{
    Rune *p, *pa, *pb;
    int c0, c;

    c0 = *s2;
    if(c0 == 0)
        return s1;
    s2++;
    for(p=runestrchr(s1, c0); p; p=runestrchr(p+1, c0)) {
        pa = p;
        for(pb=s2;; pb++) {
            c = *pb;
            if(c == 0)
                return p;
            if(c != *++pa)
                break;
        }
    }
    return nil;
}
```

Uses `runestrchr()` 87b.

9.4 UTF8

The UTF-8 functions bridge the gap between byte strings and rune strings: they search and measure byte strings while respecting multi-byte character boundaries. `utflen()`[89a] counts the number of runes (not bytes) in a UTF-8 string. `utfrune()`[89b] and `utfrrune()`[89c] find a rune in a UTF-8 string (forward and backward). `utfutf()`[90] finds a UTF-8 substring within another.

The key optimization throughout is the `Runeself` test: any byte below 0x80 is ASCII and can be handled without calling `chartorune()`[84d]. Since most text in practice is ASCII, this fast path avoids the overhead of the

full decoder for the common case. For ASCII-only runes, `utfrune()` even delegates directly to `strchr()`[80c].

⟨*function* `utflen` 89a⟩≡ (461b)

```
int
utflen(char *s)
{
    int c;
    long n;
    Rune rune;

    n = 0;
    for(;;) {
        c = *(uchar*)s;
        if(c < Runeself) {
            if(c == '\0')
                return n;
            s++;
        } else
            s += chartorune(&rune, s);
        n++;
    }
}
```

Uses `chartorune()` 84d.

⟨*function* `utfrune` 89b⟩≡ (462c)

```
char*
utfrune(char *s, long c)
{
    long c1;
    Rune r;
    int n;

    if(c < Runesync)        /* not part of utf sequence */
        return strchr(s, c);

    for(;;) {
        c1 = *(uchar*)s;
        if(c1 < Runeself) { /* one byte rune */
            if(c1 == 0)
                return nil;
            if(c1 == c)
                return s;
            s++;
            continue;
        }
        n = chartorune(&r, s);
        if(r == c)
            return s;
        s += n;
    }
}
```

Uses `chartorune()` 84d and `strchr()` 80c.

⟨*function* `utfrrune` 89c⟩≡ (462b)

```
char*
utfrrune(char *s, long c)
{
    long c1;
    Rune r;
    char *s1;
```

```
    if(c < Runesync)        /* not part of utf sequence */
        return strrchr(s, c);

    s1 = 0;
    for(;;) {
        c1 = *(uchar*)s;
        if(c1 < Runeself) { /* one byte rune */
            if(c1 == 0)
                return s1;
            if(c1 == c)
                s1 = s;
            s++;
            continue;
        }
        c1 = chartorune(&r, s);
        if(r == c)
            s1 = s;
        s += c1;
    }
}
```

Uses `chartorune()` 84d and `strrchr()` 80d.

⟨*function* `utfutf` 90⟩≡ (462d)

```
/*
 * Return pointer to first occurrence of s2 in s1,
 * 0 if none
 */
char*
utfutf(char *s1, char *s2)
{
    char *p;
    long f, n1, n2;
    Rune r;

    n1 = chartorune(&r, s2);
    f = r;
    if(f <= Runesync)       /* represents self */
        return strstr(s1, s2);

    n2 = strlen(s2);
    for(p=s1; p=utfrune(p, f); p+=n1)
        if(strncmp(p, s2, n2) == 0)
            return p;
    return 0;
}
```

Uses `chartorune()` 84d, `strlen()` 80a, `strncmp()` 443c, `strstr()` 82b, and `utfrune()` 89b.

9.5 Quoted strings

Plan 9 uses single-quote quoting for strings that contain whitespace or special characters (similar to shell quoting). The `quotestrconv`/`unquotestrdup` functions handle conversion to and from this format, used in file names and the `rc` shell.

9.6 Extensible strings

`libstring` (declared in `str.h`) provides growable, reference-counted strings—something C lacks natively. A `String` wraps a `malloc`'d character buffer with a write pointer that advances as data is appended; when the buffer fills, `s_grow()`[94b] reallocates with amortized doubling. Reference counting allows multiple owners to share a string cheaply via `s_incref()`[100b]; mutation triggers copy-on-write through `s_unique()`[100c]. The library also integrates with `libbio` for line-oriented I/O: `s_read_line()`[96c], `s_getline()`[97b], and `s_read()`[97a] append data from a `Biobuf`[195a] directly into a `String`.

A typical use looks like this:

```
String *s;

s = s_new();
s_append(s, "hello ");
s_append(s, "world");
print("%s\n", s_to_c(s));   /* "hello world" */
s_free(s);
```

9.6.1 Data structure: String

A `String` has three pointers into its buffer: `base` (start of allocated memory), `end` (one past the last allocated byte), and `ptr` (the current write position). The content of the string is the bytes between `base` and `ptr`; the space between `ptr` and `end` is available for appending. The `ref` field tracks how many owners share this string, and `fixed` marks strings that wrap a caller-owned buffer (via `s_array()`[100d]) and must not be reallocated. The embedded `Lock` protects the reference count for concurrent access.

⟨*struct* `String` 91⟩≡ (383b)

```
/* extensible Strings */
struct String {
    Lock;

    char *base; /* base of String */
    char *end; /* end of allocated space+1 */

    char *ptr; /* ptr into String */

    short ref;
    uchar fixed;
};
```

After calling `s_copy("hello")`, the string looks like this:

```
String
+--------+
| ref=1  |
| fixed=0|
+--------+
| base   |---> [h e l l o \0 _ _ _ ... _ _ _]
| ptr    |---> ...............^                 (points to \0)
| end    |---> ................................^  (128 bytes total)
+--------+

s_len(s) = ptr - base = 5
s_to_c(s) = base = "hello"
```

The macros `s_to_c()`[92a], `s_len()`[92b], and `s_clone()`[92c] provide quick access: `s_to_c(s)` returns `s->base` (the underlying C string), `s_len(s)` returns `s->ptr - s->base` (the length), and `s_clone(s)` makes an independent copy via `s_copy()`[93a].

⟨*macro* `s_to_c` 92a⟩≡ (383b)

```
#define s_to_c(s) ((s)->base)
```

⟨*macro* `s_len` 92b⟩≡ (383b)

```
#define s_len(s) ((s)->ptr-(s)->base)
```

⟨*macro* `s_clone` 92c⟩≡ (383b)

```
#define s_clone(s) s_copy((s)->base)
```

9.6.2 Allocation: `s_new()`, `s_copy()`, `s_free()`

`s_new()`[92d] creates an empty string with a default 128-byte buffer (`STRLEN`). `s_copy()`[93a] creates a string initialized from a C string, allocating at least enough to hold it. Both delegate to `_s_alloc()`[92f] (which allocates the `String` header with `ref=1`) and `s_newalloc()`[93b] (which allocates the character buffer).

⟨*function* `s_new` 92d⟩≡ (584d)

```
/* create a new ‘short’ String */
extern String *
s_new(void)
{
    String *sp;

    sp = _s_alloc();
    if(sp == nil)
        sysfatal("s_new: %r");
    sp->base = sp->ptr = malloc(STRLEN);
    if (sp->base == nil)
        sysfatal("s_new: %r");
    sp->end = sp->base + STRLEN;
    s_terminate(sp);
    return sp;
}
```

Uses `STRLEN-334` 92e, `_s_alloc()` 92f, `malloc()` 63g, `s_terminate()` 95a, and `sysfatal()` 236a.

⟨*constant* `STRLEN` 92e⟩≡ (584d)

```
#define STRLEN 128
```

⟨*function* `_s_alloc` 92f⟩≡ (584d)

```
/* allocate a String head */
extern String *
_s_alloc(void)
{
    String *s;

    s = mallocz(sizeof *s, 1);
    if(s == nil)
        return s;
    s->ref = 1;
    s->fixed = 0;
    return s;
}
```

Uses `mallocz()` 67a.

⟨*function* s_copy 93a⟩≡ (584g)
```
/* return a String containing a copy of the passed char array */
extern String*
s_copy(char *cp)
{
    String *sp;
    int len;

    len = strlen(cp)+1;
    sp = s_newalloc(len);
    setmalloctag(sp, getcallerpc(&cp));
    strcpy(sp->base, cp);
    sp->ptr = sp->base + len - 1;  /* point to 0 terminator */
    return sp;
}
```
Uses getcallerpc() 396f, s_newalloc() 93b, setmalloctag() 69a, strcpy() 81c, and strlen() 80a.

⟨*function* s_newalloc 93b⟩≡ (584d)
```
/* create a new 'short' String */
extern String *
s_newalloc(int len)
{
    String *sp;

    sp = _s_alloc();
    if(sp == nil)
        sysfatal("s_newalloc: %r");
    setmalloctag(sp, getcallerpc(&len));
    if(len < STRLEN)
        len = STRLEN;
    sp->base = sp->ptr = malloc(len);
    if (sp->base == nil)
        sysfatal("s_newalloc: %r");
    setmalloctag(sp->base, getcallerpc(&len));

    sp->end = sp->base + len;
    s_terminate(sp);
    return sp;
}
```
Uses STRLEN-334 92e, _s_alloc() 92f, getcallerpc() 396f, malloc() 63g, s_terminate() 95a, setmalloctag() 69a, and sysfatal() 236a.

`s_free()`[93c] decrements the reference count under a lock. If other references remain, it returns without freeing. Only when the count drops to zero does it free the buffer (unless `fixed`) and the header. This is the reference-counting half of the copy-on-write scheme—sharing is cheap, copying only happens when a shared string is mutated.

⟨*function* s_free 93c⟩≡ (584d)
```
extern void
s_free(String *sp)
{
    if (sp == nil)
        return;
    lock(sp);
    if(--(sp->ref) != 0){
        unlock(sp);
        return;
    }
    unlock(sp);
```

```
    if(sp->fixed == 0 && sp->base != nil)
        free(sp->base);
    free(sp);
}
```

Uses `free()` 63i, `lock()` 245b, and `unlock()` 245c.

9.6.3 Growing: `s_putc()`, `s_grow()`, `s_terminate()`

These three functions form the low-level append machinery. `s_putc()`[94a] appends a single byte, growing the buffer if `ptr` has reached `end`. It refuses to modify shared strings (`ref > 1`)—the caller must call `s_unique()`[100c] first. `s_grow()`[94b] reallocates using amortized doubling: it grows by at least the requested increment, but if that increment is less than half the current size, it grows by 50% instead—this keeps the total cost of n appends at $O(n)$ rather than $O(n^2)$. `s_terminate()`[95a] writes the null terminator at `ptr` without advancing it, growing if necessary.

⟨*function* `s_putc` 94a⟩≡ (585f)

```
void
s_putc(String *s, int c)
{
    if(s->ref > 1)
        sysfatal("can't s_putc a shared string");
    if (s->ptr >= s->end)
        s_grow(s, 2);
    *(s->ptr)++ = c;
}
```

Uses `s_grow()` 94b and `sysfatal()` 236a.

⟨*function* `s_grow` 94b⟩≡ (585b)

```
/* grow a String's allocation by at least 'incr' bytes */
extern String*
s_grow(String *s, int incr)
{
    char *cp;
    int size;

    if(s->fixed)
        sysfatal("s_grow of constant string");
    s = s_unique(s);

    /*
     *  take a larger increment to avoid mallocing too often
     */
    size = s->end-s->base;
    if(size/2 < incr)
        size += incr;
    else
        size += size/2;

    cp = realloc(s->base, size);
    if (cp == 0)
        sysfatal("s_grow: %r");
    s->ptr = (s->ptr - s->base) + cp;
    s->end = cp + size;
    s->base = cp;

    return s;
}
```

Uses `realloc()` 67b, `s_unique()` 100c, and `sysfatal()` 236a.

⟨*function* s_terminate 95a⟩≡ (586d)
```
void
s_terminate(String *s)
{
    if(s->ref > 1)
        sysfatal("can't s_terminate a shared string");
    if (s->ptr >= s->end)
        s_grow(s, 1);
    *s->ptr = 0;
}
```
Uses s_grow() 94b and sysfatal() 236a.

9.6.4 Appending: s_append(), s_nappend(), s_memappend()

Three variants for appending data to a String. s_append()[95b] appends a null-terminated C string. s_nappend()[95c] appends up to n characters, stopping early at a null terminator. s_memappend()[95d] appends exactly n bytes regardless of null bytes—useful for binary data. All three create a new string if passed nil, call s_putc()[94a] in a loop, and null-terminate at the end.

⟨*function* s_append 95b⟩≡ (584e)
```
/* append a char array to a String */
String *
s_append(String *to, char *from)
{
    if (to == 0)
        to = s_new();
    if (from == 0)
        return to;
    for(; *from; from++)
        s_putc(to, *from);
    s_terminate(to);
    return to;
}
```
Uses s_new() 92d, s_putc() 94a, and s_terminate() 95a.

⟨*function* s_nappend 95c⟩≡ (585d)
```
/* append a char array ( of up to n characters) to a String */
String *
s_nappend(String *to, char *from, int n)
{
    if (to == 0)
        to = s_new();
    if (from == 0)
        return to;
    for(; n && *from; from++, n--)
        s_putc(to, *from);
    s_terminate(to);
    return to;
}
```

⟨*function* s_memappend 95d⟩≡ (585c)
```
/* append a char array ( of up to n characters) to a String */
String *
s_memappend(String *to, char *from, int n)
{
    char *e;

    if (to == 0)
```

```
        to = s_new();
    if (from == 0)
        return to;
    for(e = from + n; from < e; from++)
        s_putc(to, *from);
    s_terminate(to);
    return to;
}
```

Uses s_new() 92d, s_putc() 94a, and s_terminate() 95a.

9.6.5 Resetting: s_reset(), s_restart()

Both functions rewind the write pointer to base, effectively clearing the string without freeing the buffer. s_reset()[96a] also null-terminates and handles nil by creating a new string. s_restart()[96b] just moves ptr back without terminating—useful when the caller will immediately start appending new content.

⟨*function* s_reset 96a⟩≡ (586c)

```
String*
s_reset(String *s)
{
    if(s != nil){
        s = s_unique(s);
        s->ptr = s->base;
        *s->ptr = '\0';
    } else
        s = s_new();
    return s;
}
```

Uses s_new() 92d and s_unique() 100c.

⟨*function* s_restart 96b⟩≡ (586c)

```
String*
s_restart(String *s)
{
    s = s_unique(s);
    s->ptr = s->base;
    return s;
}
```

Uses s_unique() 100c.

9.6.6 I/O: s_read_line(), s_read(), s_getline()

These functions read from a Biobuf[195a] (see Chapter 14) into a String, combining libbio's buffered reading with libstring's growable storage. They are only available when bio.h has been included (guarded by #ifdef BGETC in str.h).

s_read_line()[96c] appends one line using Brdline()[202] (the zero-copy line reader), copying the result into the String. The trailing newline is preserved. s_read()[97a] appends up to len raw bytes using Bread()[206], growing the string as needed. s_getline()[97b] is the most sophisticated: it strips leading whitespace, skips comment lines starting with #, and handles backslash-newline continuation. This is the format used by Plan 9 configuration files like /lib/ndb/local.

⟨*function* s_read_line 96c⟩≡ (586b)

```
/* Append an input line to a String.
 *
 * Returns a pointer to the character string (or 0).
 * Trailing newline is left on.
```

```
 */
extern char *
s_read_line(Biobuf *fp, String *to)
{
    char *cp;
    int llen;

    if(to->ref > 1)
        sysfatal("can't s_read_line a shared string");
    s_terminate(to);
    cp = Brdline(fp, '\n');
    if(cp == 0)
        return 0;
    llen = Blinelen(fp);
    if(to->end - to->ptr < llen)
        s_grow(to, llen);
    memmove(to->ptr, cp, llen);
    cp = to->ptr;
    to->ptr += llen;
    s_terminate(to);
    return cp;
}
```

Uses `Blinelen()` 212c, `Brdline()` 202, `memmove()` 48, `s_grow()` 94b, `s_terminate()` 95a, and `sysfatal()` 236a.

⟨*function* `s_read` 97a⟩≡ (586a)

```
/* Append up to 'len' input bytes to the string 'to'.
 *
 * Returns the number of characters read.
 */
extern int
s_read(Biobuf *fp, String *to, int len)
{
    int rv;
    int n;

    if(to->ref > 1)
        sysfatal("can't s_read a shared string");
    for(rv = 0; rv < len; rv += n){
        n = to->end - to->ptr;
        if(n < Minread){
            s_grow(to, Minread);
            n = to->end - to->ptr;
        }
        if(n > len - rv)
            n = len - rv;
        n = Bread(fp, to->ptr, n);
        if(n <= 0)
            break;
        to->ptr += n;
    }
    s_terminate(to);
    return rv;
}
```

Uses `Bread()` 206, `Minread-335` 586a, `s_grow()` 94b, `s_terminate()` 95a, and `sysfatal()` 236a.

⟨*function* `s_getline` 97b⟩≡ (585a)

```
/* Append an input line to a String.
 *
 * Returns a pointer to the character string (or 0).
 * Leading whitespace and newlines are removed.
```

```
 *
 * Empty lines and lines starting with '#' are ignored.
 */
extern char *
s_getline(Biobuf *fp, String *to)
{
    int c;
    int len=0;

    s_terminate(to);

    /* end of input */
    if ((c = Bgetc(fp)) < 0)
        return 0;

    /* take care of inconsequentials */
    for(;;) {
        /* eat leading white */
        while(c==' ' || c=='\t' || c=='\n' || c=='\r')
            c = Bgetc(fp);

        if(c < 0)
            return 0;

        /* take care of comments */
        if(c == '#'){
            do {
                c = Bgetc(fp);
                if(c < 0)
                    return 0;
            } while(c != '\n');
            continue;
        }

        /* if we got here, we've gotten something useful */
        break;
    }

    /* gather up a line */
    for(;;) {
        len++;
        switch(c) {
        case -1:
            s_terminate(to);
            return len ? to->ptr-len : 0;
        case '\\':
            c = Bgetc(fp);
            if (c != '\n') {
                s_putc(to, '\\');
                s_putc(to, c);
            }
            break;
        case '\n':
            s_terminate(to);
            return len ? to->ptr-len : 0;
        default:
            s_putc(to, c);
            break;
        }
        c = Bgetc(fp);
```

```
    }
}
```

Uses Bgetc() 200a, s_putc() 94a, and s_terminate() 95a.

9.6.7 Parsing: s_parse()

s_parse()[99b] extracts the next whitespace-delimited field from a String, advancing ptr past it. Fields can be single-quoted or double-quoted to include whitespace. The extracted field is appended to a destination String (or a freshly allocated one if nil). This is used for parsing configuration lines where fields may contain spaces—for example, mail addresses or file paths in Plan 9 system files.

⟨*macro* isspace 99a⟩≡ (585e)

```
#define isspace(c) ((c)==' ' || (c)=='\t' || (c)=='\n')
```

⟨*function* s_parse 99b⟩≡ (585e)

```
/* Get the next field from a String.  The field is delimited by white space,
 * single or double quotes.
 */
String *
s_parse(String *from, String *to)
{
    if (*from->ptr == '\0')
        return 0;
    if (to == 0)
        to = s_new();
    if (*from->ptr == '\'') {
        from->ptr++;
        for (;*from->ptr != '\'' && *from->ptr != '\0'; from->ptr++)
            s_putc(to, *from->ptr);
        if (*from->ptr == '\'')
            from->ptr++;
    } else if (*from->ptr == '"') {
        from->ptr++;
        for (;*from->ptr != '"' && *from->ptr != '\0'; from->ptr++)
            s_putc(to, *from->ptr);
        if (*from->ptr == '"')
            from->ptr++;
    } else {
        for (;!isspace(*from->ptr) && *from->ptr != '\0'; from->ptr++)
            s_putc(to, *from->ptr);
    }
    s_terminate(to);

    /* crunch trailing white */
    while(isspace(*from->ptr))
        from->ptr++;

    return to;
}
```

Uses isspace-333 99a, s_new() 92d, s_putc() 94a, and s_terminate() 95a.

9.6.8 Miscellaneous: s_tolower(), s_rdinstack()

s_tolower()[100a] converts the string content to lower case in place. s_incref()[100b] and s_unique()[100c] implement the copy-on-write mechanism: s_incref() bumps the reference count (under a lock, for thread safety), while s_unique() ensures the caller has an exclusive copy—if ref > 1, it clones the string and decrements the

original's reference. `s_array()`[100d] wraps a caller-owned buffer as a `String` with `fixed=1`, meaning it must not be reallocated or freed.

`s_rdinstack()`[102] reads lines from a stack of files, following `#include` directives to descend into nested files (up to 32 deep). When an included file reaches EOF, it pops back to the parent. This is used by Plan 9's mail system for reading configuration files with includes.

⟨*function* `s_tolower` 100a⟩≡ (586e)

```
/* convert String to lower case */
void
s_tolower(String *sp)
{
    char *cp;

    for(cp=sp->ptr; *cp; cp++)
        *cp = tolower(*cp);
}
```

Uses `tolower()` 29d.

⟨*function* `s_incref` 100b⟩≡ (584d)

```
/* get another reference to a string */
extern String *
s_incref(String *sp)
{
    lock(sp);
    sp->ref++;
    unlock(sp);

    return sp;
}
```

Uses `lock()` 245b and `unlock()` 245c.

⟨*function* `s_unique` 100c⟩≡ (586f)

```
String*
s_unique(String *s)
{
    String *p;

    if(s->ref > 1){
        p = s;
        s = s_clone(p);
        s_free(p);
    }
    return s;
}
```

Uses `s_free()` 93c.

⟨*function* `s_array` 100d⟩≡ (584f)

```
/* return a String containing a character array (this had better not grow) */
extern String *
s_array(char *cp, int len)
{
    String *sp = _s_alloc();

    sp->base = sp->ptr = cp;
    sp->end = sp->base + len;
    sp->fixed = 1;
    return sp;
}
```

Uses `_s_alloc()` 92f.

⟨*function* `s_allocinstack` 101a⟩≡ (585g)
```
/* initialize */
extern Sinstack *
s_allocinstack(char *file)
{
    Sinstack *sp;
    Biobuf *fp;

    fp = Bopen(file, OREAD);
    if(fp == nil)
        return nil;

    sp = malloc(sizeof *sp);
    sp->depth = 0;
    sp->fp[0] = fp;
    return sp;
}
```
Uses `Bopen()` 198d and `malloc()` 63g.

⟨*function* `s_freeinstack` 101b⟩≡ (585g)
```
extern void
s_freeinstack(Sinstack *sp)
{
    while(sp->depth >= 0)
        Bterm(sp->fp[sp->depth--]);
    free(sp);
}
```
Uses `Bterm()` 199c and `free()` 63i.

⟨*function* `rdline` 101c⟩≡ (585g)
```
/*  Append an input line to a String.
 *
 *  Empty lines and leading whitespace are removed.
 */
static char *
rdline(Biobuf *fp, String *to)
{
    int c;
    int len = 0;

    c = Bgetc(fp);

    /* eat leading white */
    while(c==' ' || c=='\t' || c=='\n' || c=='\r')
        c = Bgetc(fp);

    if(c < 0)
        return 0;

    for(;;){
        switch(c) {
        case -1:
            goto out;
        case '\\':
            c = Bgetc(fp);
            if (c != '\n') {
                s_putc(to, '\\');
                s_putc(to, c);
                len += 2;
            }
```

```
            break;
        case '\r':
            break;
        case '\n':
            if(len != 0)
                goto out;
            break;
        default:
            s_putc(to, c);
            len++;
            break;
        }
        c = Bgetc(fp);
    }
out:
    s_terminate(to);
    return to->ptr - len;
}
```

Uses Bgetc() 200a, s_putc() 94a, and s_terminate() 95a.

⟨*function* s_rdinstack 102⟩≡ (585g)

```
/* Append an input line to a String.
 *
 * Returns a pointer to the character string (or 0).
 * Leading whitespace and newlines are removed.
 * Lines starting with #include cause us to descend into the new file.
 * Empty lines and other lines starting with '#' are ignored.
 */
extern char *
s_rdinstack(Sinstack *sp, String *to)
{
    char *p;
    Biobuf *fp, *nfp;

    s_terminate(to);
    fp = sp->fp[sp->depth];

    for(;;){
        p = rdline(fp, to);
        if(p == nil){
            if(sp->depth == 0)
                break;
            Bterm(fp);
            sp->depth--;
            return s_rdinstack(sp, to);
        }

        if(strncmp(p, "#include", 8) == 0 && (p[8] == ' ' || p[8] == '\t')){
            to->ptr = p;
            p += 8;

            /* sanity (and looping) */
            if(sp->depth >= nelem(sp->fp))
                sysfatal("s_recgetline: includes too deep");

            /* skip white */
            while(*p == ' ' || *p == '\t')
                p++;

            nfp = Bopen(p, OREAD);
```

```
            if(nfp == nil)
                continue;
            sp->depth++;
            sp->fp[sp->depth] = nfp;
            return s_rdinstack(sp, to);
        }

        /* got milk? */
        if(*p != '#')
            break;

        /* take care of comments */
        to->ptr = p;
        s_terminate(to);
    }
    return p;
}
```

Uses Bopen() 198d, Bterm() 199c, rdline() 101c, s_rdinstack() 102, s_terminate() 95a, strncmp() 443c, and sysfatal() 236a.

Chapter 10

Regular Expressions

`libregexp` (declared in `regexp.h`) provides regular expression compilation and matching. The API has three steps: compile a pattern into a `Reprog`[105b] with `regcomp()`[107a], match it against a string with `regexec()`[109], and optionally perform substitution with `regsub()`[113]. The library handles UTF-8 natively—patterns and strings are byte strings, but character classes and `.` operate on runes, not bytes. Rune-string variants (`rregexec()`[117b], `rregsub()`[118]) are also provided.

10.1 Regular expression principles

Before diving into `libregexp`'s data structures, it helps to step back. A regex engine is a two-phase tool: a *compiler* turns a pattern string (the tiny language of alternation, repetition, character classes, and grouping) into some internal form, and a *matcher* runs that internal form against input strings. The API usually hides both phases behind two or three function calls, so it is easy to assume the implementation question is uninteresting. It is not. Different engines pick radically different internal forms, and the choice determines whether a match against a sufficiently nasty pattern takes microseconds or days. The textbook example is `(a?){30}a{30}` matched against thirty `a`'s: work that is almost free in theory, yet hangs some popular engines for minutes.

Three families of regex engines dominate in practice. *NFA simulation*, introduced by Ken Thompson in 1968 for the CTSS editor `qed` and soon reused in `ed` and `grep`, compiles the pattern to a nondeterministic finite automaton and at match time tracks the *set* of all states the automaton could currently be in. It runs in linear time and linear memory, and it is the algorithm behind Plan 9's `libregexp`, Go's `regexp` package, the RE2 library, and Rust's `regex` crate. *DFA compilation* converts the NFA to a deterministic automaton up front by powerset construction, making matching one table lookup per input byte—the fastest inner loop around—at the risk of exponential state blow-up, so real implementations fall back to NFA simulation on tricky patterns or build the DFA lazily on first use. This is the approach of classic `lex` and `flex`, of `fgrep` and Aho-Corasick multi-string matchers, and (with lazy construction) of RE2. *Backtracking interpretation* compiles the pattern to a syntax tree and matches by recursive trial and error, saving a stack frame at every alternation so the matcher can back up and try the other branch. It is easy to implement and supports features the other two approaches cannot—back-references, look-behind, recursive patterns—but it is exponential in the worst case, and the pattern `^(a+)+$` matched against `aaaaaaaaaaaaaaaaX` is the textbook disaster case that underlies almost every "ReDoS" regex denial-of-service CVE. Perl, PCRE, Python's `re`, JavaScript's `RegExp`, Ruby, and Java's `java.util.regex` all backtrack.

The key insight behind the NFA approach, due to Thompson, is that at any point during matching you do not need to guess which branch of an alternation to take—you can simulate all branches in parallel by tracking the *full set* of currently-active NFA states. At each input character you advance every active state through its transition function, producing a new set. The set is bounded by the number of states in the NFA (linear in the pattern length), and you advance it exactly once per input character, so the total work is bounded by pattern-length × input-length. The approach is equivalent to building a DFA on the fly, with the subset of states

reachable from the current position computed on demand rather than memoized—which is why it avoids both the exponential matching time of backtracking and the exponential space blow-up of precomputed DFAs. Russ Cox's series of articles at https://swtch.com/~rsc/regexp/ is the standard modern exposition and should be required reading for anyone touching regex performance.

Plan 9's `libregexp` is a specific variant of Thompson's idea known as the *Pike VM*, named after Rob Pike who first encoded it this way in 1987. Instead of representing the NFA as a graph of state pointers, the compiler emits a small *bytecode program* whose instructions are roughly `RUNE` (match one rune and advance), `OR` (nondeterministic branch to two successors), `CCLASS`/`NCCLASS` (character class), `LBRA`/`RBRA` (record the start or end of a submatch), `BOL`/`EOL` (anchors), and `END`. The matcher is then a tiny virtual machine that tracks the set of currently-active "program counters"—confusingly called *threads* in the Pike VM literature, with no relation to operating-system threads—and advances them all on each input character until one reaches `END`. This is why the `Reprog` struct below calls its `firstinst` array "the `.text` segment" and its `class` array "the `.data` segment": a compiled regex really is a little program. The rest of this chapter is the compiler for that program (`regcomp`), the interpreter that runs it (`regexec`), and the substituter that layers text rewriting on top (`regsub`).

A typical use looks like this:

```
Reprog *prog;
Resub match[10];

prog = regcomp("([0-9]+)-([0-9]+)");
if(regexec(prog, "port 80-443", match, 10)) {
    /* match[0] is the whole match: "80-443" */
    /* match[1] is first group: "80" */
    /* match[2] is second group: "443" */
}
free(prog);
```

⟨*signatures* `regxxxx` *functions* 105a⟩≡ (383a)

```
extern Reprog   *regcomp(char*);
extern int  regexec(Reprog*, char*, Resub*, int);
extern void regsub(char*, char*, int, Resub*, int);
```

10.2 Data structures: `Reprog`, `Reinst`, `Reclass`, `Resub`

Mechanically, a `Reprog` is three fields: `startinst`, a pointer to the instruction where matching begins (the Pike VM's initial program counter); `class`, the array of character-class tables; and `firstinst`, the instruction array itself. `firstinst` is declared with only five entries, but `regcomp()`[107a] mallocs a `Reprog`[105b] large enough for the full compiled program and lets the array extend past the end of the struct—the standard C "struct hack" used when a trailing array's length is only known at allocation time.

⟨*struct* `Reprog` 105b⟩≡ (383a)

```
/*
 *  Reprogram definition
 */
struct Reprog{
    Reinst  *startinst; /* start pc */
    Reclass class[16];  /* .data */
    Reinst  firstinst[5];   /* .text */
};
```

Each instruction (`Reinst`[106a]) has a `type` and two union fields whose meaning depends on the type:

```
Type     Field 1              Field 2
```

```
------   -----------------   ------------------
RUNE     r (char to match)   next (next inst)
ANY      (unused)            next
CCLASS   cp (class pointer)  next
NCCLASS  cp (negated class)  next
BOL      (unused)            next
EOL      (unused)            next
OR       right (alt branch)  left (main branch)
LBRA     subid (group num)   next
RBRA     subid (group num)   next
END      (unused)            (unused)
```

The second union overlays `left` and `next`—for `OR` instructions, `left` is the main branch and `right` is the alternative; for all other instructions, `next` points to the following instruction. This dual use saves space and is safe because `OR` never uses `next`.

⟨*struct* `Reinst` 106a⟩≡ (383a)

```
/*
 *  Machine instructions
 */
struct Reinst{
    int type;
    union   {
        Reclass *cp;        /* class pointer */
        Rune    r;      /* character */
        int subid;      /* sub-expression id for RBRA and LBRA */
        Reinst  *right;     /* right child of OR */
    };
    union { /* regexp relies on these two being in the same union */
        Reinst *left;       /* left child of OR */
        Reinst *next;       /* next instruction for CAT & LBRA */
    };
};
```

A character class like `[a-zA-Z0-9]` is stored as an array of rune pairs in `Reclass`[106b]. Each pair defines an inclusive range: `spans[0]`–`spans[1]` is the first range, `spans[2]`–`spans[3]` the second, and so on. The `end` pointer marks the end of the valid pairs. Matching is a linear scan: if the input rune falls within any range, the class matches.

⟨*struct* `Reclass` 106b⟩≡ (383a)

```
/*
 *  character class, each pair of rune's defines a range
 */
struct Reclass{
    Rune    *end;
    Rune    spans[64];
};
```

`Resub`[106c] records where a subexpression matched. Each element has a start pointer (`s.sp` or `s.rsp`) and an end pointer (`e.ep` or `e.rep`), using unions so the same structure works for both `char*` and `Rune*` strings. Element 0 is always the overall match; elements 1 through n correspond to parenthesized groups.

⟨*struct* `Resub` 106c⟩≡ (383a)

```
/*
 *  Sub expression matches
 */
struct Resub{
    // Start/End matches
    // old: kencc-ext: this used to be anon unions but gcc/clang don't accept it
```

```
    // and for goken we need mk to be compatible with a libregexp gcc/clang can
    // compile hence the }s; and }e; below
    union
    {
        char *sp;
        Rune *rsp;
    }s;
    union
    {
        char *ep;
        Rune *rep;
    }e;
};
```

10.3 Compilation: regcomp()

`regcomp()`[107a] parses a regular expression string and compiles it into a `Reprog`[105b]. The compiler is a classic operator-precedence parser with two stacks: an operand stack (`andstack`) holding NFA fragments, and an operator stack (`atorstack`) holding pending operators. Implicit concatenation is inserted between adjacent operands. The output is a linear array of `Reinst`[106a] instructions that encode the NFA.

Three entry points are provided: `regcomp()` treats `.` as "any character except newline" (the usual behavior), `regcomplit` treats `.` as "any character except newline" but also makes character classes case-insensitive, and `regcompnl` treats `.` as "any character including newline"—useful for multi-line matching.

For example, the pattern `a(b|c)*d` compiles to:

```
0: RUNE 'a'  -> 1
1: LBRA 1    -> 2      (start of group 1)
2: OR        -> 3 / 5  (left=3, right=5)
3: RUNE 'b'  -> 4
4: NOP       -> 6
5: RUNE 'c'  -> 6
6: RBRA 1    -> 7      (end of group 1)
7: OR        -> 2 / 8  (* = loop back or continue)
8: RUNE 'd'  -> 9
9: END
```

⟨*function* `regcomp` 107a⟩≡ (581c)

```
extern Reprog*
regcomp(char *s)
{
    return regcomp1(s, 0, ANY);
}
```

Uses `ANY` 569b and `regcomp1()` 107b.

⟨*function* `regcomp1` 107b⟩≡ (581c)

```
static Reprog*
regcomp1(char *s, int literal, int dot_type)
{
    int token;
    Reprog *pp;

    /* get memory for the program */
    pp = malloc(sizeof(Reprog) + 6*sizeof(Reinst)*strlen(s));
    if(pp == 0){
        regerror("out of memory");
```

```
        return 0;
    }
    freep = pp->firstinst;
    classp = pp->class;
    errors = 0;

    if(setjmp(regkaboom))
        goto out;

    /* go compile the sucker */
    lexdone = 0;
    exprp = s;
    nclass = 0;
    nbra = 0;
    atorp = atorstack;
    andp = andstack;
    subidp = subidstack;
    lastwasand = FALSE;
    cursubid = 0;

    /* Start with a low priority operator to prime parser */
    pushator(START-1);
    while((token = lex(literal, dot_type)) != END){
        if((token&0300) == OPERATOR)
            operator(token);
        else
            operand(token);
    }

    /* Close with a low priority operator */
    evaluntil(START);

    /* Force END */
    operand(END);
    evaluntil(START);
#ifdef DEBUG
    dumpstack();
#endif
    if(nbra)
        rcerror("unmatched left paren");
    --andp; /* points to first and only operand */
    pp->startinst = andp->first;
#ifdef DEBUG
    dump(pp);
#endif
    pp = optimize(pp);
#ifdef DEBUG
    print("start: %d\n", andp->first-pp->firstinst);
    dump(pp);
#endif
out:
    if(errors){
        free(pp);
        pp = 0;
    }
    return pp;
}
```

Uses END 569b, OPERATOR 569b, START 569b, andp-340 573a, andstack-339 572h, atorp-342 573c, atorstack-341 573b, classp-351 573l, cursubid-343 573d, errors-353 573n, evaluntil() 576b, exprp-348 573i, free() 63i, freep-352 573m,

lastwasand-346 573g, lex() 579b, lexdone-349 573j, malloc() 63g, nbra-347 573h, nclass-350 573k, operand() 574b, operator() 574c, optimize() 577, pushator() 575c, rcerror() 573r, regerror() 119, regkaboom-356 573q, strlen() 80a, subidp-345 573f, and subidstack-344 573e.

10.4 Execution: `regexec()`

`regexec()`[109] runs the compiled NFA against a string using Thompson's simulation. The key idea is to maintain two lists of active threads (NFA states): the "current list" for the character being processed and the "next list" for the character that follows. At each input character, every thread in the current list is advanced: if a thread's instruction matches the character, a new thread is added to the next list; if it doesn't match, the thread dies. After processing one character, the lists are swapped and the cycle repeats.

The algorithm guarantees $O(nm)$ time where n is the pattern size and m is the string length, because each character is processed once and the list can never exceed n entries (one per NFA state, with duplicates suppressed by `_renewthread()`[570b]).

Two optimizations are notable. First, if the pattern starts with a literal character or `^`, `regexec()` uses `utfrune` or `strchr` to skip ahead to the first possible match position—avoiding the NFA overhead for positions that can't possibly match. Second, execution starts with small stack-allocated lists (`LISTSIZE` = 10 entries); if these overflow, `regexec2()`[112] retries with heap-allocated lists 25 times larger (`BIGLISTSIZE`).

⟨*function* regexec 109⟩≡ (583b)

```
extern int
regexec(Reprog *progp, /* program to run */
    char *bol, /* string to run machine on */
    Resub *mp, /* subexpression elements */
    int ms)  /* number of elements at mp */
{
    Reljunk j;
    Relist relist0[LISTSIZE], relist1[LISTSIZE];
    int rv;

    /*
   *  use user-specified starting/ending location if specified
     */
    j.starts = bol;
    j.eol = 0;
    if(mp && ms>0){
        if(mp->s.sp)
            j.starts = mp->s.sp;
        if(mp->e.ep)
            j.eol = mp->e.ep;
    }
    j.starttype = 0;
    j.startchar = 0;
    if(progp->startinst->type == RUNE && progp->startinst->r < Runeself) {
        j.starttype = RUNE;
        j.startchar = progp->startinst->r;
    }
    if(progp->startinst->type == BOL)
        j.starttype = BOL;

    /* mark space */
    j.relist[0] = relist0;
    j.relist[1] = relist1;
    j.reliste[0] = relist0 + nelem(relist0) - 2;
    j.reliste[1] = relist1 + nelem(relist1) - 2;
```

```
    rv = regexec1(progp, bol, mp, ms, &j);
    if(rv >= 0)
        return rv;
    rv = regexec2(progp, bol, mp, ms, &j);
    if(rv >= 0)
        return rv;
    return -1;
}
```

Uses BOL 569b, LISTSIZE 569b, RUNE 569b, regexec1() 110, and regexec2() 112.

⟨*function* regexec1 110⟩≡ (583b)

```
/*
 *  return 0 if no match
 *  >0 if a match
 *  <0 if we ran out of _relist space
 */
static int
regexec1(Reprog *progp, /* program to run */
    char *bol, /* string to run machine on */
    Resub *mp, /* subexpression elements */
    int ms,  /* number of elements at mp */
    Reljunk *j
)
{
    int flag=0;
    Reinst *inst;
    Relist *tlp;
    char *s;
    int i, checkstart;
    Rune r, *rp, *ep;
    int n;
    Relist* tl;  /* This list, next list */
    Relist* nl;
    Relist* tle;  /* ends of this and next list */
    Relist* nle;
    int match;
    char *p;

    match = 0;
    checkstart = j->starttype;
    if(mp)
        for(i=0; i<ms; i++) {
            mp[i].s.sp = 0;
            mp[i].e.ep = 0;
        }
    j->relist[0][0].inst = 0;
    j->relist[1][0].inst = 0;

    /* Execute machine once for each character, including terminal NUL */
    s = j->starts;
    do{
        /* fast check for first char */
        if(checkstart) {
            switch(j->starttype) {
            case RUNE:
                p = utfrune(s, j->startchar);
                if(p == 0 || s == j->eol)
                    return match;
                s = p;
                break;
```

```
    case BOL:
        if(s == bol)
            break;
        p = utfrune(s, '\n');
        if(p == 0 || s == j->eol)
            return match;
        s = p+1;
        break;
    }
}
r = *(uchar*)s;
if(r < Runeself)
    n = 1;
else
    n = chartorune(&r, s);

/* switch run lists */
tl = j->relist[flag];
tle = j->reliste[flag];
nl = j->relist[flag^=1];
nle = j->reliste[flag];
nl->inst = 0;

/* Add first instruction to current list */
if(match == 0)
    _renewemptythread(tl, progp->startinst, ms, s);

/* Execute machine until current list is empty */
for(tlp=tl; tlp->inst; tlp++){ /* assignment = */
    for(inst = tlp->inst; ; inst = inst->next){
        switch(inst->type){
        case RUNE: /* regular character */
            if(inst->r == r){
                if(_renewthread(nl, inst->next, ms, &tlp->se)==nle)
                    return -1;
            }
            break;
        case LBRA:
            tlp->se.m[inst->subid].s.sp = s;
            continue;
        case RBRA:
            tlp->se.m[inst->subid].e.ep = s;
            continue;
        case ANY:
            if(r != '\n')
                if(_renewthread(nl, inst->next, ms, &tlp->se)==nle)
                    return -1;
            break;
        case ANYNL:
            if(_renewthread(nl, inst->next, ms, &tlp->se)==nle)
                    return -1;
            break;
        case BOL:
            if(s == bol || *(s-1) == '\n')
                continue;
            break;
        case EOL:
            if(s == j->eol || r == 0 || r == '\n')
                continue;
            break;
```

```
                case CCLASS:
                    ep = inst->cp->end;
                    for(rp = inst->cp->spans; rp < ep; rp += 2)
                        if(r >= rp[0] && r <= rp[1]){
                            if(_renewthread(nl, inst->next, ms, &tlp->se)==nle)
                                return -1;
                            break;
                        }
                    break;
                case NCCLASS:
                    ep = inst->cp->end;
                    for(rp = inst->cp->spans; rp < ep; rp += 2)
                        if(r >= rp[0] && r <= rp[1])
                            break;
                    if(rp == ep)
                        if(_renewthread(nl, inst->next, ms, &tlp->se)==nle)
                            return -1;
                    break;
                case OR:
                    /* evaluate right choice later */
                    if(_renewthread(tlp, inst->right, ms, &tlp->se) == tle)
                        return -1;
                    /* efficiency: advance and re-evaluate */
                    continue;
                case END: /* Match! */
                    match = 1;
                    tlp->se.m[0].e.ep = s;
                    if(mp != 0)
                        _renewmatch(mp, ms, &tlp->se);
                    break;
                }
                break;
            }
        }
        if(s == j->eol)
            break;
        checkstart = j->starttype && nl->inst==0;
        s += n;
    }while(r);
    return match;
}
```

Uses ANY 569b, ANYNL 569b, BOL 569b, CCLASS 569b, END 569b, EOL 569b, LBRA 569b, NCCLASS 569b, OR 569b, RBRA 569b, RUNE 569b, _renewemptythread() 571a, _renewmatch() 570a, _renewthread() 570b, chartorune() 84d, and utfrune() 89b.

⟨*function* regexec2 112⟩≡ (583b)

```
static int
regexec2(Reprog *progp, /* program to run */
    char *bol, /* string to run machine on */
    Resub *mp, /* subexpression elements */
    int ms,  /* number of elements at mp */
    Reljunk *j
)
{
    int rv;
    Relist *relist0, *relist1;

    /* mark space */
    relist0 = malloc(BIGLISTSIZE*sizeof(Relist));
    if(relist0 == nil)
        return -1;
```

```
    relist1 = malloc(BIGLISTSIZE*sizeof(Relist));
    if(relist1 == nil){
        free(relist1);
        return -1;
    }
    j->relist[0] = relist0;
    j->relist[1] = relist1;
    j->reliste[0] = relist0 + BIGLISTSIZE - 2;
    j->reliste[1] = relist1 + BIGLISTSIZE - 2;

    rv = regexec1(progp, bol, mp, ms, j);
    free(relist0);
    free(relist1);
    return rv;
}
```

Uses BIGLISTSIZE 569b, free() 63i, malloc() 63g, and regexec1() 110.

10.5 Substitution: regsub()

regsub()[113] performs substitution using the match results from regexec()[109]. It scans a template string (sp) and copies it to a destination buffer (dp), replacing backreferences as it goes: \0 through \9 expand to the corresponding subexpression match, and & expands to the entire match (mp[0]). The output is bounded by dlen to prevent buffer overflows.

⟨*function* regsub 113⟩≡ (583c)

```
/* substitute into one string using the matches from the last regexec() */
extern void
regsub(char *sp, /* source string */
    char *dp, /* destination string */
    int dlen,
    Resub *mp, /* subexpression elements */
    int ms)  /* number of elements pointed to by mp */
{
    char *ssp, *ep;
    int i;

    ep = dp+dlen-1;
    while(*sp != '\0'){
        if(*sp == '\\'){
            switch(*++sp){
            case '0':
            case '1':
            case '2':
            case '3':
            case '4':
            case '5':
            case '6':
            case '7':
            case '8':
            case '9':
                i = *sp-'0';
                if(mp!=0 && mp[i].s.sp != 0 && ms>i)
                    for(ssp = mp[i].s.sp;
                         ssp < mp[i].e.ep;
                         ssp++)
                        if(dp < ep)
                            *dp++ = *ssp;
                break;
```

```
            case '\\':
                if(dp < ep)
                    *dp++ = '\\';
                break;
            case '\0':
                sp--;
                break;
            default:
                if(dp < ep)
                    *dp++ = *sp;
                break;
            }
        }else if(*sp == '&'){
            if(mp!=0 && mp[0].s.sp != 0 && ms>0)
                for(ssp = mp[0].s.sp;
                     ssp < mp[0].e.ep; ssp++)
                    if(dp < ep)
                        *dp++ = *ssp;
        }else{
            if(dp < ep)
                *dp++ = *sp;
        }
        sp++;
    }
    *dp = '\0';
}
```

10.6 Rune variants: rregexec(), rregsub()

`rregexec()`[117b] and `rregsub()`[118] are the `Rune*` counterparts of `regexec()`[109] and `regsub()`[113]. They operate on arrays of decoded runes rather than UTF-8 byte strings, which simplifies the inner loop—each element is one character, so there is no need for `chartorune()`[84d] decoding. The compiled `Reprog`[105b] is shared: the same program works for both byte and rune execution because the instruction set operates on rune values.

⟨*function* rregexec1 114⟩≡ (583d)

```
/*
 *  return 0 if no match
 *  >0 if a match
 *  <0 if we ran out of _relist space
 */
static int
rregexec1(Reprog *progp, /* program to run */
    Rune *bol,  /* string to run machine on */
    Resub *mp,  /* subexpression elements */
    int ms,   /* number of elements at mp */
    Reljunk *j)
{
    int flag=0;
    Reinst *inst;
    Relist *tlp;
    Rune *s;
    int i, checkstart;
    Rune r, *rp, *ep;
    Relist* tl;  /* This list, next list */
    Relist* nl;
    Relist* tle;  /* ends of this and next list */
    Relist* nle;
    int match;
```

```
Rune *p;

match = 0;
checkstart = j->startchar;
if(mp)
    for(i=0; i<ms; i++) {
        mp[i].s.rsp = 0;
        mp[i].e.rep = 0;
    }
j->relist[0][0].inst = 0;
j->relist[1][0].inst = 0;

/* Execute machine once for each character, including terminal NUL */
s = j->rstarts;
do{
    /* fast check for first char */
    if(checkstart) {
        switch(j->starttype) {
        case RUNE:
            p = runestrchr(s, j->startchar);
            if(p == 0 || s == j->reol)
                return match;
            s = p;
            break;
        case BOL:
            if(s == bol)
                break;
            p = runestrchr(s, '\n');
            if(p == 0 || s == j->reol)
                return match;
            s = p+1;
            break;
        }
    }

    r = *s;

    /* switch run lists */
    tl = j->relist[flag];
    tle = j->reliste[flag];
    nl = j->relist[flag^=1];
    nle = j->reliste[flag];
    nl->inst = 0;

    /* Add first instruction to current list */
    _rrenewemptythread(tl, progp->startinst, ms, s);

    /* Execute machine until current list is empty */
    for(tlp=tl; tlp->inst; tlp++){
        for(inst=tlp->inst; ; inst = inst->next){
            switch(inst->type){
            case RUNE: /* regular character */
                if(inst->r == r)
                    if(_renewthread(nl, inst->next, ms, &tlp->se)==nle)
                        return -1;
                break;
            case LBRA:
                tlp->se.m[inst->subid].s.rsp = s;
                continue;
            case RBRA:
```

```
				tlp->se.m[inst->subid].e.rep = s;
				continue;
			case ANY:
				if(r != '\n')
					if(_renewthread(nl, inst->next, ms, &tlp->se)==nle)
						return -1;
				break;
			case ANYNL:
				if(_renewthread(nl, inst->next, ms, &tlp->se)==nle)
						return -1;
				break;
			case BOL:
				if(s == bol || *(s-1) == '\n')
					continue;
				break;
			case EOL:
				if(s == j->reol || r == 0 || r == '\n')
					continue;
				break;
			case CCLASS:
				ep = inst->cp->end;
				for(rp = inst->cp->spans; rp < ep; rp += 2)
					if(r >= rp[0] && r <= rp[1]){
						if(_renewthread(nl, inst->next, ms, &tlp->se)==nle)
							return -1;
						break;
					}
				break;
			case NCCLASS:
				ep = inst->cp->end;
				for(rp = inst->cp->spans; rp < ep; rp += 2)
					if(r >= rp[0] && r <= rp[1])
						break;
				if(rp == ep)
					if(_renewthread(nl, inst->next, ms, &tlp->se)==nle)
						return -1;
				break;
			case OR:
				/* evaluate right choice later */
				if(_renewthread(tlp, inst->right, ms, &tlp->se) == tle)
					return -1;
				/* efficiency: advance and re-evaluate */
				continue;
			case END:	/* Match! */
				match = 1;
				tlp->se.m[0].e.rep = s;
				if(mp != 0)
					_renewmatch(mp, ms, &tlp->se);
				break;
			}
			break;
		}
	}
	if(s == j->reol)
		break;
	checkstart = j->startchar && nl->inst==0;
	s++;
}while(r);
return match;
}
```

Uses ANY 569b, ANYNL 569b, BOL 569b, CCLASS 569b, END 569b, EOL 569b, LBRA 569b, NCCLASS 569b, OR 569b, RBRA 569b, RUNE 569b, _renewmatch() 570a, _renewthread() 570b, _rrenewemptythread() 571b, and runestrchr() 87b.

⟨*function* rregexec2 117a⟩≡ (583d)

```
static int
rregexec2(Reprog *progp, /* program to run */
    Rune *bol, /* string to run machine on */
    Resub *mp, /* subexpression elements */
    int ms,  /* number of elements at mp */
    Reljunk *j
)
{
    Relist relist0[5*LISTSIZE], relist1[5*LISTSIZE];

    /* mark space */
    j->relist[0] = relist0;
    j->relist[1] = relist1;
    j->reliste[0] = relist0 + nelem(relist0) - 2;
    j->reliste[1] = relist1 + nelem(relist1) - 2;

    return rregexec1(progp, bol, mp, ms, j);
}
```

Uses LISTSIZE 569b and rregexec1() 114.

⟨*function* rregexec 117b⟩≡ (583d)

```
extern int
rregexec(Reprog *progp, /* program to run */
    Rune *bol, /* string to run machine on */
    Resub *mp, /* subexpression elements */
    int ms)  /* number of elements at mp */
{
    Reljunk j;
    Relist relist0[LISTSIZE], relist1[LISTSIZE];
    int rv;

    /*
   *  use user-specified starting/ending location if specified
     */
    j.rstarts = bol;
    j.reol = 0;
    if(mp && ms>0){
        if(mp->s.sp)
            j.rstarts = mp->s.rsp;
        if(mp->e.ep)
            j.reol = mp->e.rep;
    }
    j.starttype = 0;
    j.startchar = 0;
    if(progp->startinst->type == RUNE && progp->startinst->r < Runeself) {
        j.starttype = RUNE;
        j.startchar = progp->startinst->r;
    }
    if(progp->startinst->type == BOL)
        j.starttype = BOL;

    /* mark space */
    j.relist[0] = relist0;
    j.relist[1] = relist1;
    j.reliste[0] = relist0 + nelem(relist0) - 2;
    j.reliste[1] = relist1 + nelem(relist1) - 2;
```

```
    rv = rregexec1(progp, bol, mp, ms, &j);
    if(rv >= 0)
        return rv;
    rv = rregexec2(progp, bol, mp, ms, &j);
    if(rv >= 0)
        return rv;
    return -1;
}
```

Uses `BOL` 569b, `LISTSIZE` 569b, `RUNE` 569b, `rregexec1()` 114, and `rregexec2()` 117a.

⟨*function* `rregsub` 118⟩≡ (584a)

```
/* substitute into one string using the matches from the last regexec() */
extern void
rregsub(Rune *sp, /* source string */
    Rune *dp, /* destination string */
    int dlen,
    Resub *mp, /* subexpression elements */
    int ms)  /* number of elements pointed to by mp */
{
    Rune *ssp, *ep;
    int i;

    ep = dp+(dlen/sizeof(Rune))-1;
    while(*sp != '\0'){
        if(*sp == '\\'){
            switch(*++sp){
            case '0':
            case '1':
            case '2':
            case '3':
            case '4':
            case '5':
            case '6':
            case '7':
            case '8':
            case '9':
                i = *sp-'0';
                if(mp[i].s.rsp != 0 && mp!=0 && ms>i)
                    for(ssp = mp[i].s.rsp;
                         ssp < mp[i].e.rep;
                         ssp++)
                        if(dp < ep)
                            *dp++ = *ssp;
                break;
            case '\\':
                if(dp < ep)
                    *dp++ = '\\';
                break;
            case '\0':
                sp--;
                break;
            default:
                if(dp < ep)
                    *dp++ = *sp;
                break;
            }
        }else if(*sp == '&'){
            if(mp[0].s.rsp != 0 && mp!=0 && ms>0)
            if(mp[0].s.rsp != 0)
```

```
                for(ssp = mp[0].s.rsp;
                     ssp < mp[0].e.rep; ssp++)
                    if(dp < ep)
                        *dp++ = *ssp;
        }else{
            if(dp < ep)
                *dp++ = *sp;
        }
        sp++;
    }
    *dp = '\0';
}
```

10.7 Error handling: `regerror()`

`regerror()`[119] is called when compilation encounters a syntax error in the pattern. The default implementation writes the error message to stderr and exits. Programs that want to recover from bad patterns can replace this function with their own version—it is declared `extern`, not static, precisely for this purpose.

⟨*function* `regerror` 119⟩≡ (583a)

```
void
regerror(char *s)
{
    char buf[132];

    strcpy(buf, "regerror: ");
    strcat(buf, s);
    strcat(buf, "\n");
    write(STDERR, buf, strlen(buf));
    exits("regerr");
}
```

Uses `exits()` 45b, `strcat()` 80b, `strcpy()` 81c, `strlen()` 80a, and `write()` 192b.

Chapter 11

Mathematics

The math library provides the standard transcendental functions (`sin`, `cos`, `exp`, `log`, `sqrt`, etc.) plus integer arithmetic helpers and number parsing. The integer division and modulus routines (`_div`, `_mod`) are in assembly because the ARM has no hardware divide instruction (see ASSEMBLER book [Pad15a]).

For a broader treatment of numerical algorithms in C—including root finding, interpolation, integration, and random number generation—see Press et al.'s *Numerical Recipes* [PTVF07], which covers much of the same ground as this chapter but in far greater depth.

⟨*signatures basic and rounding functions* 120a⟩≡ (368b)

```
extern  int     abs(int);
extern  double  fabs(double);

extern  double  floor(double);
extern  double  ceil(double);
extern  double  fmod(double, double);
```

⟨*signatures special float functions* 120b⟩≡ (368b)

```
extern  double  NaN(void);
extern  double  Inf(int);

extern  int     isNaN(double);
extern  int     isInf(double, int);
```

⟨*signatures exponential and powers functions* 120c⟩≡ (368b)

```
extern  double  exp(double);
extern  double  log(double);
extern  double  log10(double);
extern  double  pow(double, double);
extern  double  pow10(int);
extern  double  sqrt(double);
```

⟨*signatures trigonometric functions* 120d⟩≡ (368b)

```
extern  double  sin(double);
extern  double  cos(double);
extern  double  tan(double);

extern  double  asin(double);
extern  double  acos(double);
extern  double  atan(double);
extern  double  atan2(double, double);

extern  double  sinh(double);
extern  double  cosh(double);
extern  double  tanh(double);
```

⟨*signatures number parsing functions* 121a⟩≡ (368b)
```
extern  int     atoi(char*);
extern  double  atof(char*);
extern  long    atol(char*);

extern  double  strtod(char*, char**);
extern  long    strtol(char*, char**, int);
```

⟨*signatures* `xxxrand` *functions* 121b⟩≡ (368b)
```
extern  int     rand(void);
extern  int     nrand(int);

extern  void    srand(long);
```

Most of these are standard C, but a few are Plan 9 additions: `NaN()`/`Inf(sign)` construct special IEEE 754 values (standard C uses macros instead), `pow10()` is a fast integer-exponent variant, and `nrand(n)` returns a uniform random integer in $[0, n)$—more convenient than the `rand() % n` idiom which has modulo bias.

11.1 Floating-point principles

Every line of code in this chapter assumes one particular answer to the question of how a real number is represented in a computer: the IEEE 754 standard for binary floating-point arithmetic, published in 1985 and revised in 2008 and 2019. Its central idea is to store a number in scientific notation in base two: a sign bit, an *exponent* field, and a *mantissa* (or *significand*) field, so that the value is $\pm\,\text{mantissa} \times 2^{\text{exponent}}$. A 64-bit `double`—the format this chapter's functions operate on exclusively—uses 1 bit of sign, 11 bits of exponent, and 52 bits of mantissa, giving about 15 decimal digits of precision. The exponent is stored *biased* (add 1023) so that bit-level comparison of two positive floats matches numeric comparison, and the mantissa's leading 1 bit is implicit (*normalized form*) so that one bit of precision is recovered for free.

The standard devotes four of its exponent encodings to *special values* that do not represent ordinary numbers but are essential to a clean numerical system. *Signed zero* ($+0$ and -0) distinguishes "approached zero from above" from "approached zero from below," so that $1/(+0) = +\infty$ and $1/(-0) = -\infty$ give the right sign. *Signed infinity* is the result of operations that overflow the exponent range, letting the computation continue instead of trapping. *NaN* ("Not a Number") is the result of operations that have no meaningful real answer ($0/0$, $\sqrt{-1}$, $\infty - \infty$) and propagates through every subsequent operation—a kind of sticky error flag at the bit level—which is why Plan 9 provides the `isNaN` predicate to detect it and why ordinary comparisons like `x == x` can return false. *Subnormal* (or *denormal*) numbers live at the bottom of the exponent range and trade precision for range, preventing a sudden "hole" where $x \neq y$ but $x - y$ underflows to zero—a property called *gradual underflow*. All four categories are why the `NaN()` and `Inf()` constructors exist in Plan 9's `libc` in the first place.

The hard problem that occupies most of this chapter is computing *transcendental* functions—`sin`, `cos`, `exp`, `log`, `sqrt`, `pow`, and so on. None has a finite closed form in the basic arithmetic operations, so every implementation must *approximate*. Three steps appear in essentially every transcendental routine. First, *argument reduction*: use an identity to translate the input into a small interval where an approximation is accurate—$\sin x = \sin(x \bmod 2\pi)$, $\exp x = 2^n \cdot \exp(r)$ with $n = \lfloor x \log_2 e \rfloor$, and so on. Second, *polynomial* (or rational) approximation of the function on that small interval, with coefficients chosen offline to minimize error. Third, *reconstruction*: undo the reduction to recover the full-range answer. Each step introduces rounding error, and keeping the total error within one "unit in the last place" (ulp) of the true answer is the numerical-analyst job that determines the coefficients in step two. In extreme cases, correctly rounding a transcendental to the nearest `double` requires evaluating the function to hundreds of bits of precision—a problem known as the *table-maker's dilemma*—which is why truly "correctly-rounded" libms like GNU's CRLIBM and the more recent CORE-MATH project are much larger and slower than what ships with most C libraries.

Plan 9's math library takes the traditional answer: for each transcendental, pick an argument reduction, then evaluate a *minimax rational approximation*—a ratio of two polynomials whose coefficients have been chosen by

a Remez-style optimization to minimize the *worst-case* error over the reduced interval, rather than minimizing the average error the way a Taylor series would. The coefficients all come from Hart et al.'s 1968 book *Computer Approximations*, which tabulates thousands of such rational approximations and is why the polynomial coefficients in this chapter look like magic numbers—they were computed by optimization, not derived by hand. Each transcendental lives in its own short file (`sin.c`, `exp.c`, `log.c`, ...), each file contains one argument reduction followed by one polynomial evaluation, and the whole library is a few hundred lines. The price Plan 9 pays for this simplicity is accuracy at the edges of the input range: it aims for "close enough for most uses," not the correctly-rounded-last-bit guarantees of CRLIBM or CORE-MATH, and not the handcrafted table-lookup tricks of Sun's `fdlibm` (the ancestor of glibc's and MUSL's `libm`). The rest of this chapter works bottom-up: basic helpers (`abs`, `fabs`, `floor`, `ceil`) first, then `sqrt`, then the exponential family (`exp`, `log`, `pow`), then the trigonometric family (`sin`, `cos`, `tan` and their inverses), and finally number parsing and random numbers.

11.2 Basics

The basic math functions are straightforward but note that `abs()` and `fabs()` have the same C source code yet produce very different assembly: `abs()` uses integer comparison and negation, while `fabs()` uses floating-point instructions.

⟨*function* `abs` 122a⟩≡ (387c)

```
int
abs(int a)
{
    if(a < 0)
        return -a;
    return a;
}
```

⟨*function* `fabs` 122b⟩≡ (395d)

```
double
fabs(double arg)
{

    if(arg < 0)
        return -arg;
    return arg;
}
```

`floor()` and `ceil()` use `modf()`[137b] to split a number into integer and fractional parts, then adjust for negative numbers.

⟨*function* `floor` 122c⟩≡ (395e)

```
/*
 * floor and ceil-- greatest integer <= arg
 * (resp least >=)
 */
double
floor(double d)
{
    double fract;

    if(d < 0) {
        fract = modf(-d, &d);
        if(fract != 0.0)
            d += 1;
        d = -d;
    } else
        modf(d, &d);
```

```
    return d;
}
```

Uses modf() 137b.

⟨*function* ceil 123a⟩≡ (395e)

```
double
ceil(double d)
{
    return -floor(-d);
}
```

Uses floor() 122c.

11.3 Integers

11.3.1 Parsing

The "ato" prefix stands for "ASCII to": **atoi** converts a string to **int**, **atol** to **long**. **atol** handles three notations: decimal, octal (leading 0), and hexadecimal (leading 0x). The more flexible **strtol** adds support for arbitrary bases up to 36, returns a pointer past the parsed number (so the caller can continue parsing), and detects overflow.

⟨*function* atoi 123b⟩≡ (389c)

```
int
atoi(char *s)
{

    return atol(s);
}
```

Uses atol() 123c.

⟨*function* atol 123c⟩≡ (389c)

```
long
atol(char *s)
{
    long n;
    int f, c;

    n = 0;
    f = 0;
    while(*s == ' ' || *s == '\t')
        s++;
    if(*s == '-' || *s == '+') {
        if(*s++ == '-')
            f = 1;
        while(*s == ' ' || *s == '\t')
            s++;
    }
    if(s[0]=='0' && s[1]) {
        if(s[1]=='x' || s[1]=='X'){
            s += 2;
            for(;;) {
                c = *s;
                if(c >= '0' && c <= '9')
                    n = n*16 + c - '0';
                else
                if(c >= 'a' && c <= 'f')
                    n = n*16 + c - 'a' + 10;
```

```
                else
                if(c >= 'A' && c <= 'F')
                    n = n*16 + c - 'A' + 10;
                else
                    break;
                s++;
            }
        } else
            while(*s >= '0' && *s <= '7')
                n = n*8 + *s++ - '0';
    } else
        while(*s >= '0' && *s <= '9')
            n = n*10 + *s++ - '0';
    if(f)
        n = -n;
    return n;
}
```

⟨*constant* LONG_MAX 124a⟩≡ (446d)

```
#define LONG_MAX    2147483647L
```

⟨*constant* LONG_MIN 124b⟩≡ (446d)

```
#define LONG_MIN    -2147483648L
```

⟨*function* strtol 124c⟩≡ (446d)

```
long
strtol(char *nptr, char **endptr, int base)
{
    char *p;
    long n, nn, m;
    int c, ovfl, v, neg, ndig;

    p = nptr;
    neg = 0;
    n = 0;
    ndig = 0;
    ovfl = 0;

    /*
     * White space
     */
    for(;; p++) {
        switch(*p) {
        case ' ':
        case '\t':
        case '\n':
        case '\f':
        case '\r':
        case '\v':
            continue;
        }
        break;
    }

    /*
     * Sign
     */
    if(*p=='-' || *p=='+')
        if(*p++ == '-')
            neg = 1;
```

```
    /*
     * Base
     */
    if(base==0) {
        base = 10;
        if(*p == '0') {
            base = 8;
            if(p[1]=='x' || p[1]=='X') {
                p += 2;
                base = 16;
            }
        }
    } else
    if(base==16 && *p=='0'){
        if(p[1]=='x' || p[1]=='X')
            p += 2;
    } else
    if(base<0 || 36<base)
        goto Return;

    /*
     * Non-empty sequence of digits
     */
    m = LONG_MAX/base;
    for(;; p++,ndig++){
        c = *p;
        v = base;
        if('0'<=c && c<='9')
            v = c - '0';
        else
        if('a'<=c && c<='z')
            v = c - 'a' + 10;
        else
        if('A'<=c && c<='Z')
            v = c - 'A' + 10;
        if(v >= base)
            break;
        if(n > m)
            ovfl = 1;
        nn = n*base + v;
        if(nn < n)
            ovfl = 1;
        n = nn;
    }

Return:
    if(ndig == 0)
        p = nptr;
    if(endptr)
        *endptr = p;
    if(ovfl){
        if(neg)
            return LONG_MIN;
        return LONG_MAX;
    }
    if(neg)
        return -n;
    return n;
}
```

Uses LONG_MAX-100 124a and LONG_MIN-101 124b.

11.4 Floats

11.4.1 Special numbers

IEEE 754 floating-point has special values: NaN (not a number), positive/negative infinity, and negative zero. These functions construct and detect them by manipulating the raw bit representation via FPdbleword[27d]. A NaN has all exponent bits set (NANEXP = 2047<<20) and a non-zero mantissa. Infinity has all exponent bits set and a zero mantissa. isNaN exploits the property that NaN is the only value not equal to itself.

⟨*function* NaN 126a⟩≡ (406b)

```
double
NaN(void)
{
    FPdbleword a;

    a.hi = NANEXP;
    a.lo = 1;
    return a.x;
}
```

Uses NANEXP-81 126b.

⟨*constant* NANEXP 126b⟩≡ (406b)

```
#define NANEXP  (2047<<20)
```

⟨*constant* NANMASK 126c⟩≡ (406b)

```
#define NANMASK (2047<<20)
```

⟨*function* isNaN 126d⟩≡ (406b)

```
bool
isNaN(double d)
{
    FPdbleword a;

    a.x = d;
    if((a.hi & NANMASK) != NANEXP)
        return false;
    return !isInf(d, 0);
}
```

Uses NANEXP-81 126b, NANMASK-82 126c, and isInf() 127a.

⟨*constant* NANSIGN 126e⟩≡ (406b)

```
#define NANSIGN (1<<31)
```

⟨*function* Inf 126f⟩≡ (406b)

```
double
Inf(int sign)
{
    FPdbleword a;

    a.hi = NANEXP;
    a.lo = 0;
    if(sign < 0)
        a.hi |= NANSIGN;
    return a.x;
}
```

Uses NANEXP-81 126b and NANSIGN-83 126e.

⟨*function* isInf 127a⟩≡ (406b)
```
bool
isInf(double d, int sign)
{
    FPdbleword a;

    a.x = d;
    if(a.lo != 0)
        return false;
    if(a.hi == NANEXP)
        return sign >= 0;
    if(a.hi == (NANEXP|NANSIGN))
        return sign <= 0;
    return false;
}
```
Uses `NANEXP-81` 126b and `NANSIGN-83` 126e.

⟨*type* FPxxx 127b⟩≡ (368a)
```
/* VFP FCR */
#define FPINEX  (1<<12)         /* trap enables for exceptions */
#define FPUNFL  (1<<11)
#define FPOVFL  (1<<10)
#define FPZDIV  (1<<9)
#define FPINVAL (1<<8)
#define FPRNR   (0<<22)
#define FPRZ    (1<<22)
#define FPRPINF (2<<22)
#define FPRNINF (3<<22)
#define FPRMASK (3<<22)
#define FPPEXT  0
#define FPPSGL  0
#define FPPDBL  0
#define FPPMASK 0
/* FSR */
#define FPAINEX (1<<4)          /* accrued exceptions */
#define FPAUNFL (1<<3)
#define FPAOVFL (1<<2)
#define FPAZDIV (1<<1)
#define FPAINVAL    (1<<0)
```

11.4.2 Parsing

Parsing a floating-point number from text is surprisingly complex. `strtod` uses a state machine to handle signs, digits, decimal points, and exponents, then converts the resulting decimal number to binary using multi-precision integer arithmetic (the `low`/`hig`/`mid` arrays). The algorithm normalizes the decimal to the range $[0.5, 1.0)$, then uses binary search between lower and upper bounds to find the closest IEEE 754 representation. This avoids the rounding errors that plague simpler float-parsing approaches.

⟨*function* atof 127c⟩≡ (389b)
```
double
atof(char *cp)
{
    return strtod(cp, nil);
}
```
Uses `strtod()` 128.

⟨*function* strtod 128⟩≡ (445d)

```
double
strtod(char *as, char **aas)
{
    int na, ona, ex, dp, bp, c, i, flag, state;
    ulong low[Prec], hig[Prec], mid[Prec], num, den;
    double d;
    char *s, a[Ndig];

    flag = 0;   // Fsign, Fesign, Fdpoint
    na = 0;     // number of digits of a[]
    dp = 0;     // na of decimal point
    ex = 0;     // exonent

    state = S0;
    for(s=as;; s++) {
        c = *s;
        if(c >= '0' && c <= '9') {
            switch(state) {
            case S0:
            case S1:
            case S2:
                state = S2;
                break;
            case S3:
            case S4:
                state = S4;
                break;

            case S5:
            case S6:
            case S7:
                state = S7;
                ex = ex*10 + (c-'0');
                continue;
            }
            if(na == 0 && c == '0') {
                dp--;
                continue;
            }
            if(na < Ndig-50)
                a[na++] = c;
            continue;
        }
        switch(c) {
        case '\t':
        case '\n':
        case '\v':
        case '\f':
        case '\r':
        case ' ':
            if(state == S0)
                continue;
            break;
        case '-':
            if(state == S0)
                flag |= Fsign;
            else
                flag |= Fesign;
        case '+':
```

```
        if(state == S0)
            state = S1;
        else
        if(state == S5)
            state = S6;
        else
            break;  // syntax
        continue;
    case '.':
        flag |= Fdpoint;
        dp = na;
        if(state == S0 || state == S1) {
            state = S3;
            continue;
        }
        if(state == S2) {
            state = S4;
            continue;
        }
        break;
    case 'e':
    case 'E':
        if(state == S2 || state == S4) {
            state = S5;
            continue;
        }
        break;
    }
    break;
}

/*
 * clean up return char-pointer
 */
switch(state) {
case S0:
    if(xcmp(s, "nan") == 0) {
        if(aas != nil)
            *aas = s+3;
        goto retnan;
    }
case S1:
    if(xcmp(s, "infinity") == 0) {
        if(aas != nil)
            *aas = s+8;
        goto retinf;
    }
    if(xcmp(s, "inf") == 0) {
        if(aas != nil)
            *aas = s+3;
        goto retinf;
    }
case S3:
    if(aas != nil)
        *aas = as;
    goto ret0;  // no digits found
case S6:
    s--;        // back over +-
case S5:
    s--;        // back over e
```

```
	break;
}
if(aas != nil)
	*aas = s;

if(flag & Fdpoint)
while(na > 0 && a[na-1] == '0')
	na--;
if(na == 0)
	goto ret0;	// zero
a[na] = 0;
if(!(flag & Fdpoint))
	dp = na;
if(flag & Fesign)
	ex = -ex;
dp += ex;
if(dp < -Maxe-Nmant/3)	/* actually -Nmant*log(2)/log(10), but Nmant/3 close enough */
	goto ret0;	// underflow by exp
else
if(dp > +Maxe)
	goto retinf;	// overflow by exp

/*
 * normalize the decimal ascii number
 * to range .[5-9][0-9]* e0
 */
bp = 0;		// binary exponent
while(dp > 0)
	divascii(a, &na, &dp, &bp);
while(dp < 0 || a[0] < '5')
	mulascii(a, &na, &dp, &bp);
a[na] = 0;

/*
 * very small numbers are represented using
 * bp = -Bias+1.  adjust accordingly.
 */
if(bp < -Bias+1){
	ona = na;
	divby(a, &na, -bp-Bias+1);
	if(na < ona){
		memmove(a+ona-na, a, na);
		memset(a, '0', ona-na);
		na = ona;
	}
	a[na] = 0;
	bp = -Bias+1;
}

/* close approx by naive conversion */
num = 0;
den = 1;
for(i=0; i<9 && (c=a[i]); i++) {
	num = num*10 + (c-'0');
	den *= 10;
}
low[0] = umuldiv(num, One, den);
hig[0] = umuldiv(num+1, One, den);
for(i=1; i<Prec; i++) {
	low[i] = 0;
```

```
        hig[i] = One-1;
    }

    /* binary search for closest mantissa */
    for(;;) {
        /* mid = (hig + low) / 2 */
        c = 0;
        for(i=0; i<Prec; i++) {
            mid[i] = hig[i] + low[i];
            if(c)
                mid[i] += One;
            c = mid[i] & 1;
            mid[i] >>= 1;
        }
        frnorm(mid);

        /* compare */
        c = fpcmp(a, mid);
        if(c > 0) {
            c = 1;
            for(i=0; i<Prec; i++)
                if(low[i] != mid[i]) {
                    c = 0;
                    low[i] = mid[i];
                }
            if(c)
                break;  // between mid and hig
            continue;
        }
        if(c < 0) {
            for(i=0; i<Prec; i++)
                hig[i] = mid[i];
            continue;
        }

        /* only hard part is if even/odd roundings wants to go up */
        c = mid[Prec-1] & (Sigbit-1);
        if(c == Sigbit/2 && (mid[Prec-1]&Sigbit) == 0)
            mid[Prec-1] -= c;
        break;  // exactly mid
    }

    /* normal rounding applies */
    c = mid[Prec-1] & (Sigbit-1);
    mid[Prec-1] -= c;
    if(c >= Sigbit/2) {
        mid[Prec-1] += Sigbit;
        frnorm(mid);
    }
    d = 0;
    for(i=0; i<Prec; i++)
        d = d*One + mid[i];
    if(flag & Fsign)
        d = -d;
    d = ldexp(d, bp - Prec*Nbits);
    return d;

ret0:
    return 0;
```

```
retnan:
    return NaN();

retinf:
    if(flag & Fsign)
        return Inf(-1);
    return Inf(+1);
}
```

Uses Bias-203 132a, Fdpoint-212 132a, Fesign-211 132a, Fsign-210 132a, Inf() 126f, Maxe-209 132a, NaN() 126a, Nbits-201 132a, Ndig-206 132a, Nmant-202 132a, One-207 132a, Prec-204 132a, S0-213 132a, S1-214 132a, S2-215 132a, S3-216 132a, S4-217 132a, S5-218 132a, S6-219 132a, S7-220 132a, Sigbit-205 132a, divascii() 135a, divby() 134a, fpcmp() 133b, frnorm() 133a, ldexp() 150i, memmove() 48, memset() 46b, mulascii() 136a, umuldiv() 155b, and xcmp() 136b.

⟨*enum* _anon_ (port/strtod.c) 132a⟩≡ (445d)

```
/*
 * This routine will convert to arbitrary precision
 * floating point entirely in multi-precision fixed.
 * The answer is the closest floating point number to
 * the given decimal number. Exactly half way are
 * rounded ala ieee rules.
 * Method is to scale input decimal between .500 and .999...
 * with external power of 2, then binary search for the
 * closest mantissa to this decimal number.
 * Nmant is is the required precision. (53 for ieee dp)
 * Nbits is the max number of bits/word. (must be <= 28)
 * Prec is calculated - the number of words of fixed mantissa.
 */
enum
{
    Nbits   = 28,               // bits safely represented in a ulong
    Nmant   = 53,               // bits of precision required
    Bias        = 1022,
    Prec    = (Nmant+Nbits+1)/Nbits,    // words of Nbits each to represent mantissa
    Sigbit  = 1<<(Prec*Nbits-Nmant),    // first significant bit of Prec-th word
    Ndig    = 1500,
    One = (ulong)(1<<Nbits),
    Half    = (ulong)(One>>1),
    Maxe    = 310,
    Fsign   = 1<<0,     // found -
    Fesign  = 1<<1,     // found e-
    Fdpoint = 1<<2,     // found .

    S0  = 0,        // _        _S0 +S1 #S2 .S3
    S1,         // _+       #S2 .S3
    S2,         // _+#      #S2 .S4 eS5
    S3,         // _+.      #S4
    S4,         // _+#.#    #S4 eS5
    S5,         // _+#.#e   +S6 #S7
    S6,         // _+#.#e+  #S7
    S7,         // _+#.#e+# #S7
};
```

Uses Nbits-201 132a, Nmant-202 132a, One-207 132a, and Prec-204 132a.

⟨*struct* Tab 132b⟩≡ (445d)

```
struct  Tab
{
    int bp;
    int siz;
    char*   cmp;
};
```

⟨*function* `frnorm` 133a⟩≡ (445d)

```
static void
frnorm(ulong *f)
{
    int i, c;

    c = 0;
    for(i=Prec-1; i>0; i--) {
        f[i] += c;
        c = f[i] >> Nbits;
        f[i] &= One-1;
    }
    f[0] += c;
}
```

Uses `Nbits-201` 132a, `One-207` 132a, and `Prec-204` 132a.

⟨*function* `fpcmp` 133b⟩≡ (445d)

```
static int
fpcmp(char *a, ulong* f)
{
    ulong tf[Prec];
    int i, d, c;

    for(i=0; i<Prec; i++)
        tf[i] = f[i];

    for(;;) {
        /* tf *= 10 */
        for(i=0; i<Prec; i++)
            tf[i] = tf[i]*10;
        frnorm(tf);
        d = (tf[0] >> Nbits) + '0';
        tf[0] &= One-1;

        /* compare next digit */
        c = *a;
        if(c == 0) {
            if('0' < d)
                return -1;
            if(tf[0] != 0)
                goto cont;
            for(i=1; i<Prec; i++)
                if(tf[i] != 0)
                    goto cont;
            return 0;
        }
        if(c > d)
            return +1;
        if(c < d)
            return -1;
        a++;
    cont:;
    }
}
```

Uses `Nbits-201` 132a, `One-207` 132a, `Prec-204` 132a, and `frnorm()` 133a.

⟨*function* `_divby` 133c⟩≡ (445d)

```
static void
_divby(char *a, int *na, int b)
{
```

```
    int n, c;
    char *p;

    p = a;
    n = 0;
    while(n>>b == 0) {
        c = *a++;
        if(c == 0) {
            while(n) {
                c = n*10;
                if(c>>b)
                    break;
                n = c;
            }
            goto xx;
        }
        n = n*10 + c-'0';
        (*na)--;
    }
    for(;;) {
        c = n>>b;
        n -= c<<b;
        *p++ = c + '0';
        c = *a++;
        if(c == 0)
            break;
        n = n*10 + c-'0';
    }
    (*na)++;
xx:
    while(n) {
        n = n*10;
        c = n>>b;
        n -= c<<b;
        *p++ = c + '0';
        (*na)++;
    }
    *p = 0;
}
```

⟨*function* `divby` 134a⟩≡ (445d)

```
static void
divby(char *a, int *na, int b)
{
    while(b > 9){
        _divby(a, na, 9);
        a[*na] = 0;
        b -= 9;
    }
    if(b > 0)
        _divby(a, na, b);
}
```

Uses `_divby()` 133c.

⟨*global* `tab1` 134b⟩≡ (445d)

```
static  Tab tab1[] =
{
     1,  0, "",
     3,  1, "7",
     6,  2, "63",
```

```
     9,  3, "511",
    13,  4, "8191",
    16,  5, "65535",
    19,  6, "524287",
    23,  7, "8388607",
    26,  8, "67108863",
    27,  9, "134217727",
};
```

⟨*function* divascii 135a⟩≡ (445d)

```
static void
divascii(char *a, int *na, int *dp, int *bp)
{
    int b, d;
    Tab *t;

    d = *dp;
    if(d >= nelem(tab1))
        d = nelem(tab1)-1;
    t = tab1 + d;
    b = t->bp;
    if(memcmp(a, t->cmp, t->siz) > 0)
        d--;
    *dp -= d;
    *bp += b;
    divby(a, na, b);
}
```

Uses `divby()` 134a, `memcmp()` 53, and `tab1-221` 134b.

⟨*function* mulby 135b⟩≡ (445d)

```
static void
mulby(char *a, char *p, char *q, int b)
{
    int n, c;

    n = 0;
    *p = 0;
    for(;;) {
        q--;
        if(q < a)
            break;
        c = *q - '0';
        c = (c<<b) + n;
        n = c/10;
        c -= n*10;
        p--;
        *p = c + '0';
    }
    while(n) {
        c = n;
        n = c/10;
        c -= n*10;
        p--;
        *p = c + '0';
    }
}
```

⟨*global* tab2 135c⟩≡ (445d)

```
static  Tab tab2[] =
{
```

```
        1,  1, "",              // dp = 0-0
        3,  3, "125",
        6,  5, "15625",
        9,  7, "1953125",
       13, 10, "1220703125",
       16, 12, "152587890625",
       19, 14, "19073486328125",
       23, 17, "11920928955078125",
       26, 19, "1490116119384765625",
       27, 19, "7450580596923828125",      // dp 8-9
  };
```

⟨*function* `mulascii` 136a⟩≡ (445d)

```
  static void
  mulascii(char *a, int *na, int *dp, int *bp)
  {
      char *p;
      int d, b;
      Tab *t;

      d = -*dp;
      if(d >= nelem(tab2))
          d = nelem(tab2)-1;
      t = tab2 + d;
      b = t->bp;
      if(memcmp(a, t->cmp, t->siz) < 0)
          d--;
      p = a + *na;
      *bp -= b;
      *dp += d;
      *na += d;
      mulby(a, p+d, p, b);
  }
```

Uses `memcmp()` 53, `mulby()` 135b, and `tab2-222` 135c.

⟨*function* `xcmp` 136b⟩≡ (445d)

```
  static int
  xcmp(char *a, char *b)
  {
      int c1, c2;

      while(c1 = *b++) {
          c2 = *a++;
          if(isupper(c2))
              c2 = tolower(c2);
          if(c1 != c2)
              return 1;
      }
      return 0;
  }
```

Uses `tolower()` 29d.

11.4.3 `sqrt()`

`sqrt` uses Newton's method (also known as Heron's method): starting from an initial approximation, it iterates `temp = 0.5 * (temp` arg/temp)+ which converges quadratically—each iteration doubles the number of correct

digits, so 5 iterations suffice for double precision. The initial approximation uses `frexp` to extract the mantissa and exponent, then applies half the exponent to get close to the true square root.

⟨*function* sqrt 137a⟩≡ (441a)
```
double
sqrt(double arg)
{
    double x, temp;
    int exp, i;

    if(arg <= 0) {
        if(arg < 0)
            return NaN();
        return 0;
    }
    if(isInf(arg, 1))
        return arg;
    x = frexp(arg, &exp);
    while(x < 0.5) {
        x *= 2;
        exp--;
    }
    /*
     * NOTE
     * this wont work on 1's comp
     */
    if(exp & 1) {
        x *= 2;
        exp--;
    }
    temp = 0.5 * (1.0+x);

    while(exp > 60) {
        temp *= (1L<<30);
        exp -= 60;
    }
    while(exp < -60) {
        temp /= (1L<<30);
        exp += 60;
    }
    if(exp >= 0)
        temp *= 1L << (exp/2);
    else
        temp /= 1L << (-exp/2);
    for(i=0; i<=4; i++)
        temp = 0.5*(temp + arg/temp);
    return temp;
}
```
Uses `NaN()` 126a, `frexp()` 151a, and `isInf()` 127a.

11.4.4 `modf()`

`modf` splits a double into integer and fractional parts by directly masking the IEEE 754 bit representation. The exponent determines how many mantissa bits belong to the integer part: bits above that threshold are kept for `*ip`, and the remainder is the fractional part. This is the function that `floor` and `ceil` rely on.

⟨*function* modf 137b⟩≡ (396e)
```
double
modf(double d, double *ip)
```

```
{
    FPdbleword x;
    int e;

    x.x = d;
    e = (x.hi >> SHIFT) & MASK;
    if(e == MASK){
        *ip = d;
        if(x.lo != 0 || (x.hi & 0xfffffL) != 0) /* NaN */
            return d;
        /* +/- Inf */
        x.hi &= 0x80000000L;
        return x.x;
    }
    if(d < 1) {
        if(d < 0) {
            x.x = modf(-d, ip);
            *ip = -*ip;
            return -x.x;
        }
        *ip = 0;
        return d;
    }
    e -= BIAS;
    if(e <= SHIFT+1) {
        x.hi &= ~(0x1fffffL >> e);
        x.lo = 0;
    } else
    if(e <= SHIFT+33)
        x.lo &= ~(0x7fffffffL >> (e-SHIFT-2));
    *ip = x.x;
    return d - x.x;
}
```

Uses BIAS-42 150g, MASK-40 150e, SHIFT-41 150f, and modf() 137b.

11.4.5 fmod()

⟨*function* fmod 138⟩≡ (395f)

```
/*
 * floating-point mod function without infinity or NaN checking
 */
double
fmod (double x, double y)
{
    int sign, yexp, rexp;
    double r, yfr, rfr;

    if (y == 0)
        return x;
    if (y < 0)
        y = -y;
    yfr = frexp(y, &yexp);
    sign = 0;
    if(x < 0) {
        r = -x;
        sign++;
    } else
        r = x;
    while(r >= y) {
```

```
        rfr = frexp(r, &rexp);
        r -= ldexp(y, rexp - yexp - (rfr < yfr));
    }
    if(sign)
        r = -r;
    return r;
}
```

Uses frexp() 151a and ldexp() 150i.

11.4.6 pow()

pow handles x^y by splitting y into integer and fractional parts. The fractional part is computed as $e^{y_1 \ln x}$ (using exp and log), while the integer part is computed by repeated squaring—much faster than calling exp/log for the full exponent. Special cases include negative bases (only valid for integer exponents), zero, and overflow detection.

⟨*function* pow 139⟩≡ (431)

```
double
pow(double x, double y) /* return x ^ y (exponentiation) */
{
    double xy, y1, ye;
    long i;
    int ex, ey, flip;

    if(y == 0.0)
        return 1.0;

    flip = 0;
    if(y < 0.){
        y = -y;
        flip = 1;
    }
    y1 = modf(y, &ye);
    if(y1 != 0.0){
        if(x <= 0.)
            goto zreturn;
        if(y1 > 0.5) {
            y1 -= 1.;
            ye += 1.;
        }
        xy = exp(y1 * log(x));
    }else
        xy = 1.0;
    if(ye > 0x7FFFFFFF){    /* should be ~0UL but compiler can't convert double to ulong */
        if(x <= 0.){
 zreturn:
            if(x==0. && !flip)
                return 0.;
            return NaN();
        }
        if(flip){
            if(y == .5)
                return 1./sqrt(x);
            y = -y;
        }else if(y == .5)
            return sqrt(x);
        return exp(y * log(x));
    }
```

```
    x = frexp(x, &ex);
    ey = 0;
    i = ye;
    if(i)
        for(;;){
            if(i & 1){
                xy *= x;
                ey += ex;
            }
            i >>= 1;
            if(i == 0)
                break;
            x *= x;
            ex <<= 1;
            if(x < .5){
                x += x;
                ex -= 1;
            }
        }
    if(flip){
        xy = 1. / xy;
        ey = -ey;
    }
    return ldexp(xy, ey);
}
```

Uses `NaN()` 126a, `exp()` 149a, `frexp()` 151a, `ldexp()` 150i, `log()` 151b, `modf()` 137b, and `sqrt()` 137a.

⟨*function* `pow10` 140a⟩≡ (432a)

```
double
pow10(int n)
{
    int m;

    if(n < 0) {
        n = -n;
        if(n < sizeof(tab)/sizeof(tab[0]))
            return 1/tab[n];
        m = n/2;
        return 1/(pow10(m) * pow10(n-m));
    }
    if(n < sizeof(tab)/sizeof(tab[0]))
        return tab[n];
    m = n/2;
    return pow10(m) * pow10(n-m);
}
```

Uses `pow10()` 140a and `tab-157` 140b.

⟨*global* `tab` 140b⟩≡ (432a)

```
/*
 * this table might overflow 127-bit exponent representations.
 * in that case, truncate it after 1.0e38.
 * it is important to get all one can from this
 * routine since it is used in atof to scale numbers.
 * the presumption is that C converts fp numbers better
 * than multipication of lower powers of 10.
 */
static
double  tab[] =
{
    1.0e0,  1.0e1,  1.0e2,  1.0e3,  1.0e4,  1.0e5,  1.0e6,  1.0e7,  1.0e8,  1.0e9,
```

```
    1.0e10, 1.0e11, 1.0e12, 1.0e13, 1.0e14, 1.0e15, 1.0e16, 1.0e17, 1.0e18, 1.0e19,
    1.0e20, 1.0e21, 1.0e22, 1.0e23, 1.0e24, 1.0e25, 1.0e26, 1.0e27, 1.0e28, 1.0e29,
    1.0e30, 1.0e31, 1.0e32, 1.0e33, 1.0e34, 1.0e35, 1.0e36, 1.0e37, 1.0e38, 1.0e39,
    1.0e40, 1.0e41, 1.0e42, 1.0e43, 1.0e44, 1.0e45, 1.0e46, 1.0e47, 1.0e48, 1.0e49,
    1.0e50, 1.0e51, 1.0e52, 1.0e53, 1.0e54, 1.0e55, 1.0e56, 1.0e57, 1.0e58, 1.0e59,
    1.0e60, 1.0e61, 1.0e62, 1.0e63, 1.0e64, 1.0e65, 1.0e66, 1.0e67, 1.0e68, 1.0e69,
    1.0e70, 1.0e71, 1.0e72, 1.0e73, 1.0e74, 1.0e75, 1.0e76, 1.0e77, 1.0e78, 1.0e79,
    1.0e80, 1.0e81, 1.0e82, 1.0e83, 1.0e84, 1.0e85, 1.0e86, 1.0e87, 1.0e88, 1.0e89,
    1.0e90, 1.0e91, 1.0e92, 1.0e93, 1.0e94, 1.0e95, 1.0e96, 1.0e97, 1.0e98, 1.0e99,
    1.0e100,1.0e101,1.0e102,1.0e103,1.0e104,1.0e105,1.0e106,1.0e107,1.0e108,1.0e109,
    1.0e110,1.0e111,1.0e112,1.0e113,1.0e114,1.0e115,1.0e116,1.0e117,1.0e118,1.0e119,
    1.0e120,1.0e121,1.0e122,1.0e123,1.0e124,1.0e125,1.0e126,1.0e127,1.0e128,1.0e129,
    1.0e130,1.0e131,1.0e132,1.0e133,1.0e134,1.0e135,1.0e136,1.0e137,1.0e138,1.0e139,
    1.0e140,1.0e141,1.0e142,1.0e143,1.0e144,1.0e145,1.0e146,1.0e147,1.0e148,1.0e149,
    1.0e150,1.0e151,1.0e152,1.0e153,1.0e154,1.0e155,1.0e156,1.0e157,1.0e158,1.0e159,
};
```

11.5 Trigonometry

The trigonometric functions (**sin**, **cos**, **tan**, **atan**, **atan2**) use polynomial approximations from Hart et al.'s tables. The shared **sinus** helper first reduces the argument to the range $[0, \pi/4]$ using the periodicity of sin/cos, then evaluates a rational polynomial $p(x)/q(x)$. The coefficients (p0–p4, q0–q3) are chosen to give double-precision accuracy over this range.

11.5.1 Direct functions

⟨*function* `cos` 141a⟩≡ (440c)

```
double
cos(double arg)
{
    if(arg < 0)
        arg = -arg;
    return sinus(arg, 1);
}
```

Uses `sinus()` 142f.

⟨*function* `sin` 141b⟩≡ (440c)

```
double
sin(double arg)
{
    return sinus(arg, 0);
}
```

Uses `sinus()` 142f.

⟨*constant p0 (port/sin.c)* 141c⟩≡ (440c)

```
#define p0      .1357884097877375669092680e8
```

⟨*constant p1 (port/sin.c)* 141d⟩≡ (440c)

```
#define p1     -.4942908100902844161158627e7
```

⟨*constant p2 (port/sin.c)* 141e⟩≡ (440c)

```
#define p2      .4401030535375266501944918e6
```

⟨*constant p3 (port/sin.c)* 141f⟩≡ (440c)

```
#define p3     -.1384727249982452873054457e5
```

⟨*constant p4 (port/sin.c)* 142a⟩≡ (440c)
```
#define p4      .145968840666576872222695963
```

⟨*constant q0 (port/sin.c)* 142b⟩≡ (440c)
```
#define q0      .8644558652922534429915149e7
```

⟨*constant q1 (port/sin.c)* 142c⟩≡ (440c)
```
#define q1      .408179225234329974939577 9e6
```

⟨*constant q2 (port/sin.c)* 142d⟩≡ (440c)
```
#define q2      .9463096101538208180571257e4
```

⟨*constant q3 (port/sin.c)* 142e⟩≡ (440c)
```
#define q3      .1326534908786136358911494e3
```

⟨*function* sinus 142f⟩≡ (440c)
```
static
double
sinus(double arg, int quad)
{
    double e, f, ysq, x, y, temp1, temp2;
    int k;

    x = arg;
    if(x < 0) {
        x = -x;
        quad += 2;
    }
    x *= 1/PIO2;    /* underflow? */
    if(x > 32764) {
        y = modf(x, &e);
        e += quad;
        modf(0.25*e, &f);
        quad = e - 4*f;
    } else {
        k = x;
        y = x - k;
        quad += k;
        quad &= 3;
    }
    if(quad & 1)
        y = 1-y;
    if(quad > 1)
        y = -y;

    ysq = y*y;
    temp1 = ((((p4*ysq+p3)*ysq+p2)*ysq+p1)*ysq+p0)*y;
    temp2 = ((((ysq+q3)*ysq+q2)*ysq+q1)*ysq+q0);
    return temp1/temp2;
}
```
Uses modf() 137b, p0-87 141c, p1-88 141d, p2-89 141e, p3-90 141f, p4-91 142a, q0-92 142b, q1-93 142c, q2-94 142d, and q3-95 142e.

⟨*global p0 (port/tan.c)* 142g⟩≡ (452b)
```
static double p0     = -0.1306820264754825668269611177e+5;
```
Uses p0-149 142g.

⟨*global p1 (port/tan.c)* 142h⟩≡ (452b)
```
static double p1      = 0.1055970901714953193602353981e+4;
```
Uses p1-150 142h.

⟨*global p2 (port/tan.c)* 143a⟩≡ (452b)

```
static double p2     = -0.15506856534832663769417057 28e+2;
```

Uses p2-151 143a.

⟨*global p3 (port/tan.c)* 143b⟩≡ (452b)

```
static double p3      = 0.34225543872410034353284704 89e-1;
```

Uses p3-152 143b.

⟨*global* p4 143c⟩≡ (452b)

```
static double p4      = 0.338663864267717209607636 9e-4;
```

Uses p4-153 143c.

⟨*global q0 (port/tan.c)* 143d⟩≡ (452b)

```
static double q0     = -0.16638952389471190018514646 61e+5;
```

Uses q0-154 143d.

⟨*global q1 (port/tan.c)* 143e⟩≡ (452b)

```
static double q1      = 0.47657513629164836989266555 81e+4;
```

Uses q1-155 143e.

⟨*global q2 (port/tan.c)* 143f⟩≡ (452b)

```
static double q2     = -0.15550331640317099669001245 74e+3;
```

Uses q2-156 143f.

⟨*function* tan 143g⟩≡ (452b)

```
double
tan(double arg)
{
    double temp, e, x, xsq;
    int flag, sign, i;

    flag = 0;
    sign = 0;
    if(arg < 0){
        arg = -arg;
        sign++;
    }
    arg = 2*arg/PIO2;   /* overflow? */
    x = modf(arg, &e);
    i = e;
    switch(i%4) {
    case 1:
        x = 1 - x;
        flag = 1;
        break;

    case 2:
        sign = !sign;
        flag = 1;
        break;

    case 3:
        x = 1 - x;
        sign = !sign;
        break;

    case 0:
        break;
    }
```

```
    xsq = x*x;
    temp = ((((p4*xsq+p3)*xsq+p2)*xsq+p1)*xsq+p0)*x;
    temp = temp/(((xsq+q2)*xsq+q1)*xsq+q0);

    if(flag) {
        if(temp == 0)
            return NaN();
        temp = 1/temp;
    }
    if(sign)
        temp = -temp;
    return temp;
}
```

Uses `NaN()` 126a, `modf()` 137b, `p0-149` 142g, `p1-150` 142h, `p2-151` 143a, `p3-152` 143b, `p4-153` 143c, `q0-154` 143d, `q1-155` 143e, and `q2-156` 143f.

11.5.2 Inverse functions

⟨*function* `asin` 144a⟩≡ (387d)

```
double
asin(double arg)
{
    double temp;
    int sign;

    sign = 0;
    if(arg < 0) {
        arg = -arg;
        sign++;
    }
    if(arg > 1)
        return NaN();
    temp = sqrt(1 - arg*arg);
    if(arg > 0.7)
        temp = PIO2 - atan(temp/arg);
    else
        temp = atan(arg/temp);
    if(sign)
        temp = -temp;
    return temp;
}
```

Uses `NaN()` 126a, `atan()` 144c, and `sqrt()` 137a.

⟨*function* `acos` 144b⟩≡ (387d)

```
double
acos(double arg)
{
    if(arg > 1 || arg < -1)
        return NaN();
    return PIO2 - asin(arg);
}
```

Uses `NaN()` 126a and `asin()` 144a.

⟨*function* `atan` 144c⟩≡ (388a)

```
/*
    atan makes its argument positive and
    calls the inner routine satan.
```

```
  */

 double
 atan(double arg)
 {

     if(arg > 0)
         return satan(arg);
     return -satan(-arg);
 }
```
Uses `satan()` 146b.

⟨*constant* `sq2p1` 145a⟩≡ (388a)
```
 #define sq2p1 2.414213562373095048802e0
```

⟨*constant* `sq2m1` 145b⟩≡ (388a)
```
 #define sq2m1  .414213562373095048802e0
```

⟨*constant* `p4` 145c⟩≡ (388a)
```
 #define p4  .161536412982230228262e2
```

⟨*constant* `p3` 145d⟩≡ (388a)
```
 #define p3  .26842548195503973794141e3
```

⟨*constant* `p2` 145e⟩≡ (388a)
```
 #define p2  .11530293515404850115428136e4
```

⟨*constant* `p1` 145f⟩≡ (388a)
```
 #define p1  .178040631643319697105464587e4
```

⟨*constant* `p0` 145g⟩≡ (388a)
```
 #define p0  .89678597403663861959987488e3
```

⟨*constant* `q4` 145h⟩≡ (388a)
```
 #define q4  .5895697050844462222791e2
```

⟨*constant* `q3` 145i⟩≡ (388a)
```
 #define q3  .536265374031215315104235e3
```

⟨*constant* `q2` 145j⟩≡ (388a)
```
 #define q2  .16667838148816337184521798e4
```

⟨*constant* `q1` 145k⟩≡ (388a)
```
 #define q1  .207933497444540981287275926e4
```

⟨*constant* `q0` 145l⟩≡ (388a)
```
 #define q0  .89678597403663861962481162e3
```

⟨*function* xatan 146a⟩≡ (388a)

```
/*
    xatan evaluates a series valid in the
    range [-0.414...,+0.414...]. (tan(pi/8))
 */

static
double
xatan(double arg)
{
    double argsq, value;

    argsq = arg*arg;
    value = ((((p4*argsq + p3)*argsq + p2)*argsq + p1)*argsq + p0);
    value = value/(((((argsq + q4)*argsq + q3)*argsq + q2)*argsq + q1)*argsq + q0);
    return value*arg;
}
```

Uses `p0-143` 145g, `p1-142` 145f, `p2-141` 145e, `p3-140` 145d, `p4-139` 145c, `q0-148` 145l, `q1-147` 145k, `q2-146` 145j, `q3-145` 145i, and `q4-144` 145h.

⟨*function* satan 146b⟩≡ (388a)

```
/*
    satan reduces its argument (known to be positive)
    to the range [0,0.414...] and calls xatan.
 */

static
double
satan(double arg)
{

    if(arg < sq2m1)
        return xatan(arg);
    if(arg > sq2p1)
        return PIO2 - xatan(1/arg);
    return PIO2/2 + xatan((arg-1)/(arg+1));
}
```

Uses `sq2m1-138` 145b, `sq2p1-137` 145a, and `xatan()` 146a.

⟨*function* atan2 146c⟩≡ (388b)

```
/*
    atan2 discovers what quadrant the angle
    is in and calls atan.
*/

double
atan2(double arg1, double arg2)
{

    if(arg1+arg2 == arg1) {
        if(arg1 >= 0)
            return PIO2;
        return -PIO2;
    }
    arg1 = atan(arg1/arg2);
    if(arg2 < 0) {
        if(arg1 <= 0)
            return arg1 + PI;
        return arg1 - PI;
    }
```

```
    return arg1;
}
```

Uses `atan()` 144c.

11.5.3 Hyperbolic functions

⟨*global* p0 147a⟩≡ (440d)

```
/*
 * sinh(arg) returns the hyperbolic sine of its floating-
 * point argument.
 *
 * The exponential function is called for arguments
 * greater in magnitude than 0.5.
 *
 * A series is used for arguments smaller in magnitude than 0.5.
 * The coefficients are #2029 from Hart & Cheney. (20.36D)
 *
 * cosh(arg) is computed from the exponential function for
 * all arguments.
 */

static  double  p0  = -0.6307673640497716991184787251e+6;
```

Uses `p0-62` 147a.

⟨*global* p1 147b⟩≡ (440d)

```
static  double  p1  = -0.8991272022039509355398013511e+5;
```

Uses `p1-63` 147b.

⟨*global* p2 147c⟩≡ (440d)

```
static  double  p2  = -0.2894211355989563807284660366e+4;
```

Uses `p2-64` 147c.

⟨*global* p3 147d⟩≡ (440d)

```
static  double  p3  = -0.2630563213397497062819489e+2;
```

Uses `p3-65` 147d.

⟨*global* q0 147e⟩≡ (440d)

```
static  double  q0  = -0.6307673640497716991212077277e+6;
```

Uses `q0-66` 147e.

⟨*global* q1 147f⟩≡ (440d)

```
static  double  q1   = 0.1521517378790019070696485176e+5;
```

Uses `q1-67` 147f.

⟨*global* q2 147g⟩≡ (440d)

```
static  double  q2  = -0.173678953558233699533450911e+3;
```

Uses `q2-68` 147g.

⟨*function* sinh 148a⟩≡ (440d)

```
double
sinh(double arg)
{
    double temp, argsq;
    int sign;

    sign = 0;
    if(arg < 0) {
        arg = -arg;
        sign++;
    }
    if(arg > 21) {
        temp = exp(arg)/2;
        goto out;
    }
    if(arg > 0.5) {
        temp = (exp(arg) - exp(-arg))/2;
        goto out;
    }
    argsq = arg*arg;
    temp = (((p3*argsq+p2)*argsq+p1)*argsq+p0)*arg;
    temp /= (((argsq+q2)*argsq+q1)*argsq+q0);
out:
    if(sign)
        temp = -temp;
    return temp;
}
```

Uses `exp()` 149a, `p0-62` 147a, `p1-63` 147b, `p2-64` 147c, `p3-65` 147d, `q0-66` 147e, `q1-67` 147f, and `q2-68` 147g.

⟨*function* cosh 148b⟩≡ (440d)

```
double
cosh(double arg)
{
    if(arg < 0)
        arg = - arg;
    if(arg > 21)
        return exp(arg)/2;
    return (exp(arg) + exp(-arg))/2;
}
```

Uses `exp()` 149a.

⟨*function* tanh 148c⟩≡ (452c)

```
/*
    tanh(arg) computes the hyperbolic tangent of its floating
    point argument.

    sinh and cosh are called except for large arguments, which
    would cause overflow improperly.
 */

double
tanh(double arg)
{

    if(arg < 0) {
        arg = -arg;
        if(arg > 21)
            return -1;
        return -sinh(arg)/cosh(arg);
```

```
    }
    if(arg > 21)
        return 1;
    return sinh(arg)/cosh(arg);
}
```

Uses `cosh()` 148b and `sinh()` 148a.

11.6 Logarithms and exponentials

The `exp()`[149a] and `log()`[151b] functions also use polynomial approximations. `exp()` reduces its argument to the range $[0, \ln 2]$ by extracting the integer part as a power of 2 (applied via `ldexp()`[150i]), then evaluates a rational polynomial for the remainder. `log()` does the reverse: it uses `frexp()`[151a] to extract the exponent, then approximates $\ln(x)$ for $x \in [0.5, 1.0)$. `ldexp()` and `frexp()` manipulate the IEEE 754 exponent field directly for exact, fast multiplication/division by powers of 2.

11.6.1 exp()

⟨*function* `exp` 149a⟩≡ (395c)

```
double
exp(double arg)
{
    double fract, temp1, temp2, xsq;
    int ent;

    if(arg == 0)
        return 1;
    if(arg < -maxf)
        return 0;
    if(arg > maxf)
        return Inf(1);

    arg *= log2e;
    ent = floor(arg);
    fract = (arg-ent) - 0.5;
    xsq = fract*fract;
    temp1 = ((p2*xsq+p1)*xsq+p0)*fract;
    temp2 = ((xsq+q2)*xsq+q1)*xsq + q0;
    return ldexp(sqrt2*(temp2+temp1)/(temp2-temp1), ent);
}
```

Uses `Inf()` 126f, `floor()` 122c, `ldexp()` 150i, `log2e-50` 149c, `maxf-52` 149b, `p0-44` 149d, `p1-45` 149e, `p2-46` 149f, `q0-47` 150a, `q1-48` 150b, `q2-49` 150c, and `sqrt2-51` 150d.

⟨*constant* `maxf` 149b⟩≡ (395c)

```
#define maxf  10000
```

⟨*constant* `log2e` 149c⟩≡ (395c)

```
#define log2e  1.4426950408889634073599247
```

⟨*constant p0 (port/exp.c)* 149d⟩≡ (395c)

```
#define p0  .2080384346694663001443843411e7
```

⟨*constant p1 (port/exp.c)* 149e⟩≡ (395c)

```
#define p1  .3028697169744036299076048876e5
```

⟨*constant p2 (port/exp.c)* 149f⟩≡ (395c)

```
#define p2  .6061485330061080841615584556e2
```

⟨*constant q0 (port/exp.c)* 150a⟩≡ (395c)
```
#define q0  .6002720360238832528230907598e7
```

⟨*constant q1 (port/exp.c)* 150b⟩≡ (395c)
```
#define q1  .3277251518082914423057964422e6
```

⟨*constant q2 (port/exp.c)* 150c⟩≡ (395c)
```
#define q2  .1749287689093076403844945335e4
```

⟨*constant* `sqrt2` 150d⟩≡ (395c)
```
#define sqrt2  1.4142135623730950488016887
```

11.6.2 `ldexp()`

`ldexp()` multiplies a double by 2^n by directly adjusting the IEEE 754 exponent field. It handles edge cases (zero, NaN, infinity, denormals, overflow, underflow) and works in a loop when `n` is too large for a single exponent adjustment. `frexp()`[151a] does the reverse: it extracts the exponent and returns the mantissa in $[0.5, 1.0)$. Both access the bit representation through `FPdbleword`.

⟨*constant MASK (port/frexp.c)* 150e⟩≡ (396e)
```
/*
 * this is big/little endian non-portable
 * it gets the endian from the FPdbleword
 * union in u.h.
 */
#define MASK    0x7ffL
```

⟨*constant* `SHIFT` 150f⟩≡ (396e)
```
#define SHIFT   20
```

⟨*constant* `BIAS` 150g⟩≡ (396e)
```
#define BIAS    1022L
```

⟨*constant* `SIG` 150h⟩≡ (396e)
```
#define SIG 52
```

⟨*function* `ldexp` 150i⟩≡ (396e)
```
double
ldexp(double d, int deltae)
{
    int e, bits;
    FPdbleword x;
    ulong z;

    if(d == 0)
        return 0;
    x.x = d;
    e = (x.hi >> SHIFT) & MASK;
    if(deltae >= 0 || e+deltae >= 1){   /* no underflow */
        e += deltae;
        if(e >= MASK){      /* overflow */
            if(d < 0)
                return Inf(-1);
            return Inf(1);
        }
    }else{  /* underflow gracefully */
        deltae = -deltae;
        /* need to shift d right deltae */
        if(e > 1){      /* shift e-1 by exponent manipulation */
```

```
            deltae -= e-1;
            e = 1;
        }
        if(deltae > 0 && e==1){ /* shift 1 by switch from 1.fff to 0.1ff */
            deltae--;
            e = 0;
            x.lo >>= 1;
            x.lo |= (x.hi&1)<<31;
            z = x.hi & ((1<<SHIFT)-1);
            x.hi &= ~((1<<SHIFT)-1);
            x.hi |= (1<<(SHIFT-1)) | (z>>1);
        }
        while(deltae > 0){      /* shift bits down */
            bits = deltae;
            if(bits > SHIFT)
                bits = SHIFT;
            x.lo >>= bits;
            x.lo |= (x.hi&((1<<bits)-1)) << (32-bits);
            z = x.hi & ((1<<SHIFT)-1);
            x.hi &= ~((1<<SHIFT)-1);
            x.hi |= z>>bits;
            deltae -= bits;
        }
    }
    x.hi &= ~(MASK << SHIFT);
    x.hi |= (long)e << SHIFT;
    return x.x;
}
```

Uses `Inf()` 126f, `MASK-40` 150e, and `SHIFT-41` 150f.

11.6.3 frexp()

⟨*function* `frexp` 151a⟩≡ (396e)

```
double
frexp(double d, int *ep)
{
    FPdbleword x, a;

    *ep = 0;
    /* order matters: only isNaN can operate on NaN */
    if(isNaN(d) || isInf(d, 0) || d == 0)
        return d;
    x.x = d;
    a.x = fabs(d);
    if((a.hi >> SHIFT) == 0){   /* normalize subnormal numbers */
        x.x = (double)(1ULL<<SIG) * d;
        *ep = -SIG;
    }
    *ep += ((x.hi >> SHIFT) & MASK) - BIAS;
    x.hi &= ~(MASK << SHIFT);
    x.hi |= BIAS << SHIFT;
    return x.x;
}
```

Uses `BIAS-42` 150g, `MASK-40` 150e, `SHIFT-41` 150f, `SIG-43` 150h, `fabs()` 122b, `isInf()` 127a, and `isNaN()` 126d.

11.6.4 log()

⟨*function* `log` 151b⟩≡ (399a)

```
double
log(double arg)
{
    double x, z, zsq, temp;
    int exp;

    if(arg <= 0)
        return NaN();
    x = frexp(arg, &exp);
    while(x < 0.5) {
        x *= 2;
        exp--;
    }
    if(x < sqrto2) {
        x *= 2;
        exp--;
    }

    z = (x-1) / (x+1);
    zsq = z*z;

    temp = ((p3*zsq + p2)*zsq + p1)*zsq + p0;
    temp = temp/(((zsq + q2)*zsq + q1)*zsq + q0);
    temp = temp*z + exp*log2;
    return temp;
}
```

Uses `NaN()` 126a, `frexp()` 151a, `log2-158` 152a, `p0-161` 152d, `p1-162` 152e, `p2-163` 152f, `p3-164` 152g, `q0-165` 152h, `q1-166` 152i, `q2-167` 152j, and `sqrto2-160` 152c.

⟨*constant* `log2` 152a⟩≡ (399a)

```
#define log2    0.693147180559945309e0
```

⟨*constant* `ln10o1` 152b⟩≡ (399a)

```
#define ln10o1  .4342944819032518276511
```

⟨*constant* `sqrto2` 152c⟩≡ (399a)

```
#define sqrto2  0.707106781186547524e0
```

⟨*constant p0 (port/log.c)* 152d⟩≡ (399a)

```
#define p0      -.240139179559210510e2
```

⟨*constant p1 (port/log.c)* 152e⟩≡ (399a)

```
#define p1      0.309572928215376501e2
```

⟨*constant p2 (port/log.c)* 152f⟩≡ (399a)

```
#define p2      -.963769093377840513e1
```

⟨*constant p3 (port/log.c)* 152g⟩≡ (399a)

```
#define p3      0.421087371217979714e0
```

⟨*constant q0 (port/log.c)* 152h⟩≡ (399a)

```
#define q0      -.120069589779605255e2
```

⟨*constant q1 (port/log.c)* 152i⟩≡ (399a)

```
#define q1      0.194809660700889731e2
```

⟨*constant q2 (port/log.c)* 152j⟩≡ (399a)

```
#define q2      -.891110902798312337e1
```

⟨*function* log10 153a⟩≡ (399a)

```
double
log10(double arg)
{

    if(arg <= 0)
        return NaN();
    return log(arg) * ln10o1;
}
```

Uses NaN() 126a, ln10o1-159 152b, and log() 151b.

11.7 Random numbers

The PRNG uses a lagged Fibonacci generator with taps at positions separated by `TAP` within a `LEN`-element ring buffer. `srand()`[154e] seeds the generator with a linear congruential step. `rand()`[153b] returns 15-bit values (for ANSI C compatibility), `lrand()`[153c] returns 31-bit values, and `nrand()`[407b] returns values in $[0, n)$ with uniform distribution (it retries when the raw value would cause modulo bias).

`lrand()` uses an additive feedback shift register (a lagged Fibonacci generator) for fast pseudo-random numbers. For cryptographic randomness, `truerand()`[155a] reads from `/dev/random`, which in the kernel collects entropy from hardware sources.

⟨*function* rand 153b⟩≡ (435b)

```
int
rand(void)
{
    return lrand() & 0x7fff;
}
```

Uses lrand() 153c.

⟨*function* lrand 153c⟩≡ (400a)

```
long
lrand(void)
{
    ulong x;

    lock(&lk);

    rng_tap--;
    if(rng_tap < rng_vec) {
        if(rng_feed == 0) {
            isrand(1);
            rng_tap--;
        }
        rng_tap += LEN;
    }
    rng_feed--;
    if(rng_feed < rng_vec)
        rng_feed += LEN;
    x = (*rng_feed + *rng_tap) & MASK;
    *rng_feed = x;

    unlock(&lk);

    return x;
}
```

Uses LEN-69 399b, MASK-71 399d, isrand() 154f, lk-80 154a, lock() 245b, rng_feed-79 154d, rng_tap-78 154c, rng_vec-77 154b, and unlock() 245c.

⟨*global* lk 154a⟩≡ (400a)
```
static  Lock    lk;
```

⟨*global* rng_vec 154b⟩≡ (400a)
```
static  ulong   rng_vec[LEN];
```
Uses LEN-69 399b.

⟨*global* rng_tap 154c⟩≡ (400a)
```
static  ulong*  rng_tap = rng_vec;
```
Uses rng_tap-78 154c and rng_vec-77 154b.

⟨*global* rng_feed 154d⟩≡ (400a)
```
static  ulong*  rng_feed = 0;
```
Uses rng_feed-79 154d.

⟨*function* srand 154e⟩≡ (400a)
```
void
srand(long seed)
{
    lock(&lk);
    isrand(seed);
    unlock(&lk);
}
```
Uses isrand() 154f, lk-80 154a, lock() 245b, and unlock() 245c.

⟨*function* isrand 154f⟩≡ (400a)
```
static void
isrand(long seed)
{
    long lo, hi, x;
    int i;

    rng_tap = rng_vec;
    rng_feed = rng_vec+LEN-TAP;
    seed = seed%M;
    if(seed < 0)
        seed += M;
    if(seed == 0)
        seed = 89482311;
    x = seed;
    /*
     *  Initialize by x[n+1] = 48271 * x[n] mod (2**31 - 1)
     */
    for(i = -20; i < LEN; i++) {
        hi = x / Q;
        lo = x % Q;
        x = A*lo - R*hi;
        if(x < 0)
            x += M;
        if(i >= 0)
            rng_vec[i] = x;
    }
}
```
Uses A-72 399e, LEN-69 399b, M-73 399f, Q-74 399g, R-75 399h, TAP-70 399c, rng_feed-79 154d, rng_tap-78 154c, and rng_vec-77 154b.

⟨*function* truerand 155a⟩≡ (498b)

```
ulong
truerand(void)
{
    ulong x;
    static fdt randfd = -1;

    if(randfd < 0)
        randfd = open("/dev/random", OREAD|OCEXEC);
    if(randfd < 0)
        sysfatal("can't open /dev/random");
    if(read(randfd, &x, sizeof(x)) != sizeof(x))
        sysfatal("can't read /dev/random");

    return x;
}
```

Uses open(), read() 192a, and sysfatal() 236a.

`truerand` reads from `/dev/random`, which the kernel sources from hardware entropy (interrupt timing, etc.). `fastrand` uses the weaker `lrand` when speed matters more than unpredictability.

11.8 muldiv

⟨*function* umuldiv 155b⟩≡ (406a)

```
ulong
umuldiv(ulong a, ulong b, ulong c)
{
    double d;

    d = ((double)a * (double)b) / (double)c;
    if(d >= 4294967296.)
        abort();
    return d;
}
```

Uses abort() 356d.

⟨*function* muldiv 155c⟩≡ (406a)

```
long
muldiv(long a, long b, long c)
{
    int s;
    long v;

    s = 0;
    if(a < 0) {
        s = !s;
        a = -a;
    }
    if(b < 0) {
        s = !s;
        b = -b;
    }
    if(c < 0) {
        s = !s;
        c = -c;
    }
    v = umuldiv(a, b, c);
    if(s)
```

```
        v = -v;
    return v;
}
```

Uses `umuldiv()` 155b.

11.9 Advanced topics

11.9.1 Arbitrary precision arithmetic

The `mpdigit` type is a single "digit" for arbitrary-precision integers, defined in `mp.h`. A big integer is represented as an array of `mpdigit` values (each holding 32 bits), enabling arithmetic on numbers larger than any fixed-width type. The full arbitrary-precision library is not covered in this book.

⟨*type* `mpdigit` 156⟩≡ (368a)

```
typedef unsigned int    mpdigit;    /* for /sys/include/mp.h */
```

Chapter 12

Compression

`libflate` (declared in `flate.h`) implements the DEFLATE compression algorithm (RFC 1951) used by `gzip`, `gunzip`, and the zlib format. The API is callback-based: the caller supplies read and write functions, and `deflate()`[180]/`inflate()`[161] process data in streaming chunks without loading the entire file into memory. Block-level variants (`deflateblock()`[184], `inflateblock()`[185a]) provide a simpler in-memory interface for small data, and zlib wrappers add the framing and checksums expected by `zip` and HTTP's `Content-Encoding: deflate`.

A typical compression looks like this. Suppose we have a 2 KB log file in memory and want to gzip it. We define a trivial reader that hands `deflate` successive chunks of the buffer, and a writer that appends to an output `Biobuf`:

```
typedef struct { char *p; int n; } Mem;

static int memread(void *r, void *buf, int n) {
    Mem *m = r;
    if(n > m->n) n = m->n;
    memmove(buf, m->p, n);
    m->p += n; m->n -= n;
    return n;
}
static int biowrite(void *w, void *buf, int n) {
    return Bwrite((Biobuf*)w, buf, n);
}

Mem in = { logbuf, 2048 };
Biobuf *out = Bopen("log.gz", OWRITE);

deflateinit();
err = deflate(out, biowrite, &in, memread, 6, 0);
if(err != FlateOk)
    sysfatal("deflate: %s", flateerr(err));
```

The level argument `6` is the standard zlib default (higher = slower but smaller). Decompression mirrors the same shape, with `inflate` taking a single-byte get callback instead of a buffered read:

```
inflateinit();
err = inflate(&outmem, memwrite, in_biobuf, Bgetc);
if(err != FlateOk)
    sysfatal("inflate: %s", flateerr(err));
```

Bgetc is reused directly as the get callback, since it already has the right signature: take an opaque pointer, return the next byte or −1 at EOF. The library never sees the Biobuf type—it just calls through the function pointer.

⟨*signature* deflate 158a⟩≡ (384a)
```
int deflate(void *wr, int (*w)(void*, void*, int), void *rr, int (*r)(void*, void*, int), int level, int debug)
```

⟨*signature* inflate 158b⟩≡ (384a)
```
int inflate(void *wr, int (*w)(void*, void*, int), void *getr, int (*get)(void*));
```

12.1 Compression principles

Compression is the art of representing the same information in fewer bits. It splits into two worlds. *Lossless* compression reconstructs the original bytes *exactly* and is the only choice for text, source code, executables, filesystem archives, network protocols, and most scientific data—anything where a changed bit is a bug. *Lossy* compression (JPEG for photos, MP3 for audio, H.264 for video) throws away information the human eye or ear is unlikely to notice, achieving ratios that would be impossible under a round-trip guarantee. The rest of this section and all of libflate are about the lossless world.

Lossless compressors are all built from just two ideas, combined in different proportions. The first is *dictionary coding*, introduced by Jacob Ziv and Abraham Lempel in two landmark papers. LZ77 (1977) scans the input with a sliding window and replaces each repeated substring with a reference to its earlier occurrence ("go back 83 bytes, copy the next 9"); LZ78 (1978) instead keeps a growing explicit dictionary of phrases seen so far and emits dictionary indexes. Most modern compressors descend from one or the other: LZSS and LZMA (used in 7-Zip and xz) from LZ77; LZW (Welch 1984, used in Unix compress and GIF) from LZ78. The second idea is *entropy coding*: given a probability distribution over symbols, assign shorter codes to the more frequent ones and longer codes to the rarer ones. David Huffman's 1952 algorithm builds the optimal *prefix* code for a known distribution; arithmetic coding, range coding, and modern asymmetric numeral systems go beyond integer bit counts to squeeze closer to the entropy limit. Real compressors chain the two: dictionary first to eliminate repetition, then entropy coding to pack whatever is left.

DEFLATE, the algorithm this chapter implements, is precisely LZ77 followed by Huffman coding. Phil Katz designed it in 1989 for the PKZIP archiver, and it was standardized as RFC 1951 in 1996. A DEFLATE stream is a sequence of blocks; each block names a pair of Huffman trees—one for literal bytes and match lengths, one for match distances—and then a sequence of Huffman-coded symbols. Match references can reach back up to 32 KB (the sliding window) and cover 3–258 bytes each. Katz deliberately designed DEFLATE to sidestep the Unisys patent on LZW—the "GIF patent" that was causing legal trouble for every free-software project in the early 1990s—and DEFLATE replaced LZW almost overnight as the universal workhorse. It is what gzip/gunzip use (RFC 1952), what zlib wraps (RFC 1950), what the PNG image format uses for pixels, what the ZIP file format uses for individual entries, and what HTTP's Content-Encoding: gzip layer uses on every compressed web page.

DEFLATE is no longer state of the art. Modern replacements include ZSTD (Facebook, 2015: a similar ratio to DEFLATE at matching compression levels but much faster to decode, and drastically better at high levels—now the default in the Linux kernel, Btrfs, and most major package managers); brotli (Google, 2013: beats DEFLATE for short text and is the current HTTP-over-Chrome/ Firefox/Safari successor); LZ4 and Snappy (extreme-speed dictionary-only schemes used inside Cassandra, Kafka, LevelDB, and filesystem fast paths); and XZ/LZMA (highest ratio, slow compression, used for Linux package archives and scientific data). libflate is not trying to beat any of them on ratio or speed: it is, like the rest of this book's code, small enough to read end to end. The rest of this chapter works through its two halves—the deflate side that scans for repetitions and builds the Huffman trees, and the inflate side that rehydrates them—plus the zlib and gzip framing layers that sit on top.

12.2 Data structures and interface

The `flate.h` header defines error codes and the function signatures. The streaming API uses C function pointers as callbacks: `deflate()`[180] takes a write callback and a read callback, while `inflate()`[161] takes a write callback and a single-byte get callback. This lets the caller wire compression to any data source—files, network connections, or in-memory buffers—without the library knowing the details.

⟨*enum* `flate_error` 159⟩≡ (384a)

```
/*
 * errors from deflate, deflateinit, deflateblock,
 * inflate, inflateinit, inflateblock.
 * convertable to a string by flateerr
 */
enum
{
    FlateOk   = 0,
    FlateNoMem  = -1,
    FlateInputFail  = -2,
    FlateOutputFail  = -3,
    FlateCorrupted  = -4,
    FlateInternal  = -5,
};
```

12.3 Overview of the DEFLATE format

Before diving into the code, it helps to understand what DEFLATE actually is. The format (RFC 1951) compresses data as a sequence of *blocks*, each independently decodable. There are three block types:

Uncompressed blocks store data verbatim with just a length header—used when the data is already incompressible.

Fixed Huffman blocks use a predefined Huffman code built into both compressor and decompressor. This avoids transmitting the code table, saving space for small blocks.

Dynamic Huffman blocks transmit a custom Huffman code at the start of each block, optimized for the block's contents. This is where most of the compression happens for real data.

Within each block, DEFLATE uses two complementary techniques. LZ77 finds repeated substrings and replaces them with (length, distance) back-references into the sliding window (up to 32 KB back). Huffman coding then encodes the resulting stream of literals, lengths, and distances using variable-length bit codes, assigning shorter codes to more frequent symbols. The combination is powerful: LZ77 removes redundancy at the string level, and Huffman coding removes redundancy at the symbol level.

Here is a tiny worked example. Take the input string `abracadabra abracadabra` (23 bytes). LZ77 finds the two repetitions of `abracadabra` and emits a back-reference for the second one:

```
Pass 1 (LZ77):
  'a' 'b' 'r' 'a' 'c' 'a' 'd' 'a' 'b' 'r' 'a' ' '
  <length=11, distance=12>

After LZ77 we have 12 literals plus 1 back-reference,
instead of 23 literals.
```

The back-reference says "copy 11 bytes from 12 positions back in the output." At decompression time the decoder walks the output window backwards by 12, then copies 11 bytes forward. Note that the source range may overlap the destination—a classic trick that lets one back-reference encode a repeating pattern like `aaaa` as `'a' <length=3,distance=1>`. Then Huffman coding assigns shorter bit codes to the more frequent symbols. With four `a`s, `b` and `r` appearing twice, and other letters once, `a` might get 2 bits while `d` gets 5. The space

character and the length/distance codes are encoded the same way. The whole output ends up being a stream of variable-length bit codes, much shorter than the original 23 bytes. Real DEFLATE is more elaborate—length/distance codes have "extra bits" for fine-grained values, and dynamic blocks ship a meta-Huffman code for the code-length table itself—but the two-pass structure is exactly this.

The following diagram shows the data flow:

```
Compression:
  input bytes -> LZ77 (find matches) -> Huffman encode -> bit stream

Decompression:
  bit stream -> Huffman decode -> expand LZ77 refs -> output bytes
```

Note how decompression is simpler: it only needs to decode Huffman symbols and copy back-references from the output window. Compression must search for matches (expensive) and build optimal Huffman trees (complex). This asymmetry is reflected in the code: `deflate.c` is roughly twice the size of `inflate.c`.

Each block starts with a tiny 3-bit header that drives the top-level dispatch in `inflate()`[161]:

```
  bit 0         bits 1-2
+--------+--------------------+
| BFINAL |       BTYPE        |
+--------+--------------------+
  |         |
  |         +-- 00 = stored (uncompressed)
  |             01 = fixed Huffman
  |             10 = dynamic Huffman
  |             11 = reserved (error)
  |
  +---- 1 if this is the last block, 0 otherwise
```

The decoder reads these 3 bits via `sregfill()`[167b], switches on `BTYPE` to pick a block parser, and loops until `BFINAL` is set. Because blocks are byte-unaligned—the next block starts wherever the current one ended, down to the bit—DEFLATE streams need a running bit buffer; that is the job of the `sreg`/`nbits` pair in `Input`.

12.4 Inflation (decompression)

The decompressor reads a DEFLATE bit stream and produces the original uncompressed data. It processes one block at a time, dispatching on the block type (uncompressed, fixed Huffman, or dynamic Huffman), then decoding symbols until the end-of-block marker.

12.4.1 Initialization: `inflateinit()`

`inflateinit()`[160] builds the fixed Huffman decoding tables used by fixed-code blocks. These tables are defined by the RFC and never change, so they only need to be built once. The function returns `FlateOk` on success.

⟨*function* `inflateinit` 160⟩≡ (602b)

```
int
inflateinit(void)
{
    char *len;
    int i, j, base;

    /* byte reverse table */
    for(i=0; i<256; i++)
```

```
        for(j=0; j<8; j++)
            if(i & (1<<j))
                revtab[i] |= 0x80 >> j;

    for(i=257,base=3; i<Nlitlen; i++) {
        litlenbase[i-257] = base;
        base += 1<<litlenextra[i-257];
    }
    /* strange table entry in spec... */
    litlenbase[285-257]--;

    for(i=0,base=1; i<Noff; i++) {
        offbase[i] = base;
        base += 1<<offextra[i];
    }

    len = malloc(MaxLeaf);
    if(len == nil)
        return FlateNoMem;

    /* static Litlen bit lengths */
    for(i=0; i<144; i++)
        len[i] = 8;
    for(i=144; i<256; i++)
        len[i] = 9;
    for(i=256; i<280; i++)
        len[i] = 7;
    for(i=280; i<Nlitlen; i++)
        len[i] = 8;

    if(!hufftab(&litlentab, len, Nlitlen, MaxFlatBits))
        return FlateInternal;

    /* static Offset bit lengths */
    for(i=0; i<Noff; i++)
        len[i] = 5;

    if(!hufftab(&offtab, len, Noff, MaxFlatBits))
        return FlateInternal;
    free(len);

    return FlateOk;
}
```

Uses `MaxFlatBits-276` 602b, `MaxLeaf-277` 602b, `Nlitlen-269` 602b, `Noff-270` 602b, `free()` 63i, `hufftab()` 165, `litlenbase-279` 602b, `litlenextra-278` 602b, `litlentab-283` 602b, `malloc()` 63g, `offbase-281` 602b, `offextra-280` 602b, `offtab-284` 602b, and `revtab-285` 602b.

12.4.2 The decompression loop: `inflate()`, `decode()`

`inflate()`[161] is the main entry point. It reads block headers from the bit stream and dispatches to `uncblock()`[599c] (uncompressed), `fixedblock()`[600a] (fixed Huffman), or `dynamicblock()`[600b] (dynamic Huffman). Each of these sets up the appropriate Huffman tables and then calls `decode()`[163], which runs the actual decoding loop.

⟨*function* `inflate` 161⟩≡ (602b)

```
int
inflate(void *wr, int (*w)(void*, void*, int), void *getr, int (*get)(void*))
{
    History *his;
    Input in;
```

```
    int final, type;

    his = malloc(sizeof(History));
    if(his == nil)
        return FlateNoMem;
    his->cp = his->his;
    his->full = 0;
    in.getr = getr;
    in.get = get;
    in.wr = wr;
    in.w = w;
    in.nbits = 0;
    in.sreg = 0;
    in.error = FlateOk;

    do {
        if(!sregfill(&in, 3))
            goto bad;
        final = in.sreg & 0x1;
        type = (in.sreg>>1) & 0x3;
        in.sreg >>= 3;
        in.nbits -= 3;
        switch(type) {
        default:
            in.error = FlateCorrupted;
            goto bad;
        case 0:
            /* uncompressed */
            if(!uncblock(&in, his))
                goto bad;
            break;
        case 1:
            /* fixed huffman */
            if(!fixedblock(&in, his))
                goto bad;
            break;
        case 2:
            /* dynamic huffman */
            if(!dynamicblock(&in, his))
                goto bad;
            break;
        }
    } while(!final);

    if(his->cp != his->his && (*w)(wr, his->his, his->cp - his->his) != his->cp - his->his) {
        in.error = FlateOutputFail;
        goto bad;
    }

    if(!sregunget(&in))
        goto bad;

    free(his);
    if(in.error != FlateOk)
        return FlateInternal;
    return FlateOk;

bad:
    free(his);
    if(in.error == FlateOk)
```

```
        return FlateInternal;
    return in.error;
}
```

`decode()` is the inner loop of the decompressor. It reads Huffman-coded symbols one at a time: a literal (0–255) is copied directly to the output; a length code (257–285) is followed by a distance code, and the pair defines a back-reference—copy `length` bytes from `distance` bytes back in the output window. Symbol 256 marks end-of-block.

⟨*function* `decode` 163⟩≡ (602b)

```
static int
decode(Input *in, History *his, Huff *litlentab, Huff *offtab)
{
    int len, off;
    uchar *hs, *hp, *hq, *he;
    int c;
    int nb;

    hs = his->his;
    he = hs + HistorySize;
    hp = his->cp;

    for(;;) {
        nb = litlentab->minbits;
        for(;;){
            if(in->nbits<nb && !sregfill(in, nb))
                return 0;
            c = litlentab->flat[in->sreg & litlentab->flatmask];
            nb = c & 0xff;
            if(nb > in->nbits){
                if(nb != 0xff)
                    continue;
                c = hdecsym(in, litlentab, c);
                if(c < 0)
                    return 0;
            }else{
                c >>= 8;
                in->sreg >>= nb;
                in->nbits -= nb;
            }
            break;
        }

        if(c < 256) {
            /* literal */
            *hp++ = c;
            if(hp == he) {
                his->full = 1;
                if((*in->w)(in->wr, hs, HistorySize) != HistorySize) {
                    in->error = FlateOutputFail;
                    return 0;
                }
                hp = hs;
            }
            continue;
        }

        if(c == 256)
            break;
```

```
if(c > 285) {
    in->error = FlateCorrupted;
    return 0;
}

c -= 257;
nb = litlenextra[c];
if(in->nbits < nb && !sregfill(in, nb))
    return 0;
len = litlenbase[c] + (in->sreg & ((1<<nb)-1));
in->sreg >>= nb;
in->nbits -= nb;

/* get offset */
nb = offtab->minbits;
for(;;){
    if(in->nbits<nb && !sregfill(in, nb))
        return 0;
    c = offtab->flat[in->sreg & offtab->flatmask];
    nb = c & 0xff;
    if(nb > in->nbits){
        if(nb != 0xff)
            continue;
        c = hdecsym(in, offtab, c);
        if(c < 0)
            return 0;
    }else{
        c >>= 8;
        in->sreg >>= nb;
        in->nbits -= nb;
    }
    break;
}

if(c > 29) {
    in->error = FlateCorrupted;
    return 0;
}

nb = offextra[c];
if(in->nbits < nb && !sregfill(in, nb))
    return 0;

off = offbase[c] + (in->sreg & ((1<<nb)-1));
in->sreg >>= nb;
in->nbits -= nb;

hq = hp - off;
if(hq < hs) {
    if(!his->full) {
        in->error = FlateCorrupted;
        return 0;
    }
    hq += HistorySize;
}

/* slow but correct */
while(len) {
    *hp = *hq;
    hq++;
```

```
            hp++;
            if(hq >= he)
                hq = hs;
            if(hp == he) {
                his->full = 1;
                if((*in->w)(in->wr, hs, HistorySize) != HistorySize) {
                    in->error = FlateOutputFail;
                    return 0;
                }
                hp = hs;
            }
            len--;
        }

    }

    his->cp = hp;

    return 1;
}
```

Uses HistorySize-266 602b, hdecsym() 167a, litlenbase-279 602b, litlenextra-278 602b, offbase-281 602b, offextra-280 602b, and sregfill() 167b.

12.4.3 Huffman tree decoding: `hufftab()`, `hdecsym()`

For dynamic blocks, the Huffman trees themselves must be read from the stream. `hufftab()`[165] builds a decoding lookup table from an array of code lengths (transmitted in the block header). `hdecsym()`[167a] reads the code-length encoding—a meta-Huffman code that compresses the code lengths using run-length encoding and its own small Huffman table. This two-level encoding is one of DEFLATE's space-saving tricks.

⟨*function* `hufftab` 165⟩≡ (602b)

```
/*
 * construct the huffman decoding arrays and a fast lookup table.
 * the fast lookup is a table indexed by the next flatbits bits,
 * which returns the symbol matched and the number of bits consumed,
 * or the minimum number of bits needed and 0xff if more than flatbits
 * bits are needed.
 *
 * flatbits can be longer than the smallest huffman code,
 * because shorter codes are assigned smaller lexical prefixes.
 * this means assuming zeros for the next few bits will give a
 * conservative answer, in the sense that it will either give the
 * correct answer, or return the minimum number of bits which
 * are needed for an answer.
 */
static int
hufftab(Huff *h, char *hb, int maxleaf, int flatbits)
{
    ulong bitcount[MaxHuffBits];
    ulong c, fc, ec, mincode, code, nc[MaxHuffBits];
    int i, b, minbits, maxbits;

    for(i = 0; i < MaxHuffBits; i++)
        bitcount[i] = 0;
    maxbits = -1;
    minbits = MaxHuffBits + 1;
    for(i=0; i < maxleaf; i++){
        b = hb[i];
```

```
    if(b){
        bitcount[b]++;
        if(b < minbits)
            minbits = b;
        if(b > maxbits)
            maxbits = b;
    }
}

h->maxbits = maxbits;
if(maxbits <= 0){
    h->maxbits = 0;
    h->minbits = 0;
    h->flatmask = 0;
    return 1;
}
code = 0;
c = 0;
for(b = 0; b <= maxbits; b++){
    h->last[b] = c;
    c += bitcount[b];
    mincode = code << 1;
    nc[b] = mincode;
    code = mincode + bitcount[b];
    if(code > (1 << b))
        return 0;
    h->maxcode[b] = code - 1;
    h->last[b] += code - 1;
}

if(flatbits > maxbits)
    flatbits = maxbits;
h->flatmask = (1 << flatbits) - 1;
if(minbits > flatbits)
    minbits = flatbits;
h->minbits = minbits;

b = 1 << flatbits;
for(i = 0; i < b; i++)
    h->flat[i] = ~0;

/*
 * initialize the flat table to include the minimum possible
 * bit length for each code prefix
 */
for(b = maxbits; b > flatbits; b--){
    code = h->maxcode[b];
    if(code == -1)
        break;
    mincode = code + 1 - bitcount[b];
    mincode >>= b - flatbits;
    code >>= b - flatbits;
    for(; mincode <= code; mincode++)
        h->flat[revcode(mincode, flatbits)] = (b << 8) | 0xff;
}

for(i = 0; i < maxleaf; i++){
    b = hb[i];
    if(b <= 0)
        continue;
```

```
        c = nc[b]++;
        if(b <= flatbits){
            code = (i << 8) | b;
            ec = (c + 1) << (flatbits - b);
            if(ec > (1<<flatbits))
                return 0; /* this is actually an internal error */
            for(fc = c << (flatbits - b); fc < ec; fc++)
                h->flat[revcode(fc, flatbits)] = code;
        }
        if(b > minbits){
            c = h->last[b] - c;
            if(c >= maxleaf)
                return 0;
            h->decode[c] = i;
        }
    }
    return 1;
}
```

Uses MaxHuffBits-268 602b and revcode() 602a.

⟨*function* hdecsym 167a⟩≡ (602b)

```
static int
hdecsym(Input *in, Huff *h, int nb)
{
    long c;

    if((nb & 0xff) == 0xff)
        nb = nb >> 8;
    else
        nb = nb & 0xff;
    for(; nb <= h->maxbits; nb++){
        if(in->nbits<nb && !sregfill(in, nb))
            return -1;
        c = revtab[in->sreg&0xff]<<8;
        c |= revtab[(in->sreg>>8)&0xff];
        c >>= (16-nb);
        if(c <= h->maxcode[nb]){
            in->sreg >>= nb;
            in->nbits -= nb;
            return h->decode[h->last[nb] - c];
        }
    }
    in->error = FlateCorrupted;
    return -1;
}
```

Uses revtab-285 602b and sregfill() 167b.

12.4.4 Bit stream: sregfill(), sregunget()

The bit-level input machinery. DEFLATE is a bit stream, not a byte stream, so the decompressor needs to read individual bits. `sregfill()`[167b] refills a shift register from the byte input, and `sregunget()`[168a] pushes unused bytes back. The shift register accumulates bits so that multi-bit reads are a single mask-and-shift rather than repeated single-bit operations.

⟨*function* sregfill 167b⟩≡ (602b)

```
static int
sregfill(Input *in, int n)
{
```

```
    int c;

    while(n > in->nbits) {
        c = (*in->get)(in->getr);
        if(c < 0){
            in->error = FlateInputFail;
            return 0;
        }
        in->sreg |= c<<in->nbits;
        in->nbits += 8;
    }
    return 1;
}
```

⟨*function* sregunget 168a⟩≡ (602b)

```
static int
sregunget(Input *in)
{
    if(in->nbits >= 8) {
        in->error = FlateInternal;
        return 0;
    }

    /* throw other bits on the floor */
    in->nbits = 0;
    in->sreg = 0;
    return 1;
}
```

12.5 Deflation (compression)

The compressor is the complex half. It must find repeated substrings (LZ77 matching), decide how to encode them (literal vs. back-reference), build optimal Huffman trees from the frequency counts, and pack everything into a bit stream. The `level` parameter (0–9) controls the trade-off between compression ratio and speed, primarily by adjusting how aggressively `lzmatch()`[170] searches for matches.

12.5.1 Initialization: `deflateinit()`

`deflateinit()`[168b] builds the fixed Huffman encoding tables, byte-reversal tables, and length/distance code mappings. Like `inflateinit()`[160], this only needs to be called once. `deflatereset()`[587e] resets the compressor state between independent compression runs without rebuilding the tables.

⟨*function* deflateinit 168b⟩≡ (592)

```
int
deflateinit(void)
{
    ulong bitcount[MaxHuffBits];
    int i, j, ci, n;

    /* byte reverse table */
    for(i=0; i<256; i++)
        for(j=0; j<8; j++)
            if(i & (1<<j))
                revtab[i] |= 0x80 >> j;

    /* static Litlen bit lengths */
    for(i=0; i<144; i++)
```

```
    litlentab[i].bits = 8;
for(i=144; i<256; i++)
    litlentab[i].bits = 9;
for(i=256; i<280; i++)
    litlentab[i].bits = 7;
for(i=280; i<Nlitlen; i++)
    litlentab[i].bits = 8;

memset(bitcount, 0, sizeof(bitcount));
bitcount[8] += 144 - 0;
bitcount[9] += 256 - 144;
bitcount[7] += 280 - 256;
bitcount[8] += Nlitlen - 280;

if(!hufftabinit(litlentab, Nlitlen, bitcount, 9))
    return FlateInternal;

/* static offset bit lengths */
for(i = 0; i < Noff; i++)
    offtab[i].bits = 5;

memset(bitcount, 0, sizeof(bitcount));
bitcount[5] = Noff;

if(!hufftabinit(offtab, Noff, bitcount, 5))
    return FlateInternal;

bitcount[0] = 0;
bitcount[1] = 0;
if(!mkgzprecode(hofftab, bitcount, 2, MaxHuffBits))
    return FlateInternal;

/* conversion tables for lens & offs to codes */
ci = 0;
for(i = LenStart; i < 286; i++){
    n = ci + (1 << litlenextra[i - LenStart]);
    litlenbase[i - LenStart] = ci;
    for(; ci < n; ci++)
        lencode[ci] = i;
}
/* patch up special case for len MaxMatch */
lencode[MaxMatch-MinMatch] = 285;
litlenbase[285-LenStart] = MaxMatch-MinMatch;

ci = 0;
for(i = 0; i < 16; i++){
    n = ci + (1 << offextra[i]);
    offbase[i] = ci;
    for(; ci < n; ci++)
        offcode[ci] = i;
}

ci = ci >> 7;
for(; i < 30; i++){
    n = ci + (1 << (offextra[i] - 7));
    offbase[i] = ci << 7;
    for(; ci < n; ci++)
        bigoffcode[ci] = i;
}
return FlateOk;
```

```
}
```

Uses LenStart-292 592, MaxHuffBits-300 592, MaxMatch-298 592, MinMatch-297 592, Nlitlen-293 592, Noff-294 592, bigoffcode-321 592, hofftab-329 592, hufftabinit() 590a, lencode-319 592, litlenbase-322 592, litlenextra-323 592, litlentab-327 592, memset() 46b, mkgzprecode() 589b, offbase-324 592, offcode-320 592, offextra-325 592, offtab-328 592, and revtab-330 592.

12.5.2 LZ77 matching: lzmatch(), lzcomp()

lzcomp()[171] is the LZ77 compressor. It slides a window over the input, and at each position calls lzmatch()[170] to find the longest match in the window. Matches are found using hash chains: hashit()[587d] hashes three-byte sequences to index into a chain of previous positions with the same hash. Longer chains mean more thorough (but slower) searching —the compression level controls the chain length.

When a match is found (length ≥ 3), it is encoded as a (length, distance) pair. Otherwise, the byte is emitted as a literal. The output goes to an intermediate buffer that will later be Huffman-coded.

⟨*function* lzmatch 170⟩≡ (592)

```
/*
 * look for the longest, closest string which matches
 * the next prefix.  the clever part here is looking for
 * a string 1 longer than the previous best match.
 *
 * follows the recommendation of limiting number of chains
 * which are checked.  this appears to be the best heuristic.
 */
static int
lzmatch(int now, int then, uchar *p, uchar *es, ushort *nexts, uchar *hist, int runlen, int check, int *m)
{
    uchar *s, *t;
    int ml, off, last;

    ml = check;
    if(runlen >= 8)
        check >>= 2;
    *m = 0;
    if(p + runlen >= es)
        return runlen;
    last = 0;
    for(; check-- > 0; then = nexts[then & (MaxOff-1)]){
        off = (ushort)(now - then);
        if(off <= last || off > MaxOff)
            break;
        s = p + runlen;
        t = hist + (((p - hist) - off) & (HistBlock-1));
        t += runlen;
        for(; s >= p; s--){
            if(*s != *t)
                goto matchloop;
            t--;
        }

        /*
         * we have a new best match.
         * extend it to it's maximum length
         */
        t += runlen + 2;
        s += runlen + 2;
        for(; s < es; s++){
            if(*s != *t)
```

```
                break;
            t++;
        }
        runlen = s - p;
        *m = off - 1;
        if(s == es || runlen > ml)
            break;
matchloop:;
        last = off;
    }
    return runlen;
}
```

Uses HistBlock-310 592 and MaxOff-296 592.

⟨*function* lzcomp 171⟩≡ (592)

```
static int
lzcomp(LZstate *lz, LZblock *lzb, uchar *ep, ushort *parse, int finish)
{
    ulong cont, excost, *litlencount, *offcount;
    uchar *p, *q, *s, *es;
    ushort *nexts, *hash;
    int v, i, h, runlen, n, now, then, m, prevlen, prevoff, maxdefer;

    litlencount = lzb->litlencount;
    offcount = lzb->offcount;
    nexts = lz->nexts;
    hash = lz->hash;
    now = lz->now;

    p = &lz->hist[lz->pos];
    if(lz->prevlen != MinMatch - 1)
        p++;

    /*
     * hash in the links for any hanging link positions,
     * and calculate the hash for the current position.
     */
    n = MinMatch;
    if(n > ep - p)
        n = ep - p;
    cont = 0;
    for(i = 0; i < n - 1; i++){
        m = now - ((MinMatch-1) - i);
        if(m < lz->dot)
            continue;
        s = lz->hist + (((p - lz->hist) - (now - m)) & (HistBlock-1));

        cont = (s[0] << 16) | (s[1] << 8) | s[2];
        h = hashit(cont);
        prevoff = 0;
        for(then = hash[h]; ; then = nexts[then & (MaxOff-1)]){
            v = (ushort)(now - then);
            if(v <= prevoff || v >= (MinMatch-1) - i)
                break;
            prevoff = v;
        }
        if(then == (ushort)m)
            continue;
        nexts[m & (MaxOff-1)] = hash[h];
        hash[h] = m;
```

```
}
for(i = 0; i < n; i++)
    cont = (cont << 8) | p[i];

/*
 * now must point to the index in the nexts array
 * corresponding to p's position in the history
 */
prevlen = lz->prevlen;
prevoff = lz->prevoff;
maxdefer = lz->maxcheck >> 2;
excost = 0;
v = lzb->lastv;
for(;;){
    es = p + MaxMatch;
    if(es > ep){
        if(!finish || p >= ep)
            break;
        es = ep;
    }

    h = hashit(cont);
    runlen = lzmatch(now, hash[h], p, es, nexts, lz->hist, prevlen, lz->maxcheck, &m);

    /*
     * back out of small matches too far in the past
     */
    if(runlen == MinMatch && m >= MinMatchMaxOff){
        runlen = MinMatch - 1;
        m = 0;
    }

    /*
     * record the encoding and increment counts for huffman trees
     * if we get a match, defer selecting it until we check for
     * a longer match at the next position.
     */
    if(prevlen >= runlen && prevlen != MinMatch - 1){
        /*
         * old match at least as good; use that one
         */
        n = prevlen - MinMatch;
        if(v || n){
            *parse++ = v | LenFlag | (n << LenShift);
            *parse++ = prevoff;
        }else
            *parse++ = prevoff << LenShift;
        v = 0;

        n = lencode[n];
        litlencount[n]++;
        excost += litlenextra[n - LenStart];

        if(prevoff < MaxOffCode)
            n = offcode[prevoff];
        else
            n = bigoffcode[prevoff >> 7];
        offcount[n]++;
        excost += offextra[n];
```

```
    runlen = prevlen - 1;
    prevlen = MinMatch - 1;
    nmatches++;
}else if(runlen == MinMatch - 1){
    /*
     * no match; just put out the literal
     */
    if(++v == MaxLitRun){
        *parse++ = v;
        v = 0;
    }
    litlencount[*p]++;
    nlits++;
    runlen = 1;
}else{
    if(prevlen != MinMatch - 1){
        /*
         * longer match now. output previous literal,
         * update current match, and try again
         */
        if(++v == MaxLitRun){
            *parse++ = v;
            v = 0;
        }
        litlencount[p[-1]]++;
        nlits++;
    }

    prevoff = m;

    if(runlen < maxdefer){
        prevlen = runlen;
        runlen = 1;
    }else{
        n = runlen - MinMatch;
        if(v || n){
            *parse++ = v | LenFlag | (n << LenShift);
            *parse++ = prevoff;
        }else
            *parse++ = prevoff << LenShift;
        v = 0;

        n = lencode[n];
        litlencount[n]++;
        excost += litlenextra[n - LenStart];

        if(prevoff < MaxOffCode)
            n = offcode[prevoff];
        else
            n = bigoffcode[prevoff >> 7];
        offcount[n]++;
        excost += offextra[n];

        prevlen = MinMatch - 1;
        nmatches++;
    }
}

/*
 * update the hash for the newly matched data
```

```
         * this is constructed so the link for the old
         * match in this position must be at the end of a chain,
         * and will expire when this match is added, ie it will
         * never be examined by the match loop.
         * add to the hash chain only if we have the real hash data.
         */
        for(q = p + runlen; p != q; p++){
            if(p + MinMatch <= ep){
                h = hashit(cont);
                nexts[now & (MaxOff-1)] = hash[h];
                hash[h] = now;
                if(p + MinMatch < ep)
                    cont = (cont << 8) | p[MinMatch];
            }
            now++;
        }
    }

    /*
     * we can just store away the lazy state and
     * pick it up next time.  the last block will have finish set
     * so we won't have any pending matches
     * however, we need to correct for how much we've encoded
     */
    if(prevlen != MinMatch - 1)
        p--;

    lzb->excost += excost;
    lzb->eparse = parse;
    lzb->lastv = v;

    lz->now = now;
    lz->prevlen = prevlen;
    lz->prevoff = prevoff;

    return p - &lz->hist[lz->pos];
}
```

Uses `HistBlock-310` 592, `LenFlag-302` 592, `LenShift-303` 592, `LenStart-292` 592, `MaxLitRun-304` 592, `MaxMatch-298` 592, `MaxOff-296` 592, `MaxOffCode-314` 592, `MinMatch-297` 592, `MinMatchMaxOff-308` 592, `bigoffcode-321` 592, `hashit-318` 587d, `lencode-319` 592, `litlenextra-323` 592, `lzmatch()` 170, `nlits-331` 592, `nmatches-332` 592, `offcode-320` 592, and `offextra-325` 592.

12.5.3 Huffman tree construction: `mkprecode()`, `huffcodes()`, `leafsort()`

After LZ77 compression, the frequency of each symbol (literals 0–255, length codes 257–285, distance codes 0–29) is known. `mkprecode()`[174] builds an optimal Huffman code from these frequencies using a boundary package-merge variant: `nextchain()`[590b] and `leafsort()`[177] implement the algorithm that finds minimum-redundancy codes respecting the maximum bit-length constraint (15 bits for DEFLATE). `huffcodes()`[176] then converts the bit lengths into canonical Huffman codes.

⟨*function* `mkprecode` 174⟩≡ (592)

```
/*
 * fast, low space overhead algorithm for max depth huffman type codes
 *
 * J. Katajainen, A. Moffat and A. Turpin, "A fast and space-economical
 * algorithm for length-limited coding," Proc. Intl. Symp. on Algorithms
 * and Computation, Cairns, Australia, Dec. 1995, Lecture Notes in Computer
 * Science, Vol 1004, J. Staples, P. Eades, N. Katoh, and A. Moffat, eds.,
```

```
 * pp 12-21, Springer Verlag, New York, 1995.
 */
static int
mkprecode(Huff *tab, ulong *count, int n, int maxbits, ulong *bitcount)
{
    Chains cs;
    Chain *c;
    int i, m, em, bits;

    /*
     * set up the sorted list of leaves
     */
    m = 0;
    for(i = 0; i < n; i++){
        tab[i].bits = -1;
        tab[i].encode = 0;
        if(count[i] != 0){
            cs.leafcount[m] = count[i];
            cs.leafmap[m] = i;
            m++;
        }
    }
    if(m < 2){
        if(m != 0){
            tab[cs.leafmap[0]].bits = 0;
            bitcount[0] = 1;
        }else
            bitcount[0] = 0;
        return 0;
    }
    cs.nleaf = m;
    leafsort(cs.leafcount, cs.leafmap, 0, m);

    for(i = 0; i < m; i++)
        cs.leafcount[i] = count[cs.leafmap[i]];

    /*
     * set up free list
     */
    cs.free = &cs.chains[2];
    cs.echains = &cs.chains[ChainMem];
    cs.col = 1;

    /*
     * initialize chains for each list
     */
    c = &cs.chains[0];
    c->count = cs.leafcount[0];
    c->leaf = 1;
    c->col = cs.col;
    c->up = nil;
    c->gen = 0;
    cs.chains[1] = cs.chains[0];
    cs.chains[1].leaf = 2;
    cs.chains[1].count = cs.leafcount[1];
    for(i = 0; i < maxbits-1; i++){
        cs.lists[i * 2] = &cs.chains[0];
        cs.lists[i * 2 + 1] = &cs.chains[1];
    }
```

```
    cs.nlists = 2 * (maxbits - 1);
    m = 2 * m - 2;
    for(i = 2; i < m; i++)
        nextchain(&cs, cs.nlists - 2);

    bits = 0;
    bitcount[0] = cs.nleaf;
    for(c = cs.lists[cs.nlists - 1]; c != nil; c = c->up){
        m = c->leaf;
        bitcount[bits++] -= m;
        bitcount[bits] = m;
    }
    m = 0;
    for(i = bits; i >= 0; i--)
        for(em = m + bitcount[i]; m < em; m++)
            tab[cs.leafmap[m]].bits = i;

    return bits;
}
```

Uses ChainMem-301 592, leafsort() 177, and nextchain() 590b.

⟨*function* huffcodes 176⟩≡ (592)

```
/*
 * make up the dynamic code tables, and return the number of bits
 * needed to transmit them.
 */
static int
huffcodes(Dyncode *dc, Huff *littab, Huff *offtab)
{
    Huff *codetab;
    uchar *codes, *codeaux;
    ulong codecount[Nclen], excost;
    int i, n, m, v, c, nlit, noff, ncode, nclen;

    codetab = dc->codetab;
    codes = dc->codes;
    codeaux = dc->codeaux;

    /*
     * trim the sizes of the tables
     */
    for(nlit = Nlitlen; nlit > 257 && littab[nlit-1].bits == 0; nlit--)
        ;
    for(noff = Noff; noff > 1 && offtab[noff-1].bits == 0; noff--)
        ;

    /*
     * make the code-length code
     */
    for(i = 0; i < nlit; i++)
        codes[i] = littab[i].bits;
    for(i = 0; i < noff; i++)
        codes[i + nlit] = offtab[i].bits;

    /*
     * run-length compress the code-length code
     */
    excost = 0;
    c = 0;
    ncode = nlit+noff;
```

```
    for(i = 0; i < ncode; ){
        n = i + 1;
        v = codes[i];
        while(n < ncode && v == codes[n])
            n++;
        n -= i;
        i += n;
        if(v == 0){
            while(n >= 11){
                m = n;
                if(m > 138)
                    m = 138;
                codes[c] = 18;
                codeaux[c++] = m - 11;
                n -= m;
                excost += 7;
            }
            if(n >= 3){
                codes[c] = 17;
                codeaux[c++] = n - 3;
                n = 0;
                excost += 3;
            }
        }
        while(n--){
            codes[c++] = v;
            while(n >= 3){
                m = n;
                if(m > 6)
                    m = 6;
                codes[c] = 16;
                codeaux[c++] = m - 3;
                n -= m;
                excost += 3;
            }
        }
    }

    memset(codecount, 0, sizeof codecount);
    for(i = 0; i < c; i++)
        codecount[codes[i]]++;
    if(!mkgzprecode(codetab, codecount, Nclen, 8))
        return -1;

    for(nclen = Nclen; nclen > 4 && codetab[clenorder[nclen-1]].bits == 0; nclen--)
        ;

    dc->nlit = nlit;
    dc->noff = noff;
    dc->nclen = nclen;
    dc->ncode = c;

    return 5 + 5 + 4 + nclen * 3 + bitcost(codetab, codecount, Nclen) + excost;
}
```

Uses `Nclen-295` 592, `Nlitlen-293` 592, `Noff-294` 592, `bitcost()` 589a, `clenorder-326` 592, `memset()` 46b, and `mkgzprecode()` 589b.

⟨*function* `leafsort` 177⟩≡ (592)

```
static void
leafsort(ulong *leafcount, ushort *leafmap, int a, int n)
```

```
{
    ulong t;
    int j, pi, pj, pn;

    while(n > 1){
        if(n > 10){
            pi = pivot(leafcount, a, n);
        }else
            pi = a + (n>>1);

        t = leafcount[pi];
        leafcount[pi] = leafcount[a];
        leafcount[a] = t;
        t = leafmap[pi];
        leafmap[pi] = leafmap[a];
        leafmap[a] = t;
        pi = a;
        pn = a + n;
        pj = pn;
        for(;;){
            do
                pi++;
            while(pi < pn && (leafcount[pi] < leafcount[a] || leafcount[pi] == leafcount[a] && leafmap[pi] > le
            do
                pj--;
            while(pj > a && (leafcount[pj] > leafcount[a] || leafcount[pj] == leafcount[a] && leafmap[pj] < lea
            if(pj < pi)
                break;
            t = leafcount[pi];
            leafcount[pi] = leafcount[pj];
            leafcount[pj] = t;
            t = leafmap[pi];
            leafmap[pi] = leafmap[pj];
            leafmap[pj] = t;
        }
        t = leafcount[a];
        leafcount[a] = leafcount[pj];
        leafcount[pj] = t;
        t = leafmap[a];
        leafmap[a] = leafmap[pj];
        leafmap[pj] = t;
        j = pj - a;

        n = n-j-1;
        if(j >= n){
            leafsort(leafcount, leafmap, a, j);
            a += j+1;
        }else{
            leafsort(leafcount, leafmap, a + (j+1), n);
            n = j;
        }
    }
}
```

Uses `leafsort()` 177.

12.5.4 Bit output: `lzput()`, `lzflushbits()`, `wrblock()`

`lzput()`[179a] appends bits to the output shift register, and `lzflushbits()`[179b] writes completed bytes to the output. `wrblock()`[179c] assembles a complete DEFLATE block: it writes the block header, the Huffman code tables (for dynamic blocks, via `wrdyncode()`[588c]), and then the Huffman-encoded symbols. `bitcost()`[589a] estimates the output size to decide whether a dynamic or fixed Huffman block is cheaper.

⟨*function* `lzput` 179a⟩≡ (592)

```
static void
lzput(LZstate *lz, ulong bits, int nbits)
{
    bits = (bits << lz->nbits) | lz->bits;
    for(nbits += lz->nbits; nbits >= 8; nbits -= 8){
        *lz->out++ = bits;
        if(lz->out == lz->eout)
            lzflush(lz);
        bits >>= 8;
    }
    lz->bits = bits;
    lz->nbits = nbits;
}
```

Uses `lzflush()` 588b.

⟨*function* `lzflushbits` 179b⟩≡ (592)

```
static void
lzflushbits(LZstate *lz)
{
    if(lz->nbits)
        lzput(lz, 0, 8 - (lz->nbits & 7));
}
```

Uses `lzput()` 179a.

⟨*function* `wrblock` 179c⟩≡ (592)

```
/*
 * write out a block of n samples,
 * given lz encoding and counts for huffman tables
 */
static void
wrblock(LZstate *out, int litoff, ushort *soff, ushort *eoff, Huff *litlentab, Huff *offtab)
{
    ushort *off;
    int i, run, offset, lit, len, c;

    if(out->debug > 2){
        for(off = soff; off < eoff; ){
            offset = *off++;
            run = offset & MaxLitRun;
            if(run){
                for(i = 0; i < run; i++){
                    lit = out->hist[litoff & (HistBlock - 1)];
                    litoff++;
                    fprint(2, "\tlit %.2ux %c\n", lit, lit);
                }
                if(!(offset & LenFlag))
                    continue;
                len = offset >> LenShift;
                offset = *off++;
            }else if(offset & LenFlag){
                len = offset >> LenShift;
                offset = *off++;
```

```
            }else{
                len = 0;
                offset >>= LenShift;
            }
            litoff += len + MinMatch;
            fprint(2, "\t<%d, %d>\n", offset + 1, len + MinMatch);
        }
    }

    for(off = soff; off < eoff; ){
        offset = *off++;
        run = offset & MaxLitRun;
        if(run){
            for(i = 0; i < run; i++){
                lit = out->hist[litoff & (HistBlock - 1)];
                litoff++;
                lzput(out, litlentab[lit].encode, litlentab[lit].bits);
            }
            if(!(offset & LenFlag))
                continue;
            len = offset >> LenShift;
            offset = *off++;
        }else if(offset & LenFlag){
            len = offset >> LenShift;
            offset = *off++;
        }else{
            len = 0;
            offset >>= LenShift;
        }
        litoff += len + MinMatch;
        c = lencode[len];
        lzput(out, litlentab[c].encode, litlentab[c].bits);
        c -= LenStart;
        if(litlenextra[c])
            lzput(out, len - litlenbase[c], litlenextra[c]);

        if(offset < MaxOffCode)
            c = offcode[offset];
        else
            c = bigoffcode[offset >> 7];
        lzput(out, offtab[c].encode, offtab[c].bits);
        if(offextra[c])
            lzput(out, offset - offbase[c], offextra[c]);
    }
}
```

Uses `HistBlock-310` 592, `LenFlag-302` 592, `LenShift-303` 592, `LenStart-292` 592, `MaxLitRun-304` 592, `MaxOffCode-314` 592, `MinMatch-297` 592, `bigoffcode-321` 592, `fprint()` 75a, `lencode-319` 592, `litlenbase-322` 592, `litlenextra-323` 592, `lzput()` 179a, `offbase-324` 592, `offcode-320` 592, and `offextra-325` 592.

12.5.5 The compression loop: `deflate()`, `deflateb()`

`deflate()`[180] is the top-level entry point. It calls `deflateb()`[181], which reads input in chunks, runs `lzcomp()`[171] to find matches, then calls `wrblock()`[179c] to emit each DEFLATE block. The compressor decides block boundaries based on when the Huffman statistics shift enough to warrant starting a fresh block with new code tables.

⟨*function* `deflate` 180⟩≡ (592)

```
int
deflate(void *wr, int (*w)(void*, void*, int), void *rr, int (*r)(void*, void*, int), int level, int debug)
{
```

```
    LZstate *lz;
    LZblock *lzb;
    int ok;

    lz = malloc(sizeof *lz + sizeof *lzb);
    if(lz == nil)
        return FlateNoMem;
    lzb = (LZblock*)&lz[1];

    deflatereset(lz, level, debug);
    lz->w = w;
    lz->wr = wr;
    lz->wbad = 0;
    lz->rbad = 0;
    lz->eof = 0;
    ok = FlateOk;
    while(!lz->eof || lz->avail){
        ok = deflateb(lz, lzb, rr, r);
        if(ok != FlateOk)
            break;
    }
    if(ok == FlateOk && lz->rbad)
        ok = FlateInputFail;
    if(ok == FlateOk && lz->wbad)
        ok = FlateOutputFail;
    free(lz);
    return ok;
}
```

Uses `deflateb()` 181, `deflatereset()` 587e, `free()` 63i, and `malloc()` 63g.

⟨*function* `deflateb` 181⟩≡ (592)

```
static int
deflateb(LZstate *lz, LZblock *lzb, void *rr, int (*r)(void*, void*, int))
{
    Dyncode dyncode, hdyncode;
    Huff dlitlentab[Nlitlen], dofftab[Noff], hlitlentab[Nlitlen];
    ulong litcount[Nlitlen];
    long nunc, ndyn, nfix, nhuff;
    uchar *slop, *hslop;
    ulong ep;
    int i, n, m, mm, nslop;

    memset(lzb->litlencount, 0, sizeof lzb->litlencount);
    memset(lzb->offcount, 0, sizeof lzb->offcount);
    lzb->litlencount[DeflateEob]++;

    lzb->bytes = 0;
    lzb->eparse = lzb->parse;
    lzb->lastv = 0;
    lzb->excost = 0;

    slop = &lz->hist[lz->pos];
    n = lz->avail;
    while(n < DeflateBlock && (!lz->eof || lz->avail)){
        /*
         * fill the buffer as much as possible,
         * while leaving room for MaxOff history behind lz->pos,
         * and not reading more than we can handle.
         *
         * make sure we read at least HistSlop bytes.
```

```
     */
    if(!lz->eof){
        ep = lz->pos + lz->avail;
        if(ep >= HistBlock)
            ep -= HistBlock;
        m = HistBlock - MaxOff - lz->avail;
        if(m > HistBlock - n)
            m = HistBlock - n;
        if(m > (HistBlock + HistSlop) - ep)
            m = (HistBlock + HistSlop) - ep;
        if(m & ~(BlockSize - 1))
            m &= ~(BlockSize - 1);

        /*
         * be nice to the caller: stop reads that are too small.
         * can only get here when we've already filled the buffer some
         */
        if(m < HistSlop){
            if(!m || !lzb->bytes)
                return FlateInternal;
            break;
        }

        mm = (*r)(rr, &lz->hist[ep], m);
        if(mm > 0){
            /*
             * wrap data to end if we're read it from the beginning
             * this way, we don't have to wrap searches.
             *
             * wrap reads past the end to the beginning.
             * this way, we can guarantee minimum size reads.
             */
            if(ep < HistSlop)
                memmove(&lz->hist[ep + HistBlock], &lz->hist[ep], HistSlop - ep);
            else if(ep + mm > HistBlock)
                memmove(&lz->hist[0], &lz->hist[HistBlock], ep + mm - HistBlock);

            lz->totr += mm;
            n += mm;
            lz->avail += mm;
        }else{
            if(mm < 0)
                lz->rbad = 1;
            lz->eof = 1;
        }
    }
    ep = lz->pos + lz->avail;
    if(ep > HistSize)
        ep = HistSize;
    if(lzb->bytes + ep - lz->pos > DeflateMaxBlock)
        ep = lz->pos + DeflateMaxBlock - lzb->bytes;
    m = lzcomp(lz, lzb, &lz->hist[ep], lzb->eparse, lz->eof);
    lzb->bytes += m;
    lz->pos = (lz->pos + m) & (HistBlock - 1);
    lz->avail -= m;
}
if(lzb->lastv)
    *lzb->eparse++ = lzb->lastv;
if(lzb->eparse > lzb->parse + nelem(lzb->parse))
    return FlateInternal;
```

```
nunc = lzb->bytes;

if(!mkgzprecode(dlitlentab, lzb->litlencount, Nlitlen, MaxHuffBits)
|| !mkgzprecode(dofftab, lzb->offcount, Noff, MaxHuffBits))
    return FlateInternal;

ndyn = huffcodes(&dyncode, dlitlentab, dofftab);
if(ndyn < 0)
    return FlateInternal;
ndyn += bitcost(dlitlentab, lzb->litlencount, Nlitlen)
    + bitcost(dofftab, lzb->offcount, Noff)
    + lzb->excost;

memset(litcount, 0, sizeof litcount);

nslop = nunc;
if(nslop > &lz->hist[HistSize] - slop)
    nslop = &lz->hist[HistSize] - slop;

for(i = 0; i < nslop; i++)
    litcount[slop[i]]++;
hslop = &lz->hist[HistSlop - nslop];
for(; i < nunc; i++)
    litcount[hslop[i]]++;
litcount[DeflateEob]++;

if(!mkgzprecode(hlitlentab, litcount, Nlitlen, MaxHuffBits))
    return FlateInternal;
nhuff = huffcodes(&hdyncode, hlitlentab, hofftab);
if(nhuff < 0)
    return FlateInternal;
nhuff += bitcost(hlitlentab, litcount, Nlitlen);

nfix = bitcost(litlentab, lzb->litlencount, Nlitlen)
    + bitcost(offtab, lzb->offcount, Noff)
    + lzb->excost;

lzput(lz, lz->eof && !lz->avail, 1);

if(lz->debug){
    fprint(2, "block: bytes=%lud entries=%ld extra bits=%d\n\tuncompressed=%lud fixed=%lud dynamic=%lud huf
        nunc, lzb->eparse - lzb->parse, lzb->excost, (nunc + 4) * 8, nfix, ndyn, nhuff);
    fprint(2, "\tnlit=%lud matches=%lud eof=%d\n", nlits, nmatches, lz->eof && !lz->avail);
}

if((nunc + 4) * 8 < ndyn && (nunc + 4) * 8 < nfix && (nunc + 4) * 8 < nhuff){
    lzput(lz, DeflateUnc, 2);
    lzflushbits(lz);

    lzput(lz, nunc & 0xff, 8);
    lzput(lz, (nunc >> 8) & 0xff, 8);
    lzput(lz, ~nunc & 0xff, 8);
    lzput(lz, (~nunc >> 8) & 0xff, 8);
    lzflush(lz);

    lzwrite(lz, slop, nslop);
    lzwrite(lz, &lz->hist[HistSlop], nunc - nslop);
}else if(ndyn < nfix && ndyn < nhuff){
    lzput(lz, DeflateDyn, 2);
```

```
        wrdyncode(lz, &dyncode);
        wrblock(lz, slop - lz->hist, lzb->parse, lzb->eparse, dlitlentab, dofftab);
        lzput(lz, dlitlentab[DeflateEob].encode, dlitlentab[DeflateEob].bits);
    }else if(nhuff < nfix){
        lzput(lz, DeflateDyn, 2);

        wrdyncode(lz, &hdyncode);

        m = 0;
        for(i = nunc; i > MaxLitRun; i -= MaxLitRun)
            lzb->parse[m++] = MaxLitRun;
        lzb->parse[m++] = i;

        wrblock(lz, slop - lz->hist, lzb->parse, lzb->parse + m, hlitlentab, hofftab);
        lzput(lz, hlitlentab[DeflateEob].encode, hlitlentab[DeflateEob].bits);
    }else{
        lzput(lz, DeflateFix, 2);

        wrblock(lz, slop - lz->hist, lzb->parse, lzb->eparse, litlentab, offtab);
        lzput(lz, litlentab[DeflateEob].encode, litlentab[DeflateEob].bits);
    }

    if(lz->eof && !lz->avail){
        lzflushbits(lz);
        lzflush(lz);
    }
    return FlateOk;
}
```

Uses `BlockSize-306` 592, `DeflateBlock-307` 592, `DeflateDyn-288` 592, `DeflateEob-289` 592, `DeflateFix-287` 592, `DeflateMaxBlock-290` 592, `DeflateUnc-286` 592, `HistBlock-310` 592, `HistSize-311` 592, `HistSlop-309` 592, `MaxHuffBits-300` 592, `MaxLitRun-304` 592, `MaxOff-296` 592, `Nlitlen-293` 592, `Noff-294` 592, `bitcost()` 589a, `fprint()` 75a, `hofftab-329` 592, `huffcodes()` 176, `litlentab-327` 592, `lzcomp()` 171, `lzflush()` 588b, `lzflushbits()` 179b, `lzput()` 179a, `lzwrite()` 588a, `memmove()` 48, `memset()` 46b, `mkgzprecode()` 589b, `nlits-331` 592, `nmatches-332` 592, `offtab-328` 592, `wrblock()` 179c, and `wrdyncode()` 588c.

12.6 Block interface: `deflateblock()`, `inflateblock()`

`deflateblock()`[184] and `inflateblock()`[185a] are convenience wrappers that compress or decompress a single in-memory buffer. They construct trivial read/write callbacks (using an internal `Block` struct with a position pointer) and delegate to the streaming `deflate()`[180] and `inflate()`[161] functions. The return value is the output size on success or a negative error code.

⟨*function* `deflateblock` 184⟩≡ (597b)

```
int
deflateblock(uchar *dst, int dsize, uchar *src, int ssize, int level, int debug)
{
    Block bd, bs;
    int ok;

    bs.pos = src;
    bs.limit = src + ssize;

    bd.pos = dst;
    bd.limit = dst + dsize;

    ok = deflate(&bd, blwrite, &bs, blread, level, debug);
    if(ok != FlateOk)
        return ok;
```

```
    return bd.pos - dst;
}
```
Uses `blread()` 596 and `deflate()` 180.

⟨*function* `inflateblock` 185a⟩≡ (605a)
```
int
inflateblock(uchar *dst, int dsize, uchar *src, int ssize)
{
    Block bd, bs;
    int ok;

    bs.pos = src;
    bs.limit = src + ssize;

    bd.pos = dst;
    bd.limit = dst + dsize;

    ok = inflate(&bd, blwrite, &bs, blgetc);
    if(ok != FlateOk)
        return ok;
    return bd.pos - dst;
}
```

12.7 Checksums: `adler32()`, `blockcrc()`

Two checksum algorithms are provided. `adler32()`[185b] computes the Adler-32 checksum used by the zlib framing format—a pair of 16-bit sums (one cumulative, one of the cumulative sums) reduced modulo the largest prime below 2^{16} (65521). The inner loop is unrolled 16× for speed, processing batches of up to 5552 bytes before reducing (the maximum before the 32-bit accumulators can overflow).

⟨*function* `adler32` 185b⟩≡ (587a)
```
ulong
adler32(ulong adler, void *vbuf, int n)
{
    ulong s1, s2;
    uchar *buf, *ebuf;
    int m;

    buf = vbuf;
    s1 = adler & 0xffff;
    s2 = (adler >> 16) & 0xffff;
    for(; n >= 16; n -= m){
        m = n;
        if(m > ADLERITERS)
            m = ADLERITERS;
        m &= ~15;
        for(ebuf = buf + m; buf < ebuf; buf += 16){
            s1 += buf[0];
            s2 += s1;
            s1 += buf[1];
            s2 += s1;
            s1 += buf[2];
            s2 += s1;
            s1 += buf[3];
            s2 += s1;
            s1 += buf[4];
            s2 += s1;
            s1 += buf[5];
```

```
            s2 += s1;
            s1 += buf[6];
            s2 += s1;
            s1 += buf[7];
            s2 += s1;
            s1 += buf[8];
            s2 += s1;
            s1 += buf[9];
            s2 += s1;
            s1 += buf[10];
            s2 += s1;
            s1 += buf[11];
            s2 += s1;
            s1 += buf[12];
            s2 += s1;
            s1 += buf[13];
            s2 += s1;
            s1 += buf[14];
            s2 += s1;
            s1 += buf[15];
            s2 += s1;
        }
        s1 %= ADLERBASE;
        s2 %= ADLERBASE;
    }
    if(n){
        for(ebuf = buf + n; buf < ebuf; buf++){
            s1 += buf[0];
            s2 += s1;
        }
        s1 %= ADLERBASE;
        s2 %= ADLERBASE;
    }
    return (s2 << 16) + s1;
}
```

Uses `ADLERBASE-265` 587a and `ADLERITERS-264` 587a.

`blockcrc()`[186] computes CRC-32 using a 256-entry lookup table built by `mkcrctab()`[587b]. The table is polynomial-specific: `gzip` uses the standard CRC-32 polynomial 0xEDB88320. Each byte is processed by XORing with the low byte of the running CRC, indexing into the table, and shifting.

⟨*function* `blockcrc` 186⟩≡ (587c)

```
ulong
blockcrc(ulong *crctab, ulong crc, void *vbuf, int n)
{
    uchar *buf, *ebuf;

    crc ^= 0xffffffff;
    buf = vbuf;
    ebuf = buf + n;
    while(buf < ebuf)
        crc = crctab[(crc & 0xff) ^ *buf++] ^ (crc >> 8);
    return crc ^ 0xffffffff;
}
```

12.8 Zlib wrappers: deflatezlib(), inflatezlib()

The zlib format (RFC 1950) wraps a raw DEFLATE stream with a 2-byte header and a trailing Adler-32 checksum of the uncompressed data. deflatezlib()[187a] writes the header, runs deflate()[180] with a wrapper that checksums data as it's read, and appends the checksum. inflatezlib()[187b] validates the header, runs inflate()[161] with a wrapper that checksums data as it's written, and verifies the trailing checksum. Block variants (deflatezlibblock()[598b], inflatezlibblock()[606b]) do the same for in-memory buffers.

⟨*function* deflatezlib 187a⟩≡ (598a)

```
int
deflatezlib(void *wr, int (*w)(void*, void*, int), void *rr, int (*r)(void*, void*, int), int level, int debug)
{
    ZRead zr;
    uchar buf[4];
    int ok;

    buf[0] = ZlibDeflate | ZlibWin32k;

    /* bogus zlib encoding of compression level */
    buf[1] = ((level > 2) + (level > 5) + (level > 8)) << 6;

    /* header check field */
    buf[1] |= 31 - ((buf[0] << 8) | buf[1]) % 31;
    if((*w)(wr, buf, 2) != 2)
        return FlateOutputFail;

    zr.rr = rr;
    zr.r = r;
    zr.adler = 1;
    ok = deflate(wr, w, &zr, zlread, level, debug);
    if(ok != FlateOk)
        return ok;

    buf[0] = zr.adler >> 24;
    buf[1] = zr.adler >> 16;
    buf[2] = zr.adler >> 8;
    buf[3] = zr.adler;
    if((*w)(wr, buf, 4) != 4)
        return FlateOutputFail;

    return FlateOk;
}
```

Uses ZlibDeflate 607, ZlibWin32k 607, deflate() 180, and zlread() 597c.

⟨*function* inflatezlib 187b⟩≡ (605c)

```
int
inflatezlib(void *wr, int (*w)(void*, void*, int), void *getr, int (*get)(void*))
{
    ZWrite zw;
    ulong v;
    int c, i;

    c = (*get)(getr);
    if(c < 0)
        return FlateInputFail;
    i = (*get)(getr);
    if(i < 0)
        return FlateInputFail;
```

```
    if(((c << 8) | i) % 31)
        return FlateCorrupted;
    if((c & ZlibMeth) != ZlibDeflate
    || (c & ZlibCInfo) > ZlibWin32k)
        return FlateCorrupted;

    zw.wr = wr;
    zw.w = w;
    zw.adler = 1;
    i = inflate(&zw, zlwrite, getr, get);
    if(i != FlateOk)
        return i;

    v = 0;
    for(i = 0; i < 4; i++){
        c = (*get)(getr);
        if(c < 0)
            return FlateInputFail;
        v = (v << 8) | c;
    }
    if(zw.adler != v)
        return FlateCorrupted;

    return FlateOk;
}
```

Uses `ZlibCInfo` 607, `ZlibDeflate` 607, `ZlibMeth` 607, `ZlibWin32k` 607, and `zlwrite()` 605b.

12.9 Error handling: `flateerr()`

`flateerr()`[188] converts a `Flate` error code to a human-readable string. The error codes cover the expected failure modes: out of memory, I/O failure (read or write callback returned an error), corrupted data (invalid Huffman codes, bad checksums), and internal errors (bugs in the library).

⟨*function* `flateerr` 188⟩≡ (599b)

```
char *
flateerr(int err)
{
    switch(err){
    case FlateOk:
        return "no error";
    case FlateNoMem:
        return "out of memory";
    case FlateInputFail:
        return "input error";
    case FlateOutputFail:
        return "output error";
    case FlateCorrupted:
        return "corrupted data";
    case FlateInternal:
        return "internal error";
    }
    return "unknown error";
}
```

Part III

System Services

This part covers the library functions that wrap system calls: file I/O and buffered I/O, directory and path operations, namespace manipulation (`bind`/`mount`), time, process management (`fork`/`exec`/`wait`), error handling, and security. Each chapter follows a recurring pattern: first the raw system call, then convenience wrappers, then higher-level abstractions built on top.

Chapter 13

File IO

File I/O in Plan 9 is built on five functions: `open`, `create`, `read`, `write`, and `close`. The Plan 9 interface differs from UNIX in a few ways: `read` and `write` are implemented as wrappers around `pread`/`pwrite` (which take an explicit offset, making them atomic for concurrent access), and file modes include Plan 9-specific bits like `OEXCL` (exclusive create) and `ORCLOSE` (remove on close).

⟨*signatures file syscall wrapper* 190a⟩≡ (379b)

```
extern long read(fdt, void*, long);
extern long write(fdt, void*, long);
extern vlong seek(fdt, vlong, int);
```

⟨*signatures file other wrapper* 190b⟩≡ (379b)

```
extern long preadv(fdt, IOchunk*, int, vlong);
extern long pwritev(fdt, IOchunk*, int, vlong);
extern long readv(fdt, IOchunk*, int);
extern long writev(fdt, IOchunk*, int);
extern long readn(fdt, void*, long);
```

13.1 Data structures

The `Qid` (unique identifier) is how Plan 9 identifies files across the entire system. Every file server assigns each file a unique `path` (a 64-bit number), a `vers` (version, incremented on each modification), and a `type` byte indicating whether it's a directory, append-only, etc. Two files are identical if and only if they have the same `Qid`.

⟨*type* `Open_flag` 190c⟩≡ (379b)

```
// enum Open_flag, open parameter
#define OREAD 0 /* open for read */
#define OWRITE 1 /* write */
#define ORDWR 2 /* read and write */
#define OEXEC 3 /* execute, == read but check execute permission */
// advanced stuff (no O_APPEND, O_CREATE, O_NONBLOCK as in Unix though)
#define OTRUNC 16 /* or'ed in (except for exec), truncate file first */
#define OCEXEC 32 /* or'ed in, close on exec */
#define ORCLOSE 64 /* or'ed in, remove on close */
#define OEXCL 0x1000 /* or'ed in, exclusive use (create only) */
```

⟨*type* `Access_flag` 190d⟩≡ (379b)

```
// enum Access_flag
#define AEXIST 0 /* accessible: exists */
#define AEXEC 1 /* execute access */
#define AWRITE 2 /* write access */
#define AREAD 4 /* read access */
```

⟨*type* Qid 191a⟩≡ (379b)
```
// Qid as in uniQue id
struct Qid {
 uvlong path;
 ulong vers;
    // bitset<Qidtype>
 uchar type;
};
```

⟨*type* Qid_type 191b⟩≡ (379b)
```
/* bits in Qid.type */
#define QTFILE  0x00  /* plain file */
#define QTDIR  0x80  /* type bit for directories */
// advanced stuff
#define QTAPPEND 0x40  /* type bit for append only files */
#define QTEXCL  0x20  /* type bit for exclusive use files */
#define QTMOUNT  0x10  /* type bit for mounted channel */
#define QTAUTH  0x08  /* type bit for authentication file */
#define QTTMP  0x04  /* type bit for not-backed-up file */
```

⟨*type* Seek_cursor 191c⟩≡ (379b)
```
// pad's stuff (but it is actually also in stdio.h)
enum Seek_cursor {
    SEEK__START = 0,
    SEEK__CUR = 1,
    SEEK__END = 2,
};
```

13.2 Syscalls: `open()`, `pread()`, etc

These are the core file I/O syscalls.

⟨*signatures file syscall* 191d⟩≡ (379b)
```
extern fdt open(char*, int);
extern int close(fdt);
extern long pread(fdt, void*, long, vlong);
extern long pwrite(fdt, void*, long, vlong);
extern int dup(int, int);
```

Note that the fundamental I/O syscalls are `pread()` and `pwrite()` ("positional read/write"), not `read()`[192a] and `write()`[192b]. They take an explicit file offset as a parameter, so the kernel does not need to maintain a per-fd offset. This avoids a race condition that plagues UNIX if two processes share a file descriptor (after `fork`), a UNIX `lseek+read` sequence is not atomic—another process can seek in between. With `pread`, the offset is part of the call itself, so no race is possible. (UNIX later adopted `pread/pwrite` too, in POSIX.1-2001, but the traditional `read/write` remain the primary interface.) In `pread(fd, buf, n, offset)`: `fd` is the file descriptor, `buf` the destination buffer, `n` the number of bytes to read, and `offset` the position in the file. Passing offset −1 means "use the current file position" (this is how the library wrappers `read/write` are implemented—see below). `dup(oldfd, newfd)` duplicates `oldfd` onto `newfd`. If `newfd` is −1, the kernel allocates the lowest available descriptor.

In the kernel, `sysopen()` resolves the file path through the namespace via `namec()`, which walks the mount table and device trees to produce a `Chan` (channel)—the kernel's representation of an open file. `sysclose()` decrements the channel's reference count and frees it when it reaches zero. `syspread()/syspwrite()` call the device driver's `read/write` methods with the given offset. `sysdup()` duplicates a file descriptor, optionally replacing an existing one.

13.3 read(), write()

`read()` and `write()` are thin wrappers around `pread()` and `pwrite()` with an offset of −1, meaning "use the current file offset." The positional variants are the real system calls in Plan 9; the traditional `read`/`write` are convenience functions.

⟨*function* `read` 192a⟩≡ (491b)
```
long
read(fdt fd, void *buf, long n)
{
    return pread(fd, buf, n, -1LL);
}
```
Uses `pread()`.

⟨*function* `write` 192b⟩≡ (499d)
```
long
write(fdt fd, void *buf, long n)
{
    return pwrite(fd, buf, n, -1LL);
}
```
Uses `pwrite()`.

13.4 readn()

`readn()` loops until exactly `n` bytes have been read or EOF is reached. This is necessary because `read` may return fewer bytes than requested (a "short read"), especially when reading from pipes or network connections. `readn` guarantees the caller gets the full amount.

⟨*function* `readn` 192c⟩≡ (435c)
```
long
readn(fdt f, void *av, long n)
{
    char *a;
    long m, t;

    a = av;
    t = 0;
    while(t < n){
        m = read(f, a+t, n-t);
        if(m <= 0){
            if(t == 0)
                return m;
            break;
        }
        t += m;
    }
    return t;
}
```
Uses `read()` 192a.

13.5 Advanced topics

13.5.1 Scatter/gather I/O

The `IOchunk` structure describes a single contiguous memory region (address and length) for scatter/gather I/O, where a single `readv`/`writev` call transfers data to or from multiple non-contiguous buffers.

⟨*type* `IOchunk` 193⟩≡ (379b)

```
struct IOchunk {
 void *addr;
 ulong len;
};
```

Chapter 14

Buffered IO

`libbio` (declared in `bio.h`) provides buffered I/O—the name stands for "Buffered I/O," and `Biobuf`[195a] for "Buffered I/O buffer." It fills the same role as UNIX's `stdio`, but with a cleaner interface: there is no `FILE *` with hidden global state, no confusion between text and binary modes, no `ungetc` limited to a single byte, and the API is small enough to learn in an afternoon. A `Biobuf` wraps a file descriptor with an 8 KB buffer, providing efficient character-at-a-time reading (`Bgetc()`[200a], `Bgetrune()`[201a]), line reading (`Brdline()`[202]), and formatted output (`Bprint()`[209a]). Unlike `stdio`, `libbio` is designed for Plan 9's UTF-8 strings from the ground up.

A typical use looks like this:

```
Biobuf bin;
char *line;

Binit(&bin, 0, OREAD);       /* wrap stdin */
while(line = Brdline(&bin, '\n')) {
    line[Blinelen(&bin)-1] = '\0';
    print("%s\n", line);
}
Bterm(&bin);
```

⟨*signatures* Bxxx *functions* 194⟩≡ (382e)

```
int Binit(Biobuf*, fdt, int);
Biobuf* Bopen(char*, int);
int Bterm(Biobufhdr*);

int Bgetc(BiobufGen*);
int Bungetc(Biobufhdr*);
long Bread(Biobufhdr*, void*, long);

int Bputc(BiobufGen*, int);
long Bwrite(BiobufGen*, void*, long);
int Bprint(BiobufGen*, char*, ...);

#pragma varargck argpos Bprint 2

void* Brdline(Biobufhdr*, int);
int Blinelen(Biobufhdr*);
```

14.1 Data structures

A `Biobuf` is a structure that embeds a `Biobufhdr`[195c] (the "header" with all the bookkeeping fields) followed by a fixed-size byte array `b` that serves as the actual buffer.

⟨*struct* Biobuf 195a⟩≡ (382e)
```
struct Biobuf
{
    Biobufhdr;
    uchar b[Bungetsize+Bsize];
};
```

The buffer is `Bsize` (8 KB) plus `Bungetsize` extra bytes. The extra bytes—`UTFmax+1`, i.e. 4 bytes—sit at the front and serve as a small "unget zone": when `Bgetc()`[200a] refills the buffer, it copies the last few bytes of the old buffer into this zone so that `Bungetc()`[200b] can back up past a buffer boundary. Without this, ungetting a multi-byte UTF-8 rune that straddled two buffer fills would fail.

⟨*constants Bxxx* 195b⟩≡ (382e) 195d ▷
```
Bsize   = 8*1024,
Bungetsize = UTFmax+1,  /* space for ungetc */
```

A `Biobuf` is designed to be stack-allocated; for heap allocation, use `Bopen()`[198d] instead.

At its core, a `Biobufhdr` is a wrapper around a file descriptor (`fid`). Instead of calling `read` and `write` directly on the descriptor, the caller goes through `libbio` functions which batch small operations into buffer-sized chunks. The header tracks everything needed to manage that buffer: the current state (reading, writing, or inactive), the buffer boundaries, and counters that record how much data is available or how much space remains.

⟨*struct* Biobufhdr 195c⟩≡ (382e)
```
struct Biobufhdr
{
    fdt fid;  /* open file */
    // enum<Bkind>
    int state;  /* r/w/inactive */

    vlong offset;  /* offset of buffer in file */
    int bsize;  /* size of buffer */

    ⟨Biobufhdr count fields 196a⟩
    ⟨Biobufhdr buffer fields 196b⟩
    ⟨Biobufhdr other fields 196c⟩
};
```

A `Biobuf` is always in one of four states. `Binactive` means the buffer has hit an error and is dead. `Bractive` and `Bwactive` are the normal reading and writing states.

⟨*constants Bxxx* 195d⟩+≡ (382e) ◁ 195b 199a ▷
```
Binactive = 0,  /* states */
Bractive,
Bwactive,
Bracteof,
```

`Bracteof` is a transitional state that enables rereading after end-of-file. Consider `tail -f`: it reads to the end of a file, sees EOF, then sleeps and tries again. Without `Bracteof`, the buffer would be stuck—`Bgetc()` would keep returning `Beof` forever. Instead, the first `Bgetc()` after EOF returns `Beof` *and* transitions the state back to `Bractive`, so the next `Bgetc()` will try a fresh `read` system call. If the file has grown in the meantime, the new data is picked up.

The header contains the real machinery. The key insight is that `icount` and `ocount` are *negative* counters: `icount` counts up from `-n` toward zero as bytes are consumed from the read buffer, while `ocount` counts up from

-bsize toward zero as bytes are added to the write buffer. When either reaches zero, it's time to refill or flush. This lets the fast path—Bgetc() and Bputc()[207a]—be a single increment-and-index with no comparisons.

⟨Biobufhdr *count fields* 196a⟩≡ (195c)
```
int icount;  /* neg num of bytes at eob */
int ocount;  /* num of bytes at bob */
```

The three pointers bbuf, ebuf, and gbuf partition the byte array. bbuf and ebuf mark the full extent of the usable buffer (after the unget zone). gbuf ("good buffer") marks where valid data starts.

⟨Biobufhdr *buffer fields* 196b⟩≡ (195c)
```
// point in Biobuf.b
uchar* bbuf;  /* pointer to beginning of buffer */
uchar* ebuf;  /* pointer to end of buffer */

uchar* gbuf;  /* pointer to good data in buf */
```

Here is the buffer layout after a read that returned 3000 bytes into an 8192-byte buffer. The data is right-aligned against ebuf:

```
Biobuf.b[]
|<-unget->|<---------- bsize = 8192 ----------->|
[_ _ _ _ _|_ _ _ _ _ _ _ _ _ _ _|d d d d d d d d]
^          ^                     ^    ^          ^
|          bbuf                  gbuf |          ebuf
|                                     |
b (start of array)              ebuf + icount
                                (next byte to return)

icount = -3000  (3000 unread bytes)
offset = 3000   (file position after read)
```

For writing, ocount works symmetrically. After writing 2000 bytes into a fresh buffer:

```
[_ _ _ _ _|w w w w w w _ _ _ _ _ _ _ _ _ _ _ _ _ _]
^          ^           ^                          ^
|          bbuf        ebuf + ocount              ebuf
|
b          |<-written->|<------- free ---------->|

ocount = -6192  (6192 bytes free, started at -8192)
```

⟨Biobufhdr *other fields* 196c⟩≡ (195c) 199b ▷
```
int rdline;   /* num of bytes after rdline */
int runesize; /* num of bytes of last getrune */
```

14.2 Initialization: Binit(), Bopen(), Bterm()

There are two ways to create a Biobuf: Binit()[197a] wraps an already-open file descriptor (the Biobuf is usually on the stack), while Bopen()[198d] opens the file and heap-allocates the buffer. In both cases the real work is done by Binits()[197b], which takes an explicit buffer pointer and size.

Binit() is a thin wrapper that passes the embedded b array to Binits():

⟨*function* Binit 197a⟩≡ (555a)
```
int
Binit(Biobuf *bp, fdt f, int mode)
{
    return Binits(bp, f, mode, bp->b, sizeof(bp->b));
}
```
Uses Binits() 197b.

Binits() first reserves Bungetsize bytes at the front of the buffer for Bungetc()[200b] support, then sets the state based on the mode. For write buffers, ocount starts at -size (the entire buffer is free); for read buffers, icount starts at zero (no data available yet, so the first Bgetc()[200a] will trigger a read system call). Write buffers are registered via install()[198a] for automatic flushing at exit.

⟨*function* Binits 197b⟩≡ (555a)
```
int
Binits(Biobufhdr *bp, fdt f, int mode, uchar *p, int size)
{

    p += Bungetsize; /* make room for Bungets */
    size -= Bungetsize;

    switch(mode&~(OCEXEC|ORCLOSE|OTRUNC)) {
    case OREAD:
        bp->state = Bractive;
        bp->ocount = 0;
        break;
    case OWRITE:
        install(bp);
        bp->state = Bwactive;
        bp->ocount = -size;
        break;
    default:
        fprint(2, "Binits: unknown mode %d\n", mode);
        return Beof;

    }
    bp->bbuf = p;
    bp->ebuf = p+size;
    bp->bsize = size;
    bp->icount = 0;
    bp->gbuf = bp->ebuf;
    bp->fid = f;
    bp->flag = 0;
    bp->rdline = 0;
    bp->offset = 0;
    bp->runesize = 0;
    return 0;
}
```
Uses fprint() 75a and install() 198a.

The atexit machinery ensures that write buffers are flushed when a program exits, even if the programmer forgets to call Bterm()[199c]. A global table wbufs holds up to 20 active write buffers. When the first write buffer is installed, an atexit handler is registered that walks the table and flushes each one.

⟨*global* wbufs 197c⟩≡ (555a)
```
static Biobufhdr* wbufs[20];
```

⟨*global* atexitflag 197d⟩≡ (555a)
```
static bool  atexitflag;
```

⟨*function* install 198a⟩≡ (555a)
```
static
void
install(Biobufhdr *bp)
{
    int i;

    deinstall(bp);
    for(i=0; i<nelem(wbufs); i++)
        if(wbufs[i] == nil) {
            wbufs[i] = bp;
            break;
        }
    if(atexitflag == false) {
        atexitflag = true;
        atexit(batexit);
    }
}
```
Uses `atexit()` 45d, `atexitflag-358` 197d, `batexit()` 198c, `deinstall()` 198b, and `wbufs-357` 197c.

⟨*function* deinstall 198b⟩≡ (555a)
```
static
void
deinstall(Biobufhdr *bp)
{
    int i;

    for(i=0; i<nelem(wbufs); i++)
        if(wbufs[i] == bp)
            wbufs[i] = nil;
}
```
Uses `wbufs-357` 197c.

⟨*function* batexit 198c⟩≡ (555a)
```
static
void
batexit(void)
{
    Biobufhdr *bp;
    int i;

    for(i=0; i<nelem(wbufs); i++) {
        bp = wbufs[i];
        if(bp != nil) {
            wbufs[i] = nil;
            Bflush(bp);
        }
    }
}
```
Uses `Bflush()` 210a and `wbufs-357` 197c.

`Bopen()` is the high-level constructor: it opens (or creates) the file, `malloc`s a `Biobuf`, and marks it with `Bmagic` so that `Bterm()` knows to `close` the descriptor and `free` the memory. If the caller used `Binit()` instead, `Bterm()` just flushes without freeing—the caller owns the storage.

⟨*function* Bopen 198d⟩≡ (555a)
```
Biobuf*
Bopen(char *name, int mode)
{
    Biobuf *bp;
```

```
    fdt f;

    switch(mode&~(OCEXEC|ORCLOSE|OTRUNC)) {
    case OREAD:
        f = open(name, mode);
        break;
    case OWRITE:
        f = create(name, mode, 0666);
        break;
    default:
        fprint(2, "Bopen: unknown mode %#x\n", mode);
        return 0;
    }
    if(f < 0)
        return nil;

    bp = malloc(sizeof(Biobuf));
    Binits(bp, f, mode, bp->b, sizeof(bp->b));
    bp->flag = Bmagic;   /* mark bp open & malloced */
    return bp;
}
```

Uses `Binits()` 197b, `create()`, `fprint()` 75a, `malloc()` 63g, and `open()`.

⟨*constants Bxxx* 199a⟩+≡ (382e) ◁195d 382d▷

```
Bmagic  = 0x314159,
```

⟨`Biobufhdr` *other fields* 199b⟩+≡ (195c) ◁196c

```
int flag;  /* magic if malloc'ed */
```

`Bterm()` tears down a `Biobuf`: it deinstalls it from the atexit table, flushes any pending writes, and—only if the buffer was heap-allocated by `Bopen()`—closes the file descriptor and frees the memory. Setting `fid` to -1 after close is a defensive touch that makes use-after-free produce an obvious `write` error rather than silently corrupting another file.

⟨*function* `Bterm` 199c⟩≡ (555a)

```
int
Bterm(Biobufhdr *bp)
{
    int r;

    deinstall(bp);
    r = Bflush(bp);
    if(bp->flag == Bmagic) {
        bp->flag = 0;
        close(bp->fid);
        bp->fid = -1;   /* prevent accidents */
        free(bp);
    }
    /* otherwise opened with Binit(s) */
    return r;
}
```

Uses `Bflush()` 210a, `close()`, `deinstall()` 198b, and `free()` 63i.

14.3 Reading

`libbio` offers reading at three granularities: byte (`Bgetc()`[200a]), rune (`Bgetrune()`[201a]), and line (`Brdline()`[202], `Brdstr()`[204b]). All share the same underlying buffer; the differences are in how much they consume per call and whether they return a pointer into the buffer (zero-copy) or allocate.

14.3.1 Byte reading: `Bgetc()`, `Bungetc()`

`Bgetc()`[200a] is the workhorse. The fast path is two instructions: if `icount` is non-zero, increment it and index `ebuf`. Since `icount` is negative and counts toward zero, the expression `ebuf[icount]` walks backward from the end of the buffer—this is why short reads are right-aligned against `ebuf`.

When `icount` reaches zero the buffer is exhausted. Before calling `read`, the last `Bungetsize` bytes from the old buffer are copied to just before `bbuf`, so that a subsequent `Bungetc()`[200b] can back up past the buffer boundary. If the `read` returns fewer than `bsize` bytes, the data is shifted to the end of the buffer (right-aligned) and `gbuf` is adjusted, maintaining the invariant that `ebuf[icount]` always points to the next byte.

⟨*function* `Bgetc` 200a⟩≡ (554a)

```
int
Bgetc(Biobufhdr *bp)
{
    int i;

loop:
    i = bp->icount;
    if(i != 0) {
        bp->icount = i+1;
        return bp->ebuf[i];
    }
    if(bp->state != Bractive) {
        if(bp->state == Bracteof)
            bp->state = Bractive;
        return Beof;
    }
    /*
     * get next buffer, try to keep Bungetsize
     * characters pre-catenated from the previous
     * buffer to allow that many ungets.
     */
    memmove(bp->bbuf-Bungetsize, bp->ebuf-Bungetsize, Bungetsize);
    i = read(bp->fid, bp->bbuf, bp->bsize);
    bp->gbuf = bp->bbuf;
    if(i <= 0) {
        bp->state = Bracteof;
        if(i < 0)
            bp->state = Binactive;
        return Beof;
    }
    if(i < bp->bsize) {
        memmove(bp->ebuf-i-Bungetsize, bp->bbuf-Bungetsize, i+Bungetsize);
        bp->gbuf = bp->ebuf-i;
    }
    bp->icount = -i;
    bp->offset += i;
    goto loop;
}
```

Uses `memmove()` 48 and `read()` 192a.

`Bungetc()` simply decrements `icount`, moving the read cursor back by one byte. This works even across buffer boundaries thanks to the `Bungetsize` bytes preserved during refill.

⟨*function* `Bungetc` 200b⟩≡ (554a)

```
int
Bungetc(Biobufhdr *bp)
{

    if(bp->state == Bracteof)
```

```
        bp->state = Bractive;
    if(bp->state != Bractive)
        return Beof;
    bp->icount--;
    return 1;
}
```

14.3.2 Rune reading: Bgetrune(), Bungetrune()

`Bgetrune()`[201a] reads a full UTF-8 sequence by calling `Bgetc()`[200a] one byte at a time. If the first byte is below `Runeself` (128), it's ASCII and we're done. Otherwise, bytes are accumulated until `fullrune()` says we have a complete sequence, then `chartorune()` decodes it. Any excess bytes consumed are pushed back with `Bungetc()`[200b]. The `runesize` field records how many bytes the rune occupied, which `Bungetrune()`[201b] needs to know how far to back up.

⟨*function* `Bgetrune` 201a⟩≡ (554e)

```
long
Bgetrune(Biobufhdr *bp)
{
    int c, i;
    Rune rune;
    char str[UTFmax];

    c = Bgetc(bp);
    if(c < Runeself) {  /* one char */
        bp->runesize = 1;
        return c;
    }
    str[0] = c;
    bp->runesize = 0;

    for(i=1;;) {
        c = Bgetc(bp);
        if(c < 0)
            return c;
        if (i >= sizeof str)
            return Runeerror;
        str[i++] = c;

        if(fullrune(str, i)) {
            /* utf is long enough to be a rune, but could be bad. */
            bp->runesize = chartorune(&rune, str);
            if (rune == Runeerror)
                bp->runesize = 0; /* push back nothing */
            else
                /* push back bytes unconsumed by chartorune */
                for(; i > bp->runesize; i--)
                    Bungetc(bp);
            return rune;
        }
    }
}
```

Uses `Bgetc()` 200a, `Bungetc()` 200b, `chartorune()` 84d, and `fullrune()` 436c.

`Bungetrune()` backs up by `runesize` bytes rather than one, undoing exactly the multi-byte sequence that `Bgetrune()` consumed. It then clears `runesize` to prevent double-ungets.

⟨*function* `Bungetrune` 201b⟩≡ (554e)

```
int
```

```
Bungetrune(Biobufhdr *bp)
{

    if(bp->state == Bracteof)
        bp->state = Bractive;
    if(bp->state != Bractive)
        return Beof;
    bp->icount -= bp->runesize;
    bp->runesize = 0;
    return 1;
}
```

14.3.3 Line reading: `Brdline()`

`Brdline()`[202] is the most complex function in `libbio`. It searches for a delimiter (usually `\n`) and returns a pointer directly into the buffer—no allocation, no copy. The caller must process the data before the next `libbio` call, since the pointer is invalidated by any operation that refills the buffer. The length of the returned line (including the delimiter) is available via `Blinelen()`[212c].

The algorithm has three phases. First, it scans the unread portion of the current buffer with `memchr()`[54]. If the delimiter is found, it returns immediately—this is the fast path that handles most lines. If not, the unconsumed data is moved to the front of the buffer (`bbuf`), and more data is read in behind it. Each time new data arrives, only the new portion is scanned. If the delimiter is found, the accumulated data is moved back to the end of the buffer (to maintain the right-alignment invariant) and a pointer is returned. If the buffer fills completely without finding the delimiter, `Brdline()` returns `nil`—the line is simply too long. The caller can detect this by checking `Blinelen()`, which will equal `bsize`.

⟨*function* `Brdline` 202⟩≡ (555f)

```
void*
Brdline(Biobufhdr *bp, int delim)
{
    char *ip, *ep;
    int i, j;

    i = -bp->icount;
    if(i == 0) {
        /*
         * eof or other error
         */
        if(bp->state != Bractive) {
            if(bp->state == Bracteof)
                bp->state = Bractive;
            bp->rdline = 0;
            bp->gbuf = bp->ebuf;
            return 0;
        }
    }

    /*
     * first try in remainder of buffer (gbuf doesn't change)
     */
    ip = (char*)bp->ebuf - i;
    ep = memchr(ip, delim, i);
    if(ep) {
        j = (ep - ip) + 1;
        bp->rdline = j;
        bp->icount += j;
        return ip;
```

```
    }

    /*
     * copy data to beginning of buffer
     */
    if(i < bp->bsize)
        memmove(bp->bbuf, ip, i);
    bp->gbuf = bp->bbuf;

    /*
     * append to buffer looking for the delim
     */
    ip = (char*)bp->bbuf + i;
    while(i < bp->bsize) {
        j = read(bp->fid, ip, bp->bsize-i);
        if(j <= 0) {
            /*
             * end of file with no delim
             */
            memmove(bp->ebuf-i, bp->bbuf, i);
            bp->rdline = i;
            bp->icount = -i;
            bp->gbuf = bp->ebuf-i;
            return 0;
        }
        bp->offset += j;
        i += j;
        ep = memchr(ip, delim, j);
        if(ep) {
            /*
             * found in new piece
             * copy back up and reset everything
             */
            ip = (char*)bp->ebuf - i;
            if(i < bp->bsize){
                memmove(ip, bp->bbuf, i);
                bp->gbuf = (uchar*)ip;
            }
            j = (ep - (char*)bp->bbuf) + 1;
            bp->rdline = j;
            bp->icount = j - i;
            return ip;
        }
        ip += j;
    }

    /*
     * full buffer without finding
     */
    bp->rdline = bp->bsize;
    bp->icount = -bp->bsize;
    bp->gbuf = bp->bbuf;
    return 0;
}
```
Uses `memchr()` 54, `memmove()` 48, and `read()` 192a.

14.3.4 Line reading (allocating): Brdstr()

`Brdstr()`[204b] solves the limitation of `Brdline()`[202]: it handles lines of arbitrary length by `malloc`ing a string that the caller must eventually `free`. The `nulldelim` parameter controls whether the delimiter is included in the returned string—if true, the trailing delimiter is replaced with `\0`.

The helper `badd()`[204a] appends data to a growable string via `realloc`. When the line fits within one buffer, `Brdstr()` calls `badd()` once and returns. For lines longer than the buffer, it loops: each time the buffer fills without finding the delimiter, the buffer contents are appended to the accumulator string and a new buffer is read. This means `Brdstr()` can handle arbitrarily long lines at the cost of allocation, while `Brdline()` is zero-copy but bounded by buffer size.

⟨*function* badd 204a⟩≡ (556a)

```
static char*
badd(char *p, int *np, char *data, int ndata, int delim, int nulldelim)
{
    int n;

    n = *np;
    p = realloc(p, n+ndata+1);
    if(p){
        memmove(p+n, data, ndata);
        n += ndata;
        if(n>0 && nulldelim && p[n-1]==delim)
            p[--n] = '\0';
        else
            p[n] = '\0';
        *np = n;
    }
    return p;
}
```

Uses `memmove()` 48 and `realloc()` 67b.

⟨*function* Brdstr 204b⟩≡ (556a)

```
char*
Brdstr(Biobufhdr *bp, int delim, int nulldelim)
{
    char *ip, *ep, *p;
    int i, j;

    i = -bp->icount;
    bp->rdline = 0;
    if(i == 0) {
        /*
         * eof or other error
         */
        if(bp->state != Bractive) {
            if(bp->state == Bracteof)
                bp->state = Bractive;
            bp->gbuf = bp->ebuf;
            return nil;
        }
    }

    /*
     * first try in remainder of buffer (gbuf doesn't change)
     */
    ip = (char*)bp->ebuf - i;
    ep = memchr(ip, delim, i);
    if(ep) {
```

```
    j = (ep - ip) + 1;
    bp->icount += j;
    return badd(nil, &bp->rdline, ip, j, delim, nulldelim);
}

/*
 * copy data to beginning of buffer
 */
if(i < bp->bsize)
    memmove(bp->bbuf, ip, i);
bp->gbuf = bp->bbuf;

/*
 * append to buffer looking for the delim
 */
p = nil;
for(;;){
    ip = (char*)bp->bbuf + i;
    while(i < bp->bsize) {
        j = read(bp->fid, ip, bp->bsize-i);
        if(j <= 0 && i == 0)
            return p;
        if(j <= 0 && i > 0){
            /*
             * end of file but no delim. pretend we got a delim
             * by making the delim \0 and smashing it with nulldelim.
             */
            j = 1;
            ep = ip;
            delim = '\0';
            nulldelim = 1;
            *ep = delim; /* there will be room for this */
        }else{
            bp->offset += j;
            ep = memchr(ip, delim, j);
        }
        i += j;
        if(ep) {
            /*
             * found in new piece
             * copy back up and reset everything
             */
            ip = (char*)bp->ebuf - i;
            if(i < bp->bsize){
                memmove(ip, bp->bbuf, i);
                bp->gbuf = (uchar*)ip;
            }
            j = (ep - (char*)bp->bbuf) + 1;
            bp->icount = j - i;
            return badd(p, &bp->rdline, ip, j, delim, nulldelim);
        }
        ip += j;
    }

    /*
     * full buffer without finding; add to user string and continue
     */
    p = badd(p, &bp->rdline, (char*)bp->bbuf, bp->bsize, 0, 0);
    i = 0;
    bp->icount = 0;
```

```
        bp->gbuf = bp->ebuf;
    }
}
```

Uses badd() 204a, memchr() 54, memmove() 48, and read() 192a.

14.3.5 Bulk reading: Bread()

Bread()[206] reads an exact number of bytes into a caller-supplied buffer, similar to UNIX's fread. It first drains whatever is left in the Biobuf's internal buffer, then issues read calls for any remaining data. Like Bgetc()[200a], it handles short reads and right-aligns partial buffers. The return value is the number of bytes actually read, or −1 on hard error with no data delivered.

⟨*function* Bread 206⟩≡ (556b)

```
long
Bread(Biobufhdr *bp, void *ap, long count)
{
    long c;
    uchar *p;
    int i, n, ic;

    p = ap;
    c = count;
    ic = bp->icount;

    while(c > 0) {
        n = -ic;
        if(n > c)
            n = c;
        if(n == 0) {
            if(bp->state != Bractive)
                break;
            i = read(bp->fid, bp->bbuf, bp->bsize);
            if(i <= 0) {
                bp->state = Bracteof;
                if(i < 0)
                    bp->state = Binactive;
                break;
            }
            bp->gbuf = bp->bbuf;
            bp->offset += i;
            if(i < bp->bsize) {
                memmove(bp->ebuf-i, bp->bbuf, i);
                bp->gbuf = bp->ebuf-i;
            }
            ic = -i;
            continue;
        }
        memmove(p, bp->ebuf+ic, n);
        c -= n;
        ic += n;
        p += n;
    }
    bp->icount = ic;
    if(count == c && bp->state == Binactive)
        return -1;
    return count-c;
}
```

Uses memmove() 48 and read() 192a.

14.4 Writing

The write side mirrors the read side: `ocount` is a negative counter that starts at `-bsize` and increments toward zero. The expression `ebuf[ocount]` walks backward from `ebuf`, filling the buffer from `bbuf` toward `ebuf`. When `ocount` reaches zero the buffer is full and `Bflush()`[210a] writes it out.

14.4.1 Byte writing: `Bputc()`

`Bputc()`[207a] is the write-side counterpart to `Bgetc()`[200a]: store the byte at `ebuf[ocount]` and increment. If `ocount` is zero (buffer full), flush first. The `for(;;)` loop retries after a successful flush; if flush fails, it returns `Beof`.

⟨*function* `Bputc` 207a⟩≡ (555d)

```
int
Bputc(Biobufhdr *bp, int c)
{
    int i;

    for(;;) {
        i = bp->ocount;
        if(i) {
            bp->ebuf[i++] = c;
            bp->ocount = i;
            return 0;
        }
        if(Bflush(bp) == Beof)
            break;
    }
    return Beof;
}
```

Uses `Bflush()` 210a.

14.4.2 Rune writing: `Bputrune()`

`Bputrune()`[207b] encodes a rune into UTF-8 and writes the bytes. ASCII runes (below `Runeself`) take the fast path through `Bputc()`[207a]. Multi-byte runes are encoded into a temporary buffer with `runetochar()`[85] and written with `Bwrite()`[208].

⟨*function* `Bputrune` 207b⟩≡ (555e)

```
int
Bputrune(Biobufhdr *bp, long c)
{
    Rune rune;
    char str[UTFmax];
    int n;

    rune = c;
    if(rune < Runeself) {
        Bputc(bp, rune);
        return 1;
    }
    n = runetochar(str, &rune);
    if(n == 0)
        return Bbad;
    if(Bwrite(bp, str, n) != n)
        return Beof;
    return n;
```

```
}
```
Uses Bputc() 207a, Bwrite() 208, and runetochar() 85.

14.4.3 Bulk writing: Bwrite()

Bwrite()[208] is the write-side counterpart to Bread()[206]: it copies data into the internal buffer, flushing whenever the buffer fills. One detail worth noting is the interrupt handling: if a write system call fails, Bwrite() checks whether the error string contains "interrupt." If so, the buffer stays active—the write was interrupted by a note (signal) but the file descriptor is still good. Any other error kills the buffer by setting Binactive.

⟨*function* Bwrite 208⟩≡ (557a)
```
long
Bwrite(Biobufhdr *bp, void *ap, long count)
{
    long c;
    uchar *p;
    int i, n, oc;
    char errbuf[ERRMAX];

    p = ap;
    c = count;
    oc = bp->ocount;

    while(c > 0) {
        n = -oc;
        if(n > c)
            n = c;
        if(n == 0) {
            if(bp->state != Bwactive)
                return Beof;
            i = write(bp->fid, bp->bbuf, bp->bsize);
            if(i != bp->bsize) {
                errstr(errbuf, sizeof errbuf);
                if(strstr(errbuf, "interrupt") == nil)
                    bp->state = Binactive;
                errstr(errbuf, sizeof errbuf);
                return Beof;
            }
            bp->offset += i;
            oc = -bp->bsize;
            continue;
        }
        memmove(bp->ebuf+oc, p, n);
        oc += n;
        c -= n;
        p += n;
    }
    bp->ocount = oc;
    return count-c;
}
```
Uses errstr(), memmove() 48, strstr() 82b, and write() 192b.

14.4.4 Formatted writing: Bprint(), Bvprint()

Bprint()[209a] is the libbio equivalent of fprintf: formatted output directly to a buffered file. It delegates to Bvprint()[209c], which plugs the Biobuf's write buffer into the fmt library's Fmt[71b] structure. The key is the

fmtBflush callback: when dofmt fills the available space, fmtBflush calls Bflush()[210a] to write the buffer to disk and resets the format output pointers. This means Bprint() can format strings larger than the buffer without any temporary allocation—the fmt library and libbio cooperate to stream the output.

⟨*function* Bprint 209a⟩≡ (555c)
```
int
Bprint(Biobufhdr *bp, char *fmt, ...)
{
    va_list arg;
    int n;

    va_start(arg, fmt);
    n = Bvprint(bp, fmt, arg);
    va_end(arg);
    return n;
}
```
Uses Bvprint() 209c.

⟨*function* fmtBflush 209b⟩≡ (556f)
```
static int
fmtBflush(Fmt *f)
{
    Biobufhdr *bp;

    bp = f->farg;
    bp->ocount = (char*)f->to - (char*)f->stop;
    if(Bflush(bp) < 0)
        return 0;
    f->stop = bp->ebuf;
    f->to = (char*)f->stop + bp->ocount;
    f->start = f->to;
    return 1;
}
```
Uses Bflush() 210a.

⟨*function* Bvprint 209c⟩≡ (556f)
```
int
Bvprint(Biobufhdr *bp, char *fmt, va_list arg)
{
    int n;
    Fmt f;

    f.runes = 0;
    f.stop = bp->ebuf;
    f.start = (char*)f.stop + bp->ocount;
    f.to = f.start;
    f.flush = fmtBflush;
    f.farg = bp;
    f.nfmt = 0;
    f.args = arg;
    n = dofmt(&f, fmt);
    bp->ocount = (char*)f.to - (char*)f.stop;
    return n;
}
```
Uses dofmt() 74 and fmtBflush() 209b.

14.4.5 Flushing: `Bflush()`

`Bflush()`[210a] has different behavior depending on the buffer's state. For write buffers (`Bwactive`), it computes how many bytes are pending (`bsize + ocount`, since `ocount` is negative), writes them, and resets `ocount` to `-bsize`. A short write kills the buffer. For read buffers, flushing discards any buffered data by resetting `icount` and `gbuf`—this is used by `Bseek()`[210b] to invalidate the buffer before seeking.

⟨*function* `Bflush` 210a⟩≡ (553d)

```
int
Bflush(Biobufhdr *bp)
{
    int n, c;

    switch(bp->state) {
    case Bwactive:
        n = bp->bsize+bp->ocount;
        if(n == 0)
            return 0;
        c = write(bp->fid, bp->bbuf, n);
        if(n == c) {
            bp->offset += n;
            bp->ocount = -bp->bsize;
            return 0;
        }
        bp->state = Binactive;
        bp->ocount = 0;
        break;

    case Bracteof:
        bp->state = Bractive;

    case Bractive:
        bp->icount = 0;
        bp->gbuf = bp->ebuf;
        return 0;
    }
    return Beof;
}
```

Uses `write()` 192b.

14.5 Seeking and position: `Bseek()`, `Boffset()`

`Bseek()`[210b] attempts to satisfy seeks without a system call when possible. For absolute seeks (`base == 0`) within a read buffer, it adjusts `icount` and checks whether the new position falls within the currently buffered data—if so, no `seek` system call is needed. Relative seeks (`base == 1`) are converted to absolute by adding `Boffset()`[211]. For write buffers, `Bseek()` flushes before seeking.

⟨*function* `Bseek` 210b⟩≡ (556c)

```
vlong
Bseek(Biobufhdr *bp, vlong offset, int base)
{
    vlong n, d;

    switch(bp->state) {
    default:
        fprint(2, "Bseek: unknown state %d\n", bp->state);
        return Beof;
```

```
    case Bracteof:
        bp->state = Bractive;
        bp->icount = 0;
        bp->gbuf = bp->ebuf;

    case Bractive:
        n = offset;
        if(base == 1) {
            n += Boffset(bp);
            base = 0;
        }

        /*
         * try to seek within buffer
         */
        if(base == 0) {
            /*
             * if d is too large for an int, icount may wrap,
             * so we need to ensure that icount hasn't wrapped
             * and points within the buffer's valid data.
             */
            d = n - Boffset(bp);
            bp->icount += d;
            if(d <= bp->bsize && bp->icount <= 0 &&
                bp->ebuf - bp->gbuf >= -bp->icount)
                return n;
        }

        /*
         * reset the buffer
         */
        n = seek(bp->fid, n, base);
        bp->icount = 0;
        bp->gbuf = bp->ebuf;
        break;

    case Bwactive:
        Bflush(bp);
        n = seek(bp->fid, offset, base);
        break;
    }
    bp->offset = n;
    return n;
}
```
Uses `Bflush()` 210a, `Boffset()` 211, `fprint()` 75a, and `seek()`.

`Boffset()` returns the current logical file position—the position the caller thinks they're at, accounting for buffered but unconsumed data. For read buffers this is `offset + icount` (the file offset minus unread bytes); for write buffers it's `offset + bsize + ocount` (the file offset plus buffered writes not yet flushed).

⟨*function* `Boffset` 211⟩≡ (555b)
```
vlong
Boffset(Biobufhdr *bp)
{
    vlong n;

    switch(bp->state) {
    default:
        fprint(2, "Boffset: unknown state %d\n", bp->state);
        n = Beof;
```

```
        break;

    case Bracteof:
    case Bractive:
        n = bp->offset + bp->icount;
        break;

    case Bwactive:
        n = bp->offset + (bp->bsize + bp->ocount);
        break;
    }
    return n;
}
```

Uses `fprint()` 75a.

14.6 Helpers: `Bbuffered()`, `Bfildes()`, `Blinelen()`

These are simple accessors. `Bbuffered()`[212a] returns how many bytes are sitting in the buffer (unread for read buffers, unwritten for write buffers). `Bfildes()`[212b] returns the underlying file descriptor. `Blinelen()`[212c] returns the length of the last line returned by `Brdline()`[202] or `Brdstr()`[204b], including the delimiter.

⟨*function* `Bbuffered` 212a⟩≡ (553b)

```
int
Bbuffered(Biobufhdr *bp)
{
    switch(bp->state) {
    case Bracteof:
    case Bractive:
        return -bp->icount;

    case Bwactive:
        return bp->bsize + bp->ocount;

    case Binactive:
        return 0;
    }
    fprint(2, "Bbuffered: unknown state %d\n", bp->state);
    return 0;
}
```

Uses `fprint()` 75a.

⟨*function* `Bfildes` 212b⟩≡ (553c)

```
fdt
Bfildes(Biobufhdr *bp)
{

    return bp->fid;
}
```

⟨*function* `Blinelen` 212c⟩≡ (555f)

```
int
Blinelen(Biobufhdr *bp)
{

    return bp->rdline;
}
```

Chapter 15

Directories and File Paths

The `Dir` structure is Plan 9's file metadata representation, equivalent to UNIX's `struct stat` but richer: it includes string fields for owner (`uid`), group (`gid`), and last modifier (`muid`), plus the `Qid` for unique identification. The `dirstat`/`dirwstat` functions convert between the kernel's wire format and this C structure using `convM2D` and `convD2M`.

⟨*signatures directory syscall wrapper* 213a⟩≡ (379b)

```
extern Dir* dirfstat(fdt);
extern Dir* dirstat(char*);
extern int dirfwstat(int, Dir*);
extern int dirwstat(char*, Dir*);

extern long dirread(int, Dir**);
extern void nulldir(Dir*);
extern long dirreadall(int, Dir**);

extern  char*   getwd(char*, int);
extern int access(char*, int); // ???
extern  char*   mktemp(char*);
```

15.1 Data structures: `Dir`

⟨*type* `DirEntry` 213b⟩≡ (379b)

```
// a similar structure is defined in the kernel!
struct Dir {
 /* system-modified data */
 ushort type; /* server type */
 uint dev; /* server subtype */

 /* file data */
 Qid qid; /* unique id from server */

 ulong mode; /* permissions */
 ulong atime; /* last read time */
 ulong mtime; /* last write time */
 vlong length; /* file length */
 char *name; /* last element of path */

 char *uid; /* owner name */
 char *gid; /* group name */
 char *muid; /* last modifier name */
};
```

⟨*type* Dir_mode 214a⟩≡ (379b)
```
/* bits in Dir.mode */
#define DMDIR  0x80000000 /* mode bit for directories */
#define DMREAD  0x4  /* mode bit for read permission */
#define DMWRITE  0x2  /* mode bit for write permission */
#define DMEXEC  0x1  /* mode bit for execute permission */
// advanced stuff
#define DMAPPEND 0x40000000 /* mode bit for append only files */
#define DMEXCL  0x20000000 /* mode bit for exclusive use files */
#define DMMOUNT  0x10000000 /* mode bit for mounted channel */
#define DMAUTH  0x08000000 /* mode bit for authentication file */
#define DMTMP  0x04000000 /* mode bit for non-backed-up files */
```

⟨*constant* DIRMAX 214b⟩≡ (379b)
```
#define DIRMAX (sizeof(Dir)+STATMAX) /* max length of Dir structure */
```

⟨*constant* STATMAX 214c⟩≡ (379b)
```
#define STATMAX 65535U /* max length of machine-independent stat structure */
```

⟨*function* nulldir 214d⟩≡ (487f)
```
void
nulldir(Dir *d)
{
    memset(d, ~0, sizeof(Dir));
    d->name = d->uid = d->gid = d->muid = "";
}
```
Uses memset() 46b.

nulldir fills a Dir with "don't care" values: all bits set to 1, and empty strings for the name fields. When passed to dirwstat, only fields that differ from this sentinel pattern are actually modified—so you can change just the mode of a file without having to know its current name, owner, or modification time.

15.2 Syscalls: create(), chdir(), etc

These syscalls manage directory entries and file metadata.

⟨*signatures directory syscall* 214e⟩≡ (379b) 214f▷
```
extern int create(char*, int, ulong);
extern int remove(char*);
extern int chdir(char*);
extern int fd2path(fdt, char*, int);
```

⟨*signatures directory syscall* 214f⟩+≡ (379b) ◁214e
```
extern int fstat(int, uchar*, int);
extern int stat(char*, uchar*, int);
extern int fwstat(int, uchar*, int);
extern int wstat(char*, uchar*, int);
```

The stat/fstat syscalls return file metadata in the 9P wire format (a packed binary encoding), and wstat/fwstat modify it. The library wrappers dirstat/dirfstat shown below convert between this wire format and the Dir C structure using convM2D/convD2M.

In the kernel, syscreate() resolves the path via namec() with the Acreate action, which walks to the parent directory and asks the device driver to create the file. sysremove() walks to the file and calls the device's remove method. chdir() changes the process's current directory. fd2path() returns the full path name associated with an open file descriptor.

Plan 9 spells create in full—in UNIX, the syscall was famously creat without the final 'e'. When asked what he would change about UNIX, Ken Thompson answered: "I'd spell creat with an e." fd2path has no

direct UNIX equivalent. In Plan 9, with per-process namespaces, the same file can be reached via different paths in different processes, and `..` is resolved lexically (by `cleanname`, shown later) rather than by following the filesystem's parent pointers. So the kernel remembers the path used to open each file descriptor and `fd2path` returns it. In UNIX, `realpath()` tries to resolve a path to a canonical form, but it does not work correctly with bind mounts and has no connection to file descriptors.

15.3 xxxfstat()

`dirfstat` and `dirstat` allocate a `Dir` structure and fill it by calling the underlying `fstat`/`stat` system calls. The two-pass loop handles the case where the initial buffer is too small: the first call returns the required size, and the second call succeeds with a properly sized buffer. The string fields in `Dir` point into memory allocated immediately after the structure itself, so a single `free` releases everything.

⟨*enum* `_anon_ (9sys/dirfstat.c)` 215a⟩≡ (477)

```
enum
{
    DIRSIZE = STATFIXLEN + 16 * 4       /* enough for encoded stat buf + some reasonable strings */
};
```

⟨*function* `dirfstat` 215b⟩≡ (477)

```
Dir*
dirfstat(int fd)
{
    Dir *d;
    uchar *buf;
    int n, nd, i;

    nd = DIRSIZE;
    for(i=0; i<2; i++){ /* should work by the second try */
        d = malloc(sizeof(Dir) + BIT16SZ + nd);
        if(d == nil)
            return nil;
        buf = (uchar*)&d[1];
        n = fstat(fd, buf, BIT16SZ+nd);
        if(n < BIT16SZ){
            free(d);
            return nil;
        }
        nd = GBIT16(buf);   /* upper bound on size of Dir + strings */
        if(nd <= n){
            convM2D(buf, n, d, (char*)&d[1]);
            return d;
        }
        /* else sizeof(Dir)+BIT16SZ+nd is plenty */
        free(d);
    }
    return nil;
}
```

Uses `DIRSIZE-237` 215a, `convM2D()` 280, `free()` 63i, `fstat()`, and `malloc()` 63g.

⟨*function* `dirfwstat` 215c⟩≡ (478a)

```
int
dirfwstat(int fd, Dir *d)
{
    uchar *buf;
    int r;
```

```
    r = sizeD2M(d);
    buf = malloc(r);
    if(buf == nil)
        return -1;
    convD2M(d, buf, r);
    r = fwstat(fd, buf, r);
    free(buf);
    return r;
}
```

Uses convD2M() 281, free() 63i, fwstat(), malloc() 63g, and sizeD2M() 465a.

15.4 xxxstat()

⟨*enum* _anon_ (9sys/dirstat.c) 216a⟩≡ (481a)

```
enum
{
    DIRSIZE = STATFIXLEN + 16 * 4       /* enough for encoded stat buf + some reasonable strings */
};
```

⟨*function* dirstat 216b⟩≡ (481a)

```
Dir*
dirstat(char *name)
{
    Dir *d;
    uchar *buf;
    int n, nd, i;

    nd = DIRSIZE;
    for(i=0; i<2; i++){ /* should work by the second try */
        d = malloc(sizeof(Dir) + BIT16SZ + nd);
        if(d == nil)
            return nil;
        buf = (uchar*)&d[1];
        n = stat(name, buf, BIT16SZ+nd);
        if(n < BIT16SZ){
            free(d);
            return nil;
        }
        nd = GBIT16((uchar*)buf);   /* upper bound on size of Dir + strings */
        if(nd <= n){
            convM2D(buf, n, d, (char*)&d[1]);
            return d;
        }
        /* else sizeof(Dir)+BIT16SZ+nd is plenty */
        free(d);
    }
    return nil;
}
```

Uses DIRSIZE-231 216a, convM2D() 280, free() 63i, malloc() 63g, and stat().

⟨*function* dirwstat 216c⟩≡ (481b)

```
int
dirwstat(char *name, Dir *d)
{
    uchar *buf;
    int r;

    r = sizeD2M(d);
```

```
    buf = malloc(r);
    if(buf == nil)
        return -1;
    convD2M(d, buf, r);
    r = wstat(name, buf, r);
    free(buf);
    return r;
}
```

Uses `convD2M()` 281, `free()` 63i, `malloc()` 63g, `sizeD2M()` 465a, and `wstat()`.

15.5 getwd()

⟨*function* `getwd` 217a⟩≡ (486c)

```
char*
getwd(char *buf, int nbuf)
{
    int n;
    fdt fd;

    fd = open(".", OREAD);
    if(fd < 0)
        return nil;
    n = fd2path(fd, buf, nbuf);
    close(fd);
    if(n < 0)
        return nil;
    return buf;
}
```

Uses `close()`, `fd2path()`, and `open()`.

15.6 The deprecated mktemp()

`mktemp()` has a classic TOCTOU (time-of-check, time-of-use) vulnerability: it checks whether a filename exists with `access()`, then expects the caller to `create` it—but another process could create that file in between. The safe alternative is `mkstemp()` (next section), which atomically creates and opens the file.

⟨*function* `mktemp` 217b⟩≡ (405f)

```
char*
mktemp(char *as)
{
    char *s;
    unsigned pid;
    int i;
    char err[ERRMAX];

    pid = getpid();
    s = as;
    while(*s++)
        ;
    s--;
    while(*--s == 'X') {
        *s = pid % 10 + '0';
        pid = pid/10;
    }
    s++;
    i = 'a';
```

```
    while(access(as, 0) != -1) {
        if (i == 'z')
            return "/";
        *s = i++;
    }
    err[0] = '\0';
    errstr(err, sizeof err);    /* clear the error */
    return as;
}
```

Uses `access()` 220, `errstr()`, and `getpid()` 232a.

15.7 The safe `mkstemp()`

15.8 Normalizing a filename: `cleanname()`

`cleanname` normalizes a file path in place: it removes redundant slashes, eliminates `.` components, and resolves `..` where possible. It also handles Plan 9's `#` device syntax (e.g., `#c/cons`) by preserving the device prefix and treating the rest as a rooted path. The two-pass approach first collapses duplicate slashes, then processes dot-dot in a single forward scan—the `d` pointer tracks the output position while `s` advances through the input.

Some worked examples, showing the rewrite as a series of steps:

```
Input                 After dedup //    After ./..        Output
-------------         --------------    --------------    --------
/usr//rsc/.           /usr/rsc/.        /usr/rsc          /usr/rsc
a/b/../c              a/b/../c          a/c               a/c
/a/../../b            /a/../../b        /b                /b
../foo                ../foo            ../foo            ../foo
""                                                        .
#c/cons               #c/cons           #c/cons           #c/cons
```

Two cases are worth noticing. `/a/../../b` becomes `/b`: at the absolute root, an extra `..` is swallowed (because `/..` equals `/` in Plan 9—and in UNIX—rather than being an error). By contrast, `../foo` stays `../foo`: a relative path can keep leading `..` components, because there is no way to know without consulting the filesystem how many levels above the cwd the result actually lives. The `d0` pointer is used to mark this: while `d` is still sitting at `d0` and the path is not rooted, the `..` branch advances `d0` past it instead of deleting anything.

This lexical resolution—working from the path string alone, with no system calls—is what makes Plan 9 namespaces possible. Because `..` is resolved by `cleanname` rather than by the filesystem, the parent of `/foo` is always `/`, even after `bind /home/x /foo` has remapped `/foo` to something completely different. UNIX's `realpath()` follows the filesystem's parent pointers instead, which gives the wrong answer through bind mounts and symbolic links and is one of the reasons Plan 9 never adopted symlinks.

⟨*macro* SEP 218a⟩≡ (393b)

```
#define SEP(x)  ((x)=='/' || (x) == '\0')
```

⟨*function* cleanname 218b⟩≡ (393b)

```
/*
 * In place, rewrite name to compress multiple /, eliminate ., and process ..
 */
char*
cleanname(char *name)
{
    char *s;    /* source of copy */
    char *d;    /* destination of copy */
```

```
char *d0;   /* start of path afer the root name */
Rune r;
bool rooted;

if(name[0] == '\0')
    return strcpy(name, ".");

rooted = false;
d0 = name;
if(d0[0] == '#'){
    if(d0[1] == '\0')
        return d0;
    d0  += 1 + chartorune(&r, d0+1); /* ignore slash: #/ */
    while(!SEP(*d0))
        d0 += chartorune(&r, d0);
    if(d0 == '\0')
        return name;
    d0++;   /* keep / after #<name> */
    rooted = true;
}else if(d0[0] == '/'){
    rooted = true;
    d0++;
}

s = d0;
if(rooted){
    /* skip extra '/' at root name */
    for(; *s == '/'; s++)
        ;
}
/* remove dup slashes */
for(d = d0; *s != '\0'; s++){
    *d++ = *s;
    if(*s == '/')
        while(s[1] == '/')
            s++;
}
*d = '\0';

d = d0;
s = d0;
while(*s != '\0'){
    if(s[0] == '.' && SEP(s[1])){
        if(s[1] == '\0')
            break;
        s+= 2;
        continue;
    }
    if(s[0] == '.' && s[1] == '.' && SEP(s[2])){
        if(d == d0){
            if(rooted){
                /* /../x -> /x */
                if(s[2] == '\0')
                    break;
                s += 3;
                continue;
            }else{
                /* ../x -> ../x; and never collect ../ */
                d0 += 3;
            }
```

```
            }
            if(d > d0){
                /* a/../x -> x */
                assert(d-2 >= d0 && d[-1] == '/');
                for(d -= 2; d > d0 && d[-1] != '/'; d--)
                        ;
                if(s[2] == '\0')
                    break;
                s += 3;
                continue;
            }
        }
        while(!SEP(*s))
            *d++ = *s++;
        if(*s == '\0')
            break;

        *d++ = *s++;
    }
    *d = '\0';
    if(d-1 > name && d[-1] == '/')  /* thanks to #/ */
        *--d = '\0';
    if(name[0] == '\0')
        strcpy(name, ".");
    return name;
}
```

Uses `SEP-53` 218a, `chartorune()` 84d, and `strcpy()` 81c.

15.9 access()

`access()` checks file accessibility by actually trying to open the file with the requested mode. For existence checks (`AEXIST`), it uses `dirstat()`[216b] instead. This is different from UNIX, where `access()` is a dedicated system call. The Plan 9 approach is more honest—it tests exactly what will happen when you try to use the file—but it's also more expensive, since it opens and immediately closes the file.

⟨*function* `access` 220⟩≡ (462f)

```
int
access(char *name, int mode)
{
    fdt fd;
    Dir *db;
    static char omode[] = {
        0,
        OEXEC,
        OWRITE,
        ORDWR,
        OREAD,
        OEXEC,  /* only approximate */
        ORDWR,
        ORDWR   /* only approximate */
    };

    if(mode == AEXIST){
        db = dirstat(name);
        free(db);
        if(db != nil)
            return 0;
        return -1;
```

```
        }
        fd = open(name, omode[mode&7]);
        if(fd >= 0){
            close(fd);
            return 0;
        }
        return -1;
    }
```

Uses `close()`, `dirstat()` 216b, `free()` 63i, and `open()`.

15.10 Path manipulation

Chapter 16

Namespace

Plan 9's namespace is a per-process view of the file system that can be customized with `bind()` and `mount()`. `bind` makes one directory appear at another location; `mount` attaches a 9P file server to a mount point. The `MBEFORE` and `MAFTER` flags create union directories where multiple file trees are stacked at the same path—a powerful mechanism that replaces UNIX's `PATH` variable, shared libraries, and much of the role of environment variables.

⟨*type* `Namespace_flag` 222a⟩≡ (379b)
```
// enum Namespace_flag, mount/bind parameter
#define MREPL 0x0000 /* mount replaces object */
#define MBEFORE 0x0001 /* mount goes before others in union directory */
#define MAFTER 0x0002 /* mount goes after others in union directory */

#define MCREATE 0x0004 /* permit creation in mounted directory */
#define MCACHE 0x0010 /* cache some data */
// bitset<Namespace_flag>
#define MORDER 0x0003 /* mask for bits defining order of mounting */

#define MMASK 0x0017 /* all bits on */
```

16.1 Syscalls: `bind()`, `mount()`, etc

These syscalls manipulate the per-process namespace.

⟨*signatures namespace syscall* 222b⟩≡ (379b)
```
extern int bind(char*, char*, int/*Mxxx*/);
extern int mount(fdt, int, char*, int/*Mxxx*/, char*);
extern int unmount(char*, char*);
```

`bind` makes one local path appear at another by inserting a mount entry in the process's namespace. `mount` does the same but for a remote file server: it takes a file descriptor (typically a pipe to a 9P server) and attaches it at a path, so that file operations on that path are forwarded over the pipe as 9P messages. Both take flags (`MBEFORE`, `MAFTER`, `MREPL`) that control how the new entry combines with existing entries in union directories. `unmount` reverses a previous `bind` or `mount`. In `mount(fd, afd, old, flags, aname)`: `fd` is the file descriptor to the 9P server, `afd` is an optional authentication fd (-1 if none), `old` is the mount point path, and `aname` names which tree to attach to on the server (often `nil`). For example:

```
bind("/arm/bin", "/bin", MBEFORE);
mount(fd, -1, "/mnt/ws", MREPL, "");
```

UNIX's `mount` is a privileged operation (root only) that attaches a disk partition to a directory. Plan 9's `mount` is unprivileged and far more general: any process can mount any 9P file server (including user-level

programs) at any point in its own namespace. `bind` has no UNIX equivalent at all—the closest is Linux's `mount --bind`, added decades later, which lacks union directory support. The combination of per-process namespaces, `bind`, and `mount` is one of Plan 9's most distinctive features. UNIX's `chroot` provides a crude approximation (change the root directory), but it is a blunt, all-or-nothing tool. Linux eventually added mount namespaces (2002), which are closer to Plan 9's model and form the basis of container technologies like Docker—but they arrived 20 years after Plan 9 had the same capability in a much simpler form.

Chapter 17

Time

This chapter covers the time-related functions: system calls for sleeping and setting alarms, time-of-day functions, timezone handling, and conversions between epoch seconds and the `Tm` structure.

⟨*signatures time functions* 224a⟩≡ (368b)
```
extern  long    time(long*);
extern  vlong   nsec(void);

extern  Tm*     gmtime(long);
extern  Tm*     localtime(long);
```

`time` returns seconds since the epoch; `nsec` returns nanoseconds (by reading `/dev/bintime`), useful for benchmarking. `gmtime` and `localtime` convert epoch seconds to a `Tm` structure (presented in the Core Data Structures chapter) and return a pointer to a static buffer.

17.1 Syscalls: `sleep()`, `alarm()`

These syscalls deal with time delays and timeouts.

⟨*signatures time syscall* 224b⟩≡ (379b)
```
extern long alarm(ulong);
extern int sleep(long); //less: could be void (ulong). 0 means yield.
```

`sleep` suspends the process for `n` milliseconds. If `n` is zero or negative, it simply yields the processor to other processes. `alarm` sets a timer: after `n` milliseconds, the kernel sends an `"alarm"` note to the process. Calling `alarm(0)` cancels a pending alarm. `alarm` returns the time remaining from any previous alarm.

UNIX's `sleep()` takes seconds (not milliseconds), and UNIX's `alarm()` also takes seconds. For sub-second timing, UNIX programs must use `nanosleep` or `setitimer`—separate, more complex interfaces. Plan 9 uses milliseconds, which is fine for most purposes. There is no sub-millisecond sleep; for nanosecond-precision timing, programs read the cycle counter via `nsec()` and busy-wait.

17.2 Time and day

Plan 9's time functions read from the file system rather than making specialized system calls. `/dev/time` provides the seconds since epoch, `/dev/bintime` provides nanosecond precision, and `/env/timezone` provides the local timezone rules. The `nsec` system call is a newer, more efficient alternative to reading `/dev/bintime`.

17.3 Time zones and `localtime()`

`localtime` converts epoch seconds to a `Tm` structure in the local timezone. It reads the timezone from `/env/timezone` (lazily, on first call), applies the standard offset, then checks whether daylight saving time is active by scanning

the dlpairs array of start/end timestamps.

⟨*function* localtime 225a⟩≡ (470)

```
Tm*
localtime(long tim)
{
    Tm *ct;
    long t, *p;
    int dlflag;

    if(timezone.stname[0] == 0)
        readtimezone();
    t = tim + timezone.stdiff;
    dlflag = 0;
    for(p = timezone.dlpairs; *p; p += 2)
        if(t >= p[0])
        if(t < p[1]) {
            t = tim + timezone.dldiff;
            dlflag++;
            break;
        }
    ct = gmtime(t);
    if(dlflag){
        strcpy(ct->zone, timezone.dlname);
        ct->tzoff = timezone.dldiff;
    } else {
        strcpy(ct->zone, timezone.stname);
        ct->tzoff = timezone.stdiff;
    }
    return ct;
}
```

Uses gmtime() 467e, strcpy() 81c, and timezone-236 467c.

asctime formats a Tm into the traditional Thu Jan 01 00:00:00 GMT 1970 string. It uses a clever trick: the day and month names are stored as a single concatenated string, and the appropriate 3-character substring is located by multiplying the index by 3. The template string is initialized with the epoch date, and individual characters are overwritten—no sprintf needed.

⟨*function* asctime 225b⟩≡ (470)

```
char*
asctime(Tm *t)
{
    char *ncp;
    static char cbuf[30];

    strcpy(cbuf, "Thu Jan 01 00:00:00 GMT 1970\n");
    ncp = &"SunMonTueWedThuFriSat"[t->wday*3];
    cbuf[0] = *ncp++;
    cbuf[1] = *ncp++;
    cbuf[2] = *ncp;
    ncp = &"JanFebMarAprMayJunJulAugSepOctNovDec"[t->mon*3];
    cbuf[4] = *ncp++;
    cbuf[5] = *ncp++;
    cbuf[6] = *ncp;
    ct_numb(cbuf+8, t->mday);
    ct_numb(cbuf+11, t->hour+100);
    ct_numb(cbuf+14, t->min+100);
    ct_numb(cbuf+17, t->sec+100);
    ncp = t->zone;
    cbuf[20] = *ncp++;
```

```
	cbuf[21] = *ncp++;
	cbuf[22] = *ncp;
	if(t->year >= 100) {
		cbuf[24] = '2';
		cbuf[25] = '0';
	}
	ct_numb(cbuf+26, t->year+100);
	return cbuf;
}
```

Uses ct_numb() 468b and strcpy() 81c.

17.4 time()

time first tries the nsec system call (nanosecond precision), falling back to oldtime which reads /dev/time as a file. The fallback is notable: because /dev/time is a regular file, a fork with shared file descriptors creates a race on the file offset, so oldtime retries up to 100 times if the read returns zero bytes.

⟨*function* time 226a⟩≡ (494c)

```
long
time(long *tp)
{
	vlong t;

	t = nsec()/1000000000LL;
	if(t == 0)
		t = oldtime(0);
	if(tp != nil)
		*tp = t;
	return t;
}
```

Uses nsec() 227 and oldtime() 226b.

⟨*function* oldtime 226b⟩≡ (494c)

```
/*
 *  After a fork with fd's copied, both fd's are pointing to
 *  the same Chan structure.  Since the offset is kept in the Chan
 *  structure, the seek's and read's in the two processes can
 *  compete at moving the offset around.  Hence the unusual loop
 *  in the middle of this routine.
 */
static long
oldtime(long *tp)
{
	char b[20];
	static int f = -1;
	int i, retries;
	long t;

	memset(b, 0, sizeof(b));
	for(retries = 0; retries < 100; retries++){
		if(f < 0)
			f = open("/dev/time", OREAD|OCEXEC);
		if(f < 0)
			break;
		if(seek(f, 0, 0) < 0 || (i = read(f, b, sizeof(b))) < 0){
			close(f);
			f = -1;
		} else {
```

```
            if(i != 0)
                break;
        }
    }
    t = atol(b);
    if(tp)
        *tp = t;
    return t;
}
```

Uses atol() 123c, close(), memset() 46b, open(), read() 192a, and seek().

⟨*function* nsec 227⟩≡ (487e)

```
vlong
nsec(void)
{
    uchar b[8];
    vlong t;
    int pid, i, f, tries;

    /*
     * Threaded programs may have multiple procs
     * with different fd tables, so we may need to open
     * /dev/bintime on a per-pid basis
     */

    /* First, look if we've opened it for this particular pid */
    if((pid = _tos->pid) == 0)      /* 9vx bug, perhaps? */
        _tos->pid = pid = getpid();
    do{
        f = -1;
        for(i = 0; i < nelem(fds); i++)
            if(fds[i].pid == pid){
                f = fds[i].fd;
                break;
            }
        tries = 0;
        if(f < 0){
            /* If it's not open for this pid, try the global pid */
            if(fd >= 0)
                f = fd;
            else{
                /* must open */
                if((f = open("/dev/bintime", OREAD|OCEXEC)) < 0)
                    return 0;
                fd = f;
                for(i = 0; i < nelem(fds); i++)
                    if(fds[i].pid == pid || fds[i].pid == 0){
                        fds[i].pid = pid;
                        fds[i].fd = f;
                        break;
                    }
            }
        }
        if(pread(f, b, sizeof b, 0) == sizeof b){
            be2vlong(&t, b);
            return t;
        }
        close(f);
        if(i < nelem(fds))
            fds[i].fd = -1;
```

```
    }while(tries++ == 0);   /* retry once */
    USED(tries);
    return 0;
}
```

Uses `be2vlong()` 487b, `close()`, `fd-255` 487c, `fds-256` 487d, `getpid()` 232a, `open()`, and `pread()`.

17.5 Conversions

`tm2sec` converts a broken-down time back to epoch seconds. It accumulates seconds year by year (handling leap years via `yrsize`), then adds the day-of-year, hours, minutes, and seconds. The timezone adjustment at the end subtracts the UTC offset, converting local time back to absolute time.

⟨*function* `tm2sec` 228⟩≡ (498a)

```
/*
 * compute seconds since Jan 1 1970 GMT
 * and convert to our timezone.
 */
long
tm2sec(Tm *tm)
{
    long secs;
    int i, yday, year, *d2m;

    if(strcmp(tm->zone, "GMT") != 0 && timezone.stname[0] == 0)
        readtimezone();
    secs = 0;

    /*
     *  seconds per year
     */
    year = tm->year + 1900;
    for(i = 1970; i < year; i++){
        d2m = yrsize(i);
        secs += d2m[0] * SEC2DAY;
    }

    /*
     *  if mday is set, use mon and mday to compute yday
     */
    if(tm->mday){
        yday = 0;
        d2m = yrsize(year);
        for(i=0; i<tm->mon; i++)
            yday += d2m[i+1];
        yday += tm->mday-1;
    }else{
        yday = tm->yday;
    }
    secs += yday * SEC2DAY;

    /*
     * hours, minutes, seconds
     */
    secs += tm->hour * SEC2HOUR;
    secs += tm->min * SEC2MIN;
    secs += tm->sec;

    /*
```

```
     * Only handles zones mentioned in /env/timezone,
     * but things get too ambiguous otherwise.
     */
    if(strcmp(tm->zone, timezone.stname) == 0)
        secs -= timezone.stdiff;
    else if(strcmp(tm->zone, timezone.dlname) == 0)
        secs -= timezone.dldiff;
    if(secs < 0)
        secs = 0;
    return secs;
}
```

Uses SEC2DAY-242 496d, SEC2HOUR-241 496c, SEC2MIN-240 496b, strcmp() 81a, timezone-239 496a, and yrsize() 496g.

Chapter 18

Fork, Exec, and Wait

Plan 9 replaces UNIX's `fork` with `rfork` ("resource fork"), which takes a bitmask specifying exactly which resources to share or copy: file descriptors (`RFFDG`), namespace (`RFNAMEG`), environment (`RFENVG`), memory (`RFMEM`), and note group (`RFNOTEG`). This single call subsumes UNIX's `fork`, `clone`, `vfork`, and `pthread_create`, providing fine-grained control over what the child process inherits.

⟨*type* `Rfork_flag` 230a⟩≡ (379b)
```
/* rfork */
enum Rfork_flags
{
    RFNAMEG  = (1<<0),
    RFENVG   = (1<<1),
    RFFDG    = (1<<2),
    RFNOTEG  = (1<<3),
    RFPROC   = (1<<4),
    RFMEM    = (1<<5),
    RFNOWAIT = (1<<6),
    RFCNAMEG = (1<<10),
    RFCENVG  = (1<<11),
    RFCFDG   = (1<<12),
    RFREND   = (1<<13),
    RFNOMNT  = (1<<14)
};
```

⟨*signatures process syscall wrapper* 230b⟩+≡ (379b) ◁40
```
extern int fork(void);
extern int execl(char*, ...);
extern Waitmsg*wait(void);
extern int waitpid(void);
```

⟨*signatures other process syscall wrapper* 230c⟩≡ (379b)
```
extern int getpid(void);
extern int getppid(void);
```

The wrappers above are convenience functions built on the raw syscalls below. `fork` calls `rfork(RFFDG|RFREND|RF` (the traditional fork semantics), `execl` is a variadic wrapper around `exec`, and `wait` parses the raw `await` string into a `Waitmsg` structure.

18.1 Syscalls: `rfork()`, `exec()`, `await()`, etc

These syscalls manage processes. The `_exits` syscall was presented in Chapter 5.

⟨*signatures process syscall* 230d⟩+≡ (379b) ◁44a
```
extern int rfork(int);
extern int exec(char*, char*[]);
extern int await(char*, int);
```

Common rfork flag combinations:

```
rfork(RFPROC|RFFDG|RFREND)       traditional fork
rfork(RFPROC|RFMEM|RFNOWAIT)     shared-memory "thread"
rfork(RFNAMEG|RFENVG)             isolate namespace+env (no new proc)
```

`await(buf, n)` fills `buf` with a space-separated string: `pid time1 time2 time3 exitstring`. The library wrapper `wait()` parses this into a `Waitmsg` structure (shown in Chapter 22).

In the kernel, `sysrfork()` creates a new process, selectively sharing or copying resources (memory, file descriptors, namespace, environment) based on the flag bits. `sysexec()` replaces the current process's memory image with a new program: it reads the executable, creates fresh text/data/bss segments, sets up the stack with `argc` and `argv`, and jumps to `_main`. `sysawait()` blocks until a child process exits, then returns a text string containing the child's pid, CPU times, and exit message.

UNIX has separate `fork()`, `clone()` (Linux), and `vfork()` syscalls for different sharing patterns. Plan 9 unifies them all into `rfork` with flags. Linux's `clone()` was in fact inspired by Plan 9's `rfork`. `await` returns a text string rather than UNIX's integer status code (see the `_exits` comparison in Chapter 5).

The fork/exec pattern is the cornerstone of UNIX process creation: `fork` duplicates the process, then the child adjusts file descriptors (for redirection), changes namespace (for sandboxing), and finally calls `exec` to replace itself with a new program. Plan 9's `rfork` makes this even more flexible by letting you choose exactly which resources to share, copy, or create fresh.

18.2 fork()

The convenience `fork` function calls `rfork` with the traditional UNIX semantics: create a new process (`RFPROC`) with copied file descriptors (`RFFDG`) and a new rendezvous group (`RFREND`).

⟨*function* `fork` 231a⟩≡ (483a)

```
int
fork(void)
{
    return rfork(RFPROC|RFFDG|RFREND);
}
```

Uses `rfork()`.

18.3 execl()

`execl` is a convenience wrapper around `exec` that takes a variable number of string arguments instead of an argv array. The `&f1+` trick works because the Plan 9 C compiler lays out variadic arguments contiguously on the stack—the address after `f` is the start of the remaining arguments, which form the argv array that `exec` expects.

⟨*function* `execl` 231b⟩≡ (395b)

```
int
execl(char *f, ...)
{

    return exec(f, &f+1);
}
```

Uses `exec()`.

18.4 wait()

The `wait` function (shown in the IPC chapter) parses the text returned by the `await` system call. In Plan 9, a child process exits with a string, not an integer, so the parent receives a text message containing the pid, CPU times, and exit string. This is richer than UNIX's `waitpid` which only returns a status integer.

18.5 getpid()

In Plan 9, `getpid` reads `#c/pid` (the console device's pid file) rather than making a system call. This is typical of Plan 9's "everything is a file" design: process information is exposed through the file system rather than through specialized system calls.

⟨*function* `getpid` 232a⟩≡ (486a)

```
int
getpid(void)
{
    char b[20];
    fdt f;

    memset(b, 0, sizeof(b));
    f = open("#c/pid", OREAD);
    if(f >= 0) {
        read(f, b, sizeof(b));
        close(f);
    }
    return atol(b);
}
```

Uses `atol()` 123c, `close()`, `memset()` 46b, `open()`, and `read()` 192a.

⟨*function* `getppid` 232b⟩≡ (486b)

```
int
getppid(void)
{
    char b[20];
    int f;

    memset(b, 0, sizeof(b));
    f = open("/dev/ppid", OREAD);
    if(f >= 0) {
        read(f, b, sizeof(b));
        close(f);
    }
    return atol(b);
}
```

Uses `atol()` 123c, `close()`, `memset()` 46b, `open()`, and `read()` 192a.

Chapter 19

Error Management

Plan 9 replaces UNIX's `errno` (an integer error code) with a per-process error *string* of up to 128 characters. The `errstr` system call swaps the current error string with a buffer provided by the caller—passing an empty buffer reads and clears the error. The `%r` format verb in `print`/`fprint` outputs the current error string, and `werrstr` sets it. This design avoids the fixed, enumerated error codes of UNIX and allows file servers to return descriptive error messages.

⟨*signatures error syscall wrapper* 233a⟩≡ (379b)
```
extern void rerrstr(char*, uint);
extern void werrstr(char*, ...);

#pragma varargck argpos werrstr 1
```

⟨*constant* `ERRMAX` 233b⟩≡ (379b)
```
#define ERRMAX 128 /* max length of error string */
```

⟨*signatures error other wrapper* 233c⟩≡ (379b)
```
extern  void    perror(char*);
extern  void    sysfatal(char*, ...);

#pragma varargck    argpos  sysfatal    1
```

19.1 Syscall: `errstr()`

This syscall handles error reporting.

⟨*signatures error syscall* 233d⟩≡ (379b)
```
extern int errstr(char*, uint);
```

`errstr` *swaps* the caller's buffer with the kernel's per-process error string. If you pass an empty buffer (`buf[0] = '\0'`), the kernel's error string is returned and cleared. If you pass a non-empty buffer, you set the error string (used by `werrstr`). This swap design is unusual but efficient: a single syscall both reads and resets the error.

UNIX uses `errno`, a global integer that each syscall sets on failure. Programs must call `strerror(errno)` or `perror()` to get a human-readable message, and the set of error codes is fixed at compile time (`ENOENT`, `EPERM`, etc.). Plan 9's per-process string avoids all three problems: it is per-process (no thread-safety issue), it is human-readable directly, and file servers can return application-specific messages that do not need to fit into a predefined enum.

19.2 rerrstr(), werrstr()

Because **errstr** *swaps* the error string (it reads the current error and replaces it with the buffer's contents), **rerrstr** must call it twice: once to read the error into a temporary buffer, and again to restore it. This read-without-clear pattern is necessary when you want to inspect the error without consuming it. **werrstr** does the opposite: it formats a message into a buffer and swaps it in, setting the process's error string.

⟨*function* rerrstr 234a⟩≡ (493a)
```
void
rerrstr(char *buf, uint nbuf)
{
    char tmp[ERRMAX];

    tmp[0] = '\0';
    errstr(tmp, sizeof tmp);

    utfecpy(buf, buf+nbuf, tmp);
    errstr(tmp, sizeof tmp);
}
```
Uses errstr() and utfecpy() 460b.

⟨*function* werrstr 234b⟩≡ (499c)
```
void
werrstr(char *fmt, ...)
{
    va_list arg;
    char buf[ERRMAX];

    va_start(arg, fmt);
    vseprint(buf, buf+ERRMAX, fmt, arg);
    va_end(arg);
    errstr(buf, sizeof buf);
}
```
Uses errstr() and vseprint() 536b.

19.3 perror()

⟨*function* perror 234c⟩≡ (408a)
```
void
perror(char *s)
{
    char buf[ERRMAX];

    buf[0] = '\0';
    errstr(buf, sizeof buf);
    if(s && *s)
        fprint(2, "%s: %s\n", s, buf);
    else
        fprint(2, "%s\n", buf);
}
```

19.4 Error return values

C has no standard convention for error return values: some functions return 0 on error (`convM2S`), some return -1 (`open`), and some return 1 (`getfields`). These typedefs and constants document the convention used by

each function, making the code more self-documenting. They were added by the author to the Plan 9 source to reduce confusion.

⟨*type* errorxxx 235a⟩≡ (368b)
```
// later: unify all of that to be more consistent!
typedef int error0; // 0 is the error value
typedef int error1; // 1 is the error value
typedef int errorneg1; // -1 is the error value
typedef int errorn; // 1 or more means error
```

⟨*constant* OKxxx 235b⟩≡ (368b)
```
#define OK_0 0
#define OK_1 1
```

⟨*constant* ERRORxxx 235c⟩≡ (368b)
```
#define ERROR_0 0
#define ERROR_1 1
#define ERROR_NEG1 (-1)
```

19.5 assert()

`assert` aborts the program when a condition is false. The macro uses the Plan 9 C compiler's implicit stringification (the string `"x"` inside the macro expands to the literal text of the condition). The indirection through a function pointer `_assert` allows `libthread` to replace the default handler with one that reports the thread ID and does thread-safe cleanup.

⟨*macro* assert 235d⟩≡ (368b)
```
#define assert(x)   do{ if(x) {} else _assert("x"); }while(0)
```

⟨*global* _assert 235e⟩≡ (387b)
```
void (*_assert)(char*) = default_assert;
```
Uses `_assert` 235e and `default_assert()` 235f.

⟨*function* default_assert 235f⟩≡ (387b)
```
void
default_assert(char *s)
{
    if(__assert)
        (*__assert)(s);
    fprint(2, "assert failed: %s\n", s);
    abort();
}
```
Uses `__assert` 235g, `abort()` 356d, and `fprint()` 75a.

⟨*global* __assert 235g⟩≡ (387b)
```
void (*__assert)(char*);
```

19.6 sysfatal()

`sysfatal` is the standard way to terminate with an error message. It formats the message, prepends the program name (`argv0`), and calls `exits`. Like `_assert`, it goes through a function pointer `_sysfatal` so `libthread` can

replace it with a thread-aware version that cleans up properly.

⟨*function* sysfatal 236a⟩≡ (493e)

```
void
sysfatal(char *fmt, ...)
{
    va_list arg;

    va_start(arg, fmt);
    (*_sysfatal)(fmt, arg);
    va_end(arg);
}
```

Uses _sysfatal 236b.

⟨*global* _sysfatal 236b⟩≡ (493e)

```
void (*_sysfatal)(char *fmt, va_list arg) = _sysfatalimpl;
```

Uses _sysfatal 236b and _sysfatalimpl() 236c.

⟨*function* _sysfatalimpl 236c⟩≡ (493e)

```
static void
_sysfatalimpl(char *fmt, va_list arg)
{
    char buf[1024];

    vseprint(buf, buf+sizeof(buf), fmt, arg);
    if(argv0)
        fprint(2, "%s: %s\n", argv0, buf);
    else
        fprint(2, "%s\n", buf);
    exits(buf);
}
```

Uses argv0, exits() 45b, fprint() 75a, and vseprint() 536b.

Chapter 20

Security

Plan 9 identifies users by name strings rather than numeric uid/gid pairs. `getuser` reads the current user name from `/dev/user`, reflecting the "everything is a file" design. In a distributed environment where different file servers may have different user databases, string-based identities are more flexible than numeric IDs.

⟨*signatures security syscall* 237a⟩≡ (379b)
```
extern int fauth(int, char*);
extern int fversion(int, int, char*, int);
```

⟨*signatures security syscall wrapper* 237b⟩≡ (379b)
```
extern  char*   getuser(void);
```

`fauth` opens an authentication channel on a file server connection (the `int` is the fd to the server, the `char*` is the aname). `fversion` negotiates the 9P protocol version and maximum message size with a server. Both are low-level plumbing used by `mount`—most programs never call them directly. `getuser` simply reads `/dev/user`.

20.1 getuser()

⟨*function* getuser 237c⟩≡ (397b)
```
char*
getuser(void)
{
    static char user[64];
    fdt fd;
    int n;

    fd = open("/dev/user", OREAD);
    if(fd < 0)
        return "none";
    n = read(fd, user, (sizeof user)-1);
    close(fd);
    if(n <= 0)
        strcpy(user, "none");
    else
        user[n] = '\0';
    return user;
}
```
Uses `close()`, `open()`, `read()` 192a, and `strcpy()` 81c.

20.2 Cryptography

The `encrypt/decrypt` functions implement DES-based block encryption, processing 7-byte blocks at a time (DES's 56-bit key = 7 bytes). The encryption is done in-place, and the buffer must be at least 8 bytes. The final

partial block is handled by overlapping with the previous block. `netcrypt` is a helper for the authentication protocol: it encrypts a challenge string and formats the first 4 bytes as hexadecimal.

⟨*function* `encrypt` 238a⟩≡ (393c)

```
/*
 * destructively encrypt the buffer, which
 * must be at least 8 characters long.
 */
int
encrypt(void *key, void *vbuf, int n)
{
    ulong ekey[32];
    uchar *buf;
    int i, r;

    if(n < 8)
        return 0;
    key_setup(key, ekey);
    buf = vbuf;
    n--;
    r = n % 7;
    n /= 7;
    for(i = 0; i < n; i++){
        block_cipher(ekey, buf, 0);
        buf += 7;
    }
    if(r)
        block_cipher(ekey, buf - 7 + r, 0);
    return 1;
}
```

⟨*function* `decrypt` 238b⟩≡ (393c)

```
/*
 * destructively decrypt the buffer, which
 * must be at least 8 characters long.
 */
int
decrypt(void *key, void *vbuf, int n)
{
    ulong ekey[128];
    uchar *buf;
    int i, r;

    if(n < 8)
        return 0;
    key_setup(key, ekey);
    buf = vbuf;
    n--;
    r = n % 7;
    n /= 7;
    buf += n * 7;
    if(r)
        block_cipher(ekey, buf - 7 + r, 1);
    for(i = 0; i < n; i++){
        buf -= 7;
        block_cipher(ekey, buf, 1);
    }
    return 1;
}
```

⟨*function* netcrypt 239a⟩≡ (406e)
```
int
netcrypt(void *key, void *chal)
{
    uchar buf[8], *p;

    strncpy((char*)buf, chal, 7);
    buf[7] = '\0';
    for(p = buf; *p && *p != '\n'; p++)
        ;
    *p = '\0';
    encrypt(key, buf, 8);
    sprint(chal, "%.2ux%.2ux%.2ux%.2ux", buf[0], buf[1], buf[2], buf[3]);
    return 1;
}
```

20.3 SSL/TLS

`pushssl` and `pushtls` insert encryption layers on an existing connection using Plan 9's SSL/TLS kernel devices (`#D/ssl` and `#a/tls`). They follow the standard Plan 9 pattern: open the `clone` file to get a new channel number, configure it by writing commands to the control file (secrets, algorithm, etc.), and return the `data` file descriptor. The caller's original `fd` is closed—reads and writes now go through the encrypted channel transparently.

⟨*function* pushssl 239b⟩≡ (489a)
```
/*
 * Since the SSL device uses decimal file descriptors to name channels,
 * it is impossible for a user-level file server to stand in for the kernel device.
 * Thus we hard-code #D rather than use /net/ssl.
 */

int
pushssl(int fd, char *alg, char *secin, char *secout, int *cfd)
{
    char buf[8];
    char dname[64];
    int n, data, ctl;

    ctl = open("#D/ssl/clone", ORDWR);
    if(ctl < 0)
        return -1;
    n = read(ctl, buf, sizeof(buf)-1);
    if(n < 0)
        goto error;
    buf[n] = 0;
    sprint(dname, "#D/ssl/%s/data", buf);
    data = open(dname, ORDWR);
    if(data < 0)
        goto error;
    if(fprint(ctl, "fd %d", fd) < 0 ||
       fprint(ctl, "secretin %s", secin) < 0 ||
       fprint(ctl, "secretout %s", secout) < 0 ||
       fprint(ctl, "alg %s", alg) < 0){
        close(data);
        goto error;
    }
    close(fd);
    if(cfd != 0)
        *cfd = ctl;
```

```
    else
        close(ctl);
    return data;
error:
    close(ctl);
    return -1;
}
```

Uses close(), fprint() 75a, open(), read() 192a, and sprint() 75b.

⟨*function* pushtls 240⟩≡ (489d)

```
// given a plain fd and secrets established beforehand, return encrypted connection
int
pushtls(int fd, char *hashalg, char *encalg, int isclient, char *secret, char *dir)
{
    char buf[8];
    char dname[64];
    int n, data, ctl, hand;

    // open a new filter; get ctl fd
    data = hand = -1;
    // /net/tls uses decimal file descriptors to name channels, hence a
    // user-level file server can't stand in for #a; may as well hard-code it.
    ctl = open("#a/tls/clone", ORDWR);
    if(ctl < 0)
        goto error;
    n = read(ctl, buf, sizeof(buf)-1);
    if(n < 0)
        goto error;
    buf[n] = 0;
    if(dir)
        sprint(dir, "#a/tls/%s", buf);

    // get application fd
    sprint(dname, "#a/tls/%s/data", buf);
    data = open(dname, ORDWR);
    if(data < 0)
        goto error;

    // get handshake fd
    sprint(dname, "#a/tls/%s/hand", buf);
    hand = open(dname, ORDWR);
    if(hand < 0)
        goto error;

    // speak a minimal handshake
    if(fprint(ctl, "fd %d 0x301", fd) < 0 ||
       fprint(ctl, "version 0x301") < 0 ||
       fprint(ctl, "secret %s %s %d %s", hashalg, encalg, isclient, secret) < 0 ||
       fprint(ctl, "changecipher") < 0 ||
       finished(hand, isclient) < 0 ||
       fprint(ctl, "opened") < 0){
        close(hand);
        hand = -1;
        goto error;
    }
    close(ctl);
    close(hand);
    close(fd);
    return data;
```

```
error:
    if(data>=0)
        close(data);
    if(ctl>=0)
        close(ctl);
    if(hand>=0)
        close(hand);
    return -1;
}
```

Uses `close()`, `finished()` 489c, `fprint()` 75a, `open()`, `read()` 192a, and `sprint()` 75b.

Part IV

Concurrency and Communication

This part covers synchronization and inter-process communication: locks and queuing locks for shared-memory processes, the IPC mechanisms (notes, pipes, shared segments, 9P marshalling), the cooperative thread scheduler and channel system in `libthread`, and the network API. The progression mirrors the layering: low-level atomic operations at the bottom, channels and threads at the top.

Chapter 21

Concurrency

When processes share memory via `rfork(RFMEM)`, they need synchronization primitives to ensure thread safety (as discussed in the Principles section of the Overview). These are the primitives that make `malloc`, `atexit`, and other `libc` functions safe for concurrent use. The higher-level abstractions (channels and cooperative threads) are built on top of these in `libthread`, covered in a later chapter.

⟨*signatures atomic functions* 243a⟩≡ (368b)
```
extern long    ainc(long*);
extern long    adec(long*);
```

⟨*signatures* Lock *functions* 243b⟩≡ (368b)
```
extern  void    lock(Lock*);
extern  void    unlock(Lock*);
extern  int     canlock(Lock*);
```

⟨*signatures* QLock *functions* 243c⟩≡ (368b)
```
extern  void    qlock(QLock*);
extern  void    qunlock(QLock*);
extern  int     canqlock(QLock*);
```

⟨*signatures* RWLock *functions* 243d⟩≡ (368b)
```
extern  void    rlock(RWLock*);
extern  void    runlock(RWLock*);
extern  int     canrlock(RWLock*);
extern  void    wlock(RWLock*);
extern  void    wunlock(RWLock*);
extern  int     canwlock(RWLock*);
```

⟨*signatures* Rendez *functions* 243e⟩≡ (368b)
```
extern  void    rsleep(Rendez*);    /* unlocks r->l, sleeps, locks r->l again */
extern  int     rwakeup(Rendez*);
```

⟨*signatures private vars functions* 243f⟩≡ (368b)
```
extern  void**  privalloc(void);
extern  void    privfree(void**);
```

The primitives above form a hierarchy. `ainc`/`adec` are atomic increment/decrement (lock-free). `Lock` is a spin lock (busy-waits). `QLock` ("queuing lock") blocks the caller instead of spinning—preferred for anything held longer than a few instructions. `RWLock` allows multiple concurrent readers or one exclusive writer. `Rendez` is a condition variable tied to a `QLock`: `rsleep` atomically releases the lock and sleeps, `rwakeup` wakes one sleeper. The `can*` variants are non-blocking: they return 1 if the lock was acquired, 0 otherwise. `privalloc`/`privfree` provide per-process private storage for shared-memory processes (like thread-local storage), useful for things like per-process error strings.

21.1 Syscalls: `rendezvous()`, etc

These syscalls provide kernel-level synchronization for shared-memory processes.

⟨*signatures concurrency syscall* 244a⟩≡ (379b) 244b ▷
```
extern void* rendezvous(void*, void*);
```

⟨*signatures concurrency syscall* 244b⟩+≡ (379b) ◁ 244a
```
extern int semacquire(long*, int);
extern long semrelease(long*, long);
extern int tsemacquire(long*, ulong);
```

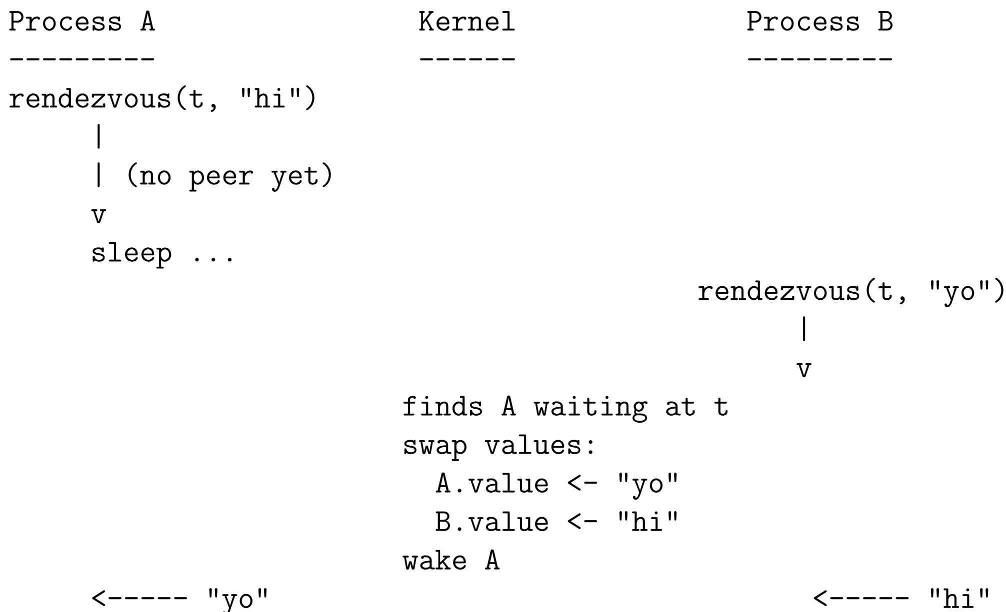
The semaphore syscalls operate on a `long` in shared memory. In `semacquire(addr, block)`, the `int` parameter controls blocking: 1 to wait, 0 to try and fail immediately. `tsemacquire` adds a timeout in milliseconds. `semrelease(addr, delta)` increments the semaphore by `delta` (usually 1) and wakes waiting processes.

In `rendezvous(tag, value)`, `tag` is an arbitrary pointer used as a meeting-point key, and `value` is the data to exchange. Two processes call `rendezvous` with the same tag; the first to arrive sleeps, and when the second arrives, they swap their `value` pointers and both wake up. For example, the `Lock` implementation uses the lock's address as the tag. `semacquire` and `semrelease` implement counting semaphores in shared memory. `semacquire` atomically decrements the semaphore; if it would go negative, the process sleeps until another process calls `semrelease`. `tsemacquire` is a variant with a timeout. These are the primitives on which `Lock` and `QLock` (shown later) are built.

UNIX has no equivalent of `rendezvous`. The closest primitives are `futex` (Linux), which also allows user-space fast paths with kernel fallback, but `futex` is more complex (it operates on memory addresses rather than abstract tags). Plan 9's `semacquire`/`semrelease` are similar in spirit to Linux's `futex WAIT` and `WAKE` operations.

The `rendezvous` system call is Plan 9's core synchronization primitive—two processes calling `rendezvous` with the same tag exchange a pointer value and both unblock. It is simpler than UNIX futexes but serves the same purpose: efficient kernel-mediated blocking when a userspace spin would waste CPU cycles. All the higher-level primitives (`QLock`, `Rendez`, channels) are built on `rendezvous` plus atomic operations.

A sequence diagram makes the meet-and-swap behavior concrete. Suppose two processes `A` and `B` both want to meet at tag `t`:



The first arrival blocks; the second arrival completes both calls atomically with the values swapped. Note that this is fundamentally a two-party rendezvous, not a broadcast: a third process arriving at `t` would have to wait

for the next pair. The library exploits this in QLock by using each QLp node's address as a unique tag, so the unlocker's rendezvous picks exactly the head waiter.

There is one subtlety the loop while((*_rendezvousp)(mp, ...) == (void*)~0) is guarding against. The rendezvous syscall returns (void*) 0 (all bits set, an otherwise impossible pointer) when it was interrupted by a note before the peer arrived. The loop simply retries in that case. This is the same pattern as the EINTR retry loop around UNIXread, but expressed with a sentinel pointer because rendezvous returns a void*, not an int errno.

21.2 Locks

The Lock uses a two-level strategy similar to Linux futexes: ainc (atomic increment) tries to acquire the lock without entering the kernel. If the lock was uncontested (key went from 0 to 1), we're done. Only if another process already holds it do we call semacquire to block in the kernel. This fast path avoids the cost of a system call in the common uncontested case.

⟨*type* Lock 245a⟩≡ (368b)
```
struct Lock {
    long    key;
    long    sem;
};
```

⟨*function* lock 245b⟩≡ (398e)
```
void
lock(Lock *l)
{
    if(ainc(&l->key) == 1)
        return; /* changed from 0 -> 1: we hold lock */

    /* otherwise wait in kernel */
    while(semacquire(&l->sem, 1) < 0){
        /* interrupted; try again */
    }
}
```
Uses semacquire().

⟨*function* unlock 245c⟩≡ (398e)
```
void
unlock(Lock *l)
{
    if(adec(&l->key) == 0)
        return; /* changed from 1 -> 0: no contention */
    semrelease(&l->sem, 1);
}
```
Uses semrelease().

⟨*function* canlock 245d⟩≡ (398e)
```
bool
canlock(Lock *l)
{
    if(ainc(&l->key) == 1)
        return true;   /* changed from 0 -> 1: success */

    /* Undo increment (but don't miss wakeup) */
    if(adec(&l->key) == 0)
        return false;   /* changed from 1 -> 0: no contention */
    semrelease(&l->sem, 1);
    return false;
```

```
}
```
Uses `semrelease()`.

21.3 Atomic operations: `ainc()` (ARM)

`ainc` and `adec` (atomic increment/decrement) are implemented in ARM assembly using the `LDREX`/`STREX` (load-exclusive/store-exclusive) instructions, which provide lock-free atomic read-modify-write on the ARM architecture. These are the foundation of all higher-level synchronization: `Lock` uses them for the fast path, and `Ref` uses them for reference counting.

21.4 Waiting queues

`QLock` (queuing lock) is a higher-level lock that maintains an explicit queue of waiters. Unlike `Lock` which may spin, `QLock` puts contending threads to sleep in FIFO order, ensuring fairness. The `QLp` nodes form a linked list of waiting processes.

⟨*type* `QLock` 246a⟩≡ (368b)
```
struct QLock {
    Lock    lock;
    int locked;

    QLp *head;
    QLp     *tail;
};
```

⟨*type* `QLp` 246b⟩≡ (368b)
```
struct QLp {
    char    state;
    int inuse;
    // Extra
    QLp *next;
};
```

⟨*function* `qlock` 246c⟩≡ (491a)
```
void
qlock(QLock *q)
{
    QLp *p, *mp;

    lock(&q->lock);
    if(!q->locked){
        q->locked = 1;
        unlock(&q->lock);
        return;
    }


    /* chain into waiting list */
    mp = getqlp();
    p = q->tail;
    if(p == nil)
        q->head = mp;
    else
        p->next = mp;
    q->tail = mp;
    mp->state = Queuing;
```

```
	unlock(&q->lock);

	/* wait */
	while((*_rendezvousp)(mp, (void*)1) == (void*)~0)
		;
	mp->inuse = 0;
}
```

Uses `Queuing-249` 490c, `_rendezvousp-253` 490d, `getqlp()` 490f, `lock()` 245b, and `unlock()` 245c.

The `qlock` function first tries the fast path: if the lock is free (`!q->locked`), it grabs it and returns. Otherwise, it allocates a `QLp` node, appends it to the wait queue, and blocks via the `rendezvous` system call. The `_rendezvousp` indirection allows `libthread` to replace the kernel `rendezvous` with its own thread-aware version.

⟨*function* `qunlock` 247a⟩≡ (491a)

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

	lock(&q->lock);
	if (q->locked == 0)
		fprint(2, "qunlock called with qlock not held, from %#p\n",
			getcallerpc(&q));
	p = q->head;
	if(p != nil){
		/* wakeup head waiting process */
		q->head = p->next;
		if(q->head == nil)
			q->tail = nil;
		unlock(&q->lock);
		while((*_rendezvousp)(p, (void*)0x12345) == (void*)~0)
			;
		return;
	}
	q->locked = 0;
	unlock(&q->lock);
}
```

Uses `_rendezvousp-253` 490d, `fprint()` 75a, `getcallerpc()` 396f, `lock()` 245b, and `unlock()` 245c.

⟨*function* `canqlock` 247b⟩≡ (491a)

```
int
canqlock(QLock *q)
{
	if(!canlock(&q->lock))
		return 0;
	if(!q->locked){
		q->locked = 1;
		unlock(&q->lock);
		return 1;
	}
	unlock(&q->lock);
	return 0;
}
```

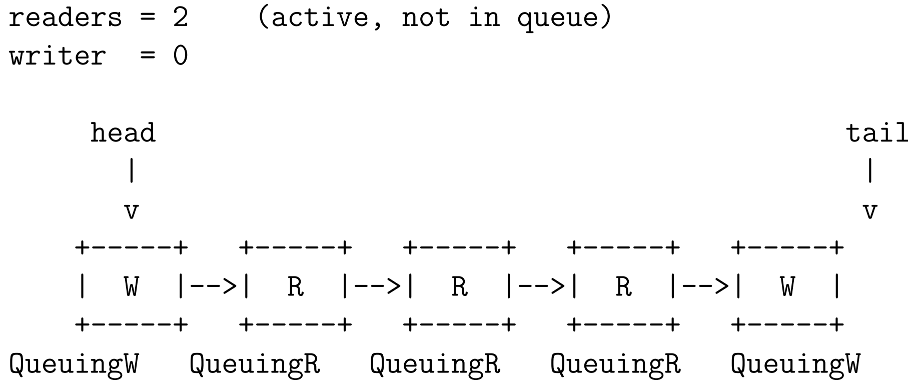
Uses `canlock()` 245d and `unlock()` 245c.

21.5 Read-write locks

`RWLock` allows multiple concurrent readers but exclusive writers. When a writer is waiting, new readers queue be-

hind it to prevent writer starvation. The `wunlock` function wakes either the next writer or a batch of consecutive readers from the wait queue.

Here is what the wait queue looks like in a typical contention scenario—two readers are active, a writer is parked behind them, and three late readers and another writer have piled up after that writer:



When the two active readers call `runlock()`[249b], the last one to leave wakes the head of the queue. Since the head is a writer, exactly one thread is released, and it becomes the lock holder. When that writer eventually calls `wunlock()`[250b], the code walks the queue as long as the head is a `QueuingR` node, releasing them all at once: the three parked readers run in parallel, and the trailing `QueuingW` stays parked. This policy prevents writer starvation (readers arriving during a writer's tenure cannot slip ahead of it) while still letting a burst of readers share the lock when one becomes available.

⟨*type* `RWLock` 248a⟩≡ (368b)

```
struct RWLock {
    Lock    lock;
    int readers;    /* number of readers */
    int writer;     /* number of writers */

    QLp *head;      /* list of waiting processes */
    QLp *tail;
};
```

⟨*function* `rlock` 248b⟩≡ (491a)

```
void
rlock(RWLock *q)
{
    QLp *p, *mp;

    lock(&q->lock);
    if(q->writer == 0 && q->head == nil){
        /* no writer, go for it */
        q->readers++;
        unlock(&q->lock);
        return;
    }

    mp = getqlp();
    p = q->tail;
    if(p == 0)
        q->head = mp;
    else
        p->next = mp;
    q->tail = mp;
    mp->next = nil;
    mp->state = QueuingR;
    unlock(&q->lock);
```

```
    /* wait in kernel */
    while((*_rendezvousp)(mp, (void*)1) == (void*)~0)
        ;
    mp->inuse = 0;
}
```

Uses QueuingR-250 490c, _rendezvousp-253 490d, getqlp() 490f, lock() 245b, and unlock() 245c.

⟨*function* canrlock 249a⟩≡ (491a)

```
int
canrlock(RWLock *q)
{
    lock(&q->lock);
    if (q->writer == 0 && q->head == nil) {
        /* no writer; go for it */
        q->readers++;
        unlock(&q->lock);
        return 1;
    }
    unlock(&q->lock);
    return 0;
}
```

Uses lock() 245b and unlock() 245c.

⟨*function* runlock 249b⟩≡ (491a)

```
void
runlock(RWLock *q)
{
    QLp *p;

    lock(&q->lock);
    if(q->readers <= 0)
        abort();
    p = q->head;
    if(--(q->readers) > 0 || p == nil){
        unlock(&q->lock);
        return;
    }

    /* start waiting writer */
    if(p->state != QueuingW)
        abort();
    q->head = p->next;
    if(q->head == 0)
        q->tail = 0;
    q->writer = 1;
    unlock(&q->lock);

    /* wakeup waiter */
    while((*_rendezvousp)(p, 0) == (void*)~0)
        ;
}
```

Uses QueuingW-251 490c, _rendezvousp-253 490d, abort() 356d, lock() 245b, and unlock() 245c.

⟨*function* wlock 249c⟩≡ (491a)

```
void
wlock(RWLock *q)
{
    QLp *p, *mp;
```

```
    lock(&q->lock);
    if(q->readers == 0 && q->writer == 0){
        /* noone waiting, go for it */
        q->writer = 1;
        unlock(&q->lock);
        return;
    }

    /* wait */
    p = q->tail;
    mp = getqlp();
    if(p == nil)
        q->head = mp;
    else
        p->next = mp;
    q->tail = mp;
    mp->next = nil;
    mp->state = QueuingW;
    unlock(&q->lock);

    /* wait in kernel */
    while((*_rendezvousp)(mp, (void*)1) == (void*)~0)
        ;
    mp->inuse = 0;
}
```

Uses QueuingW-251 490c, _rendezvousp-253 490d, getqlp() 490f, lock() 245b, and unlock() 245c.

⟨*function* canwlock 250a⟩≡ (491a)

```
int
canwlock(RWLock *q)
{
    lock(&q->lock);
    if (q->readers == 0 && q->writer == 0) {
        /* no one waiting; go for it */
        q->writer = 1;
        unlock(&q->lock);
        return 1;
    }
    unlock(&q->lock);
    return 0;
}
```

Uses lock() 245b and unlock() 245c.

⟨*function* wunlock 250b⟩≡ (491a)

```
void
wunlock(RWLock *q)
{
    QLp *p;

    lock(&q->lock);
    if(q->writer == 0)
        abort();
    p = q->head;
    if(p == nil){
        q->writer = 0;
        unlock(&q->lock);
        return;
    }
    if(p->state == QueuingW){
```

```
        /* start waiting writer */
        q->head = p->next;
        if(q->head == nil)
            q->tail = nil;
        unlock(&q->lock);
        while((*_rendezvousp)(p, 0) == (void*)~0)
            ;
        return;
    }

    if(p->state != QueuingR)
        abort();

    /* wake waiting readers */
    while(q->head != nil && q->head->state == QueuingR){
        p = q->head;
        q->head = p->next;
        q->readers++;
        while((*_rendezvousp)(p, 0) == (void*)~0)
            ;
    }
    if(q->head == nil)
        q->tail = nil;
    q->writer = 0;
    unlock(&q->lock);
}
```

Uses `QueuingR-250` 490c, `QueuingW-251` 490c, `_rendezvousp-253` 490d, `abort()` 356d, `lock()` 245b, and `unlock()` 245c.

21.6 Rendez-vous

`Rendez` is Plan 9's condition variable: a process can sleep on a `Rendez` (via `rsleep`) until another process wakes it (via `rwakeup`). Unlike POSIX condition variables, a `Rendez` is tied to a specific `QLock`—the caller must hold that lock when calling `rsleep`, and the lock is atomically released while sleeping and reacquired on wakeup. The key subtlety in `rsleep` is that it passes the `QLock` to the next waiter before going to sleep, preventing the lock from being held by a sleeping process.

⟨*type* `Rendez` 251a⟩≡ (368b)

```
struct Rendez {
    QLock   *l;

    QLp *head;
    QLp *tail;
};
```

⟨*function* `rsleep` 251b⟩≡ (491a)

```
void
rsleep(Rendez *r)
{
    QLp *t, *me;

    if(!r->l)
        abort();
    lock(&r->l->lock);
    /* we should hold the qlock */
    if(!r->l->locked)
        abort();
```

```
    /* add ourselves to the wait list */
    me = getqlp();
    me->state = Sleeping;
    if(r->head == nil)
        r->head = me;
    else
        r->tail->next = me;
    me->next = nil;
    r->tail = me;

    /* pass the qlock to the next guy */
    t = r->l->head;
    if(t){
        r->l->head = t->next;
        if(r->l->head == nil)
            r->l->tail = nil;
        unlock(&r->l->lock);
        while((*_rendezvousp)(t, (void*)0x12345) == (void*)~0)
            ;
    }else{
        r->l->locked = 0;
        unlock(&r->l->lock);
    }

    /* wait for a wakeup */
    while((*_rendezvousp)(me, (void*)1) == (void*)~0)
        ;
    me->inuse = 0;
}
```

Uses Sleeping-252 490c, _rendezvousp-253 490d, abort() 356d, getqlp() 490f, lock() 245b, and unlock() 245c.

⟨*function* rwakeup 252⟩≡ (491a)

```
int
rwakeup(Rendez *r)
{
    QLp *t;

    /*
     * take off wait and put on front of queue
     * put on front so guys that have been waiting will not get starved
     */

    if(!r->l)
        abort();
    lock(&r->l->lock);
    if(!r->l->locked)
        abort();

    t = r->head;
    if(t == nil){
        unlock(&r->l->lock);
        return 0;
    }

    r->head = t->next;
    if(r->head == nil)
        r->tail = nil;

    t->next = r->l->head;
    r->l->head = t;
```

```
    if(r->l->tail == nil)
        r->l->tail = t;

    t->state = Queuing;
    unlock(&r->l->lock);
    return 1;
}
```

Uses `Queuing-249` 490c, `abort()` 356d, `lock()` 245b, and `unlock()` 245c.

⟨*function* `rwakeupall` 253a⟩≡ (491a)

```
int
rwakeupall(Rendez *r)
{
    int i;

    for(i=0; rwakeup(r); i++)
        ;
    return i;
}
```

Uses `rwakeup()` 252.

21.7 Coroutines

`jmp_buf` is the minimal state needed for a context switch: just the stack pointer and program counter (two `long` values on ARM). `setjmp` saves SP and PC; `longjmp` restores them. This is the mechanism behind both `libthread`'s cooperative scheduling and user-level exception handling. The cooperative thread scheduler uses `setjmp`/`longjmp` to switch between threads without involving the kernel.

⟨*type* `jmp_buf` 253b⟩≡ (368a)

```
typedef long    jmp_buf[2];
```

⟨*type* `jmpbufxxx` 253c⟩≡ (368a)

```
#define JMPBUFSP    0
#define JMPBUFPC    1
#define JMPBUFDPC   0
```

Chapter 22

IPC

Plan 9 provides several IPC mechanisms: pipes for byte streams, notes (signals) for asynchronous notification, the `await()` system call for parent-child communication via exit strings, and shared memory segments. This chapter also covers the 9P protocol marshalling functions used to communicate with file servers.

⟨*signatures ipc helpers* 254a⟩≡ (379b)
```
extern  char*   getenv(char*);
extern  int     putenv(char*, char*);
```

Note that Plan 9's `putenv` takes two arguments (name, value) rather than the single `"name=value"` string of UNIX.

22.1 Syscalls: `pipe()`, `notify()`, `await()`, `segattach()`

These syscalls provide the remaining IPC mechanisms.

⟨*signatures ipc syscall* 254b⟩≡ (379b) 254c ▷
```
extern int pipe(int*);
```

⟨*signatures ipc syscall* 254c⟩+≡ (379b) ◁ 254b 254d ▷
```
extern int noted(int);
extern int notify(void(*)(void*, char*));
```

⟨*signatures ipc syscall* 254d⟩+≡ (379b) ◁ 254c
```
extern void* segattach(int, char*, void*, ulong);
extern void* segbrk(void*, void*);
extern int segdetach(void*);
extern int segflush(void*, ulong);
extern int segfree(void*, ulong);
```

`pipe(fd)` fills `fd[0]` and `fd[1]` with two file descriptors for a bidirectional pipe. `notify(handler)` registers a function to be called when the process receives a note. The handler receives a pointer to the saved registers and the note string: `int handler(void *ureg, char *note)`. `noted(v)` tells the kernel what to do after the handler returns (`NCONT` to continue, `NDFLT` to terminate). In `segattach(attr, class, va, len)`: `class` is a name like `"shared"` that identifies the segment type, `va` is the desired virtual address (0 to let the kernel choose), and `len` is the size in bytes. `segdetach` unmaps it, `segbrk` extends it, `segflush` invalidates cached pages, and `segfree` releases pages without detaching.

Plan 9's pipes are bidirectional—both ends can read and write—unlike UNIX pipes which are unidirectional (you need two pipes for two-way communication). Notes are Plan 9's replacement for UNIX signals. The key improvement is that notes carry a string message (e.g., `"alarm"` or `"sys: write on closed pipe"`) rather than just a number (`SIGALRM`, `SIGPIPE`). The handler also receives a string, not just a signal number, so it can make decisions based on the message content. `segattach` is Plan 9's equivalent of UNIX's `mmap` for shared memory (though not for file mapping). Unlike `mmap`, it uses named segment classes rather than anonymous mappings.

22.2 Helpers

`getfields` splits a string into fields by replacing separator characters with null bytes—a destructive tokenizer used to parse text-based protocols. The `mflag` parameter controls whether consecutive separators produce empty fields (like `awk`'s default) or are merged into one (like `awk -F`).

⟨*function* `getfields` 255a⟩≡ (397a)

```
int
getfields(char *str, char **args, int max, int mflag, char *set)
{
    Rune r;
    int nr, intok, narg;

    if(max <= 0)
        return 0;

    narg = 0;
    args[narg] = str;
    if(!mflag)
        narg++;
    intok = 0;
    for(;; str += nr) {
        nr = chartorune(&r, str);
        if(r == 0)
            break;
        if(utfrune(set, r)) {
            if(narg >= max)
                break;
            *str = 0;
            intok = 0;
            args[narg] = str + nr;
            if(!mflag)
                narg++;
        } else {
            if(!intok && mflag)
                narg++;
            intok = 1;
        }
    }
    return narg;
}
```

Uses `chartorune()` 84d and `utfrune()` 89b.

22.3 Parent-child IPC

In Plan 9, parent-child communication is asymmetric: the parent passes data to the child via environment variables (files in `/env/`), and the child reports its exit status back to the parent via the `await` system call, which returns a text string containing the child's pid, CPU times, and exit message.

22.3.1 Wait message

`Waitmsg` wraps the text returned by `await` into a structured form. The `wait` function parses the five whitespace-separated fields (pid, three time values, and the exit message) and allocates the `Waitmsg` and its string data in a single `malloc` call, so the caller only needs one `free`.

⟨*type* `Waitmsg` 255b⟩≡ (379b)

```
/* keep /sys/src/ape/lib/ap/plan9/sys9.h in sync with this -rsc */
struct Waitmsg {
 int pid;  /* of loved one */
 ulong time[3]; /* of loved one & descendants */
 // ref_own?<string>?
 char *msg;
};
```

⟨*function* `wait` 256a⟩≡ (499a)

```
Waitmsg*
wait(void)
{
    int n, l;
    char buf[512];
    char *fld[5];
    Waitmsg *w;

    n = await(buf, sizeof buf-1);
    if(n < 0)
        return nil;
    buf[n] = '\0';
    if(tokenize(buf, fld, nelem(fld)) != nelem(fld)){
        werrstr("couldn't parse wait message");
        return nil;
    }
    l = strlen(fld[4])+1;
    w = malloc(sizeof(Waitmsg)+l);
    if(w == nil)
        return nil;
    w->pid = atoi(fld[0]);
    w->time[0] = atoi(fld[1]);
    w->time[1] = atoi(fld[2]);
    w->time[2] = atoi(fld[3]);
    w->msg = (char*)&w[1];
    memmove(w->msg, fld[4], l);
    return w;
}
```

Uses `atoi()` 123b, `await()`, `malloc()` 63g, `memmove()` 48, `strlen()` 80a, `tokenize()` 454b, and `werrstr()` 234b.

⟨*function* `waitpid` 256b⟩≡ (499b)

```
int
waitpid(void)
{
    int n;
    char buf[512];
    char *fld[5];

    n = await(buf, sizeof buf-1);
    if(n <= 0)
        return -1;
    buf[n] = '\0';
    if(tokenize(buf, fld, nelem(fld)) != nelem(fld)){
        werrstr("couldn't parse wait message");
        return -1;
    }
    return atoi(fld[0]);
}
```

Uses `atoi()` 123b, `await()`, `tokenize()` 454b, and `werrstr()` 234b.

22.3.2 Environment

In Plan 9, environment variables are files under `/env/`. `getenv` opens `/env/name`, reads its contents, and returns a malloc'd string. Multi-valued variables (like `path`) store values separated by null bytes; `getenv` replaces those nulls with spaces to present a single string. `putenv` creates or overwrites the file. Because the environment is part of the namespace, `rfork(RFCENVG)` gives a child a clean environment, and `rfork(RFENVG)` gives it a copy.

⟨*function* getenv 257a⟩≡ (483b)
```
char*
getenv(char *name)
{
    int r;
    fdt f;
    long s;
    char *ans;
    char *p, *ep, ename[100];

    if(strchr(name, '/') != nil)
        return nil;
    snprint(ename, sizeof ename, "/env/%s", name);
    if(strcmp(ename+5, name) != 0)
        return nil;
    f = open(ename, OREAD);
    if(f < 0)
        return 0;
    s = seek(f, 0, 2);
    ans = malloc(s+1);
    if(ans) {
        setmalloctag(ans, getcallerpc(&name));
        seek(f, 0, 0);
        r = read(f, ans, s);
        if(r >= 0) {
            ep = ans + s - 1;
            for(p = ans; p < ep; p++)
                if(*p == '\0')
                    *p = ' ';
            ans[s] = '\0';
        }
    }
    close(f);
    return ans;
}
```
Uses `close()`, `getcallerpc()` 396f, `malloc()` 63g, `open()`, `read()` 192a, `seek()`, `setmalloctag()` 69a, `snprint()` 535b, `strchr()` 80c, and `strcmp()` 81a.

⟨*function* putenv 257b⟩≡ (490a)
```
int
putenv(char *name, char *val)
{
    fdt f;
    char ename[100];
    long s;

    if(strchr(name, '/') != nil)
        return -1;
    snprint(ename, sizeof ename, "/env/%s", name);
    if(strcmp(ename+5, name) != 0)
        return -1;
    f = create(ename, OWRITE, 0664);
    if(f < 0)
```

```
        return -1;
    s = strlen(val);
    if(write(f, val, s) != s){
        close(f);
        return -1;
    }
    close(f);
    return 0;
}
```

Uses close(), create(), snprint() 535b, strchr() 80c, strcmp() 81a, strlen() 80a, and write() 192b.

22.4 Notes

Notes are Plan 9's equivalent of UNIX signals, but they carry a string message instead of a number. When a note arrives, the handler can return `NCONT` (continue execution), `NDFLT` (terminate), `NSAVE` (save state for later restoration), or `NRSTR` (restore saved state). The string payload makes notes more informative than signals—a note might say `"alarm"` or `"sys: write on closed pipe"` rather than just `SIGALRM` or `SIGPIPE`.

⟨*type* Note_flag 258a⟩≡ (379b)

```
#define NCONT 0 /* continue after note */
#define NDFLT 1 /* terminate after note */
#define NSAVE 2 /* clear note but hold state */
#define NRSTR 3 /* restore saved state */
```

22.4.1 Posting

`postnote` sends a note to a process or process group by writing to `/proc/pid/note` or `/proc/pid/notepg`. This is the file-system-based equivalent of UNIX's `kill`: instead of a system call taking a signal number, you write a string to a file.

⟨*type* PostnoteKind 258b⟩≡ (368b)

```
enum
{
    PNPROC      = 1,
    PNGROUP     = 2,
};
```

⟨*function* postnote 258c⟩≡ (487g)

```
int
postnote(int group, int pid, char *note)
{
    char file[128];
    fdt f;
    int r;

    switch(group) {
    case PNPROC:
        sprint(file, "/proc/%d/note", pid);
        break;
    case PNGROUP:
        sprint(file, "/proc/%d/notepg", pid);
        break;
    default:
        return -1;
    }
```

```
    f = open(file, OWRITE);
    if(f < 0)
        return -1;

    r = strlen(note);
    if(write(f, note, r) != r) {
        close(f);
        return -1;
    }
    close(f);
    return 0;
}
```

Uses close(), open(), sprint() 75b, strlen() 80a, and write() 192b.

22.4.2 Notified

atnotify registers note handlers, analogous to UNIX's **signal** function. Up to 33 handlers can be registered. The actual note-handling function, **notifier**, tries each registered handler in order; the first one that returns non-zero "handles" the note and execution continues. If no handler claims the note, the default action (NDFLT) terminates the process.

⟨*function* atnotify 259⟩≡ (389a)

```
int
atnotify(int (*f)(void*, char*), int in)
{
    int i, n, ret;
    static bool init;

    if(!init){
        notify(notifier);
        init = true;        /* assign = */
    }
    ret = 0;
    lock(&onnotlock);
    if(in){
        for(i=0; i<NFN; i++)
            if(onnot[i] == 0) {
                onnot[i] = f;
                ret = 1;
                break;
            }
    }else{
        n = 0;
        for(i=0; i<NFN; i++)
            if(onnot[i]){
                if(ret==0 && onnot[i]==f){
                    onnot[i] = nil;
                    ret = 1;
                }else
                    n++;
            }
        if(n == 0){
            init = false;
            notify(0);
        }
    }
    unlock(&onnotlock);
    return ret;
```

```
}
```
Uses `NFN-59` 260a, `lock()` 245b, `notifier()` 260d, `notify()`, `onnot-60` 260b, `onnotlock-61` 260c, and `unlock()` 245c.

⟨*constant NFN (port/atnotify.c)* 260a⟩≡ (389a)
```
#define NFN 33
```

⟨*global* `onnot` 260b⟩≡ (389a)
```
static  int (*onnot[NFN])(void*, char*);
```
Uses `NFN-59` 260a.

⟨*global* `onnotlock` 260c⟩≡ (389a)
```
static  Lock    onnotlock;
```

⟨*function* `notifier` 260d⟩≡ (389a)
```
static
void
notifier(void *v, char *s)
{
    int i;

    for(i=0; i<NFN; i++)
        if(onnot[i] && ((*onnot[i])(v, s))){
            noted(NCONT);
            return;
        }
    noted(NDFLT);
}
```
Uses `NFN-59` 260a, `noted()`, and `onnot-60` 260b.

22.5 Misc

⟨*function* `notejmp` *(arm)* 260e⟩≡ (538d)
```
void
notejmp(void *vr, jmp_buf j, int ret)
{
    struct Ureg *r = vr;

    r->r0 = ret;
    if(ret == 0)
        r->r0 = 1;
    r->pc = j[JMPBUFPC];
    r->r13 = j[JMPBUFSP];
    noted(NCONT);
}
```
Uses `noted()`.

22.6 Pipes

Plan 9 pipes are bidirectional (unlike UNIX's unidirectional pipes) and carry structured 9P messages when used with `mount`. The `pipe` system call returns two file descriptors that are endpoints of a two-way communication channel.

22.7 Shared segments

`segattach` maps a shared memory segment into the process's address space. The `SG_RONLY` flag makes it read-only, and `SG_CEXEC` detaches it on `exec`. Shared segments are the low-level mechanism underlying `rfork(RFMEM)`'s memory sharing between processes.

⟨*type* `Segattach_flag` 261a⟩≡ (379b)
```
/* Segattch */
#define SG_RONLY 0040 /* read only */
#define SG_CEXEC 0100 /* detach on exec */
```

22.8 Fileservers

9P is the protocol that Plan 9 file servers speak. Every file operation—open, read, write, stat, walk—is a message in this protocol. The `Fcall` structure represents a single 9P message, and the `convM2S`/`convS2M` functions convert between the wire format (a byte stream) and this C structure. Programs that implement file servers (like `rio`, `exportfs`, or user-level file systems) use these functions heavily.

22.8.1 `Fcall`

`Fcall` uses a C union to overlay the fields of all message types into a single structure. Each 9P message type (Tversion, Tattach, Twalk, Tread, etc.) uses a different subset of fields. The `type` field selects the variant, the `fid` identifies the file handle, and `tag` is a transaction identifier that lets clients match responses to requests in the face of concurrent operations.

⟨*type* `Fcall` 261b⟩≡ (386)
```
struct  Fcall
{
    // enum<FcallType>
    uchar   type;

    u32int  fid;
    ushort  tag;

    union {
        struct {
            u32int  msize;      /* Tversion, Rversion */
            char    *version;   /* Tversion, Rversion */
        };
        struct {
            ushort  oldtag;     /* Tflush */
        };
        struct {
            char    *ename;     /* Rerror */
        };
        struct {
            Qid qid;             /* Rattach, Ropen, Rcreate */
            u32int  iounit;     /* Ropen, Rcreate */
        };
        struct {
            Qid aqid;       /* Rauth */
        };
        struct {
            u32int  afid;       /* Tauth, Tattach */
            char    *uname;     /* Tauth, Tattach */
            char    *aname;     /* Tauth, Tattach */
```

```
        };
        struct {
            u32int  perm;       /* Tcreate */
            char    *name;      /* Tcreate */
            uchar   mode;       /* Tcreate, Topen */
        };
        struct {
            u32int  newfid;     /* Twalk */
            ushort  nwname;     /* Twalk */
            char    *wname[MAXWELEM];   /* Twalk */
        };
        struct {
            ushort  nwqid;      /* Rwalk */
            Qid wqid[MAXWELEM];     /* Rwalk */
        };
        struct {
            vlong   offset;     /* Tread, Twrite */
            u32int  count;      /* Tread, Twrite, Rread */
            char    *data;      /* Twrite, Rread */
        };
        struct {
            ushort  nstat;      /* Twstat, Rstat */
            uchar   *stat;      /* Twstat, Rstat */
        };
    };
};
```

⟨*type* FcallType 262⟩≡ (386)

```
enum FcallType
{
    Tversion =  100,
    Rversion,
    Tauth =     102,
    Rauth,
    Tattach =   104,
    Rattach,
    Terror =    106,    /* illegal */
    Rerror,
    Tflush =    108,
    Rflush,
    Twalk =     110,
    Rwalk,
    Topen =     112,
    Ropen,
    Tcreate =   114,
    Rcreate,
    Tread =     116,
    Rread,
    Twrite =    118,
    Rwrite,
    Tclunk =    120,
    Rclunk,
    Tremove =   122,
    Rremove,
    Tstat =     124,
    Rstat,
    Twstat =    126,
    Rwstat,

    Tmax,
```

```
};
```

⟨*constant* MAXWELEM 263a⟩≡ (386)

```
#define MAXWELEM    16
```

22.8.2 Dumping

`fcallfmt` is a custom format verb (registered as `%F`) that prints a human-readable representation of any 9P message. Each message type has its own format string showing the relevant fields. This function is invaluable for debugging file servers: you can trace every 9P message with a single `fprint(2, "%F\backslash n", \&f)`.

⟨*constant* QIDFMT 263b⟩≡ (482c)

```
#define QIDFMT  "(%.16llux %lud %s)"
```

⟨*function* fcallfmt 263c⟩≡ (482c)

```
int
fcallfmt(Fmt *fmt)
{
    Fcall *f;
    int fid, type, tag, i;
    char buf[512], tmp[200];
    char *p, *e;
    Dir *d;
    Qid *q;

    e = buf+sizeof(buf);
    f = va_arg(fmt->args, Fcall*);

    type = f->type;
    fid = f->fid;
    tag = f->tag;

    switch(type){
    case Tversion:  /* 100 */
        seprint(buf, e, "Tversion tag %ud msize %ud version '%s'", tag, f->msize, f->version);
        break;
    case Rversion:
        seprint(buf, e, "Rversion tag %ud msize %ud version '%s'", tag, f->msize, f->version);
        break;
    case Tauth: /* 102 */
        seprint(buf, e, "Tauth tag %ud afid %d uname %s aname %s", tag,
            f->afid, f->uname, f->aname);
        break;
    case Rauth:
        seprint(buf, e, "Rauth tag %ud qid " QIDFMT, tag,
            f->aqid.path, f->aqid.vers, qidtype(tmp, f->aqid.type));
        break;
    case Tattach:   /* 104 */
        seprint(buf, e, "Tattach tag %ud fid %d afid %d uname %s aname %s", tag,
            fid, f->afid, f->uname, f->aname);
        break;
    case Rattach:
        seprint(buf, e, "Rattach tag %ud qid " QIDFMT, tag,
            f->qid.path, f->qid.vers, qidtype(tmp, f->qid.type));
        break;
    case Rerror:    /* 107; 106 (Terror) illegal */
        seprint(buf, e, "Rerror tag %ud ename %s", tag, f->ename);
        break;
    case Tflush:    /* 108 */
```

```
	seprint(buf, e, "Tflush tag %ud oldtag %ud", tag, f->oldtag);
	break;
case Rflush:
	seprint(buf, e, "Rflush tag %ud", tag);
	break;
case Twalk: /* 110 */
	p = seprint(buf, e, "Twalk tag %ud fid %d newfid %d nwname %d ", tag, fid, f->newfid, f->nwname);
	if(f->nwname <= MAXWELEM)
		for(i=0; i<f->nwname; i++)
			p = seprint(p, e, "%d:%s ", i, f->wname[i]);
	break;
case Rwalk:
	p = seprint(buf, e, "Rwalk tag %ud nwqid %ud ", tag, f->nwqid);
	if(f->nwqid <= MAXWELEM)
		for(i=0; i<f->nwqid; i++){
			q = &f->wqid[i];
			p = seprint(p, e, "%d:" QIDFMT " ", i,
				q->path, q->vers, qidtype(tmp, q->type));
		}
	break;
case Topen: /* 112 */
	seprint(buf, e, "Topen tag %ud fid %ud mode %d", tag, fid, f->mode);
	break;
case Ropen:
	seprint(buf, e, "Ropen tag %ud qid " QIDFMT " iounit %ud ", tag,
		f->qid.path, f->qid.vers, qidtype(tmp, f->qid.type), f->iounit);
	break;
case Tcreate:	/* 114 */
	seprint(buf, e, "Tcreate tag %ud fid %ud name %s perm %M mode %d", tag, fid, f->name, (ulong)f->perm, f
	break;
case Rcreate:
	seprint(buf, e, "Rcreate tag %ud qid " QIDFMT " iounit %ud ", tag,
		f->qid.path, f->qid.vers, qidtype(tmp, f->qid.type), f->iounit);
	break;
case Tread: /* 116 */
	seprint(buf, e, "Tread tag %ud fid %d offset %lld count %ud",
		tag, fid, f->offset, f->count);
	break;
case Rread:
	p = seprint(buf, e, "Rread tag %ud count %ud ", tag, f->count);
		dumpsome(p, e, f->data, f->count);
	break;
case Twrite:	/* 118 */
	p = seprint(buf, e, "Twrite tag %ud fid %d offset %lld count %ud ",
		tag, fid, f->offset, f->count);
	dumpsome(p, e, f->data, f->count);
	break;
case Rwrite:
	seprint(buf, e, "Rwrite tag %ud count %ud", tag, f->count);
	break;
case Tclunk:	/* 120 */
	seprint(buf, e, "Tclunk tag %ud fid %ud", tag, fid);
	break;
case Rclunk:
	seprint(buf, e, "Rclunk tag %ud", tag);
	break;
case Tremove:	/* 122 */
	seprint(buf, e, "Tremove tag %ud fid %ud", tag, fid);
	break;
case Rremove:
```

```
        seprint(buf, e, "Rremove tag %ud", tag);
        break;
    case Tstat: /* 124 */
        seprint(buf, e, "Tstat tag %ud fid %ud", tag, fid);
        break;
    case Rstat:
        p = seprint(buf, e, "Rstat tag %ud ", tag);
        if(f->nstat > sizeof tmp)
            seprint(p, e, " stat(%d bytes)", f->nstat);
        else{
            d = (Dir*)tmp;
            convM2D(f->stat, f->nstat, d, (char*)(d+1));
            seprint(p, e, " stat ");
            fdirconv(p+6, e, d);
        }
        break;
    case Twstat:    /* 126 */
        p = seprint(buf, e, "Twstat tag %ud fid %ud", tag, fid);
        if(f->nstat > sizeof tmp)
            seprint(p, e, " stat(%d bytes)", f->nstat);
        else{
            d = (Dir*)tmp;
            convM2D(f->stat, f->nstat, d, (char*)(d+1));
            seprint(p, e, " stat ");
            fdirconv(p+6, e, d);
        }
        break;
    case Rwstat:
        seprint(buf, e, "Rwstat tag %ud", tag);
        break;
    default:
        seprint(buf, e,  "unknown type %d", type);
    }
    return fmtstrcpy(fmt, buf);
}
```

Uses `convM2D()` 280, `dumpsome()` 482b, `fdirconv()` 481d, `fmtstrcpy()` 505a, `qidtype()` 265, and `seprint()` 534c.

⟨*function* `qidtype` 265⟩≡ (482c)

```
static char*
qidtype(char *s, uchar t)
{
    char *p;

    p = s;
    if(t & QTDIR)
        *p++ = 'd';

    if(t & QTAPPEND)
        *p++ = 'a';
    if(t & QTEXCL)
        *p++ = 'l';
    if(t & QTAUTH)
        *p++ = 'A';

    *p = '\0';
    return s;
}
```

22.8.3 Marshalling macros

The GBIT/PBIT macros read and write integers in little-endian byte order, the wire format of the 9P protocol. They work byte-at-a-time to be portable across architectures with different native byte orders. The BIT8SZ through BIT64SZ constants give the wire sizes. IOHDRSZ is the maximum overhead a 9P message header can add beyond the payload data.

⟨*macros GBITxxx* 266a⟩≡ (386)
```
#define GBIT8(p)    ((p)[0])
#define GBIT16(p)   ((p)[0]|((p)[1]<<8))
#define GBIT32(p)   ((p)[0]|((p)[1]<<8)|((p)[2]<<16)|((p)[3]<<24))
#define GBIT64(p)   ((u32int)((p)[0]|((p)[1]<<8)|((p)[2]<<16)|((p)[3]<<24)) |\
                ((vlong)((p)[4]|((p)[5]<<8)|((p)[6]<<16)|((p)[7]<<24)) << 32))
```

⟨*macros PBITxxx* 266b⟩≡ (386)
```
#define PBIT8(p,v)  (p)[0]=(v)
#define PBIT16(p,v) do{(p)[0]=(v);(p)[1]=(v)>>8;}while(0)
#define PBIT32(p,v) do{(p)[0]=(v);(p)[1]=(v)>>8;(p)[2]=(v)>>16;(p)[3]=(v)>>24;}while(0)
#define PBIT64(p,v) do{(p)[0]=(v);(p)[1]=(v)>>8;(p)[2]=(v)>>16;(p)[3]=(v)>>24;\
            (p)[4]=(v)>>32;(p)[5]=(v)>>40;(p)[6]=(v)>>48;(p)[7]=(v)>>56;}while(0)
```

⟨*macros BITxxx* 266c⟩≡ (386)
```
#define BIT8SZ      1
#define BIT16SZ     2
#define BIT32SZ     4
#define BIT64SZ     8
```

⟨*constant* IOHDRSZ 266d⟩≡ (386)
```
#define IOHDRSZ     24  /* ample room for Twrite/Rread header (iounit) */
```

22.8.4 readp9msg()

read9pmsg reads a complete 9P message from a file descriptor. It first reads the 4-byte size header, then reads exactly that many remaining bytes with readn. The two-step read is necessary because 9P is a length-prefixed protocol: you must read the size before you know how much data to expect.

⟨*function* read9pmsg 266e⟩≡ (491c)
```
error0
read9pmsg(fdt fd, void *abuf, uint n)
{
    int m, len;
    uchar *buf;

    buf = abuf;

    /* read count */
    m = readn(fd, buf, BIT32SZ);
    if(m != BIT32SZ){
        if(m < 0)
            return ERROR_NEG1;
        return ERROR_0;
    }

    len = GBIT32(buf);
    if(len <= BIT32SZ || len > n){
        werrstr("bad length in 9P2000 message header");
        return ERROR_NEG1;
    }
    len -= BIT32SZ;
```

```
    m = readn(fd, buf+BIT32SZ, len);
    if(m < len)
        return ERROR_0;
    return BIT32SZ+m;
}
```

Uses `readn()` 192c and `werrstr()` 234b.

22.8.5 Parsing

String to message, `convM2S()`

Every 9P message on the wire shares a 7-byte preamble: a 4-byte `size` (total message length including the size field itself), a 1-byte `type`, and a 2-byte `tag`. The remainder depends on `type`. Here is a `Twrite` message asking to write 5 bytes at offset 100 on `fid` 4:

```
offset  bytes  field       value
------  -----  ----------  -------------
 0       4     size        24
 4       1     type        118 (Twrite)
 5       2     tag         7
 7       4     fid         4
11       8     offset      100
19       4     count       5
23       5     data        "hello"
                           -------
                           24 bytes total
```

All integers are little-endian, independent of the host architecture—this is what `GBIT32`/`PBIT32` encode. Strings on the wire are a 2-byte length followed by the raw bytes (no null terminator), which is why the parser cannot just cast the buffer into an `Fcall`.

`convM2S` deserializes a 9P message from its wire format into an `Fcall` structure. The function is careful about buffer overruns: every field extraction checks that enough bytes remain before reading. String fields are handled by `gstring`, which performs an in-place trick: it shifts the string data down by one byte to overwrite the length prefix, making room for a null terminator—so the strings in the `Fcall` point directly into the input buffer with no extra allocation.

⟨*function* `convM2S` 267⟩≡ (466c)

```
/*
 * no syntactic checks.
 * three causes for error:
 *  1. message size field is incorrect
 *  2. input buffer too short for its own data (counts too long, etc.)
 *  3. too many names or qids
 * gqid() and gstring() return nil if they would reach beyond buffer.
 * main switch statement checks range and also can fall through
 * to test at end of routine.
 */
error0
convM2S(uchar *ap, uint nap, Fcall *f)
{
    uchar *p, *ep;
    uint i, size;

    p = ap;
    ep = p + nap;
```

```
if(p+BIT32SZ+BIT8SZ+BIT16SZ > ep)
    return ERROR_0;
// redo work done in read9pmsg
size = GBIT32(p);
p += BIT32SZ;

if(size < BIT32SZ+BIT8SZ+BIT16SZ)
    return ERROR_0;

f->type = GBIT8(p);
p += BIT8SZ;
f->tag = GBIT16(p);
p += BIT16SZ;

switch(f->type)
{
case Tversion:
    if(p+BIT32SZ > ep)
        return ERROR_0;
    f->msize = GBIT32(p);
    p += BIT32SZ;
    p = gstring(p, ep, &f->version);
    break;

case Tflush:
    if(p+BIT16SZ > ep)
        return ERROR_0;
    f->oldtag = GBIT16(p);
    p += BIT16SZ;
    break;

case Tauth:
    if(p+BIT32SZ > ep)
        return ERROR_0;
    f->afid = GBIT32(p);
    p += BIT32SZ;
    p = gstring(p, ep, &f->uname);
    if(p == nil)
        break;
    p = gstring(p, ep, &f->aname);
    if(p == nil)
        break;
    break;

case Tattach:
    if(p+BIT32SZ > ep)
        return ERROR_0;
    f->fid = GBIT32(p);
    p += BIT32SZ;
    if(p+BIT32SZ > ep)
        return ERROR_0;
    f->afid = GBIT32(p);
    p += BIT32SZ;
    p = gstring(p, ep, &f->uname);
    if(p == nil)
        break;
    p = gstring(p, ep, &f->aname);
    if(p == nil)
        break;
```

```
		break;

	case Twalk:
		if(p+BIT32SZ+BIT32SZ+BIT16SZ > ep)
			return ERROR_0;
		f->fid = GBIT32(p);
		p += BIT32SZ;
		f->newfid = GBIT32(p);
		p += BIT32SZ;
		f->nwname = GBIT16(p);
		p += BIT16SZ;
		if(f->nwname > MAXWELEM)
			return ERROR_0;
		for(i=0; i<f->nwname; i++){
			p = gstring(p, ep, &f->wname[i]);
			if(p == nil)
				break;
		}
		break;

	case Topen:
		if(p+BIT32SZ+BIT8SZ > ep)
			return ERROR_0;
		f->fid = GBIT32(p);
		p += BIT32SZ;
		f->mode = GBIT8(p);
		p += BIT8SZ;
		break;

	case Tcreate:
		if(p+BIT32SZ > ep)
			return ERROR_0;
		f->fid = GBIT32(p);
		p += BIT32SZ;
		p = gstring(p, ep, &f->name);
		if(p == nil)
			break;
		if(p+BIT32SZ+BIT8SZ > ep)
			return ERROR_0;
		f->perm = GBIT32(p);
		p += BIT32SZ;
		f->mode = GBIT8(p);
		p += BIT8SZ;
		break;

	case Tread:
		if(p+BIT32SZ+BIT64SZ+BIT32SZ > ep)
			return ERROR_0;
		f->fid = GBIT32(p);
		p += BIT32SZ;
		f->offset = GBIT64(p);
		p += BIT64SZ;
		f->count = GBIT32(p);
		p += BIT32SZ;
		break;

	case Twrite:
		if(p+BIT32SZ+BIT64SZ+BIT32SZ > ep)
			return ERROR_0;
		f->fid = GBIT32(p);
```

```
		p += BIT32SZ;
		f->offset = GBIT64(p);
		p += BIT64SZ;
		f->count = GBIT32(p);
		p += BIT32SZ;
		if(p+f->count > ep)
			return ERROR_0;
		f->data = (char*)p;
		p += f->count;
		break;

	case Tclunk:
	case Tremove:
		if(p+BIT32SZ > ep)
			return ERROR_0;
		f->fid = GBIT32(p);
		p += BIT32SZ;
		break;

	case Tstat:
		if(p+BIT32SZ > ep)
			return ERROR_0;
		f->fid = GBIT32(p);
		p += BIT32SZ;
		break;

	case Twstat:
		if(p+BIT32SZ+BIT16SZ > ep)
			return ERROR_0;
		f->fid = GBIT32(p);
		p += BIT32SZ;
		f->nstat = GBIT16(p);
		p += BIT16SZ;
		if(p+f->nstat > ep)
			return ERROR_0;
		f->stat = p;
		p += f->nstat;
		break;

/*
 */
	case Rversion:
		if(p+BIT32SZ > ep)
			return ERROR_0;
		f->msize = GBIT32(p);
		p += BIT32SZ;
		p = gstring(p, ep, &f->version);
		break;

	case Rerror:
		p = gstring(p, ep, &f->ename);
		break;

	case Rflush:
		break;

	case Rauth:
		p = gqid(p, ep, &f->aqid);
		if(p == nil)
			break;
```

```
    break;

case Rattach:
    p = gqid(p, ep, &f->qid);
    if(p == nil)
        break;
    break;

case Rwalk:
    if(p+BIT16SZ > ep)
        return ERROR_0;
    f->nwqid = GBIT16(p);
    p += BIT16SZ;
    if(f->nwqid > MAXWELEM)
        return ERROR_0;
    for(i=0; i<f->nwqid; i++){
        p = gqid(p, ep, &f->wqid[i]);
        if(p == nil)
            break;
    }
    break;

case Ropen:
case Rcreate:
    p = gqid(p, ep, &f->qid);
    if(p == nil)
        break;
    if(p+BIT32SZ > ep)
        return ERROR_0;
    f->iounit = GBIT32(p);
    p += BIT32SZ;
    break;

case Rread:
    if(p+BIT32SZ > ep)
        return ERROR_0;
    f->count = GBIT32(p);
    p += BIT32SZ;
    if(p+f->count > ep)
        return ERROR_0;
    f->data = (char*)p;
    p += f->count;
    break;

case Rwrite:
    if(p+BIT32SZ > ep)
        return ERROR_0;
    f->count = GBIT32(p);
    p += BIT32SZ;
    break;

case Rclunk:
case Rremove:
    break;

case Rstat:
    if(p+BIT16SZ > ep)
        return ERROR_0;
    f->nstat = GBIT16(p);
    p += BIT16SZ;
```

```
        if(p+f->nstat > ep)
            return ERROR_0;
        f->stat = p;
        p += f->nstat;
        break;

    case Rwstat:
        break;
    default:
        return ERROR_0;
    }

    if(p==nil || p>ep)
        return ERROR_0;
    if(ap+size == p)
        return size;
    return ERROR_0;
}
```

Uses gqid() 272b and gstring() 272a.

⟨*function* gstring 272a⟩≡ (466c)

```
static
uchar*
gstring(uchar *p, uchar *ep, char **s)
{
    uint n;

    if(p+BIT16SZ > ep)
        return nil;
    n = GBIT16(p);
    p += BIT16SZ - 1;
    if(p+n+1 > ep)
        return nil;
    /* move it down, on top of count, to make room for '\0' */
    memmove(p, p + 1, n);
    p[n] = '\0';
    *s = (char*)p;
    p += n+1;
    return p;
}
```

Uses memmove() 48.

⟨*function* gqid 272b⟩≡ (466c)

```
static
uchar*
gqid(uchar *p, uchar *ep, Qid *q)
{
    if(p+QIDSZ > ep)
        return nil;
    q->type = GBIT8(p);
    p += BIT8SZ;
    q->vers = GBIT32(p);
    p += BIT32SZ;
    q->path = GBIT64(p);
    p += BIT64SZ;
    return p;
}
```

Message to string, `convS2M()`

`convS2M` serializes an `Fcall` into the 9P wire format. It first calls `sizeS2M` to compute the exact output size, then writes the header (size, type, tag) and the type-specific fields. The `pstring` helper writes a length-prefixed string, with a subtle optimization: it copies the string data before writing the length, so you can serialize a struct whose string fields point into the output buffer itself (used when relaying 9P messages).

⟨*function* `convS2M` 273⟩≡ (466d)

```
error0
convS2M(Fcall *f, uchar *ap, uint nap)
{
    uchar *p;
    uint i, size;

    size = sizeS2M(f);
    if(size == 0)
        return ERROR_0;
    if(size > nap)
        return ERROR_0;

    p = (uchar*)ap;

    PBIT32(p, size);
    p += BIT32SZ;

    PBIT8(p, f->type);
    p += BIT8SZ;
    PBIT16(p, f->tag);
    p += BIT16SZ;

    switch(f->type)
    {
    case Tversion:
        PBIT32(p, f->msize);
        p += BIT32SZ;
        p = pstring(p, f->version);
        break;

    case Tflush:
        PBIT16(p, f->oldtag);
        p += BIT16SZ;
        break;

    case Tauth:
        PBIT32(p, f->afid);
        p += BIT32SZ;
        p  = pstring(p, f->uname);
        p  = pstring(p, f->aname);
        break;

    case Tattach:
        PBIT32(p, f->fid);
        p += BIT32SZ;
        PBIT32(p, f->afid);
        p += BIT32SZ;
        p  = pstring(p, f->uname);
        p  = pstring(p, f->aname);
        break;

    case Twalk:
```

```
	PBIT32(p, f->fid);
	p += BIT32SZ;
	PBIT32(p, f->newfid);
	p += BIT32SZ;
	PBIT16(p, f->nwname);
	p += BIT16SZ;
	if(f->nwname > MAXWELEM)
		return ERROR_0;
	for(i=0; i<f->nwname; i++)
		p = pstring(p, f->wname[i]);
	break;

case Topen:
	PBIT32(p, f->fid);
	p += BIT32SZ;
	PBIT8(p, f->mode);
	p += BIT8SZ;
	break;

case Tcreate:
	PBIT32(p, f->fid);
	p += BIT32SZ;
	p = pstring(p, f->name);
	PBIT32(p, f->perm);
	p += BIT32SZ;
	PBIT8(p, f->mode);
	p += BIT8SZ;
	break;

case Tread:
	PBIT32(p, f->fid);
	p += BIT32SZ;
	PBIT64(p, f->offset);
	p += BIT64SZ;
	PBIT32(p, f->count);
	p += BIT32SZ;
	break;

case Twrite:
	PBIT32(p, f->fid);
	p += BIT32SZ;
	PBIT64(p, f->offset);
	p += BIT64SZ;
	PBIT32(p, f->count);
	p += BIT32SZ;
	memmove(p, f->data, f->count);
	p += f->count;
	break;

case Tclunk:
case Tremove:
	PBIT32(p, f->fid);
	p += BIT32SZ;
	break;

case Tstat:
	PBIT32(p, f->fid);
	p += BIT32SZ;
	break;
```

```
    case Twstat:
        PBIT32(p, f->fid);
        p += BIT32SZ;
        PBIT16(p, f->nstat);
        p += BIT16SZ;
        memmove(p, f->stat, f->nstat);
        p += f->nstat;
        break;
/*
 */

    case Rversion:
        PBIT32(p, f->msize);
        p += BIT32SZ;
        p = pstring(p, f->version);
        break;

    case Rerror:
        p = pstring(p, f->ename);
        break;

    case Rflush:
        break;

    case Rauth:
        p = pqid(p, &f->aqid);
        break;

    case Rattach:
        p = pqid(p, &f->qid);
        break;

    case Rwalk:
        PBIT16(p, f->nwqid);
        p += BIT16SZ;
        if(f->nwqid > MAXWELEM)
            return ERROR_0;
        for(i=0; i<f->nwqid; i++)
            p = pqid(p, &f->wqid[i]);
        break;

    case Ropen:
    case Rcreate:
        p = pqid(p, &f->qid);
        PBIT32(p, f->iounit);
        p += BIT32SZ;
        break;

    case Rread:
        PBIT32(p, f->count);
        p += BIT32SZ;
        memmove(p, f->data, f->count);
        p += f->count;
        break;

    case Rwrite:
        PBIT32(p, f->count);
        p += BIT32SZ;
        break;
```

```
    case Rclunk:
        break;

    case Rremove:
        break;

    case Rstat:
        PBIT16(p, f->nstat);
        p += BIT16SZ;
        memmove(p, f->stat, f->nstat);
        p += f->nstat;
        break;

    case Rwstat:
        break;
    default:
        return ERROR_0;

    }
    if(size != p-ap)
        return ERROR_0;
    return size;
}
```

Uses `memmove()` 48, `pqid()` 276b, `pstring()` 276a, and `sizeS2M()` 277.

⟨*function* `pstring` 276a⟩≡ (466d)

```
static
uchar*
pstring(uchar *p, char *s)
{
    uint n;

    if(s == nil){
        PBIT16(p, 0);
        p += BIT16SZ;
        return p;
    }

    n = strlen(s);
    /*
     * We are moving the string before the length,
     * so you can S2M a struct into an existing message
     */
    memmove(p + BIT16SZ, s, n);
    PBIT16(p, n);
    p += n + BIT16SZ;
    return p;
}
```

Uses `memmove()` 48 and `strlen()` 80a.

⟨*function* `pqid` 276b⟩≡ (466d)

```
static
uchar*
pqid(uchar *p, Qid *q)
{
    PBIT8(p, q->type);
    p += BIT8SZ;
    PBIT32(p, q->vers);
    p += BIT32SZ;
    PBIT64(p, q->path);
```

```
    p += BIT64SZ;
    return p;
}
```

⟨*function* sizeS2M 277⟩≡ (466d)

```
uint
sizeS2M(Fcall *f)
{
    uint n;
    int i;

    n = 0;
    n += BIT32SZ;   /* size */
    n += BIT8SZ;    /* type */
    n += BIT16SZ;   /* tag */

    switch(f->type)
    {
    case Tversion:
        n += BIT32SZ;
        n += stringsz(f->version);
        break;

    case Tflush:
        n += BIT16SZ;
        break;

    case Tauth:
        n += BIT32SZ;
        n += stringsz(f->uname);
        n += stringsz(f->aname);
        break;

    case Tattach:
        n += BIT32SZ;
        n += BIT32SZ;
        n += stringsz(f->uname);
        n += stringsz(f->aname);
        break;

    case Twalk:
        n += BIT32SZ;
        n += BIT32SZ;
        n += BIT16SZ;
        for(i=0; i<f->nwname; i++)
            n += stringsz(f->wname[i]);
        break;

    case Topen:
        n += BIT32SZ;
        n += BIT8SZ;
        break;

    case Tcreate:
        n += BIT32SZ;
        n += stringsz(f->name);
        n += BIT32SZ;
        n += BIT8SZ;
        break;
```

```
    case Tread:
        n += BIT32SZ;
        n += BIT64SZ;
        n += BIT32SZ;
        break;

    case Twrite:
        n += BIT32SZ;
        n += BIT64SZ;
        n += BIT32SZ;
        n += f->count;
        break;

    case Tclunk:
    case Tremove:
        n += BIT32SZ;
        break;

    case Tstat:
        n += BIT32SZ;
        break;

    case Twstat:
        n += BIT32SZ;
        n += BIT16SZ;
        n += f->nstat;
        break;
/*
 */

    case Rversion:
        n += BIT32SZ;
        n += stringsz(f->version);
        break;

    case Rerror:
        n += stringsz(f->ename);
        break;

    case Rflush:
        break;

    case Rauth:
        n += QIDSZ;
        break;

    case Rattach:
        n += QIDSZ;
        break;

    case Rwalk:
        n += BIT16SZ;
        n += f->nwqid*QIDSZ;
        break;

    case Ropen:
    case Rcreate:
        n += QIDSZ;
        n += BIT32SZ;
        break;
```

```
    case Rread:
        n += BIT32SZ;
        n += f->count;
        break;

    case Rwrite:
        n += BIT32SZ;
        break;

    case Rclunk:
        break;

    case Rremove:
        break;

    case Rstat:
        n += BIT16SZ;
        n += f->nstat;
        break;

    case Rwstat:
        break;
    default:
        return 0;

    }
    return n;
}
```
Uses `stringsz()` 279a.

⟨*function* `stringsz` 279a⟩≡ (466d)
```
static
uint
stringsz(char *s)
{
    if(s == nil)
        return BIT16SZ;

    return BIT16SZ+strlen(s);
}
```
Uses `strlen()` 80a.

String to directory entry, `convM2D()`

`convM2D` deserializes a `Dir` from the 9P stat format. The stat format is a fixed-length header (type, dev, qid, mode, times, length) followed by four length-prefixed strings (name, uid, gid, muid). The `strs` parameter provides scratch space where the string data is copied, so the resulting `Dir`'s string pointers are valid as long as that buffer lives.

⟨*constant* `QIDSZ` 279b⟩≡ (386)
```
#define QIDSZ   (BIT8SZ+BIT32SZ+BIT64SZ)
```

⟨*constant* `STATFIXLEN` 279c⟩≡ (386)
```
/* STATFIXLEN includes leading 16-bit count */
/* The count, however, excludes itself; total size is BIT16SZ+count */
#define STATFIXLEN  (BIT16SZ+QIDSZ+5*BIT16SZ+4*BIT32SZ+1*BIT64SZ)   /* amount of fixed length data in a stat bu
```

⟨*function* convM2D 280⟩≡ (466b)

```
uint
convM2D(uchar *buf, uint nbuf, Dir *d, char *strs)
{
    uchar *p, *ebuf;
    char *sv[4];
    int i, ns;

    if(nbuf < STATFIXLEN)
        return 0;

    p = buf;
    ebuf = buf + nbuf;

    p += BIT16SZ;   /* ignore size */
    d->type = GBIT16(p);
    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);
    p += BIT64SZ;

    for(i = 0; i < 4; i++){
        if(p + BIT16SZ > ebuf)
            return 0;
        ns = GBIT16(p);
        p += BIT16SZ;
        if(p + ns > ebuf)
            return 0;
        if(strs){
            sv[i] = strs;
            memmove(strs, p, ns);
            strs += ns;
            *strs++ = '\0';
        }
        p += ns;
    }

    if(strs){
        d->name = sv[0];
        d->uid = sv[1];
        d->gid = sv[2];
        d->muid = sv[3];
    }else{
        d->name = nullstring;
        d->uid = nullstring;
        d->gid = nullstring;
        d->muid = nullstring;
```

```
    }

    return p - buf;
}
```
Uses `memmove()` 48 and `nullstring-259` 466a.

Directory entry to string, `convD2M()`

⟨*function* `convD2M` 281⟩≡ (465b)
```
uint
convD2M(Dir *d, uchar *buf, uint nbuf)
{
    uchar *p, *ebuf;
    char *sv[4];
    int i, ns, nsv[4], ss;

    if(nbuf < BIT16SZ)
        return 0;

    p = buf;
    ebuf = buf + nbuf;

    sv[0] = d->name;
    sv[1] = d->uid;
    sv[2] = d->gid;
    sv[3] = d->muid;

    ns = 0;
    for(i = 0; i < 4; i++){
        if(sv[i])
            nsv[i] = strlen(sv[i]);
        else
            nsv[i] = 0;
        ns += nsv[i];
    }

    ss = STATFIXLEN + ns;

    /* set size before erroring, so user can know how much is needed */
    /* note that length excludes count field itself */
    PBIT16(p, ss-BIT16SZ);
    p += BIT16SZ;

    if(ss > nbuf)
        return BIT16SZ; // ugly error code

    PBIT16(p, d->type);
    p += BIT16SZ;
    PBIT32(p, d->dev);
    p += BIT32SZ;

    PBIT8(p, d->qid.type);
    p += BIT8SZ;
    PBIT32(p, d->qid.vers);
    p += BIT32SZ;
    PBIT64(p, d->qid.path);
    p += BIT64SZ;

    PBIT32(p, d->mode);
```

```
    p += BIT32SZ;
    PBIT32(p, d->atime);
    p += BIT32SZ;
    PBIT32(p, d->mtime);
    p += BIT32SZ;

    PBIT64(p, d->length);
    p += BIT64SZ;

    for(i = 0; i < 4; i++){
        ns = nsv[i];
        if(p + ns + BIT16SZ > ebuf)
            return 0;
        PBIT16(p, ns);
        p += BIT16SZ;
        if(ns)
            memmove(p, sv[i], ns);
        p += ns;
    }

    if(ss != p - buf)
        return 0;

    return p - buf;
}
```

Uses `memmove()` 48 and `strlen()` 80a.

Chapter 23

Channels and Cooperative Threads

`libthread` provides cooperative threads ("green threads") and channels for structured concurrency, directly inspiring Go's goroutines and channels. In Plan 9's terminology, a "proc" is an OS-level process (created with `rfork(RFMEM)`), while a "thread" is a cooperatively-scheduled coroutine within a proc. Because threads within the same proc never run concurrently, they don't need locks to access shared data—a major simplification over preemptive threading. Channels provide type-safe, blocking communication between threads, and `alt` enables waiting on multiple channels simultaneously (like Go's `select`).

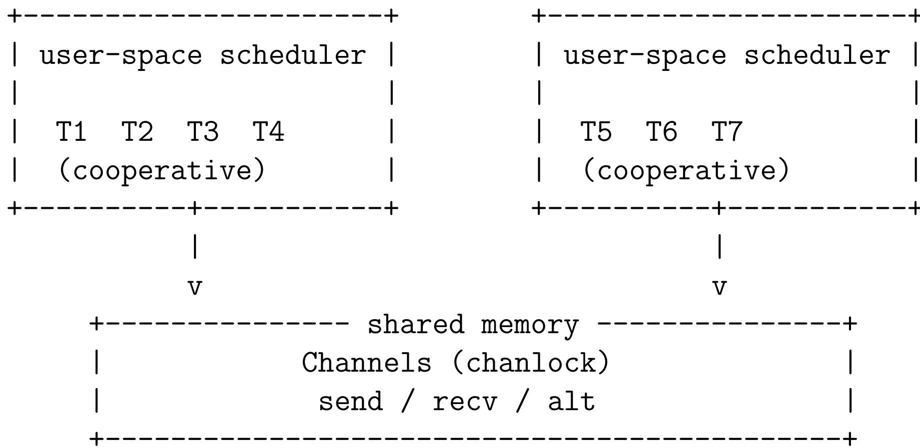
23.1 Concurrency model principles

Concurrency is the art of letting multiple logical flows of control make progress "at the same time." Every concurrency system has to answer two questions: how do flows *communicate*, and how do they *synchronize*. The two dominant answers form two philosophies. *Shared-memory* concurrency (C/POSIX threads, Java, C++ threads, C#, Rust's default `std::thread`) lets every flow see a common heap; communication is just reads and writes, and synchronization is via locks, condition variables, and atomic operations—simple to set up, notoriously hard to get right. *Message-passing* concurrency (Erlang, Go, occam, Ada tasks, Rust with channels, and Hoare's original CSP paper from 1978) gives each flow private state and requires explicit messages over channels; no locks, no barriers, no atomic primitives in user code. The price is that everything must flow through the channel, even when sharing would be natural.

Orthogonal to the communication model is the question of *how* the flows are actually scheduled onto CPUs. *Preemptive OS threads* (Linux `pthreads`, Windows threads, FreeBSD threads) are scheduled by the kernel: each thread is a full kernel-schedulable entity, and a timer interrupt can yank the CPU at any instruction, so every shared-memory access is a potential race. *Cooperative user-space threads*, also known as *green threads*, are scheduled entirely in user space by a library: they only yield at explicit synchronization points—a `send` or `recv` on an empty channel, a `sleep`, or an explicit `yield`. Cooperative threads within one OS process never run concurrently, so they sidestep the race-condition problem for anything they don't share across OS-process boundaries: no locks, no atomics, no memory barriers in user code. The price is that a thread hogging the CPU blocks all its siblings, and blocking on a slow syscall freezes everyone unless the library arranges to hand that syscall off to a helper. Green threads are an old idea: Modula-2 coroutines, Saber C, Stackless Python, Lua coroutines, OCaml's `lwt`, and early Go all used variants of it.

Plan 9's `libthread` picks a specific and unusually productive point in this design space: *message passing for communication, cooperative scheduling for responsiveness, and multiple OS procs for parallelism.* Inside a single proc (a full OS-level process created with `rfork(RFMEM)`), threads never race because they are cooperative. Across procs, threads really are parallel; communication between them goes through channels, which are therefore shared-memory objects protected by one global lock. This is the classic *M:N threading* design—M user threads mapped onto N OS threads—with a user-space scheduler handling the fine-grained context switches:

```
 Proc 1  (rfork RFMEM)            Proc 2  (rfork RFMEM)
```




```
Within a proc:  only one thread runs at a time (cooperative)
Across procs:   real parallelism via rfork RFMEM
```

The same design was later carried into the early Go runtime by Rob Pike and Russ Cox, and the direct language lineage runs Plan 9 → Newsqueak (Pike) → Alef (Winterbottom) → `libthread` → Go channels—which is why Go's `go func()` and `<-chan` syntax feel so closely related to the `send` and `recv` below. (Go later added preemptive goroutine scheduling in its runtime, for latency reasons, but the M:N shape is unchanged.)

`libthread` itself is split into a cooperative scheduler, channel operations (including the multi-channel `alt`, analogous to Go's `select`), proc and thread creation, and a set of wrappers that turn blocking system calls into channel operations, so that one cooperative thread's disk read doesn't freeze every other thread in its proc. The public API is `threadcreate`, `proccreate`, `send`, `recv`, and `alt`; everything with an `_` prefix is internal. The rest of this chapter walks these pieces in that order, starting with a small producer/consumer program that exercises all five of the public functions at once.

23.1.1 The return of coroutines

The cooperative threads in `libthread` are part of a much older story than they might appear. *Coroutines*—functions that can suspend and resume, yielding control to each other without returning—were invented by Melvin Conway in 1958 for a COBOL compiler (the same Conway who later gave us "Conway's Law": organizations produce systems that mirror their own communication structure). The idea is that two pieces of code can take turns running, each maintaining its own local state, without either being "the caller" or "the callee." Simula (1967) used coroutines for simulation; Modula-2 (1978) had explicit coroutine primitives; and many early concurrency experiments were built on cooperative switching.

Preemptive *kernel threads* largely displaced coroutines in the 1980s and 1990s. POSIX `pthreads` (standardized 1995), Java threads (1995), and Windows threads gave programmers real parallelism on SMP hardware, and the "just lock your shared data" programming model seemed workable. But as multi-core machines became standard in the 2000s, the difficulty of writing correct shared-memory concurrent code became apparent: data races, deadlocks, priority inversions, and subtle memory-ordering bugs turned out to be so common that many experienced engineers began calling threads a fundamentally broken abstraction for application-level concurrency.

The response, starting around 2010, was a *return to coroutines*—repackaged in modern clothing. C#'s `async/await` (2012) lets the programmer write sequential-looking code that the compiler transforms into a state machine of coroutine resumptions. Python adopted the same syntax in 2015; JavaScript's `async/await` followed in 2017; Rust added it in 2019. Go's goroutines (2009) are explicitly coroutines multiplexed onto OS threads by a user-space scheduler—the exact M:N model that Plan 9's `libthread` pioneered in the early 1990s. The insight that makes the comeback work is *channels* (or their equivalent): coroutines alone give you cooperative scheduling, but coroutines *plus channels* give you a structured way to communicate between concurrent flows

without shared mutable state—which is what makes the bugs go away. Hoare's CSP paper (1978) proposed exactly this combination, and the lineage from CSP through Plan 9's `libthread` to Go is a forty-year arc from theory to mainstream practice.

23.1.2 Events, threads, and the colored-function problem

For most applications, plain OS threads with blocking I/O work perfectly well. A web server handling a few hundred concurrent connections can dedicate one thread to each, block on `read` when waiting for data, and let the kernel schedule. Modern Linux handles thousands of threads with reasonable memory, and for the vast majority of programs this is the simplest, most readable, and most debuggable model. The push toward alternatives came from a specific scale requirement: Dan Kegel's 1999 "C10K problem" asked how a single server could handle ten thousand simultaneous connections. At that scale, one-thread-per-connection costs too much memory (each stack is 1–8 MB). The *event model*—a single thread running a non-blocking I/O loop with callbacks (`nginx`, Node.js, Redis)—solved the memory problem but created "callback hell": deeply nested, inside-out code that is hard to read. John Ousterhout's 1996 talk "Why Threads Are A Bad Idea (for most purposes)" argued for events; von Behren, Condit, and Brewer's 2003 paper "Why Events Are A Bad Idea" argued back. Both had a point.

Async/await (C# 2012, Python 2015, JavaScript 2017, Rust 2019) was supposed to be the synthesis: event-loop performance with sequential-looking code. It largely delivered—except for one structural flaw that Bob Nystrom crystallized in his 2015 blog post *"What Color Is Your Function?"*. In an async/await language, every function is either "sync" or "async." A sync function *cannot* call an async function without becoming async itself, and this "color" propagates up the entire call stack. The result is two parallel worlds that do not compose: libraries must be duplicated (sync and async variants), error handling diverges, and the boundary between the two colors is a persistent source of bugs. This is not a flaw in any particular implementation; it is inherent in the async/await model itself.

The most promising recent answer is *algebraic effects*, as implemented in OCaml 5's `eio` library (2022). An effect handler sits at a boundary in the call stack and intercepts operations like "read from this file descriptor" without the code *performing* the read knowing whether it will block, be scheduled cooperatively, or run asynchronously. The function simply calls `read`—no `async`, no `await`, no coloring—and the handler decides. This gives event-loop scalability with genuinely uncolored sequential code. Plan 9's `libthread` was reaching for the same goal two decades earlier: its I/O proc mechanism (a dedicated OS process performing blocking syscalls on behalf of cooperative threads) exists precisely so that thread code can call blocking `read` and `write` without freezing siblings—sequential code, scheduling concern handled separately. The effect-handler model generalizes this idea from a runtime trick into a language-level primitive.

23.2 Overview

23.2.1 Code organization

The source files group naturally along the five pieces listed in the principles section above: `sched.c` is the cooperative scheduler, `channel.c` the channel operations and `alt`, `create.c` the `threadcreate/proccreate` machinery, and the `io*.c` files the blocking-syscall wrappers. The public API lives in `thread.h`; private implementation details are in `threadimpl.h`.

23.2.2 Software architecture

Two implementation details worth naming up front. First, the user-space scheduler uses `setjmp/longjmp` for the context switch between threads, so no hand-written assembly is needed. Second, channels live in shared

memory and are protected by one global `chanlock`; cross-proc wakeups go through the `rendezvous` system call introduced in the IPC chapter.

23.3 A toy example

Before diving into the data structures, here is a complete `libthread` program that exercises the main features—channels, `threadcreate`, and the `threadmain` entry point. The program forks a producer thread and a consumer thread that communicate through a 4-element buffered channel:

```
#include <u.h>
#include <libc.h>
#include <thread.h>

enum { Nmsg = 5 };

static void
producer(void *arg)
{
	Channel *c = arg;
	int i;

	for(i = 1; i <= Nmsg; i++){
		print("producer: sending %d\n", i);
		send(c, &i);
	}
	chanclose(c);
	threadexits(nil);
}

static void
consumer(void *arg)
{
	Channel *c = arg;
	int v;

	while(recv(c, &v) > 0)
		print("consumer: got %d\n", v);
	threadexits(nil);
}

void
threadmain(int argc, char *argv[])
{
	Channel *c;

	c = chancreate(sizeof(int), 4);
	threadcreate(producer, c, 8192);
	threadcreate(consumer, c, 8192);
	threadexits(nil);
}
```

Notice that `main` does not exist—`libthread` replaces it with `threadmain` (we will see how in the `libthread/main.c` section). The runtime starts a single proc, runs `threadmain` inside it, and only when `threadmain` returns does the proc tear down. The two `threadcreate` calls do *not* create OS processes; they create coroutines that share the proc's address space and run cooperatively. Because both threads live in the same proc, accessing the channel does not need any explicit lock—only `chanlock` inside the library.

Here is the rough sequence of events when this program runs. Each box is a scheduler step; the arrows show which thread is running and what it is doing:

```
threadmain          producer            consumer        channel c
----------          --------            --------        ---------
chancreate(int,4) --------------------------------> [_,_,_,_]
threadcreate(producer)
threadcreate(consumer)
threadexits ----.
                |
                v
                send 1                                [1,_,_,_]
                send 2                                [1,2,_,_]
                send 3                                [1,2,3,_]
                send 4                                [1,2,3,4]
                send 5 (full, block)
                                    recv -> 1         [2,3,4,_]
                                    recv -> 2         [3,4,_,_]
                send 5 (wakes)                        [3,4,5,_]
                chanclose(c)        recv -> 3         [4,5,_,_]
                                    recv -> 4         [5,_,_,_]
                                    recv -> 5         [_,_,_,_]
                                    recv -> 0 (closed)
                                    threadexits
```

The block-on-full and block-on-empty cases are exactly where `libthread`'s scheduler kicks in: the blocked thread does a `longjmp` back to the proc's scheduler context, which then picks the next ready thread. With one proc and two threads, the scheduling is trivial—there is always at most one ready thread at any moment, and the "switch" is just a `setjmp/longjmp` pair. With multiple procs (using `proccreate` instead), real OS parallelism kicks in and the `chanlock` starts to matter.

23.4 Core data structures

23.4.1 Concurrency buiding blocks

`Ref` is a thread-safe reference counter. `incref` and `decref` use atomic operations (`ainc/adec`) so they work correctly even when called from different procs sharing memory. `decref` returns the new count, so the caller can free the object when it reaches zero.

⟨*struct* `Ref` 287a⟩≡ (384b)

```
struct Ref {
    long ref;
};
```

⟨*function* `incref` 287b⟩≡ (558a)

```
void
incref(Ref *r)
```

```
{
    ainc(&r->ref);
}
```

⟨*function* decref 288a⟩≡ (558a)

```
long
decref(Ref *r)
{
    return adec(&r->ref);
}
```

⟨*global* xincport_lock 288b⟩≡ (560b)

```
static Lock xincport_lock;
```

⟨*function* _xinc 288c⟩≡ (560b)

```
void
_xinc(long *p)
{

    lock(&xincport_lock);
    (*p)++;
    unlock(&xincport_lock);
}
```

Uses `lock()` 245b and `unlock()` 245c.

⟨*function* _xdec 288d⟩≡ (560b)

```
long
_xdec(long *p)
{
    long r;

    lock(&xincport_lock);
    r = --(*p);
    unlock(&xincport_lock);
    return r;
}
```

Uses `lock()` 245b and `unlock()` 245c.

23.4.2 Channel

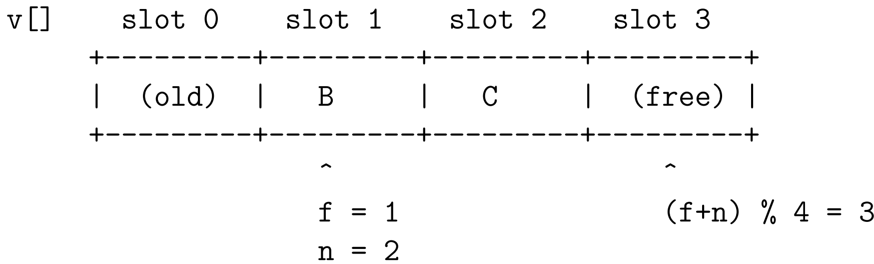
A `Channel` is a typed, optionally-buffered communication pipe between threads. When `s` (the buffer capacity) is zero, it is unbuffered: a send blocks until a receiver is ready, and vice versa—this is a rendezvous. When `s` is positive, up to `s` messages can be queued before the sender blocks. The buffer is a circular queue stored in `v` (which is allocated inline at the end of the struct), with `f` as the extraction index and `n` as the count. The `qentry` array holds pointers to `Alt` structures of threads blocked waiting on this channel. A single global `chanlock` protects all channel metadata—since threads within the same proc never run concurrently, this lock only contends between procs.

⟨*struct* Channel 288e⟩≡ (384b)

```
/*
 * Channel structure.  s is the size of the buffer.  For unbuffered channels
 * s is zero.  v is an array of s values.  If s is zero, v is unused.
 * f and n represent the state of the queue pointed to by v.
 */
struct Channel {
    int s;  /* Size of the channel (may be zero) */

    uint f;  /* Extraction point (insertion pt: (f+n) % s) */
```

```
	uint n;  /* Number of values in the channel */
	int e;  /* Element size */
	int freed;  /* Set when channel is being deleted */

	volatile Alt **qentry; /* Receivers/senders waiting (malloc) */
	volatile int nentry; /* # of entries malloc-ed */
	volatile int closed; /* channel is closed */

	// must be at the end of the struct!
	byte v[1];  /* Array of s values in the channel */
};
```

The buffer `v` is a classic circular queue: `f` is the head (next element to extract), the count `n` is how many slots are filled, and the tail (next insertion slot) is `(f+n)\%s`. Here is a buffered channel with `s=4` just after three sends followed by one receive, holding two live elements:




```
After the first recv, the ‘‘A’’ in slot 0 is stale but not
cleared; f moved from 0 to 1, and n dropped from 3 to 2.
```

Allocating `v` inline at the end of the struct (via the `byte v[1]` trailing-array trick) means `chancreate` does one `malloc` for the header and the payload together. This matters because channels are the fundamental coordination primitive—a program with many small channels should not pay a double-allocation overhead per channel.

⟨*global* `chanlock` 289a⟩≡ (563b)
```
static Lock chanlock;  /* central channel access lock */
```

⟨*function* `chancreate` 289b⟩≡ (563b)
```
Channel*
chancreate(int elemsize, int elemcnt)
{
	Channel *c;

	if(elemcnt < 0 || elemsize <= 0)
		return nil;
	c = _threadmalloc(sizeof(Channel) + elemsize*elemcnt, 1);
	c->e = elemsize;
	c->s = elemcnt;
	_threaddebug(DBGCHAN, "chancreate %p", c);
	return c;
}
```
Uses `DBGCHAN` 334c, `_threaddebug()` 334g, and `_threadmalloc()` 321a.

⟨*function* `chaninit` 289c⟩≡ (563b)
```
int
chaninit(Channel *c, int elemsize, int elemcnt)
{
	if(elemcnt < 0 || elemsize <= 0 || c == nil)
		return -1;
	c->f = 0;
	c->n = 0;
```

```
    c->closed = 0;
    c->freed = 0;
    c->e = elemsize;
    c->s = elemcnt;
    _threaddebug(DBGCHAN, "chaninit %p", c);
    return 1;
}
```
Uses DBGCHAN 334c and _threaddebug() 334g.

⟨*function* chanfree 290a⟩≡ (563b)
```
void
chanfree(Channel *c)
{
    lock(&chanlock);
    _chanfree(c);
    unlock(&chanlock);
}
```
Uses _chanfree() 290b, chanlock-7 289a, lock() 245b, and unlock() 245c.

⟨*function* _chanfree 290b⟩≡ (563b)
```
static void
_chanfree(Channel *c)
{
    int i, inuse;

    if(c->closed == 1)   /* chanclose is ongoing */
        inuse = 1;
    else{
        inuse = 0;
        for(i = 0; i < c->nentry; i++) /* alt ongoing */
            if(c->qentry[i])
                inuse = 1;
    }
    if(inuse)
        c->freed = 1;
    else{
        if(c->qentry)
            free(c->qentry);
        free(c);
    }
}
```
Uses free() 63i.

⟨*enum* _anon_ (libthread/channel.c) 290c⟩≡ (563b)
```
/* Value to indicate the channel is closed */
enum {
    CHANCLOSD = 0xc105ed,
};
```

⟨*constant* Closed 290d⟩≡ (563b)
```
#define Closed ((void*)CHANCLOSD)
```

chanclose marks a channel as closed, then wakes all threads that are blocked waiting to send (and receivers if the buffer is empty). A closed channel returns an error on send but still allows receiving buffered values—exactly like Go's closed channels. The two-phase close (state 1 = closing, state 2 = fully closed) prevents races between close and free.

⟨*function* chanclose 290e⟩≡ (563b)
```
int
chanclose(Channel *c)
```

```
{
    Alt *a;
    int i, s;

    s = _procsplhi(); /* note handlers; see :/^alt */
    lock(&chanlock);
    if(c->closed){
        /* Already close; we fail but it's ok. don't print */
        unlock(&chanlock);
        _procsplx(s);
        return -1;
    }
    c->closed = 1;  /* Being closed */
    /*
     * Locate entries that will fail due to close
     * (send, and receive if nothing buffered) and wake them up.
     * the situation cannot change because all queries
     * should be committed by now and new ones will find the channel
     * closed.  We still need to take the lock during the iteration
     * because we can wake threads on qentrys we have not seen yet
     * as in alt and there would be a race in the access to *a.
     */
    for(i = 0; i < c->nentry; i++){
        if((a = c->qentry[i]) == nil || *a->tag != nil)
            continue;

        if(a->op != CHANSND && (a->op != CHANRCV || c->n != 0))
            continue;
        *a->tag = c;
        unlock(&chanlock);
        _procsplx(s);
        while(_threadrendezvous(a->tag, Closed) == Intred)
            ;
        s = _procsplhi();
        lock(&chanlock);
    }

    c->closed = 2;  /* Fully closed */
    if(c->freed)
        _chanfree(c);
    unlock(&chanlock);
    _procsplx(s);
    return 0;
}
```

Uses `Closed-8` 290d, `Intred-9` 325b, `_chanfree()` 290b, `_procsplhi()` 328a, `_procsplx()` 328b, `_threadrendezvous()` 339e, `chanlock-7` 289a, `lock()` 245b, and `unlock()` 245c.

⟨*function* `chanclosing` 291⟩≡ (563b)

```
int
chanclosing(Channel *c)
{
    int n, s;

    s = _procsplhi(); /* note handlers; see :/^alt */
    lock(&chanlock);
    if(c->closed == 0)
        n = -1;
    else
        n = c->n;
    unlock(&chanlock);
```

```
    _procsplx(s);
    return n;
}
```
Uses _procsplhi() 328a, _procsplx() 328b, chanlock-7 289a, lock() 245b, and unlock() 245c.

23.4.3 Proc and Pqueue

A Proc represents an OS-level process containing one or more cooperative threads. Its sched jmp_buf is the scheduler's saved context—when a thread yields, it longjmps to p->sched, and the scheduler picks the next ready thread. Each proc maintains two thread queues: threads (all threads in this proc) and ready (runnable threads). The splhi/pending/nonotes fields handle note deferral: while holding the channel lock, notes are delayed to prevent deadlock.

⟨*struct* Proc 292⟩≡ (558c)
```
struct Proc
{
    int  pid;   /* process id */

    jmp_buf sched;   /* for context switches */

    Thread *thread;  /* running thread */




    bool  splhi;   /* delay notes */

    int  needexec;
    Execargs exec;   /* exec argument */
    Proc *newproc;  /* fork argument */
    char exitstr[ERRMAX]; /* exit status */

    int  rforkflag;

    Tqueue threads;  /* All threads of this proc */
    int  nthreads;

    Tqueue ready;   /* Runnable threads */
    Lock readylock;

    char printbuf[Printsize];
    int  blocked;  /* In a rendezvous */
    int  pending;  /* delayed note pending */
    int  nonotes;  /* delay notes */
    uint nextID;   /* ID of most recently created thread */


    void *arg;   /* passed between shared and unshared stk */
    char str[ERRMAX];  /* used by threadexits to avoid malloc */

    void* wdata;   /* Lib(worker) per-proc data pointer */
    void* udata;   /* User per-proc data pointer */
    char threadint;  /* tag for threadexitsall() */

    // Extra
    Proc *next;   /* linked list of Procs */

    Lock lock;
```

```
};
```
Uses Printsize 293a.

⟨*constant* Printsize 293a⟩≡ (558b)
```
Printsize = 2048,
```

⟨*global* procp 293b⟩≡ (560c)
```
// used to be in main.c
static Proc **procp;
```

⟨*function* _threadgetproc 293c⟩≡ (560c)
```
Proc*
_threadgetproc(void)
{
    return *procp;
}
```
Uses procp-27 293b.

⟨*function* _threadsetproc 293d⟩≡ (560c)
```
void
_threadsetproc(Proc *p)
{
    *procp = p;
}
```
Uses procp-27 293b.

⟨*global* _threadpq 293e⟩≡ (560c)
```
// used to be in create.c
Pqueue _threadpq;
```

⟨*struct* Pqueue 293f⟩≡ (558c)
```
struct Pqueue {  /* Proc queue */
    Proc  *head;
    Proc  **tail;

    // Extra
    Lock  lock;
};
```

⟨*function* proccreate 293g⟩≡ (563a)
```
int
proccreate(void (*f)(void*), void *arg, uint stacksize)
{
    return procrfork(f, arg, stacksize, 0);
}
```
Uses procrfork() 293h.

⟨*function* procrfork 293h⟩≡ (563a)
```
int
procrfork(void (*f)(void *), void *arg, uint stacksize, int rforkflag)
{
    Proc *p;
    int id;

    p = _threadgetproc();
    assert(p->newproc == nil);
    p->newproc = _newproc(f, arg, stacksize, nil, p->thread->grp, rforkflag);
    id = p->newproc->threads.head->id;
    _sched();
    return id;
}
```
Uses _newproc() 294a, _sched() 304a, and _threadgetproc() 293c.

⟨*function* _newproc 294a⟩≡ (563a)
```
/*
 * Create and initialize a new Proc structure with a single Thread
 * running inside it.  Add the Proc to the global process list.
 */
Proc*
_newproc(void (*f)(void *arg), void *arg, uint stacksize, char *name, int grp, int rforkflag)
{
    Proc *p;

    p = _threadmalloc(sizeof *p, 1);
    p->pid = -1;
    p->rforkflag = rforkflag;
    newthread(p, f, arg, stacksize, name, grp);

    lock(&_threadpq.lock);
    if(_threadpq.head == nil)
        _threadpq.head = p;
    else
        *_threadpq.tail = p;
    _threadpq.tail = &p->next;
    unlock(&_threadpq.lock);

    return p;
}
```
Uses _threadmalloc() 321a, _threadpq 293e, lock() 245b, newthread() 296c, and unlock() 245c.

⟨*function* _freeproc 294b⟩≡ (563a)
```
void
_freeproc(Proc *p)
{
    Thread *t, *nextt;

    for(t = p->threads.head; t; t = nextt){
        if(t->cmdname)
            free(t->cmdname);
        assert(t->stk != nil);
        free(t->stk);
        nextt = t->nextt;
        free(t);
    }
    free(p);
}
```
Uses free() 63i.

⟨*struct* Execargs 294c⟩≡ (558c)
```
struct Execargs
{
    char *prog;
    char **args;
    int  fd[2];
};
```

23.4.4 Thread and Tqueue

A Thread is a cooperatively-scheduled coroutine within a Proc. It has its own stack (allocated with malloc), a jmp_buf for context switching, and state tracking. The chan/alt fields record which channel operation the

thread is currently blocked on, used by the scheduler and debugger. The stack is filled with `0xFE` on creation so that stack usage can be measured by scanning for untouched bytes.

⟨*struct* `Thread` 295a⟩≡ (558c)
```
struct Thread
{
    int  id;  /* thread id */
    char *cmdname; /* ptr to name of thread */

    jmp_buf sched;  /* for context switches */

    Proc *proc;  /* proc of this thread */

    uint stksize; /* stack size */
    uchar *stk;  /* top of stack (lowest address of stack) */

    State state;  /* run state */
    State nextstate; /* next run state */

    Chanstate chan;  /* which channel operation is current */
    Alt  *alt;  /* pointer to current alt structure (debugging) */



    int  grp;  /* thread group */
    int  moribund; /* thread needs to die */

    int  ret;  /* return value for Exec, Fork */


    int  inrendez;
    Thread *rendhash; /* Trgrp linked list */
    void* rendtag; /* rendezvous tag */
    void* rendval; /* rendezvous value */
    int  rendbreak; /* rendezvous has been taken */


    void* udata[NPRIV]; /* User per-thread data pointer */

    // Extra
    Lock lock;  /* protects thread data structure */

    Thread *next;  /* next on ready queue */

    Thread *nextt;  /* next on list of threads in this proc*/


};
```
Uses `NPRIV` 295b.

⟨*constant* `NPRIV` 295b⟩≡ (558b)
```
NPRIV = 8,
```

⟨*function* `threadid` 295c⟩≡ (560e)
```
int
threadid(void)
{
    return _threadgetproc()->thread->id;
}
```
Uses `_threadgetproc()` 293c.

⟨*function* nextID 296a⟩≡ (563a)

```
static int
nextID(void)
{
    static Lock l;
    static int id;
    int i;

    lock(&l);
    i = ++id;
    unlock(&l);
    return i;
}
```

Uses lock() 245b and unlock() 245c.

⟨*function* threadcreate 296b⟩≡ (563a)

```
/*
 * Create a new thread and schedule it to run.
 * The thread grp is inherited from the currently running thread.
 */
int
threadcreate(void (*f)(void *arg), void *arg, uint stacksize)
{
    return newthread(_threadgetproc(), f, arg, stacksize, nil, threadgetgrp());
}
```

Uses _threadgetproc() 293c, newthread() 296c, and threadgetgrp() 337b.

⟨*function* newthread 296c⟩≡ (563a)

```
/*
 * Create and initialize a new Thread structure attached to a given proc.
 */
static int
newthread(Proc *p, void (*f)(void *arg), void *arg, uint stacksize, char *name, int grp)
{
    int id;
    Thread *t;

    if(stacksize < 32)
        sysfatal("bad stacksize %d", stacksize);
    t = _threadmalloc(sizeof(Thread), 1);

    t->stksize = stacksize;
    t->stk = _threadmalloc(stacksize, 0);
    memset(t->stk, 0xFE, stacksize);
    _threadinitstack(t, f, arg);

    t->grp = grp;
    if(name)
        t->cmdname = strdup(name);
    t->id = nextID();
    id = t->id;
    t->next = (Thread*)~0;
    t->proc = p;
    _threaddebug(DBGSCHED, "create thread %d.%d name %s", p->pid, t->id, name);

    lock(&p->lock);
    p->nthreads++;
    if(p->threads.head == nil)
        p->threads.head = t;
    else
```

```
        *p->threads.tail = t;
    p->threads.tail = &t->nextt;

    t->nextt = nil;
    t->state = Ready;
    _threadready(t);
    unlock(&p->lock);

    return id;
}
```

Uses DBGSCHED 334b, Ready 298c, _threaddebug() 334g, _threadmalloc() 321a, _threadready() 305a, lock() 245b, memset() 46b, nextID() 296a, strdup() 82a, sysfatal() 236a, and unlock() 245c.

Thread creation sets up a stack frame so that when the scheduler `longjmp`s to the new thread, it begins executing `launcher386`, which calls the user's function and then `threadexits`. The `_threadinitstack` function constructs this initial stack frame manually: it pushes the function pointer and argument onto the thread's stack, then sets `JMPBUFPC` to `launcher386` and `JMPBUFSP` to the top of the prepared stack.

⟨*function* `launcher386` 297a⟩≡ (557c)

```
static void
launcher386(void (*f)(void *arg), void *arg)
{
    (*f)(arg);
    threadexits(nil);
}
```

⟨*function* `_threadinitstack` 297b⟩≡ (557c)

```
void
_threadinitstack(Thread *t, void (*f)(void*), void *arg)
{
    ulong *tos;

    tos = (ulong*)&t->stk[t->stksize&~7];
    *--tos = (ulong)arg;
    *--tos = (ulong)f;
    t->sched[JMPBUFPC] = (ulong)launcher386+JMPBUFDPC;
    t->sched[JMPBUFSP] = (ulong)tos - 8;  /* old PC and new PC */
}
```

⟨*function* `_freethread` 297c⟩≡ (563a)

```
void
_freethread(Thread *t)
{
    Proc *p;
    Thread **l;

    p = t->proc;
    lock(&p->lock);
    for(l=&p->threads.head; *l; l=&(*l)->nextt){
        if(*l == t){
            *l = t->nextt;
            if(*l == nil)
                p->threads.tail = l;
            break;
        }
    }
    unlock(&p->lock);
    if (t->cmdname)
        free(t->cmdname);
    assert(t->stk != nil);
```

```
    free(t->stk);
    free(t);
}
```

Uses free() 63i, lock() 245b, and unlock() 245c.

⟨*function* threadpid 298a⟩≡ (560e)

```
int
threadpid(int id)
{
    int pid;
    Proc *p;
    Thread *t;

    if (id < 0)
        return -1;
    if (id == 0)
        return _threadgetproc()->pid;
    lock(&_threadpq.lock);
    for (p = _threadpq.head; p; p = p->next){
        lock(&p->lock);
        for (t = p->threads.head; t; t = t->nextt)
            if (t->id == id){
                pid = p->pid;
                unlock(&p->lock);
                unlock(&_threadpq.lock);
                return pid;
            }
        unlock(&p->lock);
    }
    unlock(&_threadpq.lock);
    return -1;
}
```

Uses _threadgetproc() 293c, _threadpq 293e, lock() 245b, and unlock() 245c.

⟨*struct* Tqueue 298b⟩≡ (558c)

```
struct Tqueue  /* Thread queue */
{
    Thread *head;
    Thread **tail;

    int  asleep;
};
```

⟨*enum* state 298c⟩≡ (558c)

```
enum state
{
    Dead,
    Running,
    Ready,
    Rendezvous,
};
```

⟨*enum* chanstate 298d⟩≡ (558c)

```
enum chanstate
{
    Channone,

    Chansend,
    Chanrecv,
```

```
    Chanalt,
};
```

⟨*function* threadsetname 299a⟩≡ (560e)

```
void
threadsetname(char *fmt, ...)
{
    int fd;
    char buf[128];
    va_list arg;
    Proc *p;
    Thread *t;

    p = _threadgetproc();
    t = p->thread;
    if (t->cmdname)
        free(t->cmdname);
    va_start(arg, fmt);
    t->cmdname = vsmprint(fmt, arg);
    va_end(arg);
    if(t->cmdname && p->nthreads == 1){
        snprint(buf, sizeof buf, "#p/%lud/args", _tos->pid); //getpid());
        if((fd = open(buf, OWRITE)) >= 0){
            write(fd, t->cmdname, strlen(t->cmdname)+1);
            close(fd);
        }
    }
}
```

Uses _threadgetproc() 293c, close(), free() 63i, open(), snprint() 535b, strlen() 80a, vsmprint() 537b, and write() 192b.

⟨*function* threadgetname 299b⟩≡ (560e)

```
char*
threadgetname(void)
{
    Proc *p;

    if((p = _threadgetproc()) && p->thread)
        return p->thread->cmdname;
    return nil;
}
```

Uses _threadgetproc() 293c.

23.4.5 `Alt`

`Alt` describes one arm of an `alt` call—the equivalent of one `case` in Go's `select`. It specifies a channel, a pointer to the value buffer, and an operation (send, receive, no-op, or end sentinel). The `CHANNOBLK` terminator makes the `alt` non-blocking: if no channel is ready, it returns immediately instead of waiting. The `tag` and `entryno` fields are used internally by the `alt` implementation to coordinate rendezvous between threads.

⟨*enum* chanop 299c⟩≡ (384b)

```
/* Channel operations for alt: */
enum chanop {
    CHANEND,

    CHANSND,
    CHANRCV,
    CHANNOP,

    CHANNOBLK,
};
```

⟨*struct* Alt 300a⟩≡ (384b)
```
struct Alt {
    Channel *c;  /* channel */
    void *v;  /* pointer to value */
    ChanOp op;  /* operation */

    char *err;  /* did the op fail? */
    /*
     * the next variables are used internally to alt
     * they need not be initialized
     */
    Channel **tag;  /* pointer to rendez-vous tag */
    int entryno; /* entry number */
};
```

23.5 main() and threadmain()

When a program uses `libthread`, the library provides its own `main` which takes over before the user's code runs. It initializes the scheduler, installs thread-aware versions of `_sysfatal`, `_dial`, and `_assert`, sets up note handling, and then creates a new proc whose first thread runs `threadmain`—the user's entry point. The `setjmp` at the top of `main` is the return point for the scheduler loop: after initialization, `_schedinit` runs and never returns to `main` directly.

⟨*global* mainp 300b⟩≡ (565a)
```
// ref<ref<Proc>
static Proc **mainp;
```

⟨*global* _mainjmp 300c⟩≡ (565a)
```
static jmp_buf _mainjmp;
```

⟨*global* mainstacksize 300d⟩≡ (565a)
```
int mainstacksize;
```

⟨*function* main 300e⟩≡ (565a)
```
void
main(int argc, char **argv)
{
    Mainarg *a;
    Proc *p;

    rfork(RFREND);
    mainp = &p;

    if(setjmp(_mainjmp))
        _schedinit(p);

    //_threaddebuglevel = (DBGSCHED|DBGCHAN|DBGREND)^~0;
    _systhreadinit();
    _qlockinit(_threadrendezvous);

    _sysfatal = _threadsysfatal;
    _dial     = _threaddial;
    __assert  = _threadassert;

    notify(_threadnote);

    if(mainstacksize == 0)
        mainstacksize = 8*1024;
```

```
    a = _threadmalloc(sizeof *a, 1);
    a->argc = argc;
    a->argv = argv;

    p = _newproc(mainlauncher, a, mainstacksize, "threadmain", 0, 0);
    _schedinit(p);

    abort(); /* not reached */
}
```

Uses __assert 235g, _dial 348a, _mainjmp-19 300c, _newproc() 294a, _qlockinit() 490e, _schedinit() 302c, _sysfatal 236b, _systhreadinit() 301a, _threadassert() 333b, _threaddial() 329d, _threadmalloc() 321a, _threadnote() 327b, _threadrendezvous() 339e, _threadsysfatal() 333a, abort() 356d, mainlauncher() 301c, mainp-20 300b, mainstacksize 300d, notify(), and rfork().

⟨*function* _systhreadinit 301a⟩≡ (560c)

```
void
_systhreadinit(void)
{
    procp = privalloc();
}
```

Uses privalloc() 488d and procp-27 293b.

⟨*struct* Mainarg 301b⟩≡ (565a)

```
struct Mainarg
{
    int  argc;
    char **argv;
};
```

⟨*function* mainlauncher 301c⟩≡ (565a)

```
static void
mainlauncher(void *arg)
{
    Mainarg *a;

    a = arg;

    // user defined threadmain()!!
    threadmain(a->argc, a->argv);

    threadexits("threadmain");
}
```

Uses threadexits() 321c and threadmain().

23.6 Threads scheduler

The scheduler is a simple loop: `_schedinit` runs in each proc, alternating between cleaning up finished threads and calling `_sched` to pick the next one. `_sched` saves the current thread's context with `setjmp`, then `longjmps` to the proc's scheduler context. The scheduler calls `runthread` to dequeue the next ready thread and `longjmps` to it. If no threads are ready, `runthread` blocks the entire proc via `rendezvous` until `_threadready` wakes it. The `yield` function simply calls `_sched`, which saves the current thread (marking it Ready) and switches to the next one. This is the core of cooperative scheduling: a thread runs until it explicitly yields, sends/receives on a channel, or exits.

The context-switch dance uses *two* `jmp_buf`s per proc, one in `Proc.sched` (the scheduler's saved context) and one per thread in `Thread.sched`. A yield from thread T_1 to thread T_2 proceeds as follows:

```
T1 running
    |
    | _sched():
    |   setjmp(T1->sched)   <-- remember where T1 paused
    |   longjmp(p->sched)   <-- jump to scheduler
    v
scheduler loop (longjmp target)
    |
    | runthread() picks T2 from p->ready
    | p->thread = T2
    | longjmp(T2->sched)    <-- resume T2 where it paused
    v
T2 running
```

The `setjmp` at the top of `_schedinit` is the landing pad for *every* `longjmp(p->sched)`: each yielding thread returns to exactly the same point in the scheduler loop, which then picks the next ready thread. The elegant part is that a brand-new thread looks identical to a paused one: `_threadinitstack` forges a `jmp_buf` whose PC is `launcher386` and whose SP points into the new stack, so the scheduler can `longjmp` into it with no special case.

⟨*global* `_psstate` 302a⟩≡ (562c)

```
static char *_psstate[] = {
    "Moribund",
    "Dead",
    "Exec",
    "Fork",
    "Running",
    "Ready",
    "Rendezvous",
};
```

⟨*function* `psstate` 302b⟩≡ (562c)

```
static char*
psstate(int s)
{
    if(s < 0 || s >= nelem(_psstate))
        return "unknown";
    return _psstate[s];
}
```

Uses _psstate-1 302a.

⟨*function* `_schedinit` 302c⟩≡ (562c)

```
void
_schedinit(void *arg)
{
    Proc *p;
    Thread *t, **l;

    p = arg;
    _threadsetproc(p);
    p->pid = _tos->pid; //getpid();

    while(setjmp(p->sched))
        ;

    _threaddebug(DBGSCHED, "top of schedinit, _threadexitsallstatus=%p",
             _threadexitsallstatus);
```

```
	if(_threadexitsallstatus)
		exits(_threadexitsallstatus);

	lock(&p->lock);
	t = p->thread;
	if(t != nil){
		p->thread = nil;

		if(t->moribund){
			t->state = Dead;
			for(l=&p->threads.head; *l; l=&(*l)->nextt)
				if(*l == t){
					*l = t->nextt;
					if(*l==nil)
						p->threads.tail = l;
					p->nthreads--;
					break;
				}
			unlock(&p->lock);
			if(t->inrendez){
				_threadflagrendez(t);
				_threadbreakrendez();
			}
			free(t->stk);
			free(t->cmdname);
			free(t); /* XXX how do we know there are no references? */
			t = nil;
			_sched();
		}
		if(p->needexec){
			t->ret = _schedexec(&p->exec);
			p->needexec = 0;
		}
		if(p->newproc){
			t->ret = _schedfork(p->newproc);
			p->newproc = nil;
		}
		t->state = t->nextstate;
		if(t->state == Ready)
			_threadready(t);
	}
	unlock(&p->lock);
	_sched();
}
```

Uses `DBGSCHED` 334b, `Dead` 298c, `Ready` 298c, `_sched()` 304a, `_schedexec()` 323a, `_schedfork()` 322c, `_threadbreakrendez()` 340d, `_threaddebug()` 334g, `_threadexitsallstatus` 321b, `_threadflagrendez()` 340c, `_threadready()` 305a, `_threadsetproc()` 293d, `exits()` 45b, `free()` 63i, `lock()` 245b, and `unlock()` 245c.

⟨*function* `needstack` 303⟩≡ (562c)

```
void
needstack(int n)
{
	int x;
	Proc *p;
	Thread *t;

	p = _threadgetproc();
	t = p->thread;

	if((uchar*)&x - n < (uchar*)t->stk){
```

```
        fprint(2, "%s %lud: &x=%p n=%d t->stk=%p\n",
            argv0, _tos->pid, &x, n, t->stk);
        fprint(2, "%s %lud: stack overflow\n", argv0, _tos->pid);
        abort();
    }
}
```

Uses _threadgetproc() 293c, abort() 356d, argv0, and fprint() 75a.

⟨*function* _sched 304a⟩≡ (562c)

```
void
_sched(void)
{
    Proc *p;
    Thread *t;

Resched:
    p = _threadgetproc();
    t = p->thread;
    if(t != nil){
        needstack(128);
        _threaddebug(DBGSCHED, "pausing, state=%s", psstate(t->state));
        if(setjmp(t->sched)==0)
            longjmp(p->sched, 1);
        return;
    }else{
        t = runthread(p);
        if(t == nil){
            _threaddebug(DBGSCHED, "all threads gone; exiting");
            _schedexit(p);
        }
        _threaddebug(DBGSCHED, "running %d.%d", t->proc->pid, t->id);
        p->thread = t;
        if(t->moribund){
            _threaddebug(DBGSCHED, "%d.%d marked to die");
            goto Resched;
        }
        t->state = Running;
        t->nextstate = Ready;
        longjmp(t->sched, 1);
    }
}
```

Uses DBGSCHED 334b, Ready 298c, Running 298c, _schedexit() 305c, _threaddebug() 334g, _threadgetproc() 293c, needstack() 303, psstate() 302b, and runthread() 304b.

⟨*function* runthread 304b⟩≡ (562c)

```
static Thread*
runthread(Proc *p)
{
    Thread *t;
    Tqueue *q;

    if(p->nthreads==0)
        return nil;
    q = &p->ready;
    lock(&p->readylock);
    if(q->head == nil){
        q->asleep = 1;
        _threaddebug(DBGSCHED, "sleeping for more work");
        unlock(&p->readylock);
        while(rendezvous(q, 0) == (void*)~0){
```

```
            if(_threadexitsallstatus)
                exits(_threadexitsallstatus);
        }
        /* lock picked up from _threadready */
    }
    t = q->head;
    q->head = t->next;
    unlock(&p->readylock);
    return t;
}
```

Uses DBGSCHED 334b, _threaddebug() 334g, _threadexitsallstatus 321b, exits() 45b, lock() 245b, rendezvous(), and unlock() 245c.

⟨*function* _threadready 305a⟩≡ (562c)

```
void
_threadready(Thread *t)
{
    Tqueue *q;

    assert(t->state == Ready);
    _threaddebug(DBGSCHED, "readying %d.%d", t->proc->pid, t->id);

    q = &t->proc->ready;
    lock(&t->proc->readylock);
    t->next = nil;
    if(q->head==nil)
        q->head = t;
    else
        *q->tail = t;
    q->tail = &t->next;
    if(q->asleep){
        q->asleep = 0;
        /* lock passes to runthread */
        _threaddebug(DBGSCHED, "waking process %d", t->proc->pid);
        while(rendezvous(q, 0) == (void*)~0){
            if(_threadexitsallstatus)
                exits(_threadexitsallstatus);
        }
    }else
        unlock(&t->proc->readylock);
}
```

Uses DBGSCHED 334b, Ready 298c, _threaddebug() 334g, _threadexitsallstatus 321b, exits() 45b, lock() 245b, rendezvous(), and unlock() 245c.

⟨*function* yield 305b⟩≡ (562c)

```
void
yield(void)
{
    _sched();
}
```

Uses _sched() 304a.

⟨*function* _schedexit 305c⟩≡ (565a)

```
void
_schedexit(Proc *p)
{
    char ex[ERRMAX];
    Proc **l;
```

```
    lock(&_threadpq.lock);
    for(l=&_threadpq.head; *l; l=&(*l)->next){
        if(*l == p){
            *l = p->next;
            if(*l == nil)
                _threadpq.tail = l;
            break;
        }
    }
    unlock(&_threadpq.lock);

    utfecpy(ex, ex+sizeof ex, p->exitstr);
    free(p);
    _exits(ex);
}
```

Uses _exits(), _threadpq 293e, free() 63i, lock() 245b, unlock() 245c, and utfecpy() 460b.

23.7 alt()

alt is the heart of libthread's channel system—even send and recv are implemented as single-entry alt calls. The algorithm has three phases: (1) scan all channels to see if any operation can proceed immediately; if multiple can, select one at random (nrand(++n)) for fairness; (2) if nothing is ready and CHANNOBLK is set, return immediately; otherwise enqueue on all channels and block; (3) when woken, dequeue from all channels and return the selected entry. The _procsplhi/_procsplx calls defer note delivery while the channel lock is held—if a note handler tried to use a channel while we hold chanlock, it would deadlock. The Intred (interrupted) return from rendezvous causes a retry rather than an error, because another thread may be about to complete the rendezvous.

⟨*function* alt 306⟩≡ (563b)

```
int
alt(Alt *alts)
{
    Alt *a, *xa, *ca;
    Channel volatile *c;
    int n, s, waiting, allreadycl;
    void* r;
    Thread *t;

    /*
     * The point of going splhi here is that note handlers
     * might reasonably want to use channel operations,
     * but that will hang if the note comes while we hold the
     * chanlock.  Instead, we delay the note until we've dropped
     * the lock.
     */
    t = _threadgetproc()->thread;
    if(t->moribund || _threadexitsallstatus)
        yield(); /* won't return */
    s = _procsplhi();

    lock(&chanlock);
    t->alt = alts;
    t->chan = Chanalt;

    /* test whether any channels can proceed */
    n = 0;
    a = nil;
```

```
for(xa=alts; xa->op!=CHANEND && xa->op!=CHANNOBLK; xa++){
    xa->entryno = -1;
    if(xa->op == CHANNOP)
        continue;

    c = xa->c;
    if(c==nil){
        unlock(&chanlock);
        _procsplx(s);
        t->chan = Channone;
        return -1;
    }

    if(isopenfor(c, xa->op) && canexec(xa))
        if(nrand(++n) == 0)
            a = xa;
}


if(a==nil){
    /* nothing can proceed */
    if(xa->op == CHANNOBLK){
        unlock(&chanlock);
        _procsplx(s);
        t->chan = Channone;
        if(xa->op == CHANNOBLK)
            return xa - alts;
    }

    /* enqueue on all channels open for us. */
    c = nil;
    ca = nil;
    waiting = 0;
    allreadycl = 0;
    for(xa=alts; xa->op!=CHANEND; xa++)
        if(xa->op==CHANNOP)
            continue;
        else if(isopenfor(xa->c, xa->op)){
            waiting = 1;
            enqueue(xa, &c);
        } else if(xa->err != errcl)
            ca = xa;
        else
            allreadycl = 1;

    if(waiting == 0)
        if(ca != nil){
            /* everything was closed, select last channel */
            ca->err = errcl;
            unlock(&chanlock);
            _procsplx(s);
            t->chan = Channone;
            return ca - alts;
        } else if(allreadycl){
            /* everything was already closed */
            unlock(&chanlock);
            _procsplx(s);
            t->chan = Channone;
            return -1;
```

```
            }
        /*
         * wait for successful rendezvous.
         * we can't just give up if the rendezvous
         * is interrupted -- someone else might come
         * along and try to rendezvous with us, so
         * we need to be here.
         * if the channel was closed, the op is done
         * and we flag an error for the entry.
         */
        Again:
        unlock(&chanlock);
        _procsplx(s);
        r = _threadrendezvous(&c, 0);
        s = _procsplhi();
        lock(&chanlock);

        if(r==Intred){  /* interrupted */
            if(c!=nil) /* someone will meet us; go back */
                goto Again;
            c = (Channel*)~0; /* so no one tries to meet us */
        }

        /* dequeue from channels, find selected one */
        a = nil;
        for(xa=alts; xa->op!=CHANEND; xa++){
            if(xa->op==CHANNOP)
                continue;
            if(xa->c == c){
                a = xa;
                a->err = nil;
                if(r == Closed)
                    a->err = errcl;
            }
            dequeue(xa);
        }
        unlock(&chanlock);
        _procsplx(s);
        if(a == nil){ /* we were interrupted */
            assert(c==(Channel*)~0);
            return -1;
        }
    }else
        altexec(a, s); /* unlocks chanlock, does splx */
    _sched();
    t->chan = Channone;
    return a - alts;
}
```

Uses Chanalt 298d, Channone 298d, Closed-8 290d, Intred-9 325b, _procsplhi() 328a, _procsplx() 328b, _sched() 304a, _threadexitsallstatus 321b, _threadgetproc() 293c, _threadrendezvous() 339e, chanlock-7 289a, dequeue() 309c, enqueue() 309b, errcl-6 309a, isopenfor() 308a, lock() 245b, nrand() 407b, unlock() 245c, and yield() 305b.

⟨*function* isopenfor 308a⟩≡ (563b)

```
static bool
isopenfor(Channel *c, int op)
{
    return c->closed == 0 || (op == CHANRCV && c->n > 0);
}
```

⟨*function* canexec 308b⟩≡ (563b)

```
static int
canexec(Alt *a)
{
    int i, otherop;
    Channel *c;

    c = a->c;
    /* are there senders or receivers blocked? */
    otherop = (CHANSND+CHANRCV) - a->op;
    for(i=0; i<c->nentry; i++)
        if(c->qentry[i] && c->qentry[i]->op==otherop && *c->qentry[i]->tag==nil){
            _threaddebug(DBGCHAN, "can rendez alt %p chan %p", a, c);
            return 1;
        }

    /* is there room in the channel? */
    if((a->op==CHANSND && c->n < c->s)
    || (a->op==CHANRCV && c->n > 0)){
        _threaddebug(DBGCHAN, "can buffer alt %p chan %p", a, c);
        return 1;
    }

    return 0;
}
```

⟨*global* `errcl` 309a⟩≡ (563b)

```
static char errcl[] = "channel was closed";
```

Uses `errcl-6` 309a.

⟨*function* `enqueue` 309b⟩≡ (563b)

```
static void
enqueue(Alt *a, Channel **c)
{
    int i;

    _threaddebug(DBGCHAN, "Queuing alt %p on channel %p", a, a->c);
    a->tag = c;
    i = emptyentry(a->c);
    a->c->qentry[i] = a;
}
```

Uses `DBGCHAN` 334c, `_threaddebug()` 334g, and `emptyentry()` 310a.

⟨*function* `dequeue` 309c⟩≡ (563b)

```
static void
dequeue(Alt *a)
{
    int i;
    Channel *c;

    c = a->c;
    for(i=0; i<c->nentry; i++)
        if(c->qentry[i]==a){
            _threaddebug(DBGCHAN, "Dequeuing alt %p from channel %p", a, a->c);
            c->qentry[i] = nil;
            /* release if freed and not closing */
            if(c->freed && c->closed != 1)
                _chanfree(c);
            return;
        }
}
```

Uses `DBGCHAN` 334c, `_chanfree()` 290b, and `_threaddebug()` 334g.

⟨*function* emptyentry 310a⟩≡ (563b)
```
static int
emptyentry(Channel *c)
{
    int i, extra;

    assert((c->nentry==0 && c->qentry==nil) || (c->nentry && c->qentry));

    for(i=0; i<c->nentry; i++)
        if(c->qentry[i]==nil)
            return i;

    extra = 16;
    c->nentry += extra;
    c->qentry = realloc((void*)c->qentry, c->nentry*sizeof(c->qentry[0]));
    if(c->qentry == nil)
        sysfatal("realloc channel entries: %r");
    memset(&c->qentry[i], 0, extra*sizeof(c->qentry[0]));
    return i;
}
```
Uses `memset()` 46b, `realloc()` 67b, and `sysfatal()` 236a.

⟨*function* altexec 310b⟩≡ (563b)
```
static int
altexec(Alt *a, int spl)
{
    volatile Alt *b;
    int i, n, otherop;
    Channel *c;
    void *me, *waiter, *buf;

    c = a->c;

    /* rendezvous with others */
    otherop = (CHANSND+CHANRCV) - a->op;
    n = 0;
    b = nil;
    me = a->v;
    for(i=0; i<c->nentry; i++)
        if(c->qentry[i] && c->qentry[i]->op==otherop && *c->qentry[i]->tag==nil)
            if(nrand(++n) == 0)
                b = c->qentry[i];
    if(b != nil){
        _threaddebug(DBGCHAN, "rendez %s alt %p chan %p alt %p", a->op==CHANRCV?"recv":"send", a, c, b);
        waiter = b->v;
        if(c->s && c->n){
            /*
             * if buffer is full and there are waiters
             * and we're meeting a waiter,
             * we must be receiving.
             *
             * we use the value in the channel buffer,
             * copy the waiter's value into the channel buffer
             * on behalf of the waiter, and then wake the waiter.
             */
            if(a->op!=CHANRCV)
                abort();
            buf = altexecbuffered(a, 1);
            altcopy(me, buf, c->e);
            altcopy(buf, waiter, c->e);
```

```
        }else{
            if(a->op==CHANRCV)
                altcopy(me, waiter, c->e);
            else
                altcopy(waiter, me, c->e);
        }
        *b->tag = c; /* commits us to rendezvous */
        _threaddebug(DBGCHAN, "unlocking the chanlock");
        unlock(&chanlock);
        _procsplx(spl);
        _threaddebug(DBGCHAN, "chanlock is %lud", *(ulong*)&chanlock);
        while(_threadrendezvous(b->tag, 0) == Intred)
            ;
        return 1;
    }

    buf = altexecbuffered(a, 0);
    if(a->op==CHANRCV)
        altcopy(me, buf, c->e);
    else
        altcopy(buf, me, c->e);

    unlock(&chanlock);
    _procsplx(spl);
    return 1;
}
```

⟨*function* altexecbuffered 311a⟩≡ (563b)

```
static void*
altexecbuffered(Alt *a, int willreplace)
{
    uchar *v;
    Channel *c;

    c = a->c;
    /* use buffered channel queue */
    if(a->op==CHANRCV && c->n > 0){
        _threaddebug(DBGCHAN, "buffer recv alt %p chan %p", a, c);
        v = c->v + c->e*(c->f%c->s);
        if(!willreplace)
            c->n--;
        c->f++;
        return v;
    }
    if(a->op==CHANSND && c->n < c->s){
        _threaddebug(DBGCHAN, "buffer send alt %p chan %p", a, c);
        v = c->v + c->e*((c->f+c->n)%c->s);
        if(!willreplace)
            c->n++;
        return v;
    }
    abort();
    return nil;
}
```

Uses DBGCHAN 334c, _threaddebug() 334g, and abort() 356d.

⟨*function* altcopy 311b⟩≡ (563b)

```
static void
altcopy(void *dst, void *src, int sz)
{
```

```
    if(dst){
        if(src)
            memmove(dst, src, sz);
        else
            memset(dst, 0, sz);
    }
}
```

Uses memmove() 48 and memset() 46b.

23.8 Send/Receive

All channel operations—**send**, **recv**, **nbsend**, **nbrecv**—are thin wrappers around **runop**, which builds a two-entry **Alt** array and calls **alt**. The comment in the code acknowledges that a direct implementation would be faster, but the simplicity of reusing **alt** is preferred. For non-blocking variants, the sentinel is CHANNOBLK instead of CHANEND.

23.8.1 runop

⟨*function* runop 312⟩≡ (563b)

```
static int
runop(int op, Channel *c, void *v, bool nb)
{
    int r;
    Alt a[2];

    /*
     * we could do this without calling alt,
     * but the only reason would be performance,
     * and i'm not convinced it matters.
     */
    a[0].op = op;
    a[0].c = c;
    a[0].v = v;
    a[0].err = nil;
    a[1].op = CHANEND;
    if(nb)
        a[1].op = CHANNOBLK;

    switch(r=alt(a)){
    case -1: /* interrupted */
        return -1;
    case 1: /* nonblocking, didn't accomplish anything */
        assert(nb);
        return 0;
    case 0:
        /*
         * Okay, but return -1 if the op is done because of a close.
         */
        if(a[0].err != nil)
            return -1;
        return 1;
    default:
        fprint(2, "ERROR: channel alt returned %d\n", r);
        abort();
        return -1;
    }
```

```
}
```
Uses `abort()` 356d, `alt()` 306, and `fprint()` 75a.

23.8.2 send()/recv()

⟨*function* `send` 313a⟩≡ (563b)
```
int
send(Channel *c, void *v)
{
    return runop(CHANSND, c, v, 0);
}
```
Uses `runop()` 312.

⟨*function* `recv` 313b⟩≡ (563b)
```
int
recv(Channel *c, void *v)
{
    return runop(CHANRCV, c, v, 0);
}
```
Uses `runop()` 312.

23.8.3 Non blocking operations: nbxxx()

⟨*function* `nbrecv` 313c⟩≡ (563b)
```
int
nbrecv(Channel *c, void *v)
{
    return runop(CHANRCV, c, v, 1);
}
```
Uses `runop()` 312.

⟨*function* `nbsend` 313d⟩≡ (563b)
```
int
nbsend(Channel *c, void *v)
{
    return runop(CHANSND, c, v, 1);
}
```
Uses `runop()` 312.

23.8.4 Typed send and receive

The typed wrappers (`sendul`/`recvul`, `sendp`/`recvp`) add a size check to catch mismatched channel/value types at runtime. Without these, sending a `ulong` on a channel created for pointers would silently corrupt data. The `channelsize` function aborts if the element size doesn't match—a runtime approximation of Go's compile-time type checking on channels.

⟨*function* `channelsize` 313e⟩≡ (563b)
```
static void
channelsize(Channel *c, int sz)
{
    if(c->e != sz){
        fprint(2, "expected channel with elements of size %d, got size %d\n",
            sz, c->e);
        abort();
    }
}
```
Uses `abort()` 356d and `fprint()` 75a.

⟨*function* `sendul` 314a⟩≡ (563b)

```
int
sendul(Channel *c, ulong v)
{
    channelsize(c, sizeof(ulong));
    return send(c, &v);
}
```

Uses `channelsize()` 313e and `send()` 313a.

⟨*function* `recvul` 314b⟩≡ (563b)

```
ulong
recvul(Channel *c)
{
    ulong v;

    channelsize(c, sizeof(ulong));
    if(recv(c, &v) < 0)
        return ~0;
    return v;
}
```

Uses `channelsize()` 313e and `recv()` 313b.

⟨*function* `sendp` 314c⟩≡ (563b)

```
int
sendp(Channel *c, void *v)
{
    channelsize(c, sizeof(void*));
    return send(c, &v);
}
```

Uses `channelsize()` 313e and `send()` 313a.

⟨*function* `recvp` 314d⟩≡ (563b)

```
void*
recvp(Channel *c)
{
    void *v;

    channelsize(c, sizeof(void*));
    if(recv(c, &v) < 0)
        return nil;
    return v;
}
```

Uses `channelsize()` 313e and `recv()` 313b.

⟨*function* `nbsendul` 314e⟩≡ (563b)

```
int
nbsendul(Channel *c, ulong v)
{
    channelsize(c, sizeof(ulong));
    return nbsend(c, &v);
}
```

Uses `channelsize()` 313e and `nbsend()` 313d.

⟨*function* `nbrecvul` 314f⟩≡ (563b)

```
ulong
nbrecvul(Channel *c)
{
    ulong v;
```

```
    channelsize(c, sizeof(ulong));
    if(nbrecv(c, &v) == 0)
        return 0;
    return v;
}
```

Uses channelsize() 313e and nbrecv() 313c.

⟨*function* nbsendp 315a⟩≡ (563b)

```
int
nbsendp(Channel *c, void *v)
{
    channelsize(c, sizeof(void*));
    return nbsend(c, &v);
}
```

Uses channelsize() 313e and nbsend() 313d.

⟨*function* nbrecvp 315b⟩≡ (563b)

```
void*
nbrecvp(Channel *c)
{
    void *v;

    channelsize(c, sizeof(void*));
    if(nbrecv(c, &v) == 0)
        return nil;
    return v;
}
```

Uses channelsize() 313e and nbrecv() 313c.

23.9 IO processus

Cooperative threads have a fundamental problem: a blocking system call (like `read` on a network connection) blocks the entire proc, freezing all threads within it. The solution is "IO procs"—dedicated processes that perform blocking I/O on behalf of threads. An `Ioproc` runs in its own process, receives I/O requests via channels, executes the blocking call, and sends the result back. The calling thread is free to yield while the IO proc is blocked. The protocol uses two channels: `c` acquires the IO proc, and `creply` exchanges the data and result. The `xioproc` function is the IO proc's event loop, and `iocall` is the client-side helper. The `io*` wrappers (`ioopen`, `ioread`, `iowrite`, etc.) provide a familiar API identical to the blocking system calls.

⟨*function* ioproc_arg 315c⟩≡ (558c)

```
#define ioproc_arg(io, type) (va_arg((io)->arg, type))
```

23.9.1 Ioproc

⟨*struct* Ioproc 315d⟩≡ (558c)

```
struct Ioproc
{
    int tid;

    Channel *c;
    Channel *creply;

    int inuse;

    long (*op)(va_list*);
```

```
    va_list arg;
    long ret;
    char err[ERRMAX];

    Ioproc *next;
};
```

⟨*constant* STACK 316a⟩≡ (566a)

```
STACK = 8192,
```

⟨*function* ioproc 316b⟩≡ (566b)

```
Ioproc*
ioproc(void)
{
    Ioproc *io;

    io = mallocz(sizeof(Ioproc), 1);
    if(io == nil)
        sysfatal("ioproc malloc: %r");
    io->c = chancreate(sizeof(void*), 0);
    io->creply = chancreate(sizeof(void*), 0);
    io->tid = proccreate(xioproc, io, STACK);
    return io;
}
```

Uses `STACK-18` 316a, `chancreate()` 289b, `mallocz()` 67a, `proccreate()` 293g, `sysfatal()` 236a, and `xioproc()` 316e.

⟨*function* closeioproc 316c⟩≡ (566b)

```
void
closeioproc(Ioproc *io)
{
    if(io == nil)
        return;
    iointerrupt(io);
    while(send(io->c, 0) == -1)
        ;
    chanfree(io->c);
    chanfree(io->creply);
    free(io);
}
```

Uses `chanfree()` 290a, `free()` 63i, `iointerrupt()` 316d, and `send()` 313a.

⟨*function* iointerrupt 316d⟩≡ (566b)

```
void
iointerrupt(Ioproc *io)
{
    if(!io->inuse)
        return;
    threadint(io->tid);
}
```

Uses `threadint()` 339a.

23.9.2 xiproc()

⟨*function* xioproc 316e⟩≡ (566b)

```
static void
xioproc(void *a)
{
```

```
    Ioproc *io, *x;
    io = a;
    /*
     * first recvp acquires the ioproc.
     * second tells us that the data is ready.
     */
    for(;;){
        while(recv(io->c, &x) == -1)
            ;
        if(x == 0) /* our cue to leave */
            break;
        assert(x == io);

        /* caller is now committed -- even if interrupted he'll return */
        while(recv(io->creply, &x) == -1)
            ;
        if(x == 0) /* caller backed out */
            continue;
        assert(x == io);

        io->ret = io->op(&io->arg);
        if(io->ret < 0)
            rerrstr(io->err, sizeof io->err);
        while(send(io->creply, &io) == -1)
            ;
        while(recv(io->creply, &x) == -1)
            ;
    }
}
```

Uses `recv()` 313b, `rerrstr()` 234a, and `send()` 313a.

23.9.3 iocall()

⟨*function* iocall 317⟩≡ (566c)

```
long
iocall(Ioproc *io, long (*op)(va_list*), ...)
{
    int ret, inted;
    Ioproc *msg;

    if(send(io->c, &io) == -1){
        werrstr("interrupted");
        return -1;
    }
    assert(!io->inuse);
    io->inuse = 1;
    io->op = op;
    va_start(io->arg, op);
    msg = io;
    inted = 0;
    while(send(io->creply, &msg) == -1){
        msg = nil;
        inted = 1;
    }
    if(inted){
        werrstr("interrupted");
        return -1;
    }
```

```
    /*
     * If we get interrupted, we have to stick around so that
     * the IO proc has someone to talk to.  Send it an interrupt
     * and try again.
     */
    inted = 0;
    while(recv(io->creply, nil) == -1){
        inted = 1;
        iointerrupt(io);
    }
    USED(inted);
    va_end(io->arg);
    ret = io->ret;
    if(ret < 0)
        errstr(io->err, sizeof io->err);
    io->inuse = 0;

    /* release resources */
    while(send(io->creply, &io) == -1)
        ;
    return ret;
}
```

Uses `errstr()`, `iointerrupt()` 316d, `recv()` 313b, `send()` 313a, and `werrstr()` 234b.

23.9.4 IO wrappers

⟨*function* `ioopen` 318a⟩≡ (567b)

```
int
ioopen(Ioproc *io, char *path, int mode)
{
    return iocall(io, _ioopen, path, mode);
}
```

Uses `_ioopen()` 318b and `iocall()` 317.

⟨*function* `_ioopen` 318b⟩≡ (567b)

```
static long
_ioopen(va_list *arg)
{
    char *path;
    int mode;

    path = va_arg(*arg, char*);
    mode = va_arg(*arg, int);
    return open(path, mode);
}
```

Uses `open()`.

⟨*function* `ioclose` 318c⟩≡ (566e)

```
int
ioclose(Ioproc *io, int fd)
{
    return iocall(io, _ioclose, fd);
}
```

Uses `_ioclose()` 319a and `iocall()` 317.

⟨*function* _ioclose 319a⟩≡ (566e)

```
static long
_ioclose(va_list *arg)
{
    int fd;

    fd = va_arg(*arg, int);
    return close(fd);
}
```

Uses `close()`.

⟨*function* ioread 319b⟩≡ (567c)

```
long
ioread(Ioproc *io, int fd, void *a, long n)
{
    return iocall(io, _ioread, fd, a, n);
}
```

Uses `_ioread()` 319c and `iocall()` 317.

⟨*function* _ioread 319c⟩≡ (567c)

```
static long
_ioread(va_list *arg)
{
    int fd;
    void *a;
    long n;

    fd = va_arg(*arg, int);
    a = va_arg(*arg, void*);
    n = va_arg(*arg, long);
    return read(fd, a, n);
}
```

Uses `read()` 192a.

⟨*function* ioreadn 319d⟩≡ (567d)

```
long
ioreadn(Ioproc *io, int fd, void *a, long n)
{
    return iocall(io, _ioreadn, fd, a, n);
}
```

Uses `_ioreadn()` 319e and `iocall()` 317.

⟨*function* _ioreadn 319e⟩≡ (567d)

```
static long
_ioreadn(va_list *arg)
{
    int fd;
    void *a;
    long n;

    fd = va_arg(*arg, int);
    a = va_arg(*arg, void*);
    n = va_arg(*arg, long);
    return readn(fd, a, n);
}
```

Uses `readn()` 192c.

⟨*function* iowrite 320a⟩≡ (568a)
```
long
iowrite(Ioproc *io, int fd, void *a, long n)
{
    return iocall(io, _iowrite, fd, a, n);
}
```
Uses _iowrite() 320b and iocall() 317.

⟨*function* _iowrite 320b⟩≡ (568a)
```
static long
_iowrite(va_list *arg)
{
    int fd;
    void *a;
    long n;

    fd = va_arg(*arg, int);
    a = va_arg(*arg, void*);
    n = va_arg(*arg, long);
    return write(fd, a, n);
}
```
Uses write() 192b.

⟨*function* iosleep 320c⟩≡ (567e)
```
int
iosleep(Ioproc *io, long n)
{
    return iocall(io, _iosleep, n);
}
```
Uses _iosleep() 320d and iocall() 317.

⟨*function* _iosleep 320d⟩≡ (567e)
```
static long
_iosleep(va_list *arg)
{
    long n;

    n = va_arg(*arg, long);
    return sleep(n);
}
```
Uses sleep().

23.10 Thread-aware libc

When `libthread` is linked in, several standard library functions need thread-aware replacements. `malloc` already uses locks, but `_threadmalloc` adds fatal-on-failure semantics and size sanity checks. `threadexits` replaces `exits`—it marks the current thread as moribund and yields to the scheduler rather than killing the process. `threadexitsall` terminates all procs. Similarly, `exec` must be coordinated through the scheduler because only one thread can exec per proc.

23.10.1 Memory

⟨*global* totalmalloc 320e⟩≡ (561b)
```
static long totalmalloc;
```

⟨*function* _threadmalloc 321a⟩≡ (561b)

```
void*
_threadmalloc(long size, int z)
{
    void *m;

    m = malloc(size);
    if (m == nil)
        sysfatal("Malloc of size %ld failed: %r", size);
    setmalloctag(m, getcallerpc(&size));
    totalmalloc += size;
    if (size > 100000000) {
        fprint(2, "Malloc of size %ld, total %ld\n", size, totalmalloc);
        abort();
    }
    if (z)
        memset(m, 0, size);
    return m;
}
```

Uses `abort()` 356d, `fprint()` 75a, `getcallerpc()` 396f, `malloc()` 63g, `memset()` 46b, `setmalloctag()` 69a, `sysfatal()` 236a, and `totalmalloc-2` 320e.

23.10.2 Exit

⟨*global* _threadexitsallstatus 321b⟩≡ (561a)

```
char *_threadexitsallstatus;
```

⟨*function* threadexits 321c⟩≡ (561a)

```
void
threadexits(char *exitstr)
{
    Proc *p;
    Thread *t;

    p = _threadgetproc();
    t = p->thread;
    t->moribund = 1;
    if(exitstr==nil)
        exitstr="";
    utfecpy(p->exitstr, p->exitstr+ERRMAX, exitstr);
    _sched();
}
```

Uses `_sched()` 304a, `_threadgetproc()` 293c, and `utfecpy()` 460b.

⟨*function* threadexitsall 321d⟩≡ (561a)

```
void
threadexitsall(char *exitstr)
{
    Proc *p;
    int pid[64];
    int i, npid, mypid;

    if(exitstr == nil)
        exitstr = "";
    _threadexitsallstatus = exitstr;
    _threaddebug(DBGSCHED, "_threadexitsallstatus set to %p", _threadexitsallstatus);
    mypid = _tos->pid; //getpid();
```

```
	/*
	 * signal others.
	 * copying all the pids first avoids other threads
	 * teardown procedures getting in the way.
	 *
	 * avoid mallocs since malloc can post a note which can
	 * call threadexitsall...
	 */
	for(;;){
		lock(&_threadpq.lock);
		npid = 0;
		for(p = _threadpq.head; p && npid < nelem(pid); p=p->next){
			if(p->threadint == 0 && p->pid != mypid){
				pid[npid++] = p->pid;
				p->threadint = 1;
			}
		}
		unlock(&_threadpq.lock);
		if(npid == 0)
			break;
		for(i=0; i<npid; i++)
			postnote(PNPROC, pid[i], "threadint");
	}

	/* leave */
	exits(exitstr);
}
```

⟨*global* `_threadwaitchan` 322a⟩≡ (561a)

```
Channel *_threadwaitchan;
```

⟨*function* `threadwaitchan` 322b⟩≡ (561a)

```
Channel*
threadwaitchan(void)
{
	if(_threadwaitchan==nil)
		_threadwaitchan = chancreate(sizeof(Waitmsg*), 16);
	return _threadwaitchan;
}
```

Uses `_threadwaitchan` 322a and `chancreate()` 289b.

23.10.3 Fork

`_schedfork` creates a new OS process that shares memory (`RFMEM`) with the parent. The child process calls `longjmp` to jump back to `main`'s `setjmp`, which re-enters `_schedinit` with the new proc as its argument—from there, the scheduler takes over and runs the new proc's threads.

⟨*function* `_schedfork` 322c⟩≡ (565a)

```
int
_schedfork(Proc *p)
{
	int pid;

	switch(pid = rfork(RFPROC|RFMEM|RFNOWAIT|p->rforkflag)){
	case 0:
		*mainp = p; /* write to stack, so local to proc */
		longjmp(_mainjmp, 1);
	default:
		return pid;
```

```
    }
}
```

Uses _mainjmp-19 300c, mainp-20 300b, and rfork().

23.10.4 Exec

Thread-aware exec is tricky because exec replaces the entire process, killing all threads within it. _schedexec forks a child process for the exec, and the parent waits for it via a pipe: the child closes the write end immediately, so if exec succeeds, the parent's read returns 0 bytes (EOF); if it fails, the child writes the error message before exiting.

⟨*function* _schedexec 323a⟩≡ (565a)

```
int
_schedexec(Execargs *e)
{
    int pid;

    switch(pid = rfork(RFREND|RFNOTEG|RFFDG|RFMEM|RFPROC)){
    case 0:
        efork(e);
    default:
        return pid;
    }
}
```

Uses efork() 323b and rfork().

⟨*function* efork 323b⟩≡ (565a)

```
static void
efork(Execargs *e)
{
    char buf[ERRMAX];

    _threaddebug(DBGEXEC, "_schedexec %s", e->prog);
    close(e->fd[0]);
    exec(e->prog, e->args);
    _threaddebug(DBGEXEC, "_schedexec failed: %r");
    rerrstr(buf, sizeof buf);
    if(buf[0]=='\0')
        strcpy(buf, "exec failed");
    write(e->fd[1], buf, strlen(buf));
    close(e->fd[1]);
    _exits(buf);
}
```

Uses DBGEXEC 334f, _exits(), _threaddebug() 334g, close(), exec(), rerrstr() 234a, strcpy() 81c, strlen() 80a, and write() 192b.

⟨*function* _schedexecwait 323c⟩≡ (565a)

```
void
_schedexecwait(void)
{
    int pid;
    Channel *c;
    Proc *p;
    Thread *t;
    Waitmsg *w;

    p = _threadgetproc();
    t = p->thread;
```

```
    pid = t->ret;
    _threaddebug(DBGEXEC, "_schedexecwait %d", t->ret);

    rfork(RFCFDG);
    for(;;){
        w = wait();
        if(w == nil)
            break;
        if(w->pid == pid)
            break;
        free(w);
    }
    if(w != nil){
        if((c = _threadwaitchan) != nil)
            sendp(c, w);
        else
            free(w);
    }
    threadexits("procexec");
}
```

Uses `DBGEXEC` 334f, `_threaddebug()` 334g, `_threadgetproc()` 293c, `_threadwaitchan` 322a, `free()` 63i, `rfork()`, `sendp()` 314c, `threadexits()` 321c, and `wait()` 256a.

⟨*constant* `PIPEMNT` 324a⟩≡ (568c)

```
#define PIPEMNT "/mnt/temp"
```

⟨*function* `procexec` 324b⟩≡ (568c)

```
void
procexec(Channel *pidc, char *prog, char *args[])
{
    int n;
    Proc *p;
    Thread *t;

    _threaddebug(DBGEXEC, "procexec %s", prog);
    /* must be only thread in proc */
    p = _threadgetproc();
    t = p->thread;
    if(p->threads.head != t || p->threads.head->nextt != nil){
        werrstr("not only thread in proc");
    Bad:
        if(pidc)
            sendul(pidc, ~0);
        return;
    }

    /*
     * We want procexec to behave like exec; if exec succeeds,
     * never return, and if it fails, return with errstr set.
     * Unfortunately, the exec happens in another proc since
     * we have to wait for the exec'ed process to finish.
     * To provide the semantics, we open a pipe with the
     * write end close-on-exec and hand it to the proc that
     * is doing the exec.  If the exec succeeds, the pipe will
     * close so that our read below fails.  If the exec fails,
     * then the proc doing the exec sends the errstr down the
     * pipe to us.
     */
    if(bind("#|", PIPEMNT, MREPL) < 0)
        goto Bad;
```

```
    if((p->exec.fd[0] = open(PIPEMNT "/data", OREAD)) < 0){
        unmount(nil, PIPEMNT);
        goto Bad;
    }
    if((p->exec.fd[1] = open(PIPEMNT "/data1", OWRITE|OCEXEC)) < 0){
        close(p->exec.fd[0]);
        unmount(nil, PIPEMNT);
        goto Bad;
    }
    unmount(nil, PIPEMNT);

    /* exec in parallel via the scheduler */
    assert(p->needexec==0);
    p->exec.prog = prog;
    p->exec.args = args;
    p->needexec = 1;
    _sched();

    close(p->exec.fd[1]);
    if((n = read(p->exec.fd[0], p->exitstr, ERRMAX-1)) > 0){ /* exec failed */
        p->exitstr[n] = '\0';
        errstr(p->exitstr, ERRMAX);
        close(p->exec.fd[0]);
        goto Bad;
    }
    close(p->exec.fd[0]);

    if(pidc)
        sendul(pidc, t->ret);

    /* wait for exec'ed program, then exit */
    _schedexecwait();
}
```

Uses `DBGEXEC` 334f, `PIPEMNT-24` 324a, `_sched()` 304a, `_schedexecwait()` 323c, `_threaddebug()` 334g, `_threadgetproc()` 293c, `bind()`, `close()`, `errstr()`, `open()`, `read()` 192a, `sendul()` 314a, `unmount()`, and `werrstr()` 234b.

⟨*function* `procexecl` 325a⟩≡ (568c)

```
void
procexecl(Channel *pidc, char *f, ...)
{
    procexec(pidc, f, &f+1);
}
```

Uses `procexec()` 324b.

23.10.5 Interruptions

When a `rendezvous` call is interrupted by a note, it returns the special value `Intred` (`~0`). The caller must not treat this as a successful exchange—it needs to retry. This is why channel operations loop on `Intred` throughout the code.

⟨*constant* `Intred` 325b⟩≡ (563b)

```
#define Intred ((void*)~0)  /* interrupted */
```

23.10.6 Notes

Thread-aware note handling adds per-process tracking (each note handler records the pid of the process that installed it) and deferred delivery. When a note arrives while a thread holds the channel lock (`splhi`), the note

is queued in a Note buffer and delivered later via delayednotes when the lock is released. This prevents the deadlock that would occur if a note handler tried to perform channel operations while the lock is held.

⟨*global* _threadnopasser 326a⟩≡ (562a)
```
int _threadnopasser;
```

⟨*constant NFN (libthread/note.c)* 326b⟩≡ (562a)
```
#define NFN  33
```

⟨*constant* ERRLEN 326c⟩≡ (562a)
```
#define ERRLEN 48
```

⟨*struct* Note 326d⟩≡ (562a)
```
struct Note
{
    Lock  inuse;
    Proc  *proc;  /* recipient */
    char  s[ERRMAX]; /* arg2 */
};
```

⟨*global* notes 326e⟩≡ (562a)
```
static Note notes[128];
```

⟨*global* enotes 326f⟩≡ (562a)
```
static Note *enotes = notes+nelem(notes);
```
Uses enotes-14 326f and notes-13 326e.

⟨*global* onnote 326g⟩≡ (562a)
```
static int  (*onnote[NFN])(void*, char*);
```
Uses NFN-11 326b.

⟨*global* onnotepid 326h⟩≡ (562a)
```
static int  onnotepid[NFN];
```
Uses NFN-11 326b.

⟨*global* onnotelock 326i⟩≡ (562a)
```
static Lock onnotelock;
```

⟨*function* threadnotify 326j⟩≡ (562a)
```
int
threadnotify(int (*f)(void*, char*), int in)
{
    int i, topid;
    int (*from)(void*, char*), (*to)(void*, char*);

    if(in){
        from = nil;
        to = f;
        topid = _threadgetproc()->pid;
    }else{
        from = f;
        to = nil;
        topid = 0;
    }
    lock(&onnotelock);
    for(i=0; i<NFN; i++)
        if(onnote[i]==from){
            onnote[i] = to;
            onnotepid[i] = topid;
            break;
```

```
        }
    unlock(&onnotelock);
    return i<NFN;
}
```

Uses NFN-11 326b, _threadgetproc() 293c, lock() 245b, onnote-15 326g, onnotelock-17 326i, onnotepid-16 326h, and unlock() 245c.

⟨*function* delayednotes 327a⟩≡ (562a)

```
static void
delayednotes(Proc *p, void *v)
{
    int i;
    Note *n;
    int (*fn)(void*, char*);

    if(!p->pending)
        return;

    p->pending = 0;
    for(n=notes; n<enotes; n++){
        if(n->proc == p){
            for(i=0; i<NFN; i++){
                if(onnotepid[i]!=p->pid || (fn = onnote[i])==nil)
                    continue;
                if((*fn)(v, n->s))
                    break;
            }
            if(i==NFN){
                _threaddebug(DBGNOTE, "Unhandled note %s, proc %p\n", n->s, p);
                if(v != nil)
                    noted(NDFLT);
                else if(strncmp(n->s, "sys:", 4)==0)
                    abort();
                threadexitsall(n->s);
            }
            n->proc = nil;
            unlock(&n->inuse);
        }
    }
}
```

Uses DBGNOTE 334e, NFN-11 326b, _threaddebug() 334g, abort() 356d, enotes-14 326f, noted(), notes-13 326e, onnote-15 326g, onnotepid-16 326h, strncmp() 443c, and unlock() 245c.

⟨*function* _threadnote 327b⟩≡ (562a)

```
void
_threadnote(void *v, char *s)
{
    Proc *p;
    Note *n;

    _threaddebug(DBGNOTE, "Got note %s", s);
    if(strncmp(s, "sys:", 4) == 0)
        noted(NDFLT);

    if(_threadexitsallstatus){
        _threaddebug(DBGNOTE, "Threadexitsallstatus = '%s'\n", _threadexitsallstatus);
        _exits(_threadexitsallstatus);
    }

    if(strcmp(s, "threadint")==0)
```

```
        noted(NCONT);

    p = _threadgetproc();
    if(p == nil)
        noted(NDFLT);

    for(n=notes; n<enotes; n++)
        if(canlock(&n->inuse))
            break;
    if(n==enotes)
        sysfatal("libthread: too many delayed notes");
    utfecpy(n->s, n->s+ERRMAX, s);
    n->proc = p;
    p->pending = 1;
    if(!p->splhi)
        delayednotes(p, v);
    noted(NCONT);
}
```

Uses `DBGNOTE` 334e, `_exits()`, `_threaddebug()` 334g, `_threadexitsallstatus` 321b, `_threadgetproc()` 293c, `canlock()` 245d, `delayednotes()` 327a, `enotes-14` 326f, `noted()`, `notes-13` 326e, `strcmp()` 81a, `strncmp()` 443c, `sysfatal()` 236a, and `utfecpy()` 460b.

⟨*function* `_procsplhi` 328a⟩≡ (562a)

```
int
_procsplhi(void)
{
    int s;
    Proc *p;

    p = _threadgetproc();
    s = p->splhi;
    p->splhi = 1;
    return s;
}
```

Uses `_threadgetproc()` 293c.

⟨*function* `_procsplx` 328b⟩≡ (562a)

```
void
_procsplx(int s)
{
    Proc *p;

    p = _threadgetproc();
    p->splhi = s;
    if(s)
        return;
    if(p->pending)
        delayednotes(p, nil);
}
```

Uses `_threadgetproc()` 293c and `delayednotes()` 327a.

23.10.7 Dialing

The thread-safe `dial` replacement uses an IO proc to perform the blocking connection in a separate process, so other threads in the calling proc can continue running.

⟨*function* `_iodial` 328c⟩≡ (567a)

```
static long
```

```
_iodial(va_list *arg)
{
    char *addr, *local, *dir;
    int *cdfp;

    addr = va_arg(*arg, char*);
    local = va_arg(*arg, char*);
    dir = va_arg(*arg, char*);
    cdfp = va_arg(*arg, int*);

    return dial(addr, local, dir, cdfp);
}
```

Uses `dial()` 347b.

⟨*function* `iodial` 329a⟩≡ (567a)

```
int
iodial(Ioproc *io, char *addr, char *local, char *dir, int *cdfp)
{
    return iocall(io, _iodial, addr, local, dir, cdfp);
}
```

Uses `_iodial()` 328c and `iocall()` 317.

⟨*enum* `_anon_ (libthread/dial.c)` 329b⟩≡ (568b)

```
enum
{
    Maxstring = 128,
    Maxpath   = 256,
};
```

⟨*struct* `DS` *(libthread/dial.c)* 329c⟩≡ (568b)

```
struct DS {
    /* dist string */
    char buf[Maxstring];
    char *netdir;
    char *proto;
    char *rem;

    /* other args */
    char *local;
    char *dir;
    int *cfdp;
};
```

Uses `Maxstring-3` 329b.

⟨*function* `_threaddial` 329d⟩≡ (568b)

```
/*
 *  the dialstring is of the form '[/net/]proto!dest'
 */
int
_threaddial(char *dest, char *local, char *dir, int *cfdp)
{
    DS ds;
    int rv;
    char err[ERRMAX], alterr[ERRMAX];

    ds.local = local;
    ds.dir = dir;
    ds.cfdp = cfdp;
```

```
    _dial_string_parse(dest, &ds);
    if(ds.netdir)
        return csdial(&ds);

    ds.netdir = "/net";
    rv = csdial(&ds);
    if(rv >= 0)
        return rv;
    err[0] = '\0';
    errstr(err, sizeof err);
    if(strstr(err, "refused") != 0){
        werrstr("%s", err);
        return rv;
    }
    ds.netdir = "/net.alt";
    rv = csdial(&ds);
    if(rv >= 0)
        return rv;

    alterr[0] = 0;
    errstr(alterr, sizeof alterr);
    if(strstr(alterr, "translate") || strstr(alterr, "does not exist"))
        werrstr("%s", err);
    else
        werrstr("%s", alterr);
    return rv;
}
```

Uses _dial_string_parse() 332, errstr(), strstr() 82b, and werrstr() 234b.

⟨*function* csdial *(libthread/dial.c)* 330⟩≡ (568b)

```
static int
csdial(DS *ds)
{
    int n, fd, rv;
    char *p, buf[Maxstring], clone[Maxpath], err[ERRMAX], besterr[ERRMAX];

    /*
     *  open connection server
     */
    snprint(buf, sizeof(buf), "%s/cs", ds->netdir);
    fd = open(buf, ORDWR);
    if(fd < 0){
        /* no connection server, don't translate */
        snprint(clone, sizeof(clone), "%s/%s/clone", ds->netdir, ds->proto);
        return call(clone, ds->rem, ds);
    }

    /*
     *  ask connection server to translate
     */
    snprint(buf, sizeof(buf), "%s!%s", ds->proto, ds->rem);
    if(write(fd, buf, strlen(buf)) < 0){
        close(fd);
        return -1;
    }

    /*
     *  loop through each address from the connection server till
     *  we get one that works.
     */
```

```
    *besterr = 0;
    rv = -1;
    seek(fd, 0, 0);
    while((n = read(fd, buf, sizeof(buf) - 1)) > 0){
        buf[n] = 0;
        p = strchr(buf, ' ');
        if(p == 0)
            continue;
        *p++ = 0;
        rv = call(buf, p, ds);
        if(rv >= 0)
            break;
        err[0] = '\0';
        errstr(err, sizeof err);
        if(strstr(err, "does not exist") == 0)
            strcpy(besterr, err);
    }
    close(fd);

    if(rv < 0 && *besterr)
        werrstr("%s", besterr);
    else
        werrstr("%s", err);
    return rv;
}
```

Uses `Maxpath-4` 329b, `Maxstring-3` 329b, `close()`, `errstr()`, `open()`, `read()` 192a, `seek()`, `snprint()` 535b, `strchr()` 80c, `strcpy()` 81c, `strlen()` 80a, `strstr()` 82b, `werrstr()` 234b, and `write()` 192b.

⟨*function* `call` *(libthread/dial.c)* 331⟩≡ (568b)

```
static int
call(char *clone, char *dest, DS *ds)
{
    int fd, cfd, n;
    char cname[Maxpath], name[Maxpath], data[Maxpath], *p;

    /* because cs is in a different name space, replace the mount point */
    if(*clone == '/'){
        p = strchr(clone+1, '/');
        if(p == nil)
            p = clone;
        else
            p++;
    } else
        p = clone;
    snprint(cname, sizeof cname, "%s/%s", ds->netdir, p);

    cfd = open(cname, ORDWR);
    if(cfd < 0)
        return -1;

    /* get directory name */
    n = read(cfd, name, sizeof(name)-1);
    if(n < 0){
        close(cfd);
        return -1;
    }
    name[n] = 0;
    for(p = name; *p == ' '; p++)
        ;
    snprint(name, sizeof(name), "%ld", strtoul(p, 0, 0));
```

```
    p = strrchr(cname, '/');
    *p = 0;
    if(ds->dir)
        snprint(ds->dir, NETPATHLEN, "%s/%s", cname, name);
    snprint(data, sizeof(data), "%s/%s/data", cname, name);

    /* connect */
    if(ds->local)
        snprint(name, sizeof(name), "connect %s %s", dest, ds->local);
    else
        snprint(name, sizeof(name), "connect %s", dest);
    if(write(cfd, name, strlen(name)) < 0){
        close(cfd);
        return -1;
    }

    /* open data connection */
    fd = open(data, ORDWR);
    if(fd < 0){
        close(cfd);
        return -1;
    }
    if(ds->cfdp)
        *ds->cfdp = cfd;
    else
        close(cfd);
    return fd;
}
```

Uses `Maxpath-4` 329b, `close()`, `open()`, `read()` 192a, `snprint()` 535b, `strchr()` 80c, `strlen()` 80a, `strrchr()` 80d, `strtoul()` 449, and `write()` 192b.

⟨*function* `_dial_string_parse` *(libthread/dial.c)* 332⟩≡ (568b)

```
/*
 *  parse a dial string
 */
static void
_dial_string_parse(char *str, DS *ds)
{
    char *p, *p2;

    strncpy(ds->buf, str, Maxstring);
    ds->buf[Maxstring-1] = 0;

    p = strchr(ds->buf, '!');
    if(p == 0) {
        ds->netdir = 0;
        ds->proto = "net";
        ds->rem = ds->buf;
    } else {
        if(*ds->buf != '/' && *ds->buf != '#'){
            ds->netdir = 0;
            ds->proto = ds->buf;
        } else {
            for(p2 = p; *p2 != '/'; p2--)
                ;
            *p2++ = 0;
            ds->netdir = ds->buf;
            ds->proto = p2;
        }
        *p = 0;
```

```
        ds->rem = p + 1;
    }
}
```
Uses Maxstring-3 329b, strchr() 80c, and strncpy() 443e.

23.11 Error management

Thread library errors are fatal: rather than returning error codes that threads might not check, the library calls abort to crash the program. This philosophy ("crash early, crash loudly") ensures that bugs like double-free, stack overflow, or channel misuse are caught immediately rather than silently corrupting state.

⟨*function* _threadsysfatal 333a⟩≡ (561b)
```
void
_threadsysfatal(char *fmt, va_list arg)
{
    char buf[1024]; /* size doesn't matter; we're about to exit */

    vseprint(buf, buf+sizeof(buf), fmt, arg);
    if(argv0)
        fprint(2, "%s: %s\n", argv0, buf);
    else
        fprint(2, "%s\n", buf);
    threadexitsall(buf);
}
```
Uses argv0, fprint() 75a, and vseprint() 536b.

⟨*function* _threadassert 333b⟩≡ (560d)
```
void
_threadassert(char *s)
{
    char buf[256];
    int n;
    Proc *p;

    p = _threadgetproc();
    if(p && p->thread)
        n = sprint(buf, "%d.%d ", p->pid, p->thread->id);
    else
        n = 0;
    snprint(buf+n, sizeof(buf)-n, "%s: assertion failed\n", s);
    write(STDERR, buf, strlen(buf));
    abort();
}
```
Uses _threadgetproc() 293c, abort() 356d, snprint() 535b, sprint() 75b, strlen() 80a, and write() 192b.

23.12 Debugging

The threading library has built-in debug tracing controlled by _threaddebuglevel, a bitmask of categories (scheduling, channels, rendezvous, etc.). Setting _threaddebuglevel = DBGSCHED|DBGCHAN produces a detailed trace of every context switch and channel operation—helpful for diagnosing deadlocks or starvation.

⟨*global* _threaddebuglevel 333c⟩≡ (560d)
```
// biset<enum<dbgxxx>>
int _threaddebuglevel;
```

⟨*constant* DBGAPPL 334a⟩≡ (558c)
```
#define DBGAPPL (1 << 0)
```

⟨*constant* DBGSCHED 334b⟩≡ (558c)
```
#define DBGSCHED (1 << 16)
```

⟨*constant* DBGCHAN 334c⟩≡ (558c)
```
#define DBGCHAN (1 << 17)
```

⟨*constant* DBGREND 334d⟩≡ (558c)
```
#define DBGREND (1 << 18)
```

⟨*constant* DBGNOTE 334e⟩≡ (558c)
```
#define DBGNOTE (1 << 20)
```

⟨*constant* DBGEXEC 334f⟩≡ (558c)
```
#define DBGEXEC (1 << 21)
```

⟨*function* _threaddebug 334g⟩≡ (560d)
```
void
_threaddebug(ulong flag, char *fmt, ...)
{
    char buf[128];
    va_list arg;
    Fmt f;
    Proc *p;

    if((_threaddebuglevel&flag) == 0)
        return;

    fmtfdinit(&f, 2, buf, sizeof buf);

    p = _threadgetproc();
    if(p==nil)
        fmtprint(&f, "noproc ");
    else if(p->thread)
        fmtprint(&f, "%d.%d ", p->pid, p->thread->id);
    else
        fmtprint(&f, "%d._ ", p->pid);

    va_start(arg, fmt);
    fmtvprint(&f, fmt, arg);
    va_end(arg);
    fmtprint(&f, "\n");
    fmtfdflush(&f);
}
```
Uses _threaddebuglevel 333c, _threadgetproc() 293c, fmtfdflush() 522a, fmtfdinit() 522b, fmtprint() 523e, and fmtvprint() 529d.

23.13 Advanced topics

⟨*global* _threadnotefd 334h⟩≡ (565a)
```
int _threadnotefd;
```

⟨*global* _threadpasserpid 334i⟩≡ (565a)
```
int _threadpasserpid;
```

⟨*function* _times 335a⟩≡ (565a)
```
static long
_times(long *t)
{
    char b[200], *p;
    int f;
    ulong r;

    memset(b, 0, sizeof(b));
    f = open("/dev/cputime", OREAD|OCEXEC);
    if(f < 0)
        return 0;
    if(read(f, b, sizeof(b)) <= 0){
        close(f);
        return 0;
    }
    p = b;
    if(t)
        t[0] = atol(p);
    p = skip(p);
    if(t)
        t[1] = atol(p);
    p = skip(p);
    r = atol(p);
    if(t){
        p = skip(p);
        t[2] = atol(p);
        p = skip(p);
        t[3] = atol(p);
    }
    return r;
}
```
Uses `atol()` 123c, `close()`, `memset()` 46b, `open()`, and `read()` 192a.

⟨*function skip (libthread/main.c)* 335b⟩≡ (565a)
```
static char*
skip(char *p)
{
    while(*p == ' ')
        p++;
    while(*p != ' ' && *p != 0)
        p++;
    return p;
}
```

23.13.1 Per thread private data

Thread-local storage is provided by `tprivalloc/tprivaddr`: each thread has an array of `NPRIV` (8) void pointers in its `Thread` structure. `tprivalloc` allocates a slot index, and `tprivaddr` returns a pointer to that slot in the current thread. This is the Plan 9 equivalent of POSIX `pthread_key_create/pthread_getspecific`.

⟨*global privlock (libthread/id.c)* 335c⟩≡ (560e)
```
static Lock privlock;
```

⟨*global* privmask 335d⟩≡ (560e)
```
//array<bool> NPRIV at least
static int privmask = 1;
```
Uses `privmask-26` 335d.

⟨*function* tprivalloc 336a⟩≡ (560e)
```
int
tprivalloc(void)
{
    int i;

    lock(&privlock);
    for(i=0; i<NPRIV; i++)
        if(!(privmask&(1<<i))){
            privmask |= 1<<i;
            unlock(&privlock);
            return i;
        }
    unlock(&privlock);
    return -1;
}
```
Uses NPRIV 295b, lock() 245b, privlock-25 335c, privmask-26 335d, and unlock() 245c.

⟨*function* tprivfree 336b⟩≡ (560e)
```
void
tprivfree(int i)
{
    if(i < 0 || i >= NPRIV)
        abort();
    lock(&privlock);
    privmask &= ~(1<<i);
}
```
Uses NPRIV 295b, abort() 356d, lock() 245b, privlock-25 335c, and privmask-26 335d.

⟨*function* tprivaddr 336c⟩≡ (560e)
```
void**
tprivaddr(int i)
{
    return &_threadgetproc()->thread->udata[i];
}
```
Uses _threadgetproc() 293c.

⟨*function* threaddata 336d⟩≡ (560e)
```
void**
threaddata(void)
{
    return &_threadgetproc()->thread->udata[0];
}
```
Uses _threadgetproc() 293c.

⟨*function* _workerdata 336e⟩≡ (560e)
```
void**
_workerdata(void)
{
    return &_threadgetproc()->wdata;
}
```
Uses _threadgetproc() 293c.

⟨*function* procdata 336f⟩≡ (560e)
```
void**
procdata(void)
{
    return &_threadgetproc()->udata;
}
```
Uses _threadgetproc() 293c.

23.13.2 Thread groups

Thread groups allow killing or interrupting multiple threads at once. Each thread inherits its creator's group ID, and **threadkillgrp** marks all threads in the group as moribund. This is useful for cleaning up a set of worker threads when a task is cancelled.

⟨*function* `threadsetgrp` 337a⟩≡ (560e)
```
int
threadsetgrp(int ng)
{
    int og;
    Thread *t;

    t = _threadgetproc()->thread;
    og = t->grp;
    t->grp = ng;
    return og;
}
```
Uses `_threadgetproc()` 293c.

⟨*function* `threadgetgrp` 337b⟩≡ (560e)
```
int
threadgetgrp(void)
{
    return _threadgetproc()->thread->grp;
}
```
Uses `_threadgetproc()` 293c.

23.13.3 Thread kills

⟨*function* `tinterrupt` 337c⟩≡ (565b)
```
static void
tinterrupt(Proc *p, Thread *t)
{
    switch(t->state){
    case Running:
        postnote(PNPROC, p->pid, "threadint");
        break;
    case Rendezvous:
        _threadflagrendez(t);
        break;
    }
}
```
Uses `Rendezvous` 298c, `Running` 298c, `_threadflagrendez()` 340c, and `postnote()` 258c.

⟨*function* `threadkillgrp` 337d⟩≡ (565b)
```
void
threadkillgrp(int grp)
{
    threadxxxgrp(grp, 1);
}
```
Uses `threadxxxgrp()` 338a.

⟨*function* `threadkill` 337e⟩≡ (565b)
```
void
threadkill(int id)
{
    threadxxx(id, 1);
}
```
Uses `threadxxx()` 338b.

⟨*function* threadxxxgrp 338a⟩≡ (565b)
```
static void
threadxxxgrp(int grp, int dokill)
{
    Proc *p;
    Thread *t;

    lock(&_threadpq.lock);
    for(p=_threadpq.head; p; p=p->next){
        lock(&p->lock);
        for(t=p->threads.head; t; t=t->nextt)
            if(t->grp == grp){
                if(dokill)
                    t->moribund = 1;
                tinterrupt(p, t);
            }
        unlock(&p->lock);
    }
    unlock(&_threadpq.lock);
    _threadbreakrendez();
}
```
Uses _threadbreakrendez() 340d, _threadpq 293e, lock() 245b, tinterrupt() 337c, and unlock() 245c.

⟨*function* threadxxx 338b⟩≡ (565b)
```
static void
threadxxx(int id, int dokill)
{
    Proc *p;
    Thread *t;

    lock(&_threadpq.lock);
    for(p=_threadpq.head; p; p=p->next){
        lock(&p->lock);
        for(t=p->threads.head; t; t=t->nextt)
            if(t->id == id){
                if(dokill)
                    t->moribund = 1;
                tinterrupt(p, t);
                unlock(&p->lock);
                unlock(&_threadpq.lock);
                _threadbreakrendez();
                return;
            }
        unlock(&p->lock);
    }
    unlock(&_threadpq.lock);
    _threaddebug(DBGNOTE, "Can't find thread to kill");
    return;
}
```
Uses DBGNOTE 334e, _threadbreakrendez() 340d, _threaddebug() 334g, _threadpq 293e, lock() 245b, tinterrupt() 337c, and unlock() 245c.

⟨*function* threadintgrp 338c⟩≡ (565b)
```
void
threadintgrp(int grp)
{
    threadxxxgrp(grp, 0);
}
```
Uses threadxxxgrp() 338a.

⟨*function* `threadint` 339a⟩≡ (565b)
```
void
threadint(int id)
{
    threadxxx(id, 0);
}
```
Uses `threadxxx()` 338b.

23.13.4 Rendez vous

`_threadrendezvous` is the thread-level replacement for the kernel `rendezvous` system call. Instead of blocking in the kernel, it uses a hash table of waiting threads (`_threadrgrp`): if a matching tag is found, the two threads exchange values immediately; otherwise, the current thread joins the hash table and yields. This allows threads within the same proc to rendezvous without leaving user space.

⟨*constant* `RENDHASH` 339b⟩≡ (558b)
```
RENDHASH = 13,
```

⟨*struct* `Rgrp` 339c⟩≡ (558c)
```
struct Rgrp
{
    Lock lock;
    Thread *hash[RENDHASH];
};
```
Uses `RENDHASH` 339b.

⟨*global* `_threadrgrp` 339d⟩≡ (562b)
```
Rgrp _threadrgrp;
```

⟨*function* `_threadrendezvous` 339e⟩≡ (562b)
```
void*
_threadrendezvous(void *tag, void *val)
{
    void *ret;
    Thread *t, **l;

    lock(&_threadrgrp.lock);
    l = &_threadrgrp.hash[((uintptr)tag)%nelem(_threadrgrp.hash)];
    for(t=*l; t; l=&t->rendhash, t=*l){
        if(t->rendtag==tag){
            _threaddebug(DBGREND, "Rendezvous with thread %d.%d", t->proc->pid, t->id);
            *l = t->rendhash;
            ret = finish(t, val);
            unlock(&_threadrgrp.lock);
            return ret;
        }
    }

    /* Going to sleep here. */
    t = _threadgetproc()->thread;
    t->rendbreak = 0;
    t->inrendez = 1;
    t->rendtag = tag;
    t->rendval = val;
    t->rendhash = *l;
    *l = t;
    t->nextstate = Rendezvous;
    _threaddebug(DBGREND, "Rendezvous for tag %p", t->rendtag);
```

```
    unlock(&_threadrgrp.lock);
    _sched();
    t->inrendez = 0;
    _threaddebug(DBGREND, "Woke after rendezvous; val is %p", t->rendval);
    return t->rendval;
}
```

Uses DBGREND 334d, Rendezvous 298c, _sched() 304a, _threaddebug() 334g, _threadgetproc() 293c, _threadrgrp 339d, finish() 340b, lock() 245b, and unlock() 245c.

⟨*global* isdirty 340a⟩≡ (562b)

```
static int isdirty;
```

⟨*function* finish 340b⟩≡ (562b)

```
static void*
finish(Thread *t, void *val)
{
    void *ret;

    ret = t->rendval;
    t->rendval = val;
    while(t->state == Running)
        sleep(0);
    lock(&t->proc->lock);
    if(t->state == Rendezvous){ /* not always true: might be Dead */
        t->state = Ready;
        _threadready(t);
    }
    unlock(&t->proc->lock);
    return ret;
}
```

Uses Ready 298c, Rendezvous 298c, Running 298c, _threadready() 305a, lock() 245b, sleep(), and unlock() 245c.

⟨*function* _threadflagrendez 340c⟩≡ (562b)

```
/*
 * This is called while holding _threadpq.lock and p->lock,
 * so we can't lock _threadrgrp.lock.  Instead our caller has
 * to call _threadbreakrendez after dropping those locks.
 */
void
_threadflagrendez(Thread *t)
{
    t->rendbreak = 1;
    isdirty = 1;
}
```

Uses isdirty-10 340a.

⟨*function* _threadbreakrendez 340d⟩≡ (562b)

```
void
_threadbreakrendez(void)
{
    int i;
    Thread *t, **l;

    if(isdirty == 0)
        return;
    lock(&_threadrgrp.lock);
    if(isdirty == 0){
        unlock(&_threadrgrp.lock);
        return;
```

```
    }
    isdirty = 0;
    for(i=0; i<nelem(_threadrgrp.hash); i++){
        l = &_threadrgrp.hash[i];
        for(t=*l; t; t=*l){
            if(t->rendbreak){
                *l = t->rendhash;
                finish(t, (void*)~0);
            }else
                 l=&t->rendhash;
        }
    }
    unlock(&_threadrgrp.lock);
}
```

Uses _threadrgrp 339d, finish() 340b, isdirty-10 340a, lock() 245b, and unlock() 245c.

23.13.5 chanprint()

chanprint formats a string and sends it through a channel in a single call—a convenience for the common pattern of sending formatted text between threads.

⟨*function* chanprint 341⟩≡ (566d)

```
int
chanprint(Channel *c, char *fmt, ...)
{
    va_list arg;
    char *p;
    int n;

    va_start(arg, fmt);
    p = vsmprint(fmt, arg);
    va_end(arg);
    if(p == nil)
        sysfatal("vsmprint failed: %r");
    n = sendp(c, p);
    yield(); /* let recipient handle message immediately */
    return n;
}
```

Uses sendp() 314c, sysfatal() 236a, vsmprint() 537b, and yield() 305b.

Chapter 24

Network

Plan 9's network API is radically different from UNIX sockets. Instead of a `socket/bind/listen/accept` sequence, networking is done through the file system: `dial("tcp!host!port")` opens a connection by writing to `/net/tcp/clone`, and `announce` listens by reading from `/net/tcp/*/listen`. The `NetConnInfo` structure holds the parsed connection metadata (local/remote addresses, service names), all obtained by reading files under the connection directory.

⟨*signatures networking functions* 342a⟩≡ (368b)

```
extern  int     dial(char*, char*, char*, int*);
extern  int     accept(int, char*);
extern  int     announce(char*, char*);
extern  int     listen(char*, char*);
```

⟨*type* `NetConnInfo` 342b⟩≡ (368b)

```
struct NetConnInfo {
    char    *dir;       /* connection directory */
    char    *root;      /* network root */
    char    *spec;      /* binding spec */
    char    *lsys;      /* local system */
    char    *lserv;     /* local service */
    char    *rsys;      /* remote system */
    char    *rserv;     /* remote service */
    char    *laddr;     /* local address */
    char    *raddr;     /* remote address */
};
```

24.1 Example: an echo server

A Plan 9 echo server is much shorter than its BSD sockets equivalent: `announce("tcp!*!echo", dir)` to listen, `listen(dir, newdir)` to accept, then `read/write` on `accept(ctl, newdir)`. There is no `socket`, `bind`, `setsockopt`, or `sockaddr_in`—the file system abstraction handles all of that.

The complete echo server fits in a few dozen lines:

```
void
main(void)
{
    char dir[40], ndir[40], buf[1024];
    int actl, lctl, fd, n;

    actl = announce("tcp!*!7", dir);
    if(actl < 0)
```

```
        sysfatal("announce: %r");
    for(;;){
        lctl = listen(dir, ndir);
        if(lctl < 0)
            sysfatal("listen: %r");
        if(fork() == 0){
            fd = accept(lctl, ndir);
            if(fd >= 0){
                while((n = read(fd, buf, sizeof buf)) > 0)
                    write(fd, buf, n);
                close(fd);
            }
            exits(nil);
        }
        close(lctl);
    }
}
```

What `announce` really does is open `/net/tcp/clone`, which causes the TCP device to allocate a fresh connection slot (say, `/net/tcp/42`), and write `announce 7` to its `ctl` file. After `announce` returns, the running namespace contains, among other things:

```
/net/tcp/42/ctl       control file (fd = actl)
/net/tcp/42/listen    blocks until a client arrives
/net/tcp/42/local     "192.168.1.4!7"
/net/tcp/42/remote    (empty until connected)
/net/tcp/42/status    "Listen"
```

`listen` then opens `/net/tcp/42/listen` and blocks until a SYN arrives. When a client connects, the open returns a control fd for a fresh connection (say, slot 43), and `accept` opens slot 43's `data` file—a plain bidirectional byte stream that we can `read` and `write` like any other file. The kernel TCP stack does the protocol work, but from the program's point of view it is just files. This is also why an SSL/TLS layer can be slid in by `bind`ing a TLS file server over `/net`: the echo server above continues to work without recompilation.

24.2 netmkaddr()

`netmkaddr` fills in missing parts of a network address using defaults. A full Plan 9 address has the form `net!host!service` (e.g., `tcp!example.com!80`). If the user provides just a hostname, `netmkaddr` prepends the default network and appends the default service. This lets programs accept flexible address formats while ensuring the result is always a complete dial string.

⟨*function* `netmkaddr` 343⟩≡ (406f)

```
/*
 *  make an address, add the defaults
 */
char *
netmkaddr(char *linear, char *defnet, char *defsrv)
{
    static char addr[256];
    char *cp;

    /*
     *  dump network name
```

```
     */
    cp = strchr(linear, '!');
    if(cp == 0){
        if(defnet==0){
            if(defsrv)
                snprint(addr, sizeof(addr), "net!%s!%s",
                    linear, defsrv);
            else
                snprint(addr, sizeof(addr), "net!%s", linear);
        }
        else {
            if(defsrv)
                snprint(addr, sizeof(addr), "%s!%s!%s", defnet,
                    linear, defsrv);
            else
                snprint(addr, sizeof(addr), "%s!%s", defnet,
                    linear);
        }
        return addr;
    }

    /*
     *  if there is already a service, use it
     */
    cp = strchr(cp+1, '!');
    if(cp)
        return linear;

    /*
     *  add default service
     */
    if(defsrv == 0)
        return linear;
    snprint(addr, sizeof(addr), "%s!%s", linear, defsrv);

    return addr;
}
```
Uses `snprint()` 535b and `strchr()` 80c.

24.3 announce(), accept(), listen(), reject()

The server-side API follows Plan 9's file system model: `announce` opens `/net/tcp/clone` to get a new connection directory, then writes `announce addr` to its control file. `listen` opens the `listen` file in that directory, which blocks until a client connects. `accept` opens the `data` file to get a bidirectional byte stream. `reject` writes a rejection message to the control file. This file-based approach means the entire networking stack is accessible through the namespace. A program can `bind` an alternative network stack (e.g., over TLS) to `/net` and all network programs automatically use it, with no code changes.

⟨*function* `announce` 344⟩≡ (464)
```
/*
 *  announce a network service.
 */
int
announce(char *addr, char *dir)
{
    int ctl, n, m;
    char buf[Maxpath];
```

```
    char buf2[Maxpath];
    char netdir[Maxpath];
    char naddr[Maxpath];
    char *cp;

    /*
     *  translate the address
     */
    if(nettrans(addr, naddr, sizeof(naddr), netdir, sizeof(netdir)) < 0)
        return -1;

    /*
     * get a control channel
     */
    ctl = open(netdir, ORDWR);
    if(ctl<0){
        werrstr("announce opening %s: %r", netdir);
        return -1;
    }
    cp = strrchr(netdir, '/');
    if(cp == nil){
        werrstr("announce arg format %s", netdir);
        close(ctl);
        return -1;
    }
    *cp = 0;

    /*
     *  find out which line we have
     */
    n = snprint(buf, sizeof(buf), "%s/", netdir);
    m = read(ctl, &buf[n], sizeof(buf)-n-1);
    if(m <= 0){
        werrstr("announce reading %s: %r", netdir);
        close(ctl);
        return -1;
    }
    buf[n+m] = 0;

    /*
     *  make the call
     */
    n = snprint(buf2, sizeof(buf2), "announce %s", naddr);
    if(write(ctl, buf2, n)!=n){
        werrstr("announce writing %s: %r", netdir);
        close(ctl);
        return -1;
    }

    /*
     *  return directory etc.
     */
    if(dir){
        strncpy(dir, buf, NETPATHLEN);
        dir[NETPATHLEN-1] = 0;
    }
    return ctl;
}
```

Uses `Maxpath-230` 462g, `close()`, `nettrans()` 463b, `open()`, `read()` 192a, `snprint()` 535b, `strncpy()` 443e, `strrchr()` 80d,

werrstr() 234b, and write() 192b.

⟨*function* listen 346a⟩≡ (464)

```
/*
 *  listen for an incoming call
 */
int
listen(char *dir, char *newdir)
{
    int ctl, n, m;
    char buf[Maxpath];
    char *cp;

    /*
     *  open listen, wait for a call
     */
    snprint(buf, sizeof(buf), "%s/listen", dir);
    ctl = open(buf, ORDWR);
    if(ctl < 0){
        werrstr("listen opening %s: %r", buf);
        return -1;
    }

    /*
     *  find out which line we have
     */
    strncpy(buf, dir, sizeof(buf) - 1);
    buf[sizeof(buf) - 1] = 0;
    cp = strrchr(buf, '/');
    if(cp == nil){
        close(ctl);
        werrstr("listen arg format %s", dir);
        return -1;
    }
    *++cp = 0;
    n = cp-buf;
    m = read(ctl, cp, sizeof(buf) - n - 1);
    if(m <= 0){
        close(ctl);
        werrstr("listen reading %s/listen: %r", dir);
        return -1;
    }
    buf[n+m] = 0;

    /*
     *  return directory etc.
     */
    if(newdir){
        strncpy(newdir, buf, NETPATHLEN);
        newdir[NETPATHLEN-1] = 0;
    }
    return ctl;

}
```

Uses Maxpath-230 462g, close(), open(), read() 192a, snprint() 535b, strncpy() 443e, strrchr() 80d, and werrstr() 234b.

⟨*function* accept 346b⟩≡ (464)

```
/*
 *  accept a call, return an fd to the open data file
 */
```

```
int
accept(int ctl, char *dir)
{
    char buf[Maxpath];
    char *num;
    long n;

    num = strrchr(dir, '/');
    if(num == nil)
        num = dir;
    else
        num++;

    n = snprint(buf, sizeof(buf), "accept %s", num);
    write(ctl, buf, n); /* ignore return value, network might not need accepts */

    snprint(buf, sizeof(buf), "%s/data", dir);
    return open(buf, ORDWR);
}
```

Uses `Maxpath-230` 462g, `open()`, `snprint()` 535b, `strrchr()` 80d, and `write()` 192b.

⟨*function* `reject` 347a⟩≡ (464)

```
/*
 *  reject a call, tell device the reason for the rejection
 */
int
reject(int ctl, char *dir, char *cause)
{
    char buf[Maxpath];
    char *num;
    long n;

    num = strrchr(dir, '/');
    if(num == 0)
        num = dir;
    else
        num++;
    snprint(buf, sizeof(buf), "reject %s %s", num, cause);
    n = strlen(buf);
    if(write(ctl, buf, n) != n)
        return -1;
    return 0;
}
```

Uses `Maxpath-230` 462g, `snprint()` 535b, `strlen()` 80a, `strrchr()` 80d, and `write()` 192b.

24.4 dial()

`dial` opens a network connection to a remote address. The implementation goes through a function pointer (`_dial`) that `libthread` replaces with a thread-safe version—the default `dialimpl` uses `rfork` internally, which conflicts with `libthread`'s process management. The dial string is parsed into network, host, and service components, and the connection is established by writing to the appropriate files under `/net`.

⟨*function* `dial` 347b⟩≡ (476c)

```
int
dial(char *dest, char *local, char *dir, int *cfdp)
{
    return (*_dial)(dest, local, dir, cfdp);
```

```
}
```

Uses _dial 348a.

⟨*global* _dial 348a⟩≡ (476c)

```
/*
 * the thread library can't cope with rfork(RFMEM|RFPROC),
 * so it must override this with a private version of dial.
 */
int (*_dial)(char *, char *, char *, int *) = dialimpl;
```

Uses _dial 348a and dialimpl() 348b.

⟨*function* dialimpl 348b⟩≡ (476c)

```
/*
 *  the dialstring is of the form '[/net/]proto!dest'
 */
static int
dialimpl(char *dest, char *local, char *dir, int *cfdp)
{
    DS ds;
    int rv;
    char err[ERRMAX], alterr[ERRMAX];

    ds.local = local;
    ds.dir = dir;
    ds.cfdp = cfdp;

    _dial_string_parse(dest, &ds);
    if(ds.netdir)
        return csdial(&ds);

    ds.netdir = "/net";
    rv = csdial(&ds);
    if(rv >= 0)
        return rv;
    err[0] = '\0';
    errstr(err, sizeof err);
    if(strstr(err, "refused") != 0){
        werrstr("%s", err);
        return rv;
    }
    ds.netdir = "/net.alt";
    rv = csdial(&ds);
    if(rv >= 0)
        return rv;

    alterr[0] = 0;
    errstr(alterr, sizeof alterr);
    if(strstr(alterr, "translate") || strstr(alterr, "does not exist"))
        werrstr("%s", err);
    else
        werrstr("%s", alterr);
    return rv;
}
```

Uses _dial_string_parse() 332, errstr(), strstr() 82b, and werrstr() 234b.

⟨*function* csdial *(9sys/dial.c)* 348c⟩≡ (476c)

```
static int
csdial(DS *ds)
{
```

```
int n, fd, rv, addrs, bleft;
char c;
char *addrp, *clone2, *dest;
char buf[Maxstring], clone[Maxpath], err[ERRMAX], besterr[ERRMAX];
Dest *dp;

werrstr("");
dp = mallocz(sizeof *dp, 1);
if(dp == nil)
    return -1;
dp->winner = -1;
dp->oalarm = alarm(0);
if (connsalloc(dp, 1) < 0) {        /* room for a single conn. */
    freedest(dp);
    return -1;
}

/*
 *  open connection server
 */
snprint(buf, sizeof(buf), "%s/cs", ds->netdir);
fd = open(buf, ORDWR);
if(fd < 0){
    /* no connection server, don't translate */
    snprint(clone, sizeof(clone), "%s/%s/clone", ds->netdir, ds->proto);
    rv = call(clone, ds->rem, ds, dp, &dp->conn[0]);
    fillinds(ds, dp);
    freedest(dp);
    return rv;
}

/*
 *  ask connection server to translate
 *  e.g., net!cs.bell-labs.com!smtp
 */
snprint(buf, sizeof(buf), "%s!%s", ds->proto, ds->rem);
if(write(fd, buf, strlen(buf)) < 0){
    close(fd);
    freedest(dp);
    return -1;
}

/*
 *  read all addresses from the connection server:
 *  /net/tcp/clone 135.104.9.78!25
 *  /net/tcp/clone 2620:0:dc0:1805::29!25
 *
 *  assumes that we'll get one record per read.
 */
seek(fd, 0, 0);
addrs = 0;
addrp = dp->nextaddr = dp->addrlist;
bleft = sizeof dp->addrlist - 2;    /* 2 is room for \n\0 */
while(bleft > 0 && (n = read(fd, addrp, bleft)) > 0) {
    if (addrp[n-1] != '\n')
        addrp[n++] = '\n';
    addrs++;
    addrp += n;
    bleft -= n;
}
```

```
    *addrp = '\0';

    /*
     * if we haven't read all of cs's output, assume the last line might
     * have been truncated and ignore it.  we really don't expect this
     * to happen.
     */
    if (addrs > 0 && bleft <= 0 && read(fd, &c, 1) == 1)
        addrs--;
    close(fd);

    *besterr = 0;
    rv = -1;                /* pessimistic default */
    dp->naddrs = addrs;
    if (addrs == 0)
        werrstr("no address to dial");
    else if (addrs == 1) {
        /* common case: dial one address without forking */
        if (parsecs(dp, &clone2, &dest) >= 0 &&
            (rv = call(clone2, dest, ds, dp, &dp->conn[0])) < 0) {
            pickuperr(besterr, err);
            werrstr("%s", besterr);
        }
    } else if (connsalloc(dp, addrs) >= 0)
        rv = dialmulti(ds, dp);

    /* fill in results */
    if (rv >= 0 && dp->winner >= 0)
        rv = fillinds(ds, dp);

    freedest(dp);
    return rv;
}
```

Uses `Maxpath-227` 471, `Maxstring-226` 471, `alarm()`, `close()`, `connsalloc()` 472d, `dialmulti()` 476b, `fillinds()` 474c, `freedest()` 473a, `mallocz()` 67a, `open()`, `parsecs()` 475a, `pickuperr()` 475b, `read()` 192a, `seek()`, `snprint()` 535b, `strlen()` 80a, `werrstr()` 234b, and `write()` 192b.

⟨*function* `call` *(9sys/dial.c)* 350⟩≡ (476c)

```
static int
call(char *clone, char *dest, DS *ds, Dest *dp, Conn *conn)
{
    int fd, cfd, n, calleralarm, oalarm;
    char cname[Maxpath], name[Maxpath], data[Maxpath], *p;

    /* because cs is in a different name space, replace the mount point */
    if(*clone == '/'){
        p = strchr(clone+1, '/');
        if(p == nil)
            p = clone;
        else
            p++;
    } else
        p = clone;
    snprint(cname, sizeof cname, "%s/%s", ds->netdir, p);

    conn->pid = getpid();
    conn->cfd = cfd = open(cname, ORDWR);
    if(cfd < 0)
        return -1;
```

```
    /* get directory name */
    n = read(cfd, name, sizeof(name)-1);
    if(n < 0){
        closeopenfd(&conn->cfd);
        return -1;
    }
    name[n] = 0;
    for(p = name; *p == ' '; p++)
        ;
    snprint(name, sizeof(name), "%ld", strtoul(p, 0, 0));
    p = strrchr(cname, '/');
    *p = 0;
    if(ds->dir)
        snprint(conn->dir, NETPATHLEN, "%s/%s", cname, name);
    snprint(data, sizeof(data), "%s/%s/data", cname, name);

    /* should be no alarm pending now; re-instate caller's alarm, if any */
    calleralarm = dp->oalarm > 0;
    if (calleralarm)
        alarm(dp->oalarm);
    else if (dp->naddrs > 1)    /* in a sub-process? */
        alarm(Maxconnms);

    /* connect */
    if(ds->local)
        snprint(name, sizeof(name), "connect %s %s", dest, ds->local);
    else
        snprint(name, sizeof(name), "connect %s", dest);
    if(write(cfd, name, strlen(name)) < 0){
        closeopenfd(&conn->cfd);
        return -1;
    }

    oalarm = alarm(0);  /* don't let alarm interrupt critical section */
    if (calleralarm)
        dp->oalarm = oalarm;    /* time has passed, so update user's */

    /* open data connection */
    conn->dfd = fd = open(data, ORDWR);
    if(fd < 0){
        closeopenfd(&conn->cfd);
        alarm(dp->oalarm);
        return -1;
    }
    if(ds->cfdp == nil)
        closeopenfd(&conn->cfd);

    n = conn - dp->conn;
    if (dp->winner < 0) {
        qlock(&dp->winlck);
        if (dp->winner < 0 && conn < dp->connend)
            dp->winner = n;
        qunlock(&dp->winlck);
    }
    alarm(calleralarm? dp->oalarm: 0);
    return fd;
}
```

Uses `Maxconnms-229` 471, `Maxpath-227` 471, `alarm()`, `closeopenfd()` 473b, `getpid()` 232a, `open()`, `qlock()` 246c, `qunlock()` 247a, `read()` 192a, `snprint()` 535b, `strchr()` 80c, `strlen()` 80a, `strrchr()` 80d, `strtoul()` 449,

and `write()` 192b.

⟨*function* `_dial_string_parse`*(9sys/dial.c)* 352⟩≡ (476c)

```
/*
 *  parse a dial string
 */
static void
_dial_string_parse(char *str, DS *ds)
{
    char *p, *p2;

    strncpy(ds->buf, str, Maxstring);
    ds->buf[Maxstring-1] = 0;

    p = strchr(ds->buf, '!');
    if(p == 0) {
        ds->netdir = 0;
        ds->proto = "net";
        ds->rem = ds->buf;
    } else {
        if(*ds->buf != '/' && *ds->buf != '#'){
            ds->netdir = 0;
            ds->proto = ds->buf;
        } else {
            /* expecting /net.alt/tcp!foo or #I1/tcp!foo */
            for(p2 = p; p2 > ds->buf && *p2 != '/'; p2--)
                ;
            *p2++ = 0;
            ds->netdir = ds->buf;
            ds->proto = p2;
        }
        *p = 0;
        ds->rem = p + 1;
    }
}
```

Part V

Debugging and Profiling

This part covers the library support for the tools described in the DEBUGGER book [Pad16b] and PROFILER book [Pad26b]: `syslog` for writing to log files, `assert` and `abort` for catching invariant violations, and the `_profin`/`_profout` functions that the linker inserts at every function entry and exit when profiling is enabled.

Chapter 25

Debugging Support

Debugging support includes `syslog` for writing to `/sys/log/` files, `assert` for compile-time invariants, and the memory allocator's built-in corruption detection (magic numbers, caller PC recording). The `abort` function generates a segmentation fault to produce a core dump for post-mortem analysis with `acid` (see DEBUGGER book [Pad16b]).

⟨*signatures logging functions* 354a⟩≡ (368b)

```
extern  void    syslog(int, char*, char*, ...);

#pragma varargck    argpos  syslog  3
```

⟨*signatures debugging functions* 354b⟩≡ (368b)

```
extern void abort(void);
```

25.1 Logging

`syslog` writes timestamped messages to `/sys/log/logname`. It caches the file descriptor but validates it on each call by comparing the `Dir` (via `eqdirdev`) against the cached copy—this detects the case where a `fork` has closed the fd behind our back. If the log file can't be opened and `cons` is set, the message goes to the system console (`#c/cons`) as a fallback. The error string is saved and restored across the call so that logging doesn't disturb `%r`.

⟨*function* `syslog` 354c⟩≡ (493f)

```
/*
 * Print
 *  sysname: time: mesg
 * on /sys/log/logname.
 * If cons or log file can't be opened, print on the system console, too.
 */
void
syslog(int cons, char *logname, char *fmt, ...)
{
    char buf[1024];
    char *ctim, *p;
    va_list arg;
    int n;
    Dir *d;
    char err[ERRMAX];

    err[0] = '\0';
    errstr(err, sizeof err);
    lock(&sl);
```

```
/*
 *  paranoia makes us stat to make sure a fork+close
 *  hasn't broken our fd's
 */
d = dirfstat(sl.fd);
if(sl.fd < 0 || sl.name == nil || strcmp(sl.name, logname) != 0 ||
   !eqdirdev(d, sl.d)){
    free(sl.name);
    sl.name = strdup(logname);
    if(sl.name == nil)
        cons = 1;
    else{
        free(sl.d);
        sl.d = nil;
        _syslogopen();
        if(sl.fd < 0)
            cons = 1;
        else
            sl.d = dirfstat(sl.fd);
    }
}
free(d);
if(cons){
    d = dirfstat(sl.consfd);
    if(sl.consfd < 0 || !eqdirdev(d, sl.consd)){
        free(sl.consd);
        sl.consd = nil;
        sl.consfd = open("#c/cons", OWRITE|OCEXEC);
        if(sl.consfd >= 0)
            sl.consd = dirfstat(sl.consfd);
    }
    free(d);
}

if(fmt == nil){
    unlock(&sl);
    return;
}

ctim = ctime(time(0));
p = buf + snprint(buf, sizeof(buf)-1, "%s ", sysname());
strncpy(p, ctim+4, 15);
p += 15;
*p++ = ' ';
errstr(err, sizeof err);
va_start(arg, fmt);
p = vseprint(p, buf+sizeof(buf)-1, fmt, arg);
va_end(arg);
*p++ = '\n';
n = p - buf;

if(sl.fd >= 0){
    seek(sl.fd, 0, 2);
    write(sl.fd, buf, n);
}

if(cons && sl.consfd >=0)
    write(sl.consfd, buf, n);

unlock(&sl);
```

```
}
```

Uses _syslogopen() 356b, ctime() 467d, dirfstat() 215b, eqdirdev() 356c, errstr(), free() 63i, lock() 245b, open(), seek(), sl-245 356a, snprint() 535b, strcmp() 81a, strdup() 82a, strncpy() 443e, sysname() 494a, time() 226a, unlock() 245c, vseprint() 536b, and write() 192b.

⟨*global* sl 356a⟩≡ (493f)

```
static struct
{
    int fd;
    int consfd;
    char    *name;
    Dir *d;
    Dir *consd;
    Lock;
} sl =
{
    -1, -1,
};
```

Uses __anon_struct_34 356a.

⟨*function* _syslogopen 356b⟩≡ (493f)

```
static void
_syslogopen(void)
{
    char buf[1024];

    if(sl.fd >= 0)
        close(sl.fd);
    snprint(buf, sizeof(buf), "/sys/log/%s", sl.name);
    sl.fd = open(buf, OWRITE|OCEXEC);
}
```

Uses close(), open(), sl-245 356a, and snprint() 535b.

⟨*function* eqdirdev 356c⟩≡ (493f)

```
static int
eqdirdev(Dir *a, Dir *b)
{
    return a != nil && b != nil &&
        a->dev == b->dev && a->type == b->type &&
        a->qid.path == b->qid.path;
}
```

25.2 abort()

⟨*function* abort 356d⟩≡ (462e)

```
void
abort(void)
{
    while(*(int*)0)
        ;
}
```

Chapter 26

Profiling Support

When a program is linked with `5l -p`, the linker inserts calls to `_profin` and `_profout` at every function entry and exit (see LINKER book [Pad15b] and PROFILER book [Pad26b]). These functions maintain a call tree of `Plink` nodes, recording the caller PC and invocation count. `_profdump` writes this data to a file that the `prof` command reads to produce human-readable profiles.

⟨*signatures profiling functions* 357a⟩≡ (368b)

```
extern  void    prof(void (*fn)(void*), void *arg, int entries, int what);
```

26.1 Data structures

⟨*type* Prof 357b⟩≡ (368b)

```
enum Profiling {
    Profoff,        /* No profiling */

    Profuser,       /* Measure user time only (default) */
    Profkernel,     /* Measure user + kernel time */
    Proftime,       /* Measure total time */
    Profsample,     /* Use clock interrupt to sample (default when there is no cycle counter) */
}; /* what */
```

⟨*struct* Plink 357c⟩≡ (432b)

```
struct  Plink
{
    Plink   *old;
    Plink   *down;
    Plink   *link;
    long    pc;
    long    count;
    vlong time;
};
```

⟨*global* khz 357d⟩≡ (432b)

```
static  ulong   khz;
```

⟨*global* perr 357e⟩≡ (432b)

```
static  ulong   perr;
```

⟨*global* havecycles 357f⟩≡ (432b)

```
static  int havecycles;
```

26.2 prof()

⟨*function* prof 358a⟩≡ (432b)
```
void
prof(void (*fn)(void*), void *arg, int entries, int what)
{
    _profinit(entries, what);
    _tos->prof.pp = _tos->prof.next;
    fn(arg);
    _profdump();
}
```
Uses _profdump() 361 and _profinit() 360a.

26.3 _profin()

_profin is called at every function entry. It walks down the Plink tree from the current node (pp) looking for a child with a matching PC. If none exists, it allocates a new node from the pre-allocated array (prof.first through prof.last). It then records the current time using the appropriate measurement method (cycle counter, kernel cycles, or clock sample) by subtracting it from the node's accumulator—the matching subtraction in _profout will yield the elapsed time.

The subtract-on-entry/add-on-exit accounting deserves a walkthrough. Suppose p->time starts at zero, and we enter a function at cycle counter t1=100 and leave at t2=140. _profin does p->time -= t1, so p->time = -100. _profout does p->time += t2, so p->time = -100 + 140 = 40, the elapsed cycles. The same trick works recursively across nested calls—each level subtracts on entry and adds on exit, so the self-time of each function is its own subtract+add minus the children's. Equivalently, p->time always holds (sum of exit times so far) minus (sum of entry times so far); when the function is not currently on the stack these balance and give the cumulative inclusive time.

```
                         t=100        t=110         t=130      t=140
 outer  enter outer ----.                                     ^---- exit outer
                        |  -100                          +140 |
                        v                                     |
                           inner enter ---.       ----- exit inner
                                          -110  +130
                                            v       ^
                                              ...

 outer.time at end = -100 + 140 = 40   (40 cycles inclusive)
 inner.time at end = -110 + 130 = 20   (20 cycles inclusive)
```

Note that the cleverness pays for itself only because the subtract and add are O(1) and can be inlined into the prologue/epilogue insertion that the linker is doing anyway—an additive accumulator avoids any per-function "start time" field.

⟨*function* _profin 358b⟩≡ (432b)
```
// Called at every procedure entry when use 5l -p, see Linker.nw
ulong
_profin(void)
{
    void *dummy;
    long pc;
    Plink *pp, *p;
    ulong arg;
```

```
    vlong t;

    arg = _savearg();
    pc = _callpc(&dummy);
    pp = _tos->prof.pp;
    if(pp == 0 || (_tos->prof.pid && _tos->pid != _tos->prof.pid))
        return arg;

    for(p=pp->down; p; p=p->link)
        if(p->pc == pc)
            goto out;
    p = _tos->prof.next + 1;
    if(p >= _tos->prof.last) {
        _tos->prof.pp = 0;
        perr++;
        return arg;
    }
    _tos->prof.next = p;
    p->link = pp->down;
    pp->down = p;
    p->pc = pc;
    p->old = pp;
    p->down = 0;
    p->count = 0;
    p->time = 0LL;

out:
    _tos->prof.pp = p;
    p->count++;
    switch(_tos->prof.what){
    case Profkernel:
        p->time = p->time - _tos->pcycles;
        goto proftime;
    case Profuser:
        /* Add kernel cycles on proc entry */
        p->time = p->time + _tos->kcycles;
        /* fall through */
    case Proftime:
    proftime:       /* Subtract cycle counter on proc entry */
        cycles((uvlong*)&t);
        p->time = p->time - t;
        break;
    case Profsample:
        p->time = p->time - _tos->clock;
        break;
    }
    return arg;     /* disgusting linkage */
}
```

Uses perr 357e.

26.4 _profout()

⟨*function* _profout 359⟩≡ (432b)

```
// Called at every procedure return when use 5l -p, see Linker.nw
ulong
_profout(void)
{
    Plink *p;
```

```
    ulong arg;
    vlong t;

    arg = _savearg();
    p = _tos->prof.pp;
    if (p == nil || (_tos->prof.pid != 0 && _tos->pid != _tos->prof.pid))
        return arg; /* Not our process */
    switch(_tos->prof.what){
    case Profkernel:        /* Add proc cycles on proc entry */
        p->time = p->time + _tos->pcycles;
        goto proftime;
    case Profuser:          /* Subtract kernel cycles on proc entry */
        p->time = p->time - _tos->kcycles;
        /* fall through */
    case Proftime:
    proftime:               /* Add cycle counter on proc entry */
        cycles((uvlong*)&t);
        p->time = p->time + t;
        break;
    case Profsample:
        p->time = p->time + _tos->clock;
        break;
    }
    _tos->prof.pp = p->old;
    return arg;
}
```

26.5 _profinit()

⟨*function* `_profinit` 360a⟩≡ (432b)

```
void
_profinit(int entries, int what)
{
    if (_tos->prof.what == 0)
        return; /* Profiling not linked in */
    _tos->prof.pp = nil;
    _tos->prof.first = mallocz(entries*sizeof(Plink),1);
    _tos->prof.last = _tos->prof.first + entries;
    _tos->prof.next = _tos->prof.first;
    _tos->prof.pid = _tos->pid;
    _tos->prof.what = what;
    _tos->clock = 1;
}
```

Uses `mallocz()` 67a.

26.6 _profmain()

`_profmain` is called from the startup code when the program was linked with profiling enabled. It reads the `/env/profsize` and `/env/proftype` files to configure the number of `Plink` entries and the measurement method, allocates the profiling buffer with `sbrk` (not `malloc`, since the allocator might not be initialized yet), and registers `_profdump` as an `atexit` handler so the profile data is written when the program exits.

⟨*function* `_profmain` 360b⟩≡ (432b)

```
// called by _mainp in _main9p.s when 5l -p, see Linker
void
_profmain(void)
```

```
{
    char ename[50];
    int n, f;

    n = 2000;
    if (_tos->cyclefreq != 0LL){
        khz = _tos->cyclefreq / 1000;   /* Report times in milliseconds */
        havecycles = 1;
    }
    f = open("/env/profsize", OREAD);
    if(f >= 0) {
        memset(ename, 0, sizeof(ename));
        read(f, ename, sizeof(ename)-1);
        close(f);
        n = atol(ename);
    }
    _tos->prof.what = Profuser;
    f = open("/env/proftype", OREAD);
    if(f >= 0) {
        memset(ename, 0, sizeof(ename));
        read(f, ename, sizeof(ename)-1);
        close(f);
        if (strcmp(ename, "user") == 0)
            _tos->prof.what = Profuser;
        else if (strcmp(ename, "kernel") == 0)
            _tos->prof.what = Profkernel;
        else if (strcmp(ename, "elapsed") == 0 || strcmp(ename, "time") == 0)
            _tos->prof.what = Proftime;
        else if (strcmp(ename, "sample") == 0)
            _tos->prof.what = Profsample;
    }
    _tos->prof.first = sbrk(n*sizeof(Plink));
    _tos->prof.last = sbrk(0);
    _tos->prof.next = _tos->prof.first;
    _tos->prof.pp = nil;
    _tos->prof.pid = _tos->pid;
    atexit(_profdump);
    _tos->clock = 1;
}
```

Uses `_profdump()` 361, `atexit()` 45d, `atol()` 123c, `close()`, `havecycles` 357f, `khz` 357d, `memset()` 46b, `open()`, `read()` 192a, `sbrk()` 58a, and `strcmp()` 81a.

26.7 _profdump()

`_profdump` writes the profiling data to a file named `prof.<pid>` (or `prof.out`). The output format is a flat array of 16-byte records: a 2-byte child index, a 2-byte sibling index, a 4-byte PC, a 4-byte call count, and a 4-byte accumulated time. The `prof` command (see Profiler book [Pad26b]) reads this file and combines it with the symbol table to produce a human-readable profile.

⟨*function* `_profdump` 361⟩≡ (432b)

```
// called by??
void
_profdump(void)
{
    int f;
    long n;
    Plink *p;
    char *vp;
```

```
char filename[64];

if (_tos->prof.what == 0)
    return; /* No profiling */
if (_tos->prof.pid != 0 && _tos->pid != _tos->prof.pid)
    return; /* Not our process */
if(perr)
    fprint(2, "%lud Prof errors\n", perr);
_tos->prof.pp = nil;
if (_tos->prof.pid)
    snprint(filename, sizeof filename - 1, "prof.%ld", _tos->prof.pid);
else
    snprint(filename, sizeof filename - 1, "prof.out");
f = create(filename, 1, 0666);
if(f < 0) {
    perror("create prof.out");
    return;
}
_tos->prof.pid = ~0;    /* make sure data gets dumped once */
switch(_tos->prof.what){
case Profkernel:
    cycles((uvlong*)&_tos->prof.first->time);
    _tos->prof.first->time = _tos->prof.first->time + _tos->pcycles;
    break;
case Profuser:
    cycles((uvlong*)&_tos->prof.first->time);
    _tos->prof.first->time = _tos->prof.first->time - _tos->kcycles;
    break;
case Proftime:
    cycles((uvlong*)&_tos->prof.first->time);
    break;
case Profsample:
    _tos->prof.first->time = _tos->clock;
    break;
}
vp = (char*)_tos->prof.first;

for(p = _tos->prof.first; p <= _tos->prof.next; p++) {

    /*
     * short down
     */
    n = 0xffff;
    if(p->down)
        n = p->down - _tos->prof.first;
    vp[0] = n>>8;
    vp[1] = n;

    /*
     * short right
     */
    n = 0xffff;
    if(p->link)
        n = p->link - _tos->prof.first;
    vp[2] = n>>8;
    vp[3] = n;
    vp += 4;

    /*
     * long pc
```

```
         */
        n = p->pc;
        vp[0] = n>>24;
        vp[1] = n>>16;
        vp[2] = n>>8;
        vp[3] = n;
        vp += 4;

        /*
         * long count
         */
        n = p->count;
        vp[0] = n>>24;
        vp[1] = n>>16;
        vp[2] = n>>8;
        vp[3] = n;
        vp += 4;

        /*
         * vlong time
         */
        if (havecycles){
            n = (vlong)(p->time / (vlong)khz);
        }else
            n = p->time;

        vp[0] = n>>24;
        vp[1] = n>>16;
        vp[2] = n>>8;
        vp[3] = n;
        vp += 4;
    }
    write(f, (char*)_tos->prof.first, vp - (char*)_tos->prof.first);
    close(f);
}
```

Uses `close()`, `create()`, `fprint()` 75a, `havecycles` 357f, `khz` 357d, `perr` 357e, `snprint()` 535b, and `write()` 192b.

26.8 `cputime()`

`cputime` returns the total CPU time (user + system + children) consumed by the process, in seconds. It reads the raw tick counts via `times` and sums all four components (user time, system time, children's user time, children's system time).

⟨*constant* `HZ` 363a⟩≡ (466e)

```
#define HZ  1000
```

⟨*function* `cputime` 363b⟩≡ (466e)

```
double
cputime(void)
{
    long t[4];
    int i;

    times(t);
    for(i=1; i<4; i++)
        t[0] += t[i];
    return t[0] / (double)HZ;
}
```

Uses `HZ-246` 363a and `times()` 495a.

Chapter 27

Conclusion

You now know how the Plan 9 core libraries work, to the smallest details, and more generally how many C standard libraries work.

This book has covered the infrastructure that every Plan 9 program depends on: the `_main()` startup code that runs before `main()`, the `Pool` memory allocator behind `malloc()/free()`, the UTF-8 string functions and `Rune` type for Unicode, buffered I/O with `Biobuf`, the `rfork()/exec()/wait()` process model, the cooperative thread scheduler in `libthread` with its `Channel`-based communication, the `dial()` networking interface, and the `libregexp` regular expression engine. Across all of these, the code is compact and direct—the entire set of libraries fits in a modest amount of C and a little ARM assembly.

27.1 Patterns and techniques

These techniques apply far beyond system programming:

- *Buffered I/O*: batching small reads and writes into larger system calls. The same amortization pattern appears in Java's `BufferedReader`, Go's `bufio`, and hardware write-back caches. Whenever per-call overhead dominates, batching is the standard fix.
- *Arena-based memory allocation*: the `Pool` allocator manages memory by coalescing free blocks and splitting large ones. The same allocator serves both user space and the kernel, parameterized by different backing functions. This pattern—one algorithm, multiple deployment contexts—appears in database buffer pools, GPU memory managers, and JVM heap allocators. Any system that manages a scarce resource in chunks faces the same fragmentation and coalescing trade-offs.
- *Channels instead of locks*: `libthread`'s threads communicate via `Channel`s rather than shared memory. Go's goroutines and channels are a direct descendant of this design; Erlang's mailboxes follow the same principle: share by communicating, don't communicate by sharing.
- *Variable-width encoding*: UTF-8 keeps ASCII unchanged while encoding wider characters in multi-byte sequences. The same design tension—fixed-width wastes space, variable-width complicates indexing—arises in protocol buffers (varints) and database row formats.
- *Connection string abstraction*: `dial()` encodes protocol, host, and service in one string and returns a file descriptor. JDBC URLs, MongoDB connection strings, and Redis URIs all use the same idea: one opaque string that the library knows how to open.

27.2 Connections to other books

The core libraries are the foundation on which every other program in Principia Softwarica is built:

- KERNEL book [Pad14]: the kernel implements the system calls that the libraries wrap—`sbrk()` for the memory allocator, `rfork()` and `exec()` for processes, `open()`/`read()`/`write()` for I/O. The kernel also shares the `Pool` allocator with user space.
- SHELL book [Pad18]: `rc` uses the process model (`rfork`, `exec`, `wait`, `pipe`) extensively. Understanding the library wrappers clarifies what the shell does at the system call level.
- WINDOWS book [Pad16d]: `rio` uses `libthread` for its multi-threaded architecture, with procs, threads, and channels as the concurrency primitives described in this book.
- COMPILER book [Pad16a]: `5c` compiles code that links against these libraries. The calling conventions, stack layout, and register usage described in the assembly chapter directly affect how library functions are called.
- LINKER book [Pad15b]: `5l` links the library object files into executables. The startup code (`_main`) and the profiling hooks (`_profin`/ `_profout`) are defined in the libraries but inserted by the linker.
- DEBUGGER book [Pad16b]: `libmach`, also part of the core libraries, provides the executable parsing and disassembly that the debugger uses.

27.3 Beyond the Plan 9 core libraries

Modern C libraries and runtime systems are considerably larger. Here are some of the areas where they differ:

- *Memory allocators*: the Plan 9 `Pool` allocator is a straightforward first-fit allocator with red-zone poisoning for debugging. Production allocators like jemalloc (FreeBSD, Firefox), tcmalloc (Google), and mimalloc (Microsoft) use thread-local caches, size-segregated freelists, and arena-based allocation to minimize contention in multi-threaded programs. These are some of the most performance-critical pieces of infrastructure in modern systems.
- *Thread libraries*: `libthread` implements cooperative (non-preemptive) user-level threads multiplexed onto OS processes. POSIX pthreads provides preemptive kernel-level threads with mutexes, condition variables, and read-write locks. Go's goroutine scheduler takes a hybrid approach: user-level goroutines are multiplexed onto OS threads by a work-stealing scheduler, combining the lightweight creation of `libthread` with the preemptive scheduling of pthreads.
- *Asynchronous I/O*: `libthread`'s model is synchronous—threads block on I/O and the scheduler switches to another thread. Modern systems provide `epoll` (Linux), `kqueue` (BSD/macOS), and `io_uring` (Linux) for handling thousands of concurrent connections efficiently without a thread per connection.
- *Unicode and internationalization*: Plan 9 pioneered UTF-8 (it was invented for Plan 9 by Ken Thompson and Rob Pike), and `libc`'s `Rune` type handles Unicode code points cleanly. Modern libraries like ICU go much further with locale-aware collation, text segmentation, bidirectional text, and complex script shaping—the infrastructure needed for worldwide internationalization.
- *Standard library scope*: the Plan 9`libc` is intentionally small. The C standard library (C11/C23) specifies threads, atomics, complex math, and wide character support. Beyond C, languages like Rust, Go, and Python ship standard libraries with collections, HTTP clients, JSON parsers, cryptography, and more—entire ecosystems that Plan 9 leaves to separate libraries or external tools.
- *Networking*: Plan 9's `dial()` provides a clean, protocol-independent interface for establishing connections. Modern systems add TLS as a baseline requirement, HTTP/2 and HTTP/3 (QUIC) protocol support, and DNS-over-HTTPS—reflecting a world where network security is no longer optional.

The Plan 9 core libraries show that the essential building blocks of systems programming—memory allocation, string handling, process management, concurrency, and I/O—can be implemented clearly in a modest amount of code. Modern libraries are larger because they optimize for different constraints (multi-core scalability, thousands of concurrent connections, worldwide internationalization), but the core abstractions remain recognizable.

Appendix A

Extra Code

A.1 include/

A.1.1 include/arch/arm/u.h

```
⟨include/arch/arm/u.h 368a⟩≡

  ⟨type uxxx 26b⟩
  ⟨type uxxxint 27a⟩
  ⟨type xxxvlong 27b⟩

  typedef signed char schar;
  ⟨type usize 27c⟩

  ⟨constant nil 31a⟩
  ⟨type uintptr 31b⟩

  ⟨type Rune 30a⟩
  ⟨type mpdigit 156⟩

  ⟨type jmpbufxxx 253c⟩
  ⟨type jmp_buf 253b⟩

  ⟨type FPxxx 127b⟩
  ⟨type FPdbleword 27d⟩
  typedef     union FPdbleword FPdbleword;

  ⟨type va_list 34b⟩

  ⟨macro va_start 34c⟩
  ⟨macro va_end 34d⟩
  ⟨macro va_arg 34e⟩
```

A.1.2 include/core/libc.h

```
⟨include/core/libc.h 368b⟩≡
  #pragma lib "libc.a"
  #pragma src "/sys/src/libc"

  typedef struct Fmt Fmt;
  typedef struct Tm Tm;
  typedef struct Lock Lock;
  typedef struct QLp QLp;
  typedef struct QLock QLock;
```

```
typedef struct RWLock RWLock;
typedef struct Rendez Rendez;
typedef struct NetConnInfo NetConnInfo;
typedef struct Qid Qid;
typedef struct Dir Dir;
typedef struct Waitmsg Waitmsg;
typedef struct IOchunk IOchunk;

//*****************************************************************************
// Foundations
//*****************************************************************************

//----------------------------------------------------------------------------
// Pad's core types
//----------------------------------------------------------------------------

// More types! Types are good!
⟨type bool 26a⟩
⟨type byte 26c⟩
typedef uchar bool_byte;

//typedef char* string; // conflict
//typedef char* filename; // conflict in sam with function filename

enum _ord {
  ORD__EQ = 0,
  ORD__INF = -1,
  ORD__SUP = 1,
};
typedef int ord;

⟨constant STDxxx 35b⟩
⟨type fdt 35a⟩

⟨constant OKxxx 235b⟩
⟨constant ERRORxxx 235c⟩
⟨type errorxxx 235a⟩

//----------------------------------------------------------------------------
// Macros (Array/Struct)
//----------------------------------------------------------------------------

⟨function nelem 31c⟩
⟨function offsetof 34a⟩

//----------------------------------------------------------------------------
// Exception
//----------------------------------------------------------------------------

extern  int     setjmp(jmp_buf);
extern  void    longjmp(jmp_buf, int);

// ??
extern  void    notejmp(void*, jmp_buf, int);


//----------------------------------------------------------------------------
// Malloc/free
//----------------------------------------------------------------------------
```

```
/*
 * malloc
 */
⟨signatures memory management functions 55⟩
// less useful
extern  void*   calloc(ulong, ulong);
extern  void*   mallocalign(ulong, ulong, long, ulong);

// internals
extern  void    setmalloctag(void*, ulong);
extern  void    setrealloctag(void*, ulong);
extern  ulong   getmalloctag(void*);
extern  ulong   getrealloctag(void*);

extern  void*   malloctopoolblock(void*);

//*****************************************************************************
// Data processing
//*****************************************************************************

//----------------------------------------------------------------------------
// Mem ops
//----------------------------------------------------------------------------

/*
 * mem routines
 */
⟨signatures of memxxx functions 46a⟩
// less useful?
extern  void*   memccpy(void*, void*, int, ulong);

//----------------------------------------------------------------------------
// Str ops
//----------------------------------------------------------------------------

/*
 * string routines
 */
// memxxx equivalent, but with special handling for '\0' (no need pass ulong)
⟨signatures of strxxx functions 78a⟩

extern  int tolower(int);
extern  int toupper(int);

// less useful?
extern  char*   strecpy(char*, char*, char*);
extern  char*   strncat(char*, char*, long);
extern  char*   strncpy(char*, char*, long);
extern  int     strncmp(char*, char*, long);
extern  char*   strrchr(char*, int);

extern  char*   strpbrk(char*, char*);

extern  long    strspn(char*, char*);
extern  long    strcspn(char*, char*);

extern  int     cistrncmp(char*, char*, int);
extern  int     cistrcmp(char*, char*);
extern  char*   cistrstr(char*, char*);
```

```
extern  char*   strtok(char*, char*);
extern  int     tokenize(char*, char**, int);

//-----------------------------------------------------------------------------
// Runes
//-----------------------------------------------------------------------------

enum
{
    ⟨constant UTFmax 30d⟩
    ⟨constant Runesync 30e⟩
    ⟨constant Runeself 30f⟩
    ⟨constant Runeerror 30g⟩
    ⟨constant Runemax 30b⟩
    ⟨constant Runemask 30c⟩
};

/*
 * rune routines
 */

// strxxx equivalent for rune
⟨signatures of runexxx functions 78b⟩

// less useful
extern  Rune*   runestrrchr(Rune*, Rune);
extern  Rune*   runestrecpy(Rune*, Rune*, Rune*);
extern  Rune*   runestrncpy(Rune*, Rune*, long);
extern  Rune*   runestrncat(Rune*, Rune*, long);
extern  int     runestrncmp(Rune*, Rune*, long);

extern  Rune    tolowerrune(Rune);
extern  Rune    toupperrune(Rune);

extern  int isalpharune(Rune);
extern  int isdigitrune(Rune);
extern  int islowerrune(Rune);
extern  int isupperrune(Rune);
extern  int isspacerune(Rune);

// ??
extern  Rune    totitlerune(Rune);
extern  Rune    tobaserune(Rune);
extern  int istitlerune(Rune);
extern  int isbaserune(Rune);

// ?????
extern  int runelen(long);
extern  int runenlen(Rune*, int);
extern  int fullrune(char*, int);

//-----------------------------------------------------------------------------
// UTF8
//-----------------------------------------------------------------------------

// char <-> rune conversion
⟨signatures rune conversion functions 78c⟩

// strxxx equivalent for utf
⟨signatures utfxxx functions 78d⟩
```

```
// less useful?
extern  int utfnlen(char*, long);
extern  char*   utfecpy(char*, char*, char*);

/* claude: 9front-style IDN helpers; currently stubs in
 * lib_security/libsec/port/idnstub.c (used by the merged 9front x509.c). */
extern int idn2utf(char*, char*, int);
extern int utf2idn(char*, char*, int);


//-----------------------------------------------------------------------------
// Print
//-----------------------------------------------------------------------------

/*
 * print routines
 */
⟨type Fmt 71b⟩

⟨type Fmt_flag 72a⟩

// pad: used to be just print() but for cg transformed in a pointer func
⟨signatures xxxprint functions 71a⟩

extern  int     vfprint(fdt, char*, va_list);
extern  char*   vseprint(char*, char*, char*, va_list);

extern  int     runesprint(Rune*, char*, ...);

// less useful
extern  char*   smprint(char*, ...);

extern  int     vsnprint(char*, int, char*, va_list);
extern  char*   vsmprint(char*, va_list);

extern  Rune*   runeseprint(Rune*, Rune*, char*, ...);
extern  int     runesnprint(Rune*, int, char*, ...);
extern  Rune*   runesmprint(char*, ...);

extern  Rune*   runevseprint(Rune*, Rune*, char*, va_list);
extern  int     runevsnprint(Rune*, int, char*, va_list);
extern  Rune*   runevsmprint(char*, va_list);

extern  int     fmtfdinit(Fmt*, int, char*, int);

extern  int     fmtfdflush(Fmt*);
extern  int     fmtstrinit(Fmt*);
extern  char*   fmtstrflush(Fmt*);
extern  int     runefmtstrinit(Fmt*);
extern  Rune*   runefmtstrflush(Fmt*);

#pragma varargck    argpos  fmtprint    2
#pragma varargck    argpos  runeseprint 3
#pragma varargck    argpos  runesmprint 1
#pragma varargck    argpos  runesnprint 3
#pragma varargck    argpos  runesprint  2
#pragma varargck    argpos  smprint     1

// %d = decimal, o = octal, x = hexa, b = binary?
```

```
#pragma varargck    type    "d" int
#pragma varargck    type    "o" int
#pragma varargck    type    "x" int
#pragma varargck    type    "b" int
// %f
#pragma varargck    type    "f" double
#pragma varargck    type    "e" double
#pragma varargck    type    "g" double
// %s
#pragma varargck    type    "s" char*
#pragma varargck    type    "q" char*
#pragma varargck    type    "S" Rune*
#pragma varargck    type    "Q" Rune*
#pragma varargck    type    "r" void
#pragma varargck    type    "%" void
#pragma varargck    type    "n" int*
#pragma varargck    type    "p" uintptr
#pragma varargck    type    "p" void*
#pragma varargck    type    "c" int
#pragma varargck    type    "C" int
#pragma varargck    type    "d" uint
#pragma varargck    type    "x" uint
#pragma varargck    type    "b" uint
#pragma varargck    type    "c" uint
#pragma varargck    type    "C" uint
#pragma varargck    type    "<" void*
#pragma varargck    type    "[" void*
#pragma varargck    type    "H" void*
#pragma varargck    type    "lH"    void*

#pragma varargck    type    "lld"   vlong
#pragma varargck    type    "llo"   vlong
#pragma varargck    type    "llx"   vlong
#pragma varargck    type    "llb"   vlong

#pragma varargck    type    "lld"   uvlong
#pragma varargck    type    "llo"   uvlong
#pragma varargck    type    "llx"   uvlong
#pragma varargck    type    "llb"   uvlong

#pragma varargck    type    "ld"    long
#pragma varargck    type    "lo"    long
#pragma varargck    type    "lx"    long
#pragma varargck    type    "lb"    long

#pragma varargck    type    "ld"    ulong
#pragma varargck    type    "lo"    ulong
#pragma varargck    type    "lx"    ulong
#pragma varargck    type    "lb"    ulong


#pragma varargck    flag    ','
#pragma varargck    flag    ' '
#pragma varargck    flag    'h'



extern  int dofmt(Fmt*, char*);
extern  int dorfmt(Fmt*, Rune*);
extern  int fmtprint(Fmt*, char*, ...);
```

```
extern  int fmtvprint(Fmt*, char*, va_list);
extern  int fmtrune(Fmt*, int);
extern  int fmtstrcpy(Fmt*, char*);
extern  int fmtrunestrcpy(Fmt*, Rune*);

/*
 * error string for %r
 * supplied on per os basis, not part of fmt library
 */
extern  int errfmt(Fmt *f);

//-----------------------------------------------------------------------------
// Quoted strings
//-----------------------------------------------------------------------------

/*
 * quoted strings
 */
extern  char*   unquotestrdup(char*);
extern  Rune*   unquoterunestrdup(Rune*);
extern  char*   quotestrdup(char*);
extern  Rune*   quoterunestrdup(Rune*);
extern  int     quotestrfmt(Fmt*);
extern  int     quoterunestrfmt(Fmt*);

extern  void    quotefmtinstall(void);

extern  int     (*doquote)(int);
extern  int     needsrcquote(int);

//*****************************************************************************
// Mathematics
//*****************************************************************************

//-----------------------------------------------------------------------------
// Random
//-----------------------------------------------------------------------------

/*
 * random number
 */
⟨signatures xxxrand functions 121b⟩

// less useful
extern  double  frand(void);
extern  ulong   truerand(void);         /* uses /dev/random */
extern  long    lrand(void);
extern  long    lnrand(long);
extern  ulong   ntruerand(ulong);       /* uses /dev/random */

//-----------------------------------------------------------------------------
// Basic math
//-----------------------------------------------------------------------------

/*
 * math
 */
⟨signatures basic and rounding functions 120a⟩
//extern  long    labs(long);
```

```
extern  double  frexp(double, int*);
extern  double  ldexp(double, int);
extern  double  modf(double, double*);

#define HUGE    3.4028234e38
⟨signatures special float functions 120b⟩

⟨signatures exponential and powers functions 120c⟩

extern  double  hypot(double, double);

//----------------------------------------------------------------------------
// Trigonometry
//----------------------------------------------------------------------------

#define PIO2    1.570796326794896619231e0
#define PI  (PIO2+PIO2)

⟨signatures trigonometric functions 120d⟩

//----------------------------------------------------------------------------
// Conversion
//----------------------------------------------------------------------------

⟨signatures number parsing functions 121a⟩

extern  vlong   atoll(char*);

extern  ulong   strtoul(char*, char**, int);
extern  vlong   strtoll(char*, char**, int);
extern  uvlong  strtoull(char*, char**, int);

//----------------------------------------------------------------------------
// Misc
//----------------------------------------------------------------------------

// internals?
extern  ulong   getfcr(void);
extern  void    setfsr(ulong);
extern  ulong   getfsr(void);
extern  void    setfcr(ulong);

extern  ulong   umuldiv(ulong, ulong, ulong);
extern  long    muldiv(long, long, long);

//******************************************************************************
// Misc
//******************************************************************************

//----------------------------------------------------------------------------
// Time
//----------------------------------------------------------------------------

/*
 * Time-of-day
 */
⟨type Tm 36⟩

⟨signatures time functions 224a⟩
```

```
extern  double  cputime(void);

extern  long    tm2sec(Tm*);

// less useful?
extern  char*   asctime(Tm*);
extern  char*   ctime(long);
extern  long    times(long*);

extern  void    cycles(uvlong*);    /* 64-bit value of the cycle counter if there is one, 0 if there isn't */


//----------------------------------------------------------------------------
// Misc
//----------------------------------------------------------------------------

/*
 * one-of-a-kind
 */

// misc
extern  double  charstod(int(*)(void*), void*);

// modified in place, so type should really be void cleanname(INOUT char*);
extern  char*   cleanname(char*);
extern  int     encodefmt(Fmt*);

extern  int     getfields(char*, char**, int, int, char*);
extern  int     gettokens(char *, char **, int, char *);

extern  int     iounit(fdt);

// ugly redefined by user code? see statusbar.c
extern  void    qsort(void*, long, long, int (*)(void*, void*));

//******************************************************************************
// Debugging and profiling
//******************************************************************************

//----------------------------------------------------------------------------
// Debugging
//----------------------------------------------------------------------------

/*
 | debugging tools
 */

⟨macro assert 235d⟩

extern  void    (*_assert)(char*);

⟨signatures debugging functions 354b⟩

⟨signatures logging functions 354a⟩

// useful for stack trace?
extern  uintptr getcallerpc(void*);

//----------------------------------------------------------------------------
// Profiling
```

```
//----------------------------------------------------------------------------

/*
 *  profiling
 */

⟨type Prof 357b⟩

⟨signatures profiling functions 357a⟩

//******************************************************************************
// Concurrency and communication
//******************************************************************************

⟨type PostnoteKind 258b⟩

//----------------------------------------------------------------------------
// Concurrency
//----------------------------------------------------------------------------

/*
 * atomic
 */
⟨signatures atomic functions 243a⟩

extern int     cas32(u32int*, u32int, u32int);
extern int     casp(void**, void*, void*);
extern int     casl(ulong*, ulong, ulong);

/*
 *  synchronization
 */
⟨type Lock 245a⟩

extern int  _tas(int*);

⟨signatures Lock functions 243b⟩

⟨type QLp 246b⟩

⟨type QLock 246a⟩

⟨signatures QLock functions 243c⟩

extern  void    _qlockinit(void* (*)(void*, void*));    /* called only by the thread library */

⟨type RWLock 248a⟩

⟨signatures RWLock functions 243d⟩

⟨type Rendez 251a⟩

⟨signatures Rendez functions 243e⟩

extern  int     rwakeupall(Rendez*);

// per-process private vars
⟨signatures private vars functions 243f⟩

//----------------------------------------------------------------------------
```

```
// Network (/net wrappers)
//-----------------------------------------------------------------------------

/*
 *  network dialing
 */
#define NETPATHLEN 40
⟨signatures networking functions 342a⟩

extern  void    setnetmtpt(char*, int, char*);
extern  int     hangup(int);
extern  char*   netmkaddr(char*, char*, char*);
extern  int     reject(int, char*, char*);

/*
 *  network services
 */
⟨type NetConnInfo 342b⟩
extern  NetConnInfo*    getnetconninfo(char*, int);
extern  void            freenetconninfo(NetConnInfo*);


//-----------------------------------------------------------------------------
// Crypto
//-----------------------------------------------------------------------------

/*
 *  encryption
 */
extern  int pushssl(int, char*, char*, char*, int*);
extern  int pushtls(int, char*, char*, int, char*, char*);

// encryption
extern  int decrypt(void*, void*, int);
extern  int encrypt(void*, void*, int);
extern  int netcrypt(void*, void*);

extern  int dec64(uchar*, int, char*, int);
extern  int enc64(char*, int, uchar*, int);
extern  int dec32(uchar*, int, char*, int);
extern  int enc32(char*, int, uchar*, int);
extern  int dec16(uchar*, int, char*, int);
extern  int enc16(char*, int, uchar*, int);

//*****************************************************************************
// Syscalls
//*****************************************************************************

/*
 * system calls
 *
 */
#include <syscall.h>


//*****************************************************************************
// ARGXXX
//*****************************************************************************

// getopt like macros
```

```
⟨signature global argv0 43b⟩
⟨macro ARGBEGIN 43d⟩
⟨macro ARGEND 43e⟩
⟨macro ARGF 43f⟩
⟨macro EARGF 43g⟩
⟨macro ARGC 43h⟩
```

A.1.3 include/core/ctype.h

```
⟨include/core/ctype.h 379a⟩≡
  #pragma src "/sys/src/libc/port"
  #pragma lib "libc.a"

  // could be merged in libc.h

  ⟨type Ctype_flag 28a⟩

  extern unsigned char _ctype[];

  ⟨macros isxxx 29a⟩
  ⟨macro _toupper 29c⟩
  ⟨macro _tolower 29e⟩
  ⟨macro toascii 29f⟩
```

A.1.4 include/core/syscall.h

```
⟨include/core/syscall.h 379b⟩≡
  // !You must include libc.h instead of this file!

  //*****************************************************************************
  // Types and constants
  //*****************************************************************************

  ⟨type Open_flag 190c⟩
  ⟨type Namespace_flag 222a⟩
  ⟨type Access_flag 190d⟩
  ⟨type Segattach_flag 261a⟩
  ⟨type Note_flag 258a⟩

  ⟨type Qid_type 191b⟩
  ⟨type Dir_mode 214a⟩

  ⟨type Rfork_flag 230a⟩

  ⟨constant STATMAX 214c⟩
  ⟨constant DIRMAX 214b⟩
  ⟨constant ERRMAX 233b⟩

  ⟨type Seek_cursor 191c⟩

  ⟨type Qid 191a⟩

  ⟨type DirEntry 213b⟩

  ⟨type Waitmsg 255b⟩

  ⟨type IOchunk 193⟩
```

```
//******************************************************************************
// Function prototypes
//******************************************************************************

// syscalls (and small wrappers around syscalls after in each category)

//----------------------------------------------------------------------------
// process
//----------------------------------------------------------------------------
⟨signatures process syscall 44a⟩

⟨signatures process syscall wrapper 40⟩
⟨signatures other process syscall wrapper 230c⟩

//----------------------------------------------------------------------------
// memory
//----------------------------------------------------------------------------
⟨signatures memory syscall 56⟩

/* this is used by sbrk and brk,  it's a really bad idea to redefine it */
extern  char    end[];

⟨signatures memory syscall wrapper 57a⟩

//----------------------------------------------------------------------------
// file
//----------------------------------------------------------------------------
⟨signatures file syscall 191d⟩

⟨signatures file syscall wrapper 190a⟩
⟨signatures file other wrapper 190b⟩
// extern int fdflush(int);

//----------------------------------------------------------------------------
// directory
//----------------------------------------------------------------------------
⟨signatures directory syscall 214e⟩

⟨signatures directory syscall wrapper 213a⟩

//----------------------------------------------------------------------------
// namespace
//----------------------------------------------------------------------------
⟨signatures namespace syscall 222b⟩

//----------------------------------------------------------------------------
// time
//----------------------------------------------------------------------------
⟨signatures time syscall 224b⟩

//----------------------------------------------------------------------------
// IPC
//----------------------------------------------------------------------------
⟨signatures ipc syscall 254b⟩
⟨signatures ipc helpers 254a⟩

// ??
extern  int     postnote(int, int, char *);
extern  int     atnotify(int(*)(void*, char*), int);
```

```
//----------------------------------------------------------------------------
// concurrency
//----------------------------------------------------------------------------
⟨signatures concurrency syscall 244a⟩

//----------------------------------------------------------------------------
// security
//----------------------------------------------------------------------------
⟨signatures security syscall 237a⟩

⟨signatures security syscall wrapper 237b⟩

//----------------------------------------------------------------------------
// error management
//----------------------------------------------------------------------------
⟨signatures error syscall 233d⟩

⟨signatures error syscall wrapper 233a⟩
⟨signatures error other wrapper 233c⟩

//----------------------------------------------------------------------------
// Misc
//----------------------------------------------------------------------------

//???
extern char* sysname(void);
```

A.1.5 include/core/internals/pool.h

⟨include/core/internals/pool.h 381a⟩≡

```
typedef struct Pool Pool;

⟨type Pool 58c⟩

extern void* poolalloc(Pool*, ulong);
extern void poolfree(Pool*, void*);

extern void* poolallocalign(Pool*, ulong, ulong, long, ulong);
extern ulong poolmsize(Pool*, void*);
extern void* poolrealloc(Pool*, void*, ulong);
extern void poolcheck(Pool*);
extern int poolcompact(Pool*);
extern void poolblockcheck(Pool*, void*);

// those globals are initialized in libc but also in the kernel itself
// so that the kernel can reuse the pool allocation code
extern Pool* mainmem;
extern Pool* imagmem;

⟨type Pool_flag 59a⟩
```

A.1.6 include/io/bio.h

⟨macro BGETC 381b⟩≡ (382e)

```
#define BGETC(bp) Bgetc(bp)
```

⟨macro BPUTC 381c⟩≡ (382e)

```
#define BPUTC(bp,c) Bputc(bp,c)
```

⟨*macro* BOFFSET 382a⟩≡ (382e)

```
#define BOFFSET(bp) Boffset(bp)
```

⟨*macro* BLINELEN 382b⟩≡ (382e)

```
#define BLINELEN(bp) Blinelen(bp)
```

⟨*macro* BFILDES 382c⟩≡ (382e)

```
#define BFILDES(bp) Bfildes(bp)
```

⟨*constants Bxxx* 382d⟩+≡ (382e) ◁199a

```
Beof  = -1,
Bbad  = -2,
```

⟨include/io/bio.h 382e⟩≡

```
#pragma src "/sys/src/libbio"
#pragma lib "libbio.a"

typedef struct Biobuf Biobuf;
typedef struct Biobufhdr Biobufhdr;

//typedef struct Biobuf BiobufGen; // for 5c in ocaml
typedef struct Biobufhdr BiobufGen; // for regular 5c

enum
{
    ⟨constants Bxxx 195b⟩
};

⟨struct Biobufhdr 195c⟩
⟨struct Biobuf 195a⟩


/* Dregs, redefined as functions for backwards compatibility */
⟨macro BGETC 381b⟩
⟨macro BPUTC 381c⟩
⟨macro BOFFSET 382a⟩
⟨macro BLINELEN 382b⟩
⟨macro BFILDES 382c⟩

⟨signatures Bxxx functions 194⟩

int Binits(Biobufhdr*, fdt, int, uchar*, int);

int Bflush(BiobufGen*);

vlong Bseek(Biobufhdr*, vlong, int);

int Bputrune(Biobufhdr*, long);
long Bgetrune(BiobufGen*);
int Bungetrune(BiobufGen*);

int Bvprint(Biobufhdr*, char*, va_list);

// Helpers
int Bbuffered(Biobufhdr*);
int Bfildes(Biobufhdr*);

int Bgetd(Biobufhdr*, double*);
vlong Boffset(Biobufhdr*);
char* Brdstr(Biobufhdr*, int, int);
```

A.1.7 include/strings/regexp.h

```
⟨include/strings/regexp.h 383a⟩≡
  #pragma src "/sys/src/libregexp"
  #pragma lib "libregexp.a"

  typedef struct Resub        Resub;
  typedef struct Reclass      Reclass;
  typedef struct Reinst       Reinst;
  typedef struct Reprog       Reprog;

  ⟨struct Resub 106c⟩

  ⟨struct Reclass 106b⟩

  ⟨struct Reinst 106a⟩

  ⟨struct Reprog 105b⟩

  ⟨signatures regxxxx functions 105a⟩
  // other less important
  extern Reprog   *regcomplit(char*);
  extern Reprog   *regcompnl(char*);
  extern void regerror(char*);
  extern int  rregexec(Reprog*, Rune*, Resub*, int);
  extern void rregsub(Rune*, Rune*, int, Resub*, int);
```

A.1.8 include/strings/str.h

```
⟨strings/str.h 383b⟩≡
  #pragma src "/sys/src/libstring"
  #pragma lib "libstring.a"

  // Extensible Strings.

  // This file was originally called string.h but this conflicts with
  // the Posix <string.h> so simpler to rename <str.h> (which anyway fits
  // well with the 3-letters utf.h, bio.h, fmt.h, etc.).

  ⟨struct String 91⟩
  typedef struct String String;

  ⟨macro s_clone 92c⟩
  ⟨macro s_to_c 92a⟩
  ⟨macro s_len 92b⟩

  extern String* s_append(String*, char*);
  extern String* s_array(char*, int);
  extern String* s_copy(char*);
  extern void s_free(String*);
  extern String* s_incref(String*);
  extern String* s_memappend(String*, char*, int);
  extern String* s_nappend(String*, char*, int);
  extern String* s_new(void);
  extern String* s_newalloc(int);
  extern String* s_parse(String*, String*);
  extern String* s_reset(String*);
  extern String* s_restart(String*);
  extern void s_terminate(String*);
```

```
extern void s_tolower(String*);
extern void s_putc(String*, int);
extern String* s_unique(String*);
extern String* s_grow(String*, int);

// Biobuf functions visible when including bio.h before str.h
#ifdef BGETC
extern int s_read(Biobuf*, String*, int);
extern char *s_read_line(Biobuf*, String*);
extern char *s_getline(Biobuf*, String*);
typedef struct Sinstack Sinstack;
#pragma incomplete Sinstack
extern char *s_rdinstack(Sinstack*, String*);
extern Sinstack *s_allocinstack(char*);
extern void s_freeinstack(Sinstack*);
#endif /* BGETC */
```

A.1.9 include/compress/flate.h

⟨compress/flate.h 384a⟩≡

```
#pragma lib "libflate.a"
#pragma src "/sys/src/libflate"

⟨enum flate_error 159⟩

// compression (make smaller)
⟨signature deflate 158a⟩
int deflateinit(void);

// decompression (make bigger)
int inflateinit(void);
⟨signature inflate 158b⟩

int inflateblock(uchar *dst, int dsize, uchar *src, int ssize);
int deflateblock(uchar *dst, int dsize, uchar *src, int ssize, int level, int debug);

int deflatezlib(void *wr, int (*w)(void*, void*, int), void *rr, int (*r)(void*, void*, int), int level, int de
int inflatezlib(void *wr, int (*w)(void*, void*, int), void *getr, int (*get)(void*));

int inflatezlibblock(uchar *dst, int dsize, uchar *src, int ssize);
int deflatezlibblock(uchar *dst, int dsize, uchar *src, int ssize, int level, int debug);

char *flateerr(int err);

ulong *mkcrctab(ulong);
ulong blockcrc(ulong *tab, ulong crc, void *buf, int n);

ulong adler32(ulong adler, void *buf, int n);
```

A.1.10 include/ipc/thread.h

⟨include/ipc/thread.h 384b⟩≡

```
#pragma src "/sys/src/libthread"
#pragma lib "libthread.a"

#pragma varargck    argpos  chanprint   2

typedef struct Alt  Alt;
```

```
typedef struct Channel  Channel;
typedef struct Ref  Ref;
/* slave I/O processes */
typedef struct Ioproc Ioproc;
#pragma incomplete Ioproc

⟨struct Channel 288e⟩

⟨enum chanop 299c⟩
typedef enum chanop ChanOp;

⟨struct Alt 300a⟩

⟨struct Ref 287a⟩

long    decref(Ref *r);         /* returns 0 iff value is now zero */
void    incref(Ref *r);

Channel* chancreate(int elemsize, int bufsize);
int chanclose(Channel*);
int chanclosing(Channel *c);
int chaninit(Channel *c, int elemsize, int elemcnt);
void chanfree(Channel *c);
int chanprint(Channel *, char *, ...);

// blocking API
int recv(Channel *c, void *v);
void* recvp(Channel *c);
ulong recvul(Channel *c);
int send(Channel *c, void *v);
int sendp(Channel *c, void *v);
int sendul(Channel *c, ulong v);

// non blocking API
int nbrecv(Channel *c, void *v);
void* nbrecvp(Channel *c);
ulong nbrecvul(Channel *c);
int nbsend(Channel *c, void *v);
int nbsendp(Channel *c, void *v);
int nbsendul(Channel *c, ulong v);

// select
int alt(Alt alts[]);

// process
int proccreate(void (*f)(void *arg), void *arg, uint stacksize);
int procrfork(void (*f)(void *arg), void *arg, uint stacksize, int flag);
void**  procdata(void);
void    procexec(Channel *, char *, char *[]);
void    procexecl(Channel *, char *, ...);

// threads
int threadcreate(void (*f)(void *arg), void *arg, uint stacksize);
void**  threaddata(void);
void    threadexits(char *);
void    threadexitsall(char *);
void    threadmain(int argc, char *argv[]);
int threadid(void);

int threadgetgrp(void); /* return thread group of current thread */
```

```
int threadsetgrp(int);      /* set thread group, return old */
char*   threadgetname(void);
void    threadsetname(char *fmt, ...);
void    threadint(int);     /* interrupt thread */
void    threadintgrp(int);  /* interrupt threads in grp */
void    threadkill(int);    /* kill thread */
void    threadkillgrp(int); /* kill threads in group */
void    threadnonotes(void);
int threadnotify(int (*f)(void*, char*), int in);
int threadpid(int);
Channel*threadwaitchan(void);

// Io proc
Ioproc* ioproc(void);
void    closeioproc(Ioproc*);
void    iointerrupt(Ioproc*);

int ioopen(Ioproc*, char*, int);
int ioclose(Ioproc*, int);
long    ioread(Ioproc*, int, void*, long);
long    ioreadn(Ioproc*, int, void*, long);
long    iowrite(Ioproc*, int, void*, long);
int iosleep(Ioproc*, long);
int iodial(Ioproc*, char*, char*, char*, int*);

long    iocall(Ioproc*, long (*)(va_list*), ...);
void    ioret(Ioproc*, int);

// misc
void    needstack(int);
int tprivalloc(void);
void    tprivfree(int);
void    **tprivaddr(int);
void    yield(void);

extern  int mainstacksize;
```

A.1.11 include/ipc/fcall.h

⟨include/ipc/fcall.h 386⟩≡

```
#pragma src "/sys/src/libc/9sys"
#pragma lib "libc.a"

#define VERSION9P   "9P2000"

⟨constant MAXWELEM 263a⟩

typedef struct  Fcall Fcall;

⟨type Fcall 261b⟩

⟨macros GBITxxx 266a⟩
⟨macros PBITxxx 266b⟩
⟨macros BITxxx 266c⟩

⟨constant QIDSZ 279b⟩

⟨constant STATFIXLEN 279c⟩
```

```
#define NOTAG       (ushort)~0U /* Dummy tag */
#define NOFID       (u32int)~0U /* Dummy fid */

⟨constant IOHDRSZ 266d⟩

⟨type FcallType 262⟩


error0    convM2S(uchar*, uint, Fcall*);
error0    convS2M(Fcall*, uchar*, uint);
uint    sizeS2M(Fcall*);

int     statcheck(uchar *abuf, uint nbuf);
uint    convM2D(uchar*, uint, Dir*, char*);
uint    convD2M(Dir*, uchar*, uint);
uint    sizeD2M(Dir*);

// dumpers
int fcallfmt(Fmt*);
int dirfmt(Fmt*);
int dirmodefmt(Fmt*);

int read9pmsg(fdt, void*, uint);

#pragma varargck    type    "F" Fcall*
#pragma varargck    type    "M" ulong
#pragma varargck    type    "D" Dir*
```

A.2 libc/port

⟨*libc includes* 387a⟩≡ (586g 584 569a 568 567 566 565 563 562 561 560 558a 557 538 537c 536 535 534 533c 532b 531 530 529 528c 523 52

```
#include    <u.h>
#include    <libc.h>
```

libc/port/_assert.c

⟨libc/port/_assert.c 387b⟩≡

```
⟨libc includes 387a⟩
⟨global __assert 235g⟩
⟨function default_assert 235f⟩
⟨global _assert 235e⟩
```

libc/port/abs.c

⟨libc/port/abs.c 387c⟩≡

```
⟨libc includes 387a⟩
⟨function abs 122a⟩
```

libc/port/asin.c

⟨libc/port/asin.c 387d⟩≡

```
/*
 * asin(arg) and acos(arg) return the arcsin, arccos,
 * respectively of their arguments.
```

```
 *
 * Arctan is called after appropriate range reduction.
 */
⟨libc includes 387a⟩
⟨function asin 144a⟩
⟨function acos 144b⟩
```

libc/port/atan.c

```
⟨libc/port/atan.c 388a⟩≡
/*
    floating-point arctangent

    atan returns the value of the arctangent of its
    argument in the range [-pi/2,pi/2].

    atan2 returns the arctangent of arg1/arg2
    in the range [-pi,pi].

    there are no error returns.

    coefficients are #5077 from Hart & Cheney. (19.56D)
*/
⟨libc includes 387a⟩
⟨constant sq2p1 145a⟩
⟨constant sq2m1 145b⟩
⟨constant p4 145c⟩
⟨constant p3 145d⟩
⟨constant p2 145e⟩
⟨constant p1 145f⟩
⟨constant p0 145g⟩
⟨constant q4 145h⟩
⟨constant q3 145i⟩
⟨constant q2 145j⟩
⟨constant q1 145k⟩
⟨constant q0 145l⟩

⟨function xatan 146a⟩
⟨function satan 146b⟩

⟨function atan 144c⟩
```

libc/port/atan2.c

```
⟨libc/port/atan2.c 388b⟩≡
⟨libc includes 387a⟩
⟨function atan2 146c⟩
```

libc/port/atexit.c

```
⟨libc/port/atexit.c 388c⟩≡
⟨libc includes 387a⟩
⟨constant NEXIT 44c⟩

typedef struct Onex Onex;
⟨struct Onex 45a⟩
```

```
⟨global onexlock 45c⟩
⟨global onex 44b⟩

⟨function atexit 45d⟩

#pragma profile off
⟨function exits 45b⟩
#pragma profile on
```
Uses Onex 388c.

libc/port/atnotify.c

```
⟨libc/port/atnotify.c 389a⟩≡
  ⟨libc includes 387a⟩
  ⟨constant NFN (port/atnotify.c) 260a⟩
  ⟨global onnot 260b⟩
  ⟨global onnotlock 260c⟩

  ⟨function notifier 260d⟩

  ⟨function atnotify 259⟩
```

libc/port/atof.c

```
⟨libc/port/atof.c 389b⟩≡
  ⟨libc includes 387a⟩
  ⟨function atof 127c⟩
```

libc/port/atol.c

```
⟨libc/port/atol.c 389c⟩≡
  ⟨libc includes 387a⟩
  ⟨function atol 123c⟩
  ⟨function atoi 123b⟩
```

libc/port/atoll.c

```
⟨function atoll 389d⟩≡                                        (389e)
  vlong
  atoll(char *s)
  {
      return strtoll(s, nil, 0);
  }
```
Uses strtoll() 447c.

```
⟨libc/port/atoll.c 389e⟩≡
  ⟨libc includes 387a⟩
  ⟨function atoll 389d⟩
```

libc/port/charstod.c

⟨*constant* ADVANCE 390a⟩≡ (391a)

```
/*
 * Reads a floating-point number by interpreting successive characters
 * returned by (*f)(vp).  The last call it makes to f terminates the
 * scan, so is not a character in the number.  It may therefore be
 * necessary to back up the input stream up one byte after calling charstod.
 */

#define ADVANCE do{*s++ = c; if(s>=e) return NaN(); c = (*f)(vp);}while(0)
```

⟨*function* charstod 390b⟩≡ (391a)

```
double
charstod(int(*f)(void*), void *vp)
{
    char str[400], *s, *e, *start;
    int c;

    s = str;
    e = str + sizeof str - 1;
    c = (*f)(vp);
    while(c == ' ' || c == '\t')
        c = (*f)(vp);
    if(c == '-' || c == '+'){
        ADVANCE;
    }
    start = s;
    while(c >= '0' && c <= '9'){
        ADVANCE;
    }
    if(c == '.'){
        ADVANCE;
        while(c >= '0' && c <= '9'){
            ADVANCE;
        }
    }
    if(s > start && (c == 'e' || c == 'E')){
        ADVANCE;
        if(c == '-' || c == '+'){
            ADVANCE;
        }
        while(c >= '0' && c <= '9'){
            ADVANCE;
        }
    }else if(s == start && (c == 'i' || c == 'I')){
        ADVANCE;
        if(c != 'n' && c != 'N')
            return NaN();
        ADVANCE;
        if(c != 'f' && c != 'F')
            return NaN();
        ADVANCE;
        if(c != 'i' && c != 'I')
            return NaN();
        ADVANCE;
        if(c != 'n' && c != 'N')
            return NaN();
        ADVANCE;
        if(c != 'i' && c != 'I')
```

```
            return NaN();
        ADVANCE;
        if(c != 't' && c != 'T')
            return NaN();
        ADVANCE;
        if(c != 'y' && c != 'Y')
            return NaN();
        ADVANCE;  /* so caller can back up uniformly */
        USED(c);
    }else if(s == str && (c == 'n' || c == 'N')){
        ADVANCE;
        if(c != 'a' && c != 'A')
            return NaN();
        ADVANCE;
        if(c != 'n' && c != 'N')
            return NaN();
        ADVANCE;  /* so caller can back up uniformly */
        USED(c);
    }
    *s = 0;
    return strtod(str, &s);
}
```

Uses `ADVANCE-195` 390a, `NaN()` 126a, and `strtod()` 128.

⟨`libc/port/charstod.c` 391a⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*constant* `ADVANCE` 390a⟩

  ⟨*function* `charstod` 390b⟩

libc/port/cistrcmp.c

⟨*function* `cistrcmp` 391b⟩≡ (391c)

```
int
cistrcmp(char *s1, char *s2)
{
    int c1, c2;

    while(*s1){
        c1 = *(uchar*)s1++;
        c2 = *(uchar*)s2++;

        if(c1 == c2)
            continue;

        if(c1 >= 'A' && c1 <= 'Z')
            c1 -= 'A' - 'a';

        if(c2 >= 'A' && c2 <= 'Z')
            c2 -= 'A' - 'a';

        if(c1 != c2)
            return c1 - c2;
    }
    return -*s2;
}
```

⟨`libc/port/cistrcmp.c` 391c⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* `cistrcmp` 391b⟩

libc/port/cistrncmp.c

⟨*function* `cistrncmp` 392a⟩≡ (392b)

```
int
cistrncmp(char *s1, char *s2, int n)
{
    int c1, c2;

    while(*s1 && n-- > 0){
        c1 = *(uchar*)s1++;
        c2 = *(uchar*)s2++;

        if(c1 == c2)
            continue;

        if(c1 >= 'A' && c1 <= 'Z')
            c1 -= 'A' - 'a';

        if(c2 >= 'A' && c2 <= 'Z')
            c2 -= 'A' - 'a';

        if(c1 != c2)
            return c1 - c2;
    }
    if(n <= 0)
        return 0;
    return -*s2;
}
```

⟨`libc/port/cistrncmp.c` 392b⟩≡

```
⟨libc includes 387a⟩
⟨function cistrncmp 392a⟩
```

libc/port/cistrstr.c

⟨*function* `cistrstr` 392c⟩≡ (393a)

```
char*
cistrstr(char *s, char *sub)
{
    int c, csub, n;

    csub = *sub;
    if(csub == '\0')
        return s;
    if(csub >= 'A' && csub <= 'Z')
        csub -= 'A' - 'a';
    sub++;
    n = strlen(sub);
    for(; c = *s; s++){
        if(c >= 'A' && c <= 'Z')
            c -= 'A' - 'a';
        if(c == csub && cistrncmp(s+1, sub, n) == 0)
            return s;
    }
    return nil;
}
```

Uses `cistrncmp()` 392a and `strlen()` 80a.

⟨libc/port/cistrstr.c 393a⟩≡
```
  ⟨libc includes 387a⟩
  ⟨function cistrstr 392c⟩
```

libc/port/cleanname.c

⟨libc/port/cleanname.c 393b⟩≡
```
  ⟨libc includes 387a⟩
  ⟨macro SEP 218a⟩
  ⟨function cleanname 218b⟩
```

libc/port/crypt.c

⟨libc/port/crypt.c 393c⟩≡
```
  /*
   *  Data Encryption Standard
   *  D.P.Mitchell  83/06/08.
   *
   *  block_cipher(key, block, decrypting)
   *
   *  these routines use the non-standard 7 byte format
   *  for DES keys.
   */
  ⟨libc includes 387a⟩
  #include <auth.h>
  #include <libsec.h>

  ⟨function encrypt 238a⟩
  ⟨function decrypt 238b⟩
```

libc/port/ctype.c

⟨libc/port/ctype.c 393d⟩≡
```
  ⟨libc includes 387a⟩
  #include <ctype.h>

  ⟨global _ctype 28b⟩
```

libc/port/encodefmt.c

⟨function encodefmt 393e⟩≡ (395a)
```
  int
  encodefmt(Fmt *f)
  {
      char *out;
      char *buf;
      int len;
      int ilen;
      int rv;
      uchar *b;
      char *p;
      char obuf[64];  // rsc optimization

      if(!(f->flags&FmtPrec) || f->prec < 1)
          goto error;
```

```
    b = va_arg(f->args, uchar*);
    if(b == 0)
        return fmtstrcpy(f, "<nil>");

    ilen = f->prec;
    f->prec = 0;
    f->flags &= ~FmtPrec;
    switch(f->r){
    case '<':
        len = (8*ilen+4)/5 + 3;
        break;
    case '[':
        len = (8*ilen+5)/6 + 4;
        break;
    case 'H':
        len = 2*ilen + 1;
        break;
    default:
        goto error;
    }

    if(len > sizeof(obuf)){
        buf = malloc(len);
        if(buf == nil)
            goto error;
    } else
        buf = obuf;

    // convert
    out = buf;
    switch(f->r){
    case '<':
        rv = enc32(out, len, b, ilen);
        break;
    case '[':
        rv = enc64(out, len, b, ilen);
        break;
    case 'H':
        rv = enc16(out, len, b, ilen);
        if(rv >= 0 && (f->flags & FmtLong))
            for(p = buf; *p; p++)
                *p = tolower(*p);
        break;
    default:
        rv = -1;
        break;
    }
    if(rv < 0)
        goto error;

    fmtstrcpy(f, buf);
    if(buf != obuf)
        free(buf);
    return 0;

error:
    return fmtstrcpy(f, "<encodefmt>");
}
```
Uses enc16() 456a, enc32() 457, enc64() 459b, fmtstrcpy() 505a, free() 63i, malloc() 63g, and tolower() 29d.

⟨libc/port/encodefmt.c 395a⟩≡
```
⟨libc includes 387a⟩
#include <ctype.h>

⟨function encodefmt 393e⟩
```

libc/port/execl.c

⟨libc/port/execl.c 395b⟩≡
```
⟨libc includes 387a⟩
⟨function execl 231b⟩
```

libc/port/exp.c

⟨libc/port/exp.c 395c⟩≡
```
/*
    exp returns the exponential function of its
    floating-point argument.

    The coefficients are #1069 from Hart and Cheney. (22.35D)
*/

⟨libc includes 387a⟩
⟨constant p0 (port/exp.c) 149d⟩
⟨constant p1 (port/exp.c) 149e⟩
⟨constant p2 (port/exp.c) 149f⟩
⟨constant q0 (port/exp.c) 150a⟩
⟨constant q1 (port/exp.c) 150b⟩
⟨constant q2 (port/exp.c) 150c⟩
⟨constant log2e 149c⟩
⟨constant sqrt2 150d⟩
⟨constant maxf 149b⟩

⟨function exp 149a⟩
```

libc/port/fabs.c

⟨libc/port/fabs.c 395d⟩≡
```
⟨libc includes 387a⟩
⟨function fabs 122b⟩
```

libc/port/floor.c

⟨libc/port/floor.c 395e⟩≡
```
⟨libc includes 387a⟩
⟨function floor 122c⟩
⟨function ceil 123a⟩
```

libc/port/fmod.c

⟨libc/port/fmod.c 395f⟩≡
```
⟨libc includes 387a⟩
⟨function fmod 138⟩
```

libc/port/frand.c

⟨*constant* MASK 396a⟩≡ (396d)

```
#define MASK    0x7fffffffL
```

⟨*constant* NORM 396b⟩≡ (396d)

```
#define NORM    (1.0/(1.0+MASK))
```

⟨*function* frand 396c⟩≡ (396d)

```
double
frand(void)
{
    double x;

    do {
        x = lrand() * NORM;
        x = (x + lrand()) * NORM;
    } while(x >= 1);
    return x;
}
```

Uses NORM-191 396b and lrand() 153c.

⟨libc/port/frand.c 396d⟩≡

```
⟨libc includes 387a⟩
⟨constant MASK 396a⟩
⟨constant NORM 396b⟩

⟨function frand 396c⟩
```

libc/port/frexp.c

⟨libc/port/frexp.c 396e⟩≡

```
⟨libc includes 387a⟩
⟨constant MASK (port/frexp.c) 150e⟩
⟨constant SHIFT 150f⟩
⟨constant BIAS 150g⟩
⟨constant SIG 150h⟩

⟨function frexp 151a⟩

⟨function ldexp 150i⟩

⟨function modf 137b⟩
```

libc/port/getcallerpc.c

⟨*function* getcallerpc 396f⟩≡ (396g)

```
uintptr
getcallerpc(void*)
{
    return 0;
}
```

⟨libc/port/getcallerpc.c 396g⟩≡

```
⟨libc includes 387a⟩
⟨function getcallerpc 396f⟩
```

libc/port/getfields.c

⟨libc/port/getfields.c 397a⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* getfields 255a⟩

libc/port/getuser.c

⟨libc/port/getuser.c 397b⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* getuser 237c⟩

libc/port/hangup.c

⟨*function* hangup 397c⟩≡ (397d)
```
/*
 *  force a connection to hangup
 */
int
hangup(int ctl)
{
    return write(ctl, "hangup", sizeof("hangup")-1) != sizeof("hangup")-1;
}
```
Uses write() 192b.

⟨libc/port/hangup.c 397d⟩≡
  ⟨*libc includes* 387a⟩
```
#include <ctype.h>
```
  ⟨*function* hangup 397c⟩

libc/port/hypot.c

⟨*function* hypot 397e⟩≡ (398a)
```
double
hypot(double p, double q)
{
    double r, s, pfac;

    if(p < 0)
        p = -p;
    if(q < 0)
        q = -q;
    if(p < q) {
        r = p;
        p = q;
        q = r;
    }
    if(p == 0)
        return 0;
    pfac = p;
    r = q = q/p;
    p = 1;
    for(;;) {
        r *= r;
        s = r+4;
        if(s == 4)
```

```
            return p*pfac;
        r /= s;
        p += 2*r*p;
        q *= r;
        r = q/p;
    }
}
```

⟨libc/port/hypot.c 398a⟩≡

```
/*
 * hypot -- sqrt(p*p+q*q), but overflows only if the result does.
 * See Cleve Moler and Donald Morrison,
 * ``Replacing Square Roots by Pythagorean Sums,''
 * IBM Journal of Research and Development,
 * Vol. 27, Number 6, pp. 577-581, Nov. 1983
 */
```

⟨*libc includes* 387a⟩
⟨*function* hypot 397e⟩

libc/port/lnrand.c

⟨*constant MASK (port/lnrand.c)* 398b⟩≡ (398d)

```
#define MASK    0x7fffffffL
```

⟨*function* lnrand 398c⟩≡ (398d)

```
long
lnrand(long n)
{
    long slop, v;

    if(n < 0)
        return n;
    slop = MASK % n;
    do
        v = lrand();
    while(v <= slop);
    return v % n;
}
```

Uses MASK-56 398b and lrand() 153c.

⟨libc/port/lnrand.c 398d⟩≡
⟨*libc includes* 387a⟩
⟨*constant MASK (port/lnrand.c)* 398b⟩

⟨*function* lnrand 398c⟩

libc/port/lock.c

⟨libc/port/lock.c 398e⟩≡
⟨*libc includes* 387a⟩
⟨*function* lock 245b⟩
⟨*function* unlock 245c⟩

⟨*function* canlock 245d⟩

libc/port/log.c

⟨libc/port/log.c 399a⟩≡
```
/*
    log returns the natural logarithm of its floating
    point argument.

    The coefficients are #2705 from Hart & Cheney. (19.38D)

    It calls frexp.
*/
```

⟨*libc includes* 387a⟩
⟨*constant* log2 152a⟩
⟨*constant* ln10o1 152b⟩
⟨*constant* sqrto2 152c⟩
⟨*constant p0 (port/log.c)* 152d⟩
⟨*constant p1 (port/log.c)* 152e⟩
⟨*constant p2 (port/log.c)* 152f⟩
⟨*constant p3 (port/log.c)* 152g⟩
⟨*constant q0 (port/log.c)* 152h⟩
⟨*constant q1 (port/log.c)* 152i⟩
⟨*constant q2 (port/log.c)* 152j⟩

⟨*function* log 151b⟩

⟨*function* log10 153a⟩

libc/port/lrand.c

⟨*constant* LEN 399b⟩≡ (400a)
```
#define LEN 607
```

⟨*constant* TAP 399c⟩≡ (400a)
```
#define TAP 273
```

⟨*constant MASK (port/lrand.c)* 399d⟩≡ (400a)
```
#define MASK    0x7fffffffL
```

⟨*constant* A 399e⟩≡ (400a)
```
#define A   48271
```

⟨*constant* M 399f⟩≡ (400a)
```
#define M   2147483647
```

⟨*constant* Q 399g⟩≡ (400a)
```
#define Q   44488
```

⟨*constant* R 399h⟩≡ (400a)
```
#define R   3399
```

⟨*constant NORM (port/lrand.c)* 399i⟩≡ (400a)
```
#define NORM    (1.0/(1.0+MASK))
```

⟨libc/port/lrand.c 400a⟩≡
```
⟨libc includes 387a⟩
/*
 *  algorithm by
 *  D. P. Mitchell & J. A. Reeds
 */

⟨constant LEN 399b⟩
⟨constant TAP 399c⟩
⟨constant MASK (port/lrand.c) 399d⟩
⟨constant A 399e⟩
⟨constant M 399f⟩
⟨constant Q 399g⟩
⟨constant R 399h⟩
⟨constant NORM (port/lrand.c) 399i⟩

⟨global rng_vec 154b⟩
⟨global rng_tap 154c⟩
⟨global rng_feed 154d⟩
⟨global lk 154a⟩

⟨function isrand 154f⟩

⟨function srand 154e⟩

⟨function lrand 153c⟩
```

libc/port/malloc.c

⟨function sbrkalloc 400b⟩≡ (403d)
```
/*
 * we do minimal bookkeeping so we can tell pool
 * whether two blocks are adjacent and thus mergeable.
 */
static void*
sbrkalloc(ulong n)
{
    ulong *x;

    n += 2*sizeof(ulong);   /* two longs for us */
    x = sbrk(n);
    if(x == (void*)-1)
        return nil;
    x[0] = (n+7)&~7;    /* sbrk rounds size up to mult. of 8 */
    x[1] = 0xDeadBeef;
    return x+2;
}
```
Uses sbrk() 58a.

⟨function sbrkmerge 400c⟩≡ (403d)
```
static int
sbrkmerge(void *x, void *y)
{
    ulong *lx, *ly;

    lx = x;
    if(lx[-1] != 0xDeadBeef)
        abort();
```

```
    if((uchar*)lx+lx[-2] == (uchar*)y) {
        ly = y;
        lx[-2] += ly[-2];
        return 1;
    }
    return 0;
}
```

Uses `abort()` 356d.

⟨*function* `plock` 401a⟩≡ (403d)

```
static void
plock(Pool *p)
{
    Private *pv;
    pv = p->private;
    lock(&pv->lk);
    if(pv->pid != 0)
        abort();
    pv->pid = _tos->pid;
}
```

Uses `abort()` 356d and `lock()` 245b.

⟨*function* `punlock` 401b⟩≡ (403d)

```
static void
punlock(Pool *p)
{
    Private *pv;
    pv = p->private;
    if(pv->pid != _tos->pid)
        abort();
    pv->pid = 0;
    unlock(&pv->lk);
}
```

Uses `abort()` 356d and `unlock()` 245c.

⟨*function* `checkenv` 401c⟩≡ (403d)

```
static int
checkenv(void)
{
    int n, fd;
    char buf[20];
    fd = open("/env/MALLOCFD", OREAD);
    if(fd < 0)
        return -1;
    if((n = read(fd, buf, sizeof buf)) < 0) {
        close(fd);
        return -1;
    }
    if(n >= sizeof buf)
        n = sizeof(buf)-1;
    buf[n] = 0;
    n = atoi(buf);
    if(n == 0)
        n = -1;
    return n;
}
```

Uses `atoi()` 123b, `close()`, `open()`, and `read()` 192a.

⟨*function* pprint 402a⟩≡ (403d)
```
static void
pprint(Pool *p, char *fmt, ...)
{
    va_list v;
    Private *pv;

    pv = p->private;
    if(pv->printfd == 0)
        pv->printfd = checkenv();

    if(pv->printfd <= 0)
        pv->printfd = 2;

    va_start(v, fmt);
    vfprint(pv->printfd, fmt, v);
    va_end(v);
}
```
Uses `checkenv()` 401c and `vfprint()` 73.

⟨*global* panicbuf 402b⟩≡ (403d)
```
static char panicbuf[256];
```

⟨*function* ppanic 402c⟩≡ (403d)
```
static void
ppanic(Pool *p, char *fmt, ...)
{
    va_list v;
    int n;
    char *msg;
    Private *pv;

    pv = p->private;
    assert(canlock(&pv->lk)==0);

    if(pv->printfd == 0)
        pv->printfd = checkenv();
    if(pv->printfd <= 0)
        pv->printfd = 2;

    msg = panicbuf;
    va_start(v, fmt);
    n = vseprint(msg, msg+sizeof panicbuf, fmt, v) - msg;
    write(2, "panic: ", 7);
    write(2, msg, n);
    write(2, "\n", 1);
    if(pv->printfd != 2){
        write(pv->printfd, "panic: ", 7);
        write(pv->printfd, msg, n);
        write(pv->printfd, "\n", 1);
    }
    va_end(v);
//  unlock(&pv->lk);
    abort();
}
```
Uses `abort()` 356d, `canlock()` 245d, `checkenv()` 401c, `panicbuf-197` 402b, `vseprint()` 536b, and `write()` 192b.

⟨*enum* _anon_ 402d⟩≡ (403d)
```
/* tracing */
enum {
```

```
    ⟨constant Npadlong 63h⟩
    MallocOffset = 0,
    ReallocOffset = 1
};
```

⟨*function* mallocalign 403a⟩≡ (403d)

```
void*
mallocalign(ulong size, ulong align, long offset, ulong span)
{
    void *v;

    v = poolallocalign(mainmem, size+Npadlong*sizeof(ulong), align, offset-Npadlong*sizeof(ulong), span);
    if(Npadlong && v != nil){
        v = (ulong*)v+Npadlong;
        setmalloctag(v, getcallerpc(&size));
        setrealloctag(v, 0);
    }
    return v;
}
```

Uses `Npadlong-198` 63h, `getcallerpc()` 396f, `mainmem` 59e, `setmalloctag()` 69a, and `setrealloctag()` 69b.

⟨*function* calloc 403b⟩≡ (403d)

```
void*
calloc(ulong n, ulong szelem)
{
    void *v;
    if(v = mallocz(n*szelem, 1))
        setmalloctag(v, getcallerpc(&n));
    return v;
}
```

Uses `getcallerpc()` 396f, `mallocz()` 67a, and `setmalloctag()` 69a.

⟨*function* malloctopoolblock 403c⟩≡ (403d)

```
void*
malloctopoolblock(void *v)
{
    if(v == nil)
        return nil;

    return &((ulong*)v)[-Npadlong];
}
```

Uses `Npadlong-198` 63h.

⟨libc/port/malloc.c 403d⟩≡

```
⟨libc includes 387a⟩
#include <pool.h>
#include <tos.h>

static void*    sbrkalloc(ulong);
static int      sbrkmerge(void*, void*);
static void     plock(Pool*);
static void     punlock(Pool*);
static void     pprint(Pool*, char*, ...);
static void     ppanic(Pool*, char*, ...);

typedef struct Private Private;
⟨struct Private 59b⟩

⟨global sbrkmempriv 59c⟩
```

```
⟨global sbrkmem 59d⟩
⟨global mainmem 59e⟩
⟨global imagmem 59f⟩

⟨function sbrkalloc 400b⟩

⟨function sbrkmerge 400c⟩

⟨function plock 401a⟩

⟨function punlock 401b⟩

⟨function checkenv 401c⟩

⟨function pprint 402a⟩

⟨global panicbuf 402b⟩
⟨function ppanic 402c⟩

/* - 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. - */

/*
 * 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.
 */

/*  non tracing
 *
enum {
    Npadlong    = 0,
    MallocOffset = 0,
    ReallocOffset = 0,
};
 *
 */

⟨enum _anon_ 402d⟩

⟨function malloc 63g⟩

⟨function mallocz 67a⟩

⟨function mallocalign 403a⟩

⟨function free 63i⟩

⟨function realloc 67b⟩

⟨function msize 68a⟩
```

⟨*function* `calloc` 403b⟩

⟨*function* `setmalloctag` 69a⟩

⟨*function* `setrealloctag` 69b⟩

⟨*function* `getmalloctag` 69c⟩

⟨*function* `getrealloctag` 70⟩

⟨*function* `malloctopoolblock` 403c⟩

Uses `Private` 59b.

libc/port/memccpy.c

⟨`libc/port/memccpy.c` 405a⟩≡
⟨*libc includes* 387a⟩
⟨*function* `memccpy` 81d⟩

libc/port/memchr.c

⟨`libc/port/memchr.c` 405b⟩≡
⟨*libc includes* 387a⟩
⟨*function* `memchr` 54⟩

libc/port/memcmp.c

⟨`libc/port/memcmp.c` 405c⟩≡
⟨*libc includes* 387a⟩
⟨*function* `memcmp` 53⟩

libc/port/memmove.c

⟨`libc/port/memmove.c` 405d⟩≡
⟨*libc includes* 387a⟩
⟨*function* `memmove` 48⟩
⟨*function* `memcpy` 49⟩

libc/port/memset.c

⟨`libc/port/memset.c` 405e⟩≡
⟨*libc includes* 387a⟩
⟨*function* `memset` 46b⟩

libc/port/mktemp.c

⟨`libc/port/mktemp.c` 405f⟩≡
⟨*libc includes* 387a⟩
⟨*function* `mktemp` 217b⟩

libc/port/muldiv.c

```
⟨libc/port/muldiv.c 406a⟩≡
  ⟨libc includes 387a⟩
  ⟨function umuldiv 155b⟩
  ⟨function muldiv 155c⟩
```

libc/port/nan.c

```
⟨libc/port/nan.c 406b⟩≡
  ⟨libc includes 387a⟩
  ⟨constant NANEXP 126b⟩
  ⟨constant NANMASK 126c⟩
  ⟨constant NANSIGN 126e⟩

  ⟨function NaN 126a⟩
  ⟨function isNaN 126d⟩

  ⟨function Inf 126f⟩
  ⟨function isInf 127a⟩
```

libc/port/needsrcquote.c

```
⟨function needsrcquote 406c⟩≡                                        (406d)
  int
  needsrcquote(int c)
  {
      if(c <= ' ')
          return 1;
      if(utfrune("`^#*[]=|\\?${}()'<>&;", c))
          return 1;
      return 0;
  }
```

Uses utfrune() 89b.

```
⟨libc/port/needsrcquote.c 406d⟩≡
  ⟨libc includes 387a⟩
  ⟨function needsrcquote 406c⟩
```

libc/port/netcrypt.c

```
⟨libc/port/netcrypt.c 406e⟩≡
  ⟨libc includes 387a⟩
  #include <auth.h>

  ⟨function netcrypt 239a⟩
```

libc/port/netmkaddr.c

```
⟨libc/port/netmkaddr.c 406f⟩≡
  ⟨libc includes 387a⟩
  #include <ctype.h>

  ⟨function netmkaddr 343⟩
```

libc/port/nrand.c

⟨*constant MASK (port/nrand.c)* 407a⟩≡ (407c)
```
#define MASK    0x7fffffffL
```

⟨*function* `nrand` 407b⟩≡ (407c)
```
int
nrand(int n)
{
    long slop, v;

    if(n < 0)
        return n;
    if(n == 1)
        return 0;
    /* and if n == 0, you deserve what you get */
    slop = MASK % n;
    do
        v = lrand();
    while(v <= slop);
    return v % n;
}
```
Uses `MASK-58` 407a and `lrand()` 153c.

⟨`libc/port/nrand.c` 407c⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*constant MASK (port/nrand.c)* 407a⟩

  ⟨*function* `nrand` 407b⟩

libc/port/ntruerand.c

⟨*function* `ntruerand` 407d⟩≡ (407e)
```
ulong
ntruerand(ulong n)
{
    ulong m, r;

    /*
     * set m to the one less than the maximum multiple of n <= 2^32,
     * so we want a random number <= m.
     */
    if(n > (1UL<<31))
        m = n-1;
    else
        /* 2^32 - 2^32%n - 1 = (2^32 - 1) - (2*(2^31%n))%n */
        m = 0xFFFFFFFFUL - (2*((1UL<<31)%n))%n;

    while((r = truerand()) > m)
        ;

    return r%n;
}
```
Uses `truerand()` 155a.

⟨`libc/port/ntruerand.c` 407e⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* `ntruerand` 407d⟩

libc/port/perror.c

⟨libc/port/perror.c 408a⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* perror 234c⟩

libc/port/pool.c

⟨*function* checklist 408b⟩≡ (426e)
```
/*
 * Tree walking
 */

static void
checklist(Free *t)
{
    Free *q;

    for(q=t->next; q!=t; q=q->next){
        assert(q->size == t->size);
        assert(q->next==nil || q->next->prev==q);
        assert(q->prev==nil || q->prev->next==q);
    //  assert(q->left==nil);
    //  assert(q->right==nil);
        assert(q->magic==FREE_MAGIC);
    }
}
```
Uses FREE_MAGIC-123 62b.

⟨*function* checktree 408c⟩≡ (426e)
```
static void
checktree(Free *t, int a, int b)
{
    assert(t->magic==FREE_MAGIC);
    assert(a < t->size && t->size < b);
    assert(t->next==nil || t->next->prev==t);
    assert(t->prev==nil || t->prev->next==t);
    checklist(t);
    if(t->left)
        checktree(t->left, a, t->size);
    if(t->right)
        checktree(t->right, t->size, b);

}
```
Uses FREE_MAGIC-123 62b, checklist() 408b, and checktree() 408c.

⟨*function* ltreewalk 408d⟩≡ (426e)
```
/* ltreewalk: return address of pointer to node of size == size */
static Free**
ltreewalk(Free **t, ulong size)
{
    assert(t != nil /* ltreewalk */);

    for(;;) {
        if(*t == nil)
            return t;

        assert((*t)->magic == FREE_MAGIC);
```

```
        if(size == (*t)->size)
            return t;
        if(size < (*t)->size)
            t = &(*t)->left;
        else
            t = &(*t)->right;
    }
}
```
Uses FREE_MAGIC-123 62b.

⟨*function* treelookup 409a⟩≡ (426e)
```
/* treelookup: find node in tree with size == size */
static Free*
treelookup(Free *t, ulong size)
{
    return *ltreewalk(&t, size);
}
```
Uses ltreewalk() 408d.

⟨*function* treeinsert 409b⟩≡ (426e)
```
/* treeinsert: insert node into tree */
static Free*
treeinsert(Free *tree, Free *node)
{
    Free **loc, *repl;

    assert(node != nil /* treeinsert */);

    loc = ltreewalk(&tree, node->size);
    if(*loc == nil) {
        node->left = nil;
        node->right = nil;
    } else {    /* replace existing node */
        repl = *loc;
        node->left = repl->left;
        node->right = repl->right;
    }
    *loc = node;
    return tree;
}
```
Uses ltreewalk() 408d.

⟨*function* treedelete 409c⟩≡ (426e)
```
/* treedelete: remove node from tree */
static Free*
treedelete(Free *tree, Free *node)
{
    Free **loc, **lsucc, *succ;

    assert(node != nil /* treedelete */);

    loc = ltreewalk(&tree, node->size);
    assert(*loc == node);

    if(node->left == nil)
        *loc = node->right;
    else if(node->right == nil)
        *loc = node->left;
    else {
```

```
        /* have two children, use inorder successor as replacement */
        for(lsucc = &node->right; (*lsucc)->left; lsucc = &(*lsucc)->left)
            ;
        succ = *lsucc;
        *lsucc = succ->right;
        succ->left = node->left;
        succ->right = node->right;
        *loc = succ;
    }

    node->left = node->right = Poison;
    return tree;
}
```

Uses `Poison-133` 63d and `ltreewalk()` 408d.

⟨*function* `treelookupgt` 410a⟩≡ (426e)

```
/* treelookupgt: find smallest node in tree with size >= size */
static Free*
treelookupgt(Free *t, ulong size)
{
    Free *lastgood; /* last node we saw that was big enough */

    lastgood = nil;
    for(;;) {
        if(t == nil)
            return lastgood;
        if(size == t->size)
            return t;
        if(size < t->size) {
            lastgood = t;
            t = t->left;
        } else
            t = t->right;
    }
}
```

⟨*function* `listadd` 410b⟩≡ (426e)

```
/* listadd: add a node to a doubly linked list */
static Free*
listadd(Free *list, Free *node)
{
    if(list == nil) {
        node->next = node;
        node->prev = node;
        return node;
    }

    node->prev = list->prev;
    node->next = list;

    node->prev->next = node;
    node->next->prev = node;

    return list;
}
```

⟨*function* `listdelete` 410c⟩≡ (426e)

```
/* listdelete: remove node from a doubly linked list */
static Free*
listdelete(Pool *p, Free *list, Free *node)
```

```
{
    if(node->next == node) {    /* singular list */
        node->prev = node->next = Poison;
        return nil;
    }
    if(node->next == nil)
        p->panic(p, "pool->next");
    if(node->prev == nil)
        p->panic(p, "pool->prev");
    node->next->prev = node->prev;
    node->prev->next = node->next;

    if(list == node)
        list = node->next;

    node->prev = node->next = Poison;
    return list;
}
```

Uses Poison-133 63d.

⟨*function* pooladd 411a⟩≡ (426e)

```
/* pooladd: add anode to the free pool */
static Free*
pooladd(Pool *p, Alloc *anode)
{
    Free *lst, *olst;
    Free *node;
    Free **parent;

    antagonism {
        memmark(_B2D(anode), 0xF7, anode->size-sizeof(Bhdr)-sizeof(Btail));
    }

    node = (Free*)anode;
    node->magic = FREE_MAGIC;
    parent = ltreewalk(&p->freeroot, node->size);
    olst = *parent;
    lst = listadd(olst, node);
    if(olst != lst) /* need to update tree */
        *parent = treeinsert(*parent, lst);
    p->curfree += node->size;
    return node;
}
```

⟨*function* pooldel 411b⟩≡ (426e)

```
/* pooldel: remove node from the free pool */
static Alloc*
pooldel(Pool *p, Free *node)
{
    Free *lst, *olst;
    Free **parent;

    parent = ltreewalk(&p->freeroot, node->size);
    olst = *parent;
    assert(olst != nil /* pooldel */);

    lst = listdelete(p, olst, node);
    if(lst == nil)
        *parent = treedelete(*parent, olst);
    else if(lst != olst)
```

```
        *parent = treeinsert(*parent, lst);

    node->left = node->right = Poison;
    p->curfree -= node->size;

    antagonism {
        memmark(_B2D(node), 0xF9, node->size-sizeof(Bhdr)-sizeof(Btail));
    }

    node->magic = UNALLOC_MAGIC;
    return (Alloc*)node;
}
```

⟨*function* dsize2bsize 412a⟩≡ (426e)

```
/* block allocation */
static ulong
dsize2bsize(Pool *p, ulong sz)
{
    sz += sizeof(Bhdr)+sizeof(Btail);
    if(sz < p->minblock)
        sz = p->minblock;
    if(sz < MINBLOCKSIZE)
        sz = MINBLOCKSIZE;
    sz = (sz+p->quantum-1)&~(p->quantum-1);
    return sz;
}
```

Uses MINBLOCKSIZE-131 63b.

⟨*function* bsize2asize 412b⟩≡ (426e)

```
static ulong
bsize2asize(Pool *p, ulong sz)
{
    sz += sizeof(Arena)+sizeof(Btail);
    if(sz < p->minarena)
        sz = p->minarena;
    sz = (sz+p->quantum)&~(p->quantum-1);
    return sz;
}
```

⟨*function* blockmerge 412c⟩≡ (426e)

```
/* both are removed from pool if necessary. */
static Alloc*
blockmerge(Pool *pool, Bhdr *a, Bhdr *b)
{
    Btail *t;

    assert(B2NB(a) == b);

    if(a->magic == FREE_MAGIC)
        pooldel(pool, (Free*)a);
    if(b->magic == FREE_MAGIC)
        pooldel(pool, (Free*)b);

    t = B2T(a);
    t->size = (ulong)Poison;
    t->magic0 = NOT_MAGIC;
    t->magic1 = NOT_MAGIC;
    PSHORT(t->datasize, NOT_MAGIC);

    a->size += b->size;
```

```
    t = B2T(a);
    t->size = a->size;
    PSHORT(t->datasize, 0xFFFF);

    b->size = NOT_MAGIC;
    b->magic = NOT_MAGIC;

    a->magic = UNALLOC_MAGIC;
    return (Alloc*)a;
}
```

Uses B2NB-115 61b, B2T-120 61g, FREE_MAGIC-123 62b, NOT_MAGIC-113 61a, PSHORT-117 61d, Poison-133 63d, and UNALLOC_MAGIC-125 62d.

⟨*function* blocksetsize 413a⟩≡ (426e)

```
/* blocksetsize: set the total size of a block, fixing tail pointers */
static Bhdr*
blocksetsize(Bhdr *b, ulong bsize)
{
    Btail *t;

    assert(b->magic != FREE_MAGIC /* blocksetsize */);

    b->size = bsize;
    t = B2T(b);
    t->size = b->size;
    t->magic0 = TAIL_MAGIC0;
    t->magic1 = TAIL_MAGIC1;
    return b;
}
```

Uses B2T-120 61g, FREE_MAGIC-123 62b, TAIL_MAGIC0-118 61e, and TAIL_MAGIC1-119 61e.

⟨*function* getdsize 413b⟩≡ (426e)

```
/* getdsize: return the requested data size for an allocated block */
static ulong
getdsize(Alloc *b)
{
    Btail *t;
    t = B2T(b);
    return b->size - SHORT(t->datasize);
}
```

Uses B2T-120 61g and SHORT-116 61c.

⟨*function* trim 413c⟩≡ (426e)

```
/* trim: trim a block down to what is needed to hold dsize bytes of user data */
Alloc*
trim(Pool *p, Alloc *b, ulong dsize)
{
    ulong extra, bsize;
    Alloc *frag;

    bsize = dsize2bsize(p, dsize);
    extra = b->size - bsize;
    if(b->size - dsize >= 0x10000 ||
      (extra >= bsize>>2 && extra >= MINBLOCKSIZE && extra >= p->minblock)) {
        blocksetsize(b, bsize);
        frag = (Alloc*) B2NB(b);

        antagonism {
            memmark(frag, 0xF1, extra);
```

```
        }

        frag->magic = UNALLOC_MAGIC;
        blocksetsize(frag, extra);
        pooladd(p, frag);
    }

    b->magic = ALLOC_MAGIC;
    blocksetdsize(p, b, dsize);
    return b;
}
```

⟨*function* freefromfront 414a⟩≡ (426e)

```
static Alloc*
freefromfront(Pool *p, Alloc *b, ulong skip)
{
    Alloc *bb;

    skip = skip&~(p->quantum-1);
    if(skip >= 0x1000 || (skip >= b->size>>2 && skip >= MINBLOCKSIZE && skip >= p->minblock)){
        bb = (Alloc*)((uchar*)b+skip);
        blocksetsize(bb, b->size-skip);
        bb->magic = UNALLOC_MAGIC;
        blocksetsize(b, skip);
        b->magic = UNALLOC_MAGIC;
        pooladd(p, b);
        return bb;
    }
    return b;
}
```

Uses MINBLOCKSIZE-131 63b, UNALLOC_MAGIC-125 62d, and blocksetsize() 413a.

⟨*function* arenasetsize 414b⟩≡ (426e)

```
/* arenasetsize: set arena size, updating tail */
static void
arenasetsize(Arena *a, ulong asize)
{
    Bhdr *atail;

    a->asize = asize;
    atail = A2TB(a);
    atail->magic = ARENATAIL_MAGIC;
    atail->size = 0;
}
```

Uses A2TB-128 62g and ARENATAIL_MAGIC-127 62f.

⟨*function* poolnewarena 414c⟩≡ (426e)

```
/* poolnewarena: allocate new arena */
static void
poolnewarena(Pool *p, ulong asize)
{
    Arena *a;
    Arena *ap, *lastap;
    Alloc *b;

    LOG(p, "newarena %lud\n", asize);
    if(p->cursize+asize > p->maxsize) {
        if(poolcompactl(p) == 0){
            LOG(p, "pool too big: %lud+%lud > %lud\n",
                p->cursize, asize, p->maxsize);
```

```
            werrstr("memory pool too large");
        }
        return;
    }

    if((a = p->alloc(asize)) == nil) {
        /* assume errstr set by p->alloc */
        return;
    }

    p->cursize += asize;

    /* arena hdr */
    a->magic = ARENA_MAGIC;
    blocksetsize(a, sizeof(Arena));
    arenasetsize(a, asize);
    blockcheck(p, a);

    /* create one large block in arena */
    b = (Alloc*)A2B(a);
    b->magic = UNALLOC_MAGIC;
    blocksetsize(b, (uchar*)A2TB(a)-(uchar*)b);
    blockcheck(p, b);
    pooladd(p, b);
    blockcheck(p, b);

    /* sort arena into descending sorted arena list */
    for(lastap=nil, ap=p->arenalist; ap > a; lastap=ap, ap=ap->down)
        ;

    if(a->down = ap)    /* assign = */
        a->down->aup = a;

    if(a->aup = lastap) /* assign = */
        a->aup->down = a;
    else
        p->arenalist = a;

    /* merge with surrounding arenas if possible */
    /* must do a with up before down with a (think about it) */
    if(a->aup)
        arenamerge(p, a, a->aup);
    if(a->down)
        arenamerge(p, a->down, a);
}
```

Uses `A2B-129` 62h, `A2TB-128` 62g, `ARENA_MAGIC-126` 62f, `UNALLOC_MAGIC-125` 62d, `arenamerge()` 416a, `arenasetsize()` 414b, `blocksetsize()` 413a, `poolcompactl()` 420a, and `werrstr()` 234b.

⟨*function* `blockgrow` 415⟩≡ (426e)

```
/* trimming it into two different blocks. */
static void
blockgrow(Pool *p, Bhdr *b, ulong nsize)
{
    if(b->magic == FREE_MAGIC) {
        Alloc *a;
        Bhdr *bnxt;
        a = pooldel(p, (Free*)b);
        blockcheck(p, a);
        blocksetsize(a, nsize);
        blockcheck(p, a);
```

```
        bnxt = B2NB(a);
        if(bnxt->magic == FREE_MAGIC)
            a = blockmerge(p, a, bnxt);
        blockcheck(p, a);
        pooladd(p, a);
    } else {
        Alloc *a;
        ulong dsize;

        a = (Alloc*)b;
        dsize = getdsize(a);
        blocksetsize(a, nsize);
        trim(p, a, dsize);
    }
}
```

Uses `B2NB-115` 61b, `FREE_MAGIC-123` 62b, `blockmerge()` 412c, `blocksetsize()` 413a, and `getdsize()` 413b.

⟨*function* `arenamerge` 416a⟩≡ (426e)

```
/* arenamerge: attempt to coalesce to arenas that might be adjacent */
static Arena*
arenamerge(Pool *p, Arena *bot, Arena *top)
{
    Bhdr *bbot, *btop;
    Btail *t;

    blockcheck(p, bot);
    blockcheck(p, top);
    assert(bot->aup == top && top > bot);

    if(p->merge == nil || p->merge(bot, top) == 0)
        return nil;

    /* remove top from list */
    if(bot->aup = top->aup) /* assign = */
        bot->aup->down = bot;
    else
        p->arenalist = bot;

    /* save ptrs to last block in bot, first block in top */
    t = B2PT(A2TB(bot));
    bbot = T2HDR(t);
    btop = A2B(top);
    blockcheck(p, bbot);
    blockcheck(p, btop);

    /* grow bottom arena to encompass top */
    arenasetsize(bot, top->asize + ((uchar*)top - (uchar*)bot));

    /* grow bottom block to encompass space between arenas */
    blockgrow(p, bbot, (uchar*)btop-(uchar*)bbot);
    blockcheck(p, bbot);
    return bot;
}
```

Uses `A2B-129` 62h, `A2TB-128` 62g, `B2PT-121` 61h, `T2HDR-122` 61i, `arenasetsize()` 414b, and `blockgrow()` 415.

⟨*function* `dumpblock` 416b⟩≡ (426e)

```
/* dumpblock: print block's vital stats */
static void
dumpblock(Pool *p, Bhdr *b)
{
```

```
    ulong *dp;
    ulong dsize;
    uchar *cp;

    dp = (ulong*)b;
    p->print(p, "pool %s block %p\nhdr %.8lux %.8lux %.8lux %.8lux %.8lux %.8lux\n",
        p->name, b, dp[0], dp[1], dp[2], dp[3], dp[4], dp[5], dp[6]);

    dp = (ulong*)B2T(b);
    p->print(p, "tail %.8lux %.8lux %.8lux %.8lux %.8lux %.8lux | %.8lux %.8lux\n",
        dp[-6], dp[-5], dp[-4], dp[-3], dp[-2], dp[-1], dp[0], dp[1]);

    if(b->magic == ALLOC_MAGIC){
        dsize = getdsize((Alloc*)b);
        if(dsize >= b->size)    /* user data size corrupt */
            return;

        cp = (uchar*)_B2D(b)+dsize;
        p->print(p, "user data ");
        p->print(p, "%.2ux %.2ux %.2ux %.2ux  %.2ux %.2ux %.2ux %.2ux",
            cp[-8], cp[-7], cp[-6], cp[-5], cp[-4], cp[-3], cp[-2], cp[-1]);
        p->print(p, " | %.2ux %.2ux %.2ux %.2ux  %.2ux %.2ux %.2ux %.2ux\n",
            cp[0], cp[1], cp[2], cp[3], cp[4], cp[5], cp[6], cp[7]);
    }
}
```

Uses `ALLOC_MAGIC-124` 62d, `B2T-120` 61g, `_B2D-134` 63e, and `getdsize()` 413b.

⟨*function* `printblock` 417a⟩≡ (426e)

```
static void
printblock(Pool *p, Bhdr *b, char *msg)
{
    p->print(p, "%s\n", msg);
    dumpblock(p, b);
}
```

Uses `dumpblock()` 416b.

⟨*function* `panicblock` 417b⟩≡ (426e)

```
static void
panicblock(Pool *p, Bhdr *b, char *msg)
{
    p->print(p, "%s\n", msg);
    dumpblock(p, b);
    p->panic(p, "pool panic");
}
```

Uses `dumpblock()` 416b.

⟨*function* `blockcheck` 417c⟩≡ (426e)

```
/* should only be called when holding pool lock */
static void
blockcheck(Pool *p, Bhdr *b)
{
    Alloc *a;
    Btail *t;
    int i, n;
    uchar *q, *bq, *eq;
    ulong dsize;

    switch(b->magic) {
    default:
```

```
        panicblock(p, b, "bad magic");
    case FREE_MAGIC:
    case UNALLOC_MAGIC:
        t = B2T(b);
        if(t->magic0 != TAIL_MAGIC0 || t->magic1 != TAIL_MAGIC1)
            panicblock(p, b, "corrupt tail magic");
        if(T2HDR(t) != b)
            panicblock(p, b, "corrupt tail ptr");
        break;
    case DEAD_MAGIC:
        t = B2T(b);
        if(t->magic0 != TAIL_MAGIC0 || t->magic1 != TAIL_MAGIC1)
            panicblock(p, b, "corrupt tail magic");
        if(T2HDR(t) != b)
            panicblock(p, b, "corrupt tail ptr");
        n = getdsize((Alloc*)b);
        q = _B2D(b);
        q += 8;
        for(i=8; i<n; i++)
            if(*q++ != 0xDA)
                panicblock(p, b, "dangling pointer write");
        break;
    case ARENA_MAGIC:
        b = A2TB((Arena*)b);
        if(b->magic != ARENATAIL_MAGIC)
            panicblock(p, b, "bad arena size");
        /* fall through */
    case ARENATAIL_MAGIC:
        if(b->size != 0)
            panicblock(p, b, "bad arena tail size");
        break;
    case ALLOC_MAGIC:
        a = (Alloc*)b;
        t = B2T(b);
        dsize = getdsize(a);
        bq = (uchar*)_B2D(a)+dsize;
        eq = (uchar*)t;

        if(t->magic0 != TAIL_MAGIC0){
            /* if someone wrote exactly one byte over and it was a NUL, we sometimes only complain. */
            if((p->flags & POOL_TOLERANCE) && bq == eq && t->magic0 == 0)
                printblock(p, b, "mem user overflow (magic0)");
            else
                panicblock(p, b, "corrupt tail magic0");
        }

        if(t->magic1 != TAIL_MAGIC1)
            panicblock(p, b, "corrupt tail magic1");
        if(T2HDR(t) != b)
            panicblock(p, b, "corrupt tail ptr");

        if(dsize2bsize(p, dsize)  > a->size)
            panicblock(p, b, "too much block data");

        if(eq > bq+4)
            eq = bq+4;
        for(q=bq; q<eq; q++){
            if(*q != datamagic[((uintptr)q)%nelem(datamagic)]){
                if(q == bq && *q == 0 && (p->flags & POOL_TOLERANCE)){
                    printblock(p, b, "mem user overflow");
```

```
                    continue;
                }
                panicblock(p, b, "mem user overflow");
            }
        }
        break;
    }
}
```

⟨*enum* _anon_ (port/pool.c)*8* 419a⟩≡ (426e)

```
/*
 * compact an arena by shifting all the free blocks to the end.
 * assumes pool lock is held.
 */
enum {
    FLOATING_MAGIC = 0xCBCBCBCB,    /* temporarily neither allocated nor in the free tree */
};
```

⟨*function* arenacompact 419b⟩≡ (426e)

```
static int
arenacompact(Pool *p, Arena *a)
{
    Bhdr *b, *wb, *eb, *nxt;
    int compacted;

    if(p->move == nil)
        p->panic(p, "don't call me when pool->move is nil\n");

    poolcheckarena(p, a);
    eb = A2TB(a);
    compacted = 0;
    for(b=wb=A2B(a); b && b < eb; b=nxt) {
        nxt = B2NB(b);
        switch(b->magic) {
        case FREE_MAGIC:
            pooldel(p, (Free*)b);
            b->magic = FLOATING_MAGIC;
            break;
        case ALLOC_MAGIC:
            if(wb != b) {
                memmove(wb, b, b->size);
                p->move(_B2D(b), _B2D(wb));
                compacted = 1;
            }
            wb = B2NB(wb);
            break;
        }
    }

    /*
     * the only free data is now at the end of the arena, pointed
     * at by wb.  all we need to do is set its size and get out.
     */
    if(wb < eb) {
        wb->magic = UNALLOC_MAGIC;
        blocksetsize(wb, (uchar*)eb-(uchar*)wb);
        pooladd(p, (Alloc*)wb);
    }

    return compacted;
}
```

⟨*function* poolcompactl 420a⟩≡ (426e)

```
/*
 * compact a pool by compacting each individual arena.
 * 'twould be nice to shift blocks from one arena to the
 * next but it's a pain to code.
 */
static int
poolcompactl(Pool *pool)
{
    Arena *a;
    int compacted;

    if(pool->move == nil || pool->lastcompact == pool->nfree)
        return 0;

    pool->lastcompact = pool->nfree;
    compacted = 0;
    for(a=pool->arenalist; a; a=a->down)
        compacted |= arenacompact(pool, a);
    return compacted;
}
```

⟨*function* B2D 420b⟩≡ (426e)

```
/*
static void*
_B2D(void *a)
{
    return (uchar*)a+sizeof(Bhdr);
}
*/

static void*
B2D(Pool *p, Alloc *a)
{
    if(a->magic != ALLOC_MAGIC)
        p->panic(p, "B2D called on unworthy block");
    return _B2D(a);
}
```

Uses ALLOC_MAGIC-124 62d and _B2D-134 63e.

⟨*function* D2B 420c⟩≡ (426e)

```
/*
static void*
_D2B(void *v)
{
    Alloc *a;
    a = (Alloc*)((uchar*)v-sizeof(Bhdr));
    return a;
}
*/

static Alloc*
D2B(Pool *p, void *v)
{
    Alloc *a;
    ulong *u;

    if((uintptr)v&(sizeof(ulong)-1))
        v = (char*)v - ((uintptr)v&(sizeof(ulong)-1));
    u = v;
```

```
    while(u[-1] == ALIGN_MAGIC)
        u--;
    a = _D2B(u);
    if(a->magic != ALLOC_MAGIC)
        p->panic(p, "D2B called on non-block %p (double-free?)", v);
    return a;
}
```

Uses ALIGN_MAGIC-130 63a, ALLOC_MAGIC-124 62d, and _D2B-135 63f.

⟨*function* poolrealloc1 421⟩≡ (426e)

```
/* poolrealloc1: attempt to grow v to ndsize bytes; assumes lock held */
static void*
poolrealloc1(Pool *p, void *v, ulong ndsize)
{
    Alloc *a;
    Bhdr *left, *right, *newb;
    Btail *t;
    ulong nbsize;
    ulong odsize;
    ulong obsize;
    void *nv;

    if(v == nil)    /* for ANSI */
        return poolalloc1(p, ndsize);
    if(ndsize == 0) {
        poolfree1(p, v);
        return nil;
    }
    a = D2B(p, v);
    blockcheck(p, a);
    odsize = getdsize(a);
    obsize = a->size;

    /* can reuse the same block? */
    nbsize = dsize2bsize(p, ndsize);
    if(nbsize <= a->size) {
    Returnblock:
        if(v != _B2D(a))
            memmove(_B2D(a), v, odsize);
        a = trim(p, a, ndsize);
        p->curalloc -= obsize;
        p->curalloc += a->size;
        v = B2D(p, a);
        return v;
    }

    /* can merge with surrounding blocks? */
    right = B2NB(a);
    if(right->magic == FREE_MAGIC && a->size+right->size >= nbsize) {
        a = blockmerge(p, a, right);
        goto Returnblock;
    }

    t = B2PT(a);
    left = T2HDR(t);
    if(left->magic == FREE_MAGIC && left->size+a->size >= nbsize) {
        a = blockmerge(p, left, a);
        goto Returnblock;
    }
```

```
    if(left->magic == FREE_MAGIC && right->magic == FREE_MAGIC
    && left->size+a->size+right->size >= nbsize) {
        a = blockmerge(p, blockmerge(p, left, a), right);
        goto Returnblock;
    }

    if((nv = poolallocl(p, ndsize)) == nil)
        return nil;

    /* maybe the new block is next to us; if so, merge */
    left = T2HDR(B2PT(a));
    right = B2NB(a);
    newb = D2B(p, nv);
    if(left == newb || right == newb) {
        if(left == newb || left->magic == FREE_MAGIC)
            a = blockmerge(p, left, a);
        if(right == newb || right->magic == FREE_MAGIC)
            a = blockmerge(p, a, right);
        assert(a->size >= nbsize);
        goto Returnblock;
    }

    /* enough cleverness */
    memmove(nv, v, odsize);
    antagonism {
        memset((char*)nv+odsize, 0xDE, ndsize-odsize);
    }
    poolfreel(p, v);
    return nv;
}
```

⟨*function* alignptr 422a⟩≡ (426e)

```
static void*
alignptr(void *v, ulong align, long offset)
{
    char *c;
    ulong off;

    c = v;
    if(align){
        off = (uintptr)c%align;
        if(off != offset){
            c += offset - off;
            if(off > offset)
                c += align;
        }
    }
    return c;
}
```

⟨*function* poolallocalignl 422b⟩≡ (426e)

```
/* poolallocalignl: allocate as described below; assumes pool locked */
static void*
poolallocalignl(Pool *p, ulong dsize, ulong align, long offset, ulong span)
{
    ulong asize;
    void *v;
    char *c;
    ulong *u;
    int skip;
```

```
Alloc *b;

/*
 * allocate block
 *  dsize bytes
 *  addr == offset (modulo align)
 *  does not cross span-byte block boundary
 *
 * to satisfy alignment, just allocate an extra
 * align bytes and then shift appropriately.
 *
 * to satisfy span, try once and see if we're
 * lucky.  the second time, allocate 2x asize
 * so that we definitely get one not crossing
 * the boundary.
 */
if(align){
    if(offset < 0)
        offset = align - ((-offset)%align);
    else
        offset %= align;
}
asize = dsize+align;
v = poolallocl(p, asize);
if(v == nil)
    return nil;
if(span && (uintptr)v/span != ((uintptr)v+asize)/span){
    /* try again */
    poolfreel(p, v);
    v = poolallocl(p, 2*asize);
    if(v == nil)
        return nil;
}

/*
 * figure out what pointer we want to return
 */
c = alignptr(v, align, offset);
if(span && (uintptr)c/span != (uintptr)(c+dsize-1)/span){
    c += span - (uintptr)c%span;
    c = alignptr(c, align, offset);
    if((uintptr)c/span != (uintptr)(c+dsize-1)/span){
        poolfreel(p, v);
        werrstr("cannot satisfy dsize %lud span %lud with align %lud+%ld", dsize, span, align, offset);
        return nil;
    }
}
skip = c - (char*)v;

/*
 * free up the skip bytes before that pointer
 * or mark it as unavailable.
 */
b = _D2B(v);
b = freefromfront(p, b, skip);
v = _B2D(b);
skip = c - (char*)v;
if(c > (char*)v){
    u = v;
    while(c >= (char*)u+sizeof(ulong))
```

```
            *u++ = ALIGN_MAGIC;
    }
    trim(p, b, skip+dsize);
    assert(D2B(p, c) == b);
    antagonism {
        memset(c, 0xDD, dsize);
    }
    return c;
}
```

⟨*function* poolallocalign 424a⟩≡ (426e)

```
void*
poolallocalign(Pool *p, ulong n, ulong align, long offset, ulong span)
{
    void *v;

    p->lock(p);
    paranoia {
        poolcheckl(p);
    }
    verbosity {
        pooldumpl(p);
    }
    v = poolallocalignl(p, n, align, offset, span);
    paranoia {
        poolcheckl(p);
    }
    verbosity {
        pooldumpl(p);
    }
    if(p->logstack && (p->flags & POOL_LOGGING)) p->logstack(p);
    LOG(p, "poolalignspanalloc %p %lud %lud %lud %ld = %p\n", p, n, align, span, offset, v);
    p->unlock(p);
    return v;
}
```

⟨*function* poolcompact 424b⟩≡ (426e)

```
int
poolcompact(Pool *p)
{
    int rv;

    p->lock(p);
    paranoia {
        poolcheckl(p);
    }
    verbosity {
        pooldumpl(p);
    }
    rv = poolcompactl(p);
    paranoia {
        poolcheckl(p);
    }
    verbosity {
        pooldumpl(p);
    }
    LOG(p, "poolcompact %p\n", p);
    p->unlock(p);
    return rv;
}
```

⟨*function* poolcheckarena 425a⟩≡ (426e)

```
/*
 * Debugging
 */

static void
poolcheckarena(Pool *p, Arena *a)
{
    Bhdr *b;
    Bhdr *atail;

    atail = A2TB(a);
    for(b=a; b->magic != ARENATAIL_MAGIC && b<atail; b=B2NB(b))
        blockcheck(p, b);
    blockcheck(p, b);
    if(b != atail)
        p->panic(p, "found wrong tail");
}
```

Uses A2TB-128 62g, ARENATAIL_MAGIC-127 62f, and B2NB-115 61b.

⟨*function* poolcheckl 425b⟩≡ (426e)

```
static void
poolcheckl(Pool *p)
{
    Arena *a;

    for(a=p->arenalist; a; a=a->down)
        poolcheckarena(p, a);
    if(p->freeroot)
        checktree(p->freeroot, 0, 1<<30);
}
```

Uses checktree() 408c and poolcheckarena() 425a.

⟨*function* poolcheck 425c⟩≡ (426e)

```
void
poolcheck(Pool *p)
{
    p->lock(p);
    poolcheckl(p);
    p->unlock(p);
}
```

Uses poolcheckl() 425b.

⟨*function* poolblockcheck 425d⟩≡ (426e)

```
void
poolblockcheck(Pool *p, void *v)
{
    if(v == nil)
        return;

    p->lock(p);
    blockcheck(p, D2B(p, v));
    p->unlock(p);
}
```

Uses D2B() 420c.

⟨*function* pooldumpl 426a⟩≡ (426e)
```
static void
pooldumpl(Pool *p)
{
    Arena *a;

    p->print(p, "pool %p %s\n", p, p->name);
    for(a=p->arenalist; a; a=a->down)
        pooldumparena(p, a);
}
```
Uses pooldumparena() 426c.

⟨*function* pooldump 426b⟩≡ (426e)
```
void
pooldump(Pool *p)
{
    p->lock(p);
    pooldumpl(p);
    p->unlock(p);
}
```
Uses pooldumpl() 426a.

⟨*function* pooldumparena 426c⟩≡ (426e)
```
static void
pooldumparena(Pool *p, Arena *a)
{
    Bhdr *b;

    for(b=a; b->magic != ARENATAIL_MAGIC; b=B2NB(b))
        p->print(p, "(%p %.8lux %lud)", b, b->magic, b->size);
    p->print(p, "\n");
}
```
Uses ARENATAIL_MAGIC-127 62f and B2NB-115 61b.

⟨*function* memmark 426d⟩≡ (426e)
```
/*
 * mark the memory in such a way that we know who marked it
 * (via the signature) and we know where the marking started.
 */
static void
memmark(void *v, int sig, ulong size)
{
    uchar *p, *ep;
    ulong *lp, *elp;
    lp = v;
    elp = lp+size/4;
    while(lp < elp)
        *lp++ = (sig<<24) ^ ((uintptr)lp-(uintptr)v);
    p = (uchar*)lp;
    ep = (uchar*)v+size;
    while(p<ep)
        *p++ = sig;
}
```

⟨libc/port/pool.c 426e⟩≡
```
/*
 * This allocator takes blocks from a coarser allocator (p->alloc) and
 * uses them as arenas.
 *
```

```
 * An arena is split into a sequence of blocks of variable size.  The
 * blocks begin with a Bhdr that denotes the length (including the Bhdr)
 * of the block.  An arena begins with an Arena header block (Arena,
 * ARENA_MAGIC) and ends with a Bhdr block with magic ARENATAIL_MAGIC and
 * size 0.  Intermediate blocks are either allocated or free.  At the end
 * of each intermediate block is a Btail, which contains information
 * about where the block starts.  This is useful for walking backwards.
 *
 * Free blocks (Free*) have a magic value of FREE_MAGIC in their Bhdr
 * headers.  They are kept in a binary tree (p->freeroot) traversible by
 * walking ->left and ->right.  Each node of the binary tree is a pointer
 * to a circular doubly-linked list (next, prev) of blocks of identical
 * size.  Blocks are added to this ``tree of lists'' by pooladd(), and
 * removed by pooldel().
 *
 * When freed, adjacent blocks are coalesced to create larger blocks when
 * possible.
 *
 * Allocated blocks (Alloc*) have one of two magic values: ALLOC_MAGIC or
 * UNALLOC_MAGIC.  When blocks are released from the pool, they have
 * magic value UNALLOC_MAGIC.  Once the block has been trimmed by trim()
 * and the amount of user-requested data has been recorded in the
 * datasize field of the tail, the magic value is changed to ALLOC_MAGIC.
 * All blocks returned to callers should be of type ALLOC_MAGIC, as
 * should all blocks passed to us by callers.  The amount of data the user
 * asked us for can be found by subtracting the short in tail->datasize
 * from header->size.  Further, the up to at most four bytes between the
 * end of the user-requested data block and the actual Btail structure are
 * marked with a magic value, which is checked to detect user overflow.
 *
 * The arenas returned by p->alloc are kept in a doubly-linked list
 * (p->arenalist) running through the arena headers, sorted by descending
 * base address (prev, next).  When a new arena is allocated, we attempt
 * to merge it with its two neighbors via p->merge.
 */

⟨libc includes 387a⟩
#include <pool.h>

typedef struct Alloc    Alloc;
typedef struct Arena    Arena;
typedef struct Bhdr Bhdr;
typedef struct Btail    Btail;
typedef struct Free Free;

⟨struct Bhdr 60⟩
⟨enum _anon_ (port/pool.c) 61a⟩
⟨macro B2NB 61b⟩

⟨macro SHORT 61c⟩
⟨macro PSHORT 61d⟩

⟨enum _anon_ (port/pool.c)2 61e⟩
⟨struct Btail 61f⟩
⟨macro B2T 61g⟩
⟨macro B2PT 61h⟩
⟨macro T2HDR 61i⟩
⟨struct Free 62a⟩
⟨enum _anon_ (port/pool.c)3 62b⟩
```

```
⟨struct Alloc 62c⟩
⟨enum _anon_ (port/pool.c)4 62d⟩

⟨struct Arena 62e⟩
⟨enum _anon_ (port/pool.c)5 62f⟩
⟨macro A2TB 62g⟩
⟨macro A2B 62h⟩

⟨enum _anon_ (port/pool.c)6 63a⟩

⟨enum _anon_ (port/pool.c)7 63b⟩

⟨global datamagic 63c⟩

⟨constant Poison 63d⟩

⟨macro _B2D 63e⟩
⟨macro _D2B 63f⟩

// static void* _B2D(void*);
// static void* _D2B(void*);
static void*    B2D(Pool*, Alloc*);
static Alloc*   D2B(Pool*, void*);
static Arena*   arenamerge(Pool*, Arena*, Arena*);
static void     blockcheck(Pool*, Bhdr*);
static Alloc*   blockmerge(Pool*, Bhdr*, Bhdr*);
static Alloc*   blocksetdsize(Pool*, Alloc*, ulong);
static Bhdr*    blocksetsize(Bhdr*, ulong);
static ulong    bsize2asize(Pool*, ulong);
static ulong    dsize2bsize(Pool*, ulong);
static ulong    getdsize(Alloc*);
static Alloc*   trim(Pool*, Alloc*, ulong);
static Free*    listadd(Free*, Free*);
static void     logstack(Pool*);
static Free**   ltreewalk(Free**, ulong);
static void     memmark(void*, int, ulong);
static Free*    pooladd(Pool*, Alloc*);
static void*    poolallocl(Pool*, ulong);
static void     poolcheckl(Pool*);
static void     poolcheckarena(Pool*, Arena*);
static int      poolcompactl(Pool*);
static Alloc*   pooldel(Pool*, Free*);
static void     pooldumpl(Pool*);
static void     pooldumparena(Pool*, Arena*);
static void     poolfreel(Pool*, void*);
static void     poolnewarena(Pool*, ulong);
static void*    poolreallocl(Pool*, void*, ulong);
static Free*    treedelete(Free*, Free*);
static Free*    treeinsert(Free*, Free*);
static Free*    treelookup(Free*, ulong);
static Free*    treelookupgt(Free*, ulong);

/*
 * Debugging
 *
 * Antagonism causes blocks to always be filled with garbage if their
 * contents are undefined.  This tickles both programs and the library.
 * It's a linear time hit but not so noticeable during nondegenerate use.
 * It would be worth leaving in except that it negates the benefits of the
 * kernel's demand-paging.  The tail magic and end-of-data magic
```

```
 * provide most of the user-visible benefit that antagonism does anyway.
 *
 * Paranoia causes the library to recheck the entire pool on each lock
 * or unlock.  A failed check on unlock means we tripped over ourselves,
 * while a failed check on lock tends to implicate the user.  Paranoia has
 * the potential to slow things down a fair amount for pools with large
 * numbers of allocated blocks.  It completely negates all benefits won
 * by the binary tree.  Turning on paranoia in the kernel makes it painfully
 * slow.
 *
 * Verbosity induces the dumping of the pool via p->print at each lock operation.
 * By default, only one line is logged for each alloc, free, and realloc.
 */

/* the if(!x);else avoids ``dangling else'' problems */
#define antagonism  if(!(p->flags & POOL_ANTAGONISM)){}else
#define paranoia    if(!(p->flags & POOL_PARANOIA)){}else
#define verbosity   if(!(p->flags & POOL_VERBOSITY)){}else

#define DPRINT  if(!(p->flags & POOL_DEBUGGING)){}else p->print
#define LOG     if(!(p->flags & POOL_LOGGING)){}else p->print

⟨function checklist 408b⟩

⟨function checktree 408c⟩

⟨function ltreewalk 408d⟩

⟨function treelookup 409a⟩

⟨function treeinsert 409b⟩

⟨function treedelete 409c⟩

⟨function treelookupgt 410a⟩

/*
 * List maintenance
 */

⟨function listadd 410b⟩
⟨function listdelete 410c⟩

/*
 * Pool maintenance
 */

⟨function pooladd 411a⟩
⟨function pooldel 411b⟩

/*
 * Block maintenance
 */
⟨function dsize2bsize 412a⟩
⟨function bsize2asize 412b⟩

/* blockmerge: merge a and b, known to be adjacent */
⟨function blockmerge 412c⟩

⟨function blocksetsize 413a⟩
```

```
⟨function getdsize 413b⟩

/* blocksetdsize: set the user data size of a block */
static Alloc*
blocksetdsize(Pool *p, Alloc *b, ulong dsize)
{
    Btail *t;
    uchar *q, *eq;

    assert(b->size >= dsize2bsize(p, dsize));
    assert(b->size - dsize < 0x10000);

    t = B2T(b);
    PSHORT(t->datasize, b->size - dsize);

    q=(uchar*)_B2D(b)+dsize;
    eq = (uchar*)t;
    if(eq > q+4)
        eq = q+4;
    for(; q<eq; q++)
        *q = datamagic[((ulong)(uintptr)q)%nelem(datamagic)];

    return b;
}

⟨function trim 413c⟩

⟨function freefromfront 414a⟩

/*
 * Arena maintenance
 */

⟨function arenasetsize 414b⟩

⟨function poolnewarena 414c⟩

/* blockresize: grow a block to encompass space past its end, possibly by */
⟨function blockgrow 415⟩

⟨function arenamerge 416a⟩

⟨function dumpblock 416b⟩

⟨function printblock 417a⟩

⟨function panicblock 417b⟩

/* blockcheck: ensure a block consistent with our expectations */
⟨function blockcheck 417c⟩

⟨enum _anon_ (port/pool.c)8 419a⟩

⟨function arenacompact 419b⟩

⟨function poolcompactl 420a⟩

/*
static int
```

```
poolcompactl(Pool*)
{
    return 0;
}
*/

/*
 * Actual allocators
 */

⟨function B2D 420b⟩
⟨function D2B 420c⟩

⟨function poolallocl 64b⟩

⟨function poolreallocl 421⟩

⟨function alignptr 422a⟩

⟨function poolallocalignl 422b⟩

⟨function poolfreel 66⟩

⟨function poolalloc 64a⟩

⟨function poolallocalign 424a⟩

⟨function poolcompact 424b⟩

⟨function poolrealloc 67c⟩

⟨function poolfree 65⟩

⟨function poolmsize 68b⟩

⟨function poolcheckarena 425a⟩

⟨function poolcheckl 425b⟩

⟨function poolcheck 425c⟩

⟨function poolblockcheck 425d⟩

⟨function pooldumpl 426a⟩

⟨function pooldump 426b⟩

⟨function pooldumparena 426c⟩

⟨function memmark 426d⟩
```
Uses `Alloc` 62c, `Arena` 62e, `Bhdr` 60, `Btail` 61f, and `Free` 62a.

libc/port/pow.c

⟨`libc/port/pow.c` 431⟩≡
```
⟨libc includes 387a⟩
⟨function pow 139⟩
```

libc/port/pow10.c

```
⟨libc/port/pow10.c 432a⟩≡
  ⟨libc includes 387a⟩
  ⟨global tab 140b⟩

  ⟨function pow10 140a⟩
```

libc/port/profile.c

```
⟨libc/port/profile.c 432b⟩≡
  ⟨libc includes 387a⟩
  #include    <tos.h>

  extern  long    _callpc(void**);
  extern  long    _savearg(void);

  ⟨global khz 357d⟩
  ⟨global perr 357e⟩
  ⟨global havecycles 357f⟩

  typedef struct  Plink   Plink;
  ⟨struct Plink 357c⟩

  #pragma profile off

  ⟨function _profin 358b⟩
  ⟨function _profout 359⟩

  ⟨function _profdump 361⟩

  ⟨function _profinit 360a⟩

  ⟨function _profmain 360b⟩

  ⟨function prof 358a⟩

  #pragma profile on
```

libc/port/qsort.c

```
⟨libc/port/qsort.c 432c⟩≡
  /*
   * qsort -- simple quicksort
   */
  #include <u.h>
  // not even libc.h

  ⟨type Sort 31d⟩

  ⟨function swapb 32b⟩
  ⟨function swapi 32c⟩

  ⟨function pivot 32d⟩

  ⟨function qsorts 33⟩

  ⟨function qsort 32a⟩
```

libc/port/quote.c

⟨*global* doquote 433a⟩≡ (435a)

```
int (*doquote)(int);
```

⟨*function* unquotestrdup 433b⟩≡ (435a)

```
char*
unquotestrdup(char *s)
{
    char *t, *ret;
    int quoting;

    ret = s = strdup(s);    /* return unquoted copy */
    if(ret == nil)
        return ret;
    quoting = 0;
    t = s;  /* s is output string, t is input string */
    while(*t!='\0' && (quoting || (*t!=' ' && *t!='\t'))){
        if(*t != '\''){
            *s++ = *t++;
            continue;
        }
        /* *t is a quote */
        if(!quoting){
            quoting = 1;
            t++;
            continue;
        }
        /* quoting and we're on a quote */
        if(t[1] != '\''){
            /* end of quoted section; absorb closing quote */
            t++;
            quoting = 0;
            continue;
        }
        /* doubled quote; fold one quote into two */
        t++;
        *s++ = *t++;
    }
    if(t != s)
        memmove(s, t, strlen(t)+1);
    return ret;
}
```

Uses memmove() 48, strdup() 82a, and strlen() 80a.

⟨*function* unquoterunestrdup 433c⟩≡ (435a)

```
Rune*
unquoterunestrdup(Rune *s)
{
    Rune *t, *ret;
    int quoting;

    ret = s = runestrdup(s);    /* return unquoted copy */
    if(ret == nil)
        return ret;
    quoting = 0;
    t = s;  /* s is output string, t is input string */
    while(*t!='\0' && (quoting || (*t!=' ' && *t!='\t'))){
        if(*t != '\''){
            *s++ = *t++;
```

```
            continue;
        }
        /* *t is a quote */
        if(!quoting){
            quoting = 1;
            t++;
            continue;
        }
        /* quoting and we're on a quote */
        if(t[1] != '\''){
            /* end of quoted section; absorb closing quote */
            t++;
            quoting = 0;
            continue;
        }
        /* doubled quote; fold one quote into two */
        t++;
        *s++ = *t++;
    }
    if(t != s)
        memmove(s, t, (runestrlen(t)+1)*sizeof(Rune));
    return ret;
}
```

Uses memmove() 48, runestrdup() 87d, and runestrlen() 86a.

⟨*function* quotestrdup 434a⟩≡ (435a)

```
char*
quotestrdup(char *s)
{
    char *t, *u, *ret;
    int quotelen;
    Rune r;

    if(_needsquotes(s, &quotelen) == 0)
        return strdup(s);

    ret = malloc(quotelen+1);
    if(ret == nil)
        return nil;
    u = ret;
    *u++ = '\'';
    for(t=s; *t; t++){
        r = *t;
        if(r == L'\'')
            *u++ = r;   /* double the quote */
        *u++ = r;
    }
    *u++ = '\'';
    *u = '\0';
    return ret;
}
```

Uses _needsquotes() 528a, malloc() 63g, and strdup() 82a.

⟨*function* quoterunestrdup 434b⟩≡ (435a)

```
Rune*
quoterunestrdup(Rune *s)
{
    Rune *t, *u, *ret;
    int quotelen;
    Rune r;
```

```
    if(_runeneedsquotes(s, &quotelen) == 0)
        return runestrdup(s);

    ret = malloc((quotelen+1)*sizeof(Rune));
    if(ret == nil)
        return nil;
    u = ret;
    *u++ = '\'';
    for(t=s; *t; t++){
        r = *t;
        if(r == L'\'')
            *u++ = r;   /* double the quote */
        *u++ = r;
    }
    *u++ = '\'';
    *u = '\0';
    return ret;
}
```

Uses _runeneedsquotes() 528b, malloc() 63g, and runestrdup() 87d.

```
⟨libc/port/quote.c 435a⟩≡
  ⟨libc includes 387a⟩
  ⟨global doquote 433a⟩

  extern int _needsquotes(char*, int*);
  extern int _runeneedsquotes(Rune*, int*);

  ⟨function unquotestrdup 433b⟩

  ⟨function unquoterunestrdup 433c⟩

  ⟨function quotestrdup 434a⟩

  ⟨function quoterunestrdup 434b⟩
```

libc/port/rand.c

```
⟨libc/port/rand.c 435b⟩≡
  ⟨libc includes 387a⟩
  ⟨function rand 153b⟩
```

libc/port/readn.c

```
⟨libc/port/readn.c 435c⟩≡
  ⟨libc includes 387a⟩
  ⟨function readn 192c⟩
```

libc/port/rune.c

```
⟨enum _anon_ (port/rune.c) 435d⟩≡                                    (437a)
  enum
  {
      Bitx    = Bit(1),

      Tx  = T(1),          /* 1000 0000 */
      Rune1 = (1<<(Bit(0)+0*Bitx))-1, /* 0000 0000 0000 0000 0111 1111 */
```

```
    Maskx   = (1<<Bitx)-1,      /* 0011 1111 */
    Testx   = Maskx ^ 0xFF,     /* 1100 0000 */

    SurrogateMin    = 0xD800,
    SurrogateMax    = 0xDFFF,

    Bad = Runeerror,
};
```

Uses `Bit`-102 84a, `Bitx`-105 435d, `Maskx`-108 435d, and `T`-103 84b.

⟨*function* `runelen` 436a⟩≡ (437a)

```
int
runelen(long c)
{
    Rune rune;
    char str[10];

    rune = c;
    return runetochar(str, &rune);
}
```

Uses `runetochar()` 85.

⟨*function* `runenlen` 436b⟩≡ (437a)

```
int
runenlen(Rune *r, int nrune)
{
    int nb, i;
    Rune c;

    nb = 0;
    while(nrune--) {
        c = *r++;
        if(c <= Rune1){
            nb++;
        } else {
            for(i = 2; i < UTFmax + 1; i++)
                if(c <= RuneX(i) || i == UTFmax){
                    nb += i;
                    break;
                }
        }
    }
    return nb;
}
```

Uses `Rune1`-107 435d and `RuneX`-104 84c.

⟨*function* `fullrune` 436c⟩≡ (437a)

```
int
fullrune(char *str, int n)
{
    int  i;
    Rune c;

    if(n <= 0)
        return 0;
    c = *(uchar*)str;
    if(c < Tx)
        return 1;
```

```
    for(i = 3; i < UTFmax + 1; i++)
        if(c < T(i))
            return n >= i - 1;
    return n >= UTFmax;
}
```
Uses T-103 84b and Tx-106 435d.

⟨libc/port/rune.c 437a⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*macro* Bit 84a⟩
  ⟨*macro* T 84b⟩
  ⟨*macro* RuneX 84c⟩

  ⟨*enum* _anon_ (port/rune.c) 435d⟩

  ⟨*function* chartorune 84d⟩

  ⟨*function* runetochar 85⟩

  ⟨*function* runelen 436a⟩

  ⟨*function* runenlen 436b⟩

  ⟨*function* fullrune 436c⟩

libc/port/runebase.c

libc/port/runebsearch.c

⟨libc/port/runebsearch.c 437b⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* _runebsearch 83c⟩

libc/port/runestrcat.c

⟨libc/port/runestrcat.c 437c⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* runestrcat 88a⟩

libc/port/runestrchr.c

⟨libc/port/runestrchr.c 437d⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* runestrchr 87b⟩

libc/port/runestrcmp.c

⟨libc/port/runestrcmp.c 437e⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* runestrcmp 87a⟩

libc/port/runestrcpy.c

⟨libc/port/runestrcpy.c 437f⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* runestrcpy 86b⟩

libc/port/runestrdup.c

⟨libc/port/runestrdup.c 438a⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* runestrdup 87d⟩

libc/port/runestrecpy.c

⟨*function* runestrecpy 438b⟩≡ (438c)

```
Rune*
runestrecpy(Rune *s1, Rune *es1, Rune *s2)
{
    if(s1 >= es1)
        return s1;

    while(*s1++ = *s2++){
        if(s1 == es1){
            *--s1 = '\0';
            break;
        }
    }
    return s1;
}
```

⟨libc/port/runestrecpy.c 438c⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* runestrecpy 438b⟩

libc/port/runestrlen.c

⟨libc/port/runestrlen.c 438d⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* runestrlen 86a⟩

libc/port/runestrncat.c

⟨*function* runestrncat 438e⟩≡ (438f)

```
Rune*
runestrncat(Rune *s1, Rune *s2, long n)
{
    Rune *os1;

    os1 = s1;
    s1 = runestrchr(s1, 0);
    while(*s1++ = *s2++)
        if(--n < 0) {
            s1[-1] = 0;
            break;
        }
    return os1;
}
```

Uses runestrchr() 87b.

⟨libc/port/runestrncat.c 438f⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* runestrncat 438e⟩

libc/port/runestrncmp.c

⟨*function* runestrncmp 439a⟩≡ (439b)

```
int
runestrncmp(Rune *s1, Rune *s2, long n)
{
    Rune c1, c2;

    while(n > 0) {
        c1 = *s1++;
        c2 = *s2++;
        n--;
        if(c1 != c2) {
            if(c1 > c2)
                return 1;
            return -1;
        }
        if(c1 == 0)
            break;
    }
    return 0;
}
```

⟨libc/port/runestrncmp.c 439b⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* runestrncmp 439a⟩

libc/port/runestrncpy.c

⟨*function* runestrncpy 439c⟩≡ (439d)

```
Rune*
runestrncpy(Rune *s1, Rune *s2, long n)
{
    int i;
    Rune *os1;

    os1 = s1;
    for(i = 0; i < n; i++)
        if((*s1++ = *s2++) == 0) {
            while(++i < n)
                *s1++ = 0;
            return os1;
        }
    return os1;
}
```

⟨libc/port/runestrncpy.c 439d⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* runestrncpy 439c⟩

libc/port/runestrrchr.c

⟨libc/port/runestrrchr.c 439e⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* runestrrchr 87c⟩

libc/port/runestrstr.c

```
⟨libc/port/runestrstr.c 440a⟩≡
  ⟨libc includes 387a⟩
  ⟨function runestrstr 88b⟩
```

libc/port/runetype.c and runetypebody-6.2.0.h

```
⟨libc/port/runetype.c 440b⟩≡
  ⟨libc includes 387a⟩
  Rune*_runebsearch(Rune c, Rune *t, int n, int ne);

  #include "runetypebody-6.2.0.h"
```

libc/port/sin.c

```
⟨libc/port/sin.c 440c⟩≡
  /*
      C program for floating point sin/cos.
      Calls modf.
      There are no error exits.
      Coefficients are #3370 from Hart & Cheney (18.80D).
  */

  ⟨libc includes 387a⟩
  ⟨constant p0 (port/sin.c) 141c⟩
  ⟨constant p1 (port/sin.c) 141d⟩
  ⟨constant p2 (port/sin.c) 141e⟩
  ⟨constant p3 (port/sin.c) 141f⟩
  ⟨constant p4 (port/sin.c) 142a⟩
  ⟨constant q0 (port/sin.c) 142b⟩
  ⟨constant q1 (port/sin.c) 142c⟩
  ⟨constant q2 (port/sin.c) 142d⟩
  ⟨constant q3 (port/sin.c) 142e⟩

  ⟨function sinus 142f⟩

  ⟨function cos 141a⟩

  ⟨function sin 141b⟩
```

libc/port/sinh.c

```
⟨libc/port/sinh.c 440d⟩≡
  ⟨libc includes 387a⟩
  ⟨global p0 147a⟩
  ⟨global p1 147b⟩
  ⟨global p2 147c⟩
  ⟨global p3 147d⟩
  ⟨global q0 147e⟩
  ⟨global q1 147f⟩
  ⟨global q2 147g⟩

  ⟨function sinh 148a⟩

  ⟨function cosh 148b⟩
```

libc/port/sqrt.c

⟨libc/port/sqrt.c 441a⟩≡
```
/*
    sqrt returns the square root of its floating
    point argument. Newton's method.

    calls frexp
*/
```
⟨*libc includes* 387a⟩
⟨*function* sqrt 137a⟩

libc/port/strcat.c

⟨libc/port/strcat.c 441b⟩≡
⟨*libc includes* 387a⟩
⟨*function* strcat 80b⟩

libc/port/strchr.c

⟨libc/port/strchr.c 441c⟩≡
⟨*libc includes* 387a⟩
⟨*function* strchr 80c⟩

libc/port/strcmp.c

⟨libc/port/strcmp.c 441d⟩≡
⟨*libc includes* 387a⟩
⟨*function* strcmp 81a⟩

libc/port/strcpy.c

⟨libc/port/strcpy.c 441e⟩≡
⟨*libc includes* 387a⟩
⟨*constant* N 81b⟩

⟨*function* strcpy 81c⟩

libc/port/strcspn.c

⟨*constant N (port/strcspn.c)* 441f⟩≡ (442a)
```
#define N   256
```

⟨*function* strcspn 441g⟩≡ (442a)
```
long
strcspn(char *s, char *b)
{
    char map[N], *os;

    memset(map, 0, N);
    for(;;) {
        map[*(uchar*)b] = 1;
        if(*b++ == 0)
            break;
```

```
    }
    os = s;
    while(map[*(uchar*)s++] == 0)
        ;
    return s - os - 1;
}
```

Uses N-98 441f and memset() 46b.

⟨libc/port/strcspn.c 442a⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*constant N (port/strcspn.c)* 441f⟩

  ⟨*function* strcspn 441g⟩

libc/port/strdup.c

⟨libc/port/strdup.c 442b⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* strdup 82a⟩

libc/port/strecpy.c

⟨*function* strecpy 442c⟩≡ (442d)

```
char*
strecpy(char *to, char *e, char *from)
{
    if(to >= e)
        return to;
    to = memccpy(to, from, '\0', e - to);
    if(to == nil){
        to = e - 1;
        *to = '\0';
    }else{
        to--;
    }
    return to;
}
```

Uses memccpy() 81d.

⟨libc/port/strecpy.c 442d⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* strecpy 442c⟩

libc/port/strlen.c

⟨libc/port/strlen.c 442e⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* strlen 80a⟩

libc/port/strncat.c

⟨*function* strncat 443a⟩≡ (443b)
```
char*
strncat(char *s1, char *s2, long n)
{
    char *os1;

    os1 = s1;
    while(*s1++)
        ;
    s1--;
    while(*s1++ = *s2++)
        if(--n < 0) {
            s1[-1] = 0;
            break;
        }
    return os1;
}
```

⟨libc/port/strncat.c 443b⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* strncat 443a⟩

libc/port/strncmp.c

⟨*function* strncmp 443c⟩≡ (443d)
```
int
strncmp(char *s1, char *s2, long n)
{
    unsigned c1, c2;

    while(n > 0) {
        c1 = *s1++;
        c2 = *s2++;
        n--;
        if(c1 != c2) {
            if(c1 > c2)
                return 1;
            return -1;
        }
        if(c1 == 0)
            break;
    }
    return 0;
}
```

⟨libc/port/strncmp.c 443d⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* strncmp 443c⟩

libc/port/strncpy.c

⟨*function* strncpy 443e⟩≡ (444a)
```
char*
strncpy(char *s1, char *s2, long n)
{
    int i;
```

```
    char *os1;

    os1 = s1;
    for(i = 0; i < n; i++)
        if((*s1++ = *s2++) == 0) {
            while(++i < n)
                *s1++ = 0;
            return os1;
        }
    return os1;
}
```

⟨libc/port/strncpy.c 444a⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* strncpy 443e⟩

libc/port/strpbrk.c

⟨*constant N (port/strpbrk.c)* 444b⟩≡ (444d)

```
#define N   256
```

⟨*function* strpbrk 444c⟩≡ (444d)

```
char*
strpbrk(char *cs, char *cb)
{
    char map[N];
    uchar *s=(uchar*)cs, *b=(uchar*)cb;

    memset(map, 0, N);
    for(;;) {
        map[*b] = 1;
        if(*b++ == 0)
            break;
    }
    while(map[*s++] == 0)
        ;
    if(*--s)
        return (char*)s;
    return 0;
}
```

Uses N-99 444b and memset() 46b.

⟨libc/port/strpbrk.c 444d⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*constant N (port/strpbrk.c)* 444b⟩

  ⟨*function* strpbrk 444c⟩

libc/port/strrchr.c

⟨libc/port/strrchr.c 444e⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* strrchr 80d⟩

libc/port/strspn.c

⟨*constant N (port/strspn.c)* 444f⟩≡ (445b)

```
#define N   256
```

⟨*function* strspn 445a⟩≡ (445b)

```
long
strspn(char *s, char *b)
{
    char map[N], *os;

    memset(map, 0, N);
    while(*b)
        map[*(uchar *)b++] = 1;
    os = s;
    while(map[*(uchar *)s++])
        ;
    return s - os - 1;
}
```

Uses N-192 444f and memset() 46b.

⟨libc/port/strspn.c 445b⟩≡

```
⟨libc includes 387a⟩
⟨constant N (port/strspn.c) 444f⟩

⟨function strspn 445a⟩
```

libc/port/strstr.c

⟨libc/port/strstr.c 445c⟩≡

```
⟨libc includes 387a⟩
⟨function strstr 82b⟩
```

libc/port/strtod.c

⟨libc/port/strtod.c 445d⟩≡

```
⟨libc includes 387a⟩
#include <ctype.h>

⟨enum _anon_ (port/strtod.c) 132a⟩

static  int xcmp(char*, char*);
static  int fpcmp(char*, ulong*);
static  void    frnorm(ulong*);
static  void    divascii(char*, int*, int*, int*);
static  void    mulascii(char*, int*, int*, int*);
static  void    divby(char*, int*, int);

typedef struct  Tab Tab;
⟨struct Tab 132b⟩

⟨function strtod 128⟩

⟨function frnorm 133a⟩

⟨function fpcmp 133b⟩

⟨function _divby 133c⟩

⟨function divby 134a⟩

⟨global tab1 134b⟩
```

⟨*function* divascii 135a⟩

⟨*function* mulby 135b⟩

⟨*global* tab2 135c⟩

⟨*function* mulascii 136a⟩

⟨*function* xcmp 136b⟩
Uses Tab 132b.

libc/port/strtok.c

⟨*constant N (port/strtok.c)* 446a⟩≡ (446c)
```
#define N   256
```

⟨*function* strtok 446b⟩≡ (446c)
```
char*
strtok(char *s, char *b)
{
    static char *under_rock;
    char map[N], *os;

    memset(map, 0, N);
    while(*b)
        map[*(uchar*)b++] = 1;
    if(s == 0)
        s = under_rock;
    while(map[*(uchar*)s++])
        ;
    if(*--s == 0)
        return 0;
    os = s;
    while(map[*(uchar*)s] == 0)
        if(*s++ == 0) {
            under_rock = s-1;
            return os;
        }
    *s++ = 0;
    under_rock = s;
    return os;
}
```
Uses N-57 446a and memset() 46b.

⟨libc/port/strtok.c 446c⟩≡
⟨*libc includes* 387a⟩
⟨*constant N (port/strtok.c)* 446a⟩

⟨*function* strtok 446b⟩

libc/port/strtol.c

⟨libc/port/strtol.c 446d⟩≡
⟨*libc includes* 387a⟩
⟨*constant* LONG_MAX 124a⟩
⟨*constant* LONG_MIN 124b⟩

⟨*function* strtol 124c⟩

libc/port/strtoll.c

⟨*constant* VLONG_MAX 447a⟩≡ (448a)
```
#define VLONG_MAX   ~(1LL<<63)
```

⟨*constant* VLONG_MIN 447b⟩≡ (448a)
```
#define VLONG_MIN   (1LL<<63)
```

⟨*function* strtoll 447c⟩≡ (448a)
```
vlong
strtoll(char *nptr, char **endptr, int base)
{
    char *p;
    vlong n, nn, m;
    int c, ovfl, v, neg, ndig;

    p = nptr;
    neg = 0;
    n = 0;
    ndig = 0;
    ovfl = 0;

    /*
     * White space
     */
    for(;; p++) {
        switch(*p) {
        case ' ':
        case '\t':
        case '\n':
        case '\f':
        case '\r':
        case '\v':
            continue;
        }
        break;
    }

    /*
     * Sign
     */
    if(*p=='-' || *p=='+')
        if(*p++ == '-')
            neg = 1;

    /*
     * Base
     */
    if(base==0){
        base = 10;
        if(*p == '0') {
            base = 8;
            if(p[1]=='x' || p[1]=='X') {
                p += 2;
                base = 16;
            }
        }
    } else
    if(base==16 && *p=='0') {
        if(p[1]=='x' || p[1]=='X')
```

```
            p += 2;
    } else
    if(base<0 || 36<base)
        goto Return;

    /*
     * Non-empty sequence of digits
     */
    m = VLONG_MAX/base;
    for(;; p++,ndig++) {
        c = *p;
        v = base;
        if('0'<=c && c<='9')
            v = c - '0';
        else
        if('a'<=c && c<='z')
            v = c - 'a' + 10;
        else
        if('A'<=c && c<='Z')
            v = c - 'A' + 10;
        if(v >= base)
            break;
        if(n > m)
            ovfl = 1;
        nn = n*base + v;
        if(nn < n)
            ovfl = 1;
        n = nn;
    }

Return:
    if(ndig == 0)
        p = nptr;
    if(endptr)
        *endptr = p;
    if(ovfl){
        if(neg)
            return VLONG_MIN;
        return VLONG_MAX;
    }
    if(neg)
        return -n;
    return n;
}
```

Uses `VLONG_MAX-37` 447a and `VLONG_MIN-38` 447b.

⟨`libc/port/strtoll.c` 448a⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*constant* `VLONG_MAX` 447a⟩
  ⟨*constant* `VLONG_MIN` 447b⟩

  ⟨*function* `strtoll` 447c⟩

libc/port/strtoul.c

⟨*constant* `ULONG_MAX` 448b⟩≡ (450a)

```
#define ULONG_MAX    4294967295UL
```

⟨*function* strtoul 449⟩≡ (450a)

```
ulong
strtoul(char *nptr, char **endptr, int base)
{
    char *p;
    ulong n, nn, m;
    int c, ovfl, neg, v, ndig;

    p = nptr;
    neg = 0;
    n = 0;
    ndig = 0;
    ovfl = 0;

    /*
     * White space
     */
    for(;;p++){
        switch(*p){
        case ' ':
        case '\t':
        case '\n':
        case '\f':
        case '\r':
        case '\v':
            continue;
        }
        break;
    }

    /*
     * Sign
     */
    if(*p=='-' || *p=='+')
        if(*p++ == '-')
            neg = 1;

    /*
     * Base
     */
    if(base==0){
        if(*p != '0')
            base = 10;
        else{
            base = 8;
            if(p[1]=='x' || p[1]=='X')
                base = 16;
        }
    }
    if(base<2 || 36<base)
        goto Return;
    if(base==16 && *p=='0'){
        if(p[1]=='x' || p[1]=='X')
            if(('0' <= p[2] && p[2] <= '9')
             ||('a' <= p[2] && p[2] <= 'f')
             ||('A' <= p[2] && p[2] <= 'F'))
                p += 2;
    }
    /*
     * Non-empty sequence of digits
```

```
     */
    n = 0;
    m = ULONG_MAX/base;
    for(;; p++,ndig++){
        c = *p;
        v = base;
        if('0'<=c && c<='9')
            v = c - '0';
        else if('a'<=c && c<='z')
            v = c - 'a' + 10;
        else if('A'<=c && c<='Z')
            v = c - 'A' + 10;
        if(v >= base)
            break;
        if(n > m)
            ovfl = 1;
        nn = n*base + v;
        if(nn < n)
            ovfl = 1;
        n = nn;
    }

    Return:
    if(ndig == 0)
        p = nptr;
    if(endptr)
        *endptr = p;
    if(ovfl)
        return ULONG_MAX;
    if(neg)
        return -n;
    return n;
}
```

Uses ULONG_MAX-194 448b.

⟨libc/port/strtoul.c 450a⟩≡

```
⟨libc includes 387a⟩
⟨constant ULONG_MAX 448b⟩

⟨function strtoul 449⟩
```

libc/port/strtoull.c

⟨*constant* UVLONG_MAX 450b⟩≡ (452a)

```
#define UVLONG_MAX  (1LL<<63)
```

⟨*function* strtoull 450c⟩≡ (452a)

```
uvlong
strtoull(char *nptr, char **endptr, int base)
{
    char *p;
    uvlong n, nn, m;
    int c, ovfl, v, neg, ndig;

    p = nptr;
    neg = 0;
    n = 0;
    ndig = 0;
    ovfl = 0;
```

```
/*
 * White space
 */
for(;; p++) {
    switch(*p) {
    case ' ':
    case '\t':
    case '\n':
    case '\f':
    case '\r':
    case '\v':
        continue;
    }
    break;
}

/*
 * Sign
 */
if(*p == '-' || *p == '+')
    if(*p++ == '-')
        neg = 1;

/*
 * Base
 */
if(base == 0) {
    base = 10;
    if(*p == '0') {
        base = 8;
        if(p[1] == 'x' || p[1] == 'X'){
            p += 2;
            base = 16;
        }
    }
} else
if(base == 16 && *p == '0') {
    if(p[1] == 'x' || p[1] == 'X')
        p += 2;
} else
if(base < 0 || 36 < base)
    goto Return;

/*
 * Non-empty sequence of digits
 */
m = UVLONG_MAX/base;
for(;; p++,ndig++) {
    c = *p;
    v = base;
    if('0' <= c && c <= '9')
        v = c - '0';
    else
    if('a' <= c && c <= 'z')
        v = c - 'a' + 10;
    else
    if('A' <= c && c <= 'Z')
        v = c - 'A' + 10;
    if(v >= base)
```

```
            break;
        if(n > m)
            ovfl = 1;
        nn = n*base + v;
        if(nn < n)
            ovfl = 1;
        n = nn;
    }

Return:
    if(ndig == 0)
        p = nptr;
    if(endptr)
        *endptr = p;
    if(ovfl)
        return UVLONG_MAX;
    if(neg)
        return -n;
    return n;
}
```

Uses UVLONG_MAX-97 450b.

⟨libc/port/strtoull.c 452a⟩≡

```
⟨libc includes 387a⟩
⟨constant UVLONG_MAX 450b⟩

⟨function strtoull 450c⟩
```

libc/port/tan.c

⟨libc/port/tan.c 452b⟩≡

```
/*
    floating point tangent

    A series is used after range reduction.
    Coefficients are #4285 from Hart & Cheney. (19.74D)
 */

⟨libc includes 387a⟩
⟨global p0 (port/tan.c) 142g⟩
⟨global p1 (port/tan.c) 142h⟩
⟨global p2 (port/tan.c) 143a⟩
⟨global p3 (port/tan.c) 143b⟩
⟨global p4 143c⟩
⟨global q0 (port/tan.c) 143d⟩
⟨global q1 (port/tan.c) 143e⟩
⟨global q2 (port/tan.c) 143f⟩

⟨function tan 143g⟩
```

libc/port/tanh.c

⟨libc/port/tanh.c 452c⟩≡

```
⟨libc includes 387a⟩
⟨function tanh 148c⟩
```

libc/port/tokenize.c

⟨*global* qsep 453a⟩≡ (454c)

```
static char qsep[] = " \t\r\n";
```

Uses qsep-96 453a.

⟨*function* qtoken 453b⟩≡ (454c)

```
static char*
qtoken(char *s, char *sep)
{
    int quoting;
    char *t;

    quoting = 0;
    t = s;  /* s is output string, t is input string */
    while(*t!='\0' && (quoting || utfrune(sep, *t)==nil)){
        if(*t != '\''){
            *s++ = *t++;
            continue;
        }
        /* *t is a quote */
        if(!quoting){
            quoting = 1;
            t++;
            continue;
        }
        /* quoting and we're on a quote */
        if(t[1] != '\''){
            /* end of quoted section; absorb closing quote */
            t++;
            quoting = 0;
            continue;
        }
        /* doubled quote; fold one quote into two */
        t++;
        *s++ = *t++;
    }
    if(*s != '\0'){
        *s = '\0';
        if(t == s)
            t++;
    }
    return t;
}
```

Uses utfrune() 89b.

⟨*function* etoken 453c⟩≡ (454c)

```
static char*
etoken(char *t, char *sep)
{
    int quoting;

    /* move to end of next token */
    quoting = 0;
    while(*t!='\0' && (quoting || utfrune(sep, *t)==nil)){
        if(*t != '\''){
            t++;
            continue;
        }
        /* *t is a quote */
```

```
        if(!quoting){
            quoting = 1;
            t++;
            continue;
        }
        /* quoting and we're on a quote */
        if(t[1] != '\''){
            /* end of quoted section; absorb closing quote */
            t++;
            quoting = 0;
            continue;
        }
        /* doubled quote; fold one quote into two */
        t += 2;
    }
    return t;
}
```

Uses `utfrune()` 89b.

⟨*function* `gettokens` 454a⟩≡ (454c)

```
int
gettokens(char *s, char **args, int maxargs, char *sep)
{
    int nargs;

    for(nargs=0; nargs<maxargs; nargs++){
        while(*s!='\0' && utfrune(sep, *s)!=nil)
            *s++ = '\0';
        if(*s == '\0')
            break;
        args[nargs] = s;
        s = etoken(s, sep);
    }

    return nargs;
}
```

Uses `etoken()` 453c and `utfrune()` 89b.

⟨*function* `tokenize` 454b⟩≡ (454c)

```
int
tokenize(char *s, char **args, int maxargs)
{
    int nargs;

    for(nargs=0; nargs<maxargs; nargs++){
        while(*s!='\0' && utfrune(qsep, *s)!=nil)
            s++;
        if(*s == '\0')
            break;
        args[nargs] = s;
        s = qtoken(s, qsep);
    }

    return nargs;
}
```

Uses `qsep-96` 453a, `qtoken()` 453b, and `utfrune()` 89b.

⟨`libc/port/tokenize.c` 454c⟩≡

⟨*libc includes* 387a⟩

```
⟨global qsep 453a⟩

⟨function qtoken 453b⟩

⟨function etoken 453c⟩

⟨function gettokens 454a⟩

⟨function tokenize 454b⟩
```

libc/port/toupper.c

```
⟨libc/port/toupper.c 455a⟩≡
  #include    <ctype.h>

  ⟨function toupper 29b⟩
  ⟨function tolower 29d⟩
```

libc/port/u16.c

```
⟨global t16e 455b⟩≡                                                    (456b)
  static char t16e[] = "0123456789ABCDEF";
```

Uses t16e-193 455b.

```
⟨function dec16 455c⟩≡                                                 (456b)
  int
  dec16(uchar *out, int lim, char *in, int n)
  {
      int c, w = 0, i = 0;
      uchar *start = out;
      uchar *eout = out + lim;

      while(n-- > 0){
          c = *in++;
          if('0' <= c && c <= '9')
              c = c - '0';
          else if('a' <= c && c <= 'z')
              c = c - 'a' + 10;
          else if('A' <= c && c <= 'Z')
              c = c - 'A' + 10;
          else
              continue;
          w = (w<<4) + c;
          i++;
          if(i == 2){
              if(out + 1 > eout)
                  goto exhausted;
              *out++ = w;
              w = 0;
              i = 0;
          }
      }
  exhausted:
      return out - start;
  }
```

⟨*function* enc16 456a⟩≡ (456b)

```
int
enc16(char *out, int lim, uchar *in, int n)
{
    uint c;
    char *eout = out + lim;
    char *start = out;

    while(n-- > 0){
        c = *in++;
        if(out + 2 >= eout)
            goto exhausted;
        *out++ = t16e[c>>4];
        *out++ = t16e[c&0xf];
    }
exhausted:
    *out = 0;
    return out - start;
}
```

Uses t16e-193 455b.

⟨libc/port/u16.c 456b⟩≡

```
⟨libc includes 387a⟩
⟨global t16e 455b⟩

⟨function dec16 455c⟩

⟨function enc16 456a⟩
```

libc/port/u32.c

⟨*function* dec32 456c⟩≡ (458a)

```
int
dec32(uchar *dest, int ndest, char *src, int nsrc)
{
    char *s, *tab;
    uchar *start;
    int i, u[8];

    if(ndest+1 < (5*nsrc+7)/8)
        return -1;
    start = dest;
    tab = "23456789abcdefghijkmnpqrstuvwxyz";
    while(nsrc>=8){
        for(i=0; i<8; i++){
            s = strchr(tab,(int)src[i]);
            u[i] = s ? s-tab : 0;
        }
        *dest++ = (u[0]<<3) | (0x7 & (u[1]>>2));
        *dest++ = ((0x3 & u[1])<<6) | (u[2]<<1) | (0x1 & (u[3]>>4));
        *dest++ = ((0xf & u[3])<<4) | (0xf & (u[4]>>1));
        *dest++ = ((0x1 & u[4])<<7) | (u[5]<<2) | (0x3 & (u[6]>>3));
        *dest++ = ((0x7 & u[6])<<5) | u[7];
        src  += 8;
        nsrc -= 8;
    }
    if(nsrc > 0){
        if(nsrc == 1 || nsrc == 3 || nsrc == 6)
            return -1;
```

```
        for(i=0; i<nsrc; i++){
            s = strchr(tab,(int)src[i]);
            u[i] = s ? s-tab : 0;
        }
        *dest++ = (u[0]<<3) | (0x7 & (u[1]>>2));
        if(nsrc == 2)
            goto out;
        *dest++ = ((0x3 & u[1])<<6) | (u[2]<<1) | (0x1 & (u[3]>>4));
        if(nsrc == 4)
            goto out;
        *dest++ = ((0xf & u[3])<<4) | (0xf & (u[4]>>1));
        if(nsrc == 5)
            goto out;
        *dest++ = ((0x1 & u[4])<<7) | (u[5]<<2) | (0x3 & (u[6]>>3));
    }
out:
    return dest-start;
}
```

Uses strchr() 80c.

⟨*function* enc32 457⟩≡ (458a)

```
int
enc32(char *dest, int ndest, uchar *src, int nsrc)
{
    char *tab, *start;
    int j;

    if(ndest <= (8*nsrc+4)/5 )
        return -1;
    start = dest;
    tab = "23456789abcdefghijkmnpqrstuvwxyz";
    while(nsrc>=5){
        j = (0x1f & (src[0]>>3));
        *dest++ = tab[j];
        j = (0x1c & (src[0]<<2)) | (0x03 & (src[1]>>6));
        *dest++ = tab[j];
        j = (0x1f & (src[1]>>1));
        *dest++ = tab[j];
        j = (0x10 & (src[1]<<4)) | (0x0f & (src[2]>>4));
        *dest++ = tab[j];
        j = (0x1e & (src[2]<<1)) | (0x01 & (src[3]>>7));
        *dest++ = tab[j];
        j = (0x1f & (src[3]>>2));
        *dest++ = tab[j];
        j = (0x18 & (src[3]<<3)) | (0x07 & (src[4]>>5));
        *dest++ = tab[j];
        j = (0x1f & (src[4]));
        *dest++ = tab[j];
        src  += 5;
        nsrc -= 5;
    }
    if(nsrc){
        j = (0x1f & (src[0]>>3));
        *dest++ = tab[j];
        j = (0x1c & (src[0]<<2));
        if(nsrc == 1)
            goto out;
        j |= (0x03 & (src[1]>>6));
        *dest++ = tab[j];
        j = (0x1f & (src[1]>>1));
```

```
        if(nsrc == 2)
            goto out;
        *dest++ = tab[j];
        j = (0x10 & (src[1]<<4));
        if(nsrc == 3)
            goto out;
        j |= (0x0f & (src[2]>>4));
        *dest++ = tab[j];
        j = (0x1e & (src[2]<<1));
        if(nsrc == 4)
            goto out;
        j |= (0x01 & (src[3]>>7));
        *dest++ = tab[j];
        j = (0x1f & (src[3]>>2));
        *dest++ = tab[j];
        j = (0x18 & (src[3]<<3));
out:
        *dest++ = tab[j];
    }
    *dest = 0;
    return dest-start;
}
```

⟨libc/port/u32.c 458a⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* dec32 456c⟩

  ⟨*function* enc32 457⟩

libc/port/u64.c

⟨*enum* _anon_ (port/u64.c) 458b⟩≡ (460a)

```
enum {
    INVAL=  255
};
```

⟨*global* t64d 458c⟩≡ (460a)

```
static uchar t64d[256] = {
   INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,
   INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,
   INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,   62,INVAL,INVAL,INVAL,   63,
      52,   53,   54,   55,   56,   57,   58,   59,   60,   61,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,
   INVAL,    0,    1,    2,    3,    4,    5,    6,    7,    8,    9,   10,   11,   12,   13,   14,
      15,   16,   17,   18,   19,   20,   21,   22,   23,   24,   25,INVAL,INVAL,INVAL,INVAL,INVAL,
   INVAL,   26,   27,   28,   29,   30,   31,   32,   33,   34,   35,   36,   37,   38,   39,   40,
      41,   42,   43,   44,   45,   46,   47,   48,   49,   50,   51,INVAL,INVAL,INVAL,INVAL,INVAL,
   INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,
   INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,
   INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,
   INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,
   INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,
   INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,
   INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,
   INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL,INVAL
};
```

Uses INVAL-84 458b.

⟨*global* t64e 458d⟩≡ (460a)

```
static char t64e[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
```

Uses t64e-86 458d.

⟨*function* dec64 459a⟩≡ (460a)

```
int
dec64(uchar *out, int lim, char *in, int n)
{
    ulong b24;
    uchar *start = out;
    uchar *e = out + lim;
    int i, c;

    b24 = 0;
    i = 0;
    while(n-- > 0){

        c = t64d[*(uchar*)in++];
        if(c == INVAL)
            continue;
        switch(i){
        case 0:
            b24 = c<<18;
            break;
        case 1:
            b24 |= c<<12;
            break;
        case 2:
            b24 |= c<<6;
            break;
        case 3:
            if(out + 3 > e)
                goto exhausted;

            b24 |= c;
            *out++ = b24>>16;
            *out++ = b24>>8;
            *out++ = b24;
            i = -1;
            break;
        }
        i++;
    }
    switch(i){
    case 2:
        if(out + 1 > e)
            goto exhausted;
        *out++ = b24>>16;
        break;
    case 3:
        if(out + 2 > e)
            goto exhausted;
        *out++ = b24>>16;
        *out++ = b24>>8;
        break;
    }
exhausted:
    return out - start;
}
```

Uses `INVAL`-84 458b and `t64d`-85 458c.

⟨*function* enc64 459b⟩≡ (460a)

```
int
enc64(char *out, int lim, uchar *in, int n)
```

```
{
    int i;
    ulong b24;
    char *start = out;
    char *e = out + lim;

    for(i = n/3; i > 0; i--){
        b24 = (*in++)<<16;
        b24 |= (*in++)<<8;
        b24 |= *in++;
        if(out + 4 >= e)
            goto exhausted;
        *out++ = t64e[(b24>>18)];
        *out++ = t64e[(b24>>12)&0x3f];
        *out++ = t64e[(b24>>6)&0x3f];
        *out++ = t64e[(b24)&0x3f];
    }

    switch(n%3){
    case 2:
        b24 = (*in++)<<16;
        b24 |= (*in)<<8;
        if(out + 4 >= e)
            goto exhausted;
        *out++ = t64e[(b24>>18)];
        *out++ = t64e[(b24>>12)&0x3f];
        *out++ = t64e[(b24>>6)&0x3f];
        *out++ = '=';
        break;
    case 1:
        b24 = (*in)<<16;
        if(out + 4 >= e)
            goto exhausted;
        *out++ = t64e[(b24>>18)];
        *out++ = t64e[(b24>>12)&0x3f];
        *out++ = '=';
        *out++ = '=';
        break;
    }
exhausted:
    *out = 0;
    return out - start;
}
```

Uses `t64e-86` 458d.

⟨`libc/port/u64.c` 460a⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*enum* `_anon_ (port/u64.c)` 458b⟩

  ⟨*global* `t64d` 458c⟩
  ⟨*global* `t64e` 458d⟩

  ⟨*function* `dec64` 459a⟩

  ⟨*function* `enc64` 459b⟩

libc/port/utfecpy.c

⟨*function* `utfecpy` 460b⟩≡ (461a)

```
char*
utfecpy(char *to, char *e, char *from)
{
    char *end;

    if(to >= e)
        return to;
    end = memccpy(to, from, '\0', e - to);
    if(end == nil){
        end = e;
        while(end>to && (*--end&0xC0)==0x80)
            ;
        *end = '\0';
    }else{
        end--;
    }
    return end;
}
```
Uses memccpy() 81d.

⟨libc/port/utfecpy.c 461a⟩≡
⟨*libc includes* 387a⟩
⟨*function* utfecpy 460b⟩

libc/port/utflen.c

⟨libc/port/utflen.c 461b⟩≡
⟨*libc includes* 387a⟩
⟨*function* utflen 89a⟩

libc/port/utfnlen.c

⟨*function* utfnlen 461c⟩≡ (462a)
```
int
utfnlen(char *s, long m)
{
    int c;
    long n;
    Rune rune;
    char *es;

    es = s + m;
    for(n = 0; s < es; n++) {
        c = *(uchar*)s;
        if(c < Runeself){
            if(c == '\0')
                break;
            s++;
            continue;
        }
        if(!fullrune(s, es-s))
            break;
        s += chartorune(&rune, s);
    }
    return n;
}
```
Uses chartorune() 84d and fullrune() 436c.

⟨`libc/port/utfnlen.c` 462a⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* `utfnlen` 461c⟩

libc/port/utfrrune.c

⟨`libc/port/utfrrune.c` 462b⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* `utfrrune` 89c⟩

libc/port/utfrune.c

⟨`libc/port/utfrune.c` 462c⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* `utfrune` 89b⟩

libc/port/utfutf.c

⟨`libc/port/utfutf.c` 462d⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* `utfutf` 90⟩

A.3 libc/9sys/

libc/9sys/abort.c

⟨`libc/9sys/abort.c` 462e⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* `abort` 356d⟩

libc/9sys/access.c

⟨`libc/9sys/access.c` 462f⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* `access` 220⟩

libc/9sys/announce.c

⟨*enum* `_anon_ (9sys/announce.c)` 462g⟩≡ (464)

```
enum
{
    Maxpath=    256,
};
```

⟨*function* identtrans 463a⟩≡ (464)

```
/*
 *  perform the identity translation (in case we can't reach cs)
 */
static int
identtrans(char *netdir, char *addr, char *naddr, int na, char *file, int nf)
{
    char proto[Maxpath];
    char *p;

    USED(nf);

    /* parse the protocol */
    strncpy(proto, addr, sizeof(proto));
    proto[sizeof(proto)-1] = 0;
    p = strchr(proto, '!');
    if(p)
        *p++ = 0;

    snprint(file, nf, "%s/%s/clone", netdir, proto);
    strncpy(naddr, p, na);
    naddr[na-1] = 0;

    return 1;
}
```

Uses `Maxpath-230` 462g, `snprint()` 535b, `strchr()` 80c, and `strncpy()` 443e.

⟨*function* nettrans 463b⟩≡ (464)

```
/*
 *  call up the connection server and get a translation
 */
static int
nettrans(char *addr, char *naddr, int na, char *file, int nf)
{
    int i, fd;
    char buf[Maxpath];
    char netdir[Maxpath];
    char *p, *p2;
    long n;

    /*
     *  parse, get network directory
     */
    p = strchr(addr, '!');
    if(p == 0){
        werrstr("bad dial string: %s", addr);
        return -1;
    }
    if(*addr != '/'){
        strncpy(netdir, "/net", sizeof(netdir));
        netdir[sizeof(netdir) - 1] = 0;
    } else {
        for(p2 = p; *p2 != '/'; p2--)
            ;
        i = p2 - addr;
        if(i == 0 || i >= sizeof(netdir)){
            werrstr("bad dial string: %s", addr);
            return -1;
        }
        strncpy(netdir, addr, i);
```

```
        netdir[i] = 0;
        addr = p2 + 1;
    }

    /*
     *  ask the connection server
     */
    snprint(buf, sizeof(buf), "%s/cs", netdir);
    fd = open(buf, ORDWR);
    if(fd < 0)
        return identtrans(netdir, addr, naddr, na, file, nf);
    if(write(fd, addr, strlen(addr)) < 0){
        close(fd);
        return -1;
    }
    seek(fd, 0, 0);
    n = read(fd, buf, sizeof(buf)-1);
    close(fd);
    if(n <= 0)
        return -1;
    buf[n] = 0;

    /*
     *  parse the reply
     */
    p = strchr(buf, ' ');
    if(p == 0)
        return -1;
    *p++ = 0;
    strncpy(naddr, p, na);
    naddr[na-1] = 0;

    if(buf[0] == '/'){
        p = strchr(buf+1, '/');
        if(p == nil)
            p = buf;
        else
            p++;
    }
    snprint(file, nf, "%s/%s", netdir, p);
    return 0;
}
```

Uses `Maxpath-230` 462g, `close()`, `identtrans()` 463a, `open()`, `read()` 192a, `seek()`, `snprint()` 535b, `strchr()` 80c, `strlen()` 80a, `strncpy()` 443e, `werrstr()` 234b, and `write()` 192b.

⟨`libc/9sys/announce.c` 464⟩≡

```
⟨libc includes 387a⟩
#include <ctype.h>

static int  nettrans(char*, char*, int na, char*, int);

⟨enum _anon_ (9sys/announce.c) 462g⟩

⟨function announce 344⟩

⟨function listen 346a⟩

⟨function accept 346b⟩

⟨function reject 347a⟩
```

⟨*function* identtrans 463a⟩

⟨*function* nettrans 463b⟩

libc/9sys/convD2M.c

⟨*function* sizeD2M 465a⟩≡ (465b)
```
uint
sizeD2M(Dir *d)
{
    char *sv[4];
    int i, ns;

    sv[0] = d->name;
    sv[1] = d->uid;
    sv[2] = d->gid;
    sv[3] = d->muid;

    ns = 0;
    for(i = 0; i < 4; i++)
        if(sv[i])
            ns += strlen(sv[i]);

    return STATFIXLEN + ns;
}
```
Uses strlen() 80a.

⟨libc/9sys/convD2M.c 465b⟩≡
⟨*libc includes* 387a⟩
```
#include    <fcall.h>
```

⟨*function* sizeD2M 465a⟩

⟨*function* convD2M 281⟩

libc/9sys/convM2D.c

⟨*function* statcheck 465c⟩≡ (466b)
```
int
statcheck(uchar *buf, uint nbuf)
{
    uchar *ebuf;
    int i;

    ebuf = buf + nbuf;

    if(nbuf < STATFIXLEN || nbuf != BIT16SZ + GBIT16(buf))
        return -1;

    buf += STATFIXLEN - 4 * BIT16SZ;

    for(i = 0; i < 4; i++){
        if(buf + BIT16SZ > ebuf)
            return -1;
        buf += BIT16SZ + GBIT16(buf);
    }
```

```
    if(buf != ebuf)
        return -1;

    return 0;
}
```

⟨*global* `nullstring` 466a⟩≡ (466b)

```
static char nullstring[] = "";
```

Uses `nullstring-259` 466a.

⟨`libc/9sys/convM2D.c` 466b⟩≡

```
⟨libc includes 387a⟩
#include    <fcall.h>

⟨function statcheck 465c⟩

⟨global nullstring 466a⟩

⟨function convM2D 280⟩
```

libc/9sys/convM2S.c

⟨`libc/9sys/convM2S.c` 466c⟩≡

```
⟨libc includes 387a⟩
#include    <fcall.h>

⟨function gstring 272a⟩

⟨function gqid 272b⟩

⟨function convM2S 267⟩
```

libc/9sys/convS2M.c

⟨`libc/9sys/convS2M.c` 466d⟩≡

```
⟨libc includes 387a⟩
#include    <fcall.h>

⟨function pstring 276a⟩

⟨function pqid 276b⟩

⟨function stringsz 279a⟩

⟨function sizeS2M 277⟩

⟨function convS2M 273⟩
```

libc/9sys/cputime.c

⟨`libc/9sys/cputime.c` 466e⟩≡

```
⟨libc includes 387a⟩
⟨constant HZ 363a⟩

⟨function cputime 363b⟩
```

libc/9sys/ctime.c

⟨*global* dmsize 467a⟩≡ (470)
```
static  char    dmsize[12] =
{
    31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
};
```

⟨*constant* TZSIZE 467b⟩≡ (470)
```
#define TZSIZE  150
```

⟨*global* timezone 467c⟩≡ (470)
```
static
struct
{
    char    stname[4];
    char    dlname[4];
    long    stdiff;
    long    dldiff;
    long    dlpairs[TZSIZE];
} timezone;
```
Uses TZSIZE-235 467b and __anon_struct_31 467c.

⟨*function* ctime 467d⟩≡ (470)
```
char*
ctime(long t)
{
    return asctime(localtime(t));
}
```
Uses asctime() 225b and localtime() 225a.

⟨*function* gmtime 467e⟩≡ (470)
```
Tm*
gmtime(long tim)
{
    int d0, d1;
    long hms, day;
    static Tm xtime;

    /*
     * break initial number into days
     */
    hms = (ulong)tim % 86400L;
    day = (ulong)tim / 86400L;
    if(hms < 0) {
        hms += 86400L;
        day -= 1;
    }

    /*
     * generate hours:minutes:seconds
     */
    xtime.sec = hms % 60;
    d1 = hms / 60;
    xtime.min = d1 % 60;
    d1 /= 60;
    xtime.hour = d1;

    /*
     * day is the day number.
```

```
     * generate day of the week.
     * The addend is 4 mod 7 (1/1/1970 was Thursday)
     */

    xtime.wday = (day + 7340036L) % 7;

    /*
     * year number
     */
    if(day >= 0)
        for(d1 = 1970; day >= dysize(d1); d1++)
            day -= dysize(d1);
    else
        for (d1 = 1970; day < 0; d1--)
            day += dysize(d1-1);
    xtime.year = d1-1900;
    xtime.yday = d0 = day;

    /*
     * generate month
     */

    if(dysize(d1) == 366)
        dmsize[1] = 29;
    for(d1 = 0; d0 >= dmsize[d1]; d1++)
        d0 -= dmsize[d1];
    dmsize[1] = 28;
    xtime.mday = d0 + 1;
    xtime.mon = d1;
    strcpy(xtime.zone, "GMT");
    return &xtime;
}
```

Uses `dmsize-234` 467a, `dysize()` 468a, and `strcpy()` 81c.

⟨*function* `dysize` 468a⟩≡ (470)

```
static int
dysize(int y)
{

    if(y%4 == 0 && (y%100 != 0 || y%400 == 0))
        return 366;
    return 365;
}
```

⟨*function* `ct_numb` 468b⟩≡ (470)

```
static
void
ct_numb(char *cp, int n)
{

    cp[0] = ' ';
    if(n >= 10)
        cp[0] = (n/10)%10 + '0';
    cp[1] = n%10 + '0';
}
```

⟨*function* `readtimezone` 468c⟩≡ (470)

```
static
void
readtimezone(void)
```

```
{
    char buf[TZSIZE*11+30], *p;
    int i;

    memset(buf, 0, sizeof(buf));
    i = open("/env/timezone", 0);
    if(i < 0)
        goto error;
    if(read(i, buf, sizeof(buf)) >= sizeof(buf)){
        close(i);
        goto error;
    }
    close(i);
    p = buf;
    if(rd_name(&p, timezone.stname))
        goto error;
    if(rd_long(&p, &timezone.stdiff))
        goto error;
    if(rd_name(&p, timezone.dlname))
        goto error;
    if(rd_long(&p, &timezone.dldiff))
        goto error;
    for(i=0; i<TZSIZE; i++) {
        if(rd_long(&p, &timezone.dlpairs[i]))
            goto error;
        if(timezone.dlpairs[i] == 0)
            return;
    }

error:
    timezone.stdiff = 0;
    strcpy(timezone.stname, "GMT");
    timezone.dlpairs[0] = 0;
}
```

Uses TZSIZE-235 467b, close(), memset() 46b, open(), read() 192a, strcpy() 81c, and timezone-236 467c.

⟨*function* rd_name 469a⟩≡ (470)

```
static int
rd_name(char **f, char *p)
{
    int c, i;

    for(;;) {
        c = *(*f)++;
        if(c != ' ' && c != '\n')
            break;
    }
    for(i=0; i<3; i++) {
        if(c == ' ' || c == '\n')
            return 1;
        *p++ = c;
        c = *(*f)++;
    }
    if(c != ' ' && c != '\n')
        return 1;
    *p = 0;
    return 0;
}
```

⟨*function* rd_long 469b⟩≡ (470)

```
static int
rd_long(char **f, long *p)
{
    int c, s;
    long l;

    s = 0;
    for(;;) {
        c = *(*f)++;
        if(c == '-') {
            s++;
            continue;
        }
        if(c != ' ' && c != '\n')
            break;
    }
    if(c == 0) {
        *p = 0;
        return 0;
    }
    l = 0;
    for(;;) {
        if(c == ' ' || c == '\n')
            break;
        if(c < '0' || c > '9')
            return 1;
        l = l*10 + c-'0';
        c = *(*f)++;
    }
    if(s)
        l = -l;
    *p = l;
    return 0;
}
```

⟨libc/9sys/ctime.c 470⟩≡

```
/*
 * This routine converts time as follows.
 * The epoch is 0000 Jan 1 1970 GMT.
 * The argument time is in seconds since then.
 * The localtime(t) entry returns a pointer to an array
 * containing
 *
 *  seconds (0-59)
 *  minutes (0-59)
 *  hours (0-23)
 *  day of month (1-31)
 *  month (0-11)
 *  year-1970
 *  weekday (0-6, Sun is 0)
 *  day of the year
 *  daylight savings flag
 *
 * The routine gets the daylight savings time from the environment.
 *
 * asctime(tvec))
 * where tvec is produced by localtime
 * returns a ptr to a character string
 * that has the ascii time in the form
 *
```

```
 *                               \\
 *  Thu Jan 01 00:00:00 GMT 1970n0
 *  012345678901234567890123456789
 *  0         1         2
 *
 * ctime(t) just calls localtime, then asctime.
 */

⟨libc includes 387a⟩
⟨global dmsize 467a⟩

/*
 * The following table is used for 1974 and 1975 and
 * gives the day number of the first day after the Sunday of the
 * change.
 */

static  int dysize(int);
static  void    ct_numb(char*, int);

⟨constant TZSIZE 467b⟩
static  void    readtimezone(void);
static  int rd_name(char**, char*);
static  int rd_long(char**, long*);
⟨global timezone 467c⟩

⟨function ctime 467d⟩

⟨function localtime 225a⟩

⟨function gmtime 467e⟩

⟨function asctime 225b⟩

⟨function dysize 468a⟩

⟨function ct_numb 468b⟩

⟨function readtimezone 468c⟩

⟨function rd_name 469a⟩

⟨function rd_long 469b⟩
```

libc/9sys/dial.c

⟨*enum* _anon_ (9sys/dial.c) 471⟩≡ (476c)

```
enum
{
    Maxstring   = 128,
    Maxpath     = 256,

    Maxcsreply  = 64*80,    /* this is probably overly generous */
    /*
     * this should be a plausible slight overestimate for non-interactive
     * use even if it's ridiculously long for interactive use.
     */
    Maxconnms   = 2*60*1000,    /* 2 minutes */
};
```

⟨*struct* DS *(9sys/dial.c)* 472a⟩≡ (476c)
```
struct DS {
    /* dist string */
    char    buf[Maxstring];
    char    *netdir;
    char    *proto;
    char    *rem;

    /* other args */
    char    *local;
    char    *dir;
    int *cfdp;
};
```
Uses Maxstring-226 471.

⟨*struct* Conn 472b⟩≡ (476c)
```
/*
 * malloc these; they need to be writable by this proc & all children.
 * the stack is private to each proc, and static allocation in the data
 * segment would not permit concurrent dials within a multi-process program.
 */
struct Conn {
    int pid;
    int dead;

    int dfd;
    int cfd;
    char    dir[NETPATHLEN+1];
    char    err[ERRMAX];
};
```

⟨*struct* Dest 472c⟩≡ (476c)
```
struct Dest {
    Conn    *conn;          /* allocated array */
    Conn    *connend;
    int nkid;

    long    oalarm;
    int naddrs;

    QLock   winlck;
    int winner;         /* index into conn[] */

    char    *nextaddr;
    char    addrlist[Maxcsreply];
};
```
Uses Maxcsreply-228 471.

⟨*function* connsalloc 472d⟩≡ (476c)
```
static int
connsalloc(Dest *dp, int addrs)
{
    Conn *conn;

    free(dp->conn);
    dp->connend = nil;
    assert(addrs > 0);

    dp->conn = mallocz(addrs * sizeof *dp->conn, 1);
    if(dp->conn == nil)
```

```
        return -1;
    dp->connend = dp->conn + addrs;
    for(conn = dp->conn; conn < dp->connend; conn++)
        conn->cfd = conn->dfd = -1;
    return 0;
}
```

Uses free() 63i and mallocz() 67a.

⟨*function* freedest 473a⟩≡ (476c)

```
static void
freedest(Dest *dp)
{
    long oalarm;

    if (dp == nil)
        return;
    oalarm = dp->oalarm;
    free(dp->conn);
    free(dp);
    if (oalarm >= 0)
        alarm(oalarm);
}
```

Uses alarm() and free() 63i.

⟨*function* closeopenfd 473b⟩≡ (476c)

```
static void
closeopenfd(int *fdp)
{
    if (*fdp >= 0) {
        close(*fdp);
        *fdp = -1;
    }
}
```

Uses close().

⟨*function* notedeath 473c⟩≡ (476c)

```
static void
notedeath(Dest *dp, char *exitsts)
{
    int i, n, pid;
    char *fields[5];            /* pid + 3 times + error */
    Conn *conn;

    for (i = 0; i < nelem(fields); i++)
        fields[i] = "";
    n = tokenize(exitsts, fields, nelem(fields));
    if (n < 4)
        return;
    pid = atoi(fields[0]);
    if (pid <= 0)
        return;
    for (conn = dp->conn; conn < dp->connend; conn++)
        if (conn->pid == pid && !conn->dead) {  /* it's one we know? */
            if (conn - dp->conn != dp->winner) {
                closeopenfd(&conn->dfd);
                closeopenfd(&conn->cfd);
            }
            strncpy(conn->err, fields[4], sizeof conn->err - 1);
            conn->err[sizeof conn->err - 1] = '\0';
```

```
            conn->dead = 1;
            return;
        }
    /* not a proc that we forked */
}
```

Uses atoi() 123b, closeopenfd() 473b, strncpy() 443e, and tokenize() 454b.

⟨*function* outstandingprocs 474a⟩≡ (476c)

```
static int
outstandingprocs(Dest *dp)
{
    Conn *conn;

    for (conn = dp->conn; conn < dp->connend; conn++)
        if (!conn->dead)
            return 1;
    return 0;
}
```

⟨*function* reap 474b⟩≡ (476c)

```
static int
reap(Dest *dp)
{
    char exitsts[2*ERRMAX];

    if (outstandingprocs(dp) && await(exitsts, sizeof exitsts) >= 0) {
        notedeath(dp, exitsts);
        return 0;
    }
    return -1;
}
```

Uses await(), notedeath() 473c, and outstandingprocs() 474a.

⟨*function* fillinds 474c⟩≡ (476c)

```
static int
fillinds(DS *ds, Dest *dp)
{
    Conn *conn;

    if (dp->winner < 0)
        return -1;
    conn = &dp->conn[dp->winner];
    if (ds->cfdp)
        *ds->cfdp = conn->cfd;
    if (ds->dir) {
        strncpy(ds->dir, conn->dir, NETPATHLEN);
        ds->dir[NETPATHLEN-1] = '\0';
    }
    return conn->dfd;
}
```

Uses strncpy() 443e.

⟨*function* connectwait 474d⟩≡ (476c)

```
static int
connectwait(Dest *dp, char *besterr)
{
    Conn *conn;

    /* wait for a winner or all attempts to time out */
```

```
    while (dp->winner < 0 && reap(dp) >= 0)
        ;

    /* kill all of our still-live kids & reap them */
    for (conn = dp->conn; conn < dp->connend; conn++)
        if (!conn->dead)
            postnote(PNPROC, conn->pid, "alarm");
    while (reap(dp) >= 0)
        ;

    /* rummage about and report some error string */
    for (conn = dp->conn; conn < dp->connend; conn++)
        if (conn - dp->conn != dp->winner && conn->dead &&
            conn->err[0]) {
            strncpy(besterr, conn->err, ERRMAX-1);
            besterr[ERRMAX-1] = '\0';
            break;
        }
    return dp->winner;
}
```

Uses postnote() 258c, reap() 474b, and strncpy() 443e.

⟨*function* parsecs 475a⟩≡ (476c)

```
static int
parsecs(Dest *dp, char **clonep, char **destp)
{
    char *dest, *p;

    dest = strchr(dp->nextaddr, ' ');
    if(dest == nil) {
        p = strchr(dp->nextaddr, '\n');
        if(p)
            *p = '\0';
        werrstr("malformed clone cmd from cs ‘%s’", dp->nextaddr);
        if(p)
            *p = '\n';
        return -1;
    }
    *dest++ = '\0';
    p = strchr(dest, '\n');
    if(p == nil)
        return -1;
    *p++ = '\0';
    *clonep = dp->nextaddr;
    *destp = dest;
    dp->nextaddr = p;       /* advance to next line */
    return 0;
}
```

Uses strchr() 80c and werrstr() 234b.

⟨*function* pickuperr 475b⟩≡ (476c)

```
static void
pickuperr(char *besterr, char *err)
{
    err[0] = '\0';
    errstr(err, ERRMAX);
    if(strstr(err, "does not exist") == 0)
        strcpy(besterr, err);
}
```

Uses errstr(), strcpy() 81c, and strstr() 82b.

⟨*function* catcher 476a⟩≡ (476c)

```
static int
catcher(void *, char *s)
{
    return strstr(s, "alarm") != nil;
}
```

Uses strstr() 82b.

⟨*function* dialmulti 476b⟩≡ (476c)

```
/*
 * try all addresses in parallel and take the first one that answers;
 * this helps when systems have ip v4 and v6 addresses but are
 * only reachable from here on one (or some) of them.
 */
static int
dialmulti(DS *ds, Dest *dp)
{
    int rv, kid, kidme;
    char *clone, *dest;
    char besterr[ERRMAX];

    dp->winner = -1;
    dp->nkid = 0;
    while(dp->winner < 0 && *dp->nextaddr != '\0' &&
        parsecs(dp, &clone, &dest) >= 0) {
        kidme = dp->nkid++;     /* make private copy on stack */
        kid = rfork(RFPROC|RFMEM);  /* spin off a call attempt */
        if (kid < 0)
            --dp->nkid;
        else if (kid == 0) {
            char err[ERRMAX];

            /* only in kid, to avoid atnotify callbacks in parent */
            atnotify(catcher, 1);

            *besterr = '\0';
            rv = call(clone, dest, ds, dp, &dp->conn[kidme]);
            if(rv < 0)
                pickuperr(besterr, err);
            _exits(besterr);    /* avoid atexit callbacks */
        }
    }
    *besterr = '\0';
    rv = connectwait(dp, besterr);
    if(rv < 0)
        werrstr("%s", (*besterr? besterr: "unknown error"));
    return rv;
}
```

Uses _exits(), atnotify() 259, catcher() 476a, connectwait() 474d, parsecs() 475a, pickuperr() 475b, rfork(), and werrstr() 234b.

⟨libc/9sys/dial.c 476c⟩≡

```
/*
 * dial - connect to a service (parallel version)
 */
⟨libc includes 387a⟩
typedef struct Conn Conn;
typedef struct Dest Dest;
typedef struct DS DS;
```

⟨*enum* _anon_ (9sys/dial.c) 471⟩

⟨*struct* DS *(9sys/dial.c)* 472a⟩

⟨*struct* Conn 472b⟩
⟨*struct* Dest 472c⟩

```
static int  call(char*, char*, DS*, Dest*, Conn*);
static int  csdial(DS*);
static void _dial_string_parse(char*, DS*);
```

⟨*function* dialimpl 348b⟩

⟨*global* _dial 348a⟩

⟨*function* dial 347b⟩

⟨*function* connsalloc 472d⟩

⟨*function* freedest 473a⟩

⟨*function* closeopenfd 473b⟩

⟨*function* notedeath 473c⟩

⟨*function* outstandingprocs 474a⟩

⟨*function* reap 474b⟩

⟨*function* fillinds 474c⟩

⟨*function* connectwait 474d⟩

⟨*function* parsecs 475a⟩

⟨*function* pickuperr 475b⟩

⟨*function* catcher 476a⟩

⟨*function* dialmulti 476b⟩

⟨*function* csdial *(9sys/dial.c)* 348c⟩

⟨*function* call *(9sys/dial.c)* 350⟩

⟨*function* _dial_string_parse *(9sys/dial.c)* 352⟩

Uses Conn 472b and Dest 472c.

libc/9sys/dirfstat.c

⟨libc/9sys/dirfstat.c 477⟩≡

⟨*libc includes* 387a⟩

```
#include <fcall.h>
```

⟨*enum* _anon_ (9sys/dirfstat.c) 215a⟩

⟨*function* dirfstat 215b⟩

libc/9sys/dirfwstat.c

⟨libc/9sys/dirfwstat.c 478a⟩≡

```
⟨libc includes 387a⟩
#include <fcall.h>

⟨function dirfwstat 215c⟩
```

libc/9sys/dirmodefmt.c

⟨global modes 478b⟩≡ (479a)

```
static char *modes[] =
{
    "---",
    "--x",
    "-w-",
    "-wx",
    "r--",
    "r-x",
    "rw-",
    "rwx",
};
```

⟨function rwx 478c⟩≡ (479a)

```
static void
rwx(long m, char *s)
{
    strncpy(s, modes[m], 3);
}
```

Uses modes-247 478b and strncpy() 443e.

⟨function dirmodefmt 478d⟩≡ (479a)

```
int
dirmodefmt(Fmt *f)
{
    static char buf[16];
    ulong m;

    m = va_arg(f->args, ulong);

    if(m & DMDIR)
        buf[0]='d';
    else if(m & DMAPPEND)
        buf[0]='a';
    else if(m & DMAUTH)
        buf[0]='A';
    else
        buf[0]='-';
    if(m & DMEXCL)
        buf[1]='l';
    else
        buf[1]='-';
    rwx((m>>6)&7, buf+2);
    rwx((m>>3)&7, buf+5);
    rwx((m>>0)&7, buf+8);
    buf[11] = 0;
    return fmtstrcpy(f, buf);
}
```

Uses fmtstrcpy() 505a and rwx() 478c.

```
⟨libc/9sys/dirmodefmt.c 479a⟩≡
  ⟨libc includes 387a⟩
  #include <fcall.h>

  ⟨global modes 478b⟩

  ⟨function rwx 478c⟩

  ⟨function dirmodefmt 478d⟩
```

libc/9sys/dirread.c

```
⟨function dirpackage 479b⟩≡                                                    (480c)
  static
  long
  dirpackage(uchar *buf, long ts, Dir **d)
  {
      char *s;
      long ss, i, n, nn, m;

      *d = nil;
      if(ts <= 0)
          return 0;

      /*
       * first find number of all stats, check they look like stats, & size all associated strings
       */
      ss = 0;
      n = 0;
      for(i = 0; i < ts; i += m){
          m = BIT16SZ + GBIT16(&buf[i]);
          if(statcheck(&buf[i], m) < 0)
              break;
          ss += m;
          n++;
      }

      if(i != ts)
          return -1;

      *d = malloc(n * sizeof(Dir) + ss);
      if(*d == nil)
          return -1;

      /*
       * then convert all buffers
       */
      s = (char*)*d + n * sizeof(Dir);
      nn = 0;
      for(i = 0; i < ts; i += m){
          m = BIT16SZ + GBIT16((uchar*)&buf[i]);
          if(nn >= n || convM2D(&buf[i], m, *d + nn, s) != m){
              free(*d);
              *d = nil;
              return -1;
          }
          nn++;
          s += m;
      }
```

```
    return nn;
}
```

Uses convM2D() 280, free() 63i, malloc() 63g, and statcheck() 465c.

⟨*function* dirread 480a⟩≡ (480c)

```
long
dirread(int fd, Dir **d)
{
    uchar *buf;
    long ts;

    buf = malloc(DIRMAX);
    if(buf == nil)
        return -1;
    ts = read(fd, buf, DIRMAX);
    if(ts >= 0)
        ts = dirpackage(buf, ts, d);
    free(buf);
    return ts;
}
```

Uses dirpackage() 479b, free() 63i, malloc() 63g, and read() 192a.

⟨*function* dirreadall 480b⟩≡ (480c)

```
long
dirreadall(int fd, Dir **d)
{
    uchar *buf, *nbuf;
    long n, ts;

    buf = nil;
    ts = 0;
    for(;;){
        nbuf = realloc(buf, ts+DIRMAX);
        if(nbuf == nil){
            free(buf);
            return -1;
        }
        buf = nbuf;
        n = read(fd, buf+ts, DIRMAX);
        if(n <= 0)
            break;
        ts += n;
    }
    if(ts >= 0)
        ts = dirpackage(buf, ts, d);
    free(buf);
    if(ts == 0 && n < 0)
        return -1;
    return ts;
}
```

Uses dirpackage() 479b, free() 63i, read() 192a, and realloc() 67b.

⟨libc/9sys/dirread.c 480c⟩≡

```
⟨libc includes 387a⟩
#include <fcall.h>

⟨function dirpackage 479b⟩
```

⟨*function* dirread 480a⟩

⟨*function* dirreadall 480b⟩

libc/9sys/dirstat.c

⟨libc/9sys/dirstat.c 481a⟩≡
```
⟨libc includes 387a⟩
#include <fcall.h>

⟨enum _anon_ (9sys/dirstat.c) 216a⟩

⟨function dirstat 216b⟩
```

libc/9sys/dirwstat.c

⟨libc/9sys/dirwstat.c 481b⟩≡
```
⟨libc includes 387a⟩
#include <fcall.h>

⟨function dirwstat 216c⟩
```

libc/9sys/fcallfmt.c

⟨*function* dirfmt 481c⟩≡ (482c)
```
int
dirfmt(Fmt *fmt)
{
    char buf[160];

    fdirconv(buf, buf+sizeof buf, va_arg(fmt->args, Dir*));
    return fmtstrcpy(fmt, buf);
}
```
Uses fdirconv() 481d and fmtstrcpy() 505a.

⟨*function* fdirconv 481d⟩≡ (482c)
```
static void
fdirconv(char *buf, char *e, Dir *d)
{
    char tmp[16];

    seprint(buf, e, "'%s' '%s' '%s' '%s' "
        "q " QIDFMT " m %#luo "
        "at %ld mt %ld l %lld "
        "t %d d %d",
            d->name, d->uid, d->gid, d->muid,
            d->qid.path, d->qid.vers, qidtype(tmp, d->qid.type), d->mode,
            d->atime, d->mtime, d->length,
            d->type, d->dev);
}
```
Uses qidtype() 265 and seprint() 534c.

⟨*constant* DUMPL 482a⟩≡ (482c)

```
/*
 * dump out count (or DUMPL, if count is bigger) bytes from
 * buf to ans, as a string if they are all printable,
 * else as a series of hex bytes
 */
#define DUMPL 64
```

⟨*function* dumpsome 482b⟩≡ (482c)

```
static uint
dumpsome(char *ans, char *e, char *buf, long count)
{
    int i, printable;
    char *p;

    if(buf == nil){
        seprint(ans, e, "<no data>");
        return strlen(ans);
    }
    printable = 1;
    if(count > DUMPL)
        count = DUMPL;
    for(i=0; i<count && printable; i++)
        if((buf[i]<32 && buf[i] !='\n' && buf[i] !='\t') || (uchar)buf[i]>127)
            printable = 0;
    p = ans;
    *p++ = '\'';
    if(printable){
        if(count > e-p-2)
            count = e-p-2;
        memmove(p, buf, count);
        p += count;
    }else{
        if(2*count > e-p-2)
            count = (e-p-2)/2;
        for(i=0; i<count; i++){
            if(i>0 && i%4==0)
                *p++ = ' ';
            sprint(p, "%2.2ux", (uchar)buf[i]);
            p += 2;
        }
    }
    *p++ = '\'';
    *p = 0;
    return p - ans;
}
```

Uses `DUMPL-233` 482a, `memmove()` 48, `seprint()` 534c, `sprint()` 75b, and `strlen()` 80a.

⟨`libc/9sys/fcallfmt.c` 482c⟩≡

```
⟨libc includes 387a⟩
#include <fcall.h>

static uint dumpsome(char*, char*, char*, long);
static void fdirconv(char*, char*, Dir*);
static char *qidtype(char*, uchar);

⟨constant QIDFMT 263b⟩

⟨function fcallfmt 263c⟩
```

⟨*function* qidtype 265⟩

⟨*function* dirfmt 481c⟩

⟨*function* fdirconv 481d⟩

⟨*constant* DUMPL 482a⟩

⟨*function* dumpsome 482b⟩

libc/9sys/fork.c

⟨libc/9sys/fork.c 483a⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* fork 231a⟩

libc/9sys/getenv.c

⟨libc/9sys/getenv.c 483b⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* getenv 257a⟩

libc/9sys/getnetconninfo.c

⟨*global* unknown 483c⟩≡ (485c)

```
static char *unknown = "???";
```

Uses unknown-263 483c.

⟨*function* getendpoint 483d⟩≡ (485c)

```
static void
getendpoint(char *dir, char *file, char **sysp, char **servp)
{
    int fd, n;
    char buf[128];
    char *sys, *serv;

    sys = serv = 0;

    snprint(buf, sizeof buf, "%s/%s", dir, file);
    fd = open(buf, OREAD);
    if(fd >= 0){
        n = read(fd, buf, sizeof(buf)-1);
        if(n>0){
            buf[n-1] = 0;
            serv = strchr(buf, '!');
            if(serv){
                *serv++ = 0;
                serv = strdup(serv);
            }
            sys = strdup(buf);
        }
        close(fd);
    }
    if(serv == 0)
        serv = unknown;
    if(sys == 0)
        sys = unknown;
```

```
    *servp = serv;
    *sysp = sys;
}
```

Uses close(), open(), read() 192a, snprint() 535b, strchr() 80c, strdup() 82a, and unknown-263 483c.

⟨*function* getnetconninfo 484⟩≡ (485c)

```
NetConnInfo*
getnetconninfo(char *dir, int fd)
{
    NetConnInfo *nci;
    char *cp;
    Dir *d;
    char spec[10];
    char path[128];
    char netname[128], *p;

    /* get a directory address via fd */
    if(dir == nil || *dir == 0){
        if(fd2path(fd, path, sizeof(path)) < 0)
            return nil;
        cp = strrchr(path, '/');
        if(cp == nil)
            return nil;
        *cp = 0;
        dir = path;
    }

    nci = mallocz(sizeof *nci, 1);
    if(nci == nil)
        return nil;

    /* copy connection directory */
    nci->dir = strdup(dir);
    if(nci->dir == nil)
        goto err;

    /* get netroot */
    nci->root = strdup(dir);
    if(nci->root == nil)
        goto err;
    cp = strchr(nci->root+1, '/');
    if(cp == nil)
        goto err;
    *cp = 0;

    /* figure out bind spec */
    d = dirstat(nci->dir);
    if(d != nil){
        sprint(spec, "#%C%d", d->type, d->dev);
        nci->spec = strdup(spec);
    }
    if(nci->spec == nil)
        nci->spec = unknown;
    free(d);

    /* get the two end points */
    getendpoint(nci->dir, "local", &nci->lsys, &nci->lserv);
    if(nci->lsys == nil || nci->lserv == nil)
        goto err;
    getendpoint(nci->dir, "remote", &nci->rsys, &nci->rserv);
```

```
    if(nci->rsys == nil || nci->rserv == nil)
        goto err;

    strecpy(netname, netname+sizeof netname, nci->dir);
    if((p = strrchr(netname, '/')) != nil)
        *p = 0;
    if(strncmp(netname, "/net/", 5) == 0)
        memmove(netname, netname+5, strlen(netname+5)+1);
    nci->laddr = smprint("%s!%s!%s", netname, nci->lsys, nci->lserv);
    nci->raddr = smprint("%s!%s!%s", netname, nci->rsys, nci->rserv);
    if(nci->laddr == nil || nci->raddr == nil)
        goto err;
    return nci;
err:
    freenetconninfo(nci);
    return nil;
}
```

Uses `dirstat()` 216b, `fd2path()`, `free()` 63i, `freenetconninfo()` 485b, `getendpoint()` 483d, `mallocz()` 67a, `memmove()` 48, `smprint()` 534e, `sprint()` 75b, `strchr()` 80c, `strdup()` 82a, `strecpy()` 442c, `strlen()` 80a, `strncmp()` 443c, `strrchr()` 80d, and `unknown-263` 483c.

⟨*function* `xfree` 485a⟩≡ (485c)

```
static void
xfree(char *x)
{
    if(x == nil || x == unknown)
        return;
    free(x);
}
```

Uses `free()` 63i and `unknown-263` 483c.

⟨*function* `freenetconninfo` 485b⟩≡ (485c)

```
void
freenetconninfo(NetConnInfo *nci)
{
    if(nci == nil)
        return;
    xfree(nci->root);
    xfree(nci->dir);
    xfree(nci->spec);
    xfree(nci->lsys);
    xfree(nci->lserv);
    xfree(nci->rsys);
    xfree(nci->rserv);
    xfree(nci->laddr);
    xfree(nci->raddr);
    free(nci);
}
```

Uses `free()` 63i and `xfree()` 485a.

⟨`libc/9sys/getnetconninfo.c` 485c⟩≡

```
⟨libc includes 387a⟩
⟨global unknown 483c⟩

⟨function getendpoint 483d⟩

⟨function getnetconninfo 484⟩

⟨function xfree 485a⟩

⟨function freenetconninfo 485b⟩
```

libc/9sys/getpid.c

⟨libc/9sys/getpid.c 486a⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* getpid 232a⟩

libc/9sys/getppid.c

⟨libc/9sys/getppid.c 486b⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* getppid 232b⟩

libc/9sys/getwd.c

⟨libc/9sys/getwd.c 486c⟩≡
  ⟨*libc includes* 387a⟩
```
static char *nsgetwd(char*, int);
```
  ⟨*function* getwd 217a⟩

libc/9sys/iounit.c

⟨*function* iounit 486d⟩≡ (486e)
```
/*
 * Format:
  3 r  M    4 (0000000000457def 11 00)   8192      512 /rc/lib/rcmain
 */

int
iounit(int fd)
{
    int i, cfd;
    char buf[128], *args[10];

    snprint(buf, sizeof buf, "#d/%dctl", fd);
    cfd = open(buf, OREAD);
    if(cfd < 0)
        return 0;
    i = read(cfd, buf, sizeof buf-1);
    close(cfd);
    if(i <= 0)
        return 0;
    buf[i] = '\0';
    if(tokenize(buf, args, nelem(args)) != nelem(args))
        return 0;
    return atoi(args[7]);
}
```
Uses atoi() 123b, close(), open(), read() 192a, snprint() 535b, and tokenize() 454b.

⟨libc/9sys/iounit.c 486e⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* iounit 486d⟩

libc/9sys/nsec.c

⟨*global* order 487a⟩≡ (487e)

```
static uvlong order = 0x0001020304050607ULL;
```

Uses order-254 487a.

⟨*function* be2vlong 487b⟩≡ (487e)

```
static void
be2vlong(vlong *to, uchar *f)
{
    uchar *t, *o;
    int i;

    t = (uchar*)to;
    o = (uchar*)&order;
    for(i = 0; i < sizeof order; i++)
        t[o[i]] = f[i];
}
```

Uses order-254 487a.

⟨*global* fd 487c⟩≡ (487e)

```
static int fd = -1;
```

Uses fd-255 487c.

⟨*global* fds 487d⟩≡ (487e)

```
static struct {
    int pid;
    int fd;
} fds[64];
```

Uses __anon_struct_37 487d.

⟨libc/9sys/nsec.c 487e⟩≡

```
⟨libc includes 387a⟩
#include <tos.h>

⟨global order 487a⟩

⟨function be2vlong 487b⟩

⟨global fd 487c⟩
⟨global fds 487d⟩

⟨function nsec 227⟩
```

libc/9sys/nulldir.c

⟨libc/9sys/nulldir.c 487f⟩≡

```
⟨libc includes 387a⟩
⟨function nulldir 214d⟩
```

libc/9sys/postnote.c

⟨libc/9sys/postnote.c 487g⟩≡

```
⟨libc includes 387a⟩
⟨function postnote 258c⟩
```

libc/9sys/privalloc.c

⟨*global privlock (9sys/privalloc.c)* 488a⟩≡ (488f)
```
static Lock privlock;
```

⟨*global* `privinit` 488b⟩≡ (488f)
```
static int  privinit;
```

⟨*global* `privs` 488c⟩≡ (488f)
```
static void **privs;
```

⟨*function* `privalloc` 488d⟩≡ (488f)
```
void **
privalloc(void)
{
    void **p;
    int i;

    lock(&privlock);
    if(!privinit){
        privinit = 1;
        if(_nprivates){
            _privates[0] = 0;
            for(i = 1; i < _nprivates; i++)
                _privates[i] = &_privates[i - 1];
            privs = &_privates[i - 1];
        }
    }
    p = privs;
    if(p != nil){
        privs = *p;
        *p = nil;
    }
    unlock(&privlock);
    return p;
}
```
Uses `lock()` 245b, `privinit-261` 488b, `privlock-260` 488a, `privs-262` 488c, and `unlock()` 245c.

⟨*function* `privfree` 488e⟩≡ (488f)
```
void
privfree(void **p)
{
    lock(&privlock);
    if(p != nil && privinit){
        *p = privs;
        privs = p;
    }
    unlock(&privlock);
}
```
Uses `lock()` 245b, `privinit-261` 488b, `privlock-260` 488a, `privs-262` 488c, and `unlock()` 245c.

⟨`libc/9sys/privalloc.c` 488f⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*global privlock (9sys/privalloc.c)* 488a⟩
  ⟨*global* `privinit` 488b⟩
  ⟨*global* `privs` 488c⟩
```
extern void **_privates;
extern int  _nprivates;
```

⟨*function* privalloc 488d⟩

⟨*function* privfree 488e⟩

libc/9sys/pushssl.c

⟨libc/9sys/pushssl.c 489a⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* pushssl 239b⟩

libc/9sys/pushtls.c

⟨*enum* _anon_ (9sys/pushtls.c) 489b⟩≡ (489d)

```
enum {
    TLSFinishedLen = 12,
    HFinished = 20,
};
```

⟨*function* finished 489c⟩≡ (489d)

```
static int
finished(int hand, int isclient)
{
    int i, n;
    uchar buf[500], buf2[500];

    buf[0] = HFinished;
    buf[1] = TLSFinishedLen>>16;
    buf[2] = TLSFinishedLen>>8;
    buf[3] = TLSFinishedLen;
    n = TLSFinishedLen+4;

    for(i=0; i<2; i++){
        if(i==0)
            memmove(buf+4, "client finished", TLSFinishedLen);
        else
            memmove(buf+4, "server finished", TLSFinishedLen);
        if(isclient == 1-i){
            if(write(hand, buf, n) != n)
                return -1;
        }else{
            if(readn(hand, buf2, n) != n || memcmp(buf,buf2,n) != 0)
                return -1;
        }
    }
    return 1;
}
```

Uses HFinished-225 489b, TLSFinishedLen-224 489b, memcmp() 53, memmove() 48, readn() 192c, and write() 192b.

⟨libc/9sys/pushtls.c 489d⟩≡
  ⟨*libc includes* 387a⟩

```
#include <mp.h>
#include <libsec.h>
```

  ⟨*enum* _anon_ (9sys/pushtls.c) 489b⟩

  ⟨*function* finished 489c⟩

  ⟨*function* pushtls 240⟩

libc/9sys/putenv.c

⟨libc/9sys/putenv.c 490a⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* putenv 257b⟩

libc/9sys/qlock.c

⟨*global* ql 490b⟩≡ (491a)
```
static struct {
    QLp *p;
    QLp x[1024];
} ql = {
    ql.x
};
```
Uses __anon_struct_35 490b and ql-248 490b.

⟨*enum* _anon_ (9sys/qlock.c) 490c⟩≡ (491a)
```
enum
{
    Queuing,
    QueuingR,
    QueuingW,
    Sleeping,
};
```

⟨*global* _rendezvousp 490d⟩≡ (491a)
```
static void*    (*_rendezvousp)(void*, void*) = rendezvous;
```
Uses _rendezvousp-253 490d and rendezvous().

⟨*function* _qlockinit 490e⟩≡ (491a)
```
/* this gets called by the thread library ONLY to get us to use its rendezvous */
void
_qlockinit(void* (*r)(void*, void*))
{
    _rendezvousp = r;
}
```
Uses _rendezvousp-253 490d.

⟨*function* getqlp 490f⟩≡ (491a)
```
/* find a free shared memory location to queue ourselves in */
static QLp*
getqlp(void)
{
    QLp *p, *op;

    op = ql.p;
    for(p = op+1; ; p++){
        if(p == &ql.x[nelem(ql.x)])
            p = ql.x;
        if(p == op)
            abort();
        if(_tas(&(p->inuse)) == 0){
            ql.p = p;
            p->next = nil;
            break;
        }
    }
    return p;
}
```
Uses abort() 356d and ql-248 490b.

⟨libc/9sys/qlock.c 491a⟩≡
⟨*libc includes* 387a⟩
⟨*global* ql 490b⟩

⟨*enum* _anon_ (9sys/qlock.c) 490c⟩

⟨*global* _rendezvousp 490d⟩

⟨*function* _qlockinit 490e⟩

⟨*function* getqlp 490f⟩

⟨*function* qlock 246c⟩

⟨*function* qunlock 247a⟩

⟨*function* canqlock 247b⟩

⟨*function* rlock 248b⟩

⟨*function* canrlock 249a⟩

⟨*function* runlock 249b⟩

⟨*function* wlock 249c⟩

⟨*function* canwlock 250a⟩

⟨*function* wunlock 250b⟩

⟨*function* rsleep 251b⟩

⟨*function* rwakeup 252⟩

⟨*function* rwakeupall 253a⟩

libc/9sys/read.c

⟨libc/9sys/read.c 491b⟩≡
⟨*libc includes* 387a⟩
⟨*function* read 192a⟩

libc/9sys/read9pmsg.c

⟨libc/9sys/read9pmsg.c 491c⟩≡
⟨*libc includes* 387a⟩
```
#include <fcall.h>
```

⟨*function* read9pmsg 266e⟩

libc/9sys/readv.c

⟨*function* ioreadv 491d⟩≡ (492c)
```
static
long
ioreadv(int fd, IOchunk *io, int nio, vlong offset)
{
```

```
    int i;
    long m, n, tot;
    char *buf, *p;

    tot = 0;
    for(i=0; i<nio; i++)
        tot += io[i].len;
    buf = malloc(tot);
    if(buf == nil)
        return -1;

    tot = pread(fd, buf, tot, offset);

    p = buf;
    n = tot;
    for(i=0; i<nio; i++){
        if(n <= 0)
            break;
        m = io->len;
        if(m > n)
            m = n;
        memmove(io->addr, p, m);
        n -= m;
        p += m;
        io++;
    }

    free(buf);
    return tot;
}
```

Uses free() 63i, malloc() 63g, memmove() 48, and pread().

⟨*function* readv 492a⟩≡ (492c)

```
long
readv(int fd, IOchunk *io, int nio)
{
    return ioreadv(fd, io, nio, -1LL);
}
```

Uses ioreadv() 491d.

⟨*function* preadv 492b⟩≡ (492c)

```
long
preadv(int fd, IOchunk *io, int nio, vlong off)
{
    return ioreadv(fd, io, nio, off);
}
```

Uses ioreadv() 491d.

⟨libc/9sys/readv.c 492c⟩≡

```
⟨libc includes 387a⟩
⟨function ioreadv 491d⟩

⟨function readv 492a⟩

⟨function preadv 492b⟩
```

libc/9sys/rerrstr.c

⟨libc/9sys/rerrstr.c 493a⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* rerrstr 234a⟩

libc/9sys/sbrk.c

⟨libc/9sys/sbrk.c 493b⟩≡
  ⟨*libc includes* 387a⟩

  ⟨*global extern declaration* end 57b⟩
  ⟨*global* bloc 57c⟩

  ⟨*enum* _anon_ (9sys/sbrk.c) 58b⟩

  ⟨*function* sbrk 58a⟩

libc/9sys/setnetmtpt.c

⟨*function* setnetmtpt 493c⟩≡ (493d)

```
void
setnetmtpt(char *net, int n, char *x)
{
    if(x == nil)
        x = "/net";

    if(*x == '/'){
        strncpy(net, x, n);
        net[n-1] = 0;
    } else {
        snprint(net, n, "/net%s", x);
    }
}
```

Uses snprint() 535b and strncpy() 443e.

⟨libc/9sys/setnetmtpt.c 493d⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* setnetmtpt 493c⟩

libc/9sys/sysfatal.c

⟨libc/9sys/sysfatal.c 493e⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* _sysfatalimpl 236c⟩

  ⟨*global* _sysfatal 236b⟩

  ⟨*function* sysfatal 236a⟩

libc/9sys/syslog.c

⟨libc/9sys/syslog.c 493f⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*global* sl 356a⟩

⟨*function* _syslogopen 356b⟩

⟨*function* eqdirdev 356c⟩

⟨*function* syslog 354c⟩

libc/9sys/sysname.c

⟨*function* sysname 494a⟩≡ (494b)

```
char*
sysname(void)
{
    int f, n;
    static char b[128];

    if(b[0])
        return b;

    f = open("#c/sysname", 0);
    if(f >= 0) {
        n = read(f, b, sizeof(b)-1);
        if(n > 0)
            b[n] = 0;
        close(f);
    }
    return b;
}
```

Uses close(), open(), and read() 192a.

⟨libc/9sys/sysname.c 494b⟩≡
⟨*libc includes* 387a⟩
⟨*function* sysname 494a⟩

libc/9sys/time.c

⟨libc/9sys/time.c 494c⟩≡
⟨*libc includes* 387a⟩
⟨*function* oldtime 226b⟩

⟨*function* time 226a⟩

libc/9sys/times.c

⟨*function skip (9sys/times.c)* 494d⟩≡ (495b)

```
static
char*
skip(char *p)
{

    while(*p == ' ')
        p++;
    while(*p != ' ' && *p != 0)
        p++;
    return p;
}
```

⟨*function* times 495a⟩≡ (495b)

```
/*
 *  after a fork with fd's copied, both fd's are pointing to
 *  the same Chan structure.  Since the offset is kept in the Chan
 *  structure, the seek's and read's in the two processes can be
 *  are competing moving the offset around.  Hence the unusual loop
 *  in the middle of this routine.
 */
long
times(long *t)
{
    char b[200], *p;
    static int f = -1;
    int i, retries;
    ulong r;

    memset(b, 0, sizeof(b));
    for(retries = 0; retries < 100; retries++){
        if(f < 0)
            f = open("/dev/cputime", OREAD|OCEXEC);
        if(f < 0)
            break;
        if(seek(f, 0, 0) < 0 || (i = read(f, b, sizeof(b))) < 0){
            close(f);
            f = -1;
        } else {
            if(i != 0)
                break;
        }
    }
    p = b;
    if(t)
        t[0] = atol(p);
    p = skip(p);
    if(t)
        t[1] = atol(p);
    p = skip(p);
    r = atol(p);
    if(t){
        p = skip(p);
        t[2] = atol(p);
        p = skip(p);
        t[3] = atol(p);
    }
    return r;
}
```

Uses `atol()` 123c, `close()`, `memset()` 46b, `open()`, `read()` 192a, and `seek()`.

⟨`libc/9sys/times.c` 495b⟩≡

```
⟨libc includes 387a⟩
⟨function skip (9sys/times.c) 494d⟩

⟨function times 495a⟩
```

libc/9sys/tm2sec.c

⟨*constant* TZSIZE *(9sys/tm2sec.c)* 495c⟩≡ (498a)

```
#define TZSIZE  150
```

⟨*global* timezone*(9sys/tm2sec.c)* 496a⟩≡ (498a)

```
static
struct
{
    char    stname[4];
    char    dlname[4];
    long    stdiff;
    long    dldiff;
    long    dlpairs[TZSIZE];
} timezone;
```

Uses TZSIZE-238 495c and __anon_struct_33 496a.

⟨*constant* SEC2MIN 496b⟩≡ (498a)

```
#define SEC2MIN 60L
```

⟨*constant* SEC2HOUR 496c⟩≡ (498a)

```
#define SEC2HOUR (60L*SEC2MIN)
```

⟨*constant* SEC2DAY 496d⟩≡ (498a)

```
#define SEC2DAY (24L*SEC2HOUR)
```

⟨*global* dmsize*(9sys/tm2sec.c)* 496e⟩≡ (498a)

```
/*
 *  days per month plus days/year
 */
static  int dmsize[] =
{
    365, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
};
```

⟨*global* ldmsize 496f⟩≡ (498a)

```
static  int ldmsize[] =
{
    366, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
};
```

⟨*function* yrsize 496g⟩≡ (498a)

```
/*
 *  return the days/month for the given year
 */
static int *
yrsize(int y)
{
    if((y%4) == 0 && ((y%100) != 0 || (y%400) == 0))
        return ldmsize;
    else
        return dmsize;
}
```

Uses dmsize-243 496e and ldmsize-244 496f.

⟨*function* readtimezone*(9sys/tm2sec.c)* 496h⟩≡ (498a)

```
static
void
readtimezone(void)
{
    char buf[TZSIZE*11+30], *p;
    int i;

    memset(buf, 0, sizeof(buf));
    i = open("/env/timezone", 0);
```

```
    if(i < 0)
        goto error;
    if(read(i, buf, sizeof(buf)) >= sizeof(buf))
        goto error;
    close(i);
    p = buf;
    if(rd_name(&p, timezone.stname))
        goto error;
    if(rd_long(&p, &timezone.stdiff))
        goto error;
    if(rd_name(&p, timezone.dlname))
        goto error;
    if(rd_long(&p, &timezone.dldiff))
        goto error;
    for(i=0; i<TZSIZE; i++) {
        if(rd_long(&p, &timezone.dlpairs[i]))
            goto error;
        if(timezone.dlpairs[i] == 0)
            return;
    }

error:
    timezone.stdiff = 0;
    strcpy(timezone.stname, "GMT");
    timezone.dlpairs[0] = 0;
}
```

Uses TZSIZE-238 495c, close(), memset() 46b, open(), read() 192a, strcpy() 81c, and timezone-239 496a.

⟨*function* rd_name *(9sys/tm2sec.c)* 497a⟩≡ (498a)

```
static int
rd_name(char **f, char *p)
{
    int c, i;

    for(;;) {
        c = *(*f)++;
        if(c != ' ' && c != '\n')
            break;
    }
    for(i=0; i<3; i++) {
        if(c == ' ' || c == '\n')
            return 1;
        *p++ = c;
        c = *(*f)++;
    }
    if(c != ' ' && c != '\n')
        return 1;
    *p = 0;
    return 0;
}
```

⟨*function* rd_long *(9sys/tm2sec.c)* 497b⟩≡ (498a)

```
static int
rd_long(char **f, long *p)
{
    int c, s;
    long l;

    s = 0;
    for(;;) {
```

```
        c = *(*f)++;
        if(c == '-') {
            s++;
            continue;
        }
        if(c != ' ' && c != '\n')
            break;
    }
    if(c == 0) {
        *p = 0;
        return 0;
    }
    l = 0;
    for(;;) {
        if(c == ' ' || c == '\n')
            break;
        if(c < '0' || c > '9')
            return 1;
        l = l*10 + c-'0';
        c = *(*f)++;
    }
    if(s)
        l = -l;
    *p = l;
    return 0;
}
```

```
⟨libc/9sys/tm2sec.c 498a⟩≡
  ⟨libc includes 387a⟩
  ⟨constant TZSIZE(9sys/tm2sec.c) 495c⟩
  static  void    readtimezone(void);
  static  int rd_name(char**, char*);
  static  int rd_long(char**, long*);
  ⟨global timezone(9sys/tm2sec.c) 496a⟩

  ⟨constant SEC2MIN 496b⟩
  ⟨constant SEC2HOUR 496c⟩
  ⟨constant SEC2DAY 496d⟩

  ⟨global dmsize(9sys/tm2sec.c) 496e⟩
  ⟨global ldmsize 496f⟩

  ⟨function yrsize 496g⟩

  ⟨function tm2sec 228⟩

  ⟨function readtimezone(9sys/tm2sec.c) 496h⟩

  ⟨function rd_name(9sys/tm2sec.c) 497a⟩

  ⟨function rd_long(9sys/tm2sec.c) 497b⟩
```

libc/9sys/truerand.c

```
⟨libc/9sys/truerand.c 498b⟩≡
  ⟨libc includes 387a⟩
  ⟨function truerand 155a⟩
```

libc/9sys/wait.c

⟨libc/9sys/wait.c 499a⟩≡
```
  ⟨libc includes 387a⟩
  #include <fcall.h>

  ⟨function wait 256a⟩
```

libc/9sys/waitpid.c

⟨libc/9sys/waitpid.c 499b⟩≡
```
  ⟨libc includes 387a⟩
  #include <fcall.h>

  ⟨function waitpid 256b⟩
```

libc/9sys/werrstr.c

⟨libc/9sys/werrstr.c 499c⟩≡
```
  ⟨libc includes 387a⟩
  ⟨function werrstr 234b⟩
```

libc/9sys/write.c

⟨libc/9sys/write.c 499d⟩≡
```
  ⟨libc includes 387a⟩
  ⟨function write 192b⟩
```

libc/9sys/writev.c

⟨*function* iowritev 499e⟩≡ (500c)
```
static
long
iowritev(int fd, IOchunk *io, int nio, vlong offset)
{
    int i;
    long tot;
    char *buf, *p;

    tot = 0;
    for(i=0; i<nio; i++)
        tot += io[i].len;
    buf = malloc(tot);
    if(buf == nil)
        return -1;

    p = buf;
    for(i=0; i<nio; i++){
        memmove(p, io->addr, io->len);
        p += io->len;
        io++;
    }

    tot = pwrite(fd, buf, tot, offset);

    free(buf);
```

```
    return tot;
}
```

Uses `free()` 63i, `malloc()` 63g, `memmove()` 48, and `pwrite()`.

⟨*function* `writev` 500a⟩≡ (500c)

```
long
writev(int fd, IOchunk *io, int nio)
{
    return iowritev(fd, io, nio, -1LL);
}
```

Uses `iowritev()` 499e.

⟨*function* `pwritev` 500b⟩≡ (500c)

```
long
pwritev(int fd, IOchunk *io, int nio, vlong off)
{
    return iowritev(fd, io, nio, off);
}
```

Uses `iowritev()` 499e.

⟨`libc/9sys/writev.c` 500c⟩≡

⟨*libc includes* 387a⟩
⟨*function* `iowritev` 499e⟩

⟨*function* `writev` 500a⟩

⟨*function* `pwritev` 500b⟩

A.4 `libc/fmt/`

`libc/fmt/fmtdef.h`

⟨`libc/fmt/fmtdef.h` 500d⟩≡

```
/*
 * dofmt -- format to a buffer
 * the number of characters formatted is returned,
 * or -1 if there was an error.
 * if the buffer is ever filled, flush is called.
 * it should reset the buffer and return whether formatting should continue.
 */

typedef int (*Fmts)(Fmt*);

typedef struct Quoteinfo Quoteinfo;
struct Quoteinfo
{
    int quoted;     /* if set, string must be quoted */
    int nrunesin;   /* number of input runes that can be accepted */
    int nbytesin;   /* number of input bytes that can be accepted */
    int nrunesout;  /* number of runes that will be generated */
    int nbytesout;  /* number of bytes that will be generated */
};

void    *_fmtflush(Fmt*, void*, int);
void    *_fmtdispatch(Fmt*, void*, int);
int _floatfmt(Fmt*, double);
int _fmtpad(Fmt*, int);
```

```
int _rfmtpad(Fmt*, int);
int _fmtFdFlush(Fmt*);

int _efgfmt(Fmt*);
int _charfmt(Fmt*);
int _countfmt(Fmt*);
int _flagfmt(Fmt*);
int _percentfmt(Fmt*);
int _ifmt(Fmt*);
int _runefmt(Fmt*);
int _runesfmt(Fmt*);
int _strfmt(Fmt*);
int _badfmt(Fmt*);
int _fmtcpy(Fmt*, void*, int, int);
int _fmtrcpy(Fmt*, void*, int n);

void    _fmtlock(void);
void    _fmtunlock(void);

#define FMTCHAR(f, t, s, c)\
    do{\
    if(t + 1 > (char*)s){\
        t = _fmtflush(f, t, 1);\
        if(t != nil)\
            s = f->stop;\
        else\
            return -1;\
    }\
    *t++ = c;\
    }while(0)

#define FMTRCHAR(f, t, s, c)\
    do{\
    if(t + 1 > (Rune*)s){\
        t = _fmtflush(f, t, sizeof(Rune));\
        if(t != nil)\
            s = f->stop;\
        else\
            return -1;\
    }\
    *t++ = c;\
    }while(0)

#define FMTRUNE(f, t, s, r)\
    do{\
    Rune _rune;\
    int _runelen;\
    if(t + UTFmax > (char*)s && t + (_runelen = runelen(r)) > (char*)s){\
        t = _fmtflush(f, t, _runelen);\
        if(t != nil)\
            s = f->stop;\
        else\
            return -1;\
    }\
    if(r < Runeself)\
        *t++ = r;\
    else{\
        _rune = r;\
        t += runetochar(t, &_rune);\
    }\
```

```
    }while(0)
```

Uses Quoteinfo 500d.

libc/fmt/dofmt.c

⟨*function* _fmtflush 502a⟩≡ (510b)

```
void *
_fmtflush(Fmt *f, void *t, int len)
{
    if(f->runes)
        f->nfmt += (Rune*)t - (Rune*)f->to;
    else
        f->nfmt += (char*)t - (char *)f->to;
    f->to = t;
    if(f->flush == 0 || (*f->flush)(f) == 0 || (char*)f->to + len > (char*)f->stop){
        f->stop = f->to;
        return nil;
    }
    return f->to;
}
```

⟨*function* _fmtpad 502b⟩≡ (510b)

```
/*
 * put a formatted block of memory sz bytes long of n runes into the output buffer,
 * left/right justified in a field of at least f->width charactes
 */
int
_fmtpad(Fmt *f, int n)
{
    char *t, *s;
    int i;

    t = f->to;
    s = f->stop;
    for(i = 0; i < n; i++)
        FMTCHAR(f, t, s, ' ');
    f->nfmt += t - (char *)f->to;
    f->to = t;
    return 0;
}
```

Uses FMTCHAR 500d.

⟨*function* _rfmtpad 502c⟩≡ (510b)

```
int
_rfmtpad(Fmt *f, int n)
{
    Rune *t, *s;
    int i;

    t = f->to;
    s = f->stop;
    for(i = 0; i < n; i++)
        FMTRCHAR(f, t, s, ' ');
    f->nfmt += t - (Rune *)f->to;
    f->to = t;
    return 0;
}
```

Uses FMTRCHAR 500d.

⟨*function* _fmtcpy 503a⟩≡ (510b)

```
int
_fmtcpy(Fmt *f, void *vm, int n, int sz)
{
    Rune *rt, *rs, r;
    char *t, *s, *m, *me;
    ulong fl;
    int nc, w;

    m = vm;
    me = m + sz;
    w = f->width;
    fl = f->flags;
    if((fl & FmtPrec) && n > f->prec)
        n = f->prec;
    if(f->runes){
        if(!(fl & FmtLeft) && _rfmtpad(f, w - n) < 0)
            return -1;
        rt = f->to;
        rs = f->stop;
        for(nc = n; nc > 0; nc--){
            r = *(uchar*)m;
            if(r < Runeself)
                m++;
            else if((me - m) >= UTFmax || fullrune(m, me-m))
                m += chartorune(&r, m);
            else
                break;
            FMTRCHAR(f, rt, rs, r);
        }
        f->nfmt += rt - (Rune *)f->to;
        f->to = rt;
        if(fl & FmtLeft && _rfmtpad(f, w - n) < 0)
            return -1;
    }else{
        if(!(fl & FmtLeft) && _fmtpad(f, w - n) < 0)
            return -1;
        t = f->to;
        s = f->stop;
        for(nc = n; nc > 0; nc--){
            r = *(uchar*)m;
            if(r < Runeself)
                m++;
            else if((me - m) >= UTFmax || fullrune(m, me-m))
                m += chartorune(&r, m);
            else
                break;
            FMTRUNE(f, t, s, r);
        }
        f->nfmt += t - (char *)f->to;
        f->to = t;
        if(fl & FmtLeft && _fmtpad(f, w - n) < 0)
            return -1;
    }
    return 0;
}
```

Uses `FMTRCHAR` 500d, `FMTRUNE` 500d, `_fmtpad()` 502b, `_rfmtpad()` 502c, `chartorune()` 84d, and `fullrune()` 436c.

⟨*function* _fmtrcpy 503b⟩≡ (510b)

```
int
```

```
_fmtrcpy(Fmt *f, void *vm, int n)
{
    Rune r, *m, *me, *rt, *rs;
    char *t, *s;
    ulong fl;
    int w;

    m = vm;
    w = f->width;
    fl = f->flags;
    if((fl & FmtPrec) && n > f->prec)
        n = f->prec;
    if(f->runes){
        if(!(fl & FmtLeft) && _rfmtpad(f, w - n) < 0)
            return -1;
        rt = f->to;
        rs = f->stop;
        for(me = m + n; m < me; m++)
            FMTRCHAR(f, rt, rs, *m);
        f->nfmt += rt - (Rune *)f->to;
        f->to = rt;
        if(fl & FmtLeft && _rfmtpad(f, w - n) < 0)
            return -1;
    }else{
        if(!(fl & FmtLeft) && _fmtpad(f, w - n) < 0)
            return -1;
        t = f->to;
        s = f->stop;
        for(me = m + n; m < me; m++){
            r = *m;
            FMTRUNE(f, t, s, r);
        }
        f->nfmt += t - (char *)f->to;
        f->to = t;
        if(fl & FmtLeft && _fmtpad(f, w - n) < 0)
            return -1;
    }
    return 0;
}
```
Uses FMTRCHAR 500d, FMTRUNE 500d, _fmtpad() 502b, and _rfmtpad() 502c.

⟨*function* _charfmt 504a⟩≡ (510b)
```
/* fmt out one character */
int
_charfmt(Fmt *f)
{
    char x[1];

    x[0] = va_arg(f->args, int);
    f->prec = 1;
    return _fmtcpy(f, x, 1, 1);
}
```
Uses _fmtcpy() 503a.

⟨*function* _runefmt 504b⟩≡ (510b)
```
/* fmt out one rune */
int
_runefmt(Fmt *f)
{
    Rune x[1];
```

```
    x[0] = va_arg(f->args, int);
    return _fmtrcpy(f, x, 1);
}
```

Uses _fmtrcpy() 503b.

⟨*function* fmtstrcpy 505a⟩≡ (510b)

```
/* public helper routine: fmt out a null terminated string already in hand */
int
fmtstrcpy(Fmt *f, char *s)
{
    int i, j;
    Rune r;

    if(!s)
        return _fmtcpy(f, "<nil>", 5, 5);
    /* if precision is specified, make sure we don't wander off the end */
    if(f->flags & FmtPrec){
        i = 0;
        for(j=0; j<f->prec && s[i]; j++)
            i += chartorune(&r, s+i);
        return _fmtcpy(f, s, j, i);
    }
    return _fmtcpy(f, s, utflen(s), strlen(s));
}
```

Uses _fmtcpy() 503a, chartorune() 84d, strlen() 80a, and utflen() 89a.

⟨*function* _strfmt 505b⟩≡ (510b)

```
/* fmt out a null terminated utf string */
int
_strfmt(Fmt *f)
{
    char *s;

    s = va_arg(f->args, char *);
    return fmtstrcpy(f, s);
}
```

Uses fmtstrcpy() 505a.

⟨*function* fmtrunestrcpy 505c⟩≡ (510b)

```
/* public helper routine: fmt out a null terminated rune string already in hand */
int
fmtrunestrcpy(Fmt *f, Rune *s)
{
    Rune *e;
    int n, p;

    if(!s)
        return _fmtcpy(f, "<nil>", 5, 5);
    /* if precision is specified, make sure we don't wander off the end */
    if(f->flags & FmtPrec){
        p = f->prec;
        for(n = 0; n < p; n++)
            if(s[n] == 0)
                break;
    }else{
        for(e = s; *e; e++)
            ;
        n = e - s;
```

```
    }
    return _fmtrcpy(f, s, n);
}
```
Uses _fmtcpy() 503a and _fmtrcpy() 503b.

⟨*function* _runesfmt 506a⟩≡ (510b)
```
/* fmt out a null terminated rune string */
int
_runesfmt(Fmt *f)
{
    Rune *s;

    s = va_arg(f->args, Rune *);
    return fmtrunestrcpy(f, s);
}
```
Uses fmtrunestrcpy() 505c.

⟨*function* _percentfmt 506b⟩≡ (510b)
```
/* fmt a % */
int
_percentfmt(Fmt *f)
{
    Rune x[1];

    x[0] = f->r;
    f->prec = 1;
    return _fmtrcpy(f, x, 1);
}
```
Uses _fmtrcpy() 503b.

⟨*enum* _anon_ (fmt/dofmt.c) 506c⟩≡ (510b)
```
enum {
    /* %,#llb could emit a sign, "0b" and 64 digits with 21 commas */
    Maxintwidth = 1 + 2 + 64 + 64/3,
};
```

⟨*function* _ifmt 506d⟩≡ (510b)
```
/* fmt an integer */
int
_ifmt(Fmt *f)
{
    char buf[Maxintwidth + 1], *p, *conv;
    uvlong vu;
    ulong u;
    uintptr pu;
    int neg, base, i, n, fl, w, isv;

    neg = 0;
    fl = f->flags;
    isv = 0;
    vu = 0;
    u = 0;
    if(f->r == 'p'){
        pu = va_arg(f->args, uintptr);
        if(sizeof(uintptr) == sizeof(uvlong)){
            vu = pu;
            isv = 1;
        }else
            u = pu;
```

```
	f->r = 'x';
	fl |= FmtUnsigned;
}else if(fl & FmtVLong){
	isv = 1;
	if(fl & FmtUnsigned)
		vu = va_arg(f->args, uvlong);
	else
		vu = va_arg(f->args, vlong);
}else if(fl & FmtLong){
	if(fl & FmtUnsigned)
		u = va_arg(f->args, ulong);
	else
		u = va_arg(f->args, long);
}else if(fl & FmtByte){
	if(fl & FmtUnsigned)
		u = (uchar)va_arg(f->args, int);
	else
		u = (char)va_arg(f->args, int);
}else if(fl & FmtShort){
	if(fl & FmtUnsigned)
		u = (ushort)va_arg(f->args, int);
	else
		u = (short)va_arg(f->args, int);
}else{
	if(fl & FmtUnsigned)
		u = va_arg(f->args, uint);
	else
		u = va_arg(f->args, int);
}
conv = "0123456789abcdef";
switch(f->r){
case 'd':
	base = 10;
	break;
case 'x':
	base = 16;
	break;
case 'X':
	base = 16;
	conv = "0123456789ABCDEF";
	break;
case 'b':
	base = 2;
	break;
case 'o':
	base = 8;
	break;
default:
	return -1;
}
if(!(fl & FmtUnsigned)){
	if(isv && (vlong)vu < 0){
		vu = -(vlong)vu;
		neg = 1;
	}else if(!isv && (long)u < 0){
		u = -(long)u;
		neg = 1;
	}
}
p = buf + sizeof buf - 1;
```

```
n = 0;
if(isv){
    while(vu){
        i = vu % base;
        vu /= base;
        if((fl & FmtComma) && n % 4 == 3){
            *p-- = ',';
            n++;
        }
        *p-- = conv[i];
        n++;
    }
}else{
    while(u){
        i = u % base;
        u /= base;
        if((fl & FmtComma) && n % 4 == 3){
            *p-- = ',';
            n++;
        }
        *p-- = conv[i];
        n++;
    }
}
if(n == 0){
    *p-- = '0';
    n = 1;
}
for(w = f->prec; n < w && p > buf+3; n++)
    *p-- = '0';
if(neg || (fl & (FmtSign|FmtSpace)))
    n++;
if(fl & FmtSharp){
    if(base == 16)
        n += 2;
    else if(base == 8){
        if(p[1] == '0')
            fl &= ~FmtSharp;
        else
            n++;
    }
}
if((fl & FmtZero) && !(fl & (FmtLeft|FmtPrec))){
    for(w = f->width; n < w && p > buf+3; n++)
        *p-- = '0';
    f->width = 0;
}
if(fl & FmtSharp){
    if(base == 16)
        *p-- = f->r;
    if(base == 16 || base == 8)
        *p-- = '0';
}
if(neg)
    *p-- = '-';
else if(fl & FmtSign)
    *p-- = '+';
else if(fl & FmtSpace)
    *p-- = ' ';
f->flags &= ~FmtPrec;
```

```
    return _fmtcpy(f, p + 1, n, n);
}
```
Uses Maxintwidth-33 506c and _fmtcpy() 503a.

⟨*function* _countfmt 509a⟩≡ (510b)
```
int
_countfmt(Fmt *f)
{
    void *p;
    ulong fl;

    fl = f->flags;
    p = va_arg(f->args, void*);
    if(fl & FmtVLong){
        *(vlong*)p = f->nfmt;
    }else if(fl & FmtLong){
        *(long*)p = f->nfmt;
    }else if(fl & FmtByte){
        *(char*)p = f->nfmt;
    }else if(fl & FmtShort){
        *(short*)p = f->nfmt;
    }else{
        *(int*)p = f->nfmt;
    }
    return 0;
}
```

⟨*function* _flagfmt 509b⟩≡ (510b)
```
int
_flagfmt(Fmt *f)
{
    switch(f->r){
    case ',':
        f->flags |= FmtComma;
        break;
    case '-':
        f->flags |= FmtLeft;
        break;
    case '+':
        f->flags |= FmtSign;
        break;
    case '#':
        f->flags |= FmtSharp;
        break;
    case ' ':
        f->flags |= FmtSpace;
        break;
    case 'u':
        f->flags |= FmtUnsigned;
        break;
    case 'h':
        if(f->flags & FmtShort)
            f->flags |= FmtByte;
        f->flags |= FmtShort;
        break;
    case 'l':
        if(f->flags & FmtLong)
            f->flags |= FmtVLong;
        f->flags |= FmtLong;
        break;
```

```
    }
    return 1;
}
```

⟨*function* _badfmt 510a⟩≡ (510b)

```
/* default error format */
int
_badfmt(Fmt *f)
{
    Rune x[3];

    x[0] = '%';
    x[1] = f->r;
    x[2] = '%';
    f->prec = 3;
    _fmtrcpy(f, x, 3);
    return 0;
}
```

Uses _fmtrcpy() 503b.

⟨libc/fmt/dofmt.c 510b⟩≡

⟨*libc includes* 387a⟩

```
#include "fmtdef.h"
```

⟨*function* dofmt 74⟩

⟨*function* _fmtflush 502a⟩

⟨*function* _fmtpad 502b⟩

⟨*function* _rfmtpad 502c⟩

⟨*function* _fmtcpy 503a⟩

⟨*function* _fmtrcpy 503b⟩

⟨*function* _charfmt 504a⟩

⟨*function* _runefmt 504b⟩

⟨*function* fmtstrcpy 505a⟩

⟨*function* _strfmt 505b⟩

⟨*function* fmtrunestrcpy 505c⟩

⟨*function* _runesfmt 506a⟩

⟨*function* _percentfmt 506b⟩

⟨*enum* _anon_ (fmt/dofmt.c) 506c⟩

⟨*function* _ifmt 506d⟩

⟨*function* _countfmt 509a⟩

⟨*function* _flagfmt 509b⟩

⟨*function* _badfmt 510a⟩

libc/fmt/dorfmt.c

⟨*function* dorfmt 511a⟩≡ (511b)

```
/* format the output into f->to and return the number of characters fmted  */

int
dorfmt(Fmt *f, Rune *fmt)
{
    Rune *rt, *rs;
    int r;
    char *t, *s;
    int nfmt;

    nfmt = f->nfmt;
    for(;;){
        if(f->runes){
            rt = f->to;
            rs = f->stop;
            while((r = *fmt++) && r != '%'){
                FMTRCHAR(f, rt, rs, r);
            }
            f->nfmt += rt - (Rune *)f->to;
            f->to = rt;
            if(!r)
                return f->nfmt - nfmt;
            f->stop = rs;
        }else{
            t = f->to;
            s = f->stop;
            while((r = *fmt++) && r != '%'){
                FMTRUNE(f, t, f->stop, r);
            }
            f->nfmt += t - (char *)f->to;
            f->to = t;
            if(!r)
                return f->nfmt - nfmt;
            f->stop = s;
        }

        fmt = _fmtdispatch(f, fmt, 1);
        if(fmt == nil)
            return -1;
    }
}
```

Uses FMTRCHAR 500d, FMTRUNE 500d, and _fmtdispatch() 520.

⟨libc/fmt/dorfmt.c 511b⟩≡

```
⟨libc includes 387a⟩
#include "fmtdef.h"

⟨function dorfmt 511a⟩
```

libc/fmt/errfmt.c

⟨*function* errfmt 511c⟩≡ (512a)

```
int
errfmt(Fmt *f)
{
    char buf[ERRMAX];
```

```
    rerrstr(buf, sizeof buf);
    return _fmtcpy(f, buf, utflen(buf), strlen(buf));
}
```

Uses _fmtcpy() 503a, rerrstr() 234a, strlen() 80a, and utflen() 89a.

⟨libc/fmt/errfmt.c 512a⟩≡

```
⟨libc includes 387a⟩
#include "fmtdef.h"

⟨function errfmt 511c⟩
```

libc/fmt/fltfmt.c

⟨*enum* _anon_ (fmt/fltfmt.c) 512b⟩≡ (517c)

```
enum
{
    FDIGIT  = 30,
    FDEFLT  = 6,
    NSIGNIF = 17,
    NEXP10  = 308,
};
```

⟨*function* xadd 512c⟩≡ (517c)

```
static int
xadd(char *a, int n, int v)
{
    char *b;
    int c;

    if(n < 0 || n >= NSIGNIF)
        return 0;
    for(b = a+n; b >= a; b--) {
        c = *b + v;
        if(c <= '9') {
            *b = c;
            return 0;
        }
        *b = '0';
        v = 1;
    }
    *a = '1';   // overflow adding
    return 1;
}
```

Uses NSIGNIF-30 512b.

⟨*function* xsub 512d⟩≡ (517c)

```
static int
xsub(char *a, int n, int v)
{
    char *b;
    int c;

    for(b = a+n; b >= a; b--) {
        c = *b - v;
        if(c >= '0') {
            *b = c;
            return 0;
        }
```

```
        *b = '9';
        v = 1;
    }
    *a = '9';   // underflow subtracting
    return 1;
}
```

⟨*function* xdtoa 513⟩≡ (517c)

```
static void
xdtoa(Fmt *fmt, char *s2, double f)
{
    char s1[NSIGNIF+10];
    double g, h;
    int e, d, i, n;
    int c1, c2, c3, c4, ucase, sign, chr, prec;

    prec = FDEFLT;
    if(fmt->flags & FmtPrec)
        prec = fmt->prec;
    if(prec > FDIGIT)
        prec = FDIGIT;
    if(isNaN(f)) {
        strcpy(s2, "NaN");
        return;
    }
    if(isInf(f, 1)) {
        strcpy(s2, "+Inf");
        return;
    }
    if(isInf(f, -1)) {
        strcpy(s2, "-Inf");
        return;
    }
    sign = 0;
    if(f < 0) {
        f = -f;
        sign++;
    }
    ucase = 0;
    chr = fmt->r;
    if(isupper(chr)) {
        ucase = 1;
        chr = tolower(chr);
    }

    e = 0;
    g = f;
    if(g != 0) {
        frexp(f, &e);
        e = e * .301029995664;
        if(e >= -150 && e <= +150) {
            d = 0;
            h = f;
        } else {
            d = e/2;
            h = f * pow10(-d);
        }
        g = h * pow10(d-e);
        while(g < 1) {
            e--;
```

```
        g = h * pow10(d-e);
    }
    while(g >= 10) {
        e++;
        g = h * pow10(d-e);
    }
}

/*
 * convert NSIGNIF digits and convert
 * back to get accuracy.
 */
for(i=0; i<NSIGNIF; i++) {
    d = g;
    s1[i] = d + '0';
    g = (g - d) * 10;
}
s1[i] = 0;

/*
 * try decimal rounding to eliminate 9s
 */
c2 = prec + 1;
if(chr == 'f')
    c2 += e;
if(c2 >= NSIGNIF-2) {
    strcpy(s2, s1);
    d = e;
    s1[NSIGNIF-2] = '0';
    s1[NSIGNIF-1] = '0';
    sprint(s1+NSIGNIF, "e%d", e-NSIGNIF+1);
    g = strtod(s1, nil);
    if(g == f)
        goto found;
    if(xadd(s1, NSIGNIF-3, 1)) {
        e++;
        sprint(s1+NSIGNIF, "e%d", e-NSIGNIF+1);
    }
    g = strtod(s1, nil);
    if(g == f)
        goto found;
    strcpy(s1, s2);
    e = d;
}

/*
 * convert back so s1 gets exact answer
 */
for(;;) {
    sprint(s1+NSIGNIF, "e%d", e-NSIGNIF+1);
    g = strtod(s1, nil);
    if(f > g) {
        if(xadd(s1, NSIGNIF-1, 1))
            e--;
        continue;
    }
    if(f < g) {
        if(xsub(s1, NSIGNIF-1, 1))
            e++;
        continue;
```

```
		}
		break;
	}

found:
	/*
	 * sign
	 */
	d = 0;
	i = 0;
	if(sign)
		s2[d++] = '-';
	else if(fmt->flags & FmtSign)
		s2[d++] = '+';
	else if(fmt->flags & FmtSpace)
		s2[d++] = ' ';

	/*
	 * copy into final place
	 * c1 digits of leading '0'
	 * c2 digits from conversion
	 * c3 digits of trailing '0'
	 * c4 digits after '.'
	 */
	c1 = 0;
	c2 = prec + 1;
	c3 = 0;
	c4 = prec;
	switch(chr) {
	default:
		if(xadd(s1, c2, 5))
			e++;
		break;
	case 'g':
		/*
		 * decide on 'e' of 'f' style convers
		 */
		if(xadd(s1, c2, 5))
			e++;
		if(e >= -5 && e <= prec) {
			c1 = -e - 1;
			c4 = prec - e;
			chr = 'h';	// flag for 'f' style
		}
		break;
	case 'f':
		if(xadd(s1, c2+e, 5))
			e++;
		c1 = -e;
		if(c1 > prec)
			c1 = c2;
		c2 += e;
		break;
	}

	/*
	 * clean up c1 c2 and c3
	 */
	if(c1 < 0)
		c1 = 0;
```

```
if(c2 < 0)
    c2 = 0;
if(c2 > NSIGNIF) {
    c3 = c2-NSIGNIF;
    c2 = NSIGNIF;
}

/*
 * copy digits
 */
while(c1 > 0) {
    if(c1+c2+c3 == c4)
        s2[d++] = '.';
    s2[d++] = '0';
    c1--;
}
while(c2 > 0) {
    if(c2+c3 == c4)
        s2[d++] = '.';
    s2[d++] = s1[i++];
    c2--;
}
while(c3 > 0) {
    if(c3 == c4)
        s2[d++] = '.';
    s2[d++] = '0';
    c3--;
}

/*
 * strip trailing '0' on g conv
 */
if(fmt->flags & FmtSharp) {
    if(0 == c4)
        s2[d++] = '.';
} else
if(chr == 'g' || chr == 'h') {
    for(n=d-1; n>=0; n--)
        if(s2[n] != '0')
            break;
    for(i=n; i>=0; i--)
        if(s2[i] == '.') {
            d = n;
            if(i != n)
                d++;
            break;
        }
}
if(chr == 'e' || chr == 'g') {
    if(ucase)
        s2[d++] = 'E';
    else
        s2[d++] = 'e';
    c1 = e;
    if(c1 < 0) {
        s2[d++] = '-';
        c1 = -c1;
    } else
        s2[d++] = '+';
    if(c1 >= 100) {
```

```
            s2[d++] = c1/100 + '0';
            c1 = c1%100;
        }
        s2[d++] = c1/10 + '0';
        s2[d++] = c1%10 + '0';
    }
    s2[d] = 0;
}
```

Uses FDEFLT-29 512b, FDIGIT-28 512b, NSIGNIF-30 512b, frexp() 151a, isInf() 127a, isNaN() 126d, pow10() 140a, sprint() 75b, strcpy() 81c, strtod() 128, tolower() 29d, xadd() 512c, and xsub() 512d.

⟨*function* _floatfmt 517a⟩≡ (517c)

```
int
_floatfmt(Fmt *fmt, double f)
{
    char s[1+NEXP10+1+FDIGIT+1];

    /*
     * The max length of a %f string is
     *  '[+-]'+"max exponent"+'.'+"max precision"+'\0'
     * which is 341 currently.
     */
    xdtoa(fmt, s, f);
    fmt->flags &= FmtWidth|FmtLeft;
    _fmtcpy(fmt, s, strlen(s), strlen(s));
    return 0;
}
```

Uses FDIGIT-28 512b, NEXP10-31 512b, _fmtcpy() 503a, strlen() 80a, and xdtoa() 513.

⟨*function* _efgfmt 517b⟩≡ (517c)

```
int
_efgfmt(Fmt *f)
{
    double d;

    d = va_arg(f->args, double);
    return _floatfmt(f, d);
}
```

Uses _floatfmt() 517a.

⟨libc/fmt/fltfmt.c 517c⟩≡

```
⟨libc includes 387a⟩
#include <ctype.h>
#include "fmtdef.h"

⟨enum _anon_ (fmt/fltfmt.c) 512b⟩

⟨function xadd 512c⟩

⟨function xsub 512d⟩

⟨function xdtoa 513⟩

⟨function _floatfmt 517a⟩

⟨function _efgfmt 517b⟩
```

libc/fmt/fmt.c

⟨*enum* _anon_ (fmt/fmt.c) 518a⟩≡ (521)

```
enum
{
    Maxfmt = 64
};
```

⟨*struct* Convfmt 518b⟩≡ (521)

```
struct Convfmt
{
    int c;
    volatile    Fmts    fmt;    /* for spin lock in fmtfmt; avoids race due to write order */
};
```

⟨*global* fmtalloc 518c⟩≡ (521)

```
struct
{
    /* lock by calling _fmtlock, _fmtunlock */
    int nfmt;
    Convfmt fmt[Maxfmt];
} fmtalloc;
```

Uses Maxfmt-34 518a and __anon_struct_9 518c.

⟨*global* knownfmt 518d⟩≡ (521)

```
static Convfmt knownfmt[] = {
    ' ',    _flagfmt,
    '#',    _flagfmt,
    '%',    _percentfmt,
    '+',    _flagfmt,
    ',',    _flagfmt,
    '-',    _flagfmt,
    'C',    _runefmt,
    'E',    _efgfmt,
    'G',    _efgfmt,
    'S',    _runesfmt,
    'X',    _ifmt,
    'b',    _ifmt,
    'c',    _charfmt,
    'd',    _ifmt,
    'e',    _efgfmt,
    'f',    _efgfmt,
    'g',    _efgfmt,
    'h',    _flagfmt,
    'l',    _flagfmt,
    'n',    _countfmt,
    'o',    _ifmt,
    'p',    _ifmt,
    'r',    errfmt,
    's',    _strfmt,
    'u',    _flagfmt,
    'x',    _ifmt,
    0,  nil,
};
```

Uses _charfmt() 504a, _countfmt() 509a, _efgfmt() 517b, _flagfmt() 509b, _ifmt() 506d, _percentfmt() 506b, _runefmt() 504b, _runesfmt() 506a, _strfmt() 505b, and errfmt() 511c.

⟨*global doquote (fmt/fmt.c)* 518e⟩≡ (521)

```
int (*doquote)(int);
```

⟨*function* _fmtinstall 519a⟩≡ (521)

```
/*
 * _fmtlock() must be set
 */
static int
_fmtinstall(int c, Fmts f)
{
    Convfmt *p, *ep;

    if(c<=0 || c>=65536)
        return -1;
    if(!f)
        f = _badfmt;

    ep = &fmtalloc.fmt[fmtalloc.nfmt];
    for(p=fmtalloc.fmt; p<ep; p++)
        if(p->c == c)
            break;

    if(p == &fmtalloc.fmt[Maxfmt])
        return -1;

    p->fmt = f;
    if(p == ep){    /* installing a new format character */
        fmtalloc.nfmt++;
        p->c = c;
    }

    return 0;
}
```

Uses `Maxfmt-34` 518a, `_badfmt()` 510a, and `fmtalloc` 518c.

⟨*function* fmtinstall 519b⟩≡ (521)

```
int
fmtinstall(int c, Fmts f)
{
    int ret;

    _fmtlock();
    ret = _fmtinstall(c, f);
    _fmtunlock();
    return ret;
}
```

Uses `_fmtinstall()` 519a, `_fmtlock()` 523b, and `_fmtunlock()` 523c.

⟨*function* fmtfmt 519c⟩≡ (521)

```
static Fmts
fmtfmt(int c)
{
    Convfmt *p, *ep;

    ep = &fmtalloc.fmt[fmtalloc.nfmt];
    for(p=fmtalloc.fmt; p<ep; p++)
        if(p->c == c){
            while(p->fmt == nil)    /* loop until value is updated */
                ;
            return p->fmt;
        }

    /* is this a predefined format char? */
```

```
    _fmtlock();
    for(p=knownfmt; p->c; p++)
        if(p->c == c){
            _fmtinstall(p->c, p->fmt);
            _fmtunlock();
            return p->fmt;
        }
    _fmtunlock();

    return _badfmt;
}
```

Uses _badfmt() 510a, _fmtinstall() 519a, _fmtlock() 523b, _fmtunlock() 523c, fmtalloc 518c, and knownfmt-35 518d.

⟨*function* _fmtdispatch 520⟩≡ (521)

```
void*
_fmtdispatch(Fmt *f, void *fmt, int isrunes)
{
    Rune rune, r;
    int i, n, w, p;
    ulong fl;
    void *ret;

    w = f->width;
    p = f->prec;
    fl = f->flags;

    f->flags = 0;
    f->width = f->prec = 0;

    for(;;){
        if(isrunes){
            r = *(Rune*)fmt;
            fmt = (Rune*)fmt + 1;
        }else{
            fmt = (char*)fmt + chartorune(&rune, fmt);
            r = rune;
        }
        f->r = r;
        switch(r){
        case '\0':
            ret = nil;
            goto end;
        case '.':
            f->flags |= FmtWidth|FmtPrec;
            continue;
        case '0':
            if(!(f->flags & FmtWidth)){
                f->flags |= FmtZero;
                continue;
            }
            /* fall through */
        case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
            i = 0;
            while(r >= '0' && r <= '9'){
                i = i * 10 + r - '0';
                if(isrunes){
                    r = *(Rune*)fmt;
                    fmt = (Rune*)fmt + 1;
                }else{
```

```
                    r = *(char*)fmt;
                    fmt = (char*)fmt + 1;
                }
            }
            if(isrunes)
                fmt = (Rune*)fmt - 1;
            else
                fmt = (char*)fmt - 1;
        numflag:
            if(f->flags & FmtWidth){
                f->flags |= FmtPrec;
                f->prec = i;
            }else{
                f->flags |= FmtWidth;
                f->width = i;
            }
            continue;
        case '*':
            i = va_arg(f->args, int);
            if(i < 0){
                /*
                 * negative precision =>
                 * ignore the precision.
                 */
                if(f->flags & FmtPrec){
                    f->flags &= ~FmtPrec;
                    f->prec = 0;
                    continue;
                }
                i = -i;
                f->flags |= FmtLeft;
            }
            goto numflag;
        }
        n = (*fmtfmt(r))(f);
        if(n < 0){
            ret = nil;
            break;
        }
        if(n == 0){
            ret = fmt;
            break;
        }
    }
end:
    f->width = w;
    f->prec = p;
    f->flags = fl;
    return ret;
}
```

Uses `chartorune()` 84d and `fmtfmt()` 519c.

⟨`libc/fmt/fmt.c` 521⟩≡

```
⟨libc includes 387a⟩
#include "fmtdef.h"

⟨enum _anon_ (fmt/fmt.c) 518a⟩

typedef struct Convfmt Convfmt;
⟨struct Convfmt 518b⟩
```

⟨*global* fmtalloc 518c⟩

⟨*global* knownfmt 518d⟩

⟨*global doquote (fmt/fmt.c)* 518e⟩

⟨*function* _fmtinstall 519a⟩

⟨*function* fmtinstall 519b⟩

⟨*function* fmtfmt 519c⟩

⟨*function* _fmtdispatch 520⟩

Uses Convfmt 518b.

libc/fmt/fmtfd.c

⟨*function* fmtfdflush 522a⟩≡ (522c)

```
/*
 * public routine for final flush of a formatting buffer
 * to a file descriptor; returns total char count.
 */
int
fmtfdflush(Fmt *f)
{
    if(_fmtFdFlush(f) <= 0)
        return -1;
    return f->nfmt;
}
```

Uses _fmtFdFlush() 535e.

⟨*function* fmtfdinit 522b⟩≡ (522c)

```
/*
 * initialize an output buffer for buffered printing
 */
int
fmtfdinit(Fmt *f, int fd, char *buf, int size)
{
    f->runes = 0;
    f->start = buf;
    f->to = buf;
    f->stop = buf + size;
    f->flush = _fmtFdFlush;
    f->farg = (void*)fd;
    f->nfmt = 0;
    return 0;
}
```

Uses _fmtFdFlush() 535e.

⟨libc/fmt/fmtfd.c 522c⟩≡

⟨*libc includes* 387a⟩

```
#include "fmtdef.h"
```

⟨*function* fmtfdflush 522a⟩

⟨*function* fmtfdinit 522b⟩

libc/fmt/fmtlock.c

⟨*global* `fmtl` 523a⟩≡ (523d)
```
static Lock fmtl;
```

⟨*function* `_fmtlock` 523b⟩≡ (523d)
```
void
_fmtlock(void)
{
    lock(&fmtl);
}
```
Uses `fmtl-32` 523a and `lock()` 245b.

⟨*function* `_fmtunlock` 523c⟩≡ (523d)
```
void
_fmtunlock(void)
{
    unlock(&fmtl);
}
```
Uses `fmtl-32` 523a and `unlock()` 245c.

⟨`libc/fmt/fmtlock.c` 523d⟩≡
```
⟨libc includes 387a⟩
⟨global fmtl 523a⟩

⟨function _fmtlock 523b⟩

⟨function _fmtunlock 523c⟩
```

libc/fmt/fmtprint.c

⟨*function* `fmtprint` 523e⟩≡ (523f)
```
/*
 * format a string into the output buffer
 * designed for formats which themselves call fmt
 */
int
fmtprint(Fmt *f, char *fmt, ...)
{
    va_list va;
    int n;

    va_start(va, fmt);
    n = fmtvprint(f, fmt, va);
    va_end(va);
    return n;
}
```
Uses `fmtvprint()` 529d.

⟨`libc/fmt/fmtprint.c` 523f⟩≡
```
⟨libc includes 387a⟩
#include "fmtdef.h"

⟨function fmtprint 523e⟩
```

libc/fmt/fmtquote.c

⟨*function* _quotesetup 524⟩≡ (528c)

```
/*
 * How many bytes of output UTF will be produced by quoting (if necessary) this string?
 * How many runes? How much of the input will be consumed?
 * The parameter q is filled in by _quotesetup.
 * The string may be UTF or Runes (s or r).
 * Return count does not include NUL.
 * Terminate the scan at the first of:
 *  NUL in input
 *  count exceeded in input
 *  count exceeded on output
 * *ninp is set to number of input bytes accepted.
 * nin may be <0 initially, to avoid checking input by count.
 */
void
_quotesetup(char *s, Rune *r, int nin, int nout, Quoteinfo *q, int sharp, int runesout)
{
    int w;
    Rune c;

    q->quoted = 0;
    q->nbytesout = 0;
    q->nrunesout = 0;
    q->nbytesin = 0;
    q->nrunesin = 0;
    if(sharp || nin==0 || (s && *s=='\0') || (r && *r=='\0')){
        if(nout < 2)
            return;
        q->quoted = 1;
        q->nbytesout = 2;
        q->nrunesout = 2;
    }
    for(; nin!=0; nin--){
        if(s)
            w = chartorune(&c, s);
        else{
            c = *r;
            w = runelen(c);
        }

        if(c == '\0')
            break;
        if(runesout){
            if(q->nrunesout+1 > nout)
                break;
        }else{
            if(q->nbytesout+w > nout)
                break;
        }

        if((c <= L' ') || (c == L'\'') || (doquote!=nil && doquote(c))){
            if(!q->quoted){
                if(runesout){
                    if(1+q->nrunesout+1+1 > nout)   /* no room for quotes */
                        break;
                }else{
                    if(1+q->nbytesout+w+1 > nout)   /* no room for quotes */
                        break;
```

```
                }
                q->nrunesout += 2;  /* include quotes */
                q->nbytesout += 2;  /* include quotes */
                q->quoted = 1;
            }
            if(c == '\'')   {
                if(runesout){
                    if(1+q->nrunesout+1 > nout) /* no room for quotes */
                        break;
                }else{
                    if(1+q->nbytesout+w > nout) /* no room for quotes */
                        break;
                }
                q->nbytesout++;
                q->nrunesout++; /* quotes reproduce as two characters */
            }
        }

        /* advance input */
        if(s)
            s += w;
        else
            r++;
        q->nbytesin += w;
        q->nrunesin++;

        /* advance output */
        q->nbytesout += w;
        q->nrunesout++;
    }
}
```

⟨*function* qstrfmt 525⟩≡ (528c)

```
static int
qstrfmt(char *sin, Rune *rin, Quoteinfo *q, Fmt *f)
{
    Rune r, *rm, *rme;
    char *t, *s, *m, *me;
    Rune *rt, *rs;
    ulong fl;
    int nc, w;

    m = sin;
    me = m + q->nbytesin;
    rm = rin;
    rme = rm + q->nrunesin;

    w = f->width;
    fl = f->flags;
    if(f->runes){
        if(!(fl & FmtLeft) && _rfmtpad(f, w - q->nrunesout) < 0)
            return -1;
    }else{
        if(!(fl & FmtLeft) && _fmtpad(f, w - q->nbytesout) < 0)
            return -1;
    }
    t = f->to;
    s = f->stop;
    rt = f->to;
    rs = f->stop;
```

```
    if(f->runes)
        FMTRCHAR(f, rt, rs, '\'');
    else
        FMTRUNE(f, t, s, '\'');
    for(nc = q->nrunesin; nc > 0; nc--){
        if(sin){
            r = *(uchar*)m;
            if(r < Runeself)
                m++;
            else if((me - m) >= UTFmax || fullrune(m, me-m))
                m += chartorune(&r, m);
            else
                break;
        }else{
            if(rm >= rme)
                break;
            r = *(uchar*)rm++;
        }
        if(f->runes){
            FMTRCHAR(f, rt, rs, r);
            if(r == '\'')
                FMTRCHAR(f, rt, rs, r);
        }else{
            FMTRUNE(f, t, s, r);
            if(r == '\'')
                FMTRUNE(f, t, s, r);
        }
    }

    if(f->runes){
        FMTRCHAR(f, rt, rs, '\'');
        USED(rs);
        f->nfmt += rt - (Rune *)f->to;
        f->to = rt;
        if(fl & FmtLeft && _rfmtpad(f, w - q->nrunesout) < 0)
            return -1;
    }else{
        FMTRUNE(f, t, s, '\'');
        USED(s);
        f->nfmt += t - (char *)f->to;
        f->to = t;
        if(fl & FmtLeft && _fmtpad(f, w - q->nbytesout) < 0)
            return -1;
    }
    return 0;
}
```

Uses FMTRCHAR 500d, FMTRUNE 500d, _fmtpad() 502b, _rfmtpad() 502c, chartorune() 84d, and fullrune() 436c.

⟨*function* _quotestrfmt 526⟩≡ (528c)

```
int
_quotestrfmt(int runesin, Fmt *f)
{
    int nin, outlen;
    Rune *r;
    char *s;
    Quoteinfo q;

    nin = -1;
    if(f->flags&FmtPrec)
        nin = f->prec;
```

```
	if(runesin){
		r = va_arg(f->args, Rune *);
		s = nil;
	}else{
		s = va_arg(f->args, char *);
		r = nil;
	}
	if(!s && !r)
		return _fmtcpy(f, "<nil>", 5, 5);

	if(f->flush)
		outlen = 0x7FFFFFFF;	/* if we can flush, no output limit */
	else if(f->runes)
		outlen = (Rune*)f->stop - (Rune*)f->to;
	else
		outlen = (char*)f->stop - (char*)f->to;

	_quotesetup(s, r, nin, outlen, &q, f->flags&FmtSharp, f->runes);
//print("bytes in %d bytes out %d runes in %d runesout %d\n", q.nbytesin, q.nbytesout, q.nrunesin, q.nrunesout);

	if(runesin){
		if(!q.quoted)
			return _fmtrcpy(f, r, q.nrunesin);
		return qstrfmt(nil, r, &q, f);
	}

	if(!q.quoted)
		return _fmtcpy(f, s, q.nrunesin, q.nbytesin);
	return qstrfmt(s, nil, &q, f);
}
```

Uses `_fmtcpy()` 503a, `_fmtrcpy()` 503b, and `qstrfmt()` 525.

⟨*function* `quotestrfmt` 527a⟩≡ (528c)

```
int
quotestrfmt(Fmt *f)
{
	return _quotestrfmt(0, f);
}
```

Uses `_quotestrfmt()` 526.

⟨*function* `quoterunestrfmt` 527b⟩≡ (528c)

```
int
quoterunestrfmt(Fmt *f)
{
	return _quotestrfmt(1, f);
}
```

Uses `_quotestrfmt()` 526.

⟨*function* `quotefmtinstall` 527c⟩≡ (528c)

```
void
quotefmtinstall(void)
{
	fmtinstall('q', quotestrfmt);
	fmtinstall('Q', quoterunestrfmt);
}
```

Uses `fmtinstall()` 519b, `quoterunestrfmt()` 527b, and `quotestrfmt()` 527a.

⟨*function* _needsquotes 528a⟩≡ (528c)
```
int
_needsquotes(char *s, int *quotelenp)
{
    Quoteinfo q;

    _quotesetup(s, nil, -1, 0x7FFFFFFF, &q, 0, 0);
    *quotelenp = q.nbytesout;

    return q.quoted;
}
```

⟨*function* _runeneedsquotes 528b⟩≡ (528c)
```
int
_runeneedsquotes(Rune *r, int *quotelenp)
{
    Quoteinfo q;

    _quotesetup(nil, r, -1, 0x7FFFFFFF, &q, 0, 0);
    *quotelenp = q.nrunesout;

    return q.quoted;
}
```

⟨libc/fmt/fmtquote.c 528c⟩≡
```
⟨libc includes 387a⟩
#include "fmtdef.h"

⟨function _quotesetup 524⟩

⟨function qstrfmt 525⟩

⟨function _quotestrfmt 526⟩

⟨function quotestrfmt 527a⟩

⟨function quoterunestrfmt 527b⟩

⟨function quotefmtinstall 527c⟩

⟨function _needsquotes 528a⟩

⟨function _runeneedsquotes 528b⟩
```

libc/fmt/fmtrune.c

⟨*function* fmtrune 528d⟩≡ (529a)
```
int
fmtrune(Fmt *f, int r)
{
    Rune *rt;
    char *t;
    int n;

    if(f->runes){
        rt = f->to;
        FMTRCHAR(f, rt, f->stop, r);
        f->to = rt;
        n = 1;
```

```
    }else{
        t = f->to;
        FMTRUNE(f, t, f->stop, r);
        n = t - (char*)f->to;
        f->to = t;
    }
    f->nfmt += n;
    return 0;
}
```

Uses `FMTRCHAR` 500d and `FMTRUNE` 500d.

⟨`libc/fmt/fmtrune.c` 529a⟩≡

```
⟨libc includes 387a⟩
#include "fmtdef.h"

⟨function fmtrune 528d⟩
```

libc/fmt/fmtstr.c

⟨*function* `fmtstrflush` 529b⟩≡ (529c)

```
char*
fmtstrflush(Fmt *f)
{
    if(f->start == nil)
        return nil;
    *(char*)f->to = '\0';
    return f->start;
}
```

⟨`libc/fmt/fmtstr.c` 529c⟩≡

```
⟨libc includes 387a⟩
⟨function fmtstrflush 529b⟩
```

libc/fmt/fmtvprint.c

⟨*function* `fmtvprint` 529d⟩≡ (530a)

```
/*
 * format a string into the output buffer
 * designed for formats which themselves call fmt
 */
int
fmtvprint(Fmt *f, char *fmt, va_list args)
{
    va_list va;
    int n;

    va = f->args;
    f->args = args;
    n = dofmt(f, fmt);
    f->args = va;
    if(n >= 0)
        return 0;
    return n;
}
```

Uses `dofmt()` 74.

⟨libc/fmt/fmtvprint.c 530a⟩≡
```
⟨libc includes 387a⟩
#include "fmtdef.h"

⟨function fmtvprint 529d⟩
```

libc/fmt/fprint.c

⟨libc/fmt/fprint.c 530b⟩≡
```
⟨libc includes 387a⟩
⟨function fprint 75a⟩
```

libc/fmt/print.c

⟨libc/fmt/print.c 530c⟩≡
```
⟨libc includes 387a⟩
⟨function libc_print 72c⟩

⟨global print 72b⟩
```

libc/fmt/runefmtstr.c

⟨*function* runefmtstrflush 530d⟩≡ (530e)
```
Rune*
runefmtstrflush(Fmt *f)
{
    if(f->start == nil)
        return nil;
    *(Rune*)f->to = '\0';
    return f->start;
}
```

⟨libc/fmt/runefmtstr.c 530e⟩≡
```
⟨libc includes 387a⟩
⟨function runefmtstrflush 530d⟩
```

libc/fmt/runeseprint.c

⟨*function* runeseprint 530f⟩≡ (530g)
```
Rune*
runeseprint(Rune *buf, Rune *e, char *fmt, ...)
{
    Rune *p;
    va_list args;

    va_start(args, fmt);
    p = runevseprint(buf, e, fmt, args);
    va_end(args);
    return p;
}
```
Uses runevseprint() 532a.

⟨libc/fmt/runeseprint.c 530g⟩≡
```
⟨libc includes 387a⟩
⟨function runeseprint 530f⟩
```

libc/fmt/runesmprint.c

⟨*function* runesmprint 531a⟩≡ (531b)

```
Rune*
runesmprint(char *fmt, ...)
{
    va_list args;
    Rune *p;

    va_start(args, fmt);
    p = runevsmprint(fmt, args);
    va_end(args);
    return p;
}
```

Uses runevsmprint() 533b.

⟨libc/fmt/runesmprint.c 531b⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* runesmprint 531a⟩

libc/fmt/runesnprint.c

⟨*function* runesnprint 531c⟩≡ (531d)

```
int
runesnprint(Rune *buf, int len, char *fmt, ...)
{
    int n;
    va_list args;

    va_start(args, fmt);
    n = runevsnprint(buf, len, fmt, args);
    va_end(args);
    return n;
}
```

Uses runevsnprint() 534a.

⟨libc/fmt/runesnprint.c 531d⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* runesnprint 531c⟩

libc/fmt/runesprint.c

⟨*function* runesprint 531e⟩≡ (531f)

```
int
runesprint(Rune *buf, char *fmt, ...)
{
    int n;
    va_list args;

    va_start(args, fmt);
    n = runevsnprint(buf, 256, fmt, args);
    va_end(args);
    return n;
}
```

Uses runevsnprint() 534a.

⟨libc/fmt/runesprint.c 531f⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* runesprint 531e⟩

libc/fmt/runevseprint.c

⟨*function* runevseprint 532a⟩≡ (532b)

```
Rune*
runevseprint(Rune *buf, Rune *e, char *fmt, va_list args)
{
    Fmt f;

    if(e <= buf)
        return nil;
    f.runes = 1;
    f.start = buf;
    f.to = buf;
    f.stop = e - 1;
    f.flush = nil;
    f.farg = nil;
    f.nfmt = 0;
    f.args = args;
    dofmt(&f, fmt);
    *(Rune*)f.to = '\0';
    return f.to;
}
```

Uses `dofmt()` 74.

⟨libc/fmt/runevseprint.c 532b⟩≡

⟨*libc includes* 387a⟩
⟨*function* runevseprint 532a⟩

libc/fmt/runevsmprint.c

⟨*function* runeFmtStrFlush 532c⟩≡ (533c)

```
static int
runeFmtStrFlush(Fmt *f)
{
    Rune *s;
    int n;

    if(f->start == nil)
        return 0;
    n = (int)(uintptr)f->farg;
    n *= 2;
    s = f->start;
    f->start = realloc(s, sizeof(Rune)*n);
    if(f->start == nil){
        f->farg = nil;
        f->to = nil;
        f->stop = nil;
        free(s);
        return 0;
    }
    f->farg = (void*)n;
    f->to = (Rune*)f->start + ((Rune*)f->to - s);
    f->stop = (Rune*)f->start + n - 1;
    return 1;
}
```

Uses `free()` 63i and `realloc()` 67b.

⟨*function* runefmtstrinit 533a⟩≡ (533c)

```
int
runefmtstrinit(Fmt *f)
{
    int n;

    memset(f, 0, sizeof *f);
    f->runes = 1;
    n = 32;
    f->start = malloc(sizeof(Rune)*n);
    if(f->start == nil)
        return -1;
    setmalloctag(f->start, getcallerpc(&f));
    f->to = f->start;
    f->stop = (Rune*)f->start + n - 1;
    f->flush = runeFmtStrFlush;
    f->farg = (void*)n;
    f->nfmt = 0;
    return 0;
}
```

Uses `getcallerpc()` 396f, `malloc()` 63g, `memset()` 46b, `runeFmtStrFlush()` 532c, and `setmalloctag()` 69a.

⟨*function* runevsmprint 533b⟩≡ (533c)

```
/*
 * print into an allocated string buffer
 */
Rune*
runevsmprint(char *fmt, va_list args)
{
    Fmt f;
    int n;

    if(runefmtstrinit(&f) < 0)
        return nil;
    f.args = args;
    n = dofmt(&f, fmt);
    if(f.start == nil)      /* realloc failed? */
        return nil;
    if(n < 0){
        free(f.start);
        return nil;
    }
    setmalloctag(f.start, getcallerpc(&fmt));
    *(Rune*)f.to = '\0';
    return f.start;
}
```

Uses `dofmt()` 74, `free()` 63i, `getcallerpc()` 396f, `runefmtstrinit()` 533a, and `setmalloctag()` 69a.

⟨`libc/fmt/runevsmprint.c` 533c⟩≡

```
⟨libc includes 387a⟩
#include "fmtdef.h"

⟨function runeFmtStrFlush 532c⟩

⟨function runefmtstrinit 533a⟩

⟨function runevsmprint 533b⟩
```

libc/fmt/runevsnprint.c

⟨*function* runevsnprint 534a⟩≡ (534b)

```
int
runevsnprint(Rune *buf, int len, char *fmt, va_list args)
{
    Fmt f;

    if(len <= 0)
        return -1;
    f.runes = 1;
    f.start = buf;
    f.to = buf;
    f.stop = buf + len - 1;
    f.flush = nil;
    f.farg = nil;
    f.nfmt = 0;
    f.args = args;
    dofmt(&f, fmt);
    *(Rune*)f.to = '\0';
    return (Rune*)f.to - buf;
}
```

Uses `dofmt()` 74.

⟨libc/fmt/runevsnprint.c 534b⟩≡

⟨*libc includes* 387a⟩
⟨*function* runevsnprint 534a⟩

libc/fmt/seprint.c

⟨*function* seprint 534c⟩≡ (534d)

```
char*
seprint(char *buf, char *e, char *fmt, ...)
{
    char *p;
    va_list args;

    va_start(args, fmt);
    p = vseprint(buf, e, fmt, args);
    va_end(args);
    return p;
}
```

Uses `vseprint()` 536b.

⟨libc/fmt/seprint.c 534d⟩≡

⟨*libc includes* 387a⟩
⟨*function* seprint 534c⟩

libc/fmt/smprint.c

⟨*function* smprint 534e⟩≡ (535a)

```
char*
smprint(char *fmt, ...)
{
    va_list args;
    char *p;

    va_start(args, fmt);
```

```
    p = vsmprint(fmt, args);
    va_end(args);
    setmalloctag(p, getcallerpc(&fmt));
    return p;
}
```

Uses `getcallerpc()` 396f, `setmalloctag()` 69a, and `vsmprint()` 537b.

⟨`libc/fmt/smprint.c` 535a⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* `smprint` 534e⟩

libc/fmt/snprint.c

⟨*function* `snprint` 535b⟩≡ (535c)

```
int
snprint(char *buf, int len, char *fmt, ...)
{
    int n;
    va_list args;

    va_start(args, fmt);
    n = vsnprint(buf, len, fmt, args);
    va_end(args);
    return n;
}
```

Uses `vsnprint()` 75c.

⟨`libc/fmt/snprint.c` 535c⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* `snprint` 535b⟩

libc/fmt/sprint.c

⟨`libc/fmt/sprint.c` 535d⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* `sprint` 75b⟩

libc/fmt/vfprint.c

⟨*function* `_fmtFdFlush` 535e⟩≡ (536a)

```
/*
 * generic routine for flushing a formatting buffer
 * to a file descriptor
 */
int
_fmtFdFlush(Fmt *f)
{
    int n;

    n = (char*)f->to - (char*)f->start;
    if(n && write((int)(uintptr)f->farg, f->start, n) != n)
        return 0;
    f->to = f->start;
    return 1;
}
```

Uses `write()` 192b.

⟨libc/fmt/vfprint.c 536a⟩≡

```
⟨libc includes 387a⟩
#include "fmtdef.h"

⟨function _fmtFdFlush 535e⟩

⟨function vfprint 73⟩
```

libc/fmt/vseprint.c

⟨*function* vseprint 536b⟩≡ (536c)

```
char*
vseprint(char *buf, char *e, char *fmt, va_list args)
{
    Fmt f;

    if(e <= buf)
        return nil;
    f.runes = 0;
    f.start = buf;
    f.to = buf;
    f.stop = e - 1;
    f.flush = nil;
    f.farg = nil;
    f.nfmt = 0;
    f.args = args;
    dofmt(&f, fmt);
    *(char*)f.to = '\0';
    return f.to;
}
```

Uses dofmt() 74.

⟨libc/fmt/vseprint.c 536c⟩≡

```
⟨libc includes 387a⟩
⟨function vseprint 536b⟩
```

libc/fmt/vsmprint.c

⟨*function* fmtStrFlush 536d⟩≡ (537c)

```
static int
fmtStrFlush(Fmt *f)
{
    char *s;
    int n;

    if(f->start == nil)
        return 0;
    n = (int)(uintptr)f->farg;
    n *= 2;
    s = f->start;
    f->start = realloc(s, n);
    if(f->start == nil){
        f->farg = nil;
        f->to = nil;
        f->stop = nil;
        free(s);
        return 0;
    }
```

```
        f->farg = (void*)n;
        f->to = (char*)f->start + ((char*)f->to - s);
        f->stop = (char*)f->start + n - 1;
        return 1;
    }
```
Uses `free()` 63i and `realloc()` 67b.

⟨*function* `fmtstrinit` 537a⟩≡ (537c)
```
    int
    fmtstrinit(Fmt *f)
    {
        int n;

        memset(f, 0, sizeof *f);
        f->runes = 0;
        n = 32;
        f->start = malloc(n);
        if(f->start == nil)
            return -1;
        setmalloctag(f->start, getcallerpc(&f));
        f->to = f->start;
        f->stop = (char*)f->start + n - 1;
        f->flush = fmtStrFlush;
        f->farg = (void*)n;
        f->nfmt = 0;
        return 0;
    }
```
Uses `fmtStrFlush()` 536d, `getcallerpc()` 396f, `malloc()` 63g, `memset()` 46b, and `setmalloctag()` 69a.

⟨*function* `vsmprint` 537b⟩≡ (537c)
```
    /*
     * print into an allocated string buffer
     */
    char*
    vsmprint(char *fmt, va_list args)
    {
        Fmt f;
        int n;

        if(fmtstrinit(&f) < 0)
            return nil;
        f.args = args;
        n = dofmt(&f, fmt);
        if(f.start == nil)      /* realloc failed? */
            return nil;
        if(n < 0){
            free(f.start);
            return nil;
        }
        setmalloctag(f.start, getcallerpc(&fmt));
        *(char*)f.to = '\0';
        return f.start;
    }
```
Uses `dofmt()` 74, `fmtstrinit()` 537a, `free()` 63i, `getcallerpc()` 396f, and `setmalloctag()` 69a.

⟨`libc/fmt/vsmprint.c` 537c⟩≡
```
    ⟨libc includes 387a⟩
    #include "fmtdef.h"
```

⟨*function* fmtStrFlush 536d⟩

⟨*function* fmtstrinit 537a⟩

⟨*function* vsmprint 537b⟩

libc/fmt/vsnprint.c

⟨libc/fmt/vsnprint.c 538a⟩≡
  ⟨*libc includes* 387a⟩
  ⟨*function* vsnprint 75c⟩

A.5 libc/arm/

libc/arm/main.s

libc/arm/argv0.s

libc/arm/memset.s

libc/arm/memmove.s

libc/arm/cycles.c

⟨*function* cycles*(arm)* 538b⟩≡ (538c)

```
void
cycles(uvlong* u)
{
    *u = 0LL;
}
```

⟨libc/arm/cycles.c 538c⟩≡
  ⟨*libc includes* 387a⟩

```
#pragma profile off
```

  ⟨*function* cycles*(arm)* 538b⟩

libc/arm/notejmp.c

⟨libc/arm/notejmp.c 538d⟩≡
  ⟨*libc includes* 387a⟩

```
#include <ureg.h>
```

  ⟨*function* notejmp*(arm)* 260e⟩

libc/arm/vlrt.c

⟨*macro* SIGN*(arm)* 538e⟩≡ (551d)

```
#define SIGN(n) (1UL<<(n-1))
```

⟨*struct* Vlong*(arm)* 538f⟩≡ (551d)

```
struct  Vlong
{
    ulong   lo;
    ulong   hi;
};
```

⟨*function* _addv *(arm)* 539a⟩≡ (551d)
```
void
_addv(Vlong *r, Vlong a, Vlong b)
{
    ulong lo, hi;

    lo = a.lo + b.lo;
    hi = a.hi + b.hi;
    if(lo < a.lo)
        hi++;
    r->lo = lo;
    r->hi = hi;
}
```

⟨*function* _subv *(arm)* 539b⟩≡ (551d)
```
void
_subv(Vlong *r, Vlong a, Vlong b)
{
    ulong lo, hi;

    lo = a.lo - b.lo;
    hi = a.hi - b.hi;
    if(lo > a.lo)
        hi--;
    r->lo = lo;
    r->hi = hi;
}
```

⟨*function* _d2v *(arm)* 539c⟩≡ (551d)
```
void
_d2v(Vlong *y, double d)
{
    union { double d; struct Vlong; } x;
    ulong xhi, xlo, ylo, yhi;
    int sh;

    x.d = d;

    xhi = (x.hi & 0xfffff) | 0x100000;
    xlo = x.lo;
    sh = 1075 - ((x.hi >> 20) & 0x7ff);

    ylo = 0;
    yhi = 0;
    if(sh >= 0) {
        /* v = (hi||lo) >> sh */
        if(sh < 32) {
            if(sh == 0) {
                ylo = xlo;
                yhi = xhi;
            } else {
                ylo = (xlo >> sh) | (xhi << (32-sh));
                yhi = xhi >> sh;
            }
        } else {
            if(sh == 32) {
                ylo = xhi;
            } else
            if(sh < 64) {
                ylo = xhi >> (sh-32);
```

```
            }
        }
    } else {
        /* v = (hi||lo) << -sh */
        sh = -sh;
        if(sh <= 10) {
            ylo = xlo << sh;
            yhi = (xhi << sh) | (xlo >> (32-sh));
        } else {
            /* overflow */
            yhi = d;    /* causes something awful */
        }
    }
    if(x.hi & SIGN(32)) {
        if(ylo != 0) {
            ylo = -ylo;
            yhi = ~yhi;
        } else
            yhi = -yhi;
    }

    y->hi = yhi;
    y->lo = ylo;
}
```

Uses SIGN-223 538e.

⟨*function* _f2v *(arm)* 540a⟩≡ (551d)

```
void
_f2v(Vlong *y, float f)
{
    _d2v(y, f);
}
```

⟨*function* _v2d *(arm)* 540b⟩≡ (551d)

```
double
_v2d(Vlong x)
{
    if(x.hi & SIGN(32)) {
        if(x.lo) {
            x.lo = -x.lo;
            x.hi = ~x.hi;
        } else
            x.hi = -x.hi;
        return -((long)x.hi*4294967296. + x.lo);
    }
    return (long)x.hi*4294967296. + x.lo;
}
```

Uses SIGN-223 538e.

⟨*function* _v2f *(arm)* 540c⟩≡ (551d)

```
float
_v2f(Vlong x)
{
    return _v2d(x);
}
```

⟨*function* dodiv *(arm)* 540d⟩≡ (551d)

```
static void
dodiv(Vlong num, Vlong den, Vlong *q, Vlong *r)
{
```

```
	ulong numlo, numhi, denhi, denlo, quohi, quolo, t;
	int i;

	numhi = num.hi;
	numlo = num.lo;
	denhi = den.hi;
	denlo = den.lo;
	/*
	 * get a divide by zero
	 */
	if(denlo==0 && denhi==0) {
	    numlo = numlo / denlo;
	}

	/*
	 * set up the divisor and find the number of iterations needed
	 */
	if(numhi >= SIGN(32)) {
	    quohi = SIGN(32);
	    quolo = 0;
	} else {
	    quohi = numhi;
	    quolo = numlo;
	}
	i = 0;
	while(denhi < quohi || (denhi == quohi && denlo < quolo)) {
	    denhi = (denhi<<1) | (denlo>>31);
	    denlo <<= 1;
	    i++;
	}

	quohi = 0;
	quolo = 0;
	for(; i >= 0; i--) {
	    quohi = (quohi<<1) | (quolo>>31);
	    quolo <<= 1;
	    if(numhi > denhi || (numhi == denhi && numlo >= denlo)) {
	        t = numlo;
	        numlo -= denlo;
	        if(numlo > t)
	            numhi--;
	        numhi -= denhi;
	        quolo |= 1;
	    }
	    denlo = (denlo>>1) | (denhi<<31);
	    denhi >>= 1;
	}

	if(q) {
	    q->lo = quolo;
	    q->hi = quohi;
	}
	if(r) {
	    r->lo = numlo;
	    r->hi = numhi;
	}
    }
```
Uses SIGN-223 538e.

⟨*function* _divvu*(arm)* 541⟩≡ (551d)

```
void
_divvu(Vlong *q, Vlong n, Vlong d)
{
    if(n.hi == 0 && d.hi == 0) {
        q->hi = 0;
        q->lo = n.lo / d.lo;
        return;
    }
    dodiv(n, d, q, 0);
}
```

⟨*function* `_modvu`*(arm)* 542a⟩≡ (551d)

```
void
_modvu(Vlong *r, Vlong n, Vlong d)
{

    if(n.hi == 0 && d.hi == 0) {
        r->hi = 0;
        r->lo = n.lo % d.lo;
        return;
    }
    dodiv(n, d, 0, r);
}
```

⟨*function* `vneg`*(arm)* 542b⟩≡ (551d)

```
static void
vneg(Vlong *v)
{

    if(v->lo == 0) {
        v->hi = -v->hi;
        return;
    }
    v->lo = -v->lo;
    v->hi = ~v->hi;
}
```

⟨*function* `_divv`*(arm)* 542c⟩≡ (551d)

```
void
_divv(Vlong *q, Vlong n, Vlong d)
{
    long nneg, dneg;

    if(n.hi == (((long)n.lo)>>31) && d.hi == (((long)d.lo)>>31)) {
        q->lo = (long)n.lo / (long)d.lo;
        q->hi = ((long)q->lo) >> 31;
        return;
    }
    nneg = n.hi >> 31;
    if(nneg)
        vneg(&n);
    dneg = d.hi >> 31;
    if(dneg)
        vneg(&d);
    dodiv(n, d, q, 0);
    if(nneg != dneg)
        vneg(q);
}
```

⟨*function* _modv *(arm)* 543a⟩≡ (551d)

```
void
_modv(Vlong *r, Vlong n, Vlong d)
{
    long nneg, dneg;

    if(n.hi == (((long)n.lo)>>31) && d.hi == (((long)d.lo)>>31)) {
        r->lo = (long)n.lo % (long)d.lo;
        r->hi = ((long)r->lo) >> 31;
        return;
    }
    nneg = n.hi >> 31;
    if(nneg)
        vneg(&n);
    dneg = d.hi >> 31;
    if(dneg)
        vneg(&d);
    dodiv(n, d, 0, r);
    if(nneg)
        vneg(r);
}
```

⟨*function* _rshav *(arm)* 543b⟩≡ (551d)

```
void
_rshav(Vlong *r, Vlong a, int b)
{
    long t;

    t = a.hi;
    if(b >= 32) {
        r->hi = t>>31;
        if(b >= 64) {
            /* this is illegal re C standard */
            r->lo = t>>31;
            return;
        }
        r->lo = t >> (b-32);
        return;
    }
    if(b <= 0) {
        r->hi = t;
        r->lo = a.lo;
        return;
    }
    r->hi = t >> b;
    r->lo = (t << (32-b)) | (a.lo >> b);
}
```

⟨*function* _rshlv *(arm)* 543c⟩≡ (551d)

```
void
_rshlv(Vlong *r, Vlong a, int b)
{
    ulong t;

    t = a.hi;
    if(b >= 32) {
        r->hi = 0;
        if(b >= 64) {
            /* this is illegal re C standard */
            r->lo = 0;
```

```
            return;
        }
        r->lo = t >> (b-32);
        return;
    }
    if(b <= 0) {
        r->hi = t;
        r->lo = a.lo;
        return;
    }
    r->hi = t >> b;
    r->lo = (t << (32-b)) | (a.lo >> b);
}
```

⟨*function* `_lshv` *(arm)* 544a⟩≡ (551d)

```
void
_lshv(Vlong *r, Vlong a, int b)
{
    ulong t;

    t = a.lo;
    if(b >= 32) {
        r->lo = 0;
        if(b >= 64) {
            /* this is illegal re C standard */
            r->hi = 0;
            return;
        }
        r->hi = t << (b-32);
        return;
    }
    if(b <= 0) {
        r->lo = t;
        r->hi = a.hi;
        return;
    }
    r->lo = t << b;
    r->hi = (t >> (32-b)) | (a.hi << b);
}
```

⟨*function* `_andv` *(arm)* 544b⟩≡ (551d)

```
void
_andv(Vlong *r, Vlong a, Vlong b)
{
    r->hi = a.hi & b.hi;
    r->lo = a.lo & b.lo;
}
```

⟨*function* `_orv` *(arm)* 544c⟩≡ (551d)

```
void
_orv(Vlong *r, Vlong a, Vlong b)
{
    r->hi = a.hi | b.hi;
    r->lo = a.lo | b.lo;
}
```

⟨*function* `_xorv` *(arm)* 544d⟩≡ (551d)

```
void
_xorv(Vlong *r, Vlong a, Vlong b)
{
```

```
    r->hi = a.hi ^ b.hi;
    r->lo = a.lo ^ b.lo;
}
```

⟨*function* _vpp *(arm)* 545a⟩≡ (551d)

```
void
_vpp(Vlong *l, Vlong *r)
{

    l->hi = r->hi;
    l->lo = r->lo;
    r->lo++;
    if(r->lo == 0)
        r->hi++;
}
```

⟨*function* _vmm *(arm)* 545b⟩≡ (551d)

```
void
_vmm(Vlong *l, Vlong *r)
{

    l->hi = r->hi;
    l->lo = r->lo;
    if(r->lo == 0)
        r->hi--;
    r->lo--;
}
```

⟨*function* _ppv *(arm)* 545c⟩≡ (551d)

```
void
_ppv(Vlong *l, Vlong *r)
{

    r->lo++;
    if(r->lo == 0)
        r->hi++;
    l->hi = r->hi;
    l->lo = r->lo;
}
```

⟨*function* _mmv *(arm)* 545d⟩≡ (551d)

```
void
_mmv(Vlong *l, Vlong *r)
{

    if(r->lo == 0)
        r->hi--;
    r->lo--;
    l->hi = r->hi;
    l->lo = r->lo;
}
```

⟨*function* _vasop *(arm)* 545e⟩≡ (551d)

```
void
_vasop(Vlong *ret, void *lv, void fn(Vlong*, Vlong, Vlong), int type, Vlong rv)
{
    Vlong t, u;

    u = *ret;
    switch(type) {
```

```
default:
    abort();
    break;

case 1: /* schar */
    t.lo = *(schar*)lv;
    t.hi = t.lo >> 31;
    fn(&u, t, rv);
    *(schar*)lv = u.lo;
    break;

case 2: /* uchar */
    t.lo = *(uchar*)lv;
    t.hi = 0;
    fn(&u, t, rv);
    *(uchar*)lv = u.lo;
    break;

case 3: /* short */
    t.lo = *(short*)lv;
    t.hi = t.lo >> 31;
    fn(&u, t, rv);
    *(short*)lv = u.lo;
    break;

case 4: /* ushort */
    t.lo = *(ushort*)lv;
    t.hi = 0;
    fn(&u, t, rv);
    *(ushort*)lv = u.lo;
    break;

case 9: /* int */
    t.lo = *(int*)lv;
    t.hi = t.lo >> 31;
    fn(&u, t, rv);
    *(int*)lv = u.lo;
    break;

case 10:    /* uint */
    t.lo = *(uint*)lv;
    t.hi = 0;
    fn(&u, t, rv);
    *(uint*)lv = u.lo;
    break;

case 5: /* long */
    t.lo = *(long*)lv;
    t.hi = t.lo >> 31;
    fn(&u, t, rv);
    *(long*)lv = u.lo;
    break;

case 6: /* ulong */
    t.lo = *(ulong*)lv;
    t.hi = 0;
    fn(&u, t, rv);
    *(ulong*)lv = u.lo;
    break;
```

```
        case 7: /* vlong */
        case 8: /* uvlong */
            fn(&u, *(Vlong*)lv, rv);
            *(Vlong*)lv = u;
            break;
        }
        *ret = u;
    }
```

Uses `abort()` 356d.

⟨*function* `_p2v` *(arm)* 547a⟩≡ (551d)

```
    void
    _p2v(Vlong *ret, void *p)
    {
        long t;

        t = (ulong)p;
        ret->lo = t;
        ret->hi = 0;
    }
```

⟨*function* `_sl2v` *(arm)* 547b⟩≡ (551d)

```
    void
    _sl2v(Vlong *ret, long sl)
    {
        long t;

        t = sl;
        ret->lo = t;
        ret->hi = t >> 31;
    }
```

⟨*function* `_ul2v` *(arm)* 547c⟩≡ (551d)

```
    void
    _ul2v(Vlong *ret, ulong ul)
    {
        long t;

        t = ul;
        ret->lo = t;
        ret->hi = 0;
    }
```

⟨*function* `_si2v` *(arm)* 547d⟩≡ (551d)

```
    void
    _si2v(Vlong *ret, int si)
    {
        long t;

        t = si;
        ret->lo = t;
        ret->hi = t >> 31;
    }
```

⟨*function* `_ui2v` *(arm)* 547e⟩≡ (551d)

```
    void
    _ui2v(Vlong *ret, uint ui)
    {
        long t;
```

```
    t = ui;
    ret->lo = t;
    ret->hi = 0;
}
```

⟨*function* _sh2v *(arm)* 548a⟩≡ (551d)

```
void
_sh2v(Vlong *ret, long sh)
{
    long t;

    t = (sh << 16) >> 16;
    ret->lo = t;
    ret->hi = t >> 31;
}
```

⟨*function* _uh2v *(arm)* 548b⟩≡ (551d)

```
void
_uh2v(Vlong *ret, ulong ul)
{
    long t;

    t = ul & 0xffff;
    ret->lo = t;
    ret->hi = 0;
}
```

⟨*function* _sc2v *(arm)* 548c⟩≡ (551d)

```
void
_sc2v(Vlong *ret, long uc)
{
    long t;

    t = (uc << 24) >> 24;
    ret->lo = t;
    ret->hi = t >> 31;
}
```

⟨*function* _uc2v *(arm)* 548d⟩≡ (551d)

```
void
_uc2v(Vlong *ret, ulong ul)
{
    long t;

    t = ul & 0xff;
    ret->lo = t;
    ret->hi = 0;
}
```

⟨*function* _v2sc *(arm)* 548e⟩≡ (551d)

```
long
_v2sc(Vlong rv)
{
    long t;

    t = rv.lo & 0xff;
    return (t << 24) >> 24;
}
```

⟨*function* _v2uc*(arm)* 549a⟩≡ (551d)
```
long
_v2uc(Vlong rv)
{

    return rv.lo & 0xff;
}
```

⟨*function* _v2sh*(arm)* 549b⟩≡ (551d)
```
long
_v2sh(Vlong rv)
{
    long t;

    t = rv.lo & 0xffff;
    return (t << 16) >> 16;
}
```

⟨*function* _v2uh*(arm)* 549c⟩≡ (551d)
```
long
_v2uh(Vlong rv)
{

    return rv.lo & 0xffff;
}
```

⟨*function* _v2sl*(arm)* 549d⟩≡ (551d)
```
long
_v2sl(Vlong rv)
{

    return rv.lo;
}
```

⟨*function* _v2ul*(arm)* 549e⟩≡ (551d)
```
long
_v2ul(Vlong rv)
{

    return rv.lo;
}
```

⟨*function* _v2si*(arm)* 549f⟩≡ (551d)
```
long
_v2si(Vlong rv)
{

    return rv.lo;
}
```

⟨*function* _v2ui*(arm)* 549g⟩≡ (551d)
```
long
_v2ui(Vlong rv)
{

    return rv.lo;
}
```

⟨*function* _testv *(arm)* 550a⟩≡ (551d)
```
int
_testv(Vlong rv)
{
    return rv.lo || rv.hi;
}
```

⟨*function* _eqv *(arm)* 550b⟩≡ (551d)
```
int
_eqv(Vlong lv, Vlong rv)
{
    return lv.lo == rv.lo && lv.hi == rv.hi;
}
```

⟨*function* _nev *(arm)* 550c⟩≡ (551d)
```
int
_nev(Vlong lv, Vlong rv)
{
    return lv.lo != rv.lo || lv.hi != rv.hi;
}
```

⟨*function* _ltv *(arm)* 550d⟩≡ (551d)
```
int
_ltv(Vlong lv, Vlong rv)
{
    return (long)lv.hi < (long)rv.hi ||
        (lv.hi == rv.hi && lv.lo < rv.lo);
}
```

⟨*function* _lev *(arm)* 550e⟩≡ (551d)
```
int
_lev(Vlong lv, Vlong rv)
{
    return (long)lv.hi < (long)rv.hi ||
        (lv.hi == rv.hi && lv.lo <= rv.lo);
}
```

⟨*function* _gtv *(arm)* 550f⟩≡ (551d)
```
int
_gtv(Vlong lv, Vlong rv)
{
    return (long)lv.hi > (long)rv.hi ||
        (lv.hi == rv.hi && lv.lo > rv.lo);
}
```

⟨*function* _gev *(arm)* 550g⟩≡ (551d)
```
int
_gev(Vlong lv, Vlong rv)
{
    return (long)lv.hi > (long)rv.hi ||
        (lv.hi == rv.hi && lv.lo >= rv.lo);
}
```

⟨*function* _lov *(arm)* 550h⟩≡ (551d)
```
int
_lov(Vlong lv, Vlong rv)
{
    return lv.hi < rv.hi ||
        (lv.hi == rv.hi && lv.lo < rv.lo);
}
```

⟨*function* _lsv *(arm)* 551a⟩≡ (551d)

```
int
_lsv(Vlong lv, Vlong rv)
{
    return lv.hi < rv.hi ||
        (lv.hi == rv.hi && lv.lo <= rv.lo);
}
```

⟨*function* _hiv *(arm)* 551b⟩≡ (551d)

```
int
_hiv(Vlong lv, Vlong rv)
{
    return lv.hi > rv.hi ||
        (lv.hi == rv.hi && lv.lo > rv.lo);
}
```

⟨*function* _hsv *(arm)* 551c⟩≡ (551d)

```
int
_hsv(Vlong lv, Vlong rv)
{
    return lv.hi > rv.hi ||
        (lv.hi == rv.hi && lv.lo >= rv.lo);
}
```

⟨libc/arm/vlrt.c 551d⟩≡

```
// very long runtime support (vlrt)

typedef unsigned long   ulong;
typedef unsigned int    uint;
typedef unsigned short  ushort;
typedef unsigned char   uchar;
typedef signed char schar;

⟨macro SIGN (arm) 538e⟩

typedef struct  Vlong   Vlong;
⟨struct Vlong (arm) 538f⟩

void    abort(void);

/* needed by profiler; can't be profiled */
#pragma profile off

⟨function _addv (arm) 539a⟩

⟨function _subv (arm) 539b⟩

#pragma profile on

⟨function _d2v (arm) 539c⟩

⟨function _f2v (arm) 540a⟩

⟨function _v2d (arm) 540b⟩

⟨function _v2f (arm) 540c⟩

/* too many of these are also needed by profiler; leave them out */
#pragma profile off
```

⟨*function* `dodiv`*(arm)* 540d⟩

⟨*function* `_divvu`*(arm)* 541⟩

⟨*function* `_modvu`*(arm)* 542a⟩

⟨*function* `vneg`*(arm)* 542b⟩

⟨*function* `_divv`*(arm)* 542c⟩

⟨*function* `_modv`*(arm)* 543a⟩

⟨*function* `_rshav`*(arm)* 543b⟩

⟨*function* `_rshlv`*(arm)* 543c⟩

⟨*function* `_lshv`*(arm)* 544a⟩

⟨*function* `_andv`*(arm)* 544b⟩

⟨*function* `_orv`*(arm)* 544c⟩

⟨*function* `_xorv`*(arm)* 544d⟩

⟨*function* `_vpp`*(arm)* 545a⟩

⟨*function* `_vmm`*(arm)* 545b⟩

⟨*function* `_ppv`*(arm)* 545c⟩

⟨*function* `_mmv`*(arm)* 545d⟩

⟨*function* `_vasop`*(arm)* 545e⟩

⟨*function* `_p2v`*(arm)* 547a⟩

⟨*function* `_sl2v`*(arm)* 547b⟩

⟨*function* `_ul2v`*(arm)* 547c⟩

⟨*function* `_si2v`*(arm)* 547d⟩

⟨*function* `_ui2v`*(arm)* 547e⟩

⟨*function* `_sh2v`*(arm)* 548a⟩

⟨*function* `_uh2v`*(arm)* 548b⟩

⟨*function* `_sc2v`*(arm)* 548c⟩

⟨*function* `_uc2v`*(arm)* 548d⟩

⟨*function* `_v2sc`*(arm)* 548e⟩

⟨*function* `_v2uc`*(arm)* 549a⟩

⟨*function* `_v2sh`*(arm)* 549b⟩

⟨*function* _v2uh*(arm)* 549c⟩

⟨*function* _v2sl*(arm)* 549d⟩

⟨*function* _v2ul*(arm)* 549e⟩

⟨*function* _v2si*(arm)* 549f⟩

⟨*function* _v2ui*(arm)* 549g⟩

⟨*function* _testv*(arm)* 550a⟩

⟨*function* _eqv*(arm)* 550b⟩

⟨*function* _nev*(arm)* 550c⟩

⟨*function* _ltv*(arm)* 550d⟩

⟨*function* _lev*(arm)* 550e⟩

⟨*function* _gtv*(arm)* 550f⟩

⟨*function* _gev*(arm)* 550g⟩

⟨*function* _lov*(arm)* 550h⟩

⟨*function* _lsv*(arm)* 551a⟩

⟨*function* _hiv*(arm)* 551b⟩

⟨*function* _hsv*(arm)* 551c⟩

A.6 libbio/

⟨*libbio includes* 553a⟩≡ (557a 556 555 554 553)

```
#include <u.h>
#include <libc.h>
#include <bio.h>
```

libbio/bbuffered.c

⟨libbio/bbuffered.c 553b⟩≡
⟨*libbio includes* 553a⟩
⟨*function* Bbuffered 212a⟩

libbio/bfildes.c

⟨libbio/bfildes.c 553c⟩≡
⟨*libbio includes* 553a⟩
⟨*function* Bfildes 212b⟩

libbio/bflush.c

⟨libbio/bflush.c 553d⟩≡
⟨*libbio includes* 553a⟩
⟨*function* Bflush 210a⟩

libbio/bgetc.c

⟨libbio/bgetc.c 554a⟩≡
  ⟨*libbio includes* 553a⟩
  ⟨*function* Bgetc 200a⟩
  ⟨*function* Bungetc 200b⟩

libbio/bgetd.c

⟨*function* Bgetdf 554b⟩≡ (554d)
```
static int
Bgetdf(void *vp)
{
    int c;
    struct bgetd *bg = vp;

    c = Bgetc(bg->b);
    if(c == Beof)
        bg->eof = 1;
    return c;
}
```
Uses Bgetc() 200a and bgetd 554d.

⟨*function* Bgetd 554c⟩≡ (554d)
```
int
Bgetd(Biobufhdr *bp, double *dp)
{
    double d;
    struct bgetd b;

    b.b = bp;
    b.eof = 0;
    d = charstod(Bgetdf, &b);
    if(b.eof)
        return -1;
    Bungetc(bp);
    *dp = d;
    return 1;
}
```
Uses Bgetdf() 554b, Bungetc() 200b, bgetd 554d, and charstod() 390b.

⟨libbio/bgetd.c 554d⟩≡
  ⟨*libbio includes* 553a⟩
```

struct bgetd
{
    Biobufhdr* b;
    int  eof;
};

```
  ⟨*function* Bgetdf 554b⟩
  ⟨*function* Bgetd 554c⟩

libbio/bgetrune.c

⟨libbio/bgetrune.c 554e⟩≡
  ⟨*libbio includes* 553a⟩
  ⟨*function* Bgetrune 201a⟩
  ⟨*function* Bungetrune 201b⟩

libbio/binit.c

⟨`libbio/binit.c` 555a⟩≡
  ⟨*libbio includes* 553a⟩
  ⟨*global* `wbufs` 197c⟩
  ⟨*global* `atexitflag` 197d⟩

  ⟨*function* `batexit` 198c⟩

  ⟨*function* `deinstall` 198b⟩
  ⟨*function* `install` 198a⟩

  ⟨*function* `Binits` 197b⟩
  ⟨*function* `Binit` 197a⟩
  ⟨*function* `Bopen` 198d⟩

  ⟨*function* `Bterm` 199c⟩

libbio/boffset.c

⟨`libbio/boffset.c` 555b⟩≡
  ⟨*libbio includes* 553a⟩
  ⟨*function* `Boffset` 211⟩

libbio/bprint.c

⟨`libbio/bprint.c` 555c⟩≡
  ⟨*libbio includes* 553a⟩
  ⟨*function* `Bprint` 209a⟩

libbio/bputc.c

⟨`libbio/bputc.c` 555d⟩≡
  ⟨*libbio includes* 553a⟩
  ⟨*function* `Bputc` 207a⟩

libbio/bputrune.c

⟨`libbio/bputrune.c` 555e⟩≡
  ⟨*libbio includes* 553a⟩
  ⟨*function* `Bputrune` 207b⟩

libbio/brdline.c

⟨`libbio/brdline.c` 555f⟩≡
  ⟨*libbio includes* 553a⟩
  ⟨*function* `Brdline` 202⟩

  ⟨*function* `Blinelen` 212c⟩

libbio/brdstr.c

⟨libbio/brdstr.c 556a⟩≡
  ⟨*libbio includes* 553a⟩
  ⟨*function* badd 204a⟩

  ⟨*function* Brdstr 204b⟩

libbio/bread.c

⟨libbio/bread.c 556b⟩≡
  ⟨*libbio includes* 553a⟩
  ⟨*function* Bread 206⟩

libbio/bseek.c

⟨libbio/bseek.c 556c⟩≡
  ⟨*libbio includes* 553a⟩
  ⟨*function* Bseek 210b⟩

libbio/btestprint.c

⟨*function* main*(btestprint.c)* 556d⟩≡ (556e)

```
void
main(int argc, char **argv)
{
    Biobuf b;
    char *s;
    int n;


    n = atoi(argv[1]);
    s = malloc(n+1);
    memset(s, 'a', n);
    s[n] = '\0';
    Binit(&b, 1, OWRITE);
    Bprint(&b, "%s\n", s);
    Bflush(&b);
}
```

Uses Bflush() 210a, Binit() 197a, Bprint() 209a, atoi() 123b, malloc() 63g, and memset() 46b.

⟨libbio/btestprint.c 556e⟩≡
  ⟨*libbio includes* 553a⟩
  ⟨*function* main*(btestprint.c)* 556d⟩

libbio/bvprint.c

⟨libbio/bvprint.c 556f⟩≡
  ⟨*libbio includes* 553a⟩
  ⟨*function* fmtBflush 209b⟩

  ⟨*function* Bvprint 209c⟩

libbio/bwrite.c

⟨libbio/bwrite.c 557a⟩≡
  ⟨*libbio includes* 553a⟩
  ⟨*function* Bwrite 208⟩

A.7 libthread/arm

⟨libthread/arm.c 557b⟩≡
  ⟨*libc includes* 387a⟩
```
#include <thread.h>
#include "threadimpl.h"

/* first argument goes in a register; simplest just to ignore it */
static void
launcherarm(int, void (*f)(void *arg), void *arg)
{
 (*f)(arg);
 threadexits(nil);
}

void
_threadinitstack(Thread *t, void (*f)(void*), void *arg)
{
 ulong *tos;

 tos = (ulong*)&t->stk[t->stksize&~7];
 *--tos = (ulong)arg;
 *--tos = (ulong)f;
 *--tos = 0; /* first arg to launchermips */
 *--tos = 0; /* place to store return PC */

 t->sched[JMPBUFPC] = (ulong)launcherarm+JMPBUFDPC;
 t->sched[JMPBUFSP] = (ulong)tos;
}
```
Uses launcherarm() 557b and threadexits() 321c.

A.8 libthread/x86

libthread/386.c

⟨libthread/386.c 557c⟩≡
  ⟨*libc includes* 387a⟩
```
#include <thread.h>

#include "threadimpl.h"
```
  ⟨*function* launcher386 297a⟩

  ⟨*function* _threadinitstack 297b⟩

A.9 libthread/

libthread/ref.c

```
⟨libthread/ref.c 558a⟩≡
  ⟨libc includes 387a⟩
  #include <thread.h>
  #include "threadimpl.h"

  ⟨function incref 287b⟩
  ⟨function decref 288a⟩
```

libthread/threadimpl.h

```
⟨enum _anon_ (libthread/threadimpl.h) 558b⟩≡                                  (558c)
  enum
  {
  ⟨constant RENDHASH 339b⟩
  ⟨constant Printsize 293a⟩
  ⟨constant NPRIV 295b⟩
  };

⟨libthread/threadimpl.h 558c⟩≡
  /*
   * Some notes on locking:
   *
   *  All the locking woes come from implementing
   *  threadinterrupt (and threadkill).
   *
   *  _threadgetproc()->thread is always a live pointer.
   *  p->threads, p->ready, and _threadrgrp also contain
   *  live thread pointers.  These may only be consulted
   *  while holding p->lock or _threadrgrp.lock; in procs
   *  other than p, the pointers are only guaranteed to be live
   *  while the lock is still being held.
   *
   *  Thread structures can only be freed by the proc
   *  they belong to.  Threads marked with t->inrendez
   *  need to be extracted from the _threadrgrp before
   *  being freed.
   *
   *  _threadrgrp.lock cannot be acquired while holding p->lock.
   */

  typedef struct Pqueue   Pqueue;
  typedef struct Rgrp Rgrp;
  typedef struct Tqueue   Tqueue;
  typedef struct Thread   Thread;
  typedef struct Execargs Execargs;
  typedef struct Proc Proc;


  /* must match list in sched.c */
  ⟨enum state 298c⟩
  typedef enum state State;

  ⟨enum chanstate 298d⟩
  typedef enum chanstate Chanstate;
```

⟨*enum* _anon_ (libthread/threadimpl.h) 558b⟩

⟨*struct* Rgrp 339c⟩

⟨*struct* Tqueue 298b⟩

⟨*struct* Thread 295a⟩

⟨*struct* Execargs 294c⟩

⟨*struct* Proc 292⟩

⟨*struct* Pqueue 293f⟩

⟨*struct* Ioproc 315d⟩

```
void    _freeproc(Proc*);
void    _freethread(Thread*);
Proc*   _newproc(void(*)(void*), void*, uint, char*, int, int);
int _procsplhi(void);
void    _procsplx(int);
void    _sched(void);
int _schedexec(Execargs*);
void    _schedexecwait(void);
void    _schedexit(Proc*);
int _schedfork(Proc*);
void    _schedinit(void*);
void    _systhreadinit(void);
void    _threadassert(char*);
void    _threadbreakrendez(void);
void    _threaddebug(ulong, char*, ...);
void    _threadexitsall(char*);
void    _threadflagrendez(Thread*);
Proc*   _threadgetproc(void);
void    _threadsetproc(Proc*);
void    _threadinitstack(Thread*, void(*)(void*), void*);
void*   _threadmalloc(long, int);
void    _threadnote(void*, char*);
void    _threadready(Thread*);
void*   _threadrendezvous(void*, void*);
void    _threadsignal(void);
void    _threadsysfatal(char*, va_list);
void**  _workerdata(void);
void    _xinc(long*);
long    _xdec(long*);

extern int      _threaddebuglevel;
extern char*        _threadexitsallstatus;
extern Pqueue       _threadpq;
extern Channel*     _threadwaitchan;
extern Rgrp     _threadrgrp;
```

⟨*constant* DBGAPPL 334a⟩
⟨*constant* DBGSCHED 334b⟩
⟨*constant* DBGCHAN 334c⟩
⟨*constant* DBGREND 334d⟩
⟨*constant* DBGNOTE 334e⟩
⟨*constant* DBGEXEC 334f⟩

⟨*function* ioproc_arg 315c⟩

Uses Execargs 294c, Pqueue 293f, Proc 292, Rgrp 339c, Thread 295a, Tqueue 298b, chanstate 298d, and state 298c.

libthread/xincarm.c

```
⟨libthread/xincarm.c 560a⟩≡
  #include "xincport.h"
```

libthread/xincport.h

```
⟨libthread/xincport.h 560b⟩≡
  ⟨libc includes 387a⟩
  #include <thread.h>

  ⟨global xincport_lock 288b⟩

  ⟨function _xinc 288c⟩
  ⟨function _xdec 288d⟩
```

libthread/globals.c

```
⟨libthread/globals.c 560c⟩≡
  ⟨libc includes 387a⟩
  #include <thread.h>
  #include "threadimpl.h"

  ⟨global _threadpq 293e⟩

  ⟨global procp 293b⟩

  ⟨function _systhreadinit 301a⟩

  ⟨function _threadgetproc 293c⟩

  ⟨function _threadsetproc 293d⟩
```

libthread/debug.c

```
⟨libthread/debug.c 560d⟩≡
  ⟨libc includes 387a⟩
  #include <thread.h>
  #include "threadimpl.h"

  ⟨global _threaddebuglevel 333c⟩

  ⟨function _threaddebug 334g⟩

  ⟨function _threadassert 333b⟩
```

libthread/id.c

```
⟨libthread/id.c 560e⟩≡
  ⟨libc includes 387a⟩
  #include <thread.h>
  #include "threadimpl.h"
  #include <tos.h>
```

```
  ⟨function threadid 295c⟩

  ⟨function threadpid 298a⟩

  ⟨function threadsetgrp 337a⟩

  ⟨function threadgetgrp 337b⟩

  ⟨function threadsetname 299a⟩

  ⟨function threadgetname 299b⟩

  ⟨function threaddata 336d⟩

  ⟨function _workerdata 336e⟩

  ⟨function procdata 336f⟩

  ⟨global privlock (libthread/id.c) 335c⟩
  ⟨global privmask 335d⟩

  ⟨function tprivalloc 336a⟩

  ⟨function tprivfree 336b⟩

  ⟨function tprivaddr 336c⟩
```

libthread/exit.c

```
⟨libthread/exit.c 561a⟩≡
  ⟨libc includes 387a⟩
  #include <thread.h>
  #include "threadimpl.h"
  #include <tos.h>

  ⟨global _threadexitsallstatus 321b⟩
  ⟨global _threadwaitchan 322a⟩

  ⟨function threadexits 321c⟩

  ⟨function threadexitsall 321d⟩

  ⟨function threadwaitchan 322b⟩
```

libthread/lib.c

```
⟨libthread/lib.c 561b⟩≡
  ⟨libc includes 387a⟩
  #include <thread.h>
  #include "threadimpl.h"

  ⟨global totalmalloc 320e⟩

  ⟨function _threadmalloc 321a⟩

  ⟨function _threadsysfatal 333a⟩
```

libthread/note.c

```
⟨libthread/note.c 562a⟩≡
  ⟨libc includes 387a⟩
  #include <thread.h>
  #include "threadimpl.h"

  typedef struct Note Note;

  ⟨global _threadnopasser 326a⟩

  ⟨constant NFN (libthread/note.c) 326b⟩
  ⟨constant ERRLEN 326c⟩
  ⟨struct Note 326d⟩

  ⟨global notes 326e⟩
  ⟨global enotes 326f⟩
  ⟨global onnote 326g⟩
  ⟨global onnotepid 326h⟩
  ⟨global onnotelock 326i⟩

  ⟨function threadnotify 326j⟩

  ⟨function delayednotes 327a⟩

  ⟨function _threadnote 327b⟩

  ⟨function _procsplhi 328a⟩

  ⟨function _procsplx 328b⟩
```

Uses Note 326d.

libthread/rendez.c

```
⟨libthread/rendez.c 562b⟩≡
  ⟨libc includes 387a⟩
  #include <thread.h>
  #include "threadimpl.h"

  ⟨global _threadrgrp 339d⟩
  ⟨global isdirty 340a⟩

  ⟨function finish 340b⟩

  ⟨function _threadrendezvous 339e⟩

  ⟨function _threadflagrendez 340c⟩

  ⟨function _threadbreakrendez 340d⟩
```

libthread/sched.c

```
⟨libthread/sched.c 562c⟩≡
  ⟨libc includes 387a⟩
  #include <thread.h>
  #include "threadimpl.h"
  #include <tos.h>
```

```
static Thread   *runthread(Proc*);

⟨global _psstate 302a⟩

⟨function psstate 302b⟩

⟨function _schedinit 302c⟩

⟨function needstack 303⟩

⟨function _sched 304a⟩

⟨function runthread 304b⟩

⟨function _threadready 305a⟩

⟨function yield 305b⟩
```

libthread/create.c

```
⟨libthread/create.c 563a⟩≡
  ⟨libc includes 387a⟩
  #include <thread.h>
  #include "threadimpl.h"

  ⟨function nextID 296a⟩

  ⟨function newthread 296c⟩

  ⟨function threadcreate 296b⟩

  ⟨function _newproc 294a⟩

  ⟨function procrfork 293h⟩

  ⟨function proccreate 293g⟩

  ⟨function _freeproc 294b⟩

  ⟨function _freethread 297c⟩
```

libthread/channel.c

```
⟨libthread/channel.c 563b⟩≡
  ⟨libc includes 387a⟩
  #include <thread.h>
  #include "threadimpl.h"

  ⟨enum _anon_ (libthread/channel.c) 290c⟩

  ⟨global errcl 309a⟩
  ⟨global chanlock 289a⟩

  static void enqueue(Alt*, Channel**);
  static void dequeue(Alt*);
  static int canexec(Alt*);
  static int altexec(Alt*, int);
```

⟨*constant* `Closed` 290d⟩
⟨*constant* `Intred` 325b⟩

⟨*function* `_chanfree` 290b⟩

⟨*function* `chanfree` 290a⟩

⟨*function* `chaninit` 289c⟩

⟨*function* `chancreate` 289b⟩

⟨*function* `isopenfor` 308a⟩

⟨*function* `alt` 306⟩

⟨*function* `chanclose` 290e⟩

⟨*function* `chanclosing` 291⟩

⟨*function* `runop` 312⟩

⟨*function* `recv` 313b⟩

⟨*function* `nbrecv` 313c⟩

⟨*function* `send` 313a⟩

⟨*function* `nbsend` 313d⟩

⟨*function* `channelsize` 313e⟩

⟨*function* `sendul` 314a⟩

⟨*function* `recvul` 314b⟩

⟨*function* `sendp` 314c⟩

⟨*function* `recvp` 314d⟩

⟨*function* `nbsendul` 314e⟩

⟨*function* `nbrecvul` 314f⟩

⟨*function* `nbsendp` 315a⟩

⟨*function* `nbrecvp` 315b⟩

⟨*function* `emptyentry` 310a⟩

⟨*function* `enqueue` 309b⟩

⟨*function* `dequeue` 309c⟩

⟨*function* `canexec` 308b⟩

⟨*function* `altexecbuffered` 311a⟩

⟨*function* `altcopy` 311b⟩

⟨*function* `altexec` 310b⟩

libthread/main.c

```
⟨libthread/main.c 565a⟩≡
  ⟨libc includes 387a⟩
  #include <thread.h>
  #include "threadimpl.h"

  typedef struct Mainarg Mainarg;

  ⟨struct Mainarg 301b⟩

  ⟨global mainstacksize 300d⟩
  ⟨global _threadnotefd 334h⟩
  ⟨global _threadpasserpid 334i⟩

  ⟨global _mainjmp 300c⟩

  static void mainlauncher(void*);
  extern void (*_sysfatal)(char*, va_list);
  extern void (*__assert)(char*);
  extern int (*_dial)(char*, char*, char*, int*);

  extern int _threaddial(char*, char*, char*, int*);

  ⟨global mainp 300b⟩

  ⟨function main 300e⟩

  ⟨function mainlauncher 301c⟩

  ⟨function skip (libthread/main.c) 335b⟩

  ⟨function _times 335a⟩

  ⟨function efork 323b⟩

  ⟨function _schedexec 323a⟩

  ⟨function _schedfork 322c⟩

  ⟨function _schedexit 305c⟩

  ⟨function _schedexecwait 323c⟩
```

Uses `Mainarg` 301b.

libthread/kill.c

```
⟨libthread/kill.c 565b⟩≡
  ⟨libc includes 387a⟩
  #include <thread.h>
  #include "threadimpl.h"

  static void tinterrupt(Proc*, Thread*);

  ⟨function threadxxxgrp 338a⟩

  ⟨function threadxxx 338b⟩

  ⟨function threadkillgrp 337d⟩
```

⟨*function* threadkill 337e⟩

⟨*function* threadintgrp 338c⟩

⟨*function* threadint 339a⟩

⟨*function* tinterrupt 337c⟩

libthread/ioproc.c

⟨*enum* _anon_ (libthread/ioproc.c) 566a⟩≡ (566b)
```
enum
{
    ⟨constant STACK 316a⟩
};
```

⟨libthread/ioproc.c 566b⟩≡
```
⟨libc includes 387a⟩
#include <thread.h>
#include "threadimpl.h"

⟨enum _anon_ (libthread/ioproc.c) 566a⟩

⟨function iointerrupt 316d⟩

⟨function xioproc 316e⟩

⟨function ioproc 316b⟩

⟨function closeioproc 316c⟩
```

libthread/iocall.c

⟨libthread/iocall.c 566c⟩≡
```
⟨libc includes 387a⟩
#include <thread.h>
#include "threadimpl.h"

⟨function iocall 317⟩
```

libthread/chanprint.c

⟨libthread/chanprint.c 566d⟩≡
```
⟨libc includes 387a⟩
#include <thread.h>

⟨function chanprint 341⟩
```

libthread/ioclose.c

⟨libthread/ioclose.c 566e⟩≡
```
⟨libc includes 387a⟩
#include <thread.h>
#include "threadimpl.h"
```

```
  ⟨function _ioclose 319a⟩

  ⟨function ioclose 318c⟩
```

libthread/iodial.c

```
⟨libthread/iodial.c 567a⟩≡
  ⟨libc includes 387a⟩
  #include <thread.h>
  #include "threadimpl.h"

  ⟨function _iodial 328c⟩

  ⟨function iodial 329a⟩
```

libthread/ioopen.c

```
⟨libthread/ioopen.c 567b⟩≡
  ⟨libc includes 387a⟩
  #include <thread.h>
  #include "threadimpl.h"

  ⟨function _ioopen 318b⟩

  ⟨function ioopen 318a⟩
```

libthread/ioread.c

```
⟨libthread/ioread.c 567c⟩≡
  ⟨libc includes 387a⟩
  #include <thread.h>
  #include "threadimpl.h"

  ⟨function _ioread 319c⟩

  ⟨function ioread 319b⟩
```

libthread/ioreadn.c

```
⟨libthread/ioreadn.c 567d⟩≡
  ⟨libc includes 387a⟩
  #include <thread.h>
  #include "threadimpl.h"

  ⟨function _ioreadn 319e⟩

  ⟨function ioreadn 319d⟩
```

libthread/iosleep.c

```
⟨libthread/iosleep.c 567e⟩≡
  ⟨libc includes 387a⟩
  #include <thread.h>
  #include "threadimpl.h"
```

⟨*function* _iosleep 320d⟩

⟨*function* iosleep 320c⟩

libthread/iowrite.c

```
⟨libthread/iowrite.c 568a⟩≡
  ⟨libc includes 387a⟩
  #include <thread.h>
  #include "threadimpl.h"

  ⟨function _iowrite 320b⟩

  ⟨function iowrite 320a⟩
```

libthread/dial.c

```
⟨libthread/dial.c 568b⟩≡
  /*
   * old single-process version of dial that libthread can cope with
   */
  ⟨libc includes 387a⟩
  typedef struct DS DS;

  static int  call(char*, char*, DS*);
  static int  csdial(DS*);
  static void _dial_string_parse(char*, DS*);

  ⟨enum _anon_ (libthread/dial.c) 329b⟩

  ⟨struct DS (libthread/dial.c) 329c⟩


  ⟨function _threaddial 329d⟩

  ⟨function csdial (libthread/dial.c) 330⟩

  ⟨function call (libthread/dial.c) 331⟩

  ⟨function _dial_string_parse (libthread/dial.c) 332⟩
```

Uses DS 329c.

libthread/exec.c

```
⟨libthread/exec.c 568c⟩≡
  ⟨libc includes 387a⟩
  #include <thread.h>
  #include "threadimpl.h"

  ⟨constant PIPEMNT 324a⟩

  ⟨function procexec 324b⟩

  ⟨function procexecl 325a⟩
```

A.10 libregexp/

⟨*libregexp includes* 569a⟩≡ (584a 583 581c 572a)
```
  ⟨libc includes 387a⟩
  #include "regexp.h"
```

libregexp/regcomp.h

⟨libregexp/regcomp.h 569b⟩≡
```
/*
 *  substitution list
 */
#define NSUBEXP 32
typedef struct Resublist Resublist;
struct Resublist
{
 Resub m[NSUBEXP];
};

/*
 * Actions and Tokens (Reinst types)
 *
 * 02xx are operators, value == precedence
 * 03xx are tokens, i.e. operands for operators
 */
#define RUNE  0177
#define OPERATOR 0200 /* Bitmask of all operators */
#define START  0200 /* Start, used for marker on stack */
#define RBRA  0201 /* Right bracket, ) */
#define LBRA  0202 /* Left bracket, ( */
#define OR  0203 /* Alternation, | */
#define CAT  0204 /* Concatentation, implicit operator */
#define STAR  0205 /* Closure, * */
#define PLUS  0206 /* a+ == aa* */
#define QUEST  0207 /* a? == a|nothing, i.e. 0 or 1 a's */
#define ANY  0300 /* Any character except newline, . */
#define ANYNL  0301 /* Any character including newline, . */
#define NOP  0302 /* No operation, internal use only */
#define BOL  0303 /* Beginning of line, ^ */
#define EOL  0304 /* End of line, $ */
#define CCLASS  0305 /* Character class, [] */
#define NCCLASS  0306 /* Negated character class, [] */
#define END  0377 /* Terminate: match found */

/*
 *  regexec execution lists
 */
#define LISTSIZE 10
#define BIGLISTSIZE (25*LISTSIZE)
typedef struct Relist Relist;
struct Relist
{
 Reinst*  inst;  /* Reinstruction of the thread */
 Resublist se;  /* matched subexpressions in this thread */
};
typedef struct Reljunk Reljunk;
struct Reljunk
{
 Relist* relist[2];
```

```
  Relist* reliste[2];
  int starttype;
  Rune startchar;
  char* starts;
  char* eol;
  Rune* rstarts;
  Rune* reol;
 };

 extern Relist* _renewthread(Relist*, Reinst*, int, Resublist*);
 extern void _renewmatch(Resub*, int, Resublist*);
 extern Relist* _renewemptythread(Relist*, Reinst*, int, char*);
 extern Relist* _rrenewemptythread(Relist*, Reinst*, int, Rune*);
```
Uses `NSUBEXP` 569b, `Relist` 569b, `Reljunk` 569b, and `Resublist` 569b.

libregexp/regaux.c

⟨*function* `_renewmatch` 570a⟩≡ (572a)
```
 /*
  *  save a new match in mp
  */
 extern void
 _renewmatch(Resub *mp, int ms, Resublist *sp)
 {
     int i;

     if(mp==0 || ms<=0)
         return;
     if(mp[0].s.sp==0 || sp->m[0].s.sp<mp[0].s.sp ||
        (sp->m[0].s.sp==mp[0].s.sp && sp->m[0].e.ep>mp[0].e.ep)){
         for(i=0; i<ms && i<NSUBEXP; i++)
             mp[i] = sp->m[i];
         for(; i<ms; i++)
             mp[i].s.sp = mp[i].e.ep = 0;
     }
 }
```
Uses `NSUBEXP` 569b.

⟨*function* `_renewthread` 570b⟩≡ (572a)
```
 /*
  * Note optimization in _renewthread:
  *  *lp must be pending when _renewthread called; if *l has been looked
  *  at already, the optimization is a bug.
  */
 extern Relist*
 _renewthread(Relist *lp, /* _relist to add to */
     Reinst *ip,  /* instruction to add */
     int ms,
     Resublist *sep)  /* pointers to subexpressions */
 {
     Relist *p;

     for(p=lp; p->inst; p++){
         if(p->inst == ip){
             if(sep->m[0].s.sp < p->se.m[0].s.sp){
                 if(ms > 1)
                     p->se = *sep;
                 else
```

```
                p->se.m[0] = sep->m[0];
            }
            return 0;
        }
    }
    p->inst = ip;
    if(ms > 1)
        p->se = *sep;
    else
        p->se.m[0] = sep->m[0];
    (++p)->inst = 0;
    return p;
}
```

⟨*function* _renewemptythread 571a⟩≡ (572a)

```
/*
 * same as renewthread, but called with
 * initial empty start pointer.
 */
extern Relist*
_renewemptythread(Relist *lp, /* _relist to add to */
    Reinst *ip,  /* instruction to add */
    int ms,
    char *sp)  /* pointers to subexpressions */
{
    Relist *p;

    for(p=lp; p->inst; p++){
        if(p->inst == ip){
            if(sp < p->se.m[0].s.sp) {
                if(ms > 1)
                    memset(&p->se, 0, sizeof(p->se));
                p->se.m[0].s.sp = sp;
            }
            return 0;
        }
    }
    p->inst = ip;
    if(ms > 1)
        memset(&p->se, 0, sizeof(p->se));
    p->se.m[0].s.sp = sp;
    (++p)->inst = 0;
    return p;
}
```

Uses memset() 46b.

⟨*function* _rrenewemptythread 571b⟩≡ (572a)

```
extern Relist*
_rrenewemptythread(Relist *lp, /* _relist to add to */
    Reinst *ip,  /* instruction to add */
    int ms,
    Rune *rsp)  /* pointers to subexpressions */
{
    Relist *p;

    for(p=lp; p->inst; p++){
        if(p->inst == ip){
            if(rsp < p->se.m[0].s.rsp) {
                if(ms > 1)
                    memset(&p->se, 0, sizeof(p->se));
```

```
                p->se.m[0].s.rsp = rsp;
            }
            return 0;
        }
    }
    p->inst = ip;
    if(ms > 1)
        memset(&p->se, 0, sizeof(p->se));
    p->se.m[0].s.rsp = rsp;
    (++p)->inst = 0;
    return p;
}
```
Uses `memset()` 46b.

⟨`libregexp/regaux.c` 572a⟩≡
```
⟨libregexp includes 569a⟩
#include "regcomp.h"

⟨function _renewmatch 570a⟩
⟨function _renewthread 570b⟩
⟨function _renewemptythread 571a⟩
⟨function _rrenewemptythread 571b⟩
```

libregexp/regcomp.c

⟨*constant* `TRUE` 572b⟩≡ (581c)
```
#define TRUE 1
```

⟨*constant* `FALSE` 572c⟩≡ (581c)
```
#define FALSE 0
```

⟨*struct* `Node` 572d⟩≡ (581c)
```
/*
 * Parser Information
 */
typedef
struct Node
{
    Reinst* first;
    Reinst* last;
}Node;
```
Uses `Node` 572d.

⟨*global* `reprog` 572e⟩≡ (581c)
```
/* max character classes per program is nelem(reprog->class) */
static Reprog *reprog;
```

⟨*constant* `NCCRUNE` 572f⟩≡ (581c)
```
/* max rune ranges per character class is nelem(classp->spans)/2 */
#define NCCRUNE nelem(classp->spans)
```

⟨*constant* `NSTACK` 572g⟩≡ (581c)
```
#define NSTACK 20
```

⟨*global* `andstack` 572h⟩≡ (581c)
```
static Node andstack[NSTACK];
```
Uses `NSTACK-338` 572g.

⟨*global* `andp` 573a⟩≡ (581c)
```
static Node *andp;
```

⟨*global* `atorstack` 573b⟩≡ (581c)
```
static int atorstack[NSTACK];
```
Uses `NSTACK-338` 572g.

⟨*global* `atorp` 573c⟩≡ (581c)
```
static int* atorp;
```

⟨*global* `cursubid` 573d⟩≡ (581c)
```
static int cursubid;  /* id of current subexpression */
```

⟨*global* `subidstack` 573e⟩≡ (581c)
```
static int subidstack[NSTACK]; /* parallel to atorstack */
```
Uses `NSTACK-338` 572g.

⟨*global* `subidp` 573f⟩≡ (581c)
```
static int* subidp;
```

⟨*global* `lastwasand` 573g⟩≡ (581c)
```
static int lastwasand; /* Last token was operand */
```

⟨*global* `nbra` 573h⟩≡ (581c)
```
static int nbra;
```

⟨*global* `exprp` 573i⟩≡ (581c)
```
static char* exprp;  /* pointer to next character in source expression */
```

⟨*global* `lexdone` 573j⟩≡ (581c)
```
static int lexdone;
```

⟨*global* `nclass` 573k⟩≡ (581c)
```
static int nclass;
```

⟨*global* `classp` 573l⟩≡ (581c)
```
static Reclass*classp;
```

⟨*global* `freep` 573m⟩≡ (581c)
```
static Reinst* freep;
```

⟨*global* `errors` 573n⟩≡ (581c)
```
static int errors;
```

⟨*global* `yyrune` 573o⟩≡ (581c)
```
static Rune yyrune;  /* last lex'd rune */
```

⟨*global* `yyclassp` 573p⟩≡ (581c)
```
static Reclass*yyclassp; /* last lex'd class */
```

⟨*global* `regkaboom` 573q⟩≡ (581c)
```
static jmp_buf regkaboom;
```

⟨*function* `rcerror` 573r⟩≡ (581c)
```
static void
rcerror(char *s)
{
    errors++;
    regerror(s);
    longjmp(regkaboom, 1);
}
```
Uses `errors-353` 573n, `regerror()` 119, and `regkaboom-356` 573q.

⟨*function* newinst 574a⟩≡ (581c)
```
static Reinst*
newinst(int t)
{
    freep->type = t;
    freep->left = 0;
    freep->right = 0;
    return freep++;
}
```
Uses freep-352 573m.

⟨*function* operand 574b⟩≡ (581c)
```
static void
operand(int t)
{
    Reinst *i;

    if(lastwasand)
        operator(CAT); /* catenate is implicit */
    i = newinst(t);

    if(t == CCLASS || t == NCCLASS)
        i->cp = yyclassp;
    if(t == RUNE)
        i->r = yyrune;

    pushand(i, i);
    lastwasand = TRUE;
}
```
Uses CAT 569b, CCLASS 569b, NCCLASS 569b, RUNE 569b, lastwasand-346 573g, newinst() 574a, operator() 574c, pushand() 575b, yyclassp-355 573p, and yyrune-354 573o.

⟨*function* operator 574c⟩≡ (581c)
```
static void
operator(int t)
{
    if(t==RBRA && --nbra<0)
        rcerror("unmatched right paren");
    if(t==LBRA){
        if(++cursubid >= NSUBEXP)
            rcerror ("too many subexpressions");
        nbra++;
        if(lastwasand)
            operator(CAT);
    } else
        evaluntil(t);
    if(t != RBRA)
        pushator(t);
    lastwasand = FALSE;
    if(t==STAR || t==QUEST || t==PLUS || t==RBRA)
        lastwasand = TRUE; /* these look like operands */
}
```
Uses CAT 569b, LBRA 569b, NSUBEXP 569b, PLUS 569b, QUEST 569b, RBRA 569b, STAR 569b, cursubid-343 573d, evaluntil() 576b, lastwasand-346 573g, nbra-347 573h, operator() 574c, pushator() 575c, and rcerror() 573r.

⟨*function* regerr2 574d⟩≡ (581c)
```
static void
regerr2(char *s, int c)
{
```

```
    char buf[100];
    char *cp = buf;
    while(*s)
        *cp++ = *s++;
    *cp++ = c;
    *cp = '\0';
    rcerror(buf);
}
```

Uses `rcerror()` 573r.

⟨*function* `cant` 575a⟩≡ (581c)

```
static void
cant(char *s)
{
    char buf[100];
    strcpy(buf, "can't happen: ");
    strcat(buf, s);
    rcerror(buf);
}
```

Uses `rcerror()` 573r, `strcat()` 80b, and `strcpy()` 81c.

⟨*function* `pushand` 575b⟩≡ (581c)

```
static void
pushand(Reinst *f, Reinst *l)
{
    if(andp >= &andstack[NSTACK])
        cant("operand stack overflow");
    andp->first = f;
    andp->last = l;
    andp++;
}
```

Uses `NSTACK-338` 572g, `andp-340` 573a, `andstack-339` 572h, and `cant()` 575a.

⟨*function* `pushator` 575c⟩≡ (581c)

```
static void
pushator(int t)
{
    if(atorp >= &atorstack[NSTACK])
        cant("operator stack overflow");
    *atorp++ = t;
    *subidp++ = cursubid;
}
```

Uses `NSTACK-338` 572g, `atorp-342` 573c, `atorstack-341` 573b, `cant()` 575a, `cursubid-343` 573d, and `subidp-345` 573f.

⟨*function* `popand` 575d⟩≡ (581c)

```
static Node*
popand(int op)
{
    Reinst *inst;

    if(andp <= &andstack[0]){
        regerr2("missing operand for ", op);
        inst = newinst(NOP);
        pushand(inst,inst);
    }
    return --andp;
}
```

Uses `andp-340` 573a, `andstack-339` 572h, `newinst()` 574a, `pushand()` 575b, and `regerr2()` 574d.

⟨*function* popator 576a⟩≡ (581c)

```
static int
popator(void)
{
    if(atorp <= &atorstack[0])
        cant("operator stack underflow");
    --subidp;
    return *--atorp;
}
```

Uses atorp-342 573c, atorstack-341 573b, cant() 575a, and subidp-345 573f.

⟨*function* evaluntil 576b⟩≡ (581c)

```
static void
evaluntil(int pri)
{
    Node *op1, *op2;
    Reinst *inst1, *inst2;

    while(pri==RBRA || atorp[-1]>=pri){
        switch(popator()){
        default:
            rcerror("unknown operator in evaluntil");
            break;
        case LBRA:  /* must have been RBRA */
            op1 = popand('(');
            inst2 = newinst(RBRA);
            inst2->subid = *subidp;
            op1->last->next = inst2;
            inst1 = newinst(LBRA);
            inst1->subid = *subidp;
            inst1->next = op1->first;
            pushand(inst1, inst2);
            return;
        case OR:
            op2 = popand('|');
            op1 = popand('|');
            inst2 = newinst(NOP);
            op2->last->next = inst2;
            op1->last->next = inst2;
            inst1 = newinst(OR);
            inst1->right = op1->first;
            inst1->left = op2->first;
            pushand(inst1, inst2);
            break;
        case CAT:
            op2 = popand(0);
            op1 = popand(0);
            op1->last->next = op2->first;
            pushand(op1->first, op2->last);
            break;
        case STAR:
            op2 = popand('*');
            inst1 = newinst(OR);
            op2->last->next = inst1;
            inst1->right = op2->first;
            pushand(inst1, inst1);
            break;
        case PLUS:
            op2 = popand('+');
            inst1 = newinst(OR);
```

```
            op2->last->next = inst1;
            inst1->right = op2->first;
            pushand(op2->first, inst1);
            break;
        case QUEST:
            op2 = popand('?');
            inst1 = newinst(OR);
            inst2 = newinst(NOP);
            inst1->left = inst2;
            inst1->right = op2->first;
            op2->last->next = inst2;
            pushand(inst1, inst2);
            break;
        }
    }
}
```

Uses CAT 569b, LBRA 569b, OR 569b, PLUS 569b, QUEST 569b, RBRA 569b, STAR 569b, atorp-342 573c, newinst() 574a, popand() 575d, popator() 576a, pushand() 575b, rcerror() 573r, and subidp-345 573f.

⟨*function* optimize 577⟩≡ (581c)

```
static Reprog*
optimize(Reprog *pp)
{
    Reinst *inst, *target;
    int size;
    Reprog *npp;
    Reclass *cl;
    int diff;

    /*
     *  get rid of NOOP chains
     */
    for(inst=pp->firstinst; inst->type!=END; inst++){
        target = inst->next;
        while(target->type == NOP)
            target = target->next;
        inst->next = target;
    }

    /*
     *  The original allocation is for an area larger than
     *  necessary.  Reallocate to the actual space used
     *  and then relocate the code.
     */
    size = sizeof(Reprog) + (freep - pp->firstinst)*sizeof(Reinst);
    npp = realloc(pp, size);
    if(npp==0 || npp==pp)
        return pp;
    diff = (char *)npp - (char *)pp;
    freep = (Reinst *)((char *)freep + diff);
    for(inst=npp->firstinst; inst<freep; inst++){
        switch(inst->type){
        case OR:
        case STAR:
        case PLUS:
        case QUEST:
            *(char **)&inst->right += diff;
            break;
        case CCLASS:
        case NCCLASS:
```

```
            *(char **)&inst->right += diff;
            cl = inst->cp;
            *(char **)&cl->end += diff;
            break;
        }
        *(char **)&inst->left += diff;
    }
    *(char **)&npp->startinst += diff;
    return npp;
}
```

Uses CCLASS 569b, END 569b, NCCLASS 569b, OR 569b, PLUS 569b, QUEST 569b, STAR 569b, freep-352 573m, and realloc() 67b.

⟨*function* dumpstack 578a⟩≡ (581c)

```
static void
dumpstack(void){
    Node *stk;
    int *ip;

    print("operators\n");
    for(ip=atorstack; ip<atorp; ip++)
        print("0%o\n", *ip);
    print("operands\n");
    for(stk=andstack; stk<andp; stk++)
        print("0%o\t0%o\n", stk->first->type, stk->last->type);
}
```

⟨*function* dump 578b⟩≡ (581c)

```
static void
dump(Reprog *pp)
{
    Reinst *l;
    Rune *p;

    l = pp->firstinst;
    do{
        print("%d:\t0%o\t%d\t%d", l-pp->firstinst, l->type,
            l->left-pp->firstinst, l->right-pp->firstinst);
        if(l->type == RUNE)
            print("\t%C\n", l->r);
        else if(l->type == CCLASS || l->type == NCCLASS){
            print("\t[");
            if(l->type == NCCLASS)
                print("^");
            for(p = l->cp->spans; p < l->cp->end; p += 2)
                if(p[0] == p[1])
                    print("%C", p[0]);
                else
                    print("%C-%C", p[0], p[1]);
            print("]\n");
        } else
            print("\n");
    }while(l++->type);
}
```

⟨*function* newclass 578c⟩≡ (581c)

```
static Reclass*
newclass(void)
{
    if(nclass >= nelem(reprog->class))
        rcerror("too many character classes; increase Reprog.class size");
```

```
    return &(classp[nclass++]);
}
```
Uses classp-351 573l, nclass-350 573k, rcerror() 573r, and reprog-336 572e.

⟨*function* nextc 579a⟩≡ (581c)
```
static int
nextc(Rune *rp)
{
    if(lexdone){
        *rp = 0;
        return 1;
    }
    exprp += chartorune(rp, exprp);
    if(*rp == L'\\'){
        exprp += chartorune(rp, exprp);
        return 1;
    }
    if(*rp == 0)
        lexdone = 1;
    return 0;
}
```
Uses chartorune() 84d, exprp-348 573i, and lexdone-349 573j.

⟨*function* lex 579b⟩≡ (581c)
```
static int
lex(int literal, int dot_type)
{
    int quoted;

    quoted = nextc(&yyrune);
    if(literal || quoted){
        if(yyrune == 0)
            return END;
        return RUNE;
    }

    switch(yyrune){
    case 0:
        return END;
    case L'*':
        return STAR;
    case L'?':
        return QUEST;
    case L'+':
        return PLUS;
    case L'|':
        return OR;
    case L'.':
        return dot_type;
    case L'(':
        return LBRA;
    case L')':
        return RBRA;
    case L'^':
        return BOL;
    case L'$':
        return EOL;
    case L'[':
        return bldcclass();
    }
```

```
    return RUNE;
}
```

Uses BOL 569b, END 569b, EOL 569b, LBRA 569b, OR 569b, PLUS 569b, QUEST 569b, RBRA 569b, RUNE 569b, STAR 569b, bldcclass() 580, nextc() 579a, and yyrune-354 573o.

⟨*function* bldcclass 580⟩≡ (581c)

```
static int
bldcclass(void)
{
    int type;
    Rune r[NCCRUNE];
    Rune *p, *ep, *np;
    Rune rune;
    int quoted;

    /* we have already seen the '[' */
    type = CCLASS;
    yyclassp = newclass();

    /* look ahead for negation */
    /* SPECIAL CASE!!! negated classes don't match \n */
    ep = r;
    quoted = nextc(&rune);
    if(!quoted && rune == L'^'){
        type = NCCLASS;
        quoted = nextc(&rune);
        *ep++ = L'\n';
        *ep++ = L'\n';
    }

    /* parse class into a set of spans */
    while(ep < &r[NCCRUNE-1]){
        if(rune == 0){
            rcerror("malformed '[]'");
            return 0;
        }
        if(!quoted && rune == L']')
            break;
        if(!quoted && rune == L'-'){
            if(ep == r){
                rcerror("malformed '[]'");
                return 0;
            }
            quoted = nextc(&rune);
            if((!quoted && rune == L']') || rune == 0){
                rcerror("malformed '[]'");
                return 0;
            }
            *(ep-1) = rune;
        } else {
            *ep++ = rune;
            *ep++ = rune;
        }
        quoted = nextc(&rune);
    }
    if(ep >= &r[NCCRUNE-1]) {
        rcerror("char class too large; increase Reclass.spans size");
        return 0;
    }
```

```
    /* sort on span start */
    for(p = r; p < ep; p += 2){
        for(np = p; np < ep; np += 2)
            if(*np < *p){
                rune = np[0];
                np[0] = p[0];
                p[0] = rune;
                rune = np[1];
                np[1] = p[1];
                p[1] = rune;
            }
    }

    /* merge spans */
    np = yyclassp->spans;
    p = r;
    if(r == ep)
        yyclassp->end = np;
    else {
        np[0] = *p++;
        np[1] = *p++;
        for(; p < ep; p += 2)
            /* overlapping or adjacent ranges? */
            if(p[0] <= np[1] + 1){
                if(p[1] >= np[1])
                    np[1] = p[1]; /* coalesce */
            } else {
                np += 2;
                np[0] = p[0];
                np[1] = p[1];
            }
        yyclassp->end = np+2;
    }

    return type;
}
```

Uses `CCLASS` 569b, `NCCLASS` 569b, `NCCRUNE-337` 572f, `newclass()` 578c, `nextc()` 579a, `rcerror()` 573r, and `yyclassp-355` 573p.

⟨*function* `regcomplit` 581a⟩≡ (581c)

```
extern Reprog*
regcomplit(char *s)
{
    return regcomp1(s, 1, ANY);
}
```

Uses `ANY` 569b and `regcomp1()` 107b.

⟨*function* `regcompnl` 581b⟩≡ (581c)

```
extern Reprog*
regcompnl(char *s)
{
    return regcomp1(s, 0, ANYNL);
}
```

Uses `ANYNL` 569b and `regcomp1()` 107b.

⟨`libregexp/regcomp.c` 581c⟩≡

```
⟨libregexp includes 569a⟩
#include "regcomp.h"

⟨constant TRUE 572b⟩
```

```
⟨constant FALSE 572c⟩

⟨struct Node 572d⟩

⟨global reprog 572e⟩

⟨constant NCCRUNE 572f⟩

⟨constant NSTACK 572g⟩
⟨global andstack 572h⟩
⟨global andp 573a⟩
⟨global atorstack 573b⟩
⟨global atorp 573c⟩
⟨global cursubid 573d⟩
⟨global subidstack 573e⟩
⟨global subidp 573f⟩
⟨global lastwasand 573g⟩
⟨global nbra 573h⟩
⟨global exprp 573i⟩
⟨global lexdone 573j⟩
⟨global nclass 573k⟩
⟨global classp 573l⟩
⟨global freep 573m⟩
⟨global errors 573n⟩
⟨global yyrune 573o⟩
⟨global yyclassp 573p⟩

/* predeclared crap */
static void operator(int);
static void pushand(Reinst*, Reinst*);
static void pushator(int);
static void evaluntil(int);
static int bldcclass(void);

⟨global regkaboom 573q⟩

⟨function rcerror 573r⟩

⟨function newinst 574a⟩

⟨function operand 574b⟩

⟨function operator 574c⟩

⟨function regerr2 574d⟩

⟨function cant 575a⟩

⟨function pushand 575b⟩

⟨function pushator 575c⟩

⟨function popand 575d⟩

⟨function popator 576a⟩

⟨function evaluntil 576b⟩

⟨function optimize 577⟩
```

```
#ifdef DEBUG
⟨function dumpstack 578a⟩
⟨function dump 578b⟩
#endif

⟨function newclass 578c⟩

⟨function nextc 579a⟩

⟨function lex 579b⟩

⟨function bldcclass 580⟩

⟨function regcomp1 107b⟩
⟨function regcomp 107a⟩

⟨function regcomplit 581a⟩

⟨function regcompnl 581b⟩
```

libregexp/regerror.c

```
⟨libregexp/regerror.c 583a⟩≡
  ⟨libregexp includes 569a⟩
  ⟨function regerror 119⟩
```

libregexp/regexec.c

```
⟨libregexp/regexec.c 583b⟩≡
  ⟨libregexp includes 569a⟩
  #include "regcomp.h"

  ⟨function regexec1 110⟩
  ⟨function regexec2 112⟩

  ⟨function regexec 109⟩
```

libregexp/regsub.c

```
⟨libregexp/regsub.c 583c⟩≡
  ⟨libregexp includes 569a⟩
  ⟨function regsub 113⟩
```

libregexp/rregexec.c

```
⟨libregexp/rregexec.c 583d⟩≡
  ⟨libregexp includes 569a⟩
  #include "regcomp.h"

  ⟨function rregexec1 114⟩
  ⟨function rregexec2 117a⟩

  ⟨function rregexec 117b⟩
```

librexexp/rregsub.c

⟨librexexp/rregsub.c 584a⟩≡
  ⟨*libregexp includes* 569a⟩
  ⟨*function* rregsub 118⟩

A.11 libstring/

⟨*libstring includes* 584b⟩≡ (586 585 584)
  ⟨*libc includes* 387a⟩
```
#include <str.h> // was string.h
```

⟨*libstring includes bis* 584c⟩≡ (586 585)
  ⟨*libc includes* 387a⟩
```
#include <bio.h> // must be before, set BGETC used in str.h
#include <str.h> // was string.h
```

libstring/s_alloc.c

⟨libstring/s_alloc.c 584d⟩≡
  ⟨*libstring includes* 584b⟩

  ⟨*constant* STRLEN 92e⟩

  ⟨*function* s_free 93c⟩

  ⟨*function* s_incref 100b⟩

  ⟨*function* _s_alloc 92f⟩

  ⟨*function* s_newalloc 93b⟩

  ⟨*function* s_new 92d⟩

libstring/s_append.c

⟨libstring/s_append.c 584e⟩≡
  ⟨*libstring includes* 584b⟩
  ⟨*function* s_append 95b⟩

libstring/s_array.c

⟨libstring/s_array.c 584f⟩≡
  ⟨*libstring includes* 584b⟩

```
extern String* _s_alloc(void);
```

  ⟨*function* s_array 100d⟩

libstring/s_copy.c

⟨libstring/s_copy.c 584g⟩≡
  ⟨*libstring includes* 584b⟩
  ⟨*function* s_copy 93a⟩

libstring/s_getline.c

⟨libstring/s_getline.c 585a⟩≡
  ⟨*libstring includes bis* 584c⟩
  ⟨*function* s_getline 97b⟩

libstring/s_grow.c

⟨libstring/s_grow.c 585b⟩≡
  ⟨*libstring includes* 584b⟩
  ⟨*function* s_grow 94b⟩

libstring/s_memappend.c

⟨libstring/s_memappend.c 585c⟩≡
  ⟨*libstring includes* 584b⟩
  ⟨*function* s_memappend 95d⟩

libstring/s_nappend.c

⟨libstring/s_nappend.c 585d⟩≡
  ⟨*libstring includes* 584b⟩
  ⟨*function* s_nappend 95c⟩

libstring/s_parse.c

⟨libstring/s_parse.c 585e⟩≡
  ⟨*libstring includes* 584b⟩

  ⟨*macro* isspace 99a⟩

  ⟨*function* s_parse 99b⟩

libstring/s_putc.c

⟨libstring/s_putc.c 585f⟩≡
  ⟨*libstring includes* 584b⟩
  ⟨*function* s_putc 94a⟩

libstring/s_rdinstack.c

⟨libstring/s_rdinstack.c 585g⟩≡
  ⟨*libstring includes bis* 584c⟩

```
struct Sinstack{
    int depth;
    Biobuf *fp[32]; /* hard limit to avoid infinite recursion */
};
```

  ⟨*function* s_allocinstack 101a⟩

  ⟨*function* s_freeinstack 101b⟩

  ⟨*function* rdline 101c⟩

  ⟨*function* s_rdinstack 102⟩

libstring/s_read.c

⟨libstring/s_read.c 586a⟩≡
```
  ⟨libstring includes bis 584c⟩

  enum
  {
      Minread= 256,
  };

  ⟨function s_read 97a⟩
```

libstring/s_read_line.c

⟨libstring/s_read_line.c 586b⟩≡
```
  ⟨libstring includes bis 584c⟩
  ⟨function s_read_line 96c⟩
```

libstring/s_reset.c

⟨libstring/s_reset.c 586c⟩≡
```
  ⟨libstring includes 584b⟩

  ⟨function s_reset 96a⟩
  ⟨function s_restart 96b⟩
```

libstring/s_terminate.c

⟨libstring/s_terminate.c 586d⟩≡
```
  ⟨libstring includes 584b⟩
  ⟨function s_terminate 95a⟩
```

libstring/s_tolower.c

⟨libstring/s_tolower.c 586e⟩≡
```
  ⟨libstring includes 584b⟩
  ⟨function s_tolower 100a⟩
```

libstring/s_unique.c

⟨libstring/s_unique.c 586f⟩≡
```
  ⟨libstring includes 584b⟩
  ⟨function s_unique 100c⟩
```

A.12 libflate/

⟨*libflate includes* 586g⟩≡ (606c 605 602b 599 598a 597b 592 587)
```
  ⟨libc includes 387a⟩
  #include <flate.h>
```

libflate/adler.c

⟨libflate/adler.c 587a⟩≡
```
⟨libflate includes 586g⟩
enum
{
    ADLERITERS = 5552, /* max iters before can overflow 32 bits */
    ADLERBASE = 65521 /* largest prime smaller than 65536 */
};

⟨function adler32 185b⟩
```

libflate/crc.c

⟨function mkcrctab 587b⟩≡ (587c)
```
ulong*
mkcrctab(ulong poly)
{
    ulong *crctab;
    ulong crc;
    int i, j;

    crctab = malloc(256 * sizeof(ulong));
    if(crctab == nil)
        return nil;

    for(i = 0; i < 256; i++){
        crc = i;
        for(j = 0; j < 8; j++){
            if(crc & 1)
                crc = (crc >> 1) ^ poly;
            else
                crc >>= 1;
        }
        crctab[i] = crc;
    }
    return crctab;
}
```
Uses malloc() 63g.

⟨libflate/crc.c 587c⟩≡
```
⟨libflate includes 586g⟩
⟨function mkcrctab 587b⟩

⟨function blockcrc 186⟩
```

libflate/deflate.c

⟨macro hashit 587d⟩≡ (592)
```
#define hashit(c) ((((ulong)(c) & 0xffffff) * 0x6b43a9b5) >> (32 - HashLog))
```

⟨function deflatereset 587e⟩≡ (592)
```
static void
deflatereset(LZstate *lz, int level, int debug)
{
    memset(lz->nexts, 0, sizeof lz->nexts);
    memset(lz->hash, 0, sizeof lz->hash);
    lz->totr = 0;
```

```
    lz->totw = 0;
    lz->pos = 0;
    lz->avail = 0;
    lz->out = lz->obuf;
    lz->eout = &lz->obuf[DeflateOut];
    lz->prevlen = MinMatch - 1;
    lz->prevoff = 0;
    lz->now = MaxOff + 1;
    lz->dot = lz->now;
    lz->bits = 0;
    lz->nbits = 0;
    lz->maxcheck = (1 << level);
    lz->maxcheck -= lz->maxcheck >> 2;
    if(lz->maxcheck < 2)
        lz->maxcheck = 2;
    else if(lz->maxcheck > 1024)
        lz->maxcheck = 1024;

    lz->debug = debug;
}
```

Uses `DeflateOut-305` 592, `MaxOff-296` 592, `MinMatch-297` 592, and `memset()` 46b.

⟨*function* `lzwrite` 588a⟩≡ (592)

```
static void
lzwrite(LZstate *lz, void *buf, int n)
{
    int nw;

    if(n && lz->w){
        nw = (*lz->w)(lz->wr, buf, n);
        if(nw != n){
            lz->w = nil;
            lz->wbad = 1;
        }else
            lz->totw += n;
    }
}
```

⟨*function* `lzflush` 588b⟩≡ (592)

```
static void
lzflush(LZstate *lz)
{
    lzwrite(lz, lz->obuf, lz->out - lz->obuf);
    lz->out = lz->obuf;
}
```

Uses `lzwrite()` 588a.

⟨*function* `wrdyncode` 588c⟩≡ (592)

```
static void
wrdyncode(LZstate *out, Dyncode *dc)
{
    Huff *codetab;
    uchar *codes, *codeaux;
    int i, v, c;

    /*
     * write out header, then code length code lengths,
     * and code lengths
     */
    lzput(out, dc->nlit-257, 5);
```

```
    lzput(out, dc->noff-1, 5);
    lzput(out, dc->nclen-4, 4);

    codetab = dc->codetab;
    for(i = 0; i < dc->nclen; i++)
        lzput(out, codetab[clenorder[i]].bits, 3);

    codes = dc->codes;
    codeaux = dc->codeaux;
    c = dc->ncode;
    for(i = 0; i < c; i++){
        v = codes[i];
        lzput(out, codetab[v].encode, codetab[v].bits);
        if(v >= 16){
            if(v == 16)
                lzput(out, codeaux[i], 2);
            else if(v == 17)
                lzput(out, codeaux[i], 3);
            else /* v == 18 */
                lzput(out, codeaux[i], 7);
        }
    }
}
```

Uses clenorder-326 592 and lzput() 179a.

⟨*function* bitcost 589a⟩≡ (592)

```
static int
bitcost(Huff *tab, ulong *count, int n)
{
    ulong tot;
    int i;

    tot = 0;
    for(i = 0; i < n; i++)
        tot += count[i] * tab[i].bits;
    return tot;
}
```

⟨*function* mkgzprecode 589b⟩≡ (592)

```
static int
mkgzprecode(Huff *tab, ulong *count, int n, int maxbits)
{
    ulong bitcount[MaxHuffBits];
    int i, nbits;

    nbits = mkprecode(tab, count, n, maxbits, bitcount);
    for(i = 0; i < n; i++){
        if(tab[i].bits == -1)
            tab[i].bits = 0;
        else if(tab[i].bits == 0){
            if(nbits != 0 || bitcount[0] != 1)
                return 0;
            bitcount[1] = 1;
            bitcount[0] = 0;
            nbits = 1;
            tab[i].bits = 1;
        }
    }
    if(bitcount[0] != 0)
        return 0;
```

```
    return hufftabinit(tab, n, bitcount, nbits);
}
```

Uses MaxHuffBits-300 592, hufftabinit() 590a, and mkprecode() 174.

⟨*function* hufftabinit 590a⟩≡ (592)

```
static int
hufftabinit(Huff *tab, int n, ulong *bitcount, int nbits)
{
    ulong code, nc[MaxHuffBits];
    int i, bits;

    code = 0;
    for(bits = 1; bits <= nbits; bits++){
        code = (code + bitcount[bits-1]) << 1;
        nc[bits] = code;
    }

    for(i = 0; i < n; i++){
        bits = tab[i].bits;
        if(bits){
            code = nc[bits]++ << (16 - bits);
            if(code & ~0xffff)
                return 0;
            tab[i].encode = revtab[code >> 8] | (revtab[code & 0xff] << 8);
        }
    }
    return 1;
}
```

Uses MaxHuffBits-300 592 and revtab-330 592.

⟨*function* nextchain 590b⟩≡ (592)

```
/*
 * calculate the next chain on the list
 * we can always toss out the old chain
 */
static void
nextchain(Chains *cs, int list)
{
    Chain *c, *oc;
    int i, nleaf, sumc;

    oc = cs->lists[list + 1];
    cs->lists[list] = oc;
    if(oc == nil)
        return;

    /*
     * make sure we have all chains needed to make sumc
     * note it is possible to generate only one of these,
     * use twice that value for sumc, and then generate
     * the second if that preliminary sumc would be chosen.
     * however, this appears to be slower on current tests
     */
    if(oc->gen){
        nextchain(cs, list - 2);
        nextchain(cs, list - 2);
        oc->gen = 0;
    }

    /*
```

```
     * pick up the chain we're going to add;
     * collect unused chains no free ones are left
     */
    for(c = cs->free; ; c++){
        if(c >= cs->echains){
            cs->col++;
            for(i = 0; i < cs->nlists; i++)
                for(c = cs->lists[i]; c != nil; c = c->up)
                    c->col = cs->col;
            c = cs->chains;
        }
        if(c->col != cs->col)
            break;
    }

    /*
     * pick the cheapest of
     * 1) the next package from the previous list
     * 2) the next leaf
     */
    nleaf = oc->leaf;
    sumc = 0;
    if(list > 0 && cs->lists[list-1] != nil)
        sumc = cs->lists[list-2]->count + cs->lists[list-1]->count;
    if(sumc != 0 && (nleaf >= cs->nleaf || cs->leafcount[nleaf] > sumc)){
        c->count = sumc;
        c->leaf = oc->leaf;
        c->up = cs->lists[list-1];
        c->gen = 1;
    }else if(nleaf >= cs->nleaf){
        cs->lists[list + 1] = nil;
        return;
    }else{
        c->leaf = nleaf + 1;
        c->count = cs->leafcount[nleaf];
        c->up = oc->up;
        c->gen = 0;
    }
    cs->free = c + 1;

    cs->lists[list + 1] = c;
    c->col = cs->col;
}
```

Uses `nextchain()` 590b.

⟨*function* `pivot` *(flate)* 591⟩≡ (592)

```
static int
pivot(ulong *c, int a, int n)
{
    int j, pi, pj, pk;

    j = n/6;
    pi = a + j; /* 1/6 */
    j += j;
    pj = pi + j; /* 1/2 */
    pk = pj + j; /* 5/6 */
    if(c[pi] < c[pj]){
        if(c[pi] < c[pk]){
            if(c[pj] < c[pk])
                return pj;
```

```
            return pk;
        }
        return pi;
    }
    if(c[pj] < c[pk]){
        if(c[pi] < c[pk])
            return pi;
        return pk;
    }
    return pj;
}
```

⟨libflate/deflate.c 592⟩≡

```
⟨libflate includes 586g⟩
typedef struct Chain Chain;
typedef struct Chains Chains;
typedef struct Dyncode Dyncode;
typedef struct HuffDeflate Huff;
typedef struct LZblock LZblock;
typedef struct LZstate LZstate;

enum
{
    /*
     * deflate format paramaters
     */
    DeflateUnc = 0,   /* uncompressed block */
    DeflateFix = 1,   /* fixed huffman codes */
    DeflateDyn = 2,   /* dynamic huffman codes */

    DeflateEob = 256,   /* end of block code in lit/len book */
    DeflateMaxBlock = 64*1024-1,  /* maximum size of uncompressed block */

    DeflateMaxExp = 10,   /* maximum expansion for a block */

    LenStart = 257,   /* start of length codes in litlen */
    Nlitlen  = 288,   /* number of litlen codes */
    Noff   = 30,   /* number of offset codes */
    Nclen  = 19,   /* number of codelen codes */

    MaxOff   = 32*1024,
    MinMatch = 3,   /* shortest match possible */
    MaxMatch = 258,   /* longest match possible */

    /*
     * huffman code paramaters
     */
    MaxLeaf   = Nlitlen,
    MaxHuffBits = 16,   /* max bits in a huffman code */
    ChainMem = 2 * (MaxHuffBits - 1) * MaxHuffBits,

    /*
     * coding of the lz parse
     */
    LenFlag  = 1 << 3,
    LenShift = 4,   /* leaves enough space for MinMatchMaxOff */
    MaxLitRun = LenFlag - 1,

    /*
     * internal lz paramaters
```

```
     */
    DeflateOut = 4096,   /* output buffer size */
    BlockSize = 8192,   /* attempted input read quanta */
    DeflateBlock = DeflateMaxBlock & ~(BlockSize - 1),
    MinMatchMaxOff = 4096,   /* max profitable offset for small match;
                          * assumes 8 bits for len, 5+10 for offset
                          * DONT CHANGE WITHOUT CHANGING LZPARSE CONSTANTS
                          */
    HistSlop = 512,   /* must be at lead MaxMatch */
    HistBlock = 64*1024,
    HistSize = HistBlock + HistSlop,

    HashLog  = 13,
    HashSize = 1<<HashLog,

    MaxOffCode = 256,   /* biggest offset looked up in direct table */

    EstLitBits = 8,
    EstLenBits = 4,
    EstOffBits = 5,
};

/*
 * knuth vol. 3 multiplicative hashing
 * each byte x chosen according to rules
 * 1/4 < x < 3/10, 1/3 x < < 3/7, 4/7 < x < 2/3, 7/10 < x < 3/4
 * with reasonable spread between the bytes & their complements
 *
 * the 3 byte value appears to be as almost good as the 4 byte value,
 * and might be faster on some machines
 */
/*
#define hashit(c) (((ulong)(c) * 0x6b43a9) >> (24 - HashLog))
*/
⟨macro hashit 587d⟩

/*
 * lempel-ziv style compression state
 */
struct LZstate
{
    uchar hist[HistSize];
    ulong pos;    /* current location in history buffer */
    ulong avail;    /* data available after pos */
    int eof;
    ushort hash[HashSize];   /* hash chains */
    ushort nexts[MaxOff];
    int now;    /* pos in hash chains */
    int dot;    /* dawn of time in history */
    int prevlen;   /* lazy matching state */
    int prevoff;
    int maxcheck;   /* compressor tuning */

    uchar obuf[DeflateOut];
    uchar *out;    /* current position in the output buffer */
    uchar *eout;
    ulong bits;    /* bit shift register */
    int nbits;
    int rbad;    /* got an error reading the buffer */
    int wbad;    /* got an error writing the buffer */
```

```
    int (*w)(void*, void*, int);
    void *wr;

    ulong totr;    /* total input size */
    ulong totw;    /* total output size */
    int debug;
};

struct LZblock
{
    ushort parse[DeflateMaxBlock / 2 + 1];
    int lastv;    /* value being constucted for parse */
    ulong litlencount[Nlitlen];
    ulong offcount[Noff];
    ushort *eparse;   /* limit for parse table */
    int bytes;    /* consumed from the input */
    int excost;    /* cost of encoding extra len & off bits */
};

/*
 * huffman code table
 */
struct HuffDeflate
{
    short bits;    /* length of the code */
    ushort encode;    /* the code */
};

/*
 * encoding of dynamic huffman trees
 */
struct Dyncode
{
    int nlit;
    int noff;
    int nclen;
    int ncode;
    Huff codetab[Nclen];
    uchar codes[Nlitlen+Noff];
    uchar codeaux[Nlitlen+Noff];
};

static int deflateb(LZstate *lz, LZblock *lzb, void *rr, int (*r)(void*, void*, int));
static int lzcomp(LZstate*, LZblock*, uchar*, ushort*, int finish);
static void wrblock(LZstate*, int, ushort*, ushort*, Huff*, Huff*);
static int bitcost(Huff*, ulong*, int);
static int huffcodes(Dyncode*, Huff*, Huff*);
static void wrdyncode(LZstate*, Dyncode*);
static void lzput(LZstate*, ulong bits, int nbits);
static void lzflushbits(LZstate*);
static void lzflush(LZstate *lz);
static void lzwrite(LZstate *lz, void *buf, int n);

static int hufftabinit(Huff*, int, ulong*, int);
static int mkgzprecode(Huff*, ulong *, int, int);

static int mkprecode(Huff*, ulong *, int, int, ulong*);
static void nextchain(Chains*, int);
static void leafsort(ulong*, ushort*, int, int);
```

```
/* conversion from len to code word */
static int lencode[MaxMatch];

/*
 * conversion from off to code word
 * off <= MaxOffCode ? offcode[off] : bigoffcode[off >> 7]
*/
static int offcode[MaxOffCode];
static int bigoffcode[256];

/* litlen code words LenStart-285 extra bits */
static int litlenbase[Nlitlen-LenStart];
static int litlenextra[Nlitlen-LenStart] =
{
/* 257 */ 0, 0, 0,
/* 260 */ 0, 0, 0, 0, 0, 1, 1, 1, 1, 2,
/* 270 */ 2, 2, 2, 3, 3, 3, 3, 4, 4, 4,
/* 280 */ 4, 5, 5, 5, 5, 0, 0, 0
};

/* offset code word extra bits */
static int offbase[Noff];
static int offextra[] =
{
    0,  0,  0,  0,  1,  1,  2,  2,  3,  3,
    4,  4,  5,  5,  6,  6,  7,  7,  8,  8,
    9,  9,  10, 10, 11, 11, 12, 12, 13, 13,
    0,  0,
};

/* order code lengths */
static int clenorder[Nclen] =
{
        16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15
};

/* static huffman tables */
static Huff litlentab[Nlitlen];
static Huff offtab[Noff];
static Huff hofftab[Noff];

/* bit reversal for brain dead endian swap in huffman codes */
static uchar revtab[256];
static ulong nlits;
static ulong nmatches;

⟨function deflateinit 168b⟩

⟨function deflatereset 587e⟩

⟨function deflate 180⟩

⟨function deflateb 181⟩

⟨function lzwrite 588a⟩

⟨function lzflush 588b⟩

⟨function lzput 179a⟩
```

```
⟨function lzflushbits 179b⟩

⟨function wrblock 179c⟩

⟨function lzmatch 170⟩

⟨function lzcomp 171⟩

⟨function huffcodes 176⟩

⟨function wrdyncode 588c⟩

⟨function bitcost 589a⟩

⟨function mkgzprecode 589b⟩

⟨function hufftabinit 590a⟩


/*
 * this should be in a library
 */
struct Chain
{
    ulong count;    /* occurances of everything in the chain */
    ushort leaf;    /* leaves to the left of chain, or leaf value */
    char col;    /* ref count for collecting unused chains */
    char gen;    /* need to generate chains for next lower level */
    Chain *up;    /* Chain up in the lists */
};

struct Chains
{
    Chain *lists[(MaxHuffBits - 1) * 2];
    ulong leafcount[MaxLeaf];  /* sorted list of leaf counts */
    ushort leafmap[MaxLeaf];  /* map to actual leaf number */
    int nleaf;    /* number of leaves */
    Chain chains[ChainMem];
    Chain *echains;
    Chain *free;
    char col;
    int nlists;
};

⟨function mkprecode 174⟩

⟨function nextchain 590b⟩

⟨function pivot(flate) 591⟩

⟨function leafsort 177⟩
```

Uses `BlockSize-306` 592, `Chain` 592, `ChainMem-301` 592, `Chains` 592, `DeflateMaxBlock-290` 592, `DeflateOut-305` 592, `Dyncode` 592, `HashLog-312` 592, `HashSize-313` 592, `HistBlock-310` 592, `HistSize-311` 592, `HistSlop-309` 592, `LZblock` 592, `LZstate` 592, `LenFlag-302` 592, `LenStart-292` 592, `MaxHuffBits-300` 592, `MaxLeaf-299` 592, `MaxMatch-298` 592, `MaxOff-296` 592, `MaxOffCode-314` 592, `Nclen-295` 592, `Nlitlen-293` 592, and `Noff-294` 592.

libflate/deflateblock.c

⟨*function* `blread` 596⟩≡ (597b)

```
static int
blread(void *vb, void *buf, int n)
{
    Block *b;

    b = vb;
    if(n > b->limit - b->pos)
        n = b->limit - b->pos;
    memmove(buf, b->pos, n);
    b->pos += n;
    return n;
}
```

Uses memmove() 48.

⟨*function* blwrite 597a⟩≡ (597b)

```
static int
blwrite(void *vb, void *buf, int n)
{
    Block *b;

    b = vb;

    if(n > b->limit - b->pos)
        n = b->limit - b->pos;
    memmove(b->pos, buf, n);
    b->pos += n;
    return n;
}
```

Uses memmove() 48.

⟨libflate/deflateblock.c 597b⟩≡

```
⟨libflate includes 586g⟩
typedef struct Block Block;

struct Block
{
    uchar *pos;
    uchar *limit;
};

⟨function blread 596⟩

⟨function blwrite 597a⟩

⟨function deflateblock 184⟩
```

Uses Block 597b.

libflate/deflatezlib.c

⟨*function* zlread 597c⟩≡ (598a)

```
static int
zlread(void *vzr, void *buf, int n)
{
    ZRead *zr;

    zr = vzr;
    n = (*zr->r)(zr->rr, buf, n);
    if(n <= 0)
```

```
        return n;
    zr->adler = adler32(zr->adler, buf, n);
    return n;
}
```
Uses adler32() 185b.

⟨libflate/deflatezlib.c 598a⟩≡
```
⟨libflate includes 586g⟩
#include "zlib.h"

typedef struct ZRead ZRead;

struct ZRead
{
    ulong adler;
    void *rr;
    int (*r)(void*, void*, int);
};

⟨function zlread 597c⟩

⟨function deflatezlib 187a⟩
```
Uses ZRead 598a.

libflate/deflatezlibblock.c

⟨*function* deflatezlibblock 598b⟩≡ (599a)
```
int
deflatezlibblock(uchar *dst, int dsize, uchar *src, int ssize, int level, int debug)
{
    ulong adler;
    int n;

    if(dsize < 6)
        return FlateOutputFail;

    n = deflateblock(dst + 2, dsize - 6, src, ssize, level, debug);
    if(n < 0)
        return n;

    dst[0] = ZlibDeflate | ZlibWin32k;

    /* bogus zlib encoding of compression level */
    dst[1] = ((level > 2) + (level > 5) + (level > 8)) << 6;

    /* header check field */
    dst[1] |= 31 - ((dst[0] << 8) | dst[1]) % 31;

    adler = adler32(1, src, ssize);
    dst[n + 2] = adler >> 24;
    dst[n + 3] = adler >> 16;
    dst[n + 4] = adler >> 8;
    dst[n + 5] = adler;

    return n + 6;
}
```
Uses ZlibDeflate 607, ZlibWin32k 607, adler32() 185b, and deflateblock() 184.

⟨libflate/deflatezlibblock.c 599a⟩≡
  ⟨*libflate includes* 586g⟩
```
#include "zlib.h"
```

  ⟨*function* deflatezlibblock 598b⟩

libflate/flateerr.c

⟨libflate/flateerr.c 599b⟩≡
  ⟨*libflate includes* 586g⟩
  ⟨*function* flateerr 188⟩

libflate/inflate.c

⟨*function* uncblock 599c⟩≡ (602b)
```
static int
uncblock(Input *in, History *his)
{
    int len, nlen, c;
    uchar *hs, *hp, *he;

    if(!sregunget(in))
        return 0;
    len = (*in->get)(in->getr);
    len |= (*in->get)(in->getr)<<8;
    nlen = (*in->get)(in->getr);
    nlen |= (*in->get)(in->getr)<<8;
    if(len != (~nlen&0xffff)) {
        in->error = FlateCorrupted;
        return 0;
    }

    hp = his->cp;
    hs = his->his;
    he = hs + HistorySize;

    while(len > 0) {
        c = (*in->get)(in->getr);
        if(c < 0)
            return 0;
        *hp++ = c;
        if(hp == he) {
            his->full = 1;
            if((*in->w)(in->wr, hs, HistorySize) != HistorySize) {
                in->error = FlateOutputFail;
                return 0;
            }
            hp = hs;
        }
        len--;
    }

    his->cp = hp;

    return 1;
}
```
Uses HistorySize-266 602b and sregunget() 168a.

⟨*function* fixedblock 600a⟩≡ (602b)

```
static int
fixedblock(Input *in, History *his)
{
    return decode(in, his, &litlentab, &offtab);
}
```

Uses decode() 163, litlentab-283 602b, and offtab-284 602b.

⟨*function* dynamicblock 600b⟩≡ (602b)

```
static int
dynamicblock(Input *in, History *his)
{
    Huff *lentab, *offtab;
    char *len;
    int i, j, n, c, nlit, ndist, nclen, res, nb;

    if(!sregfill(in, 14))
        return 0;
    nlit = (in->sreg&0x1f) + 257;
    ndist = ((in->sreg>>5) & 0x1f) + 1;
    nclen = ((in->sreg>>10) & 0xf) + 4;
    in->sreg >>= 14;
    in->nbits -= 14;

    if(nlit > Nlitlen || ndist > Noff || nlit < 257) {
        in->error = FlateCorrupted;
        return 0;
    }

    /* huff table header */
    len = malloc(Nlitlen+Noff);
    lentab = malloc(sizeof(Huff));
    offtab = malloc(sizeof(Huff));
    if(len == nil || lentab == nil || offtab == nil){
        in->error = FlateNoMem;
        goto bad;
    }
    for(i=0; i < Nclen; i++)
        len[i] = 0;
    for(i=0; i<nclen; i++) {
        if(!sregfill(in, 3))
            goto bad;
        len[clenorder[i]] = in->sreg & 0x7;
        in->sreg >>= 3;
        in->nbits -= 3;
    }

    if(!hufftab(lentab, len, Nclen, ClenBits)){
        in->error = FlateCorrupted;
        goto bad;
    }

    n = nlit+ndist;
    for(i=0; i<n;) {
        nb = lentab->minbits;
        for(;;){
            if(in->nbits<nb && !sregfill(in, nb))
                goto bad;
            c = lentab->flat[in->sreg & lentab->flatmask];
            nb = c & 0xff;
```

```
        if(nb > in->nbits){
            if(nb != 0xff)
                continue;
            c = hdecsym(in, lentab, c);
            if(c < 0)
                goto bad;
        }else{
            c >>= 8;
            in->sreg >>= nb;
            in->nbits -= nb;
        }
        break;
    }

    if(c < 16) {
        j = 1;
    } else if(c == 16) {
        if(in->nbits<2 && !sregfill(in, 2))
            goto bad;
        j = (in->sreg&0x3)+3;
        in->sreg >>= 2;
        in->nbits -= 2;
        if(i == 0) {
            in->error = FlateCorrupted;
            goto bad;
        }
        c = len[i-1];
    } else if(c == 17) {
        if(in->nbits<3 && !sregfill(in, 3))
            goto bad;
        j = (in->sreg&0x7)+3;
        in->sreg >>= 3;
        in->nbits -= 3;
        c = 0;
    } else if(c == 18) {
        if(in->nbits<7 && !sregfill(in, 7))
            goto bad;
        j = (in->sreg&0x7f)+11;
        in->sreg >>= 7;
        in->nbits -= 7;
        c = 0;
    } else {
        in->error = FlateCorrupted;
        goto bad;
    }

    if(i+j > n) {
        in->error = FlateCorrupted;
        goto bad;
    }

    while(j) {
        len[i] = c;
        i++;
        j--;
    }
}

if(!hufftab(lentab, len, nlit, LitlenBits)
|| !hufftab(offtab, &len[nlit], ndist, OffBits)){
```

```
        in->error = FlateCorrupted;
        goto bad;
    }

    res = decode(in, his, lentab, offtab);

    free(len);
    free(lentab);
    free(offtab);

    return res;

bad:
    free(len);
    free(lentab);
    free(offtab);
    return 0;
}
```

Uses ClenBits-275 602b, LitlenBits-273 602b, Nclen-271 602b, Nlitlen-269 602b, Noff-270 602b, clenorder-282 602b, decode() 163, free() 63i, hdecsym() 167a, hufftab() 165, malloc() 63g, and sregfill() 167b.

⟨*function* revcode 602a⟩≡ (602b)

```
static int
revcode(int c, int b)
{
    /* shift encode up so it starts on bit 15 then reverse */
    c <<= (16-b);
    c = revtab[c>>8] | (revtab[c&0xff]<<8);
    return c;
}
```

Uses revtab-285 602b.

⟨libflate/inflate.c 602b⟩≡

```
⟨libflate includes 586g⟩
enum {
    HistorySize= 32*1024,
    BufSize= 4*1024,
    MaxHuffBits= 17, /* maximum bits in a encoded code */
    Nlitlen= 288, /* number of litlen codes */
    Noff=  32, /* number of offset codes */
    Nclen=  19, /* number of codelen codes */
    LenShift= 10, /* code = len<<LenShift|code */
    LitlenBits= 7, /* number of bits in litlen decode table */
    OffBits= 6, /* number of bits in offset decode table */
    ClenBits= 6, /* number of bits in code len decode table */
    MaxFlatBits= LitlenBits,
    MaxLeaf= Nlitlen
};

typedef struct Input Input;
typedef struct History History;
typedef struct Huff Huff;

struct Input
{
    int error;  /* first error encountered, or FlateOk */
    void *wr;
    int (*w)(void*, void*, int);
    void *getr;
    int (*get)(void*);
```

```
    ulong sreg;
    int nbits;
};

struct History
{
    uchar his[HistorySize];
    uchar *cp;  /* current pointer in history */
    int full;  /* his has been filled up at least once */
};

struct Huff
{
    int maxbits; /* max bits for any code */
    int minbits; /* min bits to get before looking in flat */
    int flatmask; /* bits used in "flat" fast decoding table */
    ulong flat[1<<MaxFlatBits];
    ulong maxcode[MaxHuffBits];
    ulong last[MaxHuffBits];
    ulong decode[MaxLeaf];
};

/* litlen code words 257-285 extra bits */
static int litlenextra[Nlitlen-257] =
{
/* 257 */ 0, 0, 0,
/* 260 */ 0, 0, 0, 0, 0, 1, 1, 1, 1, 2,
/* 270 */ 2, 2, 2, 3, 3, 3, 3, 4, 4, 4,
/* 280 */ 4, 5, 5, 5, 5, 0, 0, 0
};

static int litlenbase[Nlitlen-257];

/* offset code word extra bits */
static int offextra[Noff] =
{
    0,  0,  0,  0,  1,  1,  2,  2,  3,  3,
    4,  4,  5,  5,  6,  6,  7,  7,  8,  8,
    9,  9,  10, 10, 11, 11, 12, 12, 13, 13,
    0,  0,
};
static int offbase[Noff];

/* order code lengths */
static int clenorder[Nclen] =
{
        16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15
};

/* for static huffman tables */
static Huff litlentab;
static Huff offtab;
static uchar revtab[256];

static int uncblock(Input *in, History*);
static int fixedblock(Input *in, History*);
static int dynamicblock(Input *in, History*);
static int sregfill(Input *in, int n);
static int sregunget(Input *in);
static int decode(Input*, History*, Huff*, Huff*);
```

```
static int hufftab(Huff*, char*, int, int);
static int hdecsym(Input *in, Huff *h, int b);

⟨function inflateinit 160⟩

⟨function inflate 161⟩

⟨function uncblock 599c⟩

⟨function fixedblock 600a⟩

⟨function dynamicblock 600b⟩

⟨function decode 163⟩

⟨function revcode 602a⟩

⟨function hufftab 165⟩

⟨function hdecsym 167a⟩

⟨function sregfill 167b⟩

⟨function sregunget 168a⟩
```

Uses `History` 602b, `HistorySize-266` 602b, `Huff` 602b, `Input` 602b, `LitlenBits-273` 602b, `MaxFlatBits-276` 602b, `MaxHuffBits-268` 602b, `MaxLeaf-277` 602b, `Nclen-271` 602b, `Nlitlen-269` 602b, and `Noff-270` 602b.

libflate/inflateblock.c

⟨*function* `blgetc` 604a⟩≡ (605a)

```
static int
blgetc(void *vb)
{
    Block *b;

    b = vb;
    if(b->pos >= b->limit)
        return -1;
    return *b->pos++;
}
```

⟨*function* `blwrite (libflate/inflateblock.c)` 604b⟩≡ (605a)

```
static int
blwrite(void *vb, void *buf, int n)
{
    Block *b;

    b = vb;

    if(n > b->limit - b->pos)
        n = b->limit - b->pos;
    memmove(b->pos, buf, n);
    b->pos += n;
    return n;
}
```

Uses `memmove()` 48.

⟨libflate/inflateblock.c 605a⟩≡
```
⟨libflate includes 586g⟩
typedef struct Block Block;

struct Block
{
    uchar *pos;
    uchar *limit;
};

⟨function blgetc 604a⟩

⟨function blwrite (libflate/inflateblock.c) 604b⟩

⟨function inflateblock 185a⟩
```

libflate/inflatezlib.c

⟨function zlwrite 605b⟩≡ (605c)
```
static int
zlwrite(void *vzw, void *buf, int n)
{
    ZWrite *zw;

    zw = vzw;
    zw->adler = adler32(zw->adler, buf, n);
    n = (*zw->w)(zw->wr, buf, n);
    if(n <= 0)
        return n;
    return n;
}
```
Uses adler32() 185b.

⟨libflate/inflatezlib.c 605c⟩≡
```
⟨libflate includes 586g⟩
#include "zlib.h"

typedef struct ZWrite ZWrite;

struct ZWrite
{
    ulong adler;
    void *wr;
    int (*w)(void*, void*, int);
};

⟨function zlwrite 605b⟩

⟨function inflatezlib 187b⟩
```
Uses ZWrite 605c.

libflate/inflatezlibblock.c

⟨function blgetc (libflate/inflatezlibblock.c) 605d⟩≡ (606c)
```
static int
blgetc(void *vb)
{
```

```
    Block *b;

    b = vb;
    if(b->pos >= b->limit)
        return -1;
    return *b->pos++;
}
```

⟨*function* blwrite (libflate/inflatezlibblock.c) 606a⟩≡ (606c)

```
static int
blwrite(void *vb, void *buf, int n)
{
    Block *b;

    b = vb;

    if(n > b->limit - b->pos)
        n = b->limit - b->pos;
    memmove(b->pos, buf, n);
    b->pos += n;
    return n;
}
```

Uses memmove() 48.

⟨*function* inflatezlibblock 606b⟩≡ (606c)

```
int
inflatezlibblock(uchar *dst, int dsize, uchar *src, int ssize)
{
    Block bd, bs;
    int ok;

    if(ssize < 6)
        return FlateInputFail;

    if(((src[0] << 8) | src[1]) % 31)
        return FlateCorrupted;
    if((src[0] & ZlibMeth) != ZlibDeflate
    || (src[0] & ZlibCInfo) > ZlibWin32k)
        return FlateCorrupted;

    bs.pos = src + 2;
    bs.limit = src + ssize - 6;

    bd.pos = dst;
    bd.limit = dst + dsize;

    ok = inflate(&bd, blwrite, &bs, blgetc);
    if(ok != FlateOk)
        return ok;

    if(adler32(1, dst, bs.pos - dst) != ((bs.pos[0] << 24) | (bs.pos[1] << 16) | (bs.pos[2] << 8) | bs.pos[3]))
        return FlateCorrupted;

    return bd.pos - dst;
}
```

Uses ZlibCInfo 607, ZlibDeflate 607, ZlibMeth 607, ZlibWin32k 607, and adler32() 185b.

⟨libflate/inflatezlibblock.c 606c⟩≡

```
⟨libflate includes 586g⟩
#include "zlib.h"
```

```
typedef struct Block Block;

struct Block
{
    uchar *pos;
    uchar *limit;
};

⟨function blgetc (libflate/inflatezlibblock.c) 605d⟩

⟨function blwrite (libflate/inflatezlibblock.c) 606a⟩

⟨function inflatezlibblock 606b⟩
```

libflate/zlib.h

⟨libflate/zlib.h 607⟩≡

```
/*
 * zlib header fields
 */
enum
{
    ZlibMeth = 0x0f,    /* mask of compression methods */
    ZlibDeflate = 0x08,

    ZlibCInfo = 0xf0,    /* mask of compression aux. info */
    ZlibWin32k = 0x70,    /* 32k history window */
};
```

Glossary

LOC	Lines Of Code

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.

A-72: 154f, 399e
A2B-129: 62h, 414c, 416a
A2TB-128: 62g, 414b, 414c, 416a, 425a
abort(): 48, 155b, 235f, 249b, 250b, 251b, 252, 300e, 303, 311a, 312, 313e, 321a, 327a, 333b, 336b, 356d, 400c, 401a, 401b, 402c, 490f, 545e
abs(): 122a
accept(): 346b
access(): 217b, 220
acos(): 144b
adler32(): 185b, 597c, 598b, 605b, 606b
ADLERBASE-265: 185b, 587a
ADLERITERS-264: 185b, 587a
ADVANCE-195: 390a, 390b
alarm(): 348c, 350, 473a
alignptr(): 422a
ALIGN_MAGIC-130: 63a, 420c
Alloc: 62c, 426e
Alloc (typedef): 426e
ALLOC_MAGIC-124: 62d, 416b, 420b, 420c
alt(): 306, 312
altcopy(): 311b
altexecbuffered(): 311a
andp-340: 107b, 573a, 575b, 575d
andstack-339: 107b, 572h, 575b, 575d
announce(): 344
ANY: 107a, 110, 114, 569b, 581a
ANYNL: 110, 114, 569b, 581b
Arena: 62e, 426e
Arena.asize: 62e
Arena.aup: 62e
Arena.down: 62e
Arena.pad: 62e
Arena (typedef): 426e
arenamerge(): 414c, 416a
arenasetsize(): 414b, 414c, 416a
ARENATAIL_MAGIC-127: 62f, 414b, 425a, 426c
ARENA_MAGIC-126: 62f, 414c
argv0: 236c, 303, 333a

asctime(): 225b, 467d
asin(): 144a, 144b
atan(): 144a, 144c, 146c
atan2(): 146c
atexit(): 45d, 198a, 360b
atexitflag-358: 197d, 198a
atnotify(): 259, 476b
atof(): 127c
atoi(): 123b, 256a, 256b, 401c, 473c, 486d, 556d
atol(): 123b, 123c, 226b, 232a, 232b, 335a, 360b, 495a
atoll(): 389d
atorp-342: 107b, 573c, 575c, 576a, 576b
atorstack-341: 107b, 573b, 575c, 576a
await(): 256a, 256b, 474b
B2D(): 420b
B2NB-115: 61b, 66, 412c, 415, 425a, 426c
B2PT-121: 61h, 66, 416a
B2T-120: 61g, 412c, 413a, 413b, 416b
Bad-112: 84d, 435d
badd(): 204a, 204b
batexit(): 198a, 198c
Bbuffered(): 212a
be2vlong(): 227, 487b
Bfildes(): 212b
Bflush(): 198c, 199c, 207a, 209b, 210a, 210b, 556d
Bgetc(): 97b, 101c, 200a, 201a, 554b
bgetd: 554b, 554c, 554d
Bgetd(): 554c
bgetd.b: 554d
bgetd.eof: 554d
Bgetdf(): 554b, 554c
Bgetrune(): 201a
Bhdr: 60, 426e
Bhdr.magic: 60
Bhdr.size: 60
Bhdr (typedef): 426e
Bias-203: 128, 132a
BIAS-42: 137b, 150g, 151a
BIGLISTSIZE: 112, 569b
bigoffcode-321: 168b, 171, 179c, 592
bind(): 324b
Binit(): 197a, 556d
Binits(): 197a, 197b, 198d
Bit-102: 84a, 435d
bitcost(): 176, 181, 589a
Bitx-105: 84d, 85, 435d, 435d
bldcclass(): 579b, 580
blgetc(): 604a
Blinelen(): 96c, 212c

bloc-257: 57c, 58a
Block: 597b, 597b
Block.limit: 597b
Block.pos: 597b
Block (typedef): 597b
blockcrc(): 186
blockgrow(): 415, 416a
blockmerge(): 66, 412c, 415
blocksetsize(): 413a, 414a, 414c, 415
BlockSize-306: 181, 592, 592
blread(): 184, 596
blwrite(): 597a
Boffset(): 210b, 211
BOL: 109, 110, 114, 117b, 569b, 579b
Bopen(): 101a, 102, 198d
Bprint(): 209a, 556d
Bputc(): 207a, 207b
Bputrune(): 207b
Brdline(): 96c, 202
Brdstr(): 204b
Bread(): 97a, 206
brk(): 58a
Bseek(): 210b
bsize2asize(): 412b
Btail: 61f, 426e
Btail.datasize: 61f
Btail.magic0: 61f
Btail.magic1: 61f
Btail.size: 61f
Btail (typedef): 426e
Bterm(): 101b, 102, 199c
BufSize-267: 602b
Bungetc(): 200b, 201a, 554c
Bungetrune(): 201b
Bvprint(): 209a, 209c
Bwrite(): 207b, 208
call(): 331
calloc(): 403b
canlock(): 245d, 247b, 327b, 402c
canqlock(): 247b
canrlock(): 249a
cant(): 575a, 575b, 575c, 576a
canwlock(): 250a
CAT: 569b, 574b, 574c, 576b
catcher(): 476a, 476b
CCLASS: 110, 114, 569b, 574b, 577, 580
ceil(): 123a
Chain: 592, 592
Chain.col: 592

Chain.count: 592
Chain.gen: 592
Chain.leaf: 592
Chain.up: 592
Chain (typedef): 592
ChainMem-301: 174, 592, 592
Chains: 592, 592
Chains.chains: 592
Chains.col: 592
Chains.echains: 592
Chains.free: 592
Chains.leafcount: 592
Chains.leafmap: 592
Chains.lists: 592
Chains.nleaf: 592
Chains.nlists: 592
Chains (typedef): 592
Chanalt: 298d, 306
CHANCLOSD-5: 290c
chanclose(): 290e
chanclosing(): 291
chancreate(): 289b, 316b, 322b
chanfree(): 290a, 316c
chaninit(): 289c
chanlock-7: 289a, 290a, 290e, 291, 306
channelsize(): 313e, 314a, 314b, 314c, 314d, 314e, 314f, 315a, 315b
Channone: 298d, 306
chanprint(): 341
Chanrecv: 298d
Chansend: 298d
chanstate: 298d, 558c
Chanstate (typedef): 558c
charstod(): 390b, 554c
chartorune(): 74, 84d, 89a, 89b, 89c, 90, 110, 201a, 218b, 255a, 461c, 503a, 505a, 520, 525, 579a
checkenv(): 401c, 402a, 402c
checklist(): 408b, 408c
checktree(): 408c, 408c, 425b
cistrcmp(): 391b
cistrncmp(): 392a, 392c
cistrstr(): 392c
classp-351: 107b, 573l, 578c
cleanname(): 218b
ClenBits-275: 600b, 602b
clenorder-282: 600b, 602b
clenorder-326: 176, 588c, 592
close(): 199c, 217a, 220, 226b, 227, 232a, 232b, 237c, 239b, 240, 257a, 257b, 258c, 299a, 319a, 323b, 324b, 330, 331, 335a, 344, 346a, 348c, 356b, 360b, 361, 401c, 463b, 468c, 473b, 483d, 486d, 494a, 495a, 496h
Closed-8: 290d, 290e, 306
closeioproc(): 316c

closeopenfd(): 350, 473b, 473c
Conn: 472b, 476c
Conn.cfd: 472b
Conn.dead: 472b
Conn.dfd: 472b
Conn.dir: 472b
Conn.err: 472b
Conn.pid: 472b
Conn (typedef): 476c
connectwait(): 474d, 476b
connsalloc(): 348c, 472d
convD2M(): 215c, 216c, 281
Convfmt: 518b, 521
Convfmt.c: 518b
Convfmt.fmt: 518b
Convfmt (typedef): 521
convM2D(): 215b, 216b, 263c, 280, 479b
convM2S(): 267
convS2M(): 273
cos(): 141a
cosh(): 148b, 148c
cputime(): 363b
create(): 198d, 257b, 361
csdial(): 330
ctime(): 354c, 467d
ct_numb(): 225b, 468b
cursubid-343: 107b, 573d, 574c, 575c
cycles(): 538b
D2B(): 66, 420c, 425d
datamagic-132: 63c
DBGAPPL: 334a
DBGCHAN: 289b, 289c, 309b, 309c, 311a, 334c
DBGEXEC: 323b, 323c, 324b, 334f
DBGNOTE: 327a, 327b, 334e, 338b
DBGREND: 334d, 339e
DBGSCHED: 296c, 302c, 304a, 304b, 305a, 334b
Dead: 298c, 302c
DEAD_MAGIC-114: 61a, 66
dec16(): 455c
dec32(): 456c
dec64(): 459a
decode(): 163, 600a, 600b
decref(): 288a
decrypt(): 238b
default_assert(): 235e, 235f
deflate(): 180, 184, 187a
deflateb(): 180, 181
deflateblock(): 184, 598b
DeflateBlock-307: 181, 592

DeflateDyn-288: 181, 592
DeflateEob-289: 181, 592
DeflateFix-287: 181, 592
deflateinit(): 168b
DeflateMaxBlock-290: 181, 592, 592
DeflateMaxExp-291: 592
DeflateOut-305: 587e, 592, 592
deflatereset(): 180, 587e
DeflateUnc-286: 181, 592
deflatezlib(): 187a
deflatezlibblock(): 598b
deinstall(): 198a, 198b, 199c
delayednotes(): 327a, 327b, 328b
dequeue(): 306, 309c
Dest: 472c, 476c
Dest.addrlist: 472c
Dest.conn: 472c
Dest.connend: 472c
Dest.naddrs: 472c
Dest.nextaddr: 472c
Dest.nkid: 472c
Dest.oalarm: 472c
Dest.winlck: 472c
Dest.winner: 472c
Dest (typedef): 476c
dial(): 328c, 347b
dialimpl(): 348a, 348b
dialmulti(): 348c, 476b
dirfmt(): 481c
dirfstat(): 215b, 354c
dirfwstat(): 215c
dirmodefmt(): 478d
dirpackage(): 479b, 480a, 480b
dirread(): 480a
dirreadall(): 480b
DIRSIZE-231: 216a, 216b
DIRSIZE-237: 215a, 215b
dirstat(): 216b, 220, 484
dirwstat(): 216c
divascii(): 128, 135a
divby(): 128, 134a, 135a
dmsize-234: 467a, 467e
dmsize-243: 496e, 496g
dofmt(): 73, 74, 75c, 209c, 529d, 532a, 533b, 534a, 536b, 537b
doquote: 518e
dorfmt(): 511a
DS: 329c, 568b
DS.buf: 329c
DS.cfdp: 329c

DS.dir: 329c
DS.local: 329c
DS.netdir: 329c
DS.proto: 329c
DS.rem: 329c
DS (typedef): 568b
dsize2bsize(): 412a
dumpblock(): 416b, 417a, 417b
DUMPL-233: 482a, 482b
dumpsome(): 263c, 482b
dynamicblock(): 600b
Dyncode: 592, 592
Dyncode.codeaux: 592
Dyncode.codes: 592
Dyncode.codetab: 592
Dyncode.nclen: 592
Dyncode.ncode: 592
Dyncode.nlit: 592
Dyncode.noff: 592
Dyncode (typedef): 592
dysize(): 467e, 468a
efork(): 323a, 323b
emptyentry(): 309b, 310a
enc16(): 393e, 456a
enc32(): 393e, 457
enc64(): 393e, 459b
encodefmt(): 393e
encrypt(): 238a
END: 107b, 110, 114, 569b, 577, 579b
enotes-14: 326f, 326f, 327a, 327b
enqueue(): 306, 309b
EOL: 110, 114, 569b, 579b
eqdirdev(): 354c, 356c
errcl-6: 306, 309a, 309a
errfmt(): 511c, 518d
ERRLEN-12: 326c
errors-353: 107b, 573n, 573r
errstr(): 208, 217b, 234a, 234b, 317, 324b, 329d, 330, 348b, 354c, 475b
EstLenBits-316: 592
EstLitBits-315: 592
EstOffBits-317: 592
etoken(): 453c, 454a
evaluntil(): 107b, 574c, 576b
exec(): 231b, 323b
Execargs: 294c, 558c
Execargs.args: 294c
Execargs.fd: 294c
Execargs.prog: 294c
Execargs (typedef): 558c

execl(): 231b
exits(): 45b, 119, 236c, 302c, 304b, 305a
exp(): 139, 148a, 148b, 149a
exprp-348: 107b, 573i, 579a
fabs(): 122b, 151a
fcallfmt(): 263c
fd-255: 227, 487c, 487c
fd2path(): 217a, 484
FDEFLT-29: 512b, 513
FDIGIT-28: 512b, 513, 517a
fdirconv(): 263c, 481c, 481d
Fdpoint-212: 128, 132a
fds-256: 227, 487d
Fesign-211: 128, 132a
fillinds(): 348c, 474c
finish(): 339e, 340b, 340d
finished(): 240, 489c
fixedblock(): 600a
flateerr(): 188
FLOATING_MAGIC-136: 419a
floor(): 122c, 123a, 149a
fmod(): 138
fmtalloc: 518c, 519a, 519c
fmtBflush(): 209b, 209c
FMTCHAR: 74, 500d, 502b
fmtfdflush(): 334g, 522a
fmtfdinit(): 73, 334g, 522b
fmtfmt(): 519c, 520
fmtinstall(): 519b, 527c
fmtl-32: 523a, 523b, 523c
fmtprint(): 334g, 523e
FMTRCHAR: 74, 500d, 502c, 503a, 503b, 511a, 525, 528d
FMTRUNE: 500d, 503a, 503b, 511a, 525, 528d
fmtrune(): 528d
fmtrunestrcpy(): 505c, 506a
Fmts (typedef): 500d
fmtstrcpy(): 263c, 393e, 478d, 481c, 505a, 505b
fmtStrFlush(): 536d, 537a
fmtstrflush(): 529b
fmtstrinit(): 537a, 537b
fmtvprint(): 334g, 523e, 529d
fork(): 231a
fpcmp(): 128, 133b
fprint(): 75a, 179c, 181, 197b, 198d, 210b, 211, 212a, 235f, 236c, 239b, 240, 247a, 303, 312, 313e, 321a, 333a, 361
frand(): 396c
Free: 62a, 426e
free(): 63i, 67b, 93c, 101b, 107b, 112, 160, 180, 199c, 215b, 215c, 216b, 216c, 220, 290b, 294b, 297c, 299a, 302c, 305c, 316c, 323c, 354c, 393e, 472d, 473a, 479b, 480a, 480b, 484, 485a, 485b, 491d, 499e, 532c, 533b,

536d, 537b, 600b
Free.left: 62a
Free.next: 62a
Free.prev: 62a
Free.right: 62a
Free (typedef): 426e
freedest(): 348c, 473a
freefromfront(): 414a
freenetconninfo(): 484, 485b
freep-352: 107b, 573m, 574a, 577
FREE_MAGIC-123: 62b, 66, 408b, 408c, 408d, 412c, 413a, 415
frexp(): 137a, 138, 139, 151a, 151b, 513
frnorm(): 128, 133a, 133b
Fsign-210: 128, 132a
fstat(): 215b
fullrune(): 201a, 436c, 461c, 503a, 525
fwstat(): 215c
getcallerpc(): 63g, 67a, 67b, 82a, 87d, 93a, 93b, 247a, 257a, 321a, 396f, 403a, 403b, 533a, 533b, 534e, 537a, 537b
getdsize(): 66, 413b, 415, 416b
getendpoint(): 483d, 484
getenv(): 257a
getfields(): 255a
getmalloctag(): 69c
getnetconninfo(): 484
getpid(): 45b, 45d, 217b, 227, 232a, 350
getppid(): 232b
getqlp(): 246c, 248b, 249c, 251b, 490f
getrealloctag(): 70
gettokens(): 454a
getuser(): 237c
getwd(): 217a
gmtime(): 225a, 467e
gqid(): 267, 272b
gstring(): 267, 272a
Half-208: 132a
hangup(): 397c
hashit-318: 171, 587d
HashLog-312: 592, 592
HashSize-313: 592, 592
havecycles: 357f, 360b, 361
hdecsym(): 163, 167a, 600b
HFinished-225: 489b, 489c
HistBlock-310: 170, 171, 179c, 181, 592, 592
History: 602b, 602b
History.cp: 602b
History.full: 602b
History.his: 602b
History (typedef): 602b

HistorySize-266: 163, 599c, 602b, 602b
HistSize-311: 181, 592, 592
HistSlop-309: 181, 592, 592
hofftab-329: 168b, 181, 592
Huff: 602b, 602b
Huff.decode: 602b
Huff.flat: 602b
Huff.flatmask: 602b
Huff.last: 602b
Huff.maxbits: 602b
Huff.maxcode: 602b
Huff.minbits: 602b
Huff (typedef): 602b
huffcodes(): 176, 181
HuffDeflate: 592
HuffDeflate.bits: 592
HuffDeflate.encode: 592
hufftab(): 160, 165, 600b
hufftabinit(): 168b, 589b, 590a
hypot(): 397e
HZ-246: 363a, 363b
identtrans(): 463a, 463b
imagmem: 59f, 59f
incref(): 287b
Inf(): 126f, 128, 149a, 150i
inflateblock(): 185a
inflateinit(): 160
inflatezlib(): 187b
inflatezlibblock(): 606b
Input: 602b, 602b
Input.error: 602b
Input.get: 602b
Input.getr: 602b
Input.nbits: 602b
Input.sreg: 602b
Input.w: 602b
Input.wr: 602b
Input (typedef): 602b
install(): 197b, 198a
Intred-9: 290e, 306, 325b
INVAL-84: 458b, 458c, 459a
iocall(): 317, 318a, 318c, 319b, 319d, 320a, 320c, 329a
ioclose(): 318c
iodial(): 329a
iointerrupt(): 316c, 316d, 317
ioopen(): 318a
Ioproc: 315d
ioproc(): 316b
Ioproc.arg: 315d

Ioproc.c: 315d
Ioproc.creply: 315d
Ioproc.err: 315d
Ioproc.inuse: 315d
Ioproc.next: 315d
Ioproc.op: 315d
Ioproc.ret: 315d
Ioproc.tid: 315d
ioproc_arg: 315c
ioread(): 319b
ioreadn(): 319d
ioreadv(): 491d, 492a, 492b
iosleep(): 320c
iounit(): 486d
iowrite(): 320a
iowritev(): 499e, 500a, 500b
isdirty-10: 340a, 340c, 340d
isInf(): 126d, 127a, 137a, 151a, 513
isNaN(): 126d, 151a, 513
isopenfor(): 306, 308a
isrand(): 153c, 154e, 154f
isspace-333: 99a, 99b
khz: 357d, 360b, 361
knownfmt-35: 518d, 519c
lastwasand-346: 107b, 573g, 574b, 574c
launcherarm(): 557b, 557b
LBRA: 110, 114, 569b, 574c, 576b, 579b
ldexp(): 128, 138, 139, 149a, 150i
ldmsize-244: 496f, 496g
leafsort(): 174, 177, 177
LEN-69: 153c, 154b, 154f, 399b
lencode-319: 168b, 171, 179c, 592
LenFlag-302: 171, 179c, 592, 592
LenShift-272: 602b
LenShift-303: 171, 179c, 592
LenStart-292: 168b, 171, 179c, 592, 592
lex(): 107b, 579b
lexdone-349: 107b, 573j, 579a
libc_print(): 72b, 72c
listadd(): 410b
listdelete(): 410c
listen(): 346a
LISTSIZE: 109, 117a, 117b, 569b
litlenbase-279: 160, 163, 602b
litlenbase-322: 168b, 179c, 592
LitlenBits-273: 600b, 602b, 602b
litlenextra-278: 160, 163, 602b
litlenextra-323: 168b, 171, 179c, 592
litlentab-283: 160, 600a, 602b

litlentab-327: 168b, 181, 592
lk-80: 153c, 154a, 154e
ln10o1-159: 152b, 153a
lnrand(): 398c
localtime(): 225a, 467d
lock(): 45d, 93c, 100b, 153c, 154e, 245b, 246c, 247a, 248b, 249a, 249b, 249c, 250a, 250b, 251b, 252, 259, 288c, 288d, 290a, 290e, 291, 294a, 296a, 296c, 297c, 298a, 302c, 304b, 305a, 305c, 306, 326j, 336a, 336b, 338a, 338b, 339e, 340b, 340d, 354c, 401a, 488d, 488e, 523b
log(): 139, 151b, 153a
log10(): 153a
log2-158: 151b, 152a
log2e-50: 149a, 149c
LONG_MAX-100: 124a, 124c
LONG_MIN-101: 124b, 124c
lrand(): 153b, 153c, 396c, 398c, 407b
ltreewalk(): 408d, 409a, 409b, 409c
LZblock: 592, 592
LZblock.bytes: 592
LZblock.eparse: 592
LZblock.excost: 592
LZblock.lastv: 592
LZblock.litlencount: 592
LZblock.offcount: 592
LZblock.parse: 592
LZblock (typedef): 592
lzcomp(): 171, 181
lzflush(): 179a, 181, 588b
lzflushbits(): 179b, 181
lzmatch(): 170, 171
lzput(): 179a, 179b, 179c, 181, 588c
LZstate: 592, 592
LZstate.avail: 592
LZstate.bits: 592
LZstate.debug: 592
LZstate.dot: 592
LZstate.eof: 592
LZstate.eout: 592
LZstate.hash: 592
LZstate.hist: 592
LZstate.maxcheck: 592
LZstate.nbits: 592
LZstate.nexts: 592
LZstate.now: 592
LZstate.obuf: 592
LZstate.out: 592
LZstate.pos: 592
LZstate.prevlen: 592
LZstate.prevoff: 592
LZstate.rbad: 592

LZstate.totr: 592
LZstate.totw: 592
LZstate.w: 592
LZstate.wbad: 592
LZstate.wr: 592
LZstate (typedef): 592
lzwrite(): 181, 588a, 588b
M-73: 154f, 399f
main-21(): 300e
main-359(): 556d
Mainarg: 301b, 565a
Mainarg.argc: 301b
Mainarg.argv: 301b
Mainarg (typedef): 565a
mainlauncher(): 300e, 301c
mainmem: 59e, 59e, 63g, 63i, 67a, 67b, 68a, 403a
mainp-20: 300b, 300e, 322c
mainstacksize: 300d, 300e
malloc(): 63g, 82a, 87d, 92d, 93b, 101a, 107b, 112, 160, 180, 198d, 215b, 215c, 216b, 216c, 256a, 257a, 321a, 393e, 434a, 434b, 479b, 480a, 491d, 499e, 533a, 537a, 556d, 587b, 600b
mallocalign(): 403a
MallocOffset-199: 69a, 69c, 402d
malloctopoolblock(): 403c
mallocz(): 67a, 92f, 316b, 348c, 360a, 403b, 472d, 484
MASK-190: 396a
MASK-40: 137b, 150e, 150i, 151a
MASK-56: 398b, 398c
MASK-58: 407a, 407b
MASK-71: 153c, 399d
Maskx-108: 85, 435d, 435d
Maxconnms-229: 350, 471
Maxcsreply-228: 471, 472c
Maxe-209: 128, 132a
maxf-52: 149a, 149b
MaxFlatBits-276: 160, 602b, 602b
Maxfmt-34: 518a, 518c, 519a
MaxHuffBits-268: 165, 602b, 602b
MaxHuffBits-300: 168b, 181, 589b, 590a, 592, 592
Maxintwidth-33: 506c, 506d
MaxLeaf-277: 160, 602b, 602b
MaxLeaf-299: 592, 592
MaxLitRun-304: 171, 179c, 181, 592
MaxMatch-298: 168b, 171, 592, 592
MaxOff-296: 170, 171, 181, 587e, 592, 592
MaxOffCode-314: 171, 179c, 592, 592
Maxpath-227: 348c, 350, 471
Maxpath-230: 344, 346a, 346b, 347a, 462g, 463a, 463b
Maxpath-4: 329b, 330, 331
Maxstring-226: 348c, 471, 472a

Maxstring-3: 329b, 329c, 330, 332
memccpy(): 81c, 81d, 442c, 460b
memchr(): 54, 202, 204b
memcmp(): 53, 135a, 136a, 489c
memcpy(): 49
memmark(): 426d
memmove(): 48, 49, 96c, 128, 181, 200a, 202, 204a, 204b, 206, 208, 256a, 272a, 273, 276a, 280, 281, 311b, 433b, 433c, 482b, 484, 489c, 491d, 499e, 596, 597a, 604b, 606a
memset(): 46b, 66, 67a, 128, 168b, 176, 181, 214d, 226b, 232a, 232b, 296c, 310a, 311b, 321a, 335a, 360b, 441g, 444c, 445a, 446b, 468c, 495a, 496h, 533a, 537a, 556d, 571a, 571b, 587e
MINBLOCKSIZE-131: 63b, 412a, 414a
MinMatch-297: 168b, 171, 179c, 587e, 592
MinMatchMaxOff-308: 171, 592
Minread-335: 97a, 586a
mkcrctab(): 587b
mkgzprecode(): 168b, 176, 181, 589b
mkprecode(): 174, 589b
mktemp(): 217b
modes-247: 478b, 478c
modf(): 122c, 137b, 137b, 139, 142f, 143g
msize(): 68a
mulascii(): 128, 136a
mulby(): 135b, 136a
muldiv(): 155c
N-192: 444f, 445a
N-55: 81b, 81c
N-57: 446a, 446b
N-98: 441f, 441g
N-99: 444b, 444c
NaN(): 126a, 128, 137a, 139, 143g, 144a, 144b, 151b, 153a, 390b
NANEXP-81: 126a, 126b, 126d, 126f, 127a
NANMASK-82: 126c, 126d
NANSIGN-83: 126e, 126f, 127a
Nbits-201: 128, 132a, 132a, 133a, 133b
nbra-347: 107b, 573h, 574c
nbrecv(): 313c, 314f, 315b
nbrecvp(): 315b
nbrecvul(): 314f
nbsend(): 313d, 314e, 315a
nbsendp(): 315a
nbsendul(): 314e
NCCLASS: 110, 114, 569b, 574b, 577, 580
NCCRUNE-337: 572f, 580
nclass-350: 107b, 573k, 578c
Nclen-271: 600b, 602b, 602b
Nclen-295: 176, 592, 592
Ndig-206: 128, 132a
needsrcquote(): 406c
needstack(): 303, 304a

netmkaddr(): 343
nettrans(): 344, 463b
newclass(): 578c, 580
newinst(): 574a, 574b, 575d, 576b
newthread(): 294a, 296b, 296c
NEXIT-54: 44b, 44c, 45b, 45d
NEXP10-31: 512b, 517a
nextc(): 579a, 579b, 580
nextchain(): 174, 590b, 590b
nextID(): 296a, 296c
NFN-11: 326b, 326g, 326h, 326j, 327a
NFN-59: 259, 260a, 260b, 260d
Nlitlen-269: 160, 600b, 602b, 602b
Nlitlen-293: 168b, 176, 181, 592, 592
nlits-331: 171, 181, 592
Nmant-202: 128, 132a, 132a
nmatches-332: 171, 181, 592
Node: 572d, 572d
Node.first: 572d
Node.last: 572d
Node (typedef): 572d
Noff-270: 160, 600b, 602b, 602b
Noff-294: 168b, 176, 181, 592, 592
NORM-191: 396b, 396c
NORM-76: 399i
Note: 326d, 562a
Note.inuse: 326d
Note.proc: 326d
Note.s: 326d
Note (typedef): 562a
noted(): 260d, 260e, 327a, 327b
notedeath(): 473c, 474b
notes-13: 326e, 326f, 327a, 327b
notifier(): 259, 260d
notify(): 259, 300e
NOT_MAGIC-113: 61a, 412c
Npadlong-198: 63g, 63h, 63i, 67a, 67b, 68a, 69a, 69b, 69c, 70, 403a, 403c
NPRIV: 295a, 295b, 336a, 336b
nrand(): 306, 407b
nsec(): 226a, 227
NSIGNIF-30: 512b, 512c, 513
NSTACK-338: 572g, 572h, 573b, 573e, 575b, 575c
NSUBEXP: 569b, 569b, 570a, 574c
ntruerand(): 407d
nulldir(): 214d
nullstring-259: 280, 466a, 466a
offbase-281: 160, 163, 602b
offbase-324: 168b, 179c, 592
OffBits-274: 602b

offcode-320: 168b, 171, 179c, 592
offextra-280: 160, 163, 602b
offextra-325: 168b, 171, 179c, 592
offtab-284: 160, 600a, 602b
offtab-328: 168b, 181, 592
oldtime(): 226a, 226b
One-207: 128, 132a, 132a, 133a, 133b
Onex: 388c, 388c
onex: 44b, 45b, 45d
Onex (typedef): 388c
onexlock: 45c, 45d
onnot-60: 259, 260b, 260d
onnote-15: 326g, 326j, 327a
onnotelock-17: 326i, 326j
onnotepid-16: 326h, 326j, 327a
onnotlock-61: 259, 260c
open(): 155a, 198d, 217a, 220, 226b, 227, 232a, 232b, 237c, 239b, 240, 257a, 258c, 299a, 318b, 324b, 330, 331, 335a, 344, 346a, 346b, 348c, 350, 354c, 356b, 360b, 401c, 463b, 468c, 483d, 486d, 494a, 495a, 496h
operand(): 107b, 574b
OPERATOR: 107b, 569b
operator(): 107b, 574b, 574c, 574c
optimize(): 107b, 577
OR: 110, 114, 569b, 576b, 577, 579b
order-254: 487a, 487a, 487b
outstandingprocs(): 474a, 474b
p0-143: 145g, 146a
p0-149: 142g, 142g, 143g
p0-161: 151b, 152d
p0-44: 149a, 149d
p0-62: 147a, 147a, 148a
p0-87: 141c, 142f
p1-142: 145f, 146a
p1-150: 142h, 142h, 143g
p1-162: 151b, 152e
p1-45: 149a, 149e
p1-63: 147b, 147b, 148a
p1-88: 141d, 142f
p2-141: 145e, 146a
p2-151: 143a, 143a, 143g
p2-163: 151b, 152f
p2-46: 149a, 149f
p2-64: 147c, 147c, 148a
p2-89: 141e, 142f
p3-140: 145d, 146a
p3-152: 143b, 143b, 143g
p3-164: 151b, 152g
p3-65: 147d, 147d, 148a
p3-90: 141f, 142f
p4-139: 145c, 146a

p4-153: 143c, 143c, 143g
p4-91: 142a, 142f
panicblock(): 417b
panicbuf-197: 402b, 402c
parsecs(): 348c, 475a, 476b
perr: 357e, 358b, 361
pickuperr(): 348c, 475b, 476b
PIPEMNT-24: 324a, 324b
pivot(): 32d
Plink: 432b
Plink (typedef): 432b
plock(): 59d, 401a
PLUS: 569b, 574c, 576b, 577, 579b
Poison-133: 63d, 409c, 410c, 412c
poolblockcheck(): 425d
poolcheck(): 425c
poolcheckarena(): 425a, 425b
poolcheckl(): 425b, 425c
poolcompactl(): 414c, 420a
pooldump(): 426b
pooldumparena(): 426a, 426c
pooldumpl(): 426a, 426b
poolfreel(): 66
poolnewarena(): 414c
popand(): 575d, 576b
popator(): 576a, 576b
postnote(): 258c, 337c, 474d
pow(): 139
pow10(): 140a, 140a, 513
ppanic(): 59d, 402c
pprint(): 59d, 402a
pqid(): 273, 276b
Pqueue: 293f, 558c
Pqueue.head: 293f
Pqueue.lock: 293f
Pqueue.tail: 293f
Pqueue (typedef): 558c
pread(): 192a, 227, 491d
preadv(): 492b
Prec-204: 128, 132a, 132a, 133a, 133b
print: 72b, 72b
printblock(): 417a
Printsize: 292, 293a
privalloc(): 301a, 488d
Private: 59b, 403d
Private.lk: 59b
Private.pid: 59b
Private.printfd: 59b
Private (typedef): 403d

privfree(): 488e
privinit-261: 488b, 488d, 488e
privlock-25: 335c, 336a, 336b
privlock-260: 488a, 488d, 488e
privmask-26: 335d, 335d, 336a, 336b
privs-262: 488c, 488d, 488e
Proc: 292, 558c
Proc.arg: 292
Proc.blocked: 292
Proc.exec: 292
Proc.exitstr: 292
Proc.lock: 292
Proc.needexec: 292
Proc.newproc: 292
Proc.next: 292
Proc.nextID: 292
Proc.nonotes: 292
Proc.nthreads: 292
Proc.pending: 292
Proc.pid: 292
Proc.printbuf: 292
Proc.ready: 292
Proc.readylock: 292
Proc.rforkflag: 292
Proc.sched: 292
Proc.splhi: 292
Proc.str: 292
Proc.thread: 292
Proc.threadint: 292
Proc.threads: 292
Proc.udata: 292
Proc.wdata: 292
Proc (typedef): 558c
proccreate(): 293g, 316b
procdata(): 336f
procexec(): 324b, 325a
procexecl(): 325a
procp-27: 293b, 293c, 293d, 301a
procrfork(): 293g, 293h
prof(): 358a
PSHORT-117: 61d, 412c
psstate(): 302b, 304a
pstring(): 273, 276a
punlock(): 59d, 401b
pushand(): 574b, 575b, 575d, 576b
pushator(): 107b, 574c, 575c
pushssl(): 239b
pushtls(): 240
putenv(): 257b

pwrite(): 192b, 499e
pwritev(): 500b
Q-74: 154f, 399g
q0-148: 145l, 146a
q0-154: 143d, 143d, 143g
q0-165: 151b, 152h
q0-47: 149a, 150a
q0-66: 147e, 147e, 148a
q0-92: 142b, 142f
q1-147: 145k, 146a
q1-155: 143e, 143e, 143g
q1-166: 151b, 152i
q1-48: 149a, 150b
q1-67: 147f, 147f, 148a
q1-93: 142c, 142f
q2-146: 145j, 146a
q2-156: 143f, 143f, 143g
q2-167: 151b, 152j
q2-49: 149a, 150c
q2-68: 147g, 147g, 148a
q2-94: 142d, 142f
q3-145: 145i, 146a
q3-95: 142e, 142f
q4-144: 145h, 146a
QIDFMT-232: 263b
qidtype(): 263c, 265, 481d
ql-248: 490b, 490b, 490f
qlock(): 246c, 350
qsep-96: 453a, 453a, 454b
qsort(): 32a
qsorts(): 32a, 33, 33
qstrfmt(): 525, 526
qtoken(): 453b, 454b
QUEST: 569b, 574c, 576b, 577, 579b
Queuing-249: 246c, 252, 490c
QueuingR-250: 248b, 250b, 490c
QueuingW-251: 249b, 249c, 250b, 490c
qunlock(): 247a, 350
quotefmtinstall(): 527c
Quoteinfo: 500d, 500d
Quoteinfo.nbytesin: 500d
Quoteinfo.nbytesout: 500d
Quoteinfo.nrunesin: 500d
Quoteinfo.nrunesout: 500d
Quoteinfo.quoted: 500d
Quoteinfo (typedef): 500d
quoterunestrdup(): 434b
quoterunestrfmt(): 527b, 527c
quotestrdup(): 434a

quotestrfmt(): 527a, 527c
R-75: 154f, 399h
rand(): 153b
RBRA: 110, 114, 569b, 574c, 576b, 579b
rcerror(): 107b, 573r, 574c, 574d, 575a, 576b, 578c, 580
rdline(): 101c, 102
rd_long(): 469b
rd_name(): 469a
read(): 155a, 192a, 192c, 200a, 202, 204b, 206, 226b, 232a, 232b, 237c, 239b, 240, 257a, 319c, 324b, 330, 331, 335a, 344, 346a, 348c, 350, 360b, 401c, 463b, 468c, 480a, 480b, 483d, 486d, 494a, 495a, 496h
read9pmsg(): 266e
readn(): 192c, 266e, 319e, 489c
readtimezone(): 468c
readv(): 492a
Ready: 296c, 298c, 302c, 304a, 305a, 340b
realloc(): 67b, 94b, 204a, 310a, 480b, 532c, 536d, 577
ReallocOffset-200: 69b, 70, 402d
reap(): 474b, 474d
recv(): 313b, 314b, 314d, 316e, 317
recvp(): 314d
recvul(): 314b
regcomp(): 107a
regcomp1(): 107a, 107b, 581a, 581b
regcomplit(): 581a
regcompnl(): 581b
regerr2(): 574d, 575d
regerror(): 107b, 119, 573r
regexec(): 109
regexec1(): 109, 110, 112
regexec2(): 109, 112
regkaboom-356: 107b, 573q, 573r
regsub(): 113
reject(): 347a
Relist: 569b, 569b
Relist.inst: 569b
Relist.se: 569b
Relist (typedef): 569b
Reljunk: 569b, 569b
Reljunk.eol: 569b
Reljunk.relist: 569b
Reljunk.reliste: 569b
Reljunk.reol: 569b
Reljunk.rstarts: 569b
Reljunk.startchar: 569b
Reljunk.starts: 569b
Reljunk.starttype: 569b
Reljunk (typedef): 569b
Rendezvous: 298c, 337c, 339e, 340b
rendezvous(): 304b, 305a, 490d

RENDHASH: 339b, 339c
reprog-336: 572e, 578c
rerrstr(): 234a, 316e, 323b, 511c
Resublist: 569b, 569b
Resublist.m: 569b
Resublist (typedef): 569b
revcode(): 165, 602a
revtab-285: 160, 167a, 602a, 602b
revtab-330: 168b, 590a, 592
rfork(): 231a, 300e, 322c, 323a, 323c, 476b
Rgrp: 339c, 558c
Rgrp.hash: 339c
Rgrp.lock: 339c
Rgrp (typedef): 558c
rlock(): 248b
rng_feed-79: 153c, 154d, 154d, 154f
rng_tap-78: 153c, 154c, 154c, 154f
rng_vec-77: 153c, 154b, 154c, 154f
Round-258: 58a, 58b
rregexec(): 117b
rregexec1(): 114, 117a, 117b
rregexec2(): 117a, 117b
rregsub(): 118
rsleep(): 251b
RUNE: 109, 110, 114, 117b, 569b, 574b, 579b
Rune1-107: 84d, 85, 435d, 436b
runeFmtStrFlush(): 532c, 533a
runefmtstrflush(): 530d
runefmtstrinit(): 533a, 533b
runelen(): 436a
runenlen(): 436b
runeseprint(): 530f
runesmprint(): 531a
runesnprint(): 531c
runesprint(): 531e
runestrcat(): 88a
runestrchr(): 86a, 87b, 87c, 88a, 88b, 114, 438e
runestrcmp(): 87a
runestrcpy(): 86b, 87d, 88a
runestrdup(): 87d, 433c, 434b
runestrecpy(): 438b
runestrlen(): 86a, 87d, 433c
runestrncat(): 438e
runestrncmp(): 439a
runestrncpy(): 439c
runestrrchr(): 87c
runestrstr(): 88b
runetochar(): 85, 207b, 436a
runevseprint(): 530f, 532a

runevsmprint(): 531a, 533b
runevsnprint(): 531c, 531e, 534a
RuneX-104: 84c, 84d, 85, 436b
runlock(): 249b
Running: 298c, 304a, 337c, 340b
runop(): 312, 313a, 313b, 313c, 313d
runthread(): 304a, 304b
rwakeup(): 252, 253a
rwakeupall(): 253a
rwx(): 478c, 478d
S0-213: 128, 132a
S1-214: 128, 132a
S2-215: 128, 132a
S3-216: 128, 132a
S4-217: 128, 132a
S5-218: 128, 132a
S6-219: 128, 132a
S7-220: 128, 132a
satan(): 144c, 146b
sbrk(): 58a, 360b, 400b
sbrkalloc(): 59d, 400b
sbrkmem-196: 59d, 59e, 59f
sbrkmempriv: 59c, 59d
sbrkmerge(): 59d, 400c
SEC2DAY-242: 228, 496d
SEC2HOUR-241: 228, 496c
SEC2MIN-240: 228, 496b
seek(): 210b, 226b, 257a, 330, 348c, 354c, 463b, 495a
semacquire(): 245b
semrelease(): 245c, 245d
send(): 313a, 314a, 314c, 316c, 316e, 317
sendp(): 314c, 323c, 341
sendul(): 314a, 324b
SEP-53: 218a, 218b
seprint(): 263c, 481d, 482b, 534c
setmalloctag(): 63g, 67a, 67b, 69a, 82a, 87d, 93a, 93b, 257a, 321a, 403a, 403b, 533a, 533b, 534e, 537a, 537b
setnetmtpt(): 493c
setrealloctag(): 63g, 67a, 67b, 69b, 403a
SHIFT-41: 137b, 150f, 150i, 151a
SHORT-116: 61c, 413b
SIG-43: 150h, 151a
Sigbit-205: 128, 132a
SIGN-223: 538e, 539c, 540b, 540d
sin(): 141b
sinh(): 148a, 148c
Sinstack: 585g
Sinstack.depth: 585g
Sinstack.fp: 585g
sinus(): 141a, 141b, 142f

sizeD2M(): 215c, 216c, 465a
sizeS2M(): 273, 277
skip(): 335b
sl-245: 354c, 356a, 356b
sleep(): 320d, 340b
Sleeping-252: 251b, 490c
smprint(): 484, 534e
snprint(): 257a, 257b, 299a, 330, 331, 333b, 343, 344, 346a, 346b, 347a, 348c, 350, 354c, 356b, 361, 463a, 463b, 483d, 486d, 493c, 535b
Sort (typedef): 31d
sprint(): 75b, 239b, 240, 258c, 333b, 482b, 484, 513
sq2m1-138: 145b, 146b
sq2p1-137: 145a, 146b
sqrt(): 137a, 139, 144a
sqrt2-51: 149a, 150d
sqrto2-160: 151b, 152c
srand(): 154e
sregfill(): 163, 167a, 167b, 600b
sregunget(): 168a, 599c
STACK-18: 316a, 316b
STAR: 569b, 574c, 576b, 577, 579b
START: 107b, 569b
stat(): 216b
statcheck(): 465c, 479b
state: 298c, 558c
State (typedef): 558c
strcat(): 80b, 119, 575a
strchr(): 80a, 80b, 80c, 80d, 82b, 89b, 257a, 257b, 330, 331, 332, 343, 350, 456c, 463a, 463b, 475a, 483d, 484
strcmp(): 81a, 228, 257a, 257b, 327b, 354c, 360b
strcpy(): 80b, 81c, 82a, 93a, 119, 218b, 225a, 225b, 237c, 323b, 330, 467e, 468c, 475b, 496h, 513, 575a
strcspn(): 441g
strdup(): 82a, 296c, 354c, 433b, 434a, 483d, 484
strecpy(): 442c, 484
stringsz(): 277, 279a
strlen(): 80a, 82a, 90, 93a, 107b, 119, 256a, 257b, 258c, 276a, 279a, 281, 299a, 323b, 330, 331, 333b, 347a, 348c, 350, 392c, 433b, 463b, 465a, 482b, 484, 505a, 511c, 517a
STRLEN-334: 92d, 92e, 93b
strncat(): 443a
strncmp(): 90, 102, 327a, 327b, 443c, 484
strncpy(): 332, 344, 346a, 354c, 443e, 463a, 463b, 473c, 474c, 474d, 478c, 493c
strpbrk(): 444c
strrchr(): 80d, 89c, 331, 344, 346a, 346b, 347a, 350, 484
strspn(): 445a
strstr(): 82b, 90, 208, 329d, 330, 348b, 475b, 476a
strtod(): 127c, 128, 390b, 513
strtok(): 446b
strtol(): 124c
strtoll(): 389d, 447c
strtoul(): 331, 350, 449

strtoull(): 450c
subidp-345: 107b, 573f, 575c, 576a, 576b
subidstack-344: 107b, 573e
SurrogateMax-111: 84d, 85, 435d
SurrogateMin-110: 84d, 85, 435d
swapb(): 32a, 32b
swapi(): 32a, 32c
sysfatal(): 92d, 93b, 94a, 94b, 95a, 96c, 97a, 102, 155a, 236a, 296c, 310a, 316b, 321a, 327b, 341
syslog(): 354c
sysname(): 354c, 494a
s_allocinstack(): 101a
s_append(): 95b
s_array(): 100d
s_copy(): 93a
s_free(): 93c, 100c
s_freeinstack(): 101b
s_getline(): 97b
s_grow(): 94a, 94b, 95a, 96c, 97a
s_incref(): 100b
s_memappend(): 95d
s_new(): 92d, 95b, 95d, 96a, 99b
s_newalloc(): 93a, 93b
s_parse(): 99b
s_putc(): 94a, 95b, 95d, 97b, 99b, 101c
s_rdinstack(): 102, 102
s_read(): 97a
s_read_line(): 96c
s_reset(): 96a
s_restart(): 96b
s_terminate(): 92d, 93b, 95a, 95b, 95d, 96c, 97a, 97b, 99b, 101c, 102
s_tolower(): 100a
s_unique(): 94b, 96a, 96b, 100c
T-103: 84b, 84d, 85, 435d, 436c
t16e-193: 455b, 455b, 456a
T2HDR-122: 61i, 66, 416a
t64d-85: 458c, 459a
t64e-86: 458d, 458d, 459b
Tab: 132b, 445d
tab-157: 140a, 140b
Tab.bp: 132b
Tab.cmp: 132b
Tab.siz: 132b
tab1-221: 134b, 135a
tab2-222: 135c, 136a
Tab (typedef): 445d
TAIL_MAGIC0-118: 61e, 413a
TAIL_MAGIC1-119: 61e, 413a
tan(): 143g
tanh(): 148c

TAP-70: 154f, 399c
Testx-109: 84d, 435d
Thread: 295a, 558c
Thread.alt: 295a
Thread.chan: 295a
Thread.cmdname: 295a
Thread.grp: 295a
Thread.id: 295a
Thread.inrendez: 295a
Thread.lock: 295a
Thread.moribund: 295a
Thread.next: 295a
Thread.nextstate: 295a
Thread.nextt: 295a
Thread.proc: 295a
Thread.rendbreak: 295a
Thread.rendhash: 295a
Thread.rendtag: 295a
Thread.rendval: 295a
Thread.ret: 295a
Thread.sched: 295a
Thread.state: 295a
Thread.stk: 295a
Thread.stksize: 295a
Thread.udata: 295a
Thread (typedef): 558c
threadcreate(): 296b
threaddata(): 336d
threadexits(): 301c, 321c, 323c, 557b
threadgetgrp(): 296b, 337b
threadgetname(): 299b
threadid(): 295c
threadint(): 316d, 339a
threadintgrp(): 338c
threadkill(): 337e
threadkillgrp(): 337d
threadmain(): 301c
threadnotify(): 326j
threadpid(): 298a
threadsetgrp(): 337a
threadsetname(): 299a
threadwaitchan(): 322b
threadxxx(): 337e, 338b, 339a
threadxxxgrp(): 337d, 338a, 338c
time(): 226a, 354c
times(): 363b, 495a
timezone-236: 225a, 467c, 468c
timezone-239: 228, 496a, 496h
tinterrupt(): 337c, 338a, 338b

TLSFinishedLen-224: 489b, 489c
tm2sec(): 228
tokenize(): 256a, 256b, 454b, 473c, 486d
tolower(): 29d, 100a, 136b, 393e, 513
totalmalloc-2: 320e, 321a
toupper(): 29b
tprivaddr(): 336c
tprivalloc(): 336a
tprivfree(): 336b
Tqueue: 298b, 558c
Tqueue.asleep: 298b
Tqueue.head: 298b
Tqueue.tail: 298b
Tqueue (typedef): 558c
treedelete(): 409c
treeinsert(): 409b
treelookup(): 409a
treelookupgt(): 410a
truerand(): 155a, 407d
Tx-106: 84d, 85, 435d, 436c
TZSIZE-235: 467b, 467c, 468c
TZSIZE-238: 495c, 496a, 496h
ULONG_MAX-194: 448b, 449
umuldiv(): 128, 155b, 155c
UNALLOC_MAGIC-125: 62d, 412c, 414a, 414c
uncblock(): 599c
unknown-263: 483c, 483c, 483d, 484, 485a
unlock(): 45d, 93c, 100b, 153c, 154e, 245c, 246c, 247a, 247b, 248b, 249a, 249b, 249c, 250a, 250b, 251b, 252, 259, 288c, 288d, 290a, 290e, 291, 294a, 296a, 296c, 297c, 298a, 302c, 304b, 305a, 305c, 306, 326j, 327a, 336a, 338a, 338b, 339e, 340b, 340d, 354c, 401b, 488d, 488e, 523c
unmount(): 324b
unquoterunestrdup(): 433c
unquotestrdup(): 433b
utfecpy(): 234a, 305c, 321c, 327b, 460b
utflen(): 89a, 505a, 511c
utfnlen(): 461c
utfrrune(): 89c
utfrune(): 89b, 90, 110, 255a, 406c, 453b, 453c, 454a, 454b
utfutf(): 90
UVLONG_MAX-97: 450b, 450c
vfprint(): 72c, 73, 75a, 402a
VLONG_MAX-37: 447a, 447c
VLONG_MIN-38: 447b, 447c
vseprint(): 234b, 236c, 333a, 354c, 402c, 534c, 536b
vsmprint(): 299a, 341, 534e, 537b
vsnprint(): 75b, 75c, 535b
wait(): 256a, 323c
waitpid(): 256b
wbufs-357: 197c, 198a, 198b, 198c

werrstr(): 234b, 256a, 256b, 266e, 317, 324b, 329d, 330, 344, 346a, 348b, 348c, 414c, 463b, 475a, 476b
wlock(): 249c
wrblock(): 179c, 181
wrdyncode(): 181, 588c
write(): 119, 192b, 208, 210a, 257b, 258c, 299a, 320b, 323b, 330, 331, 333b, 344, 346b, 347a, 348c, 350, 354c, 361, 397c, 402c, 463b, 489c, 535e
writev(): 500a
wstat(): 216c
wunlock(): 250b
xadd(): 512c, 513
xatan(): 146a, 146b
xcmp(): 128, 136b
xdtoa(): 513, 517a
xfree(): 485a, 485b
xioproc(): 316b, 316e
xsub(): 512d, 513
yield(): 305b, 306, 341
yrsize(): 228, 496g
yyclassp-355: 573p, 574b, 580
yyrune-354: 573o, 574b, 579b
ZlibCInfo: 187b, 606b, 607
ZlibDeflate: 187a, 187b, 598b, 606b, 607
ZlibMeth: 187b, 606b, 607
ZlibWin32k: 187a, 187b, 598b, 606b, 607
zlread(): 187a, 597c
zlwrite(): 187b, 605b
ZRead: 598a, 598a
ZRead.adler: 598a
ZRead.r: 598a
ZRead.rr: 598a
ZRead (typedef): 598a
ZWrite: 605c, 605c
ZWrite.adler: 605c
ZWrite.w: 605c
ZWrite.wr: 605c
ZWrite (typedef): 605c
_B2D-134: 63e, 416b, 420b
_D2B-135: 63f, 420c
_addv(): 539a
_assert: 235e, 235e
_badfmt(): 510a, 519a, 519c
_chanfree(): 290a, 290b, 290e, 309c
_charfmt(): 504a, 518d
_countfmt(): 509a, 518d
_ctype: 28b
_dial_string_parse(): 329d, 332, 348b
_dial: 300e, 347b, 348a, 348a
_divby(): 133c, 134a
_efgfmt(): 517b, 518d

_exits(): 45b, 305c, 323b, 327b, 476b
_flagfmt(): 509b, 518d
_floatfmt(): 517a, 517b
_fmtFdFlush(): 73, 522a, 522b, 535e
_fmtcpy(): 503a, 504a, 505a, 505c, 506d, 511c, 517a, 526
_fmtdispatch(): 74, 511a, 520
_fmtflush(): 74, 502a
_fmtinstall(): 519a, 519b, 519c
_fmtlock(): 519b, 519c, 523b
_fmtpad(): 502b, 503a, 503b, 525
_fmtrcpy(): 503b, 504b, 505c, 506b, 510a, 526
_fmtunlock(): 519b, 519c, 523c
_freeproc(): 294b
_freethread(): 297c
_ifmt(): 506d, 518d
_ioclose(): 318c, 319a
_iodial(): 328c, 329a
_ioopen(): 318a, 318b
_ioread(): 319b, 319c
_ioreadn(): 319d, 319e
_iosleep(): 320c, 320d
_iowrite(): 320a, 320b
_mainjmp-19: 300c, 300e, 322c
_needsquotes(): 434a, 528a
_newproc(): 293h, 294a, 300e
_percentfmt(): 506b, 518d
_procsplhi(): 290e, 291, 306, 328a
_procsplx(): 290e, 291, 306, 328b
_profdump(): 358a, 360b, 361
_profin(): 358b
_profinit(): 358a, 360a
_profmain(): 360b
_profout(): 359
_psstate-1: 302a, 302b
_qlockinit(): 300e, 490e
_quotestrfmt(): 526, 527a, 527b
_rendezvousp-253: 246c, 247a, 248b, 249b, 249c, 250b, 251b, 490d, 490d, 490e
_renewemptythread(): 110, 571a
_renewmatch(): 110, 114, 570a
_renewthread(): 110, 114, 570b
_rfmtpad(): 502c, 503a, 503b, 525
_rrenewemptythread(): 114, 571b
_runebsearch(): 83c
_runefmt(): 504b, 518d
_runeneedsquotes(): 434b, 528b
_runesfmt(): 506a, 518d
_sched(): 293h, 302c, 304a, 305b, 306, 321c, 324b, 339e
_schedexec(): 302c, 323a
_schedexecwait(): 323c, 324b

_schedexit(): 304a, 305c
_schedfork(): 302c, 322c
_schedinit(): 300e, 302c
_strfmt(): 505b, 518d
_subv(): 539b
_sysfatalimpl(): 236b, 236c
_sysfatal: 236a, 236b, 236b, 300e
_syslogopen(): 354c, 356b
_systhreadinit(): 300e, 301a
_s_alloc(): 92d, 92f, 93b, 100d
_threadassert(): 300e, 333b
_threadbreakrendez(): 302c, 338a, 338b, 340d
_threaddebug(): 289b, 289c, 296c, 302c, 304a, 304b, 305a, 309b, 309c, 311a, 323b, 323c, 324b, 327a, 327b, 334g, 338b, 339e
_threaddebuglevel: 333c, 334g
_threaddial(): 300e, 329d
_threadexitsallstatus: 302c, 304b, 305a, 306, 321b, 327b
_threadflagrendez(): 302c, 337c, 340c
_threadgetproc(): 293c, 293h, 295c, 296b, 298a, 299a, 299b, 303, 304a, 306, 321c, 323c, 324b, 326j, 327b, 328a, 328b, 333b, 334g, 336c, 336d, 336e, 336f, 337a, 337b, 339e
_threadinitstack(): 557b
_threadmalloc(): 289b, 294a, 296c, 300e, 321a
_threadnopasser: 326a
_threadnote(): 300e, 327b
_threadnotefd: 334h
_threadpasserpid: 334i
_threadpq: 293e, 294a, 298a, 305c, 338a, 338b
_threadready(): 296c, 302c, 305a, 340b
_threadrendezvous(): 290e, 300e, 306, 339e
_threadrgrp: 339d, 339e, 340d
_threadsetproc(): 293d, 302c
_threadsysfatal(): 300e, 333a
_threadwaitchan: 322a, 322b, 323c
_times(): 335a
_workerdata(): 336e
_xdec(): 288d
_xinc(): 288c
__anon_enum_15: 458b
__anon_enum_16: 435d
__anon_enum_17: 61a
__anon_enum_18: 61e
__anon_enum_19: 62b
__anon_enum_1: 329b
__anon_enum_20: 62d
__anon_enum_21: 62f
__anon_enum_22: 63a
__anon_enum_23: 63b
__anon_enum_24: 419a
__anon_enum_25: 402d

__anon_enum_26: 132a
__anon_enum_27: 489b
__anon_enum_28: 471
__anon_enum_29: 462g
__anon_enum_2: 290c
__anon_enum_30: 216a
__anon_enum_32: 215a
__anon_enum_36: 490c
__anon_enum_38: 58b
__anon_enum_39: 587a
__anon_enum_3: 566a
__anon_enum_40: 602b
__anon_enum_41: 607
__anon_enum_42: 592
__anon_enum_43: 586a
__anon_enum_5: 558b
__anon_enum_6: 512b
__anon_enum_7: 506c
__anon_enum_8: 518a
__anon_struct_1.d: 539c
__anon_struct_14.cmp: 31d
__anon_struct_14.es: 31d
__anon_struct_14.swap: 31d
__anon_struct_14: 31d, 31d
__anon_struct_1: 539c
__anon_struct_31.dldiff: 467c
__anon_struct_31.dlname: 467c
__anon_struct_31.dlpairs: 467c
__anon_struct_31.stdiff: 467c
__anon_struct_31.stname: 467c
__anon_struct_31: 467c, 467c
__anon_struct_33.dldiff: 496a
__anon_struct_33.dlname: 496a
__anon_struct_33.dlpairs: 496a
__anon_struct_33.stdiff: 496a
__anon_struct_33.stname: 496a
__anon_struct_33: 496a, 496a
__anon_struct_34.consd: 356a
__anon_struct_34.consfd: 356a
__anon_struct_34.d: 356a
__anon_struct_34.fd: 356a
__anon_struct_34.name: 356a
__anon_struct_34: 356a, 356a
__anon_struct_35.p: 490b
__anon_struct_35.x: 490b
__anon_struct_35: 490b, 490b
__anon_struct_37.fd: 487d
__anon_struct_37.pid: 487d
__anon_struct_37: 487d, 487d

__anon_struct_9.fmt: 518c
__anon_struct_9.nfmt: 518c
__anon_struct_9: 518c, 518c
__assert: 235f, 235g, 300e

Bibliography

[Han96] David R. Hanson. *C Interfaces and Implementations: Techniques for Creating Reusable Software*. Addison-Wesley, 1996. cited page(s) 12, 13, 36

[HS02] Samuel P. Harbison and Guy L. Steele. *C: A Reference Manual*. Prentice Hall, 5th edition, 2002. cited page(s) 13

[Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. cited page(s) 13

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. cited page(s) 13

[Pad09] Yoann Padioleau. Syncweb, literate programming meets unison, 2009. https://github.com/aryx/syncweb. cited page(s) 13

[Pad14] Yoann Padioleau. *Principia Softwarica: The Plan 9 Kernel 9*. 2014. cited page(s) 13, 21, 36, 37, 38, 41, 44, 366

[Pad15a] Yoann Padioleau. *Principia Softwarica: The ARM Assembler 5a*. 2015. cited page(s) 13, 17, 19, 36, 120

[Pad15b] Yoann Padioleau. *Principia Softwarica: The ARM Linker 5l*. 2015. cited page(s) 17, 19, 20, 40, 357, 366

[Pad16a] Yoann Padioleau. *Principia Softwarica: The Plan 9 C Compiler 5c*. 2016. cited page(s) 36, 366

[Pad16b] Yoann Padioleau. *Principia Softwarica: The Plan 9 Debuggers and Tracers*. 2016. cited page(s) 353, 354, 366

[Pad16c] Yoann Padioleau. *Principia Softwarica: The Plan 9 Graphics System draw*. 2016. cited page(s) 15

[Pad16d] Yoann Padioleau. *Principia Softwarica: The Plan 9 Windowing System rio*. 2016. cited page(s) 366

[Pad18] Yoann Padioleau. *Principia Softwarica: The Plan 9 Shell rc*. 2018. cited page(s) 366

[Pad26a] Yoann Padioleau. *Principia Softwarica: The Plan 9 Widgets libpanel*. 2026. cited page(s) 15

[Pad26b] Yoann Padioleau. *Principia Softwarica: The The Plan 9 Profilers*. 2026. cited page(s) 353, 357, 361

[Pla92] P. J. Plauger. *The Standard C Library*. Prentice Hall, 1992. cited page(s) 12, 13

[PTVF07] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 3rd edition, 2007. cited page(s) 120