Principia Softwarica: The Plan 9 Profilers
version 0.1

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
Ken Thompson and Rob Pike

March 24, 2026

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

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

Contents

1 Introduction 5
1.1 Motivations . . . 5
1.2 The Plan 9 profilers: `prof`, `tprof`, `trace`, `stats` . . . 5
1.3 Other profilers . . . 6
1.4 Getting started . . . 6
1.5 Requirements . . . 6
1.6 About this document . . . 7
1.7 Copyright . . . 7
1.8 Acknowledgments . . . 7

2 Overview 8
2.1 Profiler principles . . . 8
2.1.1 User time vs system time vs real time . . . 8
2.1.2 Ticks . . . 8
2.1.3 Timer interrupt and preemptive scheduling . . . 9
2.1.4 Instrumentation profilers . . . 10
2.1.5 Sampling profilers . . . 10
2.1.6 Tracing and monitoring profilers . . . 10
2.2 `time` command-line interface . . . 10
2.3 `prof` and `tprof` command-line interfaces . . . 10
2.4 `trace` and `stats` graphical user interfaces . . . 11
2.5 `iostats` filesystem interface . . . 11
2.6 Profiling `hello` . . . 12
2.7 Code organization . . . 13
2.8 Software Architecture . . . 14
2.9 Book structure . . . 15

3 Kernel Support 16
3.1 Ticks, `HZ`, and times . . . 16
3.2 Wait message . . . 16
3.3 `/proc/<pid>/status` and `/dev/cputime` . . . 17
3.4 `/proc/<pid>/profile` . . . 17
3.5 `/proc/trace` . . . 17
3.6 `TOS` . . . 18

4 Linker Support 19
4.1 `5l -p` . . . 19
4.2 `5l -p1` . . . 20
4.3 The symbol table . . . 20

5 Timing a Command: `/bin/time` **22**
5.1 `main()` . . . 22
5.2 Error management . . . 23
5.3 Extra display . . . 24

6 Instrumentation-based Profiler: `/bin/prof` **26**
6.1 Data structures . . . 26
6.2 `main()` . . . 27
6.3 Symbol table loading: `syms()` . . . 29
6.4 Profiling data loading: `datas()` . . . 29
6.5 Displaying data: `plot()` . . . 30
6.6 Call graph, `prof -d` . . . 33
6.6.1 `prof -d -r` . . . 35

7 Sampling-based Profiler: `/bin/tprof` **36**
7.1 Example: `longrun.c` . . . 36
7.2 `main()` . . . 37
7.3 Reading the symbol table from the binary . . . 38
7.4 Reading the profile data from `/proc/<pid>/profile` . . . 39
7.5 Displaying profile . . . 39
7.6 Error management . . . 40

8 Profiling the Kernel: `/bin/kprof` **42**

9 Show real-time Behavior: `/bin/trace` **46**
9.1 Data structures . . . 46
9.2 `main()` . . . 47

10 System Monitoring `/bin/stats` **59**
10.1 Data structures . . . 59
10.2 `main()` . . . 63

11 IO Monitoring: `/bin/iostats` **67**

12 Conclusion **68**
12.1 Patterns and techniques . . . 68
12.2 Connections to other books . . . 68
12.3 Beyond the Plan 9 profilers . . . 69

A Extra Code **70**
A.1 `profilers/` . . . 70
A.1.1 `misc/time.c` . . . 70
A.1.2 `misc/kprof.c` . . . 70
A.1.3 `misc/prof.c` . . . 70
A.1.4 `misc/trace.c` . . . 72
A.1.5 `misc/tprof.c` . . . 74
A.1.6 `misc/stats.c` . . . 75
A.2 `iostats/` . . . 98
A.2.1 `iostats/statfs.h` . . . 98
A.2.2 `iostats/globals.c` . . . 101
A.2.3 `iostats/iostats.c` . . . 102

A.2.4 `iostats/statsrv.c` . . . 114

Glossary 127

Index 128

References 136

Chapter 1

Introduction

The goal of this book is to present in full details the source code of a few profilers.

1.1 Motivations

Writing correct code is only half the job—making it fast enough is the other half. A profiler tells you *where* your program actually spends its time, replacing guesswork with measurement. Without a profiler, programmers often optimize the wrong function, or worse, make the code harder to read for no measurable gain. As the saying goes: "premature optimization is the root of all evil" (Knuth), but *informed* optimization, guided by profiling data, is simply good engineering.

Understanding how a profiler works gives you insight into the interplay between user programs, the compiler toolchain, and the kernel. A profiler like `prof` relies on the linker to instrument function calls, while `tprof` relies on the kernel to sample the program counter at regular intervals. Studying these tools therefore ties together concepts from LINKER book [Pad15] and KERNEL book [Pad14].

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

- How do you measure the time spent by a program? What is the difference between user time, system time, and real time (wall-clock time)?
- How does a sampling profiler work? How does the kernel periodically record where a program is executing?
- How does an instrumenting profiler work? What code does the linker insert at function entry and exit to record call counts and timing?
- How can you trace system calls and visualize system-wide activity in real time?
- What is the role of the `/proc` filesystem and the `TOS` (Top of Stack) structure in supporting profiling on Plan 9?

1.2 The Plan 9 profilers: `prof`, `tprof`, `trace`, `stats`

Unlike most books in Principia Softwarica, which focus on a single program (one assembler, one linker, one shell), this book covers a *collection* of small profiling tools. The reason is that profiling is not a single problem but a spectrum: you might want to time an entire command (`time`), identify which functions are hot (`prof`, `tprof`), trace individual system calls (`trace`), monitor system-wide resource usage (`stats`), or measure the I/O cost of a program (`iostats`). Each tool addresses a different point on that spectrum, and each is small enough—a few hundred lines of C—that together they still form a manageable codebase.

Like for most books in Principia Softwarica, I chose Plan 9 programs because those programs are simple, small, elegant, open source, and they form together a coherent set. The Plan 9 profilers total roughly 3000 lines of code, compared to the tens of thousands of lines in Linux's `perf` tool alone.

1.3 Other profilers

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

- `gprof` [?] is the classic UNIX profiler. It combines instrumentation (the compiler inserts code at each function entry to record call arcs) with statistical sampling (the kernel periodically interrupts the program to record its program counter). The combination gives both call counts and time estimates per function. `gprof` was introduced with 4.2BSD and became the standard profiling tool on UNIX systems for decades. The Plan 9 `prof` tool works on a similar principle but is simpler, relying on the linker rather than the compiler for instrumentation. GNU `gprof`, part of the `binutils` package, is a reimplementation of the original BSD tool that follows the same design (compiler-inserted call arcs plus PC sampling) and reads the same `gmon.out` file format. The original BSD `gprof` was about 3000 LOC; GNU `gprof` (part of `binutils`) is roughly 15 000 LOC.

- OProfile is a system-wide statistical profiler for Linux that uses hardware performance counters when available. Unlike `gprof`, which requires recompiling the target program, OProfile can profile any running program, including the kernel itself, with low overhead.

- `perf` is the modern Linux profiling framework. It subsumes most of what OProfile and `gprof` do, and adds tracing, hardware counter access, and visualization. However, the `perf` codebase is enormous—tens of thousands of lines of C—making it unsuitable for a teaching book.

Figure 1.1 presents a timeline of major profiling tools. The Plan 9 profilers are small and self-contained, making them ideal for a teaching book. They cover the three main profiling approaches—timing (`time`), instrumentation (`prof`), and sampling (`tprof`)—plus system-wide monitoring (`stats`, `trace`) and I/O tracing (`iostats`), all in a fraction of the code found in modern tools like `perf`.

1.4 Getting started

To get started you first need to get the source of the Plan 9 version described in this book. See https://www.principia-softwarica.org/getting-started.html. To use `prof`, you must link your target program with the `-p` flag so that the linker inserts profiling instrumentation. The `tprof` tool instead relies on kernel support (`/proc/<pid>/profile`) and requires no special linking.

Note that the profiling tools described in this book (`prof`, `tprof`, `kprof`, `trace`, `stats`) depend on kernel support (`/proc`, `profclock()`, the `Tos` structure) and can only run under a full Plan 9 system—they are not available through plan9port. Only `time`, which simply measures elapsed time, could work outside Plan 9.

1.5 Requirements

This book assumes familiarity with C and with Plan 9's basic conventions (the `/proc` filesystem, the `Waitmsg` structure, the `rfork` system call). Readers who have gone through Kernel book [Pad14] will find the kernel-support chapter straightforward; readers who have read Linker book [Pad15] will immediately understand how `5l -p` instruments function calls.

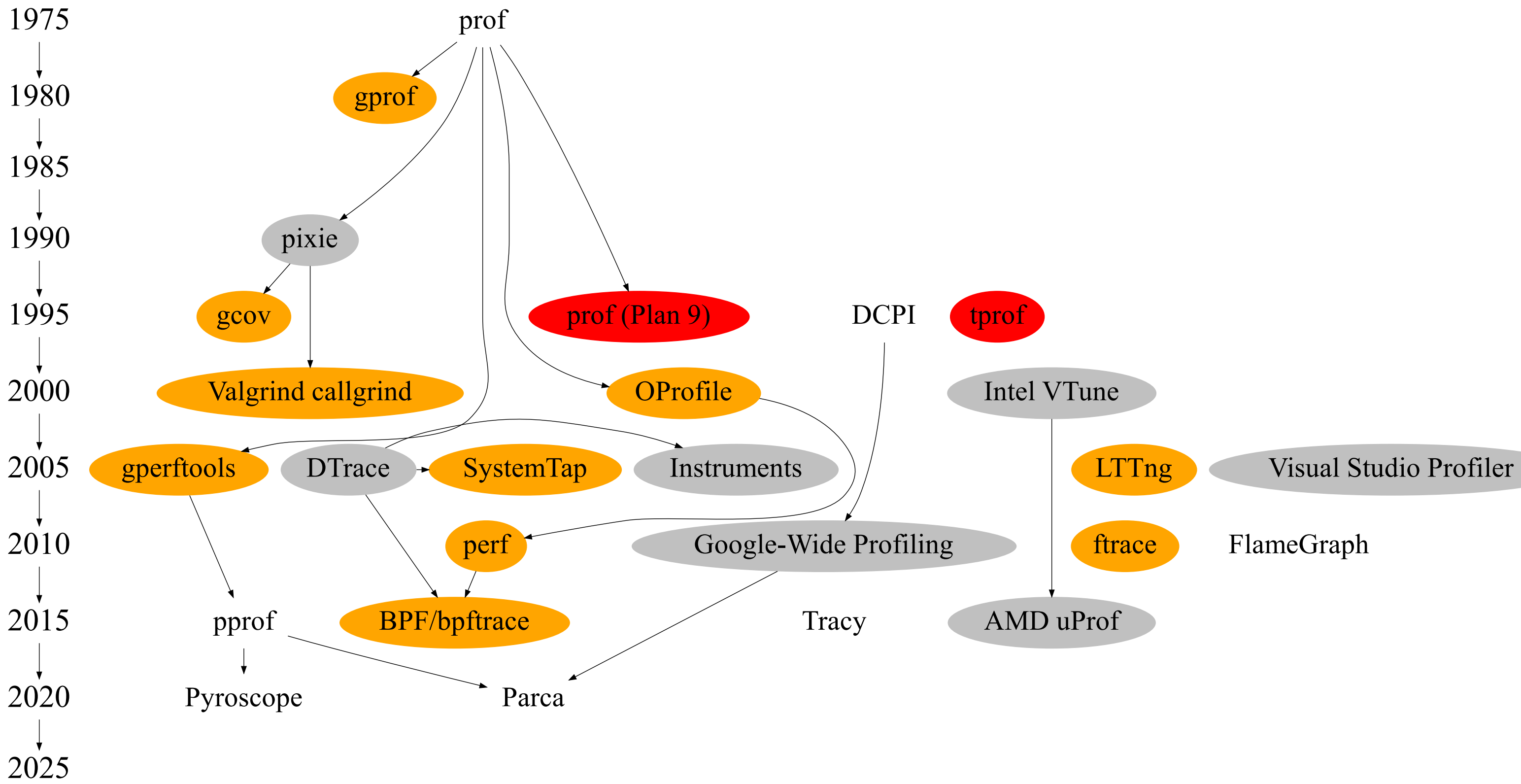


Figure 1.1: Profilers timeline

1.6 About this document

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

1.7 Copyright

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

1.8 Acknowledgments

Chapter 2

Overview

In this chapter, I will give a bird's-eye view of the Plan 9 profiling tools before diving into the code in later chapters. I will first recall general profiling principles, then present the command-line interfaces of each tool, show a sample profiling session, and finally describe the software architecture and code organization.

2.1 Profiler principles

There are two fundamental approaches to profiling. The first is instrumentation: the toolchain (compiler or linker) inserts extra code at each function entry and exit to record call counts and elapsed time. This is what Plan 9's `prof` does—the linker `5l -p` inserts calls to `_profin` and `_profout` around every function. The result is a precise call tree with exact counts, but the overhead can be significant.

The second approach is sampling: the kernel periodically interrupts the running program (typically on each clock tick) and records its current program counter. After enough samples, the histogram reveals where the program spends most of its time. This is what `tprof` does, via the `/proc/<pid>/profile` file. Sampling has low overhead but gives only statistical estimates—it cannot tell you how many times a function was called, only how often the CPU was found executing within it.

2.1.1 User time vs system time vs real time

Before diving into the profiling tools themselves, it is worth clarifying a few concepts that come up repeatedly.

When you time a program, three numbers are reported: user time, system time, and real time (also called wall-clock time). User time is the time the CPU spent executing the program's own code. System time is the time the CPU spent in the kernel on behalf of the program (handling system calls, page faults, etc.). Real time is the elapsed time from start to finish, including time the program spent waiting (for I/O, for the scheduler to give it a turn, etc.). A program that does heavy computation will have user time close to real time. A program that does lots of I/O will have high system time. A program that sleeps or waits for input will have real time much larger than user + system.

2.1.2 Ticks

How does the kernel actually measure these times? It uses a periodic timer interrupt called the clock tick. On each tick (at a frequency called `HZ`, e.g., 100 or 1000 times per second), the kernel interrupts whatever is running and updates the current process's time accounting: if the process was in user mode, its user time is incremented; if in kernel mode, its system time.

This tick-based accounting is coarse-grained: the kernel simply checks `p->insyscall` when the tick fires and increments user or system time accordingly. If a short syscall starts and finishes between two ticks, it is never seen in kernel mode and the time goes to user. Conversely, if the tick fires during a syscall, the entire tick counts

as system time. (The `Tos` structure used by `prof` provides a separate, cycle-accurate mechanism via `kcycles` and `pcycles`—but that is for per-function profiling, not for the user/sys/real counters.) Figure 2.1 illustrates this with a concrete example.

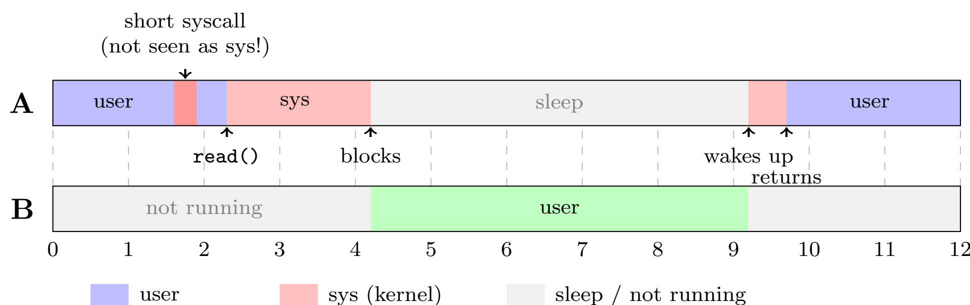


Figure 2.1: Tick-based time accounting with two interleaved processes. Ticks are numbered 0–12 along the bottom. The short syscall between ticks 1 and 2 is invisible to tick accounting (counted as user). The `read()` blocks shortly after tick 4 and A wakes up shortly after tick 9. The syscall returns at 9.7, but tick 10 is counted as user because `insyscall` is already false when it fires. After tick 12: A has user=5, sys=2, real=12; B has user=5, sys=0, real=8.

Note that the timer interrupt handler itself is kernel code, but it executes so quickly (a few microseconds) compared to a full tick interval (10 milliseconds at `HZ`=100) that its cost is negligible and not separately accounted for.

2.1.3 Timer interrupt and preemptive scheduling

The tick interrupt is also what makes preemptive scheduling possible: it gives the kernel a chance to check whether the current process has used its time slice and should yield the CPU to another process.

Profilers exploit this interrupt in two ways. First, the tick is when per-process time counters are updated (which timing tools eventually read). Second, a separate timer at a different frequency can sample the program counter for sampling-based profiling. Using a frequency that is not a multiple of `HZ` avoids aliasing—without this, the profiler might systematically miss code that runs in sync with the clock. The idea behind program counter sampling is simple: when the timer fires, the kernel looks at the interrupted instruction's address (the PC) and increments a counter for that address range. After many samples, functions where the program spends the most time will have the highest counts.

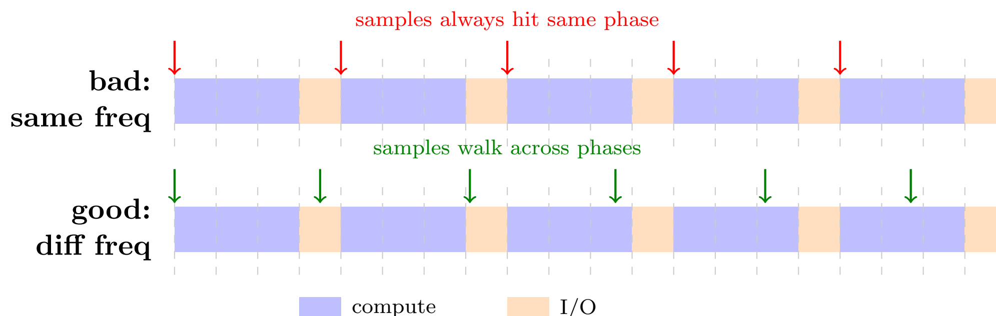


Figure 2.2: The aliasing problem. Top: when the profiling timer fires at the same frequency as the program's repeating pattern, every sample hits the same phase (here, always "compute"), missing the I/O portion entirely. Bottom: using a different frequency (e.g., 113 instead of 100), samples gradually walk across all phases, giving a representative picture.

2.1.4 Instrumentation profilers

An instrumentation profiler modifies the program so that extra code runs at each function entry and exit. This instrumentation can collect several kinds of data:

- *Call counts*: how many times each function was called. This is the simplest form of instrumentation.
- *Per-function time*: how much time was spent inside each function, including or excluding its callees.
- *Call graph*: which function called which, and how much time was spent along each call arc.

The main advantage of instrumentation is precision: you get exact counts and a complete call tree. The main disadvantage is overhead—every function call now executes extra instructions, which can slow the program down and distort the very timings you are trying to measure. Moreover, the instrumentation needs extra data structures to store the counts and times—typically a buffer of profiling records allocated at program startup.

2.1.5 Sampling profilers

A sampling profiler takes a different approach: it periodically interrupts the program and records its current program counter (PC). Over thousands of samples, the histogram of PCs reveals which functions consume the most CPU time. In principle, the kernel could also walk the stack at each sample to record not just the current PC but also the chain of callers. Some modern profilers (like Linux's `perf`) do this, but it requires the kernel to understand the stack frame layout, which is architecture-specific and can be fragile. Plan 9's `tprof` keeps things simple and only records the top-level PC.

Sampling has two key advantages: it requires no recompilation or relinking, and it has very low overhead (just one interrupt every few milliseconds). The disadvantage is that you only get statistical estimates—a function that runs for a very short time may never be sampled, and you cannot recover call counts or call graphs from sampling data alone. (The binary must still contain a symbol table so that the profiler can map PC values back to function names, but no relinking or recompilation is needed.)

2.1.6 Tracing and monitoring profilers

The Plan 9 profiling tools also include a third category: tracing and monitoring. `trace` shows real-time scheduler events (process switches, deadlines, releases) in a graphical timeline, while `stats` displays rolling graphs of system-wide counters (syscalls/sec, context switches, memory usage, etc.). These tools do not profile a single program but rather observe the whole system. These tools are especially useful when debugging performance problems that involve multiple processes or the kernel itself—for instance, understanding why a system feels sluggish, or spotting a process that consumes too many context switches or syscalls.

2.2 `time` command-line interface

⟨`main()` *(time.c) usage if no args* 10a⟩≡ (22)

```
if(argc <= 1){
    fprint(2, "usage: time command\n");
    exits("usage");
}
```

2.3 `prof` and `tprof` command-line interfaces

⟨`main()` *(prof.c) print usage and exit* 10b⟩≡ (27d)

```
fprint(STDERR, "usage: prof [-dr] [8.out] [prof.out]\n");
exits("usage");
```

`prof` reads a profile data file (`prof.<pid>`) produced by an instrumented program and correlates it with the program's symbol table. For each function, it prints the percentage of time spent there, the absolute time, and the call count. The `-d` flag prints the dynamic call graph instead, showing the tree of calls indented by depth. The profiling runtime writes its data to `prof.<pid>` (not `prof.out`), so in practice you pass the filename as a second argument: `prof 5.out prof.123`.

⟨`main()` *(tprof.c) sanity check argc and print usage* 11a⟩≡ (37)

```
if(argc != 2 && argc != 3)
    error(false, "usage: tprof pid [binary]");
```

`tprof` works differently: it reads `/proc/<pid>/profile`, which contains interrupt-time samples of the program counter collected by the kernel. Since sampling cannot track calls, `tprof` has no `-d` (call graph) option—it only reports time per function.

To enable sampling, you first write "`profile`" to `/proc/<pid>/ctl`, let the program run, then run `tprof <pid>` while the process is still alive.

⟨`main()` *(kprof.c) sanity check* `argc` *and print usage* 11b⟩≡ (42)

```
if(argc != 3)
    error(0, "usage: kprof text data");
```

`kprof` is a variant of `tprof` that profiles the kernel itself. It reads the data accumulated by the `kprof(3)` device (`/dev/kpdata`), which records one count per clock tick per 8-byte range of kernel text.

2.4 `trace` and `stats` graphical user interfaces

`trace` and `stats` are graphical programs that display system activity in real time. `trace` reads events from `/proc/trace` and draws a timeline for each traced process: colored blocks show when a process is running, and arrows mark scheduling events (releases, deadlines, yields, overruns). The user can zoom in and out with `+`/`-` and pause with `p`.

`stats` reads counters from kernel pseudo-files (`#c/sysstat`, `#c/swap`, `/net/ether0/0/stats`) and plots rolling graphs updated once per second. It can display dozens of metrics: system load, memory usage, context switches, syscalls, interrupts, TLB misses, network packets, and more. It can even monitor multiple machines simultaneously, drawing their graphs in adjacent columns. `stats` is similar in spirit to tools like GNOME System Monitor on Linux or Activity Monitor on macOS, but it runs in a terminal-sized window and focuses on kernel-level counters rather than per-process resource usage.

The lower-case flags select which metrics to display. The most commonly used are `-l` for system load (the default), `-m` for memory usage, `-c` for context switches per second, `-s` for syscalls per second, `-i` for interrupts per second, and `-e` for network packets. Upper-case flags control the display: `-S` sets a scale factor, `-L` switches to logarithmic axes, and `-Y` adds value markers along the y axis.

⟨*function usage (misc/stats.c)* 11c⟩≡ (95)

```
void
usage(void)
{
    fprint(2, "usage: stats [-O] [-S scale] [-LY] [-%s] [machine...]\n", argchars);
    exits("usage");
}
```

Uses `argchars` 62h.

2.5 `iostats` filesystem interface

`iostats` takes yet another approach: it is a user-level file server that interposes itself between a program and the regular file server, intercepting all 9P messages. When the program exits, `iostats` prints a report showing how

much data was read and written, the number and latency of each 9P message type, and a per-file I/O summary. This is a classic Plan 9 design: instead of adding tracing hooks inside the kernel, you insert a proxy file server in the namespace. For example, `iostats ls` runs `ls` through the proxy and reports how many `read`, `stat`, and `open` messages that simple command generates.

2.6 Profiling `hello`

To get a feel for the tools, consider profiling the simplest possible program, `hello.c`:

⟨hello.c 12a⟩≡
```
#include <u.h>
#include <libc.h>

void main() {
  print("hello world\n");
  exits(nil);
}
```

For a more interesting example, consider a program that calls `foo` in a loop and `bar` once, where `foo` itself calls `bar` as in `funcs.c`:

⟨funcs.c 12b⟩≡
```
#include <u.h>
#include <libc.h>

void bar(void) { /* some work */ }
void foo(void) { bar(); /* more work */ }
void main(void) {
    int i;
    for(i = 0; i < 100; i++)
        foo();
    bar();
    exits(nil);
}
```

With `time`:

```
% time hello
hello world
0.00u 0.00s 0.01r     hello
```

The three numbers are user time, system time, and real (wall-clock) time, all in seconds.

With `prof`, you first link with `-p` to enable instrumentation, run the program, then display the results:

```
% 5l -p -o hello hello.5
% ./hello
hello world
% prof hello hello.42
    %     Time    Calls  Name
  80.0    0.004       1  main
  20.0    0.001       1  libc_print
   0.0    0.000       1  memset
   0.0    0.000       1  getpid
   0.0    0.000       1  read
...
```

For `funcs` we get instead:

```
% 5l -p -o funcs funcs.5
% ./funcs
% prof funcs funcs.42
  %     Time    Calls  Name
100.0   0.003       1  main
  0.0   0.000       1  memset
  0.0   0.000     101  bar
  0.0   0.000     100  foo
...
```

The `-d` flag shows the dynamic call graph instead:

```
% prof -d funcs funcs.42
main:3
. foo:0/100
.   bar:0/100
. bar:0
. getpid:0
.   memset:0
...
```

This reveals that `bar` was called from two sites: 100 times from `foo` and once directly from `main`.

With `tprof`, you start a longer-running program, enable profiling, and then read the sampling data:

```
% longrun &
% echo profile > /proc/$apid/ctl
% # ... let it run ...
% tprof $apid
  %     Time  Name
 45.0    4.50  compute
 30.0    3.00  process
 15.0    1.50  sort
 10.0    1.00  main
```

(In `rc`, `$apid` holds the pid of the last process started with `&`.)

With `iostats`, you see the 9P traffic:

```
% iostats hello
hello world
read     32 bytes, 1 ops     ...
write    12 bytes, 1 ops     ...
protocol 48 bytes
rpc       4
```

2.7 Code organization

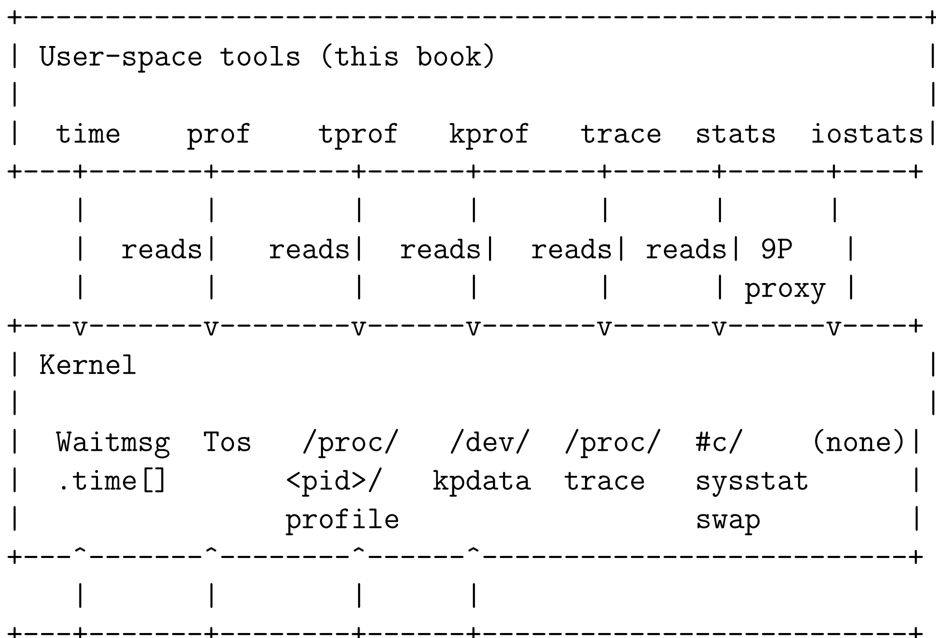
Table 2.1 presents the source files of the profiling tools, together with the main entities (e.g., structures, functions) each file defines and the corresponding chapter in which its code is discussed.

Function	Ch.	File	Entities	LOC
timing a command	5	misc/time.c	main()X add()[23b]	111
profiling data structures	6	misc/prof.c	Data[26a] Pc[27c] Acc[27a]	414
profiling display	6	misc/prof.c	main()X syms()[29b] graph()[34c] plot()[30e]	
sampling profiler	7	misc/tprof.c	main()X COUNTER	159
kernel profiler	8	misc/kprof.c	main()X COUNTER	154
trace data structures	9	misc/trace.c	TEvent[73c] Task[73c]	841
trace GUI	9	misc/trace.c	threadmain()[48a] doevent()[54] drawtrace()[55]	
stats data structures	10	misc/stats.c	Graph[59b] Machine[60a]	1622
stats GUI	10	misc/stats.c	main()X mkcol()[76a]	
iostats data structures	11	iostats/statfs.h	Fid Stats Rpc	174
iostats file server	11	iostats/iostats.c	main()	638
9P message handling	11	iostats/statsrv.c	rread() rwrite()	726
iostats globals	11	iostats/globals.c		45
Total				4884

Table 2.1: Chapters and source files of the profiling tools.

2.8 Software Architecture

The profiling infrastructure spans three layers of the system. At the bottom, the *kernel* provides the raw data: the Waitmsg.time array (user/sys/real times for time), the /proc/<pid>/profile file (PC sampling histogram for tprof), /proc/trace (scheduler events for trace), and /dev/kpdata (kernel PC histogram for kprof). In the middle, the *linker* (5l -p) instruments programs by inserting calls to _profin/_profout at function entry and exit, and the runtime library (lib_core/libc/profile.c) manages the profiling buffer and writes prof.<pid> on exit. At the top, the *user-space tools* presented in this book read this data, correlate it with symbol tables, and present it to the programmer.



```
| Linker (5l -p)                                        |
|                                                       |
| _profin/_profout    profclock   devkprof               |
| instrumentation     sampling   kernel sampling        |
+-------------------------------------------------------+
```

2.9 Book structure

The rest of the book is organized as follows. Chapters 3 and 4 describe the infrastructure that the profiling tools depend on: how the kernel exposes timing data and PC histograms, and how the linker instruments function calls. Then each tool gets its own chapter: `time` (Chapter 5, the simplest tool), `prof` (Chapter 6, instrumentation-based profiling), `tprof` (Chapter 7, sampling-based profiling), `kprof` (Chapter 8, kernel profiling), `trace` (Chapter 9, real-time scheduler visualization), `stats` (Chapter 10, system-wide monitoring), and `iostats` (Chapter 11, 9P tracing file server). Chapter 12 concludes.

Chapter 3

Kernel Support

In this chapter, I will summarize the kernel infrastructure that the profiling tools rely on. The full details can be found in KERNEL book [Pad14]; here I present just enough to understand the code in later chapters.

3.1 Ticks, HZ, and times

In Plan 9, the clock interrupt fires at a frequency `HZ` (typically 100 on ARM). On each tick, the kernel calls `accountime()`, which checks the `insyscall` flag of the current process to decide whether to increment `p->time[TUser]` or `p->time[TSys]`. Real time is tracked differently: `p->time[TReal]` stores the global tick count at process creation, so elapsed real time is computed as `CPUS(0)->ticks - p->time[TReal]`.

Consider two processes A and B interleaved on a single CPU. Each tick, the kernel attributes time to whichever process happens to be running:

```
tick:  1    2    3    4    5    6    7    8    9   10
CPU:  [  A-user  ][ A-sys ][ B-user ][ A-user ][ B ]
       +u        +u  +s  +s   +u  +u   +u  +u   +u

After tick 10:
  A: user=4  sys=2  real=10 (created at tick 0)
  B: user=3  sys=0  real=5  (created at tick 5)
```

Process A's user+sys (6) is less than its real time (10) because B ran in between. Process B's real time starts from when it was created, not from tick 0. One Plan 9-specific detail not shown in Figure 2.1 is the accumulation of children's times. When a child exits, its user and system times are added to the parent's `TCUser` and `TCSys` counters. This is how `/bin/time` can report the total CPU time consumed by a command and all its descendants.

3.2 Wait message

The simplest form of profiling—timing a command—requires almost no special kernel support. Each process maintains five time counters (user, system, real, cumulative child user, cumulative child system), incremented by `accountime()` on every clock tick. When a process exits, `pexit()` packages three values (user, system, real) into the `Waitmsg` structure that the parent receives via `wait()`:

```
struct Waitmsg {
    ...
    ulong time[3];  /* user, sys, real (ms) */
    ...
};
```

The /bin/time command (Chapter 5) simply forks, execs the target program, calls wait(), and prints w->time[0], w->time[1], w->time[2]. The kernel does all the accounting work.

3.3 /proc/<pid>/status and /dev/cputime

While Waitmsg only gives timing for *completed* processes, these two files expose the same five counters for *running* processes. /dev/cputime returns the counters of the calling process (user, sys, real, child user, child sys, all in milliseconds). /proc/<pid>/status returns those of any process, which is useful for monitoring long-running programs without waiting for them to exit.

3.4 /proc/<pid>/profile

Tick-based profiling (tprof) uses a periodic timer that fires at a frequency deliberately not aligned with HZ (to avoid aliasing with other periodic events). On each tick, profclock() checks whether the current process is running in user mode and, if so, increments a counter in the profile array attached to the text segment.

The profile array is an array of ulong (4-byte unsigned integers), one entry per 8-byte range of the text segment:

```
profile[]:  ulong (4 bytes each)
+----------+----------+----------+----------+-----+
| total ms | bucket 0 | bucket 1 | bucket 2 | ... |
+----------+----------+----------+----------+-----+
index: 0        1          2          3

text segment:
+--------+--------+--------+--------+--------+-----+
| 8 bytes| 8 bytes| 8 bytes| 8 bytes| 8 bytes| ... |
+--------+--------+--------+--------+--------+-----+
base   base+8  base+16  base+24  base+32
         ^        ^        ^        ^
         |        |        |        |
      bucket 0 bucket 1 bucket 2 bucket 3
```

The array covers the entire text segment with a resolution of 8 bytes per bucket (LRESPROF = 3, so pc >> 3 gives the bucket index). The first element profile[0] holds the total execution time in milliseconds; subsequent elements hold per-bucket times. The profile data is attached to the *segment*, not the process, so multi-threaded programs sharing the same text accumulate a single combined profile. To activate profiling, the user writes "profile" to /proc/<pid>/ctl, which allocates the profile array. The /bin/tprof tool then reads /proc/<pid>/profile and correlates the buckets with symbol information from the binary.

3.5 /proc/trace

The /proc/trace file provides a stream of scheduler events for processes that have tracing enabled. Each event is a Traceevent structure containing a process ID, an event type (SReady, SRun, SDead, SSleep, or SUser), and a timestamp. The events are stored in a kernel-side circular buffer. Hooks in ready(), runproc(), sleep(), and pexit() generate events whenever a process changes state. To enable tracing for a process, the user writes "trace" to /proc/<pid>/ctl. The /bin/trace tool reads the event stream and draws the graphical timeline. The event types are defined in /sys/include/trace.h, which is shared between the kernel and /bin/trace.

3.6 TOS

Function-level profiling (`prof`) needs to measure the time spent in each function. The naive approach would be to call a time function (like `nsec()` or `times()`) at each function entry and exit to record timestamps. But these functions are system calls—they trap into the kernel, which is orders of magnitude slower than a regular function call. A program with thousands of small functions would spend more time measuring than computing.

The kernel solves this problem by mapping the `Tos` (Top of Stack) structure solves this by mapping the `Tos` (Top of Stack) structure at the top of every user stack, giving user code direct read access to timing information without entering the kernel:

```
struct Tos {
    struct { ... } prof;  /* profiling linked list */
    uvlong cyclefreq;     /* cycle counter frequency */
    vlong  kcycles;       /* cycles in kernel */
    vlong  pcycles;       /* cycles in process */
    ulong  clock;         /* millisecond clock */
    ulong  pid;
    ...
};
```

The `Tos` structure sits at the very top of the user stack, just below `USTKTOP`. The kernel can write to it (it is mapped in the kernel's address space too), and user code can read it directly without a syscall:

```
USTKTOP
+----------------------------+
|  Tos                       |  <-- tos = USTKTOP - sizeof(Tos)
|    prof.pp, prof.next ...  |  \
|    cyclefreq               |   |  read by _profin/_profout
|    kcycles  <-- updated    |   |  in user space
|    pcycles     on every    |   |
|    clock       kernel      |  /
|    pid         entry/exit  |
+----------------------------+
|  exec arguments            |
+----------------------------+
|                            |
|  user stack frames         |
|  (grows downward)          |
|          |                 |
|          v                 |
+----------------------------+
```

The kernel updates `tos->kcycles` and `tos->pcycles` on every kernel entry and exit (in `syscall()`, `trap()`, and `arch__kexit()`), and `tos->clock` on every profiling tick. The profiling runtime library (`lib_core/libc/profile` reads these fields to compute per-function times entirely in user space. The `prof` field in `Tos` holds the linked list of profiling records that the linker's instrumentation code (`_profin`/`_profout`) maintains. When the program exits, the runtime writes this data to prof.<pid>.

Chapter 4

Linker Support

In this chapter, I will summarize how the linker `5l` instruments programs for profiling. The full details can be found in LINKER book [Pad15]; here I present just enough to understand how `prof` gets its data.

Surprisingly, in Plan 9 profiling is enabled by a flag of the *linker* (`5l -p`), not the compiler. This is possible because the linker operates on the instruction stream and can insert extra instructions after `TEXT` (function entry) and before `RET` (function return). The linker supports two profiling strategies, selected by `5l -p` (the default) or `5l -p -1`.

4.1 5l -p

The default strategy (`5l -p`) measures the *time spent in each function.* The linker inserts a `BL _profin` call right after every `TEXT` instruction and a `BL _profout` call right before every `RET`:

```
TEXT foo(SB), $0        TEXT foo(SB), $0
                   ->   BL _profin(SB)
...                     ...
                        BL _profout(SB)
RET                ->   RET
```

The `_profin()` and `_profout()` functions are implemented in `lib_core/libc/profile.c`. They use the `Tos` structure (described in the previous chapter) to read the current time without making a system call, and maintain a linked list of profiling records that track the call tree.

For example, if `main` calls `foo` which calls `bar`, the profiling records form a tree mirroring the call stack:

```
tos->prof.pp --> [main: 12ms, 1 call]
                     |
                     +---> [foo: 8ms, 3 calls]
                               |
                               +---> [bar: 2ms, 5 calls]
```

Each `_profin` pushes a new record (or increments an existing one), and each `_profout` pops back to the caller, accumulating the elapsed time.

The `-p` flag also changes the program's entry point from `_main` to `_mainp`, which calls `_profmain()` to initialize the profiling data structures before entering `main()`. When the program exits, the runtime writes the accumulated data to `prof.<pid>`, which `/bin/prof` then reads.

4.2 5l -p1

The simpler strategy (`5l -p -1`) only counts *how many times each function is called*, without measuring time. The linker allocates a global array `__mcount` where each function gets two 4-byte entries: one for the function's address and one for the call count. After each `TEXT` instruction, the linker inserts three instructions that load the counter, increment it, and store it back:

```
TEXT foo(SB), $0        TEXT foo(SB), $0
                 ->     MOVW __mcount+8(SB), R11
                        ADD  $1, R11
...                     MOVW R11, __mcount+8(SB)
                        ...
RET              ->     RET
```

This strategy has lower overhead than `5l -p` (no function call, just a load-increment-store) but provides less information: you learn which functions are hot by call count, but not how much time each call takes.

For a program with functions `main`, `foo`, and `bar`, the `__mcount` array in memory looks like:

```
__mcount:
offset  contents
+0      5            (total size in words: 1 + 2*nfuncs)
+4      &main        (address of main)
+8      0            (call count for main, incremented at runtime)
+12     &foo         (address of foo)
+16     0            (call count for foo)
+20     &bar         (address of bar)
+24     0            (call count for bar)
```

4.3 The symbol table

Both `prof` and `tprof` ultimately need to map a program counter (PC) value to a function name. This is done by reading the symbol table embedded in the executable. The linker produces this table: for each function, it records the function's name and the starting address of its code. Given a PC, the profiler finds the symbol whose address is closest below the PC—that is the function the PC belongs to. In Plan 9, the tool library `libmach` provides functions to read symbol tables from Plan 9 executables. Both `prof` and `tprof` use these to correlate addresses (from the profiling data or the sampling histogram) with human-readable function names. Without the symbol table, profiling data would just be a list of raw addresses.

A Plan 9 executable on disk has the following layout (see also LINKER book [Pad15]):

```
+-------------------+
|  header           |  magic, sizes, entry point
+-------------------+
|  text segment     |  machine code
+-------------------+
|  data segment     |  initialized globals
+-------------------+
|  symbol table     |  name, type, address for each symbol
+-------------------+
|  PC/line table    |  maps PC ranges to source lines
+-------------------+
```

```
symbol table entry:
+--------+------+---------+
|  value  | type |  name   |
| (addr)  |      | (string)|
+--------+------+---------+
```

The profiler reads the symbol table, sorts the entries by address, and for each PC in the profiling data, performs a binary search to find the enclosing function.

Chapter 5

Timing a Command: /bin/time

`time` is the simplest profiling tool—barely 60 lines of C. As explained in Chapter 3, the kernel already delivers timing data in `Waitmsg`, so all `time` has to do is fork, exec the target command, call `wait()`, and print the three numbers.

As a reminder from Section 2.6:

```
% time hello
hello world
0.00u 0.00s 0.01r     hello
```

The three numbers are user time, system time, and real time in seconds. Even this trivial program takes 10 milliseconds of real time (process creation overhead), while its actual user and system time round to zero.

5.1 main()

The implementation follows the classic UNIX`fork`/`exec`/`wait` pattern: fork a child, exec the target command in the child, and in the parent call `wait()` to collect the `Waitmsg` with the timing data. The kernel does all the accounting; `time` just formats and prints the three numbers.

⟨*function* `main` *(time.c)* 22⟩≡ (70a)

```
void
main(int argc, char *argv[])
{
    Waitmsg *w;
    long l;
    ⟨main() (time.c) other locals 23d⟩

    ⟨main() (time.c) usage if no args 10a⟩

    switch(fork()){
    case 0:
        // in child
        exec(argv[1], &argv[1]);

        ⟨main() (time.c) in child after exec 23c⟩
        error(argv[1]);
    case -1:
        error("fork");
    }
    // else, in parent

    notify(notifyf);
```

```
    loop:
    w = wait();
    ⟨main() (time.c) if w == nil 24b⟩
    l = w->time[0];
    add("%ld.%.2ldu", l/1000, (l%1000)/10);
    l = w->time[1];
    add("%ld.%.2lds", l/1000, (l%1000)/10);
    l = w->time[2];
    add("%ld.%.2ldr", l/1000, (l%1000)/10);
    add("\t");

    ⟨main() (time.c) display also argv 24e⟩
    ⟨main() (time.c) display also wait message and status 25⟩
    fprint(STDERR, "%s\n", output);
    exits(w->msg);
}
```

Uses `add()` 23b, `notifyf()` 24c, and `output` 23a.

The three time values in `Waitmsg` are in milliseconds. The formatting divides by 1000 for the integer part and extracts hundredths from the remainder, appending `u` (user), `s` (system), or `r` (real). For example, if `w->time[0]` is 15050 (milliseconds), the output is `15.05u` (15050/1000 = 15, (15050%1000)/10 = 5).

```
⟨global output (time.c) 23a⟩≡                                        (70a)
  char output[4096];
```

```
⟨function add (time.c) 23b⟩≡                                         (70a)
  void
  add(char *a, ...)
  {
      static bool beenhere=false;
      va_list arg;

      if(beenhere)
          strcat(output, " ");
      va_start(arg, a);
      vseprint(output+strlen(output), output+sizeof(output), a, arg);
      va_end(arg);
      beenhere = true;
  }
```

Uses `output` 23a.

Unlike UNIX's `execvp`, Plan 9's `exec()` does not search a path—it requires an exact file name. In Plan 9, path searching is the shell's job (using the `path` variable in `rc`). Here `time` does a minimal lookup by trying `/bin/` if the bare name fails. This is sufficient in practice because Plan 9 uses union mounts to bind all binary directories into `/bin/`, unlike UNIX which scatters binaries across `/usr/bin`, `/usr/local/bin`, etc.

```
⟨main() (time.c) in child after exec 23c⟩≡                             (22)
  if(argv[1][0] != '/' && strncmp(argv[1], "./", 2) &&
     strncmp(argv[1], "../", 3)){
      sprint(output, "/bin/%s", argv[1]);
      exec(output, &argv[1]);
  }
```

Uses `output` 23a.

5.2 Error management

```
⟨main() (time.c) other locals 23d⟩≡                               (22) 24d▷
  char err[ERRMAX];
```

⟨*function* `error` *(time.c)* 24a⟩≡ (70a)

```
static void
error(char *s)
{

    fprint(STDERR, "time: %s: %r\n", s);
    exits(s);
}
```

If the user presses the interrupt key (e.g., Delete) while the child is running, `wait()` returns `nil` with the error string "interrupted". In that case, `time` simply retries `wait()`—the interrupt was meant for the child, not for `time` itself. The `notifyf` handler registered via `notify()` ensures that interrupts are caught (`NCONT`) rather than killing `time`.

⟨`main()` *(time.c) if* `w == nil` 24b⟩≡ (22)

```
if(w == nil){
    errstr(err, sizeof err);
    if(strcmp(err, "interrupted") == 0)
        goto loop;
    error("wait");
}
```

⟨*function* `notifyf` *(time.c)* 24c⟩≡ (70a)

```
void
notifyf(void *a, char *s)
{
    USED(a);
    if(strcmp(s, "interrupt") == 0)
        noted(NCONT);
    noted(NDFLT);
}
```

5.3 Extra display

Besides the timing data, `time` also prints the command line (truncated to 4 arguments) and, if the child exited with an error, the exit status string.

See for instance:

```
% time ls /xxx
ls: /xxx: '/xxx' file does not exit
0.00u 0.01s 0.02r     ls /xxx # status= errors
%
```

⟨`main()` *(time.c) other locals* 24d⟩+≡ (22) ◁23d

```
int i;
char *p;
```

⟨`main()` *(time.c) display also argv* 24e⟩≡ (22)

```
for(i=1; i<argc; i++){
    add("%s", argv[i], 0);
    if(i>4){
        add("...");
        break;
    }
}
```

Uses `add()` 23b.

The exit status in Plan 9 is a string (not a number). By convention it has the form "`program: message`", so `time` strips the program name prefix and shows just the message part after the colon.

⟨`main()` *(time.c) display also wait message and status* 25⟩≡ (22)

```
if(w->msg[0]){
    p = utfrune(w->msg, ':');
    if(p && p[1])
        p++;
    else
        p = w->msg;
    add(" # status=%s", p);
}
```

Uses `add()` 23b.

Chapter 6

Instrumentation-based Profiler: /bin/prof

prof reads the profiling data written by the instrumented program (the prof.<pid> file) and the symbol table from the executable, then presents the results either as a flat profile (the default) or as an indented call graph (-d flag). Recall the example from Section 2.6: the flat profile shows per-function time and call counts, while -d shows which function called which.

6.1 Data structures

The profiling data file contains an array of Data[26a] records written by the runtime library (lib_core/libc/profile.c
Each record represents a function call in the dynamic call tree:

⟨*struct* Data 26a⟩≡ (70c)
```
struct Data
{
    ushort down;
    ushort right;

    ulong pc;
    ulong count;
    ulong time;
};
```

The down and right fields encode a tree using array indices (not pointers): down points to the first callee, right to the next sibling at the same call depth. The sentinel value 0xFFFF means "no child" or "no sibling." The pc field identifies the function, count is the number of times it was called at this point in the tree, and time is the cumulative time in milliseconds (including callees).

⟨*global* data 26b⟩≡ (70c)
```
// ref_own<array<Data> (len = ndata)
Data* data;
```

⟨*global* ndata 26c⟩≡ (70c)
```
// len(data)
long ndata;
```

For the example program from Section 2.6 where main calls foo 100 times and bar once, and foo calls bar, the data array encodes the following tree:

```
data[0]:  pc=main   count=1   time=100  down=1   right=FFFF
data[1]:  pc=foo    count=100 time=80   down=3   right=2
data[2]:  pc=bar    count=1   time=10   down=FFFF right=FFFF
data[3]:  pc=bar    count=100 time=20   down=FFFF right=FFFF
```

```
Tree:     main (100ms, 1 call)
            |
            +--down--> foo (80ms, 100 calls)
            |             |
            |             +--down--> bar (20ms, 100 calls)
            |
            +--right--> bar (10ms, 1 call)
```

At runtime, `_profin()` and `_profout()` maintain a linked list of `Plink` records (stored in `Tos.prof`) that track the current call chain. When the program exits, `_profmain()` calls `_profdump()` which serializes this linked list into the flat `Data` array written to `prof.<pid>`.

The `Acc`[27a] structure is used to accumulate a flat per-function summary from the tree: for each function, it records the function's name, address, total *self* time (excluding callees), and call count.

⟨*struct* `Acc` 27a⟩≡ (70c)
```
struct Acc
{
    char *name;
    ulong pc;
    ulong ms;
    ulong calls;
};
```

⟨*global* `acc` 27b⟩≡ (70c)
```
Acc* acc;
```

The `Pc`[27c] structure is only used by `graph()`[34c] to detect recursion when printing the call graph with `-d`. It forms a linked list of PCs representing the current call chain during the tree walk.

⟨*struct* `Pc` 27c⟩≡ (70c)
```
struct Pc
{
    Pc *next;
    ulong pc;
};
```

6.2 `main()`

The main function has three phases: load the symbol table from the executable via `syms()`[29b], load the profiling data from `prof.out` via `datas()`[29e], then display the results via either `plot()`[30e] (flat profile) or `graph()`[34c] (call graph with `-d`).

⟨*function main (misc/prof.c)* 27d⟩≡ (70c)
```
void
main(int argc, char *argv[])
{
    char *s;

    ⟨main() (prof.c) adjust tabstop 35c⟩

    ARGBEGIN{
    ⟨main() (prof.c) command-line parsing 28c⟩
    default:
        ⟨main() (prof.c) print usage and exit 10b⟩
    }ARGEND

    Binit(&bout, STDOUT, OWRITE);
```

```
    if(argc > 0)
        syms(argv[0]);
    else
        syms(defaout());

    if(argc > 1)
        datas(argv[1]);
    else
        datas("prof.out");

    if(ndata){
        ⟨main() (prof.c) if dflag 34b⟩
        else
            plot();
    }
    exits(nil);
}
```

⟨*global* bout 28a⟩≡ (70c)

```
Biobuf bout;
```

⟨*global* verbose 28b⟩≡ (70c)

```
int verbose;
```

⟨main() *(prof.c) command-line parsing* 28c⟩≡ (27d) 34a ▷

```
case 'v':
    verbose = 1;
    break;
```

If no executable is given on the command line, `prof` defaults to the standard output name for the current architecture (e.g., `5.out` for ARM, `8.out` for x86), determined from the `objtype` environment variable.

⟨*function* defaout 28d⟩≡ (70c)

```
char*
defaout(void)
{
    char *p;
    int i;

    p = getenv("objtype");
    if(p)
    for(i=0; trans[i]; i+=2)
        if(strcmp(p, trans[i]) == 0)
            return trans[i+1];
    return trans[1];
}
```

Uses `trans` 28e.

⟨*global* trans 28e⟩≡ (70c)

```
char* trans[] =
{
    "386",  "8.out",
    "68020",  "2.out",
    "alpha",  "7.out",
    "amd64", "6.out",
    "arm",  "5.out",
    "mips",  "v.out",
    "power",  "q.out",
    "sparc",  "k.out",
    "spim",  "0.out",
    0,0
};
```

6.3 Symbol table loading: syms()

syms()[29b] opens the executable, parses its header with crackhdr(), and initializes the symbol table with syminit(). These are libmach library functions. After this call, functions like textsym() and findsym() can be used to look up function names by address.

```
⟨global nsym 29a⟩≡                                                      (70c)
  long nsym;

⟨function syms 29b⟩≡                                                    (70c)
  void
  syms(char *cout)
  {
      Fhdr f;
      fdt fd;

      fd = open(cout, 0);
      ⟨syms() sanity check fd 29c⟩
      if (!crackhdr(fd, &f)) {
          fprint(STDERR, "can't read text file header\n");
          exits("read");
      }
      ⟨syms() sanity check f.type 29d⟩
      if (syminit(fd, &f) < 0) {
          fprint(STDERR, "syminit: %r\n");
          exits("syms");
      }
      close(fd);
  }

⟨syms() sanity check fd 29c⟩≡                                           (29b)
  if(fd < 0){
      perror(cout);
      exits("open");
  }

⟨syms() sanity check f.type 29d⟩≡                                       (29b)
  if (f.type == FNONE) {
      fprint(STDERR, "text file not an a.out\n");
      exits("file type");
  }
```

6.4 Profiling data loading: datas()

datas()[29e] reads the prof.out file, which is simply a flat array of Data[26a] records written by the profiling runtime. The number of records is computed from the file size. Since Plan 9 uses big-endian encoding for the profile data (regardless of the host architecture), each record must be byte-swapped via swapdata()[30a].

```
⟨function datas 29e⟩≡                                                   (70c)
  void
  datas(char *dout)
  {
      fdt fd;
      Dir *d;
      int i;

      fd = open(dout, 0);
      ⟨datas() sanity check fd 30b⟩
```

```
    d = dirfstat(fd);
    ⟨datas() sanity check d 30c⟩
    ndata = d->length/sizeof(data[0]);
    data = malloc(ndata*sizeof(Data));
    ⟨datas() sanity check data 30d⟩
    if(read(fd, data, d->length) != d->length){
        fprint(STDERR, "prof: can't read data file\n");
        exits("data read");
    }
    free(d);
    close(fd);
    for (i = 0; i < ndata; i++)
        swapdata(data+i);
}
```

Uses `data` 26b, `ndata` 26c, and `swapdata()` 30a.

⟨*function* `swapdata` 30a⟩≡ (70c)

```
void
swapdata(Data *dp)
{
    dp->down = beswab(dp->down);
    dp->right = beswab(dp->right);
    dp->pc = beswal(dp->pc);
    dp->count = beswal(dp->count);
    dp->time = beswal(dp->time);
}
```

⟨`datas()` *sanity check* `fd` 30b⟩≡ (29e)

```
if(fd < 0){
    perror(dout);
    exits("open");
}
```

⟨`datas()` *sanity check* `d` 30c⟩≡ (29e)

```
if(d == nil){
    perror(dout);
    exits("stat");
}
```

⟨`datas()` *sanity check* `data` 30d⟩≡ (29e)

```
if(data == 0){
    fprint(STDERR, "prof: can't malloc data\n");
    exits("data malloc");
}
```

Uses `data` 26b.

6.5 Displaying data: `plot()`

`plot()`[30e] produces the flat profile. It first builds the `acc`[27b] array by iterating over all text symbols, then calls `sum()`[31c] to walk the call tree and accumulate per-function self time. Finally, it sorts by time (ascending, so the hottest functions come last) and prints each function with a non-zero call count.

⟨*function* `plot` 30e⟩≡ (70c)

```
void
plot(void)
{
    Symbol s;
```

```
    for (nsym = 0; textsym(&s, nsym); nsym++) {
        acc = realloc(acc, (nsym+1)*sizeof(Acc));
        ⟨plot() sanity check acc 31b⟩
        acc[nsym].name = s.name;
        acc[nsym].pc = s.value;
        acc[nsym].calls = acc[nsym].ms = 0;
    }

    sum(data[0].down);
    qsort(acc, nsym, sizeof(Acc), acmp);

    Bprint(&bout, "  %%     Time     Calls  Name\n");
    if(ms == 0)
        ms = 1;

    while (--nsym >= 0) {
        if(acc[nsym].calls)
            Bprint(&bout, "%4.1f %8.3f %8lud\t%s\n",
                (100.0*acc[nsym].ms)/ms,
                acc[nsym].ms/1000.0,
                acc[nsym].calls,
                acc[nsym].name);
    }
}
```

Uses `acc` 27b, `acmp()` 33b, `bout` 28a, `data` 26b, `ms` 31a, `nsym` 29a, and `sum()` 31c.

For the `foo`/`bar` example, after `sum()` walks the tree, the `acc` array contains self times (total minus children):

```
acc[]:
name    pc        ms    calls
main    0x1000    10    1        (100 - 80 - 10 = 10)
foo     0x1040    60    100      (80 - 20 = 60)
bar     0x1080    30    101      (20 + 10 = 30)

After qsort (ascending by ms) and printing (descending):
  %     Time     Calls  Name
 60.0    0.060     100   foo
 30.0    0.030     101   bar
 10.0    0.010       1   main
```

⟨*global* `ms` 31a⟩≡ (70c)

```
ulong ms;
```

⟨`plot()` *sanity check* `acc` 31b⟩≡ (30e)

```
if(acc == 0){
    fprint(2, "prof: malloc fail\n");
    exits("acc malloc");
}
```

Uses `acc` 27b.

`sum()` recursively walks the call tree encoded in the `data`[26b] array. For each node, it computes the *self time* by subtracting the children's time (`dtime`) from the node's total time. This self time is attributed to the function identified by `symind()`[33a] (a binary search in the sorted `acc` array). The `right` links are followed to accumulate sibling times, returning the total time of the subtree.

⟨*function* `sum` 31c⟩≡ (70c)

```
ulong
sum(ulong i)
```

```
{
    long j, dtime, time;
    int k;
    static int indent;

    ⟨sum() sanity check i 32a⟩

    j = symind(data[i].pc);
    time = data[i].time;
    if(time < 0)
        time += data[0].time;

    ⟨sum() if verbose 32b⟩
    dtime = 0;
    if(data[i].down != 0xFFFF){
        indent++;
        dtime = sum(data[i].down);
        indent--;
    }
    j = symind(data[i].pc);
    if (j >= 0) {
        acc[j].ms += time - dtime;
        ms += time - dtime;
        acc[j].calls += data[i].count;
    }
    if(data[i].right == 0xFFFF)
        return time;
    return time + sum(data[i].right);
}
```

Uses `acc` 27b, `data` 26b, `ms` 31a, `sum()` 31c, and `symind()` 33a.

The traversal of `sum()` for the example tree proceeds as follows:

```
sum(0) -- main: time=100
  |  sum(1) -- foo: time=80      (down from main)
  |     |  sum(3) -- bar: time=20 (down from foo)
  |     |  return 20
  |    dtime=20, self=80-20=60 -> acc[foo]+=60
  |    sum(2) -- bar: time=10    (right from foo)
  |    dtime=0,  self=10-0=10  -> acc[bar]+=10
  |    return 80+10=90
  dtime=90, self=100-90=10     -> acc[main]+=10
```

⟨`sum()` *sanity check* `i` 32a⟩≡ (31c)

```
if(i >= ndata){
    fprint(STDERR, "prof: index out of range %ld [max %ld]\n", i, ndata);
    return 0;
}
```

Uses `ndata` 26c.

⟨`sum()` *if* `verbose` 32b⟩≡ (31c)

```
if (verbose){
    for(k = 0; k < indent; k++)
        print(" ");
    print("%lud: %ld/%lud", i, data[i].time, data[i].count);
    if (j >= 0)
        print(" %s\n", acc[j].name);
    else
```

```
        print(" 0x%lux\n", data[i].pc);
}
```

Uses acc 27b and data 26b.

symind() maps a PC to an index in the acc array using binary search. It finds the entry whose address is the largest one not exceeding the PC—that is the function the PC belongs to.

⟨*function* symind 33a⟩≡ (70c)

```
/*
 * assume acc is ordered by increasing text address.
 */
long
symind(ulong pc)
{
    int top, bot, mid;

    bot = 0;
    top = nsym;
    for (mid = (bot+top)/2; mid < top; mid = (bot+top)/2) {
        if (pc < acc[mid].pc)
            top = mid;
        else
        if (mid != nsym-1 && pc >= acc[mid+1].pc)
            bot = mid;
        else
            return mid;
    }
    return -1;
}
```

Uses acc 27b and nsym 29a.

⟨*function* acmp 33b⟩≡ (70c)

```
int
acmp(void *va, void *vb)
{
    Acc *a, *b;
    ulong ua, ub;

    a = va;
    b = vb;
    ua = a->ms;
    ub = b->ms;

    if(ua > ub)
        return 1;
    if(ua < ub)
        return -1;
    return 0;
}
```

6.6 Call graph, prof -d

With the -d flag, prof displays the dynamic call graph instead of a flat profile. The graph()[34c] function walks the same Data[26a] tree as sum()[31c], but instead of accumulating totals, it prints each node indented by call depth, showing the function name, time, and call count.

⟨*global* dflag 33c⟩≡ (70c)

```
int dflag;
```

⟨main() *(prof.c) command-line parsing* 34a⟩+≡ (27d) ◁28c 35f▷

```
case 'd':
    dflag = 1;
    break;
```

⟨main() *(prof.c) if* dflag 34b⟩≡ (27d)

```
if(dflag)
    graph(0, data[0].down, 0);
```

To handle recursive functions, `graph()` maintains a linked list of `Pc`[27c] nodes representing the current call chain. Before descending into a child, it checks whether the same PC already appears in the chain. If so, it prints "`...`" to indicate the recursion and stops. The `-r` flag disables this compression, printing the full (potentially very long) recursive call graph.

⟨*function* graph 34c⟩≡ (70c)

```
void
graph(int ind, ulong i, Pc *pc)
{
    long time, count, prgm;
    Pc lpc;

    ⟨graph() sanity check i 34d⟩
    count = data[i].count;
    time = data[i].time;
    prgm = data[i].pc;
    if(time < 0)
        time += data[0].time;
    if(data[i].right != 0xFFFF)
        graph(ind, data[i].right, pc);
    indent(ind);
    if(count == 1)
        Bprint(&bout, "%s:%lud\n", name(prgm), time);
    else
        Bprint(&bout, "%s:%lud/%lud\n", name(prgm), time, count);
    if(data[i].down == 0xFFFF)
        return;
    lpc.next = pc;
    lpc.pc = prgm;

    if(!rflag){
        while(pc){
            if(pc->pc == prgm){
                indent(ind+1);
                Bprint(&bout, "...\n");
                return;
            }
            pc = pc->next;
        }
    }

    graph(ind+1, data[i].down, &lpc);
}
```

Uses `bout` 28a, `data` 26b, `indent()` 35d, `name()` 35a, and `rflag` 35e.

⟨graph() *sanity check* i 34d⟩≡ (34c)

```
if(i >= ndata){
    fprint(STDERR, "prof: index out of range %ld [max %ld]\n", i, ndata);
    return;
}
```

Uses `ndata` 26c.

⟨*function* `name` 35a⟩≡ (70c)
```
char*
name(ulong pc)
{
    Symbol s;
    static char buf[16];

    if (findsym(pc, CTEXT, &s))
        return(s.name);
    snprint(buf, sizeof(buf), "#%lux", pc);
    return buf;
}
```

⟨*global* `tabstop` 35b⟩≡ (70c)
```
int tabstop = 4;
```
Uses `tabstop` 35b.

⟨`main()` *(prof.c) adjust* `tabstop` 35c⟩≡ (27d)
```
s = getenv("tabstop");
if(s!=nil && strtol(s,0,0)>0)
    tabstop = strtol(s,0,0);
```

⟨*function* `indent` 35d⟩≡ (70c)
```
void
indent(int ind)
{
    int j;

    j = 2*ind;
    while(j >= tabstop){
        Bwrite(&bout, ".\t", 2);
        j -= tabstop;
    }
    if(j)
        Bwrite(&bout, ".                            ", j);
}
```
Uses `bout` 28a and `tabstop` 35b.

6.6.1 `prof -d -r`

⟨*global* `rflag` 35e⟩≡ (70c)
```
int rflag;
```

⟨`main()` *(prof.c) command-line parsing* 35f⟩+≡ (27d) ◁34a
```
case 'r':
    rflag = 1;
    break;
```

Chapter 7

Sampling-based Profiler: /bin/tprof

While prof relies on linker instrumentation and reads a data file after the program exits, tprof takes a different approach: it reads a live PC histogram from the kernel while the program is still running. The implementation is even simpler than prof—just read the profile data, correlate with the symbol table, sort, and print.

7.1 Example: longrun.c

Since tprof samples a running process, we need a program that runs long enough to collect meaningful data. Here is a simple example with two functions that do different amounts of work, interleaved with sleep calls to keep the program running long enough:

⟨longrun.c 36⟩≡

```
#include <u.h>
#include <libc.h>

long compute1(void) {
    long i, sum = 0;
    for(i = 0; i < 20000000; i++)
        sum += i;
    return sum;
}
long compute2(void) {
    long i, sum = 0;
    for(i = 0; i < 10000000; i++)
        sum += i;
    return sum;
}
void main(void) {
    int i;
    for(i = 0; i < 10000; i++){
        compute1();
        compute2();
        sleep(100);  /* 100 ms */
    }
    exits(nil);
}
```

To profile this program with tprof:

```
% 8c longrun.c && 8l -o longrun longrun.8
% longrun &
% echo profile > /proc/$apid/ctl
% # ... wait a few seconds ...
```

```
% tprof $apid
total: 80
TEXT 00001000
    ms      %   sym
    60   75.0   compute1
    20   25.2   compute2
% ...
% tprof $apid
total: 3650
TEXT 00001000
    ms      %   sym
    2790   76.4   compute1
    860    23.5   compute2
```

The profile shows `compute1` taking roughly three times as much time as `compute2`—more than the expected 2:1 ratio of loop counts. The `sleep` calls between iterations may affect the ratio: `compute1` runs right after `sleep` returns (cold cache), while `compute2` benefits from the cache warmed by `compute1`. Note that `sleep` does not appear at all—because `sleep` blocks the process, the kernel does not sample its PC during that time. Only code that is actually running on the CPU gets sampled.

`tprof` is the sampling-based counterpart to `prof`. Instead of reading instrumentation data from a `prof.<pid>` file, it reads the PC histogram from `/proc/<pid>/profile`, which the kernel populates via `profclock()` (see Chapter 3). The tool must be run while the target process is still alive, since the profile data lives in the process's text segment.

7.2 `main()`

Like `prof`, `main()` has three phases: read the symbol table (from `/proc/<pid>/text` or a given binary), read the profile data (from `/proc/<pid>/profile`), and display the results. The key difference from `prof` is that the profile data is a flat array of counters (one per 8-byte bucket), not a call tree.

⟨*function main (misc/tprof.c)* 37⟩≡ (74)

```
void
main(int argc, char *argv[])
{
    // for symbol table
    fdt fd; // /proc/<pid>/text and then /proc/<pid>/profile
    Fhdr f;
    // for profile data
    char filebuf[128], *file; // /proc/<pid>/profile
    Dir *d;
    ulong *data; // profile content
    // for output
    Biobuf outbuf;
    ⟨main() (tprof.c) other locals 38a⟩

    ⟨main() (tprof.c) sanity check argc and print usage 11a⟩

    /*
     * Read symbol table
     */
    ⟨main() (tprof.c) read symbol table 38g⟩

    /*
     * Read timing data
```

```
     */
    ⟨main() (tprof.c) read timing data 39a⟩

    ⟨main() (tprof.c) displaying profile 39d⟩

    exits(nil);
}
```

⟨main() *(tprof.c) other locals* 38a⟩≡ (37)

```
long i, j, k, n;
char *name;
ulong tbase, sum;
long delta;
Symbol s;
struct COUNTER *cp;
```

Uses COUNTER 39c.

⟨main() *(tprof.c) sanity check* fd 38b⟩≡ (39a 38g)

```
if(fd < 0)
    error(true, file);
```

⟨main() *(tprof.c) sanity check* f.type 38c⟩≡ (38g)

```
if (f.type == FNONE)
    error(false, "text file not an a.out");
```

⟨main() *(tprof.c) sanity check* d 38d⟩≡ (39a)

```
if(d == nil)
    error(true, "stat");
```

⟨main() *(tprof.c) sanity check* n 38e⟩≡ (39a)

```
if(n < 2)
    error(false, "data file too short");
```

⟨main() *(tprof.c) sanity check* data 38f⟩≡ (39a)

```
if(data == nil)
    error(true, "malloc");
```

7.3 Reading the symbol table from the binary

The symbol table loading is almost identical to `syms()`[29b] in `prof.c`. The only difference is that `tprof` can read the binary directly from `/proc/<pid>/text` (a convenient Plan 9 feature: each process exports its executable via the `proc` filesystem), so there is no need to know the executable's path on disk.

⟨main() *(tprof.c) read symbol table* 38g⟩≡ (37)

```
if(argc == 2){
    file = filebuf;
    snprint(filebuf, sizeof filebuf, "/proc/%s/text", argv[1]);
}else
    file = argv[2];

fd = open(file, OREAD);
⟨main() (tprof.c) sanity check fd 38b⟩

if (!crackhdr(fd, &f))
    error(true, "read text header");
⟨main() (tprof.c) sanity check f.type 38c⟩

machbytype(f.type);
if (syminit(fd, &f) < 0)
    error(true, "syminit");
close(fd);
```

7.4 Reading the profile data from `/proc/<pid>/profile`

The profile data is a flat array of `ulong` values. As described in Chapter 3, `data[0]` holds the total execution time in milliseconds, and `data[j]` holds the time spent in the 8-byte range starting at `tbase + (j-2)*PCRES` (where `tbase` is the page-aligned start of the text segment). The data must be byte-swapped since the kernel writes in big-endian format.

⟨`main()` *(tprof.c) read timing data* 39a⟩≡ (37)

```
file = smprint("/proc/%s/profile", argv[1]);
fd = open(file, OREAD);
⟨main() (tprof.c) sanity check fd 38b⟩
free(file);

d = dirfstat(fd);
⟨main() (tprof.c) sanity check d 38d⟩

n = d->length/sizeof(data[0]);
⟨main() (tprof.c) sanity check n 38e⟩
data = malloc(d->length);
⟨main() (tprof.c) sanity check data 38f⟩
if(read(fd, data, d->length) < 0)
    error(true, "text read");
close(fd);

for(i=0; i<n; i++)
    data[i] = machdata->swal(data[i]);
```

7.5 Displaying profile

The display logic walks the symbol table and the profile array in parallel. For each function (identified by a pair of consecutive symbol addresses), it sums the profile buckets that fall within that function's address range. Functions with non-zero sample counts are collected into `COUNTER`[39c] records, sorted by time, and printed. The `PCRES` constant (8) matches the kernel's `LRESPROF` bucket size.

⟨*constant PCRES (misc/tprof.c)* 39b⟩≡ (74)

```
#define PCRES 8
```

⟨*struct COUNTER (misc/tprof.c)* 39c⟩≡ (74)

```
struct COUNTER
{
    char  *name;  /* function name */
    long time;  /* ticks spent there */
};
```

⟨`main()` *(tprof.c) displaying profile* 39d⟩≡ (37)

```
delta = data[0]-data[1];
print("total: %ld\n", data[0]);
if(data[0] == 0)
    exits(nil);
// else
if (!textsym(&s, 0))
    error(0, "no text symbols");

tbase = s.value & ~(mach->pgsize-1); /* align down to page */

print("TEXT %.8lux\n", tbase);

/*
```

```
 * Accumulate counts for each function
 */
cp = 0;
k = 0;
for (i = 0, j = (s.value-tbase)/PCRES+2; j < n; i++) {
    name = s.name;  /* save name */
    if (!textsym(&s, i)) /* get next symbol */
        break;
    sum = 0;
    while (j < n && j*PCRES < s.value-tbase)
        sum += data[j++];
    if (sum) {
        cp = realloc(cp, (k+1)*sizeof(struct COUNTER));
        if (cp == 0)
            error(1, "realloc");
        cp[k].name = name;
        cp[k].time = sum;
        k++;
    }
}
if (!k)
    error(0, "no counts");
cp[k].time = 0;   /* "etext" can take no time */

/*
 * Sort by time and print
 */
qsort(cp, k, sizeof(struct COUNTER), compar);
Binit(&outbuf, 1, OWRITE);
Bprint(&outbuf, "    ms      %%   sym\n");
while(--k>=0)
    Bprint(&outbuf, "%6ld\t%3lld.%lld\t%s\n",
            cp[k].time,
            100LL*cp[k].time/delta,
            (1000LL*cp[k].time/delta)%10,
            cp[k].name);
```
Uses COUNTER 39c, PCRES-50 39b, and mach 62e.

⟨*function compar (misc/tprof.c)* 40a⟩≡ (74)
```
int
compar(void *va, void *vb)
{
    struct COUNTER *a, *b;

    a = va;
    b = vb;
    if(a->time < b->time)
        return -1;
    if(a->time == b->time)
        return 0;
    return 1;
}
```
Uses COUNTER 39c.

7.6 Error management

⟨*function error (misc/tprof.c)* 40b⟩≡ (74)
```
static void
```

```
error(bool perr, char *s)
{
    fprint(STDERR, "tprof: %s", s);
    if(perr){
        fprint(STDERR, ": ");
        perror(0);
    }else
        fprint(STDERR, "\n");
    exits(s);
}
```

Chapter 8

Profiling the Kernel: /bin/kprof

`kprof` is essentially the same program as `tprof`, but for the kernel instead of a user process. It reads the profile data from `/dev/kpdata` (the `kprof(3)` device) and the symbol table from the kernel binary (e.g., `/386/9pcdisk` or `/arm/9`). The usage is:

```
% bind -a '#K' /dev
% echo start > /dev/kpctl
% # ... run workload ...
% echo stop > /dev/kpctl
% kprof /arm/9 /dev/kpdata
```

The code is nearly identical to `tprof.c`. The main differences are: (1) the symbol table comes from a file on disk (the kernel binary) rather than `/proc/<pid>/text`, (2) the text base address is `mach->kbase` (the kernel's load address) rather than a page-aligned user text address, and (3) the output distinguishes time spent "in kernel text" from time spent "outside kernel text" (e.g., in user mode or idle).

⟨*function main (misc/kprof.c)* 42⟩≡ (70b)

```
void
main(int argc, char *argv[])
{
    fdt fd;
    Fhdr f;
    Dir *d;
    Biobuf outbuf;
    ulong *data;
    ⟨main() (kprof.c) other locals 43a⟩

    ⟨main() (kprof.c) sanity check argc and print usage 11b⟩

    /*
     * Read symbol table
     */
    ⟨main() (kprof.c) read symbol table 43b⟩

    /*
     * Read timing data
     */
    ⟨main() (kprof.c) read profile data 43c⟩

    ⟨main() (kprof.c) display profile data 43f⟩

    exits(0);
}
```

⟨main() *(kprof.c) other locals* 43a⟩≡ (42)

```
long i, j, k, n;
char *name;
vlong tbase;
ulong sum;
long delta;
Symbol s;
struct COUNTER *cp;
```

Uses COUNTER 39c.

⟨main() *(kprof.c) read symbol table* 43b⟩≡ (42)

```
fd = open(argv[1], OREAD);
if(fd < 0)
    error(1, argv[1]);
if (!crackhdr(fd, &f))
    error(1, "read text header");
if (f.type == FNONE)
    error(0, "text file not an a.out");
if (syminit(fd, &f) < 0)
    error(1, "syminit");
close(fd);
```

⟨main() *(kprof.c) read profile data* 43c⟩≡ (42)

```
fd = open(argv[2], OREAD);
if(fd < 0)
    error(1, argv[2]);
d = dirfstat(fd);
if(d == nil)
    error(1, "stat");
n = d->length/sizeof(data[0]);
if(n < 2)
    error(0, "data file too short");
data = malloc(d->length);
if(data == 0)
    error(1, "malloc");
if(read(fd, data, d->length) < 0)
    error(1, "text read");
close(fd);
for(i=0; i<n; i++)
    data[i] = beswal(data[i]);
```

⟨*constant* PCRES 43d⟩≡ (70b)

```
#define PCRES 8
```

⟨*struct* COUNTER 43e⟩≡ (70b)

```
struct COUNTER
{
    char  *name;  /* function name */
    long time;  /* ticks spent there */
};
```

⟨main() *(kprof.c) display profile data* 43f⟩≡ (42)

```
delta = data[0]-data[1];
print("total: %ld in kernel text: %ld outside kernel text: %ld\n",
    data[0], delta, data[1]);
if(data[0] == 0)
    exits(0);
if (!textsym(&s, 0))
    error(0, "no text symbols");
```

```
tbase = mach->kbase;
if(tbase != s.value & ~0xFFF)
    print("warning: kbase %.8llux != tbase %.8llux\n",
        tbase, s.value&~0xFFF);
print("KTZERO %.8llux PGSIZE %dKb\n", tbase, mach->pgsize/1024);
/*
 * Accumulate counts for each function
 */
cp = 0;
k = 0;
for (i = 0, j = 2; j < n; i++) {
    name = s.name;  /* save name */
    if (!textsym(&s, i)) /* get next symbol */
        break;
    s.value -= tbase;
    s.value /= PCRES;
    sum = 0;
    while (j < n && j < s.value)
        sum += data[j++];
    if (sum) {
        cp = realloc(cp, (k+1)*sizeof(struct COUNTER));
        if (cp == 0)
            error(1, "realloc");
        cp[k].name = name;
        cp[k].time = sum;
        k++;
    }
}
if (!k)
    error(0, "no counts");
cp[k].time = 0;   /* "etext" can take no time */
/*
 * Sort by time and print
 */
qsort(cp, k, sizeof(struct COUNTER), compar);
Binit(&outbuf, 1, OWRITE);
Bprint(&outbuf, "ms   %% sym\n");
while(--k>=0)
    Bprint(&outbuf, "%ld\t%3lld.%lld\t%s\n",
            cp[k].time,
            100LL*cp[k].time/delta,
            (1000LL*cp[k].time/delta)%10,
            cp[k].name);
```

Uses `COUNTER` 39c, `PCRES-65` 43d, and `mach` 62e.

⟨*function error (misc/kprof.c)* 44a⟩≡ (70b)

```
static void
error(int perr, char *s)
{
    fprint(2, "kprof: %s", s);
    if(perr){
        fprint(2, ": ");
        perror(0);
    }else
        fprint(2, "\n");
    exits(s);
}
```

⟨*function* `compar` 44b⟩≡ (70b)

```
static int
```

```
compar(void *va, void *vb)
{
    struct COUNTER *a, *b;

    a = va;
    b = vb;
    if(a->time < b->time)
        return -1;
    if(a->time == b->time)
        return 0;
    return 1;
}
```

Uses COUNTER 39c.

Chapter 9

Show real-time Behavior: /bin/trace

The profilers presented so far (`time`, `prof`, `tprof`, `kprof`) analyze a single program in isolation. `trace` takes a different approach: it visualizes the scheduling behavior of one or more processes in real time, showing when each process runs, sleeps, and is ready on a graphical timeline. This makes it invaluable for debugging latency issues in concurrent and real-time systems. The implementation reads a stream of `Traceevent` records from `/proc/trace` (see Section 3), accumulates them per-process, and draws a scrolling timeline where each process gets a horizontal row. The main data structures are `TEvent` (a `Traceevent` extended with a display duration) and `Task` (the per-process state, including the event history and display color).

9.1 Data structures

⟨*struct* TEvent 46a⟩≡ (73c)
```
struct TEvent {
    Traceevent;
    vlong etime; /* length of block to draw */
};
```

⟨*struct* Task 46b⟩≡ (73c)
```
struct Task {
    int pid;
    char *name;
    int nevents;
    TEvent *events;
    vlong tstart;
    vlong total;
    vlong runtime;
    vlong runmax;
    vlong runthis;
    long runs;
    ulong tevents[Nevent];
};
```

⟨*enum* _anon_ (misc/trace.c) 46c⟩≡ (73c)
```
enum {
    Nevents = 1024,
    Ncolor = 6,
    K = 1024,
};
```

⟨*struct* scale 46d⟩≡ (73c)
```
struct scale {
    vlong scale;
    vlong bigtics;
```

```
    vlong littletics;
    int sleep;
};
```

⟨*global* scales 47a⟩≡ (73c)

```
struct scale scales[] = {
    { US(500), US(100), US(50),    0},
    { US(1000), US(500), US(100),   0},
    { US(2000), US(1000), US(200),   0},
    { US(5000), US(1000), US(500),   0},
    { MS(10),  MS(5),  MS(1),   20},
    { MS(20),  MS(10),  MS(2),   20},
    { MS(50),  MS(10),  MS(5),   20},
    { MS(100), MS(50),  MS(10),   20}, /* starting scaleno */
    { MS(200), MS(100), MS(20),   20},
    { MS(500), MS(100), MS(50),   50},
    { MS(1000), MS(500), MS(100), 100},
    { MS(2000), MS(1000), MS(200), 100},
    { MS(5000), MS(1000), MS(500), 100},
    { S(10),  S(50),  S(1),  100},
    { S(20),  S(10),  S(2),  100},
    { S(50),  S(10),  S(5),  100},
    { S(100),  S(50),  S(10),  100},
    { S(200),  S(100),  S(20),  100},
    { S(500),  S(100),  S(50),  100},
    { S(1000), S(500),  S(100),  100},
};
```

Uses `MS-54` 72c, `S-55` 72d, `US-53` 72b, and `scale` 46d 63b.

9.2 main()

⟨*global verbose (misc/trace.c)* 47b⟩≡ (73c)

```
static int verbose;
```

⟨*global* tasks 47c⟩≡ (73c)

```
Task *tasks;
```

⟨*global* cols 47d⟩≡ (73c)

```
static Image *cols[Ncolor][4];
```

Uses `Ncolor-61` 46c.

⟨*global* profdev 47e⟩≡ (73c)

```
char*profdev = "/proc/trace";
```

Uses `profdev` 47e.

⟨*function* usage 47f⟩≡ (73c)

```
static void
usage(void)
{
    fprint(2, "Usage: %s [-d profdev] [-w] [-v] [-t triggerproc] [processes]\n", argv0);
    exits(nil);
}
```

⟨*function* threadmain 48a⟩≡ (73c)
```
void
threadmain(int argc, char **argv)
{
    int fd, i;
    char fname[80];

    fmtinstall('t', timeconv);
    ARGBEGIN {
    case 'd':
        profdev = EARGF(usage());
        break;
    case 'v':
        verbose = 1;
        break;
    case 'w':
        newwin++;
        break;
    case 't':
        triggerproc = (int)strtol(EARGF(usage()), nil, 0);
        break;
    default:
        usage();
    }
    ARGEND;

    fname[sizeof fname - 1] = 0;
    for(i = 0; i < argc; i++){
        snprint(fname, sizeof fname - 2, "/proc/%s/ctl",
                    argv[i]);
        if((fd = open(fname, OWRITE)) < 0){
            fprint(2, "%s: cannot open %s: %r\n",
                        argv[0], fname);
            continue;
        }

        if(fprint(fd, "trace 1") < 0)
            fprint(2, "%s: cannot enable tracing on %s: %r\n",
                        argv[0], fname);
        close(fd);
    }

    drawtrace();
}
```
Uses drawtrace() 55, newwin 72h, profdev 47e, timeconv() 58, and triggerproc 73c.

⟨*function* mkcol 48b⟩≡ (73c)
```
static void
mkcol(int i, int c0, int c1, int c2)
{
    cols[i][0] = allocimagemix(display, c0, DWhite);
    cols[i][1] = allocimage(display, Rect(0,0,1,1), view->chan, 1, c1);
    cols[i][2] = allocimage(display, Rect(0,0,1,1), view->chan, 1, c2);
    cols[i][3] = allocimage(display, Rect(0,0,1,1), view->chan, 1, c0);
}
```

⟨*function* colinit 48c⟩≡ (73c)
```
static void
colinit(void)
{
```

```
    mediumfont = openfont(display, "/lib/font/bit/lucidasans/unicode.10.font");
    if(mediumfont == nil)
        mediumfont = font;
    tinyfont = openfont(display, "/lib/font/bit/lucidasans/unicode.7.font");
    if(tinyfont == nil)
        tinyfont = font;
    topmargin = mediumfont->height+2;
    bottommargin = tinyfont->height+2;

    /* Peach */
    mkcol(0, 0xFFAAAAFF, 0xFFAAAAFF, 0xBB5D5DFF);
    /* Aqua */
    mkcol(1, DPalebluegreen, DPalegreygreen, DPurpleblue);
    /* Yellow */
    mkcol(2, DPaleyellow, DDarkyellow, DYellowgreen);
    /* Green */
    mkcol(3, DPalegreen, DMedgreen, DDarkgreen);
    /* Blue */
    mkcol(4, 0x00AAFFFF, 0x00AAFFFF, 0x0088CCFF);
    /* Grey */
    cols[5][0] = allocimage(display, Rect(0,0,1,1), view->chan, 1, 0xEEEEEEFF);
    cols[5][1] = allocimage(display, Rect(0,0,1,1), view->chan, 1, 0xCCCCCCFF);
    cols[5][2] = allocimage(display, Rect(0,0,1,1), view->chan, 1, 0x888888FF);
    cols[5][3] = allocimage(display, Rect(0,0,1,1), view->chan, 1, 0xAAAAAAFF);
    grey = cols[5][2];
    red = allocimage(display, Rect(0,0,1,1), view->chan, 1, 0xFF0000FF);
    green = allocimage(display, Rect(0,0,1,1), view->chan, 1, 0x00FF00FF);
    blue = allocimage(display, Rect(0,0,1,1), view->chan, 1, 0x0000FFFF);
    bg = display->white;
    fg = display->black;
}
```

Uses `bg` 73c, `blue` 73c, `bottommargin` 72l, `fg` 73c, `green` 73c, `grey` 73c, `red` 73c, `tinyfont` 73c, and `topmargin` 72k.

⟨*function* `time2x` 49a⟩≡ (73c)

```
#define time2x(t) ((int)(((t) - oldestts) / ppp))
```

⟨*function* `redraw` 49b⟩≡ (73c)

```
static void
redraw(int scaleno)
{
    int n, i, j, x;
    char buf[256];
    Point p, q;
    Rectangle r, rtime;
    Task *t;
    vlong ts, oldestts, newestts, period, ppp, scale, s, ss;

    scale = scales[scaleno].scale;
    period = scale + scales[scaleno].littletics;
    ppp = period / Width; // period per pixel.

    /* Round 'now' to a nice number */
    newestts = now - (now % scales[scaleno].bigtics) +
            (scales[scaleno].littletics>>1);

    oldestts = newestts - period;

//print("newestts %t, period %t, %d-%d\n", newestts, period, time2x(oldestts), time2x(newestts));
    if (prevts < oldestts){
        oldestts = newestts - period;
```

```
    prevts = oldestts;
    draw(view, view->r, bg, nil, ZP);
}else{
    /* just white out time */
    rtime = view->r;
    rtime.min.x = rtime.max.x - stringwidth(mediumfont, "00000000000.000s");
    rtime.max.y = rtime.min.y + mediumfont->height;
    draw(view, rtime, bg, nil, ZP);
}
p = view->r.min;
for (n = 0; n != ntasks; n++) {
    t = &tasks[n];
    /* p is upper left corner for this task */
    rtime = Rpt(p, addpt(p, Pt(500, mediumfont->height)));
    draw(view, rtime, bg, nil, ZP);
    snprint(buf, sizeof(buf), "%d %s", t->pid, t->name);
    q = string(view, p, fg, ZP, mediumfont, buf);
    s = now - t->tstart;
    if(t->tevents[SRelease])
        snprint(buf, sizeof(buf), " per %t | avg: %t max: %t",
            (vlong)(s/t->tevents[SRelease]),
            (vlong)(t->runtime/t->tevents[SRelease]),
            t->runmax);
    else if((s /=1000000000LL) != 0)
        snprint(buf, sizeof(buf), " per 1s | avg: %t total: %t",
            t->total/s,
            t->total);
    else
        snprint(buf, sizeof(buf), " total: %t", t->total);
    string(view, q, fg, ZP, tinyfont, buf);
    p.y += Height;
}
x = time2x(prevts);

p = view->r.min;
for (n = 0; n != ntasks; n++) {
    t = &tasks[n];

    /* p is upper left corner for this task */

    /* Move part already drawn */
    r = Rect(p.x, p.y + topmargin, p.x + x, p.y+Height);
    draw(view, r, view, nil, Pt(p.x + Width - x, p.y + topmargin));

    r.max.x = view->r.max.x;
    r.min.x += x;
    draw(view, r, bg, nil, ZP);

    line(view, addpt(p, Pt(x, Height - lineht)), Pt(view->r.max.x, p.y + Height - lineht),
        Endsquare, Endsquare, 0, cols[n % Ncolor][1], ZP);

    for (i = 0; i < t->nevents-1; i++)
        if (prevts < t->events[i + 1].time)
            break;

    if (i > 0) {
        memmove(t->events, t->events + i, (t->nevents - i) * sizeof(TEvent));
        t->nevents -= i;
    }
```

```
for (i = 0; i != t->nevents; i++) {
    TEvent *e = &t->events[i], *_e;
    int sx, ex;

    switch (e->etype & 0xffff) {
    case SAdmit:
        if (e->time > prevts && e->time <= newestts) {
            sx = time2x(e->time);
            line(view, addpt(p, Pt(sx, topmargin)),
                addpt(p, Pt(sx, Height - bottommargin)),
                Endarrow, Endsquare, 1, green, ZP);
        }
        break;
    case SExpel:
        if (e->time > prevts && e->time <= newestts) {
            sx = time2x(e->time);
            line(view, addpt(p, Pt(sx, topmargin)),
                addpt(p, Pt(sx, Height - bottommargin)),
                Endsquare, Endarrow, 1, red, ZP);
        }
        break;
    case SRelease:
        if (e->time > prevts && e->time <= newestts) {
            sx = time2x(e->time);
            line(view, addpt(p, Pt(sx, topmargin)),
                addpt(p, Pt(sx, Height - bottommargin)),
                Endarrow, Endsquare, 1, fg, ZP);
        }
        break;
    case SDeadline:
        if (e->time > prevts && e->time <= newestts) {
            sx = time2x(e->time);
            line(view, addpt(p, Pt(sx, topmargin)),
                addpt(p, Pt(sx, Height - bottommargin)),
                Endsquare, Endarrow, 1, fg, ZP);
        }
        break;

    case SYield:
    case SUser:
        if (e->time > prevts && e->time <= newestts) {
            sx = time2x(e->time);
            line(view, addpt(p, Pt(sx, topmargin)),
                addpt(p, Pt(sx, Height - bottommargin)),
                Endsquare, Endarrow, 0,
                (e->etype == SYield)? green: blue, ZP);
        }
        break;
    case SSlice:
        if (e->time > prevts && e->time <= newestts) {
            sx = time2x(e->time);
            line(view, addpt(p, Pt(sx, topmargin)),
                addpt(p, Pt(sx, Height - bottommargin)),
                Endsquare, Endarrow, 0, red, ZP);
        }
        break;

    case SRun:
        sx = time2x(e->time);
```

```
			ex = time2x(e->etime);
			if(ex == sx)
				ex++;

			r = Rect(sx, topmargin + 8, ex, Height - lineht);
			r = rectaddpt(r, p);

			draw(view, r, cols[n % Ncolor][e->etype==SRun?1:3], nil, ZP);

			if(t->pid == triggerproc && ex < Width)
				paused ^= 1;

			for(j = 0; j < t->nevents; j++){
				_e = &t->events[j];
				switch(_e->etype & 0xffff){
				case SInts:
					if (_e->time > prevts && _e->time <= newestts){
						sx = time2x(_e->time);
						line(view, addpt(p, Pt(sx, topmargin)),
									addpt(p, Pt(sx, Height / 2 - bottommargin)),
									Endsquare, Endsquare, 0,
									green, ZP);
					}
					break;
				case SInte:
					if (_e->time > prevts && _e->time <= newestts) {
						sx = time2x(_e->time);
						line(view, addpt(p, Pt(sx, Height / 2 - bottommargin)),
									addpt(p, Pt(sx, Height - bottommargin)),
									Endsquare, Endsquare, 0,
									blue, ZP);
					}
					break;
				}
			}
			break;
		}
	}
	p.y += Height;
}

ts = prevts + scales[scaleno].littletics - (prevts % scales[scaleno].littletics);
x = time2x(ts);

while(x < Width){
	p = view->r.min;
	for(n = 0; n < ntasks; n++){
		int height, width;

		/* p is upper left corner for this task */
		if ((ts % scales[scaleno].scale) == 0){
			height = 10 * Height;
			width = 1;
		}else if ((ts % scales[scaleno].bigtics) == 0){
			height = 12 * Height;
			width = 0;
		}else{
			height = 13 * Height;
			width = 0;
		}
```

```
            height >>= 4;

            line(view, addpt(p, Pt(x, height)), addpt(p, Pt(x, Height - lineht)),
                Endsquare, Endsquare, width, cols[n % Ncolor][2], ZP);

            p.y += Height;
        }
        ts += scales[scaleno].littletics;
        x = time2x(ts);
    }

    rtime = view->r;
    rtime.min.y = rtime.max.y - tinyfont->height + 2;
    draw(view, rtime, bg, nil, ZP);
    ts = oldestts + scales[scaleno].bigtics - (oldestts % scales[scaleno].bigtics);
    x = time2x(ts);
    ss = 0;
    while(x < Width){
        snprint(buf, sizeof(buf), "%t", ss);
        string(view, addpt(p, Pt(x - stringwidth(tinyfont, buf)/2, - tinyfont->height - 1)),
            fg, ZP, tinyfont, buf);
        ts += scales[scaleno].bigtics;
        ss += scales[scaleno].bigtics;
        x = time2x(ts);
    }

    snprint(buf, sizeof(buf), "%t", now);
    string(view, Pt(view->r.max.x - stringwidth(mediumfont, buf), view->r.min.y),
        fg, ZP, mediumfont, buf);

    flushimage(display, 1);
    prevts = newestts;
}
```

Uses Height 72j, Ncolor-61 46c, Width 72i, bg 73c, blue 73c, bottommargin 72l, fg 73c, green 73c, lineht 72m, now 73c, ntasks 73c, p 103a, paused 73c, prevts 73c, red 73c, scale 46d 63b, scales 47a, tasks 47c, time2x-63 49a, tinyfont 73c, topmargin 72k, and triggerproc 73c.

⟨*function* newtask 53⟩≡ (73c)

```
Task*
newtask(ulong pid)
{
    Task *t;
    char buf[64], *p;
    int fd,n;

    tasks = realloc(tasks, (ntasks + 1) * sizeof(Task));
    assert(tasks);

    t = &tasks[ntasks++];
    memset(t, 0, sizeof(Task));
    t->events = nil;
    snprint(buf, sizeof buf, "/proc/%ld/status", pid);
    t->name = nil;
    fd = open(buf, OREAD);
    if (fd >= 0){
        n = read(fd, buf, sizeof buf);
        if(n > 0){
            p = buf + sizeof buf - 1;
            *p = 0;
            p = strchr(buf, ' ');
```

```
            if (p) *p = 0;
            t->name = strdup(buf);
        }else
            print("%s: %r\n", buf);
        close(fd);
    }else
        print("%s: %r\n", buf);
    t->pid = pid;
    prevts = 0;
    if (newwin){
        fprint(wctlfd, "resize -dx %d -dy %d\n",
            Width + 20, (ntasks * Height) + 5);
    }else
        Height = ntasks ? Dy(view->r)/ntasks : Dy(view->r);
    return t;
}
```

Uses `Height` 72j, `Width` 72i, `newwin` 72h, `ntasks` 73c, `p` 103a, `prevts` 73c, `tasks` 47c, and `wctlfd` 72n.

⟨*function* `doevent` 54⟩≡ (73c)

```
void
doevent(Task *t, Traceevent *ep)
{
    int i, n;
    TEvent *event;
    vlong runt;

    t->tevents[ep->etype & 0xffff]++;
    n = t->nevents++;
    t->events = realloc(t->events, t->nevents*sizeof(TEvent));
    assert(t->events);
    event = &t->events[n];
    memmove(event, ep, sizeof(Traceevent));
    event->etime = 0;

    switch(event->etype & 0xffff){
    case SRelease:
        if (t->runthis > t->runmax)
            t->runmax = t->runthis;
        t->runthis = 0;
        break;

    case SSleep:
    case SYield:
    case SReady:
    case SSlice:
        for(i = n-1; i >= 0; i--)
            if (t->events[i].etype == SRun)
                break;
        if(i < 0 || t->events[i].etime != 0)
            break;
        runt = event->time - t->events[i].time;
        if(runt > 0){
            t->events[i].etime = event->time;
            t->runtime += runt;
            t->total += runt;
            t->runthis += runt;
            t->runs++;
        }
        break;
    case SDead:
```

```
print("task died %ld %t %s\n", event->pid, event->time, schedstatename[event->etype & 0xffff]);
        free(t->events);
        free(t->name);
        ntasks--;
        memmove(t, t+1, sizeof(Task)*(&tasks[ntasks]-t));
        if (newwin)
            fprint(wctlfd, "resize -dx %d -dy %d\n",
                Width + 20, (ntasks * Height) + 5);
        else
            Height = ntasks ? Dy(view->r)/ntasks : Dy(view->r);
        prevts = 0;
    }
}
```

Uses `Height` 72j, `Width` 72i, `event` 73a, `newwin` 72h, `ntasks` 73c, `prevts` 73c, `schedstatename` 73b, `tasks` 47c, and `wctlfd` 72n.

⟨*function* `drawtrace` 55⟩≡ (73c)

```
void
drawtrace(void)
{
    char *wsys, line[256];
    int wfd, logfd;
    Mousectl *mousectl;
    Keyboardctl *keyboardctl;
    int scaleno;
    Rune r;
    int i, n;
    Task *t;
    Traceevent *ep;

    eventbuf = malloc(Nevents*sizeof(Traceevent));
    assert(eventbuf);

    if((logfd = open(profdev, OREAD)) < 0)
        sysfatal("%s: Cannot open %s: %r", argv0, profdev);

    if(newwin){
        if((wsys = getenv("wsys")) == nil)
            sysfatal("%s: Cannot find windowing system: %r",
                        argv0);

        if((wfd = open(wsys, ORDWR)) < 0)
            sysfatal("%s: Cannot open windowing system: %r",
                        argv0);

        snprint(line, sizeof(line), "new -pid %d -dx %d -dy %d",
                getpid(), Width + 20, Height + 5);
        line[sizeof(line) - 1] = '\0';
        rfork(RFNAMEG);

        if(mount(wfd, -1, "/mnt/wsys", MREPL, line) < 0)
            sysfatal("%s: Cannot mount %s under /mnt/wsys: %r",
                        argv0, line);

        if(bind("/mnt/wsys", "/dev", MBEFORE) < 0)
            sysfatal("%s: Cannot bind /mnt/wsys in /dev: %r",
                        argv0);

    }
    if((wctlfd = open("/dev/wctl", OWRITE)) < 0)
        sysfatal("%s: Cannot open /dev/wctl: %r", argv0);
```

```
if(initdraw(nil, nil, "trace") < 0)
    sysfatal("%s: initdraw failure: %r", argv0);

Width = Dx(view->r);
Height = Dy(view->r);

if((mousectl = initmouse(nil, view)) == nil)
    sysfatal("%s: cannot initialize mouse: %r", argv0);

if((keyboardctl = initkeyboard(nil)) == nil)
    sysfatal("%s: cannot initialize keyboard: %r", argv0);

colinit();

paused = 0;
scaleno = 7; /* 100 milliseconds */
now = nsec();
for(;;) {
    Alt a[] = {
        { mousectl->c,   nil,  CHANRCV  },
        { mousectl->resizec, nil,  CHANRCV  },
        { keyboardctl->c,  &r,   CHANRCV  },
        { nil,     nil,  CHANNOBLK },
    };

    switch (alt(a)) {
    case 0:
        continue;

    case 1:
        if(getwindow(display, Refnone) < 0)
            sysfatal("drawrt: Cannot re-attach window");
        if(newwin){
            if(Dx(view->r) != Width ||
                Dy(view->r) != (ntasks * Height)){
                fprint(2, "resize: x: have %d, need %d; y: have %d, need %d\n",
                        Dx(view->r), Width + 8, Dy(view->r), (ntasks * Height) + 8);
                fprint(wctlfd, "resize -dx %d -dy %d\n",
                        Width + 8, (ntasks * Height) + 8);
            }
        }
        else{
            Width = Dx(view->r);
            Height = ntasks? Dy(view->r)/ntasks:
                        Dy(view->r);
        }
        break;

    case 2:

        switch(r){
        case 'r':
            for(i = 0; i < ntasks; i++){
                tasks[i].tstart = now;
                tasks[i].total = 0;
                tasks[i].runtime = 0;
                tasks[i].runmax = 0;
                tasks[i].runthis = 0;
                tasks[i].runs = 0;
                memset(tasks[i].tevents, 0, Nevent*sizeof(ulong));
```

```
        }
        break;

    case 'p':
        paused ^= 1;
        prevts = 0;
        break;

    case '-':
        if (scaleno < nelem(scales) - 1)
            scaleno++;
        prevts = 0;
        break;

    case '+':
        if (scaleno > 0)
            scaleno--;
        prevts = 0;
        break;

    case 'q':
        threadexitsall(nil);

    case 'v':
        verbose ^= 1;

    default:
        break;
    }
    break;

case 3:
    now = nsec();
    while((n = read(logfd, eventbuf, Nevents*sizeof(Traceevent))) > 0){
        assert((n % sizeof(Traceevent)) == 0);
        nevents = n / sizeof(Traceevent);
        for (ep = eventbuf; ep < eventbuf + nevents; ep++){
            if ((ep->etype & 0xffff) >= Nevent){
                print("%ld %t Illegal event %ld\n",
                    ep->pid, ep->time, ep->etype & 0xffff);
                continue;
            }
            if (verbose)
                print("%ld %t %s\n",
                    ep->pid, ep->time, schedstatename[ep->etype & 0xffff]);

            for(i = 0; i < ntasks; i++)
                if(tasks[i].pid == ep->pid)
                    break;

            if(i == ntasks){
                t = newtask(ep->pid);
                t->tstart = ep->time;
            }else
                t = &tasks[i];

            doevent(t, ep);
        }
    }
```

```
            if(!paused)
                redraw(scaleno);
        }
        sleep(scales[scaleno].sleep);
    }
}
```

Uses Height 72j, Nevents-60 46c, Width 72i, doevent() 54, eventbuf 72p, nevents 72o, newtask() 53, newwin 72h, now 73c, ntasks 73c, paused 73c, prevts 73c, profdev 47e, scales 47a, schedstatename 73b, tasks 47c, and wctlfd 72n.

⟨*function* timeconv 58⟩≡ (73c)

```
int
timeconv(Fmt *f)
{
    char buf[128], *sign;
    vlong t;

    buf[0] = 0;
    switch(f->r) {
    case 'U':
        t = va_arg(f->args, vlong);
        break;
    case 't':  // vlong in nanoseconds
        t = va_arg(f->args, vlong);
        break;
    default:
        return fmtstrcpy(f, "(timeconv)");
    }
    if (t < 0) {
        sign = "-";
        t = -t;
    }else
        sign = "";
    if (t > S(1)){
        t += OneRound;
        sprint(buf, "%s%d.%.3ds", sign, (int)(t / S(1)), (int)(t % S(1))/1000000);
    }else if (t > MS(1)){
        t += MilliRound;
        sprint(buf, "%s%d.%.3dms", sign, (int)(t / MS(1)), (int)(t % MS(1))/1000);
    }else if (t > US(1))
        sprint(buf, "%s%d.%.3dµs", sign, (int)(t / US(1)), (int)(t % US(1)));
    else
        sprint(buf, "%s%dns", sign, (int)t);
    return fmtstrcpy(f, buf);
}
```

Uses MS-54 72c, MilliRound-59 72g, OneRound-58 72g, S-55 72d, and US-53 72b.

Chapter 10

System Monitoring /bin/stats

While the previous tools focus on individual programs, `stats` monitors the entire system: CPU load, memory usage, context switches, syscalls, interrupts, network traffic, and more. It reads kernel pseudo-files (`#c/sysstat`, `#c/swap`, `/net/ether0/0/stats`) once per second and plots rolling graphs, one per selected metric. It can even monitor multiple machines simultaneously over the network, drawing their graphs in adjacent columns. At roughly 1600 lines, `stats` is the largest tool in this book, mostly because of its graphical display code. The core architecture is simple: a `Machine` struct holds a network connection to each monitored host, and a `Graph` struct holds the display state for each metric. The main loop reads all machines, updates all graphs, and sleeps for one second.

10.1 Data structures

⟨*constant* MAXNUM 59a⟩≡ (95)

```
// a GUI system monitoring tool

#define MAXNUM 10 /* maximum number of numbers on data line */
```

⟨*struct* Graph 59b⟩≡ (95)

```
struct Graph
{
    int  colindex;
    Rectangle r;
    uvlong  *data;
    int  ndata;
    char  *label;
    void  (*newvalue)(Machine*, uvlong*, uvlong*, int);
    void  (*update)(Graph*, uvlong, uvlong);
    Machine  *mach;
    int  overflow;
    Image  *overtmp;
};
```

⟨*enum* _anon_ (misc/stats.c) 59c⟩≡ (95)

```
enum
{
    /* old /dev/swap */
    Mem  = 0,
    Maxmem,
    Swap,
    Maxswap,

    /* /dev/sysstats */
    Procno = 0,
```

```
    Context,
    Interrupt,
    Syscall,
    Fault,
    TLBfault,
    TLBpurge,
    Load,
    Idle,
    InIntr,
    /* /net/ether0/stats */
    In  = 0,
    Link,
    Out,
    Err0,
};
```

⟨*struct* Machine 60a⟩≡ (95)

```
struct Machine
{
    char  *name;
    char  *shortname;
    int  remote;
    int  statsfd;
    int  swapfd;
    int  etherfd;
    int  ifstatsfd;
    int  batteryfd;
    int  bitsybatfd;
    int  tempfd;
    int  disable;

    uvlong  devswap[4];
    uvlong  devsysstat[10];
    uvlong  prevsysstat[10];
    int  nproc;
    int  lgproc;
    uvlong  netetherstats[8];
    uvlong  prevetherstats[8];
    uvlong  batterystats[2];
    uvlong  netetherifstats[2];
    uvlong  temp[10];

    /* big enough to hold /dev/sysstat even with many processors */
    char  buf[8*1024];
    char  *bufp;
    char  *ebufp;
};
```

⟨*enum* _anon_ (misc/stats.c)2 60b⟩≡ (95)

```
enum
{
    Mainproc,
    Mouseproc,
    NPROC,
};
```

⟨*enum* _anon_ (misc/stats.c)3 60c⟩≡ (95)

```
enum
{
    Ncolor  = 6,
```

```
    Ysqueeze = 2, /* vertical squeezing of label text */
    Labspace = 2, /* room around label */
    Dot  = 2, /* height of dot */
    Opwid  = 5, /* strlen("add  ") or strlen("drop ") */
    Nlab  = 3, /* max number of labels on y axis */
    Lablen  = 16, /* max length of label */
    Lx  = 4, /* label tick length */
};
```

⟨*enum* `Menu2` 61a⟩≡ (95)

```
enum Menu2
{
    Mbattery,
    Mcontext,
    Mether,
    Methererr,
    Metherin,
    Metherout,
    Mfault,
    Midle,
    Minintr,
    Mintr,
    Mload,
    Mmem,
    Mswap,
    Msyscall,
    Mtlbmiss,
    Mtlbpurge,
    Msignal,
    Mtemp,
    Nmenu2,
};
```

⟨*global* `menu2str` 61b⟩≡ (95)

```
char *menu2str[Nmenu2+1] = {
    "add  battery ",
    "add  context ",
    "add  ether   ",
    "add  ethererr",
    "add  etherin ",
    "add  etherout",
    "add  fault   ",
    "add  idle    ",
    "add  inintr  ",
    "add  intr    ",
    "add  load    ",
    "add  mem     ",
    "add  swap    ",
    "add  syscall ",
    "add  tlbmiss ",
    "add  tlbpurge",
    "add  802.11b ",
    "add  temp    ",
    nil,
};
```

Uses `Nmenu2-49` 61a.

⟨*global* `menu2` 61c⟩≡ (95)

```
Menu menu2 = {menu2str, nil};
```

Uses `menu2str` 61b.

⟨*global* present 62a⟩≡ (95)
```
int present[Nmenu2];
```
Uses Nmenu2-49 61a.

⟨*global* newvaluefn 62b⟩≡ (95)
```
void (*newvaluefn[Nmenu2])(Machine*, uvlong*, uvlong*, int init) = {
    batteryval,
    contextval,
    etherval,
    ethererrval,
    etherinval,
    etheroutval,
    faultval,
    idleval,
    inintrval,
    intrval,
    loadval,
    memval,
    swapval,
    syscallval,
    tlbmissval,
    tlbpurgeval,
    signalval,
    tempval,
};
```
Uses Nmenu2-49 61a, batteryval() 89b, contextval() 86c, ethererrval() 89a, etherinval() 88d, etheroutval() 88e, etherval() 88c, faultval() 87b, idleval() 88a, inintrval() 88b, intrval() 86d, loadval() 87e, memval() 86a, signalval() 89c, swapval() 86b, syscallval() 87a, tempval() 89d, tlbmissval() 87c, and tlbpurgeval() 87d.

⟨*global cols (misc/stats.c)* 62c⟩≡ (95)
```
Image *cols[Ncolor][3];
```
Uses Ncolor-23 60c.

⟨*global* graph 62d⟩≡ (95)
```
Graph *graph;
```

⟨*global* mach 62e⟩≡ (95)
```
Machine *mach;
```

⟨*global* mediumfont 62f⟩≡ (95)
```
Font *mediumfont;
```

⟨*global* mysysname 62g⟩≡ (95)
```
char *mysysname;
```

⟨*global* argchars 62h⟩≡ (95)
```
char argchars[] = "8bceEfiImlnpstwz";
```
Uses argchars 62h.

⟨*global* pids 62i⟩≡ (95)
```
int pids[NPROC];
```
Uses NPROC-22 60b.

⟨*global* parity 62j⟩≡ (95)
```
int  parity; /* toggled to avoid patterns in textured background */
```

⟨*global* nmach 62k⟩≡ (95)
```
int nmach;
```

⟨*global* ngraph 63a⟩≡ (95)
```
int ngraph; /* totaly number is ngraph*nmach */
```

⟨*global* scale 63b⟩≡ (95)
```
double scale = 1.0;
```
Uses scale 46d 63b.

⟨*global* logscale 63c⟩≡ (95)
```
int logscale = 0;
```
Uses logscale 63c.

⟨*global* ylabels 63d⟩≡ (95)
```
int ylabels = 0;
```
Uses ylabels 63d.

⟨*global* oldsystem 63e⟩≡ (95)
```
int oldsystem = 0;
```
Uses oldsystem 63e.

⟨*global* sleeptime 63f⟩≡ (95)
```
int  sleeptime = 1000;
```
Uses sleeptime 63f.

⟨*global* procnames 63g⟩≡ (95)
```
char *procnames[NPROC] = {"main", "mouse"};
```
Uses NPROC-22 60b.

10.2 main()

⟨*function main (misc/stats.c)* 63h⟩≡ (95)
```
void
main(int argc, char *argv[])
{
    int i, j;
    double secs;
    uvlong v, vmax, nargs;
    char args[100];

    nmach = 1;
    mysysname = getenv("sysname");
    if(mysysname == nil){
        fprint(2, "stats: can't find $sysname: %r\n");
        exits("sysname");
    }
    mysysname = estrdup(mysysname);

    nargs = 0;
    ARGBEGIN{
    case 'T':
        secs = atof(EARGF(usage()));
        if(secs > 0)
            sleeptime = 1000*secs;
        break;
    case 'S':
        scale = atof(EARGF(usage()));
        if(scale <= 0)
            usage();
```

```
        break;
    case 'L':
        logscale++;
        break;
    case 'Y':
        ylabels++;
        break;
    case 'O':
        oldsystem = 1;
        break;
    default:
        if(nargs>=sizeof args || strchr(argchars, ARGC())==nil)
            usage();
        args[nargs++] = ARGC();
    }ARGEND

    if(argc == 0){
        mach = emalloc(nmach*sizeof(Machine));
        initmach(&mach[0], mysysname);
        readmach(&mach[0], 1);
    }else{
        for(i=j=0; i<argc; i++){
            if (addmachine(argv[i]))
                readmach(&mach[j++], 1);
        }
        if (j == 0)
            exits("connect");
    }

    for(i=0; i<nargs; i++)
    switch(args[i]){
    default:
        fprint(2, "stats: internal error: unknown arg %c\n", args[i]);
        usage();
    case 'b':
        addgraph(Mbattery);
        break;
    case 'c':
        addgraph(Mcontext);
        break;
    case 'e':
        addgraph(Mether);
        break;
    case 'E':
        addgraph(Metherin);
        addgraph(Metherout);
        break;
    case 'f':
        addgraph(Mfault);
        break;
    case 'i':
        addgraph(Mintr);
        break;
    case 'I':
        addgraph(Mload);
        addgraph(Midle);
        addgraph(Minintr);
        break;
    case 'l':
        addgraph(Mload);
```

```
    break;
case 'm':
    addgraph(Mmem);
    break;
case 'n':
    addgraph(Metherin);
    addgraph(Metherout);
    addgraph(Methererr);
    break;
case 'p':
    addgraph(Mtlbpurge);
    break;
case 's':
    addgraph(Msyscall);
    break;
case 't':
    addgraph(Mtlbmiss);
    addgraph(Mtlbpurge);
    break;
case '8':
    addgraph(Msignal);
    break;
case 'w':
    addgraph(Mswap);
    break;
case 'z':
    addgraph(Mtemp);
    break;
}

if(ngraph == 0)
    addgraph(Mload);

for(i=0; i<nmach; i++)
    for(j=0; j<ngraph; j++)
        graph[i*ngraph+j].mach = &mach[i];

if(initdraw(nil, nil, "stats") < 0){
    fprint(2, "stats: initdraw failed: %r\n");
    exits("initdraw");
}
colinit();
einit(Emouse);
notify(nil);
startproc(mouseproc, Mouseproc);
pids[Mainproc] = getpid();
display->locking = 1; /* tell library we're using the display lock */

resize();

unlockdisplay(display); /* display is still locked from initdraw() */
for(;;){
    for(i=0; i<nmach; i++)
        readmach(&mach[i], 0);
    lockdisplay(display);
    parity = 1-parity;
    for(i=0; i<nmach*ngraph; i++){
        graph[i].newvalue(graph[i].mach, &v, &vmax, 0);
        graph[i].update(&graph[i], v, vmax);
    }
```

```
        flushimage(display, 1);
        unlockdisplay(display);
        sleep(sleeptime);
    }
}
```

Chapter 11

IO Monitoring: `/bin/iostats`

`iostats` is unique among the profiling tools: it is not a measurement program but a proxy file server. It interposes itself between a program and the regular file server, intercepting all 9P messages. When the program exits, `iostats` prints a report showing the count, size, and latency of each 9P message type, plus a per-file summary of bytes read and written. This is a quintessentially Plan 9 approach to profiling: instead of adding tracing hooks inside the kernel, you insert a transparent proxy in the namespace. The program being profiled is completely unaware—it just opens and reads files as usual, and `iostats` records every 9P message that passes through. The implementation (roughly 1500 lines across `iostats.c`, `statsrv.c`, and `statfs.h`) is structured as a multi-threaded 9P server. For each 9P message type (`Tread`, `Twrite`, `Topen`, etc.), it records a count, total bytes transferred, and cumulative time, then forwards the message to the real server. On exit, it prints the accumulated statistics. The code for `iostats` is in `Profiler_extra.nw`.

Chapter 12

Conclusion

You now know how the Plan 9 profiling tools work—from the simple `fork/exec/wait` pattern of `time`, through the linker-inserted instrumentation of `prof`, the kernel's PC-sampling histogram read by `tprof`, to the real-time graphical displays of `trace` and `stats`—and more generally how many profilers work.

Despite totaling only about 4900 lines of C, these tools cover the three main approaches to profiling: instrumentation (`prof`, via the linker's `_profin`/`_profout` hooks), sampling (`tprof` and `kprof`, via the kernel's `profclock()` timer), and tracing/monitoring (`trace` and `stats`, via `/proc/trace` and kernel pseudo-files). The simplest tool, `time`, is barely 60 lines. The most complex, `stats`, reaches about 1600 lines, mostly because of its graphical display code. Along the way, you have seen how profiling depends on the cooperation of three layers: the kernel (time accounting, PC sampling, scheduler events), the linker (function call instrumentation), and user-space tools (reading and presenting the data). You have also seen the `Tos` structure, an elegant mechanism that lets user code read precise timing data without the overhead of a system call.

12.1 Patterns and techniques

These techniques apply far beyond performance measurement:

- *Instrumentation vs. sampling*: these are the two fundamental approaches to observing any running system. The same trade-off appears in distributed tracing (OpenTelemetry spans vs. tail sampling) and database monitoring (query logging vs. periodic snapshots): exact data with overhead, or statistical data cheaply.

- *Shared-memory communication*: the `Tos` structure lets user code read kernel-maintained counters without a system call. Linux's `vDSO` provides `gettimeofday()` the same way. Whenever crossing a boundary is expensive, mapping shared memory is the standard fix.

- *Wrapper process pattern*: `time` and `prof` fork a child, exec the target, then collect results. This "wrap and observe" pattern is how `strace`, `valgrind`, and container runtimes work: interpose a supervisor around an unmodified program to add behavior it was never designed for.

12.2 Connections to other books

- Kernel book [Pad14] describes the kernel-side infrastructure that the profiling tools rely on: `accountime()` for per-process time tracking, `profclock()` and the `profile` array for PC sampling, `/proc/trace` for scheduler events, and the `Tos` structure for cycle counting. The kernel's `kprof(3)` device provides kernel self-profiling.

- Linker book [Pad15] explains how `5l -p` instruments programs by inserting `BL _profin`/`BL _profout` around every function, and how `5l -p -1` implements the simpler `__mcount` call-counting strategy.

- Libcore book [Pad16a] details the code of `_profin`, `_profout`, `_profmain`, and `_profdump`.
- Debugger book [Pad16b] covers `libmach`, the library that `prof`, `tprof`, and `kprof` use to read symbol tables from executables and map program counter values to function names.
- Graphics book [Pad16c] covers the draw library used by `trace` and `stats` for their graphical displays.

12.3 Beyond the Plan 9 profilers

The Plan 9 profilers are intentionally simple. Modern profiling ecosystems are considerably more powerful, though also more complex:

- *Hardware performance counters*: modern CPUs provide counters for cache misses, branch mispredictions, TLB misses, and dozens of other microarchitectural events. Linux's `perf` can read these to give a much richer picture of performance than time alone.
- *Call graph from sampling*: tools like `perf` walk the stack at each sample to reconstruct call graphs, combining the low overhead of sampling with the structural information of instrumentation. Plan 9's `prof` achieves something similar through its `_profin`/`_profout` approach, at the cost of higher runtime overhead.
- *Flame graphs*: a visualization technique (invented by Brendan Gregg) that displays sampled call stacks as nested rectangles, making it easy to spot which call paths consume the most time. Flame graphs have become the standard way to explore profiling data in most languages and platforms.
- *Dynamic instrumentation*: tools like DTrace (Solaris/macOS) and eBPF (Linux) can instrument arbitrary points in a running program or kernel without recompilation, by patching instructions or inserting bytecode at runtime. DTrace was considered revolutionary when it appeared, and eBPF has become one of the most important Linux kernel technologies, enabling not just profiling but also networking, security, and observability.
- *Continuous profiling*: production systems increasingly run always-on profilers (Google-Wide Profiling, Parca, Pyroscope) that sample across an entire fleet and aggregate results over time. This is a different world from Plan 9's single-machine tools, driven by the scale of cloud deployments.

The Plan 9 profilers demonstrate that effective profiling does not require enormous complexity. A few hundred lines of C, combined with simple kernel support, are enough to answer the most common performance questions: where is my program spending its time, and how many times is each function called? The fundamental techniques—instrumentation, sampling, and tracing—are the same ones that `perf` and DTrace use; the Plan 9 tools simply present them in their clearest form.

Appendix A

Extra Code

A.1 profilers/

A.1.1 misc/time.c

⟨misc/time.c 70a⟩≡

```
#include <u.h>
#include <libc.h>

⟨global output (time.c) 23a⟩

//forward decl
void add(char*, ...);
void notifyf(void*, char*);

⟨function error (time.c) 24a⟩
⟨function main (time.c) 22⟩
⟨function add (time.c) 23b⟩
⟨function notifyf (time.c) 24c⟩
```

A.1.2 misc/kprof.c

⟨misc/kprof.c 70b⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <mach.h>

⟨constant PCRES 43d⟩

⟨struct COUNTER 43e⟩

⟨function error (misc/kprof.c) 44a⟩

⟨function compar 44b⟩
⟨function main (misc/kprof.c) 42⟩
```

A.1.3 misc/prof.c

⟨misc/prof.c 70c⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
```

```
#include <mach.h>

typedef struct Data Data;
typedef struct Pc Pc;
typedef struct Acc Acc;

⟨struct Data 26a⟩

⟨struct Pc 27c⟩

⟨struct Acc 27a⟩

⟨global data 26b⟩
⟨global acc 27b⟩
⟨global ms 31a⟩
⟨global nsym 29a⟩
⟨global ndata 26c⟩
⟨global dflag 33c⟩
⟨global rflag 35e⟩
⟨global bout 28a⟩
⟨global tabstop 35b⟩
⟨global verbose 28b⟩

void syms(char*);
void datas(char*);
void graph(int, ulong, Pc*);
void plot(void);
char* name(ulong);
void indent(int);
char* defaout(void);

⟨function main (misc/prof.c) 27d⟩

⟨function swapdata 30a⟩

⟨function acmp 33b⟩

⟨function syms 29b⟩

⟨function datas 29e⟩

⟨function name 35a⟩

⟨function graph 34c⟩
⟨function symind 33a⟩

⟨function sum 31c⟩

⟨function plot 30e⟩

⟨function indent 35d⟩

⟨global trans 28e⟩

⟨function defaout 28d⟩
```
Uses `Acc` 27a, `Data` 26a, and `Pc` 27c.

A.1.4 misc/trace.c

⟨*function* NS 72a⟩≡ (73c)
```
#define NS(x) ((vlong)x)
```

⟨*function* US 72b⟩≡ (73c)
```
#define US(x) (NS(x) * 1000ULL)
```

⟨*function* MS 72c⟩≡ (73c)
```
#define MS(x) (US(x) * 1000ULL)
```

⟨*function* S 72d⟩≡ (73c)
```
#define S(x) (MS(x) * 1000ULL)
```

⟨*function* numblocks 72e⟩≡ (73c)
```
#define numblocks(a, b) (((a) + (b) - 1) / (b))
```

⟨*function* roundup 72f⟩≡ (73c)
```
#define roundup(a, b) (numblocks((a), (b)) * (b))
```

⟨*enum* _anon_ 72g⟩≡ (73c)
```
enum {
    OneRound = MS(1)/2LL,
    MilliRound = US(1)/2LL,
};
```
Uses MS-54 72c and US-53 72b.

⟨*global* newwin 72h⟩≡ (73c)
```
int newwin;
```

⟨*global* Width 72i⟩≡ (73c)
```
int Width = 1000;
```
Uses Width 72i.

⟨*global* Height 72j⟩≡ (73c)
```
int Height = 100;  // Per task
```
Uses Height 72j.

⟨*global* topmargin 72k⟩≡ (73c)
```
int topmargin = 8;
```
Uses topmargin 72k.

⟨*global* bottommargin 72l⟩≡ (73c)
```
int bottommargin = 4;
```
Uses bottommargin 72l.

⟨*global* lineht 72m⟩≡ (73c)
```
int lineht = 12;
```
Uses lineht 72m.

⟨*global* wctlfd 72n⟩≡ (73c)
```
int wctlfd;
```

⟨*global* nevents 72o⟩≡ (73c)
```
int nevents;
```

⟨*global* eventbuf 72p⟩≡ (73c)
```
Traceevent *eventbuf;
```

⟨*global* event 73a⟩≡ (73c)

```
TEvent *event;
```

⟨*global* schedstatename 73b⟩≡ (73c)

```
char *schedstatename[] = {
    [SReady] = "Ready",
    [SRun] = "Run",
    [SDead] = "Dead",
    [SSleep] = "Sleep",
    [SUser] =  "User",

    [SAdmit] = "Admit",
    [SRelease] = "Release",
    [SYield] = "Yield",
    [SSlice] = "Slice",
    [SDeadline] = "Deadline",
    [SExpel] = "Expel",
    [SInts] = "Ints",
    [SInte] = "Inte",
};
```

⟨misc/trace.c 73c⟩≡

```
#include <u.h>
#include <tos.h>
#include <libc.h>
#include <thread.h>
#include <ip.h>
#include <bio.h>

#include <draw.h>
#include <window.h>
#include <mouse.h>
#include <cursor.h>
#include <keyboard.h>

#include <trace.h>

// a GUI tracer for kernel scheduler events

#pragma varargck type "t"  vlong
#pragma varargck type "U"  uvlong

⟨function NS 72a⟩
⟨function US 72b⟩
⟨function MS 72c⟩
⟨function S 72d⟩

⟨function numblocks 72e⟩
⟨function roundup 72f⟩

⟨enum _anon_ 72g⟩

typedef struct TEvent TEvent;
typedef struct Task Task;
⟨struct TEvent 46a⟩

⟨struct Task 46b⟩

⟨enum _anon_ (misc/trace.c) 46c⟩
```

```
  vlong now, prevts;

  ⟨global newwin 72h⟩
  ⟨global Width 72i⟩
  ⟨global Height 72j⟩
  ⟨global topmargin 72k⟩
  ⟨global bottommargin 72l⟩
  ⟨global lineht 72m⟩
  ⟨global wctlfd 72n⟩
  ⟨global nevents 72o⟩
  ⟨global eventbuf 72p⟩
  ⟨global event 73a⟩

  void drawtrace(void);
  int schedparse(char*, char*, char*);
  int timeconv(Fmt*);

  ⟨global schedstatename 73b⟩

  ⟨struct scale 46d⟩

  ⟨global scales 47a⟩

  int ntasks, triggerproc, paused;
  ⟨global verbose (misc/trace.c) 47b⟩
  ⟨global tasks 47c⟩
  ⟨global cols 47d⟩
  static Font *mediumfont, *tinyfont;
  Image *grey, *red, *green, *blue, *bg, *fg;
  ⟨global profdev 47e⟩

  ⟨function usage 47f⟩

  ⟨function threadmain 48a⟩

  ⟨function mkcol 48b⟩

  ⟨function colinit 48c⟩

  ⟨function time2x 49a⟩

  ⟨function redraw 49b⟩

  ⟨function newtask 53⟩

  ⟨function doevent 54⟩

  ⟨function drawtrace 55⟩

  ⟨function timeconv 58⟩
```
Uses TEvent 73c and Task 73c.

A.1.5 misc/tprof.c

```
⟨misc/tprof.c 74⟩≡
  #include <u.h>
  #include <libc.h>
  #include <bio.h>
  #include <mach.h>
```

⟨*constant PCRES (misc/tprof.c)* 39b⟩

⟨*struct COUNTER (misc/tprof.c)* 39c⟩

⟨*function error (misc/tprof.c)* 40b⟩

⟨*function compar (misc/tprof.c)* 40a⟩
⟨*function main (misc/tprof.c)* 37⟩

A.1.6 misc/stats.c

⟨*function* killall 75a⟩≡ (95)

```
void
killall(char *s)
{
    int i, pid;

    pid = getpid();
    for(i=0; i<NPROC; i++)
        if(pids[i] && pids[i]!=pid)
            postnote(PNPROC, pids[i], "kill");
    exits(s);
}
```

Uses NPROC-22 60b and pids 62i.

⟨*function* emalloc 75b⟩≡ (95)

```
void*
emalloc(ulong sz)
{
    void *v;
    v = malloc(sz);
    if(v == nil) {
        fprint(2, "stats: out of memory allocating %ld: %r\n", sz);
        killall("mem");
    }
    memset(v, 0, sz);
    return v;
}
```

Uses killall() 75a.

⟨*function* erealloc 75c⟩≡ (95)

```
void*
erealloc(void *v, ulong sz)
{
    v = realloc(v, sz);
    if(v == nil) {
        fprint(2, "stats: out of memory reallocating %ld: %r\n", sz);
        killall("mem");
    }
    return v;
}
```

Uses killall() 75a.

⟨*function* estrdup 75d⟩≡ (95)

```
char*
estrdup(char *s)
{
```

```
    char *t;
    if((t = strdup(s)) == nil) {
        fprint(2, "stats: out of memory in strdup(%.10s): %r\n", s);
        killall("mem");
    }
    return t;
}
```

Uses killall() 75a.

⟨*function mkcol (misc/stats.c)* 76a⟩≡ (95)

```
void
mkcol(int i, int c0, int c1, int c2)
{
    cols[i][0] = allocimagemix(display, c0, DWhite);
    cols[i][1] = allocimage(display, Rect(0,0,1,1), CMAP8, 1, c1);
    cols[i][2] = allocimage(display, Rect(0,0,1,1), CMAP8, 1, c2);
}
```

⟨*function colinit (misc/stats.c)* 76b⟩≡ (95)

```
void
colinit(void)
{
    mediumfont = openfont(display, "/lib/font/bit/pelm/latin1.8.font");
    if(mediumfont == nil)
        mediumfont = font;

    /* Peach */
    mkcol(0, 0xFFAAAAFF, 0xFFAAAAFF, 0xBB5D5DFF);
    /* Aqua */
    mkcol(1, DPalebluegreen, DPalegreygreen, DPurpleblue);
    /* Yellow */
    mkcol(2, DPaleyellow, DDarkyellow, DYellowgreen);
    /* Green */
    mkcol(3, DPalegreen, DMedgreen, DDarkgreen);
    /* Blue */
    mkcol(4, 0x00AAFFFF, 0x00AAFFFF, 0x0088CCFF);
    /* Grey */
    cols[5][0] = allocimage(display, Rect(0,0,1,1), CMAP8, 1, 0xEEEEEEFF);
    cols[5][1] = allocimage(display, Rect(0,0,1,1), CMAP8, 1, 0xCCCCCCFF);
    cols[5][2] = allocimage(display, Rect(0,0,1,1), CMAP8, 1, 0x888888FF);
}
```

⟨*function* loadbuf 76c⟩≡ (95)

```
int
loadbuf(Machine *m, int *fd)
{
    int n;


    if(*fd < 0)
        return 0;
    seek(*fd, 0, 0);
    n = read(*fd, m->buf, sizeof m->buf-1);
    if(n <= 0){
        close(*fd);
        *fd = -1;
        return 0;
    }
    m->bufp = m->buf;
    m->ebufp = m->buf+n;
```

```
    m->buf[n] = 0;
    return 1;
}
```

⟨*function* label 77a⟩≡ (95)

```
void
label(Point p, int dy, char *text)
{
    char *s;
    Rune r[2];
    int w, maxw, maxy;

    p.x += Labspace;
    maxy = p.y+dy;
    maxw = 0;
    r[1] = '\0';
    for(s=text; *s; ){
        if(p.y+mediumfont->height-Ysqueeze > maxy)
            break;
        w = chartorune(r, s);
        s += w;
        w = runestringwidth(mediumfont, r);
        if(w > maxw)
            maxw = w;
        runestring(view, p, display->black, ZP, mediumfont, r);
        p.y += mediumfont->height-Ysqueeze;
    }
}
```

Uses Labspace-25 60c and Ysqueeze-24 60c.

⟨*function* paritypt 77b⟩≡ (95)

```
Point
paritypt(int x)
{
    return Pt(x+parity, 0);
}
```

Uses parity 62j.

⟨*function* datapoint 77c⟩≡ (95)

```
Point
datapoint(Graph *g, int x, uvlong v, uvlong vmax)
{
    Point p;
    double y;

    p.x = x;
    y = ((double)v)/(vmax*scale);
    if(logscale){
        /*
         * Arrange scale to cover a factor of 1000.
         * vmax corresponds to the 100 mark.
         * 10*vmax is the top of the scale.
         */
        if(y <= 0.)
            y = 0;
        else{
            y = log10(y);
            /* 1 now corresponds to the top; -2 to the bottom; rescale */
            y = (y+2.)/3.;
        }
```

```
    }
    if(y < 0x7fffffff){ /* avoid floating overflow */
        p.y = g->r.max.y - Dy(g->r)*y - Dot;
        if(p.y < g->r.min.y)
            p.y = g->r.min.y;
        if(p.y > g->r.max.y-Dot)
            p.y = g->r.max.y-Dot;
    }else
        p.y = g->r.max.y-Dot;
    return p;
}
```
Uses `Dot-26` 60c, `logscale` 63c, and `scale` 46d 63b.

⟨*function* `drawdatum` 78a⟩≡ (95)
```
void
drawdatum(Graph *g, int x, uvlong prev, uvlong v, uvlong vmax)
{
    int c;
    Point p, q;

    c = g->colindex;
    p = datapoint(g, x, v, vmax);
    q = datapoint(g, x, prev, vmax);
    if(p.y < q.y){
        draw(view, Rect(p.x, g->r.min.y, p.x+1, p.y), cols[c][0], nil, paritypt(p.x));
        draw(view, Rect(p.x, p.y, p.x+1, q.y+Dot), cols[c][2], nil, ZP);
        draw(view, Rect(p.x, q.y+Dot, p.x+1, g->r.max.y), cols[c][1], nil, ZP);
    }else{
        draw(view, Rect(p.x, g->r.min.y, p.x+1, q.y), cols[c][0], nil, paritypt(p.x));
        draw(view, Rect(p.x, q.y, p.x+1, p.y+Dot), cols[c][2], nil, ZP);
        draw(view, Rect(p.x, p.y+Dot, p.x+1, g->r.max.y), cols[c][1], nil, ZP);
    }

}
```
Uses `Dot-26` 60c, `datapoint()` 77c, and `paritypt()` 77b.

⟨*function redraw (misc/stats.c)* 78b⟩≡ (95)
```
void
redraw(Graph *g, uvlong vmax)
{
    int i, c;

    c = g->colindex;
    draw(view, g->r, cols[c][0], nil, paritypt(g->r.min.x));
    for(i=1; i<Dx(g->r); i++)
        drawdatum(g, g->r.max.x-i, g->data[i-1], g->data[i], vmax);
    drawdatum(g, g->r.min.x, g->data[i], g->data[i], vmax);
    g->overflow = 0;
}
```
Uses `drawdatum()` 78a and `paritypt()` 77b.

⟨*function* `update1` 78c⟩≡ (95)
```
void
update1(Graph *g, uvlong v, uvlong vmax)
{
    char buf[48];
    int overflow;

    if(g->overflow && g->overtmp!=nil)
```

```
        draw(view, g->overtmp->r, g->overtmp, nil, g->overtmp->r.min);
    draw(view, g->r, view, nil, Pt(g->r.min.x+1, g->r.min.y));
    drawdatum(g, g->r.max.x-1, g->data[0], v, vmax);
    memmove(g->data+1, g->data, (g->ndata-1)*sizeof(g->data[0]));
    g->data[0] = v;
    g->overflow = 0;
    if(logscale)
        overflow = (v>10*vmax*scale);
    else
        overflow = (v>vmax*scale);
    if(overflow && g->overtmp!=nil){
        g->overflow = 1;
        draw(g->overtmp, g->overtmp->r, view, nil, g->overtmp->r.min);
        sprint(buf, "%llud", v);
        string(view, g->overtmp->r.min, display->black, ZP, mediumfont, buf);
    }
}
```

Uses `drawdatum()` 78a, `logscale` 63c, and `scale` 46d 63b.

⟨*function* `readnums` 79a⟩≡ (95)

```
/* read one line of text from buffer and process integers */
int
readnums(Machine *m, int n, uvlong *a, int spanlines)
{
    int i;
    char *p, *q, *ep;

    if(spanlines)
        ep = m->ebufp;
    else
        for(ep=m->bufp; ep<m->ebufp; ep++)
            if(*ep == '\n')
                break;
    p = m->bufp;
    for(i=0; i<n && p<ep; i++){
        while(p<ep && (!isascii(*p) || !isdigit(*p)) && *p!='-')
            p++;
        if(p == ep)
            break;
        a[i] = strtoull(p, &q, 10);
        p = q;
    }
    if(ep < m->ebufp)
        ep++;
    m->bufp = ep;
    return i == n;
}
```

⟨*function* `filter` 79b⟩≡ (95)

```
/* Network on fd1, mount driver on fd0 */
static int
filter(int fd)
{
    int p[2];

    if(pipe(p) < 0){
        fprint(2, "stats: can't pipe: %r\n");
        killall("pipe");
    }
```

```
	switch(rfork(RFNOWAIT|RFPROC|RFFDG)) {
	case -1:
		sysfatal("rfork record module");
	case 0:
		dup(fd, 1);
		close(fd);
		dup(p[0], 0);
		close(p[0]);
		close(p[1]);
		execl("/bin/aux/fcall", "fcall", nil);
		fprint(2, "stats: can't exec fcall: %r\n");
		killall("fcall");
	default:
		close(fd);
		close(p[0]);
	}
	return p[1];
}
```

Uses killall() 75a.

⟨*function* connect9fs 80a⟩≡ (95)

```
/*
 * 9fs
 */
int
connect9fs(char *addr)
{
	char dir[256], *na;
	int fd;

	fprint(2, "connect9fs...");
	na = netmkaddr(addr, 0, "9fs");

	fprint(2, "dial %s...", na);
	if((fd = dial(na, 0, dir, 0)) < 0)
		return -1;

	fprint(2, "dir %s...", dir);
//	if(strstr(dir, "tcp"))
//		fd = filter(fd);
	return fd;
}
```

⟨*function* old9p 80b⟩≡ (95)

```
int
old9p(int fd)
{
	int p[2];

	if(pipe(p) < 0)
		return -1;

	switch(rfork(RFPROC|RFFDG|RFNAMEG)) {
	case -1:
		return -1;
	case 0:
		if(fd != 1){
			dup(fd, 1);
			close(fd);
		}
```

```
        if(p[0] != 0){
            dup(p[0], 0);
            close(p[0]);
        }
        close(p[1]);
        if(0){
            fd = open("/sys/log/cpu", OWRITE);
            if(fd != 2){
                dup(fd, 2);
                close(fd);
            }
            execl("/bin/srvold9p", "srvold9p", "-ds", nil);
        } else
            execl("/bin/srvold9p", "srvold9p", "-s", nil);
        return -1;
    default:
        close(fd);
        close(p[0]);
    }
    return p[1];
}
```

⟨*function* connectexportfs 81⟩≡ (95)

```
/*
 * exportfs
 */
int
connectexportfs(char *addr)
{
    char buf[ERRMAX], dir[256], *na;
    int fd, n;
    char *tree;
    AuthInfo *ai;

    tree = "/";
    na = netmkaddr(addr, 0, "exportfs");
    if((fd = dial(na, 0, dir, 0)) < 0)
        return -1;

    ai = auth_proxy(fd, auth_getkey, "proto=p9any role=client");
    if(ai == nil)
        return -1;

    n = write(fd, tree, strlen(tree));
    if(n < 0){
        close(fd);
        return -1;
    }

    strcpy(buf, "can't read tree");
    n = read(fd, buf, sizeof buf - 1);
    if(n!=2 || buf[0]!='O' || buf[1]!='K'){
        buf[sizeof buf - 1] = '\0';
        werrstr("bad remote tree: %s\n", buf);
        close(fd);
        return -1;
    }

//  if(strstr(dir, "tcp"))
//    fd = filter(fd);
```

```
    if(oldsystem)
        return old9p(fd);

    return fd;
}
```
Uses old9p() 80b and oldsystem 63e.

⟨*function* readswap 82a⟩≡ (95)
```
int
readswap(Machine *m, uvlong *a)
{
    if(strstr(m->buf, "memory\n")){
        /* new /dev/swap - skip first 3 numbers */
        if(!readnums(m, 7, a, 1))
            return 0;
        a[0] = a[3];
        a[1] = a[4];
        a[2] = a[5];
        a[3] = a[6];
        return 1;
    }
    return readnums(m, nelem(m->devswap), a, 0);
}
```
Uses readnums() 79a.

⟨*function* shortname 82b⟩≡ (95)
```
char*
shortname(char *s)
{
    char *p, *e;

    p = estrdup(s);
    e = strchr(p, '.');
    if(e)
        *e = 0;
    return p;
}
```
Uses estrdup() 75d.

⟨*function* ilog10 82c⟩≡ (95)
```
int
ilog10(uvlong j)
{
    int i;

    for(i = 0; j >= 10; i++)
        j /= 10;
    return i;
}
```

⟨*function* initmach 82d⟩≡ (95)
```
int
initmach(Machine *m, char *name)
{
    int n, fd;
    uvlong a[MAXNUM];
    char *p, mpt[256], buf[256];
```

```
p = strchr(name, '!');
if(p)
    p++;
else
    p = name;
m->name = estrdup(p);
m->shortname = shortname(p);
m->remote = (strcmp(p, mysysname) != 0);
if(m->remote == 0)
    strcpy(mpt, "");
else{
    snprint(mpt, sizeof mpt, "/n/%s", p);
    fd = connectexportfs(name);
    if(fd < 0){
        fprint(2, "can't connect to %s: %r\n", name);
        return 0;
    }
    /* BUG? need to use amount() now? */
    if(mount(fd, -1, mpt, MREPL, "") < 0){
        fprint(2, "stats: mount %s on %s failed (%r); trying /n/sid\n", name, mpt);
        strcpy(mpt, "/n/sid");
        if(mount(fd, -1, mpt, MREPL, "") < 0){
            fprint(2, "stats: mount %s on %s failed: %r\n", name, mpt);
            return 0;
        }
    }
}

snprint(buf, sizeof buf, "%s/dev/swap", mpt);
m->swapfd = open(buf, OREAD);
if(loadbuf(m, &m->swapfd) && readswap(m, a))
    memmove(m->devswap, a, sizeof m->devswap);
else{
    m->devswap[Maxswap] = 100;
    m->devswap[Maxmem] = 100;
}

snprint(buf, sizeof buf, "%s/dev/sysstat", mpt);
m->statsfd = open(buf, OREAD);
if(loadbuf(m, &m->statsfd)){
    for(n=0; readnums(m, nelem(m->devsysstat), a, 0); n++)
        ;
    m->nproc = n;
}else
    m->nproc = 1;
m->lgproc = ilog10(m->nproc);

snprint(buf, sizeof buf, "%s/net/ether0/stats", mpt);
m->etherfd = open(buf, OREAD);
if(loadbuf(m, &m->etherfd) && readnums(m, nelem(m->netetherstats), a, 1))
    memmove(m->netetherstats, a, sizeof m->netetherstats);

snprint(buf, sizeof buf, "%s/net/ether0/ifstats", mpt);
m->ifstatsfd = open(buf, OREAD);
if(loadbuf(m, &m->ifstatsfd)){
    /* need to check that this is a wavelan interface */
    if(strncmp(m->buf, "Signal: ", 8) == 0 && readnums(m, nelem(m->netetherifstats), a, 1))
        memmove(m->netetherifstats, a, sizeof m->netetherifstats);
}
```

```
    snprint(buf, sizeof buf, "%s/mnt/apm/battery", mpt);
    m->batteryfd = open(buf, OREAD);
    m->bitsybatfd = -1;
    if(m->batteryfd >= 0){
        if(loadbuf(m, &m->batteryfd) && readnums(m, nelem(m->batterystats), a, 0))
            memmove(m->batterystats, a, sizeof(m->batterystats));
    }else{
        snprint(buf, sizeof buf, "%s/dev/battery", mpt);
        m->bitsybatfd = open(buf, OREAD);
        if(loadbuf(m, &m->bitsybatfd) && readnums(m, 1, a, 0))
            memmove(m->batterystats, a, sizeof(m->batterystats));
    }
    snprint(buf, sizeof buf, "%s/dev/cputemp", mpt);
    m->tempfd = open(buf, OREAD);
    if(loadbuf(m, &m->tempfd))
        for(n=0; n < nelem(m->temp) && readnums(m, 2, a, 0); n++)
             m->temp[n] = a[0];
    return 1;
}
```

Uses MAXNUM-1 59a, Maxmem-3 59c, Maxswap-5 59c, connectexportfs() 81, estrdup() 75d, ilog10() 82c, loadbuf() 76c, mysysname 62g, readnums() 79a, readswap() 82a, and shortname() 82b.

⟨*global* catchalarm 84a⟩≡ (95)

```
jmp_buf catchalarm;
```

⟨*function* alarmed 84b⟩≡ (95)

```
void
alarmed(void *a, char *s)
{
    if(strcmp(s, "alarm") == 0)
        notejmp(a, catchalarm, 1);
    noted(NDFLT);
}
```

Uses catchalarm 84a.

⟨*function* needswap 84c⟩≡ (95)

```
int
needswap(int init)
{
    return init | present[Mmem] | present[Mswap];
}
```

Uses Mmem-42 61a, Mswap-43 61a, and present 62a.

⟨*function* needstat 84d⟩≡ (95)

```
int
needstat(int init)
{
    return init | present[Mcontext]  | present[Mfault] | present[Mintr] | present[Mload] | present[Midle] |
        present[Minintr] | present[Msyscall] | present[Mtlbmiss] | present[Mtlbpurge];
}
```

Uses Mcontext-32 61a, Mfault-37 61a, Midle-38 61a, Minintr-39 61a, Mintr-40 61a, Mload-41 61a, Msyscall-44 61a, Mtlbmiss-45 61a, Mtlbpurge-46 61a, and present 62a.

⟨*function* needether 84e⟩≡ (95)

```
int
needether(int init)
{
    return init | present[Mether] | present[Metherin] | present[Metherout] | present[Methererr];
}
```

Uses Mether-33 61a, Methererr-34 61a, Metherin-35 61a, Metherout-36 61a, and present 62a.

⟨*function* needbattery 85a⟩≡ (95)

```
int
needbattery(int init)
{
    return init | present[Mbattery];
}
```

Uses Mbattery-31 61a and present 62a.

⟨*function* needsignal 85b⟩≡ (95)

```
int
needsignal(int init)
{
    return init | present[Msignal];
}
```

Uses Msignal-47 61a and present 62a.

⟨*function* needtemp 85c⟩≡ (95)

```
int
needtemp(int init)
{
    return init | present[Mtemp];
}
```

Uses Mtemp-48 61a and present 62a.

⟨*function* readmach 85d⟩≡ (95)

```
void
readmach(Machine *m, int init)
{
    int n, i;
    uvlong a[nelem(m->devsysstat)];
    char buf[32];

    if(m->remote && (m->disable || setjmp(catchalarm))){
        if (m->disable++ >= 5)
            m->disable = 0; /* give it another chance */
        memmove(m->devsysstat, m->prevsysstat, sizeof m->devsysstat);
        memmove(m->netetherstats, m->prevetherstats, sizeof m->netetherstats);
        return;
    }
    snprint(buf, sizeof buf, "%s", m->name);
    if (strcmp(m->name, buf) != 0){
        free(m->name);
        m->name = estrdup(buf);
        free(m->shortname);
        m->shortname = shortname(buf);
        if(display != nil) /* else we're still initializing */
            eresized(0);
    }
    if(m->remote){
        notify(alarmed);
        alarm(5000);
    }
    if(needswap(init) && loadbuf(m, &m->swapfd) && readswap(m, a))
        memmove(m->devswap, a, sizeof m->devswap);
    if(needstat(init) && loadbuf(m, &m->statsfd)){
        memmove(m->prevsysstat, m->devsysstat, sizeof m->devsysstat);
        memset(m->devsysstat, 0, sizeof m->devsysstat);
        for(n=0; n<m->nproc && readnums(m, nelem(m->devsysstat), a, 0); n++)
            for(i=0; i<nelem(m->devsysstat); i++)
```

```
                m->devsysstat[i] += a[i];
    }
    if(needether(init) && loadbuf(m, &m->etherfd) && readnums(m, nelem(m->netetherstats), a, 1)){
        memmove(m->prevetherstats, m->netetherstats, sizeof m->netetherstats);
        memmove(m->netetherstats, a, sizeof m->netetherstats);
    }
    if(needsignal(init) && loadbuf(m, &m->ifstatsfd) && strncmp(m->buf, "Signal: ", 8)==0 && readnums(m, nelem(
        memmove(m->netetherifstats, a, sizeof m->netetherifstats);
    }
    if(needbattery(init) && loadbuf(m, &m->batteryfd) && readnums(m, nelem(m->batterystats), a, 0))
        memmove(m->batterystats, a, sizeof(m->batterystats));
    if(needbattery(init) && loadbuf(m, &m->bitsybatfd) && readnums(m, 1, a, 0))
        memmove(m->batterystats, a, sizeof(m->batterystats));
    if(needtemp(init) && loadbuf(m, &m->tempfd))
        for(n=0; n < nelem(m->temp) && readnums(m, 2, a, 0); n++)
             m->temp[n] = a[0];
    if(m->remote){
        alarm(0);
        notify(nil);
    }
}
```

Uses alarmed() 84b, catchalarm 84a, eresized() 94a, estrdup() 75d, loadbuf() 76c, needbattery() 85a, needether() 84e, needsignal() 85b, needstat() 84d, needswap() 84c, needtemp() 85c, readnums() 79a, readswap() 82a, and shortname() 82b.

⟨*function* memval 86a⟩≡ (95)

```
void
memval(Machine *m, uvlong *v, uvlong *vmax, int)
{
    *v = m->devswap[Mem];
    *vmax = m->devswap[Maxmem];
}
```

Uses Maxmem-3 59c and Mem-2 59c.

⟨*function* swapval 86b⟩≡ (95)

```
void
swapval(Machine *m, uvlong *v, uvlong *vmax, int)
{
    *v = m->devswap[Swap];
    *vmax = m->devswap[Maxswap];
}
```

Uses Maxswap-5 59c and Swap-4 59c.

⟨*function* contextval 86c⟩≡ (95)

```
void
contextval(Machine *m, uvlong *v, uvlong *vmax, int init)
{
    *v = m->devsysstat[Context]-m->prevsysstat[Context];
    *vmax = sleeptime*m->nproc;
    if(init)
        *vmax = sleeptime;
}
```

Uses Context-7 59c and sleeptime 63f.

⟨*function* intrval 86d⟩≡ (95)

```
/*
 * bug: need to factor in HZ
 */
void
intrval(Machine *m, uvlong *v, uvlong *vmax, int init)
```

```
{
    *v = m->devsysstat[Interrupt]-m->prevsysstat[Interrupt];
    *vmax = sleeptime*m->nproc*10;
    if(init)
        *vmax = sleeptime*10;
}
```

Uses Interrupt-8 59c and sleeptime 63f.

⟨*function* syscallval 87a⟩≡ (95)

```
void
syscallval(Machine *m, uvlong *v, uvlong *vmax, int init)
{
    *v = m->devsysstat[Syscall]-m->prevsysstat[Syscall];
    *vmax = sleeptime*m->nproc;
    if(init)
        *vmax = sleeptime;
}
```

Uses Syscall-9 59c and sleeptime 63f.

⟨*function* faultval 87b⟩≡ (95)

```
void
faultval(Machine *m, uvlong *v, uvlong *vmax, int init)
{
    *v = m->devsysstat[Fault]-m->prevsysstat[Fault];
    *vmax = sleeptime*m->nproc;
    if(init)
        *vmax = sleeptime;
}
```

Uses Fault-10 59c and sleeptime 63f.

⟨*function* tlbmissval 87c⟩≡ (95)

```
void
tlbmissval(Machine *m, uvlong *v, uvlong *vmax, int init)
{
    *v = m->devsysstat[TLBfault]-m->prevsysstat[TLBfault];
    *vmax = (sleeptime/1000)*10*m->nproc;
    if(init)
        *vmax = (sleeptime/1000)*10;
}
```

Uses TLBfault-11 59c and sleeptime 63f.

⟨*function* tlbpurgeval 87d⟩≡ (95)

```
void
tlbpurgeval(Machine *m, uvlong *v, uvlong *vmax, int init)
{
    *v = m->devsysstat[TLBpurge]-m->prevsysstat[TLBpurge];
    *vmax = (sleeptime/1000)*10*m->nproc;
    if(init)
        *vmax = (sleeptime/1000)*10;
}
```

Uses TLBpurge-12 59c and sleeptime 63f.

⟨*function* loadval 87e⟩≡ (95)

```
void
loadval(Machine *m, uvlong *v, uvlong *vmax, int init)
{
    *v = m->devsysstat[Load];
    *vmax = 1000*m->nproc;
    if(init)
```

```
        *vmax = 1000;
}
```
Uses `Load-13` 59c.

⟨*function* `idleval` 88a⟩≡ (95)
```
void
idleval(Machine *m, uvlong *v, uvlong *vmax, int)
{
    *v = m->devsysstat[Idle]/m->nproc;
    *vmax = 100;
}
```
Uses `Idle-14` 59c.

⟨*function* `inintrval` 88b⟩≡ (95)
```
void
inintrval(Machine *m, uvlong *v, uvlong *vmax, int)
{
    *v = m->devsysstat[InIntr]/m->nproc;
    *vmax = 100;
}
```
Uses `InIntr-15` 59c.

⟨*function* `etherval` 88c⟩≡ (95)
```
void
etherval(Machine *m, uvlong *v, uvlong *vmax, int init)
{
    *v = m->netetherstats[In]-m->prevetherstats[In] + m->netetherstats[Out]-m->prevetherstats[Out];
    *vmax = sleeptime*m->nproc;
    if(init)
        *vmax = sleeptime;
}
```
Uses `In-16` 59c, `Out-18` 59c, and `sleeptime` 63f.

⟨*function* `etherinval` 88d⟩≡ (95)
```
void
etherinval(Machine *m, uvlong *v, uvlong *vmax, int init)
{
    *v = m->netetherstats[In]-m->prevetherstats[In];
    *vmax = sleeptime*m->nproc;
    if(init)
        *vmax = sleeptime;
}
```
Uses `In-16` 59c and `sleeptime` 63f.

⟨*function* `etheroutval` 88e⟩≡ (95)
```
void
etheroutval(Machine *m, uvlong *v, uvlong *vmax, int init)
{
    *v = m->netetherstats[Out]-m->prevetherstats[Out];
    *vmax = sleeptime*m->nproc;
    if(init)
        *vmax = sleeptime;
}
```
Uses `Out-18` 59c and `sleeptime` 63f.

⟨*function* ethererrval 89a⟩≡ (95)

```
void
ethererrval(Machine *m, uvlong *v, uvlong *vmax, int init)
{
    int i;

    *v = 0;
    for(i=Err0; i<nelem(m->netetherstats); i++)
        *v += m->netetherstats[i];
    *vmax = (sleeptime/1000)*10*m->nproc;
    if(init)
        *vmax = (sleeptime/1000)*10;
}
```

Uses Err0-19 59c and sleeptime 63f.

⟨*function* batteryval 89b⟩≡ (95)

```
void
batteryval(Machine *m, uvlong *v, uvlong *vmax, int)
{
    *v = m->batterystats[0];
    if(m->bitsybatfd >= 0)
        *vmax = 184;  // at least on my bitsy...
    else
        *vmax = 100;
}
```

⟨*function* signalval 89c⟩≡ (95)

```
void
signalval(Machine *m, uvlong *v, uvlong *vmax, int)
{
    ulong l;

    *vmax = sleeptime;
    l = m->netetherifstats[0];
    /*
     * Range is seen to be from about -45 (strong) to -95 (weak); rescale
     */
    if(l == 0){ /* probably not present */
        *v = 0;
        return;
    }
    *v = 20*(l+95);
}
```

Uses sleeptime 63f.

⟨*function* tempval 89d⟩≡ (95)

```
void
tempval(Machine *m, uvlong *v, uvlong *vmax, int)
{
    ulong l;

    *vmax = sleeptime;
    l = m->temp[0];
    if(l == ~0 || l == 0)
        *v = 0;
    else
        *v = (l-20)*27;
}
```

Uses sleeptime 63f.

⟨*function* addgraph 90a⟩≡ (95)

```
void
addgraph(int n)
{
    Graph *g, *ograph;
    int i, j;
    static int nadd;

    if(n > nelem(menu2str))
        abort();
    /* avoid two adjacent graphs of same color */
    if(ngraph>0 && graph[ngraph-1].colindex==nadd%Ncolor)
        nadd++;
    ograph = graph;
    graph = emalloc(nmach*(ngraph+1)*sizeof(Graph));
    for(i=0; i<nmach; i++)
        for(j=0; j<ngraph; j++)
            graph[i*(ngraph+1)+j] = ograph[i*ngraph+j];
    free(ograph);
    ngraph++;
    for(i=0; i<nmach; i++){
        g = &graph[i*ngraph+(ngraph-1)];
        memset(g, 0, sizeof(Graph));
        g->label = menu2str[n]+Opwid;
        g->newvalue = newvaluefn[n];
        g->update = update1; /* no other update functions yet */
        g->mach = &mach[i];
        g->colindex = nadd%Ncolor;
    }
    present[n] = 1;
    nadd++;
}
```

Uses `Ncolor-23` 60c, `Opwid-27` 60c, `emalloc()` 75b, `mach` 62e, `menu2str` 61b, `newvaluefn` 62b, `ngraph` 63a, `nmach` 62k, `present` 62a, and `update1()` 78c.

⟨*function* dropgraph 90b⟩≡ (95)

```
void
dropgraph(int which)
{
    Graph *ograph;
    int i, j, n;

    if(which > nelem(menu2str))
        abort();
    /* convert n to index in graph table */
    n = -1;
    for(i=0; i<ngraph; i++)
        if(strcmp(menu2str[which]+Opwid, graph[i].label) == 0){
            n = i;
            break;
        }
    if(n < 0){
        fprint(2, "stats: internal error can't drop graph\n");
        killall("error");
    }
    ograph = graph;
    graph = emalloc(nmach*(ngraph-1)*sizeof(Graph));
    for(i=0; i<nmach; i++){
        for(j=0; j<n; j++)
            graph[i*(ngraph-1)+j] = ograph[i*ngraph+j];
```

```
        free(ograph[i*ngraph+j].data);
        freeimage(ograph[i*ngraph+j].overtmp);
        for(j++; j<ngraph; j++)
            graph[i*(ngraph-1)+j-1] = ograph[i*ngraph+j];
    }
    free(ograph);
    ngraph--;
    present[which] = 0;
}
```

Uses `Opwid-27` 60c, `emalloc()` 75b, `killall()` 75a, `menu2str` 61b, `ngraph` 63a, `nmach` 62k, and `present` 62a.

⟨*function* `addmachine` 91a⟩≡ (95)

```
int
addmachine(char *name)
{
    if(ngraph > 0){
        fprint(2, "stats: internal error: ngraph>0 in addmachine()\n");
        usage();
    }
    if(mach == nil)
        nmach = 0; /* a little dance to get us started with local machine by default */
    mach = erealloc(mach, (nmach+1)*sizeof(Machine));
    memset(mach+nmach, 0, sizeof(Machine));
    if (initmach(mach+nmach, name)){
        nmach++;
        return 1;
    } else
        return 0;
}
```

Uses `erealloc()` 75c, `initmach()` 82d, `mach` 62e, `ngraph` 63a, and `nmach` 62k.

⟨*function* `labelstrs` 91b⟩≡ (95)

```
void
labelstrs(Graph *g, char strs[Nlab][Lablen], int *np)
{
    int j;
    uvlong v, vmax;

    g->newvalue(g->mach, &v, &vmax, 1);
    if(logscale){
        for(j=1; j<=2; j++)
            sprint(strs[j-1], "%g", scale*pow(10., j)*(double)vmax/100.);
        *np = 2;
    }else{
        for(j=1; j<=3; j++)
            sprint(strs[j-1], "%g", scale*(double)j*(double)vmax/4.0);
        *np = 3;
    }
}
```

Uses `Lablen-29` 60c, `Nlab-28` 60c, `logscale` 63c, and `scale` 46d 63b.

⟨*function* `labelwidth` 91c⟩≡ (95)

```
int
labelwidth(void)
{
    int i, j, n, w, maxw;
    char strs[Nlab][Lablen];

    maxw = 0;
```

```
    for(i=0; i<ngraph; i++){
        /* choose value for rightmost graph */
        labelstrs(&graph[ngraph*(nmach-1)+i], strs, &n);
        for(j=0; j<n; j++){
            w = stringwidth(mediumfont, strs[j]);
            if(w > maxw)
                maxw = w;
        }
    }
    return maxw;
}
```

Uses Lablen-29 60c, Nlab-28 60c, labelstrs() 91b, ngraph 63a, and nmach 62k.

⟨*function* resize 92⟩≡ (95)

```
void
resize(void)
{
    int i, j, k, n, startx, starty, x, y, dx, dy, ly, ondata, maxx, wid, nlab;
    Graph *g;
    Rectangle machr, r;
    uvlong v, vmax;
    char buf[128], labs[Nlab][Lablen];

    draw(view, view->r, display->white, nil, ZP);

    /* label left edge */
    x = view->r.min.x;
    y = view->r.min.y + Labspace+mediumfont->height+Labspace;
    dy = (view->r.max.y - y)/ngraph;
    dx = Labspace+stringwidth(mediumfont, "0")+Labspace;
    startx = x+dx+1;
    starty = y;
    for(i=0; i<ngraph; i++,y+=dy){
        draw(view, Rect(x, y-1, view->r.max.x, y), display->black, nil, ZP);
        draw(view, Rect(x, y, x+dx, view->r.max.y), cols[graph[i].colindex][0], nil, paritypt(x));
        label(Pt(x, y), dy, graph[i].label);
        draw(view, Rect(x+dx, y, x+dx+1, view->r.max.y), cols[graph[i].colindex][2], nil, ZP);
    }

    /* label top edge */
    dx = (view->r.max.x - startx)/nmach;
    for(x=startx, i=0; i<nmach; i++,x+=dx){
        draw(view, Rect(x-1, starty-1, x, view->r.max.y), display->black, nil, ZP);
        j = dx/stringwidth(mediumfont, "0");
        n = mach[i].nproc;
        if(n>1 && j>=1+3+mach[i].lgproc){ /* first char of name + (n) */
            j -= 3+mach[i].lgproc;
            if(j <= 0)
                j = 1;
            snprint(buf, sizeof buf, "%.*s(%d)", j, mach[i].shortname, n);
        }else
            snprint(buf, sizeof buf, "%.*s", j, mach[i].shortname);
        string(view, Pt(x+Labspace, view->r.min.y + Labspace), display->black, ZP, mediumfont, buf);
    }

    maxx = view->r.max.x;

    /* label right, if requested */
    if(ylabels && dy>Nlab*(mediumfont->height+1)){
        wid = labelwidth();
```

```
		if(wid < (maxx-startx)-30){
			/* else there's not enough room */
			maxx -= 1+Lx+wid;
			draw(view, Rect(maxx, starty, maxx+1, view->r.max.y), display->black, nil, ZP);
			y = starty;
			for(j=0; j<ngraph; j++, y+=dy){
				/* choose value for rightmost graph */
				g = &graph[ngraph*(nmach-1)+j];
				labelstrs(g, labs, &nlab);
				r = Rect(maxx+1, y, view->r.max.x, y+dy-1);
				if(j == ngraph-1)
					r.max.y = view->r.max.y;
				draw(view, r, cols[g->colindex][0], nil, paritypt(r.min.x));
				for(k=0; k<nlab; k++){
					ly = y + (dy*(nlab-k)/(nlab+1));
					draw(view, Rect(maxx+1, ly, maxx+1+Lx, ly+1), display->black, nil, ZP);
					ly -= mediumfont->height/2;
					string(view, Pt(maxx+1+Lx, ly), display->black, ZP, mediumfont, labs[k]);
				}
			}
		}
	}

	/* create graphs */
	for(i=0; i<nmach; i++){
		machr = Rect(startx+i*dx, starty, maxx, view->r.max.y);
		if(i < nmach-1)
			machr.max.x = startx+(i+1)*dx - 1;
		y = starty;
		for(j=0; j<ngraph; j++, y+=dy){
			g = &graph[i*ngraph+j];
			/* allocate data */
			ondata = g->ndata;
			g->ndata = Dx(machr)+1; /* may be too many if label will be drawn here; so what? */
			g->data = erealloc(g->data, g->ndata*sizeof(g->data[0]));
			if(g->ndata > ondata)
				memset(g->data+ondata, 0, (g->ndata-ondata)*sizeof(g->data[0]));
			/* set geometry */
			g->r = machr;
			g->r.min.y = y;
			g->r.max.y = y+dy - 1;
			if(j == ngraph-1)
				g->r.max.y = view->r.max.y;
			draw(view, g->r, cols[g->colindex][0], nil, paritypt(g->r.min.x));
			g->overflow = 0;
			r = g->r;
			r.max.y = r.min.y+mediumfont->height;
			r.max.x = r.min.x+stringwidth(mediumfont, "999999999999");
			freeimage(g->overtmp);
			g->overtmp = nil;
			if(r.max.x <= g->r.max.x)
				g->overtmp = allocimage(display, r, view->chan, 0, -1);
			g->newvalue(g->mach, &v, &vmax, 0);
			redraw(g, vmax);
		}
	}

	flushimage(display, 1);
}
```

Uses Lablen-29 60c, Labspace-25 60c, Lx-30 60c, Nlab-28 60c, erealloc() 75c, label() 77a, labelstrs() 91b, labelwidth() 91c, mach 62e, ngraph 63a, nmach 62k, paritypt() 77b, and ylabels 63d.

⟨*function* eresized 94a⟩≡ (95)
```
void
eresized(int new)
{
    lockdisplay(display);
    if(new && getwindow(display, Refnone) < 0) {
        fprint(2, "stats: can't reattach to window\n");
        killall("reattach");
    }
    resize();
    unlockdisplay(display);
}
```
Uses killall() 75a and resize() 92.

⟨*function* mouseproc 94b⟩≡ (95)
```
void
mouseproc(void)
{
    Mouse mouse;
    int i;

    for(;;){
        mouse = emouse();
        if(mouse.buttons == 4){
            lockdisplay(display);
            for(i=0; i<Nmenu2; i++)
                if(present[i])
                    memmove(menu2str[i], "drop ", Opwid);
                else
                    memmove(menu2str[i], "add  ", Opwid);
            i = emenuhit(3, &mouse, &menu2);
            if(i >= 0){
                if(!present[i])
                    addgraph(i);
                else if(ngraph > 1)
                    dropgraph(i);
                resize();
            }
            unlockdisplay(display);
        }
    }
}
```
Uses Nmenu2-49 61a, Opwid-27 60c, addgraph() 90a, dropgraph() 90b, menu2 61c, menu2str 61b, ngraph 63a, present 62a, and resize() 92.

⟨*function* startproc 94c⟩≡ (95)
```
void
startproc(void (*f)(void), int index)
{
    int pid;

    switch(pid = rfork(RFPROC|RFMEM|RFNOWAIT)){
    case -1:
        fprint(2, "stats: fork failed: %r\n");
        killall("fork failed");
    case 0:
        f();
```

```
        fprint(2, "stats: %s process exits\n", procnames[index]);
        if(index >= 0)
            killall("process died");
        exits(nil);
    }
    if(index >= 0)
        pids[index] = pid;
}
```

Uses killall() 75a, pids 62i, and procnames 63g.

⟨misc/stats.c 95⟩≡

```
#include <u.h>
#include <libc.h>
#include <ctype.h>
#include <auth.h>
#include <fcall.h>
#include <draw.h>
#include <window.h>
#include <event.h>

⟨constant MAXNUM 59a⟩

typedef struct Graph Graph;
typedef struct Machine Machine;

⟨struct Graph 59b⟩

⟨enum _anon_ (misc/stats.c) 59c⟩

⟨struct Machine 60a⟩

⟨enum _anon_ (misc/stats.c)2 60b⟩

⟨enum _anon_ (misc/stats.c)3 60c⟩

⟨enum Menu2 61a⟩

⟨global menu2str 61b⟩


void contextval(Machine*, uvlong*, uvlong*, int),
    etherval(Machine*, uvlong*, uvlong*, int),
    ethererrval(Machine*, uvlong*, uvlong*, int),
    etherinval(Machine*, uvlong*, uvlong*, int),
    etheroutval(Machine*, uvlong*, uvlong*, int),
    faultval(Machine*, uvlong*, uvlong*, int),
    intrval(Machine*, uvlong*, uvlong*, int),
    inintrval(Machine*, uvlong*, uvlong*, int),
    loadval(Machine*, uvlong*, uvlong*, int),
    idleval(Machine*, uvlong*, uvlong*, int),
    memval(Machine*, uvlong*, uvlong*, int),
    swapval(Machine*, uvlong*, uvlong*, int),
    syscallval(Machine*, uvlong*, uvlong*, int),
    tlbmissval(Machine*, uvlong*, uvlong*, int),
    tlbpurgeval(Machine*, uvlong*, uvlong*, int),
    batteryval(Machine*, uvlong*, uvlong*, int),
    signalval(Machine*, uvlong*, uvlong*, int),
    tempval(Machine*, uvlong*, uvlong*, int);

⟨global menu2 61c⟩
```

⟨*global* present 62a⟩
⟨*global* newvaluefn 62b⟩

⟨*global cols (misc/stats.c)* 62c⟩
⟨*global* graph 62d⟩
⟨*global* mach 62e⟩
⟨*global* mediumfont 62f⟩
⟨*global* mysysname 62g⟩
⟨*global* argchars 62h⟩
⟨*global* pids 62i⟩
⟨*global* parity 62j⟩
⟨*global* nmach 62k⟩
⟨*global* ngraph 63a⟩
⟨*global* scale 63b⟩
⟨*global* logscale 63c⟩
⟨*global* ylabels 63d⟩
⟨*global* oldsystem 63e⟩
⟨*global* sleeptime 63f⟩

⟨*global* procnames 63g⟩

⟨*function* killall 75a⟩

⟨*function* emalloc 75b⟩

⟨*function* erealloc 75c⟩

⟨*function* estrdup 75d⟩

⟨*function mkcol (misc/stats.c)* 76a⟩

⟨*function colinit (misc/stats.c)* 76b⟩

⟨*function* loadbuf 76c⟩

⟨*function* label 77a⟩

⟨*function* paritypt 77b⟩

⟨*function* datapoint 77c⟩

⟨*function* drawdatum 78a⟩

⟨*function redraw (misc/stats.c)* 78b⟩

⟨*function* update1 78c⟩

⟨*function* readnums 79a⟩

⟨*function* filter 79b⟩

⟨*function* connect9fs 80a⟩

⟨*function* old9p 80b⟩

⟨*function* connectexportfs 81⟩

⟨*function* readswap 82a⟩

⟨*function* `shortname` 82b⟩

⟨*function* `ilog10` 82c⟩

⟨*function* `initmach` 82d⟩

⟨*global* `catchalarm` 84a⟩

⟨*function* `alarmed` 84b⟩

⟨*function* `needswap` 84c⟩

⟨*function* `needstat` 84d⟩

⟨*function* `needether` 84e⟩

⟨*function* `needbattery` 85a⟩

⟨*function* `needsignal` 85b⟩

⟨*function* `needtemp` 85c⟩

⟨*function* `readmach` 85d⟩

⟨*function* `memval` 86a⟩

⟨*function* `swapval` 86b⟩

⟨*function* `contextval` 86c⟩

⟨*function* `intrval` 86d⟩

⟨*function* `syscallval` 87a⟩

⟨*function* `faultval` 87b⟩

⟨*function* `tlbmissval` 87c⟩

⟨*function* `tlbpurgeval` 87d⟩

⟨*function* `loadval` 87e⟩

⟨*function* `idleval` 88a⟩

⟨*function* `inintrval` 88b⟩

⟨*function* `etherval` 88c⟩

⟨*function* `etherinval` 88d⟩

⟨*function* `etheroutval` 88e⟩

⟨*function* `ethererrval` 89a⟩

⟨*function* `batteryval` 89b⟩

⟨*function* `signalval` 89c⟩

⟨*function* tempval 89d⟩

⟨*function usage (misc/stats.c)* 11c⟩

⟨*function* addgraph 90a⟩

⟨*function* dropgraph 90b⟩

⟨*function* addmachine 91a⟩

⟨*function* labelstrs 91b⟩

⟨*function* labelwidth 91c⟩

⟨*function* resize 92⟩

⟨*function* eresized 94a⟩

⟨*function* mouseproc 94b⟩

⟨*function* startproc 94c⟩

⟨*function main (misc/stats.c)* 63h⟩

Uses Graph 59b and Machine 60a.

A.2 iostats/

A.2.1 iostats/statfs.h

⟨*constant* DEBUGFILE 98a⟩≡ (100d)

```
/*
 * statfs.h - definitions for statistic gathering file server
 */

#define DEBUGFILE "iostats.out"
```

⟨*constant* DONESTR 98b⟩≡ (100d)

```
#define DONESTR  "done"
```

⟨*constant* DEBUG 98c⟩≡ (100d)

```
#define DEBUG  if(dbg)fprint
```

⟨*constant* MAXPROC 98d⟩≡ (100d)

```
#define MAXPROC  16
```

⟨*constant* FHASHSIZE 98e⟩≡ (100d)

```
#define FHASHSIZE 64
```

⟨*function* fidhash 98f⟩≡ (100d)

```
#define fidhash(s) fhash[s%FHASHSIZE]
```

⟨*enum* _anon_ (iostats/statfs.h) 98g⟩≡ (100d)

```
enum{
    Maxfdata = 8192, /* max size of data in 9P message */
    Maxrpc   = 20000,/* number of RPCs we'll log */
};
```

⟨*struct* Frec 99a⟩≡ (100d)
```
struct Frec
{
    Frec *next;
    char *op;
    ulong nread;
    ulong nwrite;
    ulong bread;
    ulong bwrite;
    ulong opens;
};
```

⟨*struct* Rpc 99b⟩≡ (100d)
```
struct Rpc
{
    char *name;
    ulong count;
    vlong time;
    vlong lo;
    vlong hi;
    ulong bin;
    ulong bout;
};
```

⟨*struct* Stats 99c⟩≡ (100d)
```
struct Stats
{
    ulong totread;
    ulong totwrite;
    ulong nrpc;
    ulong nproto;
    Rpc rpc[Maxrpc];
};
```
Uses Maxrpc 98g.

⟨*struct* Fsrpc 99d⟩≡ (100d)
```
struct Fsrpc
{
    int busy;   /* Work buffer has pending rpc to service */
    uintptr pid;   /* Pid of slave process executing the rpc */
    int canint;   /* Interrupt gate */
    int flushtag;  /* Tag on which to reply to flush */
    Fcall work;   /* Plan 9 incoming Fcall */
    uchar buf[IOHDRSZ+Maxfdata]; /* Data buffer */
};
```
Uses Maxfdata 98g.

⟨*struct* Fid 99e⟩≡ (100d)
```
struct Fid
{
    int fid;   /* system fd for i/o */
    File *f;   /* File attached to this fid */
    int mode;
    int nr;   /* fid number */
    Fid *next;   /* hash link */
    ulong nread;
    ulong nwrite;
    ulong bread;
    ulong bwrite;
    vlong offset;   /* for directories */
};
```

⟨*struct* File 100a⟩≡ (100d)
```
struct File
{
    char *name;
    Qid qid;
    int inval;
    File *parent;
    File *child;
    File *childlist;
};
```

⟨*struct* Proc 100b⟩≡ (100d)
```
struct Proc
{
    uintptr pid;
    int busy;
    Proc *next;
};
```

⟨*enum* _anon_ (iostats/statfs.h)2 100c⟩≡ (100d)
```
enum
{
    Nr_workbufs  = 40,
    Dsegpad  = 8192,
    Fidchunk = 1000,
};
```

⟨iostats/statfs.h 100d⟩≡
```
⟨constant DEBUGFILE 98a⟩
⟨constant DONESTR 98b⟩
⟨constant DEBUG 98c⟩
⟨constant MAXPROC 98d⟩
⟨constant FHASHSIZE 98e⟩
⟨function fidhash 98f⟩

⟨enum _anon_ (iostats/statfs.h) 98g⟩

typedef struct Fsrpc Fsrpc;
typedef struct Fid Fid;
typedef struct File File;
typedef struct Proc Proc;
typedef struct Stats Stats;
typedef struct Rpc Rpc;
typedef struct Frec Frec;

⟨struct Frec 99a⟩

⟨struct Rpc 99b⟩

⟨struct Stats 99c⟩

⟨struct Fsrpc 99d⟩

⟨struct Fid 99e⟩

⟨struct File 100a⟩

⟨struct Proc 100b⟩

⟨enum _anon_ (iostats/statfs.h)2 100c⟩
```

```
extern Fsrpc *Workq;
extern int   dbg;
extern File *root;
extern Fid **fhash;
extern Fid *fidfree;
extern int qid;
extern Proc *Proclist;
extern int done;
extern Stats *stats;
extern Frec *frhead;
extern Frec *frtail;
extern int myiounit;

/* File system protocol service procedures */
void Xcreate(Fsrpc*), Xclunk(Fsrpc*);
void Xversion(Fsrpc*), Xauth(Fsrpc*), Xflush(Fsrpc*);
void Xattach(Fsrpc*), Xwalk(Fsrpc*), Xauth(Fsrpc*);
void Xremove(Fsrpc*), Xstat(Fsrpc*), Xwstat(Fsrpc*);
void slave(Fsrpc*);

void reply(Fcall*, Fcall*, char*);
Fid  *getfid(int);
int freefid(int);
Fid *newfid(int);
Fsrpc *getsbuf(void);
void initroot(void);
void fatal(char*);
void makepath(char*, File*, char*);
File *file(File*, char*);
void slaveopen(Fsrpc*);
void slaveread(Fsrpc*);
void slavewrite(Fsrpc*);
void blockingslave(void);
void reopen(Fid *f);
void noteproc(int, char*);
void flushaction(void*, char*);
void catcher(void*, char*);
ulong msec(void);
void fidreport(Fid*);
```
Uses `Fid` 99e, `File` 100a, `Frec` 99a, `Fsrpc` 99d, `Proc` 100b, `Rpc` 99b, and `Stats` 99c.

A.2.2 iostats/globals.c

⟨*global* `Workq` 101a⟩≡ (102h)
```
Fsrpc *Workq;
```

⟨*global* `dbg` 101b⟩≡ (102h)
```
int   dbg;
```

⟨*global* `root` 101c⟩≡ (102h)
```
File *root;
```

⟨*global* `fhash` 101d⟩≡ (102h)
```
Fid **fhash;
```

⟨*global* `fidfree` 101e⟩≡ (102h)
```
Fid *fidfree;
```

⟨*global* qid 102a⟩≡ (102h)

```
int qid;
```

⟨*global* Proclist 102b⟩≡ (102h)

```
Proc *Proclist;
```

⟨*global* done 102c⟩≡ (102h)

```
int done;
```

⟨*global* stats 102d⟩≡ (102h)

```
Stats *stats;
```

⟨*global* frhead 102e⟩≡ (102h)

```
Frec *frhead;
```

⟨*global* frtail 102f⟩≡ (102h)

```
Frec *frtail;
```

⟨*global* myiounit 102g⟩≡ (102h)

```
int myiounit;
```

⟨iostats/globals.c 102h⟩≡

```
#include <u.h>
#include <libc.h>
#include <auth.h>
#include <fcall.h>

#include "statfs.h"

⟨global Workq 101a⟩
⟨global dbg 101b⟩
⟨global root 101c⟩
⟨global fhash 101d⟩
⟨global fidfree 101e⟩
⟨global qid 102a⟩
⟨global Proclist 102b⟩
⟨global done 102c⟩
⟨global stats 102d⟩
⟨global frhead 102e⟩
⟨global frtail 102f⟩
⟨global myiounit 102g⟩
```

A.2.3 iostats/iostats.c

⟨*global* fcalls 102i⟩≡ (113)

```
void (*fcalls[])(Fsrpc*) =
{
     [Tversion] Xversion,
     [Tauth] Xauth,
     [Tflush] Xflush,
     [Tattach] Xattach,
     [Twalk]  Xwalk,
     [Topen]  slave,
     [Tcreate] Xcreate,
     [Tclunk] Xclunk,
     [Tread]  slave,
     [Twrite] slave,
     [Tremove] Xremove,
     [Tstat]  Xstat,
```

```
    [Twstat] Xwstat,
};
```

Uses `Xattach()` 116a, `Xauth()` 115b, `Xclunk()` 118a, `Xcreate()` 119, `Xflush()` 115c, `Xremove()` 120a, `Xstat()` 118b, `Xversion()` 115a, `Xwalk()` 116b, `Xwstat()` 120b, and `slave()` 121.

⟨*global* p 103a⟩≡ (113)

```
int p[2];
```

⟨*function usage (iostats/iostats.c)* 103b⟩≡ (113)

```
static void
usage(void)
{
    fprint(2, "usage: iostats [-d] [-f debugfile] cmds [args ...]\n");
    exits("usage");
}
```

⟨*function main (iostats/iostats.c)* 103c⟩≡ (113)

```
void
main(int argc, char **argv)
{
    Fsrpc *r;
    Rpc *rpc;
    Proc *m;
    Frec *fr;
    Fid *fid;
    ulong ttime;
    char *dbfile, *s;
    char buf[128];
    float brpsec, bwpsec, bppsec;
    int type, cpid, fspid, n;

    dbfile = DEBUGFILE;

    ARGBEGIN{
    case 'd':
        dbg++;
        break;
    case 'f':
        dbfile = ARGF();
        break;
    default:
        usage();
    }ARGEND

    if(argc == 0)
        usage();

    if(dbg) {
        close(2);
        create(dbfile, OWRITE, 0666);
    }

    if(pipe(p) < 0)
        fatal("pipe");

    switch(cpid = fork()) {
    case -1:
        fatal("fork");
    case 0:
        close(p[1]);
```

```
		if(getwd(buf, sizeof(buf)) == 0)
			fatal("no working directory");

		rfork(RFENVG|RFNAMEG|RFNOTEG);
		if(mount(p[0], -1, "/", MREPL, "") < 0)
			fatal("mount /");

		bind("#c/pid", "/dev/pid", MREPL);
		bind("#e", "/env", MREPL|MCREATE);
		close(0);
		close(1);
		close(2);
		open("/fd/0", OREAD);
		open("/fd/1", OWRITE);
		open("/fd/2", OWRITE);

		if(chdir(buf) < 0)
			fatal("chdir");

		runprog(argv);
	default:
		close(p[0]);
	}

	switch(fspid = fork()) {
	default:
		while(cpid != waitpid())
			;
		postnote(PNPROC, fspid, DONESTR);
		while(fspid != waitpid())
			;
		exits(0);
	case -1:
		fatal("fork");
	case 0:
		break;
	}

	/* Allocate work queues in shared memory */
	malloc(Dsegpad);
	Workq = malloc(sizeof(Fsrpc)*Nr_workbufs);
	stats = malloc(sizeof(Stats));
	fhash = mallocz(sizeof(Fid*)*FHASHSIZE, 1);

	if(Workq == 0 || fhash == 0 || stats == 0)
		fatal("no initial memory");

	memset(Workq, 0, sizeof(Fsrpc)*Nr_workbufs);
	memset(stats, 0, sizeof(Stats));

	stats->rpc[Tversion].name = "version";
	stats->rpc[Tauth].name = "auth";
	stats->rpc[Tflush].name = "flush";
	stats->rpc[Tattach].name = "attach";
	stats->rpc[Twalk].name = "walk";
	stats->rpc[Topen].name = "open";
	stats->rpc[Tcreate].name = "create";
	stats->rpc[Tclunk].name = "clunk";
	stats->rpc[Tread].name = "read";
	stats->rpc[Twrite].name = "write";
```

```
stats->rpc[Tremove].name = "remove";
stats->rpc[Tstat].name = "stat";
stats->rpc[Twstat].name = "wstat";

for(n = 0; n < Maxrpc; n++)
    stats->rpc[n].lo = 10000000000LL;

fmtinstall('M', dirmodefmt);
fmtinstall('D', dirfmt);
fmtinstall('F', fcallfmt);

if(chdir("/") < 0)
    fatal("chdir");

initroot();

DEBUG(2, "statfs: %s\n", buf);

notify(catcher);

for(;;) {
    r = getsbuf();
    if(r == 0)
        fatal("Out of service buffers");

    n = read9pmsg(p[1], r->buf, sizeof(r->buf));
    if(done)
        break;
    if(n < 0)
        fatal("read server");

    if(convM2S(r->buf, n, &r->work) == 0)
        fatal("format error");

    stats->nrpc++;
    stats->nproto += n;

    DEBUG(2, "%F\n", &r->work);

    type = r->work.type;
    rpc = &stats->rpc[type];
    rpc->count++;
    rpc->bin += n;
    (fcalls[type])(r);
}

/* Clear away the slave children */
for(m = Proclist; m; m = m->next)
    postnote(PNPROC, m->pid, "kill");

rpc = &stats->rpc[Tread];
brpsec = (float)stats->totread / (((float)rpc->time/1e9)+.000001);

rpc = &stats->rpc[Twrite];
bwpsec = (float)stats->totwrite / (((float)rpc->time/1e9)+.000001);

ttime = 0;
for(n = 0; n < Maxrpc; n++) {
    rpc = &stats->rpc[n];
    if(rpc->count == 0)
```

```
            continue;
        ttime += rpc->time;
    }

    bppsec = (float)stats->nproto / ((ttime/1e9)+.000001);

    fprint(2, "\nread      %lud bytes, %g Kb/sec\n", stats->totread, brpsec/1024.0);
    fprint(2, "write     %lud bytes, %g Kb/sec\n", stats->totwrite, bwpsec/1024.0);
    fprint(2, "protocol  %lud bytes, %g Kb/sec\n", stats->nproto, bppsec/1024.0);
    fprint(2, "rpc       %lud count\n\n", stats->nrpc);

    fprint(2, "%-10s %5s %5s %5s %5s %5s          T       R\n",
          "Message", "Count", "Low", "High", "Time", "Averg");

    for(n = 0; n < Maxrpc; n++) {
        rpc = &stats->rpc[n];
        if(rpc->count == 0)
            continue;
        fprint(2, "%-10s %5lud %5llud %5llud %5llud %5llud ms %8lud %8lud bytes\n",
            rpc->name,
            rpc->count,
            rpc->lo/1000000,
            rpc->hi/1000000,
            rpc->time/1000000,
            rpc->time/1000000/rpc->count,
            rpc->bin,
            rpc->bout);
    }

    for(n = 0; n < FHASHSIZE; n++)
        for(fid = fhash[n]; fid; fid = fid->next)
            if(fid->nread || fid->nwrite)
                fidreport(fid);
    if(frhead == 0)
        exits(0);

    fprint(2, "\nOpens    Reads  (bytes)   Writes  (bytes) File\n");
    for(fr = frhead; fr; fr = fr->next) {
        s = fr->op;
        if(*s) {
            if(strcmp(s, "/fd/0") == 0)
                s = "(stdin)";
            else
            if(strcmp(s, "/fd/1") == 0)
                s = "(stdout)";
            else
            if(strcmp(s, "/fd/2") == 0)
                s = "(stderr)";
        }
        else
            s = "/.";

        fprint(2, "%5lud %8lud %8lud %8lud %8lud %s\n", fr->opens, fr->nread, fr->bread,
                            fr->nwrite, fr->bwrite, s);
    }

    exits(0);
}
```

⟨*function* reply 106⟩≡ (113)

```
void
reply(Fcall *r, Fcall *t, char *err)
{
    uchar data[IOHDRSZ+Maxfdata];
    int n;

    t->tag = r->tag;
    t->fid = r->fid;
    if(err) {
        t->type = Rerror;
        t->ename = err;
    }
    else
        t->type = r->type + 1;

    DEBUG(2, "\t%F\n", t);

    n = convS2M(t, data, sizeof data);
    if(write(p[1], data, n)!=n)
        fatal("mount write");
    stats->nproto += n;
    stats->rpc[t->type-1].bout += n;
}
```

Uses `DEBUG` 98c, `Maxfdata` 98g, `fatal()` 111a, `p` 103a, and `stats` 102d.

⟨*function* `getfid` 107a⟩≡ (113)

```
Fid *
getfid(int nr)
{
    Fid *f;

    for(f = fidhash(nr); f; f = f->next)
        if(f->nr == nr)
            return f;

    return 0;
}
```

Uses `fidhash` 98f.

⟨*function* `freefid` 107b⟩≡ (113)

```
int
freefid(int nr)
{
    Fid *f, **l;

    l = &fidhash(nr);
    for(f = *l; f; f = f->next) {
        if(f->nr == nr) {
            *l = f->next;
            f->next = fidfree;
            fidfree = f;
            return 1;
        }
        l = &f->next;
    }

    return 0;
}
```

Uses `fidfree` 101e and `fidhash` 98f.

⟨*function* newfid 108a⟩≡ (113)

```
Fid *
newfid(int nr)
{
    Fid *new, **l;
    int i;

    l = &fidhash(nr);
    for(new = *l; new; new = new->next)
        if(new->nr == nr)
            return 0;

    if(fidfree == 0) {
        fidfree = mallocz(sizeof(Fid) * Fidchunk, 1);
        if(fidfree == 0)
            fatal("out of memory");

        for(i = 0; i < Fidchunk-1; i++)
            fidfree[i].next = &fidfree[i+1];

        fidfree[Fidchunk-1].next = 0;
    }

    new = fidfree;
    fidfree = new->next;

    memset(new, 0, sizeof(Fid));
    new->next = *l;
    *l = new;
    new->nr = nr;
    new->fid = -1;
    new->nread = 0;
    new->nwrite = 0;
    new->bread = 0;
    new->bwrite = 0;

    return new;
}
```

Uses `Fidchunk` 100c, `fatal()` 111a, `fidfree` 101e, and `fidhash` 98f.

⟨*function* getsbuf 108b⟩≡ (113)

```
Fsrpc *
getsbuf(void)
{
    static int ap;
    int look;
    Fsrpc *wb;

    for(look = 0; look < Nr_workbufs; look++) {
        if(++ap == Nr_workbufs)
            ap = 0;
        if(Workq[ap].busy == 0)
            break;
    }

    if(look == Nr_workbufs)
        fatal("No more work buffers");

    wb = &Workq[ap];
    wb->pid = 0;
```

```
    wb->canint = 0;
    wb->flushtag = NOTAG;
    wb->busy = 1;

    return wb;
}
```
Uses Nr_workbufs 100c, Workq 101a, and fatal() 111a.

⟨*function* strcatalloc 109a⟩≡ (113)
```
char *
strcatalloc(char *p, char *n)
{
    char *v;

    v = realloc(p, strlen(p)+strlen(n)+1);
    if(v == 0)
        fatal("no memory");
    strcat(v, n);
    return v;
}
```
Uses fatal() 111a.

⟨*function* file 109b⟩≡ (113)
```
File *
file(File *parent, char *name)
{
    char buf[128];
    File *f, *new;
    Dir *dir;

    DEBUG(2, "\tfile: 0x%p %s name %s\n", parent, parent->name, name);

    for(f = parent->child; f; f = f->childlist)
        if(strcmp(name, f->name) == 0)
            break;

    if(f != nil && !f->inval)
        return f;
    makepath(buf, parent, name);
    dir = dirstat(buf);
    if(dir == nil)
        return 0;
    if(f != nil){
        free(dir);
        f->inval = 0;
        return f;
    }

    new = malloc(sizeof(File));
    if(new == 0)
        fatal("no memory");

    memset(new, 0, sizeof(File));
    new->name = strdup(name);
    if(new->name == nil)
        fatal("can't strdup");
    new->qid.type = dir->qid.type;
    new->qid.vers = dir->qid.vers;
    new->qid.path = ++qid;
```

```
    new->parent = parent;
    new->childlist = parent->child;
    parent->child = new;

    free(dir);
    return new;
}
```
Uses DEBUG 98c, fatal() 111a, makepath() 110b, and qid 102a.

⟨*function* initroot 110a⟩≡ (113)
```
void
initroot(void)
{
    Dir *dir;

    root = malloc(sizeof(File));
    if(root == 0)
        fatal("no memory");

    memset(root, 0, sizeof(File));
    root->name = strdup("/");
    if(root->name == nil)
        fatal("can't strdup");
    dir = dirstat(root->name);
    if(dir == nil)
        fatal("root stat");

    root->qid.type = dir->qid.type;
    root->qid.vers = dir->qid.vers;
    root->qid.path = ++qid;
    free(dir);
}
```
Uses fatal() 111a, qid 102a, and root 101c.

⟨*function* makepath 110b⟩≡ (113)
```
void
makepath(char *as, File *p, char *name)
{
    char *c, *seg[100];
    int i;
    char *s;

    seg[0] = name;
    for(i = 1; i < 100 && p; i++, p = p->parent){
        seg[i] = p->name;
        if(strcmp(p->name, "/") == 0)
            seg[i] = ""; /* will insert slash later */
    }

    s = as;
    while(i--) {
        for(c = seg[i]; *c; c++)
            *s++ = *c;
        *s++ = '/';
    }
    while(s[-1] == '/')
        s--;
    *s = '\0';
    if(as == s) /* empty string is root */
        strcpy(as, "/");
```

```
  }
```

⟨*function* fatal 111a⟩≡ (113)

```
  void
  fatal(char *s)
  {
      Proc *m;

      fprint(2, "iostats: %s: %r\n", s);

      /* Clear away the slave children */
      for(m = Proclist; m; m = m->next)
          postnote(PNPROC, m->pid, "exit");

      exits("fatal");
  }
```

Uses Proclist 102b.

⟨*function* rdenv 111b⟩≡ (113)

```
  char*
  rdenv(char *v, char **end)
  {
      int fd, n;
      char *buf;
      Dir *d;
      if((fd = open(v, OREAD)) == -1)
          return nil;
      d = dirfstat(fd);
      if(d == nil || (buf = malloc(d->length + 1)) == nil)
          return nil;
      n = (int)d->length;
      n = read(fd, buf, n);
      close(fd);
      if(n <= 0){
          free(buf);
          buf = nil;
      }else{
          if(buf[n-1] != '\0')
              buf[n++] = '\0';
          *end = &buf[n];
      }
      free(d);
      return buf;
  }
```

⟨*global* Defaultpath 111c⟩≡ (113)

```
  char Defaultpath[] = ".\0/bin";
```

Uses Defaultpath 111c.

⟨*function* runprog 111d⟩≡ (113)

```
  void
  runprog(char *argv[])
  {
      char *path, *ep, *p;
      char arg0[256];

      path = rdenv("/env/path", &ep);
      if(path == nil){
          path = Defaultpath;
          ep = path+sizeof(Defaultpath);
```

```
    }
    for(p = path; p < ep; p += strlen(p)+1){
        snprint(arg0, sizeof arg0, "%s/%s", p, argv[0]);
        exec(arg0, argv);
    }
    fatal("exec");
}
```

Uses `Defaultpath` 111c, `fatal()` 111a, and `rdenv()` 111b.

⟨*function* `catcher` 112a⟩≡ (113)

```
void
catcher(void *a, char *msg)
{
    USED(a);
    if(strcmp(msg, DONESTR) == 0) {
        done = 1;
        noted(NCONT);
    }
    if(strcmp(msg, "exit") == 0)
        exits("exit");

    noted(NDFLT);
}
```

Uses `DONESTR` 98b and `done` 102c.

⟨*function* `fidreport` 112b⟩≡ (113)

```
void
fidreport(Fid *f)
{
    char *p, path[128];
    Frec *fr;

    p = path;
    makepath(p, f->f, "");

    for(fr = frhead; fr; fr = fr->next) {
        if(strcmp(fr->op, p) == 0) {
            fr->nread += f->nread;
            fr->nwrite += f->nwrite;
            fr->bread += f->bread;
            fr->bwrite += f->bwrite;
            fr->opens++;
            return;
        }
    }

    fr = malloc(sizeof(Frec));
    if(fr == 0 || (fr->op = strdup(p)) == 0)
        fatal("no memory");

    fr->nread = f->nread;
    fr->nwrite = f->nwrite;
    fr->bread = f->bread;
    fr->bwrite = f->bwrite;
    fr->opens = 1;
    if(frhead == 0) {
        frhead = fr;
        frtail = fr;
    }
    else {
```

```
        frtail->next = fr;
        frtail = fr;
    }
    fr->next = 0;
}
```
Uses `fatal()` 111a, `frhead` 102e, `frtail` 102f, and `makepath()` 110b.

```
⟨iostats/iostats.c 113⟩≡
/*
 * iostats - Gather file system information
 */
#include <u.h>
#include <libc.h>
#include <auth.h>
#include <fcall.h>

#include "statfs.h"

void runprog(char**);

⟨global fcalls 102i⟩

⟨global p 103a⟩

⟨function usage (iostats/iostats.c) 103b⟩

⟨function main (iostats/iostats.c) 103c⟩

⟨function reply 106⟩

⟨function getfid 107a⟩

⟨function freefid 107b⟩

⟨function newfid 108a⟩

⟨function getsbuf 108b⟩

⟨function strcatalloc 109a⟩

⟨function file 109b⟩

⟨function initroot 110a⟩

⟨function makepath 110b⟩

⟨function fatal 111a⟩

⟨function rdenv 111b⟩

⟨global Defaultpath 111c⟩
⟨function runprog 111d⟩

⟨function catcher 112a⟩

⟨function fidreport 112b⟩
```

A.2.4 iostats/statsrv.c

⟨*global* Ebadfid 114a⟩≡ (126)

```
char Ebadfid[] = "Bad fid";
```

Uses Ebadfid 114a.

⟨*global* Enotdir 114b⟩≡ (126)

```
char Enotdir[] ="Not a directory";
```

Uses Enotdir 114b.

⟨*global* Edupfid 114c⟩≡ (126)

```
char Edupfid[] = "Fid already in use";
```

Uses Edupfid 114c.

⟨*global* Eopen 114d⟩≡ (126)

```
char Eopen[] = "Fid already opened";
```

Uses Eopen 114d.

⟨*global* Exmnt 114e⟩≡ (126)

```
char Exmnt[] = "Cannot .. past mount point";
```

Uses Exmnt 114e.

⟨*global* Enoauth 114f⟩≡ (126)

```
char Enoauth[] = "iostats: Authentication failed";
```

Uses Enoauth 114f.

⟨*global* Ebadver 114g⟩≡ (126)

```
char Ebadver[] = "Unrecognized 9P version";
```

Uses Ebadver 114g.

⟨*function* okfile 114h⟩≡ (126)

```
int
okfile(char *s, int mode)
{
    if(strncmp(s, "/fd/", 3) == 0){
        /* 0, 1, and 2 we handle ourselves */
        if(s[4]=='/' || atoi(s+4) > 2)
            return 0;
        return 1;
    }
    if(strncmp(s, "/net/ssl", 8) == 0)
        return 0;
    if(strncmp(s, "/net/tls", 8) == 0)
        return 0;
    if(strncmp(s, "/srv/", 5) == 0 && ((mode&3) == OWRITE || (mode&3) == ORDWR))
        return 0;
    return 1;
}
```

⟨*function* update 114i⟩≡ (126)

```
void
update(Rpc *rpc, vlong t)
{
    vlong t2;

    t2 = nsec();
    t = t2 - t;
    if(t < 0)
        t = 0;
```

```
    rpc->time += t;
    if(t < rpc->lo)
        rpc->lo = t;
    if(t > rpc->hi)
        rpc->hi = t;
}
```

⟨*function* Xversion 115a⟩≡ (126)

```
void
Xversion(Fsrpc *r)
{
    Fcall thdr;
    vlong t;

    t = nsec();

    if(r->work.msize > IOHDRSZ+Maxfdata)
        thdr.msize = IOHDRSZ+Maxfdata;
    else
        thdr.msize = r->work.msize;
    myiounit = thdr.msize - IOHDRSZ;
    if(strncmp(r->work.version, "9P2000", 6) != 0){
        reply(&r->work, &thdr, Ebadver);
        r->busy = 0;
        return;
    }
    thdr.version = "9P2000";
    /* BUG: should clunk all fids */
    reply(&r->work, &thdr, 0);
    r->busy = 0;

    update(&stats->rpc[Tversion], t);
}
```

Uses `Ebadver` 114g, `Maxfdata` 98g, `myiounit` 102g, `reply()` 106, `stats` 102d, and `update()` 114i.

⟨*function* Xauth 115b⟩≡ (126)

```
void
Xauth(Fsrpc *r)
{
    Fcall thdr;
    vlong t;

    t = nsec();

    reply(&r->work, &thdr, Enoauth);
    r->busy = 0;

    update(&stats->rpc[Tauth], t);
}
```

Uses `Enoauth` 114f, `reply()` 106, `stats` 102d, and `update()` 114i.

⟨*function* Xflush 115c⟩≡ (126)

```
void
Xflush(Fsrpc *r)
{
    Fsrpc *t, *e;
    Fcall thdr;

    e = &Workq[Nr_workbufs];
```

```
    for(t = Workq; t < e; t++) {
        if(t->work.tag == r->work.oldtag) {
            DEBUG(2, "\tQ busy %d pid %p can %d\n", t->busy, t->pid, t->canint);
            if(t->busy && t->pid) {
                t->flushtag = r->work.tag;
                DEBUG(2, "\tset flushtag %d\n", r->work.tag);
                if(t->canint)
                    postnote(PNPROC, t->pid, "flush");
                r->busy = 0;
                return;
            }
        }
    }

    reply(&r->work, &thdr, 0);
    DEBUG(2, "\tflush reply\n");
    r->busy = 0;
}
```

Uses `DEBUG` 98c, `Nr_workbufs` 100c, `Workq` 101a, and `reply()` 106.

⟨*function* `Xattach` 116a⟩≡ (126)

```
void
Xattach(Fsrpc *r)
{
    Fcall thdr;
    Fid *f;
    vlong t;

    t = nsec();

    f = newfid(r->work.fid);
    if(f == 0) {
        reply(&r->work, &thdr, Ebadfid);
        r->busy = 0;
        return;
    }

    f->f = root;
    thdr.qid = f->f->qid;
    reply(&r->work, &thdr, 0);
    r->busy = 0;

    update(&stats->rpc[Tattach], t);
}
```

Uses `Ebadfid` 114a, `newfid()` 108a, `reply()` 106, `root` 101c, `stats` 102d, and `update()` 114i.

⟨*function* `Xwalk` 116b⟩≡ (126)

```
void
Xwalk(Fsrpc *r)
{
    char errbuf[ERRMAX], *err;
    Fcall thdr;
    Fid *f, *n;
    File *nf;
    vlong t;
    int i;

    t = nsec();
```

```
    f = getfid(r->work.fid);
    if(f == 0) {
        reply(&r->work, &thdr, Ebadfid);
        r->busy = 0;
        return;
    }
    n = nil;
    if(r->work.newfid != r->work.fid){
        n = newfid(r->work.newfid);
        if(n == 0) {
            reply(&r->work, &thdr, Edupfid);
            r->busy = 0;
            return;
        }
        n->f = f->f;
        f = n; /* walk new guy */
    }

    thdr.nwqid = 0;
    err = nil;
    for(i=0; i<r->work.nwname; i++){
        if(i >= MAXWELEM)
            break;
        if(strcmp(r->work.wname[i], "..") == 0) {
            if(f->f->parent == 0) {
                err = Exmnt;
                break;
            }
            f->f = f->f->parent;
            thdr.wqid[thdr.nwqid++] = f->f->qid;
            continue;
        }

        nf = file(f->f, r->work.wname[i]);
        if(nf == 0) {
            errstr(errbuf, sizeof errbuf);
            err = errbuf;
            break;
        }

        f->f = nf;
        thdr.wqid[thdr.nwqid++] = nf->qid;
        continue;
    }

    if(err == nil && thdr.nwqid == 0 && r->work.nwname > 0)
        err = "file does not exist";

    if(n != nil && (err != 0 || thdr.nwqid < r->work.nwname)){
        /* clunk the new fid, which is the one we walked */
        freefid(n->nr);
    }

    if(thdr.nwqid > 0)
        err = nil;
    reply(&r->work, &thdr, err);
    r->busy = 0;

    update(&stats->rpc[Twalk], t);
}
```

Uses Ebadfid 114a, Edupfid 114c, Exmnt 114e, file() 109b, freefid() 107b, getfid() 107a, newfid() 108a, reply() 106, stats 102d, and update() 114i.

⟨*function* Xclunk 118a⟩≡ (126)
```
void
Xclunk(Fsrpc *r)
{
    Fcall thdr;
    Fid *f;
    vlong t;
    int fid;

    t = nsec();

    f = getfid(r->work.fid);
    if(f == 0) {
        reply(&r->work, &thdr, Ebadfid);
        r->busy = 0;
        return;
    }

    if(f->fid >= 0)
        close(f->fid);

    fid = r->work.fid;
    reply(&r->work, &thdr, 0);
    r->busy = 0;

    update(&stats->rpc[Tclunk], t);

    if(f->nread || f->nwrite)
        fidreport(f);

    freefid(fid);
}
```
Uses Ebadfid 114a, fidreport() 112b, freefid() 107b, getfid() 107a, reply() 106, stats 102d, and update() 114i.

⟨*function* Xstat 118b⟩≡ (126)
```
void
Xstat(Fsrpc *r)
{
    char err[ERRMAX], path[128];
    uchar statbuf[STATMAX];
    Fcall thdr;
    Fid *f;
    int s;
    vlong t;

    t = nsec();

    f = getfid(r->work.fid);
    if(f == 0) {
        reply(&r->work, &thdr, Ebadfid);
        r->busy = 0;
        return;
    }
    makepath(path, f->f, "");
    if(!okfile(path, -1)){
        snprint(err, sizeof err, "iostats: can't simulate %s", path);
        reply(&r->work, &thdr, err);
```

```
        r->busy = 0;
        return;
    }

    if(f->fid >= 0)
        s = fstat(f->fid, statbuf, sizeof statbuf);
    else
        s = stat(path, statbuf, sizeof statbuf);

    if(s < 0) {
        errstr(err, sizeof err);
        reply(&r->work, &thdr, err);
        r->busy = 0;
        return;
    }
    thdr.stat = statbuf;
    thdr.nstat = s;
    reply(&r->work, &thdr, 0);
    r->busy = 0;

    update(&stats->rpc[Tstat], t);
}
```

Uses `Ebadfid` 114a, `getfid()` 107a, `makepath()` 110b, `okfile()` 114h, `reply()` 106, `stats` 102d, and `update()` 114i.

⟨*function* `Xcreate` 119⟩≡ (126)

```
void
Xcreate(Fsrpc *r)
{
    char err[ERRMAX], path[128];
    Fcall thdr;
    Fid *f;
    File *nf;
    vlong t;

    t = nsec();

    f = getfid(r->work.fid);
    if(f == 0) {
        reply(&r->work, &thdr, Ebadfid);
        r->busy = 0;
        return;
    }


    makepath(path, f->f, r->work.name);
    f->fid = create(path, r->work.mode, r->work.perm);
    if(f->fid < 0) {
        errstr(err, sizeof err);
        reply(&r->work, &thdr, err);
        r->busy = 0;
        return;
    }

    nf = file(f->f, r->work.name);
    if(nf == 0) {
        errstr(err, sizeof err);
        reply(&r->work, &thdr, err);
        r->busy = 0;
        return;
    }
```

```
    f->mode = r->work.mode;
    f->f = nf;
    thdr.iounit = myiounit;
    thdr.qid = f->f->qid;
    reply(&r->work, &thdr, 0);
    r->busy = 0;

    update(&stats->rpc[Tcreate], t);
}
```

Uses `Ebadfid` 114a, `file()` 109b, `getfid()` 107a, `makepath()` 110b, `myiounit` 102g, `reply()` 106, `stats` 102d, and `update()` 114i.

⟨*function* `Xremove` 120a⟩≡ (126)

```
void
Xremove(Fsrpc *r)
{
    char err[ERRMAX], path[128];
    Fcall thdr;
    Fid *f;
    vlong t;

    t = nsec();

    f = getfid(r->work.fid);
    if(f == 0) {
        reply(&r->work, &thdr, Ebadfid);
        r->busy = 0;
        return;
    }

    makepath(path, f->f, "");
    DEBUG(2, "\tremove: %s\n", path);
    if(remove(path) < 0) {
        errstr(err, sizeof err);
        reply(&r->work, &thdr, err);
        freefid(r->work.fid);
        r->busy = 0;
        return;
    }

    f->f->inval = 1;
    if(f->fid >= 0)
        close(f->fid);
    freefid(r->work.fid);

    reply(&r->work, &thdr, 0);
    r->busy = 0;

    update(&stats->rpc[Tremove], t);
}
```

Uses `DEBUG` 98c, `Ebadfid` 114a, `freefid()` 107b, `getfid()` 107a, `makepath()` 110b, `reply()` 106, `stats` 102d, and `update()` 114i.

⟨*function* `Xwstat` 120b⟩≡ (126)

```
void
Xwstat(Fsrpc *r)
{
    char err[ERRMAX], path[128];
    Fcall thdr;
    Fid *f;
    int s;
```

```
	vlong t;

	t = nsec();

	f = getfid(r->work.fid);
	if(f == 0) {
		reply(&r->work, &thdr, Ebadfid);
		r->busy = 0;
		return;
	}
	if(f->fid >= 0)
		s = fwstat(f->fid, r->work.stat, r->work.nstat);
	else {
		makepath(path, f->f, "");
		s = wstat(path, r->work.stat, r->work.nstat);
	}
	if(s < 0) {
		errstr(err, sizeof err);
		reply(&r->work, &thdr, err);
	}
	else
		reply(&r->work, &thdr, 0);

	r->busy = 0;
	update(&stats->rpc[Twstat], t);
}
```

Uses `Ebadfid` 114a, `getfid()` 107a, `makepath()` 110b, `reply()` 106, `stats` 102d, and `update()` 114i.

⟨*function* `slave` 121⟩≡ (126)

```
void
slave(Fsrpc *f)
{
	int r;
	Proc *p;
	uintptr pid;
	static int nproc;

	for(;;) {
		for(p = Proclist; p; p = p->next) {
			if(p->busy == 0) {
				f->pid = p->pid;
				p->busy = 1;
				pid = (uintptr)rendezvous((void*)p->pid, f);
				if(pid != p->pid)
					fatal("rendezvous sync fail");
				return;
			}
		}

		if(++nproc > MAXPROC)
			fatal("too many procs");

		r = rfork(RFPROC|RFMEM);
		if(r < 0)
			fatal("rfork");

		if(r == 0)
			blockingslave();

		p = malloc(sizeof(Proc));
```

```
        if(p == 0)
            fatal("out of memory");

        p->busy = 0;
        p->pid = r;
        p->next = Proclist;
        Proclist = p;

        rendezvous((void*)p->pid, p);
    }
}
```

Uses MAXPROC 98d, Proclist 102b, blockingslave() 122, and fatal() 111a.

⟨*function* blockingslave 122⟩≡ (126)

```
void
blockingslave(void)
{
    Proc *m;
    uintptr pid;
    Fsrpc *p;
    Fcall thdr;

    notify(flushaction);

    pid = getpid();

    m = rendezvous((void*)pid, 0);

    for(;;) {
        p = rendezvous((void*)pid, (void*)pid);
        if(p == (void*)~0)   /* Interrupted */
            continue;

        DEBUG(2, "\tslave: %p %F b %d p %p\n", pid, &p->work, p->busy, p->pid);
        if(p->flushtag != NOTAG)
            return;

        switch(p->work.type) {
        case Tread:
            slaveread(p);
            break;
        case Twrite:
            slavewrite(p);
            break;
        case Topen:
            slaveopen(p);
            break;
        default:
            reply(&p->work, &thdr, "exportfs: slave type error");
        }
        if(p->flushtag != NOTAG) {
            p->work.type = Tflush;
            p->work.tag = p->flushtag;
            reply(&p->work, &thdr, 0);
        }
        p->busy = 0;
        m->busy = 0;
    }
}
```

Uses DEBUG 98c, flushaction() 125c, reply() 106, slaveopen() 123a, slaveread() 123b, and slavewrite() 125a.

⟨*function* slaveopen 123a⟩≡ (126)

```
void
slaveopen(Fsrpc *p)
{
    char err[ERRMAX], path[128];
    Fcall *work, thdr;
    Fid *f;
    vlong t;

    work = &p->work;

    t = nsec();

    f = getfid(work->fid);
    if(f == 0) {
        reply(work, &thdr, Ebadfid);
        return;
    }
    if(f->fid >= 0) {
        close(f->fid);
        f->fid = -1;
    }

    makepath(path, f->f, "");
    DEBUG(2, "\topen: %s %d\n", path, work->mode);

    p->canint = 1;
    if(p->flushtag != NOTAG)
        return;

    if(!okfile(path, work->mode)){
        snprint(err, sizeof err, "iostats can't simulate %s", path);
        reply(work, &thdr, err);
        return;
    }

    /* There is a race here I ignore because there are no locks */
    f->fid = open(path, work->mode);
    p->canint = 0;
    if(f->fid < 0) {
        errstr(err, sizeof err);
        reply(work, &thdr, err);
        return;
    }

    DEBUG(2, "\topen: fd %d\n", f->fid);
    f->mode = work->mode;
    thdr.iounit = myiounit;
    thdr.qid = f->f->qid;
    reply(work, &thdr, 0);

    update(&stats->rpc[Topen], t);
}
```

Uses `DEBUG` 98c, `Ebadfid` 114a, `getfid()` 107a, `makepath()` 110b, `myiounit` 102g, `okfile()` 114h, `reply()` 106, `stats` 102d, and `update()` 114i.

⟨*function* slaveread 123b⟩≡ (126)

```
void
slaveread(Fsrpc *p)
{
```

```
    char data[Maxfdata], err[ERRMAX];
    Fcall *work, thdr;
    Fid *f;
    int n, r;
    vlong t;

    work = &p->work;

    t = nsec();

    f = getfid(work->fid);
    if(f == 0) {
        reply(work, &thdr, Ebadfid);
        return;
    }

    n = (work->count > Maxfdata) ? Maxfdata : work->count;
    p->canint = 1;
    if(p->flushtag != NOTAG)
        return;
    /* can't just call pread, since directories must update the offset */
    if(f->f->qid.type&QTDIR){
        if(work->offset != f->offset){
            if(work->offset != 0){
                snprint(err, sizeof err, "can't seek in directory from %lld to %lld", f->offset, work->offset);
                reply(work, &thdr, err);
                return;
            }
            if(seek(f->fid, 0, 0) != 0){
                errstr(err, sizeof err);
                reply(work, &thdr, err);
                return;
            }
            f->offset = 0;
        }
        r = read(f->fid, data, n);
        if(r > 0)
            f->offset += r;
    }else
        r = pread(f->fid, data, n, work->offset);
    p->canint = 0;
    if(r < 0) {
        errstr(err, sizeof err);
        reply(work, &thdr, err);
        return;
    }

    DEBUG(2, "\tread: fd=%d %d bytes\n", f->fid, r);

    thdr.data = data;
    thdr.count = r;
    stats->totread += r;
    f->nread++;
    f->bread += r;
    reply(work, &thdr, 0);

    update(&stats->rpc[Tread], t);
}
```

Uses `DEBUG` 98c, `Ebadfid` 114a, `Maxfdata` 98g, `getfid()` 107a, `reply()` 106, `stats` 102d, and `update()` 114i.

⟨*function* slavewrite 125a⟩≡ (126)
```
void
slavewrite(Fsrpc *p)
{
	char err[ERRMAX];
	Fcall *work, thdr;
	Fid *f;
	int n;
	vlong t;

	work = &p->work;

	t = nsec();

	f = getfid(work->fid);
	if(f == 0) {
		reply(work, &thdr, Ebadfid);
		return;
	}

	n = (work->count > Maxfdata) ? Maxfdata : work->count;
	p->canint = 1;
	if(p->flushtag != NOTAG)
		return;
	n = pwrite(f->fid, work->data, n, work->offset);
	p->canint = 0;
	if(n < 0) {
		errstr(err, sizeof err);
		reply(work, &thdr, err);
		return;
	}

	DEBUG(2, "\twrite: %d bytes fd=%d\n", n, f->fid);

	thdr.count = n;
	f->nwrite++;
	f->bwrite += n;
	stats->totwrite += n;
	reply(work, &thdr, 0);

	update(&stats->rpc[Twrite], t);
}
```
Uses DEBUG 98c, Ebadfid 114a, Maxfdata 98g, getfid() 107a, reply() 106, stats 102d, and update() 114i.

⟨*function* reopen 125b⟩≡ (126)
```
void
reopen(Fid *f)
{
	USED(f);
	fatal("reopen");
}
```
Uses fatal() 111a.

⟨*function* flushaction 125c⟩≡ (126)
```
void
flushaction(void *a, char *cause)
{
	USED(a);
	if(strncmp(cause, "kill", 4) == 0)
		noted(NDFLT);
```

```
    noted(NCONT);
}
```

⟨iostats/statsrv.c 126⟩≡

```
#include <u.h>
#include <libc.h>
#include <auth.h>
#include <fcall.h>

#include "statfs.h"
```

⟨*global* Ebadfid 114a⟩
⟨*global* Enotdir 114b⟩
⟨*global* Edupfid 114c⟩
⟨*global* Eopen 114d⟩
⟨*global* Exmnt 114e⟩
⟨*global* Enoauth 114f⟩
⟨*global* Ebadver 114g⟩

⟨*function* okfile 114h⟩

⟨*function* update 114i⟩

⟨*function* Xversion 115a⟩

⟨*function* Xauth 115b⟩

⟨*function* Xflush 115c⟩

⟨*function* Xattach 116a⟩

⟨*function* Xwalk 116b⟩

⟨*function* Xclunk 118a⟩

⟨*function* Xstat 118b⟩

⟨*function* Xcreate 119⟩

⟨*function* Xremove 120a⟩

⟨*function* Xwstat 120b⟩

⟨*function* slave 121⟩

⟨*function* blockingslave 122⟩

⟨*function* slaveopen 123a⟩

⟨*function* slaveread 123b⟩

⟨*function* slavewrite 125a⟩

⟨*function* reopen 125b⟩

⟨*function* flushaction 125c⟩

Glossary

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.

Acc: 27a, 70c
acc: 27b, 30e, 31b, 31c, 32b, 33a
Acc.calls: 27a
Acc.ms: 27a
Acc.name: 27a
Acc.pc: 27a
Acc (typedef): 70c
acmp(): 30e, 33b
add(): 22, 23b, 24e, 25
addgraph(): 90a, 94b
addmachine(): 91a
alarmed(): 84b, 85d
argchars: 11c, 62h, 62h
batteryval(): 62b, 89b
bg: 48c, 49b, 73c
blockingslave(): 121, 122
blue: 48c, 49b, 73c
bottommargin: 48c, 49b, 72l, 72l
bout: 28a, 30e, 34c, 35d
catchalarm: 84a, 84b, 85d
catcher(): 112a
colinit(): 76b
cols: 62c
compar(): 40a
connect9fs(): 80a
connectexportfs(): 81, 82d
Context-7: 59c, 86c
contextval(): 62b, 86c
COUNTER: 38a, 39c, 39d, 40a, 43a, 43f, 44b
COUNTER.name: 39c
COUNTER.time: 39c
Data: 26a, 70c
data: 26b, 29e, 30d, 30e, 31c, 32b, 34c
Data.count: 26a
Data.down: 26a
Data.pc: 26a
Data.right: 26a
Data.time: 26a

Data (typedef): 70c
datapoint(): 77c, 78a
datas(): 29e
dbg: 101b
DEBUG: 98c, 106, 109b, 115c, 120a, 122, 123a, 123b, 125a
DEBUGFILE: 98a
defaout(): 28d
Defaultpath: 111c, 111c, 111d
dflag: 33c
doevent(): 54, 55
done: 102c, 112a
DONESTR: 98b, 112a
Dot-26: 60c, 77c, 78a
drawdatum(): 78a, 78b, 78c
drawtrace(): 48a, 55
dropgraph(): 90b, 94b
Dsegpad: 100c
Ebadfid: 114a, 114a, 116a, 116b, 118a, 118b, 119, 120a, 120b, 123a, 123b, 125a
Ebadver: 114g, 114g, 115a
Edupfid: 114c, 114c, 116b
emalloc(): 75b, 90a, 90b
Enoauth: 114f, 114f, 115b
Enotdir: 114b, 114b
Eopen: 114d, 114d
erealloc(): 75c, 91a, 92
eresized(): 85d, 94a
Err0-19: 59c, 89a
error(): 40b
estrdup(): 75d, 82b, 82d, 85d
ethererrval(): 62b, 89a
etherinval(): 62b, 88d
etheroutval(): 62b, 88e
etherval(): 62b, 88c
event: 54, 73a
eventbuf: 55, 72p
Exmnt: 114e, 114e, 116b
fatal(): 106, 108a, 108b, 109a, 109b, 110a, 111a, 111d, 112b, 121, 125b
Fault-10: 59c, 87b
faultval(): 62b, 87b
fcalls: 102i
fg: 48c, 49b, 73c
fhash: 101d
FHASHSIZE: 98e
Fid: 99e, 100d
Fid.bread: 99e
Fid.bwrite: 99e
Fid.f: 99e
Fid.fid: 99e
Fid.mode: 99e

Fid.next: 99e
Fid.nr: 99e
Fid.nread: 99e
Fid.nwrite: 99e
Fid.offset: 99e
Fid (typedef): 100d
Fidchunk: 100c, 108a
fidfree: 101e, 107b, 108a
fidhash: 98f, 107a, 107b, 108a
fidreport(): 112b, 118a
File: 100a, 100d
file(): 109b, 116b, 119
File.child: 100a
File.childlist: 100a
File.inval: 100a
File.name: 100a
File.parent: 100a
File.qid: 100a
File (typedef): 100d
filter(): 79b
flushaction(): 122, 125c
Frec: 99a, 100d
Frec.bread: 99a
Frec.bwrite: 99a
Frec.next: 99a
Frec.nread: 99a
Frec.nwrite: 99a
Frec.op: 99a
Frec.opens: 99a
Frec (typedef): 100d
freefid(): 107b, 116b, 118a, 120a
frhead: 102e, 112b
frtail: 102f, 112b
Fsrpc: 99d, 100d
Fsrpc.buf: 99d
Fsrpc.busy: 99d
Fsrpc.canint: 99d
Fsrpc.flushtag: 99d
Fsrpc.pid: 99d
Fsrpc.work: 99d
Fsrpc (typedef): 100d
getfid(): 107a, 116b, 118a, 118b, 119, 120a, 120b, 123a, 123b, 125a
getsbuf(): 108b
Graph: 59b, 95
graph(): 34c
Graph.colindex: 59b
Graph.data: 59b
Graph.label: 59b
Graph.mach: 59b

Graph.ndata: 59b
Graph.newvalue: 59b
Graph.overflow: 59b
Graph.overtmp: 59b
Graph.r: 59b
Graph.update: 59b
Graph (typedef): 95
green: 48c, 49b, 73c
grey: 48c, 73c
Height: 49b, 53, 54, 55, 72j, 72j
Idle-14: 59c, 88a
idleval(): 62b, 88a
ilog10(): 82c, 82d
In-16: 59c, 88c, 88d
indent(): 34c, 35d
InIntr-15: 59c, 88b
inintrval(): 62b, 88b
initmach(): 82d, 91a
initroot(): 110a
Interrupt-8: 59c, 86d
intrval(): 62b, 86d
K-62: 46c
killall(): 75a, 75b, 75c, 75d, 79b, 90b, 94a, 94c
label(): 77a, 92
labelstrs(): 91b, 91c, 92
labelwidth(): 91c, 92
Lablen-29: 60c, 91b, 91c, 92
Labspace-25: 60c, 77a, 92
lineht: 49b, 72m, 72m
Link-17: 59c
Load-13: 59c, 87e
loadbuf(): 76c, 82d, 85d
loadval(): 62b, 87e
logscale: 63c, 63c, 77c, 78c, 91b
Lx-30: 60c, 92
mach: 39d, 43f, 62e, 90a, 91a, 92
Machine: 60a, 95
Machine.batteryfd: 60a
Machine.batterystats: 60a
Machine.bitsybatfd: 60a
Machine.buf: 60a
Machine.bufp: 60a
Machine.devswap: 60a
Machine.devsysstat: 60a
Machine.disable: 60a
Machine.ebufp: 60a
Machine.etherfd: 60a
Machine.ifstatsfd: 60a
Machine.lgproc: 60a

Machine.name: 60a
Machine.netetherifstats: 60a
Machine.netetherstats: 60a
Machine.nproc: 60a
Machine.prevetherstats: 60a
Machine.prevsysstat: 60a
Machine.remote: 60a
Machine.shortname: 60a
Machine.statsfd: 60a
Machine.swapfd: 60a
Machine.temp: 60a
Machine.tempfd: 60a
Machine (typedef): 95
main-51(): 37
main-64(): 22
main-66(): 42
Mainproc-20: 60b
makepath(): 109b, 110b, 112b, 118b, 119, 120a, 120b, 123a
Maxfdata: 98g, 99d, 106, 115a, 123b, 125a
Maxmem-3: 59c, 82d, 86a
MAXNUM-1: 59a, 82d
MAXPROC: 98d, 121
Maxrpc: 98g, 99c
Maxswap-5: 59c, 82d, 86b
Mbattery-31: 61a, 85a
Mcontext-32: 61a, 84d
mediumfont: 62f
Mem-2: 59c, 86a
memval(): 62b, 86a
Menu2: 61a
menu2: 61c, 94b
menu2str: 61b, 61c, 90a, 90b, 94b
Mether-33: 61a, 84e
Methererr-34: 61a, 84e
Metherin-35: 61a, 84e
Metherout-36: 61a, 84e
Mfault-37: 61a, 84d
Midle-38: 61a, 84d
MilliRound-59: 58, 72g
Minintr-39: 61a, 84d
Mintr-40: 61a, 84d
mkcol(): 76a
Mload-41: 61a, 84d
Mmem-42: 61a, 84c
mouseproc(): 94b
Mouseproc-21: 60b
ms: 30e, 31a, 31c
MS-54: 47a, 58, 72c, 72g
Msignal-47: 61a, 85b

Mswap-43: 61a, 84c
Msyscall-44: 61a, 84d
Mtemp-48: 61a, 85c
Mtlbmiss-45: 61a, 84d
Mtlbpurge-46: 61a, 84d
myiounit: 102g, 115a, 119, 123a
mysysname: 62g, 82d
name(): 34c, 35a
Ncolor-23: 60c, 62c, 90a
Ncolor-61: 46c, 47d, 49b
ndata: 26c, 29e, 32a, 34d
needbattery(): 85a, 85d
needether(): 84e, 85d
needsignal(): 85b, 85d
needstat(): 84d, 85d
needswap(): 84c, 85d
needtemp(): 85c, 85d
nevents: 55, 72o
Nevents-60: 46c, 55
newfid(): 108a, 116a, 116b
newtask(): 53, 55
newvaluefn: 62b, 90a
newwin: 48a, 53, 54, 55, 72h
ngraph: 63a, 90a, 90b, 91a, 91c, 92, 94b
Nlab-28: 60c, 91b, 91c, 92
nmach: 62k, 90a, 90b, 91a, 91c, 92
Nmenu2-49: 61a, 61b, 62a, 62b, 94b
notifyf(): 22, 24c
now: 49b, 55, 73c
NPROC-22: 60b, 62i, 63g, 75a
Nr_workbufs: 100c, 108b, 115c
NS-52: 72a
nsym: 29a, 30e, 33a
ntasks: 49b, 53, 54, 55, 73c
numblocks-56: 72e
okfile(): 114h, 118b, 123a
old9p(): 80b, 81
oldsystem: 63e, 63e, 81
OneRound-58: 58, 72g
Opwid-27: 60c, 90a, 90b, 94b
Out-18: 59c, 88c, 88e
output: 22, 23a, 23b, 23c
p: 49b, 53, 103a, 106
parity: 62j, 77b
paritypt(): 77b, 78a, 78b, 92
paused: 49b, 55, 73c
Pc: 27c, 70c
Pc.next: 27c
Pc.pc: 27c

Pc (typedef): 70c
PCRES-50: 39b, 39d
PCRES-65: 43d, 43f
pids: 62i, 75a, 94c
plot(): 30e
present: 62a, 84c, 84d, 84e, 85a, 85b, 85c, 90a, 90b, 94b
prevts: 49b, 53, 54, 55, 73c
Proc: 100b, 100d
Proc.busy: 100b
Proc.next: 100b
Proc.pid: 100b
Proc (typedef): 100d
Proclist: 102b, 111a, 121
procnames: 63g, 94c
Procno-6: 59c
profdev: 47e, 47e, 48a, 55
qid: 102a, 109b, 110a
rdenv(): 111b, 111d
readmach(): 85d
readnums(): 79a, 82a, 82d, 85d
readswap(): 82a, 82d, 85d
red: 48c, 49b, 73c
redraw(): 78b
reopen(): 125b
reply(): 106, 115a, 115b, 115c, 116a, 116b, 118a, 118b, 119, 120a, 120b, 122, 123a, 123b, 125a
resize(): 92, 94a, 94b
rflag: 34c, 35e
root: 101c, 110a, 116a
roundup-57: 72f
Rpc: 99b, 100d
Rpc.bin: 99b
Rpc.bout: 99b
Rpc.count: 99b
Rpc.hi: 99b
Rpc.lo: 99b
Rpc.name: 99b
Rpc.time: 99b
Rpc (typedef): 100d
runprog(): 111d
S-55: 47a, 58, 72d
scale: 46d, 47a, 49b, 63b, 63b, 77c, 78c, 91b
scale.bigtics: 46d
scale.littletics: 46d
scale.scale: 46d
scale.sleep: 46d
scales: 47a, 49b, 55
schedstatename: 54, 55, 73b
shortname(): 82b, 82d, 85d
signalval(): 62b, 89c

slave(): 102i, 121
slaveopen(): 122, 123a
slaveread(): 122, 123b
slavewrite(): 122, 125a
sleeptime: 63f, 63f, 86c, 86d, 87a, 87b, 87c, 87d, 88c, 88d, 88e, 89a, 89c, 89d
startproc(): 94c
Stats: 99c, 100d
stats: 102d, 106, 115a, 115b, 116a, 116b, 118a, 118b, 119, 120a, 120b, 123a, 123b, 125a
Stats.nproto: 99c
Stats.nrpc: 99c
Stats.rpc: 99c
Stats.totread: 99c
Stats.totwrite: 99c
Stats (typedef): 100d
strcatalloc(): 109a
sum(): 30e, 31c, 31c
Swap-4: 59c, 86b
swapdata(): 29e, 30a
swapval(): 62b, 86b
symind(): 31c, 33a
syms(): 29b
Syscall-9: 59c, 87a
syscallval(): 62b, 87a
tabstop: 35b, 35b, 35d
Task: 73c, 73c
Task (typedef): 73c
tasks: 47c, 49b, 53, 54, 55
tempval(): 62b, 89d
TEvent: 73c, 73c
TEvent (typedef): 73c
threadmain(): 48a
time2x-63: 49a, 49b
timeconv(): 48a, 58
tinyfont: 48c, 49b, 73c
TLBfault-11: 59c, 87c
tlbmissval(): 62b, 87c
TLBpurge-12: 59c, 87d
tlbpurgeval(): 62b, 87d
topmargin: 48c, 49b, 72k, 72k
trans: 28d, 28e
triggerproc: 48a, 49b, 73c
update(): 114i, 115a, 115b, 116a, 116b, 118a, 118b, 119, 120a, 120b, 123a, 123b, 125a
update1(): 78c, 90a
US-53: 47a, 58, 72b, 72g
usage(): 103b
verbose: 47b
wctlfd: 53, 54, 55, 72n
Width: 49b, 53, 54, 55, 72i, 72i
Workq: 101a, 108b, 115c

Xattach(): 102i, 116a
Xauth(): 102i, 115b
Xclunk(): 102i, 118a
Xcreate(): 102i, 119
Xflush(): 102i, 115c
Xremove(): 102i, 120a
Xstat(): 102i, 118b
Xversion(): 102i, 115a
Xwalk(): 102i, 116b
Xwstat(): 102i, 120b
ylabels: 63d, 63d, 92
Ysqueeze-24: 60c, 77a
__anon_enum_1: 98g
__anon_enum_2: 100c
__anon_enum_3: 59c
__anon_enum_4: 60b
__anon_enum_5: 60c

Bibliography

[Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. cited page(s) 7

[Pad09] Yoann Padioleau. Syncweb, literate programming meets unison, 2009. https://github.com/aryx/syncweb. cited page(s) 7

[Pad14] Yoann Padioleau. *Principia Softwarica: The Plan 9 Kernel 9*. 2014. cited page(s) 5, 6, 16, 68

[Pad15] Yoann Padioleau. *Principia Softwarica: The ARM Linker 5l*. 2015. cited page(s) 5, 6, 19, 20, 68

[Pad16a] Yoann Padioleau. *Principia Softwarica: The Plan 9 Core Libraries*. 2016. cited page(s) 69

[Pad16b] Yoann Padioleau. *Principia Softwarica: The Plan 9 Debuggers and Tracers*. 2016. cited page(s) 69

[Pad16c] Yoann Padioleau. *Principia Softwarica: The Plan 9 Graphics System draw*. 2016. cited page(s) 69