Principia Softwarica: The Plan 9 Profilers

version 0.1

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
Ken Thompson and Rob Pike

April 28, 2026

Contents

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 what 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:

- The original UNIXprof from V6 and V7 is the historical ancestor of all the tools in this list. It was a simple sampling profiler: the program called `monitor()` to enable PC sampling, ran normally, and on exit wrote a `mon.out` file that `prof` read to produce a flat time histogram. Its source is publicly available in the Unix V6 and V7 archives (released under the Caldera license in 2002). However, it is entirely superseded by its successors, and Plan 9's own `prof` covers the same sampling idea in a cleaner design without the `monitor()` indirection.

- `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 "g" stands for graph (call graph), not GNU—the name predates the GNU project. 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. It uses hardware performance counters via the Linux kernel's `perf_events` interface to trigger sampling. OProfile is now essentially unmaintained—its last significant development activity was around 2015—and has been largely displaced by newer tools built on the same kernel interface.

- `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—orders of magnitude larger than the Plan 9 tools in this book—making it unsuitable for a teaching book.

- eBPF (extended Berkeley Packet Filter) represents a fundamentally different approach from the tools above. The original BPF (1992) was a tiny in-kernel filter for network packets; eBPF (Linux 3.18, 2014) generalized it into a safe, general-purpose virtual machine that can run user-supplied programs inside the kernel. Instead of passively sampling or instrumenting a target program, eBPF programs attach to kernel hooks—tracepoints, function entry/exit, syscalls, network events—and can aggregate data entirely in-kernel before reporting to userspace. `perf` can load eBPF programs as event handlers; `bpftrace` (a high-level tracing language) and `bcc` (a Python/C++ toolkit) compile to eBPF and load it. The eBPF ecosystem (`linux/kernel/bpf`, `libbpf`, `bpftrace`, `bcc`) is now the dominant Linux tracing substrate, but its complexity far exceeds the scope of this 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`.

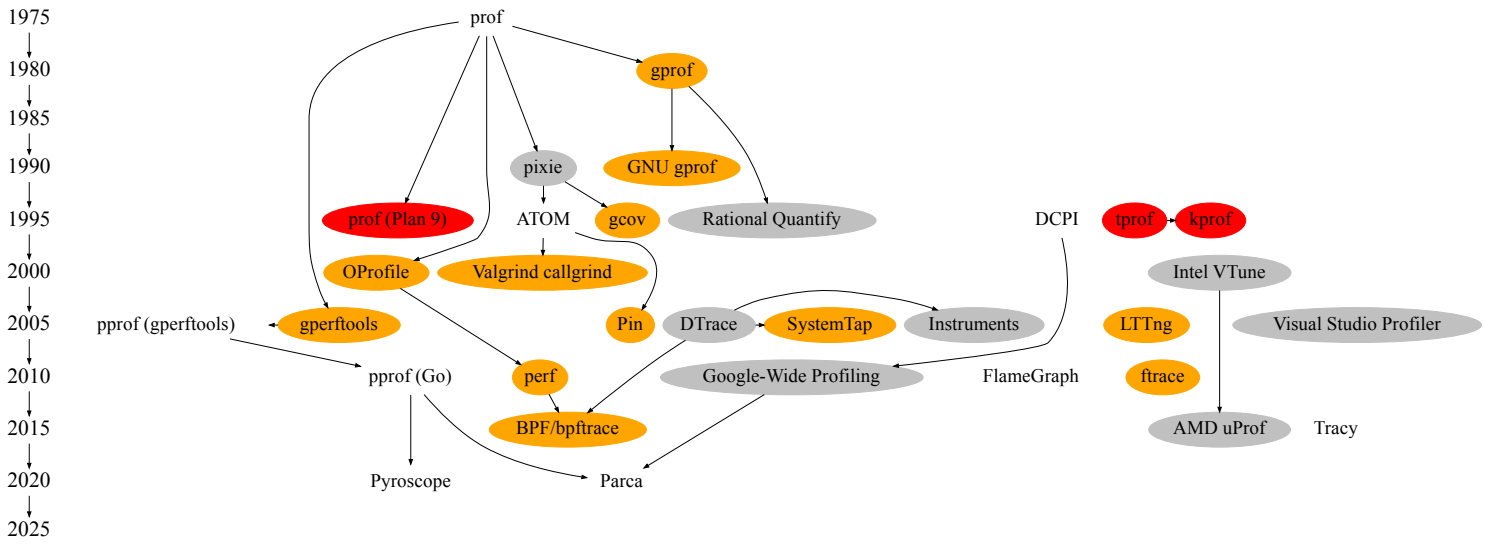


Figure 1.1: Profilers timeline

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. For a broader treatment of performance analysis methodology and tools, see Gregg's *Systems Performance* [Gre20], which covers much of the same territory from a Linux perspective.

If, while reading this book, you have specific questions on the command-line interface of the Plan 9 profiling tools, I suggest you to consult the manual pages in my Plan 9 repository: `docs/man/1/prof` (and `tprof`), `docs/man/1/time`, `docs/man/1/trace`, and `docs/man/1/ratrace` for the user-space tools, plus `docs/man/3/proc` for the `/proc/<pid>/profile` file read by `tprof` and described in the kernel-support chapter. The related administration tools `8/stats` and `8/statusbar` are documented in `docs/man/8/`.

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

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.

The combination of instrumentation and sampling in a single tool was pioneered by `gprof` (Susan Graham, Peter Kessler, and Marshall McKusick, Berkeley, 1982)—the first widely-deployed profiler and the reason we have the term "profile" in the first place.

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.

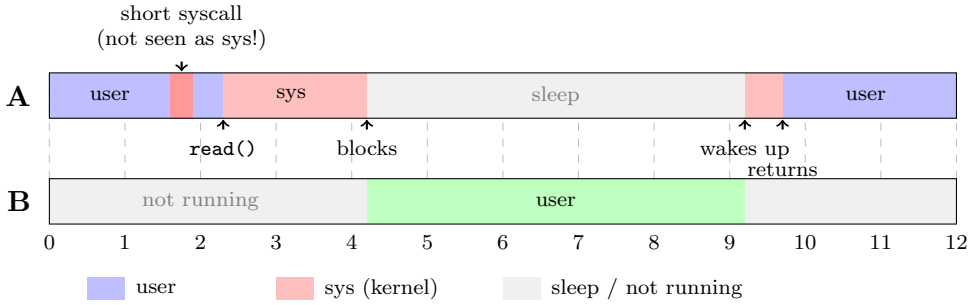


Figure 2.1: Tick-based time accounting with two interleaved processes. Ticks are numbered 0–12 along the bottom. The short syscall between ticks 1 and 2 is invisible to tick accounting (counted as user). The `read()` blocks shortly after tick 4 and A wakes up shortly after tick 9. The syscall returns 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.

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

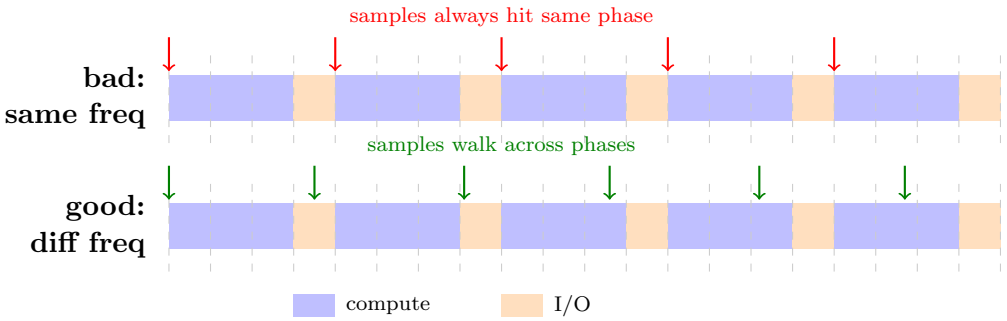


Figure 2.2: The aliasing problem. Top: when the profiling timer fires at the same frequency as the program's repeating pattern, every sample hits the same phase (here, always "compute"), missing the I/O portion entirely. Bottom: using a different frequency (e.g., 113 instead of 100), samples gradually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

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.

Instrumentation profilers come in roughly three flavors, differentiated by who inserts the hooks:

who instruments	example tools
the compiler/linker	\plan 5l -p, gcc -pg (gprof), Go's -cover, Rust's -C profile-generate
a binary rewriter	Valgrind/Callgrind, Pin, DynamoRIO, Intel SDE
the language runtime	Java -agentlib:hprof, -XX:+DTrace, Python cProfile, Ruby TracePoint, Node.js --prof

Plan 9's `5l -p` is at the simplest end of that spectrum: the linker has already lowered the program to machine code, so it can see every `TEXT` pseudo-instruction and emit a `BL _profin` / `BL _profout` pair right next to it. Binary rewriters like Valgrind go one step further—they interpret every instruction in a JIT, which lets them observe things like memory access patterns, but at a 10–50x slowdown. Runtime-level instrumentation (Java agents, Python cProfile) hooks into the interpreter's function-dispatch path, so it sees exactly what the language already considers a "call"—closer to the source-level picture but blind to inlined and native code.

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.)

Every modern operating system ships a sampling profiler, and they differ mainly in what the "sample" is allowed to contain:

OS / tool	sample source	what else per sample
\plan tprof/kprof	kernel profclock tick	just PC (8-byte bucket)
Linux perf	PMU overflow IRQ	PC + user/kernel stack, HW counters
Linux eBPF profile	perf_event + BPF prog	arbitrary BPF-computed
macOS Instruments	Mach thread_suspend	PC + stack + DTrace-derived data
Solaris/illumos DTrace	profile:::profile-Nhz	PC + stack + kernel probe arguments
Windows ETW / Xperf	PROFILE event	PC + stack, kernel flight recorder
Intel VTune	PEBS (precise PMU)	PC + regs + memory addr + latency

The progression is cumulative: each newer tool collects strictly more data per sample than the one before. `tprof` at one extreme records only the interrupted PC, which is why its output is a single flat histogram. Linux `perf` walks the user stack at each sample (using frame pointers, DWARF unwind info, or the kernel's LBR registers), which is what turns sampled profiles into the flame graphs you see everywhere today. PEBS on Intel even tells you the exact faulting memory address, enabling per-cache-line profiling that no software solution can reproduce. The price of all that data is a more intricate toolchain—`tprof` is under 200 lines precisely because it only needs to map a bucket index to a symbol name.

2.1.6 Self time vs cumulative time

Once a profiler has collected its data, there is still a choice in how to present it. The same numbers can be totalled two different ways, and the difference is the single most common source of confusion when reading profile output.

Self time (also called flat or exclusive time) is the time spent executing inside a function's own body, not counting time spent in functions it calls. Cumulative time (also called inclusive or total time) includes every callee transitively. The two views answer different questions: self time answers "which function is slow?", cumulative time answers "which function is responsible for the slowness?".

A tiny example makes the difference concrete. Suppose `main` calls `foo`, which calls `bar`, and `bar` is the only function that does real work:

function	self	cumulative
main	0%	100%
foo	0%	100%
bar	100%	100%

The cumulative view puts `main` at the top of the list (it "took" 100% of the run), but that is true of `main` in every program ever profiled. The self view correctly fingers `bar` as the hot spot. Conversely, if a slow operation is spread across ten small leaf functions called from one parent, the self view scatters the cost across ten rows while the cumulative view collapses it onto the parent, which is what you want when deciding where to cut.

Plan 9's `prof` shows self time by default (the `%` column in its main output) and switches to a cumulative call-tree presentation under `prof -d`. `gprof` on Unix showed both columns side by side from the start; modern tools (`pprof`, `perf report`, Instruments, async-profiler) let the user toggle. Sampling profilers like `tprof` can only report self time directly, because without stack walking they do not know who called the sampled PC—which is one reason stack-sampling perf-style profilers are now standard on Linux and macOS.

2.1.7 Flat list, call tree, flamegraph

The same per-function histogram can be presented in three standard ways, and which ones a tool offers is a good indicator of how much data it collected. The choice is independent of whether the data came from instrumentation or sampling—it is purely a question of how to display totals, callers, and visual proportion.

The flat list is the simplest: one row per function, sorted by self time, with a percentage column. Plan 9's `prof` without `-d` shows exactly this. It answers "which function should I open first?" in a single glance, but says nothing about who called whom.

The call tree adds caller-callee relationships: each row is indented under its caller, with the tree rooted at `main`. This is also where the cumulative view from the previous subsection becomes actually useful—each parent shows the total time inherited from everything beneath it. `prof -d` shows a call tree, as do `gprof`'s "call graph" section and `pprof --tree`. A call tree needs more data than a flat list—either an instrumentation profiler that records each call arc (the `prof` route), or a sampling profiler that walks the stack at each tick to recover the call chain.

The flamegraph (Brendan Gregg, 2011) is the third view. Each stack sample becomes a vertical stack of boxes, callers at the bottom and callees on top, and identical adjacent columns are merged horizontally:

```
|             bar             |  <- top row, hottest leaf
|             foo             |
|            main             |  <- bottom row, root
```

Box width is proportional to time, so wide boxes near the top are the hot spots and narrow boxes are noise. The flamegraph needs the same caller-chain data as a call tree but visualizes it as area instead of indentation, which scales to thousands of unique stacks in one page. `tprof` cannot produce one because it samples only the top-level PC, with no stack information; producing flamegraphs is the main reason modern sampling profilers (`perf`, `async-profiler`, Instruments, Chrome DevTools, `speedscope`, `py-spy`) all walk the stack on every sample.

2.1.8 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. The modern standard-bearer for this category is DTrace (Bryan Cantrill, Sun Microsystems, 2005), which was revolutionary because it let you instrument a running production system without restarting or recompiling—a capability that Linux's eBPF has since replicated.

2.1.9 From profiling to observability

The profiler's core question—*where is the time going?*—has not changed in forty years, but the scale has. In 1982, `gprof` asked which *function* in one program was slow. In 2005, DTrace asked what was happening across an entire *machine* in production. By 2010, Google's Dapper paper introduced *distributed tracing*—following a single request across dozens of services in a datacenter—and tools like Zipkin and Jaeger made that idea accessible to everyone. By 2017, the term *observability* (popularized by Charity Majors) had emerged to describe the convergence of traces, metrics, and logs as "three pillars" of understanding a running system, with OpenTelemetry as the standard that unifies them.

Plan 9's `trace` and `stats` are interesting precursors in this lineage: they already observe the whole system (multiple processes, kernel and user space, scheduler events and system-wide counters) rather than just one program. They are closer in spirit to modern observability tools than `gprof` ever was. The tools in this book

are the single-machine ancestors of that entire distributed-observability ecosystem, and the "where is the time going?" question that drives `prof` and `tprof` is the same question that drives a Jaeger trace across a hundred microservices.

2.2 `time` command-line interface

⟨`main()` *(time.c) usage if no args* 14a⟩≡ (26)
```
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* 14b⟩≡ (31d)
```
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* 14c⟩≡ (42)
```
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* 14d⟩≡ (47)
```
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)* 15a⟩≡ (106)

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

Uses `argchars` 71f.

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` 15b⟩≡

```
#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` 15c⟩≡

```
#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.

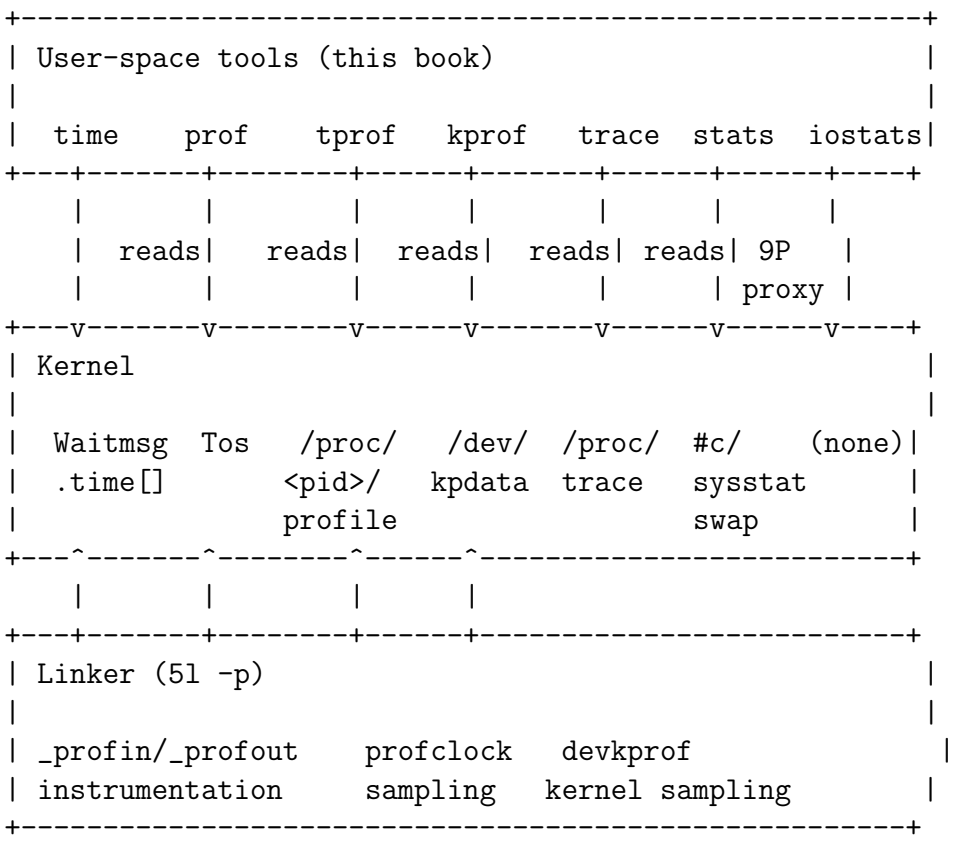
Function	Ch.	File	Entities	LOC
timing a command	5	`misc/time.c`	`main()X add()`[27b]	111
profiling data structures	6	`misc/prof.c`	`Data`[30a] `Pc`[31c] `Acc`[31a]	414
profiling display	6	`misc/prof.c`	`main()X syms()`[33b] `graph()`[38c] `plot()`[34e]	
sampling profiler	7	`misc/tprof.c`	`main()X COUNTER`	159
kernel profiler	8	`misc/kprof.c`	`main()X COUNTER`	154
trace data structures	9	`misc/trace.c`	`TEvent`[84c] `Task`[84c]	841
trace GUI	9	`misc/trace.c`	`threadmain()`[55a] `doevent()`[61b] `drawtrace()`[63]	
stats data structures	10	`misc/stats.c`	`Graph`[68b] `Machine`[68d]	1622
stats GUI	10	`misc/stats.c`	`main()X mkcol()`[87a]	
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.



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.

To make the whole loop concrete, here is what happens when the profiling timer fires while a process is executing instruction at PC `0x1048` (inside `foo`, whose entry is at `0x1040`):

```
1. HW timer interrupt fires on the CPU
     \|/
2. kernel trap entry saves user PC in Ureg
     \|/
3. profclock(Ureg *ur)  (runs in interrupt context)
     p = up                              ; current process
     if (!p || !p->prof.pp) return       ; not profiling?
     seg = p->seg[TSEG]                  ; text segment
```

```
        if (seg->profile == nil) return   ; segment has no buckets
        if (up in user mode) {
            bucket = (ur->pc - seg->base) >> LRESPROF
                   = (0x1048  -  0x1000 ) >> 3    = 9
            seg->profile[bucket + 2]++    ; +2 for [0]=total, [1]=outside
            seg->profile[0]++             ; total ms counter
        } else {
            seg->profile[1]++             ; outside text
        }
        \|/
    4. trap return; user code resumes at 0x1048
```

Three things are worth calling out. First, the hook runs entirely on the interrupt stack and does not touch any sleeping lock—it is a handful of integer operations, so the overhead per tick is measured in dozens of nanoseconds. Second, buckets `0` and `1` exist precisely so `tprof` can later compute `delta = data[0] - data[1]` (in-text time), which is the denominator of every percentage it prints. Third, the "outside text" branch catches PCs that fall in the kernel, in shared libraries, or in `etext` padding—ticks are never lost, they are just binned into `profile[1]` when they have nowhere else to go.

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`.

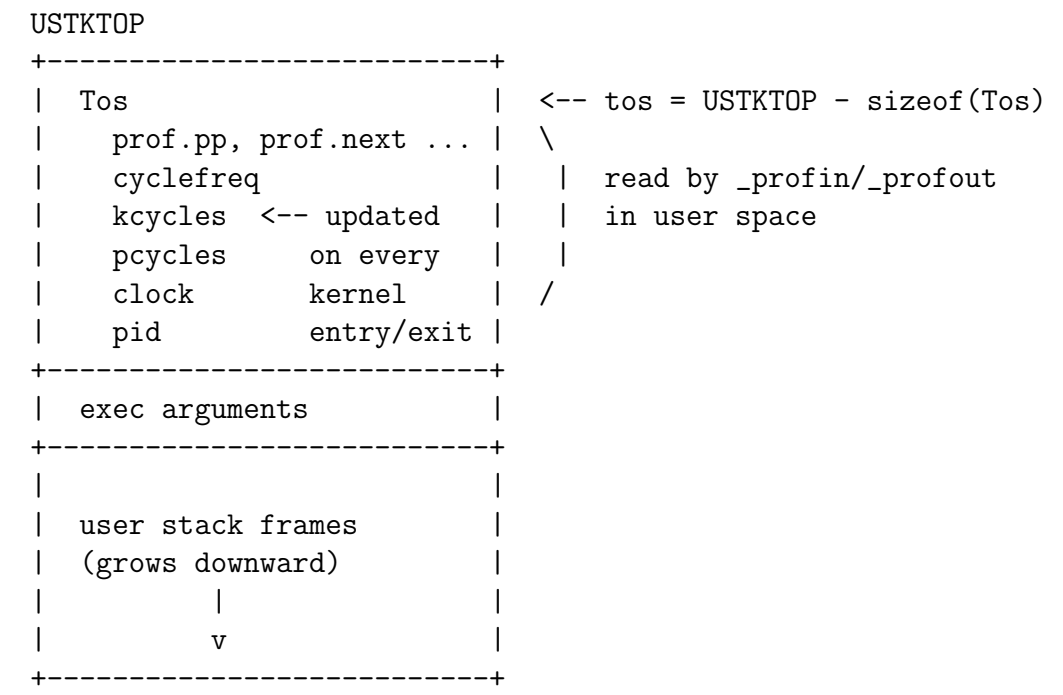
3.6 Cycle-accurate timing: TOS

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:



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`.

This is the opposite of the usual division of labor. `gcc -pg` (gprof's instrumentation flag) emits the `mcount` call from the C compiler itself, so every `.o` file must be recompiled with `-pg`; LLVM's `-fprofile-instr-generate` does the same. The argument for compile-time instrumentation is that the compiler already knows function entry and exit points and can use that information to optimize the call (for instance, omitting the hook in leaf functions). The argument for link-time instrumentation is that nothing has to be recompiled to enable profiling, and every function—including ones from libraries you do not have source for—is uniformly instrumented. Plan 9 picks the second trade-off, which is consistent with its overall style: the linker already understands the calling convention well enough to insert a `BL _profin` without clobbering arguments, so there is no reason to duplicate that knowledge in the compiler.

4.1 Time-per-function profiling: `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 Call counts: `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 Entry point: `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)* 26⟩≡ (81a)

```
void
main(int argc, char *argv[])
{
    Waitmsg *w;
    long l;
    ⟨main()(time.c) other locals 27d⟩

    ⟨main()(time.c) usage if no args 14a⟩

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

        ⟨main()(time.c) in child after exec 27c⟩
        error(argv[1]);
    case -1:
        error("fork");
    }
    // else, in parent

    notify(notifyf);
```

```
    loop:
    w = wait();
    ⟨main() (time.c) if w == nil 28b⟩
    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 28e⟩
    ⟨main() (time.c) display also wait message and status 29⟩
    fprint(STDERR, "%s\n", output);
    exits(w->msg);
}
```

Uses `add()` 27b, `notifyf()` 28c, and `output` 27a.

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)* 27a⟩≡ (81a)

```
char output[4096];
```

⟨*function* `add` *(time.c)* 27b⟩≡ (81a)

```
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` 27a.

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` 27c⟩≡ (26)

```
if(argv[1][0] != '/' && strncmp(argv[1], "./", 2) &&
   strncmp(argv[1], "../", 3)){
    sprint(output, "/bin/%s", argv[1]);
    exec(output, &argv[1]);
}
```

Uses `output` 27a.

5.2 Error management

⟨`main()` *(time.c) other locals* 27d⟩≡ (26) 28d ▷

```
char err[ERRMAX];
```

⟨*function* `error` *(time.c)* 28a⟩≡ (81a)
```
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` 28b⟩≡ (26)
```
if(w == nil){
    errstr(err, sizeof err);
    if(strcmp(err, "interrupted") == 0)
        goto loop;
    error("wait");
}
```

⟨*function* `notifyf` *(time.c)* 28c⟩≡ (81a)
```
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* 28d⟩+≡ (26) ◁27d
```
int i;
char *p;
```

⟨`main()` *(time.c) display also argv* 28e⟩≡ (26)
```
for(i=1; i<argc; i++){
    add("%s", argv[i], 0);
    if(i>4){
        add("...");
        break;
    }
}
```
Uses `add()` 27b.

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* 29⟩≡ (26)

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

Uses `add()` 27b.

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`[30a] records written by the runtime library (`lib_core/libc/profile.c`). Each record represents a function call in the dynamic call tree:

⟨*struct* `Data` 30a⟩≡ (81c)

```
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` 30b⟩≡ (81c)

```
// ref_own<array<Data> (len = ndata)
Data* data;
```

⟨*global* `ndata` 30c⟩≡ (81c)

```
// 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`[31a] 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` 31a⟩≡ (81c)
```
struct Acc
{
    char *name;
    ulong pc;
    ulong ms;
    ulong calls;
};
```

⟨*global* `acc` 31b⟩≡ (81c)
```
Acc* acc;
```

The `Pc`[31c] structure is only used by `graph()`[38c] 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` 31c⟩≡ (81c)
```
struct Pc
{
    Pc *next;
    ulong pc;
};
```

6.2 Entry point: `main()`

The main function has three phases: load the symbol table from the executable via `syms()`[33b], load the profiling data from `prof.out` via `datas()`[33e], then display the results via either `plot()`[34e] (flat profile) or `graph()`[38c] (call graph with `-d`).

⟨*function main (misc/prof.c)* 31d⟩≡ (81c)
```
void
main(int argc, char *argv[])
{
    char *s;

    ⟨main() (prof.c) adjust tabstop 39d⟩

    ARGBEGIN{
    ⟨main() (prof.c) command-line parsing 32c⟩
    default:
        ⟨main() (prof.c) print usage and exit 14b⟩
    }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 38b⟩
        else
            plot();
    }
    exits(nil);
}
```

⟨*global* bout 32a⟩≡ (81c)
```
Biobuf bout;
```

⟨*global* verbose 32b⟩≡ (81c)
```
int verbose;
```

⟨main() (prof.c) command-line parsing 32c⟩≡ (31d) 38a▷
```
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 32d⟩≡ (81c)
```
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 32e.

⟨*global* trans 32e⟩≡ (81c)
```
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()`[33b] 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` 33a⟩≡ (81c)
```
long nsym;
```

⟨*function* `syms` 33b⟩≡ (81c)
```
void
syms(char *cout)
{
    Fhdr f;
    fdt fd;

    fd = open(cout, 0);
    ⟨syms() sanity check fd 33c⟩
    if (!crackhdr(fd, &f)) {
        fprint(STDERR, "can't read text file header\n");
        exits("read");
    }
    ⟨syms() sanity check f.type 33d⟩
    if (syminit(fd, &f) < 0) {
        fprint(STDERR, "syminit: %r\n");
        exits("syms");
    }
    close(fd);
}
```

⟨`syms()` *sanity check* `fd` 33c⟩≡ (33b)
```
if(fd < 0){
    perror(cout);
    exits("open");
}
```

⟨`syms()` *sanity check* `f.type` 33d⟩≡ (33b)
```
if (f.type == FNONE) {
    fprint(STDERR, "text file not an a.out\n");
    exits("file type");
}
```

6.4 Profiling data loading: `datas()`

`datas()`[33e] reads the `prof.out` file, which is simply a flat array of `Data`[30a] 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()`[34a].

⟨*function* `datas` 33e⟩≡ (81c)
```
void
datas(char *dout)
{
    fdt fd;
    Dir *d;
    int i;

    fd = open(dout, 0);
    ⟨datas() sanity check fd 34b⟩
```

```
    d = dirfstat(fd);
    ⟨datas() sanity check d 34c⟩
    ndata = d->length/sizeof(data[0]);
    data = malloc(ndata*sizeof(Data));
    ⟨datas() sanity check data 34d⟩
    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 30b, ndata 30c, and swapdata() 34a.

⟨*function* swapdata 34a⟩≡ (81c)

```
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 34b⟩≡ (33e)

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

⟨datas() *sanity check* d 34c⟩≡ (33e)

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

⟨datas() *sanity check* data 34d⟩≡ (33e)

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

Uses data 30b.

6.5 Displaying data: plot()

plot()[34e] produces the flat profile. It first builds the acc[31b] array by iterating over all text symbols, then calls sum()[35c] 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 34e⟩≡ (81c)

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

```
    for (nsym = 0; textsym(&s, nsym); nsym++) {
        acc = realloc(acc, (nsym+1)*sizeof(Acc));
        ⟨plot() sanity check acc 35b⟩
        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 31b, acmp() 37b, bout 32a, data 30b, ms 35a, nsym 33a, and sum() 35c.

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 35a⟩≡ (81c)

```
ulong ms;
```

⟨plot() *sanity check* acc 35b⟩≡ (34e)

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

Uses acc 31b.

sum() recursively walks the call tree encoded in the data[30b] 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()[37a] (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 35c⟩≡ (81c)

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

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

    ⟨sum() sanity check i 36a⟩

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

    ⟨sum() if verbose 36b⟩
    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 31b, data 30b, ms 35a, sum() 35c, and symind() 37a.

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 36a⟩≡ (35c)

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

Uses ndata 30c.

⟨sum() *if* verbose 36b⟩≡ (35c)

```
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 31b and data 30b.

`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 37a⟩≡ (81c)

```
/*
 * 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 31b and nsym 33a.

⟨*function* acmp 37b⟩≡ (81c)

```
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()`[38c] function walks the same `Data`[30a] tree as `sum()`[35c], but instead of accumulating totals, it prints each node indented by call depth, showing the function name, time, and call count.

⟨*global* dflag 37c⟩≡ (81c)

```
int dflag;
```

⟨main() *(prof.c) command-line parsing* 38a⟩+≡ (31d) ◁32c 40b▷

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

⟨main() *(prof.c) if* dflag 38b⟩≡ (31d)

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

To handle recursive functions, `graph()` maintains a linked list of `Pc`[31c] 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.

The `pc` parameter threads a stack-allocated linked list through the recursive calls—each frame of `graph()` adds one `Pc` record on its own C stack frame, so there is no heap allocation. Suppose `main` calls `f`, and `f` recursively calls itself, which calls `g`. Walking into the nested `f` the chain looks like:

```
C stack of graph()           pc argument seen at each level
-------------------------    ------------------------------
graph(main, pc=nil)          nil
    graph(f, pc=&{main})           main -> nil
        graph(f, pc=&{f,main})     f -> main -> nil
            graph(g, ...)          <-- before descending, scan:
                                       new prgm = f? YES -> print "...",
                                       return (no recursion into inner f).
```

Two details are worth noticing. First, the list is built head-first: each frame prepends its own `prgm` with `lpc.next = pc`, so the most recent caller is at the head and the root is at the tail. Second, the recursion check happens *before* the descent but *after* the current node is printed—so the first occurrence of a recursive function still shows up with its time; only the re-entry is suppressed. Without this trick, a tight recursive loop in the traced program would make `graph()` loop forever walking the tree path that encodes the same cycle many times.

⟨*function* graph 38c⟩≡ (81c)

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

    ⟨graph() sanity check i 39a⟩
    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 32a, data 30b, indent() 39e, name() 39b, and rflag 40a.

⟨graph() *sanity check* i 39a⟩≡ (38c)

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

Uses ndata 30c.

⟨*function* name 39b⟩≡ (81c)

```
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 39c⟩≡ (81c)

```
int tabstop = 4;
```

Uses tabstop 39c.

⟨main()*(prof.c) adjust* tabstop 39d⟩≡ (31d)

```
s = getenv("tabstop");
if(s!=nil && strtol(s,0,0)>0)
    tabstop = strtol(s,0,0);
```

⟨*function* indent 39e⟩≡ (81c)

```
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 32a and tabstop 39c.

⟨*global* `rflag` 40a⟩≡ (81c)

```
int rflag;
```

⟨`main()` *(prof.c) command-line parsing* 40b⟩+≡ (31d) ◁38a

```
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 41⟩≡

```
#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 Entry point: `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)* 42⟩≡ (85)

```
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 43a⟩

    ⟨main() (tprof.c) sanity check argc and print usage 14c⟩

    /*
     * Read symbol table
     */
    ⟨main() (tprof.c) read symbol table 43g⟩

    /*
     * Read timing data
```

```
     */
    ⟨main() (tprof.c) read timing data 44⟩

    ⟨main() (tprof.c) displaying profile 45c⟩

    exits(nil);
}
```

⟨main() (tprof.c) other locals 43a⟩≡ (42)
```
long i, j, k, n;
char *name;
ulong tbase, sum;
long delta;
Symbol s;
struct COUNTER *cp;
```
Uses COUNTER 45b.

⟨main() (tprof.c) sanity check fd 43b⟩≡ (44 43g)
```
if(fd < 0)
    error(true, file);
```

⟨main() (tprof.c) sanity check f.type 43c⟩≡ (43g)
```
if (f.type == FNONE)
    error(false, "text file not an a.out");
```

⟨main() (tprof.c) sanity check d 43d⟩≡ (44)
```
if(d == nil)
    error(true, "stat");
```

⟨main() (tprof.c) sanity check n 43e⟩≡ (44)
```
if(n < 2)
    error(false, "data file too short");
```

⟨main() (tprof.c) sanity check data 43f⟩≡ (44)
```
if(data == nil)
    error(true, "malloc");
```

7.3 Reading the symbol table from the binary

The symbol table loading is almost identical to `syms()`[33b] 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 43g⟩≡ (42)
```
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 43b⟩

if (!crackhdr(fd, &f))
    error(true, "read text header");
⟨main() (tprof.c) sanity check f.type 43c⟩

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* 44⟩≡ (42)

```
file = smprint("/proc/%s/profile", argv[1]);
fd = open(file, OREAD);
⟨main() (tprof.c) sanity check fd 43b⟩
free(file);

d = dirfstat(fd);
⟨main() (tprof.c) sanity check d 43d⟩

n = d->length/sizeof(data[0]);
⟨main() (tprof.c) sanity check n 43e⟩
data = malloc(d->length);
⟨main() (tprof.c) sanity check data 43f⟩
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`[45b] records, sorted by time, and printed. The `PCRES` constant (8) matches the kernel's `LRESPROF` bucket size.

The parallel walk is the core of the algorithm and worth visualizing. Suppose the binary has three functions `main`, `foo`, `bar` at addresses `0x1000`, `0x1040`, `0x1080` respectively, and after sampling we get the following `data` array:

```
text segment (8-byte buckets, PCRES=8):

address    function   bucket index   sample count
--------   --------   ------------   ------------
0x1000     main       data[2]=8      8
0x1008     main       data[3]=2      2
0x1010     main       data[4]=0      0
...        main       ...            0
0x1040     foo        data[10]=30    30  <-- s.value moves here
0x1048     foo        data[11]=25    25
0x1050     foo        data[12]=5     5
...        foo        ...
0x1080     bar        data[18]=15    15  <-- s.value moves here
...        bar
```

The walk uses two indices: `i` iterates over symbols (via `textsym(&s, i)`), and `j` iterates over the `data` array. For each function, the inner `while` loop accumulates all buckets whose address falls before the *next* symbol's

address—that is, all buckets within the current function's range. The result is a per-function sum, which is recorded in the `cp` array if non-zero. After the walk:

```
cp[0] = {name="main", time=10}   (8+2)
cp[1] = {name="foo",  time=60}   (30+25+5)
cp[2] = {name="bar",  time=15}
```

The `+2` in the initial value of `j` accounts for the two reserved entries at the start of `data`: `data[0]` holds the total time and `data[1]` holds the time outside the text segment (used to compute `delta`).

⟨*constant PCRES (misc/tprof.c)* 45a⟩≡ (85)

```
#define PCRES 8
```

⟨*struct COUNTER (misc/tprof.c)* 45b⟩≡ (85)

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

⟨`main()` *(tprof.c) displaying profile* 45c⟩≡ (42)

```
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 45b, PCRES-50 45a, and mach 71c.

⟨*function compar (misc/tprof.c)* 46a⟩≡ (85)

```
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 45b.

7.6 Error management

⟨*function error (misc/tprof.c)* 46b⟩≡ (85)

```
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
```

8.1 Kernel profiling chicken-and-egg

Kernel profiling has different principles from user-space profiling because the profiler is itself kernel code. It cannot be attached to or paused from outside, so it has to self-sample: a high-priority timer interrupt fires on every tick, and the ISR records the interrupted PC into a histogram keyed by absolute kernel virtual address. There is no `pid` to address either, so enabling and disabling profiling goes through a dedicated control device `/dev/kpctl` (the `#K` device) that toggles a global flag the ISR checks each tick. Samples taken while the kernel is in a critical section with interrupts disabled are simply lost, which is the price of running inside the component being profiled.

Every modern OS solves this same chicken-and-egg in its own way. Linux folds kernel profiling into `perf record -a` ("-a" meaning all CPUs including kernel), reusing the same tool as user-space sampling. macOS uses `kdebug` and Instruments' "System Trace". Windows uses the ETW Kernel Logger. Solaris and illumos use `lockstat -k` or `dtrace -n 'profile-997'`. Plan 9 keeps `kprof` as a separate tool because all its other profilers assume a target process, and the kernel simply does not have one.

8.2 `kprof` vs `tprof`

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)* 47⟩≡ (81b)

```
void
main(int argc, char *argv[])
{
    fdt fd;
    Fhdr f;
```

```
    Dir *d;
    Biobuf outbuf;
    ulong *data;
    ⟨main() (kprof.c) other locals 48a⟩

    ⟨main() (kprof.c) sanity check argc and print usage 14d⟩

    /*
     * Read symbol table
     */
    ⟨main() (kprof.c) read symbol table 48b⟩

    /*
     * Read timing data
     */
    ⟨main() (kprof.c) read profile data 48c⟩

    ⟨main() (kprof.c) display profile data 49c⟩

    exits(0);
}
```

⟨main() (kprof.c) other locals 48a⟩≡ (47)

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

Uses COUNTER 45b.

⟨main() (kprof.c) read symbol table 48b⟩≡ (47)

```
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 48c⟩≡ (47)

```
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 49a⟩≡ (81b)
```
#define PCRES 8
```

⟨*struct* COUNTER 49b⟩≡ (81b)
```
struct COUNTER
{
    char  *name;  /* function name */
    long time;  /* ticks spent there */
};
```

The display logic mirrors `tprof`'s parallel walk over the symbol table and the data array, with two kprof-specific twists. First, it splits `data[0]` (total ticks) into in-kernel-text (ticks where the PC fell within a kernel function) and outside-kernel-text (ticks attributed to `data[1]`, which the kernel uses for "not in any text symbol"—typically idle or user-mode time on the CPU). Second, it converts each symbol's address to a bucket index relative to the kernel base (`mach->kbase`) rather than to a page-aligned user text address. The header line "`KTZERO PGSIZE`" lets the user double-check that the kernel binary they passed matches the running kernel: `mach->kbase` should agree with the first text symbol rounded down to a page boundary, otherwise the buckets would be misaligned and the per-function counts would be meaningless.

⟨`main()` *(kprof.c) display profile data* 49c⟩≡ (47)
```
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 45b, PCRES-65 49a, and mach 71c.

⟨*function error (misc/kprof.c)* 50a⟩≡ (81b)

```
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 50b⟩≡ (81b)

```
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 45b.

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

`TEvent` is a thin extension of the kernel's `Traceevent` struct (the raw event read from `/proc/trace`) with one extra field: `etime`, the elapsed time from this event to the next one for the same process. The kernel only timestamps each event individually; computing the duration of each state (running, ready, sleeping) is the user-space tool's job. The anonymous struct `Traceevent;` is a kencc extension that embeds all of `Traceevent`'s fields directly into `TEvent`, so `e->pid` reads the embedded `Traceevent.pid` without needing `e->traceevent.pid`.

A concrete example makes the raw event stream and the derived `etime` clearer. Suppose a process with pid 42 is scheduled, runs for 5 ms, sleeps on a channel for 3 ms, is made ready, runs another 2 ms, and exits. The kernel would emit this sequence on `/proc/trace`:

```
raw Traceevent stream (one entry per state change):

ts=1000  pid=42 etype=SRun    (start of first run)
ts=1005  pid=42 etype=SSleep  (blocks on channel)
ts=1008  pid=42 etype=SReady  (wakeup)
ts=1009  pid=42 etype=SRun    (scheduled again, 1ms later)
ts=1011  pid=42 etype=SDead   (exits)

after user-space post-processing into TEvent:

  [SRun, ts=1000, etime=5]     -> draw 5ms "run" bar
  [SSleep, ts=1005, etime=3]   -> draw 3ms "sleep" gap
  [SReady, ts=1008, etime=1]   -> 1ms "ready but not running"
  [SRun, ts=1009, etime=2]     -> draw 2ms "run" bar
  [SDead, ts=1011, etime=0]    -> terminates the row
```

The post-processing rule is simple: `etime[i] = ts[i+1] - ts[i]` for consecutive events belonging to the same pid. The reason the kernel does not precompute `etime` itself is that at the moment an event is emitted, the

kernel genuinely does not know when the next state change will happen—the process might sleep indefinitely. Only a tool that can see the whole (rolling) stream can close each interval.

This design is essentially the same as every kernel flight-recorder tool across OSes, but spelled differently:

OS	kernel source	format	user-space consumer
\plan	/proc/trace	binary	/bin/trace
Linux	ftrace ring buffer	text/raw	trace-cmd, KernelShark
Linux	perf_event ring	binary	perf sched, perfetto
Linux	LTTng ust/kernel	CTF	babeltrace, tracecompass
macOS/iOS	os_signpost + ktrace	signpost	Instruments
Solaris	DTrace aggregations	varies	dtrace(1)
Windows	ETW sessions	ETL	wpa.exe, PerfView

All of these share three design choices that `trace` also makes: (1) the kernel writes into a bounded circular buffer so a slow reader cannot back-pressure the scheduler; (2) events carry a timestamp but not a duration, because duration is a user-space concept that depends on which event you pair with which; (3) the event type is a small enum, so the producer side is cheap enough to run on every context switch. `trace` is thus the smallest reasonable point in this design space: one enum (`Sstate`), one ring buffer, one GUI. Every other tool in the table adds features—but also thousands of lines of plumbing.

⟨*struct* `TEvent` 52⟩≡ (84c)

```
struct TEvent {
    Traceevent;
    vlong etime; /* length of block to draw */
};
```

`Task` is the per-process state that `trace` maintains. It collects:

- identification: `pid` and `name`;
- the event ring buffer: `events` (size `nevents`);
- statistics for the running state: `runtime` (total running time), `runmax` (longest single run), `runthis` (current run), `runs` (number of times scheduled);
- per-event-type counters: `tevents[]` (e.g., how many `SSleep` vs `SRun` events occurred);
- display state: `tstart` (when this row started) and `total` (total wall-clock time observed).

The event ring buffer is fixed-size (`Nevents=1024`), so old events are overwritten when the buffer fills—which keeps `trace` bounded in memory regardless of how long it runs.

The per-task `events` array is used as a circular buffer indexed modulo `Nevents`. `nevents` counts the total number of events ever stored (not the current occupancy), so the slot where the next event lands is `events[nevents \% Nevents]`, and the oldest still-live event sits at `events[(nevents - Nevents) \% Nevents]` once the buffer has wrapped. Picture a 1024-slot ring for a busy task that has already received 1030 events:

```
events[]  (Nevents = 1024 slots, arranged circularly)

          slot 0     slot 5     slot 6        slot 1023
           ...          |          |              ...
           v            v          v               v
        [1024]------[1029]------[1030?]--------[1023]
          ^            ^          ^               ^
          |            |          |               |
```

```
        wrote "1024th"  "1029th"  next write        "1023rd"
        over old [0]    over [5]  will overwrite      (still
        (already        (already   slot 6 when        alive,
         overwritten)   overwritten)  nevents=1030)   oldest)

  head = nevents % Nevents      = 6
  tail = (nevents - Nevents) % Nevents = 7
```

This is the same flight-recorder discipline Linux's `perf_event` and ftrace use, but with the indirection stripped out: there is no separate head/tail pair, no producer/consumer synchronization, and no reader catching up. The drawing code simply iterates the most recent `min(nevents, Nevents)` slots in reverse when redrawing the timeline, and accepts that anything older than `Nevents` events ago is gone forever—which is fine for a visualization tool because screen pixels run out long before 1024 events do.

⟨*struct* `Task` 53a⟩≡ (84c)
```
  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)` 53b⟩≡ (84c)
```
  enum {
      Nevents = 1024,
      Ncolor = 6,
      K = 1024,
  };
```

`trace` is interactive: the user can zoom in and out on the timeline. Each zoom level is a fixed entry in the `scales` table, ranging from 500 μs/screen to 1000 s/screen. Each entry says how wide the screen represents in time (`scale`), how often to draw a major tick mark (`bigtics`) and a minor one (`littletics`), and how long to sleep between redraws (`sleep` in milliseconds—larger scales need less frequent updates). Pressing `+` / `-` in the `trace` window cycles through this table.

⟨*struct* `scale` 53c⟩≡ (84c)
```
  struct scale {
      vlong scale;
      vlong bigtics;
      vlong littletics;
      int sleep;
  };
```

⟨*global* `scales` 53d⟩≡ (84c)
```
  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 83c, S-55 83d, US-53 83b, and scale 53c 71k.

9.2 Entry point: main()

trace's entry point is threadmain()[55a] (it is a libthread program because it needs to multiplex the GUI event loop, the trace reader, and the redraw loop). The startup sequence is:

1. Parse command-line flags (the -d device override, the -w new-window flag, the -t trigger PID, and the list of PIDs to enable tracing on).
2. For each PID on the command line, write trace 1 to its /proc/<pid>/ctl to enable kernel-side tracing.
3. Call drawtrace() which initializes the GUI (window, fonts, colors), spawns the trace reader thread, and enters the redraw/event loop.

The kernel is doing the heavy lifting: hooks in ready(), runproc(), sleep(), and pexit() generate events into a circular buffer, which the user-space tool then drains through /proc/trace.

⟨*global verbose (misc/trace.c)* 54a⟩≡ (84c)
```
static int verbose;
```

⟨*global* tasks 54b⟩≡ (84c)
```
Task *tasks;
```

⟨*global* cols 54c⟩≡ (84c)
```
static Image *cols[Ncolor][4];
```

Uses Ncolor-61 53b.

⟨*global* profdev 54d⟩≡ (84c)
```
char*profdev = "/proc/trace";
```

Uses profdev 54d.

⟨*function* usage 54e⟩≡ (84c)
```
static void
usage(void)
{
    fprint(2, "Usage: %s [-d profdev] [-w] [-v] [-t triggerproc] [processes]\n", argv0);
    exits(nil);
}
```

⟨*function* `threadmain` 55a⟩≡ (84c)

```
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()` 63, `newwin` 83h, `profdev` 54d, `timeconv()` 66, and `triggerproc` 84c.

Each process gets a different color, cycling through six. `mkcol()`[87a] allocates four shades for each color: a background (light), a fill (medium), a border (dark), and a separator. The four shades let `trace` distinguish the running, ready, and sleeping states for the same process in the same row without confusion. We can see the four shades for the six colors below:

⟨*function* `mkcol` 55b⟩≡ (84c)

```
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 56a⟩≡ (84c)

```
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` 84c, `blue` 84c, `bottommargin` 83l, `fg` 84c, `green` 84c, `grey` 84c, `red` 84c, `tinyfont` 84c, and `topmargin` 83k.

⟨*function* time2x 56b⟩≡ (84c)

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

`redraw()`[89b] redraws the entire timeline. The main idea is to map time to X coordinates: a fixed window width (in pixels) represents a fixed time span (`scales[scaleno].scale`), so each pixel covers `ppp = period / Width` nanoseconds. The `time2x` macro converts a timestamp to the corresponding X coordinate. For each task, the function:

1. Scrolls the existing image left to make room for new events on the right (via `draw()X` from the image to itself with an offset);
2. Discards events that have scrolled off the left edge;
3. Iterates through the remaining events, drawing a colored horizontal bar from each `SRun` timestamp to its `etime` end timestamp.

The scrolling trick (rather than redrawing every event from scratch) is what keeps `trace` responsive even at high event rates—only the new portion at the right edge needs to be redrawn each frame.

⟨*function* redraw 56c⟩≡ (84c)

```
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 83j, Ncolor-61 53b, Width 83i, bg 84c, blue 84c, bottommargin 83l, fg 84c, green 84c, lineht 83m, now 84c, ntasks 84c, p 114a, paused 84c, prevts 84c, red 84c, scale 53c 71k, scales 53d, tasks 54b, time2x-63 56b, tinyfont 84c, topmargin 83k, and triggerproc 84c.

`newtask()`[61a] allocates a new `Task` entry for a previously unseen PID. The process name is read from `/proc/<pid>/status` (the first space-separated token, which is the program name that `exec` was called with).

When `newwin` is set, the window is resized to add a row for the new process; otherwise the existing window's height is divided evenly among all tasks.

⟨*function* `newtask` 61a⟩≡ (84c)

```
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` 83j, `Width` 83i, `newwin` 83h, `ntasks` 84c, `p` 114a, `prevts` 84c, `tasks` 54b, and `wctlfd` 83n.

`doevent()`[61b] is the heart of the trace processing logic. For each new event from the kernel, it: (1) increments the per-event-type counter `t->tevents[etype]`, (2) appends the event to the task's event ring, (3) for state-leaving events (`SSleep`, `SYield`, `SReady`, `SSlice`), looks back for the matching `SRun` event and computes the run duration, updating `runtime`, `runmax`, and `runthis`, and (4) for `SDead`, removes the task from the task list. The look-back loop is the key: when we see "the process left the run state", we want to know how long it had been running. The kernel reports each transition individually with a timestamp, and the user-space tool pairs them up.

⟨*function* `doevent` 61b⟩≡ (84c)

```
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 83j, Width 83i, event 84a, newwin 83h, ntasks 84c, prevts 84c, schedstatename 84b, tasks 54b, and wctlfd 83n.

`drawtrace()`[63] is the main GUI loop. After initializing the window, mouse, keyboard, and color palette, it enters an `alt()` loop that multiplexes:

- mouse events (channel 0): unused for now;
- resize events (channel 1): re-attach the window;
- keyboard events (channel 2): pause/resume (`p`), zoom in/out (`+`/`-`), reset stats (`r`), quit (`q`);
- the default branch (no channel ready): read pending events from `/proc/trace` via `read()X`, dispatch each to `doevent()`, and call `redraw()`.

The default-branch trick (a `CHANNOBLK Alt` entry) makes the `alt()` return immediately if no input event is pending, keeping the redraw loop running at the rate set by `scales[scaleno].sleep`. This is how `trace` combines event-driven input handling with periodic time-driven redraw.

⟨*function* `drawtrace` 63⟩≡ (84c)

```
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 83j, Nevents-60 53b, Width 83i, doevent() 61b, eventbuf 83p, nevents 83o, newtask() 61a, newwin 83h, now 84c, ntasks 84c, paused 84c, prevts 84c, profdev 54d, scales 53d, schedstatename 84b, tasks 54b, and wctlfd 83n.

`timeconv()`[66] is a custom format handler installed via `fmtinstall('t', timeconv)` in `threadmain()`, so that `print("%t", t)` anywhere in `trace` formats a `vlong` nanosecond value into the most readable unit (`ns`

/ μs / ms / s). The trick is the unit selection cascade: pick seconds if t > 1s, milliseconds if t > 1ms, etc., and always print three decimal digits in the chosen unit. The OneRound / MilliRound additions implement banker-style rounding so that boundaries like 999.9ms do not silently truncate to 999ms.

⟨*function* timeconv 66⟩≡ (84c)

```
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 83c, MilliRound-59 83g, OneRound-58 83g, S-55 83d, and US-53 83b.

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

`stats` uses two main data structures: `Graph` (one per displayed metric) and `Machine` (one per monitored host). Each `Graph` has its own ring buffer of historical samples plus two function pointers: `newvalue` reads a new sample from the `Machine` (e.g., "read context-switch counter"), and `update` redraws the graph with the new value. This function-pointer approach means `stats` can support arbitrary metrics without a giant switch statement—the metric type is encoded in which functions a `Graph` points to.

The `data` array inside `Graph`[68b] is a sliding window of samples sized to the graph's pixel width: `ndata` equals the width of `r` in pixels, so each stored sample maps directly to one column on screen. On every tick, `stats` shifts the window left by one (dropping the oldest sample) and appends the new reading at the right edge, producing the classic right-to-left scrolling behavior you see in `top`, Task Manager's CPU graph, Activity Monitor, and `htop`:

```
tick N:
  data[]:  [s0 | s1 | s2 | s3 | s4 | s5 | s6 | s7 | s8 ]
            |                                        ^
            |                                        |
        oldest (left edge of graph)        newest (right edge)

tick N+1 (shift left, append fresh sample s9):
  data[]:  [s1 | s2 | s3 | s4 | s5 | s6 | s7 | s8 | s9 ]
                                                     ^
                                                  appended
```

Because `ndata` is tied to pixel width rather than wall-clock time, resizing the window transparently re-allocates `data` and changes how much history is visible without changing the sampling rate. The rolling maximum used to scale the Y axis is recomputed on each tick from whatever values are still in the window—so old spikes

eventually scroll off the left edge and let the Y axis recover to a tighter scale, whereas a static high-water mark would leave the graph permanently compressed after a single burst.

⟨*constant* MAXNUM 68a⟩≡ (106)
```
// a GUI system monitoring tool

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

⟨*struct* Graph 68b⟩≡ (106)
```
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) 68c⟩≡ (106)
```
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,
};
```

A `Machine` is the connection to one host being monitored. It holds open file descriptors for each kernel pseudo-file (`#c/sysstat`, `#c/swap`, the ethernet stats, the battery, the temperature sensor, etc.) and the most recent values read from each. The `prev*` fields hold the previous reading so that the graphs can compute deltas (e.g., context switches per second = current - previous, since the kernel exposes monotonic counters). Multiple `Machine`s can be active at once when `stats` is invoked with several remote hosts via `srv` / `mount` over 9P.

⟨*struct* Machine 68d⟩≡ (106)
```
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 69a⟩≡ (106)

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

⟨*enum* _anon_ (misc/stats.c)3 69b⟩≡ (106)

```
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 69c⟩≡ (106)

```
enum Menu2
{
    Mbattery,
    Mcontext,
    Mether,
    Methererr,
    Metherin,
    Metherout,
    Mfault,
    Midle,
```

```
    Minintr,
    Mintr,
    Mload,
    Mmem,
    Mswap,
    Msyscall,
    Mtlbmiss,
    Mtlbpurge,
    Msignal,
    Mtemp,
    Nmenu2,
};
```

⟨*global* menu2str 70a⟩≡ (106)

```
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 69c.

⟨*global* menu2 70b⟩≡ (106)

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

Uses menu2str 70a.

⟨*global* present 70c⟩≡ (106)

```
int present[Nmenu2];
```

Uses Nmenu2-49 69c.

`newvaluefn` is the dispatch table that maps each menu entry (metric type) to its sample-reading function. When `stats` adds a graph for, say, `Mcontext` (context switches), the graph's `newvalue` field is set to `contextval`, which knows how to read the context-switch counter from `/dev/sysstat` and compute the delta from the previous reading. Each `*val` function follows the same pattern: read the current value, compute the difference from the saved previous value, save the new value as previous, and return both the current value and the rolling maximum (used to scale the graph's Y axis).

⟨*global* newvaluefn 70d⟩≡ (106)

```
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 69c, batteryval() 100b, contextval() 97c, ethererrval() 100a, etherinval() 99d, etheroutval() 99e, etherval() 99c, faultval() 98b, idleval() 99a, inintrval() 99b, intrval() 97d, loadval() 98e, memval() 97a, signalval() 100c, swapval() 97b, syscallval() 98a, tempval() 100d, tlbmissval() 98c, and tlbpurgeval() 98d.

⟨*global cols (misc/stats.c)* 71a⟩≡ (106)
```
Image *cols[Ncolor][3];
```
Uses Ncolor-23 69b.

⟨*global* graph 71b⟩≡ (106)
```
Graph *graph;
```

⟨*global* mach 71c⟩≡ (106)
```
Machine *mach;
```

⟨*global* mediumfont 71d⟩≡ (106)
```
Font *mediumfont;
```

⟨*global* mysysname 71e⟩≡ (106)
```
char *mysysname;
```

⟨*global* argchars 71f⟩≡ (106)
```
char argchars[] = "8bceEfiImlnpstwz";
```
Uses argchars 71f.

⟨*global* pids 71g⟩≡ (106)
```
int pids[NPROC];
```
Uses NPROC-22 69a.

⟨*global* parity 71h⟩≡ (106)
```
int  parity; /* toggled to avoid patterns in textured background */
```

⟨*global* nmach 71i⟩≡ (106)
```
int nmach;
```

⟨*global* ngraph 71j⟩≡ (106)
```
int ngraph; /* totaly number is ngraph*nmach */
```

⟨*global* scale 71k⟩≡ (106)
```
double scale = 1.0;
```
Uses scale 53c 71k.

⟨*global* logscale 71l⟩≡ (106)
```
int logscale = 0;
```
Uses logscale 71l.

⟨*global* `ylabels` 72a⟩≡ (106)
```
int ylabels = 0;
```
Uses `ylabels` 72a.

⟨*global* `oldsystem` 72b⟩≡ (106)
```
int oldsystem = 0;
```
Uses `oldsystem` 72b.

⟨*global* `sleeptime` 72c⟩≡ (106)
```
int  sleeptime = 1000;
```
Uses `sleeptime` 72c.

⟨*global* `procnames` 72d⟩≡ (106)
```
char *procnames[NPROC] = {"main", "mouse"};
```
Uses `NPROC-22` 69a.

10.2 Entry point: `main()`

⟨*function main (misc/stats.c)* 72e⟩≡ (106)
```
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);
        }
    }
```

The main loop is short and elegant: read all machines, update all graphs, sleep for one tick, repeat. The `graph[i].newvalue` / `graph[i].update` indirection through function pointers means the loop is the same regardless of which metrics the user selected—each graph knows how to read its own value and how to draw it. This is the strength of the function-pointer approach for dispatch tables: adding a new metric just requires

writing a `*val` function and adding an entry to `newvaluefn`, no changes to the main loop. The `parity` toggle alternates between two background shades each frame; combined with the textured backgrounds defined in `colinit()`[87b], this avoids visible diagonal patterns in the graph background that would otherwise distract from the data.

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`.

The "observe I/O without modifying the program" problem has several well-known solutions, and comparing `iostats` to them is instructive because `iostats` solves it without any kernel-side support at all:

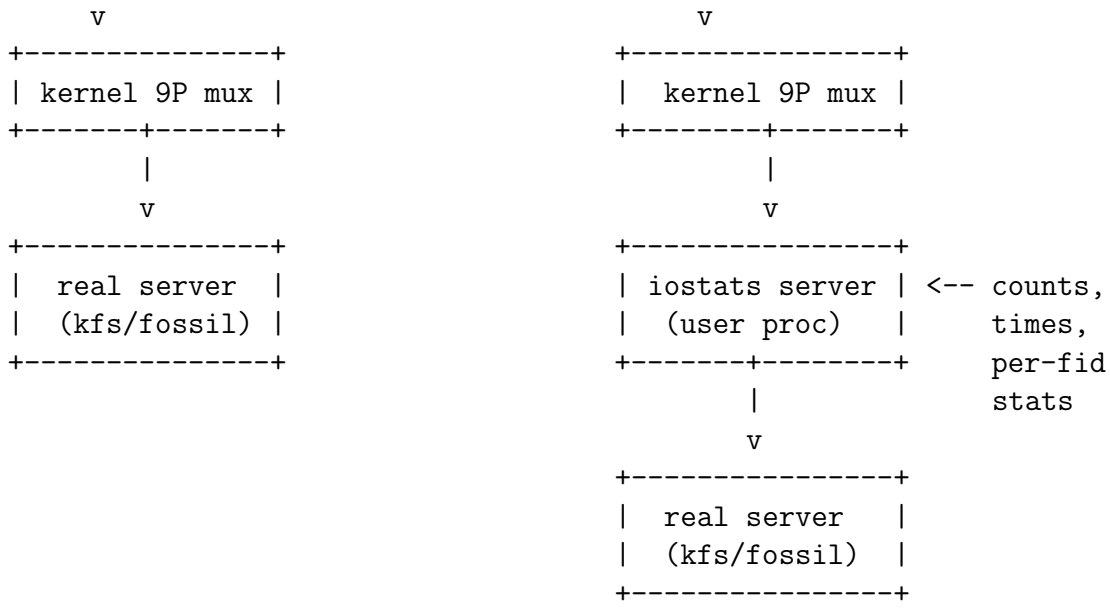
```
tool                  mechanism              where the hook lives
-------------------   --------------------   --------------------
\plan iostats         9P proxy file server   user-space namespace
Linux strace          ptrace(PTRACE_SYSCALL) kernel trap trampoline
Linux ltrace          breakpoints on PLT     user-space dyld hooks
Solaris/macOS dtruss  DTrace syscall probes  kernel probes
Linux bpftrace -e 't' tracepoint + BPF       kernel tracepoint
Windows procmon       filesystem mini-filter kernel filter driver
LD_PRELOAD wrappers   function interposition user-space linker
```

Every entry in this table except `iostats` and the LD_PRELOAD trick needs help from the kernel: either a debugger hook (`ptrace`), a tracing framework (DTrace, bpftrace, ETW), or an in-kernel filter driver. `iostats` cheats in the cleanest way possible: because on Plan 9 every I/O is a 9P message and every filesystem is a user-space server accessed via the namespace, you can put a *new user process* between the client and the real server and be done. No kernel code, no debugger tricks, no function-interposition voodoo—just `bind` the proxy into the target's namespace and `exec` the program. This is the same move that makes Plan 9's `rio` a window system, `import` a remote-mount, and `fuse2p` a FUSE bridge: structural interposition replaces ad-hoc hooks.

Concretely, `iostats ls /tmp` reshapes the target's namespace in one `bind`. Before the child `exec`s, its mount table looks like the leftmost diagram; after the interposition `bind`, it looks like the rightmost:

```
Before iostats:                    After 'iostats ls /tmp':

 child process                      child process
     |                                  |
     | fd = open("/tmp/x")              | fd = open("/tmp/x")
```



Every `Tread`, `Twrite`, `Topen` issued by the child now traverses the `iostats` process, which logs it into a `Rpc` slot (one per 9P message type) and a `Fid` entry (one per open file descriptor), forwards the message verbatim to the real server, copies the reply back, and updates the byte and latency counters. The child cannot tell the difference because the wire protocol is unchanged—`iostats` speaks the same 9P on both sides, like an HTTP reverse proxy that just happens to count requests. When the child exits, the post-mortem report walks `stats->rpc[]` and the `File` tree (built from the observed `Twalk` messages, not a real directory listing) and prints both per-message-type totals and per-file byte counts.

The two accounting tables behind the report are worth seeing side by side. `Stats.rpc[]` is indexed by 9P message type—one slot per `T`-message—and records call count, cumulative time, min/max latency, and total bytes in and out. `fhash[]` is a small closed-chain hash table (size `FHASHSIZE=64`) indexed by fid number, holding per-open-file `Fid`[110e] records that in turn point into the `File`[111a] tree built from observed `Twalk`s:

```
stats->rpc[]                          fhash[]  (FHASHSIZE=64)
(one slot per 9P T-message)

  index    name       count  time     slot  chain (fid % 64)
  -----    ----       -----  ----     ----  ----------------
  Tversion "version"    1     3us     [0]-> Fid{0}--Fid{64}--nil
  Tattach  "attach"     1    12us     [1]-> nil
  Twalk    "walk"      47   830us     [2]-> Fid{2}--Fid{66}--nil
  Topen    "open"      12   210us     [3]-> nil
  Tread    "read"     128    41ms     ...
  Twrite   "write"     33     8ms     [n]-> Fid{n}-----+
  Tclunk   "clunk"     12    40us                      |
  ...                                                  v
                                              File{"/tmp/x",
                                                   qid, parent ->
                                                   File{"/tmp",
                                                        parent -> root}}
```

The two tables are orthogonal: `Stats.rpc[]` answers "how often and how slow is each 9P message type?" while the `fhash`/`File` structure answers "which files received how many bytes?" Because fids are reused across the lifetime of a connection—`Tclunk` frees the `Fid` entry and the number can be reassigned—the per-file byte totals live in `File` (keyed by `qid.path`) rather than in `Fid`, so a short-lived open on the same file gets its bytes added

to the same `File` node. This matches the design of every filesystem monitor (Linux's `iotop`, macOS's `fs_usage`, Windows procmon): message-type histograms tell you *what* is slow, per-path counters tell you *where*, and you need both to distinguish a hot file from a hot syscall.

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 81a⟩≡
```
#include <u.h>
#include <libc.h>

⟨global output(time.c) 27a⟩

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

⟨function error(time.c) 28a⟩
⟨function main(time.c) 26⟩
⟨function add(time.c) 27b⟩
⟨function notifyf(time.c) 28c⟩
```

A.1.2 misc/kprof.c

⟨misc/kprof.c 81b⟩≡
```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <mach.h>

⟨constant PCRES 49a⟩

⟨struct COUNTER 49b⟩

⟨function error (misc/kprof.c) 50a⟩

⟨function compar 50b⟩
⟨function main (misc/kprof.c) 47⟩
```

A.1.3 misc/prof.c

⟨misc/prof.c 81c⟩≡
```
#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 30a⟩

⟨struct Pc 31c⟩

⟨struct Acc 31a⟩

⟨global data 30b⟩
⟨global acc 31b⟩
⟨global ms 35a⟩
⟨global nsym 33a⟩
⟨global ndata 30c⟩
⟨global dflag 37c⟩
⟨global rflag 40a⟩
⟨global bout 32a⟩
⟨global tabstop 39c⟩
⟨global verbose 32b⟩

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) 31d⟩

⟨function swapdata 34a⟩

⟨function acmp 37b⟩

⟨function syms 33b⟩

⟨function datas 33e⟩

⟨function name 39b⟩

⟨function graph 38c⟩
⟨function symind 37a⟩

⟨function sum 35c⟩

⟨function plot 34e⟩

⟨function indent 39e⟩

⟨global trans 32e⟩

⟨function defaout 32d⟩
```

Uses Acc 31a, Data 30a, and Pc 31c.

A.1.4 misc/trace.c

⟨*function* NS 83a⟩≡ (84c)
```
#define NS(x) ((vlong)x)
```

⟨*function* US 83b⟩≡ (84c)
```
#define US(x) (NS(x) * 1000ULL)
```

⟨*function* MS 83c⟩≡ (84c)
```
#define MS(x) (US(x) * 1000ULL)
```

⟨*function* S 83d⟩≡ (84c)
```
#define S(x) (MS(x) * 1000ULL)
```

⟨*function* numblocks 83e⟩≡ (84c)
```
#define numblocks(a, b) (((a) + (b) - 1) / (b))
```

⟨*function* roundup 83f⟩≡ (84c)
```
#define roundup(a, b) (numblocks((a), (b)) * (b))
```

⟨*enum* _anon_ 83g⟩≡ (84c)
```
enum {
    OneRound = MS(1)/2LL,
    MilliRound = US(1)/2LL,
};
```
Uses MS-54 83c and US-53 83b.

⟨*global* newwin 83h⟩≡ (84c)
```
int newwin;
```

⟨*global* Width 83i⟩≡ (84c)
```
int Width = 1000;
```
Uses Width 83i.

⟨*global* Height 83j⟩≡ (84c)
```
int Height = 100;  // Per task
```
Uses Height 83j.

⟨*global* topmargin 83k⟩≡ (84c)
```
int topmargin = 8;
```
Uses topmargin 83k.

⟨*global* bottommargin 83l⟩≡ (84c)
```
int bottommargin = 4;
```
Uses bottommargin 83l.

⟨*global* lineht 83m⟩≡ (84c)
```
int lineht = 12;
```
Uses lineht 83m.

⟨*global* wctlfd 83n⟩≡ (84c)
```
int wctlfd;
```

⟨*global* nevents 83o⟩≡ (84c)
```
int nevents;
```

⟨*global* eventbuf 83p⟩≡ (84c)
```
Traceevent *eventbuf;
```

⟨*global* event 84a⟩≡ (84c)
```
TEvent *event;
```

⟨*global* schedstatename 84b⟩≡ (84c)
```
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 84c⟩≡
```
#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 83a⟩
⟨*function* US 83b⟩
⟨*function* MS 83c⟩
⟨*function* S 83d⟩

⟨*function* numblocks 83e⟩
⟨*function* roundup 83f⟩

⟨*enum* _anon_ 83g⟩

```
typedef struct TEvent TEvent;
typedef struct Task Task;
```
⟨*struct* TEvent 52⟩

⟨*struct* Task 53a⟩

⟨*enum* _anon_ (misc/trace.c) 53b⟩

```
vlong now, prevts;

⟨global newwin 83h⟩
⟨global Width 83i⟩
⟨global Height 83j⟩
⟨global topmargin 83k⟩
⟨global bottommargin 83l⟩
⟨global lineht 83m⟩
⟨global wctlfd 83n⟩
⟨global nevents 83o⟩
⟨global eventbuf 83p⟩
⟨global event 84a⟩

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

⟨global schedstatename 84b⟩

⟨struct scale 53c⟩

⟨global scales 53d⟩

int ntasks, triggerproc, paused;
⟨global verbose (misc/trace.c) 54a⟩
⟨global tasks 54b⟩
⟨global cols 54c⟩
static Font *mediumfont, *tinyfont;
Image *grey, *red, *green, *blue, *bg, *fg;
⟨global profdev 54d⟩

⟨function usage 54e⟩

⟨function threadmain 55a⟩

⟨function mkcol 55b⟩

⟨function colinit 56a⟩

⟨function time2x 56b⟩

⟨function redraw 56c⟩

⟨function newtask 61a⟩

⟨function doevent 61b⟩

⟨function drawtrace 63⟩

⟨function timeconv 66⟩
```

Uses TEvent 84c and Task 84c.

A.1.5 misc/tprof.c

```
⟨misc/tprof.c 85⟩≡
  #include <u.h>
  #include <libc.h>
  #include <bio.h>
  #include <mach.h>
```

⟨*constant PCRES (misc/tprof.c)* 45a⟩

⟨*struct COUNTER (misc/tprof.c)* 45b⟩

⟨*function error (misc/tprof.c)* 46b⟩

⟨*function compar (misc/tprof.c)* 46a⟩
⟨*function main (misc/tprof.c)* 42⟩

A.1.6 misc/stats.c

⟨*function* killall 86a⟩≡ (106)
```
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 69a and pids 71g.

⟨*function* emalloc 86b⟩≡ (106)
```
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() 86a.

⟨*function* erealloc 86c⟩≡ (106)
```
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() 86a.

⟨*function* estrdup 86d⟩≡ (106)
```
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() 86a.

⟨*function mkcol (misc/stats.c)* 87a⟩≡ (106)

```
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)* 87b⟩≡ (106)

```
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 87c⟩≡ (106)

```
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 88a⟩≡ (106)

```
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 69b and Ysqueeze-24 69b.

⟨*function* paritypt 88b⟩≡ (106)

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

Uses parity 71h.

⟨*function* datapoint 88c⟩≡ (106)

```
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 69b, logscale 71l, and scale 53c 71k.

⟨*function* drawdatum 89a⟩≡ (106)

```
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 69b, datapoint() 88c, and paritypt() 88b.

⟨*function redraw (misc/stats.c)* 89b⟩≡ (106)

```
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() 89a and paritypt() 88b.

⟨*function* update1 89c⟩≡ (106)

```
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()` 89a, `logscale` 71l, and `scale` 53c 71k.

⟨*function* `readnums` 90a⟩≡ (106)

```
/* 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` 90b⟩≡ (106)

```
/* 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() 86a.

⟨*function* connect9fs 91a⟩≡ (106)

```
/*
 * 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 91b⟩≡ (106)

```
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 92⟩≡ (106)

```
/*
 * 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() 91b and oldsystem 72b.

⟨*function* readswap 93a⟩≡ (106)

```
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() 90a.

⟨*function* shortname 93b⟩≡ (106)

```
char*
shortname(char *s)
{
    char *p, *e;

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

Uses estrdup() 86d.

⟨*function* ilog10 93c⟩≡ (106)

```
int
ilog10(uvlong j)
{
    int i;

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

⟨*function* initmach 93d⟩≡ (106)

```
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 68a, Maxmem-3 68c, Maxswap-5 68c, connectexportfs() 92, estrdup() 86d, ilog10() 93c, loadbuf() 87c, mysysname 71e, readnums() 90a, readswap() 93a, and shortname() 93b.

⟨*global* catchalarm 95a⟩≡ (106)

```
jmp_buf catchalarm;
```

⟨*function* alarmed 95b⟩≡ (106)

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

Uses catchalarm 95a.

⟨*function* needswap 95c⟩≡ (106)

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

Uses Mmem-42 69c, Mswap-43 69c, and present 70c.

⟨*function* needstat 95d⟩≡ (106)

```
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 69c, Mfault-37 69c, Midle-38 69c, Minintr-39 69c, Mintr-40 69c, Mload-41 69c, Msyscall-44 69c, Mtlbmiss-45 69c, Mtlbpurge-46 69c, and present 70c.

⟨*function* needether 95e⟩≡ (106)

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

Uses Mether-33 69c, Methererr-34 69c, Metherin-35 69c, Metherout-36 69c, and present 70c.

⟨*function* needbattery 96a⟩≡ (106)

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

Uses Mbattery-31 69c and present 70c.

⟨*function* needsignal 96b⟩≡ (106)

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

Uses Msignal-47 69c and present 70c.

⟨*function* needtemp 96c⟩≡ (106)

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

Uses Mtemp-48 69c and present 70c.

⟨*function* readmach 96d⟩≡ (106)

```
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() 95b, catchalarm 95a, eresized() 105a, estrdup() 86d, loadbuf() 87c, needbattery() 96a, needether() 95e, needsignal() 96b, needstat() 95d, needswap() 95c, needtemp() 96c, readnums() 90a, readswap() 93a, and shortname() 93b.

⟨*function* memval 97a⟩≡ (106)

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

Uses Maxmem-3 68c and Mem-2 68c.

⟨*function* swapval 97b⟩≡ (106)

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

Uses Maxswap-5 68c and Swap-4 68c.

⟨*function* contextval 97c⟩≡ (106)

```
    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 68c and sleeptime 72c.

⟨*function* intrval 97d⟩≡ (106)

```
    /*
     * 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 68c and sleeptime 72c.

⟨*function* syscallval 98a⟩≡ (106)

```
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 68c and sleeptime 72c.

⟨*function* faultval 98b⟩≡ (106)

```
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 68c and sleeptime 72c.

⟨*function* tlbmissval 98c⟩≡ (106)

```
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 68c and sleeptime 72c.

⟨*function* tlbpurgeval 98d⟩≡ (106)

```
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 68c and sleeptime 72c.

⟨*function* loadval 98e⟩≡ (106)

```
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 68c.

⟨*function* idleval 99a⟩≡ (106)

```
void
idleval(Machine *m, uvlong *v, uvlong *vmax, int)
{
    *v = m->devsysstat[Idle]/m->nproc;
    *vmax = 100;
}
```

Uses Idle-14 68c.

⟨*function* inintrval 99b⟩≡ (106)

```
void
inintrval(Machine *m, uvlong *v, uvlong *vmax, int)
{
    *v = m->devsysstat[InIntr]/m->nproc;
    *vmax = 100;
}
```

Uses InIntr-15 68c.

⟨*function* etherval 99c⟩≡ (106)

```
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 68c, Out-18 68c, and sleeptime 72c.

⟨*function* etherinval 99d⟩≡ (106)

```
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 68c and sleeptime 72c.

⟨*function* etheroutval 99e⟩≡ (106)

```
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 68c and sleeptime 72c.

⟨*function* `ethererrval` 100a⟩≡ (106)

```
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` 68c and `sleeptime` 72c.

⟨*function* `batteryval` 100b⟩≡ (106)

```
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` 100c⟩≡ (106)

```
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` 72c.

⟨*function* `tempval` 100d⟩≡ (106)

```
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` 72c.

⟨*function* addgraph 101a⟩≡ (106)

```
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 69b, Opwid-27 69b, emalloc() 86b, mach 71c, menu2str 70a, newvaluefn 70d, ngraph 71j, nmach 71i, present 70c, and update1() 89c.

⟨*function* dropgraph 101b⟩≡ (106)

```
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 69b, emalloc() 86b, killall() 86a, menu2str 70a, ngraph 71j, nmach 71i, and present 70c.

⟨*function* addmachine 102a⟩≡ (106)

```
    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() 86c, initmach() 93d, mach 71c, ngraph 71j, and nmach 71i.

⟨*function* labelstrs 102b⟩≡ (106)

```
    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 69b, Nlab-28 69b, logscale 71l, and scale 53c 71k.

⟨*function* labelwidth 102c⟩≡ (106)

```
    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 69b, Nlab-28 69b, labelstrs() 102b, ngraph 71j, and nmach 71i.

⟨*function* resize 103⟩≡ (106)

```
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 69b, Labspace-25 69b, Lx-30 69b, Nlab-28 69b, erealloc() 86c, label() 88a, labelstrs() 102b, labelwidth() 102c, mach 71c, ngraph 71j, nmach 71i, paritypt() 88b, and ylabels 72a.

⟨*function* eresized 105a⟩≡ (106)
```
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() 86a and resize() 103.

⟨*function* mouseproc 105b⟩≡ (106)
```
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 69c, Opwid-27 69b, addgraph() 101a, dropgraph() 101b, menu2 70b, menu2str 70a, ngraph 71j, present 70c, and resize() 103.

⟨*function* startproc 105c⟩≡ (106)
```
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() 86a, pids 71g, and procnames 72d.

⟨misc/stats.c 106⟩≡

```
#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 68a⟩

```
typedef struct Graph Graph;
typedef struct Machine Machine;
```

⟨*struct* Graph 68b⟩

⟨*enum* _anon_ (misc/stats.c) 68c⟩

⟨*struct* Machine 68d⟩

⟨*enum* _anon_ (misc/stats.c)2 69a⟩

⟨*enum* _anon_ (misc/stats.c)3 69b⟩

⟨*enum* Menu2 69c⟩

⟨*global* menu2str 70a⟩

```
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 70b⟩

⟨*global* `present` 70c⟩
⟨*global* `newvaluefn` 70d⟩

⟨*global cols (misc/stats.c)* 71a⟩
⟨*global* `graph` 71b⟩
⟨*global* `mach` 71c⟩
⟨*global* `mediumfont` 71d⟩
⟨*global* `mysysname` 71e⟩
⟨*global* `argchars` 71f⟩
⟨*global* `pids` 71g⟩
⟨*global* `parity` 71h⟩
⟨*global* `nmach` 71i⟩
⟨*global* `ngraph` 71j⟩
⟨*global* `scale` 71k⟩
⟨*global* `logscale` 71l⟩
⟨*global* `ylabels` 72a⟩
⟨*global* `oldsystem` 72b⟩
⟨*global* `sleeptime` 72c⟩

⟨*global* `procnames` 72d⟩

⟨*function* `killall` 86a⟩

⟨*function* `emalloc` 86b⟩

⟨*function* `erealloc` 86c⟩

⟨*function* `estrdup` 86d⟩

⟨*function mkcol (misc/stats.c)* 87a⟩

⟨*function colinit (misc/stats.c)* 87b⟩

⟨*function* `loadbuf` 87c⟩

⟨*function* `label` 88a⟩

⟨*function* `paritypt` 88b⟩

⟨*function* `datapoint` 88c⟩

⟨*function* `drawdatum` 89a⟩

⟨*function redraw (misc/stats.c)* 89b⟩

⟨*function* `update1` 89c⟩

⟨*function* `readnums` 90a⟩

⟨*function* `filter` 90b⟩

⟨*function* `connect9fs` 91a⟩

⟨*function* `old9p` 91b⟩

⟨*function* `connectexportfs` 92⟩

⟨*function* `readswap` 93a⟩

⟨*function* `shortname` 93b⟩

⟨*function* `ilog10` 93c⟩

⟨*function* `initmach` 93d⟩

⟨*global* `catchalarm` 95a⟩

⟨*function* `alarmed` 95b⟩

⟨*function* `needswap` 95c⟩

⟨*function* `needstat` 95d⟩

⟨*function* `needether` 95e⟩

⟨*function* `needbattery` 96a⟩

⟨*function* `needsignal` 96b⟩

⟨*function* `needtemp` 96c⟩

⟨*function* `readmach` 96d⟩

⟨*function* `memval` 97a⟩

⟨*function* `swapval` 97b⟩

⟨*function* `contextval` 97c⟩

⟨*function* `intrval` 97d⟩

⟨*function* `syscallval` 98a⟩

⟨*function* `faultval` 98b⟩

⟨*function* `tlbmissval` 98c⟩

⟨*function* `tlbpurgeval` 98d⟩

⟨*function* `loadval` 98e⟩

⟨*function* `idleval` 99a⟩

⟨*function* `inintrval` 99b⟩

⟨*function* `etherval` 99c⟩

⟨*function* `etherinval` 99d⟩

⟨*function* `etheroutval` 99e⟩

⟨*function* `ethererrval` 100a⟩

⟨*function* `batteryval` 100b⟩

⟨*function* `signalval` 100c⟩

⟨*function* `tempval` 100d⟩

⟨*function usage (misc/stats.c)* 15a⟩

⟨*function* `addgraph` 101a⟩

⟨*function* `dropgraph` 101b⟩

⟨*function* `addmachine` 102a⟩

⟨*function* `labelstrs` 102b⟩

⟨*function* `labelwidth` 102c⟩

⟨*function* `resize` 103⟩

⟨*function* `eresized` 105a⟩

⟨*function* `mouseproc` 105b⟩

⟨*function* `startproc` 105c⟩

⟨*function main (misc/stats.c)* 72e⟩

Uses `Graph` 68b and `Machine` 68d.

A.2 iostats/

A.2.1 iostats/statfs.h

⟨*constant* `DEBUGFILE` 109a⟩≡ (111d)

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

#define DEBUGFILE "iostats.out"
```

⟨*constant* `DONESTR` 109b⟩≡ (111d)

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

⟨*constant* `DEBUG` 109c⟩≡ (111d)

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

⟨*constant* `MAXPROC` 109d⟩≡ (111d)

```
#define MAXPROC  16
```

⟨*constant* `FHASHSIZE` 109e⟩≡ (111d)

```
#define FHASHSIZE 64
```

⟨*function* `fidhash` 109f⟩≡ (111d)

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

⟨*enum* `_anon_ (iostats/statfs.h)` 109g⟩≡ (111d)

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

⟨*struct* `Frec` 110a⟩≡ (111d)
```
struct Frec
{
    Frec *next;
    char *op;
    ulong nread;
    ulong nwrite;
    ulong bread;
    ulong bwrite;
    ulong opens;
};
```

⟨*struct* `Rpc` 110b⟩≡ (111d)
```
struct Rpc
{
    char *name;
    ulong count;
    vlong time;
    vlong lo;
    vlong hi;
    ulong bin;
    ulong bout;
};
```

⟨*struct* `Stats` 110c⟩≡ (111d)
```
struct Stats
{
    ulong totread;
    ulong totwrite;
    ulong nrpc;
    ulong nproto;
    Rpc rpc[Maxrpc];
};
```
Uses `Maxrpc` 109g.

⟨*struct* `Fsrpc` 110d⟩≡ (111d)
```
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` 109g.

⟨*struct* `Fid` 110e⟩≡ (111d)
```
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 111a⟩≡ (111d)

```
struct File
{
    char *name;
    Qid qid;
    int inval;
    File *parent;
    File *child;
    File *childlist;
};
```

⟨*struct* Proc 111b⟩≡ (111d)

```
struct Proc
{
    uintptr pid;
    int busy;
    Proc *next;
};
```

⟨*enum* _anon_ (iostats/statfs.h)2 111c⟩≡ (111d)

```
enum
{
    Nr_workbufs  = 40,
    Dsegpad  = 8192,
    Fidchunk = 1000,
};
```

⟨iostats/statfs.h 111d⟩≡

⟨*constant* DEBUGFILE 109a⟩
⟨*constant* DONESTR 109b⟩
⟨*constant* DEBUG 109c⟩
⟨*constant* MAXPROC 109d⟩
⟨*constant* FHASHSIZE 109e⟩
⟨*function* fidhash 109f⟩

⟨*enum* _anon_ (iostats/statfs.h) 109g⟩

```
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 110a⟩

⟨*struct* Rpc 110b⟩

⟨*struct* Stats 110c⟩

⟨*struct* Fsrpc 110d⟩

⟨*struct* Fid 110e⟩

⟨*struct* File 111a⟩

⟨*struct* Proc 111b⟩

⟨*enum* _anon_ (iostats/statfs.h)2 111c⟩

```
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 110e, File 111a, Frec 110a, Fsrpc 110d, Proc 111b, Rpc 110b, and Stats 110c.

A.2.2 iostats/globals.c

⟨*global* Workq 112a⟩≡ (113h)

```
Fsrpc *Workq;
```

⟨*global* dbg 112b⟩≡ (113h)

```
int   dbg;
```

⟨*global* root 112c⟩≡ (113h)

```
File *root;
```

⟨*global* fhash 112d⟩≡ (113h)

```
Fid **fhash;
```

⟨*global* fidfree 112e⟩≡ (113h)

```
Fid *fidfree;
```

⟨*global* `qid` 113a⟩≡ (113h)
```
int qid;
```

⟨*global* `Proclist` 113b⟩≡ (113h)
```
Proc *Proclist;
```

⟨*global* `done` 113c⟩≡ (113h)
```
int done;
```

⟨*global* `stats` 113d⟩≡ (113h)
```
Stats *stats;
```

⟨*global* `frhead` 113e⟩≡ (113h)
```
Frec *frhead;
```

⟨*global* `frtail` 113f⟩≡ (113h)
```
Frec *frtail;
```

⟨*global* `myiounit` 113g⟩≡ (113h)
```
int myiounit;
```

⟨`iostats/globals.c` 113h⟩≡
```
#include <u.h>
#include <libc.h>
#include <auth.h>
#include <fcall.h>

#include "statfs.h"

⟨global Workq 112a⟩
⟨global dbg 112b⟩
⟨global root 112c⟩
⟨global fhash 112d⟩
⟨global fidfree 112e⟩
⟨global qid 113a⟩
⟨global Proclist 113b⟩
⟨global done 113c⟩
⟨global stats 113d⟩
⟨global frhead 113e⟩
⟨global frtail 113f⟩
⟨global myiounit 113g⟩
```

A.2.3 `iostats/iostats.c`

⟨*global* `fcalls` 113i⟩≡ (124)
```
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() 127a, Xauth() 126b, Xclunk() 129a, Xcreate() 130, Xflush() 126c, Xremove() 131a, Xstat() 129b, Xversion() 126a, Xwalk() 127b, Xwstat() 131b, and slave() 132.

⟨*global* p 114a⟩≡ (124)

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

⟨*function usage (iostats/iostats.c)* 114b⟩≡ (124)

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

⟨*function main (iostats/iostats.c)* 114c⟩≡ (124)

```
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 117⟩≡ (124)

```
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 109c, Maxfdata 109g, fatal() 122a, p 114a, and stats 113d.

⟨*function* getfid 118a⟩≡ (124)

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

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

    return 0;
}
```

Uses fidhash 109f.

⟨*function* freefid 118b⟩≡ (124)

```
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 112e and fidhash 109f.

⟨*function* `newfid` 119a⟩≡ (124)

```
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` 111c, `fatal()` 122a, `fidfree` 112e, and `fidhash` 109f.

⟨*function* `getsbuf` 119b⟩≡ (124)

```
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 111c, Workq 112a, and fatal() 122a.

⟨*function* strcatalloc 120a⟩≡ (124)

```
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() 122a.

⟨*function* file 120b⟩≡ (124)

```
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 109c, fatal() 122a, makepath() 121b, and qid 113a.

⟨*function* initroot 121a⟩≡ (124)

```
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() 122a, qid 113a, and root 112c.

⟨*function* makepath 121b⟩≡ (124)

```
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` 122a⟩≡ (124)

```
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` 113b.

⟨*function* `rdenv` 122b⟩≡ (124)

```
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` 122c⟩≡ (124)

```
char Defaultpath[] = ".\0/bin";
```

Uses `Defaultpath` 122c.

⟨*function* `runprog` 122d⟩≡ (124)

```
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 122c, fatal() 122a, and rdenv() 122b.

⟨*function* catcher 123a⟩≡ (124)

```
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 109b and done 113c.

⟨*function* fidreport 123b⟩≡ (124)

```
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()` 122a, `frhead` 113e, `frtail` 113f, and `makepath()` 121b.

⟨`iostats/iostats.c` 124⟩≡

```
/*
 * 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` 113i⟩

⟨*global* `p` 114a⟩

⟨*function usage (iostats/iostats.c)* 114b⟩

⟨*function main (iostats/iostats.c)* 114c⟩

⟨*function* `reply` 117⟩

⟨*function* `getfid` 118a⟩

⟨*function* `freefid` 118b⟩

⟨*function* `newfid` 119a⟩

⟨*function* `getsbuf` 119b⟩

⟨*function* `strcatalloc` 120a⟩

⟨*function* `file` 120b⟩

⟨*function* `initroot` 121a⟩

⟨*function* `makepath` 121b⟩

⟨*function* `fatal` 122a⟩

⟨*function* `rdenv` 122b⟩

⟨*global* `Defaultpath` 122c⟩
⟨*function* `runprog` 122d⟩

⟨*function* `catcher` 123a⟩

⟨*function* `fidreport` 123b⟩

A.2.4 iostats/statsrv.c

⟨*global* Ebadfid 125a⟩≡ (137)
```
char Ebadfid[] = "Bad fid";
```
Uses Ebadfid 125a.

⟨*global* Enotdir 125b⟩≡ (137)
```
char Enotdir[] ="Not a directory";
```
Uses Enotdir 125b.

⟨*global* Edupfid 125c⟩≡ (137)
```
char Edupfid[] = "Fid already in use";
```
Uses Edupfid 125c.

⟨*global* Eopen 125d⟩≡ (137)
```
char Eopen[] = "Fid already opened";
```
Uses Eopen 125d.

⟨*global* Exmnt 125e⟩≡ (137)
```
char Exmnt[] = "Cannot .. past mount point";
```
Uses Exmnt 125e.

⟨*global* Enoauth 125f⟩≡ (137)
```
char Enoauth[] = "iostats: Authentication failed";
```
Uses Enoauth 125f.

⟨*global* Ebadver 125g⟩≡ (137)
```
char Ebadver[] = "Unrecognized 9P version";
```
Uses Ebadver 125g.

⟨*function* okfile 125h⟩≡ (137)
```
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 125i⟩≡ (137)
```
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 126a⟩≡ (137)

```
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 125g, Maxfdata 109g, myiounit 113g, reply() 117, stats 113d, and update() 125i.

⟨*function* Xauth 126b⟩≡ (137)

```
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 125f, reply() 117, stats 113d, and update() 125i.

⟨*function* Xflush 126c⟩≡ (137)

```
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 109c, Nr_workbufs 111c, Workq 112a, and reply() 117.

⟨*function* Xattach 127a⟩≡ (137)

```
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 125a, newfid() 119a, reply() 117, root 112c, stats 113d, and update() 125i.

⟨*function* Xwalk 127b⟩≡ (137)

```
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 125a, Edupfid 125c, Exmnt 125e, file() 120b, freefid() 118b, getfid() 118a, newfid() 119a, reply() 117, stats 113d, and update() 125i.

⟨*function* Xclunk 129a⟩≡ (137)

```
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 125a, fidreport() 123b, freefid() 118b, getfid() 118a, reply() 117, stats 113d, and update() 125i.

⟨*function* Xstat 129b⟩≡ (137)

```
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 125a, getfid() 118a, makepath() 121b, okfile() 125h, reply() 117, stats 113d, and update() 125i.

⟨*function* Xcreate 130⟩≡ (137)

```
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 125a, file() 120b, getfid() 118a, makepath() 121b, myiounit 113g, reply() 117, stats 113d, and update() 125i.

⟨*function* Xremove 131a⟩≡ (137)

```
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 109c, Ebadfid 125a, freefid() 118b, getfid() 118a, makepath() 121b, reply() 117, stats 113d, and update() 125i.

⟨*function* Xwstat 131b⟩≡ (137)

```
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 125a, getfid() 118a, makepath() 121b, reply() 117, stats 113d, and update() 125i.

⟨*function* slave 132⟩≡ (137)

```
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 109d, Proclist 113b, blockingslave() 133, and fatal() 122a.

⟨*function* blockingslave 133⟩≡ (137)

```
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 109c, flushaction() 136c, reply() 117, slaveopen() 134a, slaveread() 134b, and slavewrite() 136a.

⟨*function* slaveopen 134a⟩≡ (137)

```
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 109c, Ebadfid 125a, getfid() 118a, makepath() 121b, myiounit 113g, okfile() 125h, reply() 117, stats 113d, and update() 125i.

⟨*function* slaveread 134b⟩≡ (137)

```
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 109c, Ebadfid 125a, Maxfdata 109g, getfid() 118a, reply() 117, stats 113d, and update() 125i.

⟨*function* slavewrite 136a⟩≡ (137)

```
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 109c, Ebadfid 125a, Maxfdata 109g, getfid() 118a, reply() 117, stats 113d, and update() 125i.

⟨*function* reopen 136b⟩≡ (137)

```
void
reopen(Fid *f)
{
    USED(f);
    fatal("reopen");
}
```

Uses fatal() 122a.

⟨*function* flushaction 136c⟩≡ (137)

```
void
flushaction(void *a, char *cause)
{
    USED(a);
    if(strncmp(cause, "kill", 4) == 0)
        noted(NDFLT);
```

```
        noted(NCONT);
    }
```

⟨iostats/statsrv.c 137⟩≡

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

    #include "statfs.h"
```

⟨*global* Ebadfid 125a⟩
⟨*global* Enotdir 125b⟩
⟨*global* Edupfid 125c⟩
⟨*global* Eopen 125d⟩
⟨*global* Exmnt 125e⟩
⟨*global* Enoauth 125f⟩
⟨*global* Ebadver 125g⟩

⟨*function* okfile 125h⟩

⟨*function* update 125i⟩

⟨*function* Xversion 126a⟩

⟨*function* Xauth 126b⟩

⟨*function* Xflush 126c⟩

⟨*function* Xattach 127a⟩

⟨*function* Xwalk 127b⟩

⟨*function* Xclunk 129a⟩

⟨*function* Xstat 129b⟩

⟨*function* Xcreate 130⟩

⟨*function* Xremove 131a⟩

⟨*function* Xwstat 131b⟩

⟨*function* slave 132⟩

⟨*function* blockingslave 133⟩

⟨*function* slaveopen 134a⟩

⟨*function* slaveread 134b⟩

⟨*function* slavewrite 136a⟩

⟨*function* reopen 136b⟩

⟨*function* flushaction 136c⟩

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.

Bibliography

[Gre20] Brendan Gregg. *Systems Performance: Enterprise and the Cloud*. Addison-Wesley, 2nd edition, 2020. cited page(s) 7

[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, 7, 19, 79

[Pad15] Yoann Padioleau. *Principia Softwarica: The ARM Linker 5l*. 2015. cited page(s) 5, 7, 23, 24, 79

[Pad16a] Yoann Padioleau. *Principia Softwarica: The Plan 9 Core Libraries*. 2016. cited page(s) 80

[Pad16b] Yoann Padioleau. *Principia Softwarica: The Plan 9 Debuggers and Tracers*. 2016. cited page(s) 80

[Pad16c] Yoann Padioleau. *Principia Softwarica: The Plan 9 Graphics System draw*. 2016. cited page(s) 80