

Systems From Scratch

From bits to cycles, measured on real hardware.

From bits to cycles, measured on RISC-V. A self-study text on computer systems and performance — data representation, machine code, the operating system layer and CPU microarchitecture — built on the xv6 teaching kernel and measured natively on hardware whose performance counters actually work.

Chris Snow · built 25 September 2026 · `bd413a3`
Prose CC-BY-NC-4.0 · code Apache-2.0

Contents

What it is about

Getting started

What this part is for

00 · Prerequisites and Setup

01 · Setting Up the Board

Part I — C, and what the machine does with it

Part I · C, and what the machine does with it

02 · Reading a Listing

03 · Memory Is One Array

04 · Kernel C Is Not Application C

05 · Reading C as Machine Code

Part II — The machine with nothing on it

Part II · The machine with nothing on it

06 · A Trap, With Nothing Else in the Machine

07 · Interrupts and Privilege Levels

08 · One Page Table, Two Harts

09 · A System Call of Your Own

10 · A Small Integer That Means a Device

11 · fork, Built Rather Than Read

Part III — What a computer does with a program

Part III · What a computer does with a program

12 · What a Computer Does With a Program

13 · Representing Information

14 · Machine-Level Code on RISC-V

15 · Linking and Loading

Part IV — The operating system layer

Part IV · The operating system layer

16 · Traps and System Calls

17 · Virtual Memory

18 · Page Faults as a Feature

19 · Interrupts and Drivers

20 · Locks and Memory Ordering

21 · Scheduling and Context Switches

22 · The File System

23 · The Same Program on Both Targets

Part V — Where the cycles go

Part V · Where the cycles go

24 · Measuring

25 · The Memory Hierarchy

- [26 · Optimising Code](#)
- [27 · The CPU](#)
- [28 · Memory Ordering on Real Hardware](#)
- [29 · The OS Layer's Cost on Real Hardware](#)
- [30 · Whole-Machine Profiling](#)
- [31 · Vectors](#)

Appendices

- [Appendix A · RISC-V Registers and CSRs](#)
- [Appendix B · gdb for Kernels and RISC-V](#)
- [Appendix C · The perf Events This Board Has](#)
- [Appendix D · An xv6 File Map](#)
- [Appendix E · Glossary](#)
- [Appendix F · AArch64 for RISC-V Readers](#)
- [Appendix G · Reading xv6 Alongside This Book](#)
- [Appendix H · Choosing the Machine](#)

Entries are links. Page numbers are omitted because pagination happens after this document is assembled.

From bits to cycles, measured on real hardware.

A self-study text on computer systems and performance, in five parts and thirty-two chapters.

What it is about

This book teaches you to find out where a program's time actually goes, and to know when your answer is wrong. It is easy to learn that a cache miss is expensive, or that a system call costs more than a function call. It is much harder to measure it yourself — on a machine in front of you, for a program you did not write — and to know whether the number you got can be trusted.

What it covers

You work up the layers, from the bits a value is made of to a whole program on real hardware: how data is represented, how C becomes machine code, how a program is linked and loaded, what the operating system does to run it, and what the CPU does with it. Each layer gets the same three questions: how does it work, what does it cost, and how do you measure that cost without fooling yourself?

What you will be able to do

Concretely, and these are the things the problems make you do rather than read about:

- Take a duration on a real machine and say whether it means anything — what the clock cost, how many repetitions, which statistic, and what the measurement could not see.
- Read the disassembly of a function you wrote and account for what the compiler did with it.
- Stop a kernel in the middle of a trap and say what state is where, and why it had to be saved.
- Find the expensive part of a program you did not write, and know when the profiler is lying to you about which line it is.
- Predict a cost from a model before measuring, then say what the gap between the two means.

What you will not get is a table of costs to memorise. Every cost in this book belongs to the one machine that produced it — not to every machine of that model, and not to Arm or to computers in general — and the line under each figure says which machine, which compiler and what day. What carries to your machine is the method, and the list above is a list of methods.

Who it is for

Someone who programs fluently — in a scripting language, or Java, or anything else with a runtime underneath it — and who has never had a reason to write C or read a kernel.

You do not need to know C. Three of [Part I](#)'s four chapters teach it, and none of them is a C tutorial: control flow, functions and operators are assumed from whatever language you already use. They teach the part your language was built to hide — that memory is one array of bytes and everything in it has an index — and then the assumptions that stop holding when there is no runtime underneath you. [ch02](#) comes first and is for everyone — it teaches reading

what the compiler produced, which every later chapter asks you to do. Then [ch03](#) gets you on to that model; [ch04](#) takes away what the runtime was doing for you; [ch05](#) sorts C’s constructs by a single question, *has the machine heard of this?*

You do not need OS internals. [Part IV](#) teaches them, and that is why the book uses a kernel small enough to read rather than one that has to be described.

You do not need any hardware background. [Part II](#) starts at a bare machine and adds one mechanism at a time, so nothing about the hardware is assumed before it is built. [ch27](#) does the same for the processor itself: a reader who has seen a textbook diagram of one still has no idea what a real core does when the program reaches an `if`, and that chapter starts from there.

Where it leads

This book is an on-ramp to *Systems Performance* GREGG (2020), which assumes you already know what a system call costs, what a cache miss is, why a profiler can blame the wrong line, and what a context switch moves. This book teaches you those things by measuring them on a machine you own.

What it deliberately does **not** cover, and what you should read that book for: tracing and BPF, flame graphs, the network and storage stacks, containers and cloud, and observability across many machines. There is no overlap to speak of. This is the layer underneath, and you should read it first, because the tools in that book all report quantities — cache misses, faults, context switches — whose meaning this book establishes.

What this book is

Three things make it the shape it is.

Every number in it was measured, and says where. No figure is typed into the prose. Each one comes from a *stamped result* — a small file in the book’s repository, under `bench/results/`, recording the machine, the kernel, the compiler and its flags, and a hash of the code that produced the number — and a check fails if a quoted figure stops matching the code. Where a machine cannot answer a question at all, the chapter says so and shows the reasoning it used instead, rather than quietly substituting a number from somewhere else.

Every problem is a test, and there is no answer key. Each chapter ends with problems that are stubs under `tests/`, with a test that passes only when you have solved it. Nothing in the repository contains the answers — which also means there is no answer key to be wrong.

Every chapter ends by saying what it could not show you. A section called *What this cannot tell you* is mandatory, and it is where the tools ran out — what could not be run, measured or shown. It is usually the most useful part of the chapter.

WHERE THIS BOOK IS

All thirty-two chapters are written, and all eight of the eight appendices. [Appendix C](#), which lists what the reference machine’s processor can count, was generated on that machine, the only place the list could honestly have come from. Four figures are marked *pending*: [ch24](#) has just grown a second instrument — a confidence interval and a rule for deciding that one thing is faster than another — and the board has not yet been asked for those four. They render as a warning containing no numbers, which is what this book does instead of estimating.

[Download the whole book as a PDF](#) — every chapter and appendix in one file, built from the same source as this site, so the two cannot disagree about what a chapter says.

Built on 2026-09-25 from commit [bd413a3](#) — compare it with the [latest on GitHub](#) to see whether a newer one exists.

How the book is arranged

Part I — C, and what the machine does with it	Four chapters. The first is for everyone; which of the rest you need depends on where you are starting. Not a C tutorial
Part II — The machine with nothing on it	A trap, an interrupt, a page table, a system call, file descriptors and <code>fork()</code> , each built from nothing on bare hardware before any kernel is read
Part III — What a computer does with a program	One program from source text to result: the toolchain, representation, machine code, linking
Part IV — The operating system layer	A kernel small enough to read, taken apart: traps, virtual memory, faults, drivers, locks, scheduling, files
Part V — Where the cycles go	Part IV ’s chapters asked again as questions about time, on hardware that can answer them

The sequence is an argument, not a filing order. [Part II](#) takes the machine’s basic mechanisms one at a time because a kernel presents them entangled — the first trap you meet in a real one arrives with a process table, a scheduler and a lock already attached. [Part III](#) is next because a kernel is a program, and you cannot usefully read one until you know what a program is and who finishes what the compiler left undone; [ch15](#) ends with the kernel starting a program, which is where [Part IV](#) begins. [Part IV](#) puts the entanglement back, and it turns out to be most of what an operating system is.

[Getting started](#) sits before all of it: [ch00](#) sets up everything that runs on your own computer and gives you a script that says what it can currently run, and [ch01](#) sets up the second machine, whenever it arrives.

[Part IV](#) reads a real kernel, and that kernel has a book of its own: *xv6: a simple, Unix-like teaching operating system* COX ET AL. (N.D.), written by its authors and free from MIT. The two do different jobs — it explains what the code does, this book asks what that costs — and they go well together. [Appendix G](#) lines them up by topic rather than by chapter number, because its numbering moves between revisions. The same appendix traces one system call through every layer it touches.

Three targets on two machines, on purpose

Almost every book on this subject picks one target and lives with its limitations. This one uses three, because *what does a program do* and *what does it cost* need different instruments.

One book, three targets, two machines

Two run under emulation and say what a program does; one is real and says what it costs. The mechanisms transfer; the numbers do not.

No operating system, under QEMU target: bare — what the hardware does	xv6 under QEMU target: xv6 — what a kernel does	Linux on real hardware target: host — what the program costs
Answers, exactly <ul style="list-style-type: none">• what the machine does at reset• what a trap leaves in mtvec and mcause• what changes when you leave M-mode• how the hardware reads a page table• what a second hart is	Answers, exactly <ul style="list-style-type: none">• which instructions run, in order• what a system call does to a process• how a page table is walked• what exec does to an address space• why a scheduler picked this thread	Answers, exactly <ul style="list-style-type: none">• cycles and instructions retired• cache and TLB miss rates• branch mispredictions• where the time went, by sampling• what four cores do to each other
Cannot tell you <ul style="list-style-type: none">✗ how many cycles anything took✗ whether a load hit in cache✗ whether a branch was predicted✗ what memory latency is	Cannot tell you <ul style="list-style-type: none">✗ how many cycles anything took✗ whether a load hit in cache✗ whether a branch was predicted✗ what memory latency is	Cannot tell you <ul style="list-style-type: none">✗ the kernel source under a breakpoint✗ a page table you can print✗ a scheduler you can instrument freely✗ an experiment you can repeat exactly

The first two columns are one machine: your laptop, running QEMU, which has no cache model, no branch predictor and no performance counters — so it will answer a timing question, and the answer will be fiction.

The division of labour. Every chapter declares which target it uses, and every figure records which one produced it.

The three are **bare** (a RISC-V processor — RISC-V is an open processor design — emulated by QEMU on your own computer, with no kernel on it at all), **xv6** (the MIT teaching kernel, running on the same emulator), and **host** (a real Linux machine, a Raspberry Pi 5 by default); **ch00** sets them up and says what each can and cannot answer. That is three targets on **two machines**: the first two are both QEMU on the computer you are reading this on, and need one cross-compiler between them — a compiler that runs on your machine and produces code for a different kind of processor. Only the third has to be real. **Part III** works on both sides of the split, and **ch12** is where it crosses.

The split is not a compromise. QEMU will happily answer a question about nanoseconds and the answer will be meaningless, because it models no cache, no branch predictor and no pipeline. Watching a program in a debugger tells you what it *does*. Only real hardware tells you what it *costs*. **ch23** puts the same program through both and makes the gap concrete.

The two emulated targets are RISC-V and the board is ARM, because the kernel small enough to read is a RISC-V kernel, and the hardware whose performance counters actually work — the processor’s own tally of what it did, which is what every cost in this book is read from — is an ARM one. **Appendix H** has the evidence.

That costs you one thing: you read disassembly in two instruction sets rather than one. Five chapters read it as AArch64, ARM’s 64-bit instruction set — **ch26**, **ch27**, **ch29**, **ch30** and **ch31** — and five read it as RISC-V, in Parts I and III. **ch02** shows one small function both ways before either matters, and **Appendix F** translates between them.

It also buys something. Two chapters are about how one processor core’s writes to memory become visible to another — **ch20** on RISC-V and **ch28** on ARM — and a reader shown only one of them would reasonably conclude that is how it works everywhere. Shown two, you learn which parts are that design’s and which are the idea.

One argument, not two tutorials

Part V is not a second book. It is **Part IV**’s chapters asked again as questions about time. Every chapter in it names the earlier chapter whose cost it measures, in its own header:

When Part V asks	You already learned the mechanism in
ch25 — where is the data, and what does each step outward cost?	ch13 layout and alignment, ch17 address translation
ch26 — what did that cost?	ch14 what the compiler emitted
ch27 — what is the core doing between fetch and finish?	ch14 the instructions themselves
ch28 — what do four cores cost each other?	ch20 locks, fences and ordering
ch29 — what does Linux charge for this?	ch16 traps, ch18 faults, ch21 switches

So you never arrive at a **Part V** chapter cold. You arrive knowing the mechanism completely and needing only the price — a better position than either half could put you in alone, and the reason the book is arranged this way rather than as theory followed by benchmarks.

Three **Part V** chapters have no counterpart, deliberately: [ch24](#) teaches measurement itself, [ch30](#) is about the whole machine rather than any one mechanism, and [ch31](#) concerns hardware **Part IV** never had reason to describe.

How the numbers work

The chapters whose reading depends on something specific about the reference machine say so in their own headers rather than letting you discover it two hundred pages in.

What you will need

A laptop for [ch00](#) and for Parts I to IV — everything there runs under emulation, free. A small Linux machine with working performance counters for [ch01](#) and **Part V**; a Raspberry Pi 5 is the reference, and one you already own may well do — [Appendix H](#) says exactly what it has to be able to do and how to check. Twenty-two chapters sit between needing the first and needing the second.

And the book’s repository. Every program a chapter shows, every problem it sets and every result behind a figure is a file in it, and the book is meant to be read here and worked through in a checkout — [ch00](#) starts by cloning it.

The book does not tell you which kernel to run. Whether a machine’s performance counters work is a property of its whole configuration — silicon, device tree, kernel, firmware — rather than of the board, and the reference board’s own counters went missing for a kernel release. So every figure records the image and kernel that produced it, and [ch00](#) hands you a script that asks your machine instead of a version number to match.

Getting started

What this part is for · 00 · Prerequisites and Setup · 01 · Setting Up the Board

PART HEADER

Chapters	ch00–ch01
Target	all three, which is the point
Assumes	nothing

What this part is for

Two machines to get working, and a script that says what each one can currently run.

The book uses three targets on two machines, and the two machines are not equally urgent. The computer you are reading this on runs `bare` and `xv6` under QEMU, needs one cross compiler, and covers everything up to [Part V](#). The board is a second machine — the only place a timing may be measured — and nothing needs it until [ch24](#).

So the two chapters are in that order, and they are separate on purpose: you can finish [ch00](#) this afternoon on a laptop, and do [ch01](#) whenever the hardware turns up. The board chapter is the next one in the book, but twenty-two chapters sit between finishing the first and *needing* the second.

What it leaves out

The decision about what to buy. That is [Appendix H](#), because it is read once, before you own anything, and then only to settle an argument with a spec sheet. This part assumes the machines exist.

Any reason for the arrangement. The [preface](#) argues for three targets on two machines and for the instruction sets not matching. These chapters take that as settled and get it working.

Where to start

[ch00](#), on whatever you are reading this on. It ends with one program compiled for two targets and run on both, agreeing about every answer, which is how you know the setup is real rather than merely installed.

Then [ch01](#) when the board arrives — or later. Nothing before [Part V](#) depends on it, and [ch00](#)'s script will tell you at any point which targets your machines can currently run.

Which machine, and what it cannot tell you

Both, which is what makes this a part rather than a chapter. The emulated targets answer questions about what a program *does*, and **nothing here or in Parts I to IV is ever timed**: QEMU models no cache, no branch predictor and no memory latency, so a duration measured inside it describes the laptop, and how the emulator translates instructions, rather than the machine being emulated. The board answers questions about cost, and is the only place in the book a timing may be taken.

Neither can tell you yet whether any of it is *worth* measuring. Setting a machine up proves it runs; it does not prove a number coming out of it means anything, and [ch24](#) is the chapter that takes that apart.

Where this leaves you

Two machines that work, and a script that says so. You will not have learned anything about computers yet. You will have the apparatus the rest of the book measures with, and a way to check at any point that it is still telling the truth.

00 · Prerequisites and Setup

CHAPTER HEADER

Target	xv6 — the teaching kernel under QEMU, whose toolchain <code>bare</code> shares
Prerequisites	none
What it measures	That the emulated targets work, and exactly what they are: <code>bench/results/setup-xv6.json</code>

The question

What do I need on my desk, and how do I know it works?

Knowing it works matters as much as having it. Every later chapter rests on a claim about a machine — this compiler, this kernel, these counters — and a setup that is *almost* right fails three chapters later as something that looks like a bug in the material. So this chapter ends with a script that interrogates the machine you are sitting at and says which of the book’s three targets it can run, and with measurements that record what those targets actually are.

The material

Three targets, and what the repository does about it

The [preface](#) makes the case for the arrangement; here is what each target is in practice.

bare is `qemu-system-riscv64` with no kernel under it at all — the machine on its own. **Part II** builds on it directly, and it shares `xv6`’s cross-compiler and setup, so one check covers both.

xv6 is the MIT teaching kernel under `qemu-system-riscv64`: a complete operating system in about nine thousand lines, which you can stop anywhere — halfway through entering the kernel, if you like — and inspect. Parts I and IV live there, and so does everything the book says about *what a program does*; **Part III** works on both sides of the split, and **ch12** is where it crosses.

host is a small Linux machine on the desk, reached over SSH — a Raspberry Pi 5 in this book. Everything about *what a program costs* is measured there, natively. **Part V** lives there.

QEMU is a functional emulator, and every later chapter depends on what that means. The [preface](#) says so too: QEMU computes what the instructions compute and models nothing else — none of the machinery that decides how long an instruction actually takes, so no cache, no branch predictor, no store buffer, no pipeline, no memory latency. Ask it how long a loop took and it will answer, and the answer describes the laptop QEMU was running on and how it happened to translate the instructions.

Almost every convenient way to observe a program changes what you are observing. A debugger stops it. A profiler samples it. A print statement in a loop makes the loop something else. Knowing which of your tools is lying to you about which question is the discipline underneath all of this, and it holds well beyond QEMU, which is why the book starts here.

So the repository enforces the split rather than trusting anyone to remember it. Every result file records where it was measured, and `scripts/verify-numbers.py` rejects two things outright: a `host` figure that was not produced natively on the hardware it claims, and an `xv6` result that contains a duration at all.

```
def provenance_problems(name: str, payload: dict[str, Any]) -> list[str]:
    """Whether this result was produced somewhere it is allowed to have been produced.

    This is the rule the book's credibility actually rests on, so it lives beside the definition
    of a result rather than inside a script. Three claims it refuses:

    * a ``host`` measurement that was not taken natively on the board. An emulated duration is
      indistinguishable from a real one once it is a number in a table, so the check has to happen
      while the provenance is still attached;
    * an ``xv6`` result that contains a duration at all. QEMU models no cache, no branch
      predictor and no pipeline, so a time measured inside it is not a slow measurement — it is
      not a measurement;
    * a ``listing`` that is anything other than disassembly. Listings are exempt from the first
      rule, so they have to be held to their own.

    A result with no ``kind`` is a measurement. That was the only thing a result could be when the
    field did not exist, and reading it that way means an old file is judged by the stricter rules
    rather than slipping past both.
    """
```

What you need

Two machines, and only one of them has to be bought. Whatever you are reading this on runs both emulated targets; `host` needs a small Linux board. The reference board is a **Raspberry Pi 5**, but what matters is a capability rather than a part number: `perf`, Linux's tool for reading the processor's performance counters, has to be able both to count events and to sample where the program was when they happened. [Appendix H](#) states that properly, and says how to check a machine you already own and what changes if yours differs from the reference. Read it before you spend anything — and then carry on here, because nothing in this chapter needs the board. [ch01](#) is where it gets set up, and twenty-two chapters go by before anything depends on it.

Setting up the xv6 target

On the Mac, via Homebrew:

```
brew tap riscv-software-src/riscv
brew install riscv-tools qemu
```

On a Debian or Ubuntu machine, including the board itself:

```
sudo apt install -y gcc-riscv64-linux-gnu qemu-system-misc qemu-user-static gdb-multiarch
```

`qemu-user-static` is worth installing even though no chapter requires it. It runs a program built for a different processor directly on your own machine — an ARM binary on an x86-64 laptop, say — which means every `host`-target example in [Part V](#) can be compiled and **checked for correctness** away from the board. It cannot tell you anything about time, and the repository does not let it try — but it is the difference between being able to work on [Part V](#) from a train and not.

The repository

```
git clone --recursive https://github.com/snowch/computer-systems.git
cd computer-systems
python3 -m pip install -r requirements.txt -r requirements-dev.txt
```

`--recursive` matters: `xv6` is a submodule, not a copy. If you have already cloned without it, `make submodule` fixes it.

`xv6` is MIT-licensed and copying it in would be perfectly legal, but a copy would hide what this book changes about the kernel. So the submodule stays pristine — a test asserts that it is byte-identical to upstream — and the book’s own material is kept separately and combined into a staging tree at build time:

```
xv6/xv6-riscv/  upstream, never modified
xv6/apps/       the book's xv6 programs
xv6/patches/    the book's kernel instrumentation, as diffs
xv6/stage/      generated: all three, combined
```

So `ls xv6/patches/` is a complete answer to “what has this book done to the kernel?”, and that answer stays true. `xv6/README.md` has the mechanics.

Booting it

```
make xv6-qemu
```

You should get a boot log, a shell prompt, and `ls` should list a couple of dozen programs.

Getting out again is `Ctrl-A` then `X`. It is a sequence rather than a chord: hold control and press A, let both go, then press X. QEMU exits immediately.

The key belongs to QEMU rather than to `xv6`. `xv6` has no way to halt the machine — no `shutdown`, no `halt`, nothing — so there is nothing to type at the shell prompt that would end the session. You are stopping the emulator out from under a kernel that has no opinion about it, which is the first of many small reminders that this is a teaching kernel and not a product.

`Ctrl-A` then `C` switches the same terminal to QEMU’s own monitor, where `quit` also exits and `info registers` works without a debugger attached. `Ctrl-A` then `C` again switches back. Learn it now rather than later: when [Appendix B](#) has QEMU halted at reset waiting for a debugger, the terminal looks frozen, and the monitor is how you confirm it is not.

The same thing non-interactively, which is how the tests do it:

```
python3 scripts/xv6-run.py -c ls -c sysprobe
```

Running the book’s programs

Every listing in this book is a real file that really compiles, and `./run` builds and runs any of them on whichever machine it belongs to.

```
./run --list
```

```
./run firstc      # a host program: compiled and run here
./run trap        # bare metal: cross-compiled and booted with no operating system
./run sysprobe --target xv6
```

Use it constantly. Change a line in the source, run it again, see what moved — that loop is what this book is asking you to do, and it is the difference between reading that a handler must advance `mepc` and watching what happens when it does not.

`./run` never touches anything committed. The figures in the chapters come from `bench/run_*.py`, which stamp a result and refuse one that has stopped demonstrating its chapter's claim; `./run` just builds and runs, so experimenting is free and cannot corrupt a number.

Checking the whole thing

```
python3 scripts/verify-setup.py
```

It reports each target separately, because most machines can run two of the three. On a laptop it confirms the cross compiler, QEMU, the submodule and a usable debugger, then says that the `host` target is read-only here — no figure can be measured on this machine — and whether a cross-built correctness path is available. On the board itself it reads the device tree and `/proc/cpuinfo`, prints whatever that kernel says identifies the core — an implementer and part number on ARM, an ISA string and three implementation IDs on RISC-V — and checks that `perf` reaches hardware.

It looks nothing up. Every fact it prints is read from the machine in front of it. A specification describes a product line; `/proc/cpuinfo` describes the silicon that is about to produce your numbers, and when the two disagree — which happens — the book cites the one it measured.

The same program in both worlds

Two commands remain. The first boots the kernel, runs a probe and stamps this chapter's first real result; the second builds the same probe for the other target and checks that the two agree about every answer.

`sysfs/include/sysfs/probe.h` asks the machine a handful of questions it can answer without a library: how big each basic type is, which addresses it is allowed to start at, what the compiler does to a struct, and which end of a multi-byte value holds its least significant byte. It is compiled twice from the same bytes, for two targets that disagree about what a C library is, and the header says so at the top — the first design decision in this book that exists entirely because of where the code has to run:

probe.h

```
/* Facts the machine will tell you about itself, asked in a way both targets can answer.
 *
 * Copyright 2026 Chris Snow. Apache-2.0 – see LICENSE-CODE.
 *
 * This header is compiled twice from the same bytes: once against glibc on Linux (the `host`
 * target) and once against xv6's freestanding user library (the `xv6` target). That constraint
 * is why it looks the way it does.
 *
 * - No #include of anything. xv6 user programs build with -ffreestanding -nostdlib and have
 *   no <stdint.h>, no <stddef.h> and no size_t. A header that needs libc cannot be shared.
 * - __alignof__ rather than _Alignof. xv6 builds with -std=gnu99, where _Alignof does not yet
 *   exist; __alignof__ is a GNU extension both compilers have understood for twenty years.
 * - Printing is the caller's problem. Both worlds have a printf(const char *, ...), but they
 *   are different functions with different capabilities, so this header hands rows to whatever
 *   the caller passes and formats them with %s and %d only – the intersection of what xv6's
 *   printf and glibc's printf both support.
 *
 * The output is one fact per line, `kind name value...`, because a format a shell can read with
 * `cut` is also a format a test can assert on without a parser.
 */
```

The rest of the header follows from that decision. Asking the machine which end of a value it puts the least significant byte at, rather than assuming, costs three lines of C:

probe.h

```
/* Byte order, asked of the machine rather than assumed.
 *
 * RISC-V is little-endian in every implementation anyone ships, but "everyone knows that" is the
 * kind of claim this book is not allowed to make. Reading the first byte of a known integer costs
 * three lines of C and turns it into something measured. */
static inline int sysfs_is_little_endian(void) {
    unsigned int word = 1u;
    const unsigned char *bytes = (const unsigned char *)&word;
    return bytes[0] == 1u;
}
```

Run it in both worlds:

```
make bench-xv6                                # boots xv6, runs the probe, stamps the result
python3 -m pytest tests/test_xv6.py -q         # asserts the two targets agree
```

What we measured

The xv6 target, described by a boot rather than by a claim:

What	This boot
Kernel	xv6-riscv @ 9e3161a
Emulator	QEMU emulator version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.18)
Machine model	qemu virt (-machine virt -bios none)
Harts started	3
User programs in the image	28
Kernel image size (bytes)	298048
Byte order	little

Conditions: target `xv6` · qemu virt (-machine virt -bios none) · QEMU emulator version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.18) · `xv6-riscv @ 9e3161a` · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · `xv6's own CFLAGS (-O -march=rv64gc -std=gnu99 -ffreestanding)` · 2026-09-17 · Source: `bench/results/setup-xv6.json`, code hash `b3e54cf82a22855a`.

The C implementation it presents, as reported from inside the kernel:

C type	Bytes	Alignment
<code>char</code>	1	1
<code>short</code>	2	2
<code>int</code>	4	4
<code>long</code>	8	8
<code>long long</code>	8	8
<code>pointer</code>	8	8
<code>float</code>	4	4
<code>double</code>	8	8

Conditions: target `xv6` · qemu virt (-machine virt -bios none) · QEMU emulator version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.18) · `xv6-riscv @ 9e3161a` · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · `xv6's own CFLAGS (-O -march=rv64gc -std=gnu99 -ffreestanding)` · 2026-09-17 · Source: `bench/results/setup-xv6.json`, code hash `b3e54cf82a22855a`.

This is the **LP64** data model: `long` and pointers are 64-bit, `int` stays 32-bit, and every basic type's alignment — the address boundary it has to start on — equals its size. That is a choice made by the ABI, the agreement between compiler and operating system about how data is laid out and passed around; RISC-V spells its variant LP64D, for the double-precision float convention RISC-V INTERNATIONAL (N.D.), and AArch64 arrives at the same layout by its own route. If you have only ever worked on 64-bit Linux this will look like the way things are. It is a choice, made twice, independently, and [ch13](#) takes it apart.

The third table matters most. Two structs, the same three members, different declaration order:

Struct	sizeof	Alignment	Bytes that hold nothing
<code>declaration_order</code>	12	4	6
<code>size_order</code>	8	4	2

Conditions: target `xv6` · qemu virt (-machine virt -bios none) · QEMU emulator version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.18) · `xv6-riscv @ 9e3161a` · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · `xv6's own CFLAGS (-O -march=rv64gc -std=gnu99 -ffreestanding)` · 2026-09-17 · Source: `bench/results/setup-xv6.json`, code hash `b3e54cf82a22855a`.

The compiler did not reorder them — C forbids it — so writing them in the order that happened to occur to you cost bytes that hold nothing at all. On one struct that is an oddity. Across an array of a few million of them it is the difference between fitting in cache and not, which is [ch25](#)'s subject and the first place this chapter's dry table turns into a number of nanoseconds.

What this cannot tell you

Everything above is *structural*: sizes, offsets, byte order, which programs are in an image, how many harts (independent hardware threads) announced themselves. Those are questions QEMU answers perfectly, because they are questions about what the instructions compute.

None of them is a question about time, and that is deliberate rather than an accident of what this chapter chose to measure. The xv6 target will never produce a timing in this book, because a timing produced there would be meaningless, and a meaningless number in a table is worse than a missing one — a missing number announces itself.

[ch01](#)'s table is the other half of the same discipline, and if it is showing a warning box rather than numbers, that is because the measurement has not been taken yet: nothing is estimated, interpolated, or carried over from a different machine. `make bench-board` refuses to run anywhere but the board, and `scripts/verify-numbers.py` rejects the result if it somehow arrives from anywhere else.

Three more limits worth naming now, since all three come up repeatedly:

A correct answer is not a fast answer. CI compiles every `host`-target example for AArch64 and runs it under user-mode QEMU. That proves the instructions are right and the answers are right, and it proves nothing whatsoever about cost. When a chapter says a result was checked in CI, it means checked, not timed.

This machine is one data point. Four out-of-order cores with a three-level cache is an ordinary shape, not a universal one. Ratios and mechanisms generalise; absolute numbers do not, and the chapters whose reading depends on this particular core say so in their own headers.

And it is a machine that changes speed. A Pi 5 throttles under sustained load, so a long run can be measuring a different clock at the end than at the start. That is not a flaw in the board — it is what most real hardware does, including the laptop you are reading this on, and a book that measured on a machine which never throttled would be teaching you to ignore something that matters. [ch24](#) deals with it properly.

Problems

Three, and each one has a test that passes only when you have solved it. There is no answer key in the back of the book — which means there is no answer key to be wrong.

0.1 — Which of these would you publish? `tests/prerequisites_and_setup/problem_1_trust.py` asks you to write one function: given a stamped result, decide whether a duration measured under those conditions is one this book may print. Five cases are waiting for it, and exactly one of them is publishable. Getting this right is the same skill as reading someone else's benchmark and noticing what they measured it on.

```
python3 -m pytest tests/prerequisites_and_setup/test_problem_1_trust.py
```

0.2 — Predict the padding. `tests/prerequisites_and_setup/problem_2_abi.py` shows you a struct with five members and asks for its size, its alignment, and the offset of each member — *before* you compile it. The test then compiles that struct for whichever architecture this machine can execute, runs it, and tells you where you were wrong. The tables above give you the sizes and alignments of the scalar types; the rest follows from one rule, and it is the same rule on both architectures.

```
python3 -m pytest tests/prerequisites_and_setup/test_problem_2_abi.py
```

0.3 — Your first xv6 program. `tests/prerequisites_and_setup/ch00ping.c` is a program that prints nothing. Make `ch00ping 41` print `pong 42`. The arithmetic is not the exercise: the exercise is the path from a file in a test directory, through the cross compiler, into xv6’s user library, onto a file system image, into QEMU, and out of a shell. If any link in that chain is missing you want to find out now, not in [ch16](#).

```
python3 -m pytest tests/prerequisites_and_setup -q          # all three, including the ones you have not
solved
```

These tests are marked `problem`, and CI deliberately does not run them. What CI does run is the scaffolding beside each one: that the puzzle compiles, that the kernel boots with your file staged into it, that the problem is answerable. The book is responsible for handing you a problem that works. Making it pass is yours.

Where to go next

The RISC-V specifications are the primary sources this book cites for anything architectural: the unprivileged ISA RISC-V INTERNATIONAL (N.D.) for instructions, the privileged architecture RISC-V INTERNATIONAL (N.D.) for CSRs, traps and paging, the ELF psABI RISC-V INTERNATIONAL (N.D.) for the calling convention and the data model measured above, and the SBI specification RISC-V INTERNATIONAL (N.D.) for how Linux reaches the counters. They are readable, and reading a specification directly is a skill this book would like you to acquire early — the habit of checking rather than remembering is most of what separates a confident answer from a correct one.

For the reference machine, Raspberry Pi’s own documentation RASPBERRY PI LTD (N.D.) gives the SoC and its cache hierarchy, and Arm’s Cortex-A76 technical reference manual ARM LTD (N.D.) gives the pipeline and the PMU events [ch27](#) reads. The RISC-V hardware [Appendix H](#) argues against is documented at STARFIVE TECHNOLOGY (N.D.) and SiFIVE, INC. (N.D.) if you want to follow that thread. Either way the caveat stands: where a document and a measurement disagree, the book prints the measurement and says so.

The study behind that decision is DISSECTING RISC-V PERFORMANCE: PRACTICAL PMU PROFILING AND HARDWARE-AGNOSTIC ROOFLINE ANALYSIS ON EMERGING PLATFORMS (2025), and it is worth reading even if you never touch RISC-V — it shows what it takes to establish what a machine can actually do, rather than what its documentation says it has.

The xv6 source COX *ET AL.* (N.D.) is worth browsing before [ch12](#), without trying to understand it. Its authors also wrote a commentary on it, which is excellent and whose structure this book deliberately does not follow; if you want a second account of the same kernel after [Part IV](#), that is the one to read.

Next is [ch01](#) if the board is on your desk, and [ch02](#) if it is not — nothing between here and [Part V](#) needs it. Further on, [ch12](#) takes a single program and follows it from source text to a result on both targets, and asks — for the first of many times — which parts of that journey cost anything.

01 · Setting Up the Board

CHAPTER HEADER

Target	host — Linux on real hardware, natively, over SSH
Prerequisites	ch00
Assumes	A Linux board whose <code>perf</code> can both count and sample
What it measures	What the board says about itself once it is working: <code>bench/results/setup-host.json</code>

The question

Is this machine telling me the truth about itself?

[ch00](#) got the emulated targets working, and they are enough for everything up to [Part V](#). This chapter sets up the other machine: the one every number in the book is measured on, and therefore the one whose word has to be good.

A counter can be present, be readable, return a number, and still be a software estimate rather than a count of anything. Most of this chapter is spent establishing that the machine in front of you is not doing that, because a figure produced by a fake counter is indistinguishable from a real one once it is in a table.

[Appendix H](#) covers the decision that comes first — what the board has to be able to do, and how to tell whether the one you own qualifies. This chapter assumes it is on your desk.

The material

A Pi is a well-trodden path and the Raspberry Pi documentation is the authority on it. What follows is the shape of the task and the parts this book depends on.

1. Write a 64-bit image. Raspberry Pi OS (64-bit), written with Raspberry Pi Imager, which will also set the hostname, your SSH key and your Wi-Fi while it writes. Use its advanced options — it saves the whole “find it on the network and change the default password” dance.

It has to be a **64-bit** image. A 32-bit userspace on ARMv7 does not get you the ARMv8 PMU — the performance monitoring unit, the block of hardware the counters live in — and you would spend an afternoon finding that out.

It also has to be an image whose **device tree describes the PMU**, and not every image does. The device tree is the file the kernel is handed at boot saying what hardware is present; the counters are in every Pi 5’s silicon, but whether Linux is told about them depends on the compiled device tree, the `.dtb`, your image ships. In the sources `RASPBERRYPI/LINUX & TORVALDS/LINUX` (N.D.), the Raspberry Pi kernel carries an `arm-pmu` node for the Cortex-A76, with one overflow interrupt per core, and every Pi 5 variant inherits it. Mainline Linux’s own BCM2712 tree carries no such node at all.

So prefer an image built on the Raspberry Pi kernel, which is what Raspberry Pi OS and the Raspberry Pi builds of other distributions use. A general-purpose distribution running a mainline kernel with mainline device trees on the same board may have no hardware PMU exposed to it whatever — not because the chip lacks one, but because nothing told the kernel it was there. One command settles it either way, and it is in *Proving the counters are real* below.

2. Boot it, wired if you can. A radio makes the machine do work you did not ask for: its driver runs on the same cores your benchmark is running on, waking up for every packet and doing its deferred work there, and a lossy link adds `sshd` wakeups on top. Link speed has nothing to do with it — nothing in **Part V** touches the network, so bandwidth, latency and the grade of cable are all irrelevant to every number in this book.

That is a mechanism, and [ch24](#) measures what it does: the same fixed workload run many times over, with the radio off and then with it on, because “the same benchmark, one thing changed that should not matter” is exactly that chapter’s subject. What it found is worth carrying here as advice rather than as a number — the radio never made the fastest run slower, and it made the typical one less predictable. Wire the board, and run it both ways yourself when you get there.

Only the machine being measured needs the cable. Your laptop can stay on Wi-Fi: its radio interrupts its own cores, not the ones running the benchmark. So for most people this costs a cable to the nearest router or switch port and nothing else — the machine gets an address and a route without being asked, and both ends are on the same network, so everything below works.

If no router port is within reach, a cable straight into a laptop’s Ethernet port — a dock’s, usually, or a cheap USB-C adapter — works too, and is arguably quieter, since nothing else on a two-host link is broadcasting at it. But that link has no DHCP server and no route out, so the machine comes up with a link-local address and no internet, and the first `apt install` fails in a way that looks like a broken image. Turn on your laptop’s internet sharing (macOS: Settings → General → Sharing → Internet Sharing, from Wi-Fi to the Ethernet adapter; Linux: set the connection to *Shared to other computers*) and both problems go away at once — it hands out the lease and routes the traffic. `.local` names resolve over a direct cable either way, so step 3 works unchanged.

3. Give it a name. In `~/.ssh/config` on your laptop:

```
Host bench
  HostName raspberrypi.local
  User pi
  ServerAliveInterval 30
```

Now `ssh bench` works, `make bench-board` over SSH works, and VS Code’s Remote-SSH extension can open it as a workspace — *Remote-SSH: Connect to Host...*, pick `bench`, and the editor runs its file operations and its terminal there while the interface stays on your laptop. That is the arrangement the rest of the book assumes: you edit on the laptop, and everything that touches a counter happens on the machine being measured.

The toolchain on the machine

```
sudo apt update
sudo apt install -y build-essential gdb git python3 python3-pip
```

`perf` is the awkward one. It ships as part of the kernel’s own tooling, so the package that provides it is tied to the running kernel. On Raspberry Pi OS:

```
sudo apt install -y linux-perf
perf --version
```

On Ubuntu, `linux-tools-$(uname -r)` or `linux-tools-raspi`. If the version `perf` reports does not match `uname -r` it will still run, and will quietly fail to open some events — which is the worst of the available outcomes, because it looks like the events do not exist rather than like a broken tool. If nothing packaged matches, build it from the kernel source tree with `make -C tools/perf` against the source for your running kernel.

Proving the counters are real

Counting is the one capability **Part V** cannot work around, and `perf` reports a failure to reach hardware in a way that is easy to skim past. So be suspicious of it.

Ask:

```
perf stat -e cycles,instructions -- true
```

A working setup prints two counts. A broken one prints `<not supported>`, which means the event never reached hardware — and if you are not reading carefully, a line saying `<not supported>` in a column of numbers looks like a number. Worse, some configurations report a count of zero rather than an error, and a zero is a number that will happily propagate into a table.

`bench/run_setup.py` therefore treats “counted” and “counted something greater than zero” as different questions:

`run_setup.py`

```
def perf_capability() -> dict[str, Any]:
    """Whether ``perf stat`` on this board reads real hardware counters, and which ones.

    This is the single most important thing the board chapter establishes, because every
    chapter in Part V assumes it. On RISC-V the counters reach Linux through the SBI PMU
    extension, so the answer depends on the firmware as much as on the silicon — which means it
    has to be asked of the machine rather than looked up.

    ``<not supported>`` and ``<not counted>`` are perf's way of saying the event did not reach
    hardware. Treating a zero as a reading is how a chapter ends up asserting that a loop
    retired no instructions.
    """
```

If nothing is counted, the usual cause on ARM is the missing device-tree node described above. Ask the kernel directly:

```
dmesg | grep -i perfevents
ls /sys/bus/event_source/devices/
```

A machine whose PMU registered says so at boot, naming the driver it bound:

```
hw perfevents: enabled with armv8_cortex_a76 PMU driver, 7 counters available
```

and `/sys/bus/event_source/devices/` contains a matching entry. No such line, or no such entry, and no amount of care in `perf`’s arguments will help: there is nothing underneath it. On a RISC-V machine the failure is usually further down — the counters are machine-mode CSRs reached through the firmware’s SBI PMU extension RISC-V INTERNATIONAL (N.D.), so check for `CONFIG_RISCV_PMU_SBI` and a firmware that provides it.

Until `perf stat` prints real counts, **Part V** cannot start, and nothing in the later chapters substitutes for it.

Counting is not sampling

There is a second capability, sampling, and it is the reason this book's `host` target is an ARM machine.

`perf stat counts`: it totals events over a whole run. `perf record samples`: it interrupts the program thousands of times a second to ask where it is, and builds a picture of where the time went from those interruptions. Sampling needs the counters to raise an interrupt — to stop the program and hand control to the kernel — when they overflow, and that is a separate hardware feature from counting.

Test it with a program that is actually running. The first of these commands looks like a test and is not:

```
perf record -F 999 -e cycles -o /tmp/perf.data -- sleep 2    # proves nothing
perf report --stats -i /tmp/perf.data | grep SAMPLE
```

A sleeping process is off the CPU, so it retires no instructions and burns no cycles, and a perfectly working PMU returns almost nothing. The command succeeds, the sample count is near zero, and you have learned nothing about the machine. Give it something to sample instead:

```
perf record -F 999 -e cycles -o /tmp/perf.data -- \
python3 -c 'x = 0
for _ in range(4_000_000): x += 1'
perf report --stats -i /tmp/perf.data | grep SAMPLE
```

Now the sample count is the answer, and `bench/run_setup.py` asks exactly this question the same way — because the first version of it ran `perf record -- true`, believed the zero exit status, and would have declared a board capable of something it had never been asked to do:

`run_setup.py`

```
def perf_can_sample() -> dict[str, Any]:
    """Whether ``perf record`` can sample on this board, which is a separate question.

    Counting and sampling are different capabilities and a board can have the first without the
    second. Sampling needs the counters to raise an overflow interrupt, which on RISC-V means the
    Sscofpmf extension @riscv-sscofpmf; a kernel without it says so at boot and then refuses.
    The SiFive U74 does not implement it. That is the difference between the OS-cost chapter,
    which counts, and the profiling chapter, which samples, so it is recorded as a fact about the
    board rather than discovered in a
    chapter.

    **This asks for samples rather than for an exit code.** An earlier version ran
    ``perf record -- true`` and believed a zero return, which is the same mistake as believing a
    counter that reads zero: ``perf`` will open the event, collect nothing from a program that
    barely runs, and exit successfully. The question is not whether the command succeeded, it is
    whether any samples came back — so this spins deliberately and counts them.
    """
```

An exit status is not evidence. It is the same mistake as believing a counter that reads zero, and it is worth meeting twice in one chapter.

On ARM, overflow interrupts are a standard PMU feature. On RISC-V they are the **Sscofpmf** extension RISC-V INTERNATIONAL (N.D.), and a kernel on a core without it says so at boot and then declines:

```
riscv-pmu-sbi: Perf sampling/filtering is not supported as sscof extension is not available
```

[ch30](#) is entirely about sampling, so on a machine that cannot do it that chapter has nothing to measure. `verify-setup.py` reports the two capabilities separately, precisely so you find out now rather than three hundred pages in.

A profiler that samples answers a different question from a counter that totals, and it fails in different ways. That distinction generalises well beyond RISC-V. [ch24](#) takes it apart properly and [ch30](#) depends on it.

What we measured

The board’s own account of itself, taken on the board: what it is, what its counters do, and whether `perf` can both count and sample. Every figure in [Part V](#) is measured on the machine this describes, which is why it is recorded rather than assumed.

What	This board
Board (device tree)	Raspberry Pi 5 Model B Rev 1.1
Operating system	Debian GNU/Linux 13 (trixie)
Kernel	Linux 6.18.50+rpt-rpi-2712
Cores online	0-3
Native compiler	gcc (Debian 14.2.0-19) 14.2.0
CPU implementer	0x41
CPU architecture	8
CPU variant	0x4
CPU part	0xd0b
CPU revision	1
Features	fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp asimdhp cpuid asimdrrm lrcpc dcpop asimddp
<code>perf stat</code> reads hardware counters	yes
Cycle counter event	cycles
<code>perf record</code> can sample	yes
Samples collected in the capability check	411

Conditions: target `host` · Raspberry Pi 5 Model B Rev 1.1 · Linux 6.18.50+rpt-rpi-2712 · gcc (Debian 14.2.0-19) 14.2.0 · `-O2 -g -Wall -Wextra` · 2026-09-16 · Source: `bench/results/setup-host.json`, code hash `a4bffd44a01dad4`.

What this cannot tell you

Whether your numbers will match the book’s. They will not, exactly, and they are not meant to. The committed figures came from one board with one kernel and one firmware; yours differs in at least the last two. What should match is the *shape* — which of two things is faster, and by roughly how much — and where it does not, the chapter that prints the figure says what it depended on.

Whether a counter counts what its name says. `perf` reports the name the kernel gives an event, and the kernel takes that from a table it was told. This chapter establishes that counting and sampling work at all; whether a given event means what you think is a question for [Appendix C](#), which is generated from the board rather than written.

Anything about the board under load. Everything here is a single quiet measurement on an idle machine. Thermal throttling, a busy neighbour and a governor that changes frequency mid-run are all real, all capable of moving a number more than the thing being measured, and all [ch24](#)'s subject.

Problems

Two, in `tests/setting_up_the_board/`. Both are Python stubs: this chapter comes before any C.

1.1 — Is this counter real? Six `perf stat` outcomes. Say which of them are counts of something the hardware did, and which are the kernel estimating or declining.

```
python3 -m pytest tests/setting_up_the_board/test_problem_1_counters.py
```

1.2 — Can this machine sample? Counting and sampling are different capabilities and a machine can have the first without the second. Given what a machine reports, say which of the book's parts it can finish.

```
python3 -m pytest tests/setting_up_the_board/test_problem_2_sampling.py
```

Where to go next

[Appendix H](#) states the requirements this chapter assumes, and `hardware/README.md` is the same thing as a checklist. The counter-overflow interrupt that sampling needs is the Sscofpmf extension RISC-V INTERNATIONAL (N.D.) on RISC-V and a standard part of the PMU on ARM; the reference board's SoC is documented by its vendor RASPBERRY PI LTD (N.D.).

[ch02](#) is next, on either machine. It teaches reading what the compiler produced, which every chapter after it assumes you can do.

Part I — C, and what the machine does with it

Part I · C, and what the machine does with it · 02 · Reading a Listing · 03 · Memory Is One Array · 04 · Kernel C Is Not Application C · 05 · Reading C as Machine Code

Part I · C, and what the machine does with it

PART HEADER

Chapters	ch02–ch05
Target	xv6 — the teaching kernel under QEMU, with ch02 also compiling for the board's architecture
Assumes	ch00, and fluency in some other language

What this part is for

Four chapters. One of them teaches reading what a compiler produced, and the other three have one job between them: enough C to read a kernel and change it, and no more.

The limit is deliberate. C is a large language and most of it never appears in the code this book reads, so a part that set out to cover the language would spend most of its length on things you are never going to meet here.

What stops people reading kernel code is not C's syntax. It is the memory model your language was built to hide: memory is one array of bytes, everything in it has an index, and a type is mostly a statement about how wide a step is. Every language has that underneath. Yours decided you did not need to see it; C decided you did.

This part is first because everything after it is written in C and half of it asks you to change some. [Part II](#) builds the machine's primitives in C and its problems have you rewriting a trap handler; [Part IV](#) reads a kernel; [Part V](#) compiles C and reads what came out. None of those is an exercise in the language. All of them are impossible without the model.

What it leaves out

Control flow, operators and functions. You already have these, and C spells them much as your language does. Where it differs the chapters say so and move on.

The standard library, which is [ch04](#)'s subject from the opposite direction: a kernel has almost none of it, and that chapter asks what it does instead. Floating point, for the same reason — the kernel does not use it, and why it does not is a better question than how it works.

And the craft of C as a language you would *build something in*: ownership discipline, API design, build systems, the long list of ways to invoke undefined behaviour. The problems in this book ask you to change a program that exists and predict what the change does — which needs the model and almost none of the craft. That is not a soft option. It is everything the rest of the book needs, and three chapters can reach it, where the craft would take far longer.

Where to start

Read the four in order. [ch02](#) is short and assumes no C, and every chapter after it puts a listing in front of you. If you already write C, [ch03](#) is the memory model you already have — skim it rather than skip it, because [ch04](#) is where it stops being the C you know.

[ch03](#) does not open with pointers: it opens with one complete program that prints four numbers, and the pointers arrive afterwards as a way of explaining numbers you have already watched appear.

Which machine, and what it cannot tell you

Most of this part is compiled for RISC-V and run on the xv6 kernel under QEMU; [ch02](#) also compiles for the board's architecture and runs nothing at all. Three of the four chapters ask you to read the disassembly of code you wrote — [ch02](#) for both of the book's instruction sets, the other two for RISC-V.

Nothing in this part is timed, and nothing in it may be. QEMU models no cache, no branch predictor, no store buffer and no memory latency, so a duration measured inside it is a fact about the laptop running the emulator, not about the machine being emulated. The figures here are therefore all structural — what a program printed, what instructions the compiler emitted — and the question of what any of it *costs* is not asked until [Part V](#), on hardware that can answer it.

Where this leaves you

Not a C programmer. Able to read a C program, and to change one and say what the change will do — which is what the rest of the book requires, and [Part II](#) starts requiring it immediately.

02 · Reading a Listing

CHAPTER HEADER

Target	xv6 and host — the same function compiled for each, and nothing executed
Prerequisites	ch00
What it measures	What the compiler made of one small function on each architecture: <code>bench/results/shapes-riscv64.json</code> , <code>bench/results/shapes-aarch64.json</code>

The question

What is the compiler telling me, and what is it not?

Almost every chapter from here on puts a listing in front of you — the instructions a compiler produced from source you can read beside it. Three of them are in the next chapter. You are never asked to write assembly, and you are never asked to know every mnemonic; you are asked to say what the compiler did, which is a much smaller skill and takes one chapter to acquire.

Acquire it now rather than in passing, because the notation that carries most of the weight — parentheses, which mean memory — is also the most likely to be read as something else.

The material

One function, two instruction sets

Here is a function with no cleverness in it at all:

shapes.c

```
/* Two conditionals and three exits. The interesting question is whether the compiler emits
 * branches or conditional moves, and the two architectures answer it differently — which makes
 * this the smallest honest demonstration that instruction selection is not a detail. */
int sysfs_clamp(int value, int low, int high) {
    if (value < low) {
        return low;
    }
    if (value > high) {
        return high;
    }
    return value;
}
```

Compiled for each of the book’s two architectures, at the same optimisation level — how hard the compiler was asked to try — by the same version of the same compiler. The conditions line under each listing says exactly which:

sysfs_clamp compiled for riscv64 at **-O2** — the **xv6** target’s instruction set, 10 instructions.

```

0000000000000000 <sysfs_clamp>:
 0:  blt    a0,a1,18 <.L4>
 4:  mv     a1,a0
 6:  blt    a2,a0,10 <.L6>
 a:  sext.w a0,a1
 e:  ret

0000000000000010 <.L6>:
10:  mv     a1,a2
12:  sext.w a0,a1
16:  ret

0000000000000018 <.L4>:
18:  mv     a0,a1
1a:  ret

```

Conditions: target `xv6` · riscv64 cross compiler, any machine — nothing here was executed · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · `-O2 -Wall -Wextra -march=rv64gc -mabi=lp64d -c` · 2026-09-17 · Source: `bench/results/shapes-riscv64.json`, code hash `96ffacfb103ddd44`.

sysfs_clamp compiled for aarch64 at `-O2` — the `host` target’s instruction set, 7 instructions.

```

0000000000000000 <sysfs_clamp>:
 0:  cmp     w0, w1
 4:  b.lt    14 <sysfs_clamp+0x14> // b.tstop
 8:  cmp     w0, w2
 c:  csel    w0, w0, w2, le
10:  ret
14:  mov     w0, w1
18:  ret

```

Conditions: target `host` · aarch64 cross compiler, any machine — nothing here was executed · aarch64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · `-O2 -Wall -Wextra -c` · 2026-09-17 · Source: `bench/results/shapes-aarch64.json`, code hash `eb70bdd44fe4bbb2`.

The rest of this chapter is how to read those two.

How a listing is laid out

The left-hand column is an address, in hexadecimal, counted from the start of the function. `0:` is the first instruction, and the number on the next line tells you how many bytes it took — so a second instruction at `4:` means the first was four bytes long. In the RISC-V listing you will also find instructions two bytes apart, because RISC-V has short forms of its commonest instructions RISC-V INTERNATIONAL (N.D.) and the compiler uses them unasked. Instructions here are *not* all the same length, and [ch06](#) turns on that fact.

Then the mnemonic: `blt` is branch-if-less-than, `mv` is move, `ret` is return. You are not expected to know these, and this book never asks you to write assembly — only to read enough of it to say what the compiler did.

Then the operands, destination first. `mv a1,a0` copies `a0` into `a1`, not the other way round. That order is the assembler’s convention RISC-V INTERNATIONAL (N.D.) and it is the one thing most likely to mislead you if you skim.

`a0`, `a1`, `a2` are registers — the processor’s own named slots, the fastest storage a program has. The `a` ones carry arguments and return values RISC-V INTERNATIONAL (N.D.), which is why a function’s first argument arrives in `a0` and its answer leaves in `a0`. [Appendix A](#) lists them all.

Parentheses mean memory, and the number in front is a byte offset. `4(a0)` is the memory at the address in `a0`, plus four bytes — not `a0` times anything. Nearly every listing in [Part I](#) turns on that one piece of notation, and the offset is usually the width of whatever is stored at that address, which is how the same C source becomes `4(a0)` in one function and `8(a0)` in another.

Everything under a listing is provenance: which compiler, which flags, which result file it was captured from. It is there so that a listing you find surprising can be regenerated rather than argued about.

What the two disagree about

Read the second comparison in each. AArch64 settles it with `csel` — compute both candidates, select one, never branch. RV64GC — the particular set of RISC-V instructions xv6 is built for, and the one every RISC-V listing in this book was compiled for — has no conditional select RISC-V INTERNATIONAL (N.D.), so the same decision has to be a branch, and the function comes out with three separate exits.

That is a real difference, and it is not a difference in speed. Nothing above says which is faster, and *What this cannot tell you* below says why not.

Two smaller things are visible in the same listings. The RISC-V one has labels — `.L4`, `.L6` — sitting inside the function, and the AArch64 one has none; that is a division of labour between two tools you meet in [ch12](#) and [ch15](#), and for now it is enough to have noticed it. And `sext.w`, which RISC-V emits on each path and AArch64 does not emit anywhere: RISC-V keeps a 32-bit `int` in a 64-bit register and has to say so, while AArch64 has a 32-bit view of the register and uses it. Neither is in the C. Both are the kind of thing [ch14](#) is for.

NONE OF THAT WAS TYPED

`bench/run_disasm.py` compiled `sysfs/lib/shapes.c` for each architecture, ran `objdump` on the object file, and wrote a stamped result. The block above is rendered from those results, and CI regenerates both on every push and fails if one instruction differs.

It can do that because a listing depends on the compiler and not on the machine — so unlike every number in [Part V](#), this one is checked automatically, every time.

What we measured

One function, compiled for each of the book's instruction sets by the same compiler at the same optimisation level. Nothing was executed and nothing was timed: a listing is a fact about a compiler, so it is the one kind of result in this book that CI can regenerate and check on every push. [ch24](#) is about the kind that cannot be.

What this cannot tell you

Which of the two is faster. This is the mistake the chapter exists to prevent, and it is tempting precisely because the listings are short enough to count. AArch64 settles the second comparison without branching and RISC-V cannot, so one comes out shorter — and that tells you nothing about time. A branch the processor guessed right is nearly free and one it guessed wrong is not; `csel` pays a fixed price either way, and has to wait for both candidates before it can choose. Which wins depends on the data the function is given, and the only way to find out is to run both on hardware that can be asked. [ch26](#) and [ch27](#) do that.

What any of these instructions costs. A mnemonic is a name, not a price. Nothing in a listing says how many cycles an instruction takes, whether its operands were in cache, or whether the core ran it at the same time as its neighbour. Those are properties of a machine and this page has not been near one.

Whether the compiler was right. It made these choices at one optimisation level, for one target, from one version of one compiler — all of which the conditions line records, because all of them change the answer. A different flag produces a different listing from identical source, which is [ch26](#)'s subject.

Anything about a program. One function with no calls in it is the easiest possible listing. Real code spends its instructions on stack frames, calling conventions and address arithmetic, and none of that is visible here. [ch14](#) is where a listing stops being a curiosity and starts being a thing you read to answer a question.

Problems

Three, in `tests/reading_a_listing/`. Each is a Python stub: this chapter comes before any C.

2.1 — How long is each instruction? Given the address column of a listing, say how many bytes each instruction occupied. RISC-V has short forms of its commonest instructions and the compiler uses them unasked, so the answer is not the same number every time.

```
python3 -m pytest tests/reading_a_listing/test_problem_1_lengths.py
```

2.2 — Does this operand touch memory? Eleven operands drawn from both instruction sets. Say which of them name a register or a literal, and which reach memory. One pair differs only by punctuation and disagrees.

```
python3 -m pytest tests/reading_a_listing/test_problem_2_memory.py
```

2.3 — How wide was the element? Given a load and its offset, say how many bytes wide the thing being stepped over was. This is the same fact the next chapter opens on, approached from the instruction rather than from the C.

```
python3 -m pytest tests/reading_a_listing/test_problem_3_width.py
```

Where to go next

[Appendix A](#) is the RISC-V register and CSR reference, and [Appendix F](#) is this page's counterpart for AArch64 — three differences that change how a listing reads. The assembler syntax and the register roles are defined in the RISC-V unprivileged specification RISC-V INTERNATIONAL (N.D.) and the psABI RISC-V INTERNATIONAL (N.D.).

[ch03](#) is next: one complete program, and its listing read the way this chapter has just shown you.

03 · Memory Is One Array

CHAPTER HEADER

Target	xv6 — the teaching kernel under QEMU
Prerequisites	ch00
What it measures	What one complete program prints: <code>bench/results/firstc.json</code> ; what <code>p + 1</code> and <code>-></code> become: <code>bench/results/declarations-riscv64.json</code>

The question

If memory is one array of bytes, what is a C declaration saying about it?

A declaration is not read left to right, and every piece of one turns into something concrete — an amount of storage, a width to step by, an instruction the compiler emits. This chapter takes declarations a few at a time and checks what each one means against what the compiler emitted, rather than against a rule for reading them.

Change the program as you go. `./run firstc` builds and runs it, and the fastest way to find out whether you have the model is to predict what an edit will print before you make it.

If you are not sure whether you want this chapter or [ch04](#), [Part I](#) routes you.

The material

One whole program

Every piece of C you have seen in this book so far has been a fragment — lifted out of a file, with no beginning and no end. Here is a complete one, and it is the shortest complete one this book has.

```

/* firstc – one complete C program, before any of the parts.
 *
 * Copyright 2026 Chris Snow. Apache-2.0 – see LICENSE-CODE.
 *
 * The memory-as-one-array chapter shows this file whole, because until now every piece of C in this
book has been a
 * fragment lifted out of somewhere larger. A reader who has never written C has never seen where
 * the pieces go.
 *
 * It is also the chapter's first measurement, and it is deliberately not an address. Printing one
 * would show a number that differs on every run and means nothing on its own; what is worth
 * seeing is the *relationships* between addresses, which are the same every time:
 *
 * that `*&x` is `x` again, so `&` and `*` undo each other;
 * that `p + 1` moves by the size of what `p` points at, and by nothing else.
 *
 * cc -o firstc firstc.c && ./firstc
 */

#include <stdint.h>
#include <stdio.h>

int main(void) {
    int value = 42;

    /* `&value` is where it lives; `*` goes back to what lives there. The two are inverses, and
 * this line is the whole of that claim. */
    int *where = &value;
    printf("firstc value %d roundtrip %d\n", value, *where);

    /* Two arrays, same length, different element type. The distance between one element and the
 * next is not one byte – it is one *element*, and how big an element is comes from the type. */
    int32_t narrow[2];
    int64_t wide[2];
    printf("firstc step int32 %td\n", (char *)&narrow[1] - (char *)&narrow[0]);
    printf("firstc step int64 %td\n", (char *)&wide[1] - (char *)&wide[0]);

    printf("end firstc\n");
    return 0;
}

```

Four things in it are not the subject of this chapter and are worth naming once so they stop being in the way.

`#include` pastes in a file of declarations, so that `printf` is a name the compiler has heard of before you use it. `main` is where a program starts, and there is exactly one. `printf` writes to the terminal, and the `%d` in its first argument is a hole the next argument fills. `return 0` at the end of `main` is the program's exit status, and zero means nothing went wrong.

Build it and run it:

```
./run firstc
```

That is the whole loop this part asks you to get into: change the source, run it again, look at what came out. `./run` knows how every program in this book is built — which compiler, which flags, which machine — so the only thing you have to think about is the source. `./run --list` shows the lot.

What the program printed	Value
<code>value</code>	42
<code>*&value</code> — the same thing again	42
Bytes from one <code>int32_t</code> to the next	4
Bytes from one <code>int64_t</code> to the next	8

Conditions: `target host · aarch64 cross build, any machine — run under user-mode QEMU for counts only · aarch64-linux-gnu-gcc · -O2 -g -Wall -Wextra -static · 2026-09-17 · Source: bench/results/firstc.json, code hash dc5dabccb8df99aa.`

The rest of the chapter explains why those four numbers are what they are.

Memory is one array, and everything has an index

The language you already use has named things that hold values, and a runtime that knows where they are. C has one array of bytes, numbered from zero, and everything you declare is a stretch of it. That is the model, not a simplification for teaching, and every difficulty later in the book is a difficulty about it.

Three questions follow, and they are the ones worth asking of any C you read:

- **Where does this live?** In the current call’s region, in one that lasts the whole run, or in memory somebody asked for.
- **How long does it stay there?** Until the function returns, for ever, or until somebody says otherwise.
- **How big is it?** A question about its *type*, and the reason types exist at all.

“Where” has a coarse answer and a precise one. The coarse one is the region. The precise one is an index — the position of the thing’s first byte in that one array — and an address is that index, and nothing else.

& and *, watched rather than described

Two operators, and they are inverses.

`&x` gives the index of where `x` lives. `*p` goes to the index held in `p` — a *pointer*, a variable whose value is an index — and reads what is there. So `*&x` is `x` again, which is the second row of the table above, printed by a program rather than asserted by a paragraph.

The program never prints an address, because an address is different every time the program runs and means nothing by itself; there is no fact in it. What is the same every time — and what is worth knowing — is the *relationship* between two addresses. The last two rows are that relationship, and they are the whole of pointer arithmetic.

Two arrays, both of two elements. The distance from the first element to the second is four bytes in one and eight in the other, and nothing in the expression that computed it said four or eight. The type did.

A pointer is an index with a type stapled to it

`int *p` says: `p` holds an index, and what lives at that index is an `int`. The type is not decoration. It is how the compiler knows how many bytes to take, how to interpret them, and — this is the part that surprises people — **how far one step is**.

That is the fact from the outside. Here it is from underneath.

Two functions, each taking a pointer and returning the element after the one it points at. The source of each is the same three characters, `p + 1`.

declarations.c

```
int32_t sysfs_step_narrow(const int32_t *p) { return *(p + 1); }
int64_t sysfs_step_wide(const int64_t *p) { return *(p + 1); }
```

If `4(a0)` and the address column are unfamiliar, [ch02](#) is the key for reading these. What is needed here is one line of it: parentheses mean memory, and the number in front is a byte offset, so `4(a0)` is the memory four bytes past the address in `a0`. Watch that number.

The narrow one:

sysfs_step_narrow compiled for riscv64 at -02 — the `xv6` target’s instruction set, 2 instructions.

```
0000000000000000 <sysfs_step_narrow>:
0:   lw      a0,4(a0)
2:   ret
```

Conditions: target `xv6` · riscv64 cross compiler, any machine — nothing here was executed · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · -02 -Wall -Wextra -march=rv64gc -mabi=lp64d -c · 2026-09-17 · Source: [bench/results/declarations-riscv64.json](#), code hash `2cf5e7f79ec7d0c2`.

The wide one:

sysfs_step_wide compiled for riscv64 at -02 — the `xv6` target’s instruction set, 2 instructions.

```
0000000000000004 <sysfs_step_wide>:
4:   ld      a0,8(a0)
6:   ret
```

Conditions: target `xv6` · riscv64 cross compiler, any machine — nothing here was executed · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · -02 -Wall -Wextra -march=rv64gc -mabi=lp64d -c · 2026-09-17 · Source: [bench/results/declarations-riscv64.json](#), code hash `2cf5e7f79ec7d0c2`.

One load each, and they are not the same load. The narrow one loads a word from four bytes along; the wide one loads a doubleword from eight. (The `ret` after it is the function returning, which is why each listing is two instructions rather than one.) **+ 1 never means “one byte”**. It means one *element*, and the element size comes from the type, and the type is not in the expression you are reading.

This is why a pointer’s type matters even when you never dereference it — never apply `*` to it — and it is why a cast to a different pointer type changes what arithmetic on it does. Nothing about `p + 1` on the page tells you the answer; you have to know what `p` was declared as.

Reading a declaration

C declarations are famously hard to read and the reason is that they are written to *mirror use*. `int *p` is not “p is a pointer to int” written awkwardly. It is “the expression `*p` has type `int`” — the declaration shows you the shape of the thing that finally yields an `int`.

Once you know that, the rule is mechanical: **start at the name, and read outwards, taking whatever binds tightest first**. `[]` and `()` bind tighter than `*`, and parentheses override.

Declaration	Read it as
<code>int *p</code>	<code>p</code> is a pointer to <code>int</code>
<code>int p[4]</code>	<code>p</code> is an array of four <code>int</code>
<code>int *p[4]</code>	<code>p</code> is an array of four pointers to <code>int</code>
<code>int (*p)[4]</code>	<code>p</code> is a pointer to an array of four <code>int</code>
<code>int *(f)(void)</code>	<code>f</code> is a function returning a pointer to <code>int</code>
<code>int (*f)(void)</code>	<code>f</code> is a pointer to a function returning <code>int</code>
<code>const int *p</code>	<code>p</code> points at an <code>int</code> you may not write through
<code>int *const p</code>	<code>p</code> is a pointer you may not repoint

The two pairs that differ only by parentheses are the ones worth doing slowly, and they are not a puzzle for its own sake: `int (*f)(void)` is how every device driver in [ch19](#) is reached, and `const int *` against `int *const` is a distinction the kernel relies on constantly to say which of two things it promises not to change.

Problem 3.1 is this table, generated rather than reproduced, with declarations you have not seen.

Arrays, and why a pointer cannot carry a length

An array is a run of elements, and its name in most expressions turns into the index of its first one. Pass an array to a function and what arrives is a pointer — the length does not travel with it, and cannot, because there is nowhere in a pointer to put it.

That is why every kernel function that takes a buffer also takes a count. It is not a style preference; there is no alternative. [ch05](#) shows the compiler discarding a length written into a parameter’s brackets, and [ch13](#) has the bug it causes.

A string in C is this with one extra convention: a run of bytes ending in a zero byte. The zero is the length, stored at the end instead of the beginning, which makes finding the length a loop rather than a lookup and makes forgetting to write it a bug that reads off the end of the array.

Structs, and what `->` really is

A struct is several objects at fixed offsets from one address. `record->third` means: take the index in `record`, add the offset of `third`, and read what is there.

```
int64_t sysfs_reach_through(const struct sysfs_record *record) {
    return record->third + record->second;
}
```

sysfs_reach_through compiled for riscv64 at -O2 — the `xv6` target’s instruction set, 4 instructions.

```
0000000000000008 <sysfs_reach_through>:
8:  lw    a4,4(a0)
a:  ld    a0,8(a0)
c:  add   a0,a0,a4
e:  ret
```

Conditions: target `xv6` · riscv64 cross compiler, any machine — nothing here was executed · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · -O2 -Wall -Wextra -march=rv64gc -mabi=lp64d -c · 2026-09-17 · Source: [bench/results/declarations-riscv64.json](#), code hash `2cf5e7f79ec7d0c2`.

Two loads and an add. **There is no member lookup.** The names `second` and `third` do not exist by the time the machine sees this; they became the numbers in those two load instructions, decided when the file was compiled. `a->b` is exactly `(*a).b`, and both are exactly “an offset on a load”.

Which offsets, and why they are not simply the sum of the sizes before them, is [ch13](#)’s subject. What matters here is that the offsets are fixed at compile time and the name is gone.

Casts, and the two kinds of arithmetic

A cast says “treat these bytes as this type instead”. Between pointer types it changes nothing about the bytes and everything about what arithmetic on them means, because the type is what sets the step size.

A kernel casts constantly between a pointer and a plain integer, because an address genuinely *is* a number there — a physical page is a number, a device register is a number — and the two kinds of arithmetic are different:

- **Pointer arithmetic** scales by the element. `p + 1` moves by `sizeof(*p)`, the size in bytes of what `p` points at.
- **Address arithmetic** does not. `(uint64)p + 1` moves by one byte.

Rounding an address to a page boundary is address arithmetic, which is why the kernel’s rounding macros cast to an integer type first and back afterwards. Problem 3.3 is writing the pair, and the thing to get right is that rounding *up* and rounding *down* are not the same expression with a different sign.

A structure that points at its own kind

The last construct, and the one that makes the next chapter possible.

```
struct node {
    int value;
    struct node *next;
};
```

A struct cannot contain itself — it would have no size — but it can contain the *index* of another one, because an index has a size regardless of what it points at. That single fact is what every list, tree and queue in the kernel is built from, and [ch04](#) shows the free list xv6 makes out of the free memory itself.

What we measured

Nothing here is a cost. Two things, and they are the same fact seen from two sides.

First, what the program printed — the chapter’s claims in a form you can reproduce: that `*x` is `x`, and that consecutive elements are one element apart rather than one byte. The runner refuses to stamp a run where either stops holding, because the chapter asks the reader to take both on trust for exactly as long as it takes to run the program.

Then three listings, captured by the machinery [ch12](#) explains and re-captured by CI on every push, showing the same two facts from underneath: that `p + 1` compiles to a different offset for different element types, and that `->` compiles to an offset on a load with no lookup of any kind.

Both are facts about what this compiler did. That they are *required* is a claim about the standard ISO/IEC JTC1/SC22/WG14 (2018) and would hold on a compiler emitting entirely different instructions.

What this cannot tell you

How to program. Deliberately. Control flow, functions, operators and the shape of a program are assumed from whatever language you already use, and C spells them much as it does. What is not assumed is the memory model, because that is the part your other language was built to hide.

What any of this costs. No chapter in this part has a clock in it. [ch25](#) is where following a pointer acquires a price, and the price turns out to depend on something no listing here can show.

Why the struct offsets are what they are. This chapter shows that `->` becomes an offset. It does not say how the offsets were chosen, and the answer is not “add up the sizes” — [ch13](#) takes it apart.

Whether your pointer is valid. C has no answer to this and neither does the hardware, until [ch18](#). A pointer is an index; nothing checks that the index means anything.

Problems

Three, in `tests/memory_is_one_array/declarations.c`.

3.1 — Say what each declaration names. The test generates declarations and asks you to classify them. The two that differ only by parentheses are the point of the exercise, and they are the two you will meet again in [ch19](#).

```
python3 -m pytest tests/memory_is_one_array/test_problem_1_declarations.py
```

3.2 — Walk a buffer without an index. Reimplement three of the routines the kernel supplies for itself, using pointers that move rather than a subscript. The versions in `kernel/string.c` are written this way, and the exercise is so that you can read them.

```
python3 -m pytest tests/memory_is_one_array/test_problem_2_walking.py
```

3.3 — Round an address both ways. Given an address and a power-of-two alignment, round up and round down. Address arithmetic, not pointer arithmetic, and the kernel does this on every page it touches.

```
python3 -m pytest tests/memory_is_one_array/test_problem_3_rounding.py
```

Where to go next

The C standard ISO/IEC JTC1/SC22/WG14 (2018) is the authority on the claims here that are about the language rather than about a compiler — §6.5.6 on what adding an integer to a pointer means, and §6.7.6 on how a declaration is assembled from the inside out.

[ch04](#) is next, and it is the chapter for a reader who already writes C. Every assumption an application programmer is entitled to make — that allocation succeeds, that the library is there, that memory is memory, that one thread is looking — stops holding, and the chapter counts what is missing rather than asserting it.

04 · Kernel C Is Not Application C

CHAPTER HEADER

Target	xv6 — the teaching kernel under QEMU
Prerequisites	ch03
What it measures	What the kernel as built does not have: <code>bench/results/kernelc-xv6.json</code>

The question

I already write applications — which of my habits stop working in a kernel?

The question is not about C, and you do not have to have written C to have the habits this chapter is about: a Java or Python programmer holds more of them than a C one, not fewer, because more has been done for them. Every assumption below is one an application programmer is entitled to make in any language, and none of them holds here.

What goes wrong is not syntax; [ch03](#) covered that.

The material

One fact, and everything else follows from it

There is nothing underneath you. An application runs on a library, which runs on a kernel, which runs on hardware. A kernel runs on hardware. Every difference in this chapter follows from removing those two layers; read it that way rather than as a list.

Here is what removing them costs, counted from the kernel that is checked in.

The kernel as built	Value
Instructions in the kernel	9,346
...that name a floating-point register	0
Heap functions it defines	0
C library functions it reimplements	13

Conditions: target `xv6` · the staged kernel, read rather than booted · `xv6-riscv @ 9e3161a` · `xv6`'s own `CFLAGS` · `riscv64-linux-gnu-objdump -d; riscv64-linux-gnu-nm --defined-only` · 2026-09-17 · Source: `bench/results/kernelc-xv6.json`, code hash `0c5f529c4ab71c65`.

Three of those rows are what the rest of this chapter is about.

No heap

There is no `malloc`, the library call an application makes when it wants memory, and no *heap*, the region `malloc` hands memory out from. Nothing in the kernel defines either, and the row in the table above is a count rather than a claim. The reason is an ordering problem rather than an omission. `malloc` is built on a kernel's memory management, and this *is* the memory management. It cannot call itself into existence.

So where do objects come from? Two places, and a kernel of this size uses both.

A fixed array, decided when the kernel was compiled.

Bound	Value	What it limits
MAXARG	32	arguments to <code>exec</code>
NBUF	30	disk blocks cached
NCPU	8	harts
NDEV	10	device drivers
NFILE	100	open files, system-wide
NINODE	50	in-memory inodes
NOFILE	16	open files, per process
NPROC	64	processes

Conditions: target `xv6` · the staged kernel, read rather than booted · `xv6-riscv @ 9e3161a` · `xv6`'s own `CFLAGS` · `riscv64-linux-gnu-objdump -d; riscv64-linux-gnu-nm --defined-only` · 2026-09-17 · Source: `bench/results/kernelc-xv6.json`, code hash `0c5f529c4ab71c65`.

Every one of those is the length of an array that exists for the whole run. The kernel does not allocate a process; it finds an unused slot in `proc[]` and marks it used. When there is no unused slot, `fork` — the call that creates a process — fails, and *that* is what the limit means. A reader used to allocation that either succeeds or raises has to get used to allocation that returns a *null pointer* — a pointer whose value is zero, C's spelling of *nothing here* — and expects the caller to check.

A free list made of the free memory itself. The other source is whole *pages* — the fixed-size blocks the hardware divides memory into, and the unit it is handed out in. A list of free pages needs a node per page, which would need memory, which is what we are trying to allocate. The kernel resolves it by writing the link *into the free page*, because a free page by definition holds nothing anybody wants.

kalloc.c

```
struct run {
    struct run *next;
};
```

xv6, `kernel/kalloc.c` — MIT licence

That is `ch03`'s self-referential struct, and here it is doing real work. This is the whole of what freeing a page does to it:

kalloc.c

```
r = (struct run *)pa;

acquire(&kmem.lock);
r->next = kmem.freelist;
kmem.freelist = r;
```

xv6, `kernel/kalloc.c` — MIT licence

Cast the page's address to a `struct run *`, write the current head of the list into the page's first bytes, and make the page the new head; allocating takes the head and follows the link it finds there. Three lines of pointer arithmetic that would be undefined behaviour in an application and are the allocator here — and one `acquire`, which is *Somebody else is running* below. [ch18](#) measures what it costs.

Undefined behaviour — the C standard's name for anything it declines to give a meaning to — is not a figure of speech here. C is defined in terms of an *abstract machine*, the standard's own model of what a C program does, written without reference to any real hardware. In that model a pointer points at an *object*: something created by a declaration, or by an allocator, with a lifetime the standard describes. `pa`, the address `kfree` was handed, is none of those. It is an integer the kernel's build and the hardware agree is the address of usable memory, cast to a pointer, and the abstract machine has no concept that would make the cast meaningful. Writing through it is undefined not because it is dangerous but because the standard has nothing to say about it.

It works anyway because the compiler is not the last word on this program. Declarations, objects and lifetimes exist so that a compiler can reason about a program without asking the hardware; here the hardware is the authority, and the kernel is asserting a fact about the machine that C has no way to express. That is the real reason kernel C cannot be read as the C of the standard with some library calls missing — it is the same language making a different bargain about who decides what an address means.

Memory that is not memory

An address in an application is somewhere your bytes are kept. In a kernel, some addresses are wires. Writing a byte to one sends a character out of a serial port; reading one twice gives two different answers because the second read consumes the next byte to arrive.

Every assumption a compiler makes about memory is wrong for those addresses — that reading twice gives the same answer, that a read nobody uses can be dropped, that two writes to one place can be combined into the last.

`volatile`, a word put on the declaration, is how you withdraw those assumptions, and [ch05](#) shows the compiler obeying, one instruction at a time.

This is why kernel source is full of a keyword application code almost never needs. `volatile` is not defensive style; it is the difference between a *driver* — the part of a kernel that talks to one device — and a program that receives one character for ever.

Somebody else is running

An application with one thread has exclusive access to its own data by default. A kernel never does: `NCPU` in the table of bounds above gives this machine eight harts, every one of them able to be inside the same function as you, on data you are halfway through changing.

What that costs is [ch20](#)'s subject and it is not small. When you read a kernel structure, ask who else can reach it, and what stops them reaching it at the same moment as you. That is not paranoia; it is the first question. `static` on a variable declared outside any function does not make it yours — it hides the name from other *files*, and every hart runs the same file.

No floating point, on purpose

Zero instructions in the whole kernel name a floating-point register. That is a decision rather than an oversight, and it is the kind of decision a kernel gets to make.

Floating-point registers are part of a process's state. Saving and restoring them on every *context switch* — the moment the kernel stops running one process and starts another — costs time on every switch, whether or not the process ever used one. xv6 declines: it does not save them, so the kernel may not use them either, because a kernel that used one would corrupt whichever process it interrupted. The cost of the feature is paid by everyone and the benefit accrues to few, so the feature is not offered.

Real kernels make the same trade differently — usually by saving the registers lazily, the first time a process touches one, which is [ch18](#)'s mechanism used for something other than memory.

Almost no library, so the kernel writes its own

The table counts the functions this kernel reimplements because nothing supplies them: the string and memory routines, and a formatted-print routine for the console. They are two short files, the plainest complete C in the tree, and written in exactly the style [ch03](#)'s second problem asked for — pointers that move, no subscripts, a length passed alongside every buffer. Read them early.

The string copy is the example to notice. The standard `strncpy` takes a size and does not promise to write the zero byte that ends a string; the kernel carries that one and, beside it, `safestrcpy`, which takes a size and always writes it. When a kernel writes its own version of something the library already has, the version usually differs on purpose, and the difference is usually about a failure the library was willing to tolerate.

Failure returns, it does not raise

There is no exception, no `finally`, no destructor and nothing to catch. A function that can fail returns something you must look at, and a caller that does not look has written the bug.

This is the habit that takes longest to acquire, because in an application ignoring a failure is usually survivable — something above you will notice. Here, nothing is above you. Problem 4.2 is deciding which of several plausible kernel functions can fail at all, which is a question about where their memory comes from — this chapter in one exercise.

What C lets you do that you must not

C will not stop you. Neither will the hardware, until [ch18](#). The three mistakes that cost the most time, all of which compile without a word of complaint:

- **A pointer to a local — a variable in the current call's region — after the function returned.** The bytes are still there and still readable, right up until the next call writes over them, which is why this produces a bug that works in testing.
- **Reading or writing past the end of an array.** Nothing checks. [ch13](#) has the arithmetic that makes a bounds check look right and be wrong.
- **Two harts writing one variable with no lock** — nothing to make one wait for the other — which works perfectly until the machine is busy. [ch20](#) is the chapter, and [ch28](#) is what it costs on real hardware.

What we measured

What the kernel does not have, counted from the kernel that is checked in rather than asserted: its instruction count, how many of those name a floating-point register, how many heap functions it defines, how much of the C library it carries its own copy of, and the compile-time bounds that stand in for an allocator.

`bench/run_kernel.c.py` refuses to stamp a census in which the kernel has acquired floating point or a heap. An absence is the weakest kind of claim to make and the easiest to go quietly stale, so the two the chapter argues from are the two that fail CI when they stop being true.

What this cannot tell you

How a production kernel does any of this. Linux has allocators, per-CPU data, lazy floating-point saving and a great deal more. xv6’s answers are the simplest ones that work, chosen so they can be read, and reading them is what makes a bigger kernel’s answers legible as choices rather than as complexity.

What any of it costs. No clock in this part. The free list is [ch18](#), locks are [ch20](#) and [ch28](#), and the context switch that declines to save floating-point registers is [ch21](#).

Whether your kernel C is correct. Nothing here is a checker. The habits in this chapter narrow where to look; they do not tell you that you have looked hard enough, and the problems in [Part IV](#) are about mechanisms that defeat careful reading entirely.

Anything about C++ or freestanding C beyond this kernel. “Freestanding” is a word the standard ISO/IEC JTC1/SC22/WG14 (2018) defines and this chapter has deliberately not used, because what actually matters is which specific things are missing here, and that is a property of xv6 rather than of a conformance mode.

Problems

Three, in `tests/c_without_a_runtime/runtime.c`.

4.1 — Hand out objects from a pool, and take them back. No allocator. A fixed array, a way to find an unused entry, and a way to return one. Getting the “none left” case right is most of the exercise, because it is the case an application programmer has never had to write.

```
python3 -m pytest tests/c_without_a_runtime/test_problem_1_pool.py
```

4.2 — Which of these can fail? Several kernel functions described by where their memory comes from. Say which can fail and what each must return when it does. The answer is never “it throws”.

```
python3 -m pytest tests/c_without_a_runtime/test_problem_2_failure.py
```

4.3 — What does a missing volatile cost? Given several loops over an address, say which the compiler may reduce and to what. One of them is a driver that receives one character for ever.

```
python3 -m pytest tests/c_without_a_runtime/test_problem_3_volatile.py
```

Where to go next

`kernel/string.c` and `kernel/kalloc.c` are the two files to read after this chapter — short enough for one sitting, and containing every idea above. [Appendix D](#) says what else is in the tree and which chapter reads it.

[ch05](#) is the last chapter of this part and the one that puts a compiler behind the claims. It sorts C's constructs by a single question — has the machine heard of this? — and answers it with disassembly rather than with assertion.

05 · Reading C as Machine Code

CHAPTER HEADER

Target	xv6 — the teaching kernel under QEMU
Prerequisites	ch04
What it measures	What the compiler emits for each construct: <code>bench/results/addresses-riscv64.json</code>

The question

Which parts of C are really about addresses, and how do I read them without flinching?

This is not a C tutorial and it does not cover the language. It covers the subset that becomes visible in machine code, sorted by a single question: **has the machine heard of this?** Some of C's most-argued-about constructs are instructions to the compiler that leave no trace at all. `volatile` is a prohibition that survives into every single load. Telling those apart makes kernel source readable, because a kernel is mostly the second kind.

The material

Everything here is an address

A pointer is an integer that happens to be the number of a byte, carrying a type that says how many bytes to take from there and how to read them. That is the whole idea, and almost every difficulty with pointers is really a difficulty about one of the questions below. [ch03](#) asked three questions of any declaration — *where does this live, how long does it stay there, how big is it*. A kernel needs a fourth, and it runs through this chapter: **who else can reach it**, and what the language lets you say about that.

`volatile` — the one qualifier the machine has heard of

Two functions, identical but for one word — a *qualifier*, one of the words C lets you attach to a type — each adding up four reads of the same address:

addresses.c

```
int sysfs_read_four(const int *slot) {
    int total = 0;
    for (int index = 0; index < READS; index++) {
        total += *slot;
    }
    return total;
}

int sysfs_read_four_volatile(const volatile int *slot) {
    int total = 0;
    for (int index = 0; index < READS; index++) {
        total += *slot;
    }
    return total;
}

/* `const int values[4]` looks like it promises four. It does not: an array parameter is a
 * pointer, the 4 is discarded, and sizeof inside this function measures a pointer. The proof is
 * that this and the function below it compile to the same instructions. */
```

The plain one:

sysfs_read_four compiled for riscv64 at -O2 — the `xv6` target's instruction set, 3 instructions.

```
0000000000000000 <sysfs_read_four>:
0:   lw      a0,0(a0)
2:   slliw   a0,a0,0x2
6:   ret
```

Conditions: target `xv6` · riscv64 cross compiler, any machine — nothing here was executed · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · -O2 -Wall -Wextra -march=rv64gc -mabi=lp64d -c · 2026-09-17 · Source: `bench/results/addresses-riscv64.json`, code hash `6ea4cbd2185fdb5b`.

One load. The compiler reasoned that nothing between the reads could change what is at that address, so it read once and multiplied — and it was right, under the rules it is given.

The `volatile` one:

sysfs_read_four_volatile compiled for riscv64 at -O2 — the `xv6` target's instruction set, 8 instructions.

```
0000000000000008 <sysfs_read_four_volatile>:
8:   lw      a5,0(a0)
a:   lw      a3,0(a0)
c:   lw      a4,0(a0)
e:   lw      a0,0(a0)
10:  addw     a5,a5,a3
12:  addw     a5,a5,a4
14:  addw     a0,a0,a5
16:  ret
```

Conditions: target `xv6` · riscv64 cross compiler, any machine — nothing here was executed · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · -O2 -Wall -Wextra -march=rv64gc -mabi=lp64d -c · 2026-09-17 · Source: `bench/results/addresses-riscv64.json`, code hash `6ea4cbd2185fdb5b`.

Four loads, in order, none of them removed.

`volatile` means one thing: **every access in the source must appear in the output, and in this order**. It does not mean *shared*, it does not mean *atomic*, and it is not a threading primitive — [ch20](#) is emphatic about that.

It matters when reading an address is not merely reading memory. A device register that returns the next byte of a serial port gives a different answer each time it is read, and a compiler that reads it once and reuses the value has turned your driver into a program that receives one character for ever. [ch19](#) writes that driver.

If you arrived from Java, this is the trap. Java's `volatile` is a threading primitive: it orders accesses between threads, and the language spells out what that guarantees. C's does none of that. It constrains the compiler and says nothing whatever to the hardware, so two harts can still see one another's writes in a different order from the one they were made in. The keyword is spelled the same and does a different job, and the instructions that do the other job are [ch20](#)'s.

An array parameter is a promise nobody keeps

addresses.c

```
/* `const int values[4]` looks like it promises four. It does not: an array parameter is a
 * pointer, the 4 is discarded, and sizeof inside this function measures a pointer. The proof is
 * that this and the function below it compile to the same instructions. */
int sysfs_sum_array(const int values[4]) {
    int total = 0;
    for (int index = 0; index < READS; index++) {
        total += values[index];
    }
    return total;
}
```

The declaration says four. The compiler discards that: an array parameter is a pointer, the size in the brackets is documentation, and `sizeof` — the operator that gives a thing's size in bytes — measures a pointer inside the function rather than an array. The proof is next to it in the same file — `sysfs_sum_pointer` takes a plain pointer, does the same work, and compiles to the same instructions:

`sysfs_sum_array` compiled for riscv64 at -O2 — the `xv6` target's instruction set, 8 instructions.

```
0000000000000018 <sysfs_sum_array>:
18:  lw    a3,0(a0)
1a:  lw    a5,4(a0)
1c:  lw    a4,8(a0)
1e:  lw    a0,12(a0)
20:  addw  a5,a5,a3
22:  addw  a5,a5,a4
24:  addw  a0,a0,a5
26:  ret
```

Conditions: target `xv6` · riscv64 cross compiler, any machine — nothing here was executed · `riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 -O2 -Wall -Wextra -march=rv64gc -mabi=lp64d -c` · 2026-09-17 · Source: `bench/results/addresses-riscv64.json`, code hash `6ea4cbd2185fdb5b`.

A test in this repository asserts that the two are identical instruction for instruction, so if a compiler ever stops agreeing, the claim fails rather than the sentence quietly becoming false.

An array is not a pointer, but it *decays* into one nearly everywhere: pass it, and what arrives is an address. That is why kernel functions take a length alongside every buffer, and why [ch13](#)'s bounds-check problem is a real bug pattern rather than an invention.

static — a word about who may know

addresses.c

```
/* Nothing outside this file can refer to this, which is what `static` means here. The compiler
 * therefore knows every call site, and is free to do what it likes with it. */
static int add_one(int value) { return value + 1; }

int sysfs_call_by_name(int value) { return add_one(value) * 2; }

int sysfs_uses_private(int value) { return add_one(value) * 2; }
```

`static` on a function means *nothing outside this file may name this*. That is a statement to the *linker* — the tool that joins compiled files into one program, met properly in [ch15](#) — and it has a consequence the compiler is quick to take advantage of: if nobody outside can call it, then the compiler knows every call site, and may do whatever it likes with them. Here is what it did:

sysfs_uses_private compiled for riscv64 at -O2 — the `xv6` target's instruction set, 3 instructions.

```
0000000000000040 <sysfs_uses_private>:
40:   addiw   a0,a0,1
42:   slliw   a0,a0,0x1
46:   ret
```

Conditions: target `xv6` · riscv64 cross compiler, any machine — nothing here was executed · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · -O2 -Wall -Wextra -march=rv64gc -mabi=lp64d -c · 2026-09-17 · Source: [bench/results/addresses-riscv64.json](#), code hash `6ea4cbd2185fdb5b`.

There is no call. The helper was *inlined* — its body copied into the caller in place of a call — and then deleted, and if you ask the object file for its symbols the helper is not among them. It has stopped existing as a separate thing.

The keyword is overloaded and the two meanings are unrelated, which is a genuine wart. On a function or a variable declared outside any function, `static` restricts *linkage* — who may refer to it. On a variable inside a function, it changes *storage duration* — the variable outlives the call and there is exactly one of it, for ever, shared by every caller. A kernel uses both constantly and means something different each time.

Calling through a variable

The last function. Instead of naming the one it calls, it takes it as an argument:

addresses.c

```
int sysfs_call_through(int (*operation)(int), int value) { return operation(value) * 2; }
```

sysfs_call_through compiled for riscv64 at -O2 — the `xv6` target's instruction set, 9 instructions.

```
0000000000000048 <sysfs_call_through>:
48:   addi     sp,sp,-16
4a:   sd       ra,8(sp)
4c:   mv       a5,a0
4e:   mv       a0,a1
50:   jalr     a5
52:   ld       ra,8(sp)
54:   slliw    a0,a0,0x1
58:   addi     sp,sp,16
5a:   ret
```

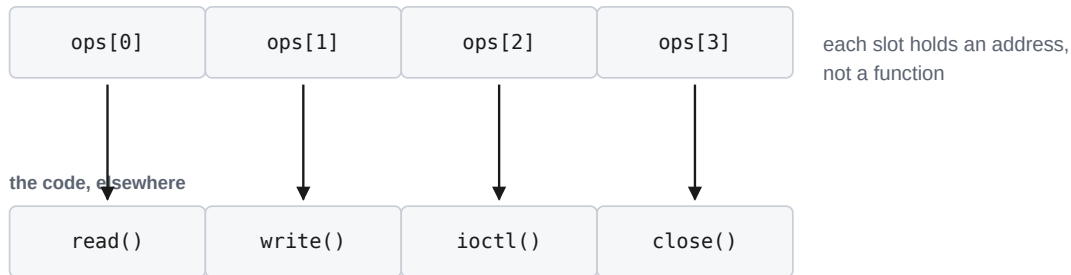
Conditions: target `xv6` · riscv64 cross compiler, any machine — nothing here was executed · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · -O2 -Wall -Wextra -march=rv64gc -mabi=lp64d -c · 2026-09-17 · Source: [bench/results/addresses-riscv64.json](#), code hash `6ea4cbd2185fdb5b`.

The jump goes to a register. The target is not in the instruction; it was loaded, and the CPU cannot know where this is going until that load has completed. That is the difference between a direct and an indirect call, and it is the mechanism behind the pattern every operating system uses to talk to hardware it has never heard of:

A table of function pointers

How a kernel does something different per device without knowing the devices.

the table, in memory



ops[n] (arg) 1. load the address out of slot n 2. jump to whatever that was

A direct call names its target inside the instruction; the CPU knows where it is going before it fetches the operand. An indirect call does not, and the CPU chapter measures what the branch predictor makes of that difference.

A table of function pointers. The kernel indexes it; the device supplies it.

A filesystem, a device driver and a network protocol all want the kernel to call *their* code when something happens. None of them can be named in the kernel's source, because they may be loaded later. So the kernel declares the shape of the table and each of them fills one in, which is how one `read()` reaches a disk, a socket and a console.

[ch19](#) uses this for real. [ch27](#) comes back to it with a cost attached, because a branch predictor faced with an indirect call has a much harder problem than one faced with a direct one.

Where it lives, and how long it stays

The three storage durations are worth knowing by their consequences rather than their names.

Automatic — an ordinary local. It lives in the current call's region of memory — its *stack frame*, which [ch14](#) shows you — and it ceases to exist the moment the function returns. A pointer to it, returned, is a pointer into space that the next call is about to use for something else.

Static — a variable outside any function, or a local marked `static`. It exists for the whole run of the program, there is exactly one, and *everybody shares it*. This is the one that produces bugs that work perfectly until the second caller arrives, which is what this chapter's second problem is.

Allocated — from `malloc`, or in a kernel from whatever the kernel has instead. It lives until somebody says otherwise, and deciding who that somebody is has consumed more engineering time than almost any other question this chapter raises.

xv6's kernel uses all three within a few hundred lines of each other, and reading it is much easier once you are asking *which of these three is this* rather than *what does this pointer mean*.

What we measured

Nothing here is a cost. Every listing above is what one compiler emitted for one file, captured by the machinery [ch12](#) explains and regenerated by CI on each push.

Each one turns an assertion into something you can check: that a plain read may be removed and a `volatile` one may not; that an array parameter and a pointer parameter produce identical code; that a `static` helper can vanish entirely; and that calling through a variable produces a different instruction from calling by name.

Those are facts about what the compiler *did*, and this chapter makes two other kinds of statement as well. Claims about what C *permits* — that `volatile` forbids removing an access, that an array parameter is adjusted to a pointer — are claims about the standard ISO/IEC JTC1/SC22/WG14 (2018) and would remain true on a compiler that produced entirely different output. Claims about what any of it *costs* are not made at all.

What this cannot tell you

Whether an indirect call is expensive. It is a different instruction, and this chapter shows that much. Whether it costs more depends on whether the predictor can guess the target, which depends on how many different functions actually flow through that call site at run time. [ch27](#) measures it; a chapter with no clock in it cannot.

Whether `volatile` is what you want. This chapter shows what it does — no access removed, no access reordered with respect to another volatile access. It says nothing about whether that is sufficient for your problem, and for anything involving two harts it is not. [ch20](#) is where that gets settled, and the answer involves instructions this chapter has not mentioned.

What the optimiser will do with `static` next time. The helper above was inlined because this compiler judged it worthwhile. A larger one would not be, and the same source would then contain a real call. “Static functions are free” is not a rule; “the compiler has more freedom with static functions” is.

How to write C. Deliberately. This chapter covers what you need to *read* a kernel and stops there — no style guidance, no idiom catalogue, and nothing about the parts of the language xv6 does not use.

Problems

Three, and all three are about the loop this chapter is really teaching: change the source, look at the output.

5.1 — Which C produced this? Four listings and four candidate functions, matched up. The listings are not stored in the repository: the test compiles each candidate with your toolchain and hands you what came out, so the puzzle is generated rather than transcribed. Solving it by eye is possible. Solving it by compiling each candidate and comparing is the point.

```
python3 -m pytest tests/c_for_people_who_will_read_a_kernel/test_problem_1_which_c.py
```

5.2 — A function that works until it is called twice.

`tests/c_for_people_who_will_read_a_kernel/scratchpad.c` turns a number into text and is correct in isolation. Called twice before either answer is used, it is not. Fix it so both answers survive, changing anything you like about the function and nothing about `main`.

The C library did exactly this for years, and some of it still does — which is why several standard functions have an `_r` on the end.

```
python3 -m pytest tests/c_for_people_who_will_read_a_kernel/test_problem_2_scratchpad.py
```

5.3 — Build the table. Four device operations and four functions that implement them. Put each function in the right slot; the test builds a dispatch table from your answer, calls through it, and checks the results. Read what the functions in `tests/c_for_people_who_will_read_a_kernel/devices.c` *do* rather than what they are called, because one of the names is a trap.

Getting the declaration of a table of function pointers right is most of the exercise, and it reads badly the first several times. That is not you.

```
python3 -m pytest tests/c_for_people_who_will_read_a_kernel/test_problem_3_dispatch.py
```

Where to go next

The C standard ISO/IEC JTC1/SC22/WG14 (2018) is the authority on which of this chapter’s claims are about the language rather than about a compiler — in particular §6.7.6.3 on the adjustment of array parameters, and §6.7.3 on what `volatile` requires.

xv6’s own source *COX ET AL.* (N.D.) is now worth opening, and this is the chapter that makes it possible. Its own commentary *COX ET AL.* (N.D.), written by the people who wrote the kernel and given away by MIT, is the best explanation of what that code does; [Appendix G](#) maps its topics onto this book’s chapters, and traces one system call through every layer both of them describe. Start with `kernel/uart.c`, which is short and contains a `volatile` device register, a static buffer with exactly the sharing problem this chapter’s second problem describes, and a lock. You will not understand the lock yet. Read it anyway and come back after [ch20](#).

[Part II](#) is next, and it leaves the kernel behind: six chapters on a machine with nothing on it, each building one mechanism — a trap, an interrupt, a page table — from the hardware alone, before any kernel is read. Reading C as machine code resumes in [ch14](#), which stops reading C and starts reading what it became: registers, the calling convention, and a stack frame you can walk by hand.

Part II — The machine with nothing on it

Part II · The machine with nothing on it · 06 · A Trap, With Nothing Else in the Machine · 07 · Interrupts and Privilege Levels · 08 · One Page Table, Two Harts · 09 · A System Call of Your Own · 10 · A Small Integer That Means a Device · 11 · fork, Built Rather Than Read

Part II · The machine with nothing on it

PART HEADER

Chapters	ch06–ch11
Target	bare — the same machine under QEMU with no operating system on it
Assumes	Part I

What this part is for

Six chapters, each building one primitive of the machine with nothing else in the way.

A kernel presents its primitives already entangled, and that is the usual reason people stall on them. The first trap you meet in xv6 arrives with a process table, a page table, a scheduler and a lock attached to it, and the code that handles it is correct about all of them at once. You are asked to learn what a trap is from a page that assumes you already know.

So: no kernel. No library, no loader, no operating system, nothing at the far end of a `printf`. The machine comes out of reset and runs your instructions. Each chapter then adds exactly one mechanism to a machine that has none — a trap, an interrupt, a page table, a system call, a descriptor table with `read` and `write` over it, and finally a second process made from the first.

The six mechanisms do not all come from the same place, and that split shapes the part. [ch06–ch09](#) are things the *hardware* hands you: a trap vector, a saved program counter and the instruction that returns from a trap are in the *privileged* specification — the half of the RISC-V manual that says what a kernel may do and a program may not — and not in anybody’s kernel. [ch10](#) and [ch11](#) are not. A descriptor table, and a second process made from the first, are inventions of software — no hardware has heard of either — and that is exactly why they are built here instead of read about later. A descriptor turns out to be an index into an array, and `fork()` turns out to be a copy; neither needs an operating system to exist, and finding that out on a machine with no operating system on it is the shortest route to believing it.

So the part sits before [Part IV](#) rather than inside it for two reasons, not one. The hardware’s mechanisms were never the kernel’s to begin with. The software’s are, and meeting them without a kernel is how you learn there is nothing magic in them. Having written the trap handler yourself and watched it work changes what xv6’s trap handling looks like: an arrangement of things you have built, rather than a wall of new ideas.

What it leaves out

Everything that makes a kernel a kernel. No scheduling policy, no file system, no device beyond the one serial port needed to see anything at all, no allocator beyond what a page table requires.

What a descriptor finds. There is no disk here and nothing to open, so the table in [ch10](#) holds two things only — the serial port, and a byte array with a cursor — which is enough to show that the code calling `read` does not change when what it reaches does, and not enough to be a file system. The several kinds of open file that make a table of them worth keeping are [ch22](#).

A pipe, and the reason is worth more than the pipe would be. A pipe is not a buffer; it is a buffer plus what happens when the buffer is empty. The reader blocks, something else runs, and somebody wakes them — and this machine’s entire scheduler is *when a process leaves, put the next one on*. It moves one way and never comes back, so a process here cannot wait for another and then continue. A shared array with a cursor is all this machine could support, which is the easy half of a pipe and teaches the wrong thing by leaving out the half that defines it. Blocking needs a scheduler that can switch both ways, which is [ch21](#), and sleeping and waking are settled there.

One omission is deliberate and needs saying plainly: **you will use a linker script — a file telling the linker where in memory each piece of the program goes — and read assembly here, and neither is explained until [Part III](#)**. Treat them as recipes. [ch14](#) covers the instructions and [ch15](#) covers the script. This part needs them working rather than understood, and the alternative ordering — linkers before traps — puts three chapters of file format between you and the first interesting thing the machine does.

Where to start

[ch06](#), in order, and this is the one part of the book with no routing in it. Each chapter’s machine is the previous chapter’s machine plus one mechanism, and each program is the previous program extended. Skipping ahead means reading code that assumes work you have not done.

Which machine, and what it cannot tell you

`qemu-system-riscv64` with no firmware and no kernel: your program is the first thing the processor executes. That is the right instrument for this part, because it is faithful about *semantics*: a trap either lands where the vector register says it does or it does not, and QEMU is exact about that.

Nothing here is timed, and nothing in it may be, for the reason [Part I](#) gives. There is a second limit as well, particular to running with no operating system: QEMU is not silicon. A real board has firmware that ran before your code ever started, errata, and a reset sequence more complicated than the idealised one here. This part teaches the architecture, not any specific chip, and a program from it will not boot a physical board unchanged.

Where this leaves you

Holding a small machine you built yourself, and the vocabulary [Part IV](#) uses without introducing.

06 · A Trap, With Nothing Else in the Machine

CHAPTER HEADER

Target	bare — the same machine under QEMU with no operating system on it
Prerequisites	Part II, ch05
What it measures	The whole of a trap in one program: where the handler was, where the interrupted instruction was, and that execution resumed after it.

The question

What is a trap, when nothing else is going on?

A trap is a jump the processor makes without being asked, to an address it was given in advance. [Part II](#) said why a kernel is a hard place to learn that: almost nothing on the page is the mechanism. Here there is nothing else — no kernel, no library, no loader, and no operating system. One handler, one deliberate trap, and every question about it answered by the machine.

The material

The whole machine

Three files, and between them they are the entire runtime. The linker script says where the program goes, because with `-bios none` — no firmware — QEMU releases the processor at a fixed address and expects to find instructions there.

bare.ld

```
SECTIONS
{
    . = 0x80000000;

    .text : {
        *(.text.entry)          /* first, because execution begins at the lowest address */
        *(.text .text.*)
    }
```

The entry code makes a stack and zeroes the memory C is entitled to assume is zero — every variable outside a function that was declared without a value. Nothing else will: there is no loader to set memory up, and no runtime to call before `main`.

start.S

```
_entry:
    csrr    t0, mhartid          # QEMU releases every hart here, at the same instant

    # sp = bare_stacks + (hartid + 1) * BARE_STACK_BYTES. Each hart gets its own and they
    # grow down into it. Two harts sharing one stack looks exactly like memory corruption.
    la      sp, bare_stacks
    li      t1, 4096
    addi    t2, t0, 1
    mul     t1, t1, t2
    add     sp, sp, t1
```

And the console is two device registers. Writing a byte to one sends it; the other says whether the device is ready for the next. That is the whole driver. The kernel's version in [ch19](#) is these same two registers underneath a great deal of machinery about sleeping and waking.

console.c

```
void bare_putc(char c) {
    while ((uart[UART_LSR] & UART_LSR_THRE) == 0) {
        /* The device has a byte in hand. Spin: there is nothing else to do and nobody to
        * yield to. A kernel would sleep here, which is the interrupts-and-drivers chapter's subject.
        */
    }
    uart[UART_THR] = (uint8)c;
}
```

Where the processor goes

One register decides where a trap lands: `mtvec`, the machine trap vector, holds the address to go to. It is one of the processor's *control and status registers* — CSRs, a set apart from `a0` and its neighbours, read and written with instructions of their own, which is what `bare_csr_write` wraps. The `m` is for *machine mode*, the most privileged state the processor has and the one it comes out of reset in; [ch07](#) is where the other modes appear.

trap.c

```
bare_csr_write(mtvec, (uint64)handler);
```

Build it and boot it on a machine with nothing on it:

```
./run trap
```

That is the installation, complete. There is no table of handlers, no registration, and nothing to tell the processor which kinds of trap this handler wants — it gets all of them. Sorting out which is which is the handler's job, and it reads `mcause` to do it — [ch07](#)'s subject.

Where it was, and the mistake that loops for ever

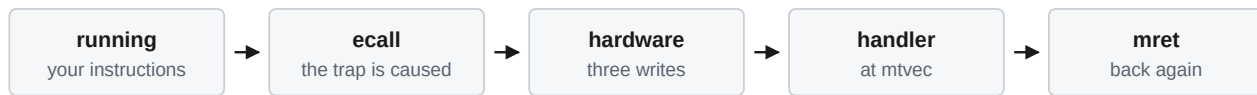
The processor also records where it was when the trap happened, in `mepc` — the saved *program counter*, the processor's record of which instruction it is on. Exactly which instruction that means is the first thing about traps that surprises people:

trap.c

```
/* mepc holds the address *of* the instruction that trapped, not the one after it. Returning
 * without moving it re-executes the ecall, which traps again, for ever. The machine does not
 * consider this an error; it is doing exactly what it was told. */
bare_csr_write(mepc, seen_epc + 4);
```

One trap, and the three registers that change

`bare` target — nothing else is running



The hardware does exactly this, and stops:

<code>mepc = address of the ecall</code>	the instruction, not the one after
<code>mcause = 11</code>	environment call from machine mode
<code>mstatus: MPIE = MIE, MIE = 0</code>	so a second trap cannot arrive yet
<code>pc = mtvec</code>	and that is all

And what it does not do:

registers	untouched — the handler's problem
stack	untouched — there is no frame

The handler must add 4 to `mepc` before returning: `mepc` is the address of the instruction that trapped, so `mret` without it re-executes the `ecall`, for ever.

Three registers and the program counter. Everything a reader associates with a trap — a saved frame, a process, a dispatch — is software somebody wrote, and none of it is here yet.

`mepc` points **at** the instruction that trapped, not past it. A handler that returns without moving it re-executes the same instruction, which traps again, into the same handler, for ever. The machine does not consider this an error. It is doing exactly what it was told, and the symptom is a program that prints nothing and never finishes.

Whether four is the right number to add is not obvious either. [ch02](#) pointed out that instructions on this architecture are not all the same length, so adding a fixed four ought to look reckless. It is safe here only because the programs in this part are built for `rv64g`, without the compressed extension — the short forms [ch02](#) mentioned — so every instruction in them really is four bytes. That is a property of the build, not of the processor, and the first problem below is about what happens when it stops holding.

What a trap does not do

It does not save anything. Not the registers, not the stack, not a frame — the processor changes `mepc`, `mcause` and a couple of bits of `mstatus`, its status register, then jumps, and that is all. Every register still holds what the interrupted code put there, and the instant the handler uses one, that value is gone.

Here the compiler covers for us:

trap.c

```
/* `interrupt("machine")` makes the compiler do two things a normal function does not: save every
 * register it touches, including the caller-saved ones an ordinary function may clobber, and end
 * with `mret` rather than `ret`. That is the whole difference between a function and a handler,
 * and the bare-metal system-call chapter is where you write it out by hand, because a handler that
 * has to read the caller's registers and change one needs them somewhere it can reach. */
__attribute__((interrupt("machine"), aligned(4))) static void handler(void) {
```

What the compiler saves is a private matter between the handler's first instructions and its last — the `mret` that jumps back to `mepc`: the C in between cannot name those slots, cannot read what the interrupted code had in `a0`, and cannot change what it will get back. That is fine here, because this handler has nothing to say to the code it interrupted. A trap that is *told* something and has to *answer* — a system call, which takes its arguments from the caller's registers and puts its result back into one — needs the saved registers laid out where the handler can reach them, and that is a frame written by hand. It is [ch09](#), and it is why that handler saves all thirty-one itself.

Proving it

The program sets a register, traps — with `ecall`, the instruction whose only job is to trap — and reads the register back afterwards. `asm volatile` is how C embeds instructions of your own; the `volatile` is [ch05](#)'s, and stops the compiler removing them.

trap.c

```
asm volatile("li    s2, 0x5eed\n"
             "la     %0, 1f\n"
             "l: \n"
             "ecall\n"
             "mv     %1, s2\n"
             : "=r" (ecall_at), "=r" (witness_after)
             : "s2", "memory");
```

What we measured

Run it yourself before reading the table — the rows below are what you should see, and a figure you have reproduced is worth more than one you have been shown:

```
./run trap
```

One trap, start to finish	Observed
<code>mtvec</code> holds the handler's address	yes
<code>mcause</code>	11
...which is <i>environment call from machine mode</i>	yes
<code>mepc</code> is the address of the <code>ecall</code> , not after it	yes
Bytes the handler must add before returning	4
A register set before the trap, still set after it	yes
Traps taken	1

Conditions: target *bare* · *qemu virt* (`-machine virt -bios none`) · *QEMU emulator version 8.2.2* (*Debian 1:8.2.2+ds-0ubuntu1.18*) · *no firmware, no kernel* · *riscv64-linux-gnu-gcc* · `-march=rv64g -mabi=lp64 -mmodel=medany -ffreestanding -nostdlib -fno-pic -fno-stack-protector -fno-builtin -O1 -g -Wall -Werror` · 2026-

09-17 · Source: `bench/results/trap-bare.json`, code hash `4d71347a1a9cb4fd`.

Every row is the machine answering rather than the chapter asserting, and `bench/run_bare.py` refuses the run if any of them stops being true — including the two that look like formalities: that `mtvec` reads back what was written, and that the program resumed, which it proves by reaching the line that prints it. If `mepc_is_the_ecall` ever came back *no*, the paragraph above about the infinite loop would be describing a different processor.

What this cannot tell you

How long a trap takes. Nothing on this target may be timed, for the reason [Part I](#) gives: a duration measured under QEMU is a fact about the laptop running the emulator. What a trap *costs* is [ch29](#), on hardware.

What a real board does before your code runs. QEMU with `-bios none` hands the processor over at the reset address with the machine in a defined state. [Part II](#) says what a physical board does instead, and why this program will not boot one unchanged.

Whether the compiler will keep covering for you. `interrupt("machine")` saved the registers because this handler had nothing to hand back. The moment a trap has to answer the code it interrupted, the saving is yours to write.

Problems

Three, and the second one is the one that catches people.

6.1 — Why four, and when is it not? The handler adds four to `mepc`. Say what that four actually is, and give a case in this instruction set where adding four would resume in the wrong place. The test asks for both, and checks the second against an assembled instruction rather than against a claim.

```
python3 -m pytest tests/a_trap_with_nothing_else/test_problem_1_advance.py
```

6.2 — Make it loop, then explain it. Remove the line that advances `mepc` and run the program. Record how the harness reports what happens, and say precisely which instruction executes how many times. The test checks that your description matches a run it performs itself, so “it hangs” does not pass.

```
python3 -m pytest tests/a_trap_with_nothing_else/test_problem_2_forever.py
```

6.3 — Take the compiler’s help away. Rewrite the handler without `interrupt("machine")`, as an ordinary function, and make the program still print `register_survived yes`. You will need to say what `mret` is and where it has to go. The test compiles your handler, boots it, and checks both the output and that the attribute is genuinely absent.

```
python3 -m pytest tests/a_trap_with_nothing_else/test_problem_3_by_hand.py
```

Where to go next

The privileged specification RISC-V INTERNATIONAL (N.D.) is the source for `mtvec`, `mepc`, `mcause` and `mret`, and is worth opening at the machine-mode trap chapter now rather than later: it is short, and everything in it is something this program did.

[ch07](#) adds the thing that arrives without being asked for. [ch16](#) is this mechanism again with a kernel around it, which is a good deal easier to read having built the middle of it.

07 · Interrupts and Privilege Levels

CHAPTER HEADER

Target	bare — the same machine under QEMU with no operating system on it
Prerequisites	ch06
What it measures	A timer interrupt taken with no kernel present, and an access refused because the program had dropped a privilege level.

The question

What arrives without being asked for, and what does a privilege level actually restrict?

[ch06](#)'s trap was caused. An instruction executed, and the trap was that instruction's consequence — remove the `ecall` and nothing happens. This chapter is about the other kind, an *interrupt*, which no instruction causes and which would arrive if the program were doing nothing at all. It is also about privilege levels: what “allowed” means in hardware is a two-bit field and a refusal.

The material

Something that arrives on its own

The timer is a device that counts. `mtime` goes up whether or not anybody reads it; when it reaches `mtimecmp`, the processor takes an interrupt. Both are addresses that are wires, in [ch04](#)'s sense — fixed places on this machine, in a block called the CLINT — and there is nothing else to it:

privilege.c

```
#define CLINT_BASE 0x02000000UL
#define CLINT_MTIMECMP (CLINT_BASE + 0x4000)
#define CLINT_MTIME (CLINT_BASE + 0xBFF8)
```

Arming it takes three writes — one to say *when*, one to enable this particular interrupt, and one to enable interrupts at all. The two enable bits do different jobs: the per-source bit says which interrupts a program is interested in, and the global bit is what a *critical section* — a stretch of code that must not be interrupted — turns off.

privilege.c

```
static void wait_for_the_timer(void) {
    volatile uint64 *mtime = (volatile uint64 *)CLINT_MTIME;
    volatile uint64 *mtimecmp = (volatile uint64 *)CLINT_MTIMECMP;

    bare_csr_write(mtvec, (uint64)handler);
    *mtimecmp = *mtime + 1000000;
    bare_csr_set(mie, MIE_MTIE);
    bare_csr_set(mstatus, MSTATUS_MIE);

    /* This loop asks for nothing. It reads no device, executes no `ecall`, and would run for
     * ever left alone. The interrupt is what stops it. */
    while (interrupt_cause == 0) {
        spins_before_interrupt++;
    }
}
```

The loop asks for nothing. It reads no device, executes no `ecall`, touches no memory it does not own, and left alone it would run until the machine was switched off. The interrupt stops it anyway, and unlike the previous chapter's trap, nothing in the loop asked for it.

Two kinds of cause, one register

`mcause` reports both kinds, and the top bit is which. That bit is also the reason the handler can be one function:

privilege.c

```
if (cause & CAUSE_INTERRUPT) {
    /* An interrupt's mepc is wherever the program happened to be. There is no instruction
     * that "caused" it, so unlike the bare-metal trap chapter's trap there is nothing to advance
    past. */
    interrupt_cause = cause;
    interrupt_epc = bare_csr_read(mepc);
    bare_csr_clear(mie, MIE_MTIE); /* once is the demonstration; twice is a loop */
    return;
}
```

The `mepc + 4` from [ch06](#) is gone. An interrupt's `mepc` is wherever the program happened to be — no instruction caused it, so there is nothing to advance past, and advancing would skip an instruction that had not run yet. The same register, read two different ways, depending on one bit.

What a privilege level refuses

Machine mode may do anything. Supervisor mode — the level below it, and the one a kernel runs in — may not, and this is the smallest demonstration of what “may not” means in hardware:

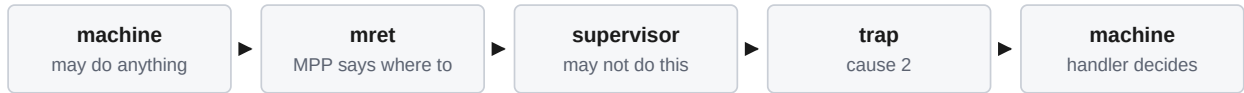
privilege.c

```
/* Supervisor mode tries to read a machine-mode register. Naked because it is not a function the
 * C world calls or returns from – it is entered by `mret` and left by trapping. */
__attribute__((naked, aligned(4))) static void supervisor_probe(void) {
    asm volatile("csrr t0, mhartid\n" /* 0xF14: machine mode only. This is the refusal. */
                "j      .\n");      /* never reached, and says so if it ever is */
}
```

One instruction, reading one register. In machine mode it succeeds; in supervisor mode the processor refuses, and the refusal is itself a trap — cause 2, illegal instruction. There is no error code, no signal and no return value. The instruction simply does not happen and control leaves for the handler.

Leaving machine mode, and the only way back

There is no instruction for gaining privilege. A trap is the entire mechanism.



<code>mstatus.MPP = 1</code>	supervisor is where mret will land
<code>mepc = &supervisor_probe</code>	and this is where it starts
<code>mret</code>	the same instruction that returns from a trap
<code>csrr t0, mhartid</code>	a machine-mode register, refused
<code>mcause = 2</code>	illegal instruction — the refusal is a trap
<code>mstatus.MPP = 3, mret</code>	the handler chooses to come back up

mret restores no registers. Machine-mode code resuming here has the supervisor caller's sp and ra, and must put its own back before it returns anywhere.

Down a level on purpose, and back only by trapping. There is no instruction for *gaining* privilege, which is why the way back has to be arranged before leaving.

Getting into supervisor mode is the same instruction that returns from a trap. `mret` goes to `mepc` at the privilege level named in `mstatus.MPP` — the field that records which level the last trap came from, and so which level `mret` goes back to — so setting that field to *supervisor* and executing `mret` is a deliberate demotion:

privilege.c

```
static void be_refused(void) {
    bare_open_memory();
    phase = PHASE_SUPERVISOR;
    bare_csr_write(mepc, (uint64)supervisor_probe);
    bare_csr_clear(mstatus, MSTATUS_MPP);
    bare_csr_set(mstatus, MSTATUS_MPP_S);

    /* `la` before `mret`, in one asm block, so the label is the instruction the handler sends us
     * back to. Split across two statements the compiler is entitled to put anything in between. */
    bare_enter_supervisor();
}
```

`bare_open_memory()` opens the *physical memory protection* regions — a small table in the processor saying which addresses the lower modes may touch — and leaving it out will cost you an afternoon. With no firmware in front of the program every one of those regions starts closed, and closed means closed to supervisor and user mode — machine mode is exempt. Without it, the supervisor program faults on its first instruction, with a cause that has nothing to do with what it was trying to do.

And getting back

Only through a trap. The handler notices which phase the program is in, puts *machine* in `MPP`, and points `mepc` at the address the program recorded before it left:

```
if (phase == PHASE_SUPERVISOR) {
    refusal_cause = cause;
    refused = 1;
    /* Go back to machine mode, at the instruction after the `mret` that left it. Changing
     * MPP is how a handler decides which mode it is returning *to* – the same field that
     * dropped us into supervisor mode brings us back out. */
    bare_csr_clear(mstatus, MSTATUS_MPP);
    bare_csr_set(mstatus, MSTATUS_MPP_M);
    bare_csr_write(mepc, bare_resume_at);
    phase = PHASE_BACK;
    return;
}
```

`mret` restores no registers at all, so anything the program wants back after the round trip has to be saved and reloaded by hand. `bare_enter_supervisor()` does that in one place, and [ch11](#) is where it stops being an inconvenience and becomes the subject.

What we measured

Run it yourself before reading the table — the rows below are what you should see, and a figure you have reproduced is worth more than one you have been shown:

```
./run privilege
```

An interrupt nobody asked for, and a refusal	Observed
The timer interrupted a loop that asked for nothing	yes
<code>mcause</code> 's top bit is set: asynchronous	yes
Cause code (machine timer)	7
Supervisor mode reading a machine register	yes
...refused with cause	2
...which is <i>illegal instruction</i>	yes
The handler returned us to machine mode	yes

Conditions: target `bare` · `qemu virt (-machine virt -bios none)` · QEMU emulator version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.18) · no firmware, no kernel · `riscv64-linux-gnu-gcc` · `-march=rv64g -mabi=lp64 -mcmodel=medany -ffreestanding -nostdlib -fno-pic -fno-stack-protector -fno-builtin -O1 -g -Wall -Werror` · 2026-09-17 · Source: `bench/results/privilege-bare.json`, code hash `832b1b25cf466972`.

What this cannot tell you

What an interrupt costs, or how long it takes to arrive. Not measurable here and not measurable under any emulator: interrupt latency is a property of a pipeline, and QEMU has none. [ch29](#) asks that question on hardware.

How real interrupt sources behave. One timer is the simplest possible case: one source, no routing, no priority, no sharing. A real machine has an interrupt controller deciding which device may interrupt which core. That controller is exactly the part this chapter leaves out, and [ch19](#) is where it arrives.

What the other privilege level does. There is a user mode below supervisor, and this chapter never enters it. Nothing here needed it, and a demonstration with a level that made no difference would have taught that levels make no difference.

Problems

Three, in `tests/interrupts_and_privilege/`.

7.1 — Interrupt an interrupt. Arrange for the timer to fire while the handler is still running, and say what happens and why. Then make it happen. The test checks both your prediction and a run. The default answer is *nothing*, for a reason in one bit of `mstatus`.

```
python3 -m pytest tests/interrupts_and_privilege/test_problem_1_nested.py
```

7.2 — Advance `mepc` on an interrupt. Make the handler treat the interrupt as [ch06](#) treated the trap, and add four. Say exactly what goes wrong and produce a run where the damage is visible in the output rather than inferred.

```
python3 -m pytest tests/interrupts_and_privilege/test_problem_2_wrong_epc.py
```

7.3 — Find another refusal. `csrr t0, mhartid` is one instruction supervisor mode may not execute. Find a second, of a different kind — not another machine-mode register — and demonstrate it, reporting its cause. The test checks the cause differs from 2 and that your program really did run in supervisor mode.

```
python3 -m pytest tests/interrupts_and_privilege/test_problem_3_another_refusal.py
```

Where to go next

The privileged specification RISC-V INTERNATIONAL (N.D.) defines the cause codes, `mstatus.MPP` and the interrupt enable bits. Its table of causes is one page and worth reading in full once: most of the entries are things you can arrange to see from a program this size.

[ch08](#) adds the other thing that arrives without being asked for, which is a second processor.

08 · One Page Table, Two Harts

CHAPTER HEADER

Target	bare — the same machine under QEMU with no operating system on it
Prerequisites	ch07
What it measures	One mapping installed by hand and an address that means something else afterwards; and a counter two harts disagree about.

The question

What does address translation do, and what does a second core break?

Two questions, and two programs, because they are two mechanisms and a chapter that ran them together would teach neither. Both are the hardware changing what memory does behind the program's back: translation does it deliberately and usefully; a second core does it simply because there is more than one processor.

The material

An address is a number until something translates it

Everything so far has used addresses that were what they said. `&x` was where `x` actually was; the UART (the console's serial device) was at the number the machine's documentation gives. *Translation* — the hardware turning every address a program uses into the one memory actually sees — makes that stop being true, and the whole of the mechanism is a table the program writes and a register that points at it.

Sv39, RISC-V's translation scheme for 64-bit machines RISC-V INTERNATIONAL (N.D.), splits a thirty-nine-bit address into three nine-bit indexes and a twelve-bit offset, which means three levels of table. This program needs none of that, because an entry at the top level can be a *leaf* — one that maps memory directly instead of pointing at the next table down — covering a whole gigabyte at once:

paging.c

```
/* A leaf entry for a whole gigabyte. The physical page number is the address shifted right by
 * twelve; the entry holds it shifted left by ten, which is the ten bits of flags underneath. */
static uint64 gigapage(uint64 physical) {
    return ((physical >> 12) << 10) | PTE_LEAF;
}
```

The entire table is three entries:

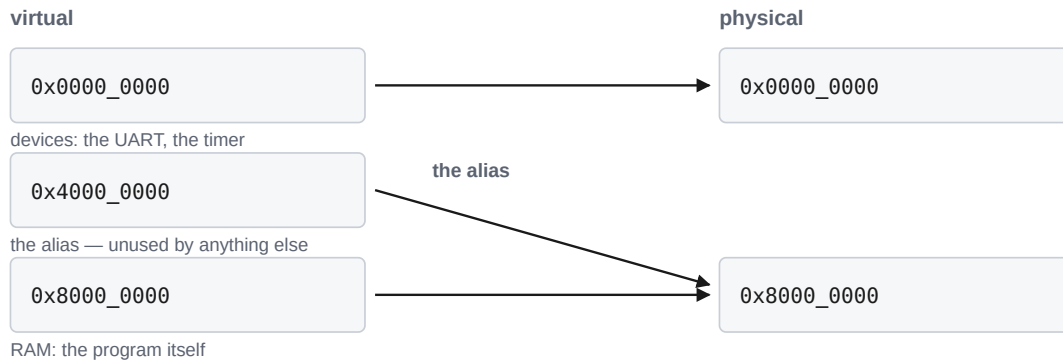
paging.c

```
static void build_the_table(void) {
    root[DEVICES_BASE / GIGABYTE] = gigapage(DEVICES_BASE);
    root[RAM_BASE / GIGABYTE] = gigapage(RAM_BASE);
    root[ALIAS_BASE / GIGABYTE] = gigapage(RAM_BASE); /* the alias: a second name for RAM */
    entries_used = 3;
}
```

Two of them map memory to itself — an *identity mapping*. They have to: the moment translation is switched on, the instruction after the switch is fetched through the table, and a table without the code in it would make the program vanish. The third points a different *virtual* address — the kind a program uses — at the same *physical* memory — the kind the chips have — so that afterwards two addresses a gigabyte apart name one byte.

One table, three entries, and an address with two names

Each entry is a leaf covering a gigabyte, so there is no second level to walk.



The two arrows into RAM are the point: addresses 1 GiB apart read the same byte. Machine mode ignores satp entirely, so none of this applies until the program leaves it.

Two entries send a gigabyte to itself, so the running program does not vanish the moment translation comes on. The third is the whole demonstration.

Switching it on is one register — `satp`, which points at the table — and one instruction:

paging.c

```
bare_csr_write(satp, SATP_SV39 | ((uint64)root >> 12));  
asm volatile("sfence.vma zero, zero"); /* the processor may have cached the old answer */
```

`sfence.vma` is there because the processor is allowed to remember translations it has already worked out, and is not obliged to notice that the table changed underneath it. Forgetting it gives a program that works until it does not, which is among the least pleasant kinds of bug.

And then the part that catches everyone: **machine mode ignores `satp` entirely**. Translation applies to supervisor and user mode; machine mode addresses are physical, always. So the program has to leave machine mode before any of this means anything, which is why [ch07](#) came first.

The demonstration itself is then one variable read twice, by its own address and through the alias, before an `ecall` asks machine mode to take the program back:

paging.c

```
/* Runs with translation on. Everything it touches is either identity-mapped or the alias. */
__attribute__((noinline)) static void in_supervisor_mode(void) {
    uint64 offset = (uint64)&marker - RAM_BASE;
    volatile uint64 *through_alias = (volatile uint64 *) (ALIAS_BASE + offset);

    reached_supervisor = 1;
    marker_direct = marker;
    marker_through_alias = *through_alias;

    asm volatile("ecall"); /* ask machine mode to take us back */
}
```

What a second core breaks

The second program has two processors in it, and one counter.

The usual way to show what goes wrong is to start both harts incrementing as fast as they can and observe that the total comes out short. That works, sometimes, and teaches the wrong thing when it does: it makes lost updates look like weather. The useful claim is narrower: `counter = counter + 1` is a load, an add and a store, and *anything at all* happening between the load and the store is enough.

So this loses an update on purpose, identically every run, by holding those three instructions apart and letting the other hart finish entirely in the gap. The two flags are the handshake that holds it open:

harts.c

```
/* The three instructions of `counter = counter + 1`, with the other hart invited into the
 * gap between the load and the store. */
stale = counter; /* 1. load: reads 0 */
hart1_may_go = 1; /* ... and here the other hart does its entire increment ... */
while (!hart1_finished) {
}
counter = stale + 1; /* 2. add, 3. store: writes 1 over the other hart's 1 */
```

The other hart's whole increment happens inside one statement of this one. Both processors incremented; the counter went up once.

Then the same shape with an *atomic* instruction — one that cannot be interrupted part-way through, because it is one instruction. `__sync_fetch_and_add` asks the compiler for it, and on this machine it is `amoadd`, a load, an add and a store in a single step:

harts.c

```
/* The same shape, with an instruction that has no gap in it. */
__sync_fetch_and_add(&atomic_counter, 1);
hart1_may_go_atomic = 1;
while (!hart1_finished_atomic) {
}
```

Nothing is lost, and nothing about the schedule changed. What changed is that there is no longer a moment at which the counter has been read but not yet written.

THIS PROVES ATOMICITY, AND ATOMICITY IS NOT ORDERING

The instruction fixed one thing: the read and the write of *this* counter cannot be separated. It says nothing about whether the other hart sees this hart's earlier writes to anything else, or in what order, and a program built on the assumption that it does will be wrong on real hardware in ways this target cannot show you. [ch20](#) is where the difference gets its own chapter, and [ch28](#) is where it costs something.

What we measured

Run them yourself before reading the tables — the rows below are what you should see, and a figure you have reproduced is worth more than one you have been shown:

```
./run paging
./run harts
```

One page table of three entries	Observed
Entries in the table	3
Translation is on, and the program still runs	yes
Two addresses, one byte	yes
Gigabytes between those two addresses	1
Machine mode read it untranslated	yes

Conditions: target `bare` · qemu virt (-machine virt -bios none) · QEMU emulator version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.18) · no firmware, no kernel · riscv64-linux-gnu-gcc · -march=rv64g -mabi=lp64 -mmodel=medany -ffreestanding -nostdlib -fno-pic -fno-stack-protector -fno-builtin -O1 -g -Wall -Werror · 2026-09-17 · Source: `bench/results/paging-bare.json`, code hash `ef29e43c6d7be324`.

What a second core costs an unguarded counter	Observed
Increments performed	2
What the counter holds	1
Updates lost	1
Increments performed, atomically	2
What that counter holds	2
Updates lost	0

Conditions: target `bare` · qemu virt (-machine virt -bios none) · QEMU emulator version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.18) · no firmware, no kernel · riscv64-linux-gnu-gcc · -march=rv64g -mabi=lp64 -mmodel=medany -ffreestanding -nostdlib -fno-pic -fno-stack-protector -fno-builtin -O1 -g -Wall -Werror · 2026-09-17 · Source: `bench/results/harts-bare.json`, code hash `0c2a2353f264220a`.

The runner refuses a run in which `updates_lost` is anything but exactly one, because if the handshake ever stopped holding the read-modify-write open, the program would still boot, still print, and still look convincing, while showing nothing at all.

What this cannot tell you

How expensive any of this is. A page-table walk costs something, a translation cache exists to avoid paying it, and two cores touching one *cache line* — the unit in which memory moves between them — cost each other a great deal. None of that is visible here. [ch25](#) and [ch28](#) are those questions on hardware, and the second one is where the atomic instruction above stops looking free.

What actually happens when two real cores race. The lost update here was arranged. On real hardware the ordering is decided by machinery this target has none of — [ch28](#) names it — and the answer is neither deterministic nor as simple as one update going missing. What transfers from this chapter is the shape of the hazard, not its likelihood.

Whether one atomic is enough. It was here, for one counter. It is not a general answer: an operation that cannot be split is not the same as a sequence of them that cannot be reordered, and the second is what a lock needs.

Problems

Three, in `tests/one_page_table_two_harts/`.

8.1 — Map a page rather than a gigabyte. Change the alias to cover four kilobytes instead of a gigabyte. You will need the two levels this chapter skipped. The test checks that the alias still reads the same byte *and* that the entry at the top level is no longer a leaf.

```
python3 -m pytest tests/one_page_table_two_harts/test_problem_1_four_kilobytes.py
```

8.2 — Break it on purpose. Remove one of the two identity mappings and predict, before running it, exactly which instruction the machine fails on and with what cause. Then run it. The test compares your prediction with the run.

```
python3 -m pytest tests/one_page_table_two_harts/test_problem_2_no_identity.py
```

8.3 — Lose an update the other way round. The program loses the second hart's increment. Rearrange the handshake so it loses the first hart's instead, without changing the counter's final value. Say which of the three instructions moved. The test checks the value, the loss, and that you did not simply swap the two harts' code.

```
python3 -m pytest tests/one_page_table_two_harts/test_problem_3_other_way.py
```

Where to go next

The privileged specification RISC-V INTERNATIONAL (N.D.) defines Sv39, the page-table entry format and `satp`; the unprivileged specification RISC-V INTERNATIONAL (N.D.) defines the atomic instruction the second program leans on. Both are worth having open — the entry format in particular is one diagram and this chapter built it by hand.

[ch09](#) turns the boundary this chapter crossed into an interface.

09 · A System Call of Your Own

CHAPTER HEADER

Target	bare — the same machine under QEMU with no operating system on it
Prerequisites	ch08
What it measures	A call number, arguments and a return value crossing the boundary, and the count of registers a handler saves when the saved registers are its interface.

The question

What has to exist before `ecall` is a system call rather than a trap?

[ch06](#) already executed an `ecall` and handled it. Nothing about that was a system call: no request was made, nothing was asked for, and the handler had nothing to decide. Four things are missing, and this chapter adds them — a number saying which call, somewhere to put arguments, somewhere to put a result, and a dispatch. A fifth is missing too, and it is the expensive one: the arguments and the result live in the caller's registers, so the handler has to save every register somewhere it can read and write them.

The material

The saved registers are the interface

[ch06](#)'s handler could let the compiler save the registers because it never needed to look at them. This one does: the call number and the arguments arrive in the caller's registers, and the result has to go back in one, so the saved registers have to be somewhere the handler's C can read and write. That is a frame the handler lays out itself.

THIS IS NOT THE USER-TO-KERNEL BOUNDARY YET

A system call on a real machine goes from *user* mode to *supervisor* mode. This one goes from supervisor to machine, which is one level up from where you are expecting it.

The mechanism is identical either way: `ecall` traps to the level above wherever it was executed, and that level decides what to do about it. Going supervisor-to-machine avoids having to build a user mode, a process and an address space before the boundary itself can be shown, which would put this chapter after [ch11](#) rather than before it. [ch16](#) is the same four ingredients at the level a reader expects.

The frame holds every register the caller had, not the three the interface needs, because a frame that holds every register is a complete description of the caller — the thing [ch11](#) copies to make a second process from the first — where a frame that holds three is not:

syscall.c

```
/* The trap entry. Naked because every instruction in it matters and a prologue the compiler
 * chose would run before the caller's registers had been saved.
 *
 * This is the file's whole point, so it is spelled out rather than generated: thirty-one stores
 * on the way in and thirty-one loads on the way out, and the only reason it is not thirty-two is
 * that x0 cannot hold anything to lose. */
__attribute__((naked, aligned(4))) static void syscall_entry(void) {
    asm volatile(
        ".macro SAVE reg, slot\n"
        "    sd \\reg, \\slot*8(sp)\n"
        ".endm\n"
        ".macro LOAD reg, slot\n"
        "    ld \\reg, \\slot*8(sp)\n"
        ".endm\n"
```

Thirty-one, not thirty-two, because `x0` is hard-wired to zero and has nothing to lose. It is written out rather than generated so that its length is visible: this is the cost of describing the caller completely, and it is paid on every single call.

The frame also becomes the interface. Once the registers are in memory, “the arguments” and “the registers” are the same thing, and the handler reads them as an array:

syscall.c

```
void syscall_dispatch(uint64 *frame) {
    uint64 cause = bare_csr_read(mcause);
    uint64 number = frame[17]; /* a7, by convention – the convention is this line */
    uint64 first = frame[10]; /* a0 */
    uint64 second = frame[11]; /* a1 */
```

`a7` holds the number and `a0` the first argument because those two lines say so, and nothing else enforces it. A convention is an agreement, and here you are both parties.

The dispatch

syscall.c

```
switch (number) {
case SYS_ADD:
    frame[10] = first + second;
    break;
case SYS_WITNESS:
    frame[10] = frame[18]; /* s2, read out of the saved frame rather than out of the CPU */
    break;
```

Writing to `frame[10]` is how a result gets back: the epilogue restores `a0` from that slot, so the caller finds it in the register the convention promised. Nothing returns anything in the C sense.

The default case decides what a call number nobody implemented does:

syscall.c

```
default:
    /* A number nobody implemented is a refusal, not a crash. Returning an error is what
     * makes this an interface rather than a trapdoor. */
    unknown_calls_refused++;
    saw_unknown_number = (number == 99);
    frame[10] = SYS_UNKNOWN_RESULT;
```

A number nobody implemented gets an error, not a stop. That is the difference between an interface and a trapdoor, and it is a decision — the hardware would have been equally happy to let the machine wander off.

Calling it

From the other side, a system call is four register moves and an instruction:

syscall.c

```
static uint64 call(uint64 number, uint64 first, uint64 second) {
    register uint64 a0 asm("a0") = first;
    register uint64 a1 asm("a1") = second;
    register uint64 a7 asm("a7") = number;
    asm volatile("ecall" : "+r"(a0) : "r"(a1), "r"(a7) : "memory");
    return a0;
}
```

The `"+r"(a0)` tells the compiler that `a0` is both an input and an output, which is exactly the claim the convention makes. Get that wrong and the compiler will cheerfully assume the register still holds what it put there.

What we measured

Run it yourself before reading the table — the rows below are what you should see, and a figure you have reproduced is worth more than one you have been shown:

```
./run syscall
```

What crosses the boundary	Observed
Registers the handler saves by hand	31
Calls dispatched	4
A call number chose what happened	yes
3 and 4 went in; this came back	7
The caller's register, read out of the saved frame	yes
...and unchanged when the caller resumed	yes
Calls refused as unknown	1
...by returning an error, not by stopping	yes

Conditions: target `bare` · qemu virt (`-machine virt -bios none`) · QEMU emulator version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.18) · no firmware, no kernel · riscv64-linux-gnu-gcc · `-march=rv64g -mabi=lp64 -mmodel=medany -ffreestanding -nostdlib -fno-pic -fno-stack-protector -fno-builtin -O1 -g -Wall -Werror` · 2026-09-17 · Source: `bench/results/syscall-bare.json`, code hash `acb9c60789cf0db5`.

Two of those rows are the same fact from opposite sides. `caller_register_in_frame` is the handler finding the caller's `s2` in the saved frame; `caller_register_intact` is the caller finding it still there afterwards. A handler that saved nothing would fail the second and never reach the first.

What this cannot tell you

What a system call costs. Thirty-one stores and thirty-one loads is a count, not a price, and the price is not thirty-one times anything. [ch29](#) measures a real one on real hardware, and the number is larger than this chapter would lead you to expect — the register saves are not where the time goes.

Whether thirty-one is the right number. It is the *safe* number, and a real kernel saves fewer by making the convention promise more. Where that is possible and what it buys is a question this target cannot answer, because the answer is entirely about cost.

What happens when the caller lies. Every argument here is a number the handler uses as a number. The moment an argument is a pointer, the handler is dereferencing an address chosen by untrusted code, and everything about that is [ch16](#)'s problem — [ch11](#) runs into the first half of it.

Problems

Three, in `tests/a_system_call_of_your_own/`.

9.1 — Add a call, and an error. Add a call that can fail for a reason other than being unknown, and return something the caller can distinguish from a valid result. Say why your choice of sentinel is safe. The test exercises both the success and the failure and checks the two cannot be confused.

```
python3 -m pytest tests/a_system_call_of_your_own/test_problem_1_add_a_call.py
```

9.2 — Save one register fewer. Remove exactly one store and its matching load from the entry stub, and produce a program in which that omission is visible in the output. The test checks that the register you dropped is the one your program shows being corrupted.

```
python3 -m pytest tests/a_system_call_of_your_own/test_problem_2_one_fewer.py
```

9.3 — Where does the frame live? The stub puts the frame on whatever stack the caller was using. Say what goes wrong if the caller arrives with a stack pointer it does not own, and change the stub so the kernel uses a stack of its own instead. The test checks the frame is no longer on the caller's stack.

```
python3 -m pytest tests/a_system_call_of_your_own/test_problem_3_whose_stack.py
```

Where to go next

The unprivileged specification RISC-V INTERNATIONAL (N.D.) defines `ecall`; the privileged one RISC-V INTERNATIONAL (N.D.) defines what it does at each privilege level, which is not the same thing at each.

[ch10](#) gives this mechanism its first API worth calling.

10 · A Small Integer That Means a Device

CHAPTER HEADER

Target	bare — the same machine under QEMU with no operating system on it
Prerequisites	ch09
What it measures	One <code>write</code> call reaching two unrelated destinations through one table, and the table itself printed before and after a descriptor is duplicated.

The question

Why does a program name what it reads by number, and what does that number find?

If you arrived from a language with a runtime, you have a picture of a file as an object: a thing with methods, which knows how to read and write itself, and which you hold a reference to. The C and Unix answer is stranger and much smaller. You hold an integer. The integer is not the file, does not point at the file, and means nothing outside the program holding it.

This is where [Part I](#)'s claim — that everything is an index — arrives one level up.

The material

Two tables, and the difference between them

A *descriptor* — the integer — is an index into a table the kernel keeps for each process; here, with no kernel, the machine-mode half of the program keeps it. What the table holds is not the file either: it holds *which open file*, and the open file is a separate thing in a separate table.

descriptors.c

```
/* Two tables, and the distinction between them is the whole subject.
 *
 * `descriptors` is indexed by the small integer and holds nothing but a number: which open file
 * this one refers to. `open_files` holds what is actually open, including how far through it we
 * are. Two descriptors can name one open file, and then they share a cursor — which is why a
 * shell can point a program's output somewhere and have it append rather than overwrite.
 *
 * Putting the cursor in the descriptor instead would be simpler and would quietly make `dup` a
 * copy rather than a second name. It is the same distinction the fork chapter's `fork` turns on: the
 * table
 * is copied, the things it refers to are not. */
#define OPEN_FILES 4
#define NO_FILE (-1)

static int descriptors[SLOTS];
static struct open_file open_files[OPEN_FILES];
static char scratch[SCRATCH_BYTES];

/* The length of what is in `scratch`, as opposed to where any descriptor currently is. */
static uint64 written_bytes;
```

Collapsing those two into one would be simpler and is the obvious first design. It is also wrong: two descriptors can refer to one open file and share its position, which the last section of this chapter shows.

Two things worth having a table for

With one possible destination, an indirection demonstrates nothing: every descriptor would find the same thing and the table would be decoration. So there are two *backends* — things a descriptor can find. One is a real device at a physical address; the other is an array of bytes with a cursor, which is a few lines long and is not a file system and is not pretending to be one.

descriptors.c

```
/* The two things a descriptor can find. A third would need nothing but another case here and
 * another `kind`, which is the point: the table is the interface. */
enum backend { BACKEND_NONE, BACKEND_CONSOLE, BACKEND_MEMORY };
```

The dispatch is a `switch`, and a third kind of backend would be one more case. The caller never learns which backend it got.

The same call, twice

descriptors.c

```
/* 1. The same call, twice, differing in one register — and reaching a device the first time
 * and an array in memory the second. Neither the calling code nor the compiler knows. */
wrote_to_console = bare_call(SYS_WRITE, 1, (uint64)message, length);
wrote_to_memory = bare_call(SYS_WRITE, 2, (uint64)message, length);
same_call_reached_both = (wrote_to_console == length) && (wrote_to_memory == length);
```

Two calls, identical but for one number, and they end up in completely unrelated places — one in a device register, one in an array in memory. Neither the calling code nor the compiler knows the difference, which is how a shell can point a program's output at a file without the program being told.

Run it and watch:

```
./run descriptors
```

The `hello` on the first line of output is the console write arriving. Everything after it is the program reporting on the other one.

The position is not a record of what you last did

Write six bytes into the array and the position is at six. So read them straight back:

descriptors.c

```
/* 2. Read it straight back, without moving the cursor. This is the part worth running the
 * program for: the write left the position at the end, so there is nothing after it to
 * read, and a call that looks obviously correct returns nothing at all. */
read_without_rewinding = bare_call(SYS_READ, 2, (uint64)taken, length);
```

It returns nothing. Not an error, not a short read — zero bytes, from a call that is plainly correct and an array that plainly has six bytes in it.

A descriptor's position is **where you are in the open file**, not a note about whether you were last reading or writing, and that is the one thing about descriptors that catches everybody. The write left it at the end, and the end is where the read started, and after the end there is nothing. There is no separate read position to fall back on, because there is no separate read position at all.

descriptors.c

```
/* 3. Move the position, then read. Same descriptor, same array, different answer. */
bare_call(SYS_SEEK, 2, 0, 0);
read_after_rewinding = bare_call(SYS_READ, 2, (uint64)taken, length);
```

Same descriptor, same array, same call — and six bytes this time, because the position moved first. `read` and `write` here are the same two lines apart from which direction the bytes go:

descriptors.c

```
/* Reads move the cursor exactly as writes do, because there is only one cursor and it is the
 * position *in the open file* rather than a position belonging to reading or to writing.
 *
 * That is the whole of why a program that has just written something and wants to read it back
 * has to move the cursor first. It is the commonest surprise about file descriptors and it is
 * not a quirk: the position is where you are, not what you last did. */
static uint64 read_from(int fd, char *into, uint64 count) {
    struct open_file *file = file_for(fd);
    if (!file || file->kind != BACKEND_MEMORY) {
        return BAD_DESCRIPTOR; /* the console here is write-only, and says so rather than lying */
    }
    uint64 taken = 0;
    while (taken < count && file->cursor < written_bytes) {
        into[taken++] = scratch[file->cursor++];
    }
    return taken;
}
```

What dup shares

`dup` copies a descriptor onto another number, and it is the reason the two tables had to be two:

descriptors.c

```
static uint64 duplicate(int fd, int onto) {
    if (onto < 0 || onto >= SLOTS || !file_for(fd)) {
        return BAD_DESCRIPTOR;
    }
    /* One number is copied. The open file it refers to is not, so afterwards two descriptors are
     * two names for one thing – including one cursor between them. */
    descriptors[onto] = descriptors[fd];
    return (uint64)onto;
}
```

One number is copied. The open file it refers to is not — so afterwards two descriptors are two names for one thing, and they share its position.

You can watch that happen rather than take it on trust. Six bytes have been written through descriptor 2; the program duplicates it onto 3 and writes one byte through the copy:

descriptors.c

```
/* 5. Duplicate, then write through the copy. Because the two numbers name one open file they
 * share its position, so this lands *after* what is already there rather than on top of
 * it. That is what makes redirecting a program's output an append rather than a
 * truncation, and it is the reason the cursor lives where it does. */
dup_result = bare_call(SYS_DUP, 2, 3, 0);
wrote_through_dup = bare_call(SYS_WRITE, 3, (uint64)!", 1);
cursor_after_dup_write = bare_call(SYS_TELL, 2, 0, 0);
```

The byte lands at offset six, the position ends at seven, and reading the array back gives seven bytes. Had the position lived in the descriptor instead, the duplicate would have started from zero and written its byte over the `h` — both programs would still run, and the difference would surface much later as a file mysteriously overwriting itself from the beginning.

That shared position is the whole reason a second write continues rather than overwrites. Two descriptors onto one open file share one offset, so the write picks up where the last one left off — a consequence of where the position already was, not a mode the write asks for.

What we measured

Run it yourself before reading the table — the rows below are what you should see, and a figure you have reproduced is worth more than one you have been shown:

```
./run_descriptors
```

One table, two destinations, one position	Observed
Descriptor slots	4
Open-file slots	4
Kinds of thing a descriptor can find	2
One <code>write</code> , two destinations, one register different	yes
Where the position is after writing six bytes	6
Bytes a read returns from there	0
...and after moving the position back to the start	6
The bytes read back are the bytes written	yes
An unopened descriptor is refused	yes
A duplicate is a second name, not a copy	yes
Position after one byte through the duplicate	7
...which landed after the six, not on them	yes
Bytes in the array altogether	7

Conditions: `target bare · qemu virt (-machine virt -bios none) · QEMU emulator version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.18) · no firmware, no kernel · riscv64-linux-gnu-gcc · -march=rv64g -mabi=lp64 -mcmodel=medany -ffreestanding -nostdlib -fno-pic -fno-stack-protector -fno-builtin -O1 -g -Wall -Werror · 2026-09-17 · Source: bench/results/descriptors-bare.json, code hash ca976726e794d978.`

What this cannot tell you

Anything about cost. A `write` here is a function call behind a trap. On a real system it is a trap, a permission check, a copy from the program's memory into the kernel's, and possibly a device. [ch29](#) prices the real one.

What an open file really contains. A kind and a cursor is the smallest thing that shows the sharing. A real one has a mode, a reference count, a position that several processes may be contending over, and a pointer to something that knows how to be read. [ch22](#) is that, in a kernel.

How the table gets entries. There is no `open` here, because there is nothing to open: the two backends are set up before the program starts. So the most interesting question about descriptors — how a name becomes a number — is entirely absent, and [ch22](#) answers it.

What the numbers 1 and 2 mean. They mean what the two lines in `main` say they mean. On a real system 0, 1 and 2 are a convention held up by the program that started you, not by the kernel, and the convention is worth exactly as much as everyone’s agreement to keep it.

Problems

Three, in `tests/a_small_integer_that_means_a_device/`.

10.1 — Duplicate onto a number in use. `dup` here overwrites whatever was at the target. Decide what should happen when the target is already open, implement it, and justify the choice. The test checks your behaviour is consistent and that you did not simply refuse every duplicate.

```
python3 -m pytest tests/a_small_integer_that_means_a_device/test_problem_1_onto_open.py
```

10.2 — Add a third backend. Add a destination that discards everything written to it and reads back as nothing. Show that the calling code does not change. The test checks the caller is byte-identical across all three descriptors.

```
python3 -m pytest tests/a_small_integer_that_means_a_device/test_problem_2_third_backend.py
```

10.3 — Put the cursor in the wrong place. Move the cursor into the descriptor table and produce a program in which that change is visible in the output. Say in one sentence which real-world behaviour would break. The test checks the shared cursor claim now fails, and that your program shows it failing rather than asserting it.

```
python3 -m pytest tests/a_small_integer_that_means_a_device/test_problem_3_wrong_place.py
```

Where to go next

There is no specification for this one: a descriptor is not a hardware concept and nothing in the RISC-V documents has heard of it. The nearest primary source is POSIX THE OPEN GROUP AND IEEE (2018) on `dup` and `write`, and it is worth reading `dup`’s wording specifically — the standard is careful about what is shared in a way that only makes sense once you have built the two tables.

[ch11](#) makes a second process, and the first question it has to answer is which of these two tables the child gets a copy of.

11 · fork, Built Rather Than Read

CHAPTER HEADER

Target	bare — the same machine under QEMU with no operating system on it
Prerequisites	ch10
What it measures	Two address spaces from one, a return value that differs between them, and the count of pages copied — beside what xv6 copies for the same call.

The question

What is the least a machine needs before two programs can run on it?

`fork` is usually introduced as a riddle: a function that returns twice, with different answers. It is not a riddle. It looks like one because it is normally met from the outside. From the inside it is a short function, and the two returns stop being mysterious once you can see that a return value is a word in a saved register set.

Everything this chapter needs already exists. [ch08](#) built an *address space* — a page table, and so a private view of memory; [ch09](#) saved a caller's registers into a frame. A process is those two things kept together, and `fork` is a copy.

The material

What a process is, here

fork.c

```
struct proc {
    uint64 frame[32]; /* the registers it had when it last left the processor */
    uint64 epc;       /* and where it was */
    uint64 *root;     /* its address space */
    uint64 *page;     /* the physical page behind USER_VA, so machine mode can look */
    int used;
};
```

A saved register set, a place it was, and an address space. That is the whole of it — and the first field is [ch09](#)'s trap frame, unchanged. The frame was built to survive a system call; keep it a little longer and it is a saved process.

Mapping one page rather than one gigabyte

[ch08](#) got away with three top-level leaf entries. A per-process page needs a real walk — an entry at each of the three levels — because four kilobytes is a leaf at the bottom level, and the levels above it have to exist first:

fork.c

```
/* Walk to the leaf entry for one virtual address, making the levels that are missing. Three
 * levels, nine bits each, because Sv39 is thirty-nine bits of address and the bottom twelve are
 * the offset within a page. The bare-metal paging chapter used gigabyte leaves and needed none of this.
 */
static uint64 *leaf_entry(uint64 *root, uint64 va) {
    uint64 *table = root;
    for (int level = 2; level > 0; level--) {
        uint64 *entry = &table[(va >> (12 + 9 * level)) & 0x1FF];
        if (*entry & PTE_V) {
            table = (uint64 *)((*entry) >> 10) << 12;
        } else {
            uint64 *next = take_page();
            *entry = (((uint64)next >> 12) << 10) | PTE_V; /* no R/W/X: a pointer, not a leaf */
            table = next;
        }
    }
    return &table[(va >> 12) & 0x1FF];
}
```

Each process gets the kernel's identity mappings — without them the code would vanish the moment its own table was installed — plus one page of its own at the same virtual address:

fork.c

```
static uint64 *build_address_space(uint64 *user_page) {
    uint64 *root = take_page();
    /* The kernel's own mappings, identity, so that the code keeps running with paging on. */
    root[DEVICES_BASE / GIGABYTE] = ((DEVICES_BASE >> 12) << 10) | PTE_LEAF;
    root[RAM_BASE / GIGABYTE] = ((RAM_BASE >> 12) << 10) | PTE_LEAF;
    *leaf_entry(root, USER_VA) = (((uint64)user_page >> 12) << 10) | PTE_DATA | PTE_U;
    return root;
}
```

Same address in both processes, different physical page behind it. That is what “its own memory” means, mechanically, and it is why an address space is worth having.

fork

fork.c

```
static int fork_current(uint64 *frame) {
    struct proc *parent = &procs[current];
    struct proc *child = &procs[current + 1];

    for (int i = 0; i < 32; i++) {
        child->frame[i] = frame[i];
    }
    /* The one edit that makes the answer differ. Everything else about the two is identical. */
    child->frame[10] = 0;
    child->epc = bare_csr_read(mepc) + 4;

    child->page = take_page();
    for (uint64 i = 0; i < BARE_PAGE_BYTES / sizeof(uint64); i++) {
        child->page[i] = parent->page[i];
    }
    pages_copied++;

    child->root = build_address_space(child->page);
    child->used = 1;
    children_made++;
    return current + 1;
}
```

The frame is copied wholesale and then one slot is changed, and that one line is why `fork` returns twice. Slot ten is `a0`, where a system call's return value lives, and the child resumes by having its frame loaded back into the registers. Two frames, identical but for one word, produce two returns with different answers.

The page is copied too — a fresh page and a byte-for-byte duplicate, so the child begins with everything the parent had and can then diverge. This is the eager version, and it is what xv6's `uvmcopy` does too: a fresh page and a copy, per page, at the moment of the call. [ch18](#) is where that stops being necessary.

Putting a process back on the processor

`syscalls.c`

```
/* Put a saved process back on the processor: its registers, and where it was.
 *
 * `sp` is loaded last because every other load is relative to it, and a frame is not somewhere
 * you can still address once you have replaced the pointer to it. */
__attribute__((naked)) void bare_enter_process(uint64 *frame, uint64 epc) {
    asm volatile("csrw mepc, a1\n"
        "mv    sp, a0\n"
        "ld x1,1*8(sp)\n ld x3,3*8(sp)\n ld x4,4*8(sp)\n ld x5,5*8(sp)\n"
        "ld x6,6*8(sp)\n ld x7,7*8(sp)\n ld x8,8*8(sp)\n ld x9,9*8(sp)\n"
        "ld x10,10*8(sp)\n ld x11,11*8(sp)\n ld x12,12*8(sp)\n ld x13,13*8(sp)\n"
        "ld x14,14*8(sp)\n ld x15,15*8(sp)\n ld x16,16*8(sp)\n ld x17,17*8(sp)\n"
        "ld x18,18*8(sp)\n ld x19,19*8(sp)\n ld x20,20*8(sp)\n ld x21,21*8(sp)\n"
        "ld x22,22*8(sp)\n ld x23,23*8(sp)\n ld x24,24*8(sp)\n ld x25,25*8(sp)\n"
        "ld x26,26*8(sp)\n ld x27,27*8(sp)\n ld x28,28*8(sp)\n ld x29,29*8(sp)\n"
        "ld x30,30*8(sp)\n ld x31,31*8(sp)\n"
        "ld x2,2*8(sp)\n"
        "mret\n");
}
```

`sp` is loaded last because every other load is relative to it, and a frame stops being addressable the instant you replace the pointer to it.

The scheduler is four lines, and calling it a scheduler is generous:

`fork.c`

```
void bare_process_left(void) {
    if (current + 1 < PROCESSES && procs[current + 1].used) {
        current++;
        bare_csr_write(satp, SATP_SV39 | ((uint64)procs[current].root >> 12));
        asm volatile("sfence.vma zero, zero");
        bare_enter_process(procs[current].frame, procs[current].epc);
    }
}
```

Install the next process's address space, load its registers, `mret`. There is no policy, no priority and no preemption — nothing takes the processor away from a process that has not given it up. [ch21](#) is where the interesting parts go back in.

The kernel cannot simply dereference what it is given

One detail here is a genuine trap, and it cost a confusing hour:

fork.c

```
/* The handler runs in machine mode, and machine mode ignores `satp`. So it cannot simply
 * dereference the address its caller gave it: `USER_VA` here would be a *physical* address,
 * and on this board there is a PCI window at that number which answers every read with ones.
 *
 * What the kernel has to do instead is translate for itself – here trivially, because it
 * remembers which physical page it gave each process. A real kernel walks the caller's page
 * table to do the same thing, and `copyin` is what that function is called. */
case SYS_PEEK:
    return procs[current].page[0];
case SYS_POKE:
    procs[current].page[0] = frame[10];
    return 0;
```

The handler runs in machine mode, and [ch08](#) established that machine mode ignores `satp`. So an address the caller supplies is not an address the handler can use — and on this machine the particular number involved lands in a region QEMU reserves for PCI devices, which answers every read with ones, so the failure is not even a fault. It is a plausible-looking wrong answer.

A real kernel walks the caller’s page table to translate by hand. That function is called `copyin`, and this is why it exists.

What we measured

Run it yourself before reading the table — the rows below are what you should see, and a figure you have reproduced is worth more than one you have been shown:

./run fork

Two processes from one	Observed
Processes	2
Address spaces	2
Pages copied	1
What <code>fork</code> returned to the parent	1
What it returned to the child	0
Both processes ran	yes
The child began with the parent's data	yes
...and its writes stayed its own	yes

Conditions: `target bare · qemu virt (-machine virt -bios none) · QEMU emulator version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.18) · no firmware, no kernel · riscv64-linux-gnu-gcc · -march=rv64g -mabi=lp64 -mmodel=medany -ffreestanding -nostdlib -fno-pic -fno-stack-protector -fno-builtin -O1 -g -Wall -Werror · 2026-09-17 · Source: bench/results/fork-bare.json, code hash 5b436c26f6ab9ccd`.

One page copied, for one page mapped. xv6 copies every mapped page of the parent the same way — `kalloc` and `memmove` per page, no sharing — so the count here is small for the same reason the program is small, not because the strategy differs.

What this cannot tell you

What `fork` costs. Copying a page has a price, copying a thousand has a much larger one, and the entire reason real kernels stopped doing this eagerly is a number this target cannot produce. [ch29](#) measures a real `fork`, and the gap between an eager copy and what Linux actually does is most of the answer.

What the child should inherit. The child here gets the address space and nothing else, because there is nothing else. A real `fork` has to decide about descriptors, the working directory, signal handlers, resource limits and more. [ch10](#)'s two tables answer the descriptor half: the descriptor table is copied, the open files it refers to are shared.

Whether two processes really run concurrently. They do not. One runs, gives up, and the other starts. Nothing here preempts anything, and both are on one hart.

What happens when the table is full. There is room for two, `fork` is called once, and the question of what a process table does when it runs out is left entirely alone. xv6's `allocproc` returns zero; what the caller should do about it is a design question this program never has to face.

Problems

Three, in `tests/fork_built_rather_than_read/`.

11.1 — Make the child run first. The parent continues and the child waits. Swap it, so the child runs to completion before the parent resumes, without changing what either prints. The test checks the order and that both still report correctly.

```
python3 -m pytest tests/fork_built_rather_than_read/test_problem_1_child_first.py
```

11.2 — Run out of processes. Call `fork` until the table is full. Decide what it returns then, implement it, and show the caller handling it. The test checks the failure is distinguishable from a successful `fork` and that the machine survives it.

```
python3 -m pytest tests/fork_built_rather_than_read/test_problem_2_full_table.py
```

11.3 — Count what a lazy fork would save. Do not implement copy-on-write — sharing the parent's pages and copying one only when somebody writes to it. Instead, say exactly which of this program's steps it would remove, what it would add, and what new trap the handler would have to deal with. The test asks for the cause code of that trap and for which page-table bit changes, both of which you have met.

```
python3 -m pytest tests/fork_built_rather_than_read/test_problem_3_lazy.py
```

Where to go next

The privileged specification RISC-V INTERNATIONAL (N.D.) has the page-table entry bits this program sets by hand, including the one problem 11.3 asks about.

This is the end of [Part II](#). [Part III](#) steps back to ask what a program is before [Part IV](#) reads a kernel — and that kernel's `fork` will look like a longer version of this one rather than a new idea, which was the point of building it.

Part III — What a computer does with a program

Part III · What a computer does with a program · 12 · What a Computer Does With a Program · 13 · Representing Information · 14 · Machine-Level Code on RISC-V · 15 · Linking and Loading

Part III · What a computer does with a program

PART HEADER

Chapters	ch12–ch15
Target	xv6 , with ch12 crossing to <code>host</code> and saying which figure came from where
Assumes	Part I

What this part is for

One program, followed the whole way: from the text you typed to instructions a processor is executing, with nothing in between left as magic.

[Part II](#) put your code at the address the machine starts from, which is the only situation in which a program simply is somewhere. Every other program gets there by a route, and the route turns out to be four separate programs, several file formats and a system call. This part walks it. On the way it has to answer what a number in memory actually is, because a program is data before it is instructions, and because the bugs that survive longest come from a value meaning something other than what its name suggests.

It sits before [Part IV](#) because a kernel is a program. You cannot usefully read one until you know what a program is, what a compiler leaves unfinished, and who finishes it. [ch15](#) ends at `exec` , the system call that puts a program into a process, which is precisely where [Part IV](#) begins.

What it leaves out

How a compiler works. Parsing, intermediate representations, register allocation as a subject in its own right — none of it is here. This part asks only what the compiler *produced* and whether you can account for it. Why it chose one thing over another, and whether a better choice existed, is [ch26](#), and it is in [Part V](#) because the answer is about cost.

Also left out: every object format and loader other than the one in front of you. What generalises is that a linker joins fragments, that some of a program's bytes are not stored because they are known to be zero, and that something has to place the result in memory before it can run. The spelling is local; the structure is not.

Where to start

[ch12](#), in order. This part is a single argument in four steps, and each chapter finishes a question the one before it left open: what the compiler produced and what it left unfinished; what the bytes in it mean; what the instructions in it do; and who fills in the unfinished parts and puts the result in memory.

If you already know ELF — the file format the rest of the part takes apart — and want the rest, [ch13](#) stands alone reasonably well; it is about representation rather than tooling, and it is the chapter most likely to change what you think you know.

Which machine, and what it cannot tell you

Mostly `xv6` : compiled with the RISC-V cross-compiler, run under QEMU, disassembled and read. [ch12](#) compiles the same source on both targets, because the claim it makes — that the stages are a property of the toolchain and not of the machine — only holds up if you watch it happen twice.

Nothing in this part is timed. Every figure here is structural: what a file contains, which instructions the compiler emitted, what a program printed. That costs the chapters nothing: a stage boundary is a fact about the toolchain and survives being observed under emulation, whereas a duration does not survive it at all.

Where this leaves you

Able to take a program you did not write, find out what it is made of, and read the part of it the processor actually sees.

12 · What a Computer Does With a Program

CHAPTER HEADER

Target	xv6 and host — every example says which
Prerequisites	ch00
What it measures	What each toolchain stage produced, and what linking added: <code>bench/results/stagewalk-riscv64.json</code> . Nothing here is timed on either target.
What it captures	What the compiler emits for the same loop twice: <code>bench/results/stages-riscv64.json</code>

The question

What actually happens between a source file and a result, and which of it costs anything?

Most of it costs nothing. A great deal of what looks like work in a C program has been finished before the machine is switched on, and the part that remains is smaller and stranger than the source suggests. Separating the two is the skill the rest of [Part III](#) is built on, because you cannot ask what a program costs until you know which parts of it still exist at the time it runs.

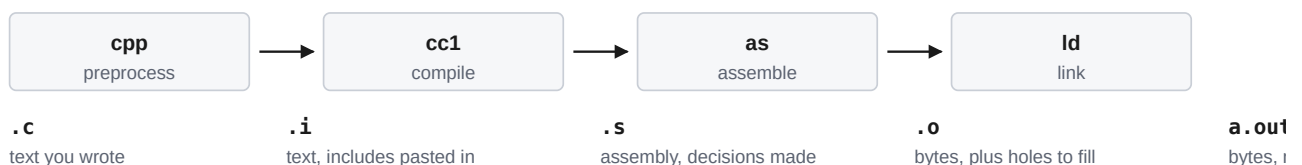
The material

One command is four programs

`gcc sameanswer.c -o sameanswer` looks like one operation. `gcc` is a launcher that runs four programs in turn, each of which leaves a file on disk that you are allowed to look at and normally never do.

One command, four programs

``gcc a.c -o a`` runs all of these. Each one hands the next a file you may look at.



and what it throws away

X comments, macro names **X** types, names, most of the header **X** mnemonics, whitespace **X** nothing — it adds

Only the second box makes decisions. The first is text substitution, the third is a lookup table, and the fourth resolves addresses — none of them can change your loop.

Four programs, four handovers. Only one of them makes decisions.

Here is the walk through them, as a shell script rather than a paragraph, so you can run it yourself:

```
# 1. Preprocess. Text in, text out: includes pasted in, macros expanded, comments deleted.
#   No C has been understood yet — cpp does not know what a function is.
$cc $flags -E "$source" -o "$outdir/$stem.i"

# 2. Compile. The only stage that makes decisions. C in, assembly out.
$cc $flags -S "$outdir/$stem.i" -o "$outdir/$stem.s"

# 3. Assemble. Assembly in, object out: instructions become bytes, and anything whose address is
#   not yet known becomes a relocation for the linker to fill in.
$cc $flags -c "$outdir/$stem.s" -o "$outdir/$stem.o"

# 4. Link. Objects and libraries in, one program out. -static so the result runs under user-mode
#   QEMU, which has no RISC-V loader to offer it.
$cc $flags -static "$outdir/$stem.o" -o "$outdir/$stem"
```

Four commands differing only in where `gcc` is told to stop. Run it on the program this chapter uses:

```
sysfs/tools/stages.sh sysfs/tools/sameanswer.c /tmp/walk
ls -l /tmp/walk
```

What each stage is allowed to know

The stages differ in what each one is allowed to know, and that difference explains most of what each one does.

The preprocessor does text substitution and nothing else. It has never heard of a function, a type, or a loop.

`#include` means *paste that file in here*; a macro means *replace this name with that text*. What comes out — one source file with everything it included pasted in — is what the compiler actually sees, and is called a *translation unit*. When the preprocessor finishes, every name it introduced is gone and no later stage can tell that a macro was ever involved. This is why a macro is not a variable and why a bug in one is reported at a line that does not look wrong.

The compiler is the only stage that decides anything. It reads C and writes assembly, and between those two it is free to do whatever it can prove does not change the result. Everything this book will later say about optimisation happens here.

The assembler does a lookup. Mnemonics become bytes. It knows the addresses of things inside the file it is reading, and nothing about anything outside it — so where your code refers to something it cannot see, it writes a hole and a note describing what belongs in it. That note is a **relocation** THE SANTA CRUZ OPERATION (N.D.).

The linker fills the holes. It gathers objects and libraries, decides where everything will live, and resolves the notes. It is the first stage that knows a real address.

The same loop, twice

Here is the whole of the program's arithmetic. Two functions, close enough that you could reasonably call them the same function:

stages.h

```
#define SYSFS_STAGES_DEFINE_SUMS \
    unsigned long sysfs_sum_folded(void) { \
        unsigned long total = 0; \
        for (unsigned n = 0; n < SYSFS_STAGES_SPAN; n++) \
            total += n; \
        return total; \
    } \
\
    unsigned long sysfs_sum_counted(unsigned span) { \
        unsigned long total = 0; \
        for (unsigned n = 0; n < span; n++) \
            total += n; \
        return total; \
    }
```

Same loop, same accumulator, same arithmetic, same optimisation level. The only difference is where the bound comes from: one is a number the compiler can read, the other arrives in a register while the program is running.

This is what the compiler did with the first one:

sysfs_sum_folded compiled for riscv64 at -O2 — the `xv6` target's instruction set, 2 instructions.

```
0000000000000000 <sysfs_sum_folded>:
0:   li    a0,2016
4:   ret
```

Conditions: target `xv6` · riscv64 cross compiler, any machine — nothing here was executed · `riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · -O2 -Wall -Wextra -march=rv64gc -mabi=lp64d -c · 2026-09-17 · Source: bench/results/stages-riscv64.json, code hash 44c82e8a3b25ae99.`

There is no loop. There is no addition. The compiler ran the loop itself, at compile time, and wrote down what it got — the entire function is a constant and a return. Whatever that sum cost, it was not paid by the machine that runs this program.

And the second:

sysfs_sum_counted compiled for riscv64 at -O2 — the `xv6` target's instruction set, 11 instructions.

```
0000000000000006 <sysfs_sum_counted>:
6:   beqz   a0,1c <.L6>
8:   slli   a4,a0,0x20
c:   srli   a4,a4,0x20
e:   li     a5,0
10:  li     a0,0

0000000000000012 <.L5>:
12:  add     a0,a0,a5
14:  addi    a5,a5,1
16:  bne     a5,a4,12 <.L5>
1a:  ret

000000000000001c <.L6>:
1c:  li     a0,0
1e:  ret
```

Conditions: target `xv6` · riscv64 cross compiler, any machine — nothing here was executed · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · `-O2 -Wall -Wextra -march=rv64gc -mabi=lp64d -c` · 2026-09-17 · Source: `bench/results/stages-riscv64.json`, code hash `44c82e8a3b25ae99`.

That is a loop: a counter, an accumulation, and a branch backwards. It also has a case for zero and a pair of shifts at the top, which are there because `span` is a 32-bit unsigned living in a 64-bit register and the compiler has to say which half of it means anything. That is [ch13](#)'s subject, arriving early.

Nothing about the source said one of these was expensive. The difference is entirely about what the compiler could *prove*. It may replace a loop with its answer only when it can establish that the loop terminates, that the bound is known, and that the result cannot depend on anything it cannot see. Take any one of those away and the loop comes back.

This book returns to that pattern constantly: **the cost of a construct is not a property of the construct**. It depends on what the compiler was able to determine about its surroundings, which is why the answer to “is this fast?” is so often “show me the rest of the file”.

What the object file still owes

Compile the program but do not link it, and it is incomplete in a specific and visible way:

```
riscv64-linux-gnu-nm -u /tmp/walk/sameanswer.o
```

`-u` lists undefined symbols — names the object uses and cannot supply. The program calls `printf`, which lives somewhere else entirely, so the assembler left a hole. After linking there are none: every name has been resolved to an address, which is most of what “linking” means.

What we measured

Nothing here is a timing. These are file sizes and symbol counts — facts about what a compiler produced, which is why they are stamped as artefacts rather than measurements and why CI regenerates them on every push rather than trusting the committed copy.

What each stage handed to the next:

Stage	Bytes it produced	Lines
<code>preprocess</code>	38189	1176
<code>compile</code>	1756	108
<code>assemble</code>	4440	10
<code>link</code>	556304	3500

Conditions: target `xv6` · riscv64 cross compiler, any machine — nothing here was executed · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · `-O2 -Wall -Wextra -march=rv64gc -mabi=lp64d` · 2026-09-17 · Source: `bench/results/stagewalk-riscv64.json`, code hash `a709eb7d4f82e541`.

Read the two columns against each other, because bytes and lines do not move together.

The preprocessed file is enormous compared with the source, and almost none of it is yours: one `#include <stdio.h>` drags in every declaration the C library wants you to have. The compiler then throws nearly all of it away — the assembly it produces is a small fraction of the text it read — because a declaration you never used generates no code. That collapse shows what a declaration is: a promise about what exists, not an instruction to build it.

Then the file gets *larger* in bytes while getting much smaller in lines, which is what happens when text becomes a container format. The object file is ELF: headers, a symbol table, relocations and a section of instructions, and the instructions are the smallest part.

And then linking:

What	Count
Object file, before linking	4440
Symbols the object left undefined	2
Linked against glibc, statically	556304
Linked by xv6, against its own user library	35424
Symbols the program leaves undefined	0

Conditions: target `xv6` · riscv64 cross compiler, any machine — nothing here was executed · `riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · -O2 -Wall -Wextra -march=rv64gc -mabi=lp64d · 2026-09-17 · Source: bench/results/stagewalk-riscv64.json, code hash a709eb7d4f82e541.`

The two linked sizes are the same program. One is linked statically — every library routine it needs copied into the file — against the GNU C library; the other is linked by xv6 against its own user library, which is a few hundred lines and knows how to do almost nothing. The difference between them is not your program — your program is identical — it is the cost of everything `printf` is prepared to do if asked. [ch15](#) opens that binary up and says where it all went.

The same program in both worlds

Build it for each target and run it:

```
make xv6-qemu          # then, at the xv6 prompt: sameanswer
python3 -m pytest tests/test_stages.py -q
```

Both print the same six lines. The folded route and the counted route agree, as they must — they compute the same sum, and the compiler is not permitted to change the answer, only the work.

What the two targets will not tell you is which route was faster. QEMU has no opinion about time worth listening to, and the board is not attached to this chapter. [ch23](#) puts that question properly, and [ch26](#) answers it.

What this cannot tell you

Nothing here is a cost. The sizes above are how much space a compiler used, and space is not time: a smaller binary is not automatically a faster one, and a function that compiles to two instructions is not automatically cheaper than one that compiles to ten. [ch27](#) shows a case where more instructions run faster, for reasons entirely outside the instruction count.

It is one compiler, at one optimisation level. Everything in this chapter is what `gcc` at `-O2` decided, and the conditions line under each table says which `gcc`. A different version may fold a loop this one leaves alone. That is why the listings are regenerated by CI instead of transcribed: if the compiler changes its mind the build fails, rather than the book quietly becoming wrong.

“The compiler folded it” is a description, not a mechanism. This chapter shows you that it happened and states the conditions under which it is allowed to. It does not show you how the decision is made, and this book never will — that is a compiler course, and an honest systems book is better off teaching you to *look at the output* than to model the optimiser.

One `#include` is not a fair account of the C library. The preprocessed size above measures `<stdio.h>` and its dependencies, not “the cost of `libc`”. A program that includes more headers preprocesses to more text and may still link to the same binary.

And that size briefly depended on where the repository sat on disk. The preprocessor writes into its output the name of every file it pasted in, spelled exactly as the command line spelled it. So an absolute include path makes the preprocessed file longer on a machine whose checkout sits deeper, and the same commit measured larger on a CI runner than on a laptop — the whole difference being the lines the preprocessor writes to say which file each piece came from, naming one header. Nothing about the number could have given that away; a size is a size. It surfaced only because CI regenerates this result instead of trusting the committed copy, and the two disagreed. `stages.sh` now names every path relative to the repository root, and the runner refuses to stamp a result if any file the walk produced mentions where the repository lives. It shows exactly what stage 1 does: it pastes text, and the paths are part of the text.

Problems

Three. Each has a test that passes only when you have solved it, and none of them has an answer stored anywhere in this repository.

12.1 — Which stage produced this? `tests/what_a_computer_does_with_a_program/problem_1_stages.py` gives you fragments of the four files and asks which program wrote each one. The fragments are real. The fastest way to solve it is not to stare at them but to run `sysfs/tools/stages.sh` yourself and look at your own.

```
python3 -m pytest tests/what_a_computer_does_with_a_program/test_problem_1_stages.py
```

12.2 — Predict the ripple. Four edits to the program. For each one, say which stages produce a different file afterwards.

This test stores no expected answers. It makes each change, runs the toolchain, and compares the bytes — so you are not being marked against an answer key, you are being marked against a compiler. When you disagree with it, it is right, and the interesting question is what you believed that was not true.

```
python3 -m pytest tests/what_a_computer_does_with_a_program/test_problem_2_ripple.py
```

12.3 — Take the folding away. Write a function that computes the same sum and that the compiler *cannot* evaluate for you. It must still contain a loop, and the machine must actually run it — the test disassembles what came out and looks for a backward branch, so returning a constant will not pass and neither will a loop the optimiser can see through.

```
python3 -m pytest tests/what_a_computer_does_with_a_program/test_problem_3_unfold.py
```

There is more than one way to do it and they are not equally good. The section on the same loop twice says what the compiler has to be able to prove; take away whichever of those you like.

Where to go next

The ELF specification [THE SANTA CRUZ OPERATION](#) (N.D.) defines the object format the assembler writes and the linker reads: sections, symbols and relocations, all of which [ch15](#) takes apart properly. It is worth skimming now purely to see that the thing `nm` printed is a documented structure rather than a convention.

The RISC-V psABI [RISC-V INTERNATIONAL](#) (N.D.) says which relocation types exist and what each one means, and is the document that explains why a call to `printf` looks the way it does in the object file.

[ch13](#) goes back to the thing the compiler was manipulating in the first place: what a number is to this machine, and when that answer bites.

13 · Representing Information

CHAPTER HEADER

Target	xv6 — the teaching kernel under QEMU
Prerequisites	ch12
What it measures	Type sizes, alignments and struct layouts, and what signed and unsigned arithmetic actually compile to: bench/results/setup-xv6.json , bench/results/signedness-riscv64.json

The question

What is a number to this machine, and when does that answer bite?

Most of the time it does not bite at all, which is the difficulty. A machine word behaves enough like an integer that a programmer can go years without meeting the difference, and then meet it in production. So this chapter is organised around the places the representation *leaks* — where two things that look identical in C turn out not to be, and the reason is always that one of them made a promise the other did not.

The material

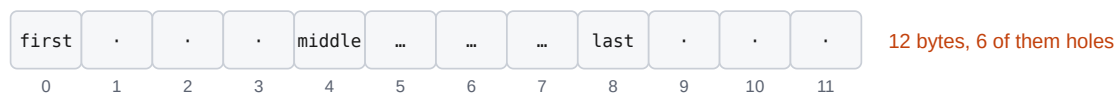
Two rules make every struct

[ch00](#) measured two structs with the same three members in different orders and found one larger than the other. A table can say that. It cannot say where the extra bytes went:

Two structs, the same three members

Declared in a different order, and one of them pays for it.

struct sysfs_declaration_order char first; int middle; char last;



struct sysfs_size_order int middle; char first; char last;



· a byte no member uses

Two rules produce every gap: a member starts at a multiple of its own alignment, and a struct is a multiple of its widest member's. The compiler may not reorder members to avoid either — C forbids it — so the order you wrote is the order you pay for.

The holes are not at the end. They are wedged between members.

Two rules produce every gap in that picture:

1. **A member starts at a multiple of its own alignment** — the spacing the hardware wants a value of that width to sit at. A four-byte integer may begin at offset zero, four, eight — not at one. Many machines load a word faster when it does not straddle two of them, and some refuse outright.
2. **A struct's size is a multiple of its own alignment**, which is the largest of its members'. Otherwise the second element of an array of them would start somewhere illegal.

And one prohibition that turns those rules into your problem rather than the compiler's: **C does not permit members to be reordered** ISO/IEC JTC1/SC22/WG14 (2018). The order you wrote is the order you get, and the padding follows from it. Every other language with a struct-like type had this same choice to make, and several of them chose differently.

On a single struct this is an oddity. On an array of a few million — a particle system, a packet buffer, a page-table cache — it is the difference between fitting in a level of cache and not, which is [ch25](#)'s subject and the first place this dry rule becomes a duration.

A comparison that is not a comparison

Here are two functions. They differ by one word.

signedness.c

```
int sysfs_signed_grows(int x) { return x + 1 > x; }

int sysfs_unsigned_grows(unsigned x) { return x + 1 > x; }
```

Both ask whether adding one to a number makes it bigger. This is what the compiler did with the signed one:

sysfs_signed_grows compiled for riscv64 at -O2 — the `xv6` target's instruction set, 2 instructions.

```
0000000000000000 <sysfs_signed_grows>:
0:   li      a0,1
2:   ret
```

Conditions: target `xv6` · riscv64 cross compiler, any machine — nothing here was executed · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · -O2 -Wall -Wextra -march=rv64gc -mabi=lp64d -c · 2026-09-17 · Source: [bench/results/signedness-riscv64.json](#), code hash `d4dace135c463c55`.

It does not add. It does not compare. It returns yes without looking at the argument, because as far as the compiler is concerned there is no argument for which the answer could be anything else.

And the unsigned one:

sysfs_unsigned_grows compiled for riscv64 at -O2 — the `xv6` target's instruction set, 3 instructions.

```
0000000000000004 <sysfs_unsigned_grows>:
4:   addi    a0,a0,1
6:   snez    a0,a0
a:   ret
```

Conditions: target `xv6` · riscv64 cross compiler, any machine — nothing here was executed · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · `-O2 -Wall -Wextra -march=rv64gc -mabi=lp64d -c` · 2026-09-17 · Source: `bench/results/signedness-riscv64.json`, code hash `d4dace135c463c55`.

That one does the work.

The difference is not that unsigned arithmetic is slower. It is that **the two types make different promises**. Unsigned overflow is defined: it wraps, and the largest unsigned value plus one really is zero, so the comparison can be false and the compiler must ask. Signed overflow is *undefined* ISO/IEC JTC1/SC22/WG14 (2018) — the standard declines to say what happens — so a compiler is entitled to assume it never occurs, and from that assumption `x + 1 > x` follows for every `x` that exists as far as it is concerned.

Undefined behaviour is not a runtime hazard; it is a licence the optimiser holds. Stay uneasy about that for the rest of the book. The danger is not that your program will crash when it overflows. The danger is that a branch you wrote to check for overflow will be deleted, because the compiler reasoned that the condition could not arise.

A division that is not a division

The same pair again, dividing instead of adding:

signedness.c

```
int sysfs_signed_quarter(int x) { return x / 4; }

unsigned sysfs_unsigned_quarter(unsigned x) { return x / 4; }
```

Unsigned first, because it is the one that behaves:

sysfs_unsigned_quarter compiled for riscv64 at `-O2` — the `xv6` target’s instruction set, 2 instructions.

```
000000000000001c <sysfs_unsigned_quarter>:
1c:  srliw  a0,a0,0x2
20:  ret
```

Conditions: target `xv6` · riscv64 cross compiler, any machine — nothing here was executed · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · `-O2 -Wall -Wextra -march=rv64gc -mabi=lp64d -c` · 2026-09-17 · Source: `bench/results/signedness-riscv64.json`, code hash `d4dace135c463c55`.

One shift. Dividing an unsigned number by a power of two is a right shift, exactly, with no edge cases.

Signed:

sysfs_signed_quarter compiled for riscv64 at `-O2` — the `xv6` target’s instruction set, 5 instructions.

```
000000000000000c <sysfs_signed_quarter>:
c:  sraiw  a5,a0,0x1f
10:  srliw  a5,a5,0x1e
14:  addw   a0,a0,a5
16:  sraiw  a0,a0,0x2
1a:  ret
```

Conditions: `target xv6 · riscv64 cross compiler, any machine — nothing here was executed · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · -O2 -Wall -Wextra -march=rv64gc -mabi=lp64d -c · 2026-09-17 ·`
Source: `bench/results/signedness-riscv64.json`, code hash `d4dace135c463c55`.

Several instructions, and the extra ones are not arithmetic — they are a correction. An arithmetic right shift rounds towards negative infinity; C requires integer division to round towards zero ISO/IEC JTC1/SC22/WG14 (2018). For positive numbers those agree. For negative ones they differ by one, so the compiler adds a bias before shifting, and the bias has to be computed from the sign bit — the top bit, which says whether the value is negative — because it only applies to negatives.

Nobody wrote that correction. It is the cost of a promise C made about rounding, paid on every signed division by a constant, by a programmer who thought they were writing a shift.

Neither of these functions is wrong, and neither compiler decision is a bug. The machine code you get is a consequence of the type you chose, and the type is a claim about what values are possible. Make a weaker claim and you get more instructions; make a stronger one and the optimiser will use it, including in ways you did not intend.

Byte order, seen rather than described

A number wider than a byte has to be laid out in memory somehow, and there are two sensible answers. Both of this book's machines choose little-endian: the lowest-addressed byte holds the least significant part. [ch00](#)'s probe asks the machine rather than asserting it, which is why the fact appears in its output and not in a footnote here.

It matters in exactly three places, and outside them you can forget it: when bytes cross a machine boundary (a file, a network, a device register), when you alias a value through a pointer of a different width, and when you are reading a memory dump by eye and the digits appear to be backwards. The first kind is the one that bites: a device register or a file written by another machine is bytes somebody else laid out, and nothing warns you when their order is not yours.

The operations worth writing once

`sysfs/lib/bits.c` holds the bit manipulations later chapters reuse. They are written out the long way rather than calling a builtin, because this is the chapter where a machine word stops being an abstraction, and a loop you wrote teaches that better than an intrinsic you called:

bits.c

```
unsigned sysfs_popcount(unsigned long word) {
    unsigned set = 0;
    while (word) {
        /* Clearing the lowest set bit each time, so this runs once per set bit rather than once per
         * bit in the word. Worth knowing as a shape: `x & (x - 1)` is the single most reused
         * bit-twiddling identity in this book. */
        word &= word - 1;
        set++;
    }
    return set;
}
```

`word &= word - 1` clears the lowest set bit. It is worth learning as a shape rather than deriving each time, and it is the single most reused identity in this book.

One function in that file is about overflow rather than bits:

```

unsigned long sysfs_round_up_pow2(unsigned long value) {
    if (value == 0) {
        return 0;
    }
    unsigned long result = 1;
    while (result < value) {
        /* The overflow check has to come before the shift, not after: once the top bit shifts out
        * there is nothing left to notice it by. Returning 0 says "no such power of two fits",
        * which a caller can test; wrapping silently to 0 would say the same thing by accident. */
        if (result > (~0UL >> 1)) {
            return 0;
        }
        result <<= 1;
    }
    return result;
}

```

The overflow check runs *before* the shift. Afterwards there is nothing left to notice it by — the bit has gone, and the value is zero, and zero is indistinguishable from a legitimate answer. That ordering is the same lesson [ch24](#) draws about checking for a condition while the evidence still exists. It arrives several chapters early because arithmetic is where it bites first.

What we measured

xv6 and host present the same C implementation — the same sizes, alignments and byte order — and [ch00](#)’s table measured it by booting the kernel and asking it. It is not repeated here, because a figure printed twice is a figure that can disagree with itself.

What this chapter adds is the machine code above. Those listings are not timings, and nothing in this chapter is. They are what one compiler emitted for one source file at one optimisation level, regenerated by CI on every push so that a compiler which changes its mind breaks the build rather than the argument.

Counting instructions is not measuring cost. A function with more instructions in it can be faster than one with fewer, and [ch27](#) shows a case where that happens for reasons entirely outside the count. The listings establish something weaker and more useful: the two functions in each pair are *not the same program*, whatever the source looked like.

What this cannot tell you

Whether any of it is slow. Signed division emits a correction; whether that correction costs a measurable amount depends on the core, on what else is in flight, and on whether the result was needed immediately. [ch26](#) is where that gets a number, on a machine that can produce one.

What another compiler does. Every listing here is one version of `gcc`. A different compiler may fold differently, and the standard permits both. The claims about *what C says* are claims about the standard ISO/IEC JTC1/SC22/WG14 (2018) and are stable; the claims about what the machine code looks like are claims about a toolchain and are stamped accordingly.

What undefined behaviour will do to you. This chapter shows one consequence — a comparison compiled away — because it is visible in a disassembly. The full set is not enumerable, and anyone who tells you the rule is “it does whatever the hardware does” has the wrong model: the hardware never sees the code that was deleted.

Floating point. It is absent from this chapter and from the xv6 target generally, and that is a decision rather than an oversight: xv6 does not save floating-point registers across a context switch, so a user program that uses them is quietly wrong the moment it is descheduled. That is a perfectly reasonable thing for a teaching kernel to decide — it makes [ch21](#)’s context switch small enough to read in one sitting — and it means floating point arrives in [Part V](#), on a machine whose kernel does save those registers.

Problems

Three, and none of their answers is anywhere in this repository.

13.1 — Two operations the library does not have. `tests/representing_information/bitops.c` asks for a byte-order swap and the index of the lowest set bit. Neither is in `sysfs/lib/bits.c`, so there is nothing to copy. The test checks *properties* rather than cases: swapping twice must give back what you started with, and the bit you name must be set with nothing set below it. A property holds for every input, so it cannot be satisfied by memorising.

```
python3 -m pytest tests/representing_information/test_problem_1_bitops.py
```

13.2 — Pack the struct. Five members, declared in an order that wastes space. Give the order that wastes least. The test compiles your arrangement and compares it against the best that arrangement of those types can do — it does not tell you the number, and the two rules at the top of this chapter are enough to work it out.

```
python3 -m pytest tests/representing_information/test_problem_2_reorder.py
```

13.3 — Find the input that makes it wrong. A bounds check that looks correct and is not. Find one triple of values where it says a read fits and the read would run off the end.

Nothing in it is undefined: the type is `unsigned`, so the machine does exactly what it was told, and the result is still wrong. That is the uncomfortable half of this chapter — undefined behaviour is dramatic and rare, and *defined* behaviour that does not match your intention is quiet and everywhere.

```
python3 -m pytest tests/representing_information/test_problem_3_break_it.py
```

Where to go next

The C standard ISO/IEC JTC1/SC22/WG14 (2018) is the document that decides which of this chapter’s surprises are the compiler’s fault and which are yours. It is not readable end to end, and it is not meant to be — but looking up one behaviour you thought you knew is a formative afternoon, and the freely available committee draft is fine for that.

The RISC-V unprivileged specification RISC-V INTERNATIONAL (N.D.) defines what the shift and division instructions above actually do, including the cases C calls undefined. Reading the two documents against each other is how you learn that “undefined behaviour” describes a *language*, not a machine: the hardware always does something specific, and knowing what does not make the C legal.

The psABI RISC-V INTERNATIONAL (N.D.) fixes the sizes and alignments this chapter’s structs are laid out by.

[ch05](#) did the same thing to the other half of C — the part that is really about addresses — and is worth rereading now that a type has been shown deciding what the machine does.

ch14 stops treating the listings as evidence for a point about C and starts reading them as the subject.

14 · Machine-Level Code on RISC-V

CHAPTER HEADER

Target	xv6 — the teaching kernel under QEMU
Prerequisites	ch05
What it measures	Frame sizes and instruction mix for four functions at -O0 and -O2: <code>bench/results/framesizes-riscv64.json</code>

The question

What did the compiler actually emit, and how do I read it?

[ch02](#) taught you to read a line of one: which column is which, what an operand means, what the number in front of the parentheses is. This chapter is about what the lines say between them, which is a different skill — and the one the listings in [Part III](#) have been quietly assuming. By the end of it you should be able to open an unfamiliar function, find where its arguments went, work out how much stack it wanted and why, and walk back up the chain of calls that reached it — by hand, and then in a debugger, and understand that these are the same operation.

The material

There are two kinds of register, and that is most of the convention

A machine has a fixed number of registers and any function may use any of them. That is a problem the moment one function calls another: the callee will want registers, and the caller still needs what was in them.

There are only two ways to settle it, and a calling convention picks both. Either **the caller saves** a register before making the call and restores it afterwards, or **the callee saves** it on entry and puts it back before returning. Every register is assigned to one group or the other RISC-V INTERNATIONAL (N.D.), and that assignment is why a prologue — the instructions that set up a function's stack frame — looks the way it does.

The consequence is about *lifetime*, not about register numbers:

- A value you need **after** a call should live in a **callee-saved** register, because the callee is obliged to hand it back unchanged. It costs a store in the prologue and a load in the epilogue — the matching instructions that take the frame down again — paid once per function.
- A value you need only **between** calls can live in a **caller-saved** one, which costs nothing at all, because nobody has promised anything about it.

The compiler makes that decision for every value in every function, and a great deal of what looks like arbitrary register choice in a disassembly is this rule being applied. [Appendix A](#) lists which register is in which group; the list is not worth memorising and the rule is.

A stack frame is a linked list

A stack frame, and how to leave it

Two fixed slots turn the frames into a linked list, which is all a backtrace is.



Only while frame pointers are kept. Compiled without them the slots are not written, the list does not exist, and a backtrace has to be reconstructed from debug tables instead — which is why a release build's stack trace is so often a disappointment.

Two fixed slots, and the frames become something you can walk.

When a function needs more room than the registers give it — somewhere to put a callee-saved register it wants to use, somewhere for an array, somewhere for arguments beyond the ones that travel in registers — it takes that room from the stack, by subtracting from the stack pointer. That block is its **frame**.

Two of the slots in it sit at fixed offsets, which is what makes a backtrace possible: the *return address* — where the call came from, and so where the function will resume — and the caller's *frame pointer*, the register that marks where a frame starts. Follow the second and you are in the caller's frame; read the first and you know where it will resume. That is a linked list, and walking it is a loop:

framewalk.c

```
for (int depth = 0; depth < SYSFS_MAX_FRAMES; depth++) {
    void *return_address = frame[RETURN_ADDRESS_SLOT];
    void **caller = (void **)frame[SAVED_FRAME_SLOT];

    /* Stop before dereferencing nonsense. The outermost frame's saved pointer is not another
     * frame, and a walker that trusts it reads whatever happens to be there — which is how a
     * stack trace comes to contain a function that was never called. */
    if (caller == NULL || caller <= frame) {
        printf("frame %d end\n", depth);
        break;
    }
}
```

Build it and run it here:

```
./run framewalk
```

Check it against the expected frames:

```
python3 -m pytest tests/test_framework.c.py -q
```

A debugger does exactly this, with two differences: it reads the frames out of another process, and it can do the walk without frame pointers by consulting tables the compiler emitted for the purpose. That second ability is why `gdb` can produce a backtrace in code compiled without `-fno-omit-frame-pointer`, and why it sometimes cannot.

`framework.c` marks all three of its call-chain functions `__noinline__`, and the comment saying why is a lesson rather than a workaround:

framework.c

```
/* `__noinline__` on all three, and the reason is the lesson rather than a workaround. Without it the
 * optimiser inlines this chain into main and the walk below finds one frame where the source has
 * four — because there genuinely is one. A frame is not a call in the source, it is a call the
 * machine still makes, and inlining is the commonest reason a stack trace is shorter than the
 * code that produced it. */
```

A frame is not a call in the source. It is a call the machine still makes. Inlining is the commonest reason a stack trace is shorter than the code that produced it, and the commonest reason a function you are certain is on the stack does not appear in the backtrace.

How much stack does a function need?

There is no general answer. Four functions, compiled twice:

Function	Frame -O0	Frame -O2	Instructions -O0	Instructions -O2	Memory ops -O0	Memory ops -O2
<code>sysfs_leaf</code>	32	0	20	3	7	0
<code>sysfs_calls_out</code>	32	16	18	8	6	2
<code>sysfs_many_locals</code>	112	0	46	12	30	0
<code>sysfs_accumulates</code>	48	48	35	32	18	13

Conditions: target `xv6` · riscv64 cross compiler, any machine — nothing here was executed · `riscv64-linux-gnu-gcc` (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · `-Wall -Wextra -march=rv64gc -mabi=lp64d -c, at -O0 and -O2` · 2026-09-17 · Source: `bench/results/framesizes-riscv64.json`, code hash `6d902f888daf3ea8`.

Read across, then down.

`sysfs_leaf` calls nothing. At `-O0` it still takes a frame and still writes to it, because `-O0` puts every named value in memory and reloads it on each use — that is what makes stepping through unoptimised code in a debugger so pleasant and what makes it so slow. At `-O2` it has no frame at all and touches no memory: arguments arrive in registers, the result leaves in one, and nothing ever needed an address.

`sysfs_many_locals` is the striking one. It takes eight arguments and computes four intermediates, and at `-O0` it wants the largest frame in the table and does more memory operations than it has instructions at `-O2`. At `-O2` it needs no frame either. Everything fit.

That pair shows what an optimiser is actually for. It is not making the arithmetic cleverer — the multiplications are the same multiplications. **It is keeping values in registers instead of in memory**, and the difference between those two is the subject of [ch25](#).

sysfs_calls_out keeps a frame at **-02**, and a small one. It calls something, so the return address in **ra** — the register a call leaves it in — is no longer safe where it is: the callee’s own **jal**, the call instruction, will overwrite it. A frame appears for exactly one reason and holds exactly one thing.

sysfs_accumulates keeps the same frame at both levels. It has a loop with a call in it, and a running total that must survive each call. That total has to live in a callee-saved register, and a callee-saved register has to be saved. The optimiser cannot make this one disappear because the requirement is not an inefficiency; it is the convention being obeyed.

Reading a prologue

Two of those functions, as the compiler emitted them. The leaf:

sysfs_leaf compiled for riscv64 at **-02** — the **xv6** target’s instruction set, 3 instructions.

```
0000000000000000 <sysfs_leaf>:
0:   mulw    a1,a0,a1
4:   addw    a0,a0,a1
6:   ret
```

Conditions: target **xv6** · riscv64 cross compiler, any machine — nothing here was executed · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · **-02 -Wall -Wextra -march=rv64gc -mabi=lp64d -c** · 2026-09-17 · Source: **bench/results/frames-riscv64.json**, code hash **6b398372e546affb**.

No prologue, no epilogue, no memory. Arguments in, answer out.

And the one that calls:

sysfs_calls_out compiled for riscv64 at **-02** — the **xv6** target’s instruction set, 8 instructions.

```
0000000000000008 <sysfs_calls_out>:
8:   addi    sp,sp,-16
a:   sd      ra,8(sp)
c:   auipc   ra,0x0
10:  jalr     ra # c <sysfs_calls_out+0x4>
14:  ld       ra,8(sp)
16:  addiw    a0,a0,1
18:  addi     sp,sp,16
1a:  ret
```

Conditions: target **xv6** · riscv64 cross compiler, any machine — nothing here was executed · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · **-02 -Wall -Wextra -march=rv64gc -mabi=lp64d -c** · 2026-09-17 · Source: **bench/results/frames-riscv64.json**, code hash **6b398372e546affb**.

You will see this shape several thousand times: make room, put the return address in it, do the work, take the return address back, release the room, return. A prologue and an epilogue are a matched pair, and the number in both is the same number.

Control flow is a comparison and a branch

There are no loops in machine code and no **if** statements. There is a comparison, and there is a branch that is taken or not — which is why **ch12**’s counted loop ended in a branch pointing backwards, and why a loop and an **if** look so similar once compiled.

RISC-V folds the comparison and the branch into one instruction, which is a design choice rather than a universal: it has no condition-code register — no register that remembers the result of the last comparison — so there is nothing to carry between them. That difference resurfaces in [ch26](#) on a machine that does have one, and it is one of the few places where the two architectures genuinely do not translate word for word.

Stepping it in the debugger

Everything above can also be done interactively against a real xv6 binary, where you can step the instructions rather than read them:

```
make xv6-gdb          # in one terminal: boots halted, waiting
```

[Appendix B](#) has the workflow — attaching, setting a breakpoint in a user program, stepping one instruction at a time, and printing the registers. Do this at least once: break on entry to a function, read the frame pointer, and find the return address yourself with `x/gx`, then type `backtrace` and watch the debugger produce the same answer.

What we measured

Frame sizes and instruction counts, read out of the disassembly, at two optimisation levels. They are counts rather than costs, stamped as artefacts, and regenerated by CI.

The table establishes one thing: **whether a function needs a frame is a property of what it does, not of how big it looks**. Two of these four shed their frames entirely under optimisation and two did not, and the two that did not are exactly the two that call something. That is a rule you can apply to a function you have never seen.

What this cannot tell you

Whether a frame costs anything. A store and a load in a prologue is two memory operations, and what they cost depends on whether that stack line is in cache — which it almost always is, for reasons [ch25](#) explains and this chapter has no way to demonstrate.

Whether `-O0` is “slower”. It is certainly more instructions and far more memory traffic, and those are the numbers above. Turning that into a ratio requires a clock and a machine, and this chapter has neither. It is a common mistake to assume the ratio follows the instruction count; [ch27](#) is largely about why it does not.

What the registers are called on the other machine. Everything here is RISC-V. AArch64 divides its registers the same way, into caller-saved and callee-saved, and gives them entirely different names and a different number of argument registers. The *rule* transfers; the table does not, and [Appendix F](#) is the translation for the reader who meets it in [ch26](#).

How the compiler chose. Register allocation is an optimisation problem with a large literature and this chapter deliberately does not enter it. It teaches how to read the *result* instead, which is the durable skill: allocators change, and prologues do not.

Problems

Three, and the first is the hardest thing in [Part III](#) so far.

14.1 — Write the C that produced this. A listing, and you write a C function that compiles to the same instructions. Not similar ones — the same ones, in order. The test compiles your attempt with the same toolchain and compares mnemonic by mnemonic, so a function that gets the right answer another way does not pass.

Reconstructing the behaviour is the easy half. Reconstructing the compiler’s route to it means asking why *that* instruction and not the obvious one, which is the question this chapter has been teaching you to ask.

```
python3 -m pytest tests/machine_level_code_on_riscv/test_problem_1_reconstruct.py
```

14.2 — Which of these needs a frame? Four functions. For each, predict whether it gets a frame at `-02` and whether it saves the return address. The test compiles each one and reads the prologue, so you are predicting a compiler rather than reciting a rule — and the reasoning that explains the table above explains all four.

```
python3 -m pytest tests/machine_level_code_on_riscv/test_problem_2_frames.py
```

14.3 — Who saves what? Name one register a function writes without saving, and one it saves before using. The test checks both against the actual disassembly *and* against which half of the convention each belongs to, so an answer that is mechanically true but comes from the wrong group fails and tells you so.

```
python3 -m pytest tests/machine_level_code_on_riscv/test_problem_3_who_saves.py
```

Where to go next

The RISC-V psABI RISC-V INTERNATIONAL (N.D.) is the document that decides everything in this chapter that is not a compiler’s choice: which registers are caller-saved, where arguments go, how the stack is aligned, and what a frame must look like. It is short by the standards of such documents and the register table in it is worth having open the first few times you read a disassembly.

The unprivileged specification RISC-V INTERNATIONAL (N.D.) defines the instructions themselves. Its instruction listing answers “what does `sd` actually do” faster than any tutorial.

xv6’s `kernel/swtch.S` COX ET AL. (N.D.) is a few dozen lines of assembly that save fourteen registers into one context and restore fourteen from another, and that is the entire mechanism of a context switch. Read it now. You will not know *why* it is called or what a context is until [ch21](#), but you can already read every instruction in it, which is a good way to find out that you can.

[ch15](#) asks where all this ends up: sections, segments, symbols, and what `exec` does with them.

15 · Linking and Loading

CHAPTER HEADER

Target	xv6 — the teaching kernel under QEMU
Prerequisites	ch14
What it measures	Sections, segments and symbol counts for xv6's own binaries, read by this book's reader: <code>bench/results/elf-xv6.json</code>

The question

How does a file on disk become an address space?

[ch12](#) left the linker as the stage that “fills in the holes” and moved on. This chapter opens the file it produced. By the end you should be able to say what is in an executable, which parts of it will exist at run time and which will not, and what `exec` has to do to turn one into a running program — and you should have written enough of a reader to believe it.

The material

Write the reader

A reader who has only ever used `readelf` — the standard tool for printing what an ELF file contains — believes an ELF file is a thing a tool understands. The fastest cure is to write a tool.

elfdump.c

```
/* Field offsets, straight out of the specification. Named rather than written inline so that a
 * reader with the document open can check each one. */
#define EH_TYPE 16
#define EH_MACHINE 18
#define EH_ENTRY 24
#define EH_PHOFF 32
#define EH_SHOFF 40
#define EH_PHENTSIZE 54
#define EH_PHNUM 56
#define EH_SHENTSIZE 58
#define EH_SHNUM 60
#define EH_SHSTRNDX 62
```

Build it and point it at something:

```
./run elfdump xv6/stage/user/_sameanswer
```

That is most of what an ELF file is: a header with a magic number — a few fixed bytes that say *this is ELF* — a handful of sizes, and the offsets of two arrays. Everything else is found by following one of those offsets. There is no `#include <elf.h>` anywhere in `sysfs/tools/elfdump.c`, deliberately — the structures are declared from the specification THE SANTA CRUZ OPERATION (N.D.), so that they read as a documented layout of bytes rather than as something only a library may know.

Two arrays, two audiences

The two arrays describe the same bytes to different readers, and confusing them is the single most common muddle about executables.

Sections are the linker's view. They are named, they are numerous, and most of them exist to be combined with the corresponding sections of other objects: put all the `.text` — the instructions — together, all the `.rodata` — the read-only data — together, resolve what refers to what. After linking, most of them have no further purpose — the debug sections in particular are enormous and are never loaded.

Segments are the loader's view, and there are far fewer:

Two views of the same bytes

Sections are for the linker. Segments are for whatever has to load the thing.

18 sections, 5 of them loaded

<code>.text</code>	2368 bytes
<code>.rodata</code>	145 bytes
<code>.eh_frame</code>	1148 bytes
<code>.data</code>	0 bytes
<code>.bss</code>	32 bytes

2 segments

<code>r-x code</code>	at 0
<code>rw- data</code>	at 4096 (32 bytes not

the loader is told: map this much, from here, and zero the rest

A segment whose memory size exceeds its file size is asking for space that is not in the file: `.bss`. Storing a megabyte of zeroes would be silly, so the format says how many there are and the loader provides them.

The same bytes, described twice. The second description is the one the machine acts on.

A segment says: *take this many bytes from this offset in the file, put them at this address, give them these permissions*. That is nearly the whole interface between a file and an address space.

Program	Sections	Loadable segments	Symbols defined	Symbols undefined
<code>ls</code>	18	2	60	0
<code>sameanswer</code>	18	2	59	0

Conditions: target `xv6` · riscv64 cross compiler, any machine — nothing here was executed · `cc` (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · the reader is built for this machine; the subjects are built by `xv6` · 2026-09-17 · Source: `bench/results/elf-xv6.json`, code hash `85308ce82a349d62`.

Eighteen sections, two segments. The collapse is by permission: everything that needs to be readable and executable goes in one, everything writable in the other. Permissions are a property of a *page*, not of a symbol, which is why the grouping is by what the hardware can enforce rather than by what the programmer named.

The segment that is partly not in the file

Program	Permissions	Loaded at	Bytes in the file	Bytes in memory	Zeroed
ls	r-x	0	4124	4124	0
ls	rw-	8192	0	48	48
sameanswer	r-x	0	3668	3668	0
sameanswer	rw-	4096	0	32	32

Conditions: target `xv6` · riscv64 cross compiler, any machine — nothing here was executed · `cc` (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · the reader is built for this machine; the subjects are built by `xv6` · 2026-09-17 · Source: `bench/results/elf-xv6.json`, code hash `85308ce82a349d62`.

Look at the writable rows. Those segments occupy bytes in memory and none at all in the file.

That is `.bss`: variables that start as zero. Storing them would mean putting a run of zeroes in every binary, so the format does not. It records how much space to provide and the loader provides it, already zeroed. A global array of a million integers costs a number in a header.

Two consequences worth carrying forward. A binary’s size on disk tells you little about its size in memory — [ch12](#)’s comparison of `glibc` against `xv6`’s library was about code, and this is the other direction. And “zero-initialised” is not a favour the language does you at some cost; it is the *cheapest* initial state, because it is the one the loader was going to produce anyway.

What `exec` actually does

The interface above is small enough that you can now read the kernel side of it. `xv6`’s `exec` opens the file, checks the magic number, walks the program headers — the segment array — and for each loadable one allocates pages and copies bytes in *COX ET AL.* (N.D.). Then it builds a stack, puts the arguments on it, and switches the process to the new address space.

Everything it does is a consequence of what the file said. It has no idea what the program does, what language it was written in, or what its functions are called — the symbol table is not consulted. **An ELF executable is a set of instructions to a loader, and the code is incidental.**

The permissions matter here in a way they do not in a file. The code segment is mapped executable-and-not-writable and the data segment writable-and-not-executable, and those two prohibitions are enforced by the page table rather than by convention. [ch17](#) is where that enforcement becomes a mechanism you can see and [ch18](#) is where breaking it becomes a fault you can catch.

The symbol table, which is not loaded

The symbol table is not loaded and is not needed to run the program. It survives in the file for the benefit of debuggers and of anybody reading it, which is why stripping a binary makes it smaller without making it slower, and why a stripped binary produces a backtrace full of addresses and no names.

It cuts both ways: the names in a backtrace are a convenience the file happens to carry, and a production binary that has been stripped has thrown them away permanently — which is why [ch30](#) spends time on keeping symbols around for the profiler.

What we measured

Sections, segments and symbol counts for two xv6 programs, read by `sysfs/tools/elfdump.c` rather than by `readelf`. Facts about files, stamped as artefacts, regenerated by CI.

xv6's binaries were chosen over Linux ones because a Linux executable's answer to "where does this get loaded" is complicated by a dynamic loader and a position-independent layout, both of which deserve their own explanation and neither of which is the mechanism underneath. xv6 links a program to a fixed address and puts it there. That is the simple version of what the complicated version is doing.

What this cannot tell you

How dynamic linking works. Every binary here is statically linked to a fixed address. Shared libraries, relocation at load time, the procedure linkage table and the global offset table are all absent, and they are the majority of what happens when you run a program on Linux. [ch29](#) touches the cost of the machinery; this chapter does not describe it.

What a linker script decides. xv6 has one, it fixes the addresses in the table above, and this chapter shows the *result* of it rather than the language it is written in. Linker scripts are a small, strange, badly documented language, so this chapter does no more than say the addresses came from somewhere and point at the file.

Anything about time. Loading a program costs something — pages have to be allocated and bytes copied — and this chapter has no way to measure it. [ch29](#) measures what a process costs to start on a machine with a clock.

Whether the symbol table is "wasted space". It is not loaded, so it costs no memory at run time; it costs file size and it buys every debugging session you will ever have. That is a trade-off rather than a defect, and which side of it you want depends on things this chapter has no opinion about.

Problems

Three, and the first is the one that makes the format stop being magic.

15.1 — Finish the reader. Two functions `sysfs/tools/elfdump.c` does not contain: how much memory a program occupies once loaded, and which section covers a given address. The second one requires the indirection this chapter describes — a section header does not hold its own name.

The expected answers are not stored. The test runs the book's finished reader on the same binary and compares, so the target moves with the file rather than going stale the next time xv6 is rebuilt.

```
python3 -m pytest tests/linking_and_loading/test_problem_1_reader.py
```

15.2 — Which of these links? Five pairs of C files. Predict whether each produces a program. The test actually links them.

One of the five links successfully and produces a program that is wrong, which is not a trick: it is the reason the C language has a reputation. Say whether it links; the chapter has already said what to make of the fact that it does.

```
python3 -m pytest tests/linking_and_loading/test_problem_2_resolve.py
```

15.3 — Read the error, name the cause. Four link failures, with the messages a linker really produced during the test run. Name the cause of each.

Linker errors are terse, they name symbols rather than files, and they come from a program that knows nothing about your intentions. Reading one is a skill, and it is the difference between twenty minutes and ten seconds.

```
python3 -m pytest tests/linking_and_loading/test_problem_3_diagnose.py
```

Where to go next

The ELF specification *THE SANTA CRUZ OPERATION* (N.D.) is the document `sysfs/tools/elfdump.c` was written from, and it is unusually pleasant for a format specification: the structures are small, the field names are the ones every tool uses, and you can check the reader against it line by line.

The RISC-V psABI *RISC-V INTERNATIONAL* (N.D.) supplies the architecture-specific half — which relocation types exist and what each computes — and is where to look when a relocation name appears in an error message.

xv6’s `kernel/exec.c` *COX ET AL.* (N.D.) is short, and it is now readable end to end. It is the shortest complete answer to “what happens when you run a program” that exists anywhere, and having written a reader for the format it consumes, you will find it contains no surprises at all. That feeling is what **Part III** was for.

ch16 begins **Part IV** by asking what happens when that program asks the kernel for something.

Part IV — The operating system layer

Part IV · The operating system layer · 16 · Traps and System Calls · 17 · Virtual Memory · 18 · Page Faults as a Feature · 19 · Interrupts and Drivers · 20 · Locks and Memory Ordering · 21 · Scheduling and Context Switches · 22 · The File System · 23 · The Same Program on Both Targets

Part IV · The operating system layer

PART HEADER

Chapters	ch16–ch23
Target	xv6 — the teaching kernel under QEMU, with ch23 crossing to host
Assumes	Part II, Part III

What this part is for

A complete operating system, small enough to read, taken apart one mechanism at a time.

Part II removed the entanglement deliberately: each primitive on its own, on a machine with nothing else running. This part puts it back, and the entanglement turns out to be most of what an operating system *is*. A trap handler that also has to find the right process, on the right hart, without losing an interrupt, while another hart is in the same code, is not a harder version of **ch06**'s handler. It is a different subject, and it is this one.

The kernel is xv6, because it is short enough to read in full and you can stop the entire machine in the middle of a trap and look at anything. That matters more than modernity here. A production kernel would have the same mechanisms buried under thirty years of necessary special cases, and you would be taking someone's word for which lines were the point.

The order of the chapters follows the dependencies. Traps come before virtual memory because a page fault — the trap taken when an address has no mapping — is a trap. Interrupts come before locks because a lock's first job is to survive one. Locks come before scheduling because a scheduler is largely the thing that needed them.

What it leaves out

Cost. Every question of the form *how long does this take* is refused here and asked again in **Part V**, where there is hardware that can answer it. A timing taken under emulation is a fact about the emulator, so a chapter that produced one would be teaching you to trust the wrong instrument.

Linux, except by contrast. xv6 is a teaching kernel and some of its choices are simplifications rather than designs; the chapters say which, and **ch29** puts the same operations to a production kernel. Left out entirely: networking, a scheduler with a policy worth arguing about, and the parts of a file system that exist because disks are large rather than because crashes happen.

Where to start

ch16, in order. Each chapter names the one before it as its prerequisite and means it.

ch23 is the exception and can be read early if you want to know where the book is going. It runs one program on both targets and is the hinge into **Part V** — the chapter that stops being about how a thing works and starts being about how you would find out what it costs.

Which machine, and what it cannot tell you

xv6 under qemu-system-riscv64, read in a debugger.

Nothing in this part is timed, and nothing in it may be. QEMU is the right instrument for structure and the wrong one for everything else. It will tell you exactly which register held the faulting address and in what order the stores happened in the program's own view; it models no cache, no branch predictor, no store buffer and no memory latency, so it cannot tell you what any of it cost or what a second hart really does to the first.

The sharper limit is that **QEMU lies convincingly**. A duration measured here is a real number, reproducible, and about the host machine and the translation strategy rather than about RISC-V — and once written into a table it is indistinguishable from a measurement. That is why the rule is absolute and why a check enforces it rather than a convention.

Where this leaves you

Knowing what the mechanisms are and where they live, and therefore able to ask what they cost — the only question left.

16 · Traps and System Calls

CHAPTER HEADER

Target	xv6 — the teaching kernel under QEMU
Prerequisites	ch15
What it measures	The length of the trap path, and what a fixed workload asks the kernel for: <code>bench/results/traps-xv6.json</code>

The question

What does the hardware do when a program asks the kernel for something?

Part III ended with a program loaded into an address space and running. It cannot do anything useful on its own — it cannot read a file, write to the console, or obtain more memory — because none of those are things a user program is permitted to do. Everything interesting requires asking the kernel, and this chapter follows one such request all the way in and back out.

The material

A trap is not a call

The instruction is `ecall`, and it is not a function call into the kernel, though it is easy to read it as one. They differ in nearly every respect, and the differences matter.

A function call is an agreement. The caller knows it is calling, has arranged its registers accordingly, and the convention tells both sides who preserves what — that was [ch14](#).

`ecall` is the one trap a program does agree to, and it gets nothing for agreeing, because **it has no path of its own**. One entry point serves all four of `ecall`, an illegal instruction, a page fault and a timer interrupt, and three of those arrive without being asked for: the interrupted code made no arrangements and has to find every register exactly as it left it. The path has to satisfy the worst of its callers, so it has no calling convention to lean on.

So the trap path cannot save “the callee-saved registers”. It has to save **everything**, and put everything back. The census at the end of this chapter shows the same point in a table: one counter, entered for several unrelated reasons.

The hardware’s half

`ecall` does a small, fixed amount of work, and it is small because this is the part implemented in silicon RISC-V INTERNATIONAL (N.D.). It records the address of the instruction that trapped, records why the trap happened, disables interrupts, switches the privilege level, and jumps to an address the kernel installed earlier.

It does *not* save registers, switch stacks, or change the page table. The processor’s contribution is to get control to a known address with the reason available, and everything else is software’s problem. The same division holds on every architecture, even where the names differ.

The software's half, and the page it lives in

What one system call actually costs to arrange

The work you asked for is the box in the middle. Everything else is getting there.



and twice, in the middle of it, the address space changes

uservec switches to the kernel page table; userret switches back. That is why both halves live in one page mapped at the same address in both — the trampoline.

uservec 44 instructions	in 83 instructions of pure state movement, out before any of your work begins
userret 39 instructions	

Counted, not timed. This target cannot say what an instruction costs, and a count is what remains true anyway: the path is this long whatever machine runs it. The OS-cost chapter prices the same shape on hardware.

The work you asked for is the box in the middle.

The diagram leaves out a problem the code has to solve. The kernel runs with a different page table from the process, so at some point the address space must change — and the instant it changes, the code that is running must still be mapped, or the next instruction fetch faults. A page table cannot be swapped from code that is only in one of the two.

xv6 solves it the way real kernels do: one page, the **trampoline**, mapped at the same virtual address in every address space, kernel and user alike. The switch happens inside that page, so whichever table is active the instruction after the switch is at an address that is still valid. [ch17](#) explains page tables properly; this is the one place in [Part IV](#) where a chapter has to promise that something later will make sense.

How long is the path?

Half of the path	Instructions	Registers saved	Registers restored	CSR ops
userret	39	0	31	1
uservec	44	31	4	3

Conditions: target `xv6` · `qemu virt (-machine virt -bios none)` · `QEMU emulator version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.18)` · `xv6-riscv @ 9e3161a · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0` · `xv6's own CFLAGS` · 2026-09-17 · Source: `bench/results/traps-xv6.json`, code hash `f78d829cb10e7c2c`.

Those counts are the state movement in full: the way in — `uservec`, the assembly in the trampoline page — saves thirty-one registers, the way out — `userret` — restores them, and neither is doing anything you asked for. Before `usertrap`, the C the assembly hands over to, has looked at why it was entered — before any argument has been examined or any work begun — the machine has executed over forty instructions of pure bookkeeping, and it will execute nearly as many again on the way out, after the work is finished.

`uservec` also does a handful of CSR operations, and one of them is the page-table switch. `userret` does fewer, because `sret` — supervisor mode's `mret` — puts the privilege level and the interrupt state back in one instruction.

This is a count and not a cost, which is why it appears here rather than in [Part V](#). What an instruction costs depends on a pipeline and a cache, and this target has neither. The path is this long on any machine. [ch29](#) takes the same shape to hardware and puts a price on it.

Counting what actually happens

The kernel has been patched to count every trap by cause and print the table when you press Ctrl-T, the way it already prints the process table on Ctrl-P:

13-trap-census.patch

```
+static void
+census_note(uint64 scause, int user)
+{
+  uint64 code = scause & 0xf;
+  if (scause >> 63)
+    census.interrupt[code]++;
+  else if ((scause & ~(uint64)0xf) == 0)
+    census.exception[code]++;
+  else
+    census.unclassified++;
+
+  if (user)
+    census.from_user++;
+  else
+    census.from_kernel++;
+}
+
+// Called from syscall() once the number is known, so the census can say *which* calls a workload
+// made rather than only how many. The per-number counts are what the traps-and-system-calls chapter
+// can reproduce exactly: a
+// program that asks for one thing a thousand times is counted a thousand times, whatever the
+// shell around it happened to do.
+void
+census_syscall(int number)
+{
+  if (number >= 0 && number < NSYSCALL)
+    census.syscall[number]++;
+}
+
```

Then run a workload and ask:

What the kernel was entered for	This run
<code>getpid</code> calls the workload asked for (syscall 11)	1000
Exception causes seen	system call (ecall from user mode)
Interrupt causes seen (counts deliberately not recorded)	5, 9

Conditions: target `xv6` · qemu virt (-machine virt -bios none) · QEMU emulator version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.18) · xv6-riscv @ 9e3161a · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · xv6's own CFLAGS · 2026-09-17 · Source: `bench/results/traps-xv6.json`, code hash `f78d829cb10e7c2c`.

Most of that table is a list rather than a count, because those counts were not reproducible — and the reason is worth more than the numbers would have been.

The obvious thing to measure was the shell: boot, run `ls`, and report how many system calls it took. That produced a satisfyingly large number — and a different one each time. How many times the shell calls `read` depends on how the console delivered its characters, which depends on timing, and elapsed time inside QEMU is a property of the laptop running it. Two runs of an identical workload disagreed, which is how this was found out rather than assumed.

An *exception* — a trap an instruction caused, as opposed to an interrupt — is reproducible for a *fixed sequence of instructions*, because the same instructions cause the same traps. The trouble is that “run the shell” is not a fixed sequence of instructions. So the workload is a program that removes the question:

trapload.c

```
int main(void) {
    int last = 0;
    for (int i = 0; i < SYSFS_TRAPLOAD_CALLS; i++) {
        last = getpid();
    }
    printf("trapload %d calls pid %d\n", SYSFS_TRAPLOAD_CALLS, last);
    exit(0);
}
```

Build it into the kernel, boot, and run it:

```
./run trapload
```

Nothing else in the system calls `getpid`, so that row of the census is a number this program decided and the shell’s noise lands elsewhere. It asks a thousand times and the kernel counts a thousand — which is a much smaller claim than the one that was almost printed, and unlike it, true on every machine.

This is where [ch00](#)’s discipline costs something. The large number was easy, impressive, and meaningless. [ch29](#) counts the lot, on a machine where elapsed time is a fact about the machine.

The register census, and why it is not thirty-two

Thirty-one registers are saved. RV64 has thirty-two. The missing one is not an oversight and working out which it is — and why a trap path is allowed to ignore it — is one of this chapter’s problems.

What we measured

A path length, read out of the built kernel, and a trap census, read out of a running one. Neither is a duration and this target could not produce one honestly.

The census is reproducible because the workload is fixed and because only the deterministic half of it is recorded. CI re-runs it on every push and compares. A kernel patch that lost some of the workload’s thousand calls, or that introduced a cause this list does not have, would move the table — and moving it silently is exactly the failure the whole stamping scheme exists to prevent.

The check deliberately cannot catch a change in how often the *shell* asks the kernel for anything, because that was never recorded. Giving up that number bought the rest of the table its reproducibility, for the reasons the section above gives.

What this cannot tell you

What any of this costs. The path is eighty-odd instructions long; whether that is expensive depends on whether they hit in cache, whether the pipeline drains, and what the page-table switch does to the TLB (the hardware cache of address translations) — three questions this target has no opinion about whatsoever. It is entirely possible for the *shorter* of two paths to be the slower one, and [ch29](#) is where that gets settled.

What a real kernel's path looks like. xv6's is short because xv6 is small. Linux's does considerably more on the way in — checking for signals, handling seccomp filters, auditing — and its fast paths exist precisely to avoid some of it. The shape is the same; the length is not.

Why the timer fires when it does. The census shows timer interrupts occurred and nothing else about them, because their frequency here is an artefact of emulation. [ch21](#) explains what the kernel does with them.

Anything about interrupts arriving during a trap. `ecall` disables them and `usertrap` turns them back on deliberately, at a point chosen for a reason the code comments explain. Nested traps, and what happens when one arrives at the wrong moment, are [ch19](#)'s subject.

Problems

Two, and the first is the one every xv6 reader should do once.

16.1 — Add a system call, end to end. The census is printed on Ctrl-T, which is no use to a program. Add `trapcount()` and a user program that calls it.

Five files have to agree before this works, and finding all five is the exercise: the number that identifies the call, the table that dispatches on it, the declaration the user program compiles against, the stub that issues `ecall`, and the kernel function itself. Leaving any one out produces an error that does not mention the others.

The test cannot check that your number is *correct* without reimplementing your counter, so it checks the two things that separate a working system call from a plausible stub: the value is not zero, and it grows between two runs. No constant passes both.

```
python3 -m pytest tests/traps_and_system_calls/test_problem_1_addcall.py
```

16.2 — Which registers must be saved, and which need not be? Count them, then say which ones `uservec` leaves alone and why each is allowed. There is more than one reason a register can end up on that list, and this chapter has given you two of them.

The test reads the count out of the kernel as built, so if you patch the trampoline you are graded against your own kernel rather than against the book's.

```
python3 -m pytest tests/traps_and_system_calls/test_problem_2_registers.py
```

Where to go next

The RISC-V privileged specification RISC-V INTERNATIONAL (N.D.) defines exactly what `ecall` and `sret` do and what each CSR in the path is for. The section on trap handling is short, and reading it alongside `trampoline.S` is the fastest way to work out which half of the path is hardware.

xv6's `kernel/trampoline.S` and `kernel/trap.c` COX *ET AL.* (N.D.) are now readable in full — the assembly is the path this chapter counted, and `usertrap` is a short piece of C. Read the assembly first and the C second; the order matters, because the C makes no sense until you know what state it has been handed.

ch17 takes the promise this chapter made — that a page table is a thing, that the trampoline is mapped into two of them — and turns it into something you can print.

17 · Virtual Memory

CHAPTER HEADER

Target	xv6 — the teaching kernel under QEMU
Prerequisites	ch16
What it measures	The page-table shape of a running process: levels, entries, and physical pages consumed per mapping: bench/results/pagetable-xv6.json

The question

What is an address, and who decides what it means?

Every chapter so far has treated an address as a number that names a place. [ch15](#) watched a loader put a program at the address its linker chose, and [ch16](#) left a promise: the trampoline page is mapped into two address spaces at once. That sentence only makes sense if an address means different things to different programs, and it does. This chapter is about the page tables that arrange it, and about what they cost — not in time, which this target cannot tell you, but in the memory it takes to say where memory is.

The material

An address is a question

The instruction `ld a0, 0(a1)` does not name a location. It names a number, and something between the core and the memory has to decide what that number refers to. On this machine the decision is made by a data structure in memory, and one register says where that structure begins.

That register is `satp`. Writing to it replaces every address in the running program's world at a stroke, which is why [ch16](#)'s trap path switches it and why the trampoline has to be mapped at the same address in both maps — the instruction after the switch has to still exist.

So “what is an address” has a precise answer. **An address is a question, `satp` names who answers, and the answer is looked up rather than computed.**

Nine, nine, nine, twelve

The scheme is called Sv39, and two decisions force the whole of its layout. There is nothing arbitrary in it to memorise.

Choose a page size, and choose how big an entry describing a page must be. Everything else follows: a table is one page of entries, so the number of entries per table is one divided by the other, and the number of bits needed to index a table is the logarithm of that. Stack enough levels to reach a useful address width and you have the whole design.

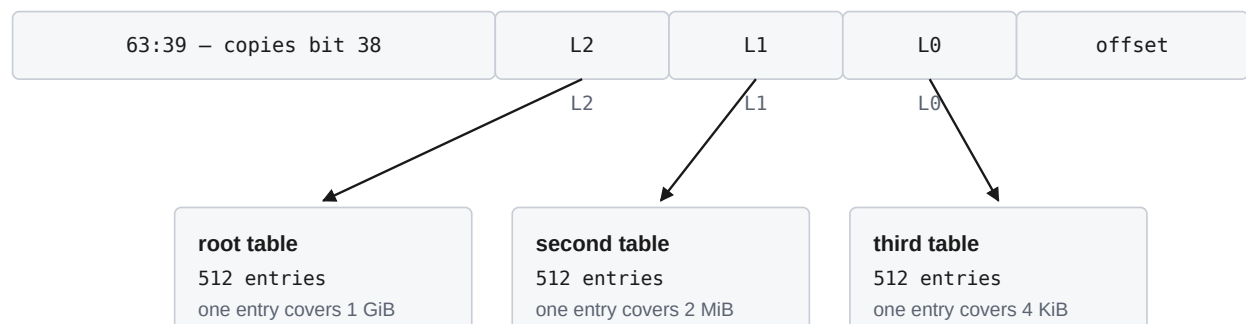
	Value	Where it comes from
Page size	4096 bytes	chosen
Page-table entry	8 bytes	chosen
Entries per table	512	page ÷ entry
Index bits per level	9	$\log_2(\text{entries})$
Levels	3	to reach 39 bits
One level-0 entry covers	4096 bytes	a page
One level-1 entry covers	2097152 bytes	512 pages
One level-2 entry covers	1073741824 bytes	512 of those

Conditions: target `xv6` · `qemu virt (-machine virt -bios none)` · QEMU emulator version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.18) · `xv6-riscv @ 9e3161a` · `riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0` · `xv6's own CFLAGS` · 2026-09-17 · Source: `bench/results/pagetable-xv6.json`, code hash `e2b798006ccdfc0d`.

Read the right-hand column. Only the first two rows were chosen. Nine bits per level is not a tuning parameter — it is what a 4096-byte page holding 8-byte entries makes you use, and a designer who picked a different page size would get a different number of levels rather than a different number of bits.

How Sv39 reads a virtual address

Nine bits per level, three levels, and twelve bits it never touches.



The offset is not translated. It is copied.

Why nine: a 4096-byte page holds 512 entries of 8 bytes, and 512 is nine bits.

Three levels of nine, plus twelve of offset, is thirty-nine — which is where the name comes from.

One virtual address, and the four things translation does with it.

Two details in that figure are easy to miss.

The offset is not translated. The bottom twelve bits are copied from the virtual address to the physical one, untouched. Translation moves pages about; it has no opinion about where in a page a byte is. That is why a page is the unit of everything in this chapter: it is the smallest thing the mechanism can say anything about.

The top twenty-five bits are not spare. Sv39 translates thirty-nine bits, and the rest of the word must all copy bit 38 — a *sign extension*, the top bit copied into every bit above it, which is how a narrow signed number is widened. An address that fails that rule is not an address that is out of range; it is not an address, and the hardware refuses it. xv6

declines to use the top half at all, capping its address space one bit below the maximum so that every address it ever forms has bit 38 clear and the question never arises.

A fault is a walk that stopped

Follow the indices down and one of three things happens. An entry with its valid bit clear ends the walk. An entry with valid set and none of read, write or execute points at the next table down. An entry with valid set and any of them is a leaf, and the walk is over.

So **a failed translation has a location**. It is not “this address is wrong” but “the walk got this far and stopped”, and which level it stopped at says something different each time. Stopping at the top level means nothing in that gigabyte of the address space exists. Stopping at the bottom means the neighbourhood is mapped and this particular page is not — a stack that has grown one page too far, say, rather than a wild pointer. [ch18](#) is about what a kernel can do with that distinction; problem 17.3 is about extracting it.

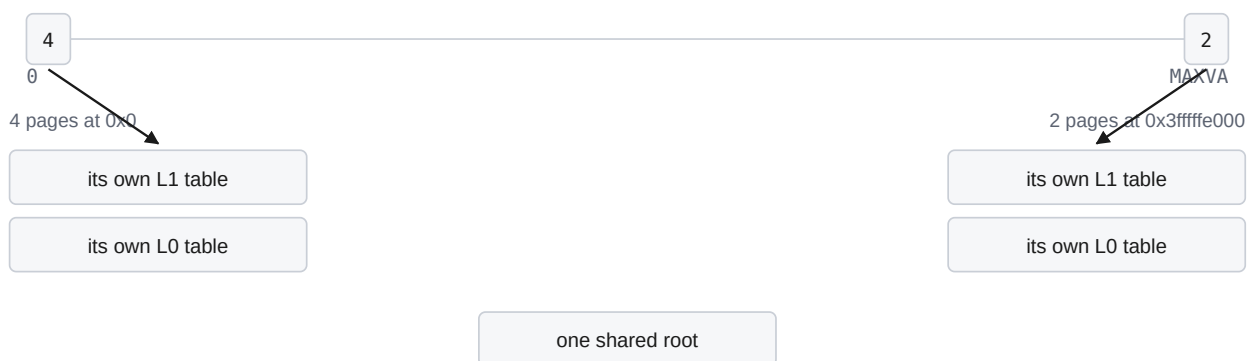
Two clusters and five tables

The rest of this section counts pages of table.

`init` is the first program xv6 runs, and the smallest interesting address space in the system. It maps six pages. Describing those six pages takes five more.

A page table's size is decided by where the pages are

`init`'s address space: six mapped pages, five pages of table to describe them.



5 pages of table for 6 pages of memory.

The kernel maps 49219 pages in 102 pages of table, because almost all of them are consecutive. Same mechanism, opposite result: the cost is per region, not per page.

Why six mapped pages need five pages of table.

The six are not in one place. Four are at the bottom of the address space, where the linker put the program. Two are at the very top: the *trapframe* — the page the trap path saves its thirty-one registers into — and the trampoline, which [ch16](#) put there. Those two clusters are separated by almost the whole of a 512-gigabyte address space, so neither can reuse any of the other’s tables — each forces its own chain down from the shared root.

A chain is two pages whether it ends in one mapping or five hundred.

The map costs memory

Put the totals side by side.

Address space	Page-table pages	Pages mapped	Separate regions	Overhead
kernel	102	49219	67	0.207%
init	5	6	2	83.3%

Conditions: target `xv6` · `qemu virt (-machine virt -bios none)` · QEMU emulator version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.18) · `xv6-riscv @ 9e3161a` · `riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0` · `xv6's own CFLAGS` · 2026-09-17 · Source: `bench/results/pagetable-xv6.json`, code hash `e2b798006ccdfc0d`.

The kernel maps an enormous amount of memory and spends almost nothing per page describing it. `init` maps almost nothing and spends nearly as much on the description as on the thing described. The last column explains both: the kernel's memory arrives in a handful of vast runs, and `init`'s in two tiny ones.

Two of the kernel's regions account for nearly everything it maps. One is the whole of the RAM it manages, direct-mapped so that a physical address and a kernel virtual address are the same number — a deliberate decision that makes the kernel's own pointers translatable without any bookkeeping at all. The other is a run of device registers so long that three separate devices, laid out end to end by the machine's designers, come out as a single region: the census cannot tell them apart, because as far as the page table is concerned they are not different things.

Almost every remaining region is a single page, and they are kernel stacks. `xv6` leaves an unmapped page between every pair of them, so that a kernel stack which overflows hits nothing rather than quietly writing into its neighbour. Each of those guard pages costs nothing — an absent mapping is an entry that is not there — and the arrangement costs one shared table for the lot, because they are all in the same two megabytes.

The most expensive two pages in the system

Go back to `init`'s five tables and ask which mapping is responsible for which.

The four pages at the bottom need a chain of two — one middle table, one last — under the shared root. Three pages of table for four pages of program, and every further page the program allocates is free until it crosses two megabytes.

The two pages at the top need another middle table and another last table, because nothing else is within a gigabyte of them. Two pages of table for two pages of mapping, and every process in the system pays it.

Those two pages are the trapframe and the trampoline. The trap path [ch16](#) measured in instructions has a second price that chapter could not express: per process, two pages of table, for as long as the process exists. Neither number is a cost in the sense this book usually means — [ch29](#) is where traps get priced in time — but both are real, and the second one is invisible unless somebody counts it.

The model and the kernel

The numbers above were obtained twice over, by two independent routes.

The kernel counts its own tables by walking them. Separately, `sysfs/lib/sv39.c` computes how many tables a set of mappings *requires*, from the addresses alone, with no machine involved:

```

uint64_t sysfs_sv39_tables(const struct sysfs_sv39_run *runs, size_t count, uint64_t out[3]) {
    /* One root, always. Below it, a level-1 table is needed for each distinct gigabyte the runs
     * touch and a level-0 table for each distinct two megabytes — so the answer depends on where
     * the pages are and not at all on how many of them there are.
     */
    /* Runs arrive sorted, so distinctness is a comparison with the previous chunk rather than a
     * set. That is not only cheaper; it is the reason this fits in a page of C. */
    uint64_t gigabytes = 0, leaves = 0;
    uint64_t previous_gib = 0, previous_2mib = 0;
    int started = 0;

    for (size_t i = 0; i < count; i++) {
        uint64_t first = runs[i].start;
        uint64_t last = first + runs[i].pages * SYSFS_SV39_PAGE_BYTES - 1;
        for (uint64_t chunk = first >> 21; chunk <= (last >> 21); chunk++) {
            uint64_t gib = chunk >> SYSFS_SV39_INDEX_BITS;
            if (!started || gib != previous_gib) {
                gigabytes++;
                previous_gib = gib;
            }
            if (!started || chunk != previous_2mib) {
                leaves++;
                previous_2mib = chunk;
            }
            started = 1;
        }
    }
}

```

`sysfs/tools/sv39.c` puts a command line on that, so you can ask about `init`'s two clusters directly — four pages at the bottom of the address space, two at the top, and the answer this chapter spent a section arriving at:

```
./run sv39 tables 0x0:4 0x3fffffe000:2
```

The two answers come from opposite ends — one from inspecting a running system, the other from a specification — and the runner refuses to record a result in which they disagree. The figures above are therefore not merely a reproducible measurement: they are two derivations agreeing, and if a future `xv6` changes its layout, `CI` will report the disagreement.

What we measured

The shape of two address spaces: how many pages of table each spends, how many pages each maps, and how many separate regions those pages fall into. Both are structural constants rather than anything a particular run produced — the kernel's map is built once from a layout fixed at compile time, and `init`'s is built by `exec` from a binary whose size the linker decided.

The shell's address space is deliberately not recorded. `xv6`'s shell calls `malloc` while it parses a command, so its size is a fact about what it has been asked to do rather than about address spaces, and [ch16](#) already found out what happens when those two get confused.

What this cannot tell you

What a translation costs. Nothing here is a duration, and the omission is not a limitation of effort. Translation is done by hardware that caches its results in a TLB, and `QEMU` models neither the cache nor the miss — so a walk measured here would take exactly as long as the emulator's bookkeeping and tell you about the laptop. [ch25](#)

measures what a miss costs on a machine that can charge for one, which is also where the three levels stop being free: a miss is not one memory access, it is up to three, before the access you asked for.

Whether a bigger page would help. Sv39 allows a leaf at the middle or top level — a two- megabyte or one- gigabyte mapping — and the census counts leaves by level precisely so that this would be visible. It counted none: xv6 never makes one. Whether that costs anything is a question about TLB reach, and this target has no TLB. The measurement above records that the format supports superpages and says nothing about whether using them is a good idea.

Anything about permissions. Every entry carries read, write, execute and user bits, and this chapter counted entries without looking at them. What those bits prevent, and what a kernel does when one of them is violated, is [ch18](#).

What a real system’s address space looks like. init maps six pages; a browser tab maps hundreds of thousands, in hundreds of regions, most of them nowhere near each other. The overhead column would look quite different, and the mechanism producing it is identical — which is the reason this chapter measured the small case, where every page of table can be accounted for by hand.

Problems

Three, and they are one program: `tests/virtual_memory/walk.c` builds a page table, follows it, and reports where following it stops. Nothing in `sysfs/` does any of those, so the repository contains no answer to any of them.

17.1 — Build a page table, and no more of one than the addresses require. Write `walk_map`, allocating interior tables as you need them.

The test does not compare your table with a stored one. It counts how many pages you took from the allocator and compares that with what this chapter’s own model derives from the addresses. Three address spaces are used, each with the same number of pages arranged differently, and they cost three different amounts. An implementation that allocates eagerly maps everything correctly and still fails — which is this chapter’s claim, turned into a check.

```
python3 -m pytest tests/virtual_memory/test_problem_1_map.py
```

17.2 — Follow it. Write `walk_translate`. The expected answers are the mappings the test asked problem 17.1 to make, so the target moves with your implementation rather than being a constant. Two things are easy to get wrong and are checked: the offset must survive, and an address that was never mapped must fault rather than return something plausible.

Handle a leaf found above the bottom level even though problem 17.1 never creates one. The format allows it, and code that assumes otherwise is wrong in a way that will not show up until it meets a kernel that uses superpages.

```
python3 -m pytest tests/virtual_memory/test_problem_2_translate.py
```

17.3 — Say where it stopped. Write `walk_first_missing_level`. Four probes: one mapped, and one each for a walk that fails at the top, the middle and the bottom. Each answer follows from where the test put the mappings, so there is nothing to look up.

This is the function a page-fault handler needs before it can decide anything, and [ch18](#) is about the decisions.

```
python3 -m pytest tests/virtual_memory/test_problem_3_fault.py
```

Where to go next

The RISC-V privileged specification RISC-V INTERNATIONAL (N.D.) defines Sv39 in about four pages: the entry format, the walk, and the rules about which reserved encodings fault. Read it after problem 17.2 rather than before — the specification is describing something you will by then have built, and it is much shorter than it looks.

xv6's `kernel/vm.c` COX ET AL. (N.D.) is now readable in full. `walk` is the function you wrote twice, with one extra argument that makes it do both jobs; `mappages` is a loop around it; `freewalk` is the same traversal a third time, and the census this chapter added is a fourth. Four uses of one traversal is a good thing to notice about a kernel.

ch18 takes the failure case seriously. A walk that stops is an opportunity, not an error — and a kernel that treats it that way can hand out memory it has not allocated, share pages until somebody writes to one, and keep a program's whole address space on disk.

18 · Page Faults as a Feature

CHAPTER HEADER

Target	xv6 — the teaching kernel under QEMU
Prerequisites	ch17
What it measures	Pages allocated and faults taken for one workload, under each of the kernel's two allocation policies: bench/results/faults-xv6.json

The question

What can a kernel do with a fault it expected?

[ch17](#) ended with a walk that stopped and a level to report it at. That was presented as a diagnosis, which is how a fault is usually introduced: something went wrong, and here is where. But a walk stops whenever an entry is absent, and **the kernel decides which entries are absent**. So a kernel can arrange to be told, by hardware, at the exact moment a particular address is touched — and the question stops being what went wrong and becomes what to do about it.

The material

The hardware will call you

A page fault is a callback, with three unusual properties.

It is **precise**: the kernel is given the address, in `stval`, not merely told that something happened. It is **synchronous**: it happens at the instruction that touched the address and not at some convenient later point. And it is **unavoidable**: the program cannot opt out, because it does not know it is happening.

No other mechanism in the machine has all three. A timer interrupt is not precise. A system call is not unavoidable — the program has to make it. This is the only way to have code run exactly when a particular byte is used, and everything in this chapter is built on that one property.

The instruction runs again

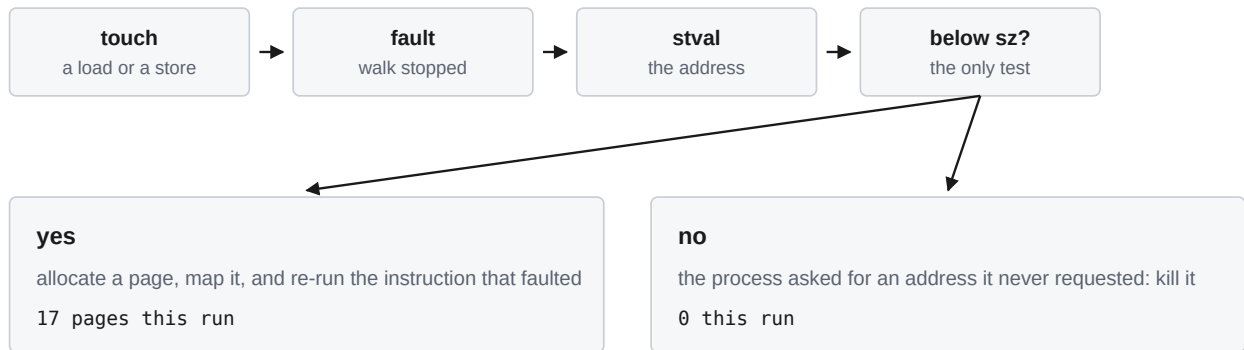
The difference from [ch16](#) is one line of the kernel.

After a system call, `usertrap` advances the saved program counter past the `ecall` before returning, because the call has been made and the program should carry on with the next instruction. After a page fault it does not. The saved program counter still points at the load or store that faulted, so returning from the handler **runs that instruction again**.

That is exactly what you want. The handler does not emulate the access, does not need to know what the instruction was, and does not have to put a value anywhere. It makes the address work and returns, and the hardware does the rest. A fault handler is therefore allowed to be ignorant of almost everything about the program it is rescuing.

A page fault is a question the kernel gets to answer

The hardware supplies the address. What happens next is entirely policy.



Unlike a system call, the faulting instruction runs again.

Change the test and you get a different feature from the same hook: copy-on-write, a guard page, a page fetched from disk. This chapter measures one of them. The others are named in the text and not measured, which is not the same as being free.

What the kernel does with a fault, and the single question that decides it.

One test, and the one next to it

xv6's handler asks whether the faulting address is below the process's size. Below it, the process asked for this memory and has not touched it yet, so the right answer is to allocate a page, map it, and return. At or above it, the process is using an address it never requested, and the right answer is to kill it.

A second test sits beside it, and it is the one that gets left out. If the page is *already* mapped, the fault cannot be a first touch — the entry is there and valid, so the walk did not stop for lack of one. It stopped because a permission was refused. Allocating a fresh page for that case would map a blank page over one the program was in the middle of using, and the program would not crash; it would get the wrong answer. Problem 18.2 is about exactly these edges.

Both policies are already here

Unusually, **this kernel already implements both allocation policies**, and lets a program pick one per call. `sbrk`, the call that grows a process's memory, allocates the pages when you ask. `sbrklazy` increases the process's size and allocates nothing, leaving the faults to do it.

So this chapter adds no policy and changes no decision. The patch counts:

15-fault-census.patch

```
+census_page(int lazy)
+{
+  struct proc *p = myproc();
+
+  if (p == 0) // during boot there is nobody to charge it to
+    return;
+  if (lazy)
+    p->fc.lazy_pages++;
+  else
+    p->fc.eager_pages++;
+}
```

Per process, because the shell is allocating too and a global total would mostly be a fact about the shell. Recorded as the process exits, because by the time anybody can press a key the interesting process has gone.

The trade, counted

The workload asks for memory both ways and touches a known amount of it:

faultload.c

```
int main(void) {
    char *sparse_lazy = sbrklazy(SYSFS_SPARSE_PAGES * PGSIZE);
    touch(sparse_lazy, SYSFS_SPARSE_TOUCHED);

    char *dense_lazy = sbrklazy(SYSFS_DENSE_PAGES * PGSIZE);
    touch(dense_lazy, SYSFS_DENSE_PAGES);

    char *eager = sbrk(SYSFS_SPARSE_PAGES * PGSIZE);
    touch(eager, SYSFS_SPARSE_TOUCHED);
}
```

Build it into the kernel, boot, and run it:

```
./run faultload
```

Three phases, chosen to bracket the trade rather than to demonstrate a win. A large lazy request barely touched; a smaller lazy request touched in full; an eager request of the same size as the first. The two lazy requests appear in the table summed rather than separately, because the census counts per process and not per call.

What the workload did	Pages
Pages requested with <code>sbrklazy</code>	80
...of those, pages ever touched	17
...so pages never allocated at all	63
Entries into the kernel that cost	17
Pages requested with <code>sbrk</code>	64
...of those, pages allocated	64
Pages <code>exec</code> allocated before <code>main</code> ran	20

Conditions: target `xv6 · qemu virt (-machine virt -bios none) · QEMU emulator version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.18) · xv6-riscv @ 9e3161a · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · xv6's own CFLAGS · 2026-09-17 · Source: bench/results/faults-xv6.json, code hash 89e452857447d851.`

Read the first four rows as one sentence. Most of what was asked for lazily was never allocated, and the pages that were allocated each cost one entry into the kernel — the whole of [ch16](#)'s trap path, plus a walk, plus an allocation, for every one of them.

That is the exchange rate, and the half usually left out is the price. Laziness is nearly always described by its saving, and the saving here is real and large. The price is real too: a number of kernel entries equal to the number of pages the program actually uses. Which of those matters depends entirely on the ratio between them, and that ratio is a property of the program rather than of the policy.

The last row is a different kind of fact: before `main` ran at all, `exec` had already allocated pages for the program's text, its data and its stack. Every process pays that, and it is the baseline everything else in the table sits on top of.

Which fault, and what this kernel does about it

The hardware does not report “a page fault”. It reports a load fault, a store fault or an instruction-fetch fault, as three different causes.

Cause	scause	Count
Load page fault	13	0
Store page fault	15	17
Faults the handler declined	—	0

Conditions: target xv6 · qemu virt (-machine virt -bios none) · QEMU emulator version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.18) · xv6-riscv @ 9e3161a · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · xv6's own CFLAGS · 2026-09-17 · Source: bench/results/faults-xv6.json, code hash 89e452857447d851.

Every page in this run arrived on a store, because the workload writes to each page it touches. A program that read a lazily-allocated page before writing it would produce load faults instead, for the same pages.

And this kernel does nothing with the difference: both causes are handled identically and get a readable, writable page. That is a real observation about xv6 rather than a gap in the measurement — the distinction exists in the hardware, and using it is what copy-on-write is. A kernel that shares a page between two processes marks it read-only in both, at which point a load fault cannot happen and a store fault means *somebody is about to modify a page they are sharing*. Same hook, same handler, one more test.

What laziness costs that is not faults

There is a second price, and it is not pages or faults: laziness changes when a program finds out it has run out of memory.

An eager request that cannot be satisfied fails at the call. The program gets a return value it can test, and can do something sensible — free a cache, use a smaller buffer, report a clear error. A lazy request always succeeds, because nothing has been allocated and there is nothing to fail. The shortage is discovered later, by a store instruction, in the middle of code that has no idea it is allocating anything. There is no return value to check at that point, and nothing the kernel can do but kill the process.

So laziness moves a failure from a place where the program can handle it to a place where it cannot. That is the whole reason an operating system which *overcommits* — promises more memory than it has — needs a policy for choosing something to kill, and problem 18.3 is about being able to say precisely when each version finds out.

What we measured

Pages allocated and faults taken for one workload, under each of the kernel's two policies, with every quantity fixed by the program rather than observed to be steady. `faultload` prints what it asked for and what it touched; the kernel counts independently; the runner refuses to stamp a result in which the two disagree, refuses one latched from a different process, and refuses one in which the handler declined a fault — which would mean the workload went somewhere it never asked for, and that the numbers describe a bug rather than a policy.

What this cannot tell you

What a fault costs. Not a single duration appears above, and it could not honestly. A fault's cost is the trap path, the walk, the allocation and whatever the TLB and the caches make of the interruption — and this target models none of the last part. [ch29](#) prices the whole shape on hardware. Until then, the fault count above is a count of kernel entries and not a claim about time.

Whether laziness is worth it. The table gives the exchange rate for one workload, chosen to show both ends. Whether trading those pages for those faults is a good deal depends on how much memory costs relative to kernel entries on the machine in question, which is a question about hardware and not about kernels.

Copy-on-write, demand paging — a page fetched from disk when first touched — and the rest. The chapter names them and measures none of them, which is deliberate and is not the same as their being free. Each is this hook with a different test, and each has its own exchange rate that this chapter has not measured. Naming a mechanism and pricing it are different things, and the book tries hard not to let the first pass for the second.

Anything about fragmentation. Every page here is the same size and comes from a free list that does not care which one it hands out. A system where physical contiguity matters — for large pages, or for a device doing its own addressing — has an allocation problem this one does not, and no measurement here would notice.

Problems

Three, in `tests/page_faults_as_a_feature/policy.c`. They are the three decisions the handler is really making, taken out of the kernel so that being right about them is separable from getting a kernel to boot. Neither this chapter's patch nor anything in `sysfs/` answers any of them: the patch counts and decides nothing.

18.1 — How many faults will these accesses cause? You are given runs of bytes a program touches and asked for the number of first-touch faults.

A program thinks in bytes and the machine charges pages, which is the trap the whole chapter rests on. Two accesses in one page cost one fault; an access of two bytes can cost two. The runs overlap, repeat and arrive in no order.

```
python3 -m pytest tests/page_faults_as_a_feature/test_problem_1_faults.py
```

18.2 — What should the handler do about this fault? Allocate, kill, or refuse to have an opinion. Nine cases, and three of them are the ones a handler written from the happy path gets wrong: the exact boundary, a page that is already mapped, and a cause that is not a page fault at all.

```
python3 -m pytest tests/page_faults_as_a_feature/test_problem_2_action.py
```

18.3 — When does a program that asks for too much find out? For each policy, say where the shortage becomes visible: at the request, at the touch, or never.

This is the section above, turned into a function. Getting it right means you can predict which of two programs — identical apart from one argument — dies in a way its author can do something about.

```
python3 -m pytest tests/page_faults_as_a_feature/test_problem_3_observed.py
```

Where to go next

The RISC-V privileged specification RISC-V INTERNATIONAL (N.D.) lists the fault causes and says exactly what the hardware guarantees about `stval` and about the state at the point of a fault. The guarantee that the faulting instruction has had no effect is the one worth reading carefully; it is what makes re-running it safe, and it is not free for the hardware to provide.

xv6's `vmfault` in `kernel/vm.c` COX ET AL. (N.D.) is the handler this chapter measured, and it is shorter than this section. Read it beside `sys_sbrk` in `kernel/sysproc.c`, which is where the two policies are chosen between — the whole difference is one branch.

[ch19](#) stays with the same mechanism and changes what raises it. A page fault is the CPU interrupting itself about something it was doing; a device interrupt is somebody else entirely, with no relationship to the instruction that happens to be running, and that difference turns out to matter more than it sounds.

19 · Interrupts and Drivers

CHAPTER HEADER

Target	xv6 — the teaching kernel under QEMU
Prerequisites	ch18
What it measures	What a fixed amount of I/O costs in interrupts, for the device where that question has an answer: bench/results/interrupts-xv6.json

The question

How does a device get the CPU's attention, and what does the CPU do about it?

[ch18](#) was about the CPU interrupting itself. A page fault happens *because of* the instruction that is running: it is caused by that instruction, it is reported at that instruction, and returning re-runs it. Everything about it is attached to the program it happened to.

A device interrupt has none of those properties. It arrives because something outside the processor finished doing something, at a moment decided by that thing, while the processor was in the middle of work with no relationship to it at all. The instruction it lands between is arbitrary. The process it lands on is very probably not the one that asked.

That difference is why device drivers are shaped the way they are, and this chapter asks which of its consequences can be counted.

The material

Getting attention costs someone else's time

The machine has one path in. A device raises a line, the interrupt controller — on this machine the PLIC, the block that decides which device may interrupt which hart — picks which device is allowed to speak, and the processor takes exactly the trap path [ch16](#) counted: the same thirty-odd register saves, the same page-table switch, the same restoration on the way out.

None of that work is the device's. It is charged to whichever process was running, which had nothing to do with the I/O and is not consulted. A process that performs no I/O at all still pays for every interrupt that arrives while it holds a core, and it pays in exactly the currency [ch16](#) measured.

So the obvious question is how many of these there are. One kind of device has an answer and the other does not.

A workload that decides its own numbers

intrload.c

```
int main(void) {
    /* Console: one write of a known number of characters, on one line so the transcript stays
     * readable. Every one of them has to reach a device that takes them one at a time. */
    static char line[SYSFS_INTRLOAD_CHARS + 1];
    for (int i = 0; i < SYSFS_INTRLOAD_CHARS; i++)
        line[i] = '.';
    write(1, line, SYSFS_INTRLOAD_CHARS);
    write(1, "\n", 1);

    /* Disk: a file written and read back, so the request count is decided here and not by a
     * cache's opinion of what is already in memory. */
    for (int i = 0; i < BLOCK; i++)
        block[i] = (char)i;

    int fd = open("intrload.dat", O_CREATE | O_WRONLY);
    for (int i = 0; i < SYSFS_INTRLOAD_BLOCKS; i++)
        write(fd, block, BLOCK);
    close(fd);

    fd = open("intrload.dat", O_RDONLY);
    for (int i = 0; i < SYSFS_INTRLOAD_BLOCKS; i++)
        read(fd, block, BLOCK);
    close(fd);
    unlink("intrload.dat");
}
```

Build it into the kernel, boot, and run it:

```
./run intrload
```

Two devices, deliberately. The console takes characters one at a time; the disk takes blocks. Both amounts are constants this program chose, so anything that varies between runs is varying for a reason that is not the workload.

The kernel counts interrupts by source, and counts two extra facts about the console driver:

16-interrupt-census.patch

```
+uartnote(int what, uint64 n)
+{
+  switch (what) {
+  case 0: icensus.tx_wakeups += n; break;
+  case 1: icensus.rx_chars += n; break;
+  case 2: icensus.chars_written += n; break;
+  default: icensus.write_sleeps += n; break;
+  }
+}
+
```

Three interrupt sources, one countable

Three sources, and only one countable answer

The work was fixed. Whether the interrupt count was fixed too depends on the device.

Disk one completion per request ----- recorded 319 interrupts, every run	Console “ready for more”, whenever that is ----- not recorded a different number every run	Timer once per tick of elapsed time ----- not recorded a fact about the host, not the guest
--	--	---

A block request is completed once. “Ready for more” is not a unit of anything.

Same image, same workload: the disk count was identical on every run and the console count was not, over a spread of nearly a third. So one of these is a property of the work and the other is a property of the afternoon, and only one of them belongs in a table.

Three sources. Same workload, repeated: one count came back identical every time and two did not.

What happened	Count
Block operations the workload performed	64
Interrupts the disk raised	319
Characters the workload asked to be written	513
Times it had to stop and wait for the device	0
Characters arriving from the keyboard	10

Conditions: target `xv6` · qemu virt (-machine virt -bios none) · QEMU emulator version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.18) · xv6-riscv @ 9e3161a · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · xv6's own CFLAGS · 2026-09-17 · Source: [bench/results/interrupts-xv6.json](#), code hash `e20981b7aa1ab1c9`.

The disk count is in the table. The console count is not, and its absence is the finding rather than an omission.

A second row is missing too, and it took longer to find. The table used to carry a count of how many characters the writing process handed to the device. It reads like a companion to the row above it and is not one: the counter behind it is incremented for every character *the kernel* sends, so it counts the boot log, the shell’s prompt and the echo of what was typed as well as the workload’s output. It reported more characters than the workload had asked for, which is what gave it away, and it came back one different between identical runs often enough to fail the check that re-runs this measurement. The same argument that keeps the console’s interrupt count out of the table keeps that row out of it.

Run the identical workload five times and the disk raises the same number of interrupts every time. Run it five times and the console raises a different number every time. Nothing about the workload changed between those runs, so the console’s count is not telling you about the workload.

The reason is in what the two devices are saying. **A block device interrupts to say that a request is finished.**

Requests are discrete, the workload decides how many there are, and each is completed exactly once — so the count follows. **A character device interrupts to say it is ready for more.** That is not a unit of anything. How many times

a transmitter announces its readiness during a burst of output depends on how the output and the announcements interleave, which depends on timing, and timing inside an emulator is a property of the laptop.

This is [ch16](#)'s rule again, from a different direction. There the uncountable thing was the timer, and the reason was obviously about elapsed time. Here two devices are doing what looks like the same job, and only one of them is countable — which is a much better demonstration that the question is about what an event *means* and not about which peripheral raised it.

The zero

One number in the table is zero, and it says more about the emulator than about the driver.

xv6's console driver is written to be asynchronous. A process writing to the console hands a character to the transmitter if it is idle, and otherwise **sleeps**, to be woken by the interrupt that says the transmitter has caught up. That machinery — the sleep, the *wait channel* the sleeper is parked on, the wakeup in the handler — is the reason the driver is split into a part that runs in the process and a part that runs in the interrupt, and it is the standard explanation for why device drivers have two halves.

In this run it never once engaged. Every character was handed over by the writing process itself, and the count of times anybody had to stop and wait is zero.

That is not a fault in the driver or in the measurement. It is the emulator: QEMU's transmitter is always ready, because there is no serial line and nothing is being clocked out at a fixed rate. The design exists to cope with a device that is slower than the program, and this target does not have one.

So this chapter can show you the structure and cannot show you the reason for it. The split into two halves is visible in the code, the interrupt that would do the waking is counted, and the thing that makes any of it necessary — a device that makes a program wait — is absent from the machine the chapter runs on. [ch29](#) is on hardware where it is not.

What the image has to do with it

One more dependency, because it caught this chapter out and would otherwise catch a reader out.

The disk figure is reproducible, but it is reproducible *for a filesystem image*. It counts the block operations xv6's filesystem performs, and how many those are depends on where the blocks are — which depends on what `mkfs`, the program that builds the image, laid down, which depends on which programs this book has added to xv6. Add another one in a later chapter and the number moves, for a reason that has nothing to do with interrupts.

Nothing in the book's stamping scheme covers the filesystem image, because until now nothing had needed it to. The result records the image's digest — a hash of its contents — so the dependency is visible, and CI regenerates the measurement on every push, so a change is reported rather than absorbed.

What we measured

Interrupts raised by the disk for a fixed number of block operations, and what the console driver did with a fixed number of characters. The census prints two counts this book does not record; the result names them rather than quietly dropping them, and problem 19.3 asks the reader to work out which they are before being told.

What this cannot tell you

What an interrupt costs. No duration appears above. The cost of an interrupt is the trap path plus the handler plus whatever the interruption does to the caches and the pipeline of the innocent process that was running — and this target models none of the last part. [ch29](#) prices it.

How often the timer really fires. The census counts timer interrupts and the book does not print the number, for [ch16](#)'s reason. The count is *steady* here over repeated runs and still must not be published: steady on one machine for one short workload is not the same as determined by the workload, and the difference is exactly the mistake this book is trying not to make.

Why the driver is split in two. The structure is here and the pressure that produced it is not, because the emulated transmitter is never busy. A reader who wants to see the asynchronous path do something has to run this on hardware, which is [Part V](#).

Anything about interrupt latency, priority or affinity. The PLIC can be told which core should take which interrupt and at what priority; xv6 uses almost none of that, and none of it shows up in a count. On a machine where a device interrupt can land on the core running your benchmark, those settings are the difference between a measurement and a puzzle, and [ch30](#) has to care.

Problems

Three. The first two are in `tests/interrupts_and_drivers/console.c`; the third is a Python stub you fill in with an opinion and are graded against the machine.

19.1 — How many characters does the console lose? Given a string of arrivals and drains, and a buffer of xv6's own size, say how many are dropped.

A handler cannot block, cannot allocate and has nobody to return an error to, so a character arriving at a full buffer is simply gone and nothing anywhere is told. This is why a terminal loses characters when you paste into it, and it is not a bug in anything.

```
python3 -m pytest tests/interrupts_and_drivers/test_problem_1_lost.py
```

19.2 — How many interrupts will this cost? Given a number of operations and how many the device reports at once, predict the count — and return “undecidable” for the device where the workload does not determine it.

The last case is the only one that matters. Being able to say *this question has no answer* is worth more than any number, and it is a skill this book keeps asking for.

```
python3 -m pytest tests/interrupts_and_drivers/test_problem_2_interrupts.py
```

19.3 — Which of these counts is a property of the workload? Six counters; mark each reproducible or not, before running anything.

You are not being graded against what this book decided. The test boots xv6 several times and compares the censuses, so a counter you called reproducible had better come back the same every time. Where the experiment cannot settle one — a varying counter that happened not to vary — you are told so rather than failed, because a test that occasionally fails a correct answer is worse than one that occasionally proves less than it hoped to.

```
python3 -m pytest tests/interrupts_and_drivers/test_problem_3_attribute.py
```

Where to go next

The RISC-V privileged specification RISC-V INTERNATIONAL (N.D.) defines the interrupt causes and the rules about when a hart will take one; the PLIC's own specification defines claiming and completing, which is the handshake `devintr` performs twice per interrupt.

xv6's `kernel/uart.c` and `kernel/plic.c` COX *ET AL.* (N.D.) are both short enough to read at a sitting. Read `uartwrite` beside `uartintr` and find the sleep and the wakeup that this chapter measured zero of — the code is written for a device that makes you wait, and seeing it never wait is the most useful thing this target can show you about it.

ch20 is about the problem this chapter has been carefully stepping around. A handler and a process share a buffer; one of them can start at any instruction of the other; and the counters in this chapter's own patch are deliberately unlocked, which is a decision that needs defending.

20 · Locks and Memory Ordering

CHAPTER HEADER

Target	xv6 — the teaching kernel under QEMU
Prerequisites	ch19
What it measures	What a lock is made of, in instructions: <code>bench/results/locks-xv6.json</code> , <code>bench/results/ordering-riscv64.json</code>

The question

What breaks when two harts touch the same memory, and what is the minimum fix?

[ch19](#) left a question unanswered. Its census counters are incremented from interrupt handlers and from process context, on three harts, with no lock anywhere — and the patch says so in a comment that amounts to *this is fine*. Either that is defensible or the book has been printing numbers from a data structure that races, so this chapter has to settle it.

The material

A race is three instructions

The sentence “two threads incrementing a counter can lose an update” is usually offered as a fact about threads. It is not. It is a fact about what an increment becomes:

sysfs_bump_plain compiled for **riscv64** at **-O2** — the **xv6** target’s instruction set, 4 instructions.

```
0000000000000000 <sysfs_bump_plain>:
0:   ld      a5,0(a0)
2:   addi    a5,a5,1
4:   sd      a5,0(a0)
6:   ret
```

Conditions: target **xv6** · **riscv64** cross compiler, any machine — nothing here was executed · **riscv64-linux-gnu-gcc** (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · **-O2 -Wall -Wextra -march=rv64gc -mabi=lp64d -c** · 2026-09-17 · Source: `bench/results/ordering-riscv64.json`, code hash `c4735c450b3a7f51`.

sysfs_bump_plain compiled for **aarch64** at **-O2** — the **host** target’s instruction set, 4 instructions.

```
0000000000000000 <sysfs_bump_plain>:
0:   ldr     x1, [x0]
4:   add     x1, x1, #0x1
8:   str     x1, [x0]
c:   ret
```

Conditions: target **host** · **aarch64** cross compiler, any machine — nothing here was executed · **aarch64-linux-gnu-gcc** (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · **-O2 -Wall -Wextra -c** · 2026-09-17 · Source: `bench/results/ordering-aarch64.json`, code hash `f73072083d1d21ee`.

Three instructions: load, add, store. Nothing in the machine holds them together. Another hart can load between your load and your store, and then both of you store the same value — one increment where there should have been two. An interrupt can land in the same gap on a single hart, which is why this is not only a multiprocessor problem.

That is the whole mechanism. Everything else in this chapter is about closing that gap. Both architectures produce the same three instructions, so the problem does not belong to an instruction set.

One instruction instead of three

The minimum fix is to ask the machine for an increment it will not interrupt:

sysfs_bump_relaxed compiled for riscv64 at -O2 — the `xv6` target's instruction set, 3 instructions.

```
0000000000000008 <sysfs_bump_relaxed>:
 8:  li    a5,1
 a:  amoadd.d    zero,a5,(a0)
 e:  ret
```

Conditions: target `xv6` · riscv64 cross compiler, any machine — nothing here was executed · `riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · -O2 -Wall -Wextra -march=rv64gc -mabi=lp64d -c · 2026-09-17 · Source: bench/results/ordering-riscv64.json, code hash c4735c450b3a7f51.`

sysfs_bump_relaxed compiled for aarch64 at -O2 — the `host` target's instruction set, 7 instructions.

```
0000000000000010 <sysfs_bump_relaxed>:
10:  stp    x29, x30, [sp, #-16]!
14:  mov    x1, x0
18:  mov    x0, #0x1                // #1
1c:  mov    x29, sp
20:  bl     0 <__aarch64_ldadd8_relax>
24:  ldp    x29, x30, [sp], #16
28:  ret
```

Conditions: target `host` · aarch64 cross compiler, any machine — nothing here was executed · `aarch64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · -O2 -Wall -Wextra -c · 2026-09-17 · Source: bench/results/ordering-aarch64.json, code hash f73072083d1d21ee.`

On RISC-V that is `amoadd.d` — atomic memory operation, add, doubleword. One instruction, so there is no gap to land in.

The AArch64 version does not get one instruction. Instead there is a **call** to a helper with a name the programmer never wrote. AArch64 gained single-instruction atomics in a later revision of the architecture, so a compiler that does not know which chip it is building for cannot use them directly; it emits a call to a routine that decides at run time. Whether the same C is one instruction or a function call is therefore not a property of the source or even of the architecture, but of what the compiler was told to assume.

[ch26](#) is where telling the compiler what to assume matters.

The ordering is not the atomicity

Two things are commonly rolled together and are separate. An operation can be *indivisible* and still be reordered against its neighbours.

sysfs_bump_ordered compiled for riscv64 at -02 — the xv6 target’s instruction set, 3 instructions.

```
0000000000000010 <sysfs_bump_ordered>:
10:  li      a5,1
12:  amodd.d.aqrL    zero,a5,(a0)
16:  ret
```

Conditions: target `xv6` · riscv64 cross compiler, any machine — nothing here was executed · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · -02 -Wall -Wextra -march=rv64gc -mabi=lp64d -c · 2026-09-17 · Source: `bench/results/ordering-riscv64.json`, code hash `c4735c450b3a7f51`.

That is the same increment with the strongest ordering the language offers, and it is the same single instruction with two letters added: `.aqrL`, *acquire* and *release* — the two halves of an ordering promise, that nothing after this instruction may be moved before it and nothing before it may be moved after. **Ordering costs no extra instruction here at all** — the atomic instruction carries its own ordering, and what would be a separate fence on another machine is two bits of an opcode on this one.

So “an atomic operation” and “an ordered operation” are different requests, and RISC-V makes the difference visible: the spelling changes and the instruction count does not.

Publishing something

The reason ordering matters separately from atomicity is the pattern every lock is built out of: write something, then announce that you have written it.

sysfs_publish compiled for riscv64 at -02 — the xv6 target’s instruction set, 5 instructions.

```
0000000000000018 <sysfs_publish>:
18:  sd      a2,0(a0)
1a:  li      a5,1
1c:  fence   rw,w
20:  sw      a5,0(a1)
22:  ret
```

Conditions: target `xv6` · riscv64 cross compiler, any machine — nothing here was executed · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · -02 -Wall -Wextra -march=rv64gc -mabi=lp64d -c · 2026-09-17 · Source: `bench/results/ordering-riscv64.json`, code hash `c4735c450b3a7f51`.

sysfs_publish compiled for aarch64 at -02 — the host target’s instruction set, 4 instructions.

```
0000000000000050 <sysfs_publish>:
50:  str     x2, [x0]
54:  mov     w0, #0x1           // #1
58:  stlr    w0, [x1]
5c:  ret
```

Conditions: target `host` · aarch64 cross compiler, any machine — nothing here was executed · aarch64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · -02 -Wall -Wextra -c · 2026-09-17 · Source: `bench/results/ordering-aarch64.json`, code hash `f73072083d1d21ee`.

The RISC-V version writes the value, then `fence rw,w`, then writes the flag. The fence is doing one job: it forbids the flag's store from becoming visible before the value's. Without it a reader on another hart could see the flag set and the value stale, and every instruction involved would have executed exactly as written.

AArch64 spells the same requirement as `stlr` — a store with release ordering built in, one instruction instead of two. Same semantics, same guarantee, different arithmetic. A reader who learned that “a barrier is an instruction you put between things” would not recognise the second version as containing a barrier at all.

Compare it with the same code written without the ordering and the difference is one instruction on RISC-V and none on AArch64 — where the plain version uses `str` and the ordered one `stlr`, so the cost is not an instruction but a choice of instruction.

What xv6's lock is made of

Now the kernel's own, disassembled as built rather than reimplemented for a book:

	Instructions	Atomic	Fences	CSR writes
<code>acquire</code>	25	1	0	0
<code>pop_off</code>	27	0	0	2
<code>push_off</code>	23	0	0	1
<code>release</code>	20	0	1	0

Conditions: target `xv6` · riscv64 cross compiler, any machine — nothing here was executed · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · `xv6's own CFLAGS` · 2026-09-17 · Source: `bench/results/locks-xv6.json`, code hash `5280bab00cd231da`.

Three things in that table are worth reading carefully.

One instruction in `acquire` does the mutual exclusion. The atomic column says one, and the instruction is `amoswap.w.aq` — swap a one in, and if what came out was already one, somebody else holds it, so go round again. Everything else in those twenty-five instructions is xv6 checking that the caller is not already holding the lock and panicking if it is. The lock is one instruction; the rest is the kernel protecting you from yourself.

`release` contains no atomic instruction at all. Releasing is an ordinary store — there is nobody to compete with, because you hold the lock. What it does contain is the fence, and that fence is the entire reason the lock works: it is what stops the work you did inside the critical section from becoming visible *after* the store that lets the next hart in.

Turning interrupts off costs more than the lock does. `push_off` and `pop_off` are, between them, more instructions than `acquire` and `release` together. They exist because a lock held across an interrupt on the same hart would *deadlock* against its own handler — the handler spinning for a lock that the code it interrupted can never release — and they are the price of that safety rather than of the mutual exclusion.

So were those unlocked counters right?

Back to the unlocked counters.

ch19's counters are incremented without a lock from several harts and from interrupt context. On the evidence above, that increment is three instructions and updates can certainly be lost. **The patch is therefore wrong in the sense that its counts can be short, and right in the sense that this is the correct trade.**

The argument is not “it is only a counter”. It is that the alternative is worse in a way that matters to this book specifically. A lock on the trap path is itself work on the trap path: `push_off`, an atomic, a fence, `pop_off` — on every entry into the kernel, added to the thing being measured. The measurement would change what it measures, and it would change it by an amount of the same order as some of the things this book wants to count.

So the census is deliberately approximate where approximation is cheap, and the results that matter — the ones in [ch18](#) and [ch19](#) — are the ones fixed by the workload rather than accumulated under a race. That is a decision rather than an oversight, and it is the kind of decision that should be written down where a reader can disagree with it.

What we measured

Instruction counts, and what kinds of instruction they are, for the lock primitives in the kernel as built and for five ways of incrementing a counter on two architectures. Nothing here was executed and nothing was timed.

The plan for this chapter promised contention counts per lock, and said the interleavings would be deterministic under QEMU. Neither survived contact. Contention is a statement about how long one hart made another wait, which is a duration this target cannot supply — and the interleavings are not deterministic either: [ch19](#) had already found the console’s interrupt count varying between identical runs, on this same emulator. The chapter measures what is actually there instead, and this paragraph is here because a plan that turned out to be wrong is worth more to a reader than a plan quietly rewritten.

What this cannot tell you

What a lock costs. The whole of it. An uncontended acquire is one atomic instruction, and what that instruction costs depends on whether the cache line is already held exclusively, which depends on what the other cores have been doing. A contended one costs however long you waited. Both are questions about hardware, and [ch28](#) asks them on a machine that can answer.

Whether the fence is doing anything here. RVWMO — RISC-V’s memory model, the rules for which reorderings a hart may perform — permits the reorderings the fence forbids, and QEMU is entitled to but does not perform them: it executes each hart’s instructions in order. So every program in this chapter would behave identically with the fences removed, on this target, and would be broken on real hardware. That is an uncomfortable property of the instrument and the reason problem 20.2 asks about the model rather than about a run.

Anything about fairness. xv6’s lock is a plain spin: whoever’s swap happens to succeed gets in, and a hart can be unlucky indefinitely. Real kernels care, use queues and tickets, and pay for them. None of that is visible in an instruction count.

Sleep locks, which the chapter has not mentioned. xv6 has them, they are what [ch19](#)’s console driver used, and they are built on top of the spinlock plus the scheduler rather than beside it. [ch21](#) has the scheduler and can say what sleeping actually means.

Problems

Three, in `tests/locks_and_memory_ordering/locking.c`. The first is the only one in this book that is graded by running it under real concurrency, because a lock is the one thing that cannot be checked by reading it.

20.1 — Write a correct lock. Four threads take it eight hundred thousand times between them, incrementing an ordinary non-atomic counter inside. If the lock works the total is exact; the empty stub loses most of them.

You have `<stdatomic.h>` and not `pthread_mutex`. Two things to get right, and only the first is about exclusion: taking the lock must be one indivisible operation, and the ordering must be such that work done inside is visible to the next thread in. A correct exchange with the wrong memory order passes on a strongly ordered machine and protects nothing on a weak one.

```
python3 -m pytest tests/locks_and_memory_ordering/test_problem_1_lock.py
```

20.2 — Which reorderings does each barrier actually prevent? Eighteen cases. The four that matter are the ones where the annotation is on the operation it does not help — an acquire on the second access, a release on the first — and the ones where the two accesses touch the same address, which settles the question before any barrier is consulted.

```
python3 -m pytest tests/locks_and_memory_ordering/test_problem_2_reorder.py
```

20.3 — Do these two paths disagree about lock order? Given the locks each path takes in order, say whether there is a pair taken both ways round.

Deliberately not “will this hang”. A conflicting pair is forbidden whether or not you can build a schedule that hangs today, because tomorrow’s schedule is not yours to choose — and two paths can share every lock they use and be perfectly safe.

```
python3 -m pytest tests/locks_and_memory_ordering/test_problem_3_order.py
```

Where to go next

The RISC-V unprivileged specification RISC-V INTERNATIONAL (N.D.) has the A extension and the memory model in two separate chapters, which is the right way round: the atomics chapter tells you what `amoswap` does and the RVWMO chapter tells you what any of it means. Read the `fence` encoding and notice that `rw,w` is two four-bit fields — the fence you saw is one of two hundred and fifty-six.

xv6’s `kernel/spinlock.c` COX ET AL. (N.D.) is short and now contains nothing you have not seen the machine code for. The comments around the `__sync_synchronize()` calls say what the fences are for in the authors’ own words, which is a useful second opinion on this chapter’s.

[ch21](#) is the other half of what the console driver did. A process that cannot get on gives up the CPU, and something has to decide what runs instead — which needs a lock, taken across a context switch, in a way that ought to be impossible.

21 · Scheduling and Context Switches

CHAPTER HEADER

Target	xv6 — the teaching kernel under QEMU
Prerequisites	ch20
What it measures	What a context switch moves, and the one switch count a workload decides: <code>bench/results/switch-xv6.json</code>

The question

What exactly is saved, and what does it mean to say a thread “runs”?

[ch20](#) ended on something that ought to be impossible: a lock held across a context switch. A process gives up the CPU while holding its own lock, another hart picks it up later, and the release happens on a different core from the acquire. That is not a bug, it is the design, and understanding why needs a precise account of what a switch actually moves.

The material

A switch is a function call, and that is the whole saving

What a switch costs to arrange	Count
Registers a context switch saves	14
Registers a trap saves (ch16)	31
Bytes a switch moves, in and out	224
Instructions in <code>switch</code>	29

Conditions: target `xv6` · `qemu virt (-machine virt -bios none)` · QEMU emulator version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.18) · `xv6-riscv @ 9e3161a` · `riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0` · `xv6's own CFLAGS` · 2026-09-17 · Source: `bench/results/switch-xv6.json`, code hash `36029a44513eb33d`.

Read the first two rows against each other.

[ch16](#)’s trap path saves thirty-one registers. `switch` saves fourteen. Both are moving “the state of a thread” and one of them moves less than half as much, which looks like an optimisation and is not.

A trap is not a call and a switch is. The interrupted program of [ch16](#) agreed to nothing, so everything it might have been using has to be preserved. `switch` is reached by an ordinary `jal` from `sched`, the function a process calls to give up the CPU, which means [ch14](#)’s calling convention has already been obeyed on the way in: anything the caller cared about and the convention does not protect is already spilled to the caller’s stack. What is left for the switch to keep is exactly the callee-saved set, plus the return address and the stack pointer — fourteen registers, and the structure that holds them has fourteen fields.

So the cost of a switch is low for the same reason the cost of a trap is high, and neither number is a property of how well the code was written. Both follow from the calling convention: the switch can rely on it and the trap path cannot.

Where the thread goes

`switch` takes two pointers, saves into the first and restores from the second, and returns — but it returns somewhere else. The `ret` at the end reads the `ra` that was just loaded from the *other* context, so control arrives in whichever function last called `switch` on that side. A thread's identity, at the instruction level, is a saved stack pointer and a saved return address. Everything else about it is in memory that neither of those moves.

This is also the answer to [ch20](#)'s impossibility. The lock is held by a *process*, and the process's state lives in memory that both harts can reach. `switch` does not release anything and does not need to: the acquire and the release are operations on a shared word, and which core performs them is not part of the agreement. What would break is holding a lock across a switch *and* expecting interrupts to stay disabled on the hart you came from — which is why `push_off` and `pop_off` count nesting per CPU rather than per process, and why [ch20](#) found them costing more instructions than the lock.

Why did it switch?

Every switch out of a process goes through one function, so counting them is easy. Attributing them takes more, and this chapter's kernel patch records which of three reasons applied: the timer took the CPU away, the process is waiting for something, or the process is not coming back.

What the workload caused	Count
Children created and exited	16
Switches out of a process that is not coming back	17

Conditions: *target xv6 · qemu virt (-machine virt -bios none) · QEMU emulator version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.18) · xv6-riscv @ 9e3161a · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · xv6's own CFLAGS · 2026-09-17 · Source: [bench/results/switch-xv6.json](#), code hash [36029a44513eb33d](#).*

Only the last is a number the workload decides. A process that exits switches away exactly once, and the count is the children the workload created plus the workload itself.

The other two are declined, and by now the reason should be familiar: a count of how often the timer intervened is a statement about elapsed time, and a count of how often something waited depends on whether the thing it waited for had already happened. [ch19](#) established this for devices and it is the same argument.

One observation did not make the table. In this workload the timer **never** took the CPU away from anything: every switch was voluntary, because every process blocked or exited before its *slice* — the stretch of time the timer allows it before intervening — ran out. Preemption is what a scheduler is for, and a short-lived workload of blocking processes never needs it. Problem 21.3 is about what a policy decides when preemption does happen; running the census yourself and trying to make the timer intervene is a more instructive ten minutes than reading about it.

Sleeping is not a state of the CPU

[ch19](#)'s console driver slept, and the word has been unexplained ever since.

Sleeping is not the hardware doing anything. It is a process marking itself not-runnable, noting what it is waiting for, and calling `sched` — after which some other thread's registers are in the CPU and the sleeper exists only as a `struct context` in memory. Waking is the reverse: somebody marks it runnable again, and eventually the scheduler picks it up and `switches` back into it, and it returns from a function call it made an arbitrary time ago.

The dangerous part is the gap. If the sleeper checks a condition, finds it false, and then sleeps, a wakeup arriving in between hits a process that is not yet asleep, hits nothing, and is not repeated. That is a lost wakeup, it is the reason sleep is handed the lock that guards the condition rather than being called after releasing it, and problem 21.2 is about being able to spot it in an ordering rather than recognising it in a diagram.

What we measured

What `switch` moves, read out of the kernel as built, and the one switch count this workload fixes. Nothing was timed: what a switch *costs* is the registers, the cache lines they touch, and what the new thread finds missing when it starts running — and only the first of those is visible here. [ch29](#) measures the rest.

What this cannot tell you

What a switch costs. Two hundred and twenty-four bytes move; whether that is expensive depends entirely on whether they are in cache, and this target has no cache. The expensive part of a real context switch is usually not the registers at all — it is that the new thread arrives to find the caches and the TLB full of somebody else’s data. [ch29](#) is where that gets a number, and the number is not small.

Whether xv6’s scheduler is any good. It is a round-robin — each runnable process in turn — over a fixed array, chosen to be readable, and it is not trying to be anything else. Real schedulers care about fairness over time, about which core a thread last ran on, and about latency for threads that have just been woken; none of that is in a few dozen lines and none of it is visible in a switch count.

Anything about more threads than cores. This workload never had enough runnable processes to make the scheduler choose, which is why the timer never intervened. A measurement of scheduling needs contention for the CPU, contention is a question about time, and this chapter has spent its whole length explaining why that question goes to [Part V](#).

What `sched` does about the lock, exactly. The dance between `sched`, `scheduler` and `p->lock` is the subtlest stretch of code in xv6 and is worth reading with the invariants in front of you — they are written in the comments. This chapter has said what the switch moves; it has not tried to prove the locking correct.

Problems

Three, in `tests/scheduling_and_context_switches/scheduling.c`.

21.1 — Must the switch save this register? Given the register’s role under [ch14](#)’s convention and whether the value is still needed, say whether `swtch` itself has to preserve it.

A context switch is an ordinary call, so most of the answer is already in the calling convention. Getting it right is what makes the fourteen against thirty-one above obvious rather than surprising.

```
python3 -m pytest tests/scheduling_and_context_switches/test_problem_1_save.py
```

21.2 — Does the sleeper wake? Six event sequences; say which loses a wakeup. Two of them use exactly the same six events in different orders and have different answers, which is the entire lesson.

```
python3 -m pytest tests/scheduling_and_context_switches/test_problem_2_wakeup.py
```

21.3 — What order does each policy run them in? Three jobs, three policies, three different answers. First-come-first-served, shortest-job-first and round-robin with a quantum — a slice — of one, on burst lengths chosen so that no two policies agree.

Producing the orders is mechanical; the value is in noticing what each one optimises and what it gives up, and that shortest-job-first needs to know something the kernel cannot generally find out.

```
python3 -m pytest tests/scheduling_and_context_switches/test_problem_3_policy.py
```

Where to go next

xv6's `kernel/proc.c` COX ET AL. (N.D.) — `scheduler`, `sched`, `yield`, `sleep` and `wakeup` are short between them and are now readable in full. The comment above `sched` stating the invariants it requires is the densest paragraph in the kernel; read it after problem 21.2 and it will say something it would not have said before.

`kernel/switch.S` is the fourteen stores and fourteen loads this chapter counted, and was the code [ch14](#) sent you to look at without explaining. It has not changed; you have.

[ch22](#) is the last of [Part IV](#)'s xv6 chapters and the one with a disk in it. A file system has to survive being interrupted at any instruction by a power cut, which is a stronger requirement than anything a lock provides, and it is met by writing things down in an order chosen so that every prefix of the sequence is survivable.

22 · The File System

CHAPTER HEADER

Target	xv6 — the teaching kernel under QEMU
Prerequisites	ch21
What it measures	What one byte costs the disk, as a difference between two runs: <code>bench/results/blocks-xv6.json</code>

The question

What has to be true on the disk for a crash mid-write to be survivable?

Every mechanism in [Part IV](#) so far has assumed the machine keeps running. A lock is held until it is released; a page is mapped until it is unmapped; a process runs until it is switched away. A file system cannot assume any of that. The power can fail between any two instructions, and what is on the platter afterwards is whatever had reached it — so the only tool available is *the order things are written in*, and the whole design follows from that one constraint.

The material

One byte

Start at the end, with what it costs.

	No bytes written	One byte written	The byte alone
Blocks read from the disk	0	3	3
Blocks written to the disk	16	26	10
Blocks entered in the log	6	10	4
Transactions committed	5	6	1

Conditions: target `xv6` · `qemu virt (-machine virt -bios none)` · QEMU emulator version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.18) · `xv6-riscv @ 9e3161a` · `riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0` · `xv6's own CFLAGS` · 2026-09-17 · Source: `bench/results/blocks-xv6.json`, code hash `92b201238488aa7b`.

The two runs are the same program making the same calls, differing in a single write. The last column is therefore the byte's and nothing else: the file is created in both, removed in both, and the shell forks and execs the program in both.

The same cost in bytes rather than blocks:

	Bytes
Bytes the program wrote	1
Bytes the disk was given, because of that byte	5120
Bytes the disk was given for a file with nothing in it	8192

Conditions: target `xv6` · `qemu virt (-machine virt -bios none)` · QEMU emulator version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.18) · `xv6-riscv @ 9e3161a` · `riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0` · `xv6's own CFLAGS` · 2026-09-17 · Source: `bench/results/blocks-xv6.json`, code hash `92b201238488aa7b`.

That is the *amplification* — how much the disk was given for the one byte the program wrote — and it is not a rounding error. Nor is it waste: every one of those writes is doing something, and the rest of this chapter says what.

Notice the last row of that second table as well. A file with nothing whatever in it, created and immediately removed, costs more disk traffic than the byte does. Most of what a file system does is bookkeeping, and the data is the small part.

Why every block is written twice

Take the last column apart. The byte caused four blocks to be modified and ten writes to reach the disk, and those two numbers are related by the design rather than by accident.

A committed transaction writes each of its blocks **twice**. Once into the log — a reserved run of blocks that is not where the data belongs — and then again to the place it actually belongs. Around those it writes the log's header twice: once to say what the transaction contains, and once afterwards to say that it no longer contains anything.

So the writes a transaction costs are twice its blocks plus two, and the check in `bench/run_blocks.py` refuses to stamp a result where that no longer holds. Problem 22.1 is that formula, and it is worth deriving before reading on.

Doubling the traffic looks like a strange thing to do on purpose. It is the only thing that works.

The commit is one write

A crash can happen between any two block writes. For the file system to be recoverable, **every prefix of what it writes must leave a state that recovery can turn into a correct one.**

Consider updating a file in place, with no log. Write the data block, then the *inode* — the on-disk record that describes a file: its size, and where its blocks are — so that it records the new size. A crash between them leaves a file whose size says one thing and whose contents say another, and nothing on the disk records which was intended. There is no recovery procedure, because there is no information to recover from.

The log removes the ambiguity by making one single write the moment everything becomes true:

1. The modified blocks are copied into the log. The header does not mention them, so as far as anything is concerned they are scratch.
2. **The header is written, naming them.** The transaction is now real.
3. The blocks are copied from the log to their homes.
4. The header is cleared. The transaction is over.

A crash before step 2 leaves blocks in the log that nothing points at, and recovery ignores them. A crash after step 2 and before step 4 leaves a header naming blocks, and recovery copies them to their homes — which is safe whether or not step 3 had already done it, because copying the same block to the same place twice is the same as copying it once.

There is no moment at which half a transaction is visible. That is what the log is for, and it is bought with exactly one thing: writing the header after the log and before the homes. Problem 22.3 asks you to check the other five orderings, and the ones that fail are the ones that look reasonable.

Idempotence is doing the real work

Step 3 is harmless to repeat, and that property is doing more work than it appears to.

Recovery does not know how far the original run got. It cannot know: the only evidence is the header, and the header says what the transaction *contains*, not what has been done about it. So recovery replays everything the header names, every time, and this is correct only because replaying an already-installed block changes nothing.

That is why the log holds **blocks and not changes**. “Set byte 40 of block 6 to `x`” is not safe to apply twice if it is expressed as an increment; “block 6 now looks like this” is safe to apply any number of times. Choosing the *idempotent* representation — the one that comes out the same however many times it is applied — is what lets recovery work without knowing anything about history.

It also explains the amplification. A log of changes would be far smaller than a log of blocks, and would not survive being replayed twice.

The seven layers, and where the traffic comes from

Under `write` there are seven things, and each one is why one of the numbers above is what it is: the file descriptor that names an open file, the inode that describes a file, the directory that maps names to inodes, the block allocator that finds free space, the log that makes a group of updates atomic, the buffer cache that keeps blocks in memory and decides when they reach the disk, and the disk driver from [ch19](#).

The log took four block-writes for one byte: the data block itself, the inode recording that the file is now one byte long and where that byte is, the *bitmap* — the allocator’s one-bit-per-block record of which blocks are free — recording that the data block is no longer free, and — once the program deletes the file at the end — the bitmap again, giving the block back. The traffic is not the file system being careless. It is a byte requiring three facts to be made true at once, by a design in which they become true together or not at all, and then undone the same way.

The cache is why the reads are so few

The read counts are nearly zero. Creating and deleting a whole file read **nothing** from the disk, and writing a byte read three blocks.

That is the buffer cache from the list above, doing the only thing a cache does. A block already in memory is not fetched, and a short workload touches the same handful of blocks — the superblock, which says where everything else on the disk is, the log header, the inode block, the bitmap — over and over. Reads are the operation a cache can eliminate entirely; writes are the operation it can only delay, and a log is a design that deliberately declines to delay them very much.

What we measured

Blocks reaching the disk for two runs of one program that differ by a single `write` call, and the difference between them. Nothing was timed: what a block write *costs* is a property of a disk, and the disk here is a file on a laptop.

The measurement is a difference because a single run would have charged the byte for the shell forking, for the kernel reading the program, and for the directory lookup that found it — all of which happen whether a byte is written or not, and all of which are larger than the thing being measured.

What this cannot tell you

What any of it costs in time. A real disk charges enormously more for a write that has to move a head than for one that does not, and orders of magnitude less for both if it is solid state. The amplification factor is the same on every disk; what it means is not, and a design that is obviously wasteful on one is obviously correct on another.

Whether the double write is worth it. The chapter has shown what the log buys and what it costs, and has not compared it with the alternatives — soft updates, copy-on-write trees, journalling only metadata — each of which makes a different trade and none of which is visible here.

Anything about the disk lying. The whole argument rests on writes reaching the platter in the order they were issued, and real drives have caches that reorder and acknowledge early. A file system that assumes otherwise is correct on paper and loses data in practice, and the machinery for saying “this one, really, now” is a write barrier — which is an instruction to a device rather than to a processor, and shares nothing with [ch20](#)’s fences but a name.

What happens when the log is too small. xv6 panics on a transaction larger than the log, which is a reasonable thing for a teaching kernel to do and not a reasonable thing for a file system to do. Making a large operation into several transactions that are each individually safe is most of what makes a real journalling file system hard.

Problems

Three, in `tests/the_file_system/filesystem.c`.

22.1 — How many block writes does a transaction cost? Derive the formula. It is graded partly against this chapter’s own measurement: whatever you arrive at has to turn the blocks the book logged into the writes the book counted, or one of the two is wrong.

```
python3 -m pytest tests/the_file_system/test_problem_1_writes.py
```

22.2 — What must recovery do after a crash at each stage? Five stages; the answer changes exactly once on the way up and once on the way down. Finding those two points is the exercise.

```
python3 -m pytest tests/the_file_system/test_problem_2_crash.py
```

22.3 — Which orderings are safe? Six orderings of the same three writes. Exactly one is safe at every point a crash could happen, and the question is not which produces the right end state — all six do, if nothing goes wrong.

```
python3 -m pytest tests/the_file_system/test_problem_3_ordering.py
```

Where to go next

xv6’s `kernel/log.c` COX ET AL. (N.D.) is the whole of this chapter. Read `commit` first — it is a handful of lines and they are in the order this chapter argued for — then `recover_from_log`, which is what makes those lines mean anything.

`kernel/fs.c` and `kernel/bio.c` are the layers underneath, and `bio.c` turns out to be a cache with a lock around it, which is a shape worth recognising: a great deal of a kernel is one or the other, and much of it is both.

ch23 is the hinge. Everything **Part IV** has established is about what a program *does*, on a target chosen because you can stop it and look. The next chapter puts the same program on a machine where you cannot, and asks what the first instrument failed to tell you — which is the question the whole of **Part V** exists to answer.

23 · The Same Program on Both Targets

CHAPTER HEADER

Target	both — <code>xv6</code> and <code>host</code> , the same source compiled and run on each
Answers the cost of	ch14 , ch16 , ch17
Prerequisites	ch22
What it measures	Everything the two targets agree about: <code>bench/results/bridge-both.json</code> ; and what the two routes cost on the board, which is not among them: <code>bench/results/bridge-host.json</code>

The question

What does watching a program in a debugger fail to tell me about what it costs?

Everything up to here has built a complete structural account of a machine. You can say what a program compiles to, where its data sits, how its address space is built, what it costs the kernel to ask for anything, and how many blocks of disk traffic a single byte becomes. All of it was obtained on a target chosen precisely because you can stop it anywhere and look.

None of it is a duration, and this chapter is about what that leaves out.

The material

One program, two routes

Here is a program small enough that the parts before this one explain it completely.

An array of cells, each holding a number and the index of another cell. The chain visits every cell exactly once. Two routes add up every number: one walks the array in order, and the other follows the chain.

bridge.h

```
long sysfs_bridge_sequential(const struct sysfs_cell *cells, long count) { \
    long total = 0; \
    for (long i = 0; i < count; i++) \
        total += cells[i].value; \
    return total; \
} \
\
/* Every value, in the order the chain gives them. Same values, same total, \
 * same number of additions, and one more load to find out where to go. */ \
```

They add the same numbers in a different order and return the same total. Compiled for the `xv6` target and for the reference machine, run on both, they agree:

What both targets say	Value
The answer both routes computed	8390656
Elements walked	4096
Instructions in the sequential route (RISC-V)	12
Instructions in the chased route (RISC-V)	15
Instructions in the sequential route (AArch64)	13
Instructions in the chased route (AArch64)	18

Conditions: target `xv6` · riscv64 cross build, any machine — run under user-mode QEMU for counts only · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · `xv6`'s own CFLAGS on one side; the book's portable flags on the other · 2026-09-17 · Source: `bench/results/bridge-both.json`, code hash `1ab750e4b1e975e9`.

That agreement is what the book has established so far. The same source produces the same answer on two machines with different kernels, different instruction sets and different everything else — because [ch13](#) checked the data model matched, and because the answer is a property of the arithmetic rather than of the machine.

What the structure predicts

Now the instructions. RISC-V first:

`sysfs_bridge_sequential` compiled for riscv64 at `-O2` — the `xv6` target's instruction set, 12 instructions.

```
0000000000000052 <sysfs_bridge_sequential>:
52:  blez    a1,6c <.L16>
56:  slli    a1,a1,0x4
58:  mv      a5,a0
5a:  add     a3,a0,a1
5e:  li      a0,0

0000000000000060 <.L15>:
60:  ld      a4,0(a5)
62:  addi    a5,a5,16
64:  add     a0,a0,a4
66:  bne     a5,a3,60 <.L15>
6a:  ret

000000000000006c <.L16>:
6c:  li      a0,0
6e:  ret
```

Conditions: target `xv6` · riscv64 cross compiler, any machine — nothing here was executed · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · `-O2 -Wall -Wextra -march=rv64gc -mabi=lp64d -c` · 2026-09-17 · Source: `bench/results/bridge-riscv64.json`, code hash `2c0d5645ba9b7b66`.

`sysfs_bridge_sequential` compiled for aarch64 at `-O2` — the `host` target's instruction set, 13 instructions.

```

0000000000000090 <sysfs_bridge_sequential>:
90:  cmp     x1, #0x0
94:  b.le    bc <sysfs_bridge_sequential+0x2c>
98:  mov     x2, x0
9c:  mov     x0, #0x0                // #0
a0:  add     x3, x2, x1, lsl #4
a4:  nop
a8:  ldr     x1, [x2], #16
ac:  add     x0, x0, x1
b0:  cmp     x2, x3
b4:  b.ne    a8 <sysfs_bridge_sequential+0x18> // b.any
b8:  ret
bc:  mov     x0, #0x0                // #0
c0:  ret

```

Conditions: target `host` · aarch64 cross compiler, any machine — nothing here was executed · aarch64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · -O2 -Wall -Wextra -c · 2026-09-17 · Source: `bench/results/bridge-aarch64.json`, code hash `140cc3a3926d377c`.

sysfs_bridge_chased compiled for riscv64 at -O2 — the `xv6` target's instruction set, 15 instructions.

```

0000000000000070 <sysfs_bridge_chased>:
70:  mv      a2,a0
72:  blez    a1,8e <.L21>
76:  li      a4,0
78:  li      a5,0
7a:  li      a0,0

000000000000007c <.L20>:
7c:  slli    a5,a5,0x4
7e:  add     a5,a5,a2
80:  ld      a3,0(a5)
82:  addi    a4,a4,1
84:  ld      a5,8(a5)
86:  add     a0,a0,a3
88:  bne     a1,a4,7c <.L20>
8c:  ret

000000000000008e <.L21>:
8e:  li      a0,0
90:  ret

```

Conditions: target `xv6` · riscv64 cross compiler, any machine — nothing here was executed · riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · -O2 -Wall -Wextra -march=rv64gc -mabi=lp64d -c · 2026-09-17 · Source: `bench/results/bridge-riscv64.json`, code hash `2c0d5645ba9b7b66`.

sysfs_bridge_chased compiled for aarch64 at -O2 — the `host` target's instruction set, 18 instructions.

```

0000000000000000c4 <sysfs_bridge_chased>:
c4:  mov     x4, x0
c8:  cmp     x1, #0x0
cc:  b.le    104 <sysfs_bridge_chased+0x40>
d0:  mov     x3, #0x0                // #0
d4:  mov     x2, #0x0                // #0
d8:  mov     x0, #0x0                // #0
dc:  nop
e0:  lsl     x2, x2, #4
e4:  add     x3, x3, #0x1
e8:  add     x6, x4, x2
ec:  ldr     x5, [x4, x2]
f0:  ldr     x2, [x6, #8]
f4:  add     x0, x0, x5
f8:  cmp     x1, x3
fc:  b.ne    e0 <sysfs_bridge_chased+0x1c> // b.any
100: ret
104: mov     x0, #0x0                // #0
108: ret

```

Conditions: target `host` · aarch64 cross compiler, any machine — nothing here was executed · aarch64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · `-O2 -Wall -Wextra -c` · 2026-09-17 · Source: `bench/results/bridge-aarch64.json`, code hash `140cc3a3926d377c`.

Read the loop bodies. The sequential route loads a value, advances a pointer by a constant, adds, and branches. The chased route does the same and then loads again to find out where to go next, and has to compute an address from an index rather than stepping.

So: one more load, and a shift and an add. Everything [Part III](#) and [Part IV](#) have to say about these two functions is in that sentence, and it predicts that the chased route costs somewhat under twice what the sequential one does.

Hold on to that prediction. It is the most that a complete structural understanding can offer, it was arrived at correctly, and [Part V](#) exists because of how wrong it is.

The number the first target could not supply

	Nanoseconds per element
Sequential route	2759
Chased route	9685
Measured ratio	3.5x
Ratio the instruction counts predict	1.4x

Conditions: target `host` · Raspberry Pi 5 Model B Rev 1.1 · Linux 6.18.50+rpt-rpi-2712 · cc (Debian 14.2.0-19) 14.2.0 · `-O2 -g -Wall -Wextra` · 2026-09-16 · Source: `bench/results/bridge-host.json`, code hash `248a965fb6193f75`.

That table came from the board and could have come from nowhere else — and it is the one place in this book where a duration appears before you have been given any way to judge one. Take the two numbers on trust for exactly one chapter: [ch24](#) is next precisely because it is about how much to trust them, and it comes back to put a spread and an interval around figures like these. A duration measured under QEMU would describe the laptop that ran the emulator: it has no cache to miss, no prefetcher to defeat, no store buffer to fill, and no memory that takes time. It would produce a number, the number would look like the others in this book, and it would mean nothing at all. So until the board answered, this table was a box saying so.

Read the last two rows against each other. The first is the prediction the section above arrived at, from the instruction counts; the second is what the machine did. Everything [Part III](#) and [Part IV](#) can say about these two functions was in that prediction, and it was arrived at correctly, and the machine disagrees with it by more than the whole of the structural account can explain.

The chased route's whole cost is that each load has to finish before the next address is known. That is a statement about a memory system, and the target that has taught you everything else in this book does not have one. [ch25](#) is where that cost gets taken apart.

Three things differ at once

The board's numbers will be easy to misattribute, and avoiding that is the skill the rest of [Part V](#) is built on.

The two runs differ in **three** ways simultaneously. One is emulated and one is not. One is xv6 and one is Linux. One is RISC-V and one is AArch64. A difference in the measurement could be caused by any of them, and the temptation — universal, and not confined to beginners — is to attribute it to whichever is most interesting.

The fix is not care. It is design: **compare two configurations that differ in exactly one thing**. xv6 under QEMU against Linux under QEMU isolates the kernel. Linux under QEMU against Linux on hardware isolates the emulation. Linux on one board against Linux on another isolates the instruction set. Nothing can be isolated by comparing the first of those with the last, and no amount of statistics applied afterwards recovers what the design gave away.

Problems 23.1 and 23.2 are that move, mechanically, until it is automatic.

What survives the crossing

It is easy to summarise everything behind you as “structure transfers and cost does not”, and that is wrong in both directions.

The instruction count is structural and does not transfer. Two instruction sets do not emit the same number of instructions for the same source — the table above shows they do not, for this very program. A model built on counting instructions is a model of one compiler for one architecture.

The memory layout is structural and does transfer, but not automatically: it transfers because both targets are LP64 — `long` and pointers are both eight bytes — with the same alignment rules, which [ch13](#) measured rather than assumed. A book that had chosen a 32-bit target for [Part III](#) would have had to say something quite different here.

And one thing transfers that is neither: **the mechanisms themselves**. A page fault is a page fault, a context switch saves the callee-saved registers, a log makes a group of writes atomic. What changes across the crossing is every number attached to them, which is exactly why [Part V](#) is arranged as [Part IV](#)'s chapters asked again.

What we measured

The answer both routes compute, on both targets, and the instructions each compiles to on both architectures — all of it reproducible, none of it a duration. The one figure this chapter is actually about — the cost of the two routes — has been measured on the board.

What the earlier parts predict about these two functions does not depend on that measurement, and neither does the argument about confounds; the measurement supplies the size of the error and not its existence.

What this cannot tell you

Why the prediction was wrong. The central claim here — that a complete structural model predicts the cost badly — is shown in one table and explained in none of this chapter. The next eight are the explanation.

Which of the three differences matters most. Naming the confound is not separating it. Doing that needs configurations this book does not ship: a Linux image for the RISC-V target, or a second board. [ch24](#) starts from the machine the book does have.

Whether the structural model is useless. It is not, and the chapter would be dishonest to imply it. It predicts the answer exactly, predicts the layout exactly, and predicts which of two programs does more work — which is often the question. What it cannot do is turn that into a ratio, and knowing which questions your model answers is worth more than a better model.

Problems

Three, in `tests/the_same_program_on_both_targets/crossing.c`. All three are graded against a table of six configurations that the tests read for themselves, so there is no key to find.

23.1 — What does comparing these two configurations isolate? All thirty-six pairs. A comparison isolates a variable only when it is the single thing that differs; otherwise it confounds, and the pair this book’s own two targets form confounds all three.

```
python3 -m pytest tests/the_same_program_on_both_targets/test_problem_1_isolates.py
```

23.2 — Which configuration would test this claim? Somebody says the difference is caused by the emulator, or the kernel, or the instruction set. Given where you are, name the configuration that would settle it — or say that none of the six will, which happens and is the right answer when it does.

```
python3 -m pytest tests/the_same_program_on_both_targets/test_problem_2_pair.py
```

23.3 — Does this observation transfer? Six things you could observe about the program. Sorting them into structure and cost gets two of them wrong.

```
python3 -m pytest tests/the_same_program_on_both_targets/test_problem_3_transfers.py
```

Where to go next

Part V. Every chapter of it names the earlier chapter whose cost it measures, and the first of them is about the instrument rather than the machine: before measuring anything, [ch24](#) asks how you would know a measurement was wrong, which on a board that throttles under sustained load is not a rhetorical question.

For the confound itself, Mytkowicz and colleagues MYTKOWICZ *ET AL.* (2009) is the paper to read — it shows measured speedups appearing and disappearing according to the size of an environment variable, which is a more alarming demonstration of this chapter’s point than anything here.

Part V — Where the cycles go

Part V · Where the cycles go · 24 · Measuring · 25 · The Memory Hierarchy · 26 · Optimising Code · 27 · The CPU · 28 · Memory Ordering on Real Hardware · 29 · The OS Layer's Cost on Real Hardware · 30 · Whole-Machine Profiling · 31 · Vectors

Part V · Where the cycles go

PART HEADER

Chapters	ch24–ch31
Target	host — Linux on real hardware, natively (ch01)
Assumes	Part IV, and a machine of your own that meets Appendix H's requirements

What this part is for

Part IV's chapters, asked again as questions about time.

Nearly every chapter here has a counterpart earlier in the book and names it in its header, so you arrive already knowing what a context switch moves, what a page fault does and what a lock is for. Only the price is left to establish. That is a better position to measure from than the usual one, because the commonest way to be wrong about performance is not arithmetic — it is attributing a cost to a mechanism you had only a rough idea about.

ch24 comes first and is the method: how to get a number you would defend, and how you would know it was wrong. Everything after it is that method applied to one layer at a time — the memory, the compiler, the core, the kernel — then to a whole machine at once, and last to the one unit a program has to be rewritten to use.

It is last because it is the only part that needs the other four to have happened.

What it leaves out

The whole of what *Systems Performance* GREGG (2020) covers, deliberately, because that book is where this one is trying to deliver you. Tracing and BPF, flame graphs, the network and storage stacks, containers, observability across a fleet: none of it is here. This part is the layer underneath — cache misses, faults and context switches are what all of those tools report, and this is where they get their meaning. Do this first, because a flame graph of a workload whose costs you cannot account for is a picture rather than an answer.

It also leaves out a list of numbers to memorise. The figures are this board's. Some of them would be different on yours by a factor that matters, and every chapter says so. What transfers is the method, the shape of the answer, and knowing which questions the instrument cannot answer.

Where to start

ch24, and this is the one instruction in the book worth being rigid about. Every chapter after it produces numbers using its method, and reading them without it is how people end up confident and wrong — a measurement is not self-describing, and the difference between a defensible figure and a plausible one is entirely in how it was taken.

After that the header rows will route you: each chapter names the earlier chapter whose cost it answers, so you can follow a single mechanism from what it does to what it charges.

Which machine, and what it cannot tell you

A small Linux machine on a desk, reached over SSH, running natively. Every number in this book that is a *cost* was measured here, and each one records the board, the kernel, the compiler and a hash of the code that produced it.

Several chapters go further than using the machine and depend on it — a particular cache hierarchy, a particular core, a particular set of performance counters exposed by a particular firmware. Those say so in an **Assumes** row of their own. On a different board the mechanism holds and the figures will not match, and where that is likely the chapter says what to expect instead of pretending the number is universal.

The disassembly here is AArch64, not the RISC-V you learned in [Part III. Appendix F](#) is the translation between the two. The change of architecture is not an accident: these chapters need performance counters that both *count* and *sample*, and no purchasable RISC-V core does both. Staying on one architecture would have cost two chapters their measurements, which is a worse trade than asking you to read a second instruction set in five of them.

And where a thing genuinely cannot be measured on the available hardware — an absent unit, a counter the firmware does not expose, a second machine that does not exist — the chapter says so, shows the reasoning it used instead, and states what it would take to measure. It does not borrow a number from somewhere else.

More than once this part sets out to reproduce a well-known effect and finds it mostly gone. Bias from where the stack sits, the penalty for false sharing, the clean step that betrays a cache line: each is real, each is in the literature, and on this newer core each came back smaller than the folklore promises, sometimes to nothing. That is not the measurement failing. A modern core spends its transistors hiding exactly these effects, so the older the demonstration the likelier the silicon has closed it, and the number you inherited was taken on a machine that no longer exists. It is said once here because you will meet it in three chapters, and the answer is the same each time: measure yours.

Where this leaves you

Able to take a machine you have never seen, a program you did not write, and a claim about why it is slow, and find out whether the claim is true.

24 · Measuring

CHAPTER HEADER

Target	host — the reference machine, natively
Prerequisites	ch23
What it measures	The instrument, before anything is measured with it: <code>bench/results/measuring-host.json</code> ; and how wrong one number is: <code>bench/results/distribution-host.json</code>

The question

How do I get a number I would defend, and how would I know it was wrong?

[ch23](#) established that the structural model does not predict cost, and pointed at a machine that can answer. It also printed two durations, and gave you no way to judge them. This chapter is that way.

It measures what reading the clock costs, changes something about the program that should make no difference and checks whether the answer moves, and then builds the two things every later chapter quotes: a spread around a median, and a rule for deciding that one thing is faster than another. By the end nothing in this book reports a duration as a bare number again.

The material

The instrument is part of the measurement

Every duration in this book is read through one header, so that “how was this timed” has one answer rather than one per chapter. It reads the *monotonic* clock — one that only ever goes forward and is never adjusted to keep wall-clock time honest, so a measurement cannot come out negative because something corrected the date halfway through:

timing.h

```
uint64_t sysfs_now_ns(void);

/* What reading the clock costs, in nanoseconds, measured by reading it repeatedly.
 *
 * This is not trivia. If a thing being measured is of the same order as the instrument, the
 * instrument is most of what is being measured – and knowing where that threshold is tells you
 * when to stop timing single operations and start timing a loop of them. */
uint64_t sysfs_clock_cost_ns(void);
```

The second function is the one people skip. Reading a clock takes time, that time is included in every measurement made with it, and if the thing being measured is of the same order then the instrument is most of what is being measured.

	Measured
Cost of reading the clock	18 ns
Smallest change the clock ever reports	18 ns
Work that would be half instrument	18 ns

Conditions: target `host` · Raspberry Pi 5 Model B Rev 1.1 · Linux 6.18.50+rpt-rpi-2712 · cc (Debian 14.2.0-19) 14.2.0 · -O2 -g -Wall -Wextra · 2026-09-16 · Source: `bench/results/measuring-host.json`, code hash 51019b1da2ae8858.

The first two numbers there measure different things. What a clock *costs* is how long the read takes. What it can *resolve* is the smallest change it will ever report — a clock can hand back nanoseconds and still only ever move in steps of a hundred of them, in which case a measurement of anything shorter is a coin toss between zero and one step. The third row is what the first implies: work that costs as much as reading the clock is half instrument.

Problem 24.2 turns this into the arithmetic you actually need: given what the work costs and what the clock costs, how many repetitions must go inside one timed region before the instrument is small enough to ignore.

The harness costs something as well

The clock is not the whole instrument. Around every measured region there is a loop, a call, and whatever the harness does between samples, and all of it is inside the timing. So measure it: run the harness around a region that does nothing and see what comes back.

NOT MEASURED YET — THIS NEEDS THE BOARD

what the harness itself costs, and what the workload costs beside it: run `make bench-board` on the reference machine and commit the result

Nothing is estimated in its place: the book prints measurements or it prints nothing.

Two things in that table and the second is the one to carry away. The empty region does not cost nothing, which is expected. It also does not cost a *constant* — it has a spread of its own, as wide as the last row shows, and that is what makes the obvious fix wrong.

The obvious fix is to subtract it. **Do not, in general.** Subtracting one number from another is sound when the first is a fixed offset; when it is a distribution, the subtraction removes the middle of it and leaves the variation behind, so the answer is a little smaller and no more certain than before. Worse, the harness cost and the work are not independent — the same interrupt lands in both — so the difference of two medians is not the median of the difference.

The fix that does work is arithmetic you can do before running anything: make the measured region so much larger than the harness that the harness cannot matter. The last row of the table is that ratio for this workload, and problem 24.2 is how to choose it for yours. Subtraction is a way of rescuing a measurement that should have been designed differently.

There is no such thing as the time it took

Run identical work, on an idle machine, two thousand times:

	Nanoseconds
Repetitions	2000
Fastest	37499 ns
Median	37518 ns
90th percentile	37519 ns
Slowest	52889 ns
Mean	37544 ns
Slowest over fastest	1.4x

Conditions: `target host` · Raspberry Pi 5 Model B Rev 1.1 · Linux 6.18.50+rpt-rpi-2712 · cc (Debian 14.2.0-19) 14.2.0 · -O2 -g -Wall -Wextra · 2026-09-16 · Source: `bench/results/measuring-host.json`, code hash `51019b1da2ae8858`.

Read the 90th percentile against the fastest, and then the slowest. Identical work, identical input, nothing else running — and yet nine runs in ten agree to within a rounding error while the slowest stands far outside them. Nothing was wrong with any of those measurements; they are all correct observations of what happened.

So “how long does it take” has no answer, and the question has to change. Report a distribution instead, which is why `sysfs_summarise` exists and why problem 24.1 asks you to write it.

Why the minimum is usually the number to look at. Everything that can happen to a measurement on a real machine makes it slower: an interrupt arrives, another process is scheduled, a page is not resident, the clock drifts. Almost nothing makes it faster. So the mean is partly a statement about the interference and the minimum is the closest available statement about the work.

And when that reasoning fails. If you care how long something takes *in service*, the interference is not noise — it is the thing your users experience, and reporting the minimum is reporting the one case that never happens to them. Two different questions, two different statistics, and the mistake is not picking the wrong one but not noticing there was a choice.

Why the first measurements are slow

The first few measurements are slower, always, for three reasons the chapters either side of this one take apart: [ch18](#)’s pages are not yet faulted in, [ch25](#)’s caches hold somebody else’s data, and the branch predictor of [ch27](#) has never seen this loop.

Discarding them is standard practice and is usually done wrong. Warm-up is a property of *position*: the first samples are slow because they are first. A slow sample in the middle is interference, it is not warm-up, and discarding everything before it throws away good measurements in order to hide a bad one. Problem 24.3 is that distinction, which is why the definition it asks you to implement has a second clause.

Cold and warm are two experiments

Which raises the question of what you are measuring in the first place. Run the same work with the caches holding its data, and again with them holding somebody else’s:

NOT MEASURED YET — THIS NEEDS THE BOARD

the same work with the caches evicted and left alone: run `make bench-board` on the reference machine and commit the result

Nothing is estimated in its place: the book prints measurements or it prints nothing.

Neither row is the noise in the other. They are answers to two different questions — *what does this cost when it is the only thing running* and *what does it cost the first time* — and a real program does some of each. What a harness must not do is let the difference arrive by accident, which is what happens when warm-up is a number somebody picked and cold is whatever the machine happened to be in.

So the harness this chapter builds takes the mode as an argument and records it in the stamp. There is no default. A default would be a decision about what the reader is measuring, made silently, by the tool.

The variable that should not matter

Here is the experiment every benchmark deserves and few get: vary something that cannot affect the result, and check that the result does not move.

The same work, the same input, the same binary, three times — differing only in how many bytes of stack were claimed before the buffer it walks was allocated. Not used. Claimed. Each run lands the data at a different address, and the runner records that address, so this is a real change to where everything sits and not padding the compiler quietly folded away.

What was changed	Fastest	Median
0 bytes of stack claimed first	204832 ns	204833 ns
64 bytes of stack claimed first	204814 ns	204833 ns
512 bytes of stack claimed first	204814 ns	204833 ns

Conditions: `target host · Raspberry Pi 5 Model B Rev 1.1 · Linux 6.18.50+rpt-rpi-2712 · cc (Debian 14.2.0-19) 14.2.0 · -O2 -g -Wall -Wextra · 2026-09-16 · Source: bench/results/measuring-host.json`, code hash `51019b1da2ae8858`.

The addresses differ; the median does not move by a nanosecond. On this core, where the data sits does not change what reaching it costs.

On other machines it does move. Mytkowicz and colleagues MYTKOWICZ ET AL. (2009) made the same change — they varied the size of an environment variable, which moves the stack, exactly as the padding here does — and found it large enough to manufacture or erase the kind of speedup papers are published about. Their machine cared. This one, measured the same way, does not, very likely because a core with this much first-level cache, and this forgiving an attitude to unaligned access, hides what a 2009 one could not.

The null result does not make the experiment less worth running. You cannot tell which machine you have by reasoning about it — Mytkowicz and colleagues could not, and neither could this chapter until the board answered. A single configuration, measured carefully, can be confidently wrong, and no amount of repetition inside it ever finds out. So do not conclude the effect is gone because this page did not find it. Vary the thing that should not matter, on your own machine, and see whether your result survives.

The floor moves

One more source of slow samples, specific to this book’s reference machine and to most small ones.

A Raspberry Pi 5 under sustained load gets hot and reduces its clock. Not gradually and not with a warning: the frequency drops, and every measurement after that point is against a different machine from the ones before it. A benchmark that runs for a minute can therefore show a “regression” in its second half that is entirely thermal — and a comparison between two implementations, run one after the other, can rank them by which went first.

The defence is to record the clock alongside the result, which is why [Appendix H](#) insists on the active cooler and why every `host` result in this book stamps the machine’s state as well as its number. It is also why the results are taken with the board idle and the laptop not driving it over a link that interrupts it, which is the next section’s subject.

What else the machine is doing

[ch01](#) recommends wiring the board rather than using its radio, and gives a mechanism: a wireless driver takes interrupts and runs deferred work on the cores being measured, which is [ch19](#)’s subject arriving where it is least wanted.

So: the same fixed workload, run many times over with the radio off and again with it on — the only thing changed being a variable that has nothing to do with the code.

	Radio off	Radio on
Fastest sample	37499 ns	37499 ns
Best run’s mean	37537 ns	37536 ns
Worst run’s mean	37561 ns	52855 ns
Runs disturbed (of 30)	0	2

Conditions: target `host` · Raspberry Pi 5 Model B Rev 1.1 · Linux 6.18.50+rpt-rpi-2712 · cc (Debian 14.2.0-19) 14.2.0 · -O2 -g -Wall -Wextra · 2026-09-16 · Source: `bench/results/interference-host.json`, code hash `b0f816ce68014b0e`.

Read the first row before any other. The fastest sample is identical to the nanosecond, radio or not, because interference can only ever add time: the fastest run is the one it missed. That is the earlier argument for the minimum, demonstrated rather than asserted — everything the radio does makes a measurement slower, so the mean carries the interference and the floor carries the work.

Now read the rest of the table. A clean run with the radio on is indistinguishable from one with it off, so a single measurement cannot tell you which you have: the radio’s background work arrives in bursts, and only a run that catches one is wrecked — its mean dragged well above the floor while its neighbours look fine. Which run you happened to take decides your answer, and nothing inside that run tells you which kind you got. It is the warm-up problem inverted: not a slow prefix you can cut, but slow samples scattered through a run that averages out clean until it does not.

Wiring the board does not make it faster. It makes the number mean something, by removing a source of slow samples the work has no say in. The radio is one such source; a busy neighbour, a background build, a cron job at the wrong minute are others, and the defence is the same — measure on a machine you have made quiet, and record

what quiet meant. Problem 24.4 is to reproduce this on your own board, where the numbers will differ and the shape should not.

How many runs is enough?

Everything so far says report a spread. It does not say how much of the spread is the machine and how much is that you only ran it four hundred times. That question has an answer you can compute from the samples you already have, and it needs no assumption about their shape.

Take the samples, draw a new set of the same size from them *with replacement*, and write down its median. Do that a couple of thousand times. You now have a couple of thousand medians, each from a run that could plausibly have happened instead of the one that did, and the middle of them is an interval on the median. That is the whole of the bootstrap, and it is about fifteen lines:

distribution.py

```
def bootstrap_median_ci(
    samples: Sequence[float],
    *,
    resamples: int = RESAMPLES,
    seed: int = SEED,
    confidence: float = 0.90,
) -> tuple[float, float]:
    """A percentile-bootstrap interval on the median of ``samples``.

    Draw ``resamples`` new sets of the same size, with replacement, take each one's median, and
    report the middle ``confidence`` of those medians. No distribution is assumed and none is
    fitted: the samples are the population.

    The interval answers one question – how much would this median move if the run were repeated
    under conditions like these – and it cannot answer any other. In particular a board that
    warms up during the run breaks the assumption that the samples are interchangeable, and the
    interval then looks tight while being wrong. That is not a flaw in the arithmetic. It is what
    the environment record beside it is for.
    """
    if not samples:
        raise ValueError("cannot bootstrap an empty sample set")
    if not 0 < confidence < 1:
        raise ValueError(f"confidence must be between 0 and 1, not {confidence}")
    if resamples < 1:
        raise ValueError("a bootstrap needs at least one resample")

    rng = random.Random(seed)
    population = list(samples)
    size = len(population)
    medians = sorted(statistics.median(rng.choices(population, k=size)) for _ in range(resamples))

    tail = (1.0 - confidence) / 2.0
    return (_at(medians, tail), _at(medians, 1.0 - tail))
```

Nothing in there fits a curve or assumes one. The samples *are* the population, which is why this works on a distribution with a long tail and a floor, and why it would still work on one shaped in a way nobody has a name for.

Now watch what it buys, by running it on more and more of one run:

NOT MEASURED YET — THIS NEEDS THE BOARD

how the interval on the median narrows with the run count: `run make bench-board` on the reference machine and commit the result

Nothing is estimated in its place: the book prints measurements or it prints nothing.

The interval narrows as the count grows, which is the part everyone expects, and then it stops, which is the part worth knowing. What averaging removes is the noise that differs between runs. Anything the same in all of them — a governor that settled at one frequency, a board a degree warmer than it started, a layout the loader chose once — is not noise to this arithmetic at all. It is the answer. Past the point where the curve flattens, more runs buy a tighter interval around a number that may still be wrong, and the honest response is to change the conditions rather than to collect more evidence under the ones you have.

Is B faster than A?

This is the question every later chapter asks, so it gets a rule rather than a judgement.

Two variants of one loop, differing only in how many elements each iteration handles. The threshold — how much difference would be worth caring about — is **two per cent, fixed before the run**, and that order matters more than the number does. A threshold chosen after seeing the result is not a threshold; it is the result, written twice.

NOT MEASURED YET — THIS NEEDS THE BOARD

the decision rule applied to two variants of one loop: `run make bench-board` on the reference machine and commit the result

Nothing is estimated in its place: the book prints measurements or it prints nothing.

B is faster than A only when both halves hold. The interval on the difference must exclude zero, which is the evidence half: an interval straddling zero is consistent with there being no difference at all, whatever the two medians happen to be. And the difference must clear the threshold, which is the significance half: a difference can be real, repeatable, and far too small for anybody to want.

distribution.py

```
excludes_zero = low > 0 or high < 0
difference = candidate_median - base_median
relative = abs(difference) / base_median if base_median else 0.0
big_enough = relative >= threshold

if excludes_zero and big_enough:
    verdict: Verdict = "faster" if difference < 0 else "slower"
else:
    verdict = "cannot tell"
```

That leaves three outcomes rather than two, and the third is the useful one. **“Cannot tell” is a result.** It says this experiment does not separate these two, which is a fact about the experiment and an instruction about what to do next — run more, or quieten the machine, or accept that the difference is smaller than you can see from here. A book that never printed it would be claiming a precision its instrument does not have, and problem 24.5 is about recognising the two quite different ways of arriving at it.

Why none of this works on xv6

Everything in this chapter needed the board, and it is worth being precise about why, because the other two targets are right there and running.

The stamping machinery will not let you. An `xv6` or `bare` result carrying anything that looks like a duration is refused before it can be written:

stamp.py

```
if target in ("xv6", "bare"):
    if machine.get("kind") != "qemu":
        problems.append(
            f"{name} declares target {target!r} but machine.kind is "
            f"{machine.get('kind')!r}, not 'qemu'."
        )
    timing = timing_keys(payload.get("summary", {}))
    if timing:
        problems.append(
            f"{name} is a {target!r} result carrying what looks like a timing: "
            f"{', '.join(timing)}. QEMU models no cache, no predictor and no pipeline, so a "
            "duration measured inside it means nothing – move the measurement to the board."
        )

return problems
```

That is not a policy about tidiness. Under QEMU a duration is the host's scheduler, the host's cache and the translation strategy, sampled through a program that thinks it is a RISC-V machine; run the same `xv6` workload while a build is running on the laptop and the number moves, because the number was always about the laptop. It would arrive in the stamp looking exactly like the figures in [Part V](#), and once it is a row in a table nothing distinguishes it.

So the refusal is machinery rather than advice. [ch00](#) made the argument; this is where the argument is enforced.

What we measured

The cost and resolution of the clock; the spread of two thousand runs of one fixed workload — a body a rounding error wide, and a tail that is all interference; a memory walk run three times with its data deliberately moved to a different address each time, which on this machine did not move the time at all; and that same workload with the radio off and on, which moved the mean and left the floor exactly where it was.

Then the second instrument, and everything the rest of [Part V](#) reports through it: an empty measured region, so the harness has a price of its own rather than an assumed one; the same work cold and warm; one run analysed at six different counts, to find where more runs stop buying anything; and two variants of one loop put to the decision rule. Every one of those keeps its samples — `bench/raw/`, one nanosecond value per line — because an interval is computed from samples and a result that threw them away could only ever be re-measured, which needs the board.

What this cannot tell you

Whether your machine behaves like this one. Almost everything above is a property of a particular board, a particular kernel and a particular thermal design. The *method* is what transfers, which is the reason this chapter comes before any result.

How much repetition is enough. Problem 24.2 gives the arithmetic for making the clock negligible, and that is a necessary condition rather than a sufficient one. How many samples you need before the distribution is trustworthy is a question about the distribution's shape, which you do not know until you have sampled it.

What to do about a bimodal result. Sometimes a benchmark genuinely has two answers — one where the data was resident and one where it was not — and every summary statistic in this chapter reports something that happened in neither case. When the histogram has two humps, the right answer is to find out what distinguishes them, which is [ch30](#)'s equipment.

Anything about the compiler having deleted your benchmark. The loop measured here is written so it cannot be optimised away, and the technique — a dependent chain feeding a value the program later uses — is stated rather than demonstrated. [ch26](#) shows what happens when it is forgotten, which is a benchmark that measures an empty loop and reports an enormous speedup.

Whether the runs were independent, which the bootstrap assumes. Resampling treats the samples as interchangeable: any one could have come in any position. A board that warms up during the run, or a governor that settles part-way through, makes that false — the later samples are not exchangeable with the earlier ones — and the interval then comes out tight while being centred in the wrong place. Nothing in the arithmetic notices. The environment recorded beside each figure is there so that a reader can, and the run-order plot is the cheapest way to look.

What happens on a busy machine. Every figure here was taken on a board made deliberately quiet. That is the right way to measure a *difference between two versions of a program*, and it is the wrong way to estimate what the program will do in service, where the interference is not noise but the thing your users experience. Two questions, two methods, and this chapter has only done the first.

Anything about rare events. Four hundred runs say a good deal about the middle of a distribution and almost nothing about its far tail. If what matters is the worst case in a million, the tail in this chapter's tables is not evidence about it — you would need a different experiment, run for far longer, and the statistic you want is not the one reported here.

Any effect smaller than the harness's own noise. A difference beneath the floor of the empty measured region cannot be detected by this instrument at any run count. That is not a limit of the statistics; more evidence does not help, because the instrument never recorded the difference in the first place. Making a smaller effect visible means a bigger measured region or a better clock, and both are changes to the experiment rather than to the analysis.

Problems

Seven. Most are in `tests/measuring/` and are graded against definitions the tests compute for themselves. One has no test at all: its answer is a property of your machine, not this one's.

24.1 — Report the distribution. Minimum, median, 90th percentile and mean, to the definitions in the stub. The even-length case is the one that matters: the median of an even set is a sample that happened, not the average of two that did.

```
python3 -m pytest tests/measuring/test_problem_1_summary.py
```

24.2 — How many repetitions? Given what the work costs and what the clock costs, the smallest count for which the instrument is within budget. One of the cases has the answer “one”, and one has the answer “this is not a question”, and both are worth getting right.

```
python3 -m pytest tests/measuring/test_problem_2_repetitions.py
```

24.3 — How much of this is warm-up? The definition has two clauses, and the second is where the difficulty is: the prefix you discard has to have actually been slow. Leave it out and a single spike in the middle lets you throw away every good measurement before it.

```
python3 -m pytest tests/measuring/test_problem_3_warmup.py
```

24.4 — Reproduce the interference on your own machine. The table above is this board’s. `python3 -m bench.run_interference` runs the same experiment — it toggles the radio for you and restores it — so run it on yours and report the three things the method should let you defend: that the floor did not move, that the mean did, and how often a run was disturbed. That last figure travels least of anything in this chapter, because it depends on your radio, your kernel and whatever else the machine was doing — which is why you have to take it yourself.

There is no test, because the answer is a property of your machine rather than a fact to check against. If the floor *does* move on yours, something more interesting than the radio is loose, and tracking it down is the skill this chapter exists to teach.

24.5 — Which kind of “cannot tell” is this? Two comparisons that both decline to call a winner, for opposite reasons: one where the interval contains zero, and one where it excludes zero and the difference is under the threshold. Say which is which, and what you would do next in each case. They look the same in a summary and mean entirely different things.

```
python3 -m pytest tests/measuring/test_problem_5_verdict.py
```

24.6 — Is your harness measuring itself? Given a timer overhead and a set of measured regions, say which of the timings are reportable. A region shorter than the instrument that measured it is not a fast measurement; it is not a measurement. The test will not accept a harness that reports one.

```
python3 -m pytest tests/measuring/test_problem_6_floor.py
```

24.7 — What has to be in the stamp? A stamp is a promise that the number can be judged later. Complete the environment record so that every field this chapter argued for is present — including the ones your machine cannot answer, which are recorded as unavailable rather than left out. A missing key and a null are different claims: one says the harness never looked.

```
python3 -m pytest tests/measuring/test_problem_7_stamp.py
```

Where to go next

Mytkowicz, Diwan, Hauswirth and Sweeney MYTKOWICZ *ET AL.* (2009) is four pages and is the single most useful thing to read after this chapter. The experiment is simple enough to repeat and the result is bad enough to change your habits.

`man 2 clock_gettime` and `man 7 vdso` are worth twenty minutes: the reason reading the clock is as cheap as it is, on Linux, is that it usually is not a system call at all — [ch16](#)’s trap path is avoided by mapping a page of kernel data into every process, which is [ch17](#)’s mechanism used for something [ch17](#) had no reason to mention.

A stamped distribution is also the shape of thing another kind of model wants as an input. The companion book *sizing-and-tco* builds Monte Carlo models of what a system will cost at a given size, and every such model needs measured constants with their uncertainty attached — which is exactly what the stamps from this chapter are. The bootstrap here is about measurement error and nothing else; propagating a distribution through a model is that book’s subject and is not attempted in this one.

[ch25](#) is the first chapter to ask the machine a question, and it settles this book’s running example: [ch23](#)’s two routes differ because of where the data is, and the next chapter measures that hierarchy rather than looking it up.

25 · The Memory Hierarchy

CHAPTER HEADER

Target	host — the reference machine, natively
Answers the cost of	ch13 , ch17
Prerequisites	ch24
Assumes	A particular cache hierarchy. The numbers are this board's; the method is not.
What it measures	The hierarchy, by asking the machine rather than reading its datasheet: <code>bench/results/hierarchy-host.json</code>

The question

Where is the data, and what does each extra step out cost?

[ch23](#) left a prediction on the table: two routes over the same array, differing by one load, so the structural model says the slower one costs somewhat under twice the faster. This chapter has the equipment to say what actually separates them, and it does not start by looking anything up.

The material

Measuring a cache instead of reading about it

A datasheet will tell you this machine's cache sizes. So will `/sys`. Neither is the same as measuring them, and [ch00](#) states the reason as policy: a specification is a claim about a product line, and a measurement is a statement about the silicon in front of you.

The instrument is a **dependent pointer chase** — a cycle of pointers, each pointing at the next, walked one at a time. That shape matters. It makes every load wait for the one before it, so the machine cannot overlap them, and what comes out is one latency rather than a throughput. A loop that walks an array with independent loads measures how many loads the machine can have in flight at once, which is a real number and a completely different one.

The cycle is also deliberately not sequential. A prefetcher that recognises a pattern will fetch ahead and answer a question nobody asked.

The steps are the levels

Walk a cycle that covers more and more memory, and plot how long one step takes:

Working set	Nanoseconds per dependent load
1 KiB	1 ns
2 KiB	1 ns
4 KiB	1 ns
8 KiB	1 ns
16 KiB	1 ns
32 KiB	1 ns
64 KiB	1 ns
128 KiB	3 ns
256 KiB	3 ns
512 KiB	3 ns
1024 KiB	3 ns
2048 KiB	3 ns
4096 KiB	6 ns
8192 KiB	6 ns
16384 KiB	6 ns
32768 KiB	6 ns
65536 KiB	6 ns

Conditions: `target host` · Raspberry Pi 5 Model B Rev 1.1 · Linux 6.18.50+rpt-rpi-2712 · cc (Debian 14.2.0-19) 14.2.0 · -O2 -g -Wall -Wextra · 2026-09-16 · Source: `bench/results/hierarchy-host.json`, code hash `7752dfc03ba6c5fa`.

The curve is flat, then steps, then flat, then steps again. Nothing in the program changed between those points except how much memory the cycle covers, so each step is the *working set* — the memory the loop touches — ceasing to fit in something, and the size at which it steps is the size of the thing it stopped fitting in.

Nothing else is needed to find the levels. The hierarchy is not inferred from a specification; it is read off a curve the machine produced when it was asked.

Problem 25.1 is that reading, on synthetic curves whose answers are known by construction.

The line is the unit

A second experiment, with the working set fixed and the gap between visits growing:

Stride	Nanoseconds per dependent load
8 bytes	1 ns
16 bytes	1 ns
32 bytes	2 ns
64 bytes	3 ns
128 bytes	5 ns
256 bytes	6 ns
512 bytes	20 ns
1024 bytes	72 ns
2048 bytes	26 ns
4096 bytes	20 ns

Conditions: target `host` · Raspberry Pi 5 Model B Rev 1.1 · Linux 6.18.50+rpt-rpi-2712 · cc (Debian 14.2.0-19) 14.2.0 · -O2 -g -Wall -Wextra · 2026-09-16 · Source: `bench/results/hierarchy-host.json`, code hash `7752dfc03ba6c5fa`.

Flat, and then it climbs. While several visits share a cache line the later ones are nearly free — the first fetch brought them all — and each doubling of the *stride*, that gap, halves how many share, so the cost climbs a step at a time rather than jumping once.

The naive reading is that the stride at which it first rises is the line size. On this machine that reading lands *below* what the vendor publishes, because the sharing thins out gradually: the curve is already moving before the last visit has a line to itself. That gap is what the vendor table below shows. So read the curve knowing that, rather than trusting the first rise — but what the line *explains* does not depend on reading it to the byte. Alignment and padding matter because memory moves in lines, and a structure straddling two costs two fetches for one field, which is the question [ch13](#) raised and could not settle.

Problem 25.3 turns that into arithmetic. The shape it asks about is a ceiling: the line effect gets steadily worse as the stride grows and then can get no worse, because one line per visit is as few as they can share. The measured curve does something past that point that the arithmetic cannot account for — it climbs steeply, then falls back — so whatever moves it there is a second effect sharing the axis. By then the stride has grown to a page and beyond, and translation is involved, which the next section measures. Why the curve falls back again is not something this chapter can say from this curve alone, and it does not guess.

Translation has a cache too

[ch17](#) established that every address is a question answered by a walk through three levels of page table. It did not say what happens when that walk is not cached, because that target has no cache to not-be-in.

Touch one pointer per page, so the *data* comfortably fits in the last-level cache while the *translations* stop fitting:

	Measured
Pages the translation cache holds	512
Memory those pages reach	2097152
Last level, for comparison	no step found

Conditions: target `host` · Raspberry Pi 5 Model B Rev 1.1 · Linux 6.18.50+rpt-rpi-2712 · cc (Debian 14.2.0-19) 14.2.0 · -O2 -g -Wall -Wextra · 2026-09-16 · Source: `bench/results/hierarchy-host.json`, code hash `7752dfc03ba6c5fa`.

The reach of a TLB is its entries multiplied by the page size, and the second row is that product. The third row was meant to put the last-level cache beside it, and on this board it cannot: the levels curve above found no step for one. Set the reach against the vendor’s figure for that cache in the table below instead, and the two are the same size — so on this core the translations run out at about the point the data does, not well before it, which is the textbook shape and not this machine’s. What survives is the mechanism. A program whose data fits in cache can still pay for a page-table walk on every access once it touches more pages than the TLB holds, and `ch17`’s three levels are then three more memory accesses. That is where the walk stops being a structural fact and becomes a cost; how large a one, on a board where the two limits coincide, is not something this experiment separates.

Comparing the curve with the datasheet

Every number above came out of a curve. Only now is it worth looking anything up.

	Measured	Vendor’s figure
Cache line	16	64
First level	65536	65536
Second level	2097152	524288
Last level	no step found	2097152

Conditions: target `host` · Raspberry Pi 5 Model B Rev 1.1 · Linux 6.18.50+rpt-rpi-2712 · cc (Debian 14.2.0-19) 14.2.0 · -O2 -g -Wall -Wextra · 2026-09-16 · Source: `bench/results/hierarchy-host.json`, code hash `7752dfc03ba6c5fa`.

Two columns that ought to agree, and are allowed not to. Where they differ the measurement is what this book prints, for the reason the chapter opened with — the right-hand column describes a product line and the left-hand one describes the chip that produced it. A disagreement is not an error in either: it is the most interesting thing on the page, and the question to ask is which column you would have believed if you had only had one of them.

Here they disagree twice. The measured line is a fraction of the vendor’s, and the section on the line said why the first rise lands early. And the curve’s second step sits at the size the vendor gives for the *last* level, not the second: the level between them, which the vendor lists at a quarter of that size, produced no step of its own. At this instrument’s resolution the curve could not separate that level from the one behind it, so the measurement reports one step where the datasheet lists two.

Back to the two routes

Now the prediction can be judged.

`ch23`’s sequential route walks an array in order: every line is used completely before the next is touched, the pattern is exactly what a prefetcher is for, and the loads do not depend on each other, so the machine can have many in flight at once. Its chased route reads one pointer per cell, in an order chosen to defeat prediction, and each load’s *address* comes from the previous load’s *result* — so no amount of parallelism helps, and each step waits for a full latency at whatever level the data happens to be at.

The structural model saw one extra load and predicted a factor under two. What separates them is not the load count. It is that one program can overlap its memory accesses and the other cannot, which is not visible in the instruction stream at all.

That is the lesson **Part V** exists for, in its general form: **the cost of a memory access is not a property of the access. It is a property of what else the machine was able to do at the same time.**

What we measured

Latency against working-set size, latency against stride, and latency against the number of pages touched — each a dependent chase, each reported as a minimum over repetitions for **ch24**'s reason, and each measured on the board.

The comparison with the vendor's figures is deliberate and the rule is stated in advance: where they disagree, the measurement is what this book prints and the disagreement is what it discusses.

What this cannot tell you

Anything about a different machine. This chapter's header says it assumes a particular hierarchy, and it means it: every number here is this board's. What transfers is the method, the shape of the curves, and the reasoning — which is why the problems are about reading curves rather than about remembering sizes.

How the cache decides what to evict. The steps say how big each level is and say nothing about associativity — how many places in the cache a given address is allowed to sit — or replacement policy, both of which are measurable with more elaborate versions of the same instrument and neither of which this chapter builds. A working set that fits but crowds into too few of those places behaves like one that does not fit, and nothing here would distinguish them.

What a write costs. Every experiment reads. Writes have their own path — store buffers, write-allocate policies, and the coherence traffic **ch28** is about — and measuring reads and assuming writes behave similarly is a good way to be wrong by a large factor.

Whether the prefetcher is helping. The instrument is built to defeat it, on purpose, so that a latency is a latency. That means every number here is a *worst case*, and a real program with a predictable access pattern may never see any of them. **ch26** is where making a pattern predictable becomes a thing you do deliberately.

Whether the largest working set reached main memory. The levels curve stops stepping at the last size tried, and nothing in it says whether that final plateau is a cache or the memory behind it. A working set larger than every cache should step again, higher; the sizes tried here did not show it, and the reason — a set still small enough, a pattern the prefetcher could follow after all, or something else — is a question for the board rather than for this page.

Problems

Three, in `tests/the_memory_hierarchy/hierarchy.c`, graded against curves the tests construct with known answers.

25.1 — Where does the curve step? Four curves, including one that drifts upward without stepping. A threshold that calls that a step finds levels in every machine, including ones that do not have them.

```
python3 -m pytest tests/the_memory_hierarchy/test_problem_1_steps.py
```

25.2 — What is the line size? The stride at which consecutive visits stop sharing. One of the curves is flat throughout, and the right answer there is that there is no answer.

```
python3 -m pytest tests/the_memory_hierarchy/test_problem_2_line.py
```

25.3 — How many lines does this loop touch? The arithmetic behind the curve, including the ceiling: past one line per element, a larger stride buys the machine nothing back.

```
python3 -m pytest tests/the_memory_hierarchy/test_problem_3_lines.py
```

Where to go next

The BCM2712's documentation RASPBERRY PI LTD (N.D.) and the Cortex-A76 technical reference manual ARM LTD (N.D.) give the figures this chapter measures. Read them **after** running the experiments, in that order, and treat any disagreement as interesting rather than as an error — a datasheet describes a design and a measurement describes a chip.

ch26 is the other half of this. Knowing where the data is tells you what a program costs; the next question is what the compiler will do about it unasked, and what it will never do however obvious it looks.

26 · Optimising Code

CHAPTER HEADER

Target	host — the reference machine, natively
Answers the cost of	ch14
Prerequisites	ch25
What it measures	What the compiler makes of five hand-optimisations: <code>bench/results/loops-aarch64.json</code> ; and what each then costs on the board: <code>bench/results/loops-host.json</code>

The question

What will the compiler do for me, and what will it never do?

[ch25](#) ended by saying that what a memory access costs depends on what else the machine could do at the same time. Given that, what is worth changing in the source? Start with the changes the compiler is going to make anyway.

The material

Five ways to write one loop

`sysfs/lib/loops.c` contains the same loop five times. All five compute the same result from the same input; they differ in which hand-optimisation has been applied.

The plain version recomputes a bound each time round and multiplies in the body. The hoisted version lifts the bound into a local. The reduced version replaces the multiply with an addition — strength reduction, done by hand. The unrolled version does four iterations per pass. And one does all three at once.

That is the list a careful programmer reaches for when they do not trust the compiler. Here is what the compiler made of it:

Written as	-O0	-O2	-O3
<code>everything</code>	102	44	60
<code>hoisted</code>	31	14	14
<code>plain</code>	32	14	14
<code>reduced</code>	32	14	14
<code>unrolled</code>	92	38	54
distinct programs	4	3	3

Conditions: target `host` · `aarch64` cross compiler, any machine — nothing here was executed · `aarch64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0` · `-O2 -g -Wall -Wextra` at each of `-O0 -O2 -O3` · 2026-09-17 ·
Source: `bench/results/loops-aarch64.json` , code hash `15de464ef0a2a9ab` .

Three of them were the same program

Read the `-O2` column first. **Plain, hoisted and reduced compile to identical numbers of instructions.** The compiler hoisted the bound. The compiler reduced the strength of the multiply. Both hand-optimisations were correct, both were unnecessary, and the code they produced was the code the compiler was going to produce from the obvious source.

This is the most common thing that happens when people optimise C by hand, and what it costs is not run time — those three are the same program. It costs a source that is now harder to read, harder to change, and no faster, and a reader who does not know this will preserve the hand-optimisation through every future edit because it looks load-bearing.

At `-O0` they are not identical: the hand-optimisations *do* something to unoptimised code. Measuring at `-O0` and concluding that a transformation helps is a reliable way to make a program worse.

And one of them backfired

Now the unrolled rows. At `-O2` the hand-unrolled loop is not shorter than the plain one. It is substantially longer, and at `-O3` the gap widens rather than closes.

The hand-unrolled source is a **different, larger program** than the one the compiler would have produced. Its four bodies with explicit indices are harder to analyse than one body with a clean loop counter, so the compiler has less to work with — less for its own unrolling, and less for vectorisation (doing several array elements per instruction, [ch31](#)'s subject), which is what `-O3` turns up. The optimisation was applied by hand, so it could not also be applied by the compiler, and the compiler's version was better.

So, the answer to this chapter's question. **The compiler will do the local, mechanical transformations better than you will. What it will not do is change your algorithm, your data layout, or your memory access pattern** — and those are the things [ch25](#) showed dominate.

The benchmark that measured nothing

A compiler this good will delete work that nothing observes, including the work a benchmark exists to time — and then every number that benchmark reports is meaningless.

`tests/optimising_code/escapes.c` contains five functions doing identical arithmetic. They differ only in what becomes of the result: dropped, returned, stored through a `volatile`, stored to a variable outside any function that nothing reads, or used in a condition that is never true.

Two of those loops are removed entirely and three survive, and which is which is not the division most people expect. Problem 26.3 is that prediction, graded by compiling the file and counting — so the answer comes from a compiler rather than from this chapter's opinion.

The rule underneath is simple to state and easy to get wrong in practice: a compiler may remove work whose result **nothing can observe**. Returning it counts. Storing it through a `volatile` counts, which is [ch05](#)'s keyword doing the job it exists for. Storing it somewhere nothing reads does not count, however much it looks like it should.

A benchmark whose work has been removed does not report zero. It reports a very small number, and a spectacular speedup, and nothing about it looks wrong.

What this cost

Written as	Measured
everything	1.72 ns per element
hoisted	1.67 ns per element
plain	1.67 ns per element
reduced	1.67 ns per element
unrolled	1.71 ns per element

Conditions: target `host` · Raspberry Pi 5 Model B Rev 1.1 · Linux 6.18.50+rpt-rpi-2712 · cc (Debian 14.2.0-19) 14.2.0 · `-O2 -g -Wall -Wextra` · 2026-09-16 · Source: `bench/results/loops-host.json`, code hash `4d834678d5e2d16a`.

Instruction counts say whether a source change survived the compiler. They do not say what the surviving differences cost; the board does. Read this table against the one above it. The three variants that compiled to the same program cost the same, as they must. The two longer programs cost more — and by a margin far smaller than the difference in their instruction counts would suggest. That gap between the count and the cost is [ch27](#)’s subject, and the reason this chapter has claimed only what the counts can support: more instructions can run in nearly the same time.

What we measured

Instruction counts for five variants at three optimisation levels, on the reference architecture, compiled and counted rather than run. CI regenerates them on every push, so the chapter’s claims about what this compiler does are checked against this compiler continuously — and if a future version stops equalising those three, the build fails rather than the book quietly becoming wrong.

The timings are a different question, and the board answered it: the table under *What this cost* is the same five variants at `-O2`, run rather than counted.

What this cannot tell you

Whether fewer instructions is faster. It very often is not, and [ch27](#) is the chapter with the equipment to say why. Instruction count is the wrong unit for the final answer and the right unit for *this* question, which is whether the source change survived at all.

Anything about a different compiler. Every claim here is about one compiler at three named levels, recorded in the conditions line. A different one may equalise a different set, and the method — compile the variants, count, compare — is what transfers.

Where the algorithmic wins are. This chapter is entirely about local transformations, because those are the ones the compiler competes with you on. Nothing in it would tell you that a different data layout removes most of the work, which is the change that usually matters and the one no compiler will make.

Whether `-O3` is worth it. The table shows `-O3` producing more instructions than `-O2` for two variants and the same for three, and says nothing about which runs faster. `-O3` trades size for speculation about what will help; whether the trade pays is a property of the program and the machine together.

Problems

Three. The first two are predictions about this book's own stamped result; the third is graded by compiling.

26.1 — Which variants does the compiler equalise? From the source alone, group the five by whether they come out the same at `-O2`. Graded against `bench/results/loops-aarch64.json`, which CI regenerates — so a future compiler that changes its mind changes the right answer rather than making the book wrong.

26.2 — Which hand-optimisation backfired, and what does `-O3` do to it? Name the variant that costs more instructions than writing the loop plainly, and say whether raising the level helps.

These two are answered in one stub and graded by one command, because they are two readings of the same table:

```
python3 -m pytest tests/optimising_code/test_problem_1_equalised.py
```

26.3 — Which benchmark loops survive the compiler? Five functions, identical arithmetic, differing only in what becomes of the result. Two are removed. Two of the five are traps and they are traps in opposite directions.

```
python3 -m pytest tests/optimising_code/test_problem_3_escapes.py
```

Where to go next

Read the `-O2` and `-O3` sections of your compiler's own documentation — `gcc(1)` lists exactly which passes each level enables, and it is a shorter and more useful document than its length suggests. The two entries worth finding are the ones that name the transformations problems 26.1 and 26.2 are about.

[ch27](#) is the chapter this one keeps deferring to. Instruction counts cannot say whether the longer program is the slower one, because a modern core does not execute instructions one at a time, and the next chapter is about what it does instead.

27 · The CPU

CHAPTER HEADER

Target	host — the reference machine, natively
Answers the cost of	ch14
Prerequisites	ch26
Assumes	An out-of-order core (Cortex-A76) and its performance counters. An in-order core would make this easier to read; see Appendix H .
What it measures	What the core is given to work with: <code>bench/results/pipeline-shapes.json</code> ; and what it does with it: <code>bench/results/pipeline-host.json</code>

The question

What is this core doing between fetching an instruction and finishing it?

ch26 ended on a deferral: instruction counts cannot say whether the longer program is the slower one. The reason is that a modern core does not execute instructions one at a time in the order they appear, and the two things it does instead — overlapping independent work, and guessing which way a branch will go — are what decide the answer.

The material

The branch that was not there

This chapter set out to measure branch misprediction, and the first attempt failed.

`sysfs/lib/pipeline.c` contains a loop that counts how many values exceed a threshold, written with an `if`. Compile it and look:

	Instructions	Conditional branches	Branches removed
<code>sum_chain1</code>	13	2	0
<code>sum_chain2</code>	31	4	0
<code>sum_chain4</code>	42	4	0
<code>sum_chain8</code>	58	4	0
<code>count_over</code>	14	2	1
<code>count_over_calling</code>	29	6	0

Conditions: target `host` · `aarch64` cross compiler, any machine — nothing here was executed · `aarch64-linux-gnu-gcc` (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · as recorded in `pipeline-aarch64.json` · 2026-09-17 · Source: `bench/results/pipeline-shapes.json`, code hash `c18a72992d8d019c`.

The `count_over` row has a branch **removed**. The compiler saw that both sides of the `if` were cheap, and replaced the branch with a conditional increment — `cinc` on AArch64 — which computes both outcomes and selects between them with no control flow at all. There is nothing left to mispredict. A measurement of branch prediction on that loop would have produced a number, and the number would have been about something else entirely.

That is [ch26](#)'s lesson again, one chapter later and at this book's expense: what you wrote and what runs are different things, and the gap is exactly where a measurement goes wrong quietly.

The fix is to make the branch un-removable, by putting something in the taken case that the compiler cannot execute unconditionally. A call will do it: the compiler cannot compute both outcomes when one of them is “call a function that might do anything”. `count_over_calling` is that loop, it keeps its branch, and the runner refuses to stamp a result in which these two stop differing.

Problem 27.1 asks you to predict which of the two keeps its branch, before compiling. The stub says plainly that this book got it wrong.

The same additions, four different amounts of parallelism

Now the four sums. They add the same numbers with the same number of additions, differing only in how many running totals they use.

Read their instruction counts in the table above and notice that they go the wrong way: the one-accumulator version is the **shortest** program. By [ch26](#)'s measure it should be the best one.

It is the slowest, and the reason is not in the instruction stream. With one accumulator, every addition needs the result of the one before it — a *dependency chain*, each step waiting on the last — so the machine can only ever have one in flight however many execution units it has. With four, four chains proceed independently and the core can overlap them. The work is identical; what differs is how much of it can happen at once.

	Nanoseconds per element	Instructions per cycle
<code>sum_chain1</code>	0.71 ns	2.23
<code>sum_chain2</code>	0.43 ns	2.26
<code>sum_chain4</code>	0.3 ns	2.78
<code>sum_chain8</code>	0.25 ns	1.85

Conditions: `target host · Raspberry Pi 5 Model B Rev 1.1 · Linux 6.18.50+rpt-rpi-2712 · cc (Debian 14.2.0-19) 14.2.0 · -O2 -g -Wall -Wextra · 2026-09-16 · Source: bench/results/pipeline-host.json , code hash d6230b84235ca06a .`

The longest program is the fastest, and the shortest is nearly three times slower than it. That is the answer [ch26](#) deferred, and it is as blunt as it looks: on this core, for this loop, the instruction count predicts the ranking backwards.

Read the second column as well, because it says the returns are running out. Instructions per cycle rises from one accumulator to four and then falls at eight — the eighth chain is still worth having in wall-clock terms, and the core is getting less done per cycle while it runs it.

Problem 27.2 is the arithmetic: the critical path is each accumulator's share of the elements plus the tree that combines them. Do it, then notice what the model predicts — that more accumulators are always better, right down to one element each. They are not, and what stops it is nowhere in the formula: accumulators live in registers, there is a fixed number of those, and past that point they spill to memory and [ch25](#) takes over.

Guessing, and what a wrong guess costs

A core that wants to keep many instructions in flight cannot wait to find out which way a branch goes. It guesses, proceeds, and — when the guess was wrong — throws away everything it did since.

Data	Mispredicted	Per element
100% predictable	0.0%	1.25 ns
90% predictable	3.2%	1.95 ns
75% predictable	7.6%	2.84 ns
50% predictable	13.4%	3.94 ns
25% predictable	16.0%	4.42 ns
0% predictable	16.2%	4.48 ns
derived cost of one mispredict		19.7 ns

Conditions: target `host` · Raspberry Pi 5 Model B Rev 1.1 · Linux 6.18.50+rpt-rpi-2712 · cc (Debian 14.2.0-19) 14.2.0 · -O2 -g -Wall -Wextra · 2026-09-16 · Source: `bench/results/pipeline-host.json`, code hash `d6230b84235ca06a`.

Two things in that table are worth reading before the last row. The time per element climbs steadily as the data becomes less predictable, which is the expected shape. But the mispredict *rate* does not reach anything like half: data the program picks at random has the predictor wrong about one time in six, not one in two, because the predictor is not guessing blind — it is doing much better than chance on data that contains no pattern the chapter put there. And the last two rows barely differ, so a predictor fed data chosen to be *wrong* every time does no worse than one fed noise.

The cost of one mispredict is **derived**, not measured, and this chapter says so in the table itself. Nothing counts nanoseconds-per-mispredict directly; what is measured is the time per element at several predictability levels and the misprediction rate at each, and the cost is the slope between them. A derived quantity is a fine thing to publish and a bad thing to publish without the word “derived” next to it — and this one rests on the rate column, so whatever makes that column flatten at the bottom is inside the number.

The predictor itself is a small saturating counter per branch, and problem 27.3 is simulating one. The alternating pattern is the case that explains the design: a one-bit predictor gets *every* branch of `TNTNTN` wrong, because it always predicts what happened last time, and two bits fixes exactly that by requiring two surprises before it changes its mind.

Counters that are not counters

One warning specific to reading a PMU, and it applies to every chapter after this one.

Some events a performance monitoring unit reports are counted by hardware. Others are computed from the ones that are — `perf` will happily report an “IPC” (instructions per cycle) or a “miss rate” that is a division of two other counters, and a “stall cycles” figure that is a subtraction. Those are derived, they inherit the error of both inputs, and on some cores they are documented as approximate.

The rule this book follows is the one the table above uses: say which. A number that is counted and a number that is computed are both useful and they are not the same kind of thing, and the difference matters most exactly when the numbers are surprising.

What we measured

The shapes: how many instructions each variant compiles to, how many conditional branches survive, and how many the compiler replaced with arithmetic. All of it compiler output, all of it regenerated by CI, and enough on its own to establish this chapter’s first finding.

The timings and the counter readings were measured on the board — time per element for four accumulator counts with instructions per cycle beside each, and time per element against misprediction rate for six kinds of data — and [ch24](#)’s discipline applies to every one of them.

What this cannot tell you

Anything about an in-order core. The header says this chapter assumes an out-of-order one, because that is what the reference machine has and because [Appendix H](#) explains at length why the alternative could not be used. Little is lost by it: every machine a reader is likely to optimise reorders, so attributing cycles on a core that reorders them is the skill that transfers.

How wide the machine is. The accumulator experiment never reached the point where another chain stopped helping — eight was still the fastest of the four tried — so it puts a floor under how much this core can overlap and no ceiling. The vendor’s manual *ARM LTD (N.D.)* has the issue width; the measurement has the consequence, and they answer different questions.

Where the mispredicted work went. A wrong guess costs the work thrown away, and nothing here says how far down the pipeline it got before being thrown. That depends on the depth of the pipeline, which is not directly observable at all and is inferred from exactly the kind of slope this chapter labels “derived”.

Whether the compiler will do any of this for you. [ch26](#) established that it does the local transformations better than you will. Breaking a dependency chain by adding accumulators is a change to the *arithmetic* — a different order of additions — and a compiler may only do that when told the reordering is permitted. Which is a flag, and a decision, and not the default.

Problems

Three.

27.1 — Which of these loops still contains a branch? Predict before compiling, and be graded against the compiler. The stub records that this book predicted wrongly.

```
python3 -m pytest tests/the_cpu/test_problem_1_branchy.py
```

27.2 — How long is the critical path? Each accumulator’s share plus the combining tree. Then notice what the model predicts and what actually stops it.

```
python3 -m pytest tests/the_cpu/test_problem_2_path.py
```

27.3 — How many does a two-bit predictor get wrong? Simulate the saturating counter. The alternating sequence is the one that explains why two bits and not one.

```
python3 -m pytest tests/the_cpu/test_problem_3_predictor.py
```

Where to go next

The Cortex-A76 technical reference manual ARM LTD (N.D.) has the pipeline description and the list of PMU events, and its event table marks which are architectural and which are implementation-defined — which is the distinction the last section of this chapter is about.

`perf list` on the board is the other half of that: it names the events this particular kernel and this particular chip agree exist, which is a smaller set than the manual's and the one you can actually use.

[ch28](#) takes the same machinery and adds more cores. Everything in this chapter assumed one thread with the whole machine to itself; the next one is about what four of them cost each other, and it is where [ch20](#)'s fences finally get a price.

28 · Memory Ordering on Real Hardware

CHAPTER HEADER

Target	host — the reference machine, natively
Answers the cost of	ch20
Prerequisites	ch27
Assumes	Four cores and this interconnect's coherence. The scaling curve moves elsewhere; the mechanism does not.
What it measures	Which counters share a cache line: <code>bench/results/sharing-layout.json</code> ; and what that sharing, and an atomic, cost on four cores: <code>bench/results/sharing-host.json</code>

The question

What do four cores cost each other, and what does a fence actually buy?

[ch20](#) established what a lock is made of and said plainly that what it *costs* is contention, which that target had no way to express. This chapter has four real cores, and it is also where the reader meets a second memory model — which is the reason this book uses two architectures.

The material

A cost with no shared data

Two threads, two counters, no shared data and no race. One of the two arrangements below costs more than twice what the other does.

	Size	Offsets	Same cache line
<code>packed</code>	16 bytes	0 and 8	yes
<code>padded</code>	128 bytes	0 and 64	no

Conditions: target `host` · `aarch64` cross compiler, any machine — nothing here was executed · `aarch64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0` · `the book's portable flags` · 2026-09-17 · Source: `bench/results/sharing-layout.json`, code hash `76da3c773e2fc037`.

The counters are the same counters. What differs is padding, and therefore whether the two land on the same cache line. The compiler and the layout settle that before anything runs, which is why this figure is measured here rather than on the board.

Coherence works in lines, not in variables. When one core writes a line, every other core's copy of that line is invalidated, and a core that wanted a *different* variable on that line has to fetch it back. Neither thread is sharing anything; they are fighting over a container that happens to hold both their things.

The name for this is false sharing, and the name is slightly misleading. The sharing is real — the line is genuinely shared — and what is false is the implication that the program meant to share anything.

	Measured
2 threads, packed	2.56 ns per increment
2 threads, padded	1.17 ns per increment
4 threads, packed	1.9 ns per increment
4 threads, padded	1.28 ns per increment

Conditions: target `host` · Raspberry Pi 5 Model B Rev 1.1 · Linux 6.18.50+rpt-rpi-2712 · cc (Debian 14.2.0-19) 14.2.0 · -O2 -g -Wall -Wextra · 2026-09-16 · Source: `bench/results/sharing-host.json`, code hash `7070f0b86e84fccc`.

The gap is widest at two threads and narrows as cores are added: more contenders do not make false sharing worse, which is the opposite of what a scaling curve would suggest. Two caches trading one line back and forth, with nothing else to overlap against, is already the expensive case.

Problem 28.1 is the analysis: given where the fields are and who writes them, which pairs will contend. The case worth getting right is two fields on one line written by the *same* thread, which costs nothing, because sharing a line is not what is at fault.

The same need, two spellings

Now the second memory model, put beside the first on purpose.

[ch20](#) printed what a release looks like on both architectures, and the two are not alike. RISC-V emits `fence rw,w` and then an ordinary store: an instruction *between* the two things being ordered. AArch64 emits `stlr` — a store that carries the ordering itself. One instruction, no fence, and a reader who learned that a barrier is something you put between two operations will not recognise it as a barrier at all.

Both satisfy the same requirement. Neither spelling is memory ordering itself.

That is why this book uses two architectures: **a reader shown one weak memory model concludes that model is memory ordering**. Shown two, they find that “weakly ordered” is a family; that a fence is an architecture-specific spelling of an architecture-independent need; and that what actually transfers is the mechanism underneath — store buffers, coherence, and the fact that another core can observe your writes in an order you did not write them in.

Problem 28.3 is that correspondence as a table, and what matters is the asymmetry between the two spellings rather than the table itself.

What ordering costs

Operation	One core	Four cores
ordered	2.16 ns	28.34 ns
plain	0.58 ns	1.83 ns
relaxed	1.58 ns	28.3 ns

Conditions: target `host` · Raspberry Pi 5 Model B Rev 1.1 · Linux 6.18.50+rpt-rpi-2712 · cc (Debian 14.2.0-19) 14.2.0 · -O2 -g -Wall -Wextra · 2026-09-16 · Source: `bench/results/sharing-host.json`, code hash `7070f0b86e84fccc`.

Two columns because the difference between them is most of what there is to know. An atomic operation on a line nobody else wants is nearly free: the core already owns the line exclusively and the operation is local. The same operation on a line four cores are all writing costs the coherence traffic to take the line away from whoever had it, every time — and the table puts that at more than an order of magnitude.

So “atomics are expensive” is not a fact about atomics. It is a fact about contention, and the same instruction has two costs that differ by a large factor depending on something that is not in the instruction.

Now read the rows against each other, because they answer the second half of this chapter’s question. Uncontended, the ordered atomic costs measurably more than the relaxed one: that difference is what the ordering buys, and it is the only place in this book where a fence has a price of its own. Contended, the two are the same to within the width of the measurement. **The ordering is free exactly when the contention is not**, because once the line has to be fetched from another core, waiting for it has already done everything the ordering was going to.

Which makes the plain row the one to be careful with. An ordinary non-atomic increment is cheaper than either atomic in both columns, and on four cores it is cheaper by a very large factor — and it is also wrong, for [ch20](#)’s reason. That row is what the lost update costs you: nothing, until it costs you everything.

Predict before you measure

[ch24](#) asked for a distribution instead of a number. This chapter asks for something else first: a **prediction**.

Amdahl’s law says what a scaling curve can look like at best. If a quarter of the work cannot be parallelised, four cores buy about two and a quarter times, and no number of cores buys four ever. That is a ceiling computed from the program, before the machine is involved at all.

A measured curve on its own tells you very little. A measured curve that falls short of a predicted one tells you there is something to find — and false sharing, atomic contention and coherence traffic are exactly the things that live in the gap. Problem 28.2 is the prediction. This chapter’s tables give two thread counts rather than a curve, which is enough to show the direction and not enough to fit anything to; the curve is yours to take, on your own machine, and the prediction is what makes it worth taking.

What we measured

Where the fields land: that decides whether the cores will fight, and finding it needs no machine. The fights themselves were then measured on the board — two layouts at two thread counts — along with three kinds of increment, each on one core and on four.

The runner refuses to stamp a layout in which the packed structure’s counters have stopped sharing a line or the padded one’s have started, because either would leave this chapter demonstrating nothing while still producing a table.

What this cannot tell you

Anything about a different interconnect. The header says four cores and this coherence fabric, and it means it. The scaling curve is this board’s; the mechanism — invalidate on write, fetch on read, in units of lines — is not.

Whether your program has false sharing. The analysis here is over a layout you can see. Real programs share lines through allocators, through arrays of small structures handed one per thread, and through padding decisions made in libraries. Finding it in something you did not write is [ch30](#)’s equipment.

What the memory model permits. This chapter measures what the machine *does*, and a weak memory model is a statement about what it is *allowed* to do. Those are different, and the difference is dangerous: a reordering that never happens on this chip may be permitted, and code that relies on not seeing it is broken on a chip that does. [ch20](#)’s problem 20.2 is about the permission, deliberately, and no measurement can replace it.

The cost of getting it wrong. Every fence in this chapter is correct. What an incorrect one costs is not a number — it is a program that works for a year and then does not, on a machine nobody has yet.

Problems

Three, in `tests/memory_ordering_on_real_hardware/sharing.c`.

28.1 — Which fields will two cores fight over? Given offsets and writers, count the contending pairs. Two fields on one line written by the same thread are not one of them.

```
python3 -m pytest tests/memory_ordering_on_real_hardware/test_problem_1_pairs.py
```

28.2 — What does the curve look like before you measure it? Amdahl’s law, applied. Do this before the board reports: a prediction you fail to reach is information, and a measurement with nothing to compare it against is much less.

```
python3 -m pytest tests/memory_ordering_on_real_hardware/test_problem_2_scaling.py
```

28.3 — Same requirement, two architectures. Give the spelling each one uses. The table is small; the observation that one of them has no separate instruction at all is the reason the problem exists.

```
python3 -m pytest tests/memory_ordering_on_real_hardware/test_problem_3_spelling.py
```

Where to go next

The ARM architecture reference manual’s memory-model chapter [ARM LTD \(N.D.\)](#) and the RISC-V unprivileged specification’s RVWMO chapter [RISC-V INTERNATIONAL \(N.D.\)](#) are worth reading in the same sitting, in either order. They describe the same kind of object with different vocabulary, and the correspondence is much easier to see when the two are half an hour apart than when they are years.

[ch29](#) returns to [Part IV](#) with the same equipment. Everything xv6 demonstrated structurally — a system call, a page fault, a context switch — has a price on this machine, and the chapter puts the two accounts side by side.

29 · The OS Layer's Cost on Real Hardware

CHAPTER HEADER

Target	host — the reference machine, natively
Answers the cost of	ch16 , ch18 , ch21
Prerequisites	ch28
What it measures	The trap instruction and the call that hides it: <code>bench/results/oscalls-aarch64.json</code> ; and what the three services, the two kinds of fault and the two routes cost: <code>bench/results/oscost-host.json</code> , <code>bench/results/faultcost-host.json</code> , <code>bench/results/vdso-host.json</code>

The question

What does Linux charge for the services xv6 showed me?

Part IV took three things apart — a system call, a page fault, a context switch — and counted what each one moved. It could not do anything else: the target has no cache, no predictor and no memory latency, so a duration measured there describes a laptop. Here those three things meet a machine with a clock, and the counts become a model that can be wrong.

The material

What Part IV left

Service	What Part IV established	Where
System call	83 instructions of trap path	ch16
...of which registers moved	62	ch16
Page fault	17 for 17 first touches	ch18
Context switch	14 registers, 224 bytes	ch21

Conditions: target `xv6` · `qemu virt (-machine virt -bios none)` · QEMU emulator version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.18) · `xv6-riscv @ 9e3161a` · `riscv64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0` · `xv6's own CFLAGS` · 2026-09-17 · Source: `bench/results/traps-xv6.json`, code hash `f78d829cb10e7c2c`.

Every figure in that table was counted rather than timed, and each is still true — an instruction count is a property of the kernel as built, not of the machine that ran it. What they are not is a cost. They are a *floor*: the work has to happen, so the call cannot be cheaper than executing it, and problem 29.2 turns the count into that bound.

A bound is worth having because it says something whichever way the measurement falls. Come in under it and the model is wrong — some of those instructions are not on the path, or are not being executed. Come in a little over it and the model explains the cost. Come in far over it and the model was never the expensive part, and the interesting question becomes what the rest is.

A trap, and the call that hides it

Here is a system call written as the instruction it is.

sysfs_raw_getpid compiled for aarch64 at -O2 — the `host` target's instruction set, 3 instructions.

```
0000000000000000 <sysfs_raw_getpid>:
 0:  mov    x8, #0xac                // #172
 4:  svc    #0x0
 8:  ret
```

Conditions: target `host` · aarch64 cross compiler, any machine — nothing here was executed · aarch64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · -O2 -Wall -Wextra -c · 2026-09-17 · Source: [bench/results/oscalls-aarch64.json](#), code hash `8b0e58e6e550e9fc`.

Three ideas and nothing else: put the call number where the kernel's convention says to put it, execute the instruction that changes privilege level, come back. There is no stack frame, because by [ch14](#)'s rule a function that calls nothing needs none — and this function does not call anything. It traps.

Note which convention that is. The register holding the call number is not one the C calling convention would have chosen, and it could not be: the process and the kernel were compiled separately, by different people, at different times, so they cannot have agreed by being compiled together. They agree by specification instead, which is the reason a system call is not a function and cannot be made into one.

Now the same request, written the way anybody would write it.

sysfs_libc_getpid compiled for aarch64 at -O2 — the `host` target's instruction set, 6 instructions.

```
0000000000000010 <sysfs_libc_getpid>:
10:  stp    x29, x30, [sp, #-16]!
14:  mov    x29, sp
18:  bl     0 <getpid>
1c:  sxtw   x0, w0
20:  ldp    x29, x30, [sp], #16
24:  ret
```

Conditions: target `host` · aarch64 cross compiler, any machine — nothing here was executed · aarch64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · -O2 -Wall -Wextra -c · 2026-09-17 · Source: [bench/results/oscalls-aarch64.json](#), code hash `8b0e58e6e550e9fc`.

A frame, a branch, and a sign-extension to widen what the kernel returned. The trap is not in this listing at all. It is somewhere past that branch, in code shipped with the machine, and nothing at the call site distinguishes it from a call that never leaves the process.

Cost in this layer is not visible in the code. Two calls that look identical in C can differ by whether a privilege boundary is crossed, and the compiler will not tell you, because the compiler does not know either.

Timing something as small as a trap

[ch24](#) built the clock and, more usefully, measured what reading it costs. This is the chapter where that second number decides whether a measurement means anything.

The trap is small. If the thing being timed is of the same order as the instrument, a harness that reads the clock on both sides of every call is charging the instrument to every call, and no number of iterations divides it away — it is added, not amortised. Read the clock once before the loop and once after, and the same overhead is divided by however many iterations there were.

Problem 29.1 is that arithmetic, and arithmetic rather than a rule because at a single iteration the two arrangements give the same answer. Putting the clock outside the loop does not remove the overhead. It divides it, and dividing by one is not a saving.

The cost of this mistake is not a slightly wrong number. It is a confident, stable, reproducible measurement of `clock_gettime`, which will look exactly like a measurement of whatever was in the loop and will not change when the loop's contents do.

What the three cost

Service	Measured	Against	Ratio
<code>fault</code>	636.0 ns	a touch of a page already mapped	×30.3
<code>switch</code>	2233.0 ns	the same round trip without leaving the thread	×8.3
<code>syscall</code>	166.31 ns	an empty function call	×396.0

Conditions: `target host · Raspberry Pi 5 Model B Rev 1.1 · Linux 6.18.50+rpt-rpi-2712 · cc (Debian 14.2.0-19) 14.2.0 · -O2 -g -Wall -Wextra · 2026-09-16 · Source: bench/results/osc cost-host.json , code hash 52aa592ba783a4c7`.

Every row has a baseline beside it, because a duration on its own is not a cost. Knowing what a system call takes is not useful until it is beside the cheapest thing the machine can do, and it is the ratio that survives being read on a different machine two years from now.

Now settle the bound. Problem 29.2 turns the trap path's instruction count into a floor, and the measured system call is far above whatever floor you compute from it — not a little over, which would mean the model explains the cost, but several times over. So the third case above is the one that happened: **executing the trap path is not what a system call mostly costs**. The registers moved, the privilege change and the page-table switch are all real and all counted, and something outside that count is larger than all of them together. This chapter does not find out what; [ch30](#) is the equipment for asking a machine where its time went.

The baselines are chosen to be unflattering. A system call compared against an empty function call; a fault compared against a write to a page that is already there; a context switch compared against the same work done without leaving the thread. In each case the comparison is between doing the thing and not doing it, which is the only comparison that answers “what did the kernel cost me”.

Read the ratio column rather than the durations, and read it knowing the baselines differ. The system call's ratio is the largest by a wide margin and the switch's the smallest, which is not a ranking of the three services: it is a statement about what each is being compared against. An empty function call is the cheapest thing in this book, so anything measured against it looks enormous. The switch's baseline already includes real work. Change the baselines and the ranking changes, which is exactly why each row names its own.

Two faults that are not the same fault

Fault	Measured	Satisfied from
none	21 ns	nothing — it is mapped and resident
minor	660 ns	a zeroed page the kernel already had
major	46223 ns	storage

Conditions: target host · Raspberry Pi 5 Model B Rev 1.1 · Linux 6.18.50+rpt-rpi-2712 · cc (Debian 14.2.0-19) 14.2.0 · -O2 -g -Wall -Wextra · 2026-09-16 · Source: bench/results/faultcost-host.json, code hash 52aa592ba783a4c7.

ch18 counted faults and made the case that a fault is a feature — the mechanism by which a page arrives only when it is wanted. It counted them because that target could not price them: the interesting difference between two faults is where the data came from, and QEMU’s storage is a host file.

Both faults in that table enter the kernel by exactly the path ch16 traced, and leave it the same way. Nothing about the trap differs. What differs is whether the kernel could answer from memory it already had or had to go and ask storage — and that one difference is worth more than everything else in this chapter put together. Read the three rows as a ladder: an access that does not fault, a fault the kernel answers from memory it already had, and a fault that has to wait for storage. Each step up is a large multiple of the one below it, and the second step is by far the larger. Problem 29.3 is the classification: four facts about an address, and the order the rules apply in.

The order is where the difficulty is. A page the process never asked for is fatal whatever the other facts say; a translation that already exists costs nothing whatever else is true; and only then does the minor-against-major question arise at all.

The call that does not trap

Call	Route	Measured
clock_gettime (vdso)	vdso	35.01 ns
getpid (trap)	trap	166.31 ns
clock_gettime (trap)	trap	205.07 ns

Conditions: target host · Raspberry Pi 5 Model B Rev 1.1 · Linux 6.18.50+rpt-rpi-2712 · cc (Debian 14.2.0-19) 14.2.0 · -O2 -g -Wall -Wextra · 2026-09-16 · Source: bench/results/vdso-host.json, code hash 52aa592ba783a4c7.

Some system calls are not system calls. The kernel maps a page of its own code into every process — the **vDSO** — and a few requests, reading the clock being the one that matters here, are served by running that code with no privilege change at all. The answer comes from memory the kernel keeps up to date, and the process never traps.

The table forces the comparison the usual way round, by making the same request both ways. The same call, the same answer, and the trap route costs several times what the vDSO route does. Notice also that the trapping clock_gettime costs more than the trapping getpid beside it: getpid is the floor of the boundary, and anything the kernel actually does after the trap is charged on top of it.

This closes a loop opened five chapters earlier. ch24’s clock is cheap enough to time things with because of this mechanism; a clock that trapped would be an instrument of the same order as much of what Part V measures, and most of this book’s timings would be impossible to take in the form they are taken.

That puts a price on the listing above: two calls, indistinguishable in C, one of which crosses a privilege boundary and one of which does not. You cannot read that off the page. You can only measure it.

What we measured

The trap instruction and the call that hides it, compiled for the reference architecture. That listing needs no board — an instruction sequence is a property of the compiler — and it is the one figure here that CI regenerates and diffs on every push.

Everything with a duration in it was measured on the machine: what the three services cost against their baselines, what separates a minor fault from a major one, and what the vDSO saves. The counts they are compared against are already in the book, taken in [Part IV](#) — which is what all that counting was for.

What this cannot tell you

What a system call costs. It cannot, and no measurement can. `getpid` is chosen precisely because the kernel does almost nothing after the trap, so what the table reports is the floor — the cost of the boundary itself. A `read` that returns a megabyte is mostly not boundary cost, and dividing this chapter’s figure into it would explain none of it.

What it costs your program. These calls are made in a loop, with the caches and the TLB warm and the branch predictor already trained on the path. A call made once, in the middle of other work, also evicts what that work had cached, and the eviction is charged to the code that runs next rather than to the call. Finding that in a real program is [ch30](#)’s equipment.

Anything about a differently configured kernel. The same source, built with different hardening options or booted with different mitigations, is a different measurement. This book takes one, on one kernel, and records which — it does not survey them, and a reader who needs the comparison has to take it themselves.

What xv6 charges. [Part IV](#)’s counts stay counts. There is no version of this chapter that prices the kernel that the reader can stop mid-trap, because the machine it runs on is a program, and that is the trade the two targets were chosen to make.

What the rest of a system call is. The bound says what the call cannot beat and the measurement came in far above it, so this chapter has shown that [Part IV](#)’s account is incomplete without showing what is missing. Pipeline effects at a privilege change, a cold predictor on the kernel’s side, the TLB, the mitigations a modern kernel carries on that path — each is a candidate and none of them is measured here. The instruction count is not where the answer will be found, which is most of what this chapter establishes.

Problems

Three, in `tests/the_os_layers_cost/oscost.c`.

29.1 — Where does the clock go? Two harnesses, identical except for where they read the clock, and what each reports. Do this before writing any timing code in the chapters that follow.

```
python3 -m pytest tests/the_os_layers_cost/test_problem_1_instrument.py
```

29.2 — What can it not be cheaper than? Turn [ch16](#)’s instruction count into a floor, given an IPC and a clock. Divide once at the end: a bound that has been rounded twice is an estimate.

```
python3 -m pytest tests/the_os_layers_cost/test_problem_2_bound.py
```

29.3 — Which fault is this? Four facts about an address and a stated precedence. The minor-against-major split is the one worth the largest factor in this chapter, and it is not a fact about the fault.

```
python3 -m pytest tests/the_os_layers_cost/test_problem_3_faults.py
```

Where to go next

The AArch64 exception model is specified in the ARM architecture reference manual ARM LTD (N.D.), and reading its description of what `svc` does beside [ch16](#)’s account of `ecall` is the fastest way to see which parts of a trap are architecture and which are xv6. The system-call numbering the listing above uses is Linux’s own, in `include/uapi/asm-generic/unistd.h` in the kernel tree.

[ch30](#) stops assuming you know which code to look at. Everything measured so far has been code this book wrote, in a loop chosen to isolate one mechanism; the next chapter is about finding the expensive part of a program nobody here has read.

30 · Whole-Machine Profiling

CHAPTER HEADER

Target	host — the reference machine, natively
Prerequisites	ch29
Assumes	that <code>perf</code> can sample. ARM PMUs support counter-overflow interrupts as standard, so this works on the reference machine — but most affordable RISC-V cores do not, and a reader following Part V on one will find this the chapter they cannot run.
What it measures	What the program under the profiler does, counted before anyone times it: <code>bench/results/tally-census.json</code> ; and where the samples landed, before and after: <code>bench/results/profile-host.json</code>

The question

How do I find the bottleneck in something I did not write?

Every measurement in this book so far has been of code the book wrote, in a loop chosen to isolate one mechanism, with the answer known before the machine was asked. That is the right shape for learning what a cache is. It is the wrong shape for the situation this is all for: a program you did not write, doing something you do not fully understand, too slowly.

The material

The program, and what it is about to do

`sysfs/tools/tally.c` reads a stream of records, turns each into a table index, and counts them. That is the whole description you get, and it is deliberately more than you would usually have.

Before profiling it, write down what it does. Not what it costs — what it *does*, counted.

Counted before the profile	Value
Records	1,048,576
Counters in the table	1,048,576 (4,194,304 bytes)
Counters the keys actually reach	663,430
Cache lines they reach	65,536 — every line in the table
Decode's conditional, taken	1,044,530 of 1,048,576
Table in play at once, partitioned	8,192 (32,768 bytes)
Key traffic partitioning adds	3,145,728 reads, 1,048,576 writes

Conditions: target host · aarch64 cross build, any machine — run under user-mode QEMU for counts only · aarch64-linux-gnu-gcc · -O2 -g -Wall -Wextra -static · 2026-09-17 · Source: bench/results/tally-census.json, code hash e19cdb43c99e15cf.

Three of those rows are worth staring at.

The keys do not reach every counter in the table, and yet they reach **every cache line in it**. A line holds a run of adjacent counters, so the untouched entries buy nothing at all: misses are counted in lines, and there are no unvisited lines. This is [ch28](#)'s observation about coherence reappearing as an observation about capacity, and both are the same fact — the machine deals in lines, and your data structure does not know that.

The conditional in the decode phase is taken almost every time. It is the most conspicuous thing in the source of that function, and [ch27](#) already established what a branch this lopsided costs a predictor.

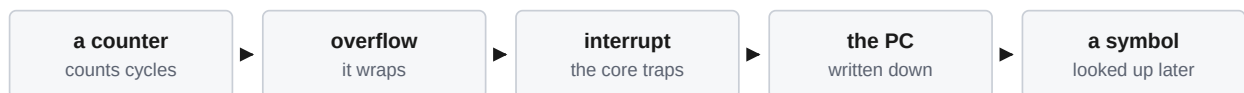
And the partitioned arrangement re-reads the keys, which adds substantial extra traffic, in order to shrink the part of the table in play at any one time. Whether that trade pays is not a question the census can answer. That open question is why there is a second arrangement at all.

The census is a prediction, and writing it down first is the method. A profile with nothing committed to beforehand is remarkably easy to agree with: whatever it blames becomes what you expected all along, and the exercise teaches nothing. `bench/run_profile.py` refuses to stamp a census in which the two arrangements stop computing the same answer, or the scattered pass stops reaching every line, or the conditional stops being lopsided — because any of those would leave the chapter pointing confidently at the wrong phase.

What a profiler actually does

A profile is a sample of where the program was, not a record of where it went

Nothing watches the program. Something interrupts it, and writes down an address.



The loop the samples land in

load the key	sequential — in cache
load the counter	scattered — this is the wait
add one, store it back	cannot start until the load returns
test and branch	where the sample is often written down

The interrupt arrives some instructions after the one that caused the stall.

So the report blames a cheap instruction standing next to an expensive one. The fix is to read the neighbourhood, never the line. The loop above is the one the compiler emitted for this book's scatter pass, 45 instructions in total.

The address written down is the one the core had reached when the interrupt arrived, and not the one that was waiting.

Nothing watches your program. A counter in the PMU is set to a large negative number, the core counts cycles into it, it overflows, and the overflow raises an interrupt — which is [ch19](#)'s mechanism, in hardware you have already met, doing a job that has nothing to do with a device. The handler writes down where the program was and returns. Later, addresses are matched against symbols.

Everything a profile can and cannot tell you follows from those sentences.

It is statistical, so a symbol holding a small share of the time has a count that is mostly noise — problem 30.2 is how many samples a claim needs before it is worth making, and the answer is often more than the profile you are looking at contains.

It is attributed to an address, so what you get is *exclusive* time: the samples that landed in each function itself. The symbol at the top of that list is very often a leaf you did not write and cannot change. Problem 30.1 is putting the callers back together, and the reason it matters is that the thing you can change is never the leaf.

And the address written down is the one the core reached when the interrupt was taken, which is not the one that was waiting.

Skid, and why the line is a lie

`sysfs_tally_scatter` compiled for aarch64 at `-02` — the `host` target's instruction set, 45 instructions.

```

0000000000000090 <sysfs_tally_scatter>:
90:  stp     x29, x30, [sp, #-48]!
94:  mov     x6, x2
98:  mov     x29, sp
9c:  stp     x19, x20, [sp, #16]
a0:  mov     x20, x0
a4:  mov     x19, x1
a8:  stp     x21, x22, [sp, #32]
ac:  mov     w22, w3
b0:  cbz     w3, 12c <sysfs_tally_scatter+0x9c>
b4:  mov     x0, x6
b8:  mov     w21, w3
bc:  ubfiz   x2, x22, #2, #32
c0:  mov     w1, #0x0                                // #0
c4:  bl      0 <memset>
c8:  mov     x6, x0
cc:  cbz     x19, fc <sysfs_tally_scatter+0x6c>
d0:  mov     x4, #0x0                                // #0
d4:  nop
d8:  ldr     w2, [x20, x4, lsl #2]
dc:  add     x4, x4, #0x1
e0:  ldr     w5, [x6, x2, lsl #2]
e4:  add     w5, w5, #0x1
e8:  str     w5, [x6, x2, lsl #2]
ec:  cmp     x19, x4
f0:  b.hi    d8 <sysfs_tally_scatter+0x48> // b.pmore
f4:  cbz     w22, 130 <sysfs_tally_scatter+0xa0>
f8:  mov     w21, w22
fc:  sub     x6, x6, #0x4
100: mov     x1, #0x1                                // #1
104: mov     x0, #0x0                                // #0
108: ldr     w3, [x6, x1, lsl #2]
10c: cmp     x21, x1
110: madd    x0, x3, x1, x0
114: add     x1, x1, #0x1
118: b.ne    108 <sysfs_tally_scatter+0x78> // b.any
11c: ldp     x19, x20, [sp, #16]
120: ldp     x21, x22, [sp, #32]
124: ldp     x29, x30, [sp], #48
128: ret
12c: cbnz    x1, d0 <sysfs_tally_scatter+0x40>
130: ldp     x19, x20, [sp, #16]
134: mov     x0, #0x0                                // #0
138: ldp     x21, x22, [sp, #32]
13c: ldp     x29, x30, [sp], #48
140: ret

```

Conditions: target `host` · aarch64 cross compiler, any machine — nothing here was executed · aarch64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · `-O2 -Wall -Wextra -c` · 2026-09-17 · Source: [bench/results/profiling-aarch64.json](#), code hash `70477bce1e82551b`.

The inner loop of the scattered pass is a handful of instructions, and the census has already told you which one will stall: the second load, the one indexed by the key, reaching into a table far too large to sit in cache. The load after it and the store after that cannot proceed until it returns.

The stall is real and it belongs to that load. The *sample* lands somewhere after it, because the interrupt is taken when the machine notices, and by then the program counter has moved on.

Instruction	Offset	Samples
d44: stp x29, x30, [sp, #-48]!	+0	0.0%
d48: mov x7, x2	+1	0.0%
d4c: mov x29, sp	+2	0.0%
d50: stp x19, x20, [sp, #16]	+3	0.0%
d54: mov x20, x0	+4	0.0%
d58: mov x19, x1	+5	0.0%
d5c: stp x21, x22, [sp, #32]	+6	0.0%
d60: mov w22, w3	+7	0.0%
d64: cbz w3, de4 <sysfs_tally_scatter+0xa0>	+8	0.0%
d68: mov x0, x7	+9	0.0%
d6c: mov w21, w3	+10	0.0%
d70: ubfiz x2, x22, #2, #32	+11	0.0%
d74: mov w1, #0x0 // #0	+12	0.0%
d78: bl 750 <memset@plt>	+13	0.0%
d7c: mov x7, x0	+14	0.0%
d80: cbz x19, db0 <sysfs_tally_scatter+0x6c>	+15	0.0%
d84: mov x5, #0x0 // #0	+16	0.0%
d88: ldr w4, [x20, x5, lsl #2]	+17	0.0%
d8c: add x5, x5, #0x1	+18	0.36%
d90: lsl x4, x4, #2	+19	0.0%
d94: ldr w6, [x7, x4]	+20	0.0%
d98: add w6, w6, #0x1	+21	92.71%
d9c: str w6, [x7, x4]	+22	0.0%
da0: cmp x19, x5	+23	0.0%
da4: b.hi d88 <sysfs_tally_scatter+0x44> // b.pmore	+24	0.0%
da8: cbz w22, de8 <sysfs_tally_scatter+0xa4>	+25	0.0%
dac: mov w21, w22	+26	0.0%
db0: sub x7, x7, #0x4	+27	0.0%
db4: mov x0, #0x0 // #0	+28	0.0%
db8: mov x1, #0x1 // #1	+29	0.0%
dbc: nop	+30	0.0%
dc0: ldr w3, [x7, x1, lsl #2]	+31	0.0%
dc4: cmp x21, x1	+32	6.39%
dc8: umaddl x0, w3, w1, x0	+33	0.18%
dcc: add x1, x1, #0x1	+34	0.0%
dd0: b.ne dc0 <sysfs_tally_scatter+0x7c> // b.any	+35	0.36%
dd4: ldp x19, x20, [sp, #16]	+36	0.0%
dd8: ldp x21, x22, [sp, #32]	+37	0.0%

Instruction	Offset	Samples
ddc: ldp x29, x30, [sp], #48	+38	0.0%
de0: ret	+39	0.0%
de4: cbnz x1, d84 <sysfs_tally_scatter+0x40>	+40	0.0%
de8: ldp x19, x20, [sp, #16]	+41	0.0%
dec: mov x0, #0x0 // #0	+42	0.0%
df0: ldp x21, x22, [sp, #32]	+43	0.0%
df4: ldp x29, x30, [sp], #48	+44	0.0%
df8: ret	+45	0.0%

Conditions: target `host` · Raspberry Pi 5 Model B Rev 1.1 · Linux 6.18.50+rpt-rpi-2712 · cc (Debian 14.2.0-19) 14.2.0 · -O2 -g -Wall -Wextra · 2026-09-16 · Source: `bench/results/skid-host.json`, code hash `fb4c63af07e67d03`.

Almost every sample in that loop landed on one instruction, and the instruction is an `add` of one to a register. Nothing about an increment is expensive on any machine in this book. What is expensive is what happened a few instructions earlier and had not finished, and the increment is merely where the core had got to when the counter overflowed.

So the rule for reading a profile at instruction granularity is to read the neighbourhood and never the line. An instruction with no reason to be expensive, sitting shortly after one with every reason, is not a mystery. It is the shape of the measurement.

Aliasing, and why it is worse than noise

There is a second way a profile misleads, and unlike skid it leaves no trace.

A profiler sampling at a fixed period, over a loop of fixed length, does not sample the loop evenly. The sample positions walk around the loop in steps of the sampling period, and that walk does not visit every position — it visits the multiples of a number that problem 30.3 asks you to find. When the two periods share a large factor, the profile lands on a handful of positions out of thousands, every time, on every run, reproducibly.

This is worse than a noisy profile, because a noisy profile looks noisy. An aliased one looks like a finding. It is stable across runs, it is consistent between machines with the same timing, and it will tell you with great confidence that one instruction in a loop is responsible for everything.

The fix is not to choose the period more carefully; there is no period that is coprime to every loop. It is to make the period slightly random, which real profilers do, and which is worth knowing about because it is the reason the tool's defaults are not round numbers.

The two profiles, before and after

Symbol	Before	After
<code>sysfs_tally_scatter</code>	99.08%	0.0%
<code>0x00000000000a4358</code>	0.89%	6.39%
<code>sysfs_tally_partitioned</code>	0.0%	93.55%

Conditions: target `host` · Raspberry Pi 5 Model B Rev 1.1 · Linux 6.18.50+rpt-rpi-2712 · cc (Debian 14.2.0-19) 14.2.0 · -O2 -g -Wall -Wextra · 2026-09-16 · Source: `bench/results/profile-host.json`, code hash `fb4c63af07e67d03`.

The change is the second arrangement: partition the keys so that every increment lands in a slice of the table small enough to stay resident, at the cost of reading the keys twice more and writing them once more.

The blame moved completely. Before, one symbol held essentially all of the samples; after, that symbol holds none and its replacement holds nearly all of them. A profile could hardly be more emphatic.

And it does not say whether the program got faster. A profile reports *shares*, and shares sum to one whatever the total is: a program that took twice as long would produce this same table if its time were distributed the same way. Reading that emphatic a move as a result is the mistake this whole part is arranged to prevent — the number that would settle it is a duration, taken [ch24](#)'s way, before and after, and this chapter did not take one. The census predicted where the time would move and the profile confirms it moved there. Whether the trade paid is a separate measurement, and it has not been made here.

What we measured

The census: what the program touches, counted, in units that do not depend on any machine. CI re-runs it on every push and refuses three specific ways it could stop describing the chapter. Alongside it, the disassembly of the scattered inner loop, which is what makes the skid discussion concrete rather than a warning.

Measured on the board: the profiles themselves, before and after, and the instruction-level samples that show where the skid put them. All of it is shares of samples; no duration was taken, which is why the section above declines to say which arrangement is faster.

What this cannot tell you

Where the time goes in a program that is not CPU-bound. Sampling on cycles finds code that is executing. A program waiting on a disk, a socket or a lock is not executing, and a cycle-sampled profile of it will be a beautifully detailed picture of the small part that was. The counters for that are different ones and the tooling is a different chapter.

Anything about a machine whose PMU cannot interrupt. The header says this and means it. The counting parts of `perf` work on far more hardware than the sampling parts; the census in this chapter was designed to be useful to a reader in that position, and the profile cannot be.

Whether the symbol is the cause. A profile is a map of where time was spent, which is a different question from what is responsible for spending it. The scattered pass is expensive because of a decision made in the decode phase about how keys are distributed, and no amount of sampling inside the loop points at that.

How to profile something that is fast. Every technique here needs the program to run long enough to collect samples. For anything shorter, the equipment is [ch24](#)'s — repetition and a distribution — and the two do not substitute for each other.

Whether the change in this chapter was an improvement. Said above and repeated here because it is the easiest thing on the page to forget: a profile ranks, and a ranking is not a total. Every claim of the form “this made it faster” in this book rests on a duration, and this chapter has none.

What it costs to measure. The interrupt has to be taken, the handler has to run, and the sample has to be written down. At a high sampling rate that is not free, and it is charged to the program being measured. Nothing in this chapter tells you how much, because measuring the cost of a measurement with the same measurement is circular.

Problems

Three, in `tests/whole_machine_profiling/profiler.c`.

30.1 — Flat to inclusive. Given a call tree and the profiler’s exclusive counts, work out what each function cost including everything it called. The symbol you can change is never the one at the top of the flat list.

```
python3 -m pytest tests/whole_machine_profiling/test_problem_1_inclusive.py
```

30.2 — How many samples is a claim? Derive the closed form for the number of samples a share needs before it is worth believing. It is small enough to do in your head with a profile on the screen, which is the point of deriving it.

```
python3 -m pytest tests/whole_machine_profiling/test_problem_2_noise.py
```

30.3 — Why the period is random. A fixed sampling period over a fixed loop visits only some positions. Work out how many, and the reason real profilers jitter will follow from the formula rather than from being told.

```
python3 -m pytest tests/whole_machine_profiling/test_problem_3_aliasing.py
```

Where to go next

`perf_event_open(2)` LINUX MAN-PAGES PROJECT (N.D.) is the interface everything in this chapter sits on, and it is worth reading once even if you never call it: the structure it takes is a list of every decision a profiler makes on your behalf. Mytkowicz and colleagues MYTKOWICZ ET AL. (2009) is the paper to read on measurement bias — it shows profilers disagreeing with each other about the same program, for reasons that are nobody’s bug, and it is the best available argument for the habit this chapter is built on: writing the prediction down first.

[ch31](#) is the last measurement in the book and the narrowest: one loop, one unit, and the question of what vectorising actually buys when it is measured against the arithmetic rather than against the version you started with.

31 · Vectors

CHAPTER HEADER

Target	host — the reference machine, natively
Prerequisites	ch30
Assumes	a vector unit — NEON on the reference core. This chapter became measurable when Part V moved to AArch64; on a RISC-V board without RVV 1.0 it reverts to reasoning about code the compiler emits but the hardware cannot run.
What it measures	Which of five loops the compiler widens, at three sets of flags: <code>bench/results/vectors-census.json</code> ; and what the widening bought: <code>bench/results/vectors-host.json</code>

The question

What does vectorising actually buy, and when will the compiler do it for me?

[ch30](#) was about finding the expensive loop in a program nobody here has read. This chapter is what you do to that loop once you have found it, and it asks the narrowest question in the book: the unit can do four of these at once, so what became of the other three?

The material

Five loops, three builds

Loop	-O2	-O3	-O3 -ffast-math
<code>sysfs_vec_gather</code>	10 (0 vector)	10 (0 vector)	10 (0 vector)
<code>sysfs_vec_running</code>	11 (0 vector)	11 (0 vector)	11 (0 vector)
<code>sysfs_vec_scale</code>	9 (0 vector)	56 (4 vector)	56 (4 vector)
<code>sysfs_vec_sum_f32</code>	8 (0 vector)	9 (0 vector)	37 (5 vector)
<code>sysfs_vec_sum_i32</code>	12 (0 vector)	40 (5 vector)	40 (5 vector)

Conditions: target `host` · `aarch64` cross compiler, any machine — nothing here was executed · `aarch64-linux-gnu-gcc` · the book's `aarch64` listing flags, then with `-O3`, then with `-O3 -ffast-math` · 2026-09-17 · Source: `bench/results/vectors-census.json`, code hash `7aadd394c6e4d2c1`.

Read the second column first. At the book's own optimisation level — the one every other listing in this book was compiled at, and the one a reader building this code gets — **not one of these loops is vectorised**. That is the first half of the answer to “when will the compiler do it for me”, and it is not a subtlety: the default is no.

At `-O3` two of them widen. At `-O3` with permission to change the answer, a third joins them. Two never do, at any setting.

So that table holds four different reasons a loop is not vectorised, and only one of them is about the optimisation level.

The one that widens

`sysfs_vec_scale` compiled for `aarch64` at `-O2` — the `host` target's instruction set, 9 instructions.

```

0000000000000000 <sysfs_vec_scale>:
 0:  cbz     x2, 20 <sysfs_vec_scale+0x20>
 4:  mov     x3, #0x0                                // #0
 8:  ldr     s1, [x1, x3, lsl #2]
c:  fmul     s1, s1, s0
10:  str     s1, [x0, x3, lsl #2]
14:  add     x3, x3, #0x1
18:  cmp     x2, x3
1c:  b.ne    8 <sysfs_vec_scale+0x8>  // b.any
20:  ret

```

Conditions: target `host` · aarch64 cross compiler, any machine — nothing here was executed · aarch64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · `-O2 -Wall -Wextra -c` · 2026-09-17 · Source: `bench/results/vectors-o2.json`, code hash `9a4601ec65522b39`.

sysfs_vec_scale compiled for aarch64 at **-O3** — the `host` target's instruction set, 56 instructions.

```

0000000000000000 <sysfs_vec_scale>:
 0: cbz     x2, 40 <sysfs_vec_scale+0x40>
 4: sub     x3, x2, #0x1
 8: cmp     x3, #0x4
 c: b.lsl   c0 <sysfs_vec_scale+0xc0> // b.plast
10: add     x4, x1, #0x4
14: mov     x3, #0x0 // #0
18: sub     x4, x0, x4
1c: cmp     x4, #0x8
20: b.hi    44 <sysfs_vec_scale+0x44> // b.pmore
24: nop
28: ldr     s1, [x1, x3, lsl #2]
2c: fmul    s1, s1, s0
30: str     s1, [x0, x3, lsl #2]
34: add     x3, x3, #0x1
38: cmp     x2, x3
3c: b.ne    28 <sysfs_vec_scale+0x28> // b.any
40: ret
44: dup     v2.4s, v0.s[0]
48: lsr     x4, x2, #2
4c: lsl     x4, x4, #4
50: ldr     q1, [x1, x3]
54: fmul    v1.4s, v1.4s, v2.4s
58: str     q1, [x0, x3]
5c: add     x3, x3, #0x10
60: cmp     x3, x4
64: b.ne    50 <sysfs_vec_scale+0x50> // b.any
68: and     x3, x2, #0xfffffffffffffc
6c: tst     x2, #0x3
70: b.eq    40 <sysfs_vec_scale+0x40> // b.none
74: ldr     s1, [x1, x3, lsl #2]
78: add     x5, x3, #0x1
7c: lsl     x4, x3, #2
80: fmul    s1, s1, s0
84: str     s1, [x0, x3, lsl #2]
88: cmp     x2, x5
8c: b.lsl   40 <sysfs_vec_scale+0x40> // b.plast
90: add     x5, x4, #0x4
94: add     x3, x3, #0x2
98: ldr     s1, [x1, x5]
9c: fmul    s1, s1, s0
a0: str     s1, [x0, x5]
a4: cmp     x2, x3
a8: b.lsl   40 <sysfs_vec_scale+0x40> // b.plast
ac: add     x4, x4, #0x8
b0: ldr     s1, [x1, x4]
b4: fmul    s0, s1, s0
b8: str     s0, [x0, x4]
bc: ret
c0: mov     x3, #0x0 // #0
c4: ldr     s1, [x1, x3, lsl #2]
c8: fmul    s1, s1, s0
cc: str     s1, [x0, x3, lsl #2]
d0: add     x3, x3, #0x1
d4: cmp     x2, x3
d8: b.ne    28 <sysfs_vec_scale+0x28> // b.any
dc: b      40 <sysfs_vec_scale+0x40>

```

Conditions: target `host` · aarch64 cross compiler, any machine — nothing here was executed · aarch64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · `-O2 -Wall -Wextra -O3 -c` · 2026-09-17 · Source: `bench/results/vectors-o3.json`, code hash `9a4601ec65522b39`.

Independent elements, one multiply each, nothing carried between iterations. The `-O3` version does four at a time: one load of a quad-word register, one multiply across four lanes, one store.

Look at what it cost. The function is several times longer. The widening is three instructions in the middle, and everything else is getting there — checking whether the pointers overlap, working out how many full vectors there are, and handling the elements left over at the end.

That tail is the part worth carrying away. A loop shorter than one vector gains nothing at all and pays for all of that code, and a loop one element past a vector pays a whole scalar iteration for it. Problem 31.1 is the arithmetic, and the shape it produces is a sawtooth rather than a slope.

The one that is refused, and why it is right

sysfs_vec_sum_f32 compiled for aarch64 at -O3 — the `host` target's instruction set, 9 instructions.

```
0000000000000180 <sysfs_vec_sum_f32>:
180:  movi    v0.2s, #0x0
184:  cbz     x1, 1a0 <sysfs_vec_sum_f32+0x20>
188:  add     x1, x0, x1, lsl #2
18c:  nop
190:  ldr     s1, [x0], #4
194:  fadd    s0, s0, s1
198:  cmp     x1, x0
19c:  b.ne    190 <sysfs_vec_sum_f32+0x10>  // b.any
1a0:  ret
```

Conditions: target `host` · aarch64 cross compiler, any machine — nothing here was executed · aarch64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · -O2 -Wall -Wextra -O3 -c · 2026-09-17 · Source: [bench/results/vectors-o3.json](#), code hash `9a4601ec65522b39`.

sysfs_vec_sum_f32 compiled for aarch64 at -O3 -ffast-math — the `host` target's instruction set, 37 instructions.

```

0000000000000180 <sysfs_vec_sum_f32>:
180:   cbz     x1, 200 <sysfs_vec_sum_f32+0x80>
184:   sub     x2, x1, #0x1
188:   cmp     x2, #0x2
18c:   b.ls    208 <sysfs_vec_sum_f32+0x88> // b.plast
190:   lsr     x3, x1, #2
194:   mov     x2, x0
198:   movi    v0.4s, #0x0
19c:   add     x3, x0, x3, lsl #4
1a0:   ldr     q1, [x2], #16
1a4:   fadd    v0.4s, v0.4s, v1.4s
1a8:   cmp     x3, x2
1ac:   b.ne    1a0 <sysfs_vec_sum_f32+0x20> // b.any
1b0:   faddp   v0.4s, v0.4s, v0.4s
1b4:   and     x2, x1, #0xffffffffffffffffc
1b8:   faddp   v0.4s, v0.4s, v0.4s
1bc:   tst     x1, #0x3
1c0:   b.eq    1fc <sysfs_vec_sum_f32+0x7c> // b.none
1c4:   ldr     s1, [x0, x2, lsl #2]
1c8:   add     x3, x2, #0x1
1cc:   lsl     x4, x2, #2
1d0:   fadd    s0, s0, s1
1d4:   cmp     x1, x3
1d8:   b.ls    1fc <sysfs_vec_sum_f32+0x7c> // b.plast
1dc:   add     x0, x0, x4
1e0:   add     x2, x2, #0x2
1e4:   ldr     s1, [x0, #4]
1e8:   fadd    s0, s0, s1
1ec:   cmp     x1, x2
1f0:   b.ls    1fc <sysfs_vec_sum_f32+0x7c> // b.plast
1f4:   ldr     s1, [x0, #8]
1f8:   fadd    s0, s0, s1
1fc:   ret
200:   movi    v0.2s, #0x0
204:   ret
208:   movi    v0.2s, #0x0
20c:   mov     x2, #0x0 // #0
210:   b       1c4 <sysfs_vec_sum_f32+0x44>

```

Conditions: target `host` · aarch64 cross compiler, any machine — nothing here was executed · aarch64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · `-O2 -Wall -Wextra -O3 -ffast-math -c` · 2026-09-17 · Source: [bench/results/vectors-o3fast.json](#), code hash `9a4601ec65522b39`.

A sum of floats. `-O3` leaves it scalar. The same loop over integers, in the table above, widens at `-O3` without being asked.

The difference is not in the loop and not in the compiler. **Integer addition is associative and floating-point addition is not.** Widening a reduction means adding the numbers in a different order — lanes accumulating independently, partials combined at the end — and a different order can give a different answer. A compiler that did that on its own would be changing your program’s output to make it faster.

The third column grants exactly that permission. `-ffast-math` says the answer may change, and the loop widens immediately, which is the proof that the refusal was never a limitation. Hold on to this one: the last table in this chapter says what the permission bought.

Problem 31.2 is the disagreement itself. You write both orders, the test supplies an input where they differ, and you compare the bit patterns. Being told that floating-point addition is not associative is not the same as watching the two answers come out different.

The two that never widen

`sysfs_vec_running` compiled for aarch64 at `-O3 -ffast-math` — the `host` target’s instruction set, 11 instructions.

```
0000000000000240 <sysfs_vec_running>:
240:   cbz     x2, 268 <sysfs_vec_running+0x28>
244:   movi    v0.2s, #0x0
248:   mov     x3, #0x0                                // #0
24c:   fmov     s2, #5.000000000000000000000000e-01
250:   ldr     s1, [x1, x3, lsl #2]
254:   fmadd    s0, s0, s2, s1
258:   str     s0, [x0, x3, lsl #2]
25c:   add     x3, x3, #0x1
260:   cmp     x2, x3
264:   b.ne     250 <sysfs_vec_running+0x10> // b.any
268:   ret
```

Conditions: target `host` · aarch64 cross compiler, any machine — nothing here was executed · aarch64-linux-gnu-gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0 · `-O2 -Wall -Wextra -O3 -ffast-math -c` · 2026-09-17 · Source: `bench/results/vectors-o3fast.json`, code hash `9a4601ec65522b39`.

Each element needs the one before it. Lanes run at the same time, and the second lane cannot start until the first has finished — so there is nothing for the width to do, and no flag changes that. This is [ch27](#)’s dependence chain again, at a different granularity and with the same conclusion.

The other refusal is the gather: the elements are independent, but their addresses are not known until the indices have been loaded. A vector unit is fastest reading a run of adjacent memory, and this loop has no run to read. It stays scalar at every setting in the table.

A note on reading a listing for vector instructions. Counting instructions that name a vector register does not answer the question. This compiler uses a half-width vector form to make a floating-point zero, so the loop the chapter calls impossible to widen contains an instruction that looks like vectorisation and is not.

`bench/run_vectors.py` counts only full-width forms, and found this out by tripping over its own guard.

What it bought

Loop	Measured	Arithmetic bound	Of the bound
<code>sysfs_vec_gather</code>	×0.95	×4.0	23.8%
<code>sysfs_vec_running</code>	×0.98	×4.0	24.4%
<code>sysfs_vec_scale</code>	×2.97	×4.0	74.3%
<code>sysfs_vec_sum_f32</code>	×1.0	×4.0	25.0%
<code>sysfs_vec_sum_i32</code>	×1.65	×4.0	41.2%

Conditions: target `host` · Raspberry Pi 5 Model B Rev 1.1 · Linux 6.18.50+rpt-rpi-2712 · cc (Debian 14.2.0-19) 14.2.0 · `-O2 -g -Wall -Wextra` · 2026-09-16 · Source: `bench/results/vectors-host.json`, code hash `b50c0d1926e9569e`.

Each measured speedup sits beside the most its lane count could possibly have bought.

Read the two loops that never widen first, and then stop reading their last column. Their measured speedup is about one, as it must be, since nothing changed — and the percentage beside it is arithmetic rather than achievement: anything that did not move at all sits at a quarter of a fourfold bound by definition. The column means something only for the rows that widened.

Of those, the one with independent elements and a multiply each got most of the way to its bound, and the integer reduction got a good deal less — the partials still have to be combined at the end, and that combining is not four-at-a-time work.

Then the float sum, which is the finding. It widened, under permission this chapter spent a whole section justifying, and it bought nothing whatever: its measured speedup is one. The compiler's refusal at `-O3` was correct, the permission that overrode it was granted, the vectorisation happened — and the loop takes what it always took, because what limits it is not arithmetic width. Getting a program's answers changed and its time not changed is the worst trade in this book, and nothing in the census could have predicted it. Only the measurement did.

Compare each speedup with its bound, then. A speedup on its own invites you to be pleased with it; as a fraction of the bound it makes you ask where the rest went — and the answer is a tail, or memory, or a loop that was never the bottleneck to begin with. Problem 31.3 is that arithmetic, and it deliberately does not clamp: a result over the bound means something other than the width changed, and the comparison has stopped being between two versions of one loop.

What we measured

Which loops the compiler widens and at what flags, counted from the disassembly, which needs no machine and so is checked by CI on every push. The runner refuses a census in which nothing widens at `-O3`, nothing further widens under `-ffast-math`, or the loop with the dependence acquires vector instructions — each of which would leave the chapter asserting something its own evidence had stopped supporting.

Measured on the board: what the widening actually bought, beside what it was allowed to buy. The five rows include the two loops that never widen, which are there as a control — a loop that did not change should measure as not having changed, and they do.

What this cannot tell you

What a different compiler does. Every refusal in that table is this compiler's, at this version. The reason behind one of them — reassociation changes the answer — is a fact about arithmetic and will hold everywhere. The other two are judgements about cost, and a different compiler is entitled to judge differently.

What a wider or different vector unit does. NEON is a fixed width. Scalable vector extensions on both of this book's architectures express the same loops without a compile-time width, which changes the tail arithmetic in problem 31.1 fundamentally rather than in degree. The reference machine does not have one, so this book has nothing to say about it that would be worth reading.

Whether to write intrinsics. Nothing here is written in intrinsics, deliberately: a chapter about what the compiler decides cannot be written in a notation that takes the decision away from it. What that costs is that the book never shows you the ceiling — a hand-written version might beat every row of the last table, and this chapter would not know.

Whether any of this is where your time is going. [ch30](#) is the chapter for that, and the order matters. A loop vectorised to a quarter of its scalar cost, in code holding a hundredth of the running time, has bought you well under one per cent.

Problems

Three, in `tests/vectors/lanes.c`.

31.1 — What the tail costs. Speedup from a lane count and a trip count. The case to get right is the loop shorter than one vector, which gains nothing and pays for all the code.

```
python3 -m pytest tests/vectors/test_problem_1_tail.py
```

31.2 — The same numbers, two orders. Write the sequential sum and the lane-wise one, and watch them disagree. This is the refusal above, reproduced rather than described.

```
python3 -m pytest tests/vectors/test_problem_2_reassociation.py
```

31.3 — The speedup you were allowed. A measured speedup as a fraction of the arithmetic bound. Do not clamp it: the value over a hundred is the one that tells you something.

```
python3 -m pytest tests/vectors/test_problem_3_bound.py
```

Where to go next

The ARM architecture reference manual's Advanced SIMD chapter [ARM LTD \(N.D.\)](#) is the specification for what those instructions do, and the RISC-V vector extension [RISC-V INTERNATIONAL \(N.D.\)](#) is worth reading beside it for the same reason [ch28](#) put two memory models side by side: it solves the tail problem in problem 31.1 by not having a compile-time width at all, and seeing one design makes the other's choices visible as choices.

That is the last measurement in the book. [Appendix A](#) collects the reference cards, and the [preface](#) says what this book set out to make you able to do. The test is not whether you remember any of these numbers, but whether, next time something is slow, the first thing you reach for is a measurement.

Appendices

[Appendix A · RISC-V Registers and CSRs](#) · [Appendix B · gdb for Kernels and RISC-V](#) · [Appendix C · The perf Events This Board Has](#) · [Appendix D · An xv6 File Map](#) · [Appendix E · Glossary](#) · [Appendix F · AArch64 for RISC-V Readers](#) · [Appendix G · Reading xv6 Alongside This Book](#) · [Appendix H · Choosing the Machine](#)

Appendix A · RISC-V Registers and CSRs

An appendix in this book is a reference, not a chapter: no argument, no narrative, and everything in it either cites a primary source or comes from a stamped result under `bench/results/`.

Everything on this page is from the RISC-V unprivileged RISC-V INTERNATIONAL (N.D.) and privileged RISC-V INTERNATIONAL (N.D.) specifications and the psABI RISC-V INTERNATIONAL (N.D.), restated for the subset this book uses. It is not a substitute for reading them; it is what to have open while you do.

The integer registers

Thirty-two, each sixty-four bits wide on RV64. `x0` is hardwired to zero — writes to it are discarded, which is how the instruction set gets a no-op, a move and a compare-against-zero without spending encodings on them.

The names on the left are the only ones you will see in this book’s listings. The hardware has no opinion about them: the roles below are a *convention* RISC-V INTERNATIONAL (N.D.), agreed between compilers. [ch14](#) shows what the convention buys a compiler, and [ch16](#) shows what it is worth to a trap — which is nothing, because a trap is not a party to it.

Register	x number	Role	Preserved across a call?
<code>zero</code>	<code>x0</code>	Hardwired zero	—
<code>ra</code>	<code>x1</code>	Return address	No
<code>sp</code>	<code>x2</code>	Stack pointer	Yes
<code>gp</code>	<code>x3</code>	Global pointer	— (set once)
<code>tp</code>	<code>x4</code>	Thread pointer	— (set once)
<code>t0 – t2</code>	<code>x5 – x7</code>	Temporaries	No
<code>s0 / fp</code>	<code>x8</code>	Saved, or frame pointer	Yes
<code>s1</code>	<code>x9</code>	Saved	Yes
<code>a0 – a1</code>	<code>x10 – x11</code>	Arguments, and return values	No
<code>a2 – a7</code>	<code>x12 – x17</code>	Arguments	No
<code>s2 – s11</code>	<code>x18 – x27</code>	Saved	Yes
<code>t3 – t6</code>	<code>x28 – x31</code>	Temporaries	No

Two consequences of that table:

Eight argument registers. A call passing eight or fewer word-sized arguments touches no memory to do it, which is most of why [ch14](#)’s `-O2` listings have no stack frame at all.

“Preserved” means the callee promised. A caller-saved register is not saved by anybody unless the caller needs it afterwards; a callee-saved one is saved by whoever wants to use it. Neither promise binds the hardware, which is why [ch16](#)’s trap path saves all thirty-one.

The control and status registers this book touches

CSRs are a separate address space, read and written with their own instructions (`csrr`, `csrw`, `csrrw` and the immediate forms). They are not the integer registers and they are not memory.

The privileged specification RISC-V INTERNATIONAL (N.D.) defines several dozen. These are the ones this book’s code reads or writes, each with the chapter that meets it.

They come in two sets, because this book runs at two privilege levels. **Part II** has no kernel under it and runs in **machine** mode, so it uses the `m` registers directly. xv6 runs in **supervisor** mode and uses the `s` ones, which do the same jobs one level down.

Machine-mode CSR	What it is	Where
<code>mtvec</code>	The address the hardware jumps to on a machine trap	ch06
<code>mepc</code>	The address of the instruction that trapped; <code>mret</code> returns here	ch06 , ch07
<code>mcause</code>	Why the trap happened, with the interrupt bit at the top	ch06 , ch07
<code>mstatus</code>	Machine status: previous privilege in <code>MPP</code> , previous interrupt-enable in <code>MPIE</code>	ch07

`mepc` points at the instruction, not past it. [ch06](#)’s first problem turns on that, and so does the rule that a handler for `ecall` has to advance it and a handler for an interrupt must not.

Supervisor-mode CSR	What it is	Where
<code>stvec</code>	The address the hardware jumps to on a supervisor trap	ch16
<code>sepc</code>	The PC at the moment of the trap; <code>sret</code> returns here	ch16
<code>scause</code>	Why the trap happened, as a code with the interrupt bit at the top	ch16 , ch19
<code>sstatus</code>	Supervisor status: previous privilege, previous interrupt-enable	ch16 , ch19
<code>sscratch</code>	One word the trap path may use before it has a usable register	ch16
<code>stval</code>	The faulting address, for a fault that has one	ch18
<code>satp</code>	The root page table and the translation mode	ch17
<code>sie</code> / <code>sip</code>	Which interrupts are enabled, and which are pending	ch19
<code>time</code>	The real-time counter — an unprivileged CSR, read here by the supervisor-mode driver	ch19

`scause`’s top bit is the one to read first. It separates an interrupt — something outside the program asking for attention — from an exception, which is this instruction refusing to complete. The rest of the field means different things in the two cases, and [ch19](#) is about why the distinction is structural rather than a numbering convenience.

`sscratch` exists because of a bootstrapping problem. A trap handler needs a register to work with and every register currently belongs to the interrupted program. `sscratch` is the one place to put something that survives the swap, and [ch16](#) traces exactly what xv6 keeps there.

Sv39, in one place

The paging scheme this book uses. [ch17](#) derives it and measures the walk; this is the field layout to keep beside that chapter.

A virtual address is thirty-nine bits of meaning: three nine-bit indices and a twelve-bit offset. A page-table entry is sixty-four bits: flags at the bottom, a physical page number above them.

Field	Bits	What it decides
V	0	Whether the entry means anything at all
R W X	1–3	Readable, writable, executable. All three clear means “this points at another table”
U	4	Reachable from user mode
G	5	Global — the same in every address space
A D	6–7	Accessed, dirty
RSW	8–9	Two bits the hardware ignores and software may use
PPN	10–53	The physical page number

The **R/W/X bits decide whether the entry is a leaf**. An entry with none of them set is a pointer to the next level; an entry with any of them set is a leaf. That is how one format serves both, and how a leaf can appear at a level above the last to map a larger page.

Where to go next

The unprivileged specification RISC-V INTERNATIONAL (N.D.) for the instruction encodings and the memory model, the privileged specification RISC-V INTERNATIONAL (N.D.) for everything on this page below the register table, and the psABI RISC-V INTERNATIONAL (N.D.) for the convention. The SBI specification RISC-V INTERNATIONAL (N.D.) describes the firmware layer beneath the kernel, which [ch01](#) meets once — a RISC-V machine reaches its performance counters through it — and this book does not otherwise use.

Appendix B · gdb for Kernels and RISC-V

An appendix in this book is a reference, not a chapter: no argument, no narrative, and everything in it either cites a primary source or comes from a stamped result under `bench/results/`.

Every sequence on this page has been run against this repository's own `make xv6-gdb`, on the submodule at its pinned commit. Where something does not work, that is recorded too, because the things that do not work are what cost the afternoon.

Getting attached

Two terminals. In the first:

```
make xv6-gdb
```

That stages the kernel, builds it, and starts QEMU halted before its first instruction with a debug stub listening. In the second:

```
gdb-multiarch xv6/stage/kernel/kernel
```

and then, inside gdb:

```
set architecture riscv:rv64
directory xv6/stage
target remote localhost:26000
```

`gdb-multiarch`, not `gdb`. This is the first thing that goes wrong and it goes wrong quietly. A distribution's plain `gdb` is on every `PATH` and is almost always built for the host architecture only. It connects to QEMU quite happily, and then fails on the first register read with

```
Truncated register 37 in remote 'g' packet
```

which does not mention architectures at all. `scripts/verify-setup.py` asks each debugger it finds whether it understands `riscv:rv64` rather than trusting its name, so `make verify` will tell you before you start.

`directory xv6/stage`. The kernel is built in the staging tree and you are running gdb from the repository root, so without this every stop reports `No such file or directory` and shows you no source. It is not that the debug information is missing; it is that the paths in it are relative to somewhere else.

The terminal will look frozen, and it is not. `make xv6-gdb` starts QEMU halted before the first instruction, so there is no boot log and no prompt — that is the `-S` doing its job. `Ctrl-A` then `C` switches to QEMU's monitor, where `info registers` answers and confirms the machine is alive and stopped; `Ctrl-A` then `C` returns. `Ctrl-A` then `X` exits, and works whether or not gdb is attached.

Port 26000. Not 1234. `xv6`'s own `.gdbinit` template says 1234 and this repository does not use it, because a fixed well-known port makes it easy to attach to somebody else's QEMU by mistake.

One hart, unless you mean it

```
python3 scripts/xv6-run.py --gdb --cpus 1
```

With the default number of harts, a breakpoint in shared kernel code is hit by whichever one gets there, and gdb switches threads under you:

```
[Switching to Thread 1.2]
Thread 2 hit Breakpoint 1, _entry () at kernel/entry.S:12
```

That is real and sometimes it is the thing you are studying, but it makes every other investigation harder. `info threads` lists them; `thread 2 moves`; `--cpus 1` makes the question go away.

Stopping in a trap

The sequence [ch16](#) is written around:

```
break usertrap
continue
printf "scause=%#lx sepc=%#lx stval=%#lx\n", $scause, $sepc, $stval
```

which prints, on the first system call of the boot:

```
scause=0x8 sepc=0x392 stval=0
```

`scause 8` is an environment call from user mode. `sepc` is a user-mode address: you are in the kernel, and the register holding “where we were” holds an address in a different address space. `stval` is zero because a system call has no faulting address; [ch18](#) is where it stops being zero.

CSRs are read as gdb convenience registers with a `$` in front, exactly like the integer ones. [Appendix A](#) lists the ones this book uses.

```
print/x $satp
$1 = 0x80000000000087fff
```

The top nibble is the mode field: `8` is Sv39. The bottom forty-four bits are the physical page number of the root page table, which is where [ch17](#)’s walk starts.

When the stack is nonsense

Stopped in `usertrap`, a backtrace looks like this:

```
#0 usertrap () at kernel/trap.c:167
#1 0x0000003fffffff09c in ?? ()
```

The second frame is not corrupt and gdb is not confused. It is the trampoline, and gdb cannot name it because the trampoline does not run at the address the linker gave it.

A breakpoint on `uservec` never fires.

```
print/x &uservec
$1 = 0x80007000
```

That is where the linker put it. It is not where it executes. The trampoline page is mapped a second time, at the top of every address space, and the trap path runs from *that* mapping. That second mapping is why the page survives the page table changing underneath it. Break on the symbol and you will wait for ever.

Break on the address it actually runs at instead:

```
break *0x3ffffff000
continue
x/4i $pc
```

```
=> 0x3ffffff000:      csrw      sscratch,a0
0x3ffffff004:      lui       a0,0x2000
0x3ffffff008:      addiw     a0,a0,-1
0x3ffffff00a:      slli      a0,a0,0xd
```

That address is `TRAMPOLINE` from `kernel/memlayout.h` — the last page below `MAXVA`. The `#1` frame in the backtrace above is the same page plus an offset, which is how you recognise it.

Inside there, `bt` shows one frame and nothing else, because there is no frame: no prologue has run, `sp` still belongs to the interrupted program, and the calling convention [ch14](#) describes is not in force. **What to use instead:** `x/i $pc` to see where you are, `info registers` to see the state, and the source of `trampoline.S` open beside it. Single-step with `stepi`, never `step`, since there is no line table to step through.

The same applies anywhere before a stack is established — `entry.S`, `start.c`'s early lines, and any point during a context switch.

Watchpoints

Hardware watchpoints work, and on a kernel they are often the fastest way to answer “who wrote this”:

```
watch ticks
continue
```

```
Hardware watchpoint 3: ticks

Old value = 1
New value = 2
clockintr () at kernel/trap.c:305
305         wakeup(&ticks);
```

On a physical address rather than a symbol, cast a literal:

```
watch *(unsigned long *)0x80000000
```

This is the form to use when the thing you care about has no name — a page you just allocated, a device register, a slot in a page table. Note that it watches the address, so if the page table changes underneath it you are now watching something else.

Delete before you re-break. A stale breakpoint in commonly-executed code will catch every `continue` and make it look as though your new one never fires. `delete` with no argument clears them all; `info breakpoints` says what is set.

A scripted session

For anything you will run more than once, put the commands in a file and pass it with `-x`. This is how every sequence on this page was checked:

```
gdb-multiarch -q -nx --batch -x session.gdb xv6/stage/kernel/kernel
```

`--batch` runs the file and exits, `-nx` ignores any `.gdbinit` you happen to have, and `-q` drops the banner. Add `set confirm off` and `set pagination off` at the top of the file or it will stop and wait for a keypress with nobody there.

What this cannot tell you

Anything about time. `gdb` stops the machine. Every duration observed under it is a duration of the debugger, and this is the `xv6` target, where a duration means nothing anyway ([ch00](#)).

What the hardware would have done. QEMU implements the architecture, not a pipeline. Stepping through `uservec` shows you the instructions in order; a real core does not execute them in that order, and [ch27](#) is the chapter about the difference.

Whether your change is correct. A debugger shows one run. The problems in this book are tests for the same reason: a run you watched go right is much weaker evidence than a check that fails when it goes wrong.

Appendix C · The perf Events This Board Has

An appendix here is a reference, not a chapter: no argument, no narrative, and everything in it either cites a primary source or comes from a stamped result. This one is generated from the reference machine, because which events a core exposes is a property of its silicon, its kernel and its firmware together and cannot be read off a datasheet. The Cortex-A76 manual ARM LTD (N.D.) says what each event *means*; `bench/results/perfevents-host.json` says which events exist *here*. Regenerate it on your own board with `python3 -m bench.run_perfevents`.

What this machine has

	This board
PMU	armv8_cortex_a76
Hardware counters	7
Events counted at once	7 — one more makes perf multiplex, scaling each to 75.0% enabled
Raw hardware events	40
perf's portable names (hardware / cache)	9 / 12
Software events the kernel keeps	26

Conditions: target `host` · Raspberry Pi 5 Model B Rev 1.1 · Linux 6.18.50+rpt-rpi-2712 · 2026-09-16 · Source: `bench/results/perfevents-host.json`, code hash `33a891ae8292682f`.

Counted, and computed

How many hardware counters this core has decides whether perf counts or estimates. Up to that many events run at once, each read straight from its own counter — a *count* of something that happened. Ask for one more and perf has no counter to give it, so it time-shares them: every event runs for a fraction of the workload and perf scales its reading up to a whole-run estimate. The number comes back looking the same and is no longer a count. The figure above found where that line falls by asking the machine — handing perf one more event at a time until one stopped running for the whole of the workload.

Carry that distinction out of [Part V](#). `perf stat -e cycles,instructions` is two counts; `perf stat` with a dozen events is a dozen estimates; and the `enabled` percentage perf prints beside each is how it tells you which you are holding. [ch30](#)'s sampling rests on the same counters and the same limit.

The hardware events

The events the PMU counts — the raw names the kernel exposes for this core. The Cortex-A76 manual ARM LTD (N.D.) defines what they mean, and this book redraws none of it.

Hardware event (Cortex-A76 PMU)			
br_mis_pred	br_mis_pred_retired	br_pred	br_retired
bus_access	bus_cycles	cid_write_retired	cpu_cycles
dtlb_walk	exc_return	exc_taken	inst_retired
inst_spec	itlb_walk	l1d_cache	l1d_cache_refill
l1d_cache_wb	l1d_tlb	l1d_tlb_refill	l1i_cache
l1i_cache_refill	l1i_tlb	l1i_tlb_refill	l2d_cache
l2d_cache_allocate	l2d_cache_refill	l2d_cache_wb	l2d_tlb
l2d_tlb_refill	l3d_cache	l3d_cache_allocate	l3d_cache_refill
ll_cache_miss_rd	ll_cache_rd	mem_access	memory_error
remote_access	stall_backend	stall_frontend	ttbr_write_retired

Conditions: target `host` · Raspberry Pi 5 Model B Rev 1.1 · Linux 6.18.50+rpt-rpi-2712 · 2026-09-16 · Source: `bench/results/perfevents-host.json`, code hash `33a891ae8292682f`.

perf also offers portable names. These are not events of their own but aliases the kernel maps onto some of the raw events above — and the mapping is the kernel’s choice, not a promise, so `cache-misses` on one machine and `cache-misses` on another need not be counting the same thing. A portable name is a convenience, not a definition.

perf's portable name	Kind
branch-instructions	hardware
branch-misses	hardware
bus-cycles	hardware
cache-misses	hardware
cache-references	hardware
cpu-cycles	hardware
instructions	hardware
stalled-cycles-backend	hardware
stalled-cycles-frontend	hardware
L1-dcache-load-misses	cache
L1-dcache-loads	cache
L1-icache-load-misses	cache
L1-icache-loads	cache
LLC-load-misses	cache
LLC-loads	cache
branch-load-misses	cache
branch-loads	cache
dTLB-load-misses	cache
dTLB-loads	cache
iTLB-load-misses	cache
iTLB-loads	cache

Conditions: target `host` · Raspberry Pi 5 Model B Rev 1.1 · Linux 6.18.50+rpt-rpi-2712 · 2026-09-16 · Source: `bench/results/perfevents-host.json`, code hash `33a891ae8292682f`.

The software events

`perf list sw` gathers a different kind: events the kernel keeps in software, with no PMU involved, so they are countable on a machine that has no counters at all — page faults, context switches, migrations, the CPU clock. Mixed in are `perf`'s own *tool* values, which count nothing and are computed or simply known: how many CPUs there are, how long the run took. The counted-and-computed line runs straight through this list too.

Software or tool event		
<code>alignment-faults</code>	<code>bpf-output</code>	<code>cgroup-switches</code>
<code>context-switches</code>	<code>cpu-clock</code>	<code>cpu-migrations</code>
<code>cs</code>	<code>dummy</code>	<code>duration_time</code>
<code>emulation-faults</code>	<code>faults</code>	<code>has_pmem</code>
<code>major-faults</code>	<code>migrations</code>	<code>minor-faults</code>
<code>num_cores</code>	<code>num_cpus</code>	<code>num_cpus_online</code>
<code>num_dies</code>	<code>num_packages</code>	<code>page-faults</code>
<code>slots</code>	<code>smt_on</code>	<code>system_time</code>
<code>task-clock</code>	<code>user_time</code>	

Conditions: target `host` · Raspberry Pi 5 Model B Rev 1.1 · Linux 6.18.50+rpt-rpi-2712 · 2026-09-16 · Source: `bench/results/perfevents-host.json`, code hash `33a891ae8292682f`.

Appendix D · An xv6 File Map

An appendix in this book is a reference, not a chapter: no argument, no narrative, and everything in it either cites a primary source or comes from a stamped result under `bench/results/`.

This one is walked from the submodule at its pinned commit rather than written, so every line count here is one that `wc -l` will agree with on the tree you have checked out — and a file renamed upstream fails CI rather than quietly making this page wrong.

How much kernel there is

The kernel, counted	Value
Files in <code>kernel/</code>	44
Lines in all of them	6603
Files a chapter reads	27
Lines in those	5137
Files nothing in this book opens	17

Conditions: target `xv6` · the `xv6` submodule's source, read rather than built · `xv6-riscv @ 9e3161a` · `none` · `none` · 2026-09-17 · Source: `bench/results/filemap-xv6.json`, code hash `72ccf96c95f8b570`.

That first figure is the reason this book uses `xv6` at all. A production kernel is far too large for anybody to read through; this one is small enough that the files [Part IV](#) opens are most of it, as the last two rows say, and the rest is not hidden from you — it is just not what any chapter needed.

Which chapter reads which file

File	Lines	Read by	For
kernel/entry.S	21	ch16	the first instructions the kernel runs, before there is a stack
kernel/start.c	66	ch16	the machine-mode setup that hands over to supervisor mode
kernel/syscall.c	151	ch16	the dispatch table, and how arguments cross the boundary
kernel/syscall.h	23	ch16	the call numbers
kernel/sysproc.c	112	ch16	the process calls, including the one the traps chapter's workload uses
kernel/trampoline.S	153	ch16	uservec and userret — the register moves the traps chapter counts
kernel/trap.c	220	ch16	where a trap is decided, and where a page fault would be handled
kernel/memlayout.h	63	ch17	what is mapped where, and why the trampoline is at the top
kernel/riscv.h	417	ch17	the Sv39 field definitions, as macros over a 64-bit word
kernel/vm.c	491	ch17	the page-table walk, mapping, and the kernel's own address space
kernel/exec.c	176	ch18	what a fresh address space is built out of
kernel/kalloc.c	82	ch18	the physical page allocator a fault ends up calling
kernel/console.c	203	ch19	a device driver small enough to read in full
kernel/kernelvec.S	64	ch19	the trap path taken when the kernel itself is interrupted
kernel/plic.c	47	ch19	which device is allowed to interrupt which hart
kernel/uart.c	155	ch19	the registers underneath it
kernel/sleeplock.c	55	ch20	the other kind, and when each is correct
kernel/spinlock.c	116	ch20	the lock, and the memory barriers around it
kernel/spinlock.h	8	ch20	what a lock is made of
kernel/proc.c	701	ch21	the scheduler, and both halves of a context switch
kernel/proc.h	104	ch21	the context the scheduling chapter counts the registers of
kernel/swtch.S	42	ch21	the switch itself — fourteen registers and nothing else
kernel/bio.c	153	ch22	the buffer cache, and why a read can cost nothing
kernel/file.c	179	ch22	what a file descriptor actually is
kernel/fs.c	741	ch22	inodes, blocks, and the path from a name to a byte
kernel/log.c	261	ch22	the write-ahead log the file-system chapter counts the amplification of
kernel/virtio_disk.c	333	ch22	the only device in the system that makes you wait

Conditions: target `xv6` · the `xv6` submodule's source, read rather than built · `xv6-riscv @ 9e3161a` · none · none · 2026-09-17 · Source: `bench/results/filemap-xv6.json`, code hash `72ccf96c95f8b570`.

Keep this open from [ch16](#) onwards. The rows are grouped by chapter because that is how you will arrive here: a chapter is discussing something, and the question is which of the files in `kernel/` to open beside it.

What is not in the table

Everything else in `kernel/` — headers describing on-disk and in-memory structures, the string and printing routines, the linker script, the parts of the build no chapter discusses. They are not listed individually because a list of them is a directory listing, which you already have.

Two absences are deliberate rather than incidental.

main.c is not mapped to a chapter. It is worth reading early and it belongs to no chapter in particular: it is the order in which everything else is initialised, which is a useful thing to have seen once and a poor thing to study in isolation.

Nothing in user/ is mapped either. The book’s own programs live in `xv6/apps/`, and the kernel’s user programs are read in passing rather than studied — with one exception: [ch16](#) explains why the shell is the worst possible thing to measure.

What this cannot tell you

Where a symbol is. This maps files, not functions. `grep -rn` over `kernel/` answers that question faster than any table could, and stays right when upstream moves something.

What the file does. The “For” column says what a chapter goes there *for*, which is narrower on purpose. `fs.c` does a great deal that [ch22](#) never mentions.

Anything about a different xv6. The commit is pinned and recorded in the conditions line under each table. MIT’s xv6 changes; a map of one version describing another is exactly the failure this page is generated to avoid.

Appendix E · Glossary

An appendix in this book is a reference, not a chapter: no argument, no narrative, and everything in it either cites a primary source or comes from a stamped result under `bench/results/`.

Assembled from the chapters after they were written, which is the only order that works: a glossary drafted first defines the words the prose did not end up using. Every entry names the chapter that establishes it, and the definition here is deliberately shorter than the chapter's — this page is for recognising a term, not for learning it.

The book's own vocabulary

These four are used throughout and mean something specific here.

Term	What it means in this book
target	One of three places code runs: <code>bare</code> (a RISC-V machine under QEMU with nothing on it), <code>xv6</code> (the teaching kernel under QEMU) or <code>host</code> (Linux on real hardware, natively). Every result declares one. ch00
stamped result	A JSON file under <code>bench/results/</code> carrying what was measured, on which machine, with which toolchain, and a hash of the code that produced it. Nothing reaches a chapter any other way. ch00
pending figure	A measurement the book owes but has not taken. It renders as a warning containing no numbers — never a placeholder and never an estimate. Preface
listing	Disassembly captured from a compiler and stamped, so a chapter can show machine code without pasting it. Re-captured by CI on every push. ch02

Part I — C, and what the machine does with it

Term	Definition	Where
disassembly	A compiled function printed back as instructions. What the compiler produced, not what you wrote.	ch02
mnemonic	The name of an instruction, as distinct from the bytes that encode it.	ch02
operand	What an instruction acts on. In both of this book's syntaxes the destination comes first.	ch02
address	A number naming a byte. Memory is one array and an address is an index into it.	ch03
pointer	A variable holding an address, with a type saying what to assume is there.	ch03
freestanding	The standard's word for C with no library underneath it. ch04 deliberately does not use it, on the grounds that what matters is which specific things are missing.	ch04
free list	An allocator that keeps its bookkeeping inside the memory it is not currently handing out.	ch04
storage duration	How long an object lives: automatic, static, or allocated. The distinction a garbage-collected language does not make you make.	ch05
function pointer	An address of code rather than of data, and how a kernel gets a virtual call without a class.	ch05

Part II — the machine with nothing on it

Term	Definition	Where
trap vector	The address the hardware jumps to when a trap happens, which software puts there before any trap can be handled.	ch06 , Appendix A
trap handler	The code at that address. On a bare machine it is a function you wrote; in a kernel it is the start of everything.	ch06 , ch16
privilege level	Which of the machine's modes is executing, and therefore what the instruction being attempted is allowed to do.	ch07
hart	A RISC-V hardware thread: one instruction stream with its own registers, sharing memory with the others.	ch08
identity mapping	A page table in which a virtual address translates to the same physical address, so translation can be switched on without anything moving.	ch08
dispatch	Turning a number a caller supplied into the function that serves it, which is all a system-call table is.	ch09
file descriptor	A small integer that means a device, because a table in the process turns it into one.	ch10
process table	The kernel's fixed array of processes, and the reason running out of them is a number rather than a shortage of memory.	ch11

Part III — the machine underneath a program

Term	Definition	Where
translation unit	One source file plus everything it included, which is what the compiler actually sees.	ch12
preprocessing	The first stage: includes pasted in, macros expanded, conditionals resolved. Its output is C.	ch12
relocation	A note in an object file saying "this address is not known yet", resolved at link time.	ch12 , ch15
object file	Compiled code with unresolved references, not yet a program.	ch12
padding	Bytes a compiler inserts inside a struct so each member lands on an address it can be accessed at.	ch13
alignment	The requirement that an object's address be a multiple of its size, or of its widest member.	ch13
undefined behaviour	A program the standard makes no promises about, which a compiler may therefore assume never happens.	ch13 , ch05
<code>volatile</code>	A promise not to cache a value in a register and not to reorder accesses to it relative to each other. Not a threading tool.	ch05
calling convention	The agreement about which registers carry arguments, which are preserved, and who cleans up. A contract between compiled code, not a hardware rule.	ch14
caller-saved / callee-saved	Whether the code making a call or the code being called is responsible for preserving a register across it.	ch14
stack frame	The region a function owns on the stack. At <code>-O2</code> many functions have none.	ch14
prologue / epilogue	The instructions at a function's start and end that establish and dismantle its frame.	ch14
leaf function	One that calls nothing, and therefore usually needs no frame and no saved return address.	ch14
section	A named region of an object file — what the <i>linker</i> deals in.	ch15
segment	A named region of a program image — what the <i>loader</i> deals in. Sections are mapped into segments.	ch15
symbol table	The list of names an object file defines and requires.	ch15

Part IV — the kernel, as structure

Term	Definition	Where
trap	Any transfer to the kernel: a system call, a fault, or an interrupt. The hardware's mechanism is the same for all three.	ch16
exception	A trap caused by the instruction being executed — it could not complete.	ch16 , ch19
interrupt	A trap caused by something outside the program asking for attention.	ch19
trampoline	A page mapped at the same address in every address space, so the trap path survives the page table changing under it.	ch16 , ch17
trap frame	Where the interrupted program's registers are put, since it agreed to no convention about which to preserve.	ch16
virtual address	An address a program uses, which means nothing without a page table.	ch17
page table	The tree the hardware walks to turn a virtual address into a physical one.	ch17
Sv39	RISC-V's three-level paging scheme: three nine-bit indices and a twelve-bit offset.	ch17 , Appendix A
page-table entry (PTE)	One sixty-four-bit word: permission flags and a physical page number, or a pointer to the next level.	ch17
TLB	The cache of recent translations. Changing a page table means telling it.	ch17 , ch29
page fault	A trap raised when translation fails. Not an error — a hook, and most of what a modern kernel does with memory is built on it.	ch18
lazy allocation	Handing out address space and only finding physical pages when they are touched.	ch18
copy-on-write	Sharing a page until somebody writes to it, using the fault as the trigger.	ch18
device driver	The code that knows a device's registers. In xv6, small enough to read in full.	ch19
interrupt controller	The hardware that decides which device may interrupt which core.	ch19
spinlock	A lock that waits by running. Correct when the wait is shorter than a context switch.	ch20
sleeplock	A lock that waits by yielding. Correct when it is not.	ch20 , ch21
memory ordering	Which of one core's writes another core may observe, and in what order. Weaker than the program order on both of this book's architectures.	ch20 , ch28
barrier / fence	An instruction, or a property of one, constraining that order.	ch20 , ch28
acquire / release	The two halves of the ordering a lock needs: nothing moves out of the critical section past either end.	ch20
context switch	Saving one thread's registers and restoring another's. Structurally small; what surrounds it is not.	ch21
scheduler	The code that decides which thread runs next.	ch21
buffer cache	The kernel's copy of recently used disk blocks, and the reason a read can cost nothing.	ch22
write-ahead log	Writing what you are about to do before doing it, so a crash leaves a recoverable state.	ch22
write amplification	How many blocks actually move for each block a program asked to write.	ch22
inode	The on-disk record of a file, distinct from any name it has.	ch22

Part V — where the cycles go

Term	Definition	Where
monotonic clock	One that cannot go backwards and is not adjusted to keep wall-clock time honest. The only kind to measure with.	ch24
minor fault	A page fault the kernel satisfies without touching storage.	ch18 , ch29
major fault	One that has to wait for storage. Orders of magnitude more expensive, and identical from the trap's point of view.	ch29
clock cost	What reading the clock costs. The number that decides whether a measurement means anything.	ch24 , ch29
distribution	What to report instead of a number: minimum, median, tail. A single duration is an anecdote.	ch24
warm-up	Discarding the start of a run, only when the discarded part is genuinely slower and you can say why.	ch24
cache line	The unit everything happens in. Not the variable, not the byte.	ch25 , ch28
working set	How much memory a piece of code touches in the window that matters.	ch25 , ch30
prefetch	The hardware fetching ahead of a predictable access pattern, and the reason a sequential walk is not a sequence of misses.	ch25
pointer chase	A walk whose next address is not known until the current load returns, which defeats prefetching.	ch23 , ch25
strength reduction	Replacing an expensive operation with a cheaper one that computes the same thing. Usually the compiler's job.	ch26
pipeline	The core overlapping the stages of consecutive instructions.	ch27
instruction-level parallelism (ILP)	How many independent instructions are available to overlap.	ch27
IPC	Instructions retired per cycle. Above one on a core that can finish several at once, and a diagnostic rather than a goal.	ch27 , ch29
dependence chain	A sequence where each instruction needs the previous one's result. Its length is a floor no width removes.	ch27 , ch31
branch predictor	The hardware guessing which way a branch goes, so the pipeline need not wait.	ch27
misprediction	A wrong guess, paid for by discarding the work done since.	ch27
branchless	Computing both sides and selecting, so there is nothing to mispredict. Worth it exactly when the branch is unpredictable.	ch27
coherence	The guarantee that cores agree about the contents of a cache line, and the traffic that provides it.	ch28
false sharing	Two cores fighting over a line neither shares data on. The sharing is real; what is false is that anyone meant to share.	ch28
Amdahl's law	The ceiling a scaling curve cannot pass, computed from the program before the machine is involved.	ch28
system call	A request to the kernel. Compiled into your program as a branch, with the trap somewhere you cannot see.	ch16 , ch29
vDSO	Kernel code mapped into every process so a few requests need no privilege change at all. The reason the clock is cheap.	ch29
sampling profiler	One that arranges to be interrupted and writes down where the program was. Everything it can and cannot tell you follows from that.	ch30
exclusive / inclusive time	Samples in a function itself, against samples in it and everything it called. The flat list blames the leaf.	ch30

Term	Definition	Where
skid	The gap between the instruction that stalled and the one the sample was attributed to. Read the neighbourhood, never the line.	ch30
aliasing (sampling)	A fixed sampling period over a fixed loop visiting only some positions, producing a stable and confidently wrong profile.	ch30
vectorisation	One instruction doing several elements. Not on by default at the level most people build at.	ch31
lane	One element's worth of a vector register.	ch31
tail	The elements left over when the trip count is not a multiple of the width, handled one at a time.	ch31
reassociation	Adding the same numbers in a different order. Free for integers, not free for floats, and the reason one reduction widens and the other does not.	ch31
arithmetic bound	The most a change could possibly buy, computed before measuring. A speedup is reported against it or not reported.	ch28 , ch31

What this cannot tell you

Enough to use any of these. A one-line definition is for recognising a term you have met. The chapter named beside it is where it is established, and none of these entries is a substitute for the measurement that made the chapter worth writing.

The standard meaning. Where a term is used here more narrowly than in general practice — and several are — this page gives the book's sense. Where the book's sense differs from the common one deliberately, the chapter says so.

Appendix F · AArch64 for RISC-V Readers

An appendix in this book is a reference, not a chapter: no argument, no narrative, and everything in it either cites a primary source or comes from a stamped result under `bench/results/`.

This is a translation rather than a reference. It is written for someone who has read [ch14](#) and is about to read [ch26](#), and it is organised as *you know this already, here it is again*. For anything not in the book's path, the architecture reference manual ARM LTD (N.D.) is the document; this page will not substitute for it and does not try.

Why there are two

[Part III](#) and [Part IV](#) are RISC-V because the kernel small enough to read in an afternoon is a RISC-V kernel. [Part V](#) is AArch64 because that is where the performance counters work: sampling needs a PMU that can raise an interrupt on counter overflow, and no affordable RISC-V core can both count and sample. [Appendix H](#) has the evidence, and [ch28](#) is where using two architectures pays off: a reader shown one weak memory model concludes that model is memory ordering.

So the crossing is deliberate, and this page is where you pay for it, once.

Registers

Thirty-one general-purpose registers plus a zero register and a stack pointer, which is nearly the same count as RISC-V and arrived at differently.

	RISC-V	AArch64
General purpose	<code>x0 – x31</code> , <code>x0</code> hardwired zero	<code>x0 – x30</code> , plus <code>xzr / wzr</code>
32-bit form	none — use <code>w</code> -suffixed instructions	<code>w0 – w30</code> , the low half of <code>x0 – x30</code>
Zero	<code>x0</code> , a register	<code>xzr</code> , an operand encoding
Stack pointer	<code>sp</code> = <code>x2</code> , an ordinary register	<code>sp</code> , separate and not <code>x31</code>
Return address	<code>ra</code> = <code>x1</code> , by convention	<code>x30 / lr</code> , written by the branch instruction
Arguments	<code>a0 – a7</code>	<code>x0 – x7</code>
Return value	<code>a0</code> , <code>a1</code>	<code>x0</code> , <code>x1</code>
Callee-saved	<code>s0 – s11</code>	<code>x19 – x28</code>
Frame pointer	<code>s0 / fp</code> , by convention	<code>x29 / fp</code> , by convention

Two differences that change how listings read.

The `w` registers are not a convention, they are the instruction. `add w0, w1, w2` is a 32-bit add that zero-extends into the full register; `add x0, x1, x2` is a 64-bit add. RISC-V spells this with separate mnemonics (`addw` against `add`). You will see `w` registers constantly in [ch26](#)'s listings wherever an `int` is involved, and the zero-extension is free rather than an extra instruction.

`x31` is two registers depending on the instruction. In most positions the encoding means `xzr`; in a few it means `sp`. This is why the zero register is described here as an operand encoding rather than a register — it does not exist as a thing you can point at.

Loads and stores

	RISC-V	AArch64
Load a word	<code>lw a0, 8(a1)</code>	<code>ldr w0, [x1, #8]</code>
Load a doubleword	<code>ld a0, 8(a1)</code>	<code>ldr x0, [x1, #8]</code>
Store	<code>sd a0, 8(a1)</code>	<code>str x0, [x1, #8]</code>
Indexed	compute the address first	<code>ldr w0, [x1, x2, lsl #2]</code>
Pair	—	<code>stp x29, x30, [sp, #-16]!</code>
Post-increment	—	<code>ldr q1, [x2], #16</code>

Three things to know before reading any AArch64 listing.

The addressing modes do arithmetic. `[x1, x2, lsl #2]` scales an index by four and adds it, in the load. RISC-V would need a shift and an add first. This is why an AArch64 inner loop over an array is often two instructions shorter than the RISC-V one for the same C, and it is visible in every listing in [ch26](#).

stp and ldp move two registers at once. Almost every non-leaf function prologue you will see is `stp x29, x30, [sp, #-16]!` — save the frame pointer and the return address, and decrement the stack pointer, in one instruction. The `!` is the write-back. [ch14](#)'s RISC-V prologue makes the room and stores the return address as two separate instructions, and would need a third if it also kept a frame pointer.

The suffix after the bracket is where the increment went. `[x2], #16` adds sixteen to `x2` *after* the access; `[x2, #16]!` adds it before. Neither exists in RISC-V, and both appear in vectorised loops in [ch31](#).

Branches and conditions

This is the largest difference and the one that changes how you read a loop.

RISC-V compares and branches in one instruction: `blt a0, a1, label`. AArch64 sets flags and then branches on them:

```
cmp x0, x1
b.lt label
```

The flags are a side effect that persists. They buy the conditional select, which has no RISC-V equivalent; compare-and-branch-on-zero sidesteps them entirely.

Conditional select. `csele x0, x1, x2, lt` writes one of two registers depending on the flags, with no branch at all. [ch27](#) is about what that is worth: a branch the predictor cannot learn costs a pipeline flush every time, and a `csele` costs one instruction always.

Compare-and-branch-on-zero. `cbz / cbnz` are the exception that does not use the flags, and they are extremely common because testing against zero is extremely common.

	RISC-V	AArch64
Compare and branch	<code>blt a0, a1, L</code>	<code>cmp x0, x1 then b.lt L</code>
Branch if zero	<code>beqz a0, L</code>	<code>cbz x0, L</code>
Select without branching	—	<code>csel x0, x1, x2, cond</code>
Call	<code>jal ra, f</code>	<code>bl f</code>
Tail call	<code>j f</code>	<code>b f</code>
Return	<code>ret (jalr x0, 0(ra))</code>	<code>ret (uses x30)</code>

Atomics and ordering

[ch20](#) prints both of these from real disassembly and [ch28](#) is the chapter about what the difference means.

	RISC-V	AArch64
Atomic read-modify-write	<code>amoswap.w.aq, amoadd.d, ...</code>	<code>ldadd, swp, ... (LSE), or ldxr/stxr</code>
Acquire	<code>fence r, rw</code> after the load	<code>ldar</code> — the load itself
Release	<code>fence rw, w</code> before the store	<code>stlr</code> — the store itself
Full barrier	<code>fence rw, rw</code>	<code>dmb ish</code>

The difference is where the ordering goes. RISC-V puts an instruction *between* the two operations being ordered; AArch64 folds the ordering into one of them. A reader who learned that a barrier is something you put between two things will not recognise `stlr` as a barrier at all, which is exactly why [ch28](#) puts them side by side rather than teaching one.

Both spell the same requirement. Memory ordering is that requirement, not either way of writing it.

Vectors

NEON, in one paragraph, because [ch31](#) is the chapter.

Thirty-two registers, `v0–v31`, 128 bits each, addressed by an *arrangement specifier* that says how to divide them: `v0.4s` is four 32-bit lanes, `v0.2d` is two 64-bit ones, `v0.16b` is sixteen bytes. The same registers are named `q0–q31` when the whole 128 bits are meant, and `s0/d0` when a single scalar float or double is meant.

That last point is a trap [ch31](#) fell into and records: an instruction naming a `v` register is not evidence of vectorisation. This compiler builds a floating-point zero with `movi v0.2s, #0`, in a function whose loop it has refused to widen.

What this cannot tell you

How the two architectures differ where this book does not go. Exceptions, privilege levels, the MMU, system registers — all different, none of it here, because [Part V](#) does not read the kernel and [Part IV](#) does not run on AArch64.

Which is better. They make different choices and the book uses both for what each is good for. Nothing on this page is an argument.

What any of it costs. This is a translation table. Every number about cost in this book is measured, and none of it is here.

Where to go next

The ARM architecture reference manual for AArch64 ARM LTD (N.D.) is the document, and its A64 instruction index is the part to keep open. The procedure call standard for AArch64 ARM LTD (N.D.) is the equivalent of the RISC-V psABI RISC-V INTERNATIONAL (N.D.) and is what decides the register roles above.

Appendix G · Reading xv6 Alongside This Book

An appendix in this book is a reference, not a chapter: no argument, no narrative, and everything in it either cites a primary source or comes from a stamped result under `bench/results/`.

Everything below was read out of the submodule at its pinned commit rather than recalled, so it describes the code you have checked out. [Appendix D](#) says what else is in that tree.

The two books ask different questions

The kernel this book uses has its own commentary *COX ET AL.* (N.D.), written by the people who wrote the kernel, distributed free by MIT. It is the best available explanation of what that code does and you should read it.

It is not a competitor and this book does not summarise it. They ask different questions of the same source:

The xv6 commentary	What does this code do, and why is it arranged this way?
This book	What does it cost, and how would I know?

That difference decides almost everything about how the two are organised. The commentary follows the kernel's own structure, because it is explaining a program. This book follows a question, which is why its [Part IV](#) takes the kernel apart mechanism by mechanism and its [Part V](#) then asks the price of each — and why a mechanism that is one section there may be a chapter here, or the reverse.

Read them in either order. A reader who has the commentary open will find this book's chapters supply the measurements it does not attempt; a reader who starts here will find the commentary supplies the surrounding code that a chapter about one mechanism does not show.

Where the coverage meets

Mapped by topic rather than by chapter number, deliberately: the commentary's numbering shifts between revisions and its topics do not.

When the commentary is on	This book is in	And the price is in
Operating system interfaces — processes, <code>fork</code> , <code>exec</code> , files	ch04 for the memory model it implies; the trace below	—
Operating system organization, isolation, privilege	ch16	ch29
Page tables and Sv39	ch17	ch25 address translation, ch29 faults
Traps, system calls, and the trap path	ch16	ch29
Page faults and what can be built on them	ch18	ch29
Interrupts and device drivers	ch19	—
Locking	ch20	ch28
Scheduling and context switching	ch21	ch29
File system, logging, buffer cache	ch22	—
Concurrency revisited, memory ordering	ch20	ch28

Two of the blank rows share a reason: the cost of an interrupt, a file system and a driver on this machine is the cost of *this machine*'s devices, and the reference board's storage is an SD card behind a bridge rather than anything a chapter could generalise from. [ch19](#) and [ch22](#) say so in their own limitations sections. The third blank, operating-system interfaces, has no single price: the trace below pays it a layer at a time.

One call through every layer: fork

`fork` is the worked example of how to use the table above, because it is the one call in the kernel that touches almost every mechanism this book takes apart separately. Reading it is the fastest way to see that the chapters are describing one system rather than six.

In the pinned tree the function is `kfork` in `kernel/proc.c` — `sys_fork` in `kernel/sysproc.c` is a one-line wrapper. Older revisions of the commentary call it `fork`.

What it does	Which layer	Where that is explained
<code>allocproc()</code> finds an unused slot in the fixed process table	allocation without a heap	ch04
...and returns <code>0</code> when there is none, which becomes <code>-1</code>	failure that returns rather than raises	ch04 's problem 4.2
<code>uvmcopy()</code> walks the parent's page table and copies every page	address translation	ch17
...with a <code>kalloc()</code> per page, which can also fail	the physical allocator	ch18
<code>*(np->trapframe) = *(p->trapframe)</code> copies the saved user registers	what a trap saves	ch16
<code>np->trapframe->a0 = 0</code>	the calling convention	ch14 , ch16
<code>filedup()</code> over the open files, and <code>idup()</code> on the working directory	reference counting	ch22
<code>np->state = RUNNABLE</code> makes it eligible to be chosen	scheduling	ch21

Nothing returns twice. The row that sets `a0` to zero is where that illusion comes from. The child is a copy of the parent — including the saved register set the trap path will restore on the way out — with a single word changed: the register the calling convention uses for a return value. Both processes then resume at the instruction after the `ecall`, each reading its own `a0`. [ch16](#) is where that saved register set is counted.

`uvmcopy` copies **eagerly**, and the pinned tree is explicit about it: a `kalloc` and a `memmove` of a whole page, per page, with no sharing. That is a deliberate simplification, and it lets [ch18](#) add copy-on-write rather than merely explain it — the mechanism is absent here, so the chapter has somewhere to put it. A production kernel shares the pages and marks them read-only, and [ch29](#) is where the difference in cost is measured.

The failure paths are where [ch04](#)'s second problem lives, in the source. `kfork` can fail in two places — no free slot, or no free page — and both return `-1` after undoing what they had done. A caller that does not look at the result has written the bug [ch04](#) is about.

What this cannot tell you

What the commentary says. This page is a map, not a summary, and it deliberately does not restate a word of it. Where the two books cover the same mechanism they do it differently and both are worth reading; where this page sends you to a chapter, the chapter is about the cost.

Anything about a different revision. The tree here is pinned, and in it the function is `kfork` and `allocproc` returns `0`. Both have been spelled otherwise. The conditions line under [Appendix D](#)'s tables names the commit; `git -C xv6/xv6-riscv log -1` is the check.

Whether the commentary's version and this one agree. They are different documents at different revisions, and where they differ the tree wins — the tree is what boots.

Appendix H · Choosing the Machine

An appendix in this book is a reference, not a chapter. This one is read once, before you spend any money, and then only to settle an argument with a spec sheet.

Part V is the only part that needs hardware of its own: it is where every number in the book is measured, and **ch01** is where you make it work. This appendix covers the decision before that: what the `host` target has to be able to do, how to check a machine you already own, and what changes if yours differs from the reference. The *argument* for a second machine at all is in the [preface](#); this is the shopping and the checking.

A **Raspberry Pi 5 with 4 GB or more**, and an active cooler. That is the whole decision. Every Pi 5 has the counters this book needs — same SoC, same four cores, no variant where they are missing — so there is no specification to compare and nothing to get wrong except the RAM, and only because **ch28** wants four cores with room to work.

	What	Why this one
Board	Raspberry Pi 5, 4 GB or 8 GB	Four identical Cortex-A76 cores whose performance counters work. See below
Cooling	The official active cooler, or a case with a fan	Not optional here. A Pi 5 under sustained load throttles
Power	The official 27 W USB-C supply, or one rated for the board	Underpowering a Pi produces instability that reads exactly like a kernel bug
Storage	A microSD card that is not the cheapest on the shelf	An NVMe drive on a PCIe HAT is nicer and not required
Network	An Ethernet cable — any working one	Wi-Fi works. Wired keeps the radio's driver from doing interrupt work on the cores you are measuring

You also need a development machine for Parts I to IV — anything that runs Homebrew or apt and holds an SSH key. It never measures anything.

The cooler earns its line in the table. A Pi 5 that throttles is running a benchmark at one clock speed and finishing it at another, which is not a slow measurement but a wrong one, and one of the more instructive ways to be wrong. **ch24** treats throttling as a measurement hazard and shows how to catch it happening; a cooler means you meet it deliberately rather than in every run you ever take.

Why a separate board, and not the laptop you are reading this on

A laptop can almost certainly count and sample — `perf` on x86-64 is mature, and on Linux you could start **Part V** this afternoon. Do not, because a laptop is too complicated to learn on. A current laptop has cores of two different kinds, a clock that moves constantly, two threads sharing one core's execution units, and a scheduler migrating your benchmark across all of it. Each of those makes a measurement harder to attribute. The Pi 5 is four identical cores with one cache hierarchy and no hyper-threading: when a number moves, something you did moved it.

That is the book's own argument applied to its own tooling — use the instrument that can answer the question. A laptop is a *faster* machine and a *worse* instrument. If a Pi is genuinely not possible, **Part V** still runs on a Linux laptop, and every chapter whose reading depends on the core's shape says so in its own header.

The Pi is not the *most* legible instrument. An **in-order** core — short pipeline, no out-of-order execution, no register renaming — makes microarchitecture plainer still: a dependent load that misses in cache stalls, visibly, for as long as the miss takes. The A76 reorders, so the connection between an instruction you wrote and a cycle that got spent runs through enough machinery that a small experiment occasionally comes out backwards.

Two reasons that is the right trade anyway. The in-order RISC-V option could not sample, which cost more than legibility bought. And **every machine you are likely to care about optimising reorders**, so learning to attribute cycles on one is the skill that transfers. [ch27](#) is harder for it, says so in its own header, and is more useful as a result. If you want the clean version too, an in-order Cortex-A53 — a Pi 3 or Pi Zero 2 W — costs very little, and running [ch24](#)’s experiments on both is an instructive afternoon.

It is an ARM machine, and everything before Part V is RISC-V

That is deliberate. The kernel small enough to read in an afternoon is a RISC-V kernel; the hardware whose counters actually work is an ARM one. Those are different machines, and pretending otherwise would mean lying about one of them.

Part V needs `perf` to do two separate things: **count** events over a run, and **sample** — interrupt the program thousands of times a second to ask where it is. Sampling needs the counters to raise an interrupt when they overflow. On ARM that is a standard part of the performance monitoring unit. On RISC-V it is an optional extension, and a 2025 study of the three RISC-V cores you can actually buy *DISSECTING RISC-V PERFORMANCE: PRACTICAL PMU PROFILING AND HARDWARE-AGNOSTIC ROOFLINE ANALYSIS ON EMERGING PLATFORMS* (2025) found that none of them wins:

	SiFive U74	T-Head C910	SpacemiT X60
Out-of-order	No	Yes	No
Vector extension	None	0.7.1 (draft)	RVV 1.0
Counter-overflow interrupt	No	Yes	Limited
Upstream Linux support	Yes	Partial	No

Read down the columns. Choosing RISC-V for **Part V** would have made two of its eight chapters unmeasurable — one needs sampling, one needs a vector unit — on boards that are hard to buy, with firmware that has broken `perf` between distribution releases. A Raspberry Pi costs none of that.

The [preface](#) says what that costs a reader. Everything outside the disassembly is method, and method does not have an architecture.

Its SoC is a BCM2712: four Arm Cortex-A76 cores at 2.4 GHz, 64 kB of L1 instruction and data cache each, 512 kB of L2 per core, and 2 MB of L3 shared between them RASPBERRY PI LTD (N.D.). Those are the vendor’s numbers, and the book does not repeat them anywhere else — [ch25](#) measures that hierarchy rather than quoting it, and comparing what it finds against this paragraph is one of the more satisfying results in **Part V**.

Three levels with a private L2 and a shared L3 is a genuinely good shape to learn on. The private level shows you locality; the shared one is where [ch28](#)’s cores collide.

If you already own something else

The book recommends one machine because one machine is enough, not because the rest of **Part V** is about Raspberry Pis. What a machine actually has to do is short:

	Requirement	Why
Must	64-bit Linux, reachable over SSH	
Must	<code>perf stat -e cycles,instructions -- true</code> returns real counts	The one requirement with no workaround. Part V does not exist without it
Must	<code>perf record</code> can sample	ch30 is entirely sampling. A different capability from counting
Must	4 GB RAM, 4 cores	ch28 measures what cores cost each other
Nice	NVMe or a fast SSD	Builds and ch22 are far less tedious
Nice	A SIMD unit the compiler targets — NEON, or RVV 1.0	ch31 measures vectorisation
Nice	Few kinds of core, and a clock that holds still	Not required. It is why the reference is a Pi rather than a laptop

An old laptop, a spare desktop, a Rock 5B, another single-board computer you have in a drawer: run `scripts/verify-setup.py` on it and it will tell you. `hardware/README.md` has the same list in a form you can hand to a search tool, for readers somewhere Raspberry Pis are hard to get.

THE PURCHASE IS YOURS

This book does not sell hardware, has no relationship with any vendor, and has tested nothing but its own reference machine. Prices, availability and listings change; nothing here is a warranty that a given machine will work for you.

In practice: **the requirement you cannot check before it arrives is the one that matters most.** No product listing can honestly promise you working performance counters, because they depend on the image as much as on the board. Buy somewhere with a return policy, and run `scripts/verify-setup.py` on day one rather than the week you reach [Part V](#).

The requirement to be suspicious about

The counters. Everything else is printed on the box; whether `perf` can read the hardware is not, and it is the one that stops the book dead.

On ARM the PMU is reached directly, but the kernel still has to be told it is there, and this is not hypothetical: the Raspberry Pi kernel's own 6.12 branch shipped a device tree for the Pi 5 with the `arm-pmu` node missing [RASPBERYPi/LINUX ISSUE 6507 \(2025\)](#). The 6.6 tree had it. On the affected images the hardware was perfectly capable, the `armv8_cortex_a76` driver never registered, and `perf` saw no hardware counters at all — quietly, because that is how this fails.

On RISC-V there is an extra layer: the counters are machine-mode CSRs, the kernel runs in supervisor mode, and the firmware bridges them through the SBI PMU extension RISC-V INTERNATIONAL (N.D.), so the answer depends on the firmware as much as on the silicon.

Read that regression as the general case rather than a Raspberry Pi anecdote. **Whether your counters work is a property of the configuration, not of the board** — silicon, device tree, kernel, firmware and `perf` build all have to agree, and four of those five change under you without the box changing at all.

THE BOOK DOES NOT TELL YOU WHICH KERNEL TO RUN

Ending this section with an image and a version number would be worse advice than it looks. A pinned version is wrong within a year, cannot be re-verified on every release, and teaches you to check a string instead of a machine — while the failure it is meant to prevent stays perfectly possible on the version that was correct when it was written.

A *minimum* version would be worse still. What went wrong on the Pi 5 was a **regression**, so the node was present in the older kernel and absent in the newer one; a floor selects for the broken configurations rather than against them. A range would work and would need maintaining forever — and it would still have to be written once per kernel tree, because the Raspberry Pi kernel and mainline are different trees that disagree about this today RASPBERRYPI/LINUX & TORVALDS/LINUX (N.D.). The version number is not what decides it. The device tree in `/boot/firmware/` is, and your machine will read it out for you.

So the book records the machine instead. Every `host` result stamps the board, the operating system, the kernel and whether `perf` could count and sample; the reference machine's own account is printed in [ch01](#). It tells you what produced the book's numbers; it is not a requirement for yours. The only requirement is that `verify-setup.py` passes on the machine in front of you, which is a question about that machine and not about a version string.

The reference machine, and why your numbers will differ

Every figure in [Part V](#) of this repository was measured on the machine that its result names — each one stamps the model, the core and the kernel that produced it, so no figure is ambiguous about where it came from.

So your numbers will not match, and that is expected rather than a problem. The book is about ratios, mechanisms and method, and those transfer.

Which chapters actually depend on the hardware

Most do not. Five chapters of [Part V](#) do, and rather than let you discover that two hundred pages in, here they are up front. Each of these says the same thing in its own header, so you cannot open one without being told.

Chapter	What it assumes	What changes on a different machine
ch25	A particular cache hierarchy — levels, sizes, line size, TLB reach	The numbers, entirely. The method is the chapter, and measuring <i>your own</i> hierarchy is the exercise
ch27	An out-of-order core, and the PMU events it exposes	Width, predictor and event names all differ. On an in-order core these experiments get easier to read, not harder
ch28	Four cores, and this interconnect's coherence behaviour	A different core count moves the scaling curve without changing the mechanism. Two cores make the chapter thin
ch30	That <code>perf</code> can sample , not only count	Standard on a mainline ARM kernel. Most affordable RISC-V cores cannot, so this is the chapter a RISC-V reader will find they cannot run
ch31	A vector unit — NEON here	On a RISC-V board without RVV 1.0 it reverts to reasoning about code the compiler emits but the hardware cannot run

A chapter's *mechanism* survives a change of hardware; its *numbers* do not, which is the same split the two targets follow. That is why the book insists on stamping every figure with the machine that produced it, and why [ch25](#) is written as an instruction rather than a table — a cache hierarchy you measured is worth more than one you read.

If none of your numbers resemble the committed ones and you want to know whether that is your board or your method: it is almost always your board, and [ch24](#) is where you learn to tell the difference.

`hardware/README.md` has the requirements, the prompt and the verification step in one place, for when you are standing in front of a shop rather than reading a chapter.

The reference machine's own account of itself

Every figure in [Part V](#) was measured on one board, and what that board says about itself is printed in [ch01](#) rather than here — it is the measurement that chapter takes, and a statement about one piece of silicon rather than about a product line. Read it beside the requirements above if you are comparing a machine you already own.

Where to go next

`hardware/README.md` in the repository carries the same requirements as a checklist, plus the evidence for choosing an ARM board over a RISC-V one: a 2025 study of the three RISC-V cores that can actually be bought [DISSECTING RISC-V PERFORMANCE: PRACTICAL PMU PROFILING AND HARDWARE-AGNOSTIC ROOFLINE ANALYSIS ON EMERGING PLATFORMS \(2025\)](#), and the counter-overflow extension a sampling profiler needs [RISC-V INTERNATIONAL \(N.D.\)](#). [Appendix C](#) is what this board turned out to expose once it reported.