

THE HEARTHWRIGHT'S PRIMER

*A Propædæutic Enchiridion for the Keeping of One's Own Machine,
Kindled from a Single Ember*

CONTENTS

Introduction	1
 Part 0 — Orientation	 5
1. How to use this book	7
 Part I — The machine	 15
2. Bytes, words, and the image	17
3. Privilege	31
4. Memory as the CPU sees it	39
5. Traps, the frame, the UART	45
6. Time and the hart	53
 Part II — A kernel	 57
7. The first process that is not the kernel	59
8. Context switch	75
9. Heap and sbrk	89
10. Pipes and a console	101
11. A block device	117

12. A file system small enough to read	131
13. Userland	149
 Part III — The shapes of an operating system	 165
14. Monolith, microkernel, exokernel, library OS	167
15. What Linux actually is from here	177
16. A look through xv6	203
17. seL4 and the verified kernel	229
 Part IV — Virtualization as hardware	 243
18. The trap-and-emulate story, then hardware	245
19. Nested paging	261
20. KVM as an API	279
21. Device models and the bus you will keep	301
22. Virtio as a protocol	319

INTRODUCTION



This book ends with you running two machines you wrote. One is *ward*: a jailer plus a Linux-KVM VMM you can leave on the NixOS hosts you already have, launching jailed microVMs whose device model you understand because you implemented the rings. The other is *hearth*: a type-1 hypervisor that is itself a unikernel — one address space, no host kernel, no syscall table — booting on the machine and entering a unikernel guest. That second object is not for sale. The first is how you learn *virtio* and guest bring-up against a host that already has files, TAP, and a clock. *hearth* is the same VMM idea after Linux is deleted.

The sibling course is *The Artificer's Primer*. That book builds an intelligence. This one houses it. An optional later joint lets *æcd* tool runs go through *ward*. That coupling does not block this book, and *hearth* is not an agent sandbox until it has a control plane you trust.

Four trees, same contract as *tinygpt* / *æc*:

- *ember* (C, RISC-V in QEMU) is a teaching kernel. Literacy. It may die.

- `wick` (Rust, Linux KVM) is a Firecracker-shaped one-process VMM. You keep this.
- `hearth` (C, RISC-V H-extension, later AArch64 EL2) is the unikernel hypervisor.
- `ward` (Go unless a later decision collapses it into `wick`) supervises the KVM path only. It does not supervise `hearth`; `hearth` *is* the machine.

This is not an undergraduate OS course. You are assumed to already be fluent in Unix, C, Go, NixOS, and building tools from scratch. What is not assumed is privilege and page tables from the CPU's side, nested paging, virtio as a protocol, why Firecracker is small, what a unikernel deleted, and what a hypervisor still has to be once there is no `/dev/kvm`.

Each chapter follows the same shape: why the layer exists, the mechanism, a walkthrough of the reference implementation, exercises, and a lab that boots — QEMU for `ember` / `hearth`, KVM_RUN for `wick` / `ward`. A chapter that only prints listings is not done. If the `hearth` image has a syscall table, it is `wick`.

The book is also available offline, rebuilt on every update: a 6×9 PDF and an EPUB.

The ten parts:

- **Part 0 — Orientation.** How to use this book. The four trees and which may die. Two capstones.
- **Part I — The machine.** C, a linker script, QEMU RISC-V virt. Bytes, privilege, Sv39, traps, time. Every chapter boots. No OS yet.
- **Part II — A kernel (`ember`).** S-mode, two user programs, a heap, pipes, virtio-blk as guest, a small file system, a shell. The guest `hearth` will run, written by you.
- **Part III — The shapes of an operating system.** Isolation budgets, Linux as the host `wick` stands on, xv6, seL4. Read and extend; do not rewrite Linux.
- **Part IV — Virtualization as hardware.** Trap-and-emulate, then the H-extension; nested paging *before* `/dev/kvm`; virtio as a protocol implemented once.
- **Part V — MicroVMs on a host kernel (`wick`).** The Tuesday path, and the device-backend test rig for rings `hearth` will steal.
- **Part VI — Unikernels.** The guest you will host twice. The contract `hearth` will honour is frozen here.
- **Part VII — A unikernel hypervisor (`hearth`).** The image that owns the machine, enter a guest, virtio without a host kernel, several guests,

interrupt injection, the AArch64 dialect.

- **Part VIII — Isolation as a menu.** The ladder from process to confidential VM. Kata is not a VMM. Policy on two floors.
- **Part IX — Supervisors and capstones.** ward on NixOS; hearth on QEMU-as-metal with no Linux in the TCB.

PART 0 — ORIENTATION

1. HOW TO USE THIS BOOK

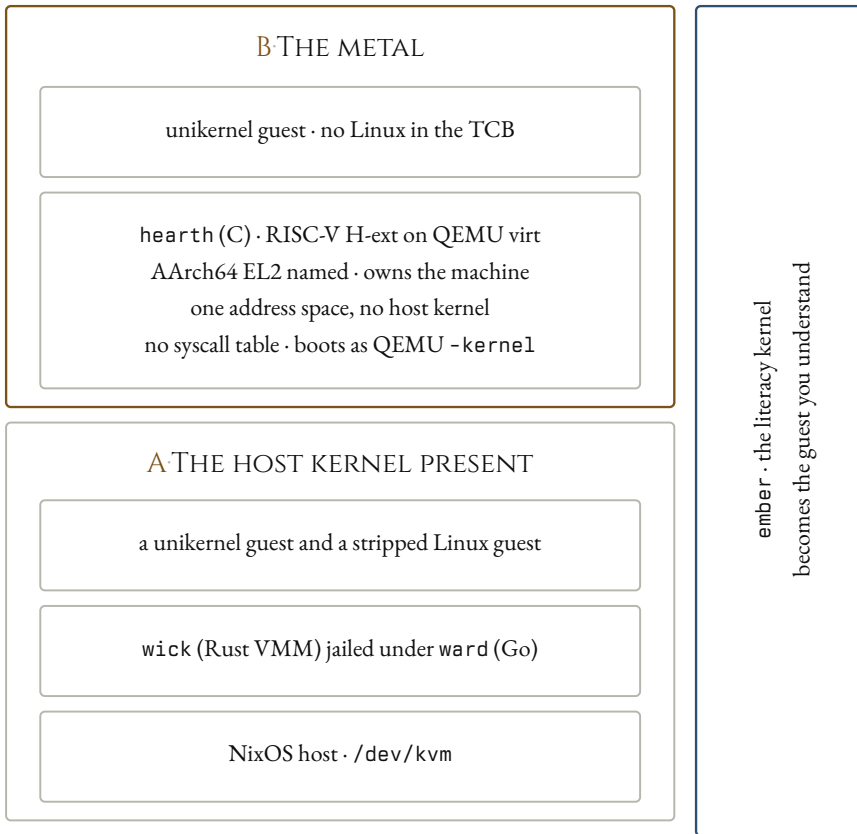
Why this book exists

You already run machines. You have NixOS hosts on a tailnet, systemd units you wrote, containers where they help, and a shell history that reads like a small distribution. What you do not have is a machine you can answer “why did it do that?” about all the way down — because between your software and the silicon sit a Linux kernel, a KVM, and a VMM, each of which you use and none of which you have written.

This book ends that with two machines you wrote.

Capstone A — ward. A jailer plus a Linux-KVM VMM that stays running on your NixOS hosts. `ward` (Go) supervises the KVM path: it holds images, identity, and policy, and launches jailed `wick` (Rust) processes — one per guest — which drive `/dev/kvm` directly. `make check` boots two guests under KVM: a unikernel on `vsock`, and a stripped Linux that runs one command and dies. This is the machine you can put next to `aecd` on a Tuesday.

Capstone B — hearth. A type-1 hypervisor that is itself a unikernel: one address space, no host kernel, no syscall table. It boots as `QEMU -kernel` on the machine, owns the page tables and the UART, and enters a unikernel guest with no Linux in the trusted computing base. The guest console comes out the hypervisor’s UART. This is the object that is not for sale. The first is how you learn `virtio` and guest bring-up against a host that already has files, TAP, and a clock; the second is the same VMM idea after Linux is deleted.



The two floors of the figure are two answers to “who owns the machine?” On the bottom floor, Linux does, and your work wraps it: *wick* under *ward*, guests on top, */dev/kvm* as the door. On the top floor, *heart h* is the machine — there is nothing under it to be jailed by or to call into. *ember*, drawn beside both floors, is the literacy kernel: a small teaching kernel you write in Part II that later becomes, with its userland stripped, the unikernel guest you understand from the inside.

What is granted, what is thin

Granted, and therefore skipped: Unix, C, Go, NixOS, podman, applied crypto, running a fleet, and building tools from scratch. There is no C primer here, no “what is a process”, no Kubernetes, no Raspberry Pi as the teaching machine, no real-mode x86 archaeology, and no hidden plan to replace your NixOS hosts with *heart h*. Capstone B is competence, not a migration plan.

Thin, and therefore where the book spends its time:

- **Privilege and page tables from the CPU's side** — M/S/U and HS/VS/VU modes, control and status registers, the Sv39 walk, what `sret` actually does.
- **Nested paging** — the two-stage translation a hypervisor owns, and why a guest page fault is not a hypervisor page fault.
- **KVM as an API** — what the `ioctl`s wrap, learned after you have driven the same interface by hand.
- **Virtio as a protocol** — descriptor chains, available and used rings, written once and fronted twice.
- **The Firecracker cut** — what a microVM deletes, and why that makes it fast and small.
- **Unikernels as a deleted kernel** — what remains when the user/kernel split inside the guest goes away.
- **What remains of a hypervisor with no `/dev/kvm`** — the question `hearth` answers with its whole design.

The four trees in the repository you cloned, and which may die:

- **ember** (C) — a teaching kernel on RISC-V in QEMU. Literacy. It may die; by the time it could, it has already done its work.
- **wick** (Rust) — a Firecracker-shaped one-process VMM on Linux KVM. Kept.
- **hearth** (C) — the type-1 unikernel hypervisor. The point.
- **ward** (Go) — supervisor for the KVM path only. It does not supervise `hearth`; `hearth` *is* the machine.

C owns anything that owns page tables. Rust owns the Linux VMM. Go owns the supervisor.

How the book is built

Four trees, one spine

The finished repository you cloned has this shape:

```
tsp/
├── ember/                # C, RISC-V teaching kernel (Part II)
├── hearth/               # C, type-1 unikernel hypervisor (Part VII)
├── wick/                 # Rust VMM on Linux KVM (Part V)
├── ward/                 # Go supervisor, KVM path only (Part IX)
├── labs/chNN/           # one directory per lab chapter
├── book/                 # this book (HonKit sources)
└── docs/superpowers/    # specs and plans (not published)
```

```
|— deploy/                # Caddy site block
|— Makefile
```

The four trees are four codebases with one contract: everything a chapter claims about them is checked by `make check`.

The chapter template

Every chapter after this one has the same six sections, in order:

1. **Why this layer exists** — what breaks without it, and where it reappears in wick or hearth.
2. **Mechanism** — the idea, with the hardware interface derived where it is used: CSRs, page-table walks, virtqueue rings, ioctls.
3. **Walkthrough** — code listings extracted from the reference source.
4. **Exercises** — short problems, answers in collapsed blocks.
5. **Lab** — a build task that boots, KVM_RUNs, or `srets` into a guest.
6. **Further reading** — primary sources, cited by title and year.

Labs boot or they are not labs

The labs run in `labs/chNN/`. QEMU hosts `ember` and `hearth`; a Linux host with `/dev/kvm` runs `wick` and `ward` through `KVM_RUN`. A lab skipped because the machine at hand has no `/dev/kvm` is documented in the lab's notes — the skip is an honest state, not a pass, and the lab is rerun on a KVM host before the part is called done. A chapter that only prints listings is not done.

Listings cannot drift

Every code listing in a chapter is a copy of a marked region of a real source file that compiles and passes its tests. In the source, the region is bracketed by comment markers — `/* listing:NAME */` and `/* end-listing:NAME */` in C, with the language's own comment form in other trees. In the chapter, the directive `<!-- listing: PATH#NAME -->` sits on the line immediately before a fenced code block, where `PATH` is relative to the repository root; the block under it is regenerated, never hand-edited.

`tools/listings` walks every Markdown file under `book/`, extracts each named region, and compares it byte-for-byte with the fenced block. `make listings` fails the build on any drift or on a dead `SUMMARY.md` link; `make sync` rewrites stale blocks from source. The consequence: a listing in this book is never a paraphrase, never a version from three refactors ago. When you read a listing, you are reading the code the tests ran against.

The test that names the object: **if the hearth image has a syscall table, it is wick**. A “bare-metal” hypervisor that secretly calls into Linux is not capstone B; the book fails itself if that ever appears.

The route: KVM is a wrapper

The spine is ordered so that interfaces arrive before the machinery that wraps them.

Parts I–III build literacy. Part I is the machine itself — C, a linker script, QEMU’s RISC-V `virt` board; bytes, privilege, Sv39, traps, time; every chapter boots. Part II writes `ember`: a guest-shaped kernel with processes, a heap, pipes, `virtio-blk` as a *guest*, a small file system, userland. Part III reads the shapes of an operating system — monolith, microkernel, exokernel, library OS; what Linux actually is from this vantage; `xv6`; `seL4` — so the vocabulary of isolation budgets exists before virtualization needs it.

Part IV derives the hardware interface first. Trap-and-emulate and the Popek/Goldberg conditions, then the RISC-V H-extension: HS-mode, VS/VU, `hstatus`, `hedeleg`. Chapter 19 is nested paging — two-stage translation, why a guest page fault is not a hypervisor page fault — and its lab is `hearth` chapter zero: identity-map a payload in VS-mode and `sret` into it with no devices. **Only then**, in chapter 20, does `/dev/kvm` appear — as Linux wrapping exactly the interface you just drove by hand. The `ioctl` is not the subject; it is the wrapper. Chapter 21 settles the device bus (`virtio-mmio`) and chapter 22 implements the `virtio` backend once, against a ring in shared memory, to be fronted in both `wick` and `hearth`.

Parts V–VI build the guests and their host. Part V builds `wick`: what a microVM deletes, booting someone else’s kernel, `block/net/vsock`, the jailer, snapshots, density. Part VI builds the unikernels the book will host twice, and chapter 32 freezes the guest contract — a packed boot-info block, not FDT — before `hearth` exists. Nothing in Part VII may change it.

Part VII builds `hearth`. The image that owns the machine; entering a guest; emulation vs passthrough; the `virtio` backend without a host kernel; several guests; interrupts into the guest. **Parts VIII–IX** climb the isolation ladder, build `ward`, and land the two capstones.

The dialects, stated plainly. The written hypervisor ISA is the RISC-V H-extension on QEMU `virt`. The second port is AArch64 EL2 (stage-2, `HCR_EL2`, `vGIC`), the subject of chapter 39, with QEMU `virt, virtualization=on` as a

first-class capstone-B target — and the path onto a Raspberry Pi with EL2 if you ever want one real board. VMX is an optional later dialect, not chapter 39. Apple Silicon is a fine QEMU host and is never a type-1 target.

Environment

Install the host toolchain with `make deps`: on Omarchy it is `omarchy pkg add` for the QEMU system emulators, cross gcc's, and `dtc`; on Debian/Ubuntu the same via `apt`; on NixOS, `nix develop` from `flake.nix` builds the dev shell with everything pinned. `rustc` and `go` come from your own toolchain (`mise`, `rustup`) — `deps` does not install them. Go 1.27 and Node 26 are pinned in `mise.toml` for the book's own tools.

Then run `make doctor`. It prints one line per check and exits non-zero on any failure:

```
ok  make                GNU Make 4.4.1
ok  rsync                rsync version 3.5.0-g471e17dc protocol
   ↪ version 32
ok  dtc                  Version: DTC v1.8.1
ok  qemu-system-riscv64  QEMU emulator version 11.1.1
ok  qemu-system-aarch64  QEMU emulator version 11.1.1
ok  riscv64-gcc          riscv64-linux-gnu-gcc (GCC) 16.2.1
   ↪ 20260810
ok  aarch64-gcc          aarch64-linux-gnu-gcc (GCC) 16.1.0
ok  rustc                rustc 1.98.1 (48a229cea 2026-09-01)
ok  go                   go version go1.27.1 linux/amd64
ok  aarch64-virt-el2     virt,virtualization=on accepted
```

Read it line by line. `make`, `rsync`, and `dtc` are the book's own plumbing. The two QEMU lines are the emulator pair the whole book runs on — one for `ember` and `hearth`, one for the chapter 39 dialect. The two cross lines are the GNU compilers for guest images; the tool accepts the naming variants distributions use (`riscv64-linux-gnu-gcc` on Arch and Debian, `riscv64-unknown-linux-gnu-gcc` under Nix `pkgsCross`, and the `*-unknown-elf-*` bare-metal form). `rustc` builds `wick`, `go` builds `ward` and the book's tools. The last line boots nothing: it asks `qemu-system-aarch64` to accept `-machine virt,virtualization=on`, because a QEMU that rejects EL2 would sink chapter 39's labs. A FAIL on any line means `make deps` (or `nix develop`) did not finish its job on that machine.

Which host plays which role:

Role	Host
QEMU host for <code>ember</code> / <code>hearth-as-metal</code>	any machine, including Apple Silicon
KVM host for <code>wick</code> / <code>ward</code>	a Linux host with <code>/dev/kvm</code> — in practice your NixOS boxes
Never a type-1 target	Apple Silicon (no EL2 under macOS)

Any machine, the Mac included, is a valid QEMU host for `ember` and for `hearth as metal` — that is TCG, and it needs no privileges. The `wick/ward` labs need a Linux host with `/dev/kvm`.

The book keeps the same host-command rules your fleet does, and its labs never ask you to break them: machines are addressed by MagicDNS name, never tailnet IP; nothing is ever piped into an interpreter; and no lab uses a foreground `sleep` as a timer — when a lab needs to wait for something it waits on the thing itself, a pipe or a socket, not on a clock.

Lab: take stock

Nothing boots in this chapter. The first image that boots is chapter 2's. What this lab produces is knowledge of your own machines, written down.

On every machine you intend to use with this book, from the repository root:

```
make doctor
```

Record, in a file outside the repository (never commit it):

- which machine is the QEMU host — the one whose doctor output is fully ok, where `ember` and `hearth-as-metal` labs will run;
- which machine is the KVM host — the Linux box with `/dev/kvm` present, where `wick` and `ward` labs will run;
- every machine where `/dev/kvm` is absent or doctor fails, and what the failing line said.

If doctor fails on the machine you meant to use as the QEMU host, fix that before chapter 2: every chapter from there on boots.

Further reading

The three documents this book keeps open:

- **The RISC-V Instruction Set Manual, Volume II: Privileged Architecture** (RISC-V International; the ratified release in current force, version dated January 2026) — modes, CSRs, the H-extension, Sv39 and two-stage translation.
- **Virtual I/O Device (VIRTIO) Version 1.2** (OASIS, 2022) — the device family and the virtqueue transport this book implements twice.
- **The Definitive KVM (Kernel-based Virtual Machine) API Documentation** (the Linux kernel tree, kept current at docs.kernel.org) — the wrapper chapter 20 unwraps.

And the paper behind the microVM cut, read in Part V:

- **Firecracker: Lightweight Virtualization for Serverless Applications** (Agache et al., NSDI 2020) — what a VMM deletes when its guests are functions, and what that buys.

PART I — THE MACHINE

2. BYTES, WORDS, AND THE IMAGE

Why this layer exists

Every later object in this book — `ember` the kernel, `heart` the type-1 hypervisor — is an image the CPU enters with no one underneath it. That sentence has consequences. A program running under an operating system has three services before its `main` is ever reached: a loader read the executable and placed its segments; an exit path exists, and someone else cleans up; and a stack was set up by runtime startup code that ran first. A type-1 hypervisor is not a kernel with extra flags — it is entered exactly like this image, which is `heart` in miniature: the CPU starts executing its first instruction with the machine entirely its own. Nobody loaded it, so it must sit where the machine already looks. Nobody will catch its exit, so the exit is a device it drives itself. Nobody set a stack, so the first instructions set one before any C runs.

`ember` is that image, small. This chapter builds it far enough to boot, print where the linker placed everything, and exit. The skill is the same one `heart` uses in Part VII when it boots as a `-kernel` with no Linux under it: know where your first byte lies, know how you were entered, own the whole machine from instruction one.

Mechanism

The reset contract

Power-on is not a blank page. On this board, with `-bios none -kernel ember.elf`, the hart (RISC-V for hardware thread: one fetch/decode/execute context with its own control registers) comes out of reset in M-mode at `0x1000`, where QEMU has placed a few instructions of ROM. They are built by `riscv_setup_rom_reset_vec` in QEMU's `hw/riscv/boot.c`:

```
1:  auipc  t0, %pcrel_hi(fw_dyn)    # t0 -> the data words that follow
    addi  a2, t0, %pcrel_lo(1b)
    csrr  a0, mhartid              # a0 = the hart's ID
```

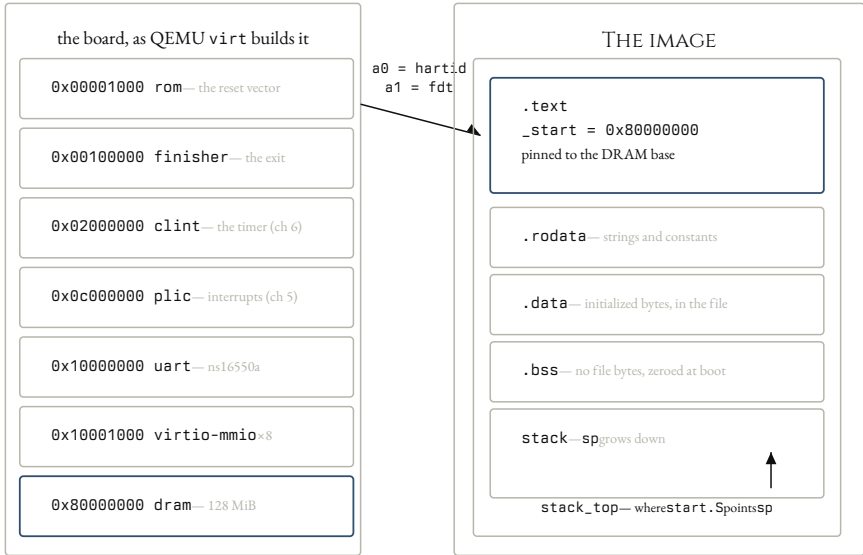
```

ld      a1, 32(t0)          # a1 = where the device tree blob
↳ was loaded
ld      t0, 24(t0)         # t0 = start_addr
jr      t0                 # jump there

```

Three facts fall out, binding for everything this book builds on this board:

1. **a0 = hart ID, a1 = device tree.** The image does not guess which hart it is; the ROM tells it. The flattened device tree (FDT) is a blob the board builder writes describing the hardware. This chapter's image never parses it — every address it uses is on this page — but the blob is there in DRAM, and a guest that wants to discover devices instead of hard-coding them reads it (the lab's last exercise dumps it).
2. **The jump target is a board constant, not your ELF's entry point.** In `hw/riscv/virt.c`, the `start_addr` handed to that ROM is `memmap[VIRT_DRAM].base` — the DRAM base, `0x80000000`. `jr t0` therefore lands on whatever byte is at `0x80000000`. Your kernel, given as `-kernel`, is loaded by `riscv_load_kernel` with its `PT_LOAD` segments copied to their physical addresses (`p_paddr`); the ELF's `e_entry` is recorded only into the firmware-info block that a firmware like OpenSBI would consume — the `-bios none` path never reads it. Link the image's first byte at `0x80000000`, or be loaded but never entered.
3. **The hart is in M-mode.** The RISC-V privileged architecture resets the hart into M-mode; with `-bios none` there is no firmware to hand off to S-mode. We take the machine bare, with everything ours — chapter 3 is the whole story of what M-mode is and how to leave it.



The figure's left column is the board's address ladder; the right is DRAM expanded into the image. The arrow between them is the ROM's jump: everything in this chapter happens along it.

What -bios none removes

The default boot, `-bios default`, loads OpenSBI into DRAM at `0x80000000` and starts it in M-mode; OpenSBI later enters your kernel in S-mode at a higher address. OpenSBI owns M-mode: the timer, the platform services, the first trap handlers. That is the comfortable division every real RISC-V system runs. `-bios none` deletes it: nothing is at `0x80000000` but your image, there is no SBI to call, no one to ask for anything. The price of taking the machine bare is that every M-mode obligation this book eventually names — the traps that stay in M, the timer shim — is yours from reset.

The ELF as a container

The linker emits an ELF with three parts that matter here: **sections** — `.text` for code, `.rodata` for read-only data, `.data` for initialized writable data, `.bss` for zero-initialized data; **program headers** — the loadable segments, each recording a file offset, a size, and the physical address `p_addr` where the bytes land; and `e_entry`, the entry point convention. QEMU's loader (in `hw/core/loader.c`) walks the program headers and copies each `PT_LOAD` segment to its physical address. In this image VMA equals LMA everywhere — the linker script pins

them together, no `AT()` anywhere — and that equality is load-bearing: the image runs untranslated in M-mode, so physical addresses in MMIO and physical addresses in the image are the same numbers, and chapter 4 can turn translation on over an identity map without moving anything.

Why does the ELF carry two addresses at all? Because in a general system they differ: a kernel linked to run at a high virtual address while loaded low needs `AT()` in its link script so the file's bytes sit at the low physical address while symbols resolve to the high virtual one. That split is the norm for S-mode kernels; this image deliberately does without it.

`ENTRY(_start)` stays in the link script anyway. It makes the ELF self-describing — `objdump -f, gdb`, and any loader that does honor entry points read it — and it would be the door in a `-bios default` flow. On this boot path it is decorative, and the chapter says so outright: the acceptance test is behavioral. The lab boots and the transcript matches; relink elsewhere and nothing runs (exercise (b) makes you watch it happen).

The linker script as a contract with the CPU

Read `ember/link.ld` in the walkthrough below; here is what each clause buys.

- `. = 0x80000000;` sets the location counter to the DRAM base. Every section that follows is placed relative to it. Move this line, move the image.
- `KEEP(*(.text.start))` pins the entry section first in `.text`. `KEEP` defeats the linker's garbage collection: `_start` is referenced by no symbol the linker can see (the reset vector jumps to an address, not to a symbol), so without `KEEP` the section could be dropped and the image's first bytes could be anything.
- `.bss (NOLOAD)` gives `.bss` no file bytes. The ELF records the section's address and size, and zero bytes of image; DRAM comes up with arbitrary contents (QEMU happens to zero it; real hardware does not promise to), so `start.S` must zero the range before any C runs. An uninitialized global in C is a promise the image itself keeps.
- The symbols the script defines — `__bss_start`, `__bss_end`, `stack_top` — are addresses, not variables. C sees them as `extern char __bss_start[]`; the value is the link-time constant. That is why the capture can print them: every number was fixed when the linker ran, before the machine booted.

The stack

1a `sp`, `stack_top` is the first instruction of the image for a reason: the ABI requires a valid, 16-byte-aligned `sp` at every call boundary, and no one else will provide one. `stack_top` is the top of a 256 KiB region the script reserves after `.bss`; the stack grows down into it. The region sits inside the image rather than at the top of DRAM to keep every address — code at `0x80000000`, devices at `0x10000000` — inside one ± 2 GiB band, which is what `-mmodel=medany` guarantees for `auipc`-relative addressing. A 128 MiB DRAM would fit either way; the discipline of keeping the image compact pays the moment it grows.

Freestanding C

`-nostdlib -nostartfiles -ffreestanding -fno-builtin` strips assumptions you did not know you had. No startup files: nothing sets a stack, clears `.bss`, or calls `main` — `start.S` does all three. No `libc`: no `memcpy`, no `printf`, no `malloc`, which is why the chapter's UART and number printing are hand-rolled. One caveat is worth stating because it bites eventually: GCC documents that even in freestanding mode it may emit calls to `memcpy`, `memmove`, `memset`, and `memcmp` for operations it chooses to synthesize. This image compiles and links clean — none were emitted — but the first time one appears, the link fails with an undefined reference, and the fix is to provide the four functions, not to hunt a flag.

The UART as raw MMIO

The `ns16550a` UART at `0x10000000` is a bank of byte-wide registers spaced one byte apart — the device tree for this board has no `reg-shift` property. The registers this chapter needs:

Register	Offset	Meaning
THR (write), RBR (read)	+0	the byte to send; the byte that arrived
IER	+1	interrupt enables (chapter 5)
FCR (write), IIR (read)	+2	FIFO control, interrupt identification (chapter 5)
LCR	+3	line control: divisor latch, data format
LSR	+5	line status: bit 5 (0x20) transmitter holding register empty, bit 0 (0x01) data ready

Sending a byte is two steps: wait for LSR bit 5, then store the byte to THR.

Receiving is the mirror: LSR bit 0 says RBR holds an unread byte. The `volatile` on the device pointer is not decoration — it is what tells the compiler the pointed-to state can change between reads, so at -O2 the poll loop cannot be hoisted out or the store to THR sunk past the poll.

One board wart, taught now because the lab shows it: the device tree's `clock-frequency` property for the UART is malformed, and no divisor is ever programmed (the divisor latch lives behind LCR bit 7). Output works anyway — the lab's transcript is proof. QEMU's 16550 model does not gate the transmitter on the divisor; it drains THR as fast as the guest polls. A physical 16550 without its divisor programmed can drop characters or send garbage framing. The lesson is not "QEMU is lax"; it is that the device decides what a register write means, and the device tree's job is to describe what the board builder wired — when it lies, the lab is the arbiter.

The finisher: the only exit the board offers

There is no `exit(2)` for an image and no parent to return to. The `virt` board carries a small device for exactly this: the `SiFive` test finisher at `0x00100000` (`hw/misc/sifive_test.c`). A write whose bits 15:0 are `0x5555` makes QEMU exit with the code in bits 31:16; low bits `0x3333` exit marking a failure; `0x7777` resets the machine. That write is `ember's halt()`. The finisher is a board device, not an ISA feature — a test fiction, named as one — and it is the honest ancestor of the power-off register a real board gives you. `hearth` will use the same register when it wants QEMU to let go.

Walkthrough

The image in five listings, in link order.

`start.S` — the no-one-else instructions. `la sp, stack_top` provides the stack nothing else will. `csrr a2, mhartid` reads the hart ID from the CSR into the third argument register; the ROM put the same number in `a0`, and the capture prints both to prove they agree. The loop zeroes `.bss` eight bytes per store until `__bss_end`, comparing addresses unsigned with `bgeu`. Then `call main`, and — because `main` returning would be undefined — a `wfi` loop (wait for interrupt: the hart idles until an interrupt is pending) that never yields.

```
.section .text.start, "ax", @progbits
.globl _start
_start:
    la      sp, stack_top
```

```

        csrr      a2, mhartid
        la        t0, __bss_start
        la        t1, __bss_end
1:      bgeu      t0, t1, 2f
        sd        zero, 0(t0)
        addi      t0, t0, 8
        j         1b
2:      call     main
3:      wfi
        j         3b

```

How `main` receives its arguments: the reset contract left `a0` (hart ID) and `a1` (FDT) in registers; `start.S` added `a2 = mhartid`. The RISC-V calling convention makes `a0–a7` the first arguments, so `main(hartid, fdt, mhartid)` reads them with no stack and no frame. `main` here is not the hosted `main` the C runtime calls; it is a plain function the image enters with the machine’s initial registers.

`link.ld` — the contract. The location counter’s first assignment pins the base; `KEEP` holds the entry section first in `.text`; `.data` absorbs the compiler’s small-data section (`.sdata`) so every initialized global lands in one place; `.bss` (`NOLOAD`) fixes the zero-promise; the reserved stack region closes the image and defines `stack_top`.

```
. = 0x80000000;
```

SECTIONS

```

{
    .text : {
        KEEP(*(.text.start))
        *(.text .text.*)
    }

    .rodata : {
        *(.rodata .rodata.*)
    }

    .data : {
        *(.data .data.*)
        *(.sdata .sdata.*)
    }

    .bss (NOLOAD) : {
        __bss_start = .;
        *(.sbss .sbss.*)
        *(.bss .bss.*)
    }
}

```

```

    *(COMMON)
    . = ALIGN(8);
    __bss_end = .;
}

/* Stack, growing down from the top of the reserved region. */
. = ALIGN(16);
. = . + 0x40000; /* 256 KiB stack region */
. = ALIGN(16);
stack_top = .;

. = ALIGN(4096);
pt_pool_start = .;
. = . + 0x80000; /* 512 KiB page-table pool (ch 4) */
pt_pool_end = .;
}

```

uart.c — the polling driver. The while on LSR bit 5 is the handshake; the store to THR is the send. uart_getc returns the byte in RBR when LSR bit 0 says one arrived, else -1 — the receive side is unused this chapter and chapter 5's interrupt path builds on it. The offsets are the byte-spaced table above; a board whose device tree declares reg-shift spaces every register by 4 instead, and exercise (e) makes the difference felt.

```

#define UART0 0x10000000UL

#define THR 0x0 /* transmit holding register (write) */
#define RBR 0x0 /* receive buffer register (read) */
#define LSR 0x5 /* line status register */

#define LSR_THR 0x20 /* THR empty: the transmitter can take another
↳ byte */
#define LSR_DR 0x01 /* data ready: RBR holds an unread byte */

static volatile unsigned char *const uart = (volatile unsigned char
↳ *)UART0;

void uart_putc(char c)
{
    while ((uart[LSR] & LSR_THR) == 0)
        ;
    uart[THR] = (unsigned char)c;
}

void uart_puts(const char *s)

```

```

{
    while (*s)
        uart_putc(*s++);
}

int uart_getc(void)
{
    if ((uart[LSR] & LSR_DR) == 0)
        return -1;
    return uart[RBR];
}

```

halt.c — the finisher write. Clean exit: 0x5555. Failure: the code packed into bits 31:16 over 0x3333. The loop after the write exists because halt must not return on any board, and on a board without a finisher it never would.

```

#define FINISHER 0x00100000UL

#define FINISHER_PASS 0x5555 /* status: shut down, exit code in bits
↳ 31:16 */
#define FINISHER_FAIL 0x3333 /* status: panic, exit code in bits 31:16
↳ */

static volatile unsigned int *const finisher = (volatile unsigned int
↳ *)FINISHER;

void halt(int code)
{
    finisher[0] = code
        ? ((unsigned int)code << 16) | FINISHER_FAIL
        : FINISHER_PASS;

    /* Not reached on this board. */
    for (;;)
        ;
}

```

main.c — the proof transcript. puthex prints a 64-bit value in hex with no library; sym_line prints one linker symbol's value — the symbol's address, not a pointer dump of memory. The two proofs the lab pins: .data survived the load (data_word == 0x600d — bytes that were in the file), and .bss starts at zero (bss_probe[0] == bss_probe[31] == 0 — bytes that were not). If a0 and mhartid disagree, the boot story is wrong; if either proof fails, the machine did not keep the ELF's promises. Then PASS, halt(0).

```

/* Linker symbols: their values are addresses, fixed at link time. */
extern char _start[], __bss_start[], __bss_end[], stack_top[];

/* In .data: the image carries both these bytes and their address. */
static volatile unsigned long data_word = 0x600d;
/* In .bss: no file bytes; start.S zeroes the range before main runs. */
static volatile unsigned long bss_probe[32];

static void puthex(unsigned long v)
{
    static const char digits[] = "0123456789abcdef";
    int started = 0;

    uart_puts("0x");
    if (v == 0) {
        uart_putc('0');
        return;
    }
    for (unsigned long shift = 60;; shift -= 4) {
        unsigned long d = (v >> shift) & 0xf;
        if (d != 0 || started || shift == 0) {
            uart_putc(digits[d]);
            started = 1;
        }
        if (shift == 0)
            break;
    }
}

#if HP_CH < 3
static void sym_line(const char *label, unsigned long addr)
{
    uart_puts("ember: ");
    uart_puts(label);
    puthex(addr);
    uart_puts("\r\n");
}
#endif

static void ch2_report(unsigned long hartid, unsigned long mhartid)
{
    if (hartid != 0 || hartid != mhartid)
        halt(1);
}

```

```

uart_puts("ember: hart a0=");
puthex(hartid);
uart_puts(" mhartid=");
puthex(mhartid);
uart_puts("\r\n");

#if HP_CH < 3
    /* Link-time addresses belong to chapter 2. Later labs must not
     * pin them: a correct mine that differs in size would fail the
     * diff before its stub ever ran. */
    sym_line("_start", (unsigned long)_start);
    sym_line("__bss_start", (unsigned long)__bss_start);
    sym_line("__bss_end", (unsigned long)__bss_end);
    sym_line("data_word", (unsigned long)&data_word);
    sym_line("stack_top", (unsigned long)stack_top);
#else
    (void)_start;
    (void)__bss_start;
    (void)__bss_end;
    (void)stack_top;
#endif

    uart_puts("ember: .data survived: data_word = ");
    puthex(data_word);
    uart_puts("\r\n");

    uart_puts("ember: .bss zero: bss_probe[0] = ");
    puthex(bss_probe[0]);
    uart_puts(", bss_probe[31] = ");
    puthex(bss_probe[31]);
    uart_puts("\r\n");
}

```

Exercises

(a) Chapter 4 translates addresses through Sv39, which slices a 64-bit virtual address as VPN[2] (bits 38:30), VPN[1] (29:21), VPN[0] (20:12), page offset (11:0). Which level-2 index does 0x80000000 fall in — the slot number in the 512-entry root table?

Answer, p. 29

(b) Relink ember at 0x80400000 — change the one line in link.ld, rebuild — and boot it with the lab's QEMU command. Describe what happens, and explain it from the reset vector.

Answer, p. 29

(c) Grow `.bss` by 2 MiB (a large uninitialized array) and rebuild. What happens to the ELF's size on disk, and why? What happens at boot?

Answer, p. 30

(d) In `start.S`, point the stack into the UART window: `li sp, 0x10000008`. What happens? Explain.

Answer, p. 30

(e) In `mine/`, poll LSR at +4 instead of +5. What do you see, and what does the missing `reg-shift` have to do with it?

Answer, p. 30

(f) Dump the board's device tree and locate every node the figure's left column shows:

```
qemu-system-riscv64 -machine virt,dumpdtb=virt.dtb -m 128M
dtc -I dtb -O dts -o virt.dts virt.dtb
```

Answer, p. 30

Lab

From the repository root:

```
make lab CH=02
```

builds the reference image, boots it under a 30-second timeout, feeds it the lab's (empty) `input.txt` on stdin, and diffs the serial transcript against the committed `capture.txt`:

```
ember: hart a0=0x0 mhartid=0x0
ember: _start      0x80000000
ember: __bss_start 0x800007a0
ember: __bss_end   0x800008a0
ember: data_word   0x80000660
ember: stack_top   0x800408a0
ember: .data survived: data_word = 0x600d
ember: .bss zero: bss_probe[0] = 0x0, bss_probe[31] = 0x0
PASS
```

The capture pins the whole chapter. `_start` at `0x80000000` is the reset contract; the symbol lines are the linker's contract; `.data` intact and `.bss` zero are the ELF's contract; the two hart IDs agreeing is the register handoff. Every number

is a link-time constant, so the transcript is byte-identical on every run — the diff, not QEMU’s exit status, is the acceptance.

Your copy is `labs/ch02/mine/`: the same tree with `uart_putc`, `uart_getc`, and `halt` stubbed as silent no-ops. Run:

```
make lab CH=02 IMPL=mime
```

and it fails loudly — an empty transcript against the capture, then the timeout’s kill message. Bring up the console and the exit: the register table and the finisher encoding are everything you need. When the diff goes quiet, you have written the image’s two device drivers, and chapter 3 can change the privilege mode they run in.

Further reading

- **QEMU RISC-V virt board** — the board every lab in this book boots: its devices, boot options, and the device tree it passes.
- **Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification, Version 1.2** (TIS Committee, May 1995) — sections, program headers, and the distinction this chapter leans on: `p_paddr` is where bytes land; `e_entry` is a convention.
- **The RISC-V Instruction Set Manual, Volume II: Privileged Architecture** (RISC-V International; the ratified release in current force, version dated January 2026) — reset behavior and M-mode are chapter 3’s subject; the CSR map is worth a skim now.

Answers

(a) `0x80000000` is bit 31 set, all else zero — a legal Sv39 address (bits 63:39 must copy bit 38, which is 0). `VPN[2]` is bits 38:30: `0x80000000 >> 30 = 2`, and 2 is below 512, so the level-2 index is 2. Each root slot spans 1 GiB, so the DRAM base sits at the start of the third gigabyte of the address space. Chapter 4’s identity map begins by pointing root slot 2 at DRAM.

(b) QEMU loads the ELF — the loader honors the program headers — and then nothing happens: no transcript, and the lab’s `timeout` kills the machine. The reset vector jumps to `start_addr`, the board’s DRAM base `0x80000000`, which your image no longer occupies. DRAM below your link base reads as zeros; the all-zero instruction is illegal, so the hart traps on entry, and with no handler installed the trap loops — but the observable is simply: your bytes are at

0x80400000, execution is at 0x80000000, and the two never meet. The board's jump target does not read your ELF.

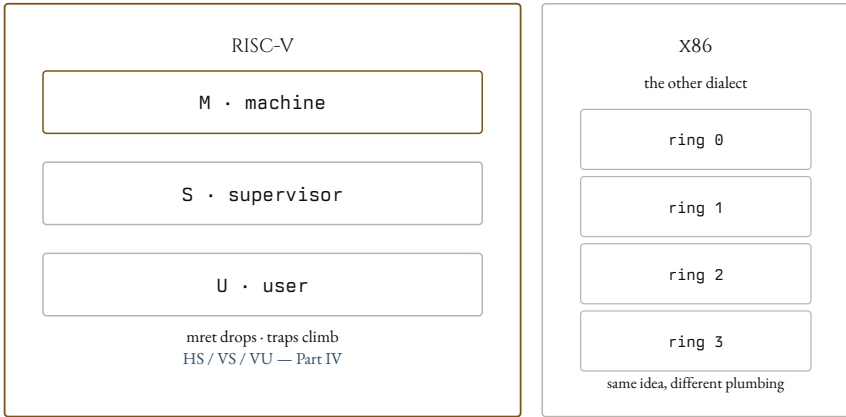
(c) The ELF barely changes — section-size fields move a little; no file bytes are added, because `.bss` (NOLOAD) records only address and size. At boot, the clear loop in `start.S` now runs ~262,144 stores (2 MiB / 8 bytes per store) — instant at emulator speeds, measurable on hardware. That is the whole point of NOLOAD: memory costs file bytes only when you initialize it.

(d) The first stack store — the return address `main`'s prologue saves — lands on the UART at 0x10000000: THR. Eight bytes of a return address go out the serial line as garbage characters, and the stack continues growing down across IER, FCR, LCR — interrupt and FIFO registers, scribbled with fragments of saved registers. Exactly what you see depends on the store order the compiler chose; what you learn does not: a device region is not write-memory. Every store has device semantics, so `sp` must name RAM.

(e) Silence — the poll never satisfies. On this board the 16550's registers are byte-spaced (no `reg-shift` in the DTB), so offset +4 is register 4: MCR, the modem control register, whose bit 5 is not transmitter-empty. The poll spins forever, nothing prints, and the timeout kills the machine. On a 32-bit-strided 16550 — `reg-shift` = 2, four address bytes per register — register 5 would live at offset +20, and offset +4 would be IER. Same chip, different wiring: the stride is a board property the device tree describes, which is why a portable driver treats it as data instead of baking it into the code.

(f) `memory@80000000` carries the DRAM base and size (0x8000000 = 128 MiB); `test@100000` is the finisher; `clint@2000000`; `plic@c000000`; `uart@10000000`; and eight `virtio_mmio` nodes from 0x10001000 to 0x10008000. The one box you will not find is the ROM: the reset vector at 0x1000 is built by QEMU (`riscv_setup_rom_reset_vec`) and is not described in the DTB, because the DTB is written for the software the ROM is about to launch.

3. PRIVILEGE



Why this layer exists

The CPU has a bit that says who is speaking. Every isolation story in this book is that bit, moved. User processes in chapter 7 are U-mode refused a page and a CSR. Nested paging in chapter 19 is HS-mode refusing a guest store. `hearth` entering a unikernel is the same `mret` this chapter executes, aimed at a different `mepc`.

A kernel is not a program that happens to live in a file named `vmlinux`. A program is fetched, has a stack someone else set, and `exits`. This image was jumped to from the reset vector with no one underneath it. The lie that “kernel” is just a program dies at `mret`: the instruction does not call a function. It writes the privilege field and the PC together, and the next fetch is a different kind of fetch.

Mechanism

The ladder

RISC-V privilege is a three-rung ladder on this hart: M, S, U. A trap always climbs (or stays); `mret` and `sret` are the only way down. There is no instruction that “calls” S-mode. HS, VS, and VU — the hypervisor rungs — exist on this board’s ISA string (h is in the DTB). They are named here and unused. Chapter 18 starts that climb; chapter 19’s lab is the first `sret` into VS.

x86 has the same idea with different plumbing: ring 0 is the kernel, ring 3 is user, rings 1 and 2 are the historical middle that almost nobody uses. The rings are not M/S/U. Do not translate CSRs into them.

CSRs

Control and status registers are the CPU’s memory that is not memory. `csrr`, `csrw`, `csrrs`, `csrrc` (and the immediate forms) are the only access. A CSR address is a 12-bit index encoded in the instruction, not a load effective address. S-mode executing `csrr t0, mhartid` is not a bad pointer. It is an illegal instruction: the encoding names a register S is not allowed to read.

The accessors this image uses:

```
#define csr_read(csr)
↳ \
    ({
↳ \
        unsigned long __v;
↳ \
        __asm__ volatile("csrr %0, " #csr : "=r"(__v));
↳ \
        __v;
↳ \
    })

#define csr_write(csr, val)
↳ \
    __asm__ volatile("csrw " #csr ", %0" : : "rK"(val))

#define csr_set(csr, val)
↳ \
    __asm__ volatile("csrs " #csr ", %0" : : "rK"(val))
```

```

#define csr_clear(csr, val)
↵ \
    __asm__ volatile("csrrc " #csr ", %0" : : "rk"(val))

/* mstatus RV64, privileged spec. Bit numbers are the spec's. */
#define MSTATUS_SIE    (1UL << 1)
#define MSTATUS_MIE    (1UL << 3)
#define MSTATUS_SPIE    (1UL << 5)
#define MSTATUS_MPIE    (1UL << 7)
#define MSTATUS_SPP    (1UL << 8)
#define MSTATUS_MPP_SHIFT 11
#define MSTATUS_MPP    (3UL << MSTATUS_MPP_SHIFT)
#define MSTATUS_MPP_U (0UL << MSTATUS_MPP_SHIFT)
#define MSTATUS_MPP_S (1UL << MSTATUS_MPP_SHIFT)
#define MSTATUS_MPP_M (3UL << MSTATUS_MPP_SHIFT)

```

mstatus is the switchboard. The fields this chapter freezes, from the privileged spec, RV64:

Field	Bits	Meaning
SIE	1	S-mode interrupt enable
MIE	3	M-mode interrupt enable
SPIE	5	SIE stacked on trap
MPiE	7	MIE stacked on trap
SPP	8	privilege stacked for sret
MPP	12:11	privilege stacked for mret (00=U, 01=S, 11=M)

mret does four things as one instruction: $PC \leftarrow mepc$; $\text{privilege} \leftarrow \text{MPP}$; $\text{MIE} \leftarrow \text{MPIE}$; $\text{MPIE} \leftarrow 1$. That is the door. mtvec holds the M-mode trap base in bits 63:2 and MODE in 1:0; MODE=0 is direct (every trap to the base). mcause is the reason: bit 63 set means interrupt, the rest is the code. Illegal instruction is 2. mtval for cause 2 is the faulting instruction word.

medeleg / mideleg are the hinge. A bit set in medeleg sends that exception to S (stvec) instead of M. This chapter leaves them zero: everything still traps to M, which is how we prove S ran. Chapter 5 turns the hinge.

The drop

```

void drop_to_s(void (*entry)(void))
{
    unsigned long s;

```

```

pmp_open();
csr_write(mtvec, (unsigned long)mshim_entry);
/* Unconditional: mshim_entry swaps through mscratch on every M
 * trap. From ch 8 the timer can fire while U runs; for traps from
 * S (ch 3-7) the shim now runs on its own stack instead of the
 * kernel's, which nothing in those chapters observes. */
csr_write(mscratch, (unsigned long)mshim_sp);
#if HP_CH >= 6
    timer_enable_m();
#endif
s = csr_read(mstatus);
s = (s & ~MSTATUS_MPP) | MSTATUS_MPP_S;
csr_write(mstatus, s);
csr_write(mepc, (unsigned long)entry);
__asm__ volatile("mret");
}

```

On this QEMU 11.1.1 virt hart, reset leaves `mstatus.MPP=U`. That is not a spec requirement; it is what the board does. The drop writes `MPP=S` before `mret`. It also programs PMP. With no matching PMP entry, this QEMU refuses S/U fetches: `mcause 1`, instruction access fault, `mtval 0`, and `S` never prints. One TOR region from 0 covering ROM, MMIO, and DRAM (`pmpaddr0 = 0x100000000 >> 2`, `pmpcfg0 = R|W|X|TOR`) is enough. The lab observed this; a different CPU may not enforce PMP the same way. Record what *this* board did.

The M-mode shim is the landing:

```

void mshim(unsigned long cause, unsigned long tval)
{
    unsigned long code = cause & ~MCAUSE_INT;
    int expected = 0;

    #if HP_CH >= 6
        if (timer_claim_m(cause))
            return;
    #endif

    uart_puts("M: trap cause ");
    uart_puthex(code);
    if (code == MCAUSE_ILLEGAL) {
        uart_puts(" (illegal instruction) mtval=");
        uart_puthex(tval);
        uart_puts(" - S-mode proven\r\n");
    }
}

```

```

} else if (code == MCAUSE_STORE_PAGE || code == MCAUSE_LOAD_PAGE)
{
    uart_puts(" (page fault) mtval=");
    uart_puthex(tval);
    uart_puts("\r\n");
} else {
    uart_puts(" mtval=");
    uart_puthex(tval);
    uart_puts("\r\n");
}

#if HP_CH == 3
    expected = (code == MCAUSE_ILLEGAL);
#elif HP_CH == 4
    expected = (code == MCAUSE_STORE_PAGE);
#endif
if (expected) {
    uart_puts("PASS\r\n");
    halt(0);
}
halt(1);
}

```

Direct `mtvec`: `mshim_entry` in `mshim.S` saves every caller-saved integer, reads `mcause` and `mtval` into `a0/a1`, and calls `mshim`. Exceptions halt. From chapter 6 a machine timer interrupt returns via `mret`; a C clobber of `a3–a5` would otherwise corrupt the interrupted S-mode code. `PASS` is only for this chapter’s illegal instruction and chapter 4’s page fault. Any other cause is a fail.

`drop_to_s` also parks a shim stack top in `mscratch` — M-mode’s scratch CSR, read/write with no hardware meaning — and `mshim_entry` opens by swapping `sp` for it and closes by swapping back before `mret`, so the shim never pushes on the interrupted world’s stack. Nothing in this chapter needs that: every trap here comes from S, whose `sp` is DRAM the shim could have used. From chapter 8 the timer can interrupt U-mode code, whose `sp` is a low virtual address that M-mode does not translate and cannot store through; chapter 8 derives the fault and the swap.

The proof

```

static void ch3_drop(unsigned long hartid)
{
    unsigned long s = csr_read(mstatus);
    unsigned long mpp = (s & MSTATUS_MPP) >> MSTATUS_MPP_SHIFT;

```

```

    uart_puts("M: hart ");
    uart_puthex(hartid);
    uart_puts("\r\n");
    uart_puts("M: mstatus MPP=");
    uart_putc(mpp == 3 ? 'M' : mpp == 1 ? 'S' : 'U');
    uart_puts("\r\n");
#ifdef HP_CH >= 5
    trap_init_m();
#endif
    drop_to_s(s_entry);
}

```

S prints S: running, then executes `csrr t0, mhartid`. That instruction's encoding is `0xf14022f3`. S is not allowed to read `mhartid`. The trap climbs to M, the shim prints cause 2 and that `mtval`, and we halt. If S had been a fiction — if we were still in M — `csrr mhartid` would have succeeded.

Walkthrough

The listing order is the drop: accessors, `drop_to_s`, the shim, then `ch3_drop` from `main.start`. S still passes `a0=mhartid` from the reset vector and `a2` from `csrr mhartid`; chapter 2 already showed they match. Chapter 3 uses `a0` only as the hart printed in M.

`HP_CH` selects how far `main` runs. Chapter 2's lab builds with `HP_CH=02` and never links `drop.c`. Chapter 3 links them. Mixing the object files into chapter 2's image would move `.data` and break its capture.

Exercises

(a) Drop $M \rightarrow U$ (`MPP=00`) instead of S. Where does `csrr mhartid` land, and why is that not a different proof?

Answer, p. 37

(b) Set `MPP=11` (M) and `mret`. What changes?

Answer, p. 37

(c) Clear `MPIE` before `mret`. Do interrupts fire?

Answer, p. 37

(d) Set `medeleg` bit 2 and `predict`, without capturing a crash.

Answer, p. 37

(e) Read `misa` and check it against the DTB ISA string.

Answer, p. 38

(f) Why can `sret` not climb?

Answer, p. 38

Lab

```
make lab CH=03
make lab CH=03 IMPL=mime
```

The reference transcript includes chapter 2's report minus its symbol lines, then M: `mstatus MPP=U` (this board's reset), S: `running`, and the cause-2 line with `mtval=0xf14022f3`. `IMPL=mime` stubs `drop_to_s` so S never runs. The skip is not a pass.

If doctor is green and the reference still traps with cause 1, PMP is the first suspect: no TOR entry, or `pmpcfg0` not R|W|X.

Further reading

- The RISC-V Instruction Set Manual, Volume II: Privileged Architecture (RISC-V International, ratified v20260120, January 2026) — `modes`, `mstatus`, `mret`, PMP, `medeleg`.
- QEMU RISC-V virt machine (QEMU, current) — the board this drop runs on.

Answers

(a) U-mode also cannot read `mhartid`. The trap still climbs to M, cause 2. You have proved a drop, not S-mode. The S: `running` line is the S-specific evidence; an `MPP=U` drop never prints it from S.

(b) Privilege stays M. `mret` is then a glorified jump through `mepc`. `csrr mhartid` succeeds. The shim never runs. The capture's cause-2 line is gone.

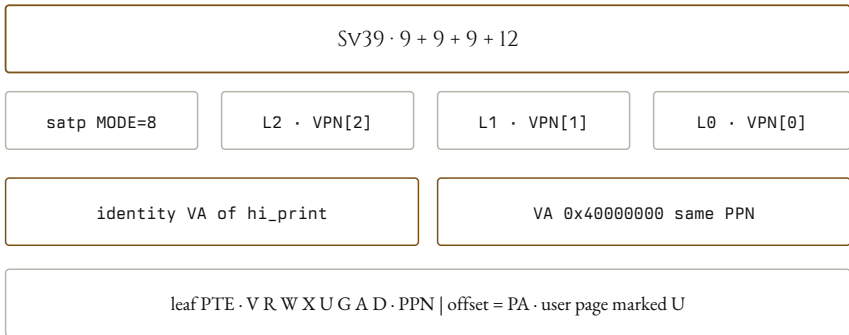
(c) Nothing interrupt-shaped happens in this chapter. `mret` copies MPIE into MIE, so clearing MPIE leaves MIE clear after the drop. Chapter 5 is where the gates matter.

(d) Cause 2 now goes to `stvec`, which is still 0. The fetch at 0 is an instruction-access fault (cause 1). That cause is not delegated, so it lands in the M shim — not a silent death, and not this chapter's PASS. Do not commit that transcript; the prediction is the exercise.

(e) `misa` bits name extensions. This board's DTB includes `h`, `sstc`, `svadu`. `misa`'s `H` bit should be set. A mismatch is a QEMU/DTB story, not a CSR bug — write down both.

(f) The spec: `sret` sets privilege from SPP, which holds U or S, never M. There is no encoding in SPP for M. Climbing is traps; descent is `*ret`.

4. MEMORY AS THE CPU SEES IT



Why this layer exists

The CPU does not fetch from “the address you typed.” It fetches from a translated view. Whoever owns that translation owns every load, store, and instruction. This walk is the object `hcart` will double (G-stage, chapter 19) and the object `wick` will consume already-built by hardware (EPT under KVM). Nested paging is promised here and not built: a guest page fault is a walk that stops in the guest’s tables; a hypervisor page fault is a walk that stops in the hypervisor’s. Two walks, two `*tvals`, two owners.

Mechanism

Physical addresses are what the bus uses. Virtual addresses are what S-mode uses once `satp.MODE` is `Sv39` (8). `satp` is one register: `MODE` in 63:60, `ASID` in 59:44, `root PPN` in 43:0. Writing it switches the world. The instruction after the write must still be fetchable, so the identity map has to cover that PC before the write.

`Sv39` splits a 39-bit VA into `VPN[2]|VPN[1]|VPN[0]|offset`, $9+9+9+12$. Each table has 512 PTEs, 4 KiB, aligned. A PTE is V, R, W, X, U, G, A, D, then PPN

at bits 53:10. A leaf is V and (R or X). W without R is reserved at every level. A non-leaf is V with R, W, and X clear. Superpages are a leaf at L1 or L2; this chapter maps 4 KiB so the walk stays readable.

A and D are set at map time. This hart's DTB reports svadu; the code does not rely on hardware PTE updates. The DTB mmu-type is sv57; we teach Sv39 because every RVA23-capable board must reach it.

```
int pte_leaf(unsigned long pte)
{
    return pte_valid(pte) && (pte & (PTE_R | PTE_X)) != 0;
}

unsigned long *walk(unsigned long va)
{
    unsigned long *table = root;
    int level;

    for (level = 2; level >= 0; level--) {
        unsigned vpn = (unsigned)((va >> (PAGE_SHIFT + 9 * level)) &
            0x1ff);
        unsigned long *slot = &table[vpn];
        if (level == 0)
            return slot;
        if (!pte_valid(*slot)) {
            void *next = alloc_table();
            *slot = make_ptr_pte(next);
        } else if (pte_leaf(*slot)) {
            return 0;
        }
        table = (unsigned long *) (pte_ppn(*slot) << PAGE_SHIFT);
    }
    return 0;
}
```

walk allocates missing non-leaf tables from a pool reserved after the stack in the linker script, outside .bss, so nothing silently overwrites it. map_page plants a leaf:

```
void map_page(unsigned long va, unsigned long pa, unsigned long flags)
{
    unsigned long *slot = walk(va);
    if (!slot)
        halt(1);
    *slot = make_leaf_pte(pa, flags);
}
```

```

void mem_init(void)
{
    unsigned long fn = (unsigned long)hi_print;
    unsigned long page = fn & ~(PAGE_SIZE - 1);
    unsigned long off = fn & (PAGE_SIZE - 1);
    void (*via_second)(void);

    pt_init();
    map_identity_dram();
    map_mmio(0x00100000UL);
    map_mmio_range(0x02000000UL, 0x100000UL);
    map_mmio_range(0x0c000000UL, 0x400000UL);
    map_mmio(0x10000000UL);
    map_page(SECOND_VA, page, PTE_R | PTE_X);
    map_page(0x81000000UL, 0x81000000UL, PTE_R | PTE_W | PTE_U);

    csr_write(satp, SATP_MODE_SV39 | pt_root_ppn());
    __asm__ volatile("sfence.vma" ::: "memory");

    via_second = (void (*)(void))(SECOND_VA + off);
    via_second();
    #if HP_CH < 5
        *(volatile unsigned long *)BAD_VA = 1UL;
        uart_puts("S: store to unmapped VA survived\r\n");
        halt(1);
    #endif
}

```

The identity map covers 128 MiB of DRAM plus the MMIO pages this image actually uses (finisher, CLINT, PLIC, UART). A second mapping at VA 0x40000000 points at the physical page of `hi_print`. That function is page-aligned, puts its string on the stack, and writes the UART with no call — RISC-V call is `auipc`, so a PC at the second VA cannot reach identity-mapped `uart_puts`. Calling through that VA is the proof of translation: same PPN, different VPN. A user page at 0x81000000 is mapped U=1 and unused; chapter 9 reclaims it. Then a store to 0x12345000, which is not mapped. Nothing is delegated, so the fault climbs to M. The shim prints `mcause 15` (store page fault) and `mtval`, not `stval` — this is still an M-mode trap. M-mode itself is untranslated; the identity map makes the numbers coincide, which is deliberate.

Walkthrough

`ch04-pte-bits`, `ch04-walk`, `ch04-map-page`, `ch04-enable-mmio`. The M shim gained a page-fault line; chapter 3's illegal-instruction line is unchanged.

Exercises

(a) Hand-walk `0x80000000` through the three VPN indices and name the L2 slot.

Answer, p. 42

(b) Flip U on the identity map. What changes in this chapter?

Answer, p. 42

(c) Map the second VA read-only and execute through it.

Answer, p. 43

(d) How many 4 KiB pages and tables does a full identity map of 128 MiB DRAM need?

Answer, p. 43

(e) Why must the identity map be live before the `satp` write?

Answer, p. 43

(f) Sv48 on paper: what changes in the split?

Answer, p. 43

Lab

```
make lab CH=04
```

```
make lab CH=04 IMPL=mime
```

The capture pins S: `second map` and `mtval=0x12345000`. `IMPL=mime` stubs `mem_init`.

Further reading

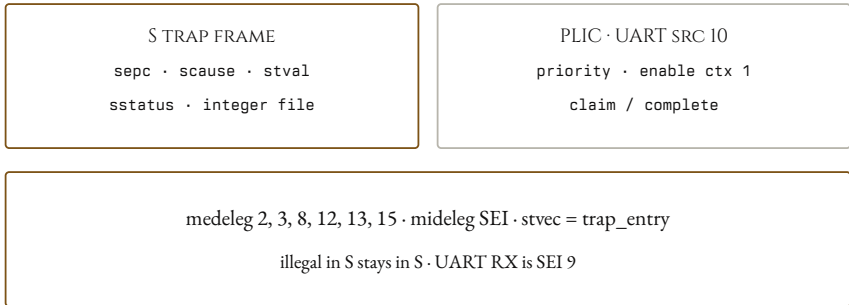
- The RISC-V Instruction Set Manual, Volume II: Privileged Architecture (RISC-V International, v20260120, January 2026) — Sv39, `satp`, PTE bits.

Answers

(a) `VPN[2] = 0x80000000 >> 30 = 2`. `VPN[1] = VPN[0] = 0`. The L2 index is 2; that slot's PPN is the L1 table for the DRAM identity.

- (b) S-mode fetch from a $U=1$ page is an instruction page fault (cause 12), immediately. Loads and stores fault unless `sstatus.SUM=1`. The bit is already a denial. Chapter 7 makes it the user/kernel split on purpose.
- (c) Cause 12, instruction page fault. X was clear.
- (d) 32768 leaf pages. 64 L0 tables ($512 \text{ PTEs} \times 4 \text{ KiB} = 2 \text{ MiB}$ per L0 table, $64 \times 2 \text{ MiB} = 128 \text{ MiB}$), one L1 table, one L2 table: 66 tables.
- (e) The next instruction's VA is the physical PC. Without that leaf, the fetch after `csrw satp` walks to nothing and the hart dies before `sfence.vma`.
- (f) A fourth VPN[3], 9 more bits, 48-bit VA, one more table level. The leaf format is the same.

5. TRAPS, THE FRAME, THE UART



Why this layer exists

A hypervisor is a trap handler with opinions. Nested paging, virtio, the KVM_RUN loop — every one of those is “something happened in the guest, here is a frame, decide.” This chapter freezes the frame on a single hart, in S-mode, with no guest yet. The same four CSRs (`sepc`, `scause`, `stval`, `sstatus`) will still be the story in chapter 34; only the cause numbers grow.

Mechanism

Chapter 3’s `medeleg`: bits 2, 3, 8, 12, 13, 15 send illegal instruction, breakpoint, `ecall` from U (`ecall` is the instruction a U program executes to ask the kernel for service), and the three page-fault causes to S. Written from M, before `mret`. Writing it from S is itself illegal and lands in the M shim with `mtval` of a `csrw medeleg`. `stvec` is an S CSR; it is written after the drop.

The S entry saves the integer file and the four CSRs (`sepc`, `scause`, `stval`, `sstatus`) into a 280-byte frame that matches `struct trapframe`. `sstatus` lives at offset 272; a 272-byte frame would leave that slot past the save. Direct mode, `stvec` MODE 0, aligned. `sret` restores `sstatus` the hardware already stacked:

SPIE→SIE, SPP→privilege. The stacked sstatus is for later chapters that read tf→sstatus, not for sret.

```
void trap_init_m(void)
{
    unsigned long medeleg = (1UL << 2) | (1UL << 3) | (1UL << 8) |
                           (1UL << 12) | (1UL << 13) | (1UL << 15);
    #if HP_CH >= 19
        /* VS ecall and the guest-page faults come to HS (chapter 19). */
        medeleg |= (1UL << 10) | (1UL << 20) | (1UL << 21) | (1UL << 22) |
        ↪ (1UL << 23);
    #endif
    csr_write(medeleg, medeleg);
    csr_write(mideleg, (1UL << 1) | (1UL << 9));
}

void trap_init(void)
{
    csr_write(stvec, (unsigned long)trap_entry);
}

void uart_irq_on(void)
{
    plic_init_uart();
    uart_enable_rx();
    /* Drain what QEMU already stuffed with interrupts masked – and
     * mask them here rather than assume they are: a driver brought up
     * before the console may already have set sstatus.SIE and
     * sie.SEIE, and the PLIC line just enabled above would then nest
     * this handler inside its own loop, between an LSR read and the
     * RBR read it belongs to. Bytes echo twice and land out of order.
     * Anything that arrives after the drain comes through the
     * interrupt. */
    csr_clear(sstatus, MSTATUS_SIE);
    handle_uart();
    csr_set(sie, SIE_SEIE);
    csr_set(sstatus, MSTATUS_SIE);
}

void trapper(struct trapframe *tf)
{
    unsigned long cause = csr_read(scause);

    #if HP_CH >= 19
        if (hyp_trap(tf))
            return;
    #endif
}
```

```

#endif
    if (cause == SCAUSE_ILLEGAL) {
        if (!saw_illegal) {
            saw_illegal = 1;
            uart_puts("S: trap sepc=");
            uart_puthex(csr_read(sepc));
            uart_puts(" scause=");
            uart_puthex(cause);
            uart_puts(" stval=");
            uart_puthex(csr_read(stval));
            uart_puts("\r\n");
        }
        skip_illegal(tf);
        return;
    }
    if (cause == SCAUSE_SSI) {
        csr_clear(sip, 1UL << 1);
        ticks++;
        uart_puts("S: tick ");
        uart_puthex(ticks);
        uart_puts("\r\n");
        if (ticks >= 3) {
            #if HP_CH == 6
                uart_puts("PASS\r\n");
                halt(0);
            #endif
        }
        #if HP_CH >= 8
            /* Ch 8's preemptive run: every tick is a scheduling point.
             * The terminator above is gated to ch 6; this is the same
             * SSIP, one chapter later, driving the switch. */
            preempt_tick();
        #endif
        return;
    }
    if (cause == SCAUSE_SEI) {
        unsigned src = plic_claim_s();
        if (src == 10)
            handle_uart();
        #if HP_CH >= 11
            if (src == blk_irq)
                blk_intr(); /* ch 11: the block device's used-buffer
                 ↪ notification */
        #endif
        if (src)

```

```

        plic_complete_s(src);
        return;
    }
    #if HP_CH >= 7
        if (cause == SCAUSE_ECALL_U) {
            syscall(tf);
            return;
        }
    #endif
    uart_puts("S: unexpected scause=");
    uart_puthex(cause);
    uart_puts("\r\n");
    halt(1);
}

```

The probe is `csrr t0, mhartid` again. Delegated, it is `scause 2` and `stval=0xf14022f3` — same encoding as chapter 3, different CSR (`stval` not `mtval`), different owner. The handler adds 4 to `sepc` and returns. Do not restore a stacked `sepc` on top of that write; the stack holds the faulting PC.

Then the UART. The PLIC is the platform-level interrupt controller: it takes every device's line and routes it to a hart context, gated by a priority and a threshold. QEMU virt wires `ns16550a` source 10 to PLIC context 1 (S-mode hart 0). Context 0 is M and is unused. Priority 1, enable bit 10, threshold 0. Claim reads the source; writing it back completes.

```

static volatile unsigned int *plic_priority(unsigned src)
{
    return (volatile unsigned int *) (PLIC + 4u * src);
}

static volatile unsigned int *plic_enable(unsigned ctx)
{
    return (volatile unsigned int *) (PLIC + 0x2000u + 0x80u * ctx);
}

static volatile unsigned int *plic_threshold(unsigned ctx)
{
    return (volatile unsigned int *) (PLIC + 0x200000u + 0x1000u *
        ↵ ctx);
}

static volatile unsigned int *plic_claim(unsigned ctx)
{
    return plic_threshold(ctx) + 1;
}

```

```

}

void plic_init_uart(void)
{
    *plic_priority(UART_SRC) = 1;
    *plic_enable(S_CTX) |= (1u << UART_SRC);
    *plic_threshold(S_CTX) = 0;
}

unsigned plic_claim_s(void)
{
    return *plic_claim(S_CTX);
}

void plic_complete_s(unsigned src)
{
    *plic_claim(S_CTX) = src;
}

```

IER bit 0 is received-data. Do not write FCR after the FIFO is live: QEMU already stuffed input.txt, and a FIFO enable/reset drops the first byte. After enabling IER the handler drains immediately, with sstatus.SIE cleared for the drain and sie.SIE still clear; the unmask comes after. The clear is not redundant with the order: in this chapter nothing has enabled interrupts yet, but a later driver that unmasks first would otherwise leave this drain running with the line live. Unmask first and the UART trap can land between the drain loop's LSR read and its RBR read — the nested handler pops the byte, the loop then reads a stale RBR with DR already clear, and one character echoes twice. The window is host-timing-sized (QEMU delivers bytes from the main loop, the guest reads on the vCPU thread), which is why it shows up under load and not on a quiet run. A 128-byte guard stops a runaway loop; the #if around it in the listing is where a later line buffer takes over that job. Newline ends the lab.

```

static void handle_uart(void)
{
    int c;
    while ((c = uart_getc()) >= 0) {
#ifdef HP_CH < 10
        if (guard >= 128)
            halt(1);
        guard++;
#endif
        uart_putc((char)c);
        nread++;
    }
}

```

```

#if HP_CH >= 10
    console_rx(c); /* ch 10: the console's line buffer bounds it
    ↵ */
#endif
    if (c == '\n' || c == '\r') {
#if HP_CH == 5
        uart_puts("chars=");
        uart_puthex(nread);
        uart_puts("\r\nPASS\r\n");
        halt(0);
#endif
    }
}

```

PLIC MMIO is not one page. Enable lives at 0x0c002080, claim at 0x0c201004. The identity map from chapter 4 has to cover that range or the first claim is scause 13.

Walkthrough

ch05-delegate, ch05-stvec-entry, ch05-claim, ch05-uart-irq. The capture pins the S-mode illegal line and chars=0x6 from the six-byte input.

Exercises

(a) Clear medeleg bit 2. Where does the probe go?

Answer, p. 51

(b) Skip the PLIC complete. What happens to the next character?

Answer, p. 51

(c) Enable RX before draining. Why does the chapter insist on drain-after-enable?

Answer, p. 51

(d) Map only the PLIC base page. Which access faults?

Answer, p. 51

(e) What is in the trap frame that sret actually consumes?

Answer, p. 51

Lab

```
make lab CH=05
make lab CH=05 IMPL=mime
IMPL=mime stubs uart_irq_on.
```

Further reading

- The RISC-V Instruction Set Manual, Volume II: Privileged Architecture (RISC-V International, v20260120, January 2026) — `medeleg`, `stvec`, `sret`.

Answers

(a) The M shim, chapter 3’s line, `mtval=0xf14022f3`. Delegation is a mask, not a suggestion.

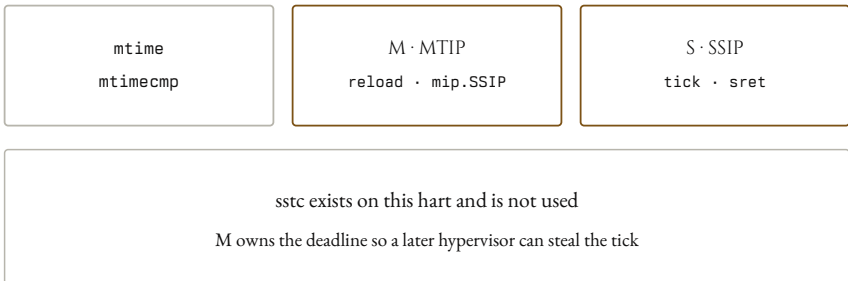
(b) A claimed source stays claimed until complete. It is not pending again, so the hart returns to `wfi` and goes silent — not a livelock. This lab drains the 6-byte file in thread context before `wfi`, so skipping complete is not visible here.

(c) QEMU has already stuffed the FIFO. Enabling IER with data waiting raises SEI; draining first in thread context is the same bytes without depending on the interrupt firing before `wfi`. Do not poke FCR to “enable the FIFO” after that stuffing — the reset drops a byte.

(d) Enable at `0x0c002080` and claim at `0x0c201004` — load page fault, `stval` of that address.

(e) `sepc` and `sstatus`. The integer file is software. `scause` and `stval` are for the handler, not for `sret`.

6. TIME AND THE HART



Why this layer exists

A guest's `rdtime` is a lie the hypervisor tells on a schedule. Before that lie, one hart has to take a tick it did not ask the S-mode timer CSR for. This chapter leaves the deadline in M: `mtime / mtimecmp` in the CLINT, MTIP not delegated. S sees a software interrupt the M shim injected. That is the same shape as a later hypervisor stealing time from a guest.

`sstc` is in the DTB. It is not used. Named so the reader does not discover it in a footnote and think the book missed it. `Sstc` would let S write `stimecmp` and take STIP without M in the path. The path is the lesson.

Mechanism

The CLINT — the core-local interruptor, the block that holds `mtime` and each hart's `mtimecmp` and raises the timer and software interrupts — sits at `0x02000000` on QEMU virt; it puts `mtimecmp` for hart 0 at `0x02004000` and `mtime` at `0x0200bff8`. The timebase is 10 MHz. A 1 Hz tick is `mtime + 10000000`. `mtimecmp` of 0 at reset means MTIP is already pending; reload before enabling `mie.MTIE`.

`mtvec` must point at the M shim before `MTIE` is set. Enabling the timer with

mtvec still 0 jumps to ROM and the hart dies silently. From S, mie.MTIE is enough: M-mode interrupts are taken whenever the current privilege is less than M. mstatus.MIE is the gate only while the hart is in M. mstatus.MPIE is not required for the tick.

The M shim is no longer a dead end. Exceptions still print and halt. MTIP (interrupt, code 7) reloads mtimecmp, sets mip.SSIP, and mrets. mideleg bit 1 lets S take that software interrupt. S clears sip.SSIP, counts, and on the third tick prints PASS.

M arms the deadline before the drop and reloads it on every MTI; the S side only enables the software interrupt it will be handed:

```
void timer_reload(void)
{
    *MTIMECMP = *MTIME + INTERVAL;
}

void timer_enable_m(void)
{
    timer_reload();
    csr_set(mie, 1UL << 7); /* MTIE. From S, mie is the gate; MIE is
↳ not. */
}

int timer_claim_m(unsigned long cause)
{
    unsigned long code = cause & ~MCAUSE_INT;

    if ((cause & MCAUSE_INT) && code == MCAUSE_MTI) {
        timer_reload();
        csr_set(mip, 1UL << 1); /* SSIP → S */
        return 1;
    }
    return 0;
}

void timer_on(void)
{
    csr_set(sie, 1UL << 1); /* SSIE */
    csr_set(sstatus, MSTATUS_SIE);
}
```

The S path is wfi until SSIP. Three lines S: tick 0x1 ... 0x3. No UART input. Chapter 5's PLIC path is still in the image and unused.

```
csr_clear(sip, 1UL << 1);
```

```

        ticks++;
        uart_puts("S: tick ");
        uart_puthex(ticks);
        uart_puts("\r\n");
        if (ticks >= 3) {
# if HP_CH == 6
            uart_puts("PASS\r\n");
            halt(0);
# endif
        }

```

Walkthrough

ch06-reload, ch06-arm, ch06-mtip-shim, ch06-tick. The M shim gained a return path; chapter 3 and 4 still halt on their causes. S does not write mtimecmp.

Exercises

(a) Leave mtimecmp at 0 and enable MTIE. What happens?

Answer, p. 56

(b) Write mideleg bit 7 (MTIP) and read it back. Where does the tick still land?

Answer, p. 56

(c) Why mip.SSIP instead of programming stimecmp?

Answer, p. 56

(d) Three ticks at 1 Hz. How would you prove the interval without a wall clock?

Answer, p. 56

(e) wfi with SIE clear. Does the tick still arrive?

Answer, p. 56

Lab

```

make lab CH=06
make lab CH=06 IMPL=mime

```

The capture pins S: tick 0x1 through 0x3 at 1 Hz, then PASS. IMPL=mime stubs timer_on.

A frame in S that survives a trap, and a tick the kernel did not ask for, are what Part II puts a process under: chapter 7 fabricates that frame to enter a program, and chapter 8 uses this tick to take the hart back from it.

Further reading

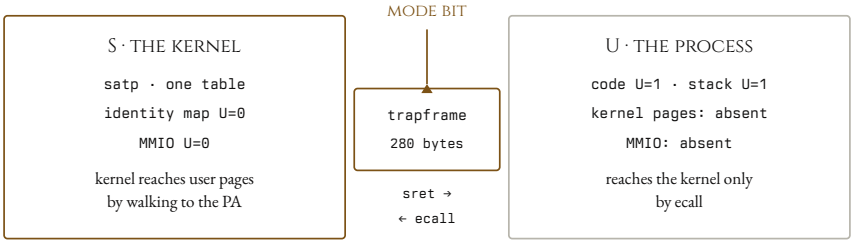
- The RISC-V Instruction Set Manual, Volume II: Privileged Architecture (RISC-V International, v20260120, January 2026) — CLINT, `mtime`, `mip/mie`, `sstc`.

Answers

- (a) MTIP is already high. If `mtvec` is set, the first trap is immediate; the shim writes `mtime+interval`, so after that one early tick the cadence is the normal 1 Hz — not a loop. If `mtvec` is 0, the hart fetches 0 and never prints `S: running`.
- (b) `mideleg` bit 7 is read-only zero. Machine timer interrupts cannot be delegated. The tick still lands in M. The “lose the deadline” story is what a writable bit 7 would have meant; this board does not have that bit.
- (c) Because `sstc` is the S-owned deadline. Injecting SSIP keeps M in the path, which is the shape of chapter 34’s timer exit.
- (d) Read `mtime` at tick 1 and tick 3. The difference is two intervals, 20 million cycles at 10 MHz.
- (e) The interrupt is pending. `wfi` may return when an interrupt is pending even if disabled; taking it still needs SIE. Clear SIE and the count stays 0.

PART II — A KERNEL

7. THE FIRST PROCESS THAT IS NOT THE KERNEL



the U bit in the PTE and the mode bit sret sets — the whole isolation
the same two objects hearth will point at a guest, with a G-stage table (Part IV)

Why this layer exists

Until this chapter, “kernel” was a lie of vocabulary: everything on the hart was the kernel. From here on some code runs *under* the kernel, and the CPU — not good intentions — keeps it there. Isolation is exactly two objects: a page table whose kernel entries lack the U bit, and the privilege mode sret sets on the way down. No third thing exists in this chapter, and none is needed: a program that cannot fetch, load, or store outside two pages is contained, whatever else it believes.

This is the whole trick hearth will repeat on guests with a G-stage table in Part IV — a different table, a different mode bit, the same two objects. If this chapter is solid, the hypervisor’s isolation story is already written; only the names change.

Mechanism

Entering U is a return

Chapter 5 built the trap *into* the kernel; this chapter needs its mirror. sret is the instruction that leaves a trap: it sets the program counter from sepc, the

privilege from `sstatus.SPP`, and interrupts from `sstatus.SPIE` (`SIE←SPIE`, `SPIE←1`). Nothing about `sret` knows a trap happened — it reads registers and jumps. So a kernel that wants to *start* a program fabricates the state a return would consume: a `struct trapframe` with `sepc` = the program's entry, `sp` = the user stack top, and `sstatus` with `SPP=0` — return to U — and `SPIE=1`, so the `sret` lands in U with interrupts enabled. A clear `SPIE` would run U masked, and chapter 8's tick preemption would never fire; the bit is set now, once, on purpose.

The frame's register file is zeroed except `sp`; the program's `_start` builds its world from there. The whole load of this program is one `memcpy` and one page of address space — QEMU's ELF load in chapter 2 was more work than this, and that asymmetry is the point: entering U is *cheaper* than loading a file, because the hardware does the privilege part for free.

```

struct proc *proc_create(const void *blob, unsigned long size,
                        unsigned long va)
{
    struct proc *p = 0;
    unsigned long code_pa = proc_page();
    unsigned long stack_pa = proc_page();
    unsigned long kstack_pa = proc_page();
    unsigned long a;
    int i, slot;

    for (slot = 0; slot < NPROC; slot++)
        if (proc_tab[slot].state == UNUSED || proc_tab[slot].state ==
            ↪ DEAD) {
            p = &proc_tab[slot];
            break;
        }
    if (!p)
        halt(1);

    p->pid = (int)(p - proc_tab) + 1;
    p->state = RUNNABLE;
    p->kstack_top = kstack_pa + PAGE_SIZE;

    /* Code pages at va, U=1. The kernel's identity map entry for the
     * same PA stays U=0: one PA, two VAs, two permissions — the
     * kernel reaches its copy, U reaches only this one. */
    for (a = 0; a < size; a += PAGE_SIZE) {
        unsigned long off = a & (PAGE_SIZE - 1);
        unsigned long n = PAGE_SIZE - off;
    }
}

```



```

    unsigned long *dst = (unsigned long *) (code_pa + a);
    const unsigned char *src = (const unsigned char *) blob + a;
    if (n > size - a)
        n = size - a;
    for (i = 0; i < (int) n; i++)
        ((unsigned char *) dst)[i] = src[i];
    map_page(va + a, code_pa + a,
             PTE_R | PTE_W | PTE_X | PTE_U);
}

/* One U page for the stack, growing down. */
map_page(va + 0x1000UL, stack_pa, PTE_R | PTE_W | PTE_U);

/* The table changed under a live satp. The hart may hold the old
 * translation for these VAs (chapter 8 maps them twice); the
 * fence orders the PTE writes before any use of the new ones. */
__asm__ volatile("sfence.vma" ::: "memory");

/* Fabricate the frame: entering U is the mirror of returning
 * from a trap. SPP=0 so sret drops privilege; SPIE=1 so sret
 * copies it into SIE — U must run with interrupts enabled or
 * chapter 8's preemption never fires. */
for (i = 0; i < 35; i++)
    ((unsigned long *) &p->tf)[i] = 0;
p->tf.sepc = va;
p->tf.sp = va + 0x2000UL;
p->tf.sstatus = MSTATUS_SPIE; /* SPP=0 | SPIE=1 */

/* The heap break starts just above the stack page; chapter 9's
 * sbrk maps pages there on demand. */
p->brk = va + 0x2000UL;

return p;
}

```

The user pages go in at `USER_BASE`, `0x10000`, below every MMIO window the kernel maps — the value `ember/user/link.ld` fixes. Physical pages come from a fixed carve above the page-table pool for now; chapter 9 replaces the carve with the allocator, and the shape — blob to U pages, a kernel stack `U=0`, a fabricated frame — is the object that survives, not the bump pointer. Three things in the listing belong to chapter 8, where they are explained: the `va` argument (one table can hold several processes only if each is mapped at its own base — here it is `USER_BASE`), the slot loop over `proc_tab` (here it finds slot 0), and the `sfence.vma` after the maps (the table changed under a live `satp`; chapter 8 maps the same

VAs a second time). Two belong to chapter 9: `proc_page`, which is the carve in this chapter's image and the page allocator from chapter 9's, and `p->brk`, the heap break `sbrk` will move.

Why the Part I trap entry cannot be entered from U

The frozen `trap_entry` starts with `addi sp, sp, -280` and then stores through `sp`. From S that is fine: `sp` is a kernel page, `U=0`. From U it is a fault machine: the incoming `sp` is a *user* stack address, its page is `U=1`, and the very first `sd` is an S-mode store to a U page without `sstatus.SUM` — cause 15, `stval` on that same user `sp`. The fault is a trap; `stvec` still points at `trap_entry`; the handler runs `addi sp, sp, -280` on the same user `sp` and faults again, cause 15, forever — each “handler” faulting before it can save anything. The fault never reaches a handler you can see; the hart is lost in a loop that no counter watches. Exercise (c') has you predict exactly this.

The fix is one instruction at the front of a *new* entry. `sscratch` is a scratch CSR the spec defines as read/write with no hardware meaning — the conventional parking place for the kernel stack pointer while U runs. `user_vec` opens with `csrrw sp, sscratch, sp`, the atomic swap: `sp` becomes the `proc`'s kernel stack (parked there by `proc_run`), `sscratch` holds the user `sp`, and the 280-byte save lands on kernel memory. One instruction, and the recursive fault is gone.

The entries

`user_vec` saves the same 31 registers and 4 CSRs into the same 280-byte layout as the frozen entry — the frame layout is the frozen object, and duplicating the entry code around it is deliberate: editing `trap.S` would move every Part I capture, and the frozen entry is *correct* for traps from S. What changes is the prologue (the swap), the epilogue (re-park, then through `user_ret`, so a `syscall`'s writes to `tf->sepc` and `tf->a0` are honored), and one `stvec` write in each direction.

```
.globl user_vec
user_vec:
    csrrw    sp, sscratch, sp
    addi     sp, sp, -280
    sd       ra, 0(sp)
    sd       gp, 16(sp)
    sd       tp, 24(sp)
    sd       t0, 32(sp)
    sd       t1, 40(sp)
    sd       t2, 48(sp)
    sd       s0, 56(sp)
```

```

sd      s1, 64(sp)
sd      a0, 72(sp)
sd      a1, 80(sp)
sd      a2, 88(sp)
sd      a3, 96(sp)
sd      a4, 104(sp)
sd      a5, 112(sp)
sd      a6, 120(sp)
sd      a7, 128(sp)
sd      s2, 136(sp)
sd      s3, 144(sp)
sd      s4, 152(sp)
sd      s5, 160(sp)
sd      s6, 168(sp)
sd      s7, 176(sp)
sd      s8, 184(sp)
sd      s9, 192(sp)
sd      s10, 200(sp)
sd      s11, 208(sp)
sd      t3, 216(sp)
sd      t4, 224(sp)
sd      t5, 232(sp)
sd      t6, 240(sp)
csrr    t0, sscratch          /* the user sp */
sd      t0, 8(sp)
csrr    t0, sepc
sd      t0, 248(sp)
csrr    t0, scause
sd      t0, 256(sp)
csrr    t0, stval
sd      t0, 264(sp)
csrr    t0, sstatus
sd      t0, 272(sp)
la      t0, trap_entry
csrw    stvec, t0             /* back to the kernel's entry */
mv      a0, sp
call    trapper

/* trapper returns only to keep running the same proc: the frame
 * at sp is authoritative (syscalls wrote tf->sepc, tf->a0).
 * Re-park the kernel sp and re-enter through user_ret - the
 * mirror of the entry swap. Stopping the proc never comes back.
 */
↵ addi    t0, sp, 280
   csrw    sscratch, t0
   mv      a0, sp

```

```
tail    user_ret
```

`stvec` alternates because it must: `stvec` is writable by S at any time, so pointing it at `user_vec` while U runs and back at `trap_entry` when the kernel runs is legal — and necessary, since each entry is correct for exactly one origin. The write happens on both sides of the crossing, in the entries themselves, so the invariant “`stvec` matches the world that is about to run” is maintained by the same code that changes the world.

```
.globl user_ret
user_ret:
    mv      t6, a0
    ld      t0, 272(t6)          /* sstatus */
    csrwr   sstatus, t0
    ld      t0, 264(t6)
    csrwr   stval, t0
    ld      t0, 256(t6)
    csrwr   scause, t0
    ld      t0, 248(t6)
    csrwr   sepc, t0
    la      t0, user_vec
    csrwr   stvec, t0
    ld      ra, 0(t6)
    ld      gp, 16(t6)
    ld      tp, 24(t6)
    ld      t0, 32(t6)
    ld      t1, 40(t6)
    ld      t2, 48(t6)
    ld      s0, 56(t6)
    ld      s1, 64(t6)
    ld      a0, 72(t6)
    ld      a1, 80(t6)
    ld      a2, 88(t6)
    ld      a3, 96(t6)
    ld      a4, 104(t6)
    ld      a5, 112(t6)
    ld      a6, 120(t6)
    ld      a7, 128(t6)
    ld      s2, 136(t6)
    ld      s3, 144(t6)
    ld      s4, 152(t6)
    ld      s5, 160(t6)
    ld      s6, 168(t6)
    ld      s7, 176(t6)
    ld      s8, 184(t6)
```

```

ld      s9, 192(t6)
ld      s10, 200(t6)
ld      s11, 208(t6)
ld      t3, 216(t6)
ld      t4, 224(t6)
ld      t5, 232(t6)
ld      sp, 8(t6)          /* the user sp, from the frame */
ld      t6, 240(t6)        /* t6's saved value, last */
sret

```

t6 holds the frame base and is restored from the frame last — every register that touches the copy is loaded from the copy before the base is dropped.

```

void proc_run(struct proc *p)
{
    curproc = p;
    p->state = RUNNING;
    csr_write(sscratch, p->kstack_top);
    csr_write(stvec, (unsigned long)user_vec);
    user_ret(&p->tf);
}

```

proc_run parks the kernel stack top in sscratch *before* the crossing — that value is the first instruction of user_vec’s world — and never returns in this chapter: the program’s exit is a halt, and proc_run’s caller treats a return as an error (halt(1)). Chapter 8, which has somewhere to switch to, will change who calls this and why; the parking order does not change.

The U bit, the fault you cannot see around, and SUM

Between kernel and user PTEs exactly one flag changed: U. The consequences run in both directions. A U-mode access to a U=0 page faults — loads cause 13, stores 15, fetches 12. An S-mode *fetch* of a U=1 page also faults (cause 12): a kernel that jumps into user code does not execute it, it crashes — the spec made S+U fetch illegal so a confused kernel cannot be lured into running attacker bytes as itself. An S-mode *load or store* of a U=1 page faults too, unless sstatus.SUM (permit Supervisor User Memory access) is set.

This kernel does not set SUM. It does not need to, because it holds something better than a permission: the page table. walk() — the same frozen walk the hardware uses — turns a user VA into a PTE; the PTE’s PPN is a physical address; and the kernel’s identity map holds that PA at a U=0 VA. The kernel copies through its own mapping and never executes a user instruction. SUM would be simpler to write (dereference the user pointer directly) and costs exactly

the safety the walk buys: with SUM, any kernel bug that leaks a user pointer into a dereference silently reads user memory; without it, that same bug is a page fault, cause 13, in the kernel's own handler, loud. The trade is real — SUM is one instruction and faster — and this kernel still refuses it. Holding the table is what makes the kernel king.

```
int copy_in_user(struct proc *p, unsigned long uva, void *kbuf,
                unsigned long n)
{
    unsigned char *dst = kbuf;
    unsigned long done = 0;

    (void)p; /* one table; the walk is the authority */
    while (done < n) {
        unsigned long va = uva + done;
        unsigned long pa = user_pa(va);
        unsigned long off = va & (PAGE_SIZE - 1);
        unsigned long chunk = PAGE_SIZE - off;
        const unsigned char *src;
        unsigned long i;

        if (!pa || chunk > n - done)
            chunk = n - done;
        if (!pa)
            return -1; /* unmapped or not a U page: the boundary */
        src = (const unsigned char *) (pa | off);
        for (i = 0; i < chunk; i++)
            dst[done + i] = src[i];
        done += chunk;
    }
    return 0;
}
```

Every byte the kernel ever accepts from a process crosses this function or its mirror `copy_out_user` (chapter 10's pipes are its first caller). A VA that is not a mapped U leaf is refused, not followed: the user cannot make the kernel read kernel memory by passing a clever pointer, because the walk only answers for U pages. This one function is the isolation boundary made concrete — the rest of the chapter is plumbing around it.

The ecall path end to end

An `ecall` is the instruction a U program executes to request service: it raises an exception to the privilege that owns the machine — cause 8 from U — with no

arguments, no memory, nothing but the register file as it stands. Chapter 5's `medeleg` bit 8 already routes it to S. The path:

1. U executes `ecall` with the syscall number in `a7`, arguments in `a0–a2`.
2. Hardware raises cause 8; delegation (`medeleg` bit 8) sends it to S; `stvec` is `user_vec`; hardware sets $SPP \leftarrow U$, $SPIE \leftarrow SIE$, $SIE \leftarrow 0$, $sepc \leftarrow$ the `ecall`'s PC.
3. `user_vec`: the `sscratch` swap, the 280-byte save on the `proc`'s kernel stack, `stvec` $\leftarrow$ `trap_entry`.
4. trapper sees cause 8 and calls `syscall(tf)`.
5. The syscall runs with kernel privileges, reads its arguments from the saved frame, and writes results back into the frame — `tf->a0` for the return value, `tf->sepc += 4` to skip the `ecall` itself.
6. `user_vec`'s tail re-parks the kernel `sp` and enters `user_ret`, which re-stores the frame, re-arms `stvec` $\leftarrow$ `user_vec`, and `srets`: $PC \leftarrow sepc$, `privilege` $\leftarrow SPP = U$, `interrupts` $\leftarrow SPIE$.

```
static long sys_exit(long code)
{
    (void)code;
    #if HP_CH < 8
        uart_puts("S: proc 1 exited\r\n");
        uart_puts("PASS\r\n");
        halt(0);
    #else
        /* Ch 8: death is a state, not a halt. The kernel says so and
         * returns to the scheduler through the same door as yield. */
        uart_puts("S: proc ");
        uart_putc((char)('0' + curproc->pid));
        uart_puts(" exited\r\n");
    #if HP_CH >= 10
        fd_close_all(curproc); /* a pipe's other end learns of this */
    #endif
        curproc->state = DEAD;
    #if HP_CH >= 13
        wakeup(curproc); /* a parent in wait sleeps on this address */
    #endif
        sched(); /* never returns: the loop picks a live world */
    #endif
        return 0; /* unreachable; the return silences -Wall */
}

static long sys_write(long fd, long buf, long n)
{

```

```

#if HP_CH >= 10
    /* Ch 10: through the descriptor table. fd 1 is still the console;
     * it is just no longer the only thing a process can write. */
    struct file *f = fd_get(curproc, fd);

    if (!f || n < 0)
        return -1;
    return file_write(f, (unsigned long)buf, n);
#else
    char kbuf[128];
    long i;

    if (fd != 1 || n < 0 || (unsigned long)n > sizeof(kbuf))
        return -1;
    if (copy_in_user(curproc, (unsigned long)buf, kbuf,
        (unsigned long)n) < 0)
        return -1;
    for (i = 0; i < n; i++)
        uart_putc(kbuf[i]);
    return n;
#endif
}

#if HP_CH >= 9
    /* Grow the calling process's heap by npages, U/R/W and never X, and
     * return the old break - the start of the new pages. sbrk(0) reads
     * the break. On an empty freelist the pages already mapped stay the
     * process's and brk already covers them; -1 says the request was not
     * met in full. Shrinking is named, not built. */
    static long sys_sbrk(long npages)
    {
        unsigned long old = curproc->brk;
        long i;

        if (npages < 0)
            return -1;
        for (i = 0; i < npages; i++) {
            unsigned long *slot = walk(curproc->brk);
            unsigned long pa;

            /* One table for everyone: the break can run into the kernel's
             * own low maps (the finisher at 0x1000000 is 238 pages up from
             * the first program's heap) or another process's pages. A
             * slot that is already valid is refused, never overwritten.
            ↵ */

```



```

    if (!slot || (*slot & PTE_V)) {
        uart_puts("S: sbrk: break hit a mapping\r\n");
        return -1;
    }
    pa = kalloc_page();
    if (!pa) {
        uart_puts("S: sbrk: out of pages\r\n");
        return -1;
    }
    map_page(curproc->brk, pa, PTE_R | PTE_W | PTE_U);
    curproc->brk += PAGE_SIZE;
}
__asm__ volatile("sfence.vma" ::: "memory");
return (long)old;
}
#endif

void syscall(struct trapframe *tf)
{
    long n = (long)tf->a7;
    long a0 = (long)tf->a0, a1 = (long)tf->a1, a2 = (long)tf->a2;

    switch (n) {
    case SYS_exit:
        a0 = sys_exit(a0);
        break;
    case SYS_write:
        a0 = sys_write(a0, a1, a2);
        break;
    #if HP_CH >= 8
    case SYS_yield:
        /* Still runnable - the loop decides for how long. */
        curproc->state = RUNNABLE;
        sched();
        a0 = 0;
        break;
    #endif
    #if HP_CH >= 9
    case SYS_sbrk:
        a0 = sys_sbrk(a0);
        break;
    #endif
    #if HP_CH >= 10
    case SYS_read:
        a0 = sys_read(a0, a1, a2);

```

```

        break;
    case SYS_close:
        a0 = sys_close(a0);
        break;
#endif
#if HP_CH >= 12
    case SYS_open:
        a0 = sys_open(a0, a1);
        break;
#endif
#if HP_CH >= 13
    case SYS_spawn:
        a0 = spawn_impl(a0, a1);
        break;
    case SYS_wait:
        a0 = wait_impl(a0);
        break;
#endif
    default:
        a0 = -1;
        break;
}
tf->a0 = (unsigned long)a0;
tf->sepc += 4; /* past the ecall */
}

```

The table has two entries and a refusal in this chapter's image: syscall 1 is `exit` (the `HP_CH < 8` branch: the proc is done, the kernel says so, `PASS`, `halt(0)` — the other branch is chapter 8's, where death becomes a state and a switch), syscall 2 is `write(fd, buf, n)`, which copies the buffer in through `copy_in_user` and prints it on the UART itself. The `SYS_yield` case sits behind `HP_CH >= 8` for the same reason: the file is compiled by every chapter from 7 on, and each chapter's additions are gated to it. The kernel *is* the syscall: the user program never touches the UART's MMIO page, because its page table has no such page — device access must trap or isolation is a story, and exercise (e) derives why from the map. Later chapters extend this table; the numbering never moves.

And the program — the `ecall` wrapper, then `run7`'s two `ecalls`, nothing else. `main` branches on `boot_mode`, a fact chapter 8's kernel places in `a0` at launch and chapter 8's `_start` captures; in this chapter's image the frame's `a0` is zero and `main` runs `run7`:

```

static long syscall13(long n, long a0, long a1, long a2)
{

```

```

register long a0r __asm__("a0") = a0;
register long a1r __asm__("a1") = a1;
register long a2r __asm__("a2") = a2;
register long a7r __asm__("a7") = n;

__asm__ volatile("ecall"
                  : "+r"(a0r)
                  : "r"(a1r), "r"(a2r), "r"(a7r)
                  : "memory");

return a0r;
}

static void run8(void);

static void run7(void)
{
    /* Two ecalls: everything the program does, the kernel does. The
     * U: prefix is the book's convention — this line was printed by
     * the kernel, on the process's behalf, from a U page. */
    syscall3(2, 1, (long)"U: user: hello from U\r\n", 23); /* write
↵ */
    syscall3(1, 0, 0, 0);                                     /* exit
↵ */
}

int main(void);
int main(void)
{
    if (boot_mode == 0) {
        run7(); /* chapter 7's shape: hello, then exit */
        return 0;
    }
    run8(); /* chapter 8's: alternate (1) or spin (2) */
    return 0; /* not reached: run8 exits */
}

```

_start zeroes .bss and sets sp from the link script's stack_top before main — the fabricated frame's sp points at the *top* of the one stack page, and _start's job is to make that page a working stack (it aligns, it does not re-map; the frame's value and the symbol's value are the same number, frozen on both sides of the build).

The U: prefix on the greeting marks which world asked for the line, not which world printed it: the kernel printed it, through the trap, from bytes that live on a U=1 page — the two facts the capture pins in one line.

Walkthrough

ch07-user-prog (the two-ecall program), ch07-proc-create (the fabricated frame and the U pages), ch07-user-vec (the swap and the save), ch07-user-ret (the mirror), ch07-enter (proc_run's parking order), ch07-copy-in (the boundary), ch07-syscall (the table). The capture pins both trap directions: S: kernel up before the first sret, U: user: hello from U printed by the kernel through the write trap, S: proc 1 exited, PASS.

Exercises

(a) Delete PTE_U from the user's code page. Predict the outcome on entry.

Answer, p. 73

(b) Point the fabricated sp at a kernel page (an identity address). Predict.

Answer, p. 73

(c) Have the kernel dereference the user's buffer pointer directly in sys_write — no SUM, no walk. Predict and explain.

Answer, p. 73

(c') Leave stvec at trap_entry and ecall. What does the first sd in the frozen entry do, and why does the fault never reach a handler you can see?

Answer, p. 73

(d) Remove medeleg bit 8 and ecall. Where does it land, and why?

Answer, p. 74

(e) Make the user program store to the UART's MMIO address. Derive what happens and why user device access must trap.

Answer, p. 74

(f) Compute what the mode bit alone — no page tables — would and would not stop.

Answer, p. 74

Lab

```
make lab CH=07
```

```
make lab CH=07 IMPL=mime
```

The capture pins the transcript across both trap directions: S: `kernel up`, then U: `user: hello from U` — printed by the kernel, on the process's behalf, from a U page — then S: `proc 1 exited`, then `PASS`. One process, no clocks, no input: deterministic by construction. `IMPL=mime` copies exactly the objects chapter 7 links and stubs `user_ret` to print `MINE: user_ret stub` and `halt` — the lab boots, prints S: `kernel up`, and fails loudly with no user lines.

Further reading

- The RISC-V Instruction Set Manual, Volume II: Privileged Architecture (RISC-V International, v20260120, January 2026) — `ecall`, `sret`, `sstatus.SPP/SPIE/SUM`, `sscratch`, `stvec`, U-bit fault causes.
- `xv6-riscv` (MIT PDOS) — `kernel/trampoline.S's uservec/userret` as a sibling: same `sscratch` swap, same per-privilege `stvec`, with a trampoline page this kernel does not need because it never switches `satp`.

Answers

(a) 12, instruction page fault. The first fetch at the fabricated `sepc` is now a U-mode fetch of a U=0 page; `stval` is the entry address. The trap is taken (page faults are delegated), `user_vec` runs — the swap works, the save lands on the kernel stack — and `trapper` sees cause 12 and halts. The isolation held in the fetch direction before any instruction executed.

(b) 15, store page fault, on the user's first push. The program may run several instructions — until `_start's` prologue or the first call needs the stack — then a U-mode store to a U=0 page faults with `stval` at the bad `sp`. Isolation holds in the data direction too: a wrong stack is a fault, not a window.

(c) Load page fault, cause 13, `stval` at the user VA. The kernel's `satp` is the same table and the page is U=1: S-mode data access to a U page faults without `SUM`. The fault traps through `trap_entry` (the kernel was running) and reaches `trapper` as cause 13 — visible, loud, debuggable. That visibility is the reason this kernel walks instead of setting `SUM`.

(c') The entry runs `addi sp, sp, -280` on the `user sp`, then the first `sd` is an S-mode store to a U=1 page: cause 15. That fault traps — back to `trap_entry`, still — which pushes another `-280` on the same `user sp` and faults on the same store, identically, forever. Each iteration is a fault whose handler faults before saving anything: no counter increments, no line prints, the hart spins on a cause-15 loop. This is why `stvec` must be switched per privilege, and why the swap

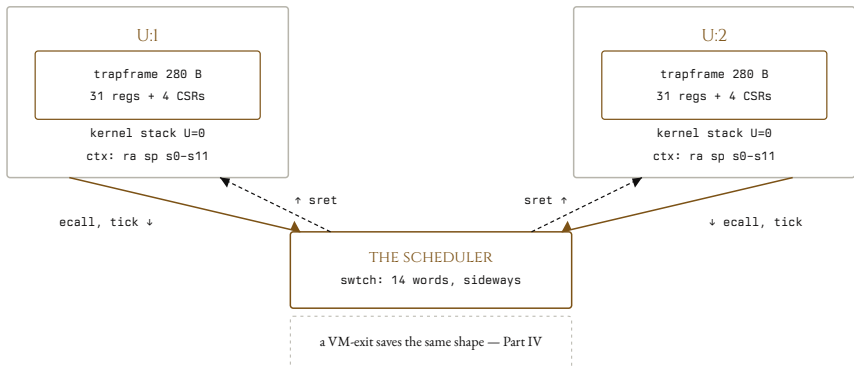
instruction lives at the very top of `user_vec`.

(d) The M shim, cause 8, by trap-not-delegated. From M, `mtval` is 0 and the shim prints and halts — chapter 3’s dead-end path. Delegation is a mask, not a suggestion (chapter 5’s exercise (a) said so for illegal instructions; this is the same rule one bit over): the ecall goes to whichever privilege the mask says, with no regard for who executed it.

(e) Cause 15: the identity map’s UART page is `U=0`, and no user entry for that PA exists anywhere in the table. The user’s only route to the device is a syscall that copies bytes across `copy_in_user` — the kernel speaks for the device, the process asks. If U could reach MMIO directly, any program could program the UART (or later the PLIC, the block device, the finisher at `0x00100000` and halt the machine) — isolation that excludes the devices is not isolation.

(f) Would stop: CSR access (`satp`, `stvec`, `medeleg`... are U-mode illegal), privileged instructions (`sret`, `sfence.vma`), and — if the OS parks a kernel sp in `sscratch` — nothing about memory. Would not stop: every load, store, and fetch anywhere in the address space, because without translation every address is physical and the mode bit does not gate the bus. A U program could read the kernel’s bytes, rewrite its code, and jump to it. The mode bit confines *actions*; the U bit confines *addresses*; this chapter needed both.

8. CONTEXT SWITCH



Why this layer exists

Chapter 7 ends with one process and a kernel that has nowhere else to go: `exit` is a `halt`, because the only alternative to running that process is stopping the machine. A kernel with one process forever is a bootloader with a page table. The object this chapter adds is the *switch* — the kernel leaves one process in the middle of a system call and resumes another in the middle of a different one — and with it two things exist for the first time: more than one process, and the possibility that a process loses the hart without asking. That second thing is preemption, and it is the whole scheduling story of a hypervisor. Chapter 37's `heart` decides which guest believes it has a hart by doing exactly what this chapter's tick handler does — taking the hart away on an interrupt and giving it to someone else — one privilege level up.

The switch is small: fourteen stores, fourteen loads, one `ret`. The reason it is small is the chapter's mechanism, and the reason it must be called from exactly the right place is the chapter's lesson.

Mechanism

Two register files, saved once each

A process in this kernel has two contexts, and the switch touches only one of them.

The first is the user register file — all 31 integer registers, `sepc`, `scause`, `stval`, `sstatus` — and it is already saved, by chapter 7’s `user_vec`, into the 280-byte frame on the `proc`’s kernel stack, the moment the process traps. By the time the kernel decides to switch, the process’s own registers are in memory and will be restored by `user_ret` when the process next runs. Nothing in this chapter saves them again.

The second is the *kernel*’s state on behalf of that process: where in the kernel it was, on which stack. That is what `swtch` saves, and it is much less than a register file. `swtch` is a C function call — `swtch(&curproc->ctx, &sched_ctx)` — and the C ABI already says what a call may destroy. The caller-saved registers (`t0–t6`, `a0–a7`, `ra` from the callee’s point of view) are dead across any call: the compiler has spilled whatever it still needs before the `call`. The callee-saved registers (`s0–s11`, `sp`) must survive; that is what “callee-saved” means. So a function that returns *somewhere else* has to preserve exactly the set a normal return would have preserved — `s0–s11`, `sp` — plus the one thing a normal return takes for granted, `ra`, which is where this world resumes. Fourteen words.

```
struct ctx {
    unsigned long ra, sp;
    unsigned long s[12]; /* s0-s11 */
};
```

The derivation is the ABI, not a design choice: save fewer and the caller’s live values are corrupted on resume; save more and you are re-saving registers the compiler already declared dead. Exercise (a) makes the first case observable.

```
.globl swtch
swtch:
    sd      ra, 0(a0)          /* where this world resumes */
    sd      sp, 8(a0)          /* the stack it resumes on */
    sd      s0, 16(a0)
    sd      s1, 24(a0)
    sd      s2, 32(a0)
    sd      s3, 40(a0)
    sd      s4, 48(a0)
    sd      s5, 56(a0)
    sd      s6, 64(a0)
```



```

sd      s7, 72(a0)
sd      s8, 80(a0)
sd      s9, 88(a0)
sd      s10, 96(a0)
sd      s11, 104(a0)
ld      ra, 0(a1)          /* the other world's resume point */
ld      sp, 8(a1)
ld      s0, 16(a1)
ld      s1, 24(a1)
ld      s2, 32(a1)
ld      s3, 40(a1)
ld      s4, 48(a1)
ld      s5, 56(a1)
ld      s6, 64(a1)
ld      s7, 72(a1)
ld      s8, 80(a1)
ld      s9, 88(a1)
ld      s10, 96(a1)
ld      s11, 104(a1)
ret

```

`swtch` stores the current world into `a0`'s `struct ctx`, loads the other world from `a1`'s, and `rets` — which jumps to the loaded `ra` on the loaded `sp`. It never touches `sstatus`, `stvec`, or `sscratch`: those describe whichever world is about to run, and the code on each side of the switch already set them. The function is a leaf that returns into a different function's frame. Everything above it on the *new* stack was put there by the new world before it last called `swtch`, and everything above it on the *old* stack waits, intact, for the old world's turn.

sp is the world

A process's kernel stack is a page, allocated in `proc_create`, whose top is parked in `sscratch` while the process runs. It holds two things: the 280-byte user frame, pushed by `user_vec` at the moment of the trap, and below it every kernel frame the trap handler grew — `trapper`, `syscall`, and, if the process is about to lose the hart, `sched` and `swtch`'s fourteen words in the `proc`'s `ctx`. Switching `sp` therefore switches all of it at once: the frame that `user_ret` will restore, the C frames that will unwind through `trapper` back into `user_vec`'s tail, and the `ra` that says where. There is no separate “which process is this” state to update on the fast path; `curproc` is set by the scheduler loop for the kernel's bookkeeping, but the hardware-visible identity of the running world is `sp`.

This is why each process needs its *own* kernel stack, and why a single shared kernel

stack — a natural first thought, since only one process runs at a time — is not a simplification but a bug. Exercises (b) and (c) derive the failure in both the cooperative and preemptive cases. The invariant is not “never switch inside a trap handler” — this chapter’s tick handler switches inside a trap handler on every tick, legally — but “each frame lives on the stack of the world it belongs to.”

The loop, and why nobody names a successor

The scheduler is a loop on the kernel’s boot stack with its own `ctx`, and every switch goes through it. A process never switches to another process; it switches to the scheduler, which picks. Two switches per handoff instead of one is the price, and what it buys is that the picking policy lives in one function, `pick`, and the two paths that give up the hart — a `yield` syscall and a timer tick — are identical from `sched` onward. The preemptive run in the lab proves the scheduler is the mechanism precisely because the tick calls the same `sched()` that `yield` calls.

```
/* The whole switch, kernel side: save this proc's context, land in
 * the scheduler's. Nothing else — picking is the loop's job. */
void sched(void)
{
    swtch(&curproc->ctx, &sched_ctx);
}

void scheduler(void)
{
    for (;;) {
        struct proc *p = pick();

        if (!p) {
#ifdef HP_CH == 8
            if (phase == 1) {
                /* the cooperative pair is DEAD: same programs again,
                 * yields removed, the tick driving the switch */
                phase_two();
                continue;
            }
#endif
            return; /* nothing runnable: the caller decides what that
                     ↪ means */
        }
        last = (int)(p - proc_tab);
        p->state = RUNNING;
        curproc = p;
    }
}
```

```

        swtch(&sched_ctx, &p->ctx);
        curproc = 0;
        /* back in the loop: the proc yielded or died */
    }
}

/* A never-run proc's fabricated ra: its own kernel stack, curproc
 * already set, and the chapter 7 crossing ahead of it. */
static void proc_entry(void)
{
    proc_run(curproc); /* returns only if U stopped without exiting
    ↵ */
    halt(1);
}

```

Three things in that listing are the chapter's shape. `sched()` is one line, the switch into the scheduler's context. `scheduler()` marks the pick `RUNNING`, sets `curproc`, switches in, and on return — the process yielded or died — clears `curproc` and goes around. `proc_entry` is the resume point of a process that has never run: `launch` fabricates its `ctx` with `ra = proc_entry` and `sp` at the top of its fresh kernel stack, so the first `swtch` into it “returns” into `proc_entry`, which calls chapter 7's `proc_run` — the parking of `sscratch`, the `stvec` write, `user_ret`. Fabricating a kernel context is the same move as chapter 7's fabricated user frame: a resume point and a stack, nothing else. `s0–s11` start at zero because the ABI lets a function begin with anything in them.

`pick` is round-robin: the first `RUNNABLE` slot after the last runner, in a table of four. The table is the chapter's size — two runs of two processes — and claims no generality. Death is a state now, not a halt: `sys_exit` prints, marks the process `DEAD`, and calls `sched()`, which never returns to it because the loop skips `DEAD` slots.

```

case SYS_yield:
    /* Still runnable – the loop decides for how long. */
    curproc->state = RUNNABLE;
    sched();
    a0 = 0;
    break;

```

`yield` is the cooperative handoff: the process is still `RUNNABLE`, it just calls `sched()`. When the loop picks it again, `swtch` returns into `sched`, `sched` returns into `syscall`, `syscall` writes `tf->a0 = 0` and `tf->sepc += 4`, `trapper` returns into `user_vec`'s tail, and `user_ret` restores the frame — the process resumes at the instruction after its `ecall`, as if the `ecall` had simply taken a while. From its

side, that is exactly what happened.

The tick is a yield the process did not ask for

Chapter 6's tick — mtime compare in M, mtimecmp reloaded by the M shim, SSIP injected into S — is untouched. What changes is what the S-mode handler does with it. In chapter 6 the handler counted; from this chapter it also calls preempt_tick:

```
/* Called from the tick handler on every SSIP — chapter 6's 1 Hz tick,
 * reused untouched. In the cooperative run, or when the kernel
↳ ↳ itself
* is the running context, there is nothing to preempt. */
void preempt_tick(void)
{
    struct proc *next;

    if (phase != 2 || !curproc)
        return;
    #if HP_CH == 8
        if (preempts >= 4) {
            /* four preempted rounds is the proof; the tick after them
             * ends the lab. Chapter 8's terminator, gated like trap.c's.
↳ /*
            uart_puts("PASS\r\n");
            halt(0);
        }
    #endif
    preempts++;
    next = pick(); /* who the loop is about to choose — announced,
↳ then done */
    if (!next)
        return;
    uart_puts("S: preempt ");
    uart_puthex(preempts);
    uart_puts(" to U:");
    uart_putc((char)('0' + next->pid));
    uart_puts("\r\n");
    curproc->state = RUNNABLE; /* still runnable; the tick says next
↳ /*
    sched();
}
```

The interrupted process is still RUNNABLE — it did nothing wrong — and sched() is called from inside the tick handler, on that process's kernel stack, exactly as

`yield` is called from inside the `ecall` handler. The frames are the same shape: `user_vec` pushed the user frame, `trapper` and `preempt_tick` grew below it, `swtch` parked the kernel context. When this process is picked again, it unwinds the same way and `srets` back to the instruction the tick interrupted. The user program, which contains no `yield` and no `exit`, ran on a hart that was taken from it and returned four times; it cannot tell.

Preemption needs interrupts on in U, and the frame chapter 7 fabricated already has `SPIE=1` for this reason. Who owns `sstatus.SIE` across a switch is worth stating, because `swtch` does not save it and the register is global to the hart, not per-context. Hardware clears SIE on every trap into S. Once a process has trapped in, the kernel never sets it again itself except once, in `timer_on` at the start of the preemptive run. So after the first trap, every kernel path runs with interrupts masked — the scheduler loop included, whatever SIE was when it was last switched away from — and a process regains interrupts only through `sret`, which copies SPIE from *its own frame's* `sstatus` into SIE. The user side of SIE lives in the frame; the kernel side is “off after the first trap.” That is a non-preemptible kernel, and it is deliberate: the tick handler can call `sched` without worrying that a second tick lands under it. Exercise (d) follows a tick that arrives while the kernel is in the middle of a syscall.

The M shim gets a stack too

One thing in Part I could not survive preemption, and it is one privilege *up*. The M-mode timer interrupt does not care what privilege the hart is in; from this chapter it can arrive while U is running. Without the `mscratch` swap chapter 3's shim carries, the shim would push its saved registers on the incoming `sp`, which is a kernel stack for every trap before this chapter. A user `sp` is a low virtual address — `0x12000` for the first program — and M-mode runs untranslated: `satp` does not apply, so the shim's first `sd` is a store to *physical* `0x12000`, which on this board is neither DRAM nor a device. The store faults — cause 7, store/AMO access fault, to M — on the same `sp`, and the shim recurses the way chapter 7's exercise (c') recursed one level down.

The fix is the same instruction one level up, through `mscratch` — the M-mode scratch CSR chapter 3 introduced, the twin of `sscratch` — where `drop.c` parks a 4 KiB shim stack's top before the first `mret` to S. `mshim_entry` opens with `csrrw sp, mscratch, sp` and closes by swapping back before `mret`. For a trap from S the shim runs on its own stack instead of the kernel's, and nothing in a transcript can observe which stack it used; the swap exists for the trap from U that this chapter makes possible.

Two programs, one table, one fact in a0

The kernel has one page table, and two processes in it need distinct virtual addresses. prog1 is linked at chapter 7's frozen 0x10000; prog2 is the same program linked at 0x20000 (`--defsym=USER_BASE=0x20000` on its link), and `proc_create` gained a `va` argument to map each where it was linked. Because the table is shared, one process could in principle address the other's pages — both are `U=1` — and this chapter does not prevent it. That is a real hole, and this kernel keeps it: ember runs on one page table through Part II and never switches `satp`. Per-process tables — the thing that makes `U` pages private, at the cost of a `satp` write and a fence on every switch — are xv6's design (chapter 16) and every production kernel's. Here the two programs are cooperating halves of one proof, and the hole is not in the proof's way.

```

struct proc *proc_create(const void *blob, unsigned long size,
                          unsigned long va)
{
    struct proc *p = 0;
    unsigned long code_pa = proc_page();
    unsigned long stack_pa = proc_page();
    unsigned long kstack_pa = proc_page();
    unsigned long a;
    int i, slot;

    for (slot = 0; slot < NPROC; slot++)
        if (proc_tab[slot].state == UNUSED || proc_tab[slot].state ==
            ↪ DEAD) {
            p = &proc_tab[slot];
            break;
        }
    if (!p)
        halt(1);

    p->pid = (int)(p - proc_tab) + 1;
    p->state = RUNNABLE;
    p->kstack_top = kstack_pa + PAGE_SIZE;

    /* Code pages at va, U=1. The kernel's identity map entry for the
     * same PA stays U=0: one PA, two VAs, two permissions — the
     * kernel reaches its copy, U reaches only this one. */
    for (a = 0; a < size; a += PAGE_SIZE) {
        unsigned long off = a & (PAGE_SIZE - 1);
        unsigned long n = PAGE_SIZE - off;
        unsigned long *dst = (unsigned long *) (code_pa + a);

```

```

    const unsigned char *src = (const unsigned char *)blob + a;
    if (n > size - a)
        n = size - a;
    for (i = 0; i < (int)n; i++)
        ((unsigned char *)dst)[i] = src[i];
    map_page(va + a, code_pa + a,
             PTE_R | PTE_W | PTE_X | PTE_U);
}

/* One U page for the stack, growing down. */
map_page(va + 0x1000UL, stack_pa, PTE_R | PTE_W | PTE_U);

/* The table changed under a live satp. The hart may hold the old
 * translation for these VAs (chapter 8 maps them twice); the
 * fence orders the PTE writes before any use of the new ones. */
__asm__ volatile("sfence.vma" ::: "memory");

/* Fabricate the frame: entering U is the mirror of returning
 * from a trap. SPP=0 so sret drops privilege; SPIE=1 so sret
 * copies it into SIE – U must run with interrupts enabled or
 * chapter 8's preemption never fires. */
for (i = 0; i < 35; i++)
    ((unsigned long *)&p->tf)[i] = 0;
p->tf.sepc = va;
p->tf.sp = va + 0x2000UL;
p->tf.sstatus = MSTATUS_SPIE; /* SPP=0 | SPIE=1 */

/* The heap break starts just above the stack page; chapter 9's
 * sbrk maps pages there on demand. */
p->brk = va + 0x2000UL;

return p;
}

```

The `sfence.vma` at the end is new in this chapter and belongs to the mapping, not to the switch. The preemptive run maps `0x10000` and `0x20000` a *second* time, to fresh physical pages, while `satp` is live. A hart may hold the old translation, and the privileged spec requires an `sfence.vma` after a PTE write before the new translation can be relied on. In this lab the old and new pages hold identical bytes — the same blob, freshly copied — so a stale entry would be invisible in the transcript; the fence is there because the spec says so, not because the capture would catch its absence. A kernel that remaps pages to *different* contents without it would run the wrong code with no fault to show for it.

The user programs branch on a fact the kernel hands them in the fabricated

frame's `a0`: 1 for the cooperative run, 2 for the preemptive one. `_start` has to capture that register *before* it zeroes `.bss`, because `.bss` is where the captured value will live:

```
void _start(void)
{
    extern char __bss_start[], __bss_end[], stack_top[];
    char *p;
    register long a0r __asm__("a0");

    __asm__ volatile("" : "+r"(a0r)); /* a0: the spawn-time fact */
    for (p = __bss_start; p < __bss_end; p++)
        *p = 0;
    boot_mode = a0r; /* after the wipe; the register held it alive */
    __asm__ volatile(
        "mv sp, %0\n"
        "call main\n"
        :: "r"(stack_top) : "memory");
    __builtin_unreachable();
}

/* Chapter 8: the same program, spawned with the fact in a0. The
 * cooperative run writes a fixed line and yields; the preemptive run
 * writes the same line and spins — no yield anywhere, the tick is
 * the only switcher. */
static void run8(void)
{
    long i;

    if (boot_mode == 1) {
        for (i = 0; i < 3; i++) {
            syscall3(2, 1, (long)"U:1\r\n", 5); /* write */
            syscall3(3, 0, 0, 0);                /* yield */
        }
    } else { /* boot_mode == 2: one line, then the spin */
        syscall3(2, 1, (long)"U:1\r\n", 5); /* write */
        for (;;)
            ; /* no yield, no exit: the tick is the only way out */
    }
    syscall3(1, 0, 0, 0); /* exit */
}
```

`prog2` is the same file with `U:2` in place of `U:1`. In the cooperative run each writes its line and yields, three times, then exits. In the preemptive run each writes its line once and spins — no yield, no exit, no way out but the tick.


```

mem_init();
trap_init();
uart_puts("S: kernel up\r\n");
sched_boot();

```

`sched_boot` launches both programs with the `fact` set to 1 and enters the loop. When the loop finds nothing runnable and both have died, it launches the same pair with the `fact` set to 2, arms the tick with chapter 6's `timer_on`, and goes around again.

Walkthrough

`ch08-main` (the entry: two launches and the loop), `ch08-swth` (fourteen words, sideways), `ch08-sched` (`sched`, the loop, `proc_entry`), `ch08-yeild` (the cooperative door), `ch08-preempt` (the same door, opened by the tick), `ch08-boot-mode` (the user prologue that reads `a0` before wiping `.bss`), `ch08-prog1-loop` (write-and-yeild, then write-and-spin). The regenerated `ch07-proc-create` shows the `va` argument, the slot loop over the table, and the fence.

Exercises

(a) Save only `ra` and `sp` in `swth` — drop `s0–s11` — and run the cooperative pair. Predict what breaks first, and where.

Answer, p. 87

(b) Give every process the same kernel stack — one page, one `kstack_top` — and run the cooperative pair. Derive the failure.

Answer, p. 87

(c) Same shared stack, but the preemptive run. Which process's frame does the second `user_vec` overwrite, and what is the invariant that a per-process kernel stack actually enforces?

Answer, p. 87

(d) Arrange for the tick to arrive while the kernel is in the middle of a `write` syscall for `prog1`. What state is already saved, by whom, and where does the tick actually land?

Answer, p. 87

(e) Compute the kernel-stack budget: the deepest path this chapter's kernel takes on a process's stack, against the page it has.

Answer, p. 88

(f) `swtch` saves no floating-point registers, and neither does the trap frame. Is that because this hart has no FPU?

Answer, p. 88

Lab

```
make lab CH=08
make lab CH=08 IMPL=mime
```

The transcript has two halves. The cooperative run alternates `U:1` and `U:2` three times, then `S: proc 1 exited`, `S: proc 2 exited`. Then `S: preemptive run, U:1`, and the tick takes over: `S: tick 0xN` is chapter 6's handler still counting, `S: preempt N to U:M` is this chapter announcing the pick, and `U:2` appears once — the second program's single line before it spins. Four preemptions, then the fifth tick prints `PASS`. The tick is 1 Hz, so the run takes about five seconds; the lab's timeout is thirty.

The order is fixed, with one dependency worth naming: the cooperative half must finish inside the first tick interval, so that the tick handler's first call finds the preemptive pair already running. It takes milliseconds under QEMU; the margin is three orders of magnitude, and the line order within each interval is fixed because a program's single `write` takes microseconds against a one-second tick. Nothing in the transcript is a counter value.

`IMPL=mime` copies exactly the objects chapter 8 links and stubs `swtch` to print `MIME: swtch stub and halt` — the lab boots, prints `S: kernel up`, and fails loudly at the first switch with no user lines.

Further reading

- The RISC-V Instruction Set Manual, Volume II: Privileged Architecture (RISC-V International, v20260120, January 2026) — `sstatus.SIE/SPIE` across `sret`, `mscratch/sscratch`, `mstatus.FS`, `sfence.vma` after PTE updates, and the interrupt-taking rule at instruction boundaries.
- `xv6-riscv` (MIT PDOS) — `kernel/swtch.S` is the same fourteen words; `kernel/proc.c`'s `scheduler/sched` pair is the same two-switch loop, with a per-CPU scheduler context this single-hart kernel collapses to one.

Answers

(a) The kernel, not a user program, and immediately after the first `U:1`. The first `yield` switches into the scheduler loop, whose live values — the loop's `p`, the table cursor — the compiler kept in callee-saved registers across the `switch` call, as the ABI allows. They come back holding whatever `prog1`'s kernel path left there. The loop dereferences a garbage `p` and takes a load page fault in `S`: the transcript reads `U:1`, then `S: unexpected scause=0xd`, then the halt. The user register file was never at risk — it lives in the frame — but the kernel's own continuation was, and the ABI is the only thing that says which registers carry it.

(b) `prog1` traps for its first `yield`: `user_vec` pushes its 280-byte frame at the top of the shared page, `trapper`, `syscall`, and `sched` grow below it, and `switch` records `prog1`'s `ctx.sp` somewhere inside that region. The loop picks `prog2`, whose fabricated `ctx.sp` is the same page top; `proc_entry` → `proc_run` → `sret`. `prog2` traps for *its* first `yield`, and `user_vec` pushes `prog2`'s frame at the same page top, over `prog1`'s, with `prog2`'s kernel frames over `prog1`'s. When the loop next resumes `prog1`, its `switch` returns onto memory that now holds `prog2`'s frames of identical shape: it unwinds through `prog2`'s `trapper` frame into `user_vec`'s tail and `user_ret` restores `prog2`'s user frame — `prog2`'s `sepc`, `prog2`'s `sp`. The hart is now running `prog2`'s code while `curproc` says `prog1`, and the transcript prints `U:2` where `U:1` was due. A stack that two worlds share is a frame that two worlds share.

(c) The same overwrite, at the tick instead of the `yield`: the first tick preempts `prog1` mid-spin, its frame sits at the page top, the loop switches to `prog2`, and `prog2`'s first trap — the second tick — pushes over it. The tempting rule, “never call `switch` from a trap handler,” is wrong: `preempt_tick` calls `sched` from inside `trapper` on every tick in the working kernel, and it is correct, because that `trapper` frame is on *the preempted process's own* stack. The invariant is that every frame — user or kernel — lives on the stack of the world it belongs to, so that switching `sp` switches all of them together and none of them is under anyone else's feet. Per-process kernel stacks are how that invariant is spelled.

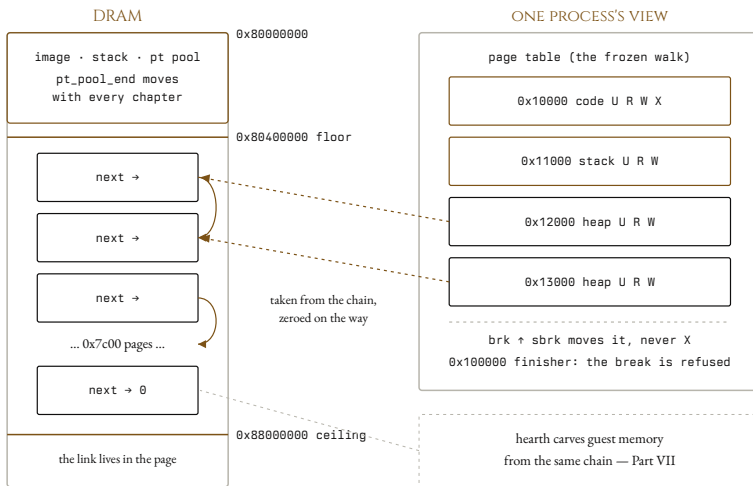
(d) It does not land in the middle of the `syscall`. Hardware cleared `sstatus.SIE` when the `ecall` trapped into `S`, and nothing in the kernel sets it again, so `SSIP` sits pending through `trapper`, `syscall`, and `user_ret`. The `sret` copies `SPIE=1` into `SIE` and the pending interrupt is taken at the very next instruction boundary — in `U`, with `sepc` at the instruction after the `ecall`. By then everything the `syscall` produced is already in the frame (`tf->a0`, the advanced `sepc`), saved by `syscall` itself, and the second trap saves the user register file again through

`user_vec`. Had the kernel run with SIE set, the tick would have entered through `trap_entry` — `stvec` is the kernel's entry while the kernel runs — pushed a frame below the `syscall`'s, and `preempt_tick` would have switched from there, on the same per-process stack, also correctly. The non-preemptible kernel is a simplification the design chooses, not a requirement the mechanism has.

(e) From the image's disassembly: `user_vec`'s frame is 280 bytes; `trapper` allocates 32; `syscall` 176 (its 128-byte `kbuf` for `write` is inlined there); `preempt_tick` 32; `uart_puthex` 32; `sched` allocates nothing and tail-calls `switch`, which uses no stack — the fourteen words go into `struct ctx` inside `struct proc`, not onto the stack. The `ecall` path is therefore $280 + 32 + 176 = 488$ bytes; the tick path $280 + 32 + 32 + 32 = 376$. Against a 4096-byte page that is under 12 %, with nothing nested because the kernel is non-preemptible. The budget is generous by design: what would eat it is not depth but a second frame on the same stack, which (b) and (c) rule out.

(f) No. Read `misa` at boot: `0x800000000001411ad`, and bits 5 and 3 — F and D — are set; QEMU's virt hart has a full double-precision unit. What makes the omission correct is `mstatus.FS`: bits 14:13 read 0, *Off*, at reset, and no code in `ember` ever sets them. With `FS=Off` every floating-point instruction traps as illegal (cause 2), so there is no floating-point state that could be live, and a save would be saving nothing. The kernel and every user program are compiled - `march=rv64ima_zicsr`, so the compiler never emits such an instruction in the first place — that is what guarantees no path needs `FS` on. The choice buys 32 registers plus `fcsr` — 264 bytes — not saved on every trap and every switch, and no dirty-tracking. A kernel that lets processes use the FPU has to turn `FS` on, save that state, and usually save it lazily; the `march` flag in Part I's build is where this kernel declined that work.

9. HEAP AND SBRK



Why this layer exists

Chapter 7 gave a process three pages by bumping a pointer past the page-table pool, and chapter 8 gave two processes six the same way. A pointer that only moves up is not an allocator: nothing ever comes back, and the kernel has no answer to “how much is left.” Every object from here on needs pages that can be handed out and taken back — a pipe’s ring buffer (chapter 10), the block device’s ring and its buffers (chapter 11), the fresh address space `spawn` builds (chapter 13) — and each needs them at run time, in an amount the link script could not have known.

`malloc` is not a law of nature. It is a library over a smaller thing the kernel provides, and the smaller thing is this chapter: physical pages, one at a time, from a region the kernel owns, into a page table the kernel builds with the frozen walk. The shape — a page-granular freelist over a physical region — is the one `hearth` uses to carve guest memory in chapter 34. The object outlives the kernel it was

born in; what changes later is the name of the thing the page is given to.

Mechanism

The page is the unit

The MMU's unit is the page, and the walk maps pages, so a kernel allocator that dealt in bytes would be lying about the granularity underneath it. Everything in this chapter is 4 KiB: the allocator hands out page-aligned physical addresses, sbrk grows a heap by whole pages, and the kernel never splits one. Sub-page allocation is a user-space concern — a malloc in the libc stub, when chapter 13 wants one — layered over sbrk exactly as it is layered over brk on Linux.

The link lives in the page

An allocator needs to know which pages are free. The obvious structure is a table — a bitmap or an array of flags — and it has a problem this early in a kernel: it has to be sized and placed before anything is allocated, which means the link script or a boot-time carve, which means one more number that moves with the image. The freelist avoids the table by using the pages themselves. A free page is, by definition, memory nobody is using; its first eight bytes can hold the physical address of the next free page. The free region describes itself, the head pointer is the whole allocator state, and both operations are a pointer swap:

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

static struct run *freelist;
static unsigned long nfree;

void kfree_page(unsigned long pa)
{
    struct run *r;

    /* Alignment and range are the only checks. A page freed twice is
     * pushed twice and the list becomes a cycle — exercise (a). */
    if ((pa & (PAGE_SIZE - 1)) || pa < KALLOC_FLOOR || pa >=
        KALLOC_CEIL)
        halt(1);
    r = (struct run *)pa;
    r->next = freelist;
    freelist = r;
    nfree++;
}
```

```

}

unsigned long kalloc_page(void)
{
    struct run *r = freelist;
    unsigned long *w = (unsigned long *)r;
    unsigned long i;

    if (!r)
        return 0; /* empty: the caller fails loudly, never wraps */
    freelist = r->next;
    nfree--;
    for (i = 0; i < PAGE_SIZE / sizeof(unsigned long); i++)
        w[i] = 0; /* the link goes too */
    return (unsigned long)r;
}

```

kfree_page pushes; kalloc_page pops. The push writes the link *into* the page, through the kernel's identity map, and that is where the design's one cost lives. A page just popped from the list still holds that link in its first word — a kernel physical address, the next free page's — and a moment later it is a U page. Hand it out unzeroed and every process gets a peek at the kernel's free-page chain: one word per allocation, but a word that says where the next allocation will land. kalloc_page zeroes the page before returning it, link included, and the probe program checks the first and last words are zero as its first act — not because zero-fill is decoration but because the alternative is a leak. QEMU zeroes DRAM at start, so on this board only the link would have been non-zero; on a warm reboot it would be whatever ran last.

The zeroing has a second consequence, less obvious, that exercise (a) turns into a demonstration: a page freed twice is pushed twice and *appears* to make the list a cycle, but the next pop zeroes the page and with it the link that was the cycle — and everything the list held beyond that page is orphaned.

A constant floor

Where does the free region start? The obvious answer is the end of the image — `pt_pool_end`, the last link-script symbol — and it is the wrong answer for this book. That symbol moves with every chapter's code size, and the allocator's bounds appear in the transcript. A capture that pinned `pt_pool_end` would fail any correct mine/ whose code was a different size, before its stub ever ran; which is why chapter 2 alone pins link-time addresses — its subject is the layout — and no later capture does.

```

void kalloc_init(void)
{
    extern char pt_pool_end[];
    unsigned long pa;

    /* The floor is a constant so the transcript never depends on code
     * size; this is the check that keeps it honest. */
    if ((unsigned long)pt_pool_end > KALLOC_FLOOR) {
        uart_puts("S: kalloc: image past the floor\r\n");
        halt(1);
    }
    /* Chapter 4 mapped one page of this region, 0x81000000, U=1 in
     * the identity map — named for a process, never used. The kernel
     * cannot write a freelist link through a U leaf without SUM
     * (cause 15), so the page comes back to the kernel first. */
    map_page(CH4_USER_PAGE, CH4_USER_PAGE, PTE_R | PTE_W | PTE_X);
    __asm__ volatile("sfence.vma" ::: "memory");
    for (pa = KALLOC_FLOOR; pa < KALLOC_CEIL; pa += PAGE_SIZE)
        kfree_page(pa);
    uart_puts("S: kalloc ");
    uart_puthex(KALLOC_FLOOR);
    uart_puts("..");
    uart_puthex(KALLOC_CEIL);
    uart_puts(" pages=");
    uart_puthex(nfree);
    uart_puts("\r\n");
}

```

So the floor is a constant, 0x80400000 — 4 MiB into DRAM — and the ceiling is -m 128M's end, 0x88000000. The image is nowhere near 4 MiB, but “nowhere near” is not a proof; the boot assertion is. If a later chapter ever grows the image past the floor, the kernel says so and stops, and the constant moves by a DECISIONS.md line, not by silent drift. The region is 31744 pages, 0x7c00, and the transcript says so in a line no reader's code size can change.

One page in that region is special, and the allocator has to know it. Chapter 4 mapped 0x81000000 into the identity map with U=1 — a user page, named for a process that chapter 7 never used, because chapter 7 carved fresh pages instead. It is inside the free region, and kfree_page's link store through the identity map is an S-mode store to a U page: cause 15, the SUM rule from chapter 7, this time biting the kernel's own bookkeeping. The kernel could not write its free-page chain through a leaf it had given away. kalloc_init takes the page back — remaps it U=0, fences — before weaving the list. That the same rule that protects the kernel from a user pointer also stops the kernel from writing

through a U leaf it forgot about is the U bit doing exactly its job.

The break

A process's heap is one contiguous range of virtual addresses that grows upward from a fixed point. The point is the *break*: the first address above the process's stack page, `va + 0x2000 — 0x12000` for the first program — set in `proc_create` and moved only by `sbrk`. `sbrk(n)` maps `n` fresh pages at the break, moves it up, and returns where it was: the start of the new pages. `sbrk(0)` reads it.

```
/* Grow the calling process's heap by npages, U|R|W and never X, and
 * return the old break — the start of the new pages. sbrk(0) reads
 * the break. On an empty freelist the pages already mapped stay the
 * process's and brk already covers them; -1 says the request was not
 * met in full. Shrinking is named, not built. */
static long sys_sbrk(long npages)
{
    unsigned long old = curproc->brk;
    long i;

    if (npages < 0)
        return -1;
    for (i = 0; i < npages; i++) {
        unsigned long *slot = walk(curproc->brk);
        unsigned long pa;

        /* One table for everyone: the break can run into the kernel's
         * own low maps (the finisher at 0x100000 is 238 pages up from
         * the first program's heap) or another process's pages. A
         * slot that is already valid is refused, never overwritten.
        ↵ */
        if (!slot || (*slot & PTE_V)) {
            uart_puts("S: sbrk: break hit a mapping\r\n");
            return -1;
        }
        pa = kalloc_page();
        if (!pa) {
            uart_puts("S: sbrk: out of pages\r\n");
            return -1;
        }
        map_page(curproc->brk, pa, PTE_R | PTE_W | PTE_U);
        curproc->brk += PAGE_SIZE;
    }
    __asm__ volatile("sfence.vma" ::: "memory");
    return (long)old;
}
```

```
}
```

Three decisions in that function are the chapter's. The pages are `U|R|W` and never `X`: heap memory is data, and a process that could execute what it wrote into its heap could execute what a pipe or a file put there too. Nothing enforces that distinction yet — exercise (e) shows the transcript does not change if you add `X` — which is exactly why the flag is set correctly now, before chapter 13 loads programs from a disk and the distinction starts to matter. The second: on an empty freelist the pages already mapped stay the process's, the break already covers them, and `-1` says the request was not met in full — a partial `sbrk` is still an `sbrk`. Shrinking the heap is named and not built; nothing in Part II frees a heap page before the process dies.

The third decision is the guard, and it exists because `ember` has one page table. Chapters 7 and 8 said the `U` pages are private to a process only by convention, since every process and the kernel share the same tree. The break makes the sharing concrete: `0x12000` upward is the first program's heap, and 238 pages up from there is `0x100000`, the finisher's identity-mapped MMIO page — the kernel's `halt`. Without the guard, `sbrk` would reach it, and `map_page` would overwrite the finisher's PTE with a heap page; the next `halt()` is then an S-mode store to a `U` page, cause 15, in a handler that halts, forever. With it, a slot that is already valid is refused: `S: sbrk: break hit a mapping`, and the caller gets `-1`. Exercise (b) runs both. In chapter 8's layout, where a second program sits 64 KiB above the first, the first heap has fourteen pages before it meets the second program's code — a limit set by where the programs were linked, not by how much DRAM there is. Per-process page tables would dissolve that limit, and that is the design every production kernel uses; this one keeps a single table through Part II and says so wherever it costs something.

Who owns a page

A page is owned by whoever mapped it: the process, once `sbrk` puts it in the table. There is no reference count, because in this kernel a page has exactly one owner and is never shared — chapter 10's pipe page is the kernel's, copied through, and never mapped into a process. The count would be bookkeeping for a sharing that does not exist. What does exist is a leak: a process that exits leaves its pages mapped and its slot `DEAD`; chapter 8's `pick` skips the slot, nothing frees the pages, and the next `proc_create` reuses the slot with fresh pages. Chapter 13's `wait` is where a parent reclaims a child's pages; until then the leak is named, small, and deliberate.

The page-table budget

Each new page the heap maps costs a leaf in a level-0 table, and each new 2 MiB region the heap enters costs a fresh level-0 table from `pt.c`'s pool — 512 KiB, 128 tables, fixed in `link.ld`. The walk is frozen and its pool cursor is private, so the kernel cannot ask `pt.c` how many it has used; it counts instead, walking from `satp`'s root: the root, every valid pointer entry in it, every valid pointer entry in those. That is a pure function of the maps, not of code size, and the transcript prints it twice — after the process is created and after it has exited — as `0x4a` of `0x80`. Part I's identity map and MMIO windows took 74 tables; the first program's pages landed in a level-0 table the finisher's mapping had already created; the four heap pages are in the same 2 MiB region as the code, so `sbrk` cost no table at all. The number to remember is the one that is left: 54. Heaps that cross 2 MiB boundaries and more processes at more bases spend from it, and a chapter that prints a higher number is spending on purpose.

Where the process starts, and a build rule

The kernel enters a user blob at its first byte — `proc_create` fabricates `sepc = va`, and `va` is where the blob's byte 0 was copied. The blob is the program's `.text` followed by `.rodata`, so byte 0 is whatever the compiler emitted first in `.text`. Nothing guarantees that is `_start`: at -O2 GCC is free to emit a static helper ahead of the function that calls it, and when it does the blob's first bytes are, say, a `puts`, and the process's first instruction is a load through `a0`, which the fabricated frame set to zero — cause 13 at `stval 0, sepc 0x10000`. So `_start` is placed in a `.text.start` section by attribute, in every program, and the link script places that section first. The entry belongs at the front by construction, not by emission order.

One build rule goes with it: the user binaries depend on their sources in the kernel's `Makefile`, so an edited program reaches the image on the next build. A transcript that came from a stale blob is not a transcript.

The probe itself:

```
int main(void);
int main(void)
{
    unsigned long old = (unsigned long)syscall13(4, HEAP_PAGES, 0, 0);
    unsigned long now = (unsigned long)syscall13(4, 0, 0, 0);
    unsigned long *w = (unsigned long *)old;
    unsigned long sum = 0, i;
```

```

puts("U: brk ");
puthex(old);
puts(" -> ");
puthex(now);
puts("\r\n");

/* A free page's first word held the kernel's freelist link; the
 * last word is as far from it as the page goes. Both must be 0.
↪ */
puts(w[0] == 0 && w[WORDS - 1] == 0 ? "U: fresh page zero\r\n"
      : "U: fresh page dirty\r\n");

for (i = 0; i < WORDS; i++)
    w[i] = (i + 1) * MULT;
for (i = 0; i < WORDS; i++)
    sum += w[i];
puts("U: sum=");
puthex(sum);
puts("\r\n");

syscall3(1, 0, 0, 0); /* exit */
return 0;
}

```

Four pages, a zero check at both ends of the new range, a pattern the reader can recompute — the sum over $i = 0 \dots 2047$ of $(i+1) \cdot 0x9E3779B97F4A7C15 \bmod 2^{64}$ — read back and printed. Every number in the transcript is a constant. The kernel's side of the run is chapter 8's launcher and loop, entered from a boot stage that prints the budget on either side:

```

mem_init();
trap_init();
uart_puts("S: kernel up\r\n");
kalloc_init();
launch(user3_blob_start,
      (unsigned long)(user3_blob_end - user3_blob_start),
      USER_BASE, 0);
pt_report();
scheduler(); /* returns when the table is empty */
pt_report();
uart_puts("PASS\r\n");
halt(0);

```

`scheduler()` returns when its table is empty — the one behavioral change to chapter 8's loop — and the caller decides what that means; here it means the

probe exited and the lab passed. The scheduler is cooperative from this chapter on: nothing arms the tick, `preempt_tick` stays chapter 8's, and a process runs until it yields, blocks, or exits. That is a choice, made here so chapters 10 and 13 inherit it knowingly.

Walkthrough

`ch09-main` (the boot stage: allocator, one launch, the loop, the budget on either side), `ch09-kalloc-init` (the assertion, the reclaimed page, the weave), `ch09-freelist` (push, pop, zero), `ch09-sbrk` (the guard, the break, the fence), `ch09-heap-probe` (four pages, two zero checks, one checksum). The regenerated `ch07-proc-create` shows `proc_page` — the carve for chapters 7 and 8, `kalloc_page` from here — and the break's initial value.

Exercises

(a) Free a page twice, then allocate three times. Derive what the list does, then run it.

Answer, p. 98

(b) Ask for 40000 pages. Predict what stops you, then run it with the guard in `sys_sbrk` removed, and again with it present.

Answer, p. 99

(c) How many processes can this kernel hold, at three pages each plus a four-page heap?

Answer, p. 99

(d) Why is there no reference count on a page, and what would it be for?

Answer, p. 99

(e) Map heap pages `U|R|W|X` instead of `U|R|W`. What breaks?

Answer, p. 99

(f) What does `hearth` change about this allocator, and what does it keep?

Answer, p. 100

Lab

```
make lab CH=09
make lab CH=09 IMPL=mime
```

The transcript, after chapter 8's boot lines: S: `kalloc 0x80400000..0x88000000 pages=0x7c00`, the constant bounds and the count; S: `pt tables 0x4a/0x80` with the probe created; U: `brk 0x12000 -> 0x16000`, the old break and the new; U: `fresh page zero`, the first and last words of the new range; U: `sum=0x1516cf4cac905400`, the checksum; S: `proc 1 exited`; the budget again, unchanged, because four pages in one 2 MiB region cost no table; PASS. No clocks, no addresses that depend on code size — the floor and ceiling are constants and the table count is a function of the maps.

IMPL=mime copies exactly the objects chapter 9 links and stubs `kalloc_page` to print MINE: `kalloc_page stub` and halt. The lab boots, prints the allocator's bounds — `kalloc_init` still weaves the list, since `kfree_page` is not the stub — and fails loudly at the first `proc_page` with no table count and no user lines.

Further reading

- The RISC-V Instruction Set Manual, Volume II: Privileged Architecture (RISC-V International, v20260120, January 2026) — the `U` bit and `sstatus.SUM` (the rule that bit the allocator's own link store), `sfence.vma` after PTE writes, the Sv39 page-table geometry behind the budget.
- xv6-riscv (MIT PDOS) — `kernel/kalloc.c` is the same freelist with the link in the page; xv6 fills freed pages with junk to catch use-after-free and zeroes on `kalloc` for the same reason this chapter does. Its `sbrk` grows and shrinks; this one only grows, and says so.

Answers

(a) Pushed twice, the page's link points at itself: `x->next == x`, a one-element cycle with the rest of the list hanging off the old head that is now unreachable behind it. The first pop returns `x` and sets the head to `x->next`, which is `x` again — and then zeroes the page, link included. The second pop returns `x` once more and sets the head to `x->next`, which is now 0. The third returns 0: the list is empty. Observed: `0x87fff000 0x87fff000 0x0`, followed by S: `proc_create: out of pages`. The double free did not loop forever; it truncated the list at the duplicate and orphaned every page beyond it — the 31743 that were behind the first push. A freelist with the link in the page has no way to notice a

double free without a scan; a bitmap would have refused the second free in one test. That is the trade the design made, and the exercise is where it shows.

(b) Not the allocator: 40000 is more than the 31741 pages the list holds, but the heap never gets that far. Without the guard, `map_page` walks up from `0x12000` and after 238 pages reaches `0x100000` — the finisher, identity-mapped by chapter 4 — and overwrites its PTE with a U heap page. The next kernel `halt()` is an S-mode store to a U page, cause 15, whose handler calls `halt()`: the transcript repeats S: unexpected `scause=0xf` until the timeout. With the guard, the 239th page is refused — S: `sbrk: break hit a mapping` — `sbrk` returns `-1` with the break at `0x100000` and 238 pages mapped, and the probe, which trusts the return value, then dereferences `-1` and faults in U with cause 13. Two lessons: in a one-table kernel the heap’s ceiling is the next mapping, not the amount of DRAM; and a refused `sbrk` is a return value the caller has to check.

(c) Three answers, and the smallest wins. By physical memory: 31744 pages at 7 each is 4534 processes — the allocator is not the limit. By virtual layout: chapter 8 spaces programs 64 KiB apart from `0x10000`, and the first kernel mapping above them is the finisher at `0x100000`, so 15 slots fit, each with at most 14 heap pages before its break meets the next program’s code. By the table: `NPROC` is 4. The kernel holds four processes because a fixed-size array says so, could hold fifteen if the array grew, and could hold thousands only with per-process page tables, which it does not have. Each limit is a different object; knowing which one binds is the whole exercise.

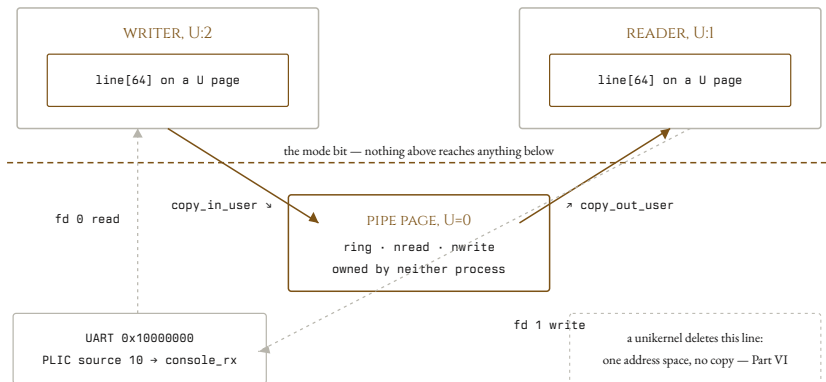
(d) A count tracks sharing, and nothing in this kernel shares a page: every page has one owner — the kernel, or the process whose table maps it — and the owner is the only party that could free it. Chapter 10’s pipe is the counterexample that is not: the pipe page belongs to the kernel and the data is copied through it, so neither process ever maps it. The count would matter the moment two tables mapped one page (a shared library, a fork that copies on write, a zero page mapped everywhere), because then “who frees it” has no single answer. What this kernel has instead of a count is a leak: an exited process’s pages stay mapped in a DEAD slot until chapter 13’s `wait` reclaims them.

(e) Nothing, observably: the transcript is identical, `PASS` and all. The probe never jumps into its heap, and no other code in this chapter can put bytes there. That is the danger, not the reassurance. From chapter 10 a pipe can write bytes into a process’s heap on behalf of another process; from chapter 12 a file can; from chapter 13 a process is *loaded* from a file. A heap that is executable turns every one of those data paths into a code path, and nothing would fail until something

exploited it. The flag is set correctly in the chapter where setting it costs nothing, so the chapters where it costs something inherit it.

(f) It keeps the object: a freelist of physical pages over a region the hypervisor owns, the link in the page, zero on hand-out. What changes is the boundary the page crosses. Here a page goes into a process's table with U set and the process addresses it through $satp$. In chapter 34 a page goes into a *G-stage* table — the second walk chapter 19 builds — and a guest addresses it through its own $satp$ and then through the hypervisor's, and never learns the physical address at all. The zeroing rule gets stricter, not looser: a guest that saw a stale link would be reading the hypervisor's free-page map from inside a VM. And the ceiling becomes a policy: how much of the machine each guest is allowed to take, which is chapter 37's scheduling question in a different unit.

10. PIPES AND A CONSOLE



Why this layer exists

Two processes exist since chapter 8, and they cannot say anything to each other. They share a page table, which is a hole, not a channel: either could scribble on the other's pages if it guessed an address, and nothing would tell the victim. What a kernel owes its processes is the opposite of that — a way to pass bytes that shares *nothing*: no page, no pointer, no lock, no address either side could name. This chapter builds one. A pipe is a kernel page neither process maps, and the two copies that move bytes in and out of it are the isolation boundary made into a data path.

A console is the same thing with the machine on one end. Until now ember talked; from here it listens, and a program that reads a line the reader typed is the first program on this hart that waits for the outside world. Waiting is the second object this chapter adds, and it is a scheduler state, not a loop.

The contrast is worth naming once. A unikernel (Part VI) deletes this boundary on purpose: one address space, the “pipe” is a function call, the copy is gone,

and with it every property the copy enforced. That deletion is a choice with a price, and this chapter is where the price is stated so that Part VI can be read as choosing.

Mechanism

A small integer

A process names what it can read or write by a file descriptor: an index into its own table of pointers into one kernel-wide table of open files. The kernel table has sixteen slots and each process eight; the object in a slot has a kind, a reference count, and permission bits. Two kinds exist this chapter — the console and a pipe end — and chapter 12 adds a file on disk as the third behind the same three syscalls.

```

struct file *file_alloc(void)
{
    int i;

    for (i = 0; i < NFILE; i++)
        if (ftab[i].ref == 0) {
            ftab[i].ref = 1;
            ftab[i].type = FD_NONE;
            ftab[i].readable = ftab[i].writable = 0;
            ftab[i].pipe = 0;
            return &ftab[i];
        }
    return 0;
}

void file_close(struct file *f)
{
    if (--f->ref > 0)
        return; /* another descriptor still names it */
    if (f->type == FD_PIPE)
        pipe_close(f->pipe, f->writable);
    #if HP_CH >= 12
        if (f->type == FD_FILE)
            fs_close(f->ino); /* the cache reaches the disk here */
    #endif
    f->type = FD_NONE;
}

long file_read(struct file *f, unsigned long uva, long n)

```

```

{
    if (!f->readable)
        return -1;
    switch (f->type) {
    case FD_CONSOLE:
        return console_read(uva, n);
    case FD_PIPE:
        return pipe_read(f->pipe, uva, n);
#if HP_CH >= 12
    case FD_FILE:
        return fs_fread(f, uva, n);
#endif
    default:
        return -1;
    }
}

long file_write(struct file *f, unsigned long uva, long n)
{
    if (!f->writable)
        return -1;
    switch (f->type) {
    case FD_CONSOLE:
        return console_write(uva, n);
    case FD_PIPE:
        return pipe_write(f->pipe, uva, n);
#if HP_CH >= 12
    case FD_FILE:
        return fs_fwrite(f, uva, n);
#endif
    default:
        return -1;
    }
}

/* The lowest free slot: 0, 1, 2 for the console, 3 for the first pipe
 * end - the POSIX shape, by construction rather than by rule. */
int fd_install(struct proc *p, struct file *f)
{
    int fd;

    for (fd = 0; fd < NOFILE; fd++)
        if (!p->ofile[fd]) {
            p->ofile[fd] = f;
            return fd;
        }
}

```

```

    }
    return -1;
}

struct file *fd_get(struct proc *p, long fd)
{
    if (fd < 0 || fd >= NOFILE)
        return 0;
    return p->ofile[fd];
}

/* On exit: every open end is closed, which is how a reader learns its
 * writer is gone. */
void fd_close_all(struct proc *p)
{
    int fd;

    for (fd = 0; fd < NOFILE; fd++)
        if (p->ofile[fd]) {
            file_close(p->ofile[fd]);
            p->ofile[fd] = 0;
        }
}

```

Nothing in that file touches user memory; it routes. `read(fd, ...)` and `write(fd, ...)` find the file and dispatch on its kind, `close(fd)` drops a reference and, at zero, tells the underlying object. The lowest free slot is handed out first, so the console lands at 0, 1, 2 and a pipe end at 3 — the POSIX layout by construction, not by rule. On exit every descriptor is closed, which is not tidiness: it is how a reader learns its writer is gone.

```

/* read is the first syscall that can wait: a pipe with nothing in it
 * or a console with no line yet puts the caller to sleep inside
 * file_read, and the return happens on a later switch. */
static long sys_read(long fd, long buf, long n)
{
    struct file *f = fd_get(curproc, fd);

    if (!f || n < 0)
        return -1;
    return file_read(f, (unsigned long)buf, n);
}

static long sys_close(long fd)
{

```

```

struct file *f = fd_get(curproc, fd);

if (!f)
    return -1;
file_close(f);
curproc->ofile[fd] = 0;
return 0;
}

```

write existed in chapter 7 as “fd 1 is the UART.” It now goes through the table; fd 1 is still the console, it is just no longer the only thing a process can write to.

The pipe

A pipe is one page from chapter 9’s allocator, used as a ring, with two cursors that only ever increase and a count of open ends on each side. The cursors are taken modulo the page size when they index the ring; their difference is the number of bytes waiting, and the ring is full when that difference is the page. Neither process maps the page. The writer’s bytes cross into the kernel through `copy_in_user` — the walk-to-PA copy chapter 7 built — and the reader’s leave through `copy_out_user`, its mirror, called here for the first time:

```

int copy_in_user(struct proc *p, unsigned long uva, void *kbuf,
                unsigned long n)
{
    unsigned char *dst = kbuf;
    unsigned long done = 0;

    (void)p; /* one table; the walk is the authority */
    while (done < n) {
        unsigned long va = uva + done;
        unsigned long pa = user_pa(va);
        unsigned long off = va & (PAGE_SIZE - 1);
        unsigned long chunk = PAGE_SIZE - off;
        const unsigned char *src;
        unsigned long i;

        if (!pa || chunk > n - done)
            chunk = n - done;
        if (!pa)
            return -1; /* unmapped or not a U page: the boundary */
        src = (const unsigned char *) (pa | off);
        for (i = 0; i < chunk; i++)
            dst[done + i] = src[i];
        done += chunk;
    }
}

```

```

    }
    return 0;
}

long pipe_write(struct pipe *pi, unsigned long uva, long n)
{
    char kbuf[128];
    long done = 0;

    while (done < n) {
        long chunk = n - done, i;

        /* Full: wake any reader, then wait for room. A pipe with no
         * reader left can never drain - refuse rather than sleep. */
        while (pi->nwrite == pi->nread + PIPE_SIZE) {
            if (pi->readers == 0)
                return -1;
            wakeup(&pi->nread);
            sleep_on(&pi->nwrite, "pipe (full)");
        }
        if (chunk > (long)sizeof(kbuf))
            chunk = sizeof(kbuf);
        if (chunk > (long)(pi->nread + PIPE_SIZE - pi->nwrite))
            chunk = (long)(pi->nread + PIPE_SIZE - pi->nwrite);
        if (copy_in_user(curproc, uva + (unsigned long)done, kbuf,
            (unsigned long)chunk) < 0)
            return -1;
        for (i = 0; i < chunk; i++)
            pi->data[pi->nwrite++ % PIPE_SIZE] = kbuf[i];
        done += chunk;
    }
    wakeup(&pi->nread);
    return done;
}

long pipe_read(struct pipe *pi, unsigned long uva, long n)
{
    char kbuf[128];
    long chunk, i;

    /* Empty with a writer still open: block. Empty with none: EOF.
     * */
    while (pi->nread == pi->nwrite && pi->writers > 0)
        sleep_on(&pi->nread, "pipe");
    chunk = (long)(pi->nwrite - pi->nread);
    if (chunk == 0)

```

```

        return 0;
    if (chunk > n)
        chunk = n;
    if (chunk > (long)sizeof(kbuf))
        chunk = sizeof(kbuf);
    for (i = 0; i < chunk; i++)
        kbuf[i] = pi->data[pi->nread++ % PIPE_SIZE];
    if (copy_out_user(curproc, uva, kbuf, (unsigned long)chunk) < 0)
        return -1;
    wakeup(&pi->nwrite); /* room for a writer blocked on full */
    return chunk;
}

```

The two copies are the design. Setting `sstatus.SUM` and letting the kernel dereference user pointers would save a walk per chunk, and it would cost exactly what chapter 7 said it costs: a wrong pointer becomes a silent read instead of a fault. This kernel keeps the walk, and the pipe inherits the refusal — a user buffer that is not a mapped U page is a -1, never a kernel read.

Three facts about the ring follow from the cursors. A read returns whatever is waiting, up to the caller's count: a pipe has no message boundaries, and exercise (a) shows a read that returns the writer's line and the start of its next write in one call. A write larger than the ring's free space fills what it can and blocks for the rest — the writer sleeps until a reader makes room, and wakes any reader before it sleeps so the two cannot wait for each other. And EOF is a fact about the *ends*: a read on an empty pipe blocks while a write end is open and returns 0 once none is, which is why `exit` closing every descriptor matters.

Blocking is a state

A reader with nothing to read does not spin. It records what it is waiting for, marks itself `WAITING`, and switches to the scheduler — chapter 8's `sched()`, from inside a syscall, on its own kernel stack, exactly as `yield` did. The loop skips `WAITING` slots. Whoever makes the awaited thing true calls `wakeup` on the same channel, and the sleeper is `RUNNABLE` again; when the loop next picks it, `sleep_on` returns into the loop that called it, which re-checks its condition and either proceeds or sleeps again.

```

void sleep_on(void *chan, const char *what)
{
    curproc->chan = chan;
    curproc->state = WAITING;
    uart_puts("S: proc ");
    uart_putc((char){ '0' + curproc->pid });
}

```

```

    uart_puts(" blocks on ");
    uart_puts(what);
    uart_puts("\r\n");
    sched(); /* returns when the loop picks this proc again */
    curproc->chan = 0;
}

void wakeup(void *chan)
{
    int i;

    for (i = 0; i < NPROC; i++) {
        struct proc *p = &proc_tab[i];

        if (p->state == WAITING && p->chan == chan) {
            p->state = RUNNABLE;
            uart_puts("S: wake proc ");
            uart_putc((char)('0' + p->pid));
            uart_puts("\r\n");
        }
    }
}

```

The channel is just an address both sides agree on — a pipe’s read cursor for readers, its write cursor for writers, the console’s line count for a console reader — and wakeup wakes everyone on it. That is why every sleep sits inside a `while`, never an `if`: a wakeup means “look again,” not “it is true,” and a second reader may have taken the bytes first. The kernel says both events aloud, `S: proc N blocks on ...` and `S: wake proc N`, so the lab’s transcript pins that the block happened and in what order the world moved.

This scheduler is cooperative — chapter 9 said so — and the lab is honest about what that means. A process that blocks gives up the hart; a process that is runnable keeps it until it blocks, yields, or exits. The writer, once woken, runs to its `exit` before the reader gets a turn, and the transcript’s order is fixed by that. Add chapter 8’s tick and the order would depend on where the tick landed; the mechanism would be the same, the transcript would not.

No fork, so the kernel wires it

POSIX gives a pipe to a process with `pipe(2)` and to a second process with `fork`: the child inherits the descriptors. `ember` has no `fork` (chapter 13 says why, and what `spawn` does instead), so a pipe syscall would hand both ends to one process with no way to give either away. The kernel does the wiring: the boot

stage creates the pipe and installs the read end in the reader's table and the write end in the writer's. Chapter 13's `spawn` attaches only the console to a child; installing a caller's descriptor in the child is what `posix_spawn`'s file actions add, and what a `sh | cat` would need. The thing POSIX has that this kernel lacks is the *inheritance*, not the pipe.

The console

Input comes in on chapter 5's path, untouched: the UART raises PLIC source 10, the S-mode handler claims it, `handle_uart` drains the receive register — echoing each byte, as a terminal in cooked mode does — and now hands each byte to `console_rx`. The console keeps a 128-byte ring of its own and a count of complete lines in it; a newline bumps the count and wakes anyone sleeping on it.

```
void console_rx(int c)
{
    if (cw - cr >= CONS_SIZE)
        return; /* full: the byte is dropped, as a tty would */
    cbuf[cw++ % CONS_SIZE] = (char)c;
    if (c == '\n') {
        lines++;
        wakeup((void *)&lines);
    }
}

long console_read(unsigned long uva, long n)
{
    char kbuf[CONS_SIZE];
    long i = 0;

    while (lines == 0)
        sleep_on((void *)&lines, "console");
    while (i < n && i < CONS_SIZE && cr != cw) {
        char c = cbuf[cr++ % CONS_SIZE];

        kbuf[i++] = c;
        if (c == '\n') {
            lines--;
            break; /* one line per read */
        }
    }
    if (copy_out_user(curproc, uva, kbuf, (unsigned long)i) < 0)
        return -1;
    return i;
}
```



```

    console_attach(reader);
    console_attach(writer);
    pipe_wire(reader, writer); /* read end → reader, write end →
    ↪ writer */
}
scheduler();
{
    /* Nothing runnable is not the same as nothing left: a process
    * asleep on a channel nobody will ever signal is a deadlock,
    * and PASS over one would be a lie. */
    int i;
    for (i = 0; i < NPROC; i++)
        if (proc_tab[i].state == WAITING) {
            uart_puts("S: deadlock: proc ");
            uart_putc((char)('0' + proc_tab[i].pid));
            uart_puts(" still waiting\r\n");
            halt(1);
        }
}
uart_puts("PASS\r\n");
halt(0);

```

The reader is launched first so that its first read finds the pipe empty and blocks — that is the sentence the lab is built to print — and the check after `scheduler()` is not decoration. “Nothing runnable” and “nothing left” are different: a process asleep on a channel nobody will ever signal is `WAITING`, not `RUNNABLE`, and the loop returns over it. `PASS` on top of a sleeping process would be a lie; the stage looks, and says `deadlock` instead.

The programs

```

int main(void);
int main(void)
{
    char line[64];
    long n;

    n = syscall3(5, 3, (long)line, sizeof(line)); /* read(3, ...):
    ↪ blocks */
    tag();
    puts(" pipe: ");
    syscall3(2, 1, (long)line, n); /* the line,
    ↪ newline included */
    n = syscall3(5, 3, (long)line, sizeof(line)); /* the writer is
    ↪ gone: 0 */

```

```

    if (n == 0) {
        tag();
        puts(" eof\r\n");
    }
    syscall3(1, 0, 0, 0);
    return 0;
}

int main(void);
int main(void)
{
    char line[64];
    long n;

    n = syscall3(5, 0, (long)line, sizeof(line)); /* read(0, ...): the
↪ console */
    tag();
    puts(" console: ");
    syscall3(2, 1, (long)line, n);
    syscall3(2, 3, (long)line, n);                /* write(3, ...):
↪ into the pipe */
    syscall3(1, 0, 0, 0);                          /* exit closes fd 3
↪ */
    return 0;
}

```

The reader reads fd 3 until it returns 0. The writer reads a line from fd 0, says what it got, writes it into fd 3, and exits; the exit closes fd 3, the pipe's last write end, and the reader's second read returns 0. Each program prints `U:` and its pid, which the kernel handed it in `a1` at launch, the same register-at-entry trick chapter 8's programs used for `a0`.

Walkthrough

`ch10-main` (the boot stage: drain, wait for the line, two launches, the console attached to each, the pipe wired, the loop, the deadlock check), `ch10-fd` (the tables and the dispatch), `ch10-read-close` (the two new syscalls), `ch07-copy-in` re-shown beside its mirror's first use, `ch10-pipe-write` and `ch10-pipe-read` (the ring, full and empty, EOF), `ch10-sleep-wake` (blocking as a state), `ch10-console-read` (the interrupt-fed ring and the line discipline), `ch10-reader` and `ch10-writer`.

Exercises

(a) Have the writer, after its line, write 64 bytes seventy times into the pipe with no `yield`, and have the reader drain until EOF. Predict where the writer sleeps and who wakes it, then run it.

Answer, p. 114

(b) Close the write end while the reader is blocked. Derive the reader's return value and the path it takes.

Answer, p. 114

(c) Map the pipe's page into both processes with `U=1` and let them read and write it directly. What did the design just give away?

Answer, p. 115

(d) Make the pipe unbounded — a writer never blocks, the kernel allocates another page whenever the ring is full. Compute the failure.

Answer, p. 115

(e) Suppose the console were read with the external interrupt masked — `sie`. SEIE clear after the boot drain. Derive what happens to a process that asks for a second line.

Answer, p. 115

(f) Name what POSIX pipes provide that this pipe omits.

Answer, p. 115

Lab

```
make lab CH=10
make lab CH=10 IMPL=mime
```

`input.txt` is one line, pipe `me`. After the boot lines: pipe `me` echoed by the drain, exactly as it arrived (a bare newline — the UART echoes what it got); S: console line ready; S: `proc 1` blocks on pipe, the reader's first read; U:2 console: pipe `me`, the writer showing the line it read from fd 0; S: wake `proc 1`, the writer's write into the pipe; S: `proc 2` exited, which closes the write end; U:1 pipe: pipe `me`, the reader's read returning; U:1 eof, the second read returning 0; S: `proc 1` exited; PASS. The order is fixed by the cooperative scheduler and by the boot stage waiting for the line before launching. Nothing

in the transcript is an address or a clock. The chapter 5 proof line (`S: trap sepc=...`) is absent from chapter 7 on: its `sepc` is a link-time address that moved with this stage's prologue, and captures pin behavior, not layout.

`IMPL=mine` copies exactly the objects chapter 10 links and stubs `pipe_write` to print `MINE: pipe_write stub` and halt. The lab boots, the reader blocks, the writer reads its line and reaches the stub — and fails loudly before any wake.

Further reading

- `pipe(2)` — Linux manual page (man-pages project) — the POSIX contract this chapter's pipe is measured against: `PIPE_BUF`, `O_NONBLOCK`, `EPIPE`, and the inheritance through `fork` that `ember` replaces with wiring.
- The RISC-V Instruction Set Manual, Volume II: Privileged Architecture (RISC-V International, v20260120, January 2026) — `wfi` and the interrupt-taking rule behind `console_wait_line`, and the `U/SUM` rules the copies rest on.
- `xv6-riscv` (MIT PDOS) — `kernel/pipe.c` is the same ring with the same two cursors; `kernel/proc.c`'s `sleep/wakeup` is the same channel idea with the lock this single-hart, non-preemptible kernel does not need.

Answers

(a) The writer's first write wakes the reader (`S: wake proc 1`) but does not yield to it: cooperative, it keeps the hart and keeps writing. After exactly 64 writes the ring holds 4096 bytes and the 65th finds it full — `S: proc 2` blocks on `pipe (full)`, on the write-cursor channel, having first woken any reader on the read-cursor channel (already awake). The loop picks the reader, whose *first* read returns 64 bytes: the 8-byte line and 56 bytes of the writer's pattern — one read, two writes, no boundary between them. Its `copy_out_user` done, `pipe_read` calls `wakeup` on the write cursor: `S: wake proc 2`. The reader, also cooperative, drains the ring to empty before it blocks again (`S: proc 1` blocks on `pipe`); the writer resumes, finishes its seventy, exits, and the reader's next read is the EOF. Observed, in that order. Each side wakes the other before it sleeps, which is why neither can wait forever on a pipe the other still holds.

(b) The reader sleeps inside `pipe_read`'s `while (empty && writers > 0)`. Closing the last write end — by `close(3)` or, as in the lab, by `exit` — runs `pipe_close`: `writers` becomes 0 and `wakeup` fires on the read cursor. The reader is picked, `sleep_on` returns, the loop re-checks: still empty, but `writers == 0`, so it falls through with zero bytes available and returns 0. That 0 is EOF,

and it is a statement about the ends, not the bytes: the same empty ring with a writer still open would have put the reader back to sleep. The lab's `U:1 eof` line is this path, taken through the writer's exit.

(c) Everything the copy enforced. The reader can see bytes the writer has not finished writing and the writer can see what the reader has still to read; either can move the cursors, so “full” and “empty” are advisory; a write is no longer bounded by the ring, because a process that ignores the cursor can write anywhere on the page; and a bad pointer on either side is a silent corruption of shared state, not a -1. There is no longer anything for the kernel to refuse, because nothing crosses it. On this kernel the argument has an extra twist: the processes already share a page table, so they *could* already reach each other's U pages by guessing. The pipe's value is precisely that its page is `U=0` — the one page in the exchange that neither side can touch — and giving that up leaves shared memory by accident, which is what the chapter set out not to build.

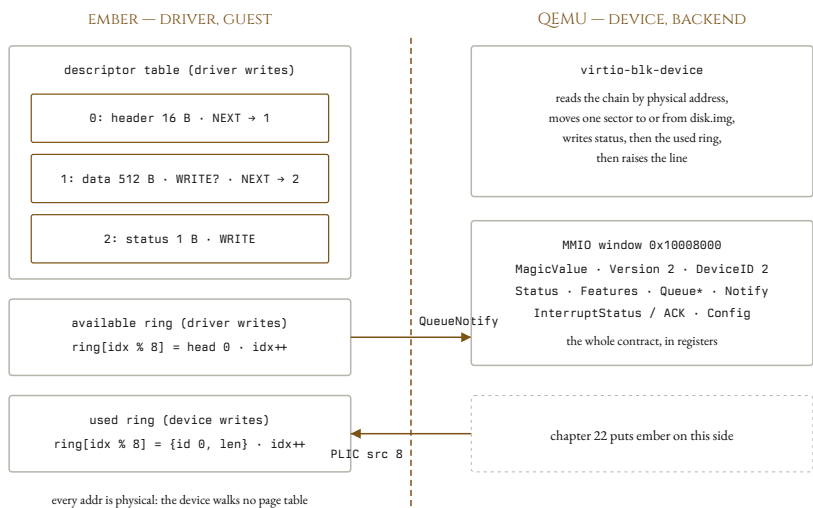
(d) A writer with a slow or absent reader grows the pipe by 4 KiB per page until chapter 9's freelist is empty: 31741 pages, about 124 MiB, then `kalloc_page` returns 0 and the writer gets -1 — from the allocator, not from the pipe, and after taking every page any other process or the kernel itself would have needed. The bound is what makes backpressure exist: a full ring is the kernel telling the writer that the reader is behind, and blocking is the only message the writer cannot ignore. Without it the reader's slowness becomes everyone's exhaustion. This is the same argument chapter 11's block device will make with a fixed ring of descriptors, and chapter 22's backend with the other side of it.

(e) The bytes arrive at the UART and the PLIC raises the line, but the hart never takes the trap: `handle_uart` never runs, `console_rx` never sees a newline, `lines` stays 0. The process's `console_read` sleeps on the line count, and nothing in the kernel can ever call wakeup on it — the only caller is the interrupt handler that has been masked off. That is not a livelock, in which something runs and nothing progresses; it is a process that will never be runnable again. Watch what the rest of the kernel does with it: the scheduler finds nothing runnable and returns, and a boot stage that printed PASS on an empty run would have reported success over a process asleep forever. This chapter's stage checks the table for WAITING first and prints S: deadlock: proc N still waiting instead. A blocking read is only correct if its wakeup path stays alive; masking the interrupt cuts the path and leaves the sleep.

(f) Named, not built: `pipe(2)` as a syscall a process calls for itself, and `fork` to inherit the ends; the `PIPE_BUF` guarantee that a write of at most that many

bytes is atomic with respect to other writers (this pipe has one writer and chunks at 128 bytes, so the question does not arise); `O_NONBLOCK`, a read or write that returns `EAGAIN` instead of sleeping; `SIGPIPE`/`EPIPE` when writing to a pipe with no reader (this pipe returns `-1`, and there is no signal to deliver); readiness — `select`, `poll` — over several descriptors at once; and named pipes in the file system. Every one of them is a refinement of the same three facts: a bounded ring, two ends, a copy in and a copy out.

11. A BLOCK DEVICE



Why this layer exists

Everything the kernel stores lives in DRAM and dies with the power. A file system needs a device that keeps bytes, and a device that keeps bytes is the first one the kernel cannot politely poll forever: a disk takes time, the request has to be handed off and the answer collected later, and the collecting is an interrupt. That is the mechanical reason. The reason this book cares is the other one: until now ember owned the machine. From this chapter it is a *guest* — it asks a device it did not write, does not control and cannot see inside for service, through a contract it has to honor exactly, and the thing on the other side of that contract is QEMU. In chapter 22 the contract is taught in full and the backend is written once; in chapter 36 hearth serves it to guests. Standing on the driver side first is how the reader learns what the other side owes.

The contract is virtio: a device model designed so that a guest and a host that

never share code can share a ring of buffers. This chapter uses the subset one synchronous request needs and says where it stops.

Mechanism

Finding a device with no bus to ask

A *virtqueue* is a ring of buffer descriptors in guest memory that a driver fills and a device drains (or the reverse), with an index on each side saying how far it has got; it is the one data structure *virtio* is built on, and the split layout this chapter uses is the original of the two the spec defines. Before there is a queue there has to be a device, and on this board there is no bus that enumerates devices. There are eight windows of MMIO at `0x10001000`, `0x10002000`, ... `0x10008000`, each a possible *virtio* transport, and the only way to know what is behind one is to look.

Looking is safe because of a magic value. The first register in every *virtio-mmio* window reads `0x74726976` — the bytes *v*, *i*, *r*, *t* in little-endian order — and a window that does not read that is not *virtio*. The second, *Version*, must read 2 for the modern transport; the third, *DeviceID*, names the device type, 2 for block, and 0 for a placeholder the spec says to skip without complaint. The driver maps the eight pages (Part I mapped none of the *virtio* windows), walks them, and stops at the first block device:

```
static void blk_probe(void)
{
    unsigned long pa;
    int i;

    /* The windows are MMIO the kernel has not mapped until now. */
    for (pa = VIRTIO_BASE; pa < VIRTIO_BASE + VIRTIO_COUNT *
        VIRTIO_STEP;
        pa += VIRTIO_STEP)
        map_page(pa, pa, PTE_R | PTE_W);
    __asm__ volatile("sfence.vma" ::: "memory");

    /* 4.2.3.1.1: MagicValue and Version first; DeviceID 0 is a
     * placeholder to be ignored without an error. */
    for (i = 0; i < VIRTIO_COUNT; i++) {
        base = VIRTIO_BASE + (unsigned long)i * VIRTIO_STEP;
        if (rd(R_MAGIC) != MAGIC_VIRT)
            continue;
        if (rd(R_DEVICE_ID) != DEVICE_ID_BLOCK)
            continue;
    }
}
```

```

    if (rd(R_VERSION) != 2) {
        uart_puts("S: blk: legacy transport\r\n");
        halt(1); /* the lab passes force-legacy=false; see board
↳ facts */
    }
    blk_irq = (unsigned)(i + 1); /* DTB: interrupts = <n+1> */
    uart_puts("S: blk: virtio-mmio at ");
    uart_puthex(base);
    uart_puts(" irq=");
    uart_puthex(blk_irq);
    uart_puts(" version=");
    uart_puthex(rd(R_VERSION));
    uart_puts(" device=");
    uart_puthex(rd(R_DEVICE_ID));
    uart_puts("\r\n");
    return;
}
uart_puts("S: blk: no blk device\r\n");
halt(1);
}

```

Two facts about this board fall out of the walk, and both are the kind that must be observed rather than remembered. The device is at the *eightth* window, 0x10008000, not the first: QEMU attaches -device instances to its transports from the top, and the board's device tree lists the eight virtio-mmio nodes in descending order for exactly that reason. And its interrupt line is PLIC source 8, because the device tree says `interrupts = <0x08>` for that node and `<0x01>` for the lowest — the eight windows are sources 1 through 8, in address order, which is what the driver's `blk_irq = i + 1` encodes. Chapter 2's `dumpdtb` exercise is where a reader confirms both.

The version check has teeth. This QEMU's virtio-mmio transport is *legacy* by default: Version reads 1, the modern feature bit is not offered, and the queue is programmed through registers the modern spec does not have. The lab passes -global virtio-mmio.force-legacy=false, and a driver that sees Version 1 anyway prints `legacy transport` and stops rather than speaking a dialect this chapter did not freeze.

Negotiation is a version handshake

A virtio device offers every feature it understands; the driver answers with the subset it accepts; nothing outside that subset exists for the rest of the conversation. The handshake is carried in the Status register as a sequence of bits the driver

sets and never clears: ACKNOWLEDGE (1, “I see a virtio device”), DRIVER (2, “I know how to drive this kind”), then the features, then FEATURES_OK (8), then DRIVER_OK (4). Writing 0 resets the device; FAILED (128) is how the driver gives up.

```
static void blk_negotiate(void)
{
    unsigned int feat_hi;

    /* 3.1.1: reset, ACKNOWLEDGE, DRIVER, then features. */
    wr(R_STATUS, 0);
    wr(R_STATUS, S_ACKNOWLEDGE);
    wr(R_STATUS, S_ACKNOWLEDGE | S_DRIVER);

    /* Read the word that holds bit 32. A modern device MUST offer
     * VIRTIO_F_VERSION_1 (6.2); a driver MUST accept it (6.1).
     ↪ Nothing
     * else is written back: a bit the driver never negotiated is a
     * promise it never made. */
    wr(R_DEV_FEAT_SEL, F_VERSION_1_WORD);
    feat_hi = rd(R_DEV_FEATURES);
    if (!(feat_hi & F_VERSION_1_BIT)) {
        wr(R_STATUS, S_FAILED);
        uart_puts("S: blk: VERSION_1 not offered\r\n");
        halt(1);
    }
    wr(R_DRV_FEAT_SEL, F_VERSION_1_WORD);
    wr(R_DRV_FEATURES, F_VERSION_1_BIT);
    wr(R_DRV_FEAT_SEL, 0);
    wr(R_DRV_FEATURES, 0);

    /* FEATURES_OK, then read it back: a device that rejects the
     ↪ subset
     * clears the bit (2.2.2), and the driver must notice. */
    wr(R_STATUS, S_ACKNOWLEDGE | S_DRIVER | S_FEATURES_OK);
    if (!(rd(R_STATUS) & S_FEATURES_OK)) {
        wr(R_STATUS, S_FAILED);
        uart_puts("S: blk: FEATURES_OK refused\r\n");
        halt(1);
    }
    uart_puts("S: blk: VERSION_1 offered and accepted; "
             "every other feature declined\r\n");
}
```

Features are 64 bits, read 32 at a time through a select register. Bits 0–23 belong to the device type, 24–40 to the transport and queue machinery, and one of them is

the whole point of the modern interface: `VIRTIO_F_VERSION_1`, bit 32, which a modern device **MUST** offer and a driver **MUST** accept. This driver accepts it and nothing else. That is not laziness; it is the rule the spec states as “**MUST NOT** accept a feature which the device did not offer” read from the other end — a bit the driver never wrote back is a promise it never made, and every feature it declined is a field in the configuration space or a request type it is not entitled to assume. Observed on this QEMU, the device offers `0x30006e54` in the low word and `0x101` in the high: `SEG_MAX`, `GEOMETRY`, `BLK_SIZE`, `FLUSH`, `TOPOLOGY`, `CONFIG_WCE`, `DISCARD`, `WRITE_ZEROES`, and the transport’s `INDIRECT_DESC` and `EVENT_IDX`; `VERSION_1` and `RING_RESET` above 32. Ten of those would each have changed what the device expects or what the driver may read. All are declined, and exercise (e) names which one would have added geometry.

`FEATURES_OK` is written and then *read back*. A device that cannot live with the subset clears the bit, and a driver that does not look would proceed on a negotiation that failed. Exercise (a) skips the step and finds QEMU forgiving; the spec allows a device not to be.

The ring: three tables and who writes which

```
static void blk_queue(void)
{
    unsigned long ring = kalloc_page(); /* zeroed: used.flags = 0
    ↪ (2.7.10.1) */
    unsigned long bufs = kalloc_page();

    if (!ring || !bufs)
        halt(1);
    desc = (volatile struct virtq_desc *)ring; /* +0x000
    ↪ */
    avail = (volatile struct virtq_avail *) (ring + 0x080); /* 16-byte
    ↪ desc end at 0x80 */
    used = (volatile struct virtq_used *) (ring + 0x100);
    req = (volatile struct blk_req *)bufs;
    req_status = (volatile unsigned char *) (bufs + 0x10);
    data = (volatile unsigned char *)kalloc_page();
    if (!data)
        halt(1);

    /* 4.2.3.2, in the spec's order. The addresses are physical: the
       * device walks no page table, and these pages are
    ↪ identity-mapped
```

```

    * kernel memory, so the kernel's pointer and the device's address
    * are the same number. That coincidence is the whole of Part IV's
    * shared-memory problem, seen from the easy side. */
wr(R_QUEUE_SEL, 0);
if (rd(R_QUEUE_READY) != 0)
    halt(1);
if (rd(R_QUEUE_NUM_MAX) < QNUM) {
    uart_puts("S: blk: queue too small\r\n");
    halt(1);
}
wr(R_QUEUE_NUM, QNUM);
wr(R_QUEUE_DESC_LO, (unsigned int)ring);
wr(R_QUEUE_DESC_HI, (unsigned int)(ring >> 32));
wr(R_QUEUE_DRV_LO, (unsigned int)(ring + 0x080));
wr(R_QUEUE_DRV_HI, (unsigned int)((ring + 0x080) >> 32));
wr(R_QUEUE_DEV_LO, (unsigned int)(ring + 0x100));
wr(R_QUEUE_DEV_HI, (unsigned int)((ring + 0x100) >> 32));
wr(R_QUEUE_READY, 1);
}

```

The split virtqueue is three arrays. The *descriptor table* — sixteen bytes per entry: a 64-bit address, a 32-bit length, flags, and a next index — describes buffers, and only the driver writes it. The *available ring* — a flags word, a free-running 16-bit `idx`, and a ring of descriptor-chain heads — is how the driver offers chains to the device, and only the driver writes it. The *used ring* — flags, `idx`, and a ring of `{id, len}` pairs — is how the device returns them, and only the device writes it. The ownership is the protocol: each side writes what it owns, reads what the other owns, and the two indices say how far each has got. The queue size is a power of two the driver chooses under the device's `QueueNumMax`; this driver picks 8 and puts all three arrays in one page — descriptors at offset 0 (8×16 bytes ends at `0x80`), the available ring at `0x80`, the used ring at `0x100` — then hands the device the three *physical* addresses and writes 1 to `QueueReady`, in the order the spec lists. The page came from `kalloc_page`, so it is zeroed, which is how the used ring's flags start at 0 as the spec requires.

Every address is physical

The device walks no page table. When it reads descriptor 1 and finds an address, it reads or writes memory *at that address on the bus* — not through `satp`, not through anything. Every pointer the driver puts in a descriptor, and every address it writes to the queue registers, is a physical address, and it had better be one the device can reach. On this kernel that is invisible: the ring and the buffers are `kalloc` pages, identity-mapped `U=0`, so the kernel's pointer and the device's

address are the same number, and the driver writes (unsigned long) ring into a register without a translation. Exercise (b) puts a virtual address there instead and shows what “success” looks like when the device writes somewhere else.

This coincidence is the seed of everything Part IV does with shared memory. A guest under hearth will fill a ring with addresses that are physical *to the guest* and not to the machine; the hypervisor will have to translate them through the G-stage table (chapter 19) before it can touch a byte, or refuse — and the spec has a feature bit for exactly that distinction, `VIRTIO_F_ACCESS_PLATFORM` (33), “the device’s access to memory is limited and/or translated,” which this device does not offer and this driver would not know what to do with. Chapter 36 is where that bit stops being a footnote.

A request, end to end

A block request is a chain of three descriptors, in the order the spec requires — every device-readable element before every device-writable one. The first is a 16-byte header the device reads: type (`VIRTIO_BLK_T_IN` = 0 for read, `_OUT` = 1 for write), a reserved word, and the sector number. The second is the 512-byte data buffer, marked device-writable for a read. The third is a single status byte the device writes: 0 for OK, 1 for an I/O error, 2 for unsupported.

```
/* One request, one chain of three: the header the device reads, one
 * sector it reads or writes, the status byte it writes (2.7.4.2:
 * ↪ every
 * device-writable element after every device-readable one). */
static void blk_op(unsigned int type, unsigned long lba)
{
    if (lba >= capacity) {
        uart_puts("S: blk: lba past capacity\r\n");
        halt(1); /* 5.2.6.1: never submit beyond capacity */
    }
    req->type = type;
    req->reserved = 0;
    req->sector = lba;
    req->status = 0xff; /* a value the device cannot leave behind */

    desc[0].addr = (unsigned long)req;
    desc[0].len = sizeof(struct blk_req);
    desc[0].flags = D_NEXT;
    desc[0].next = 1;
    desc[1].addr = (unsigned long)data;
    desc[1].len = BLK_SECTOR;
```

```

    desc[1].flags = (unsigned short)(D_NEXT | (type == T_IN ? D_WRITE
↵ : 0));
    desc[1].next = 2;
    desc[2].addr = (unsigned long)req_status;
    desc[2].len = 1;
    desc[2].flags = D_WRITE;
    desc[2].next = 0;

    /* 2.7.13: place the head, barrier, bump idx, barrier, notify. */
    avail->ring[avail->idx % QNUM] = 0;
    fence();
    avail->idx++;
    fence();
    wr(R_QUEUE_NOTIFY, 0); /* 4.2.3.3: the queue index */
    blk_wait();
}

void blk_read(unsigned long lba, void *buf)
{
    unsigned long i;

    blk_op(T_IN, lba);
    for (i = 0; i < BLK_SECTOR; i++)
        ((unsigned char *)buf)[i] = data[i];
}

void blk_write(unsigned long lba, const void *buf)
{
    unsigned long i;

    for (i = 0; i < BLK_SECTOR; i++)
        data[i] = ((const unsigned char *)buf)[i];
    blk_op(T_OUT, lba);
}

```

Then the driver's side of 2.7.13: put the head index in the available ring, a memory barrier so the device sees the descriptors before the index, bump `idx`, another barrier, and ring the doorbell — a write of the queue number to `QueueNotify`, which is an MMIO store and therefore the one moment the driver's world and the device's touch. The status byte is preset to `0xff`, a value the device cannot leave behind, so a completion that never wrote it is caught.

```

/* 4.2.3.4.1: read InterruptStatus, handle, acknowledge the same bits.
 * Called from the S trap handler when the PLIC delivers our source,

```



```

    * and from blk_wait after a wfi — idempotent, so either may be first.
    ↪ */
void blk_intr(void)
{
    unsigned int st = rd(R_INT_STATUS);

    if (!st)
        return;
    wr(R_INT_ACK, st);
#if HP_CH == 11
    if (st & 1)
        uart_puts("S: blk: interrupt\r\n");
#endif
}

static void blk_wait(void)
{
    /* The device wrote used.idx before raising the line (2.7.8.2).
    ↪ The
    * hart sleeps in wfi; wfi resumes on a pending, enabled interrupt
    * whether or not sstatus.SIE lets the trap be taken, so the loop
    * also polls the acknowledge path. */
    while (used->idx == last_used) {
        __asm__ volatile("wfi");
        blk_intr();
    }
    fence();
    if (used->ring[last_used % QNUM].id != 0) {
        uart_puts("S: blk: unexpected used id\r\n");
        halt(1);
    }
    last_used++;
    if (*req_status != S_OK) {
        uart_puts("S: blk: status ");
        uart_puthex(*req_status);
        uart_puts("\r\n");
        halt(1);
    }
}

```

The device, when it is done, writes the status, writes {id, len} into the used ring, bumps the used idx — in that order, the spec says, so a driver that sees the index can trust the rest — and raises its line. The line is PLIC source 8; chapter 5’s handler claims it and now calls `blk_intr`, which reads `InterruptStatus`, acknowledges the same bits into `InterruptACK`, and (in this chapter’s image) says

so. The driver, meanwhile, has been in `wfi`. Two details of the wait are deliberate. `wfi` resumes on a pending, enabled interrupt whether or not `sstatus.SIE` would let the trap be taken — the spec says the global enable does not affect it — so in a later chapter where the kernel runs with interrupts masked, the loop still wakes; and because it might wake without the handler having run, the loop calls `blk_intr` itself, which is idempotent, so either path may be first. Under QEMU the device completes the request inside the doorbell write, the interrupt is taken on the very next instruction, and the transcript shows `S: blk: interrupt` before each `ok` — the same order every run.

One at a time, on purpose

This driver submits one request, waits for it, and only then submits the next; the ring never holds more than one chain, `last_used` only ever counts by one, and the queue could have been size 1. That is the subset one synchronous path needs and not a model of a real driver, which keeps the ring full and matches completions to requests by `id`. Chapter 22 builds the protocol in full, on the other side; chapter 12's file system is the only caller this driver has, and it asks for one sector at a time.

```
mem_init();
trap_init();
uart_puts("S: kernel up\r\n");
kalloc_init();
blk_init(); /* probe, negotiate, one queue, DRIVER_OK */
blk_lab(); /* read, write, read back; constants in, constants out
↳ */
uart_puts("PASS\r\n");
halt(0);
```

No process runs in this chapter — the kernel is the only client — so the boot stage is the probe, the negotiation, the queue, and a check: read a sector of the pattern `mkdisk.c` wrote, write a different pattern to another sector, read it back. Constants in, constants out.

Walkthrough

`ch11-main` (the stage), `ch11-probe` (the windows, the magic, the version, the eighth slot and its source), `ch11-negotiate` (the status sequence, bit 32, the read-back), `ch11-ring` (three tables in one page, physical addresses in the registers, the spec's order), `ch11-submit` (a three-descriptor chain, barriers, the doorbell), `ch11-complete` (the acknowledge path, `wfi`, the used ring, the status byte).

Exercises

(a) Skip `FEATURES_OK` — go from `DRIVER` straight to `DRIVER_OK`. Predict which devices tolerate it, then run it.

Answer, p. 128

(b) Put a virtual address in the data descriptor — `0x40000000` plus the buffer's page offset, Part I's second mapping of a kernel page — and read a sector. Derive what the device does, then run it.

Answer, p. 128

(c) Busy-wait on the used ring with the external interrupt masked — no `SEIE`, no `wfi`, just a loop. Does it work? Compute what it costs.

Answer, p. 129

(d) Read two sectors: once as two chains of three, once as one chain of four (header, data, data, status). Do the ring arithmetic.

Answer, p. 129

(e) Which declined feature would have added geometry to the configuration space, and what did the driver correctly refuse to assume?

Answer, p. 129

(f) Name the chapter 22 moment: what changes when `ember` is on the other side of this ring?

Answer, p. 129

Lab

```
make lab CH=11
make lab CH=11 IMPL=mime
```

`labs/ch11/extra.mk` adds two things to the QEMU line: `-global virtio-mmio.force-legacy=false`, without which the device would present the legacy transport and the driver would stop at legacy transport; and `-drive file=disk.img,format=raw,if=none,id=hd0 -device virtio-blk-device,drive=hd0`. `disk.img` is generated, never committed: `mkdisk.c` writes 2048 sectors in which byte *i* of sector *s* is $(s + i) \& 0xff$, and a rebuilt image is byte-identical to the last one — determinism of the image is part of the acceptance.

After the boot lines: S: blk: virtio-mmio at 0x10008000 irq=0x8 version=0x2 device=0x2, the probe's finding; the negotiation line; S: blk: capacity=0x800 sectors, queue 0 size=0x8 — 2048 sectors is the image's size, the queue size is the driver's choice; then three pairs of S: blk: interrupt and an ok: the read of sector 7 against the pattern, the write of sector 9, and the read-back of sector 9 against what was written; PASS. The window address and the source number are this board's, from the device tree; the capacity is the image's; nothing is a clock.

IMPL=mime copies exactly the objects chapter 11 links and stubs blk_init to print MINE: blk_init stub and halt: the lab boots and fails before any probe line.

Further reading

- Virtual I/O Device (VIRTIO) Version 1.2, Committee Specification 01 (OASIS, July 2022) — every constant in this chapter: §2.1 device status, §2.2 feature bits, §2.7 split virtqueues, §4.2.2 the MMIO register layout, §4.2.3 MMIO initialization, §5.2 the block device, §6 reserved feature bits (VIRTIO_F_VERSION_1, VIRTIO_F_ACCESS_PLATFORM). The PDF edition has the same text with page numbers.
- xv6-riscv (MIT PDOS) — kernel/virtio_disk.c is the same driver with the ring kept full and completions matched by id, and its Makefile passes the same force-legacy=false.

Answers

(a) QEMU tolerates it: the run is unchanged, every sector reads and writes correctly. The spec does not promise that. FEATURES_OK is the device's one chance to say the subset is unacceptable — “otherwise it MUST fail to set the FEATURES_OK device status bit” — and a device that never sees the bit written has no moment at which to refuse; whether it then serves requests or ignores them is up to the device. A strict device, or a hypervisor written to the spec (chapter 36's hearth will be), can reasonably treat DRIVER_OK without FEATURES_OK as a broken driver. This driver does the step, and reads the bit back, because “QEMU lets it slide” is not a property of the protocol.

(b) The device does not know it is a virtual address; there is no such thing on its side of the line. It writes 512 bytes to physical 0x40000000, which on this board is not DRAM and not a device, so the bytes go nowhere; it writes status 0 — from its point of view the transfer happened — and the used ring, and raises

the line. The driver sees a completed request with `VIRTIO_BLK_S_OK`, copies its untouched buffer, and the check fails on the first byte: `S: blk: read lba 0x7 MISMATCH at 0x0`. Observed. Nothing faulted, because a device DMA is not a load or a store the MMU sees; the only thing that noticed was the driver's own comparison. That silence is why `VIRTIO_F_ACCESS_PLATFORM` and an IOMMU exist, and why chapter 36 has to translate every guest-physical address before a real device sees it.

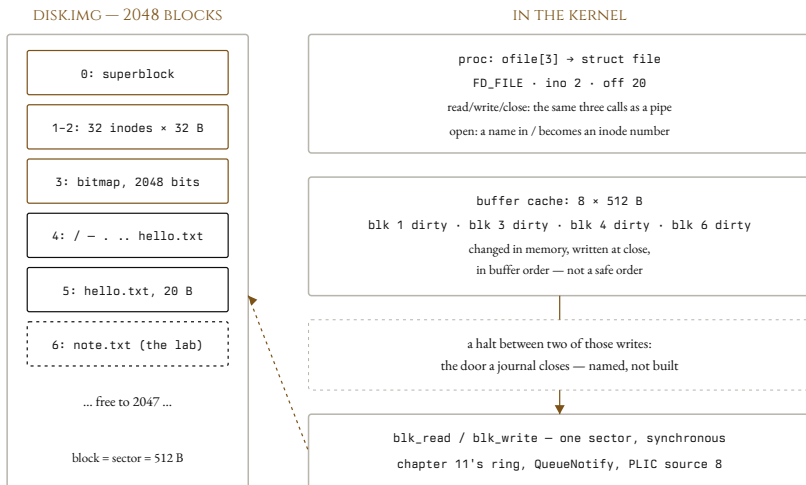
(c) It works: the loop reads `used.idx`, the device has already bumped it inside the doorbell write, and every check passes. Observed — with one visible change: the `S: blk: interrupt` lines move. With the trap never taken, the acknowledge happens only when the *next* request's wait loop polls, so each request's interrupt is acknowledged one request late and the last one never; the device's line stays asserted at the PLIC with nobody to claim it. The cost is what the hart does while it waits: every cycle, on a real disk for milliseconds per request, on a hart that could have run a process instead — and a device that needs the acknowledge to raise the next interrupt eventually stalls behind an unacknowledged one. The interrupt path costs a trap per request and buys every cycle in between, which is the same arithmetic chapter 5 did for the UART with a keyboard on the other end.

(d) Two chains: six descriptors of the eight, two heads in the available ring, `idx` advanced by two, two used entries, two interrupts (or one, if the device coalesces). One chain: four descriptors, one head, `idx` by one, one used entry whose `len` is 1025 (two sectors plus the status byte), one interrupt. Both are legal — the spec says the device “MUST NOT make assumptions about the particular arrangement of descriptors” — and the header's sector number is the start of a contiguous run either way, so the one-chain form is a two-sector read, not two reads. The driver's `last_used`, which counts by one, would have to count completions rather than requests to handle the first form with both chains in flight; that is the bookkeeping this chapter declined and chapter 22 owns.

(e) `VIRTIO_BLK_F_GEOMETRY`, bit 4, which the device offers (0x30006e54 has it). Negotiated, it makes the `cylinders/heads/sectors` struct in `virtio_blk_config` meaningful; declined, those bytes are present but mean nothing, and reading them is reading noise. The driver reads exactly one configuration field, `capacity`, because the spec says it “is always present” — every other field is conditional on a bit, and the driver holds none. The same discipline applies to `BLK_SIZE` (bit 6, also offered): without it the sector is 512 bytes by the protocol's definition, and the driver assumes nothing finer. Declining a feature is not just declining a capability; it is declining to read the fields that come with it.

(f) Every “the device” in this chapter becomes “we.” The backend reads the descriptor table it did not write, follows next through a chain it must not trust (a loop, an address outside the guest, a length past the buffer), translates each address from guest-physical to something it can touch — in wick a host pointer into the guest’s mapped memory, in hearth a G-stage walk — performs the transfer, writes the status byte and the used entry *in that order*, and raises an interrupt it now has to inject rather than receive. The magic value, the version, the feature bits and FEATURES_OK are things it must present and enforce rather than check. Same three tables, same indices, same doorbell; the only thing that moved is which side of the dashed line the reader’s code is on.

12. A FILE SYSTEM SMALL ENOUGH TO READ



Why this layer exists

A file system is the first kernel object in this book the reader has almost certainly only ever *used*. Every object before it — a page table, a trap frame, a scheduler, a ring — was new; a file system is familiar, and familiarity is the problem, because what is familiar is the interface (`open`, a path, `read`, a byte offset) and not the thing under it: a fixed arrangement of bytes on a disk that two programs, a formatter and a kernel, agree on without ever running at the same time. This chapter builds one that fits on a page. Small enough to read means small enough to trust, and later, in Part VI, small enough to hand to a guest whole: a unikernel that carries a disk image carries a format, and this is a format the reader will have written both ends of.

Nothing here is ext2 or FFS. The layout is this book's own — a superblock, an inode array, a bitmap, data — chosen so that every number in it can be derived

by hand and checked with `xxd`. What it leaves out, it names.

Mechanism

The layout, on one page

```
#define FS_MAGIC      0x52424d45u /* "EMBR", little-endian */
#define FS_VERSION    1u
#define FS_BLOCK      512u
#define FS_NBLOCKS    2048u      /* 1 MiB */
#define FS_NINODES    32u
#define FS_INODE_START 1u        /* first inode block */
#define FS_INODES_PER_BLOCK (FS_BLOCK / sizeof(struct dinode))
#define FS_BITMAP_BLOCK 3u
#define FS_DATA_START  4u
#define FS_NDIRECT     6u
#define FS_MAXFILE     (FS_NDIRECT * FS_BLOCK) /* 3072 bytes */
#define FS_NAMELEN     14u
#define FS_ROOT_INO    1u

struct superblock {
    unsigned int magic;
    unsigned int version;
    unsigned int nblocks;
    unsigned int ninodes;
    unsigned int inode_start;
    unsigned int bitmap_block;
    unsigned int data_start;
    unsigned int pad;
};

#define T_FREE 0
#define T_FILE 1
#define T_DIR 2

struct dinode {          /* 32 bytes */
    unsigned short type;
    unsigned short nlink;
    unsigned int size;    /* bytes */
    unsigned int direct[FS_NDIRECT];
};

struct dirent {          /* 16 bytes */
    unsigned short inum;  /* 0: unused entry */
```



```

char name[FS_NAMELEN];    /* NUL-padded, not necessarily terminated
↳ */
};

```

A block is a sector is 512 bytes, chapter 11's unit kept so that no arithmetic hides a conversion. A 1 MiB image is 2048 blocks. Block 0 is the superblock: a magic value (EMBR, so a kernel that mounts the wrong disk says so), a version, and the counts and starting blocks the rest of the layout is derived from. Blocks 1 and 2 hold 32 inodes of 32 bytes, sixteen to a block. An *inode* is the fixed-size record that *is* a file as far as the disk is concerned: a type, a link count, a size in bytes, and six block numbers — where the file's first six blocks are. Names are not in it; a file's name lives in a directory, which is a file whose bytes are 16-byte entries of an inode number and a 14-character name. Block 3 is a bitmap, one bit per block, 2048 bits in a 512-byte block with room to spare. Data starts at block 4, and the root directory — inode 1 — is there, with two entries pointing at itself, `.` and `..`, before any file.

Six direct blocks and nothing else is a decision, not an oversight: the largest file is $6 \times 512 = 3072$ bytes. Exercise (a) prices the indirect block this design leaves out; the reason it is left out is that the walk from an inode to a byte should be one array index and one block read, with no second level to explain before the first is understood.

Two programs, one header

```

int main(int argc, char **argv)
{
    unsigned char *sb = block(0);
    unsigned next_ino = FS_ROOT_INO + 1;    /* 2: the first file */
    unsigned next_blk = FS_DATA_START + 1; /* 5: block 4 is the root
↳ dir */
    unsigned root_direct[FS_NDIRECT] = { FS_DATA_START, 0, 0, 0, 0, 0
↳ };
    unsigned char *root = block(FS_DATA_START);
    unsigned nent = 0, b;
    int i;

    if (argc < 2) {
        fprintf(stderr, "usage: mkfs IMAGE [FILE...]\n");
        return 2;
    }

    /* Superblock: the numbers the kernel checks and derives from. */

```

```

put32(sb + 0, FS_MAGIC);
put32(sb + 4, FS_VERSION);
put32(sb + 8, FS_NBLOCKS);
put32(sb + 12, FS_NINODES);
put32(sb + 16, FS_INODE_START);
put32(sb + 20, FS_BITMAP_BLOCK);
put32(sb + 24, FS_DATA_START);

/* Blocks 0..4 are spoken for before any file: superblock, two
 * inode blocks, the bitmap, the root directory's one block. */
for (b = 0; b <= FS_DATA_START; b++)
    bitmap_set(b);

/* The root directory: "." and ".." both name inode 1. */
put16(root + 16 * nent, FS_ROOT_INO);
memcpy(root + 16 * nent + 2, ".", 1);
nent++;
put16(root + 16 * nent, FS_ROOT_INO);
memcpy(root + 16 * nent + 2, "..", 2);
nent++;

for (i = 2; i < argc; i++) {
    const char *base = strrchr(argv[i], '/');
    unsigned direct[FS_NDIRECT] = { 0 };
    unsigned size = 0, k = 0;
    FILE *f;
    size_t n;

    base = base ? base + 1 : argv[i];
    if (strlen(base) > FS_NAMELEN || next_ino >= FS_NINODES ||
        nent * sizeof(struct dirent) >= FS_BLOCK) {
        fprintf(stderr, "mkfs: cannot add %s\n", base);
        return 1;
    }
    f = fopen(argv[i], "rb");
    if (!f) {
        perror(argv[i]);
        return 1;
    }
    while ((n = fread(block(next_blk), 1, FS_BLOCK, f)) > 0) {
        if (k == FS_NDIRECT) {
            fprintf(stderr, "mkfs: %s exceeds %u bytes\n", base,
                FS_MAXFILE);
            return 1;
        }

```

```

        direct[k++] = next_blk;
        bitmap_set(next_blk);
        next_blk++;
        size += (unsigned)n;
    }
    fclose(f);
    write_inode(next_ino, T_FILE, 1, size, direct);
    put16(root + 16 * nent, next_ino);
    memcpy(root + 16 * nent + 2, base, strlen(base));
    nent++;
    next_ino++;
}
write_inode(FS_ROOT_INO, T_DIR, 2, nent * (unsigned)sizeof(struct
↵ dirent),
           root_direct);

{
    FILE *out = fopen(argv[1], "wb");

    if (!out || fwrite(image, 1, sizeof image, out) != sizeof
↵ image ||
        fclose(out) != 0) {
        perror(argv[1]);
        return 1;
    }
}
return 0;
}

```

`mkfs` runs on the host. It writes the superblock, marks blocks 0 through 4 used, lays down the root directory with `.` and `..`, and then, for each file named on its command line, takes the next inode and the next data blocks in order and appends an entry to the root. Inodes are handed out from 2, data blocks from 5, in argument order, so the image is a function of its inputs and nothing else — a rebuild is byte-identical, and the lab checks that it is. The kernel and the formatter include the same `fs.h`; the constants and the two structs are the whole agreement, and neither side can drift from it without the other noticing at the magic value.

A formatter on the host is the honest place for it (exercise (e) asks why): formatting is not mounting. The kernel that reads this layout never needs to create it, and a guest with no way to get files in but the disk itself is served by a host that writes the disk.

Where the kernel keeps blocks

```
#define NBUF 8

struct buf {
    unsigned int blockno;
    int valid, dirty;
    unsigned char data[FS_BLOCK];
};

static struct buf bufs[NBUF];
static int victim;

/* The block, in memory. A miss evicts the next buffer in rotation,
 * writing it back first if it is dirty. */
static struct buf *bget(unsigned int b)
{
    struct buf *bp;
    int i;

    for (i = 0; i < NBUF; i++)
        if (bufs[i].valid && bufs[i].blockno == b)
            return &bufs[i];
    bp = &bufs[victim];
    victim = (victim + 1) % NBUF;
    if (bp->valid && bp->dirty)
        blk_write(bp->blockno, bp->data);
    blk_read(b, bp->data);
    bp->blockno = b;
    bp->valid = 1;
    bp->dirty = 0;
    return bp;
}

/* Every dirty block to the disk, in buffer order. Returns the count.
 * */
static unsigned int bsync(void)
{
    unsigned int n = 0;
    int i;

    for (i = 0; i < NBUF; i++)
        if (bufs[i].valid && bufs[i].dirty) {
            blk_write(bufs[i].blockno, bufs[i].data);
            bufs[i].dirty = 0;
            n++;
        }
}
```

```

    }
    return n;
}

```

Every block the kernel touches goes through a cache of eight buffers. `bget` returns the block in memory, reading it through chapter 11's `blk_read` on a miss and evicting the next buffer in rotation — writing it back first if it is dirty. A block that is changed is marked dirty and stays in memory; `bsync`, called from `fs_close`, writes every dirty buffer to the disk and counts them. The cache is not there for speed — with synchronous single-sector I/O nothing here is fast — it is there so that the order in which changes reach the disk is a *fact* in the code, visible and discussable, rather than an accident of when each write happened to be issued. That order is the subject of the section on consistency below.

Inodes are read out of and written back into their block by value: with eight buffers and a rotating victim, a pointer into a buffer does not survive another `bget`, so `iread` and `iwrite` copy.

Inodes, blocks, and a name

```

/* The first free inode, marked as the given type. Inode 0 is never
 * handed out: it is the number that means "none". */
static unsigned int ialloc(unsigned short type)
{
    struct dinode di;
    unsigned int ino;

    for (ino = 1; ino < sb.ninodes; ino++) {
        iread(ino, &di);
        if (di.type == T_FREE) {
            unsigned int k;

            di.type = type;
            di.nlink = 1;
            di.size = 0;
            for (k = 0; k < FS_NDIRECT; k++)
                di.direct[k] = 0;
            iwrite(ino, &di);
            return ino;
        }
    }
    return 0;
}

```

```

/* The first free data block, zeroed: a file's new block must not
↵ show
* what the block held last. */
static unsigned int balloc(void)
{
    struct buf *bm = bget(sb.bitmap_block);
    unsigned int b;

    for (b = sb.data_start; b < sb.nblocks; b++)
        if (!(bm->data[b / 8] & (1u << (b % 8)))) {
            struct buf *bp;
            unsigned int i;

            bm->data[b / 8] |= (unsigned char)(1u << (b % 8));
            bm->dirty = 1;
            bp = bget(b);
            for (i = 0; i < FS_BLOCK; i++)
                bp->data[i] = 0;
            bp->dirty = 1;
            return b;
        }
    return 0;
}

```

`ialloc` scans the array from inode 1 for a free one and stamps it with a type; inode 0 is never handed out because 0 is the number that means “no inode” in a directory entry. `balloc` scans the bitmap from the first data block for a clear bit, sets it, and zeroes the block in the cache — a file’s new block must not show what the block held last, which on a fresh image is nothing and on a used one is somebody’s old data.

```

long fs_read(unsigned int ino, unsigned long off, void *kbuf, long n)
{
    struct dinode di;
    long done = 0;

    iread(ino, &di);
    if (off >= di.size)
        return 0;
    if ((unsigned long)n > di.size - off)
        n = (long)(di.size - off);
    while (done < n) {
        unsigned long o = off + (unsigned long)done;
        unsigned int b = di.direct[o / FS_BLOCK];
        unsigned long in = o % FS_BLOCK;

```

```

    long chunk = (long)(FS_BLOCK - in);
    struct buf *bp;

    if (chunk > n - done)
        chunk = n - done;
    bp = bget(b);
    kmemcpy((unsigned char *)kbuf + done, bp->data + in,
        (unsigned long)chunk);
    done += chunk;
}
return done;
}

```

Reading is the walk the layout promised: the inode's size bounds the request, `off / 512` picks a direct block, `off % 512` an offset in it, and the bytes come out of the cached block. Writing is the same in reverse, allocating a block the first time an index is touched and raising the size. Neither knows anything about names.

/ One component under the root. Returns the inode number, creating
 * the file if asked and it is absent. Nothing is opened in the sense
 * of a kernel object here – that is fd.c's struct file; this is the
 * name-to-inode step the directory exists for. */*

```

int fs_open(const char *name, int flags)
{
    unsigned int ino = dir_lookup(name);

    if (ino == 0) {
        if (!(flags & O_CREAT))
            return -1;
        ino = ialloc(T_FILE);
        if (ino == 0 || dir_link(name, ino) < 0)
            return -1;
        uart_puts("S: fs: create ");
        uart_puts(name);
        uart_puts(" inode ");
        uart_puthex(ino);
        uart_puts("\r\n");
        return (int)ino;
    }
    uart_puts("S: fs: open ");
    uart_puts(name);
    uart_puts(" inode ");
    uart_puthex(ino);
    uart_puts(" size ");
}

```

```

    uart_puthex(fs_size(ino));
    uart_puts("\r\n");
    return (int)ino;
}

```

`fs_open` is where a name becomes an inode. It reads the root directory as a file — 16 bytes at a time through `fs_read` on inode 1 — and compares names; if the name is absent and `O_CREAT` was asked, it allocates an inode and appends an entry, which is a `fs_write` on inode 1. That the directory is a file, read and written through the same two functions as everything else, is the trick that keeps the design on one page. Paths are one component under the root; a hierarchy is the same lookup repeated per component on a directory inode instead of the root, and it is left out because repetition is not a mechanism.

```

/* The path comes in through the walk like any user bytes, bounded
   ↪ and
   * NUL-terminated by the kernel, not trusted to be. */
static long sys_open(long path, long flags)
{
    char name[FS_NAMELEN + 2];
    struct file *f;
    int ino;
    unsigned int i;

    for (i = 0; i < sizeof name; i++)
        name[i] = 0;
    if (fs_copy_name(curproc, (unsigned long)path, name, sizeof name)
        ↪ < 0)
        return -1;
    ino = fs_open(name, (int)flags);
    if (ino < 0)
        return -1;
    f = file_alloc();
    if (!f)
        return -1;
    f->type = FD_FILE;
    f->ino = (unsigned int)ino;
    f->off = 0;
    f->readable = 1;
    f->writable = (flags & O_WRONLY) != 0;
    return fd_install(curproc, f);
}

```

The user's side is the third kind of descriptor. `open` copies the path in through the walk, bounded and NUL-terminated by the kernel, asks `fs_open` for an

inode, and installs a `struct file` of kind `FD_FILE` with that inode and an offset of 0. From then on `read`, `write`, and `close` are the three calls chapter 10 built, dispatched by kind: a file's `read` moves bytes from the cache to the user through `copy_out_user` and advances the offset; its `close` is the moment the cache reaches the disk. A process cannot tell a pipe from a file by the calls it makes on it, which is the point of the descriptor table.

What a halt in the middle leaves behind

The lab creates `note.txt`, writes 17 bytes, and closes it. Between the create and the close, four blocks are dirty: block 1 (inode 3 now says file, size 17, block 6; inode 1's size grew by one entry), block 4 (the root's new entry names inode 3), block 3 (bit 6 now set), and block 6 (the bytes). `bsync` writes them in buffer order, which is the order the buffers were first used in this run: 1, 4, 3, 6 — inode, directory, bitmap, data. Now halt the machine between any two of those writes and read the disk.

After 1 alone: inode 3 exists with size 17 and block 6, the root's size covers a fourth entry, but block 4 on disk still has zeros there — an entry with inode 0, which lookup skips. An allocated inode nobody can name, and block 6 not marked used: an orphan, and a block the next `ballocc` will hand out again. After 1 and 4: the file has a name and an inode, its block is not marked used, and the block holds zeros; the next file to allocate a block gets block 6 too, and two files now share it — the next write to either silently corrupts the other. After 1, 4 and 3: the file is complete in every structure and its content is zeros. Exercise (d) asks which single lost write is worst; the answer is the bitmap, because a lost bitmap write is not a wrong file but a *future* double allocation, and nothing on the disk records that it happened.

The rule real file systems follow is that a change must never be *reachable* before it is *consistent*: write the data and the bitmap and the inode, then the directory entry that names them — and never in the order this cache happens to have. Either the writes are ordered by hand (FFS's soft updates), or every group of writes is first written whole to a log and replayed after a crash (a journal, which is what ext3 added to ext2 and what every file system the reader runs has). This chapter builds neither. The cache makes the order visible; the reader now knows what a journal is *for*, which is more than the reader knew before, and a journal itself is a chapter this book does not have.

```
mem_init();
trap_init();
uart_puts("S: kernel up\r\n");
```

```

kalloc_init();
blk_init();    /* chapter 11: the disk */
fs_init();     /* the superblock: is this a disk we understand? */
{
    struct proc *p = launch(user6_blob_start,
                            (unsigned long)(user6_blob_end -
                                             user6_blob_start),
                            USER_BASE, 0);

    console_attach(p);
}
scheduler();
{
    int i;
    for (i = 0; i < NPROC; i++)
        if (proc_tab[i].state == WAITING) {
            uart_puts("S: deadlock: proc ");
            uart_putc((char)('0' + proc_tab[i].pid));
            uart_puts(" still waiting\r\n");
            halt(1);
        }
}
uart_puts("PASS\r\n");
halt(0);

```

The boot stage mounts — reads the superblock and refuses anything but its own magic and version — and launches the probe with a console. `fs_init`'s line and the four lines the file system prints are the transcript's evidence that the layout on disk and the layout in the header agree.

```

int main(void);
int main(void)
{
    static const char note[] = "written by ember\n";
    char buf[64];
    long fd, n, i, same = 1;

    fd = syscall3(7, (long)"hello.txt", O_RDONLY, 0); /* open */
    n = syscall3(5, fd, (long)buf, sizeof buf);        /* read */
    tag();
    puts(" hello.txt: ");
    syscall3(2, 1, (long)buf, n);
    syscall3(6, fd, 0, 0);                             /* close */

    fd = syscall3(7, (long)"note.txt", O_CREAT | O_WRONLY, 0);
    syscall3(2, fd, (long)note, slen(note));            /* write */

```

```

    syscall13(6, fd, 0, 0);                                /* close: the
↪  sync */

    fd = syscall13(7, (long)"note.txt", O_RDONLY, 0);
    n = syscall13(5, fd, (long)buf, sizeof buf);
    syscall13(6, fd, 0, 0);
    if (n != strlen(note))
        same = 0;
    for (i = 0; same && i < n; i++)
        if (buf[i] != note[i])
            same = 0;

    tag();
    puts(same ? " note.txt: readback equal\r\n" : " note.txt: readback
↪  DIFFERS\r\n");

    syscall13(1, 0, 0, 0);
    return 0;
}

```

Walkthrough

ch12-super (the layout: constants and two structs), ch12-mkfs (the formatter's core), ch12-bcache (eight buffers, the rotation, the sync), ch12-ialloc (free inode, free block), ch12-read (inode → block → byte), ch12-open (name → inode, create), ch12-sys-open (the third descriptor kind), ch12-main (mount and launch), ch12-probe.

Exercises

(a) Compute the largest file this layout allows, and what one indirect block would add and cost.

Answer, p. 145

(b) Suppose an `unlink` existed that removed a directory entry and decremented the inode's link count, freeing it at zero. Apply it to `.` and to `..` and derive what goes wrong.

Answer, p. 145

(c) After the lab, read `disk.img` on the host with `xxd` and find the planted file and the created one by derived offsets.

Answer, p. 145

(d) Of the four blocks the lab's close writes, which single lost write does the most damage?

Answer, p. 146

(e) Why does mkfs run on the host and not in the kernel?

Answer, p. 146

(f) What would a unikernel guest (Part VI) delete from this design, and what would it keep?

Answer, p. 146

Lab

```
make lab CH=12
make lab CH=12 IMPL=mime
```

labs/ch12/extra.mk adds chapter 11's device flags and builds disk.img from files/hello.txt with the reference mkfs — and rebuilds it before *every* run, because the lab creates a file: a second run on the same image would find note.txt already there and print a different transcript. The image is generated and gitignored; mkfs.c, fs.h and files/ are what is committed, and a rebuilt image is byte-identical.

After the block device's lines: S: fs: superblock ok blocks=0x800 inodes=0x20 data from 0x4, the mount; S: fs: open hello.txt inode 0x2 size 0x14, the lookup; U:1 hello.txt: hello from the disk — the file's bytes, newline included, printed by the process; S: fs: create note.txt inode 0x3; S: fs: sync wrote 0x4 blocks, the close of the written file — the four blocks the consistency section walks through; S: fs: open note.txt inode 0x3 size 0x11, the second open finding what the first created; U: 1 note.txt: readback equal; S: proc 1 exited; PASS. Every number is a property of the layout or of the two files' lengths.

IMPL=mime copies exactly the objects chapter 12 links and stubs fs_open to print MINE: fs_open stub and halt: the mount succeeds and the first open fails loudly.

Further reading

- A Fast File System for UNIX (McKusick, Joy, Leffler, Fabry; Computer Systems Research Group, UC Berkeley, 1984) — the paper behind the

inode-array-plus-bitmap shape this layout is a miniature of, and behind the write-ordering discipline the consistency section names.

- xv6-riscv (MIT PDOS) — `kernel/fs.c` and `mkfs/mkfs.c` are the same two-program shape with a log this chapter names and does not build; its inode has twelve direct blocks and one indirect, which is exercise (a) done.
- `pipe(2)` — Linux manual page and its neighbours `open(2)`, `unlink(2)`, `rmdir(2)` (man-pages project) — the contracts behind exercises (b) and (f); the `./..` rules are stated there as errors a call must return.

Answers

(a) Six direct blocks of 512 bytes: 3072 bytes. An indirect block is a data block that holds block numbers instead of bytes — $512 / 4 = 128$ of them — so one indirect pointer in the inode adds $128 \times 512 = 65536$ bytes, for a maximum of 68608. Its cost is one extra block per file that uses it (the pointer block itself, a data block taken from the same pool), and one extra read on every access past 3 KiB: inode, then the indirect block, then the data. A doubly indirect block would add $128 \times 128 \times 512 = 8$ MiB behind two extra reads. The layout stays at six because the first level should be understood before the second is added, and because every file in this book’s labs is smaller than a page.

(b) Both entries name the root itself, inode 1, which is why its link count is 2 and not 1. Unlinking `.` removes an entry and decrements inode 1 to 1; unlinking `..` then removes the other and decrements it to 0, and the naive rule frees the root directory — the very file the entries were being removed from, with the entries for `hello.txt` still in its block, now pointing into an inode marked free, and a bitmap that still says block 4 is in use. On a hierarchy the failure is quieter and worse: a subdirectory’s `..` is a second link to its parent, so a parent’s count is 2 plus one per child, and an unlink that treated `..` like a name would free parents that still had children pointing at them. `unlink` must refuse the two names outright, and a directory removal must check that nothing but those two remain — which is exactly what POSIX `unlink(2)` and `rmdir(2)` do, and why the two entries exist at all: they make “parent” a lookup rather than a separate structure.

(c) Offsets are block $\times$ 512 plus the slot. The superblock is at 0: 45 4d 42 52 is EMBR, then version 1, 0x800 blocks, 0x20 inodes, inode start 1, bitmap 3, data 4. Inode i is at $0x200 + 32 \cdot i$: inode 1 at 0x220 reads type 2, nlink 2, size 0x30 (three entries before the lab, four after: 0x40) and `direct[0] = 4`; inode 2 at 0x240 reads type 1, nlink 1, size 0x14, `direct[0] = 5`; inode 3 at 0x260, created by the

lab, reads type 1, size 0x11, direct[0] = 6. The root directory is block 4 at 0x800: 01 00 2e is inode 1 named ., then .. at 0x810, then 02 00 68 65 6c 6c 6f 2e 74 78 74 — inode 2, hello.txt — at 0x820, and after the lab 03 00 6e 6f 74 65 2e 74 78 74 at 0x830. Block 5 at 0xa00 is hello from the disk; block 6 at 0xc00 is written by ember. The bitmap's first byte at 0x600 is 7f: blocks 0 through 6 used. Observed on this image; the image is regenerated before every run, so a reader sees the same bytes.

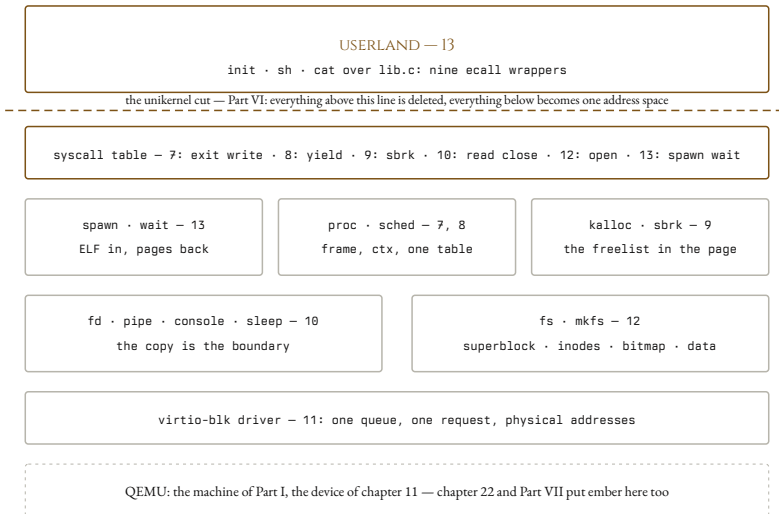
(d) The bitmap. Lose the inode block and note.txt vanishes cleanly — the directory entry is there but names an inode the disk still calls free, and lookup would find a file whose inode says T_FREE with size 0 — plus one leaked block. Lose the directory block and the inode and its data exist with no name: an orphan and a leaked block, invisible but harmless. Lose the data block and the file exists, correctly sized, full of zeros — wrong, but wrong in a way a reader of the file can see. Lose the bitmap write and everything about note.txt is correct, and block 6 is marked free: the next `ballocc` hands it to the next file, two inodes now point at one block, and the next write through either corrupts the other. No structure on the disk records the error, no read of either file reveals it until it happens, and it happens later. That is the shape of the worst file-system bug: not a missing file but a shared block, and it is why the bitmap is written before anything that depends on it in every design that takes crashes seriously.

(e) Because formatting and mounting are different jobs with different inputs. Mounting reads a layout that already exists and needs nothing but the disk. Formatting creates the layout from files — and the kernel has no files until the layout exists; the only way bytes get into this guest is the disk itself. The host has the files, a C compiler, `xxd` to check the result, and no need to be small. It also keeps the kernel honest: a kernel that could format its own disk would be tempted to “fix” a layout it did not understand, and the magic-value refusal in `fs_init` would have an escape hatch. Real systems draw the same line: `mke2fs` is a program, not a syscall, and the kernel's `ext4` driver has never created a file system.

(f) It keeps the on-disk layout — that is the contract with whoever wrote the image, and a unikernel is often *defined* by shipping one — and it keeps the cache, or at least the write ordering the cache makes visible. It deletes the boundary this chapter built the file system behind: there is no `U` mode, so `open` is a function call and not an `ecall`, the path is a pointer and not a copy through the walk, `read` fills the caller's buffer directly and `copy_out_user` is gone, and the descriptor table — the indirection that let a process treat a pipe and a file alike — becomes an ordinary struct the application holds. What that deletion buys is a copy per

byte and a trap per call; what it gives up is that a bug in the application's use of a pointer is now a bug in the file system's memory. Chapter 32 fixes what a guest may assume about its disk; this chapter is what it would be assuming.

13. USERLAND



Why this layer exists

An operating system is proven by the programs it runs, and until now every program ember ran was baked into the kernel's .rodata at build time and launched by the boot stage. That is a test harness, not an OS. This chapter closes the loop: programs are files on the disk of chapter 12, loaded by the kernel when a running program asks, given a console and an argument, waited for, and reclaimed. At the end there is an `init`, a shell, and a `cat`, and a reader who types `cat hello.txt` at the shell is running a program the kernel found by name, checked, and mapped — the whole of Part II in one keystroke.

The second reason is Part VI's. A unkernel is an image with the userland cut off and the rest inlined; to cut a userland off there has to be one worth cutting, with a syscall boundary that can be seen and deleted. This chapter's image is that: an

S-mode kernel, U-mode programs, nine syscalls between them, IPC, a device, a file system, a shell. hearth will be allowed to run this same image, stripped, as a guest — something the reader wrote, not bought.

Mechanism

spawn is proc_create fed from a file

Chapter 7's `proc_create` took a blob and an address and produced a process: pages, a fabricated frame, a kernel stack. `spawn` is the same call with the blob read from a file:

```
struct proc *spawn_path(const char *name, const char *argstr)
{
    const struct elf64_phdr *ph;
    struct proc *p;
    unsigned int size;
    int ino;

    ino = fs_open(name, O_RDONLY);
    if (ino < 0)
        return 0;
    size = fs_size((unsigned int)ino);
    if (size > sizeof img)
        return (struct proc *)reject("file larger than the loader's
        ↪ buffer");
    fs_read((unsigned int)ino, 0, img, (long)size);
    ph = elf_check(size);
    if (!ph)
        return 0;

    /* Chapter 7's shape: the segment's file bytes become the code
    ↪ page
    * at p_vaddr, and the page beyond p_filesz — the .bss the ELF
    * promised in p_memsz — is zero because kalloc zeroed it. Then a
    * stack page, a kernel stack, a fabricated frame at e_entry. */
    p = launch(img + ph->p_offset, ph->p_filesz, ph->p_vaddr, 0);
    console_attach(p);
    p->parent = curproc; /* 0 for init, spawned by the boot stage */

    /* One argument, as a string at the bottom of the stack page; the
    * child finds its address in a2. Not argv — named, not built. */
    if (argstr) {
        unsigned long *slot = walk(ph->p_vaddr + PAGE_SIZE);
```

```

    unsigned char *dst = (unsigned char *)(((slot >> 10) <<
    ↪ PAGE_SHIFT));
    unsigned int i;

    for (i = 0; i < 63 && argstr[i]; i++)
        dst[i] = (unsigned char)argstr[i];
    dst[i] = 0;
    p->tf.a2 = ph->p_vaddr + PAGE_SIZE;
}

uart_puts("S: spawn ");
uart_puts(name);
uart_puts(" pid ");
uart_puthex((unsigned long)p->pid);
uart_puts(" entry ");
uart_puthex(ph->p_vaddr);
uart_puts("\r\n");
return p;
}

long spawn_impl(long path_uva, long arg_uva)
{
    char name[FS_NAMELEN + 2];
    char argbuf[64];
    struct proc *p;
    unsigned int i;

    for (i = 0; i < sizeof name; i++)
        name[i] = 0;
    for (i = 0; i < sizeof argbuf; i++)
        argbuf[i] = 0;
    if (fs_copy_name(curproc, (unsigned long)path_uva, name, sizeof
    ↪ name) < 0)
        return -1;
    if (arg_uva &&
        fs_copy_name(curproc, (unsigned long)arg_uva, argbuf, sizeof
    ↪ argbuf) < 0)
        return -1;
    p = spawn_path(name, arg_uva ? argbuf : 0);
    return p ? p->pid : -1;
}

```

The name goes to chapter 12's `fs_open`, the bytes to `fs_read`, and the result to launch — chapter 8's `create-and-fabricate` — with the segment's file bytes as the blob and the segment's virtual address as the base. The child gets a console

(`console_attach`, fds 0–2), a parent (whoever called), and, if the caller passed one, a single argument string copied to the bottom of the child’s stack page with its address in `a2`. That is not `argv`; it is one string, and the chapter says so rather than pretending. The kernel prints what it launched, and the child’s first instruction is the libc stub’s `_start`.

Why `spawn` and not `fork` and `exec`? Because `fork` needs a second copy of an address space, and this kernel has one page table. `fork` on `ember` would mean either copying every U page of the parent into fresh pages at *different* virtual addresses (which the program was not linked for) or per-process page tables with copy-on-write, which is a chapter of its own and the design every production kernel has. `exec` without `fork` replaces the caller, and a shell that replaced itself with `cat` would not be there to prompt again. `spawn` — create a new process from a file, in one call — is what POSIX offers as `posix_spawn`, for exactly the cases where `fork`’s copy is the wrong price; `xv6`, which has per-process tables, does `fork` and `exec`, and Part III’s chapter 16 reads it. `ember` keeps the one table, and one consequence follows: each program is linked at its own base — `init` at `0x10000`, `sh` at `0x20000`, `cat` at `0x30000` — and a second `sh` at once would collide with the first. Named, not fixed.

The kernel is the loader now

Chapter 2 read ELF headers to learn what QEMU loads and where. This chapter does what QEMU did:

```
struct elf64_ehdr {
    unsigned char e_ident[16];
    unsigned short e_type, e_machine;
    unsigned int e_version;
    unsigned long e_entry, e_phoff, e_shoff;
    unsigned int e_flags;
    unsigned short e_ehsize, e_phentsize, e_phnum;
    unsigned short e_shentsize, e_shnum, e_shstrndx;
};

struct elf64_phdr {
    unsigned int p_type, p_flags;
    unsigned long p_offset, p_vaddr, p_paddr, p_filesz, p_memsz,
    ↵ p_align;
};

#define ET_EXEC      2
#define EM_RISCV     243
```

```

#define ELFCLASS64 2
#define ELFDATA2LSB 1
#define PT_LOAD 1

static unsigned char img[4096]; /* a file is at most 3 KiB (fs.h) */

static const char *reject(const char *why)
{
    uart_puts("S: spawn: ");
    uart_puts(why);
    uart_puts("\r\n");
    return 0;
}

/* Validate the file in img[] and return its one PT_LOAD, or 0. Every
 * check names what a malformed file would have made the kernel do.
 * */
static const struct elf64_phdr *elf_check(unsigned int size)
{
    const struct elf64_ehdr *eh = (const struct elf64_ehdr *)img;
    const struct elf64_phdr *ph;
    unsigned int i;

    if (size < sizeof *eh)
        return (const struct elf64_phdr *)reject("file shorter than a
        ↵ header");
    if (eh->e_ident[0] != 0x7f || eh->e_ident[1] != 'E' ||
        eh->e_ident[2] != 'L' || eh->e_ident[3] != 'F')
        return (const struct elf64_phdr *)reject("bad magic");
    if (eh->e_ident[4] != ELFCLASS64 || eh->e_ident[5] != ELFDATA2LSB)
        return (const struct elf64_phdr *)reject("not ELF64
        ↵ little-endian");
    if (eh->e_type != ET_EXEC || eh->e_machine != EM_RISCV)
        return (const struct elf64_phdr *)reject("not a RISC-V
        ↵ executable");
    if (eh->e_phentsize != sizeof *ph || eh->e_phnum == 0 ||
        eh->e_phoff + (unsigned long)eh->e_phnum * sizeof *ph > size)
        return (const struct elf64_phdr *)reject("program headers out
        ↵ of range");
    for (i = 0; i < eh->e_phnum; i++) {
        ph = (const struct elf64_phdr *) (img + eh->e_phoff) + i;
        if (ph->p_type != PT_LOAD)
            continue;
        if (ph->p_vaddr & (PAGE_SIZE - 1))

```

```

        return (const struct elf64_phdr *)reject("segment not
        ↪ page-aligned");
    if (ph->p_offset + ph->p_filesz > size || ph->p_filesz >
        ↪ ph->p_memsz)
        return (const struct elf64_phdr *)reject("segment out of
        ↪ range");
    if (ph->p_memsz > PAGE_SIZE)
        return (const struct elf64_phdr *)reject("segment larger
        ↪ than a page");
    if (eh->e_entry != ph->p_vaddr)
        return (const struct elf64_phdr *)reject("entry is not the
        ↪ segment start");
    return ph; /* the first PT_LOAD is the program */
}
return (const struct elf64_phdr *)reject("no loadable segment");
}

```

The fields are the System V gABI's: the four magic bytes, the class and data bytes in `e_ident` (64-bit, little-endian), `e_type` = 2 for an executable, `e_machine` = 243 for RISC-V — the psABI's number, and the `f3 00` at byte 18 of every binary this toolchain makes — `e_entry`, and the program header table's offset, entry size and count. Then the program headers, of which one matters: the first `PT_LOAD` says where in the file the program's bytes are (`p_offset`, `p_filesz`), where in memory they go (`p_vaddr`), and how much memory the program expects (`p_memsz`). Every check names what a malformed file would have done to the kernel: a segment past the end of the file is a read past the loader's buffer; a segment larger than a page is a copy over the stack page; an entry that is not the segment's start is a jump into the middle of something. Exercise (a) feeds the loader a text file.

`p_memsz` is larger than `p_filesz` when the program has a `.bss`: bytes that are zero at start and take no space in the file. The loader does nothing for them, and that is correct, because `proc_create` copies the file bytes into a page that `kalloc_page` zeroed — the `.bss` is zero for free, and the stub's `_start` zeroes it again anyway. Exercise (b) puts numbers on it.

The user build makes ELF's the loader can take, which is not what this toolchain does on its own. It links position-independent executables by default, with `.interp`, `.dynamic`, a GOT and an unwind table, and a load segment that starts below the program's base; the flat blobs of chapters 7–12 never noticed, because `objcopy -O binary` discarded all of it. As files they would be 6 KiB, over chapter 12's 3 KiB limit. `-fno-pie -no-pie`, no unwind tables, no build ID, and `-N` so the load segment is not padded to a page in the file: the shell is 1304 bytes on disk. That is chapter 2's lesson from the other side — the ELF is a container, and

what the linker puts in it is a choice.

The stub is the ABI

```
void _start(void) __attribute__((section(".text.start"), noreturn));
void _start(void)
{
    extern char __bss_start[], __bss_end[], stack_top[];
    char *p;
    register long a1r __asm__("a1");
    register long a2r __asm__("a2");

    __asm__ volatile("" : "+r"(a1r), "+r"(a2r)); /* before .bss is
↳ wiped */
    for (p = __bss_start; p < __bss_end; p++)
        *p = 0;
    pid = a1r;
    arg = (const char *)a2r;
    __asm__ volatile(
        "mv sp, %0\n"
        "call main\n"
        "li a7, 1\n" /* exit(main()) - a0 already holds the value
↳ */
        "ecall\n"
        :: "r"(stack_top) : "memory");
    __builtin_unreachable();
}

static long syscall3(long n, long a0, long a1, long a2)
{
    register long a0r __asm__("a0") = a0;
    register long a1r __asm__("a1") = a1;
    register long a2r __asm__("a2") = a2;
    register long a7r __asm__("a7") = n;

    __asm__ volatile("ecall"
        : "+r"(a0r)
        : "r"(a1r), "r"(a2r), "r"(a7r)
        : "memory");

    return a0r;
}
```

lib.c is the whole syscall interface as a user program sees it: `_start`, which captures the two registers the kernel set (the pid in `a1`, the argument in `a2`), zeroes `.bss`, sets `sp`, calls `main`, and turns `main`'s return into `exit`; and nine

wrappers, one `ecall` each, with the numbers the kernel's table froze in the order the chapters added them: `exit` 1, `write` 2, `yield` 3, `sbrk` 4, `read` 5, `close` 6, `open` 7, `spawn` 8, `wait` 9. A stub, not a library — nothing here formats, allocates or buffers, and the three programs are written against that. `_start` sits in `.text`. `start` so it is the first byte of the segment, which is the byte `e_entry` names; exercise (c) is what happens if either of its duties is skipped.

wait, and the leak closes

```
/* The child's pages come back: every U page from its base to its
* break, unmapped and freed; its kernel stack; its slot. Chapter 9
* named this leak; a parent that waits is who closes it. */
static void reap(struct proc *c)
{
    unsigned long va;

    for (va = c->base; va < c->brk; va += PAGE_SIZE) {
        unsigned long *slot = walk(va);

        if (slot && (*slot & PTE_V)) {
            kfree_page((*slot >> 10) << PAGE_SHIFT);
            *slot = 0;
        }
    }
    __asm__ volatile("sfence.vma" ::: "memory");
    kfree_page(c->kstack_top - PAGE_SIZE);
    c->state = UNUSED;
    c->parent = 0;
    uart_puts("S: reap proc ");
    uart_putc((char)('0' + c->pid));
    uart_puts("\r\n");
}

long wait_impl(long pid)
{
    struct proc *c;

    if (pid < 1 || pid > NPROC)
        return -1;
    c = &proc_tab[pid - 1];
    if (c->state == UNUSED || c->parent != curproc)
        return -1; /* not a child of the caller */
    while (c->state != DEAD)
        sleep_on(c, "child"); /* exit wakes on the child's own address
    ↳ */
```



```

    reap(c);
    return pid;
}

```

`wait(pid)` is chapter 10's `sleep_on` with the child's own address as the channel; `exit` calls `wakeup` on that address after marking the child `DEAD`. When the parent wakes it *reaps*: every user page from the child's base to its break, unmapped through the frozen walk and returned to the freelist; the child's kernel stack; the slot. Chapter 9 named this leak — a process that exits leaves its pages mapped in a `DEAD` slot — and said that a parent that waits is who closes it. Only a parent may wait, and only for its own child; the kernel says `S: reap proc N` when it does.

The programs

```

int main(void)
{
    long sh;

    puts_("init: up\r\n");
    sh = spawn("sh", 0);
    if (sh < 0) {
        puts_("init: no sh\r\n");
        return 1;
    }
    wait(sh);
    return 0;
}

int main(void)
{
    char line[64];

    puts_("sh: up\r\n");
    for (;;) {
        long n = read(0, line, sizeof line - 1);
        char *a = 0;
        long i, child;

        if (n <= 0)
            return 0; /* EOF on the console */
        if (line[n - 1] == '\n')
            n--;
        line[n] = 0;
        puts_("$ ");
    }
}

```

```

puts_(line);
puts_("\r\n");
for (i = 0; i < n; i++)
    if (line[i] == ' ') {
        line[i] = 0;
        a = &line[i + 1];
        break;
    }
if (line[0] == 0)
    continue;
if (streq(line, "exit"))
    return 0;
child = spawn(line, a);
if (child < 0) {
    puts_("sh: cannot spawn ");
    puts_(line);
    puts_("\r\n");
    continue;
}
wait(child);
}
}

```

init spawns the shell and waits for it; when the shell exits, init exits, the table is empty, and the boot stage's scheduler() returns — to a wait on init itself, since the boot stage is init's parent, and then to PASS. The shell is the userland's scheduler of one: read a line from fd 0, print it after a prompt, split it at the first space into a command and an argument, spawn the command, wait for it, loop. exit is the one built-in. cat opens its argument, copies it to fd 1 in 64-byte reads until read returns 0, and returns — a program that would work unchanged on a pipe end, which is exercise (d).

```

mem_init();
trap_init();
uart_puts("S: kernel up\r\n");
kalloc_init();
blk_init();
fs_init();
uart_irq_on();           /* the console: chapter 5's path, chapter 10's
↳ buffer */
console_wait_line(2);    /* the lab's script is two lines; see
↳ console.c */
if (!spawn_path("init", 0))
    halt(1);             /* no init on the disk: nothing to run */
scheduler();             /* returns when init has exited */

```

```

wait_impl(1);           /* init's parent is the boot stage: reap it
↳  too */
{
    int i;
    for (i = 0; i < NPROC; i++)
        if (proc_tab[i].state == WAITING) {
            uart_puts("S: deadlock: proc ");
            uart_putc((char)('0' + proc_tab[i].pid));
            uart_puts(" still waiting\r\n");
            halt(1);
        }
}
uart_puts("PASS\r\n");
halt(0);

```

The boot stage mounts the disk, brings up the console, and spawns `init` by name through the same `spawn_path` a program would use. Nothing is embedded; the kernel image carries no user code from this chapter on. The console wait is two lines because the lab's script is two lines — the same determinism argument as chapter 10, and the same admission that a human at the console would block in the shell's read instead.

Walkthrough

ch13-main (mount, console, init by name, the loop, the deadlock check), ch13-spawn (file to process: `fs_open`, `fs_read`, the check, launch, the argument, the parent), ch13-elf-load (the fields and every rejection), ch13-wait (sleep on the child, reap), ch13-libc-start (the two registers, `.bss`, `sp`, `main`, `exit`), ch13-init, ch13-sh.

Exercises

(a) Type `hello.txt` at the shell — spawn a text file. Predict which check rejects it, then run it.

Answer, p. 161

(b) `readelf -l` on the shell shows `FileSiz 0x30a` and `MemSiz 0x320`. Derive what the 22 bytes are and who makes them zero.

Answer, p. 161

(c) What must the stub's `_start` do before it calls anything, and what fails if it skips either?

Answer, p. 161

(d) `cat` reads a descriptor from `open` and writes to `fd 1`. What would change if the descriptor were a pipe end instead of a file?

Answer, p. 162

(e) Strip the userland: on paper, what does the Part VI cut keep and what does it delete?

Answer, p. 162

(f) Count the syscalls in this ABI and set the number beside Linux's.

Answer, p. 162

Lab

```
make lab CH=13
make lab CH=13 IMPL=mime
```

`input.txt` is the shell session: `cat hello.txt`, then `exit`. `extra.mk` builds the disk from the reference `init`, `sh` and `cat` — stripped ELF's from `ember/user/build/fs` — and `files/hello.txt`, regenerated before every run.

After the mount: the two lines echoed as they arrive; S: console 0x2 lines ready; S: fs: open init inode 0x2 size 0x3f0 and S: spawn init pid 0x1 entry 0x10000, the boot stage loading the first program by name; init: up; the same pair for `sh` at 0x20000; S: proc 1 blocks on child, init in wait; sh: up; \$ `cat hello.txt`, the shell showing what it parsed; `cat` opened, checked, spawned at 0x30000, the shell blocking on it; S: fs: open `hello.txt` inode 0x5 size 0x14, `cat`'s own open; the file's bytes; S: proc 3 exited, S: wake proc 2, S: reap proc 3; \$ `exit`; the shell exits, init wakes and reaps it, init exits; S: reap proc 1 — init has no parent process, so the boot stage, which spawned it, is the one that waits for it and gives its pages back; PASS. The three file sizes are the stripped binaries' sizes on this toolchain; the reader's `mime` runs the same binaries, so its transcript has the same numbers.

One ordering rule in this stage is load-bearing. The block device is brought up before the console, and `blk_init` enables `sstatus.SIE`; a console drain that merely ordered “drain, then unmask” would then run with the UART's line live, the handler would nest inside the drain loop, and both loops would echo and buffer the same bytes. Chapter 5's drain clears `sstatus.SIE` itself for exactly

this reason. Interrupt enable is global, and a driver that turns it on changes every later assumption about it.

IMPL=mime copies exactly the objects chapter 13 links and stubs `spawn_path` to print MINE: `spawn_path` stub for `init` and `halt`: the mount succeeds and the first program never loads.

Further reading

- System V Application Binary Interface — ELF Header and Program Header (gABI 4.1+, Linux Foundation referenced specifications) — `Elf64_Ehdr`, `Elf64_Phdr`, `ET_EXEC`, `PT_LOAD`, the magic bytes; every field the loader reads.
- `xv6-riscv` (MIT PDOS) — `kernel/exec.c` is the sibling that *replaces*: the same header checks, then a new page table swapped for the old, which is what `per-process` tables buy and `ember` does without.
- `pipe(2)` — Linux manual page (man-pages project) — for exercise (d), the contract a pipe end keeps that lets `cat` not care.

Answers

(a) The first one that can. `hello.txt` is 20 bytes, an ELF header is 64, and the loader checks the size before it looks at a single byte of the header: `S: spawn: file shorter than a header`, then the shell's `sh: cannot spawn hello.txt`. Observed. A longer text file would fall to `bad magic`; a real ELF for the wrong machine to not a RISC-V executable; a file whose program-header offset points past its end to program headers out of range. Each rejection is a read or a copy the kernel would otherwise have done on the file's say-so, and the order of the checks is the order of the damage they prevent.

(b) They are the shell's `.bss` — `pid` and `arg` from the stub, eight bytes each, plus alignment — variables that must start at zero and therefore need no bytes in the file. The gABI says the loader supplies `p_memsz - p_filesz` zero bytes after the file's bytes. This loader does not, explicitly: it copies `p_filesz` bytes into a page that `kalloc_page` zeroed, and the remaining 22 bytes of that page are zero because every byte of it was. Then `_start` zeroes `__bss_start` to `__bss_end` a second time, which on this kernel is redundant and on a kernel that reused pages without zeroing would be the only thing standing between the program and someone else's old data. Two mechanisms, either of which suffices; the one in the program is the one the program can rely on.

(c) Two things, in this order: capture what is in the registers the kernel set, and set up the world C expects. The frame's `a1` and `a2` exist only until the first instruction that clobbers them, so `_start` reads them into registers the compiler is told to keep (`register long a1r asm("a1")`) before anything else. Then it zeroes `.bss` — but not before the captures, because `pid` and `arg` *are* in `.bss` and would be wiped — and sets `sp` to `stack_top`, because the fabricated frame's `sp` is the kernel's guess and the link script's symbol is the program's. Skip the captures and the shell has no `pid` to print and `cat` has no argument. Skip the `.bss` zeroing and on this kernel nothing changes, which is exercise (b)'s point. Skip `sp` and the first `call` pushes onto whatever the frame said, which here happens to be the same page top — chapter 7 chose them to agree — and on any other kernel would be a fault at the first push.

(d) Nothing in `cat`. It calls `read` until it returns 0 and writes what it got; a pipe's `read` returns bytes while a writer is open and 0 when none is, which is the same contract a file's `read` keeps at end of file, and `close` on either drops a reference. The only thing this kernel lacks is a way to *hand* a pipe end to a spawned process — `spawn` attaches the console and nothing else, so `sh | cat` would need `spawn` to install a caller's descriptor in the child, the way `posix_spawn`'s file actions do. The uniformity is chapter 10's descriptor table earning its keep: three kinds behind three calls, and a program that cannot tell them apart.

(e) Deleted: the three programs as separate address spaces, the stub, the `ecall` and everything on its path — `user_vec`, the frame save, the syscall table, `copy_in_user` and `copy_out_user` — and the U bit as a boundary, since there is one privilege level left. `spawn`, `wait`, the per-process kernel stacks and the scheduler go with them, or shrink to threads. Kept: the allocator, the file system and its cache, the block driver and the ring, the console's interrupt path, the page table as a map of the machine rather than as a wall between programs, and the M-mode shim underneath it all. What was `cat hello.txt` becomes a function that calls `fs_open` and `fs_read` directly and prints through `uart_putc`. The image is smaller and every bug in the application is now a bug in the kernel; chapter 32 fixes what such an image may assume of its host, and chapter 40 is the list of what it refuses.

(f) Nine: `exit`, `write`, `yield`, `sbrk`, `read`, `close`, `open`, `spawn`, `wait`. The riscv64 Linux table, which uses the generic numbering, declares `__NR_syscalls` as 472 in this host's `asm-generic/unistd.h`. The contrast is named and not surveyed: the gap is not nine things done badly against 472 done well, it is nine things a shell and a `cat` need against everything thirty years of programs have asked for, and the ABI is the part of a kernel that can never be taken back. A

unikernel's answer to the 472 is to have none — which is why chapter 32's contract is a boot-info block and not a syscall table, and why an image with a syscall table is wick, not hearth.

PART III — THE SHAPES OF AN OPERATING SYSTEM

14. MONOLITH, MICROKERNEL, EXOKERNEL, LIBRARY OS

WHAT IS SHARED	WHO IS THE TCB	WHERE THE COPY IS
MONOLITHLinux;ember is here, with a U bit		
one kernel space, every subsystem in it	the whole kernel: drivers, fs, net	by default at the syscall boundary, user to kernel
MICROKERNELseL4		
nothing by default; a space per server	the kernel alone; servers sit outside	an IPC message, through the kernel
EXOKERNELAegis, 1995		
raw resources, split by secure binding	the thin multiplexer; the libOS is yours	none in the kernel; the libOS decides
LIBRARY OSUnikraft;hearth is one, hosting these		
one address space, everything in it	the whole image, app and libraries	none: a call is a function call

Each shape spends the same three things; the rows differ in where the cost lands.

Why this layer exists

The reader has built a kernel and is about to build two more objects: a hypervisor in Part VII and, in Part VI, the unikernels it hosts. These are not history. This

chapter names four shapes an operating system can take, and three of the objects in this book are placed among them: the kernel the reader owns, the hypervisor and the VMM. A reader who can place each on the same three axes can say what an isolation claim in any later chapter actually costs.

Every design spends the same three things, which this book calls the *isolation budget*. **What is shared:** the address space, the page table, the kernel objects two programs can both name. **Who is trusted:** the code whose bug breaks the isolation, the *trusted computing base* (TCB) of that isolation. **Where the copy is:** the boundary a byte crosses when it moves from one program to another. A design cannot spend none of them. Sharing less costs a copy; trusting less costs an interface that has to be narrow enough to check; removing the copy costs a shared page and therefore a bigger TCB. The shapes are four different ways of making that trade, and chapter 41 comes back to the same three axes to place the isolation mechanisms of Part VIII.

Mechanism

The three axes, from what Part II built

The axes are not new vocabulary; ember already has a worked example of each.

Shared. Chapter 7 put the kernel and every process into one page table. What separates them is a bit in the PTEs, not a second table, so what two ember processes share is *everything the U bit does not hide*: the kernel's identity map is there for both, and each other's user pages are there too, mapped $U=1$, reachable by a stray address. Chapter 10 is the opposite choice in miniature: a pipe is a kernel page that neither process maps. The processes share a kernel object (the open-file table) and no memory at all.

Trusted. Chapter 7 reduced isolation to two objects: the U bit in the page table and the privilege mode that `sret` restores. The TCB of ember's isolation is therefore every line that can set either one wrongly: every `map_page` call that passes `PTE_U`, and every path that can write `sstatus.SPP` or execute `sret`. That is not small: it includes `sbrk`, which maps user pages as the heap grows, and the loader that fabricates a frame. But it is countable, and the count is what the word means. A bug anywhere else in ember / is a crash; a bug there is an escape.

The copy. Chapter 10's `copy_in_user` and `copy_out_user` are the boundary made into code: the kernel walks the page table itself, finds the physical page behind a user address, and moves the bytes through its own mapping, refusing any address that is not a mapped U page. Chapter 13's `spawn` argument is two

such copies in a row: the parent's string comes in through `copy_in_user`, and the kernel then writes it into the child's stack page, whose address the child finds in `a2`. No byte reaches the child without the kernel handling it.

Four shapes

A **monolith** is a kernel in which every service (scheduler, file system, network stack, drivers) runs in one address space at the highest privilege, so a call from one service to another is a function call. Linux is the shape's working example, and *ember* is a small one. By default the copy is at the syscall boundary: a program's `ecall` enters the kernel at one door and the bytes that cross it are copied. The TCB is the whole kernel, including every loaded driver. By default *what is shared*, between two programs, is the kernel and nothing else: each has its own user address space over one shared kernel. A program can spend on the shared axis when it asks the kernel to map a page into both address spaces (`mmap` with `MAP_SHARED`, POSIX or System V shared memory). Such a page is shared memory, and bytes written to it cross no copy, exactly the trade chapter 10's exercise (c) makes. `fork` shares differently: it maps every writable page *copy-on-write*, read-only in both processes until one of them stores to it and is given a private copy of that page; chapter 16 builds that. Chapter 15 reads Linux from inside a boot of it.

A **microkernel** is a kernel reduced to what cannot be done outside it. Liedtke's criterion, in *On μ -Kernel Construction* (1995), is that a concept is tolerated inside the kernel only if moving it outside, permitting competing implementations, would prevent the implementation of the system's required functionality. File systems, drivers and network stacks become *servers*: ordinary programs, each in its own address space. *seL4* is the shape's working example. A program that wants a file sends a message to the file server, and the copy is that message: the kernel moves a few words from the sender to the receiver (a comment in *seL4*'s `copyMRs` calls them "IPC MRs"; `libseL4/include/seL4/syscalls.h` spells MR out as *message register*). The kernel is the TCB of the isolation between two programs; a server is trusted only by the programs that depend on it. A *capability* is an unforgeable reference to a kernel object that carries the right to use it; *seL4* uses one where *ember* uses a descriptor, for endpoints, address spaces and the rest. Chapter 17 reads them.

An **exokernel** inverts the microkernel's question. Engler, Kaashoek and O'Toole (1995) describe the architecture as a thin kernel that *multiplexes* physical resources securely and leaves the *abstractions* to the application's own library. A **library OS** is that library: the code that would be a file system or a network stack in a monolith, linked into the program and running in its address space with no

special privilege. The mechanism that lets an untrusted library hold a resource is the **secure binding**: in the paper’s words, a protection mechanism that decouples authorization from the actual use of a resource, performing the authorization only at bind time. The kernel checks once that a program may have a physical page, a disk block or a TLB entry, then gets out of the way. The exokernel’s TCB is the multiplexer, which is small because it manages nothing: it tracks ownership and checks bindings. The kernel has no copy of its own; what crosses between two programs is whatever their libraries arrange.

A **unikernel** is the library OS with the kernel deleted. Madhavapeddy and colleagues (2013) describe unikernels as specialised, sealed, single-purpose library OS VMs that run directly on the hypervisor, all services from the scheduler to the device drivers to the network stack implemented as libraries linked with the application. There is one address space. There is no copy, because there is no boundary inside the image: a call from the application to the network stack is a function call. The price is the TCB: the application, every library and every driver are all one image, and a bug in any of them is a bug in all of them. Isolation moves outside the image, to whatever hosts it. Chapter 29 builds the idea out; chapter 30 builds and reads a real one.

Placing ember and hearth

After chapter 13, *ember* is a monolith with a U bit: one kernel address space, one page table shared by kernel and processes, the U bit and the mode bit as the whole of the isolation, and the copy at the syscall boundary. It has a boundary and little else, which is why it teaches the boundary: there is no per-process table to hide the mechanism.

hearth is a library OS whose “applications” are library OSes. It is one image that owns the machine, with no syscall table, and what it runs are unikernel guests. Between a guest and the machine sits a second translation, the G-stage table that chapter 19 builds, and it is the object that makes *hearth*’s copy line: a guest’s physical address means nothing until the hypervisor has translated it or refused. That is one paragraph of placement; the mechanism is Part IV and Part VII and none of it is used here.

What the budget says about the two floors

wick inherits Linux’s spend. It is a process on a Linux host, so the host kernel is in its TCB (as it is in *ward*’s, which jails it from a program on the same kernel), and whatever it hands a guest passes through that kernel’s syscall boundary. *hearth*

spends nothing it did not write. There is no host kernel in its TCB and no syscall table on its boundary; every line in the TCB is a line the reader wrote in Part VII. This is what “no Linux in the trusted computing base” means, and the two floors of the book are the two columns of that sentence.

Walkthrough

There is no listing. The walkthrough is five kernels on the three axes, three answers each (**Shared**, **TCB**, **Copy**), and for each answer the file at its pin that justifies it. ember paths are in this tree; xv6, Linux and seL4 are at the pins in labs/ch14/README.md. The Linux files are named, not quoted: chapter 15 reads most of them.

ember (this tree)

- **Shared:** one page table for the kernel and every process; the U bit tells them apart (ember/proc.h, ember/proc.c).
- **TCB:** all of ember/; in particular every PTE_U mapping (ember/proc.c, ember/syscall.c’s sbrk) and the path to sret (ember/proc.S, ember/trap.S).
- **Copy:** copy_in_user and copy_out_user in ember/copyio.c.

xv6-riscv

- **Shared:** the kernel; each process has its own user page table (kernel/vm.c: kvmmake, uvmmcreate).
- **TCB:** the whole kernel; kernel/syscall.c is the table.
- **Copy:** copyin and copyout in kernel/vm.c.

Linux

- **Shared:** one kernel address space under each process’s own user space (kernel/fork.c).
- **TCB:** the whole kernel and every loaded module.
- **Copy:** the syscall boundary, entered in arch/riscv/kernel/entry.S; the bytes cross in copy_from_user and copy_to_user (include/linux/uaccess.h).

seL4

- **Shared:** nothing by default; an address space is an object a program holds (src/arch/riscv/kernel/vspace.c).
- **TCB:** the kernel; servers are programs (src/object/endpoint.c).

- **Copy:** an IPC message: `sendIPC (src/object/endpoint.c)` calls `doIPCtransfer (src/kernel/thread.c)`, which moves the words with `copyMRs (src/object/tcb.c)`.

Unikraft

- **Shared:** one address space, application and libraries linked together.
- **TCB:** the whole image.
- **Copy:** none: a call is a function call (read in chapter 30).

The Unikraft answers are placed from the unikernel definition above (Madhavapeddy et al.), not from a checkout; chapter 30 reads the tree.

Exercises

(a) Place `ember` after chapter 13 on the three axes. Name the file behind each answer.

Answer, p. 173

(b) Which of the three axes does chapter 10's pipe spend, and which would a shared-memory pipe spend instead?

Answer, p. 173

(c) State an exokernel's *secure binding* in Engler's terms, and name the Part II object that `ember` has closest to one.

Answer, p. 174

(d) Why is a microkernel's IPC copy the same object as a monolith's syscall copy, and what differs?

Answer, p. 174

(e) A unikernel's TCB is the whole image. State what that buys and what it costs, in the budget's terms, using chapter 13's exercise (e) as the worked case.

Answer, p. 174

(f) Place `hearth` on the three axes from the spec alone, and say which axis Part IV's nested paging changes.

Answer, p. 175

Lab

A reading lab: nothing boots. For each of `ember`, `xv6`, `Linux`, `seL4` and `Unikraft`, write three one-line answers, **Shared** (what two programs share), **TCB** (which code is trusted for the isolation between them) and **Copy** (where a byte crosses from one to the other), each with the file or line that justifies it. `labs/ch14/README.md` lists the sources and their pins. The Walkthrough's lists are the reference; an answer that disagrees with one is either a mistake or a better reading, and the file named in the answer says which.

Further reading

- Exokernel: An Operating System Architecture for Application-Level Resource Management, Engler, Kaashoek and O'Toole, SOSP 1995: secure bindings, visible revocation and the abort protocol; Aegis is the kernel, ExOS the library OS.
- On μ -Kernel Construction, Liedtke, SOSP 1995: the minimality criterion quoted above, and the IPC costs that made it viable.
- Unikernels: Library Operating Systems for the Cloud, Madhavapeddy et al., ASPLOS 2013: the library OS as a sealed single-purpose VM image on a hypervisor.
- The living systems are read where the book builds on them: Unikraft in chapter 30, the contrast with POSIX-shaped unikernels in chapter 31, and the library OS idea in chapter 29.

Answers

(a) The Walkthrough's `ember` list. *Shared*: one page table for kernel and processes, stated in the header comment of `ember/proc.h` ("One page table holds everything... what separates the two worlds is the U bit in the PTEs and the mode bit `sret` sets") and built by `proc_create` in `ember/proc.c`. *TCB*: the sites a `grep` for `PTE_U` finds that map a user page, which are `proc_create` (code and stack), `sbrk` in `ember/syscall.c`, and the one fixed page `mem_init` maps in `ember/mem.c`; and the writes that reach the privilege the next `sret` restores: the frame `proc_create` fabricates with `SPP=0`, the `csrwr sstatus` in `ember/proc.S`, and the `sret` in `ember/trap.S`. *Copy*: `copy_in_user` and `copy_out_user` in `ember/copyio.c`, reached from `ember/syscall.c`, `ember/pipe.c`, `ember/console.c` and `ember/fs.c`. Everything else in `ember/` is outside the isolation claim: a bug in the allocator crashes the machine, a bug in a `PTE_U` mapping lets a program read another's pages.

(b) The pipe spends *the copy*. `ember/pipe.c`'s header says it in so many words: neither process maps the page, the writer's bytes come in through `copy_in_user` and the reader's leave through `copy_out_user`, and "the two processes share nothing". The kernel's ring is shared only in the sense that the kernel holds it. A pipe page mapped into both processes with `U=1` (chapter 10's exercise (c)) spends *shared* instead and gets the copy back for free. The price is on the TCB axis too: each process is now in the other's TCB, because either can move the cursors, write past the ring, or hand the other a bad pointer, and the kernel has nothing left to refuse because nothing crosses it.

(c) Engler, Kaashoek and O'Toole define a secure binding as a protection mechanism that decouples authorization from the actual use of a resource, and they say it performs authorization only at bind time. The kernel checks once that an application may have a resource (they name physical pages and TLB entries), and later uses need no check. The closest `ember` has is the file descriptor of chapter 10: `open` checks the name and allocates a slot in the kernel's open-file table, the `struct file` in `ember/fd.h` records `readable` and `writable` when it is allocated, and what the program holds afterwards is a small integer that only indexes its own table. The resemblance stops at the resource. A secure binding is to a hardware resource (a page, a block); an `ember` descriptor is to a kernel abstraction (a console, a pipe, a file), and the kernel still copies every byte through that abstraction on every use. An exokernel hands out the page and leaves the abstraction to the library.

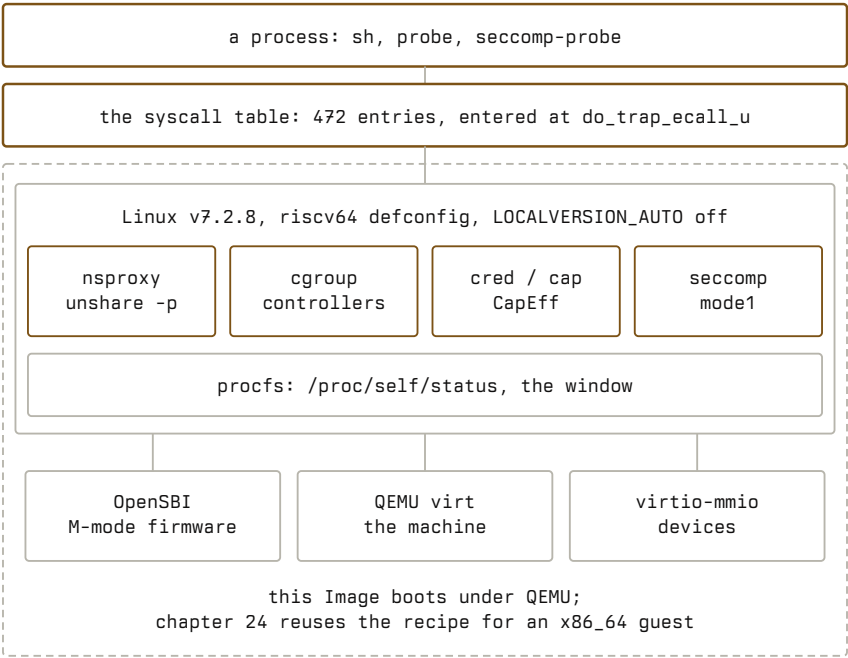
(d) Both are the kernel moving bytes between two address spaces so that neither side ever dereferences a pointer into the other's. `ember`'s `copy_in_user` moves them through the kernel's own mapping of the user's page; `seL4`'s `copyMRs` moves them from the sender's message registers to the receiver's (the call chain is the Walkthrough's `seL4 Copy` answer). What differs is who is on the other side. In a monolith the far end of a copy is the kernel, whose code is the TCB. In a microkernel the far end is another program, a server, and the kernel is only the courier. So the same boundary now separates two things outside the TCB, and the TCB shrinks to the courier, at the cost of a message for every service call that would have been a function call.

(e) Chapter 13's exercise (e) strips `ember` on paper: delete the programs as separate address spaces, the stub, the `ecall` path (the frame save, the syscall table, `copy_in_user` and `copy_out_user`) and the `U` bit as a boundary, and keep the allocator, file system, block driver and console. `cat hello.txt` becomes a function that calls `fs_open` and `fs_read` directly. On the budget it lands where the unikernel paragraph above puts it: no copy, everything shared, the whole image trusted.

It buys the removal of every cost the boundary had: no trap, no frame save, no copy, no table, and a smaller image. It costs the boundary's protection: every bug in the application is now a bug in the kernel. The isolation moves to whatever hosts the image. This is the exokernel's *library OS* with the kernel's last duty, multiplexing, handed to a hypervisor.

(f) The spec (docs/superpowers/specs/2026-09-29-hp-course-design.md) calls *hearth* a type-1 hypervisor that is itself a unikernel: "one address space, no host kernel, no syscall table", entering a unikernel guest, and says chapter 32 boots the *ember* image "with its userland stripped as a guest". It puts nested paging in chapter 19 as the lab that is *hearth*'s chapter zero. On the axes: *Shared*: nothing between guests; each sees only what the hypervisor maps into its G-stage table. *TCB*: *hearth*'s own image and nothing under it: no host kernel, which is the difference from *wick*. *Copy*: the G-stage translation, as placed above (chapter 11 already notes that a guest's ring holds guest-physical addresses the hypervisor must translate or refuse). Nested paging changes the *shared* axis: chapter 4 says it is two walks with two owners, so the page table stops being one object the trusted code owns alone and becomes two, the guest's own and the hypervisor's. The U bit and the mode bit of chapter 7 are the same two objects with new names; the copy line is not a second change, it is where the second table puts the boundary.

15. WHAT LINUX ACTUALLY IS FROM HERE



The table and the four hooks are the five objects; procfs is the window the probe reads them through.

Why this layer exists

wick and ward run on Linux. Chapter 24 boots a Linux kernel as the guest of wick. The KVM hosts are x86_64, so that chapter reuses this chapter’s recipe (the pins, defconfig, a BusyBox initramfs) for an x86_64 guest; the riscv64 Image built here boots under QEMU. Chapter 26’s jailer is made of the objects this chapter reads. A reader who has written a kernel with nine syscalls and has to put a second, much larger kernel in the trusted computing base of the second floor should know what that kernel’s isolation is made of, and know it from its

source, not from a container tutorial.

Linux's riscv64 syscall table is 472 entries long (`__NR_syscalls`), the number chapter 13 took from this host's headers; here it comes with its source. It has the same three axes chapter 14 defined and a great deal more machinery on each. Five of its objects are the ones wick's host stands on: the syscall table, namespaces, cgroups, capabilities and seccomp. The chapter reads each in the kernel source at tag `v7.2.8` while the kernel runs in QEMU next to the tree, and reads a line the running kernel prints for each. You do not implement any of them. The lab is a probe: a shell script, a BusyBox initramfs and a short C program that asks the kernel five questions, and the capture pins its answers.

Mechanism

Every quoted block below is headed with its file and line range at `v7.2.8`, copied from the pinned checkout the build fetched, not retyped. The kernel is configured by riscv64 `defconfig` and built with the cross toolchain `make doctor` checks; `labs/ch15/build.sh` records the pins.

The syscall table, grown up

Chapter 13's table is a switch on `a7` in `ember/syscall.c`. Linux's has the same shape with more around it. The first is how a user `ecall` reaches C. The trap vector saves the frame and indexes a table of handlers by `scause`; `cause 8`, an `ecall` from U mode, is the ninth entry:

```
// linux v7.2.8, arch/riscv/kernel/entry.S:485-494
SYM_DATA_START_LOCAL(excp_vect_table)
    RISCVPTR do_trap_insn_misaligned
    ALT_INSN_FAULT(RISCV_PTR do_trap_insn_fault)
    RISCVPTR do_trap_insn_illegal
    RISCVPTR do_trap_break
    RISCVPTR do_trap_load_misaligned
    RISCVPTR do_trap_load_fault
    RISCVPTR do_trap_store_misaligned
    RISCVPTR do_trap_store_fault
    RISCVPTR do_trap_ecall_u /* system call */
```

The handler is the function that chapter 13's `syscall()` grew into:

```
// linux v7.2.8, arch/riscv/kernel/traps.c:324-345
asmlinkage __visible __trap_section __no_stack_protector
void do_trap_ecall_u(struct pt_regs *regs)
{
```

```

if (user_mode(regs)) {
    long syscall = regs->a7;

    regs->epc += 4;
    regs->orig_a0 = regs->a0;
    regs->a0 = -ENOSYS;

    riscv_v_vstate_discard(regs);

    syscall = syscall_enter_from_user_mode(regs, syscall);

    add_random_kstack_offset();

    if (syscall >= 0 && syscall < NR_syscalls) {
        syscall = array_index_nospec(syscall,
        NR_syscalls);
        syscall_handler(regs, syscall);
    }

    syscall_exit_to_user_mode(regs);

```

The moves are ember's: step epc over the ecall (ember does it at the end of `syscall()`, `tf->sepc += 4`), take the number from `a7`, store `-ENOSYS` in `a0` as the answer to a number that is not in the table, and dispatch. Two things are new. The bound is `NR_syscalls`, and the index is passed through `array_index_nospec` so that a mispredicted branch cannot speculatively index past the table. And `syscall_enter_from_user_mode` runs before the dispatch. It is the one place where everything that wants to veto or observe a syscall gets its turn: `ptrace` stops there and `seccomp` runs there (the `seccomp` section follows the call down). The table's size is a constant in the generic header:

```

// linux v7.2.8, include/uapi/asm-generic/unistd.h:866-867
#undef __NR_syscalls
#define __NR_syscalls 472

```

472 is the table's length, the last number plus one. On chapter 14's axes this is the copy, the boundary every byte crosses at the same door, now with a hook on the way in.

PID 1 and /proc

The probe runs as a child of PID 1. The kernel runs whatever `rdinit=` names as the first user process, and `/init` in the `initramfs` is a shell script, so `/proc/1/comm` is `init`, the basename of the file the kernel `execed`. `/proc` is a file system

the kernel synthesizes: reading `/proc/self/status` calls a function that prints the fields of the running task's kernel structures into a buffer. The lines the probe keeps come from two helpers in `fs/proc/array.c`. The capability masks are printed by `render_cap_t` through `task_cap`; the seccomp state by `task_seccomp`:

```
// linux v7.2.8, fs/proc/array.c:302-308
static void render_cap_t(struct seq_file *m, const char *header,
                        kernel_cap_t *a)
{
    seq_puts(m, header);
    seq_put_hex_ll(m, NULL, a->val, 16);
    seq_putc(m, '\n');
}

// linux v7.2.8, fs/proc/array.c:325-329
render_cap_t(m, "CapInh:\t", &cap_inheritable);
render_cap_t(m, "CapPrm:\t", &cap_permitted);
render_cap_t(m, "CapEff:\t", &cap_effective);
render_cap_t(m, "CapBnd:\t", &cap_bset);
render_cap_t(m, "CapAmb:\t", &cap_ambient);

// linux v7.2.8, fs/proc/array.c:332-336
static inline void task_seccomp(struct seq_file *m, struct task_struct
    ↪ *p)
{
    seq_put_decimal_ull(m, "NoNewPrivs:\t", task_no_new_privs(p));
#ifdef CONFIG_SECCOMP
    seq_put_decimal_ull(m, "\nSeccomp:\t", p->seccomp.mode);

```

The `\t` in each header is why the capture has real tab characters after the colons. `Seccomp_filters:` follows on the next lines; the probe's regex, anchored with the colon directly after `Seccomp`, does not match it. `/proc` is not an isolation mechanism. It is the observation window onto the four that are, and on chapter 14's axes it adds no boundary: it is a read-only window through which a program sees the kernel's state.

Namespaces

A **namespace** wraps one global resource of the kernel, such as the set of process ids, the mount table or the network stack, so that the processes inside see their own instance of it (the wording is namespaces(7)'s). A task's view of most of these hangs off one pointer, `task->nsproxy`:

```
// linux v7.2.8, include/linux/nsproxy.h:32-41

```



```

struct nsproxy {
    refcount_t count;
    struct uts_namespace *uts_ns;
    struct ipc_namespace *ipc_ns;
    struct mnt_namespace *mnt_ns;
    struct pid_namespace *pid_ns_for_children;
    struct net                *net_ns;
    struct time_namespace *time_ns;
    struct time_namespace *time_ns_for_children;
    struct cgroup_namespace *cgroup_ns;

```

Two entries are the namespaces that the task's *future children* will be born into, `pid_ns_for_children` and `time_ns_for_children`, and they differ from the task's own once someone has asked for a new one. For `pid`, the task's own namespace is reached through `task_active_pid_ns` (which `fs/proc/root.c` calls below), not through this struct. The user namespace, which owns the others, is not here either; it is in the task's credentials, which is where the capability section looks.

The probe lists `/proc/self/ns`, which `fs/proc/namespaces.c` generates from a table gated by the `CONFIG_*_NS` symbols that `defconfig` turns on (all =y, below):

```

// linux v7.2.8, fs/proc/namespaces.c:15-40
static const struct proc_ns_operations *const ns_entries[] = {
#ifdef CONFIG_NET_NS
    &netns_operations,
#endif
#ifdef CONFIG_UTS_NS
    &utsns_operations,
#endif
#ifdef CONFIG_IPC_NS
    &ipcns_operations,
#endif
#ifdef CONFIG_PID_NS
    &pidns_operations,
    &pidns_for_children_operations,
#endif
#ifdef CONFIG_USER_NS
    &usersns_operations,
#endif
    &mntns_operations,
#ifdef CONFIG_CGROUPS
    &cgroupns_operations,
#endif

```

```

#ifdef CONFIG_TIME_NS
    &timens_operations,
    &timens_for_children_operations,
#endif
};

```

The probe's output, `cgroup ipc mnt net pid pid_for_children time time_for_children user uts`, is that table in alphabetical order, because `ls` sorts. The table has ten rows, `pid` and `time` each contributing two (the task's own and the one its children will get), so the list has ten names from eight mechanisms: `net`, `uts`, `pid`, `user`, `mnt`, `cgroup` and `time`.

A namespace is born in one of two places: `clone(2)`, through `copy_namespaces` when a task is created, or `unshare(2)`, through `unshare_nsproxy_namespaces` when a task changes its own set. Both call `create_new_namespaces`, which builds a new `nsproxy`. The probe's `pid` namespace comes from the second. The `unshare` path checks `CAP_SYS_ADMIN` in the relevant user namespace (the new one if the same call also asks for `CLONE_NEWUSER`) and calls `create_new_namespaces`:

```

// linux v7.2.8, kernel/nsproxy.c:211-235
int unshare_nsproxy_namespaces(unsigned long unshare_flags,
    struct nsproxy **new_nsp, struct cred *new_cred, struct
    ↪ fs_struct *new_fs)
{
    struct user_namespace *user_ns;
    u64 flags = unshare_flags;
    int err = 0;

    if (!(flags & (CLONE_NS_ALL & ~CLONE_NEWUSER)))
        return 0;

    user_ns = new_cred ? new_cred->user_ns : current_user_ns();
    if (!ns_capable(user_ns, CAP_SYS_ADMIN))
        return -EPERM;

    /*
     * Convert the 32-bit UNSHARE_EMPTY_MNTNS (which aliases
     * CLONE_PARENT_SETTID) to the unique 64-bit
    ↪ CLONE_EMPTY_MNTNS.
     */
    if (flags & UNSHARE_EMPTY_MNTNS) {
        flags &= ~(u64)UNSHARE_EMPTY_MNTNS;
        flags |= CLONE_EMPTY_MNTNS;
    }
}

```

```

        *new_nsp = create_new_namespaces(flags, current, user_ns,
                                         new_fs ? new_fs :
↳   current->fs);

```

and `create_new_namespaces` is where the pid namespace for the task's future children is made:

```

// linux v7.2.8, kernel/nsproxy.c:118-119
    new_nsp->pid_ns_for_children =
        copy_pid_ns(flags, user_ns,
↳   tsk->nsproxy->pid_ns_for_children);

```

The clone path is the one every ordinary fork takes. `copy_namespaces` reads the user namespace from the new task's credentials; if the clone flags ask for none of the namespace-creating flags it takes another reference on the parent's nsproxy and returns, which creates nothing, and otherwise it checks `CAP_SYS_ADMIN` in that user namespace (the new task's, which is the new one under `CLONE_NEWUSER`) before building a new set:

```

// linux v7.2.8, kernel/nsproxy.c:169-181
int copy_namespaces(u64 flags, struct task_struct *tsk)
{
    struct nsproxy *old_ns = tsk->nsproxy;
    struct user_namespace *user_ns = task_cred_xxx(tsk, user_ns);
    struct nsproxy *new_ns;

    if (likely(!(flags & (CLONE_NS_ALL & ~CLONE_NEWUSER)))) {
        if ((flags & CLONE_VM) ||
↳   likely(old_ns->time_ns_for_children ==
old_ns->time_ns)) {
            get_nsproxy(old_ns);
            return 0;
        }
    } else if (!ns_capable(user_ns, CAP_SYS_ADMIN))

```

The probe's child `sh` takes the fast path: it is forked with no namespace flags, and inherits the nsproxy that unshare already replaced in its parent.

The probe's unshare `-f -p --mount-proc sh -c ...` needs three facts from the source. `unshare(2)` with `CLONE_NEWPID` does not move the caller: in the manual's words the calling process “is not moved into the new namespace”, and “the first child created by the calling process will have the process ID 1”. That is why the probe passes `-f`: BusyBox's unshare calls `unshare(2)` (`util-linux/unshare.c:305` at BusyBox 1_38_0) and then, because of `-f`, creates a child with

xvfork (line 336; BusyBox's checked vfork, include/libbb.h:1377) before it runs sh, so the sh is the first child. The place that the child's number is decided is fork, which allocates a pid from the parent's pid_ns_for_children, not from its own namespace:

```
// linux v7.2.8, kernel/fork.c:2306-2307
    if (pid != &init_struct_pid) {
        pid = alloc_pid(p->nsproxy->pid_ns_for_children,
↳   args->set_tid,
```

and the allocation loop walks the namespace chain and insists that the first pid handed out in a new namespace is 1:

```
// linux v7.2.8, kernel/pid.c:300-305
    * PID 1 (init) must be created first.
    */
    if (!READ_ONCE(tmp->child_reaper) && nr != 1) {
        retval = -EINVAL;
        goto out_free;
    }
```

So pid inside: 1 is the answer to the question “what number did alloc_pid give the first process in a namespace”, and it is fixed by that check, not by an accident of timing. The third fact explains processes visible inside: 1. /proc lists the pids of one pid namespace, the one its mounter was in, fixed at mount time:

```
// linux v7.2.8, fs/proc/root.c:341-353
static int proc_init_fs_context(struct fs_context *fc)
{
    struct proc_fs_context *ctx;

    ctx = kzalloc_obj(struct proc_fs_context);
    if (!ctx)
        return -ENOMEM;

    ctx->pid_ns = get_pid_ns(task_active_pid_ns(current));
    put_user_ns(fc->user_ns);
    fc->user_ns = get_user_ns(ctx->pid_ns->user_ns);
    fc->fs_private = ctx;
    fc->ops = &proc_fs_context_ops;
```

--mount-proc is the flag that mounts a fresh /proc inside the child, after the child is in the new namespace, so task_active_pid_ns(current) is the new one and /proc/[0-9]* has one entry, the shell. Without the flag the glob would count the host namespace's pids, a number that depends on how many kernel

threads the boot started (Exercise (b) runs it), and the capture would move. The probe counts with the shell's glob so that nothing forks inside the namespace (an `ls` would be a second process, and the count would be two).

A namespace is a change to **what is shared** and nothing else: the kernel, the syscall table and the address-space machinery are unchanged, and what differs is which instance of a named resource a name reaches. Every bug in the kernel still crosses the boundary; isolation by namespace is a statement about naming, not about who is trusted.

Control groups

A **cgroup** is a node in a tree of process groups to which the kernel attaches *controllers*: per-resource accounting and limits, such as memory, cpu time, or the number of processes. Version 2 has a single tree, mounted by the `mount -t cgroup2` in `/init`. The file `cgroup.controllers` at its root lists the controllers that the tree offers; the probe prints it, and the answer is `cpuset cpu io memory hugetlb pids`.

The read is `cgroup_controllers_show`, which prints a mask of controllers from `cgroup_control`:

```
// linux v7.2.8, kernel/cgroup/cgroup.c:465-482
static u32 cgroup_control(struct cgroup *cgrp)
{
    struct cgroup *parent = cgroup_parent(cgrp);
    u32 root_ss_mask = cgrp->root->subsys_mask;

    if (parent) {
        u32 ss_mask = parent->subtree_control;

        /* threaded cgroups can only have threaded controllers
         ↪ */
        if (cgroup_is_threaded(cgrp))
            ss_mask &= cgrp->df1_threaded_ss_mask;
        return ss_mask;
    }

    if (cgroup_on_df1(cgrp))
        root_ss_mask &= ~(cgrp->df1_inhibit_ss_mask |
                        cgrp->df1_implicit_ss_mask);
    return root_ss_mask;
}
```

At the root of the default (v2) hierarchy the mask is everything the kernel was built with, minus two sets: `cgrp->df1_inhibit_ss_mask` and `cgrp->df1_implicit_`

`ss_mask`. The controllers a kernel is built with are a list gated by config symbols, in a header included under a macro that turns each entry into an array slot:

```
// linux v7.2.8, include/linux/cgroup_subsys.h:12-30
#if IS_ENABLED(CONFIG_CPUSETS)
SUBSYS(cpuset)
#endif

#if IS_ENABLED(CONFIG_CGROUP_SCHED)
SUBSYS(cpu)
#endif

#if IS_ENABLED(CONFIG_CGROUP_CPUACCT)
SUBSYS(cpuacct)
#endif

#if IS_ENABLED(CONFIG_BLK_CGROUP)
SUBSYS(io)
#endif

#if IS_ENABLED(CONFIG_MEMCG)
SUBSYS(memory)
#endif
```

The output order is this list's order. Six of the twelve controllers in this `.config` appear; the other six are removed by the two masks, and the loop that fills them is where the reason lives:

```
// linux v7.2.8, kernel/cgroup/cgroup.c:6509-6512
    if (ss->implicit_on_dfl)
        cgrp_dfl_implicit_ss_mask |= 1 << ss->id;
    else if (!ss->dfl_cftypes)
        cgrp_dfl_inhibit_ss_mask |= 1 << ss->id;
```

A controller that has no files for the default hierarchy (`dfl_cftypes` is null) is inhibited there. The probe's missing names are each that case: `cpuacct` (`kernel/sched/cpuacct.c`, which sets only `.legacy_cftypes`), `devices` (`security/device_cgroup.c`, the same), `freezer` (`kernel/cgroup/legacy_freezer.c`), `net_cls` (`net/core/netclassid_cgroup.c`) and `net_prio` (`net/core/netprio_cgroup.c`). A controller marked `implicit_on_dfl` is also not listed, because it is always on and not something to enable: `perf_event` (`kernel/events/core.c`, which sets `.implicit_on_dfl = true`). That accounts for all six: twelve of the sixteen symbols in the header are `=y` in this `.config` (`CGROUP_MISC`, `RDMA`, `DMEM` and `DEBUG` are not), and six of the twelve reach the file. `memory`, `cpu` and `pids` are controllers that bound how much of

the machine a group of processes can use. On chapter 14's axes a cgroup is none of the three; it is a fourth thing, a budget on *use*: it bounds how much of the machine a program can consume and says nothing about what it can touch.

Capabilities

A **capability**, in the POSIX sense Linux implements, is one named piece of the superuser's power (`CAP_SYS_ADMIN`, `CAP_NET_RAW`, ...) that a task holds or does not. It is unrelated to seL4's kind, which is a reference to an object; the word has two meanings and the book uses each in its own chapter. Each task's credentials (`struct cred`) carry several sets of them as 64-bit masks. `CapEff` is the *effective* set, the one that permission checks consult.

`CapEff`: `000001ffffffffffff` is 41 one-bits. The bit count comes from the number of the last capability, and the mask from `CAP_VALID_MASK`:

```
// linux v7.2.8, include/uapi/linux/capability.h:421-423
#define CAP_CHECKPOINT_RESTORE      40

#define CAP_LAST_CAP                  CAP_CHECKPOINT_RESTORE

// linux v7.2.8, include/linux/capability.h:64-67
#define CAP_VALID_MASK                (BIT_ULL(CAP_LAST_CAP+1)-1)

# define CAP_EMPTY_SET                ((kernel_cap_t) { 0 })
# define CAP_FULL_SET                  ((kernel_cap_t) { CAP_VALID_MASK })
```

`CAP_LAST_CAP` is 40, `BIT_ULL(41) - 1` is $2^{41} - 1$, which is `0x1ffffffffffff`, and `render_cap_t` (above) prints it in sixteen hex digits, which gives the two leading zeros the probe shows. `Init` has all of them because the kernel's first task is born with the full set:

```
// linux v7.2.8, init/init_task.c:90-94
    .securebits           = SECUREBITS_DEFAULT,
    .cap_inheritable      = CAP_EMPTY_SET,
    .cap_permitted        = CAP_FULL_SET,
    .cap_effective        = CAP_FULL_SET,
    .cap_bset             = CAP_FULL_SET,
```

and every process the `initramfs` runs is a descendant that is root (`uid 0`) and inherits them: `handle_privileged_root` in `security/commoncap.c` is the step at `execve` that gives a root process its capability sets (Exercise (a) shows a case where it gives none). The check that uses the masks is `cap_capable`, and its helper `cap_capable_helper` walks up the user namespaces from the resource's to the credential's:

```

// linux v7.2.8, security/commoncap.c:68-106
static inline int cap_capable_helper(const struct cred *cred,
                                     struct user_namespace *target_ns,
                                     const struct user_namespace
                                     ↪ *cred_ns,
                                     int cap)
{
    struct user_namespace *ns = target_ns;

    /* See if cred has the capability in the target user namespace
     * by examining the target user namespace and all of the
    ↪ target
        * user namespace's parents.
        */
    for (;;) {
        /* Do we have the necessary capabilities? */
        if (likely(ns == cred_ns))
            return cap_raised(cred->cap_effective, cap) ?
                ↪ 0 : -EPERM;

        /*
        ↪ looking for,
            * If we're already at a lower level than we're
            * we're done searching.
            */
        if (ns->level <= cred_ns->level)
            return -EPERM;

        /*
        ↪ the
            * The owner of the user namespace in the parent of
            * user namespace has all caps.
            */
        if ((ns->parent == cred_ns) && uid_eq(ns->owner,
        ↪ cred->euid))
            return 0;

        /*
        ↪ you have
            * If you have a capability in a parent user ns, then
            * it over all children user namespaces as well.
            */
        ns = ns->parent;
    }
}

```



```

        /* We never get here */
    }

```

The test is per namespace: a task holds a capability *over a user namespace* and, by the last branch, over its descendants. That is why capabilities granted in a new user namespace are worth nothing against the host; Exercise (a) runs `unshare -U` and shows the sets a process ends with. A capability is a change to **who is trusted**: it divides root into parts so that a program can be handed *fewer* of them, which is the direction a jailer moves in.

Seccomp

Seccomp (“secure computing”) is a per-task mode that restricts which syscalls the task may make. The mode is stored in the task (`p->seccomp.mode`, the `Seccomp:` line); the probe prints `0`, meaning disabled, for the shell, and then the C program turns on the simplest mode, *strict*, which is one `prctl` away. `prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT)` reaches `prctl_set_seccomp`, which translates the mode into an operation for `do_seccomp`, which calls `seccomp_set_mode_strict`, which assigns the mode:

```

// linux v7.2.8, kernel/seccomp.c:1428-1441
static long seccomp_set_mode_strict(void)
{
    const unsigned long seccomp_mode = SECCOMP_MODE_STRICT;
    long ret = -EINVAL;

    spin_lock_irq(&current->sighand->siglock);

    if (!seccomp_may_assign_mode(seccomp_mode))
        goto out;

#ifdef TIF_NOTSC
    disable_TSC();
#endif
    seccomp_assign_mode(current, seccomp_mode, 0);
}

```

`seccomp_assign_mode` also sets the task’s `SYSCALL_WORK_SECCOMP` flag, and the check is on the syscall path the trap handler called: `syscall_enter_from_user_mode` calls `syscall_enter_from_user_mode_work`, which calls `syscall_trace_enter` when any entry-work flag is set, and that function calls `__secure_computing` when this one is:

```

// linux v7.2.8, include/linux/entry-common.h:104-109
/* Do seccomp after ptrace, to catch any tracer changes. */

```

```

if (work & SYSCALL_WORK_SECCOMP) {
    ret = __secure_computing();
    if (ret == -1L)
        return ret;
}

```

On riscv64 HAVE_ARCH_SECCOMP_FILTER is selected (arch/riscv/Kconfig), so the function that runs is the second of the two `__secure_computing` definitions in `kernel/seccomp.c`, the one that switches on the mode:

```

// linux v7.2.8, kernel/seccomp.c:1388-1413
int __secure_computing(void)
{
    int mode = current->seccomp.mode;
    int this_syscall;

    if (IS_ENABLED(CONFIG_CHECKPOINT_RESTORE) &&
        unlikely(current->ptrace & PT_SUSPEND_SECCOMP))
        return 0;

    this_syscall = syscall_get_nr(current, current_pt_regs());

    switch (mode) {
    case SECCOMP_MODE_STRICT:
        __secure_computing_strict(this_syscall); /* may call
↪ do_exit */
        return 0;
    case SECCOMP_MODE_FILTER:
        return __seccomp_filter(this_syscall, false);
        /* Surviving SECCOMP_RET_KILL_* must be proactively
↪ impossible. */
    case SECCOMP_MODE_DEAD:
        WARN_ON_ONCE(1);
        do_exit(SIGKILL);
        return -1;
    default:
        BUG();
    }
}

```

Strict mode's whole policy is one table and one function:

```

// linux v7.2.8, kernel/seccomp.c:1051-1065
/*
 * Secure computing mode 1 allows only read/write/exit/sigreturn.
 * To be fully secure this must be combined with rlimit

```

```

* to limit the stack allocations too.
*/
static const int mode1_syscalls[] = {
    __NR_seccomp_read, __NR_seccomp_write, __NR_seccomp_exit,
    ↪ __NR_seccomp_sigreturn,
#ifdef __NR_uretprobe
    __NR_uretprobe,
#endif
#ifdef __NR_uprobe
    __NR_uprobe,
#endif
    -1, /* negative terminated */
};

// linux v7.2.8, kernel/seccomp.c:1067-1085
static void __secure_computing_strict(int this_syscall)
{
    const int *allowed_syscalls = mode1_syscalls;
#ifdef CONFIG_COMPAT
    if (in_compat_syscall())
        allowed_syscalls = get_compat_mode1_syscalls();
#endif
    do {
        if (*allowed_syscalls == this_syscall)
            return;
    } while (++allowed_syscalls != -1);

#ifdef SECCOMP_DEBUG
    dump_stack();
#endif
    current->seccomp.mode = SECCOMP_MODE_DEAD;
    seccomp_log(this_syscall, SIGKILL, SECCOMP_RET_KILL_THREAD,
    ↪ true);
    do_exit(SIGKILL);
}

```

The table has read, write, exit and rt_sigreturn (`__NR_seccomp_*` in `include/asm-generic/seccomp.h` map to `__NR_read`, `__NR_write`, `__NR_exit` and `__NR_rt_sigreturn`, which are 63, 64, 93 and 139 in `include/uapi/asm-generic/unistd.h`), and, where the architecture defines them, `uretprobe` and `uprobe`. `getpid` is 172, is not in the table, and the loop falls out of the `do ... while` into the tail: the task's mode is set to dead, the attempt is logged, and `do_exit(SIGKILL)` runs, in the task's own context, before the syscall's handler is ever called. The process is killed by signal 9; BusyBox's shell prints `Killed` and reports `128 + 9 = 137`, the two lines the capture pins after `strict` mode on.

The program's last write never runs.

Seccomp is a change to **who is trusted**, by shrinking the interface: the kernel's TCB is unchanged, but the *set of doors into it* that this task can reach is four. Filter mode, which the jailer uses, replaces the table with a program; Exercise (d) reads where.

The configuration and the build's identity

defconfig is the kernel's own statement of what it is. The config symbols the probe depends on are these, read from the .config that make defconfig wrote for this tag, and every one is =y:

```
CONFIG_CGROUPS=y
CONFIG_NAMESPACES=y
CONFIG_UTS_NS=y
CONFIG_IPC_NS=y
CONFIG_USER_NS=y
CONFIG_PID_NS=y
CONFIG_NET_NS=y
CONFIG_BLK_DEV_INITRD=y
CONFIG_SECCOMP=y
CONFIG_SECCOMP_FILTER=y
CONFIG_DEVTMPFS=y
CONFIG_DEVTMPFS_MOUNT=y
CONFIG_VIRTIO_BLK=y
CONFIG_SERIAL_8250_CONSOLE=y
CONFIG_POWER_RESET_SYSCON=y
CONFIG_POWER_RESET_SYSCON_POWEROFF=y
CONFIG_VIRTIO_MMIO=y
```

Nothing had to be enabled on top of defconfig; the only edit build.sh makes to the kernel's configuration is to disable CONFIG_LOCALVERSION_AUTO, which scripts/setlocalversion could otherwise extend on a shallow clone with a + or -g suffix, so that uname -r is exactly 7.2.8. The first sixteen hex digits of the SHA-256 of the final .config are 4f23a572ce2f5876 on straylight; build.sh writes them to out/config.sha. The hash covers every line of the file, including the compiler's own version string: CONFIG_CC_VERSION_TEXT and CONFIG_GCC_VERSION are in .config, so a different toolchain gives a different hash, and talos, whose cross compiler is GCC 16.2.1 against straylight's 16.2.0, wrote 7fcf186d9ad26d19 for the same tag and the same CONFIG_ answers. The hash is the identity of *this* build, not of the tag: the tag says which source, the CONFIG_ lines above say which question the source was asked, and the hash says which compiler built the answer. The capture is identical on both hosts.

Walkthrough

Nothing here quotes the kernel again; the kernel blocks are in Mechanism. These three listings are ours.

/init

```
# PID 1. Mount the three kernel views a probe needs, run the probe
# between two sentinels the lab's capture is cut at, then power off:
# the kernel's own log lines carry timestamps and never enter the
# capture.
mount -t proc proc /proc
mount -t sysfs sysfs /sys
mount -t devtmpfs devtmpfs /dev 2>/dev/null
mkdir -p /sys/fs/cgroup
mount -t cgroup2 none /sys/fs/cgroup
# Only emergencies reach the console from here: a kernel line landing
# between the sentinels would make the capture depend on timing.
dmesg -n 1
echo '=== probe begin'
/probe
echo "=== probe end rc=$?"
poweroff -f
```

PID 1 mounts three views of the kernel before anything else can be asked: /proc for the per-task files, /sys for the device and cgroup trees, and devtmpfs on /dev, which the kernel populates by itself. CONFIG_DEVTMPFS_MOUNT does not apply to an initramfs (its help text in drivers/base/Kconfig says the file system always has to be mounted by hand after the rootfs is), so /init mounts it; the probe never reads /dev, and the 2>/dev/null only keeps a failed mount's message off the console. cgroup2 goes on /sys/fs/cgroup, which is what makes cgroup.controllers exist. dmesg -n 1 sets the console log level so that only emergencies print, because a kernel message that lands between the sentinels would make the capture depend on timing. Then the two sentinels bracket the probe, and the closing one carries the probe's exit status. poweroff -f powers the machine off at once, without signalling init, which is this script. The virt machine has a power-off device (OpenSBI's boot banner names syscon-poweroff as the platform shutdown device), the console shows reboot: Power down, and QEMU exits on its own without the timeout firing.

/probe

```
# Every line here is a fact about the running kernel that the chapter
```

```

# reads in the source. Nothing prints a timestamp, an address or a
↳ pid
# from outside the new namespace (those depend on how many kernel
# threads started first). Inside unshare, the count uses the shell's
# glob and forks nothing, so the answer is the shell alone.
echo "kernel $(uname -r)"
echo "pid1 $(cat /proc/1/comm)"
grep -E '^(CapEff|Seccomp|NoNewPrivs):' /proc/self/status
echo namespaces $(ls /proc/self/ns)
echo "cgroup controllers: $(cat /sys/fs/cgroup/cgroup.controllers)"
if [ "${$$}" -ne 1 ]; then echo 'pid outside: not 1'; fi
unshare -f -p --mount-proc sh -c 'echo "pid inside: ${$$}"; set --
↳ /proc/[0-9]*; echo "processes visible inside: $#"'
echo "seccomp strict, then a forbidden syscall:"
/seccomp-probe
echo "seccomp-probe rc=$?"

```

Each line asks one question of one kernel object. `uname -r` is the version the build pinned. `/proc/1/comm` names PID 1. The `grep` picks three fields of `proc_pid_status` for the probe's own process. `/proc/self/ns` lists the namespace files of the process; `cgroup.controllers` lists the controllers of the root cgroup. The `if` line is the outside view of pid numbering: the probe is `init`'s child, so `$$` (the shell's own pid, spelled so the book's math markup leaves it alone) is not 1 and the line prints `pid outside: not 1`, the counterpart of `pid inside: 1`, and unlike the pid itself the sentence does not depend on how many kernel threads started before the probe. The `unshare` line is the namespace experiment, and the `set --` and `$#` count the entries `/proc/[0-9]*` expands to without forking. The last two lines run the C program and print its exit status.

seccomp.c

```

/* Enter seccomp strict mode – read, write, _exit and sigreturn are
↳ the
* whole syscall table from here – then call one that is not on it.
↳ The
* kernel answers with SIGKILL; the shell reports 137. Chapter 15
↳ reads
* the check in kernel/seccomp.c. */
#include <stdio.h>
#include <unistd.h>
#include <sys/prctl.h>
#include <linux/seccomp.h>

int main(void)

```

```

{
    if (prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT, 0, 0, 0) != 0) {
        perror("prctl");
        return 2;
    }
    write(1, "strict mode on\n", 15);
    getpid(); /* not read, write, _exit or sigreturn */
    write(1, "still alive (no seccomp?)\n", 26);
    return 0;
}

```

The program checks the `prctl` call, writes one line, makes the forbidden call, and would write a second line if it survived. The first write is allowed, so the capture has `strict mode on`; the `getpid` call is not, so the second write never runs and the string `still alive` never appears. `getpid()` on riscv64 is a real `ecall`, which the run shows by the program dying. The write's return value is ignored, the compiler says so when the program is built, and the warning is correct and harmless: a program whose next instruction might be its own death has nowhere to put an error.

Exercises

(a) Predict `CapEff` for a process that runs `unshare -U` (a new user namespace), then run it in the guest and answer with the value observed and the kernel's reason.

Answer, p. 197

(b) Drop `--mount-proc` from the probe's `unshare` line. What do the two lines print, and why does only one of them change? Cite `fs/proc/root.c`.

Answer, p. 198

(c) Which `cgroup.controllers` entry would disappear if `CONFIG_MEMCG` were off? Cite the table.

Answer, p. 198

(d) Replace `SECCOMP_MODE_STRICT` with a filter. What does `SECCOMP_MODE_FILTER` need that `strict` does not, and where does the kernel run it?

Answer, p. 199

(e) Which of the five objects has `ember` an analogue of, and which has it none of? Use chapter 14's axes.

Answer, p. 200

(f) Which of the five does the jailer of chapter 26 use on wick? Answer from the syllabus alone.

Answer, p. 200

Lab

```
make lab CH=15
make lab CH=15 IMPL=mime
```

labs/ch15/build.sh fetches Linux v7.2.8 from <https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git> and BusyBox 1_38_0 from <https://git.busybox.net/busybox>, each a shallow clone of one tag, into `${XDG_CACHE_HOME:-$HOME/.cache}/hp/ch15/`, outside the tree. A clean clone of the book does not rebuild them and `rm -rf _build` does not remove them. It runs `defconfig`, builds Image for riscv64, builds BusyBox static with `CONFIG_TC` off (its `tc.c` needs CBQ definitions that the toolchain's kernel headers do not have), installs it into an `initramfs` tree, adds `/init`, `/probe` and the static `seccomp-probe`, and packs the tree with `cpio` and `gzip`. The cross prefix is whichever of `riscv64-linux-gnu-` and `riscv64-unknown-linux-gnu-` is installed, the two spellings of the same toolchain that make `doctor` accepts. Only `out/Image` and `out/initramfs.cpio.gz` leave the cache. `out/Image` is a riscv64 kernel that boots under QEMU here; it is not chapter 24's guest (the opening of this chapter says why).

Timings, measured: on `straylight` (32 cores) on 2026-09-30, the first run took 4 min 3 s wall, including the copy of the tree, the clone of both trees and the kernel's configuration and build, and ended where BusyBox's build stopped at `tc.c` (the reason for `CONFIG_TC` off); a complete cold rebuild with `CONFIG_TC` off, on 2026-10-01, took 302 s wall including the `rsync` and the boot; the run that then built BusyBox, the `initramfs` and booted took 22 s; a run with a warm cache (`build.sh` skips the builds and repacks) takes about 10 s. On `talos` (8 cores), the first run took 18 min 4 s wall for the clone, both builds and the boot (BusyBox and the kernel both built there).

The boot is `qemu-system-riscv64 -machine virt -m 512M -smp 1 -nographic -monitor none -kernel out/Image -initrd out/initramfs.cpio.gz -append 'console=ttyS0 rdinit=/init'`. There is no `-bios` none: that is ember's rule. Here OpenSBI, QEMU's default firmware, runs in M mode and hands the kernel to S mode. That is this boot only. A RISC-V

guest under KVM runs no M-mode firmware: the host kernel answers its SBI calls itself, in `arch/riscv/kvm/vcpu_sbi.c` and its neighbours. Chapter 24's x86_64 guest has no M mode to begin with.

The capture is the probe's lines between `=== probe begin` and `=== probe end`, cut out of the console with `sed` and stripped of carriage returns. The kernel's own boot log, with timestamps and addresses, is in `out/console.txt` and is not pinned: a timestamp moves on every boot, and a capture of it would fail on a cold cache. Two runs on `straylight` were byte-identical. The capture holds three lines with a tab character after the colon, because that is what the kernel prints; a diff that complains about those lines has had its tabs replaced.

`IMPL=mime` rebuilds the `initramfs` from `mime/` against the same kernel. The reader's copy has the `seccomp` program stubbed to print `MINE: seccomp probe stub` and exit 1: the boot and every line before it match, and the diff is the three lines after `seccomp strict`, then a forbidden `syscall`:

Further reading

- Control Group v2 (Linux kernel documentation) — the authoritative description of the hierarchy, the controllers and the default-hierarchy rules that `cgroup_control` implements.
- Seccomp BPF (SECure COMPUting with filters) (Linux kernel documentation) — filter mode, the return actions and `no_new_privs`, for Exercise (d).
- `namespaces(7)` (Linux man-pages) — the namespace types and what each wraps.
- `capabilities(7)` (Linux man-pages) — the sets, the `execve` transformation, and the user-namespace rules Exercise (a) runs into.
- `unshare(2)` (Linux man-pages) — why `CLONE_NEWPID` moves the children and not the caller.
- The kernel source at v7.2.8 — every file quoted above, at the tag the build fetches.

Answers

(a) Observed on `talos`, from a scratch copy of `probe.sh` with `unshare -U sh -c 'grep -E "^Cap(Prm|Eff|Bnd):" /proc/self/status; id -u'` appended, rebuilt and booted: `CapPrm: 0000000000000000, CapEff: 0000000000000000, CapBnd: 000001ffffffffffff`, and `id -u` prints 65534. The prediction that a new user namespace gives a process every capability is half

right, and the half that is wrong is the interesting one. `set_cred_user_ns` in `kernel/user_namespace.c` does give the task the full set when the namespace is created:

```
// linux v7.2.8, kernel/user_namespace.c:44-53
static void set_cred_user_ns(struct cred *cred, struct user_namespace
↳ *user_ns)
{
    /* Start with the same capabilities as init but useless for
    ↳ doing
    * anything as the capabilities are bound to the new user
    ↳ namespace.
    */
    cred->securebits = SECUREBITS_DEFAULT;
    cred->cap_inheritable = CAP_EMPTY_SET;
    cred->cap_permitted = CAP_FULL_SET;
    cred->cap_effective = CAP_FULL_SET;
    cred->cap_ambient = CAP_EMPTY_SET;
```

but the shell is then `execed`, and at `execve` the capability sets are recomputed. `cap_bprm_creds_from_file` in `security/commoncap.c` (line 935) computes `root_uid = make_kuid(new->user_ns, 0)` and passes it to `handle_privileged_root`, which grants the root sets only if the process's uid is that uid. In a new user namespace with no uid map, `make_kuid` calls `map_id_down`, which returns `(u32) -1` for an id that is not mapped (`kernel/user_namespace.c`, `map_id_range_down`), so there is no root in the namespace for the process to be, `CapPrm` and `CapEff` come out as zero, and `id -u` shows the overflow uid 65534. The bounding set is untouched. With a mapping of root (`unshare -U -r`; the map is `0 0 1`) the same probe prints `CapPrm`, `CapEff` and `CapBnd` all `000001ffffffffffff` and uid 0. What the new namespace's capabilities are worth is the last branch of `cap_capable_helper`: they are held over the new namespace and its descendants, not over the namespaces above it.

(b) Observed on `talos`, from a scratch copy of `probe.sh` with a copy of the `unshare` line appended, `--mount-proc` removed and its two echoes relabelled `no mount-proc: pid inside` and `no mount-proc: visible`, rebuilt and booted: `no mount-proc: pid inside 1` and `no mount-proc: visible 47`. The pid does not change, because the flag has nothing to do with allocation: Mechanism's `alloc_pid` still hands the first child of the new namespace pid 1. The count changes because `/proc` is still the mount `/init` made, and `proc_init_fs_context` in `fs/proc/root.c` fixed that mount's namespace to `task_active_pid_ns(current)` of its mounter, the host's. The glob lists the host's processes and kernel threads, a count of this boot that the probe therefore never prints.

(c) memory. In `include/linux/cgroup_subsys.h` the line `SUBSYS(memory)` is inside `#if IS_ENABLED(CONFIG_MEMCG)` (lines 28 to 30 at the tag), and the `cgroup_subsys[]` array in `kernel/cgroup/cgroup.c` is built by including that header, so with the symbol off the slot does not exist, the controller is not in the mask that `cgroup_control` returns, and the word does not print. The other five are gated by other symbols: `cpuset` by `CONFIG_CPUSETS`, `cpu` by `CONFIG_CGROUP_SCHED`, `io` by `CONFIG_BLK_CGROUP`, `hugetlb` by `CONFIG_CGROUP_HUGETLB` and `pids` by `CONFIG_CGROUP_PIDS`.

(d) A BPF program, passed in by the caller, and one of two things about the caller's privilege. The program is what strict mode's fixed table is replaced with. The privilege rule is in `kernel/seccomp.c`:

```
// linux v7.2.8, kernel/seccomp.c:684-693

/*
 * Installing a seccomp filter requires that the task has
 * CAP_SYS_ADMIN in its namespace or be running with no_new_privs.
 * This avoids scenarios where unprivileged tasks can affect the
 * behavior of privileged children.
 */
if (!task_no_new_privs(current) &&&
    !ns_capable_noaudit(current_user_ns(), CAP_SYS_ADMIN))
    return ERR_PTR(-EACCES);
```

A task installs a filter only if it has set `no_new_privs` (so that a program it later runs cannot gain privilege through the filter) or holds `CAP_SYS_ADMIN` in its user namespace. The kernel runs the program at the same point strict mode's check runs: `__secure_computing`'s `SECCOMP_MODE_FILTER` case calls `__seccomp_filter`, which calls `seccomp_run_filters`, and that evaluates every attached filter with `bpf_prog_run_pin_on_cpu` and keeps the lowest action:

```
// linux v7.2.8, kernel/seccomp.c:404-430
static u32 seccomp_run_filters(const struct seccomp_data *sd,
                              struct seccomp_filter **match)
{
    u32 ret = SECCOMP_RET_ALLOW;
    /* Make sure cross-thread synced filter points somewhere sane.
     */
    struct seccomp_filter *f =
        READ_ONCE(current->seccomp.filter);

    /* Ensure unexpected behavior doesn't result in failing open. */
    if (WARN_ON(f == NULL))
        return SECCOMP_RET_KILL_PROCESS;
```

```

if (seccomp_cache_check_allow(f, sd))
    return SECCOMP_RET_ALLOW;

/*
 * All filters in the list are evaluated and the lowest BPF return
 * value always takes priority (ignoring the DATA).
 */
for (; f; f = f->prev) {
    u32 cur_ret = bpf_prog_run_pin_on_cpu(f->prog, sd);

    if (ACTION_ONLY(cur_ret) < ACTION_ONLY(ret)) {
        ret = cur_ret;
        *match = f;
    }
}
}

```

so the decision is an integer the program returns (allow, errno, trap, kill), not always a signal. The verdict is reached before the syscall's handler runs, as in strict mode.

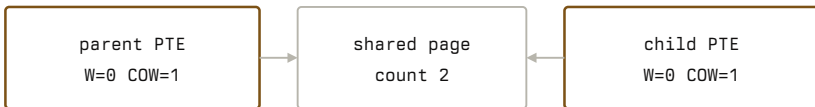
(e) The syscall table, namespaces and capabilities have analogues; cgroups and seccomp have none. The table is chapter 13's switch on a7 in `ember/syscall.c`, the same door with nine entries and no hook before it. A namespace is a per-task table of names that reach objects; `ember`'s is the descriptor table, `ofile[NOFILE]` in `ember/proc.h`, which maps a small integer to an entry in the kernel-wide open-file table (`ember/fd.h`): two processes that both hold descriptor 1 are not naming the same thing unless the kernel put them there, which is the *shared* axis (what a name reaches) made per-process. The capability analogue is the same table read differently: a descriptor is the only way a program names a pipe end, and the `readable` and `writable` bits in `struct file` are rights bound when it is created, which chapter 14's exercise (c) already used for exokernel secure binding; it is the closest `ember` has to `seL4`'s capabilities and a long way from POSIX's, which divide *root*. It has no analogue of a cgroup, because nothing in it accounts for or bounds a process's use of anything except the fixed `NOFILE` and `NOFILE` tables, and no analogue of `seccomp`, because all nine syscalls are open to every process: `syscall()` in `ember/syscall.c` has no per-task filter before its switch. `Procfs`, the window rather than one of the five, has no analogue either: none of the nine syscalls reports the kernel's own state to a program.

(f) `SYLLABUS.md`'s entry for chapter 26 reads "Namespaces, cgroups, `pivot_root`, `drop uid`, `seccomp` on the *VMM*." Three of the five are named outright: namespaces, cgroups and `seccomp`. "Drop uid" is the fourth, the

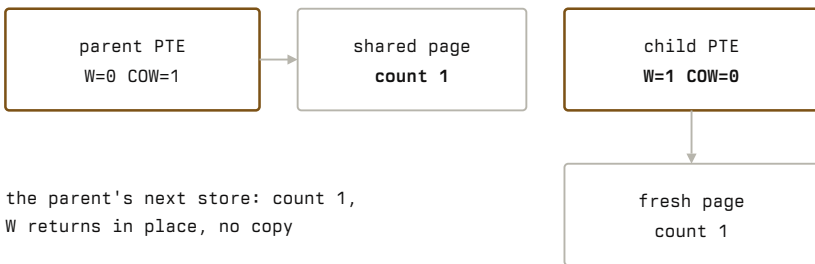
credential and capability object: a process that gives up root loses the capability sets that `CapEff` shows. The syscall table is not chosen by the jailer; it is the boundary the other objects guard, and `seccomp` narrows it. `Procfs` is not one of the five and is not used as a mechanism: the jailer only needs the mount in its namespace to be correct. The entry also says where the five apply: to the VMM process, not to the guest. The jailer does not reach into the guest kernel, which runs as a virtual machine; everything the five do is about `wick`, the process on the host.

16. A LOOK THROUGH XV6

BEFORE



AFTER THE CHILD WRITES



the copy happens on the write that needs it, and only for that page

Why this layer exists

Chapter 13 gave `ember spawn` and not `fork`, and said why: `fork` needs a second address space, and `ember` has one page table. It named the cure in one clause, per-process page tables with copy-on-write (chapter 14's term), and left it. `xv6` is the kernel that has the per-process tables, in a teaching Unix small enough to read whole. At the pinned commit its `fork` copies every page of the parent. This chapter makes that `fork` cheap, and reads the kernel's trap, page-table and allocator code while doing it.

It is one extension, not a port of `ember` to `xv6`'s structure and not a tour of `xv6`. The extension is the one chapter 13 postponed. It touches the three places where a page's meaning changes hands: the allocator, which has to know when a page

is free; the page table, which has to say that a page is shared; and the trap handler, which has to turn a store to a shared page into a private one.

Mechanism

Every block headed `xv6-riscv 06aad25` is copied from a checkout of `https://github.com/mit-pdos/xv6-riscv` at that commit, by line range. Blocks headed `+ 0001-cow-fork.patch` are hunks of `labs/ch16/patches/0001-cow-fork.patch`, the lab's one patch, and `labs/ch16/build.sh` applies it to that checkout. At this commit the kernel functions that implement system calls carry a `k` prefix (`kfork`, `kexit`, `kwait`, `kexec`), so the fork system call is `kfork` in the kernel, and the call that sleeps is `pause`. The *xv6* book (rev4, August 2024) describes the same design under the older names, and says that fork is implemented with `uvmcopy`, “which allocates physical memory for the child and copies the parent's memory into it”.

The fork this chapter starts from

`kfork` allocates a process, gives it a page table, and fills the table by calling `uvmcopy`:

```
// xv6-riscv 06aad25, kernel/proc.c:270-275
// Copy user memory from parent to child.
if (uvmcopy(p->pagetable, np->pagetable, p->sz) < 0) {
    freeproc(np);
    release(&np->lock);
    return -1;
}
```

`uvmcopy` walks the parent's user pages, one `PGSIZE` (4096 bytes) at a time:

```
// xv6-riscv 06aad25, kernel/vm.c:292-326
// Given a parent process's page table, copy
// its memory into a child's page table.
// Copies both the page table and the
// physical memory.
// returns 0 on success, -1 on failure.
// frees any allocated pages on failure.
int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
    pte_t *pte;
    uint64 pa, i;
    uint flags;
```



```

char *mem;

for (i = 0; i < sz; i += PGSIZE) {
    if ((pte = walk(old, i, 0)) == 0)
        continue; // page table entry hasn't been allocated
    if ((*pte & PTE_V) == 0)
        continue; // physical page hasn't been allocated
    pa = PTE2PA(*pte);
    flags = PTE_FLAGS(*pte);
    if ((mem = kalloc()) == 0)
        goto err;
    memmove(mem, (char *)pa, PGSIZE);
    if (mappages(new, i, PGSIZE, (uint64)mem, flags) != 0) {
        kfree(mem);
        goto err;
    }
}
return 0;

err:
    uvmunmap(new, 0, i / PGSIZE, 1);
    return -1;
}

```

For each page that is present, walk returns the leaf *page-table entry* (PTE) at that address in the parent's table, PTE2PA takes the physical address out of it, PTE_FLAGS takes its low ten bits (the permissions V, R, W, X, U, and their neighbours), kalloc finds a fresh page, memmove copies 4096 bytes into it, and mappages makes the child's PTE point at the copy with the same flags. The cost is the size of the parent, paid at the moment of fork, whether or not the child ever touches the memory. For the commonest use of fork the child touches almost none of it: the shell forks and the child immediately calls exec, which discards the address space the copy just built.

The comment at the top of labs/ch16/cowtest.c states the problem, and the arithmetic is short. Physical memory is PHYSTOP - KERNBASE, 128 MiB (kernel/memlayout.h), and its simple test grows the process to two thirds of that and forks. A copy needs two thirds again; there is a third. Run against the unpatched kernel, with only cowtest added to the image, the test prints the failure:

```

$ cowtest
simple: fork() failed

```

Copy-on-write, and a count

Under copy-on-write (COW from here on), the child's page table is built to point at the parent's physical pages; nothing is copied; and both tables are marked so that the first store through either one is caught and given a private copy of that one page. Memory is copied on the write that needs it, for the pages that are written, and never for the rest.

Three things have to be true for that to work.

The share must be visible to the hardware. A page that two page tables map must fault when either writes. RISC-V's PTE has a write bit, *W*, and a store to a page with *W* clear raises a *store page fault*, scause 15 (the privileged specification: "Attempting to execute a store, store-conditional, AMO, or cache-block zero instruction whose effective address lies within a page without write permissions raises a store page-fault exception"; sic, the doubled word is the specification's). So *uvmcopy* clears *W* in the parent's PTE as well as the child's. A page that was read-only to begin with (the program's text) stays as it is and is shared without any further mark: nothing will ever store to it.

The fault must be able to tell a share from a mistake. A store to a page with *W* clear is also what a store to the program's own text looks like, and that one has to kill the process. The handler needs one bit of its own. The PTE reserves two for exactly this: the *RSW* field, bits 9 and 8, which the privileged specification calls "reserved for use by supervisor software; the implementation shall ignore this field" (Figure 22, Sv39 page table entry, the format chapter 4 introduced). The patch names bit 8 PTE_COW:

```
// xv6-riscv 06aad25 + 0001-cow-fork.patch, kernel/riscv.h
@@ -397,6 +397,7 @@ typedef uint64 *pagetable_t; // 512 PTEs
#define PTE_W (1L << 2)
#define PTE_X (1L << 3)
#define PTE_U (1L << 4) // user can access
+#define PTE_COW (1L << 8) // copy-on-write: RSW bit, ours (ch16)

// shift a physical address to the right place for a PTE.
#define PA2PTE(pa) (((uint64)pa) >> 12) << 10)
```

A PTE now has one of three states for a user page: *W*=1, *COW*=0 (private and writable), *W*=0, *COW*=1 (shared, or formerly shared, until the next store), and *W*=0, *COW*=0 (read-only: text). *W*=1 together with *COW*=1 is never written, and the design depends on that.

The kernel must know when a shared page is free. Upstream *kfree* puts

a page on the free list, and a process's pages are freed when its parent reaps it with `wait` (`kwait` calls `freeproc`, which reaches `uvmunmap` with `do_free` set, below; `kexit` only makes a zombie, which keeps its references until then). With shared pages the first process to be reaped would free pages the other still maps. The allocator needs a *reference count* per page: the number of holders of the page, where a holder is a PTE (or a kernel structure that owns the page outright: `pipealloc` takes a page from `kalloc` for a `struct pipe`, and the pipe is its one holder). `kalloc` hands a page out with a count of one. `kfree` becomes “drop a reference” and frees the page only when the count reaches zero. `uvmcopy` raises the count of every page it shares, read-only ones included, because a text page is held by every process running the program.

The count is a static array indexed by page number, from `KERNBASE` to `PHYSTOP`: 32768 ints, 128 KiB of the kernel's BSS. It lives in the allocator, not in the page (the patch's comment: a page with a count is exactly a page someone may still be reading), and not in the PTE, which is per table where the count is per physical page. The invariants the rest of the chapter keeps are two. A page's count is the number of holders that name it, at every point. No PTE has `W` and `COW` both set.

The fault

A store to a `PTE_COW` page traps into `usertrap` with `scause` 15. The handler has two cases, and they are the whole point of the count.

If the count is greater than one, the page really is shared: the handler allocates a page, copies the 4096 bytes, points the faulting process's PTE at the copy with `W` set and `COW` clear, and calls `kfree` on the old page, which drops one reference (the process's own) and leaves the others. If the count is one, every other holder has already let go (the sibling was reaped, or took its own copy first), and the page is this process's alone; copying it would be waste, so the handler sets `W`, clears `COW`, and returns. The page the parent held read-only through the whole fork becomes writable again in place, with no copy at all.

The handler writes the PTE directly. The obvious tool, `mappages`, refuses:

```
// xv6-riscv 06aad25, kernel/vm.c:164-167
if ((pte = walk(pagetable, a, 1)) == 0)
    return -1;
if (*pte & PTE_V)
    panic("mappages: remap");
```

A mapping that exists is a bug to `mappages`, so replacing a PTE goes around it. The fault returns to the same store, which now succeeds. The handler needs no

`sfence.vma`, the instruction that tells a hart to discard its cached translations, to make the hart see the new PTE: `kernel/trampoline.S`'s `userret` executes one on every return to user mode, and the return from `usertrap` goes through it:

```
// xv6-riscv 06aad25, kernel/trampoline.S:100-112
.globl userret
userret:
    # usertrap() returns here, with user satp in a0.
    # return from kernel to user.

    # flush icache, in case this is the first time
    # we're running this proc on this hart.
    fence.i

    # switch to the user page table.
    sfence.vma zero, zero
    csrw satp, a0
    sfence.vma zero, zero
```

The first `sfence.vma` waits for earlier memory operations to finish under the table they started with; both `sfence.vma zero, zero` discard every cached translation, and the one after `csrw satp, a0` is the one that matters here: nothing the hart remembered about this process's table before the kernel changed it survives into user mode. A PTE rewritten while the process is in the kernel is therefore never seen stale: `uvmcopy` rewrites the *parent's* PTEs during `fork` and the parent returns through the same code.

In `usertrap` the patch puts the new case first. Order is a matter of cost here, not of correctness: upstream's next case, `vmfault`, handles the *lazily allocated* pages `sbrklazy` creates (a page the kernel promised and has not backed with memory yet) and returns zero at once for a page that is already mapped:

```
// xv6-riscv 06aad25, kernel/vm.c:458-469
uint64
vmfault(pagetable_t pagetable, uint64 psz, uint64 va, int read)
{
    uint64 mem;

    if (va >= psz)
        return 0;
    va = PGROUNDDOWN(va);
    if (ismapped(pagetable, va)) {
        return 0;
    }
    mem = (uint64)kalloc();
```

so a COW page falls through `vmfault` to the `else` that kills the process, which is why the COW case must exist at all, and why either order works. The COW case goes first because it is exact (scause 15 only: a COW page keeps R, so only stores fault on it) and `vmfault` is the general one. A page that was lazily left unallocated in the parent is skipped by `uvmcopy` (`if ((*pte & PTE_V) == 0) continue`), so the child's copy of that range is unmapped too and the child faults it in the old way.

The kernel writes too

User code is not the only writer of user memory. `copyout` copies from the kernel into a process, by looking the user address up in the page table and writing to the physical page through the kernel's own mapping of RAM. It never stores through the user's PTE, so no store page fault can happen to catch it, and the only guard upstream has is a software test:

```
// xv6-riscv 06aad25, kernel/vm.c:362-365
pte = walk(pagetable, va0, 0);
// forbid copyout over read-only user text pages.
if ((*pte & PTE_W) == 0)
    return -1;
```

That test refuses a COW page, so a read into shared memory fails with `-1`; without the test, the write would go through to a page the parent still reads. Either way a COW page needs a fault taken on the process's behalf: the patch's `copyout` hunk calls `cowfault` itself, looks the physical address up again (the PTE now points at a copy) and goes on. Only `copyout` needs it. `copyin` and `copyinstr` only read, and the one other writer, `exec`'s `loadseg` (`kernel/exec.c`, through `walkaddr`), writes into pages it has just allocated in a fresh table, which no one shares.

What the count already handles

`uvmunmap`, which reaping a child (`kwait` to `freeproc`), `exec` and a failed `uvmcopy` all reach, frees a page with `kfree`:

```
// xv6-riscv 06aad25, kernel/vm.c:207-211
if (do_free) {
    uint64 pa = PTE2PA(*pte);
    kfree((void *)pa);
}
*pte = 0;
```

With counts, that call is “drop this process's reference”, which is what a holder

letting go of a shared page should do, and the change that makes it so is in `kfree`, not here. `uvmcopy`'s own failure path (`uvmunmap(new, 0, i / PGSIZE, 1)`) is right for the same reason: every page already mapped into the child has had its count raised, and the page that failed to map has not.

The kernel's own page table maps RAM directly and writable, with no COW bit, so a kernel store to a user page lands on the physical page whatever the user's PTE says: that is why copyout is the one place that has to check. The trapframe and the trampoline are mapped into every process's table above `sz`, outside the range `uvmcopy` walks. The trapframe page is the child's own; the trampoline is one page in every table, never counted and never freed (`proc_freepagetable` unmmaps both with `do_free` zero).

Walkthrough

The patch is fourteen hunks in six files, in the order `git diff` lists them, which is alphabetical by path; this section takes them by mechanism, not by path.

`kernel/kalloc.c` carries five. The first holds the count array and the two functions the rest of the kernel uses to read and raise it; the comment in the patch is the specification of the invariant:

```
// xv6-riscv 06aad25 + 0001-cow-fork.patch, kernel/kalloc.c
@@ -18,11 +18,42 @@ struct run {
    struct run *next;
};

+// listing:ch16-refcount
+// One reference count per physical page from KERNBASE to PHYSTOP,
+// indexed by page number. A page is free when its count reaches
+//   zero;
+// fork's copy-on-write share raises it, and a write fault or a kfree
+// lowers it. The count lives in the kernel, not in the page: a page
+// with a count is exactly a page someone may still be reading.
+#define PGREF(pa) (refs[((uint64)(pa) - KERNBASE) / PGSIZE])
+static int refs[(PHYSTOP - KERNBASE) / PGSIZE];
+
+struct {
+    struct spinlock lock;
+    struct run *freelist;
+} kmem;

+// Raise the count of the page holding pa. Called by uvmcopy when a
+// child comes to share a parent's page.
```

```

+void
+krefinc(void *pa)
+{
+  acquire(&kmem.lock);
+  PGREF(pa)++;
+  release(&kmem.lock);
+}
+
+// The count, for cowfault's "am I the only one" test.
+int
+krefcount(void *pa)
+{
+  int n;
+  acquire(&kmem.lock);
+  n = PGREF(pa);
+  release(&kmem.lock);
+  return n;
+}
+// end-listing:ch16-refcount
+
+void
+kinit()
+{

```

`krefinc` and `krefcount` take `kmem.lock`, the allocator's existing spinlock, rather than a lock of their own: the count is read and written in `kfree` under the same lock, and two locks around one number is a bug waiting for a second hart. `freerange` runs once at boot, before anything has a count, and calls `kfree` for every page of RAM; `kfree` now expects a count of at least one, so `freerange` sets one first. The same hunk replaces `kfree`'s header comment:

```

// xv6-riscv 06aad25 + 0001-cow-fork.patch, kernel/kalloc.c
@@ -35,14 +66,13 @@ freerange(void *pa_start, void *pa_end)
{
  char *p;
  p = (char *)PGROUNDUP((uint64)pa_start);
-  for (; p + PGSIZE <= (char *)pa_end; p += PGSIZE)
+  for (; p + PGSIZE <= (char *)pa_end; p += PGSIZE) {
+    PGREF(p) = 1; // so that kfree below frees it
+    kfree(p);
+  }
}

-// Free the page of physical memory pointed at by pa,
-// which normally should have been returned by a

```

```

-// call to kalloc(). (The exception is when
-// initializing the allocator; see kinit above.)
+// Drop one reference; free the page when nobody holds it.
void
kfree(void *pa)
{

```

kfree decrements under the lock, panics if the count was already zero (a page freed twice, the invariant's negative), and returns without touching the page if anyone still holds it. Only the last release fills the page with junk and links it into the free list, as before:

```

// xv6-riscv 06aad25 + 0001-cow-fork.patch, kernel/kalloc.c
@@ -51,6 +81,16 @@ kfree(void *pa)
    if (((uint64)pa % PGSIZE) != 0 || (char *)pa < end || (uint64)pa >=
↳ PHYSTOP)
        panic("kfree");

+   acquire(&kmem.lock);
+   if (PGREF(pa) < 1)
+       panic("kfree: refcount");
+   PGREF(pa)--;
+   if (PGREF(pa) > 0) {
+       release(&kmem.lock);
+       return;
+   }
+   release(&kmem.lock);
+
    // Fill with junk to catch dangling refs.
    memset(pa, 1, PGSIZE);

```

The last two change kalloc's comment and kalloc itself, which sets the count of a page it hands out to one:

```

// xv6-riscv 06aad25 + 0001-cow-fork.patch, kernel/kalloc.c
@@ -62,7 +102,7 @@ kfree(void *pa)
    release(&kmem.lock);
}

-// Allocate one 4096-byte page of physical memory.
+// Allocate one 4096-byte page of physical memory, with a count of
↳ one.
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
void *

```



```
// xv6-riscv 06aad25 + 0001-cow-fork.patch, kernel/kalloc.c
@@ -72,8 +112,10 @@ kalloc(void)

    acquire(&kmem.lock);
    r = kmem.freelist;
-   if (r)
+   if (r) {
        kmem.freelist = r->next;
+       PGREF(r) = 1;
+   }
    release(&kmem.lock);

    if (r)
```

kernel/defs.h gains the three prototypes, and Makefile adds cowtest to UPROGS:

```
// xv6-riscv 06aad25 + 0001-cow-fork.patch, kernel/defs.h
@@ -60,6 +60,8 @@ void          ireclaim(int);
void*          kalloc(void);
void           kfree(void *);
void           kinit(void);
+void          krefinc(void *);
+int           krefcount(void *);

// log.c
void           initlog(int, struct superblock*);

// xv6-riscv 06aad25 + 0001-cow-fork.patch, kernel/defs.h
@@ -156,6 +158,7 @@ void          kvminit(void);
void           kvminithart(void);
void           kvmmap(pagetable_t, uint64, uint64, uint64, int);
int            mappages(pagetable_t, uint64, uint64, uint64, int);
+int           cowfault(pagetable_t, uint64);
pagetable_t    uvmcreate(void);
uint64         uvmalloc(pagetable_t, uint64, uint64, int);
uint64         uvmdealloc(pagetable_t, uint64, uint64);

// xv6-riscv 06aad25 + 0001-cow-fork.patch, Makefile
@@ -143,6 +143,7 @@ UPROGS=\
    $_sh\
    $_stressfs\
    $_usertests\
+   $_cowtest\
    $_grind\
    $_wc\
    $_zombie\
```

uvmcopy is the first of four hunks in kernel/vm.c. The diff shows the old body going and the new one arriving in two adjacent hunks, the comment with the dropped mem, and the loop:

```
// xv6-riscv 06aad25 + 0001-cow-fork.patch, kernel/vm.c
@@ -289,19 +289,19 @@ uvmfree(pagetable_t pagetable, uint64 sz)
    freewalk(pagetable);
}

-// Given a parent process's page table, copy
-// its memory into a child's page table.
-// Copies both the page table and the
-// physical memory.
-// returns 0 on success, -1 on failure.
-// frees any allocated pages on failure.
+// listing:ch16-uvmcopy
+// Given a parent process's page table, give a child the same
+↪ physical
+// pages, copy-on-write: every writable page is mapped read-only in
+↪ both
+// tables with PTE_COW set and its reference count raised. Nothing is
+// copied here; the copy happens in cowfault, on the first write, for
+↪ the
+// pages that are written, and never for the rest. Returns 0 on
+↪ success,
+// -1 on failure, freeing what it mapped.
int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
    pte_t *pte;
    uint64 pa, i;
    uint flags;
-   char *mem;

    for (i = 0; i < sz; i += PGSIZE) {
        if ((pte = walk(old, i, 0)) == 0)

// xv6-riscv 06aad25 + 0001-cow-fork.patch, kernel/vm.c
@@ -309,14 +309,13 @@ uvmcopy(pagetable_t old, pagetable_t new, uint64
    ↪ sz)
        if ((*pte & PTE_V) == 0)
            continue; // physical page hasn't been allocated
        pa = PTE2PA(*pte);
-   flags = PTE_FLAGS(*pte);
-   if ((mem = kalloc()) == 0)
-       goto err;
```

```

- memmove(mem, (char *)pa, PGSIZE);
- if (mappages(new, i, PGSIZE, (uint64)mem, flags) != 0) {
-     kfree(mem);
-     goto err;
+ if (*pte & PTE_W) {
+     *pte = (*pte & ~PTE_W) | PTE_COW;
+ }
+ flags = PTE_FLAGS(*pte);
+ if (mappages(new, i, PGSIZE, pa, flags) != 0)
+     goto err;
+ krefinc((void *)pa);
+ }
return 0;

```

The loop is upstream's with the allocation and the copy gone. For a writable page it replaces the parent's PTE with `(*pte & ~PTE_W) | PTE_COW`, then takes the flags from that, so the child inherits them; `mappages` builds the child's PTE; `krefinc` raises the count. No `kalloc`, no `memmove`. Read-only pages go through the same lines without the first `if`.

`cowfault` is the fault handler as a function the trap path and `copyout` can both call. Its return value has three meanings (not a COW page: 0; handled: 1; out of memory: -1), because its two callers need to tell them apart:

```

// xv6-riscv 06aad25 + 0001-cow-fork.patch, kernel/vm.c
@@ -324,6 +323,40 @@ err:
    uvmunmap(new, 0, i / PGSIZE, 1);
    return -1;
}
+// end-listing:ch16-uvmcopy
+
+// listing:ch16-cowfault
+// A store to a PTE_COW page. If this process is the page's only
+↪ holder,
+// the page becomes writable in place; otherwise it gets its own copy
+↪ and
+// drops its reference to the shared one. Returns 0 if va is not a
+↪ COW
+// page (the caller decides what that means), 1 on success, -1 when
+↪ out
+// of memory.
+int
+cowfault(pagetable_t pagetable, uint64 va)
+{

```

```

+ pte_t *pte;
+ uint64 pa;
+ char *mem;
+
+ if (va >= MAXVA)
+     return 0;
+ va = PGROUNDDOWN(va);
+ pte = walk(pagetable, va, 0);
+ if (pte == 0 || (*pte & PTE_V) == 0 || (*pte & PTE_U) == 0 || (*pte
↳ & PTE_COW) == 0)
+     return 0;
+ pa = PTE2PA(*pte);
+ if (krefcount((void *)pa) == 1) {
+     *pte = (*pte | PTE_W) & ~PTE_COW;
+     return 1;
+ }
+ if ((mem = kalloc()) == 0)
+     return -1;
+ memmove(mem, (char *)pa, PGSIZE);
+ *pte = PA2PTE(mem) | ((PTE_FLAGS(*pte) | PTE_W) & ~PTE_COW);
+ kfree((void *)pa); // one fewer holder of the shared page
+ return 1;
+}
+// end-listing:ch16-cowfault

// mark a PTE invalid for user access.
// used by exec for the user stack guard page.

```

The guards before the count check are not decoration: *va* is the faulting address, any 64-bit value, so it is checked against *MAXVA* before *walk* sees it; the PTE must exist, be valid, be a user page and carry *PTE_COW*. A *scause 15* fault on anything else returns 0 and the old path decides. The *krefcount(...)* == 1 branch is the in-place take-over. The other branch builds the PTE from the new page with the old flags, *W* forced on and *COW* forced off, and then calls *kfree* on the old page: the count drops by this process's one. Neither branch holds a lock across the decision, and neither needs to. A count of one means the faulting process's own PTE is the only reference, and only a fork by that same single-threaded process could raise it, which cannot happen while it is in this handler. On the copy path the faulting's own reference keeps the page alive while the other holders let go.

copyout and *usertrap* get the last two hunks, one call site each:

```

// xv6-riscv 06aad25 + 0001-cow-fork.patch, kernel/vm.c
@@ -360,6 +393,14 @@ copyout(pagetable_t pagetable, uint64 psz, uint64
↳ dstva, char *src, uint64 len)

```

```

    }

    pte = walk(pagetable, va0, 0);
+   if (*pte & PTE_COW) {
+       // the kernel is about to write a shared page on the user's
↳   behalf:
+       // take the same fault the user would have taken.
+       if (cowfault(pagetable, va0) != 1)
+           return -1;
+       pa0 = walkaddr(pagetable, va0);
+       pte = walk(pagetable, va0, 0);
+   }
    // forbid copyout over read-only user text pages.
    if ((*pte & PTE_W) == 0)
        return -1;

// xv6-riscv 06aad25 + 0001-cow-fork.patch, kernel/trap.c
@@ -68,10 +68,14 @@ usertrap(void)
    syscall();
    } else if ((which_dev = devintr()) != 0) {
        // ok
+   // listing:ch16-trap
+   } else if (r_scause() == 15 && cowfault(p->pagetable, r_stval()) ==
↳   1) {
+       // store to a copy-on-write page: now the process's own
    } else if ((r_scause() == 15 || r_scause() == 13) &&
                vmfault(p->pagetable, p->sz, r_stval(),
                    (r_scause() == 13) ? 1 : 0) != 0) {
        // page fault on lazily-allocated page
+   // end-listing:ch16-trap
    } else {
        printk("usertrap(): unexpected scause 0x%lx pid=%d\n",
↳   r_scause(), p->pid);
        printk("          sepc=0x%lx stval=0x%lx\n", r_sepc(),
↳   r_stval());

```

If `cowfault` returns -1, memory ran out while copying. In `usertrap` the process falls through `vmfault`, which refuses a mapped page, and the final `else` kills it. In copyout the same -1 makes the system call return -1 and the process survives. A fork that cannot be paid for at the first user store kills the writer, which is what an unbackable lazy `sbrk` page already does upstream; a system call that meets the same shortage fails instead.

cowtest.c

The test program is ours, after MIT's 6.1810 copy-on-write test, and `build.sh` copies it into `xv6's user/` before the build:

```
// cowtest: copy-on-write fork, proven three ways. Without COW,
↳ "simple"
// cannot fork at all: the parent holds most of physical memory and a
// copying fork has nowhere to put the copy.
//
// After cowtest.c from MIT 6.1810's copy-on-write lab (Kaashoek,
↳ Morris,
// Cox; MIT license, as xv6's LICENSE file), with the comments
↳ rewritten
// for this chapter.
#include "kernel/types.h"
#include "kernel/memlayout.h"
#include "user/user.h"

// Allocate more than half of physical memory, then fork. Without COW
// the fork fails; with COW the child shares the pages, writes to
↳ each,
// and exits; the parent's copies are unchanged.
void
simpletest()
{
    uint64 phys_size = PHYSTOP - KERNBASE;
    int sz = (phys_size / 3) * 2;

    printf("simple: ");
    char *p = sbrk(sz);
    if (p == (char *)0xffffffffffffffffL) {
        printf("sbrk(%d) failed\n", sz);
        exit(-1);
    }
    for (char *q = p; q < p + sz; q += 4096)
        *(int *)q = getpid();

    int pid = fork();
    if (pid < 0) {
        printf("fork() failed\n");
        exit(-1);
    }
    if (pid == 0)
        exit(0);
    wait(0);
}
```

```

if (sbrk(-sz) == (char *)0xffffffffffffL) {
    printf("sbrk(-%d) failed\n", sz);
    exit(-1);
}
printf("ok\n");
}

// Three processes all writing COW memory: the counts reach 3, and
↳ each
// write copies. The parent's bytes survive the children's writes.
void
threetest()
{
    uint64 phys_size = PHYSTOP - KERNBASE;
    int sz = phys_size / 4;
    int pid1, pid2;

    printf("three: ");
    char *p = sbrk(sz);
    if (p == (char *)0xffffffffffffL) {
        printf("sbrk(%d) failed\n", sz);
        exit(-1);
    }
    pid1 = fork();
    if (pid1 < 0) {
        printf("fork failed\n");
        exit(-1);
    }
    if (pid1 == 0) {
        pid2 = fork();
        if (pid2 < 0) {
            printf("fork failed");
            exit(-1);
        }
        if (pid2 == 0) {
            for (char *q = p; q < p + (sz / 5) * 4; q += 4096) {
                *(int *)q = getpid();
            }
            for (char *q = p; q < p + (sz / 5) * 4; q += 4096) {
                if (*(int *)q != getpid()) {
                    printf("wrong content\n");
                    exit(-1);
                }
            }
        }
    }
}

```

```

    exit(-1);
}
for (char *q = p; q < p + (sz / 2); q += 4096) {
    *(int *)q = 9999;
}
exit(0);
}
for (char *q = p; q < p + sz; q += 4096) {
    *(int *)q = getpid();
}
wait(0);
pause(1);
for (char *q = p; q < p + sz; q += 4096) {
    if (*(int *)q != getpid()) {
        printf("wrong content\n");
        exit(-1);
    }
}
if (sbrk(-sz) == (char *)0xffffffffffffffffL) {
    printf("sbrk(-%d) failed\n", sz);
    exit(-1);
}
printf("ok\n");
}

char junk1[4096];
int fds[2];
char junk2[4096];
char buf[4096];
char junk3[4096];

// copyout() must fault a COW page in before writing to it: read a
// ↪ pipe
// into a page the child shares with the parent. Without the fault-in,
// the read writes through the shared page or fails.
void
filetest()
{
    printf("file: ");
    buf[0] = 99;
    for (int i = 0; i < 4; i++) {
        if (pipe(fds) != 0) {
            printf("pipe() failed\n");
            exit(-1);
        }
    }
}

```



```

int pid = fork();
if (pid < 0) {
    printf("fork failed\n");
    exit(-1);
}
if (pid == 0) {
    pause(1);
    if (read(fds[0], buf, sizeof(i)) != sizeof(i)) {
        printf("error: read failed\n");
        exit(1);
    }
    pause(1);
    int j = *(int *)buf;
    if (j != i) {
        printf("error: read the wrong value\n");
        exit(1);
    }
    exit(0);
}
if (write(fds[1], &i, sizeof(i)) != sizeof(i)) {
    printf("error: write failed\n");
    exit(-1);
}
}

int xstatus = 0;
for (int i = 0; i < 4; i++) {
    wait(&xstatus);
    if (xstatus != 0) {
        exit(1);
    }
}
if (buf[0] != 99) {
    printf("error: child overwrote parent\n");
    exit(1);
}
printf("ok\n");
}

int
main(int argc, char *argv[])
{
    simpletest();
    simpletest();
    threetest();
    threetest();
}

```

```

threetest();
filetest();
printf("ALL COW TESTS PASSED\n");
exit(0);
}

```

simple is the test COW exists for: it holds two thirds of memory and forks. Without COW its output is the simple: `fork()` failed above; with it, the child shares the parent's pages, exits without writing, and the parent gives the memory back with `sbrk(-sz)`. It runs twice because the second run starts from whatever state the first left in the allocator: a count that was not restored shows up there.

three has no failing counterpart without COW; against the unpatched kernel it prints three: ok, because a copying fork of 32 MiB three times fits. It exists to check the implementation: a parent and two descendants write the same pages in different orders, so the counts reach three and come back, and every process re-reads what it wrote and the parent re-reads after its children are gone. A count that is off by one makes a page be freed while a process still maps it, or never; a missed copy makes the parent see a child's 9999. `file` also passes on the unpatched kernel. Against the patched kernel without the copyout hunk it is the test that fails, and Exercise (c) shows how.

Exercises

(a) Delete `krefinc((void *)pa);` from `uvmcopy`, rebuild, and run `cowtest`. Say which line of the output is first and what the kernel was checking when it stopped.

Answer, p. 225

(b) Make `cowfault` always copy, even when the count is 1, so `cowtest` still passes; then count the extra copies with a counter in a scratch build. What do they cost?

Answer, p. 225

(c) Remove the copyout hunk and leave the rest of the patch. Run `file` alone (a scratch `cowtest` whose main calls only `filetest`). What fails first, and why there?

Answer, p. 226

(d) Chapter 4's PTE has G at bit 5 and A at bit 6. Make PTE_COW bit 6 and run `cowtest`. What happens, and does it show the bit is a safe choice?

Answer, p. 226

(e) Chapter 13's spawn is on ember's single page table. Of this patch's fourteen hunks, which would still mean something on ember, and which could not exist?

Answer, p. 227

(f) The count is read and written under `kmem.lock`. Name the race that removing the lock would allow when xv6 runs with three harts, and then try to catch it.

Answer, p. 227

Lab

```
make lab CH=16
```

```
make lab CH=16 IMPL=mime
```

`labs/ch16/build.sh` fetches `xv6-riscv` at commit `06aad25c735fd3159bdfae5680be4eab7` from <https://github.com/mit-pdos/xv6-riscv> into `${XDG_CACHE_HOME:-$HOME/.cache}/hp/ch16/`, outside the tree, resets that checkout to the pin on every run, applies every `*.patch` in the patch directory it is given, copies `cowtest.c` into the checkout's `user/` and builds the kernel and the file system image with xv6's own Makefile: `kernel/kernel` and `fs.img`, one hart (`CPUS=1`), the cross prefix of whichever of `riscv64-linux-gnu-` and `riscv64-unknown-linux-gnu-` is installed. The patch is the object you read and edit. A cached checkout is never edited by hand: the build resets it, so a change you want to keep goes in the patch (`git -C <checkout> diff` writes one).

The boot is xv6's own: `-machine virt -bios none -kernel out/kernel -m 128M -smp 1`, with `-global virtio-mmio.force-legacy=false` and the file system as a `virtio-blk` disk. `-snapshot` keeps QEMU's writes to the disk out of `out/fs.img`, and `build.sh` copies a fresh image every run, so no run starts from the file system the last one left. `input.txt` holds the two commands the shell runs: `cowtest`, then `usertests -q`. `run.sh` feeds them two seconds after QEMU starts. Fed at once, QEMU hands the first byte to the UART before xv6 has initialised it, and the driver discards it; the first run of this lab printed `owtest` and `exec owtest` failed.

xv6 has no `poweroff`. `run.sh` follows the console and stops QEMU when `ALL TESTS PASSED` or a `panic: line` appears; `timeout (300 s)` bounds a run that reaches neither. The capture is the console from xv6 kernel is booting to `ALL TESTS PASSED`. Three things in it are not what they look like:

- `usertests` provokes faults on purpose (writes through null, past `MAXVA`, into kernel memory) and the kernel reports each as a `usertrap(): unexpected` `scause` line with the process id and then `sepc` and `stval`. The process ids were identical in every run on both hosts (`straylight`: four transcripts including a cold rebuild; `talos`: three). If one ever drifts, add `s/pid=[0-9]*/pid=PID/` to `run.sh`'s `sed` beside the `sepc` and `stval` rules and re-capture; the verdicts and fault counts would stay pinned. The two addresses move when the toolchain does (the same source built with `riscv64-linux-gnu-` and with `riscv64-unknown-linux-gnu-` differs in code addresses), so `run.sh` replaces them with `ADDR`. Nothing else in the capture is edited.
- `$ cowtest` is followed at once by `usertests -q`: the console driver echoes characters when they arrive (`kernel/console.c:173-174`), not when the shell reads them, and both lines arrive together. The echo is the second command's, before the first one's output.
- A `usertests` test that provokes a fault prints `test NAME:` , then the kernel's `usertrap` lines, then `OK`: its verdict lands on a line of its own.

The capture is the transcript of `cowtest`'s six `ok` lines, `ALL COW TESTS PASSED`, then `usertests -q`'s verdicts and `ALL TESTS PASSED`. Two runs on `straylight` were byte-identical to each other and to `capture.txt`, and the build on `talos` matches it.

Timings, measured on 2026-09-30. `usertests -q` under QEMU's TCG (software CPU emulation): on `straylight` `run.sh` reports the pass marker 39 s after QEMU started (the two-second feed delay included); on `talos`, 89 s. A cold `make lab CH=16` on `talos` with an empty cache (the clone, the build and the run) took 1 min 27 s.

`IMPL=mine` applies `labs/ch16/mine/patches/0001-cow-fork.patch`, which is the reference patch with `cowfault`'s body replaced by `printk("MINE: cowfault stub\n"); return 0; (xv6's kernel prints with printk). The kernel builds and boots, the banner and $ cowtest match, and the transcript diverges at the line after usertests -q where simple: ok was expected: MINE: cowfault stub and usertrap(): unexpected scause 0xf pid=3, the shell's child; then the same for the shell, pid=2; then for init, pid=1; then panic: init exiting. The stub kills the parent as well because uvmcopy cleared W in the parent's PTEs, and the first store to the parent's stack after the fork returns faults. run.sh stops at the panic.`

Further reading

- xv6-riscv at commit 06aad25 (MIT PDOS) — the pinned commit; every file quoted above is read at it.
- *xv6: a simple, Unix-like teaching operating system* (Russ Cox, Frans Kaashoek, Robert Morris, rev4, August 2024) — the book that describes this design; its names for the kernel's functions predate the pinned commit's `k` prefixes.
- The RISC-V Instruction Set Manual, Volume II: Privileged Architecture (RISC-V International, v20260120) — the Sv39 page table entry, the RSW field, the accessed bit and `sfence.vma`.

Answers

(a) Observed, in a scratch copy of the pinned tree with the patch applied and that one line deleted, on one hart:

```
$ cowtest
simple: usertrap(): unexpected scause 0x2 pid=3
      sepc=0x1000 stval=0xb000
panic: kfree: refcount
```

The first casualty is not a test but the process running one: `pid=3` is `cowtest` itself (the shell is 2), killed for an illegal instruction (`scause 2`) after it has printed `simple: .` The panic is the check `if (PGREF(pa) < 1) panic("kfree: refcount")`: some page's count was already zero when a `kfree` reached it, which is a page freed twice. By the code, that is what the missing line allows. Without `krefinc` every shared page has a count of one while two page tables map it. The first share is `init`'s fork of `sh` at boot, before any shell exists; each later shell command repeats it. The child's `exec` releases the table it copied, which takes each count to zero and puts the parent's live pages on the free list, where the next `kalloc` (the new program's own pages) finds them. The damage starts before `cowtest`'s first fork, and the kernel does not stop until a page is released twice.

(b) Observed with two scratch builds, each with a pair of global counters (`cow_take` bumped where the reference takes the page over, `cow_copy` where a page is copied) printed from `kexit` when the exiting process is named `cow...`, one hart, one run of `cowtest` from boot. The reference build ends at `take=12295 copy=44251`. The always-copy build ends at `take=12798 copy=56546`, and `cowtest` prints `ALL COW TESTS PASSED` in both (the counters are kernel-wide since boot, so `take` in the second build counts the stores that *would* have been take-overs).

The difference in copies, $56546 - 44251 = 12295$, is exactly the reference build's number of take-overs: every one became a `kalloc`, a 4096-byte `memmove` and a `kfree`, 28 percent more copies in a run whose only work is writing memory. The cost is not correctness, which is why the always-copy kernel passes; it is that the take-over is the case COW exists to make free (a parent writing after its children are gone), and it is also the case that cannot fail for want of memory, which a copy can.

(c) Observed, in a scratch tree with the patch applied and the `if (*pte & PTE_COW) { ... }` block deleted from `copyout`:

```
$ cowtest
file: pipe() failed
$ error: read failed
```

Two failures, in two processes, and the first is not the read the test is about. `pipe(fds)` is a system call that stores two integers into the user's `fds` array, through `copyout`; `fds` is a global, so after the first fork its page is COW in the parent, and `copyout's (*pte & PTE_W) == 0` test refuses the store. That is the second iteration of the parent's loop. The child's `read(fds[0], buf, ...)` fails the same test on `buf`. With the patch intact, `file: ok`. The lesson is the breadth of the hunk: a system call that looks like it only reads (`pipe`) writes user memory, and every such call goes through the one function.

(d) `cowtest` prints ALL COW TESTS PASSED with `PTE_COW` as bit 6, and with bit 5; the test does not distinguish them, which is the reason a test is not an argument here. The privileged specification gives both bits a meaning the hardware acts on. Bit 6 is the accessed bit: "When a virtual page is accessed and the A bit is clear, the PTE is updated to set the A bit" (the scheme in force without the *Svade* extension, under which it faults instead). Bit 5, G, is a global mapping, and "marking a non-global mapping as global is a software bug". Linux's `arch/riscv/include/asm/pgtable-bits.h` at v7.2.8 (lines 16 to 19) says the same of the layout: `_PAGE_GLOBAL (1 << 5)`, `_PAGE_ACCESSED (1 << 6)` "Set by hardware on any access", `_PAGE_SOFT (3 << 8)` "Reserved for software".

The failure the bit-6 build allows is a store to read-only text. A scratch program forks, and the child stores into `main`. With `PTE_COW` at bit 8 the child is killed (`usertrap(): unexpected scause 0xf pid=4`) and the parent prints parent done. With bit 6, on the same kernel and QEMU: child stored into its own text. QEMU sets the accessed bit in the PTE when the text page is fetched from (the no-*Svade* scheme quoted above), and `cowfault` read that bit as `PTE_COW`; the page's count was two (parent and child), so it gave the child a private,

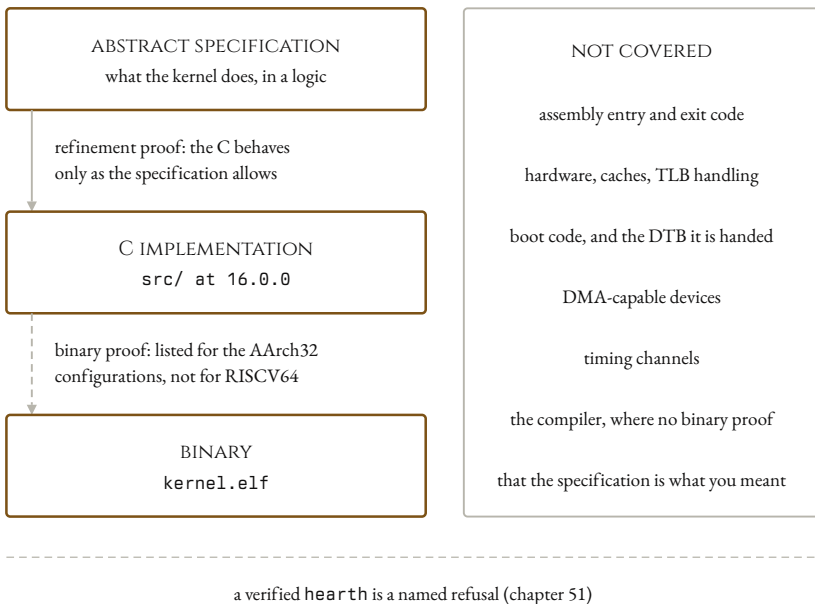
writable copy of its own code. Only RSW bits 8 and 9 are the software's.

(e) Read from the sources. The ones that cannot exist are the two `uvmcopy` hunks: it builds a child's table from a parent's, and `ember` has one table (`satp` is never switched, as `ember/proc.c` says in its header; chapter 13: "this kernel has one page table"), so there is no second table to point at the parent's pages, and a second mapping of the same page in the one table would have to be at a different virtual address, which a program linked for its own base cannot use. Every other hunk has a counterpart but nothing in `ember` to trigger it. `PTE_COW`: `ember/pt.h` defines the PTE bits through `PTE_D` (bit 7), so bit 8 is free. The count and the changed `kfree`: `ember/kalloc.c` is a free list with no metadata, and the comment on `kfree_page` says a page freed twice is pushed twice (chapter 9, Exercise (a), shows what the list then does); a count is the fix for that bug as well as for sharing. The trap hook: chapter 5 delegates page faults to `S` (`medeleg` bits 12, 13 and 15, `ember/trap.c`), but `trapper` handles only illegal instruction, the two interrupts and `ecall`, and anything else prints `S: unexpected scause=` and halts; a store page fault has no case. `copyout`: `ember/copyio.c`'s `copy_out_user` finds the physical page from the PTE's `V` and `U` bits and writes to it; it has no `W` test at all, so on a COW `ember` that function would be the first thing to change and would need more than `xv6`'s one hunk. `Makefile` and `defs.h` are `xv6`'s plumbing and only their names change.

(f) The race is two holders of one page changing its count at once. `PGREF(pa)++` is a load, an add and a store; so is `PGREF(pa)--`. A parent and its child share a page (count 2). The parent forks again, which raises it, while the child forks, which raises it, on two harts: both load 2, both store 3, and the page has four holders and a count of three. The last `kfree` frees it with one holder still mapping it. The same interleaving between a `krefinc` and a `kfree` has two outcomes: a lost release leaves the count too high and leaks the page, and a lost increment leaves it too low and frees the page early. The `krefcount(...) == 1` test in `cowfault` takes the lock to read the count but holds none across the decision that follows, and is safe for the reason given with that function: a count of one has no other holder that could change it. Any two holders will do: a parent and its child here, or two children of one parent forking at once.

Observed: with `acquire` and `release` removed from `krefinc` and from the decrement in `kfree`, `CPUS=3` and the harts up (`hart 2` starting, `hart 1` starting), three runs of `cowtest` in a row passed in each of two boots. The race is in the source and the tests do not trigger it, which is the use of a lock that the reader can only get by reading, not by running.

17. seL4 AND THE VERIFIED KERNEL



Why this layer exists

Chapter 14 placed five kernels on three axes, and the middle one, *who is the TCB*, was a count: every line that can set the U bit or the mode bit wrongly. seL4 is the kernel where that cell is answered by a proof. The book names it so that *verified* has a fixed meaning by the time chapter 51 lists a `verified heart` among the things it did not build: a statement about a specific kernel, of a specific form, resting on specific assumptions, and not a grade.

seL4 is also a hypervisor, and the book names that too so that nobody takes the next step. The white paper (revision 1.4.1 of 2026-07-17, fetched 2026-

10-01T01:36Z from <https://sel4.systems/About/sel4-whitepaper.pdf>) says that seL4 “runs in the appropriate hypervisor mode (EL2 on Arm, Root Ring-0 on x86, HS on RISC-V)” and hands VM exits to a user-level VMM. That is the design. This is the tree at the pin:

```
cd $C/sel4-16.0.0
grep -rn 'HypervisorSupport\|CONFIG_RISCV_HYP\|hyp' src/arch/riscv
↪ config.cmake kernel-config*
```

kernel-config* matches no file at 16.0.0. src/arch/riscv matches nothing. The only hits are config.cmake:314 and config.cmake:461, both KernelArmHypervisorSupport, which src/arch/arm/config.cmake:79 defines. The tree at 16.0.0 has no RISC-V hypervisor configuration, and the verified-configurations page says the same for the verified RISC-V64 configurations (“Hypervisor mode: No”). A kernel plus a user-level VMM is also a different shape from the one image that chapter 14 gave hearth, and hearth is not that.

This is a reading chapter. seL4 is not built: its build wants its own toolchain and Python environment, and what the chapter is about is what the proof says, not how to boot sel4test. The Lab section records the exception.

Mechanism

What is proved

A **refinement proof** shows that a concrete description of a system has no behaviour the abstract one forbids. The 2009 paper puts it as a correspondence “between a high-level (abstract) and a low-level (concrete, or refined) representation of a system”, and as a consequence: properties proved of the abstract model also hold of the refined one. The white paper’s version for seL4 is that the possible behaviours of the C code “are a subset of those allowed by the abstract model”. That is the first arrow in the figure: the abstract specification is the kernel’s behaviour written in higher-order logic, the C implementation is the code under src/, parsed into the same logic by a C parser that gives it a semantics, and the proof says the second refines the first.

The proofs sit on three different arrows, each from a page fetched on 2026-10-01 UTC:

- **Functional correctness**, the arrow in the figure: the C refines the abstract specification. The white paper’s gloss is that the kernel “cannot behave in ways that are not allowed by the spec” (<https://sel4.systems/About/sel4-whitepaper.pdf>, fetched 01:36Z), other than in time.

- **Binary correctness**, the dashed arrow: the compiled binary refines the C. A tool checks it by translation validation, an automatic comparison of the compiler’s output against the C’s logic rather than a manual proof.
- **Integrity and availability, and confidentiality**, a third arrow that the figure leaves out: they are proved of the abstract specification, not of the C, and carry down to the code through the refinement. Integrity: no entity modifies data without write access. Confidentiality: none reads or infers data without read access, in “a correctly configured system”. The white paper adds that the confidentiality proofs rule out covert storage channels and “presently not covert timing channels”, and that the integrity and availability proofs do not cover timeliness (same fetch).

Which claim holds for which architecture is in the verified-configurations page (<https://docs.sel4.systems/projects/sel4/verified-configurations.html>, fetched 2026-10-01T01:37Z). RISCv64: “C-level functional correctness, no fast path”, integrity and availability, and confidentiality. ARM (AArch32): the same C-level claim *including* the fast path, integrity and availability, confidentiality, and “binary correctness, covering C functions that have C-level verification”. AArch64: C-level functional correctness including the fast path, integrity and availability, and confidentiality, with the kernel “running in EL2” as a hypervisor. ARM_HYP and X64: C-level functional correctness only. The *fast path* is an optimised version of the two IPC system calls, `Call` and `ReplyRecv`: `traps.S` jumps to it (`c_handle_fastpath_call`, line 117) before the general system-call code when `CONFIG_FASTPATH` is set, and “no fast path” means the page does not claim it is covered. The tree and the page differ here, as they do for the binary claim: `configs/include/RISCV64_verified_include.cmake` sets `KernelFastpath ON` and `KernelBinaryVerificationBuild ON`, and the page’s list is the status. Exercise (e) reads the binary one.

The 2009 paper (SOSP 2009, fetched 01:41Z) was the first of the three, and it had an executable specification as a middle layer between the abstract one and the C. The chain the white paper draws today is abstract specification, C, binary.

What is assumed

The assumptions page (<https://sel4.systems/Verification/assumptions.html>, fetched 01:37Z) lists seven. Four matter to this book:

- **Assembly.** “the seL4 kernel, like all operating system kernels, contains some assembly code”, mainly “entry to and exit from the kernel, as well as direct hardware accesses”, and “For the proof, we assume this code

is correct”. On RISC-V that is `src/arch/riscv/traps.S`, `head.S`, and the inline assembly of `restore_user_context` in `c_traps.c` (the path to `sret`).

- **Hardware**, and, separately, **hardware management**: cache consistency and TLB maintenance are assumed to be done correctly in that assembly and to have no effect on the kernel’s behaviour. The `sfence()` after a page-table write (`vspace.c:1088` and `vspace.c:1131`) is that assumption being used.
- **Boot code**. The proof starts after the kernel has been loaded and brought into a consistent minimal state. The boot code is not part of it.
- **DMA**. The CPU and MMU are assumed to be the only devices that touch memory directly. A driver for a DMA-capable device has to be verified, or the proof does not carry over to it.

Two more apply to the configuration the book would actually run. The configurations page says that none of the verified configurations “take into account address translation for devices (System MMU or IOMMU), debug/profiling/printing interfaces, or the kernel startup at boot”. And where no binary proof is listed, nothing checks the compiler’s output: the assumptions page’s own sentence is that the compiler and linker need no trust “on architectures that are supported by our binary verification”.

The device tree appears on neither page as a name. It enters through the boot code, and the chapter’s own observation is where. `src/plat/qemu-riscv-virt/config.cmake` declares the board this book uses, `qemu-riscv-virt`, and builds its device tree by asking QEMU for it (`dumpdtb`, lines 60-65 and 133-138). That blob is the kind `ember` was handed in `a1` in chapter 2 and dumped in that chapter’s exercise (f). `seL4`’s `init_kernel` receives its address and size from the loader (`src/arch/riscv/kernel/boot.c:456-457`). The boot-time blob is outside the proof because the boot code is; the build-time DTB becomes constants in the preprocessed C (`kernel_all_pp.c`, the configurations page says) that verification does cover. And `grep -rnil 'libfdt\|fdt_' src` prints nothing at the pin: no kernel code parses it.

The kernel on the axes

Chapter 14’s three axes, for this kernel:

Shared. Nothing, by default. An address space is an object that a program holds a capability to, and `src/arch/riscv/kernel/vspace.c` is where the RISC-V one is made and edited.

The copy. An IPC message, which only an endpoint can carry (`src/object/endpoint.c`).

Trusted. The kernel, and the proof is about exactly that TCB. The word *capability* is chapter 14’s: an unforgeable reference to a kernel object that carries the right to use it. The design makes a capability the only way to reach an object: the white paper says that invoking a capability is “the one and only way of performing an operation on a system object”. What the proofs add is that access control is enforced: integrity and confidentiality, in a correctly configured system. Together the TCB question has an answer by type. There are two comparisons with what the reader built, and each is a rule, not a line count.

The first is chapter 7’s trap entry. `ember`’s `user_vec` begins `csrrw sp, sscratch, sp` (`ember/proc.S:25`): the hardware does not switch stacks on a trap, so the first instruction swaps the user’s `sp` for a kernel stack parked in `sscratch`, and nothing is stored before it. `seL4`’s entry has two forms. With SMP it is the same swap (`traps.S:28`); without SMP it swaps `t0` instead and leaves `sp` alone until every general register is saved and `sp` itself is stored as data in the frame. The kernel stack is then a link-time constant (`traps.S:94`). The rule is that no value the user supplied is used as an address before it has been put somewhere safe.

The second is chapter 4’s `map_page`. `ember`’s `map_page` is a function any kernel code can call with any virtual address, physical address and flags, and chapter 14’s TCB count for `ember` was every `map_page` call that passes `PTE_U` together with the paths that write `sstatus` or execute `sret`. `seL4`’s `map` functions, `performPageTableInvocationMap` and `performPageInvocationMapPTE` in `vspace.c`, are called from one place each, from the decode functions at `vspace.c:798` and `vspace.c:907`, and those are reached from `decodeRISCVMMUInvocation` (`vspace.c:933`), which switches on the type of the capability the caller invoked. No capability, no mapping, with one exception: the boot code builds the first user program’s tables without one (`map_it_pt_cap`, `vspace.c:167`, and `map_it_frame_cap`, `vspace.c:197`, both `BOOT_CODE`), which is the boot-code assumption again. Nothing is counted; the type system of the kernel’s own object model says there is one way in.

Walkthrough

Read in the README’s order. Each excerpt is `sed -n` from the pinned checkout, and the header names the lines.

1. src/plat/qemu-riscv-virt/config.cmake. The platform is declared in line 8, qemu-riscv-virt, for riscv64 and riscv32, with OpenSBI's generic platform (line 13). The kernel runs in S mode and M mode is OpenSBI's, where ember's -bios none image owns M mode itself. The device tree is not hand-written for this board:

```
// sel4 16.0.0, src/plat/qemu-riscv-virt/config.cmake:16-26
# If neither QEMU_DTS nor QEMU_DTB is set explicitly, the device
↳ tree is
# extracted from QEMU. This keeps it nicely up to date with the the
↳ actual
# QEMU versions that is used, and it's quite convenient for
↳ development.
if(NOT DEFINED QEMU_DTS)

    set(QEMU_DTS "${CMAKE_BINARY_DIR}/qemu-riscv-virt.dts")

    if(NOT DEFINED QEMU_DTB)

        message(STATUS "Extracting device tree from QEMU")
        set(QEMU_DTB "${CMAKE_BINARY_DIR}/qemu-riscv-virt.dtb")
```

Line 185 appends the extracted DTS and the overlay to KernelDTSList, and the top-level config.cmake (lines 410-424) concatenates that list into one DTS and compiles it to kernel.dtb.

2. src/plat/qemu-riscv-virt/overlay-qemu-riscv-virt.dts. The overlay changes three things. It names which nodes are kernel devices, adds a clint node whose size comes from QEMU's source, and reserves the first 2 MiB of DRAM, from 0x80000000, for SBI. That is the address ember's image is entered at with -bios none:

```
// sel4 16.0.0,
↳ src/plat/qemu-riscv-virt/overlay-qemu-riscv-virt.dts:7-20
/ {
    chosen {
        /*
        * - elfloader and kernel use SBI console by default
        * - QEMU emulates a SiFive PLIC and CLINT by default
        *
        * Nothing needed for elfloader
        *     sel4,elfloader-devices = ... ;
        *
        */
        sel4,kernel-devices =
```

```

    &{/soc/clint@20000000},
    &{/soc/plic@c0000000};
};

```

3. `src/arch/riscv/kernel/boot.c`. There is no M-to-S drop here, because the loader and OpenSBI did it. The kernel's first C, `init_kernel`, ends up in `init_cpu`, whose first act is the chapter 4 act, a kernel page table in force (`activate_kernel_vspace`, after `map_kernel_window` at line 227), and whose second is `stvec`, chapter 5's:

```

// sel4 16.0.0, src/arch/riscv/kernel/boot.c:126-143
BOOT_CODE static void init_cpu(void)
{

    activate_kernel_vspace();
    /* Write trap entry address to stvec */
    write_stvec((word_t)trap_entry);
    initLocalIRQController();
#ifdef CONFIG_KERNEL_MCS
    initTimer();
#endif

    /* disable FPU access */
    set_fs_off();

#ifdef CONFIG_HAVE_FPU
    init_fpu();
#endif
}

```

`head.S:42-52` documents the arguments the loader hands over in `a0-a5`, `a4` and `a5` being the DTB's address and size.

4. `src/arch/riscv/kernel/vspace.c`. The walk. `CONFIG_PT_LEVELS` is 3 by default (`src/arch/riscv/config.cmake:11`, whose comment ties 3 to Sv39), and the loop descends only while the slot it is looking at is a page table:

```

// sel4 16.0.0, src/arch/riscv/kernel/vspace.c:392-415
lookupPTSlot_ret_t lookupPTSlot(pte_t *lv1pt, vptr_t vptr)
{
    lookupPTSlot_ret_t ret;

    word_t level = CONFIG_PT_LEVELS - 1;
    pte_t *pt = lv1pt;

```

```

    /* this is how many bits we potentially have left to decode.
    ↪ Initially we have the
    * full address space to decode, and every time we walk this will
    ↪ be reduced. The
    * final value of this after the walk is the size of the frame
    ↪ that can be inserted,
    * or already exists, in ret.ptSlot. The following formulation is
    ↪ an invariant of
    * the loop: */
    ret.ptBitsLeft = PT_INDEX_BITS * level + sel4_PageBits;
    ret.ptSlot = pt + ((vptr >> ret.ptBitsLeft) &
    ↪ MASK(PT_INDEX_BITS));

    while (isPTEPageTable(ret.ptSlot) && likely(0 < level)) {
        level--;
        ret.ptBitsLeft -= PT_INDEX_BITS;
        pt = getPPtrFromHWPTE(ret.ptSlot);
        ret.ptSlot = pt + ((vptr >> ret.ptBitsLeft) &
    ↪ MASK(PT_INDEX_BITS));
    }

    return ret;
}

```

The map invocations dispatch on capability type:

```

// sel4 16.0.0, src/arch/riscv/kernel/vspace.c:933-942
exception_t decodeRISCVMMUInvocation(word_t label, word_t length,
    ↪ cptr_t cptr,
                                cte_t *cte, cap_t cap, bool_t
    ↪ call, word_t *buffer)
{
    switch (cap_get_capType(cap)) {

    case cap_page_table_cap:
        return decodeRISCVPageTableInvocation(label, length, cte,
    ↪ cap, buffer);

    case cap_frame_cap:
        return decodeRISCVFrameInvocation(label, length, cte, cap,
    ↪ call, buffer);
    }
}

```

5. src/arch/riscv/traps.S and c_traps.c. The entry, whose two stack forms Mechanism explains:

```

// sel4 16.0.0, src/arch/riscv/traps.S:24-38

```



```

BEGIN_FUNC(trap_entry)

#ifdef ENABLE_SMP_SUPPORT
/* The sscratch contains the stack for the current core */
    csrrw sp, sscratch, sp
/* Now we have a valid kernel stack */
    STORE t0, (-2*REGBYTES)(sp)
    LOAD  t0, (-1*REGBYTES)(sp)
#else
    csrrw t0, sscratch, t0
#endif
    STORE ra, (0*REGBYTES)(t0)
#ifdef ENABLE_SMP_SUPPORT
    STORE sp, (1*REGBYTES)(t0)
#endif

```

The kernel stack is loaded only after every register is saved (`traps.S:93-95`), and `restore_user_context` in `c_traps.c` writes `sscratch` back with the thread's register block before `sret` (`c_traps.c:77-80`), which is what makes the non-SMP swap in line 33 find the frame:

```

// sel4 16.0.0, src/arch/riscv/traps.S:92-95
#else
/* Load kernel's stack address */
    la sp, (kernel_stack_alloc + BIT(CONFIG_KERNEL_STACK_BITS))
#endif

```

6. `src/object/endpoint.c`. The sender's half of IPC. The case the message actually moves in is the receiver already waiting:

```

// sel4 16.0.0, src/object/endpoint.c:62-86
case EPState_Recv: {
    tcb_queue_t queue;
    tcb_t *dest;

    /* Get the head of the endpoint queue. */
    queue = ep_ptr_get_queue(epptr);
    dest = queue.head;

    /* Haskell error "Receive endpoint queue must not be empty"
       ↪ */
    assert(dest);

    /* Dequeue the first TCB */
#ifdef CONFIG_KERNEL_MCS
    tcbEPDequeue(dest, ep_ptr);

```

```

else
    queue = tcbEPDequeue(dest, queue);
    ep_ptr_set_queue(epptr, queue);

    if (!queue.head) {
        endpoint_ptr_set_state(epptr, EPState_Idle);
    }
#endif /* CONFIG_KERNEL_MCS */

    /* Do the transfer */
    doIPCTransfer(thread, epptr, badge, canGrant, dest);

```

doIPCTransfer is in `src/kernel/thread.c` and the word copy, copyMRs, in `src/object/tcb.c`, as chapter 14's Walkthrough says. This file is where the decision to copy is made.

Exercises

(a) The assumptions page has no entry named for the device tree. Where does the DTB enter a seL4 build and boot on this board, and which of the page's assumptions does each entry fall under?

Answer, p. 239

(b) Find the Sv39 level count in `vspace.c`, and say what the seL4 walk does at a level whose table does not exist, against `ember/pt.c`'s walk.

Answer, p. 240

(c) On the non-SMP path, what does `traps.S` do with the user's `t0` and `sp`, and what does that say about kernel stacks, against `ember/proc.S`?

Answer, p. 240

(d) A pipe in chapter 10 buffers: a write completes whether or not a reader is there. What does `sendIPC` do with a non-blocking send to an endpoint nobody is receiving on?

Answer, p. 241

(e) From the verified-configurations page: on which architectures is binary correctness listed, and is `riscv64` one of them at the fetch date? What does the tree say?

Answer, p. 241

(f) State in one sentence what “hearth is not verified” will mean in chapter 51, using the three boxes of the figure.

Answer, p. 241

Lab

A reading lab: nothing boots. Fetch the pinned tree (read-only; nothing in this repository builds it):

```
C=${XDG_CACHE_HOME:-$HOME/.cache}/hp/ch17
mkdir -p $C && git clone --depth 1 --branch 16.0.0
↳ https://github.com/seL4/seL4 $C/seL4-16.0.0
```

labs/ch17/README.md lists the six files in the order of the Walkthrough. Read them there, and answer the six exercises with the line that answers each. The exception to “labs boot” is recorded in DECISIONS.md (2026-09-30) as one of Part III’s two reading labs. seL4’s build needs its own toolchain and Python environment, and the chapter’s subject is the statement of the proof, which a sel4test run does not show.

Further reading

- seL4: Formal Verification of an OS Kernel, Klein et al., SOSP 2009 (DOI 10.1145/1629575.1629596): the first functional-correctness proof of an OS kernel, and the definition of refinement used above.
- The seL4 Microkernel: An Introduction, Gernot Heiser, seL4 Foundation, revision 1.4.1 of 2026-07-17: the verification story, the proof chain and the three assumptions, and seL4 as a hypervisor.
- What the Proofs Assume and Verified Configurations, seL4 Foundation: the assumption list and the per-architecture property list this chapter quotes.
- seL4 at 16.0.0, commit 6e7c3b733d296cfd88d5fbf635c96e447a882374: the tree the Walkthrough quotes by line range.

Answers

(a) Two places. At build time, the top-level config.cmake (lines 410–424) joins kernelDTSList, which for this board is QEMU’s dumped DTS plus the overlay, into kernel.dts and compiles it to kernel.dtb; tools/hardware/fdt.py:76–77 has a function that returns the nodes listed under seL4, kernel-devices, the property the overlay sets (overlay-qemu-riscv-virt.dts:17–19, the CLINT

and the PLIC). A wrong DTS here is closest to the page’s *hardware* assumption failing (the machine not as the model describes it), with the mistake made in the build instead of the silicon; that is the chapter’s own reading, not a line on the page. At boot, `init_kernel` receives `dtb_addr_p` and `dtb_size` (`boot.c:456-457`) and, if the size is not 0, copies the blob into the boot-info block for the first user program (`boot.c:314-324`, `SEL4_BOOTINFO_HEADER_FDT`). Nothing in the kernel parses it (`grep -rnil 'libfdt\|fdt_' src` prints nothing). The kernel passes the DTB on, so the party that must read it correctly at run time is the root task, which is user code. The boot entry is the page’s *boot code* assumption.

So “the DTB is outside the proof” is true, and what follows from it is narrower than it sounds. The kernel’s *run-time* behaviour does not depend on interpreting a DTB. The build does.

(b) Three. `CONFIG_PT_LEVELS` defaults to 3 (`src/arch/riscv/config.cmake:11`; its comment, lines 8–10, ties 2, 3 and 4 to Sv32, Sv39 and Sv48), and `lookupPTSlot` starts at `CONFIG_PT_LEVELS - 1` (`vspace.c:396`), the same three levels as `ember/pt.c:72`’s for (`level = 2`; `level >= 0`; `level--`). The difference is the missing level. `ember`’s `walk` calls `alloc_table()` when a slot is not valid (`pt.c:77-79`). `lookupPTSlot` never allocates: its loop (`vspace.c:407-412`) stops at the first slot that is not a page-table entry and returns it with `ptBitsLeft` saying how big a mapping that slot could take. `decodeRISCVFrameInvocation` compares that to the frame’s size and, if it differs, fails with `sel4_FailedLookup` and a `lookup_fault_missing_capability` (`vspace.c:860-863`). The caller must first map a page table into the missing level with the `RISCVPageTableMap` invocation (`vspace.c:709`), using a capability to a page-table object it already owns. In `ember` the kernel’s allocator is the policy. In `sel4` the policy is the user’s, and the kernel’s allocator is absent: that is the price of having no allocation in the walk that a proof has to cover.

(c) Mechanism gives the swap; what it leaves behind is the user’s `t0`, which sits in `sscratch` and is recovered by `csrr x1, sscratch` and stored at slot 4 (`traps.S:67-73`), with the user’s `sp` stored as data in slot 1 (line 37). Then every trap starts at the same stack top, `la sp, (kernel_stack_alloc + BIT(CONFIG_KERNEL_STACK_BITS))` (line 94). `ember` does not: its `user_vec` finds the process’s own kernel stack top in `sscratch` (`ember/proc.S:25`, parked by `proc.c:132`), because chapter 8’s `swtch` can leave a process asleep halfway down it. The frame in `sel4` lives in the thread control block, not on a stack, and a trap never resumes a half-run kernel path on an old stack. Whether that is why the proof is tractable is not something this file says; what it says is that the kernel has

no per-thread stack to reason about.

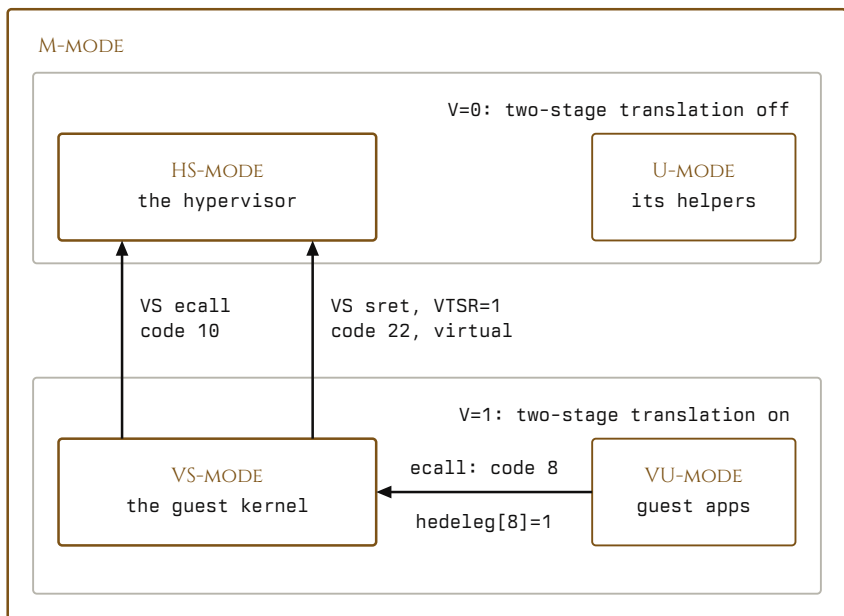
(d) It drops the message. The state is `EPState_Idle` and the first case (`endpoint.c:28-30`) does its work only if (`blocking`): it marks the sender `BlockedOnSend` and queues it on the endpoint (lines 32-57). For `blocking == false` there is no `else`, and control reaches `break` (line 60). Nothing was copied and nothing was queued. An endpoint has no buffer. A blocking sender waits with its message still in its own message registers, and the copy happens only in the `EPState_Recv` case (lines 62-86), at `doIPCTransfer` (line 86), when both threads are present. Chapter 10’s ring is a third party that holds bytes between a writer and a reader. An endpoint is a rendezvous, and the kernel is never the store.

(e) Binary correctness is listed under the `AArch32` (ARM) configurations and under no other (Mechanism has the lists; fetched 2026-10-01T01:37Z). So at that date `riscv64` is not one, and on `riscv64` the compiler and linker are in the trusted base (the assumptions page, above). The tree adds a wrinkle that the page does not resolve. `configs/include/RISCV64_verified_include.cmake:14` sets `KernelBinaryVerificationBuild ON`, but the option’s own help text (`config.cmake:177-183`) says enabling it “does NOT imply that you are using a verified kernel”. It restricts optimisations that would interfere with a binary proof. The white paper’s description of translation validation names an L3 model of the RISC-V ISA. The tooling for the step exists on RISC-V; the page’s property list is the status. And `qemu-riscv-virt` is not a platform in the `RISCV64` list, and `grep -rn 'qemu-riscv-virt|QEMURiscv' configs` finds nothing: `configs/RISCV64_verified.cmake` builds for `hifive`. The board this book uses is therefore not a verified configuration, which is why the lab reads it and does not claim it.

(f) `hearth` has no abstract-specification box and no refinement arrow from its `C` to one, so what its tests show is the behaviour of a run, and nothing in the book claims the `C` does only what a specification allows; the side box (its assembly entry and exit, its boot code and the DTB it is handed, the hardware under it) is the part of `hearth` that a verified `hearth` would still have to assume. The refusal is in the spec (`docs/superpowers/specs/2026-09-29-hp-course-design.md`, “Named refusals (ch 51)”: “a verified `hearth`”) and in the syllabus’s chapter 51 entry; the figure’s dashed rule is the same sentence.

PART IV — VIRTUALIZATION AS HARDWARE

18. THE TRAP-AND-EMULATE STORY, THEN HARDWARE



Popek and Goldberg's theorem, implemented as a mode bit

Why this layer exists

Chapter 7 built isolation from two objects: a page table whose kernel entries lack the U bit, and the privilege mode `sret` sets on the way down. Those two keep a *program* out of a kernel. A hypervisor has a harder tenant. It has to run a *kernel*, code written to own the machine: it writes `satp`, executes `sret`, reads the clock, idles with `wfi`. The hypervisor must let that code run at full speed and still not let it own anything. That needs a third object: a way to run supervisor-level code

under a layer that sees every instruction which would touch machine-owned state, or answers it from a private copy.

Popek and Goldberg worked out in 1974 when the hardware that already exists is enough: *trap-and-emulate* works if every instruction that touches such state traps. The Intel Pentium failed that test on seventeen instructions, which Robin and Irvine counted in 2000. x86 got its third object added later, as VT-x and SVM. The RISC-V hypervisor extension (the *H-extension*) was designed with it rather than retrofitted, and the RISC-V ISA is where this book writes its hypervisor: the course design (§3, the infrastructure table) names “RISC-V H-extension on QEMU virt” as the written hypervisor ISA, AArch64 EL2 as the second, and x86 VMX as an optional later dialect. That is why *hearth* is written on RISC-V first.

This chapter reads the H-extension as a claim about instructions: for each instruction a guest kernel executes, one of three things happens, and the table in the Walkthrough says which. The vocabulary defined here (*trap-and-emulate*, *sensitive* and *privileged* instructions, the virtualization mode *V*, HS-, VS- and VU-mode, the *virtual instruction exception*, *VMCS* and *VMCB*) is what chapters 19 through 22 and Part VII use without defining again.

Mechanism

Every section number below is from the pages as fetched on 2026-10-01 UTC: the privileged spec, hypervisor chapter, v20260120 (<https://docs.riscv.org/reference/isa/v20260120/priv/hypervisor.html>, fetched 04:45Z), and its machine chapter (.../priv/machine.html, same fetch). Quotes are verbatim. Section numbers in the machine chapter are written 2.1.x as that page numbers them. “§14.1” alone means the hypervisor chapter’s introduction, the text before §14.1.1.

Trap-and-emulate, sensitive, privileged

Trap-and-emulate is the technique: run the guest’s instructions directly on the real hart, at a privilege level below the hypervisor’s. When the guest executes something that would change or reveal machine state the hypervisor owns, the hart traps to the hypervisor, which emulates the instruction’s effect on the guest’s *virtual* copy of that state and resumes the guest after it. Almost every guest instruction is an add or a load and never traps; the cost is paid only where an instruction is about the machine.

Popek and Goldberg give the vocabulary, for a deliberately simplified machine with a supervisor/user mode and a relocation register (“Formal Requirements for Virtualizable Third Generation Architectures”, CACM 17(7), July 1974, pp. 412–421; page numbers below are the journal’s). An instruction is **privileged** if it traps when executed in user mode and does not trap in supervisor mode. The paper is explicit that the trap is the definition: an instruction that is merely a no-op in user mode “should not be called a privileged instruction; maybe ‘user mode NOP’ would be more accurate” (p. 415). An instruction is **sensitive** if it is *control sensitive*, meaning it attempts to change the resources available or “affects the processor mode without going through the memory trap sequence” (p. 415), or *behavior sensitive*, meaning the effect of its execution depends on the mode or on the relocation register (p. 416). Their basic theorem is Theorem 1 (p. 417):

For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.

Read it as a bug report waiting to happen. A sensitive instruction that is not privileged is a hole: the guest kernel executes it, it acts on the real hart, and the hypervisor never gets a trap. The formal definitions are for the paper’s simplified machine; this book uses *sensitive* in the looser sense that fits a real ISA: an instruction whose effect or result depends on, or changes, state the hypervisor owns.

What x86 did

Robin and Irvine (“Analysis of the Intel Pentium’s Ability to Support a Secure Virtual Machine Monitor”, 9th USENIX Security Symposium, 2000) applied the theorem to a real ISA. They analysed the Pentium’s “approximately 250 instructions” (abstract) and found that “seventeen instructions violate” their Requirement 3 (§3), their form of the theorem’s condition. The paper’s own summary of the consequence (§2.2) is the cost model of the whole subject: when a VM runs in user mode and a sensitive instruction is privileged, it traps and the VMM emulates it; “However, if sensitive, non-privileged instructions exist, it may be necessary for the VMM to examine all instructions before execution to force a trap to the VMM”. The standing example is `popf`, which pops a word into the flags register. Executed in a guest kernel that the VMM runs at user level, without enough privilege to change the interrupt flag, “an exception is not generated” (§3.1.3): the flag stays as it was and the guest never learns. The VMM never sees the instruction at all. `sgdt`, `siddt`, `sldt` and `smsw` (§3.1) fail the other way, reading machine state without a trap.

The fix was in hardware. Intel's VT-x and AMD's SVM add a second operating state in which guest code runs (the Intel manual's name is VMX non-root operation) and a memory-resident structure that holds the guest's processor state: the **VMCS** (virtual-machine control structure) on Intel, the **VMCB** (virtual-machine control block) on AMD. Of the Intel one, the SDM says only what this book needs: "VM entries load processor state from these fields and VM exits store processor state into these fields" (Vol. 3C §27.4). That is a trap frame the hardware fills in: the same object as chapter 5's `struct trapframe`, except that the hardware, not the entry stub, writes it. The SDM chapters are named in Further reading; nothing else in this book depends on them.

The H-extension: modes and V

The chapter's opening sentence of purpose: the extension "virtualizes the supervisor-level architecture to support the efficient hosting of guest operating systems atop a type-1 or type-2 hypervisor" (§14.1). It does so by changing S-mode into **HS-mode**, "hypervisor-extended supervisor mode", and adding a second S-mode and U-mode for guests:

HS-mode acts the same as S-mode, but with additional instructions and CSRs that control the new stage of address translation and support hosting a guest OS in virtual S-mode (VS-mode). Regular S-mode operating systems can execute without modification either in HS-mode or as VS-mode guests. (§14.1)

The S-mode ember has run in since chapter 3 is HS-mode on a hart that has the H-extension; `mret` picks it by `MPP=S` with `mstatus.MPV` clear (§14.1.6.4, Table 9). `drop_to_s` does not touch MPV, and the machine chapter (§2.1.4) does not specify its reset value, so the book relies on QEMU's reset value, and chapter 19's boot into HS depends on it: a hypervisor CSR write that does not trap is the proof that V was 0. What separates HS from VS is a new piece of hart state, the **virtualization mode V**, which §14.1.1 says tells whether the hart is executing in a guest:

When V=1, the hart is either in virtual S-mode (VS-mode), or in virtual U-mode (VU-mode) atop a guest OS running in VS-mode. When V=0, the hart is either in M-mode, in HS-mode, or in U-mode atop an OS running in HS-mode. The virtualization mode also indicates whether two-stage address translation is active (V=1) or inactive (V=0). (§14.1.1)

So there are five modes: M, HS and U with V=0, and VS and VU with V=1 (the

figure draws them). “HS-mode is more privileged than VS-mode, and VS-mode is more privileged than VU-mode” (§14.1.1). Two-stage translation, the second half of that paragraph, is chapter 19; here it only says $V=1$ is where it applies.

hstatus: the guest’s control panel

hstatus is HS-mode’s register for tracking and controlling the exception behaviour of a VS-mode guest (§14.1.2.1). The fields this chapter needs, with their bit positions on RV64 as the pinned Linux tree defines them (`arch/riscv/include/asm/csr.h:144–152` at `v7.2.8`, the tree chapter 15 builds):

Field	Bit	What it controls
SPV	7	Set by the hart on a trap into HS to the value of V at the time of the trap
SPVP	8	Set to the guest privilege (VS or VU) only when V was 1, otherwise unchanged; sets the effective privilege of the <code>h1v/hsv</code> accesses
HU	9	Whether the <code>h1v/h1vx/hsv</code> instructions may also run in U-mode
VTVM	20	Makes <code>satp</code> accesses and <code>sfence.vma</code> from VS trap
VTW	21	Makes <code>wfi</code> from VS trap
VTSR	22	Makes <code>sret</code> from VS trap

SPV is the hypervisor’s entry switch as well as the trap’s record: “the SPV bit in *hstatus* is set to the value of the virtualization mode V at the time of the trap. When an SRET instruction is executed when $V=0$, V is set to SPV.” To start or resume a guest, HS sets SPV to 1 and *sstatus.SPP* to 1, writes the guest’s program counter into *sepc*, and executes *sret*; the hart lands at that address in VS-mode with $V=1$ (§14.1.6.4, Table 10). The same instruction chapter 7 used to enter U enters a guest. SPVP is the guest-side counterpart of *sstatus.SPP*: when V was 1, “bit SPVP (Supervisor Previous Virtual Privilege) is set to the nominal privilege mode at the time of the trap, the same as *sstatus.SPP*”, and it also sets the effective privilege of the virtual-machine load and store instructions. Without it, “even with $HU=1$, U-mode could not access virtual machine memory at VS-level, because to enter U-mode using SRET always leaves *SPP*=0”. HU is for a hypervisor split into a small HS part and a larger U part: “The HU bit allows a portion of a hypervisor to be run in U-mode for greater protection against software bugs, while still retaining access to a virtual machine’s memory.” *hstatus* has more fields (GVA, VGEIN, VSBE, VSXL); GVA is chapter 19’s.

The three trap bits are one idea:

The `hstatus` fields `VTSR`, `VTW`, and `VTVM` are defined analogously to the `mstatus` fields `TSR`, `TW`, and `TVM`, but affect execution only in VS-mode, and cause virtual-instruction exceptions instead of illegal-instruction exceptions. (§14.1.2.1)

`mstatus.TSR`, `TW` and `TVM` already let M-mode make S-mode's `sret`, `wfi` and `satp/sfence.vma` trap (§2.1.1.6.6; “Trapping the WFI instruction can trigger a world switch to another guest OS, rather than wastefully idling in the current guest”). The H-extension gives HS the same three levers over VS, without going through M.

VS CSRs: the copy

A guest kernel is unmodified supervisor code: it writes `sstatus`, `stvec`, `satp`. If those writes reached the hart's real `sstatus`, `stvec` and `satp`, they would overwrite the hypervisor's own. The extension's answer is a second set of registers, chosen by V:

Furthermore, several *virtual supervisor* CSRs (VS CSRs) are replicas of the normal supervisor CSRs. For example, `vsstatus` is the VS CSR that duplicates the usual `sstatus` CSR.

When `V=1`, the VS CSRs substitute for the corresponding supervisor CSRs, taking over all functions of the usual supervisor CSRs except as specified otherwise. Instructions that normally read or modify a supervisor CSR shall instead access the corresponding VS CSR. (§14.1.2)

The replicas are `vsstatus`, `vsip/vsie`, `vstvec`, `vsscratch`, `vsepc`, `vscause`, `vstval` and `vsatp` (§14.1.2.11–§14.1.2.18), plus `vstimecmp` with the `Sstc` extension (§14.1.2.19). The real ones are not lost: while `V=1` they “retain their values but do not affect the behavior of the machine unless specifically documented to do so” (§14.1.2). A few supervisor CSRs have no replica (`senvcfg`, `scounteren`, `scontext`), and for those “Hypervisor software is expected to manually swap the contents of these registers as needed.” The guest's `csrwr satp` is an ordinary instruction that happens to write `vsatp`. This is the sense in which the guest's state is *virtualised* rather than *trapped*: no exit, no emulation, the hart addresses a different register.

An access in the other direction does trap. “When `V=1`, an attempt to read or write a VS CSR directly by its own separate CSR address causes a virtual-instruction exception” (§14.1.2), and the same holds for the H-level CSRs: the

guest cannot read `hstatus` to find out it is a guest.

The three outcomes, and the virtual instruction exception

A guest instruction that touches machine-owned state therefore ends one of three ways:

1. **It runs on the VS copy** (or, for `cycle` and `wfi`, on the real hart with no trap). `sret` with `VTSR=0` (§14.1.6.4: “When executed in VS-mode (i.e., `V=1`), `SRET` sets the privilege mode according to Table 11, in `vsstatus` sets `SPP=0`, `SIE=SPIE`, and `SPIE=1`, and lastly sets `pc=vsepc`”), the CSR accesses above, `time` and `cycle` reads when `hcounteren` and `mcounteren` allow them.
2. **It traps to the hypervisor as a *virtual instruction exception*.**
3. **It traps as its ordinary cause**, the same exception a non-virtualised hart would raise: `ecall` is the case that matters.

The **virtual instruction exception** is exception code 22. It is how the hardware tells HS that the instruction would have been valid in HS-mode but was stopped because `V=1`, which for a VS-mode guest kernel usually means one it is entitled to execute:

When `V=1`, a virtual-instruction exception (code 22) is normally raised instead of an illegal-instruction exception if the attempted instruction is *HS-qualified* but is prevented from executing when `V=1` either due to insufficient privilege or because the instruction is expressly disabled by a supervisor or hypervisor CSR such as `scounteren` or `hcounteren`. An instruction is *HS-qualified* if it would be valid to execute in HS-mode (for some values of the instruction’s register operands), assuming fields `TSR` and `TVM` of CSR `mstatus` are both zero. (§14.1.6.1)

The split matters because the two exceptions call for different handlers. A virtual instruction exception is a trap the hypervisor can emulate (the spec: “It is not unusual that hypervisors must emulate the instructions that raise virtual-instruction exceptions”): the guest did something legal for a kernel, and the hypervisor supplies the effect. “When not emulating the trapping instruction, a hypervisor should convert a virtual-instruction trap into an illegal-instruction exception for the guest virtual machine” (§14.1.6.1), so it can still be handed to the guest as an error. The spec adds a performance reason: “Machine level is expected ordinarily to delegate virtual-instruction traps directly to HS-level, whereas illegal-instruction traps are likely to be processed first in M-mode before

being conditionally delegated (by software) to HS-level.”

Exception codes and delegation

The extension adds codes to `mcause` and `scause`: “Codes are added for VS-level interrupts (interrupts 2, 6, 10), for supervisor-level guest external interrupts (interrupt 12), for virtual-instruction exceptions (exception 22), and for guest-page faults (exceptions 20, 21, 23). Furthermore, environment calls from VS-mode are assigned cause 10, whereas those from HS-mode or S-mode use cause 9 as usual.” (§14.1.6.1, Table 7). Five exception codes are new to this book:

Code	Table 7 name	Where it is used
10	Environment call from VS-mode	this chapter, chapter 19
20	Instruction guest-page fault	chapter 19
21	Load guest-page fault	chapter 19
22	Virtual instruction	this chapter
23	Store/AMO guest-page fault	chapter 19

The guest-page faults belong to the second translation stage; chapter 19 builds that stage and defines them. They are listed here because a hypervisor’s trap handler reads the code to tell all five apart.

Delegation is a chain, and the extension adds a link. “When a trap occurs in VS-mode or VU-mode, it goes to M-mode, unless delegated by `medeleg` or `mideleg`, in which case it goes to HS-mode, unless further delegated by `hedeleg` or `hideleg`, in which case it goes to VS-mode.” (§14.1.6.2). Which bits of `hedeleg` can be set is fixed by §14.1.2.2, Table 2: bit 8 (environment call from U-mode or VU-mode) is writable, and bits 9, 10 and 11 (environment calls from HS, VS and M) and bits 20–23 (the guest-page faults and the virtual instruction exception) are read-only zero. Two consequences. A guest never receives a virtual instruction exception or a guest-page fault. Those are HS’s when `medeleg` delegates them, otherwise M’s. And a VS-mode `ecall`, code 10, can never be delegated to the guest. The spec gives the reason the code exists, “HS-mode and VS-mode ECALLs use different cause values so they can be delegated separately” (§14.1.6.1), so that M can delegate one to HS and not the other; the reason the hypervisor cannot pass it on follows from what `ecall` is, a request to the layer above, which for VS is HS: “An HS-mode hypervisor is expected to implement the SBI for its VS-mode guest” (§14.1). The SBI, the Supervisor Binary Interface, is the calling convention by which supervisor code asks the layer below it for a service; chapter 19’s guest makes one such call. A VU-mode `ecall` is a different

matter: it is code 8, the guest kernel’s own system call, and `hedeleg[8]` lets HS pass it straight to the guest without a world switch. The figure’s three arrows are these cases; the two upward arrows reach HS when `medeleg` delegates codes 10 and 22 to it, and M otherwise.

Placing this on chapter 14’s axes

Chapter 14’s three axes, for the object this chapter describes. *What is shared:* the hart, and, from chapter 19, the memory it translates; most of the guest’s S-level registers are not shared, because each has a VS replica. *Who is trusted:* HS-mode and M-mode. The hypervisor depends on the guest kernel for nothing: the guest’s S-mode is not part of the hypervisor’s TCB, which is the point of the part. *Where the copy is:* the VS CSRs are the guest’s registers, held by the hart while $V=1$, and everything else (the guest’s general-purpose registers) is saved into a trap frame by HS’s entry code, the way chapter 5’s `trap.S` saves them for `ember`. On x86 the VMCS and VMCB are where the copy lives; on RISC-V it is a set of CSRs and a frame HS writes itself.

Walkthrough

There is no code. The table is the lab’s answer key: ten rows, each an instruction (or class) a guest kernel executes in VS-mode, what an unmodified S-mode kernel would see, and which of the three outcomes the H-extension gives it, with the spec section that says so. “Virtual instruction” is exception code 22.

In- struc- tion in VS- mode	Unmodified S-mode semantics	With $V=1$	Spec
<code>sret</code>	Returns from a trap: mode from <code>sstatus</code> . SPP, pc from <code>sepc</code> (chapter 7)	Copy if <code>hstatus.VTSR=0</code> : pops <code>vsstatus</code> , <code>pc=vsepc</code> . Virtual instruction if <code>VTSR=1</code> .	§14.1.6.4; §14.1.2.1

In- struc- tion in VS- mode	Unmodified S-mode semantics	With V=1	Spec
wfi	Stalls the hart until an interrupt may need service; may be a NOP (chapter 6)	Runs if VTW=0 and mstatus.TW=0. If VTW=1 and mstatus.TW=0: virtual instruction , unless it completes in an implementation-specific bounded time. If mstatus.TW=1: ordinary illegal instruction (code 2), likewise unless it completes in bounded time	§14.1.2.1; §14.1.6.1; §2.1.1.6.6
csrr/csrrsatp	Reads or writes the S-stage translation root (chapter 4)	Copy if VTVM=0: accesses vsatp; a write with an unsupported MODE is ignored. Virtual instruction if VTVM=1	§14.1.2.18; §14.1.2.1
sfence.vma	Orders and flushes address-translation caches for satp (chapter 4)	Copy if VTVM=0: applies to the current satp, which at V=1 is vsatp. Virtual instruction if VTVM=1	§14.1.3.2; §14.1.2.1
sstatus, sie, sip, stvec, sscratch, sepc, scause, stval	The S-level register	Copy : the instruction reads or writes the vs* replica; HS's own value is untouched	§14.1.2; §14.1.2.11–§14.1.2.17
csrr time, csrr cycle	Readable when the mcounteren bit is set, else illegal instruction	Both bits set: cycle runs on the real counter, and time is a copy that returns time + htimedelta. hcounteren bit clear and mcounteren bit set: virtual instruction . mcounteren bit clear: ordinary illegal instruction	§14.1.2.6; §14.1.2.7; §14.1.6.1; §2.1.1.11
ecall	A request to the layer above: from S-mode it is code 9	Ordinary : code 10, always; hedeleg[10] is read-only zero, so it cannot reach VS. Lands in HS if medeleg[10] is set, else in M	§14.1.6.1, Table 7; §14.1.2.2, Table 2; §14.1.6.2
hlv.*, hlvx. *, hsv.*	Do not exist; illegal instruction	Virtual instruction at V=1. In HS they are the hypervisor's way to read and write guest memory	§14.1.3.1; §14.1.6.1

In- struc- tion in VS- mode	Unmodified S-mode semantics	With V=1	Spec
hfence. vma, hfence. gvma csrr/csrrw of vsatp, hstatus, hedeleg ... by their own ad- dresses	Do not exist; illegal instruction Do not exist; illegal instruction	Virtual instruction at V=1; neither mstatus.TVM nor hstatus.VTVM makes hfence.vma trap in HS Virtual instruction at V=1: the guest cannot touch the replicas or the control CSRs directly	§14.1.3.2; §14.1.6.1 §14.1.2; §14.1.6.1

Three things the table shows that no row says alone.

First, the conditions sit in two places. VTSR, VTW, VTVM, hcounteren and hedeleg are HS's; mstatus.TW and mcounteren are M's. HS decides what is *virtual instruction* versus *copy*; M's bits decide whether the outcome becomes an *ordinary* illegal instruction instead. The spec closes the loop with one sentence: "Because TSR and TVM in mstatus are intended to impact only S-mode (HS-mode), they are ignored for determining exceptions in VS-mode" (§14.1.6.1). TW is not in that sentence, which is why wfi has the extra case above.

Second, VU-mode is stricter than VS. The spec's list (§14.1.6.1) has VU-mode sret and sfence instructions, supervisor CSR accesses that HS-mode could make, and wfi with mstatus.TW=0 all raise virtual-instruction exceptions, and bit 22 of hedeleg is read-only zero, so they are never the guest's (HS's when medeleg delegates code 22, otherwise M's): a guest's user code can never run a supervisor instruction.

Third, the three outcomes cover every row, and this is the check of the theorem for this table: no row is sensitive and silent. The spec does not state Popek and Goldberg's condition as a goal; the table is the reader's evidence for the instructions it lists, not a proof about the whole ISA.

Exercises

(a) Why does `popf` defeat trap-and-emulate on a 1990s Pentium, in Popek and Goldberg's terms? Name the two properties it has and the one it lacks.

Answer, p. 257

(b) Which `hstatus` bit makes `wfi` trap from VS-mode, what is the exception code, and which other CSR bit changes the code?

Answer, p. 258

(c) A VS-mode `ecall` lands in HS with code 10. What does a VU-mode `ecall` do when `hedeleg[8]=0`, and when `hedeleg[8]=1`, and what must be true of `medeleg` in both cases? Which `hstatus` and `vsstatus` fields change?

Answer, p. 258

(d) Name the VS CSR that replaces `satp` and the `hstatus` bit that makes the guest's accesses to `satp` trap. What happens to HS's own `satp` while the guest runs?

Answer, p. 258

(e) The VMCS is a trap frame. List the fields of `ember's struct trapframe` (chapter 5), and say which three VMCS guest-state fields they correspond to.

Answer, p. 259

(f) On chapter 14's axes, what does the H-extension add to the TCB of a hypervisor?

Answer, p. 259

Lab

A reading lab: nothing boots. `labs/ch18/README.md` has an empty table with the same ten rows as the Walkthrough, in the same order. For each row, fill in the three columns: what an unmodified S-mode kernel sees, which of the three outcomes (**copy**, **virtual instruction**, **ordinary**) the H-extension gives at `V=1` and under which `hstatus`, `mstatus` or counter-enable bits, and the section of the privileged spec that says so. Read the spec pages from the links in Further reading; the Walkthrough's table is the answer key, and a disagreement between your table and the spec's text is a finding to report, not to paper over. Chapter 19 is the first lab that boots a guest on this hart.

Further reading

- The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, hypervisor extension chapter (RISC-V International, v20260120; fetched 2026-10-01T04:45Z), and the machine chapter: §14.1.1 (modes and V), §14.1.2.1 (hstatus), §14.1.2.2 (hedeleg, Table 2), §14.1.6.1 (cause codes, the virtual instruction exception), §14.1.6.4 (trap return); §2.1.1.6.6 (TVM, TW, TSR) in the machine chapter.
- Formal Requirements for Virtualizable Third Generation Architectures, Gerald J. Popek and Robert P. Goldberg, CACM 17(7), July 1974, pp. 412–421: the definitions of privileged and sensitive instructions and Theorem 1. The DOI page refuses automated fetches (HTTP 403); the bibliographic record is confirmed by Crossref, and the text read for this chapter is the copy at https://www.princeton.edu/~rblee/ELE572Papers/Fall04Readings/secureOS/popek_virtualizable.pdf (fetched 2026-10-01T04:47Z).
- Analysis of the Intel Pentium’s Ability to Support a Secure Virtual Machine Monitor, John Scott Robin and Cynthia E. Irvine, 9th USENIX Security Symposium, 2000: §3 is the seventeen instructions; §3.1.3 is popf. The paper text was read from the USENIX proceedings PDF, https://www.usenix.org/legacy/publications/library/proceedings/sec2000/full_papers/robin/robin.pdf (fetched 2026-10-01T04:46Z).
- Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C (the manual set is at version 093 on the landing page, fetched 2026-10-01T04:46Z): Chapter 27, “Virtual Machine Control Structures”, §27.4 “Guest-State Area”; Chapter 30, “VM Exits”. The names are all this book uses.
- AMD64 Architecture Programmer’s Manual Volume 2: System Programming, AMD, publication 24593: Chapter 15, “Secure Virtual Machine”, and Appendix B, “Layout of VMCB”. The chapter and appendix names were read from revision 3.39 of the manual; the vendor’s document viewer does not render to a fetch, so the revision 3.45 link is verified by its search listing only.

Answers

(a) Robin and Irvine §3.1.3: executed in a VM running at user level with insufficient privilege, popf raises no exception, and the interrupt-flag bits are left unchanged, so a guest kernel’s attempt to re-enable interrupts is silently dropped

and the VMM never sees the instruction. It is *sensitive*: its effect on the flags depends on the privilege level the VM runs at (behavior sensitive, Popek and Goldberg p. 416), and, in this chapter's looser sense of the word, it changes state the monitor owns. It lacks the third property: it is not *privileged*. Popek and Goldberg's definition requires the user-mode execution to trap, and their p. 415 names the case that does not, "user mode NOP". Theorem 1 (p. 417) needs sensitive $\subseteq$ privileged, and popf is a counterexample to it. The hardware fix did not make popf trap: the guest's flags live in a field (Intel's guest-state area holds the guest RFLAGS, SDM Vol. 3C §27.4.1), and the guest's popf writes the guest's copy.

(b) VTW, bit 21 (csr.h:145 at v7.2.8). With VTW=1 and mstatus.TW=0, a VS-mode wfi raises a virtual-instruction exception, code 22 (§14.1.6.1), unless it completes within an implementation-specific, bounded time (§14.1.2.1). With mstatus.TW=1 the answer changes to code 2, an illegal-instruction exception: §2.1.1.6.6 says that when TW=1, a wfi executed in "any less-privileged mode" that does not complete in bounded time causes one, and §14.1.2.1 states VTW's effect only "assuming mstatus.TW=0". So mstatus.TW sits above VTW, and the code that arrives in HS depends on a bit the hypervisor may not own.

(c) A VU-mode ecall is code 8 (Table 7, "Environment call from U-mode or VU-mode"). In both cases medeleg[8] must be set; otherwise the trap goes to M, because the chain starts at medeleg (§14.1.6.2). Chapter 5's trap_init_m value (medeleg bits 2, 3, 8, 12, 13 and 15) sets it, the same bit chapter 7 uses to take a user ecall in S. With hedeleg[8]=0 the trap goes to HS: V becomes 0, hstatus.SPV=1 and sstatus.SPP=0 (Table 10, VU-mode), SPVP is set to the same value as SPP, 0, and sepc, scause, stval and the two hypervisor trap registers are written (§14.1.6.2): htval, which holds a faulting guest-physical address and is zero for an ecall (§14.1.2.8), and htinst, zero or a transformed copy of the trapping instruction (§14.1.2.9). With hedeleg[8]=1 the trap goes to VS: vsstatus.SPP=0 (Table 11), vsepc, vscause and vstval are written, and, in the spec's words, "hstatus and the HS-level sstatus are not modified, and the virtualization mode V remains 1". The second case is the one that costs HS nothing: the guest kernel sees an ordinary system call, as it would on bare metal. The corresponding VS ecall has no such case because hedeleg[10] is read-only zero (Table 2).

(d) vsatp (§14.1.2.18) and VTVM, bit 20 (§14.1.2.1; csr.h:146). With VTVM=1 the same bit also makes sfence.vma and sinval.vma trap, with code 22. HS's own satp is not overwritten and not consulted: the supervisor CSRs that have VS replicas "retain their values but do not affect the behavior of the machine

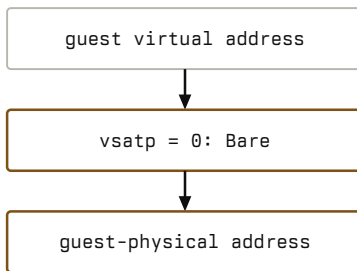
unless specifically documented to do so” (§14.1.2), so HS does not have to change its own `satp` to run a guest. HS may still choose to trap the guest’s `satp` writes. The reason is the one §2.1.1.6.6 gives for the machine-level TVM bit: trapping them “provides the hooks necessary to lazily populate shadow page tables”, the software technique that a second translation stage replaces.

(e) `struct trapframe` in `ember/trap.h` has 35 fields, 280 bytes (chapter 5): the 31 general-purpose registers `ra`, `sp`, `gp`, `tp`, `t0–t6`, `s0–s11`, `a0–a7`, and the four CSRs `sepc`, `scause`, `stval`, `sstatus`. Intel SDM Vol. 3C §27.4.1 lists the guest-state area’s register fields: control registers CR0, CR3 and CR4, DR7, “RSP, RIP, and RFLAGS”, the segment and descriptor-table registers, and some MSRs. Three line up: `sepc` with RIP, `sp` with RSP, `sstatus` with RFLAGS (RFLAGS holds the interrupt-enable flag, as `sstatus.SIE` does). The list has no other general-purpose register, so on VMX the guest’s remaining GPRs are the VMM’s to save in software, as `ember`’s entry stub saves the other 30 here. `scause` and `stval` have no guest-state counterpart: they record *why* the guest stopped, and the VMCS keeps that elsewhere than the guest-state area (the exit information fields; this chapter does not read those). On RISC-V the guest-side copy of all four is the VS CSRs, which the hart already holds.

(f) Nothing outside HS-mode and M-mode. The hypervisor is HS plus the firmware below it; the guest’s S-mode is below HS in the order §14.1.1 gives (“HS-mode is more privileged than VS-mode”), and the hardware routes every trap in `V=1` up the chain in §14.1.6.2. Table 2 keeps that chain closed in HS’s favour (the Mechanism section’s delegation paragraph): chapter 5’s `medeleg` delegates neither code 10 nor 20–23, and chapter 19’s gated branch adds them, so in this book they are HS’s. What is added to the TCB is configuration, not code: every `hstatus`, `hedeleg` and `medeleg` value HS or M writes decides who sees a trap. A `hedeleg[8]` set by mistake delivers a guest’s user `ecall` to the guest kernel when the hypervisor meant to intercept it. The guest’s own S-mode, a whole kernel in a conventional design, falls out of the TCB entirely, and that is the point of Part IV: chapter 14’s unikernel was trusted as a whole image for its own isolation, and under `hearth` no part of it is trusted by the thing that hosts it.

19. NESTED PAGING

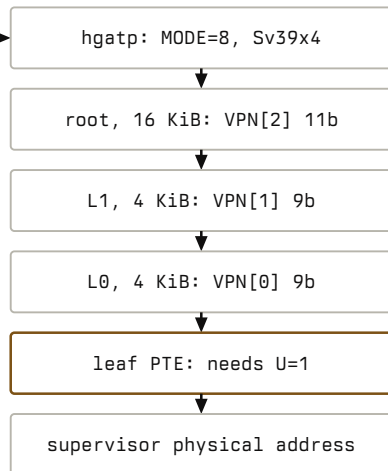
VS-STAGE, VSATP



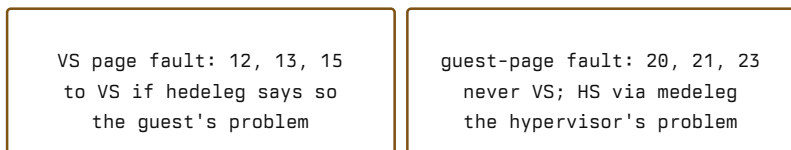
The guest's own tables sit in guest-physical memory: each read of them is G-stage translated too.

Chapter 19: vsatp = 0, so GVA = GPA and one walk runs.

G-STAGE, HGATP



WHEN A WALK FAILS



Two walks; a guest page fault is not a hypervisor page fault

Why this layer exists

Chapter 7 put a *program* under the kernel with one page table and the U bit. Chapter 18 said what a hypervisor needs beyond that: a way to run a *kernel* at full speed and still own the machine. The part of that problem this chapter solves is memory. A guest kernel builds page tables whose entries it believes name physical memory, and it believes that memory starts where its platform says and is as large as it was told. The hypervisor cannot let that be true of the real memory. The guest's "physical" pages have to be pages the hypervisor chose, and the guest must not be able to name any other. Before hardware had a second translation stage, the hypervisor got there by trapping every write to the guest's `satp` and every edit of its page tables, and keeping a *shadow* page table of its own that composed the two mappings. Chapter 18 met the hook for that technique, the `hstatus` bit that makes the guest's `satp` accesses trap. The second stage replaces the shadow table with a second hardware walk.

This is heartth chapter zero: the first place in the book where code the reader wrote runs a guest. HS-mode, which is *ember* after its drop to S on a hart that has the H-extension (chapter 18), builds an identity map for a guest of two pages, executes `sret` into VS-mode, and reads the guest's exit out of the trap frame. The guest is eleven instructions; there is no guest kernel, no device and no interrupt. What the lab shows is three things that stay true when the guest is Linux: a page table the guest does not own, a fault class the guest never sees, and a trap entry (chapter 5's `trap_entry`, unchanged) that catches the guest's exit. Chapter 20 gives the same job to KVM and runs the same payload bytes; there a memory slot replaces `gmap`, and KVM builds the G-stage from it a page at a time, on the guest's faults.

Mechanism

Section numbers and quotes are from the privileged spec, hypervisor chapter, v20260120 (<https://docs.riscv.org/reference/isa/v20260120/priv/hypervisor.html>), fetched 2026-10-01T05:26Z, with the machine chapter ([../priv/machine.html](https://docs.riscv.org/reference/isa/v20260120/priv/machine.html)) and the supervisor chapter ([../priv/supervisor.html](https://docs.riscv.org/reference/isa/v20260120/priv/supervisor.html), §11.x) from the same fetch. Quotes are verbatim.

Two stages, three addresses

When `V=1` every memory access goes through two translations. The terms below are defined here; chapter 18's modes and `V` are used as defined there.

For any virtual memory access, the original virtual address is converted in the first stage by VS-level address translation, as controlled by the `vsatp` register, into a *guest physical address*. The guest physical address is then converted in the second stage by guest physical address translation, as controlled by the `hgatp` register, into a supervisor physical address. The two stages are known also as VS-stage and G-stage translation. (§14.1.5)

So there are three kinds of address. A **guest virtual address** (GVA) is what a guest instruction names. A **guest-physical address** (GPA) is what the guest's own page tables, or no page table at all, turn it into; it is the guest's idea of physical memory. A **supervisor physical address** is a real address on the bus. The **VS-stage** is the walk the guest controls (its `vsatp`, its tables, which sit in guest-physical memory). The **G-stage** is the walk the hypervisor controls (`hgatp`, tables in real memory). The spec adds a sentence that decides this lab's shape: "Although there is no option to disable two-stage address translation when $V=1$, either stage of translation can be effectively disabled by zeroing the corresponding `vsatp` or `hgatp` register." The payload has no page tables, so `vsatp` is zero (MODE 0, Bare, as in `satp`): its GVAs are GPAs, and one walk, the G-stage, does all the work. The guest's own tables, when it has them, are read through the G-stage as well: "When $V=1$, memory accesses that would normally bypass address translation are subject to G-stage address translation alone. This includes memory accesses made in support of VS-stage address translation, such as reads and writes of VS-level page tables." (§14.1.5). A guest page-table walk of three levels therefore costs three G-stage walks of its own, one per level, plus one for the final address.

The two walks fail in two different ways, and which one failed decides who hears about it. A VS-stage failure is an ordinary page fault (codes 12, 13, 15). It goes to VS if `hedeleg` says so: Table 2 lists bits 12, 13 and 15 as writable, so the hypervisor may hand it to the guest, which is the guest's own memory management at work. A G-stage failure is a **guest-page fault**: code 20 for an instruction fetch, 21 for a load, 23 for a store or atomic. Table 2 lists bits 20 to 23 as read-only zero, and §14.1.5.2 says the same from the other side: "Guest-page-fault traps may be delegated from M-mode to HS-mode under the control of CSR `medeleg`, but cannot be delegated to other privilege modes." A guest never sees one. The guest believes it has memory at that address; the hypervisor decides it does not, and only the hypervisor is told. The figure's lower half is this rule: a guest page fault is not a hypervisor page fault, because the hypervisor's own page faults are the ordinary kind, from its own `satp`, and these are neither.

Sv39x4: the G-stage table

hgatp selects the G-stage scheme with its MODE field (Table 5; bits 63:60 on RV64, with the page number in bits 43:0, as the pinned Linux tree’s arch/riscv/include/asm/csr.h:167–170 has them, the tree chapter 15 builds): 0 is Bare (“guest physical addresses are equal to supervisor physical addresses”), 8 is Sv39x4. Sv39x4 is Sv39, the scheme chapter 4 built, with the incoming address widened by two bits:

To accommodate the 2 extra bits, the root page table (only) is expanded by a factor of four to be 16 KiB instead of the usual 4 KiB. Matching its larger size, the root page table also must be aligned to a 16 KiB boundary instead of the usual 4 KiB page boundary. Except as noted, all other aspects of Sv32, Sv39, Sv48, or Sv57 are adopted unchanged for G-stage translation. (§14.1.5.1)

The arithmetic: a guest-physical address is 41 bits (“Address bits 63:41 must all be zeros, or else a guest-page-fault exception occurs”), so the top-level index, VPN[2], is bits 40:30, eleven bits, not nine. $2^{11} \times 8 \text{ bytes} = 16384$ is the 16 KiB root; the two lower levels are Sv39’s, 512 entries and 4 KiB each, indexed by bits 29:21 and 20:12, with the low twelve bits the page offset. hgatp holds the root’s physical page number; because the root is 16 KiB aligned, “the lowest two bits of the physical page number (PPN) in hgatp always read as zeros”. A page at guest-physical `0x80401000`, as an example, has VPN[2] = 2, VPN[1] = 2, VPN[0] = 1: the same bits Sv39 would index with, except that the top index could now run to 2047. The PTE format is not changed: “G-stage address translation uses the identical format for PTEs as regular address translation, even including the U bit”.

ember’s root is a static array, unsigned long groot[2048] aligned to 16384. The two lower levels come from kalloc_page(), each a zeroed 4 KiB page.

The leaf: U, A and D

A G-stage leaf needs one flag that no chapter-4 leaf needed in this role, and it is the one the first mistake forgets. §14.1.5.1 says G-stage translation is Sv39’s algorithm, except that, among other things, “when checking the U bit, the current privilege mode is always taken to be U-mode”, and, in the next paragraph, “all memory accesses (including those made to access data structures for VS-stage address translation) are considered to be user-level accesses, as though executed in U-mode”. In Sv39 a U-mode access to a page whose U bit is clear is a page fault. In the G-stage every access is a U-mode access, so a leaf without U faults for every

guest access, the fetch included, however many of R, W and X it carries. `gmap` sets `PTE_U` on every leaf for that reason, and `exercise(a)` removes it.

Pointer entries (non-leaf) carry `V` alone, as in `Sv39`. The `A` and `D` bits are set at creation: the supervisor chapter (§11.1.3.1) defines two schemes, “The *Svade* extension: when a virtual page is accessed and the `A` bit is clear, or is written and the `D` bit is clear, a page-fault exception is raised”, and, without *Svade*, the hardware updates the bits. A G-stage fault of that kind is a guest-page fault. Setting `A` and `D` in the entry makes the lab independent of which scheme the hart implements. The `G` bit has no defined use in G-stage entries, and software is told to leave it clear.

hgatp and the fence

`hgatp` is written with `MODE 8` and the root’s page number. The `VMID` field, which tags cached translations per guest, is left zero: “The number of `VMID` bits is `UNSPECIFIED` and may be zero”, and the lab runs one guest. Then:

Note that writing `hgatp` does not imply any ordering constraints between page-table updates and subsequent G-stage address translations. If the new virtual machine’s guest physical page tables have been modified, or if a `VMID` is reused, it may be necessary to execute an `HFENCE.GVMA` instruction (see 14.1.3.2. Hypervisor Memory-Management Fence Instructions) before or after writing `hgatp`. (§14.1.2.10)

`hfence.gvma zero, zero` is the form with both operands `x0`: no address and no `VMID` named, so it covers every guest-physical address and every `VMID`. Here it is required, not optional: §14.1.3.2 says “If `hgatp.MODE` is changed for a given `VMID`, an `HFENCE.GVMA` with `rs1=x0` (and `rs2` set to either `x0` or the `VMID`) must be executed to order subsequent guest translations with the `MODE` change—even if the old `MODE` or new `MODE` is `Bare`.” The machine resets with `hgatp.MODE zero, Bare`, so the lab’s write is a `Bare` to `Sv39x4` change, and the fence also orders the stores that built the table before the first G-stage walk.

Entering the guest

Everything `hyp_lab` writes before `sret` is the guest’s initial state. In order:

- `hedeleg = 0` and `hideleg = 0`: nothing a guest does is handled by the guest; every trap from `VS` comes to `HS`. This is the only choice for a

payload that has no trap handler, and for the bits that matter here it is not a choice: Table 2 makes `hedeleg` bits 9 to 11 and 20 to 23 read-only zero (chapter 18).

- `vsatp = 0`: the VS-stage is Bare. `vsstatus = 0` and `vstvec = 0`: replicas of `sstatus` and `stvec` (chapter 18), zeroed, because the guest never reads or traps through them.
- `hstatus = VSXL | SPV`. `SPV` (bit 7) is the entry switch of chapter 18: `sret` at `V=0` sets `V` to `SPV`. `VSXL` (bits 33:32, `csr.h:141–142`) selects `VSXLEN`, the guest’s register width: “`VSXL` is a WARL field that is encoded the same as the `MXL` field of `misa`”, and the machine chapter’s Table 1 encodes 64 bits as 2, so `HSTATUS_VSXL64` is `2UL << 32`.
- `sstatus.SPP = 1` and `sstatus.SPIE = 0`. `sret` in HS-mode reads two bits to choose the mode it enters: “`SRET` first determines what the new privilege mode will be according to the values in `hstatus.SPV` and `sstatus.SPP`, as encoded in Table 10. `SRET` then sets `hstatus.SPV=0`, and in `sstatus` sets `SPP=0`, `SIE=SPIE`, and `SPIE=1`. Lastly, `SRET` sets the privilege mode as previously determined, and sets `pc=sepc`.” (§14.1.6.4). Table 10 maps (`SPV=1`, `SPP=1`) to VS-mode. `SIE=SPIE` then loads HS’s own interrupt-enable bit from the cleared one. `priv.h` names only the `mstatus` positions, `MSTATUS_SPP` and `MSTATUS_SPIE`; the bits sit at the same places in `sstatus`.
- `sepc = code`: the guest-physical address of its first instruction, which is also its physical address because the map is an identity map.
- `hyp_enter(data)`: the first argument register, `a0`, carries the data page’s guest-physical address into the guest; `sret` leaves every other register as it was.

Which side of V the lab starts on. All of this assumes the hart is in HS, `V=0`, when `hyp_lab` runs. `drop_to_s` does not set `mstatus.MPV`, and the machine chapter does not specify its reset value (chapter 18 says so); the book depends on QEMU’s. The proof is the first write: a hypervisor CSR write from a hart with `V=1` would be a virtual instruction exception, and one from a hart without the extension would be an illegal instruction. Neither happens, and `csrw hgatp` is the line that would show it. (Without the extension the run does not print an error: the illegal write enters `trapper`, whose first act reads `hstatus`, which traps again; the image hangs with nothing after the allocator line, exercise (f).) `misa` would show whether the extension is implemented but is M-only, and the S-mode kernel cannot read it, so the lab does not look.

The interrupt mask. The rule for when an interrupt reaches HS is in §14.1.2.3:

“An interrupt i will trap to HS-mode whenever all of the following are true: (a) either the current operating mode is HS-mode and the SIE bit in the `sstatus` register is set, or the current operating mode has less privilege than HS-mode; (b) bit i is set in both `sip` and `sie`, or in both `hip` and `hie`; and (c) bit i is not set in `hideleg`.” With $V=1$ the mode is VS or VU, less privileged than HS, so clause (a) holds whatever `sstatus.SIE` says; the gate that is left is clause (b), HS’s own `sie`. The M shim’s timer (chapter 6) raises a supervisor software interrupt every second; if `sie.SSIE` were set, the tick would be taken in the middle of the guest, with `hstatus.SPV` set. `hyp_lab` therefore clears `sie` bit 1 before entering. In the committed stage nothing sets that bit, so the clear changes nothing, and the lab ends in microseconds, long before a tick. It is there for the reader’s next stage. Observed on `talos` in a scratch copy with the payload spinning for a few seconds and `timer_on()` (chapter 6) called before the guest: without the mask, `S: tick 0x1` printed from inside the guest; with it, nothing. The exit is the same in both.

Which traps come to HS

A trap taken while $V=1$ starts at M and follows chapter 18’s delegation chain (§14.1.6.2): `medeleg` passes it to HS, then `hedeleg` to VS. With `hedeleg = 0`, the question is whether `medeleg` sends it to HS. `trap_init_m` delegates bits 2, 3, 8, 12, 13 and 15 (chapter 5), which says nothing about codes 10 or 20 to 23. Without them a VS-mode `ecall` goes to the M shim: observed on `talos`, a build with the gated line below made empty stops at M: `trap cause 0xa mtval=0x0` and the guest’s exit never reaches `hyp_trap`. So a VS `ecall` reaches HS only through `medeleg` bit 10, and the five bits the chapter adds are 10 (environment call from VS), 20, 21 and 23 (the guest-page faults), and 22 (virtual instruction). Like every `medeleg` write, this one happens in `trap_init_m`, from M, before the drop: from S the write is itself illegal (chapter 5).

The exit: an `ecall` that asks

A guest with no device has no way to say it is done. It cannot write the test finisher at `0x1000000`; the guest-physical map has two pages and the finisher is not in it, and a store there would be a guest-page fault, which is an error, not an exit. `ecall` is the one instruction that traps whatever the G-stage holds, and the RISC-V convention for what to ask is already written down. The SBI (chapter 18) fixes the registers: the extension ID goes in `a7`, the function ID in `a6`, arguments in `a0` onward, then `ecall`. The System Reset extension has the ID `0x53525354`, the ASCII bytes of “SRST”, and function 0, `sbi_system_reset(reset_type, reset_reason)`, with `reset_type 0` meaning shutdown

(SBI specification v3.0, System Reset Extension). The extension's text says who answers: "When supervisor software is running inside a virtual machine, the SBI implementation is provided by a hypervisor." Chapter 18 quoted the same fact from the privileged side (§14.1).

The payload therefore loads `a7 = 0x53525354`, `a6 = 0`, `a0 = 0` (shutdown), `a1 = 0` (reason: none), and executes `ecall`. From VS that is cause 10. `a1` is overwritten by the call's own argument, so the proof that the guest's load saw its own store is the other register, `t1`: the payload loads into `t1` before it builds the call, and `trap_entry` saved `t1` in the frame, where `hyp_trap` reads `tf->t1`. The trap frame is chapter 5's. The hardware did not switch register files, so the guest's registers were the hart's registers and `trap_entry` saved them like any other trap's.

What a guest-page fault reports

`scause` has the code. `stval` has "the faulting guest virtual address as usual", and `htval` has the faulting guest-physical address, shifted right by two: "When a guest-page-fault trap is taken into HS-mode, `htval` is written with either zero or the guest physical address that faulted, shifted right by 2 bits" (§14.1.2.8). The "either zero or" is the spec's licence for an implementation to report nothing; QEMU reports the address. `hyp_trap` prints `(htval << 2) & 0xfff`, the offset within the page, not the page: the page is wherever `kalloc_page` put it, and the capture pins behaviour, not layout. The two address bits shifted out are, by the spec, "ordinarily the same as the least-significant two bits of the faulting virtual address in `stval`"; the lab's accesses are 8-byte aligned so they are zero. A fault inside the guest's own page-table walk would report the address of the table entry that could not be read, not the address the guest asked for (§14.1.2.8); `vsatp` is Bare here, so that case cannot arise.

The sp the guest chooses

`hyp_enter` is `sret` and nothing else, so the guest starts with HS's `sp`, `gp`, `tp` and every other register as `hyp_lab` left them, and from the first instruction the registers are the guest's to change. That matters at the first trap. `trap_entry` does `addi sp, sp, -280` and stores the frame through whatever `sp` then holds, under HS's own `satp`; nothing checks that it is a stack. Whether the guest can reach that page through its G-stage is irrelevant: the store is HS's, so a guest can aim HS's frame store anywhere in HS's address space by setting `sp`. Observed on `talos` in a scratch copy: a payload that begins `li sp, 0x87ffe0d0` (the data page's address plus 280 minus 72, in this layout, so that the frame's `a0` slot

lands on the first word of the data page) ends with H: guest read back t1= 0x600d5eed and H: data page holds 0x0: HS's own trap entry overwrote the word the guest had stored, with the guest's a0. The payload never wrote 0x0 there. Chapter 7's sscratch swap (user_vec opens with csrrw sp, sscratch, sp, so sp becomes a kernel stack the kernel parked) is therefore mandatory for guest entry, not optional, and hearth adopts it in Part VII.

What EPT and NPT change

Nothing structural. Intel's EPT and AMD's nested page tables are the same idea under other names: a second table, rooted in a field of the guest's control structure (the VMCS or VMCB of chapter 18) instead of hgatp, with entries in the format of the first stage, walked on every guest access, with a fault class of its own that goes to the hypervisor and never to the guest (an EPT violation on Intel, a nested page fault on AMD). Where they differ from Sv39x4 is detail: permissions and memory types in the entry, accessed and dirty bits that can be turned off, a different root size. The names of the SDM and APM chapters are in Further reading; nothing else in this book depends on them.

Walkthrough

The payload, with the SBI call from the last section. `ember/payload.S` is built by its own rule in `ember/Makefile`: linked at address 0, flattened with `objcopy` to `build/payload.bin`, and embedded by `payload_blob.S` the way `user_blob.S` embeds a user program.

```
# The guest. Position-independent, no stack, no devices. It touches
↳ one
# data page through the G-stage (a0 is its guest-physical address),
# proves the read-back in t1, and asks the hypervisor to stop through
# the SBI system-reset call — which is an ecall, code 10, from VS.
.section .text
.globl _start
_start:
    li    t0, 0x600d5eed
    sd    t0, 0(a0)
    ld    t1, 0(a0)
    li    a7, 0x53525354      # SBI extension SRST ("SRST")
    li    a6, 0               # fid 0: sbi_system_reset
    li    a0, 0               # reset_type: shutdown
    li    a1, 0               # reset_reason: none
    ecall
1:    j     1b
```

The payload is eleven instructions and 0x2c bytes: each `li` of a 32-bit constant expands to `lui` plus `addiw`, so the two of them are four; `li a6, li a0` and `li a1` are one each; with `sd, ld, ecall` and the `j` that spins, eleven. `riscv64-linux-gnu-objdump -d ember/build/payload.elf` shows them, and the capture's `payload bytes=0x2c` is the same count times four.

`gmap` maps one page. The three indices are the walk of the Mechanism section; each level without an entry yet gets a fresh zeroed page from `kalloc_page`, entered as a pointer (`V` alone); the leaf is the physical page number shifted into the PTE with the caller's `R/W/X` and the four flags every leaf carries.

```
/*
 * Map one 4 KiB guest-physical page to a physical page in the G-stage
 * table. Same PTE format as Sv39 (§14.1.5: "identical format"), one
 * more root bit pair, and every leaf carries U: G-stage accesses are
 * always treated as user-level accesses.
 */
static void gmap(unsigned long gpa, unsigned long pa, unsigned long
↳ flags)
{
    unsigned long *t = groot;
    unsigned long i2 = (gpa >> 30) & 0x7ff;    /* 11 bits: Sv39's 9 +
↳ 2 */
    unsigned long i1 = (gpa >> 21) & 0x1ff;
    unsigned long i0 = (gpa >> 12) & 0x1ff;
    unsigned long *t1, *t0;

    if (!(t[i2] & PTE_V)) {
        unsigned long p = kalloc_page();
        if (!p) {
            uart_puts("H: gmap: out of pages\r\n");
            halt(1);
        }
        t[i2] = ((p >> 12) << 10) | PTE_V;    /* pointer: V only
↳ */
    }
    t1 = (unsigned long *)((t[i2] >> 10) << 12);
    if (!(t1[i1] & PTE_V)) {
        unsigned long p = kalloc_page();
        if (!p) {
            uart_puts("H: gmap: out of pages\r\n");
            halt(1);
        }
        t1[i1] = ((p >> 12) << 10) | PTE_V;
    }
}
```

```

    t0 = (unsigned long *)((t1[i1] >> 10) << 12);
    t0[i0] = ((pa >> 12) << 10) | flags | PTE_U | PTE_A | PTE_D |
↳   PTE_V;
}

```

hyp_enter is the whole of the entry; everything it relies on is set up by hyp_lab (its body is not a listing: it is the sequence the Mechanism section walked, in the same order).

```

# sret with hstatus.SPV=1 and sstatus.SPP=1 (set by hyp_lab) leaves HS
# for VS at sepc. Every other register is whatever HS left in it. The
# guest's sp is the register trap_entry will store HS's frame through,
# with HS's own satp: a guest can aim that store anywhere in HS's
↳   address
# space (chapter 19, "The sp the guest chooses"). Chapter 7's sscratch
↳   swap
# is mandatory for guest entry; hearth adopts it.
    .section .text
    .globl hyp_enter
hyp_enter:
    sret

```

hyp_trap runs at the top of trapper, through the gated branch below, before chapter 5's illegal-instruction, tick, device and system-call branches (trap.c and main.c also include hyp.h under the same HP_CH >= 19 guard). A trap is the guest's if hstatus.SPV is set: the hardware sets it on every trap into HS from V=1 (Table 10), and sret cleared it on the way in, and a trap taken from V=0 writes SPV=0 (Table 10), so a trap that HS itself takes, as chapter 5's illegal-instruction skip does, leaves it clear. hyp_trap owns every synchronous trap with SPV set: the exit, the guest-page faults and the virtual instruction exception have their own lines, and any other cause prints H: guest trap cause=... sepc offset=... and stops. Interrupts (scause bit 63) return 0 and take the branches they always took. The placement matters: an earlier version called hyp_trap after those branches, and a guest whose first word was .word 0xffffffff was caught by chapter 5's illegal-instruction branch, which skips the instruction and srets back into the guest, so HS silently emulated it as a no-op (exercise (g)).

```

/* Called first from trapper() when HP_CH >= 19, before every other
 * branch: a trap from VS is one whose hstatus.SPV is set (V was 1
↳   when it
 * was taken), and every synchronous one is this function's.
↳   Interrupts
 * (scause bit 63) return 0 and take trapper()'s own paths. */
int hyp_trap(struct trapframe *tf)

```

```

{
    unsigned long cause = csr_read(scause);

    if (!(csr_read(hstatus) & HSTATUS_SPV))
        return 0;                                /* not the guest's */
    if (cause & MCAUSE_INT)
        return 0;                                /* an interrupt: trapper's
        ↪ */

    if (cause == CAUSE_ECALL_VS) {
        uart_puts("H: VS ecall ext=");
        uart_puthex(tf->a7);
        uart_puts(" fid=");
        uart_puthex(tf->a6);
        uart_puts(" type=");
        uart_puthex(tf->a0);
        uart_puts("\r\n");
        uart_puts("H: guest read back t1=");
        uart_puthex(tf->t1);
        uart_puts("\r\n");
        uart_puts("H: data page holds ");
        uart_puthex(*(unsigned long *)guest_data);
        uart_puts("\r\n");
        uart_puts("PASS\r\n");
        halt(0);
    }
    if (cause == CAUSE_GPF_INST || cause == CAUSE_GPF_LOAD ||
        cause == CAUSE_GPF_STORE) {
        uart_puts("H: guest-page fault code=");
        uart_puthex(cause);
        uart_puts(" gpa offset=");
        uart_puthex((csr_read(htval) << 2) & 0xffff);
        uart_puts("\r\n");
        halt(1);
    }
    if (cause == CAUSE_VIRT_INST) {
        uart_puts("H: virtual instruction exception\r\n");
        halt(1);
    }
    /* Any other synchronous trap from the guest is not one HS
    ↪ emulates:
    * stop, rather than let a lower branch of trapper() resume it.
    ↪ */
    uart_puts("H: guest trap cause=");
    uart_puthex(cause);

```

```

    uart_puts(" sepc offset=");
    uart_puthex(csr_read(sepc) & 0xffff);
    uart_puts("\r\n");
    halt(1);
    return 1;
}

```

The two gated branches in `ember/trap.c`, quoted as they stand. The first is the delegation (it sits in `trap_init_m`, after chapter 5's `medeleg` value and before the write); the second is the call into `hyp_trap`, placed at the top of `trapper`, before chapter 5's branches.

```

#if HP_CH >= 19
    /* VS ecall and the guest-page faults come to HS (chapter 19). */
    medeleg |= (1UL << 10) | (1UL << 20) | (1UL << 21) | (1UL << 22) |
↳ (1UL << 23);
#endif

#if HP_CH >= 19
    if (hyp_trap(tf))
        return;
#endif

```

The chapter's boot stage in `ember/main.c`: chapter 13's prefix up to the allocator, then the hypervisor instead of `init`. The disk, the console and the file system are not started, so the lab needs no `input.txt` bytes and no drive.

```

mem_init();
trap_init();
uart_puts("S: kernel up\r\n");
kalloc_init();
hyp_lab();           /* builds the G-stage, enters VS; never
↳ returns */

```

The `Makefile` gates the new objects by chapter (`hyp.c` `hyp.S` `payload_blob.S` link only when `HP_CH >= 19`) so that every earlier capture is unchanged: linking later files into an earlier image moves symbols.

Exercises

Each exercise below was run on `talos` (QEMU 11.1.1, `riscv64-linux-gnu-GCC 16.2.1`) in a scratch copy of `ember/`, one edit at a time, with `-cpu rv64`, `h=true` unless it says otherwise. `hyp_trap` prints codes in hex: `0x14` is 20, `0x15` is 21, `0x17` is 23.

(a) Remove PTE_U from gmap's leaf and run. Which fault, at what offset, and why with R and X set?

Answer, p. 276

(b) Map the data page R only. Which fault does the payload take, and at which instruction?

Answer, p. 276

(c) Make the payload load from the page after the data page. What does it do, and what does a load from an address that is certainly unmapped report?

Answer, p. 276

(d) Set `hstatus.VTSR` (bit 22) before entering and put an `sret` at the start of the payload. What arrives, and what does `hyp_trap` print?

Answer, p. 277

(e) Replace the three-level map with one 1 GiB leaf at the root covering `0x80000000` to `0xC0000000` (root index 2), pass the address of that entry to the guest in `a0`, and have the payload read it. What does the guest see, and what does chapter 14's budget say about the leaf?

Answer, p. 277

(f) Run `make lab CH=19` with `-cpu rv64,h=false` in `extra.mk`. What does it print, and how does it end?

Answer, p. 277

(g) Put `.word 0xffffffff`, an illegal instruction, at the start of the payload. What happens, and what did it do before `hyp_trap` moved to the top of `trapper`?

Answer, p. 278

Lab

```
make lab CH=19
```

```
make lab CH=19 IMPL=mine
```

`ember/` is the reference; `labs/ch19/mine/` is a copy of it in which `gmap` is a stub that prints `MINE: gmap stub` and maps nothing. `labs/ch19/extra.mk` passes `-cpu rv64,h=true` explicitly: this QEMU accepts `h=true` and `h=false`, and the lab never relies on the default.

The first thing to observe is the failure. `IMPL=mime` prints the common prefix, then:

```
+MINE: gmap stub
+MINE: gmap stub
H: payload bytes=0x2c entering VS
-H: VS ecall ext=0x53525354 fid=0x0 type=0x0
-H: guest read back t1=0x600d5eed
-H: data page holds 0x600d5eed
-PASS
+H: guest-page fault code=0x14 gpa offset=0x0
```

(the diff of `capture.txt` against the run; the stub runs twice, once for each page). The guest's first fetch finds no G-stage entry: code 0x14, 20, the instruction guest-page fault, at offset 0. Before `gmap` has a body the guest cannot take a single step, and the fault says why in the spec's terms, not in `ember`'s. Writing the three index lines, the table allocations and the leaf is the lab; the failing run is how you know a leaf without U and a missing leaf both look the same from HS (exercise (a)).

With the reference the capture is the common prefix (chapters 2 to 9, through the allocator line), then:

```
H: payload bytes=0x2c entering VS
H: VS ecall ext=0x53525354 fid=0x0 type=0x0
H: guest read back t1=0x600d5eed
H: data page holds 0x600d5eed
PASS
```

The capture pins the payload's size, the registers it sent (`a7`, `a6`, `a0`), the value the guest read back from its own store, and the value HS reads in the same page through its own address, which is the same page. It does not print the address of the guest's pages or of the page-table pages (they are whatever `kalloc_page` returned), and on a fault it prints a page offset, not a page. The run is deterministic: two runs on `talos` are byte-identical to each other and to `capture.txt`.

Further reading

- The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, hypervisor extension chapter (RISC-V International, v20260120; fetched 2026-10-01T05:26Z): §14.1.5 (two-stage translation; §14.1.5.1 the G-stage walk, the U rule; §14.1.5.2 guest-page faults), §14.1.2.10 (`hgatp`, Table 5), §14.1.2.8 (`htval`), §14.1.2.1 (`hstatus`), §14.1.3.2 (`HFENCE.GVMA`), §14.1.6.2 and §14.1.6.4 (trap entry and return). The supervisor chap-

ter has the A and D bit rules (§11.1.3.1) and the machine chapter the interrupt registers (§2.1.1.9).

- RISC-V Supervisor Binary Interface Specification, RISC-V International, release 3.0 (ratified); the System Reset Extension is the file `src/ext-sys-reset.adoc` at the `v3.0` tag.
- Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C (order number 326019-093US, September 2026, the revision chapter 18 cites): Chapter 31, “VMX Support for Address Translation”, §31.3, “The Extended Page Table Mechanism (EPT)” (chapter and section names read from the PDF on 2026-10-01).
- AMD64 Architecture Programmer’s Manual Volume 2: System Programming, AMD, publication 24593: Chapter 15, “Secure Virtual Machine”, the Nested Paging section (numbered 15.25 in the copy of the manual that `sift` returned for the section name; the number moves between revisions, and the vendor viewer does not render to a fetch).

Answers

(a) Observed: H: `guest-page fault code=0x14 gpa offset=0x0`, code 20, the instruction fetch at the first byte of the code page. R and X do not help: by the U rule of the Mechanism section every G-stage access is a U-mode access. It is the same fault the lab’s `IMPL=mime` run gets from an empty map, and nothing in the fault tells a missing leaf from a leaf the access may not use: both are code 20 at the same offset.

(b) Observed: H: `guest-page fault code=0x17 gpa offset=0x0`, code 23, the store/AMO guest-page fault, with the payload’s first access to the data page, its `sd`, as the instruction that took it (the fetch of the code page succeeded, so the fault is not the code’s). Offset 0 is the data page’s address plus zero: `sd t0, 0(a0)`. The load that would have followed never ran. Permissions are checked in the G-stage the same way as in the VS-stage (§14.1.5.1: “Access type permissions—readable, writable, or executable—are checked during G-stage translation the same as for VS-stage translation”); a store to a read-only guest page is reported to the hypervisor as a guest-page fault, which is the mechanism dirty tracking and copy-on-write for guest memory are built on.

(c) The first attempt, `li t2, 0x1008, add t2, t2, a0, ld t1, 0(t2)` (the load offset is too large for the instruction’s 12-bit immediate, hence the add), did not fault: `t1` was `0x13b700553023`, the payload’s own code word at byte 8. `kalloc_init` pushes the pages onto the free list from the floor upward, so

the list head is the highest page and `kalloc_page` hands pages out downward: the code page, allocated first, sits directly above the data page, allocated second, and is mapped. The page after the data page is not an unmapped page, and it is the allocator's layout that decides, not anything in the map. With `li t2, 0x2008` and `ld t1, 0(t2)` instead (a guest-physical address below DRAM that `gmap` never maps), the payload takes H: guest-page fault code=0x15 gpa offset=0x8: code 21, the load guest-page fault, and an offset of 8, because `htval` carries the address of the access, not the start of the page. The two low address bits are not in `htval`; 0x2008's are zero.

(d) Observed: H: virtual instruction exception. Chapter 18's table: with `VTSR=1` a VS-mode `sret` raises a virtual instruction exception, code 22 (§14.1.2.1). `medeleg` bit 22, in the gated branch, is what lets it reach HS; `hedeleg` bit 22 is read-only zero (Table 2), so it could not have gone anywhere else. With `VTSR=0`, the same `sret` would not have trapped at all: it would run on `vsstatus` and `vsepc` (chapter 18's outcome 1) and, with both zero, resume the guest at address 0, in VU-mode (`vsstatus.SPP=0` on entry, Table 11, so the `sret` returns to VU). The payload does not get to find out: observed with `VTSR` clear, the next event is H: guest-page fault code=0x14 gpa offset=0x0, the fetch at guest-physical 0, which is unmapped.

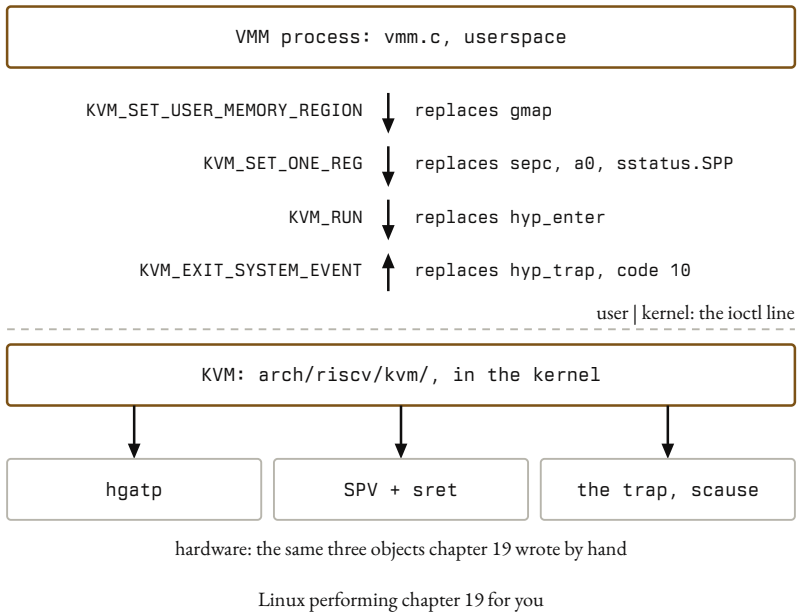
(e) With `groot[2] = ((0x80000000 >> 12) << 10) | R | W | X | U | A | D | V` and the payload's `sd` removed so that `t1` is the guest's read of its own G-stage entry, observed: H: guest read back `t1=0x200000df`. That is the leaf itself: PPN field 0x80000 shifted to bit 10, flags 0xdf. The guest is reading the hypervisor's page table, because `groot` is in the image at 0x80000000-and-up and the leaf maps all of DRAM, the hypervisor's text, data, stacks and tables included, readable and writable. On the budget: *what is shared* is everything, the guest's view of memory is the host's; *who is trusted* now includes the guest, which can rewrite `groot` and so its own map (the W bit on a leaf that covers the table is the whole attack); *where the copy is* disappears, which is why nothing had to be copied for the guest to read it. A superpage is the cheapest map to build and walk, and it gives the guest the host.

(f) QEMU accepts the flag. Observed on `talos`: nothing prints after S: `kalloc 0x80400000..0x88000000 pages=0x7c00`, not even H: payload bytes, and QEMU runs until timeout kills it (exit 124). With `-d int -D log` the first trap is `csrw hgatp (tval 0x68079073: csw with CSR number 0x680)`, an illegal instruction in S-mode on a hart without the extension. Then `csrrs x15, hstatus, x0 (tval 0x600027f3, CSR 0x600, hstatus)` traps, and traps again at the same address, forever. The cause is the placement from the last exercise:

every illegal-instruction trap enters `trapper`, whose first act is `hyp_trap's csr_read(hstatus)`, itself illegal without `H`, so the trap re-enters `trap_entry` 280 bytes lower on the same stack, without end. (Before `hyp_trap` moved to the top, chapter 5's handler skipped each `H` write in silence and the payload ran as plain `S` code, ending at `M: trap cause 0x9`.) The lesson: from chapter 19 on every trap path reads `hstatus`, so the image has no `H`-less fallback. The evidence that the extension is on is reaching `H: payload bytes`, and then the three `H: result` lines.

(g) Now: `H: guest trap cause=0x2 sepc offset=0x0` and exit 1: cause 2, illegal instruction, taken with `hstatus.SPV` set, at offset 0 of the code page, is not a trap `HS` emulates. Before the move (the committed first version, which called `hyp_trap` after the other branches), the run printed `S: trap sepc=0x87fff000 scause=0x2 stval=0xffffffff` and then the normal exit lines and `PASS`: chapter 5's illegal-instruction branch skipped the guest's instruction and `sreted` back into it. `HS` had silently run the guest's illegal instruction as a no-op. The lesson is the order of the dispatch: every branch of a handler that predates the guest answers for the guest too, unless the guest's check comes first.

20. KVM AS AN API



Why this layer exists

Chapter 19 wrote the hypervisor's three jobs by hand: a G-stage table (`gmap`), an entry (`hyp_enter`, an `sret` with `hstatus.SPV` set) and an exit handler (`hyp_trap`, which read the guest's `ecall` out of a trap frame). This chapter runs the same payload, byte for byte, under a hypervisor someone else wrote: KVM, the one built into Linux. The ordering is the point. KVM's interface is a set of `ioctl` calls, and an `ioctl` can be read as a plain API only by someone who does not know what is behind it. A reader who has just written `gmap` reads `KVM_SET_USER_MEMORY_REGION` as the call that makes `gmap` unnecessary, `KVM_RUN` as `hyp_enter` plus the wait for `hyp_trap`, and `KVM_EXIT_SYSTEM_EVENT` as the same

eca11 arriving a second way: chapter 19's `hyp_trap` read it from a trap frame in `ember`; here the VMM reads it from a shared page after `KVM_RUN` returns.

One ISA runs throughout. The kernel is chapter 15's pinned Linux, `riscv64`, built once more with KVM compiled in; QEMU provides the H-extension (`-cpu rv64,h=true`, as in chapter 19); the VMM is a static `riscv64 C` program, and the guest is the eleven-instruction payload. KVM does no general instruction emulation: it runs the guest on the hart, and decodes only the loads and stores that reach a device (chapter 21) and the few instructions the H-extension traps to it, `wfi` among them (exercise (a)). The lab's QEMU command has no `-accel`, so QEMU itself emulates the whole hart, H-extension included, and inside that emulated hart Linux executes `sret` into VS-mode exactly as `ember` did. Two levels are in play: the QEMU-booted Linux is the KVM host, and the payload is its guest. The ioctls this part uses (`KVM_CREATE_VM`, `KVM_CREATE_VCPU`, `KVM_SET_USER_MEMORY_REGION`, `KVM_RUN`) are architecture-independent; the register ids are not, and are the one place the lab is RISC-V's alone. `wick` (Part V) is a VMM of this shape on a Linux host with `/dev/kvm`; chapters 21 and 22 grow this program a device and a `virtqueue`.

Mechanism

Three words, defined where this chapter first needs them. A **vCPU** is the object KVM keeps for one guest hart: its register file, its CSR shadows and its run state, all in kernel memory, and the file descriptor userspace uses to set them and run them. A **memory slot** is a record “guest-physical range *G* is backed by this range of the VMM's own address space”, the unit in which the VMM hands a guest its RAM. A **VM exit** is the guest stopping for the hypervisor: on RISC-V, any trap from `V=1` into HS (chapter 18). Most are finished inside KVM and the guest re-enters; `KVM_RUN` returning to the VMM is the subset of exits that KVM cannot finish itself.

Three descriptors

`open("/dev/kvm")` gives a handle on KVM itself. The first call on it is the version check, and the documentation says what to do with the answer:

```
// linux v7.2.8, Documentation/virt/kvm/api.rst:139-146
:Returns: the constant KVM_API_VERSION (=12)
```

This identifies the API version as the stable `kvm` API. It is not expected that this number will change. However, Linux 2.6.20 and 2.6.21 report earlier versions; these are not documented and not

supported. Applications should refuse to run if `KVM_GET_API_VERSION` returns a value other than 12. If this check passes, all ioctls described as 'basic' will be available.

`KVM_CREATE_VM` on that descriptor returns a second one, the VM. It “has no virtual cpus and no memory”:

```
// linux v7.2.8, Documentation/virt/kvm/api.rst:156-158
:Returns: a VM fd that can be used to control the new virtual machine.
```

The new VM has no virtual cpus and no memory.

`KVM_CREATE_VCPU` on the VM descriptor returns the third, the vCPU. Three descriptors, three levels: `ioctl` on the first is about the whole of KVM (`KVM_GET_API_VERSION`), on the second about one guest’s memory and devices, on the third about one hart.

The memory slot feeds the G-stage

Chapter 19’s gmap walked an `Sv39x4` table and wrote a leaf for each page. The VMM does none of that. It mmaps anonymous memory in its own address space and tells KVM that this range is the guest’s RAM at a guest-physical address:

```
// linux v7.2.8, Documentation/virt/kvm/api.rst:1381-1384
This ioctl allows the user to create, modify or delete a guest
↳ physical
memory slot. Bits 0-15 of "slot" specify the slot id and this value
should be less than the maximum number of user memory slots supported
↳ per
VM. The maximum allowed slots can be queried using
↳ KVM_CAP_NR_MEMSLOTS.
```

The last two sentences of the next paragraph decide how the VMM then loads the guest:

```
// linux v7.2.8, Documentation/virt/kvm/api.rst:1398-1402
Memory for the region is taken starting at the address denoted by the
field userspace_addr, which must point at user addressable memory for
the entire memory slot size. Any object may back this memory,
↳ including
anonymous memory, ordinary files, and hugetlbfs. Changes in the
↳ backing
of the memory region are automatically reflected into the guest.
```

The slot’s backing memory is the VMM’s, and it stays the VMM’s. The lab’s fread of the payload into the slot writes the guest’s code page; its read of mem +

0x1000 after the exit reads the guest's data page. No copy crosses a boundary; the VMM and the guest share the pages, which is chapter 14's "shared" axis spent in the open.

What the `ioctl` does not do is build the G-stage table. `kvm_arch_init_vm` calls `kvm_riscv_mmu_alloc_pgd` when the VM is created: a zeroed root of `kvm_riscv_gstage_pgd_size` bytes, the four-times-wide root chapter 19 declared as `groot`. The table's depth is the widest G-stage mode the hart supports (`kvm_riscv_gstage_max_pgd_levels`), so on a hart with a wider mode than Sv39x4 the table is deeper than chapter 19's three levels; the root is 16 KiB either way.

```
// linux v7.2.8, arch/riscv/kvm/mmu.c:664-682
int kvm_riscv_mmu_alloc_pgd(struct kvm *kvm)
{
    struct page *pgd_page;

    if (kvm->arch.pgd != NULL) {
        kvm_err("kvm_arch already initialized?\n");
        return -EINVAL;
    }

    pgd_page = alloc_pages(GFP_KERNEL | __GFP_ZERO,
                           get_order(kvm_riscv_gstage_pgd_size));
    if (!pgd_page)
        return -ENOMEM;
    kvm->arch.pgd = page_to_virt(pgd_page);
    kvm->arch.pgd_phys = page_to_phys(pgd_page);
    kvm->arch.pgd_levels = kvm_riscv_gstage_max_pgd_levels;

    return 0;
}
```

The slot is checked when it is registered (`kvm_arch_prepare_memory_region`, same file, line 171, rejects a range outside the guest-physical space the table can address and a writable slot over a read-only mapping) and the table is filled in later, one page at a time, by the fault the guest takes when it first touches the page. A guest-page fault, the fault class chapter 19 showed going to HS and never to the guest, is where KVM finds the slot:

```
// linux v7.2.8, arch/riscv/kvm/vcpu_exit.c:25-31
fault_addr = (trap->htval << 2) | (trap->stval & 0x3);
gfn = fault_addr >> PAGE_SHIFT;
memslot = gfn_to_memslot(vcpu->kvm, gfn);
hva = gfn_to_hva_memslot_prot(memslot, gfn, &writable);
```

```

if (kvm_is_error_hva(hva) ||
    (trap->scause == EXC_STORE_GUEST_PAGE_FAULT && !writable))
    ↪ {

```

If the guest-physical address belongs to a slot, KVM finds the host page behind it and installs a G-stage leaf, then resumes the guest without returning to the VMM:

```

// linux v7.2.8, arch/riscv/kvm/vcpu_exit.c:65-71
ret = kvm_riscv_mmu_map(vcpu, memslot, fault_addr, hva,
    (trap->scause ==
    ↪ EXC_STORE_GUEST_PAGE_FAULT) ? true
    ↪ : false,
    &host_map);

if (ret < 0)
    return ret;

return 1;

```

`kvm_riscv_mmu_map` is `gmap` made lazy: it walks the same table, allocates the lower levels from a per-vCPU cache, and writes the leaf with the permissions `kvm_riscv_gstage_map_page` chose. Chapter 19's first mistake was a leaf without `U`. KVM's protection values are Linux's own page-protection macros, and every one of them is built on `_PAGE_BASE`:

```

// linux v7.2.8, arch/riscv/include/asm/pgtable.h:167-175
#define _PAGE_BASE          (_PAGE_PRESENT | _PAGE_ACCESSED |
    ↪ _PAGE_USER)

#define PAGE_NONE            __pgprot(_PAGE_PROT_NONE |
    ↪ _PAGE_READ)
#define PAGE_READ            __pgprot(_PAGE_BASE | _PAGE_READ)
#define PAGE_WRITE           __pgprot(_PAGE_BASE | _PAGE_READ |
    ↪ _PAGE_WRITE)
#define PAGE_EXEC            __pgprot(_PAGE_BASE | _PAGE_EXEC)
#define PAGE_READ_EXEC       __pgprot(_PAGE_BASE | _PAGE_READ
    ↪ | _PAGE_EXEC)
#define PAGE_WRITE_EXEC      __pgprot(_PAGE_BASE |
    ↪ _PAGE_READ |
                                _PAGE_EXEC | _PAGE_WRITE)

```

`_PAGE_USER` is in the base of every combination, so every G-stage leaf KVM writes has `U=1` (`gstage.c` picks among these in lines 239 to 249). The rule from §14.1.5.1 chapter 19 quoted holds for Linux too, and Linux satisfies it by routing G-stage leaves through the user-page macros.

When the address has no slot, the fault is not a bug. It is a device: a load or store to an address nobody backed with memory is exactly what a memory-mapped register looks like from inside a guest. KVM decodes the instruction (the hardware's `htinst` register, or a read of the guest's instruction) and fills `kvm_run` with what the guest tried to do. This is where a device model takes over (chapter 21 defines the term and builds one); this chapter's VMM reports the exit and fails. An instruction fetch from an unbacked address is different: the same function reflects an instruction access fault back into the guest (`vcpu_exit.c:41-58`), as a bare machine with no memory there would, rather than passing a fetch up as a device.

Registers: the id encoding

A vCPU's registers are reached with `KVM_SET_ONE_REG` and `KVM_GET_ONE_REG`, each taking a `struct kvm_one_reg` of a 64-bit id and a pointer. RISC-V's ids are documented in `api.rst`:

```
// linux v7.2.8, Documentation/virt/kvm/api.rst:2787-2791
RISC-V core registers represent the general execution state of a Guest
↪ VCPU
and it has the following id bit patterns::
```

```
0x8020 0000 02 <index into the kvm_riscv_core struct:24> (32bit
↪ Host)
0x8030 0000 02 <index into the kvm_riscv_core struct:24> (64bit
↪ Host)
```

The documentation lists the register behind each index, and the table starts:

```
// linux v7.2.8, Documentation/virt/kvm/api.rst:2798-2800
0x80x0 0000 0200 0000 regs.pc    Program counter
0x80x0 0000 0200 0001 regs.ra    Return address
0x80x0 0000 0200 0002 regs.sp    Stack pointer
```

The index is the word offset of the register inside `struct kvm_riscv_core`, and the header defines the structure and the macro that computes the index from a member name:

```
// linux v7.2.8, arch/riscv/include/uapi/asm/kvm.h:65-68
struct kvm_riscv_core {
    struct user_regs_struct regs;
    unsigned long mode;
};

// linux v7.2.8, arch/riscv/include/uapi/asm/kvm.h:257-259
```



```
#define KVM_REG_RISCV_CORE                (0x02 <<
↳ KVM_REG_RISCV_TYPE_SHIFT)
#define KVM_REG_RISCV_CORE_REG(name)      \
        (offsetof(struct kvm_riscv_core, name) /
↳ sizeof(unsigned long))
```

struct user_regs_struct is Linux's ptrace register layout, pc first, then ra, sp, gp, tp, t0 to t2, s0, s1, a0 to a7, s2 to s11, t3 to t6: regs.pc is index 0, regs.a0 is 10, regs.a7 is 17, and mode follows the 32 registers at index 32. offsetof takes a member path, so KVM_REG_RISCV_CORE_REG(regs.a0) is valid C; the full id is KVM_REG_RISCV (0x8000000000000000, the architecture tag) ORed with KVM_REG_SIZE_U64 (0x0030000000000000, the register is 8 bytes), KVM_REG_RISCV_CORE (0x02 << 24, the register group) and the index: 0x8030000002000000 for pc, which is the 0x8030 0000 02 <index> pattern above.

The mode register is the guest's privilege mode and KVM_RISCV_MODE_S is 1. What it sets is the bit chapter 19 set by hand: sstatus.SPP in the vCPU's saved state, the second of the two inputs to sret (hstatus.SPV, the first, KVM sets itself):

```
// linux v7.2.8, arch/riscv/kvm/vcpu_onereg.c:250-256
    else if (reg_num == KVM_REG_RISCV_CORE_REG(mode)) {
        if (reg_val == KVM_RISCV_MODE_S)
            cntx->sstatus |= SR_SPP;
        else
            cntx->sstatus &= ~SR_SPP;
    } else
        return -ENOENT;
```

KVM's reset state already has both bits (kvm_riscv_vcpu_context_reset sets them), and exercise (a) reads that.

KVM_RUN and the shared page

KVM_RUN has no argument. The result comes back through a page shared with the kernel, which the VMM maps from the vCPU descriptor, its size given by KVM_GET_VCPU_MMAP_SIZE:

```
// linux v7.2.8, Documentation/virt/kvm/api.rst:412-416
This ioctl is used to run a guest virtual cpu. While there are no explicit parameters, there is an implicit parameter block that can be obtained by mmap()ing the vcpu fd at offset 0, with the size given by KVM_GET_VCPU_MMAP_SIZE. The parameter block is formatted as a 'struct kvm_run' (see below).
```

The first fields of struct kvm_run are the exit reason and, in a union, the detail

for each reason:

```
// linux v7.2.8, include/uapi/linux/kvm.h:224-234
```

```
struct kvm_run {
    /* in */
    __u8 request_interrupt_window;
    __u8 HINT_UNSAFE_IN_KVM(immediate_exit);
    __u8 padding1[6];

    /* out */
    __u32 exit_reason;
    __u8 ready_for_interrupt_injection;
    __u8 if_flag;
    __u16 flags;
```

KVM_RUN returns 0 when the guest has stopped for a reason userspace has to see, and `run->exit_reason` says which. Everything else a trap can be, KVM handles and re-enters: a guest-page fault with a slot behind it, an SBI call it implements, a timer. The kernel dispatches in one function, `kvm_riscv_vcpu_exit`, and its cases are the same traps chapter 19's `hyp_trap` took, behind the same test: a trap is the guest's if `hstatus.SPV` is set.

```
// linux v7.2.8, arch/riscv/kvm/vcpu_exit.c:251-260
```

```
case EXC_INST_GUEST_PAGE_FAULT:
case EXC_LOAD_GUEST_PAGE_FAULT:
case EXC_STORE_GUEST_PAGE_FAULT:
    if (vcpu->arch.guest_context.hstatus & HSTATUS_SPV)
        ret = gstage_page_fault(vcpu, run, trap);
    break;
case EXC_SUPERVISOR_SYSCALL:
    if (vcpu->arch.guest_context.hstatus & HSTATUS_SPV)
        ret = kvm_riscv_vcpu_sbi_ecall(vcpu, run);
    break;
```

The entry is the other half. KVM's switch function restores the guest's registers and ends the way `hyp_enter` did:

```
// linux v7.2.8, arch/riscv/kvm/vcpu_switch.S:194-206
```

```
/*
 * Parameters:
 * A0 <= Pointer to struct kvm_vcpu_arch
 */
SYM_FUNC_START(__kvm_riscv_switch_to)
    SAVE_HOST_GPRS

    SAVE_HOST_AND_RESTORE_GUEST_CSRS .Lkvm_switch_return
```

```
RESTORE_GUEST_GPRS
```

```
/* Resume Guest using SRET */
sret
```

That `sret` is chapter 19's `hyp_enter`, with a swap of the host's and guest's register files around it (`SAVE_HOST_GPRS`, `RESTORE_GUEST_GPRS`) that `ember` skipped because `hyp_1ab` never returns: HS had nothing in its registers worth keeping. Exercise (e) lists what else KVM does that chapter 19 did not need to.

Exits: the two this part uses

The exit reasons are a long list (`kvm.h`:152-195, 0 to 43 at the tag). This part uses two.

`KVM_EXIT_SYSTEM_EVENT` (24) is the guest asking the platform to stop, restart or suspend. On RISC-V it is not a special instruction; it is an SBI call. The SBI is chapter 18's and the system-reset call chapter 19's: `a7 = 0x53525354`, `a6 = 0` for `sbi_system_reset`, `a0 = 0` for shutdown. KVM, not the VMM, is the SBI implementation the guest sees. Its table lists the extensions it implements, and the first entries are the ones a Linux guest needs from its firmware:

```
// linux v7.2.8, arch/riscv/kvm/vcpu_sbi.c:36-64
static const struct kvm_riscv_sbi_extension_entry sbi_ext[] = {
    {
        .ext_idx = KVM_RISCV_SBI_EXT_V01,
        .ext_ptr = &vcpu_sbi_ext_v01,
    },
    {
        .ext_idx = KVM_RISCV_SBI_EXT_MAX, /* Can't be disabled
        ↪ */
        .ext_ptr = &vcpu_sbi_ext_base,
    },
    {
        .ext_idx = KVM_RISCV_SBI_EXT_TIME,
        .ext_ptr = &vcpu_sbi_ext_time,
    },
    {
        .ext_idx = KVM_RISCV_SBI_EXT_IPI,
        .ext_ptr = &vcpu_sbi_ext_ipi,
    },
    {
        .ext_idx = KVM_RISCV_SBI_EXT_RFENCE,
        .ext_ptr = &vcpu_sbi_ext_rfence,
    },
}
```

```

    },
    {
        .ext_idx = KVM_RISCV_SBI_EXT_SRST,
        .ext_ptr = &vcpu_sbi_ext_srst,
    },
    {
        .ext_idx = KVM_RISCV_SBI_EXT_HSM,
        .ext_ptr = &vcpu_sbi_ext_hsm,
    },

```

An `ecall` from VS arrives at the `EXC_SUPERVISOR_SYSCALL` case above and goes to `kvm_riscv_vcpu_sbi_ecall`, which finds the extension by the value of `a7` and calls its handler. The `SRST` handler, for shutdown:

```

// linux v7.2.8, arch/riscv/kvm/vcpu_sbi_replace.c:157-162
case SBI_EXT_SRST_RESET:
    switch (type) {
        case SBI_SRST_RESET_TYPE_SHUTDOWN:
            kvm_riscv_vcpu_sbi_system_reset(vcpu, run,

```

`KVM_SYSTEM_EVENT_SHUTDOWN,`

```

            ,
            reason);

```

`kvm_riscv_vcpu_sbi_system_reset` stops every vCPU and writes the exit into the shared page:

```

// linux v7.2.8, arch/riscv/kvm/vcpu_sbi.c:150-169
void kvm_riscv_vcpu_sbi_system_reset(struct kvm_vcpu *vcpu,
                                     struct kvm_run *run,
                                     u32 type, u64 reason)
{
    unsigned long i;
    struct kvm_vcpu *tmp;

    kvm_for_each_vcpu(i, tmp, vcpu->kvm) {
        spin_lock(&tmp->arch.mp_state_lock);
        WRITE_ONCE(tmp->arch.mp_state.mp_state,
        ↪ KVM_MP_STATE_STOPPED);
        spin_unlock(&tmp->arch.mp_state_lock);
    }
    kvm_make_all_cpus_request(vcpu->kvm, KVM_REQ_SLEEP);

    memset(&run->system_event, 0, sizeof(run->system_event));
    run->system_event.type = type;
    run->system_event.ndata = 1;

```

```

run->system_event.data[0] = reason;
run->exit_reason = KVM_EXIT_SYSTEM_EVENT;
}

```

ndata is 1, data[0] is the reset reason from a1. The handler sets retdata->uexit (it is the line after the call at `vcpu_sbi_replace.c:163`), which makes `kvm_riscv_vcpu_sbi_ecall` return 0 and KVM_RUN return to userspace without advancing sepc. The guest's registers stay as they were at the ecall: the VMM can read a7, a6 and t1 after the exit and finds the same values `hyp_trap` found in the trap frame in chapter 19.

This is why the guest needs no firmware of its own. The host kernel runs on OpenSBI (QEMU's default firmware for virt; chapter 19's ember ran with `-bios none` and was its own firmware); the guest has KVM, which answers its ecalls itself.

KVM_EXIT_MMIO (6) is the device exit and chapter 21's subject. Its payload in `kvm_run` and the code that fills it:

```

// linux v7.2.8, include/uapi/linux/kvm.h:274-280
/* KVM_EXIT_MMIO */
struct {
    __u64 phys_addr;
    __u8 data[8];
    __u32 len;
    __u8 is_write;
} mmio;

// linux v7.2.8, arch/riscv/kvm/vcpu_insn.c:598-616
/* Update MMIO details in kvm_run struct */
run->mmio.is_write = true;
run->mmio.phys_addr = fault_addr;
run->mmio.len = len;

/* Try to handle MMIO access in the kernel */
if (!kvm_io_bus_write(vcpu, KVM_MMIO_BUS,
                    fault_addr, len, run->mmio.data)) {
    /* Successfully handled MMIO access in the kernel so
     ↳ resume */
    vcpu->stat.mmio_exit_kernel++;
    kvm_riscv_vcpu_mmio_return(vcpu, run);
    return 1;
}

/* Exit to userspace for MMIO emulation */
vcpu->stat.mmio_exit_user++;

```

```
run->exit_reason = KVM_EXIT_MMIO;

return 0;
```

`kvm_io_bus_write` first offers the access to devices that live inside the kernel; only if none claims it does `KVM_RUN` return with `KVM_EXIT_MMIO`. When the VMM resumes the vCPU after a read, `kvm_arch_vcpu_ioctl_run` completes the load with the bytes the VMM wrote into `run->mmio.data` before re-entering the guest:

```
// linux v7.2.8, arch/riscv/kvm/vcpu.c:872-876
switch (run->exit_reason) {
case KVM_EXIT_MMIO:
    /* Process MMIO value returned from user-space */
    ret = kvm_riscv_vcpu_mmio_return(vcpu, vcpu->run);
    break;
```

This chapter's loop prints an MMIO exit and gives up; exercise (b) provokes one.

What this VMM does not do, and where it runs

It injects no interrupts, and no lab in this part does (chapter 22's guest driver polls; injection, through `hvip` on the hardware and an `ioctl` chapter 21 names, is Part V's and Part VII's). It starts one vCPU, loads no kernel and no device tree, and does not handle `KVM_EXIT_RISCV_SBI`, the exit KVM uses for extensions it forwards rather than implements. The H-extension has to be present: make doctor checks that this host's QEMU accepts `-cpu rv64,h=true`, and a host without it fails there, not as a hang here. The lab needs no KVM on the machine you run it on, only QEMU with `h=true`: the KVM is the one in the QEMU-booted Linux, built with `CONFIG_KVM=y`, and the first line of its output is the check that it worked, `/dev/kvm` existing in that host. Without it the VMM dies on open and the diff fails; there is no skip path. The documented skip for a machine without KVM (spec §3) is Part V's, where `wick` runs on the real host.

Walkthrough

The `initramfs`'s `init` is the chapter 15 shape: mount `/proc`, `/sys`, `devtmpfs`, quiet the kernel's console with `dmesg -n 1`, and run the lab between two sentinel lines. The test for `/dev/kvm` is one line of output, and `/vmm` is the program below, given the payload.

```
mount -t proc proc /proc
mount -t sysfs sysfs /sys
```

```
mount -t devtmpfs devtmpfs /dev 2>/dev/null
dmesg -n 1
echo '=== lab begin'
[ -c /dev/kvm ] && echo /dev/kvm
/vmm /payload.bin
echo "vmm rc=$?"
echo '=== lab end'
poweroff -f
```

kvm_open opens the device and refuses any API version but 12, as the documentation said to.

```
static int kvm_open(void)
{
    int kvm = open("/dev/kvm", O_RDWR | O_CLOEXEC);
    if (kvm < 0) die("open /dev/kvm");
    int api = ioctl(kvm, KVM_GET_API_VERSION, 0);
    if (api != 12) { fprintf(stderr, "vmm: api %d\n", api); exit(1); }
    printf("vmm: KVM_GET_API_VERSION %d\n", api);
    return kvm;
}
```

vm_memory is gmap's replacement. A 2 MiB anonymous mapping becomes slot 0 at guest-physical 0x80000000, where virt's RAM starts; the payload is position-independent, so any address would do. The guest-physical address has its low 21 bits clear; mmap promises only page alignment for the host address, so the documentation's recommendation that both addresses agree in their low 21 bits (so the host can back the slot with large pages) is not guaranteed, which costs nothing for a 2 MiB test slot. The function returns the host pointer: the VMM writes the guest's code and reads its data through it.

/ Guest-physical memory is a host mapping handed to KVM with a slot:
* KVM fills the G-stage from this slot on guest-page faults. */*

```
static void *vm_memory(int vm)
{
    void *mem = mmap(NULL, GUEST_SIZE, PROT_READ | PROT_WRITE,
                     MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (mem == MAP_FAILED) die("mmap");
    struct kvm_userspace_memory_region r = {
        .slot = 0, .flags = 0,
        .guest_phys_addr = GUEST_BASE,
        .memory_size = GUEST_SIZE,
        .userspace_addr = (uint64_t)(uintptr_t)mem,
    };
    if (ioctl(vm, KVM_SET_USER_MEMORY_REGION, &r) < 0)
        die("KVM_SET_USER_MEMORY_REGION");
}
```

```

    printf("vmm: memory slot 0: 0x%lx bytes at gpa 0x%lx\n",
    ↪ GUEST_SIZE, GUEST_BASE);
    return mem;
}

```

`core_id`, `vcpu_set_reg` and `vcpu_get_reg` are the id encoding of the Mechanism section in code. A register value passes by address, and the register's width is in the id, so one pair of functions covers every core register.

```

/* One RISC-V core register by its index in struct kvm_riscv_core
 * (asm/kvm.h: KVM_REG_RISCV_CORE_REG); api.rst: 0x8030 0000 02
 ↪ <index>. */
static uint64_t core_id(unsigned long index)
{
    return KVM_REG_RISCV | KVM_REG_SIZE_U64 | KVM_REG_RISCV_CORE |
    ↪ index;
}

static void vcpu_set_reg(int vcpu, unsigned long index, unsigned long
    ↪ val)
{
    struct kvm_one_reg r = { .id = core_id(index), .addr =
    ↪ (uint64_t)(uintptr_t)&val };
    if (ioctl(vcpu, KVM_SET_ONE_REG, &r) < 0) die("KVM_SET_ONE_REG");
}

static unsigned long vcpu_get_reg(int vcpu, unsigned long index)
{
    unsigned long val = 0;
    struct kvm_one_reg r = { .id = core_id(index), .addr =
    ↪ (uint64_t)(uintptr_t)&val };
    if (ioctl(vcpu, KVM_GET_ONE_REG, &r) < 0) die("KVM_GET_ONE_REG");
    return val;
}

```

`vcpu_run_until_exit` is `KVM_RUN` with a switch on the exit reason, wrapped in a loop that chapter 21 makes iterate (every case here returns). It accepts one outcome, a shutdown system event, and every other exit is a failure with a message. A lab that returned success after any `KVM_RUN` would pass even if the guest never ran. The `KVM_EXIT_MMIO` case prints what a device model will need, the direction, the address and the width, and refuses; chapter 21 replaces that case.

```

/* Run until the guest stops. Any exit but a shutdown system event is
 ↪ a
 * failure here: chapter 21 is where MMIO exits become a device. */
static int vcpu_run_until_exit(int vcpu, struct kvm_run *run)
{
    for (;;) {

```



```

    if (ioctl(vcpu, KVM_RUN, 0) < 0) die("KVM_RUN");
    switch (run->exit_reason) {
    case KVM_EXIT_SYSTEM_EVENT:
        printf("vmm: KVM_RUN exit %u (KVM_EXIT_SYSTEM_EVENT) type
↳ %u (%s) ndata %u\n",
            run->exit_reason, run->system_event.type,
            run->system_event.type == KVM_SYSTEM_EVENT_SHUTDOWN
↳ ? "SHUTDOWN" : "other",
            run->system_event.ndata);
        return run->system_event.type == KVM_SYSTEM_EVENT_SHUTDOWN
↳ ? 0 : 1;
    case KVM_EXIT_MMIO:
        printf("vmm: KVM_RUN exit %u (KVM_EXIT_MMIO) %s gpa
↳ 0x%llx len %u: no device here\n",
            run->exit_reason, run->mmio.is_write ? "write" :
↳ "read",
            run->mmio.phys_addr, run->mmio.len);
        return 1;
    default:
        printf("vmm: KVM_RUN exit %u: unexpected\n",
↳ run->exit_reason);
        return 1;
    }
}

```

main is the sequence of the figure: open, create the VM, register the memory, copy the payload to the start of the slot, create the vCPU (the helper also maps `kvm_run`; its code is `vmm.c`'s `vcpu_create`, not a listing because it is three checked calls), set `pc` to the slot's base, `a0` to the data page's guest-physical address (the same two things `sepc` and `hyp_enter`'s argument carried in chapter 19) and mode to `S`, run, then read `a7`, `a6` and `t1` back and the data page directly from the slot. The verdict needs both halves: the exit must be a shutdown, and the data page must hold the word the guest stored. The payload size in the capture is the file's: `0x2c`, the same eleven instructions chapter 19 counted.

Exercises

Each was run on `straylight` (QEMU 11.1.1, the lab's kernel and `initramfs`) in a scratch copy of `labs/ch20/`, one edit at a time.

(a) Remove the mode register write and run. Then set mode to `KVM_RISCV_MODE_U` and run. What happens, and why?

Answer, p. 296

(b) Point `a0` at `0x90000000`, outside the slot. What exit arrives, and what does `vcpu_insn.c` put in it?

Answer, p. 297

(c) Read the vCPU's `satp` through `KVM_REG_RISCV_CSR` and print it after the exit. What is it, and what did chapter 19 write to the equivalent register?

Answer, p. 298

(d) Change the payload's SBI extension id from `SRST` to an id KVM does not implement (`0x12345678`). What does the VMM see, and what did chapter 19 do with the same `ecall`?

Answer, p. 298

(e) List what chapter 19 did that this VMM did not have to do, and name the file under `arch/riscv/kvm/` or the layer below it that does each.

Answer, p. 298

(f) Place KVM on chapter 14's three axes for two guests of one host, and compare with chapter 19's answer.

Answer, p. 299

Lab

```
make lab CH=20
```

```
make lab CH=20 IMPL=mime
```

`labs/ch20/build.sh` is chapter 15's recipe with three changes: the cache is `~/.cache/hp/ch20/`, `scripts/config --enable KVM` follows the `defconfig` step, and the `initramfs` is `BusyBox` plus every file in an overlay directory the caller names. `defconfig` at this tag has `CONFIG_KVM=m`, and a module needs a `/lib/modules` tree that the `initramfs` would have to carry; `=y` puts KVM in the `Image`. The kernel is built a second time because chapter 15's has no KVM, and it is built once more for the three KVM chapters together: chapters 21 and 22 call `labs/ch20/build.sh` with their own overlay and output directories and reuse the cache. The script also runs `make headers_install` into `~/.cache/hp/ch20/hdr/`, and the Makefile compiles the VMM with `-I$(CACHE)/hdr/include`: the VMM is built against the kernel it runs on, not against the host's `linux-`

headers package. One header detail is worth knowing: the installed `asm/kvm.h` uses `offsetof` without including `<stddef.h>`, so `vmm.c` includes it first.

`ember/payload.S` is built by `make -C ../../ember HP_CH=19 build/payload.bin` and copied into the overlay: the bytes chapter 19 embeds are the bytes the VMM loads. The VMM is `labs/ch20/vmm.c`, compiled static with the host's cross compiler (`riscv64-linux-gnu-` or `riscv64-unknown-linux-gnu-`, found the way chapter 15 finds it), and the Makefile rebuilds it on every run because `IMPL` changes its source file without changing either file's modification time.

Timings, measured on 2026-09-30: on `straylight` the cold start (clone both trees, build the KVM kernel and `BusyBox`) took 247 s, and a warm `make lab CH=20` takes 11 to 12 s. On `talos`, with 8 build jobs, the kernel build alone took 933 s; the rest of the first run (headers, `BusyBox`, payload, boot) took 86 s, and a warm run takes 16 s. The first run on a new host is the long one; every later chapter run and every run of chapters 21 and 22 starts from the cache.

`IMPL=mime` replaces `vm_memory`'s body with `printf("MINE: memory slot stub\n"); exit(1);`: the chapter's job is the slot, and the stub is the failing version of it. The run diverges there:

```
/dev/kvm
vmm: KVM_GET_API_VERSION 12
-vmm: memory slot 0: 0x200000 bytes at gpa 0x800000000
-vmm: payload 0x2c bytes at gpa 0x800000000, data page at gpa
↳ 0x80001000
-vmm: pc=0x80000000 a0=0x80001000 mode=S
-vmm: KVM_RUN exit 24 (KVM_EXIT_SYSTEM_EVENT) type 1 (SHUTDOWN) ndata
↳ 1
-vmm: guest a7=0x53525354 a6=0x0 t1=0x600d5eed
-vmm: data page holds 0x600d5eed
-vmm: PASS
-vmm rc=0
+MINE: memory slot stub
+vmm rc=1
```

(the diff of `capture.txt` against the run). Writing `mmap` and `KVM_SET_USER_MEMORY_REGION` is the lab; `/dev/kvm` and the API version line, which come before it, still match.

With the reference the capture is:

```
=== lab begin
/dev/kvm
```

```

vmm: KVM_GET_API_VERSION 12
vmm: memory slot 0: 0x200000 bytes at gpa 0x80000000
vmm: payload 0x2c bytes at gpa 0x80000000, data page at gpa 0x80001000
vmm: pc=0x80000000 a0=0x80001000 mode=S
vmm: KVM_RUN exit 24 (KVM_EXIT_SYSTEM_EVENT) type 1 (SHUTDOWN) ndata 1
vmm: guest a7=0x53525354 a6=0x0 t1=0x600d5eed
vmm: data page holds 0x600d5eed
vmm: PASS
vmm rc=0
=== lab end

```

It pins the exit reason (24) and system-event type (1), the registers the guest set (a7, a6, t1) and the data word; it refuses KVM_GET_VCPU_MMAP_SIZE, any address the host or the kernel chose, and the kernel's boot messages (dmesg -n 1 before the begin sentinel). Two runs on straylight are byte-identical to each other and to capture.txt; the local run on talos matches too.

Further reading

- Documentation/virt/kvm/api.rst at the pinned tag: §4.1, §4.2, §4.7, §4.10, §4.35, §4.68, §4.69 and the kvm_run structure at the end.
- arch/riscv/kvm/ at the same tag: the files the lab quotes are vcpu_exit.c, vcpu_sbi.c, vcpu_sbi_replace.c, vcpu_insn.c, mmu.c, gstage.c, vcpu_switch.S and vcpu_onereg.c.
- RISC-V Supervisor Binary Interface Specification, release 3.0: the System Reset Extension, the call the payload makes.
- Firecracker: Lightweight Virtualization for Serverless Applications (Agache et al., NSDI 2020): a minimal VMM over this ioctl set, the shape wick takes.

Answers

(a) Without the write, the run is the capture, byte for byte: the lab's diff passes. (The mode=S line is the VMM's own printf, so the text does not change; the guest's behaviour is what matters, and it still reaches the shutdown with t1=0x600d5eed.) KVM's reset path already put the vCPU in S:

```

// linux v7.2.8, arch/riscv/kvm/vcpu.c:74-74
    cntx->sstatus = SR_SPP | SR_SPIE;

// linux v7.2.8, arch/riscv/kvm/vcpu.c:76-78
    cntx->hstatus |= HSTATUS_VTW;
    cntx->hstatus |= HSTATUS_SPVP;

```

```
cntx->hstatus |= HSTATUS_SPV;
```

SR_SPP is the sstatus.SPP bit that KVM_RISCV_MODE_S would have set, and HSTATUS_SPV is the other half of chapter 19's sret contract, so the explicit write is redundant; the VMM keeps it so that the lab does not depend on a default.

With KVM_RISCV_MODE_U (mode = 0, clearing SPP) the run changes: KVM_RUN does not return. Observed: the output stops after the pc= line and QEMU is killed by timeout (25 s in this run). The source shows where the guest could go: the guest enters VU, its ecall is cause 8, and KVM's default hedeleg delegates that cause to the guest, where chapter 19 set hedeleg = 0:

```
// linux v7.2.8, arch/riscv/kvm/vcpu_config.c:9-15
#define KVM_HEDELEG_DEFAULT      (BIT(EXC_INST_MISALIGNED) | \
                                BIT(EXC_INST_ILLEGAL)      | \
                                BIT(EXC_BREAKPOINT)        | \
                                BIT(EXC_SYSCALL)           | \
                                BIT(EXC_INST_PAGE_FAULT)   | \
                                BIT(EXC_LOAD_PAGE_FAULT)   | \
                                BIT(EXC_STORE_PAGE_FAULT))
```

(EXC_SYSCALL is cause 8, an ecall from U or VU.) So the guest traps into its own kernel at vstvec, which is 0, and nothing there is backed by a slot. Only the hang was observed; the loop through vstvec is what the source implies, not something the run printed. The point of the exercise is the contrast with chapter 19: a VU ecall is a trap the hypervisor can hand to the guest, and a VS ecall, the one the payload makes from S, cannot be (chapter 18: hedeleg bits 9 to 11 are read-only zero), which is why the shutdown call reaches KVM at all.

(b) Observed:

```
vmm: KVM_RUN exit 6 (KVM_EXIT_MMIO) write gpa 0x90000000 len 8: no
↳ device here
vmm: guest a7=0x0 a6=0x0 t1=0x0
vmm: data page holds 0x0
vmm: FAIL
vmm rc=1
```

(The pc=... a0=... line above it still prints 0x80001000: it prints the constant the lab means to set, not the register.) The payload's first memory access is sd t0, 0(a0), a store to a guest-physical address no slot covers. That is a store guest-page fault (code 23) into HS, gstage_page_fault finds no host address for the page and routes a store to kvm_riscv_vcpu_mmio_store (vcpu_exit.c: 30-38), which decodes the 8-byte store, sets run->mmio.is_write, phys_addr and len (vcpu_insn.c:599-601, quoted in the Mechanism section), offers it

to in-kernel devices, and, finding none, sets `KVM_EXIT_MMIO`. `a7` and `t1` read 0 because the guest never got past the store: `t1` is loaded after it. The data page is 0 because the VMM zero-filled it by mmap and nothing wrote to it.

(c) Add after the run:

```
unsigned long satp = 0xdead;
struct kvm_one_reg r = {
    .id = KVM_REG_RISCV | KVM_REG_SIZE_U64 | KVM_REG_RISCV_CSR |
        KVM_REG_RISCV_CSR_GENERAL | KVM_REG_RISCV_CSR_REG(satp),
    .addr = (uint64_t)(uintptr_t)&satp,
};
ioctl(vcpu, KVM_GET_ONE_REG, &r);
printf("vmm: guest satp=0x%lx\n", satp);
```

Observed: `vmm: guest satp=0x0`. `satp` here is the guest's: `struct kvm_riscv_csr` (`asm/kvm.h`) lists `sstatus` to `satp` in the order of KVM's saved VS-level registers (`struct kvm_vcpu_csr`, `kvm_host.h:148-160`, where the ninth field, `satp's slot`, is `vsatp`), and `kvm_arch_vcpu_put` (`vcpu.c:681`) reads the hart's `vsatp` into that field. Zero is Bare: the guest never enabled translation, so the VS-stage did nothing. It is the value chapter 19's `hyp_lab` wrote to `vsatp` by hand; here a fresh vCPU starts with it (the reset `memset` of the CSR block, `vcpu.c:64`).

(d) Observed: the run hangs. The output stops after the `pc=` line, `KVM_RUN` never returns, and QEMU is killed by `timeout`. KVM looks the extension up, finds nothing, and answers the guest itself:

```
// linux v7.2.8, arch/riscv/kvm/vcpu_sbi.c:609-613
    } else {
        /* Return error for unsupported SBI calls */
        cp->a0 = SBI_ERR_NOT_SUPPORTED;
        goto ecall_done;
    }
```

`SBI_ERR_NOT_SUPPORTED` goes into the guest's `a0`, `ecall_done` adds 4 to `sepc` (`vcpu_sbi.c:642-644`), and the guest resumes after its `ecall`, at the payload's `j 1b`, where it spins. No exit happens, because nothing in the path asks for one. The same edit under chapter 19's `ember` (`make lab CH=19` in a scratch copy on `talos`, payload changed to `li a7, 0x12345678`) prints `H: VS ecall ext=0x12345678 fid=0x0 type=0x0` and the rest of the capture, `PASS` included: `hyp_trap` takes any VS `ecall` for the exit and never looks at `a7` to decide. KVM treats the SBI as a contract it implements, one extension id at a time; chapter 19's handler was a shutdown button.

(e) Four things, and one line each for where they went.

M-mode setup. Chapter 19's gated `medeleg` branch and the PMP window from `drop.c` were the machine-mode half of the job. In this lab OpenSBI, QEMU's default firmware for `virt`, does both before Linux starts; its boot banner lists the PMP-protected regions it set up, and KVM's host side runs at HS with that delegation in place and adds only the guest-facing half: `kvm_arch_enable_virtualization_cpu` zeroes `hedeleg` and `hideleg` (`kvm/main.c:44-45`), and each vCPU load writes that vCPU's own values (`vcpu_config.c:76-94`). The M-mode half is not in `arch/riscv/kvm/`.

The G-stage. `gmap`'s table, with U on every leaf, is `mmu.c` and `gstage.c`: `kvm_riscv_mmu_alloc_pgd` for the root, `kvm_riscv_mmu_map` and `kvm_riscv_gstage_map_page` for the leaves, filled on guest-page faults rather than up front.

The hgatp write and hfence.gvma. Chapter 19 required the fence after changing `hgatp`'s mode. KVM writes `hgatp` in `kvm_riscv_mmu_update_hgatp` (`mmu.c:709-722`) and, when the hart has no VMID bits, fences through `kvm_riscv_local_hfence_gvma_all` in `tlb.c`, which is chapter 19's `hfence.gvma zero, zero`:

```
// linux v7.2.8, arch/riscv/kvm/tlb.c:75-78
void kvm_riscv_local_hfence_gvma_all(void)
{
    asm volatile(HFENCE_GVMA(zero, zero) : : : "memory");
}
```

(With VMID bits, which KVM uses to tag translations per VM, the fence is needed less often; `vmid.c` manages the tags.)

The trap frame. `trap_entry` and `struct trapframe` were chapter 5's, reused. KVM's equivalent is `vcpu_switch.S: __kvm_riscv_switch_to` saves the host's registers, restores the guest's, and `srets` (quoted above), and the guest's trap comes back to the label that follows. The difference is the one the quote's comment shows: KVM has two register files to keep apart, a host's and a guest's, and chapter 19 had one.

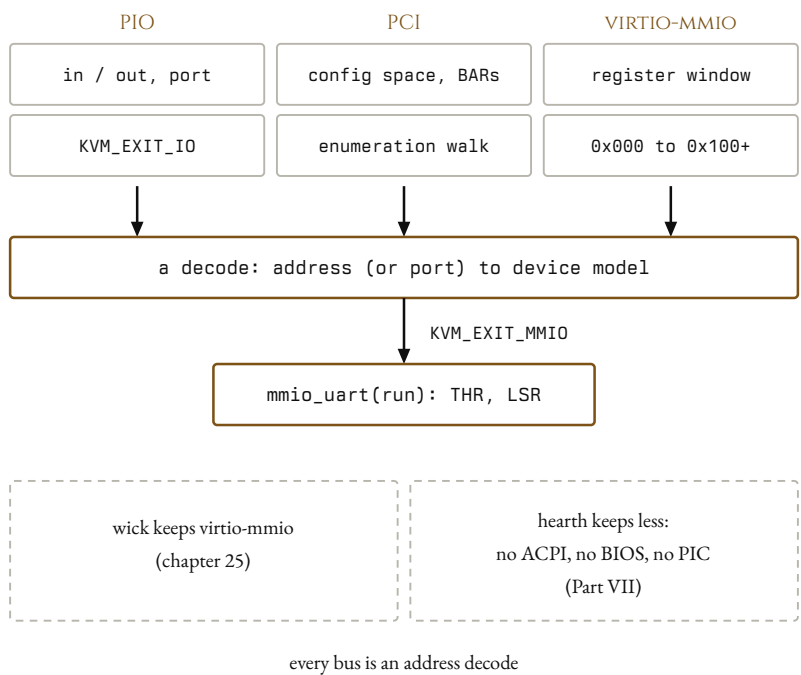
(f) *Shared.* Two guests of one host share no guest-physical memory with each other: `kvm_arch_init_vm` allocates a G-stage root per VM (`vm.c:31-36`), and each guest's slot is a separate mapping in its own VMM process. What each guest does share is the kernel under it and, with its own VMM, the slot's pages: this lab's VMM reads the guest's data page through its own mapping with no copy. Chapter 19's guest shared the page with HS the same way, through the identity

map.

TCB. The code whose bug breaks the isolation is the hardware, the host kernel with KVM in it (chapter 15's whole Linux: a monolith, with `CONFIG_KVM=y`) and the VMM process for what it is allowed to do to its guest. KVM runs at HS in the kernel's address space, so a bug in any driver of that kernel is in the guests' TCB. Chapter 19's was *ember's* image, with nothing under it but the firmware. This is the line chapter 14 drew between *wick*, which has a host kernel, and *hearth*, which does not.

Copy. The guest's memory is not copied on the way to the VMM; the VMM already has it mapped. A device access is the copy: an MMIO exit hands the VMM up to eight bytes through `run->mmio.data` and takes the answer back through the same field (chapter 21), one trap per register access. Chapter 19 had no such boundary because it had no device. What the chapter 14 budget says of the pair: KVM spends more on *trusted* (a kernel) to spend less on writing a hypervisor (an `ioctl` set), and the copy it cannot avoid is per device access, not per page.

21. DEVICE MODELS AND THE BUS YOU WILL KEEP



Why this layer exists

A guest kernel expects a machine, and a machine is more than a hart and RAM. It has a console to print on, a disk to read, a way to find out which of those exist. Chapters 19 and 20 ran a payload that touched memory and made one `ecall`; nothing in them was a device. The hart has no device in it. Every device a guest sees is something the hypervisor pretends is there, and the pretending is a program: a load or a store arrives at an address, the hypervisor works out which pretend device owns that address, and runs the code that does what the hardware would have done.

This chapter builds the smallest such program and then reads the shelf of buses around it. The program is a 16550 UART with two registers, and the guest that uses it is not new: the payload's loop is `ember's chapter 2 uart_putc`: poll the line-status register until bit 5 says the transmitter can take a byte, then store the byte to the transmit register. On QEMU's `virt` board that driver talks to a UART QEMU models. Here the same two accesses go to an address no memory slot backs, and the VMM from chapter 20 answers them. The guest's driver does not change; the machine under it does. That is the whole job of a device model, and every bus in the museum below (port I/O, PCI, `virtio-mmio`) turns out to be a different way of telling the guest which addresses to use.

Mechanism

Three terms, defined where this chapter first needs them. A **device model** is the code that does what a device's hardware would do in response to accesses: it owns the device's state, answers reads from that state and acts on writes. `mmio_uart` below is one; a file under QEMU's `hw/` is one; a Rust struct in Firecracker is one. An **MMIO exit** is `KVM_RUN` returning with `exit_reason KVM_EXIT_MMIO`: the guest executed a load or store to a guest-physical address that no memory slot covers and no device inside the kernel claims, and KVM has stopped the guest with the access described in `kvm_run` for the VMM to complete. A **bus** is whatever lets a guest find a device and name a register in it. Every bus in this chapter reduces to the same act, a decode of an address (or, on x86, a port number) to a device model and an offset into it; they differ in who assigns the addresses and how the guest learns them.

The smallest device: two registers of a 16550

The PC's serial chip is the 16550. Its programming model is a handful of byte-wide registers selected by three address lines; the datasheet's register-address table (the National Semiconductor PC16550D datasheet as republished by Texas Instruments, Table 2) puts them at offsets 0 to 7. Two are enough to print:

- Offset 0, written, is the transmit holding register, `THR`: the byte stored there is sent. The same offset, read, is the receive buffer register, `RBR`; the datasheet's table has one row for both, "Receiver Buffer (read), Transmitter Holding Register (write)", and the direction of the access picks the register. That holds while the divisor-latch access bit (bit 7 of the line control register) is clear, its reset state; with it set, offset 0 is the divisor latch instead. The model never lets the guest set it.

- Offset 5, read, is the line status register, LSR. Two of its bits matter here, in the datasheet's words (§8.6.3). Bit 5: "This bit is the Transmitter Holding Register Empty (THRE) indicator. Bit 5 indicates that the UART is ready to accept a new character for transmission." Bit 6: "This bit is the Transmitter Empty (TEMT) indicator. Bit 6 is set to a logic 1 whenever the Transmitter Holding Register (THR) and the Transmitter Shift Register (TSR) are both empty."

The datasheet's reset table (Table 3) gives the line status register's state after a master reset as 0110 0000, bits 5 and 6 set: 0x60, a transmitter with nothing in it. The model returns exactly that on every read of LSR, and its THR write finishes before the exit returns, so the transmitter really is always empty. The real chip clears THRE while a character is in flight; the model has no flight time. The rest of the chip (the baud-rate divisor latches, the interrupt-enable and FIFO-control registers, the receive path, the modem lines) is not modelled, and the model says so by refusing any other offset.

The exit path, from instruction to run->mmio

The guest's `lbu t2, 5(t0)` is a load from 0x10000005. Nothing maps that address in the G-stage, so the load takes a load guest-page fault into HS, the fault class chapter 19 showed and chapter 20 followed to `gstage_page_fault`: with no memory slot behind the address, a load goes to `kvm_riscv_vcpu_mmio_load` (and a store to its twin, quoted in chapter 20). The function's first job is to learn what the guest tried to do. The hardware helps: on a guest-page fault it records a transformed copy of the faulting instruction in `htinst`, and when the copy is not available KVM reads the instruction out of guest memory at `sepc`.

```
// linux v7.2.8, arch/riscv/kvm/vcpu_insn.c:383-390
/* Determine trapped instruction */
if (htinst & 0x1) {
    /*
     * Bit[0] == 1 implies trapped instruction value is
     * transformed instruction or custom instruction.
     */
    insn = htinst | INSN_16BIT_MASK;
    insn_len = (htinst & BIT(1)) ? INSN_LEN(insn) : 2;
}
```

The second job is the width. Nothing in the fault says how many bytes the guest wanted; the opcode does, and KVM decodes it:

```
// linux v7.2.8, arch/riscv/kvm/vcpu_insn.c:409-417
/* Decode length of MMIO and shift */
```

```

if ((insn & INSN_MASK_LW) == INSN_MATCH_LW) {
    len = 4;
    shift = 8 * (sizeof(ulong) - len);
} else if ((insn & INSN_MASK_LB) == INSN_MATCH_LB) {
    len = 1;
    shift = 8 * (sizeof(ulong) - len);
} else if ((insn & INSN_MASK_LBU) == INSN_MATCH_LBU) {
    len = 1;

```

len is therefore the instruction's width, not a property of the device: a lbu is 1, a lw is 4, an ld is 8. The function then refuses a misaligned access with an error (-EIO, not an exit) and fills in kvm_run:

```

// linux v7.2.8, arch/riscv/kvm/vcpu_insn.c:452-466
/* Fault address should be aligned to length of MMIO */
if (fault_addr & (len - 1))
    return -EIO;

/* Save instruction decode info */
vcpu->arch.mmio_decode.insn = insn;
vcpu->arch.mmio_decode.insn_len = insn_len;
vcpu->arch.mmio_decode.shift = shift;
vcpu->arch.mmio_decode.len = len;
vcpu->arch.mmio_decode.return_handled = 0;

/* Update MMIO details in kvm_run struct */
run->mmio.is_write = false;
run->mmio.phys_addr = fault_addr;
run->mmio.len = len;

```

Before the exit, the access is offered to devices that live inside the kernel; this lab registers none, so every access to the UART exits:

```

// linux v7.2.8, arch/riscv/kvm/vcpu_insn.c:477-481
/* Exit to userspace for MMIO emulation */
vcpu->stat.mmio_exit_user++;
run->exit_reason = KVM_EXIT_MMIO;

return 0;

```

KVM_RUN returns and the VMM reads run->mmio. The documentation says what the VMM owes in return, in a sentence (api.rst:6734-6735): the data member “contains the written data if ‘is_write’ is true, and should be filled by application code otherwise”, and, from the paragraph below it (api.rst:6737), “in its first ‘len’ bytes, the value as it would appear if the VCPU performed a load or store of the appropriate width directly to the byte array.” For the UART read the VMM

stores 0x60 in `data[0]` and calls `KVM_RUN` again. A VMM that forgets to fill `data` gets no error: the next `KVM_RUN` consumes whatever bytes are there.

The guest has not moved on yet. The faulting load has neither completed nor been skipped, and `KVM_RUN`'s first act on the next call is to finish it, before any guest instruction runs:

```
// linux v7.2.8, arch/riscv/kvm/vcpu.c:872-876
switch (run->exit_reason) {
case KVM_EXIT_MMIO:
    /* Process MMIO value returned from user-space */
    ret = kvm_riscv_vcpu_mmio_return(vcpu, vcpu->run);
    break;
```

`kvm_riscv_vcpu_mmio_return` does the two things a load would have done. It puts the byte from `run->mmio.data` into the destination register, and, for stores and loads alike, it steps `sepc` over the instruction:

```
// linux v7.2.8, arch/riscv/kvm/vcpu_insn.c:641-652
if (run->mmio.is_write)
    goto done;

len = vcpu->arch.mmio_decode.len;
shift = vcpu->arch.mmio_decode.shift;

switch (len) {
case 1:
    data8 = *((u8 *)run->mmio.data);
    SET_RD(insn, &vcpu->arch.guest_context,
           (long)((ulong)data8 << shift) >> shift);
    break;
```

The shift was set by the decode: `1bu` leaves it 0, so the byte is zero-extended into the register, while `1b` sets it to 56 and the arithmetic right shift sign-extends. The UART's 0x60 reads the same either way.

```
// linux v7.2.8, arch/riscv/kvm/vcpu_insn.c:672-674
done:
    /* Move to next instruction */
    vcpu->arch.guest_context.sepc +=
    <- vcpu->arch.mmio_decode.insn_len;
```

So an MMIO exit is a synchronous call from the guest into the VMM with a small ABI: one instruction in, at most eight bytes either way. The guest sees an ordinary load that took a long time. The VMM sees a switch case. The model's continue in the exit loop is that second `KVM_RUN`.

The exit count follows from this. The payload’s instruction fetches and the `lbu` that reads the string from `.rodata` touch the memory slot, where a G-stage fault is KVM’s own to resolve and never returns to the VMM (chapter 20); the shutdown `ecall` returns as a system event, not an MMIO exit. `mmio` exits counts the accesses the model answered and nothing else.

Port I/O, named

On x86 the PC architecture has a second address space beside memory: 16-bit port numbers reached by the `in` and `out` instructions. KVM reports such an access as `KVM_EXIT_IO`, with a different record:

```
// linux v7.2.8, Documentation/virt/kvm/api.rst:6695-6704
/* KVM_EXIT_IO */
struct {
#define KVM_EXIT_IO_IN  0
#define KVM_EXIT_IO_OUT 1
    __u8 direction;
    __u8 size; /* bytes */
    __u16 port;
    __u32 count;
    __u64 data_offset; /* relative to kvm_run
    ↪ start */
} io;
```

The documentation recommends both exits, rather than `KVM_EXIT_HYPERCALL` (a guest’s explicit call to the hypervisor, with a number and six arguments), for the same job (`api.rst:6768-6772`; “It is strongly recommended that userspace use `KVM_EXIT_IO` (x86) or `KVM_EXIT_MMIO` (all except `s390`) to implement functionality that requires a guest to interact with host userspace”) and adds a note: “.. note:: `KVM_EXIT_IO` is significantly faster than `KVM_EXIT_MMIO`.” RISC-V has no port space, so this lab’s only device exit is `KVM_EXIT_MMIO`, and `KVM_EXIT_IO` is named here and not used. A model for a port would key on `run->io.port` where this one keys on `run->mmio.phys_addr`, and find its `data_offset` bytes into the `kvm_run` page.

PCI: a bus that tells the guest where the devices are

PCI solves discovery. A guest does not need to be told which devices exist or where: each device exposes a small block of configuration registers, addressed by (bus, device, function), whose first fields are a vendor ID and a device ID, and an OS enumerates by walking every address and noting which ones answer. The same block holds the **BARs** (base address registers), where the OS writes the

address at which the device’s own registers should appear. The guest, not the hypervisor, assigns the decode addresses. The cost to the hypervisor is a model of the host bridge, of configuration space for every device, of BAR sizing, and of the interrupt plumbing PCI brings (MSI and MSI-X). Virtio over PCI is §4.1 of the virtio 1.2 specification: “Any PCI device with PCI Vendor ID 0x1AF4, and PCI Device ID 0x1000 through 0x107F inclusive is a virtio device” (§4.1.2). wick would need that transport to boot a guest that expects PCI, and does not build it: it hands its guests virtio-mmio instead.

virtio-mmio: no discovery, one window

The virtio specification introduces the memory-mapped transport as the one for the machines without a PCI bus: “Virtual environments without PCI support (a common situation in embedded devices models) might use simple memory mapped device (“virtio-mmio”) instead of the PCI device.” (§4.2). What it gives up is discovery, and §4.2.1 says so: “Unlike PCI, MMIO provides no generic device discovery mechanism. For each device, the guest OS will need to know the location of the registers and interrupt(s) used.” That is chapter 11’s situation from the guest side: ember probes eight windows at $0x10001000 + n * 0x1000$ because QEMU’s virt board put devices there. A VMM that builds the machine chooses the addresses and tells the guest (the specification’s suggested binding is a device-tree node with the window’s base, its length and an interrupt number).

What sits in a window is the register table of §4.2.2 (Table 4.1, fetched 2026-10-01). The registers ember’s chapter 11 reads and writes, with the specification’s direction column:

Offset	Name	Direction
0x000	MagicValue	R
0x004	Version	R
0x008	DeviceID	R
0x010	DeviceFeatures	R
0x014	DeviceFeaturesSel	W
0x020	DriverFeatures	W
0x024	DriverFeaturesSel	W
0x030	QueueSel	W
0x034	QueueNumMax	R
0x038	QueueNum	W
0x044	QueueReady	RW
0x050	QueueNotify	W
0x060	InterruptStatus	R
0x064	InterruptACK	W

Offset	Name	Direction
0x070	Status	RW
0x080, 0x084	QueueDescLow, QueueDescHigh	W
0x090, 0x094	QueueDriverLow, QueueDriverHigh	W
0x0a0, 0x0a4	QueueDeviceLow, QueueDeviceHigh	W
0x100+	Config (device-specific)	RW

Two lines of §4.2.2.2 change what a model must check. First: “The driver **MUST** only use 32 bit wide and aligned reads and writes to access the control registers described in table 4.1.” So for the control registers `mmio_virtio`’s width test is `len != 4` (the device-specific config space at `0x100` is accessed at the width of each field), where `mmio_uart`’s is `len != 1`; the offset is still `phys_addr - base`. Second: the driver “**MUST NOT** write to the read-only registers (direction R) and **MUST NOT** read from the write-only registers (direction W)”, so the direction column is the list of accesses the model may refuse, the same refusal exercise (c) asks `mmio_uart` to make for a read of `THR`. The register that matters for what comes next is `QueueNotify`: “Writing a value to this register notifies the device that there are new buffers to process in a queue.” One 32-bit store is the doorbell for an arbitrarily large transfer, which the 16550’s one-store-per-byte interface cannot be. Chapter 22 is that device.

What Firecracker deleted, and what a unikernel hypervisor deletes after it

Firecracker’s paper (Agache et al., NSDI 2020) is a list of buses not built. It “does not offer a BIOS, cannot boot arbitrary kernels, does not emulate legacy devices nor PCI, and does not support VM migration” (§1), and the reason is stated: “Lower-level features, such as additional devices (USB, PCI, sound, video, etc), BIOS, and CPU instruction emulation are simply not implemented because they are not needed by typical serverless container and function workloads.” What it keeps is “device emulation for disk, networking and serial console” (§3), and the cost of the part that matters: “The entire `virtio` block implementation in Firecracker (including MMIO and data structures) is around 1400 lines of Rust” (§3). A device model is cheap and a bus is expensive, and a guest that has no PCI needs no PCI model. That is chapter 23’s subject for `wick`; this chapter’s point is that each deleted bus is a decode the VMM no longer has to implement correctly.

`hearth` deletes further, because its guests are not Linux. A guest written against chapter 32’s contract is handed a boot-info block, not a platform description to discover: it does not need a table language (ACPI), a boot firmware to call (BIOS), or the PC’s legacy interrupt controller (the 8259 PIC) to find out what

machine it is on. Each is a model the hypervisor does not have to write and a surface a guest cannot attack. That is the shorter shelf in the figure, and Part VII's argument.

The three budget axes of chapter 14 apply to a device model without change, and exercise (f) places `mmio_uart` on them.

What this lab does not do

No interrupt reaches the guest: the payload polls LSR and nothing signals it. Interrupt injection is a separate mechanism with its own chapter in hearth (chapter 38) and, on a KVM host, its own ioctl: `KVM_INTERRUPT` (`api.rst` §4.16, which lists RISC-V) when KVM has no in-kernel model of the interrupt controller, an *irqchip*, and `KVM_IRQ_LINE` when it has one, which `arch/riscv/kvm/vm.c` implements and refuses without one (line 62) although §4.25 does not list RISC-V; chapter 22's driver polls too. The model also has no receive path, no FIFO, and no state beyond the counter in the VMM.

Walkthrough

The payload is the driver. `t0` holds the UART's base; `a2` walks the string in `.rodata`; the inner loop is `uart_putc`'s two steps, LSR read and mask, then the THR store. The end is chapter 19's shutdown call, the same `ecall` with `a7 = SRST`.

```
# The guest: print "hi from VS\n" through a 16550's two registers,
↳ then
# ask for shutdown as chapter 19's payload did. Position independent,
↳ no
# stack. Every load and store through t0 is to an address no memory
↳ slot
# backs: each one is a VM exit that the VMM turns into a device
↳ access.
    .option norelax          # keep auipc+addi: the linker would fold
    ↳ them to an absolute li at address 0
    .section .text
    .globl _start
_start:
    li    t0, 0x10000000      # UART base (guest-physical)
    lla   a2, msg             # PC-relative, no GOT: the payload is
    ↳ loaded at 0x80000000
1: lbu   a3, 0(a2)           # next byte of the string
    beqz a3, 3f              # NUL: done
```

```

2: lbu  t2, 5(t0)          # LSR
   andi t2, t2, 0x20       # THRE: transmitter holding register
   ↵ empty
   beqz t2, 2b
   sb   a3, 0(t0)          # THR
   addi a2, a2, 1
   j    1b
3: li   a7, 0x53525354     # SBI SRST, as chapter 19
   li   a6, 0
   li   a0, 0
   li   a1, 0
   ecall
4: j    4b

.section .rodata
msg:
.asciz "hi from VS\n"

```

The Makefile assembles it with chapter 19's flags (`-march=rv64ima_zicsr -mabi=lp64 -nostdlib -static -Wl,-Ttext=0 -Wl,--build-id=none`) and extracts `.text` and `.rodata` with `objcopy -O binary`, so the string follows the code in the file and in the guest's memory. Two details are the toolchain's. `1a a2, msg` on this cross toolchain expands to a load from the global offset table, which `objcopy -j .text -j .rodata` leaves out of the binary the VMM loads. `11a` is the PC-relative form (`auipc` and `addi`), and with `-Ttext=0` the linker relaxes that pair into an absolute `li a2, 68`; `.option norelax` keeps the pair, and the payload position-independent, which it has to be: it is linked at 0 and loaded at `0x80000000`.

The device model is the `ch21-uart` listing. `UART_BASE` is the address the payload uses, and the offsets and the two LSR bits are the datasheet's. `mmio_uart` returns 0 for the two accesses it understands, -1 for anything else, including a width other than 1.

```

#define UART_BASE 0x10000000UL
#define UART_THR  0
#define UART_LSR  5
#define LSR_THRE  0x20          /* transmitter holding register
↵ empty */
#define LSR_TENT  0x40          /* transmitter empty */

/* The device model: what the hardware would have done, done by us,
 * after the exit. Reads fill run->mmio.data for the next KVM_RUN. */
static int mmio_uart(struct kvm_run *run)

```

```

{
    unsigned long off = run->mmio.phys_addr - UART_BASE;
    if (run->mmio.len != 1) return -1;
    if (run->mmio.is_write && off == UART_THR) {
        putchar(run->mmio.data[0]);
        fflush(stdout);
        return 0;
    }
    if (!run->mmio.is_write && off == UART_LSR) {
        run->mmio.data[0] = LSR_THRE | LSR_TEMT;
        return 0;
    }
    return -1;
}

```

The exit loop is chapter 20's with one case changed. An exit inside the window `UART_BASE` to `UART_BASE + 8` goes to the model, and on success the loop counts it and re-enters (continue is the second `KVM_RUN` of the Mechanism section). An access the model refuses, or any access outside the window, prints the line chapter 20 printed and fails the run.

```

/* Run until the guest stops. An MMIO exit inside the UART's window
↳ goes
* to the device model and the loop re-enters; any other exit but a
* shutdown system event is a failure. */
static int vcpu_run_until_exit(int vcpu, struct kvm_run *run)
{
    for (;;) {
        if (ioctl(vcpu, KVM_RUN, 0) < 0) die("KVM_RUN");
        switch (run->exit_reason) {
            case KVM_EXIT_SYSTEM_EVENT:
                printf("vmm: KVM_RUN exit %u (KVM_EXIT_SYSTEM_EVENT) type
↳ %u (%s) ndata %u\n",
                    run->exit_reason, run->system_event.type,
                    run->system_event.type == KVM_SYSTEM_EVENT_SHUTDOWN
↳ ? "SHUTDOWN" : "other",
                    run->system_event.ndata);
                return run->system_event.type == KVM_SYSTEM_EVENT_SHUTDOWN
↳ ? 0 : 1;
            case KVM_EXIT_MMIO:
                if (run->mmio.phys_addr >= UART_BASE &&
↳ run->mmio.phys_addr < UART_BASE + 8) {
                    if (mmio_uart(run) == 0) { mmio_exits++; continue; }
                }

```

```

        printf("vmm: KVM_RUN exit %u (KVM_EXIT_MMIO) %s gpa
↳ 0x%llx len %u: no device here\n",
               run->exit_reason, run->mmio.is_write ? "write" :
↳ "read",
               run->mmio.phys_addr, run->mmio.len);
        return 1;
    default:
        printf("vmm: KVM_RUN exit %u: unexpected\n",
↳ run->exit_reason);
        return 1;
    }
}
}

```

main is chapter 20's with the data page gone: no `a0`, no word to read back from the slot. It prints the exit count after the run and the two registers `a7` and `a6`, and its verdict is a shutdown exit with `a7` equal to the SRST extension id. The string itself is checked by the capture diff, not by the VMM: a model that dropped the THR writes would pass the verdict and fail the diff.

The lab starts red. With chapter 20's VMM (which refuses every MMIO exit) and this payload, the guest's first access is the LSR read:

```

vmm: KVM_RUN exit 6 (KVM_EXIT_MMIO) read gpa 0x100000005 len 1: no
↳ device here

```

The address is `UART_BASE + 5`, the length is the 1bu's, and the run fails there: the payload cannot print until the VMM is a bus.

Exercises

Each was run on `talos` (QEMU 11.1.1, the lab's kernel and `initramfs`) in a scratch copy of `labs/ch21/`, one edit at a time.

(a) Change the model so that the first LSR read after each THR write (and the first read of the run) reports not-empty (0), and later reads report 0x60. What does `mmio` exits become, and why?

Answer, p. 315

(b) Change the payload's sb `a3, 0(t0)` to sw `a3, 0(t0)`, a four-byte store to the same address. What fails, and what does `len` mean?

Answer, p. 315

(c) Add a `lbu t3, 0(t0)` before the poll loop: a read of `THR`'s offset. What does the model do, and what would a real 16550 return?

Answer, p. 315

(d) Move the UART to `0x10001000` in both the payload and `UART_BASE`. What changes in the capture?

Answer, p. 316

(e) Make the payload's string 1024 bytes of `x` and run. How many exits does it cost, and what does the virtio ring (chapter 22) exist to replace?

Answer, p. 316

(f) Place `mmio_uart` on chapter 14's three axes, as it runs in the VMM process, and say what changes in hearth.

Answer, p. 316

Lab

```
make lab CH=21
make lab CH=21 IMPL=mime
```

`labs/ch21/Makefile` is chapter 20's with two changes. It calls `../ch20/build.sh overlay out`, so the kernel, BusyBox and kernel headers come from the same `hp/ch20` cache, and a host that has run chapter 20 builds nothing new here. And it assembles the payload itself from `payload.S` instead of copying one from `ember/`. The VMM is `labs/ch21/vmm.c`, chapter 20's file with the device model, the exit-loop case and the counter added and the data page removed; the `ch21-init` script in `overlay/init` is chapter 20's.

`IMPL=mime` replaces the body of `mmio_uart` with `printf("MIME: mmio stub\n"); return -1;`. The exit loop then falls through to "no device here" on the first LSR read, and the run diverges there:

```
vmm: pc=0x80000000 mode=S
-hi from VS
-vmm: KVM_RUN exit 24 (KVM_EXIT_SYSTEM_EVENT) type 1 (SHUTDOWN) ndata
↳ 1
-vmm: mmio exits 22
-vmm: guest a7=0x53525354 a6=0x0
-vmm: PASS
-vmm rc=0
+MIME: mmio stub
```

```
+vmm: KVM_RUN exit 6 (KVM_EXIT_MMIO) read gpa 0x10000005 len 1: no
↳ device here
+vmm: mmio exits 0
+vmm: guest a7=0x0 a6=0x0
+vmm: FAIL
+vmm rc=1
```

(the tail of the diff of capture.txt against the run). Writing the two registers is the lab.

With the reference the capture is:

```
=== lab begin
/dev/kvm
vmm: KVM_GET_API_VERSION 12
vmm: memory slot 0: 0x200000 bytes at gpa 0x80000000
vmm: payload 0x54 bytes at gpa 0x80000000
vmm: pc=0x80000000 mode=S
hi from VS
vmm: KVM_RUN exit 24 (KVM_EXIT_SYSTEM_EVENT) type 1 (SHUTDOWN) ndata 1
vmm: mmio exits 22
vmm: guest a7=0x53525354 a6=0x0
vmm: PASS
vmm rc=0
=== lab end
```

The 22 was observed before it was explained: the payload prints 11 bytes (hi from VS is ten characters and the newline), and with a model that always reports empty each byte is one LSR read and one THR write, two exits. The `ecall` is not an MMIO exit: it ends the run as the system event on the line above. The capture pins the exit reasons, the printed string, the exit count and `a7` and `a6`; it does not pin the guest-physical addresses of the exits or their widths (the exercises read those) beyond the payload's byte count, 0x54, the size of the assembled file. The run on `talos` and two runs on `straylight` are byte-identical to `capture.txt`.

Further reading

- Virtual I/O Device (VIRTIO) Version 1.2 (OASIS Committee Specification 01, July 2022): §4.1 (the PCI transport, §4.1.2 discovery), §4.2 (the MMIO transport), §4.2.1 discovery, §4.2.2 the register layout and its driver requirements.
- PC16550D Universal Asynchronous Receiver/Transmitter with FIFOs (National Semiconductor, now Texas Instruments, datasheet SNLS378C, revised May 2015, as hosted by a distributor; the publisher's own URL

did not resolve when checked): Table 2 (register addresses), Table 3 (reset state), §8.6.3 (the line status register).

- Firecracker: Lightweight Virtualization for Serverless Applications (Agache et al., NSDI 2020): §1 for what it does not implement, §3 for the devices it does.
- Documentation/virt/kvm/api.rst at the pinned tag: the `kvm_run` structure’s `KVM_EXIT_IO` and `KVM_EXIT_MMIO` records and the note after them; §4.16 `KVM_INTERRUPT`.
- `arch/riscv/kvm/vcpu_insn.c` at the same tag: `kvm_riscv_vcpu_mmio_load`, `kvm_riscv_vcpu_mmio_store`, `kvm_riscv_vcpu_mmio_return`.

Answers

(a) Observed: `vmm: mmio exits 33`, with the string and every other line of the capture unchanged. Each byte now costs three exits instead of two: an LSR read that says “busy”, the payload’s `beqz` back to the poll, a second LSR read that says “empty”, then the `THR` store. Eleven bytes at three is 33. Polling a register through a device model is paid in VM exits, one per poll; real hardware’s `THRE` would stay clear for a character time and the guest would spin at memory speed, here each spin is a round trip to the VMM. The model’s always-empty LSR is the cheapest honest answer, and it is also the reason the capture’s 22 is a lower bound for this payload.

(b) Observed:

```
vmm: KVM_RUN exit 6 (KVM_EXIT_MMIO) write gpa 0x10000000 len 4: no
↳ device here
vmm: mmio exits 1
vmm: guest a7=0x0 a6=0x0
vmm: FAIL
vmm rc=1
```

The count is 1: the first LSR read was served, and the run died at the first store. The address is right, the direction is right and `len` is 4. `len` is the width of the instruction, decoded from its opcode in `kvm_riscv_vcpu_mmio_store` (`vcpu_insn.c:539-542`: `INSN_MATCH_SW` sets `len = 4`, `INSN_MATCH_SB` sets 1), and it says nothing about what the device accepts; `data[0..3]` hold the four bytes of the register’s value, as the documentation’s “first ‘len’ bytes” says. `mmio_uart’s len != 1` test is a decision that this device is byte-wide. A model that accepted the store would have to choose what a word to a byte register means (take the low byte, or the whole word as four characters); refusing is the choice that cannot print the wrong thing.

(c) Observed: `vmm: KVM_RUN exit 6 (KVM_EXIT_MMIO) read gpa 0x10000000 len 1: no device here and vmm: mmio exits 0, then FAIL`. The model refuses the read because it implements THR, the write side of offset 0, and not RBR, the read side (offset 0 is those two only while the divisor-latch access bit is clear, which it always is here). The datasheet's register table has one row for both at offset 0 (Table 2: "Receiver Buffer (read), Transmitter Holding Register (write)"); the chip returns whatever its receive buffer holds, and LSR bit 0 (data ready) says whether that byte is fresh. A model with no receive path could return 0, which would tell the guest a NUL byte arrived. Refusing is safer, and costs nothing for a driver that checks data-ready first: the model's LSR is 0x60, bit 0 clear, and ember's `uart_getc (uart[LSR] & LSR_DR, chapter 2)` never reads RBR in that case.

(d) Nothing: `diff` against `capture.txt` reports no difference: string, exit count and all. (The edit is two constants, `li t0, 0x10001000` and `#define UART_BASE 0x10001000UL`.) `0x10001000` is where QEMU's virt board puts its first virtio-mmio window, the address ember's chapter 11 probes; in this VM it is an address like any other, with a model where the VMM put one. The guest cannot tell, and neither can the exit: `phys_addr` is only a key into the VMM's decode. A bus is that mapping, and who writes it down (a datasheet, a BAR, a device tree node) is the only difference among the shelves of the figure.

(e) Observed: `vmm: mmio exits 2048`, with PASS (the payload is 0x449 bytes: the code, 1024 characters and a NUL). Two exits per byte, 2048 in all, each a trip from the guest through HS and the kernel's instruction decoder to userspace and back through KVM_RUN. Nothing about the exit count depends on the device being fast; it is set by the interface, a register per byte. A virtqueue replaces the byte-per-store interface with a descriptor in guest memory that names the whole buffer: the memory is a slot the VMM already has mapped (chapter 20), so reading the buffer costs no exit, and the guest signals the device with a single write to `QueueNotify` (offset 0x50, Table 4.1) for the batch. One doorbell exit, then, where this device pays 2048; chapter 22 builds the backend and counts.

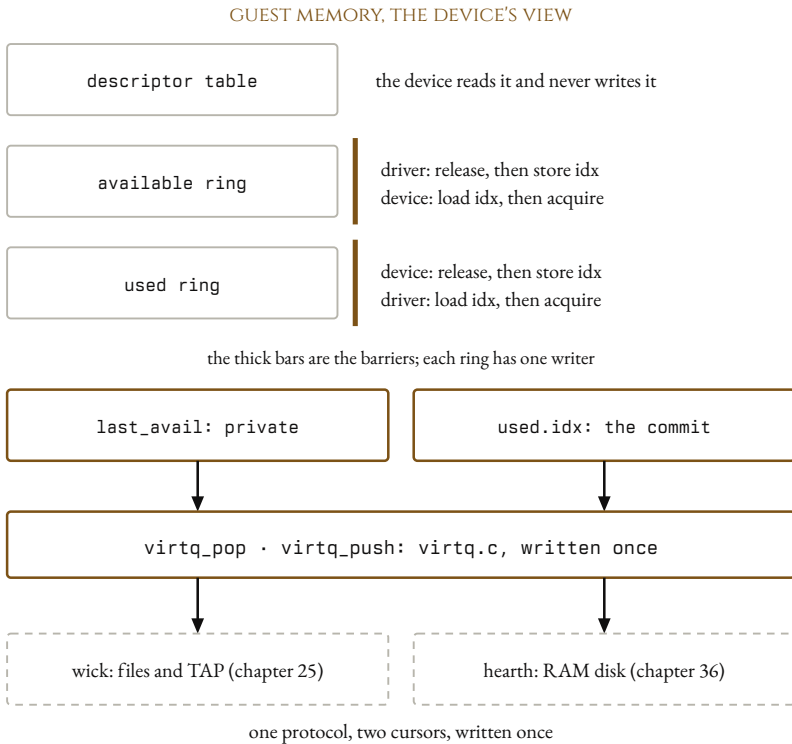
(f) *Shared*. The guest and the model share one thing, the register window: the guest-physical addresses `UART_BASE` to `UART_BASE + 7`, with nothing behind them except what the VMM chooses to answer. They share no memory beyond chapter 20's slot, and the model's state (the `mmio_exits` counter, `stdout`) is the VMM process's own.

TCB. `mmio_uart` runs in the VMM process, at the privilege of a userspace program under the host kernel, so a bug in it is a bug in a process that the host

can confine (chapter 26's jailer for wick); what the guest reaches in the kernel is the exit path quoted above, and the instruction decoder in it is KVM's, not the VMM's. In hearth there is no process to confine: the device model is part of the hypervisor image, so every line of it is in the TCB, which is the argument for the short shelf in the figure.

Copy. At most eight bytes cross per exit, through `run->mmio.data`; the capture's 22 exits (`talos` and `straylight`) move 11 bytes out and 11 status bytes in. The copy is small and the crossing is the cost, which is the number exercise (e) turns into 2048.

22. VIRTIO AS A PROTOCOL



Why this layer exists

Chapter 21 ended on a number: a 1024-character string through the 16550 cost 2048 exits, two for every byte, because the device's interface is one register access per byte and every access is a trip from the guest through the kernel to the VMM and back. A device that moves a disk sector cannot be built that way. It needs an interface in which the bytes cost no exits at all: the guest leaves the data in its own memory, which the VMM already has mapped (chapter 20), and tells the

device where, and the device reads or writes it in place. What is left to agree on is the shape of that memory, who may write which part of it and in what order, and how the device knows there is something to look at. That agreement is the virtqueue, and virtio is the standard that fixes it.

Chapter 11 wrote the driver’s half, with `ember` as the guest and QEMU as the device. This chapter writes the other half, once. `virtq.c` is the device side of a split virtqueue in seventy lines of freestanding C: it takes a ring that a driver filled and returns chains of buffers, and takes finished chains and hands them back. A host-side harness drives it through six checks, several with more than one case, and the harness is the contract. Its six output lines open the capture, and much of what it checks prints nothing: the counters’ wrap at 65536, every malformed ring but one, a ring offered exactly full, a push that ignores a scribbled `used_idx`. `wick`’s Rust ring meets the same checks in chapter 25 and `hearth`’s serves the same protocol to its guests in chapter 36. Then the same file goes into chapter 21’s VMM behind a virtio-mmio block device over a RAM disk, and a bare guest driver in C, running as the guest under KVM, reads one sector through it. The lab stops there on purpose: one queue, one request type, one thread, and no interrupts. The Mechanism section says where each of those limits is set and what lifting it costs.

Mechanism

The ring, from the side that does not write the table

A virtqueue is the ring chapter 11 defined: a table of buffer descriptors in guest memory that a driver fills and a device drains, with an index on each side saying how far each has got. That chapter built it from the driver’s chair. The specification’s second sentence about the split layout is the rule the whole design turns on (virtio 1.2 §2.7, fetched 2026-10-01):

The split virtqueue format separates the virtqueue into several parts, where each part is write-able by either the driver or the device, but not both.

There are three parts, and the specification’s table gives each its alignment and size for a queue of *N* entries:

Part	Alignment	Size	Written by
Descriptor table	16	16 * N	the driver
Available ring	2	6 + 2 * N	the driver

Part	Alignment	Size	Written by
Used ring	4	$6 + 8 * N$	the device

The device’s whole job follows from the third column. It reads the descriptor table and the available ring and writes the used ring and the buffers the descriptors name, and it never writes a descriptor. The specification says the last part twice over (§2.7.5.1): “A device **MUST NOT** write to a device-readable buffer, and a device **SHOULD NOT** read a device-writable buffer (it **MAY** do so for debugging or diagnostic purposes). A device **MUST NOT** write to any descriptor table entry.” This is why `virtq_pop` takes the table through a pointer it only loads from, and why `struct virtq` has no function that stores into `desc`.

A descriptor is 16 bytes (§2.7.5): a guest-physical address, a length, flags and a next index. `VIRTQ_DESC_F_NEXT` (1) says the buffer continues in the descriptor next names; `VIRTQ_DESC_F_WRITE` (2) says the device may write it, and without that flag the device may only read. A chain of descriptors is one request. The available ring (§2.7.6) is a 16-bit `flags`, a 16-bit `idx` and a ring of 16-bit descriptor indices, each the head of a chain: “`idx` field indicates where the driver would put the next descriptor entry in the ring (modulo the queue size). This starts at 0, and increases.” The used ring (§2.7.8) has the same two header fields and a ring of `{id, len}` pairs, `id` the head of a finished chain and `len` the number of bytes the device wrote. Version 2 of the transport takes three separate ring addresses where the legacy interface had one contiguous, page-aligned region (§2.7.2). `virtq.h` declares the three structures with the natural C layout, which is the specification’s layout (descriptor 16 bytes; `flags` and `idx` at offsets 0 and 2, the ring at offset 4 in both rings, 8-byte used elements), and does not pack them: the atomics below need an addressable, naturally aligned `idx`, and packing would take that away for nothing.

Two cursors

The driver owns `avail.idx` and the device owns `used.idx`; both only ever increase, wrap naturally at 65536 (§2.7.13.3: “`idx` always increments, and wraps naturally at 65536”), and name a slot by the index modulo the queue size. Because the queue size is a power of two, 65536 is a multiple of it and the slot arithmetic stays consistent across the wrap: slot 7 is followed by slot 0 whether the counter is at 7 or at 65535. The device keeps two private numbers: `last_avail`, the count of chains it has taken, and `next_used`, its own copy of `used.idx`. The copy exists because the used ring is guest memory too and a driver can

write `used.idx`; reading it back to decide where the next element goes would let the driver choose the slot. Nothing in guest memory says which chains the device has seen; `avail.idx - last_avail` does, and it cannot exceed the queue size in a driver that follows the rules, because it may not offer more than the descriptor table holds (§2.7.13: “the 16-bit `idx` value can always distinguish between a full and empty buffer”). A difference greater than the queue size is a driver that broke the rule, and `virtq_pop` refuses it. So `virtq_pop` has one empty test, `avail.idx == last_avail`, and one commit point, `last_avail++`; and `virtq_push` has one commit point, the store to `used.idx`. Two cursors, one private (`last_avail`) and one published (`used.idx`, shadowed by the private `next_used`), are the entire state the backend keeps.

Barriers: who writes what, and when it is visible

The driver offers a buffer in this order (§2.7.13): it fills the descriptors, writes the chain’s head into the next ring slot, performs “a suitable memory barrier to ensure the device sees the updated descriptor table and available ring before the next step”, and only then increases `avail.idx`; requirement §2.7.13.3.1 states the same ordering as an obligation: “The driver MUST perform a suitable memory barrier before the `idx` update, to ensure the device sees the most up-to-date copy.” That is a release: everything written before `idx` is visible to anyone who sees the new `idx`. The device’s half is the matching acquire, a load of `avail.idx` after which the ring slot and the descriptors may be read. The specification names the driver’s barrier and not this one; it is the reader’s half of the pair, and `virtq_pop` makes it with `__ATOMIC_ACQUIRE` on the load of `avail.idx`.

The device’s own obligation is the mirror image, and the specification does state it (§2.7.8.2): “The device MUST set `len` prior to updating the `used.idx`. The device MUST write at least `len` bytes to descriptor, beginning at the first device-writable buffer, prior to updating the `used.idx`.” So `virtq_push` writes the element, and `__ATOMIC_RELEASE` on the store to `used.idx` orders the element and every byte the device wrote into the buffers before the index that publishes them. The driver’s acquire on `used.idx` completes that pair. The figure marks those four: a release by each writer before its index, an acquire by each reader after it. They are not the complete set. After the device stores `used.idx` it decides whether to notify the driver by loading `avail.flags` (§2.7.7.2), a store followed by a load, which release and acquire do not order; on a second thread the load can be satisfied before the store is visible and a notification can be lost. The specification puts the matching MUST on the driver side (§2.7.13.4.1: “The driver MUST perform a suitable memory barrier before reading `flags` or `avail_event`, to avoid

missing a notification”), and the device needs the same full fence for the same reason, so `blk_service` issues `__ATOMIC_SEQ_CST` between the push and the flags read.

The strength of those barriers is the specification’s to set, and it sets it in the feature bits (§6, `VIRTIO_F_ORDER_PLATFORM`, bit 36): “If this feature bit is not negotiated, then the device and driver are assumed to be implemented in software, that is they can be assumed to run on identical CPUs in an SMP configuration. Thus a weaker form of memory barriers is sufficient to yield better performance.” Here both sides are software, the bit is not offered, and the barriers are the ones threads use on one machine, which is what the `__atomic` acquire and release orderings give; a device behind real hardware (a PCI card) would need the stronger kind and would negotiate the bit.

Nothing in this lab exercises them, and the chapter says so rather than pretending. The harness is one thread. The VMM is one thread too: `KVM_RUN` returns to the VMM with the vCPU stopped, the doorbell handler runs the ring before the next `KVM_RUN`, and the guest cannot store anything while it does; nothing the guest does can race with the drain. The barriers are in the code because the code is the contract: wick’s device will run on its own thread beside the vCPU thread (chapter 25), and the ring it serves has to be correct there. Exercise (b) is a two-thread test that shows what the order of the stores does once there is a second reader.

A driver that is not trusted

The ring is the driver’s memory, and the driver is the guest; a guest that is hostile, or merely buggy, can put anything in it. `virtq_pop` therefore treats every field as an input and makes four decisions about each chain before returning it.

Every field is read once. The descriptor table is guest memory that the guest may change while the device runs, so a copy is only a copy if the compiler really makes it; reading `flags` from the table to test it and again to follow it would check one value and use another. `virtq_pop` loads each field of each descriptor, and the ring entry, with a relaxed atomic load and works from those values. Copying the descriptor into a local struct looks like one read and is not guaranteed to be one, because a compiler may load the fields lazily at their uses; the explicit loads cannot be turned back into two reads.

The indices are bounds-checked. An `avail.idx` more than `num` ahead of `last_avail` is not a ring the protocol allows (the device could only serve stale slots), a head or a next of `num` or more is not a descriptor, and each is refused. The

chain length is bounded by `VIRTQ_MAX_SEGS` (8): the descriptor table's size is "the maximum possible descriptor chain length" (§2.7.5) and §2.7.5.2 forbids loops ("this implies that loops in the descriptor chain are forbidden!"), so a chain longer than the table is a loop, and the bound is what stops a loop that a hostile driver built anyway. The specification allows a device to set its own limit (§2.7.4.1: "The device MAY have a reasonable limit of descriptors it will allow in a chain"); here the limit is the queue size this device offers, and a device with a larger queue would raise it to the queue size.

The order is enforced. §2.7.4.2: "The driver MUST place any device-writable descriptor elements after any device-readable descriptor elements." A chain with a readable buffer after a writable one is malformed, and `virtq_pop` returns -1 on it. The rule is what lets a device treat the chain as "the bytes I read, then the places I write" with no bookkeeping, and it is the one a block device leans on: header first, status last.

The device's own writes cannot feed the loop. The three areas are disjoint, and `QueueReady = 1` refuses a ring in which they overlap. That matters for two reasons: a used ring over the descriptor table would make the device write the table, which §2.7.5.1 forbids, and a used ring over the available ring would have each push store the device's index into `avail_idx`, erasing the chains the driver offered (an exact alias with three chains offered served one and dropped the other two). As a second defence the drain serves at most `num` chains per doorbell: a driver more than `num` ahead is refused, so `num` chains are everything that can be offered at that moment. The bound earns its place against history: an earlier `virtq_push` read `used_idx` back from guest memory, and with the used ring aliased onto the available ring every push then bumped `avail_idx` and the drain ran a million chains on one doorbell. That is exactly why the device keeps its own `next_used`.

Every address is translated. A descriptor's `addr` is a guest-physical address, which means nothing to the VMM until `gpa2hva` turns it into a host pointer; `virtq_pop` passes the length too, and a NULL back means some byte of the buffer is outside what the guest owns, which is also malformed. The translation is the caller's because it is the one thing the backend cannot know: in this lab it is an offset into the one memory slot, and in `hearth` it will be the guest's G-stage (chapter 19) rather than anything a host process maps.

On any of these `virtq_pop` returns -1 and leaves `last_avail` where it was: the request stays at the head of the ring, un-consumed, for the caller to deal with. What the caller does is the device's policy, not the protocol's. The VMM sets the

device status bit `DEVICE_NEEDS_RESET` (64) and the configuration-change bit of `InterruptStatus`, as §2.1.2 asks of a device in an error state it cannot recover from, and stops serving the queue: `QueueNotify` is refused while the bit is set (like every refusal, it ends the VM with the failure verdict, so a driver must reset the device before ringing again), and a `Status` write keeps it (§2.1.1: the driver must not clear a status bit); only a reset, a write of 0, clears it.

used. len, and what a driver may trust

The `len` the device returns is the “total of bytes written into the buffer” (§2.7.8), counted over the device-writable part of the chain. For a block read that is 512 data bytes and one status byte, 513; for a request the device answers with an error status it is 0, because the first device-writable buffer is the data segment and nothing was written to it. The specification lets a device under-report: it “MAY write more than `len` bytes” (§2.7.8.2), and in exchange it tells the driver what it may assume (§2.7.8.3): “The driver MUST NOT make assumptions about data in device-writable buffers beyond the first `len` bytes, and SHOULD ignore this data.” A driver that follows that rule and sees a short `len` has to treat the tail as not written. For this request layout that is a real tension: the status byte sits after the 512 data bytes, so with `len` 0 (an error answer) a driver that trusts `len` never looks at the status, and with `len` 513 it can. A `len`-trusting driver can see an error status only if the device also writes the data. Chapter 11’s driver does not trust `len` at all; exercise (c) reads its source and what that choice costs.

The MMIO transport as a state machine

Chapter 21 gave the register table; what it did not need was the order, and the order is what a device model has to enforce. The driver walks a fixed sequence (§3.1.1, which §4.2.3.1 sends the MMIO driver to): reset; set `ACKNOWLEDGE` (1); set `DRIVER` (2); read the device’s feature bits and write back the subset it accepts; set `FEATURES_OK` (8); “re-read device status to ensure the `FEATURES_OK` bit is still set: otherwise, the device does not support our subset of features and the device is unusable”; set up the queues; set `DRIVER_OK` (4), “at this point the device is ‘live’”. The `Status` register carries the bits; writing zero resets the device (§4.2.2: “Writing zero (0x0) to this register triggers a device reset”), and a reset clears `InterruptStatus` and every `QueueReady` (§4.2.2.1).

Three places in that sequence need the model to do something.

Features. A device offers all it understands and the driver answers with a subset (§2.2). This one understands exactly one feature, `VIRTIO_F_VERSION_1`, bit 32

(§6: “This indicates compliance with this specification, giving a simple way to detect legacy devices or drivers”), which “A device **MUST** offer” (§6.2) and a driver **MUST** accept (§6.1). Feature words are read 32 bits at a time through a select register, so bit 32 is bit 0 of word 1: `DeviceFeaturesSel` reads 1 when `DeviceFeaturesSel` is 1 and 0 otherwise. The model checks the answer when the driver sets `FEATURES_OK`: it stays set only if the driver accepted word 1 equal to 1 and word 0 equal to 0, and the model clears it otherwise, which is the signal the driver’s read-back at step 6 is looking for. The driver’s `Status` read is therefore a real check and not a formality; the guest prints it, `0xb`, which is `ACKNOWLEDGE | DRIVER | FEATURES_OK`.

The block device has features of its own (§5.2.3: `FLUSH`, `BLK_SIZE`, `SEG_MAX` and others), and the model offers none of them. §5.2.5.2 says “Devices **SHOULD** always offer `VIRTIO_BLK_F_FLUSH`”; a RAM disk has nothing to flush, and the departure is a documented one, not a silent one.

The moment the device learns the ring. The driver programs queue 0 in seven steps (§4.2.3.2): select it with `QueueSel`; read `QueueReady` and expect zero; read `QueueNumMax`; zero the memory; write the size to `QueueNum`; write the three guest-physical addresses to the `QueueDesc`, `QueueDriver` and `QueueDevice` low/high pairs; “Write `0x1` to `QueueReady`”. Until the last of those, the device is only collecting numbers. The model stores each write in its `vdev`, and `QueueReady` = 1 is where it builds the `struct virtq`: it checks that the size is a power of two no larger than `QueueNumMax` and that each of the three addresses has the specification’s alignment (16, 2 and 4), translates each through `gpa2hva` with the size the table gives ($16 * N$, $6 + 2 * N$, $6 + 8 * N$) so that a ring that leaves the memory slot is refused, and records the pointers. §4.2.2.1 draws the line from the other side: “The device **MUST NOT** access virtual queue contents when `QueueReady` is zero (`0x0`)”. A device that read the addresses as they were written would be dereferencing numbers the driver was still in the middle of choosing.

The doorbell. `QueueNotify` is written with the queue index (§4.2.3.3, when `VIRTIO_F_NOTIFICATION_DATA` has not been negotiated). It is the one exit per request, and §2.1.2 gives the model its precondition: “The device **MUST NOT** consume buffers or send any used buffer notifications to the driver before `DRIVER_OK`.” A write to it before `DRIVER_OK`, or before the queue is ready, or with an index other than 0, is a driver bug, and the model refuses it the way it refuses every register access it cannot answer.

Width. §4.2.2.2: “The driver **MUST** only use 32 bit wide and aligned reads and writes to access the control registers described in table 4.1. For the device-specific

configuration space, the driver **MUST** use 8 bit wide accesses for 8 bit wide fields, 16 bit wide and aligned accesses for 16 bit wide fields and 32 bit wide and aligned accesses for 32 and 64 bit wide fields.” The model therefore accepts `len == 4` and nothing else, in the control registers and at `Config + 0` and `Config + 4`, the two halves of the 64-bit `capacity` field. A single 64-bit load of `capacity` is a driver bug and the model says so (exercise (e) shows it). The same clause makes the direction column of Table 4.1 binding on the driver, “**MUST NOT** write to the read-only registers (direction R) and **MUST NOT** read from the write-only registers (direction W)”, so the model’s switch has a separate list for reads and for writes and falls to -1 on everything else. `VendorID` is the one register the specification leaves meaningless (“Virtio Subsystem Vendor ID”); the model returns `0x554d4551`, the ASCII bytes `QEMU` in little-endian order, and nothing in the lab reads it. A general driver also reads a field wider than 32 bits inside a loop on `ConfigGeneration` (§2.5.1: drivers “**MUST NOT** assume reads from fields greater than 32 bits wide are atomic”), so that a change between the two halves is caught; this model’s generation is constant, and the lab’s driver omits the loop because its extra reads would change the pinned exit count.

The block request

A block request (§5.2.6) is a header, the data, and a status byte:

```
struct virtio_blk_req {
    le32 type;
    le32 reserved;
    le64 sector;
    u8 data[];
    u8 status;
};
```

On the ring the header is the first descriptor (16 bytes, device-readable), the data the second (512 bytes, device-writable for a read) and the status byte the third (one byte, device-writable), in the order §2.7.4.2 requires. `type 0` is `VIRTIO_BLK_T_IN`, a read; the other types, a write among them, are not served (only a request shaped as a read, with another type, is answered, with `UNSUPP`; the next paragraph says what happens to the rest), and the status byte the device writes is “either `VIRTIO_BLK_S_OK` for success, `VIRTIO_BLK_S_IOERR` for device or driver error or `VIRTIO_BLK_S_UNSUPP` for a request unsupported by device”: 0, 1 and 2. The sector number is in units of 512 bytes, and the device’s size is in the same units: “The capacity of the device (expressed in 512-byte sectors) is always present” (§5.2.4), a `le64` at offset 0 of the device-specific configuration, which sits at `0x100` of the MMIO window. The RAM disk has 16 sectors, and

sector s , byte i holds $(s * 7 + i) \& 0xff$: sector 0 starts 00 01 02 ..., sector 1 starts 07 08 09 ..., and a read of the wrong sector, or a copy to the wrong place, shows in the first bytes. The driver “MUST NOT submit a request which would cause a read or write beyond capacity” (§5.2.6.1); the device does not take that on trust, and answers a sector past the end with IOERR.

The device model serves one arrangement of descriptors, exactly these three, and sets `DEVICE_NEEDS_RESET` on any other. That is narrower than the specification asks of a full device: §2.7.4.1 says “The device MUST NOT make assumptions about the particular arrangement of descriptors”, so a complete block backend gathers the readable descriptors into the header and the writable ones into data and status, however the driver cut them. The model checks the arrangement because it is cheap and the lab’s driver is the only one.

Why polling is honest here

When the device uses a buffer it is obliged to say so (§2.7.7.2): with the available ring’s flags at 0, “the device MUST send a notification” (this driver never sets the flag, and the device honours it if it did). On the MMIO transport that notification is bit 0 of `InterruptStatus`, “the interrupt was asserted because the device has used a buffer in at least one of the active virtual queues”, and a line to the interrupt controller. The model sets the bit; nothing asserts a line, because no lab in this part injects an interrupt into a guest (the mechanism is `hvip` on the hardware and `KVM_INTERRUPT` or `KVM_IRQ_LINE` through KVM, as chapter 21 noted, and `hearth` injects its own in chapter 38). The guest polls `used.idx` instead and still performs the register half of the protocol: it reads `InterruptStatus`, sees 1, and writes the value back to `InterruptACK` (§4.2.3.4.1: “After the interrupt is handled, the driver MUST acknowledge it by writing a bit mask corresponding to the handled events to the `InterruptACK` register”). Polling is honest because the data path is the ring and the doorbell. An interrupt tells the driver when to look at `used.idx`; it changes when the driver finds the completion and nothing about how the device produced it.

The three axes

Chapter 14’s axes apply to the backend as they did to the UART. *Shared*: the three tables and the buffers they name, all in guest memory the VMM maps through the slot, and the register window; the device’s cursors and the RAM disk are the VMM’s own. *TCB*: `virtq_pop` and `virtq_push` are the code that reads untrusted memory, fifty-five lines between them, comments included; in `wick` they run in a process the jailer confines (chapter 26) and in `hearth` they are

part of the hypervisor image, where a bug in a bounds check is a bug in the TCB. *Copy*: one memcpy of 512 bytes from the RAM disk into the guest's buffer per request, and one exit; the descriptors and the buffers cost no exit and no second copy.

Walkthrough

The contract is `virtq.h`: the three structures of the Mechanism section and the two prototypes. `virtq_pop` is the device's half of §2.7.13.

```
/* The ring is the driver's memory, and the driver may be hostile:
↳ every
* field is read once, with a relaxed atomic load, so the value
↳ checked is
* the value used. */
#define LOAD(x) __atomic_load_n(&(x), __ATOMIC_RELAXED)

int virtq_pop(struct virtq *q, struct virtq_chain *c)
{
    /* Acquire: everything the driver wrote before it bumped idx (the
↳ ring
* entry, the descriptors) is visible after this load (2.7.13.3.1
↳ is
* the driver's half of this pair). */
    uint16_t avail_idx = __atomic_load_n(&q->avail->idx,
↳ __ATOMIC_ACQUIRE);
    if (avail_idx == q->last_avail)
        return 0; /* 2.7.6: nothing
↳ new */
    if ((uint16_t)(avail_idx - q->last_avail) > q->num)
        return -1; /* 2.7.13: more
↳ than the table holds */
    uint16_t head = LOAD(q->avail->ring[q->last_avail % q->num]);
    uint16_t i = head;
    unsigned n = 0;
    int seen_writable = 0;

    c->head = head;
    for (;;) {
        if (i >= q->num || n >= VIRTQ_MAX_SEGS)
            return -1; /* bad index, or
↳ too long / a loop */
        struct virtq_desc *dp = &q->desc[i]; /* the table is
↳ the driver's: read, never write (2.7.5.1) */
        uint64_t addr = LOAD(dp->addr);
```

```

uint32_t len    = LOAD(dp->len);
uint16_t flags  = LOAD(dp->flags);
uint16_t next   = LOAD(dp->next);
if (flags & VIRTQ_DESC_F_INDIRECT)
    return -1;                                /* 2.7.5.3: not
    ↳ negotiated */
int w = (flags & VIRTQ_DESC_F_WRITE) != 0;
if (seen_writable && !w)
    return -1;                                /* 2.7.4.2 */
seen_writable |= w;
void *hva = q->gpa2hva(addr, len, q->ctx);
if (!hva)
    return -1;
c->seg[n].hva = hva;
c->seg[n].len = len;
c->seg[n].writable = w;
n++;
if (!(flags & VIRTQ_DESC_F_NEXT))
    break;
i = next;
}
c->nsegs = n;
q->last_avail++;
return 1;
}

```

The `LOAD` macro at the top of the listing is the single-read rule from the Mechanism section: a relaxed atomic load of one field. Everything the function returns is a copy of values it loaded once, and `last_avail` is incremented only on the path that returns 1.

```

void virtq_push(struct virtq *q, uint16_t head, uint32_t written)
{
    uint16_t idx = q->next_used;                /* private: the
    ↳ driver can write used.idx, so it is never read back */
    q->used->ring[idx % q->num].id = head;
    q->used->ring[idx % q->num].len = written;
    q->next_used = (uint16_t)(idx + 1);
    /* 2.7.8.2: len, and the bytes it counts, are written before idx.
     * Release: the element and the data are visible before the index.
    ↳ */
    __atomic_store_n(&q->used->idx, q->next_used, __ATOMIC_RELEASE);
}

```

`virtq_push` is the device's obligation from §2.7.8.2: element first, index after, with the release between them.

The harness is the driver and the memory. It keeps guest-physical memory as a 4096-byte array and a `gpa2hva` that adds an address to its base, plays the driver's half of §2.7.13 in `driver_offer` (ring entry, release, index), and checks six things. The second is the one a block device lives on: the chain of three in §2.7.4.2's order.

```
/* 2. A block read in 2.7.4.2's order: header (device-readable),
 * data and status (device-writable). */
queue_reset();
set_desc(0, HDR_OFF, 16, VIRTQ_DESC_F_NEXT, 1);
set_desc(1, DATA_OFF, 512, VIRTQ_DESC_F_NEXT | VIRTQ_DESC_F_WRITE, 2);
set_desc(2, STAT_OFF, 1, VIRTQ_DESC_F_WRITE, 0);
driver_offer(0);
if (pop(&c) != 1) fail("3-seg chain: pop did not return 1");
char pat[3 * VIRTQ_MAX_SEGS + 1] = "";
for (unsigned i = 0; i < c.nsegs; i++)
    strcat(pat, i ? (c.seg[i].writable ? ",wo" : ",ro") :
↳ (c.seg[i].writable ? "wo" : "ro"));
if (c.head != 0 || c.nsegs != 3 || strcmp(pat, "ro,wo,wo") != 0 ||
    c.seg[0].hva != mem + HDR_OFF || c.seg[0].len != 16 ||
    c.seg[1].hva != mem + DATA_OFF || c.seg[1].len != 512 ||
    c.seg[2].hva != mem + STAT_OFF || c.seg[2].len != 1)
    fail("3-seg chain: wrong segments");
printf("harness: 3-seg read chain: head %u segs %u %s\n", c.head,
↳ c.nsegs, pat);
```

The other five are an empty queue (pop returns 0), the used ring after a push, twenty chains through an eight-entry ring with reused descriptors, a run that begins at 65530 and carries the counters across 65536 without printing anything, and the malformed rings and chains: writable before readable, which the output names, and also an `avail.idx` more than a table ahead, a head past the table, a next past the table, a loop, a descriptor with the `INDIRECT` flag (not negotiated), and a buffer outside the memory, each refused with `last_avail` unmoved. Two more silent checks guard the other direction: a ring offered exactly full (num chains, all of them served, then 0) is legal and must not be refused, and a push ignores a `used.idx` the driver has scribbled on. The harness has caught the bugs it was written for: removing the order check in `virtq_pop` stops it at `wo,ro`: not refused; removing the segment bound makes the loop case run off the end of the segment array and abort; not advancing `last_avail` or not reducing the used slot modulo the queue size stops it in step 4; accepting an unmappable address stops it at `unmappable`: not refused. Each of those is a one-line edit to a scratch copy of `virtq.c`.

The VMM adds two things to chapter 21's, the register file and the request loop. `mmio_virtio` is the model the Mechanism section described: width first, then a read switch and two write switches (the second is for queue 0 only). Every return `-1` is a refusal, and the exit loop turns a refusal into the same “no device here” line that chapter 21 printed.

```
/* The register file. The driver MUST use 32-bit aligned accesses to
↪ the
↪ * control registers and to 64-bit config fields (4.2.2.2), MUST NOT
↪ read
↪ * the write-only registers or write the read-only ones, and MUST NOT
↪ touch
↪ * anything not in Table 4.1: this model refuses all of it. */
static int mmio_virtio(struct kvm_run *run, struct vdev *dev, unsigned
↪ char *mem)
{
    unsigned long off = run->mmio.phys_addr - VIRTIO_BASE;
    uint32_t v;

    if (run->mmio.len != 4) return -1;
    if (!run->mmio.is_write) {
        switch (off) {
            case R_MAGIC:      v = 0x74726976; break;          /*
↪ "virt" */
            case R_VERSION:    v = 2; break;
            case R_DEVICE_ID:  v = 2; break;                  /*
↪ 5.2.1: block */
            case R_VENDOR_ID:  v = 0x554d4551; break;         /*
↪ "QEMU", little-endian: any value is allowed */
            case R_DEV_FEATURES: v = dev->dev_feat_sel == 1;   /* bit
↪ 32, VERSION_1, and no other */
                                break;
            case R_QUEUE_NUM_MAX: v = dev->queue_sel == 0 ? QUEUE_MAX : 0;
↪ break;
            case R_QUEUE_READY: v = dev->queue_sel == 0 ? dev->queue_ready
↪ : 0; break;
            case R_INT_STATUS:  v = dev->int_status; break;
            case R_STATUS:     v = dev->status; break;
            case R_CONFIG_GEN:  v = 0; break;                  /* the
↪ config never changes */
            case R_CONFIG:     v = DISK_SECTORS; break;        /*
↪ 5.2.4: capacity, low half */
            case R_CONFIG + 4:  v = 0; break;                  /* high
↪ half */
            default: return -1;
        }
    }
}
```



```

    }
    memcpy(run->mmio.data, &v, 4);
    return 0;
}

memcpy(&v, run->mmio.data, 4);
switch (off) {
case R_DEV_FEAT_SEL: dev->dev_feat_sel = v; return 0;
case R_DRV_FEAT_SEL: dev->drv_feat_sel = v; return 0;
case R_DRV_FEATURES:
    if (dev->drv_feat_sel > 1) return -1;
    dev->drv_feat[dev->drv_feat_sel] = v;
    return 0;
case R_QUEUE_SEL:    dev->queue_sel = v; return 0;
case R_STATUS:
    if (v == 0) { dev_reset(dev); return 0; } /*
    ↪ 4.2.2: writing 0 resets */
    if ((v & S_FEATURES_OK) && !features_ok(dev)) v &=
    ↪ ~S_FEATURES_OK;
    dev->status = v | (dev->status & S_NEEDS_RESET); /*
    ↪ NEEDS_RESET survives driver writes; only the 0 reset clears it */
    return 0;
case R_INT_ACK:      dev->int_status &= ~v; return 0;
case R_QUEUE_NOTIFY:
    /* The value is the queue index (4.2.3.3, NOTIFICATION_DATA
    ↪ not
    * negotiated). The device MUST NOT consume buffers before
    * DRIVER_OK (2.1.2) nor touch the queue before QueueReady
    * (4.2.2.1): a driver that rings the bell early is refused.
    ↪ */
    if (v != 0 || !(dev->status & S_DRIVER_OK) ||
    ↪ !dev->queue_ready ||
        (dev->status & S_NEEDS_RESET)) return -1;
    blk_service(dev);
    return 0;
}
/* Everything below is about queue 0 only. */
if (dev->queue_sel != 0) return -1;
switch (off) {
case R_QUEUE_NUM:    dev->queue_num = v; return 0;
case R_QUEUE_DESC_LO: dev->desc = (dev->desc & ~0xffffffffFULL)
    ↪ | v; return 0;
case R_QUEUE_DESC_HI: dev->desc = (dev->desc & 0xffffffffFULL) |
    ↪ (uint64_t)v << 32; return 0;

```

```

case R_QUEUE_DRV_LO: dev->avail = (dev->avail & ~0xffffffffFULL)
↳ | v; return 0;
case R_QUEUE_DRV_HI: dev->avail = (dev->avail & 0xffffffffFULL) |
↳ (uint64_t)v << 32; return 0;
case R_QUEUE_DEV_LO: dev->used = (dev->used & ~0xffffffffFULL)
↳ | v; return 0;
case R_QUEUE_DEV_HI: dev->used = (dev->used & 0xffffffffFULL) |
↳ (uint64_t)v << 32; return 0;
case R_QUEUE_READY:
    if (v == 0) { dev->queue_ready = 0; return 0; }
    if (v != 1 || queue_setup(dev, mem) < 0) return -1;
    dev->queue_ready = 1;
    return 0;
}
return -1;
}

```

blk_service runs on the doorbell: it pops chains until the ring is empty, checks the shape, reads the header out of guest memory, serves the read from the RAM disk, writes the status byte, and pushes.

```

/* QueueNotify: drain the available ring. Each chain is one block
↳ request,
* 5.2.6: a 16-byte header the device reads, 512 bytes it writes, a
↳ status
* byte it writes. Anything else is a driver the device cannot serve:
↳ it
* sets DEVICE_NEEDS_RESET and a configuration-change notification
* (2.1.2) and stops. At most num chains are served per doorbell: that
↳ is
* everything the driver can have offered (virtq_pop refuses a driver
↳ more
* than num ahead), and it bounds the loop even if the rings overlap.
↳ */

```

```

static void blk_service(struct vdev *dev)
{
    static int told;
    struct virtq_chain c;
    int r = 0;

    for (unsigned n = 0; n < dev->q.num && (r = virtq_pop(&dev->q,
↳ &c)) == 1; n++) {
        if (c.nsegs != 3 ||
            c.seg[0].writable || c.seg[0].len != 16 ||
            !c.seg[1].writable || c.seg[1].len != 512 ||
            !c.seg[2].writable || c.seg[2].len != 1) {

```

```

        r = -1;
        break;
    }
    uint32_t type;
    uint64_t sector;
    memcpy(&type, c.seg[0].hva, 4);
    ↪ little-endian host */
    memcpy(&sector, (unsigned char *)c.seg[0].hva + 8, 8);

    uint8_t status = BLK_S_OK;
    uint32_t written = 512 + 1;
    ↪ data and the status byte */
    if (type != BLK_T_IN) {
        status = BLK_S_UNSUPP;
        written = 0;
    }
    ↪ data segment is untouched (2.7.8.2) */
    } else if (sector >= DISK_SECTORS) {
        status = BLK_S_IOERR;
        written = 0;
    } else {
        memcpy(c.seg[1].hva, disk[sector], 512);
    }
    *(uint8_t *)c.seg[2].hva = status;
    virtq_push(&dev->q, c.head, written);
    dev->served++;
    ↪ The used.idx store above and this load of avail.flags are a
    ↪ store
    ↪ * then a load: release and acquire do not order them, a full
    ↪ fence does. */
    __atomic_thread_fence(__ATOMIC_SEQ_CST);
    ↪ 2.7.7.2: with avail.flags 0, the device notifies. */
    if (!(__atomic_load_n(&dev->q.avail->flags, __ATOMIC_RELAXED)
    ↪ & 1))
        dev->int_status |= 1;
}
if (r < 0) {
    if (virtq_mine_stub && !told) {
        printf("MINE: virtq_pop stub\n");
        fflush(stdout);
        told = 1;
    }
    dev->status |= S_NEEDS_RESET;
    dev->int_status |= 2;
}
}
}

```

The bytes the driver sees, in the order it sees them, are the memcpy into `seg[1]`, the status store into `seg[2]`, and the push that publishes both. A request of the right shape for a read, but for a sector outside the RAM disk, is answered with `IOERR`, and one with a type other than 0 with `UNSUPP`, each with a `len` of 0. A request of any other shape (a conformant write, whose data segment is device-readable, a flush, a get-ID) is not answered at all: it puts the device in `DEVICE_NEEDS_RESET`.

The guest is a freestanding C program for `rv64ima_zicsr` linked at `0x80000000`, the base of the VMM's 2 MiB slot, by `guest/link.ld`, with its stack at the top of that slot. `guest/start.S` sets the stack pointer, zeroes `.bss` (the file holds only `.text`, `.rodata` and `.data`, as `objcopy -O binary -j extracts` them), calls `main`, and then makes chapter 19's shutdown call, the SBI `SRST` `ecall` that KVM turns into the VM's shutdown. Its address arithmetic is `auipc` and `addi` pairs with `relaxation off` (`.option norelax` in the assembly and `-mno-relax` for the C), as in chapter 21, for the same reason: the image is linked and loaded at one address and must not depend on a linker's folding of it. The three rings are static arrays aligned to 4096 in `.bss`. The guest runs with its VS-stage translation Bare, so its addresses are guest-physical addresses, and the address the compiler gives an array is the one the driver writes into the queue registers; the driver prints nothing about any of them. The driver then follows the specification in order:

```
static int blk_read_sector0(void)
{
    if (rd(R_MAGIC) != 0x74726976u) { fail("MagicValue"); return 1; }
    ↪ /* 4.2.3.1.1 */
    show("MagicValue", 0x74726976u);
    if (rd(R_VERSION) != 2) { fail("Version"); return 1; }
    show("Version", 2);
    if (rd(R_DEVICE_ID) != 2) { fail("DeviceID"); return 1; }
    ↪ /* 5.2.1: block */
    show("DeviceID", 2);

    /* 3.1.1: reset, ACKNOWLEDGE, DRIVER, features, FEATURES_OK, read
    ↪ back. */
    wr(R_STATUS, 0);
    wr(R_STATUS, S_ACKNOWLEDGE);
    wr(R_STATUS, S_ACKNOWLEDGE | S_DRIVER);
    wr(R_DEV_FEAT_SEL, 1);
    ↪ /* word 1 holds bit 32 */
    uint32_t f = rd(R_DEV_FEATURES);
    show("DeviceFeatures[1]", f);
}
```

```

if (!(f & 1)) { fail("VERSION_1 not offered"); return 1; }
↳ /* section 6 */
wr(R_DRV_FEAT_SEL, 1);
wr(R_DRV_FEATURES, 1);
↳ /* VERSION_1, nothing else */
wr(R_DRV_FEAT_SEL, 0);
wr(R_DRV_FEATURES, 0);
wr(R_STATUS, S_ACKNOWLEDGE | S_DRIVER | S_FEATURES_OK);
uint32_t st = rd(R_STATUS);
show("Status", st);
if (!(st & S_FEATURES_OK)) { fail("FEATURES_OK refused"); return
↳ 1; }

/* 4.2.3.2: queue 0. */
wr(R_QUEUE_SEL, 0);
if (rd(R_QUEUE_READY) != 0) { fail("queue already ready"); return
↳ 1; }
uint32_t max = rd(R_QUEUE_NUM_MAX);
show("QueueNumMax", max);
if (max < QNUM) { fail("queue too small"); return 1; }
wr(R_QUEUE_NUM, QNUM);
uint64_t d = (uint64_t)(uintptr_t)desc, a =
↳ (uint64_t)(uintptr_t)&avail, u = (uint64_t)(uintptr_t)&used;
wr(R_QUEUE_DESC_LO, (uint32_t)d); wr(R_QUEUE_DESC_HI,
↳ (uint32_t)(d >> 32));
wr(R_QUEUE_DRV_LO, (uint32_t)a); wr(R_QUEUE_DRV_HI,
↳ (uint32_t)(a >> 32));
wr(R_QUEUE_DEV_LO, (uint32_t)u); wr(R_QUEUE_DEV_HI,
↳ (uint32_t)(u >> 32));
wr(R_QUEUE_READY, 1);
wr(R_STATUS, S_ACKNOWLEDGE | S_DRIVER | S_FEATURES_OK |
↳ S_DRIVER_OK);

/* 5.2.4: capacity is a le64 at config offset 0; 4.2.2.2 says
↳ 64-bit
   * fields are read as two aligned 32-bit halves. */
uint64_t cap = rd(R_CONFIG) | (uint64_t)rd(R_CONFIG + 4) << 32;
show("capacity", cap);

/* 5.2.6: header (device-readable), 512 bytes and a status byte
   * (device-writable), chained in that order (2.7.4.2). */
req.type = 0;
↳ /* VIRTIO_BLK_T_IN */
req.reserved = 0;
req.sector = 0;

```

```

    req_status = 0xff;
↪ /* a value the device cannot leave behind */
    desc[0] = (struct virtq_desc){ (uint64_t)(uintptr_t)&req, 16,
↪ VIRTQ_DESC_F_NEXT, 1 };
    desc[1] = (struct virtq_desc){ (uint64_t)(uintptr_t)data, 512,
↪ VIRTQ_DESC_F_NEXT | VIRTQ_DESC_F_WRITE, 2 };
    desc[2] = (struct virtq_desc){ (uint64_t)(uintptr_t)&req_status,
↪ 1, VIRTQ_DESC_F_WRITE, 0 };

    /* 2.7.13: head into the ring, barrier, idx, barrier, notify. */
    avail_ring[avail_idx % QNUM] = 0;
    wmb();
    __atomic_store_n(&avail_idx, (uint16_t)(avail_idx + 1),
↪ __ATOMIC_RELEASE);
    __asm__ volatile("fence" ::: "memory");
↪ /* idx before the doorbell */
    wr(R_QUEUE_NOTIFY, 0);
↪ /* 4.2.3.3: the queue index */

    while (__atomic_load_n(&used_idx, __ATOMIC_ACQUIRE) != 1)
        ↪ /* polled: no interrupt */
        ;
    show("status", req_status);
    show("used len", used_ring[0].len);
    puts("guest: sector 0:");
    for (int i = 0; i < 16; i++) {
        uart_putc(' ');
        put_nibbles(data[i], 2);
    }
    uart_putc('\n');

    uint32_t is = rd(R_INT_STATUS);
    ↪ /* 4.2.2: bit 0, used buffer */
    show("InterruptStatus", is);
    wr(R_INT_ACK, is);
    return 0;
}

```

The first three checks of the driver are §4.2.3.1.1's, on MagicValue, Version and DeviceID; the next block is §3.1.1 steps 1 to 6; the queue block is §4.2.3.2; the request is §5.2.6 and §2.7.13, with the head into the ring, a release, the index, and a full fence before the doorbell so that the index is visible before the exit begins. The loop polls `used_idx` with an acquire load. Every number the guest prints is a value it computed from the device's answers: the three identification registers, the features word, the status, `QueueNumMax`, `capacity`, the request's status byte,

`used.ring[0].len`, the first 16 bytes of the sector and `InterruptStatus`.

The lab starts red in both halves. With `mine/virtq.c`, whose `virtq_pop` sets `virtq_mine_stub` and returns -1, the harness stops at its first check and the guest's read never completes; the Lab section shows the diff.

Exercises

Each was run on `talos` (QEMU 11.1.1, the lab's kernel and `initramfs`) in a scratch copy of `labs/ch22/`, one edit at a time, except (b), whose program is committed as `labs/ch22/stress.c`.

(a) Give the harness a queue of 16 (`NUM 16`, `AVAIL_OFF 0x100`, `USED_OFF 0x180`) and replace step 2's chain by nine chained 16-byte descriptors. What does `virtq_pop` do, and where does the number 8 come from?

Answer, p. 342

(b) In the guest driver, store `avail.idx` before `avail.ring[0]`, with no barrier between them (and put the chain at descriptors 4, 5, 6 so the head is not the zero the array starts as). Does the lab still pass? Then run `make stress` and say why the answer to the first question does not protect `wick`.

Answer, p. 342

(c) Make the device report `used.len` as 512 instead of 513 for a successful read. What changes, and what does chapter 11's driver do with `len`, by its source?

Answer, p. 343

(d) Make the guest read sector 16, one past the RAM disk's end. What does the device answer? Then delete the sector check from `blk_service` and read it again.

Answer, p. 343

(e) The capture says `vmm: mmio exits 613`. Derive it from the driver, then say what a second read would cost; then change the driver to read capacity with one 64-bit load and observe what the device does.

Answer, p. 344

(f) Write `virtq.h` as the Rust types and signatures `wick's ring` will have. Types only.

Answer, p. 345

Lab

```
make lab CH=22
make lab CH=22 IMPL=mime
make -C labs/ch22 stress
```

labs/ch22/Makefile is chapter 21's with these additions. A harness step compiles harness.c and the chosen virtq.c with the host's cc -O2 -Wall -Wextra -std=c11 and runs it, and the run does not stop on its failure, so that an IMPL=mime diff shows both halves. A guest step builds guest/start.S and guest/blk.c with -march=rv64ima_zicsr -mabi=lp64 -nostdlib -ffreestanding -mmodel=medany -mno-relax -O2, links them with guest/link.ld and extracts .text, .rodata and .data into overlay/guest.bin. The VMM is compiled with the chosen virtq.c, and overlay/init runs /vmm / guest.bin. The kernel, BusyBox and kernel headers come from chapter 20's build script and its hp/ch20 cache, as in chapter 21, so a host that has run chapter 20 builds nothing new. The captured run is the harness's six lines followed by the sentinel section of the boot.

IMPL=mime swaps in mime/virtq.c, where virtq_pop is the part to write: it sets virtq_mime_stub and returns -1. The backend cannot print, so the harness and the VMM do, once each, when they see -1 with the flag set; the reference never prints that line. The harness stops at its first check. In the VM the VMM's virtq_pop returns -1, the VMM prints the line and marks the device DEVICE_NEEDS_RESET, and the guest, which polls used.idx, never leaves its loop. The run ends when timeout ends QEMU, after TIMEOUT seconds (180), and the diff is:

```
-harness: empty queue pops 0
-harness: 3-seg read chain: head 0 segs 3 ro,wo,wo
-harness: push used idx 1 id 0 len 513
-harness: wrap: 20 chains, last_avail 20, used idx 20
-harness: malformed wo,ro refused
-harness: PASS
+MIME: virtq_pop stub
+harness: FAIL: empty queue: pop did not return 0
=== lab begin
...
guest: capacity 0x10
-guest: status 0x0
-guest: used len 0x201
...
+MIME: virtq_pop stub
```



```
+qemu-system-riscv64: terminating on signal 15 from pid 4107609
↳ (timeout)
```

(the head and tail of the diff of capture.txt against the run on straylight; the lines between are the guest's remaining output and the VMM's summary, absent in the mine run, and the process id in the last line is that run's). Writing virtq_pop is the lab.

With the reference the capture is:

```
harness: empty queue pops 0
harness: 3-seg read chain: head 0 segs 3 ro,wo,wo
harness: push used idx 1 id 0 len 513
harness: wrap: 20 chains, last_avail 20, used idx 20
harness: malformed wo,ro refused
harness: PASS
=== lab begin
/dev/kvm
vmm: KVM_GET_API_VERSION 12
vmm: memory slot 0: 0x200000 bytes at gpa 0x80000000
vmm: guest loaded at gpa 0x80000000
vmm: pc=0x80000000 mode=S
guest: MagicValue 0x74726976
guest: Version 0x2
guest: DeviceID 0x2
guest: DeviceFeatures[1] 0x1
guest: Status 0xb
guest: QueueNumMax 0x8
guest: capacity 0x10
guest: status 0x0
guest: used len 0x201
guest: sector 0: 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
guest: InterruptStatus 0x1
vmm: KVM_RUN exit 24 (KVM_EXIT_SYSTEM_EVENT) type 1 (SHUTDOWN) ndata 1
vmm: mmio exits 613
vmm: requests served 1
vmm: PASS
vmm rc=0
=== lab end
```

The 613 was observed before it was explained, and exercise (e) is the explanation. The harness is the contract and its six lines open the capture; the boot lines are the proof that the contract is enough to read a sector in a guest that did not share a line of code with the backend. Sector 0's bytes are the formula $(s * 7 + i) \& 0xff$ at $s = 0$, and the capture does not print a ring address, a pointer or the

guest image's size, none of which is a property of the protocol. A `make lab CH=22` took 20.2 seconds of wall time on `talos` and 10.9 seconds on `straylight` (14.9 for the run after a source edit), both with chapter 20's kernel cached; the `IMPL=mime` run on `straylight` took 189.8 seconds, the 180 of the timeout and the build. The run on `talos` and two runs on `straylight` are byte-identical to `capture.txt`.

Further reading

- Virtual I/O Device (VIRTIO) Version 1.2 (OASIS Committee Specification 01, July 2022): §2.7 (split virtqueues; §2.7.4.2, §2.7.5.1, §2.7.8.2, §2.7.13.3.1 for the rules this chapter's code enforces), §3.1.1 (device initialization), §4.2.2 and §4.2.3 (the MMIO register table and its driver requirements), §5.2 (the block device; §5.2.6 for the request), §6 (reserved feature bits).
- `ember/virtio.c` in this repository and chapter 11: the driver's half of the same protocol, in the same register names.
- `rust-vm/vm-virtio` and its `virtio-queue` crate: the Rust implementation of the split-queue device side that `wick` may read and need not depend on; its `DescriptorChain` and `Queue` are the shapes exercise (f) designs toward.

Answers

(a) Observed:

```
harness: empty queue pops 0
harness: FAIL: 3-seg chain: pop did not return 1
```

`virtq_pop` returns -1 at the ninth descriptor: `n` has reached `VIRTQ_MAX_SEGS` before the chain ends, and the bound is what stands between a driver that builds a loop and a backend that follows it. With `VIRTQ_MAX_SEGS` raised to 16 the same chain pops (the harness then stops one line later, at `3-seg chain: wrong segments`, because it expected three). The 8 is the queue size, the bound the Mechanism section sets. A bound lower than the queue size is a policy the device must document, and a bound above it is no bound at all.

(b) Observed: the lab passes, with `diff` against `capture.txt` reporting nothing. This is the single-threaded doorbell of the Mechanism section's barrier paragraph: both stores have completed, in whichever order, before `blk_service` reads the ring.

make stress runs the reference `virtq_pop` and `virtq_push` against a second thread that plays the driver. With the entry stored first and `avail.idx` after (§2.7.13's order) the device popped 20,000,000 chains with 0 wrong heads; with `avail.idx` first, `idx-first` order: 20000000 chains, 639 popped with the wrong head on one run on talos and 13809 on the next. The count is a race, and it varies by orders of magnitude; the point is that it is not zero. A device thread that sees the new `idx` reads the ring slot, and the slot still holds the previous lap's head. The failure here is the order of the stores in the program, which can fail on any machine (a release store orders the operations before it, not the plain store after it). A missing barrier with the right order is a different bug, and talos (x86-64, whose stores stay in program order) would not show it; it is not exercised here, and a weaker memory model such as RISC-V's can show it. wick's device thread runs beside the vCPU thread, so the ring it serves depends on both the order and the barrier.

(c) Observed: one line of the capture changes, `guest: used len 0x200` for `0x201`; the status, the sector bytes and everything else are as before, because the device still wrote the status byte, only without counting it. 512 is a legal under-report (§2.7.8.2), and by §2.7.8.3 the status byte at offset 512 is then past what a driver may trust: a driver that trusts `len` must treat the request as having no status.

ember's `blk_wait` never reads `len`: it compares the used ring's `id` with its head and reads `*req_status`, which `blk_op` set to `0xff` before submitting. Read from the source (the lab does not run ember), against this device with `len 512` it would behave exactly as before, and it would be correct only because of the sentinel: a status byte the device did not write reads `0xff`, which is not `S_OK`, so a missing write is a failure rather than a pass. A driver that left stale zeros there would read `S_OK` from a status the device never wrote.

(d) Observed, with the check: `guest: status 0x1 (IOERR)`, `guest: used len 0x0` (nothing in the data segment was written, and the status byte sits after it), and the 16 bytes of the data buffer are all `00`, the buffer as the guest left it. The device refused to read and said so in the protocol's own terms.

Observed, without it: `guest: status 0x0`, `guest: used len 0x201` and the bytes `00 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00`. That is a successful read of 512 bytes from `disk[16]`, which is not in the array: it is whatever the VMM process holds after it, handed to the guest as a disk sector. The bytes depend on the VMM's layout, and that dependence is what the exercise is for. The sector check is a memory-safety boundary between the guest and the VMM's

own memory, and the specification's "driver MUST NOT" does not move it.

(e) The count is two terms. The UART: the guest prints 11 lines, and `grep '^guest:' capture.txt | wc -c` prints 291 bytes, each one an LSR read and a THR write as in chapter 21, which is 582 exits. The device, from `blk_read_sector0`:

Phase	Reads	Writes
MagicValue, Version, DeviceID	3	0
reset, ACKNOWLEDGE, DRIVER	0	3
features: writes of DeviceFeaturesSel, DriverFeaturesSel (twice), DriverFeatures (twice) and Status (FEATURES_OK); reads of DeviceFeatures and Status	2	6
queue: QueueSel, QueueReady and QueueNumMax reads, QueueNum, the six address words, QueueReady, DRIVER_OK	2	10
capacity, two halves	2	0
the doorbell, QueueNotify	0	1
InterruptStatus and InterruptACK	1	1

That is 10 reads and 21 writes, 31 in all; 582 and 31 make 613, the observed count (the number was pinned by the run before this table was written, and the table is the explanation). The request itself, the whole sector, costs one of them: the doorbell. The other 612 are the console and the setup.

A second read, run as this driver is written (a silent second request, with the same interrupt handling), observed 616: the doorbell, InterruptStatus and InterruptACK, three exits, and none for the descriptors, the ring entry or the 512 bytes. A driver that polls the used ring and never touches the interrupt registers observed 614, one exit, the doorbell: the point of the ring. A second read never touches the queue-setup registers again; that is the saving.

With `capacity read as *(volatile uint64_t *) (VIRTIO_BASE + R_CONFIG)`, observed:

```
vmm: KVM_RUN exit 6 (KVM_EXIT_MMIO) read gpa 0x10001100 len 8: no
↳ device here
vmm: mmio exits 302
vmm: requests served 0
vmm: FAIL
vmm rc=1
```

The address is `Config + 0`, the direction is right, and the width is the load's: 8.

§4.2.2.2 asks for 32-bit accesses to 64-bit fields, and the model accepts nothing else.

(f) This is a design, and nothing checks it but the C: the harness’s six lines are the test it must pass in chapter 25. The mapping, field by field:

```
pub const MAX_SEGS: usize = 8;
pub enum Seg { Read(GuestSlice), Write(GuestSlice) } // virtq_seg:
↳ hva, len, writable
pub struct Chain { pub head: u16, pub segs: ArrayVec<Seg, MAX_SEGS> }
pub trait GuestMemory {                               // gpa2hva and
↳ its ctx
    fn translate(&self, gpa: u64, len: u32) -> Option<GuestSlice>;
}
pub enum Malformed { BadIndex, TooLong, WritableBeforeReadable,
↳ Unmappable, Indirect, AheadOfTable }
pub struct Virtq<M: GuestMemory> {                    // struct virtq
    num: u16, desc: u64, avail: u64, used: u64,        //
    ↳ guest-physical, not pointers
    last_avail: Wrapping<u16>,
    next_used: Wrapping<u16>,                          // the private copy
    ↳ of used.idx
    mem: M,
}
impl<M: GuestMemory> Virtq<M> {
    pub fn pop(&mut self) -> Result<Option<Chain>, Malformed>; // 1,
    ↳ 0, -1
    pub fn push(&mut self, head: u16, written: u32);
}
```

The return code 1, 0 and -1 becomes `Ok(Some)`, `Ok(None)` and `Err`, with the reason the C had to leave out as a variant for each of the six refusals in `virtq`. h. The function pointer and its context become a trait. The ring addresses stay guest-physical numbers in the struct, which is a change from the C: every access goes through `translate`, and a Rust ring that held raw pointers would have the same hostile-driver surface with more ways to keep it. `GuestSlice` is a type that cannot be a plain `&[u8]`, because the guest can change the bytes under any reference that outlives the next instruction: it is a pointer and a length whose accessors are volatile reads and writes, which is the role the single-load rule plays in the C. `last_avail` and `next_used` are `Wrapping<u16>` so that the free-running counter’s wrap is the type’s, not the programmer’s. The atomics in `pop` and `push` stay as they are, an acquire load of `avail.idx` and a release store of `used.idx`, on the guest memory the `GuestMemory` hands back. Where the C’s block backend validated “exactly three segments”, this signature validates

nothing about arrangement and leaves it to the device above it, which is where §2.7.4.1's gather belongs.