

OPERATING SYSTEMS ENGINEERING

Build and test the mechanisms behind traps, scheduling, concurrency, virtual memory, devices, filesystems, recovery, and isolation.

OFFICIAL DOCUMENTATION

November 2026

CONTENTS

CROSSING THE KERNEL BOUNDARY

The Operating-System Contract

Traps, Interrupts, and System Calls

Boot and Kernel Organization

PROCESSES AND CPU TIME

Processes, Threads, and Context Switches

Process API, IPC, and Signals

CPU Scheduling as Policy

Scheduler Lab: Weighted Virtual Time in xv6

CONCURRENCY AND LIVENESS

Interleavings and Atomicity

Locks, Atomics, and Memory Ordering

Condition Variables and Semaphores

Deadlock, Liveness, and Priority Inversion

VIRTUAL MEMORY

Address Spaces and Translation

Page Tables and TLBs

Page Faults, Copy-on-Write, and mmap

Replacement, Working Sets, and Thrashing

DEVICES AND I/O

Device I/O, Interrupts, and DMA

Storage Devices and I/O Scheduling

Device-Driver Lab: A Defensible Virtio Queue

PERSISTENCE

Filesystem Interface and Names

Filesystem Implementation

Crash Consistency and Journaling

Filesystem Lab: Recoverable Rename

PROTECTION, ISOLATION, AND SYNTHESIS

Protection and Authority

Virtual Machines

Containers and Resource Control

Multicore Scalability and Observability

Comparative Kernel Design

Operating Systems Capstone

COURSE PRACTICE

Operating Systems Practice Workshop

Crossing the kernel boundary

SECTION DETAIL

The Operating-System Contract

A storage controller accepts physical addresses. A malicious process knows the controller's register layout and submits a direct-memory-access descriptor targeting another process's page. If the CPU permits the write, the controller faithfully corrupts the victim while the attacker remains in user mode. A filesystem API, process identifier, and graphical desktop can all exist on such a machine; none supplies protection unless access to the underlying mechanism is mediated.

The operating system is necessary because hardware resources are shared and because the requests presented to them are not intrinsically trustworthy. Its central contract is stronger than "provide convenient functions":

> ***Kernel mediation contract.*** *Every transition that can affect protected state must be reached through a controlled entry, checked against the caller's authority, executed while preserving global invariants, and reported through an interface whose failure semantics are explicit.*

Boundary prediction — the hostile DMA descriptor

Before reading the model below, write four lines:

1. the state the attacker wants to change;
2. the first component that can prevent the change;
3. the authority evidence that component needs; and
4. one assumption under which prevention still fails.

Separate the CPU's privilege check from the device's ability to access memory.

The protected state is the victim's physical page. The CPU can prevent user code from programming the controller by mapping its registers as supervisor-only; an IOMMU can separately constrain the device's DMA addresses. The kernel needs both the caller's right to use the device and ownership of the submitted buffers. Protection can still fail if the kernel validates the wrong length, retains a stale mapping, misprograms the IOMMU, or trusts malicious firmware. Naming only "permissions" misses the two distinct access paths: CPU loads/stores and device DMA.

Six ideas that must not collapse into one

The same implementation may serve several roles, but the roles answer different questions.

DEFINITION

An operating-system design is not characterized by one abstraction label; it is characterized by the resources it represents, the choices it makes, and the authority it mediates.

Idea	Precise question	Example	Non-example
Abstraction	Which details does the interface hide while preserving useful operations?	A file exposes byte reads instead of disk-sector commands.	Renaming a sector number `file_id` without adding semantics.
Virtualization	How does one physical resource appear as multiple logical instances or as a different resource?	Each process observes a private virtual address space.	Merely serializing access to one global buffer.
Multiplexing	Which claimant receives a finite resource, when, and for how long?	A scheduler allocates CPU intervals among runnable threads.	Mapping two names to the same object without allocating time or capacity.
Mechanism	What operations make a policy enforceable?	Timer interrupts, saved contexts, and a ready queue permit preemption.	"Be fair" without a way to stop a running thread.
Policy	Which allowed outcome should the mechanism choose?	Select the runnable thread with the earliest eligible virtual deadline.	The context-switch assembly that saves registers.
Protection	Which subject may perform which operation on which object?	Page-table permissions reject a user-mode store to a kernel page.	Hiding an address while leaving it writable.

An abstraction can exist without isolation: two processes may hold descriptors for the same open file description. Virtualization can be partial: CPU time is virtualized while cache capacity remains visibly shared. Protection can be correct yet unavailable: a kernel that denies every request preserves confidentiality but offers no useful progress.

Retrieval under changed names

A storage service assigns each tenant a "virtual disk," queues all operations in one FIFO, and performs no access check because identifiers are 128-bit random values. Classify the design's three claims: abstraction, multiplexing, and protection.

The block interface and tenant-specific names provide an abstraction. The FIFO multiplexes one physical service path. Protection is not established: identifier unpredictability is not an authorization proof, and accidental disclosure grants access. A correct answer that calls the random identifier a capability must also establish unforgeability, controlled transfer, revocation semantics, and complete mediation; length alone does none of those things.

MECHANISM VERSUS POLICY CLASSIFICATION MATRIX

Evaluate each statement regarding how kernel mechanisms and scheduling policies are separated in operating systems architecture:

Answer: Hardware timer interrupt delivery, trapframe register saving, and low-level context switching are mechanisms.; Selecting the runnable thread with the highest priority or smallest accumulated virtual runtime is policy.

Mechanisms supply capabilities (timer traps, register saving, context switches) while policies define decision rules (thread priorities, quantum lengths, allocation strategies).

A kernel operation as a transition system

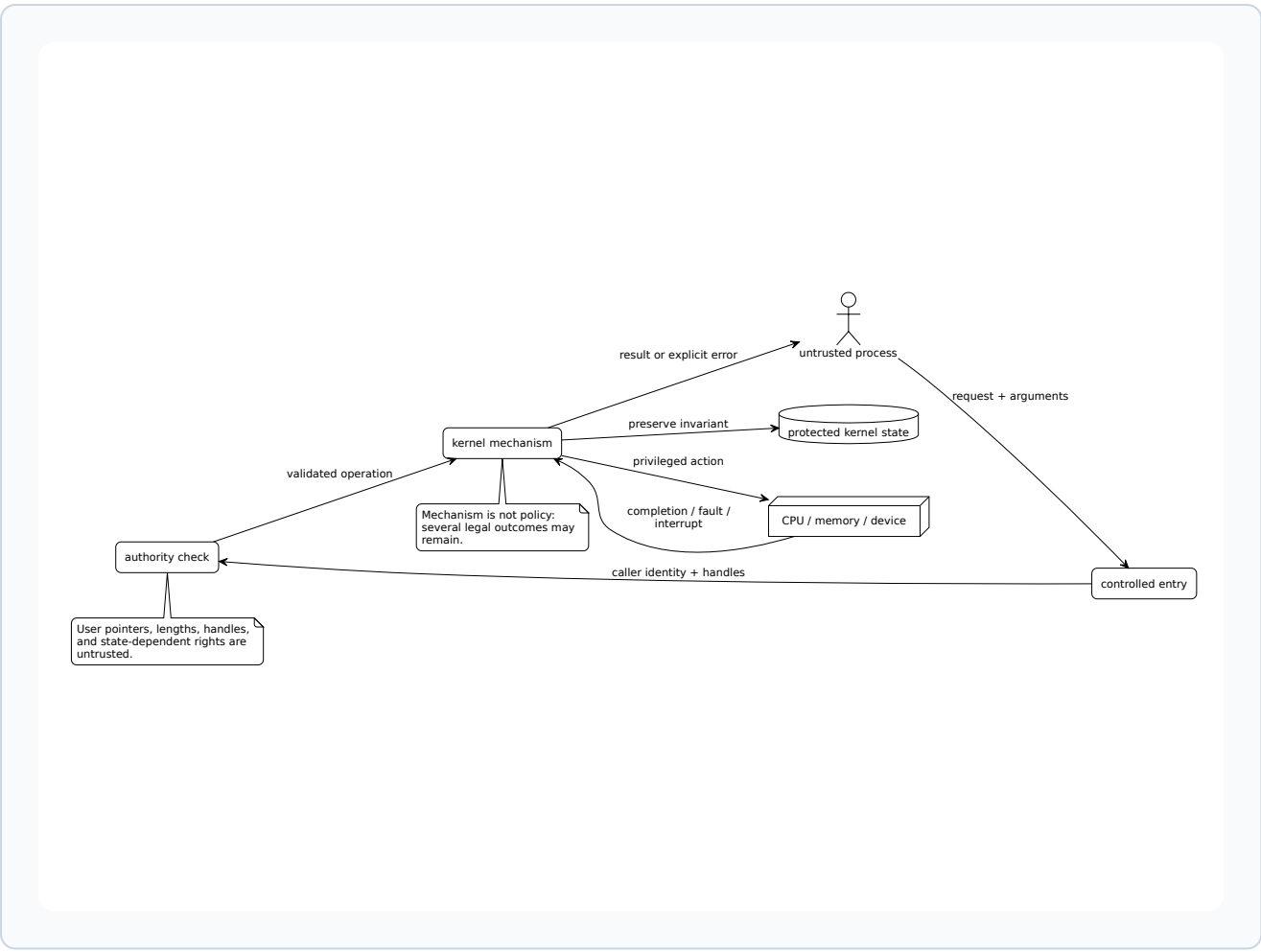
Let the machine state be

$$S = (H, K, P_1, \ldots, P_n, D),$$

where H is architecturally visible hardware state, K is kernel-owned state, each P_i is a protection domain, and D is device-visible state. An event e proposes a transition $S \to S'$. Events include explicit requests such as a system call, synchronous faults caused by the current instruction, and asynchronous interrupts raised by devices or timers.

The transition is acceptable only if three obligations hold:

1. **Authorization:** the subject represented by P_i has authority for the requested operation on the target object.
2. **Safety:** an invariant $I(S)$ implies $I(S')$. Examples include "a physical frame has at most one mutable owner unless sharing is explicit" and "a runnable thread is not simultaneously linked into two run queues."
3. **Progress under assumptions:** if required resources and external events eventually become available, the request either completes or returns a documented error. This is conditional liveness, not a promise against broken hardware or infinite contention.



The diagram is a proof obligation, not a literal function-call graph. Some kernels fuse entry and validation; a device completion may awaken a different thread; and hardware may reject a request before kernel code runs. The invariant must survive every path, including errors and cancellation.

Application — define an invariant before code

A kernel exposes `map_shared(frame_handle, va, writable)`. Two processes may map the same frame. Write one safety invariant that is strong enough to prevent unintended mutable aliasing but weak enough to permit deliberate shared memory. Then list the information the authorization check needs.

Ownership and writability need a relation; a count of mappings is insufficient.

One usable invariant is: “For every physical frame, either exactly one protection domain has a writable mapping, or every writable mapping belongs to a sharing group explicitly authorized by the frame owner; all other mappings are read-only.” The check needs the caller identity, the frame object denoted by the handle, the handle’s map and write rights, the requested virtual interval and permissions, the caller’s current address-space map, and the sharing-group policy. “At most one mapping” is too strong; “all handles may map” ignores rights encoded by the handle.

Worked trace: four bytes from a file

Assume one process calls `read(fd, buf, 4)`. The descriptor already denotes a readable open file; `buf` is a user virtual address; the requested bytes are not cached.

EXAMPLE

The trace records the controller and invariant at each cut so that a fast-path implementation can change without erasing the boundary proof.

Cut	Controller	Relevant state	Obligation before advancing
0	user process	`fd`, `buf`, register state	User code may prepare a request but cannot issue disk DMA.
1	trap hardware	saved user PC, cause, privilege state	Entry must land at a kernel-configured vector without trusting a user target.
2	kernel dispatcher	call number and raw arguments	Reject unknown call numbers, invalid descriptors, overflowed ranges, and unwritable destinations.
3	filesystem/device path	file offset, cache state, request queue	Serialize updates to shared file state; bind DMA to kernel-owned buffers.
4	scheduler	caller is blocked; another thread runs	A blocked thread must not remain marked running on the same CPU.
5	interrupt path	device reports completion	Match the completion to a live request before waking a waiter.
6	return path	four bytes copied; result register is `4`	Restore the caller's user state and no kernel-only register or mapping.

This trace crosses abstraction layers but does not imply a copy at each boundary. A production kernel might satisfy the read from a page cache, perform readahead, or map data into user space. The contract remains: descriptor authority and user memory are checked, shared offsets change atomically according to the interface, and failure is reported without exposing unrelated state.

Transfer — a cancellation cut

At cut 4, another thread requests cancellation of the blocked operation. The disk later completes anyway. Give one safe ownership protocol for the request object and explain why "free it on cancellation" is wrong.

Use a request state machine such as `QUEUED → IN_FLIGHT → COMPLETING → DONE`, with cancellation recording `CANCEL_REQUESTED` rather than freeing an in-flight object. The completion path owns the final transition, detaches the DMA mapping, records either completion or cancellation, wakes waiters, and releases the last reference. Immediate free permits the later interrupt to dereference reused memory or to copy device data into a new owner's buffer. The missing concept in that wrong answer is asynchronous lifetime, not merely locking.

Worked trace: one CPU, two processes, one page frame

Consider processes A and B on one CPU. A is running; B is runnable. No free physical frame exists. A requests a page that may be supplied by evicting a clean page owned by B.

> **EXAMPLE**
>
> CPU preemption and frame reassignment virtualize different resources, yet both require revoking old ownership before publishing new ownership.

1. A timer event transfers control to the kernel. The scheduler saves A's execution context, changes A from running to runnable, chooses B, and restores B.
2. Later, A runs and faults on an unmapped virtual page. The fault is synchronous: the current instruction cannot retire with its requested effect.
3. The kernel chooses a victim frame using a replacement policy. The unmap and TLB invalidation mechanisms make that policy enforceable.
4. Before reassigning the frame, every mapping that could write it as B's page must cease to authorize access. The kernel records the new owner and mapping for A.
5. The faulting instruction is retried. From A's perspective, a private page appeared; from the machine's perspective, CPU time and a frame were multiplexed at different timescales.

The CPU scheduler and memory manager solve different allocation problems but share one structure: observable abstraction, protected metadata, revocable hardware mechanism, and policy under scarcity.

Quantitative check — capacity is not a policy

Suppose A demands 8 ms of CPU every 20 ms and B demands 15 ms every 25 ms. Compute average utilization demand and decide what the number proves.

Add demand rates, then separate a necessary capacity condition from a schedule proof.

A demands $8/20 = 0.4$ CPU and B demands $15/25 = 0.6$, so average demand is 1.0. This establishes no scheduling guarantee. At exactly full utilization there is no slack for kernel work, release jitter, cache effects, or unfavorable alignment, and average feasibility does not prove every deadline. The diagnostic distinction is capacity versus policy plus timing assumptions.

Mechanism, policy, and evidence

"The kernel should be fair" is not implementable until fairness has an observable definition. Equal CPU time, weighted shares, bounded response time, and deadline satisfaction are different policies. A timer interrupt and context switch are mechanisms. A run queue is partly mechanism and partly policy because its ordering encodes a choice.

WARNING

Evidence for one claim layer cannot be promoted into another: elapsed time does not prove authorization, and a type-checked call does not prove progress.

Use this claim decomposition for any operating-system feature:

Claim layer	Required evidence
Interface	Inputs, outputs, error cases, concurrency semantics, and lifetime rules
Safety	Invariant plus an argument over all state transitions, including failure paths
Liveness	Workload and fairness assumptions; a bound or a counterexample when possible
Performance	Cost model, controlled measurements, uncertainty, and a competing explanation
Security	Threat model, authority path, trusted computing base, and revocation story

A benchmark cannot prove isolation. A type signature cannot prove progress. A successful happy-path test cannot prove recovery. Evidence must match the claim.

Synthesis — audit a “zero-copy” service

A proposed service maps a network device’s receive ring into every client for “zero-copy performance.” Clients are mutually hostile. Produce a compact mechanism map with: abstract objects, protected state, authority checks, one safety invariant, one liveness risk, and one measurement needed before accepting the performance claim.

A defensible design exposes per-client receive buffers or immutable packet views, not the shared writable ring. Protected state includes descriptor ownership, producer/consumer indices, DMA mappings, and buffer lifetimes. Authority checks bind a client to its queue and map only buffers assigned to it. A useful invariant is: each writable descriptor and DMA target has exactly one current owner; clients cannot advance device-owned indices. A slow client can retain buffers and exhaust the ring, so liveness needs quotas, revocation, or copying under pressure. Measure end-to-end throughput and tail latency while accounting for IOMMU operations, cache-line contention, retained buffers, and fallback copies. A response that says only “use an IOMMU” omits ownership and denial-of-service.

Boundaries of the contract

The kernel cannot create guarantees absent from its assumptions.

> **WARNING**
>
> Every unconditional whole-system guarantee is suspect until the hardware, trusted-code, load, and durability assumptions are named.

- **Hardware failure:** a CPU that violates privilege checks or memory ordering invalidates the software proof.
- **Trusted-code defect:** a monolithic kernel bug may bypass every user/kernel boundary because the buggy code already has supervisor authority.
- **Information leakage:** page isolation does not automatically hide timing through shared caches or queues.
- **Permanent overload:** no scheduler can complete unbounded demand with finite capacity.
- **Durability:** a successful write to volatile cache is not a persistence guarantee unless the interface defines and enforces a durability point.

The trusted computing base (TCB) is the set of components whose correct behavior is necessary for a security property. Smaller is not automatically correct, but every additional trusted parser, driver, or privileged service enlarges what must be audited.

> **REMARK**
>
> An omitted mechanism is not assumed away in an engineering assessment; any design that depends on one must state it as an external precondition.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 1.1.1: The Hostile DMA Descriptor and Privilege Boundaries

Description application

A storage controller accepts physical addresses directly from software descriptors. Explain why CPU privilege mode checks alone cannot prevent a hostile user process from corrupting physical memory via the controller, and identify the hardware and software mediation mechanisms required to enforce isolation.

Separate the CPU privilege check from the peripheral device memory bus access path.

The CPU privilege mode check restricts CPU instructions (loads and stores) from writing peripheral device registers when mapped as supervisor-only. However, once a peripheral initiates a direct memory access (DMA) transaction across the system bus, the access originates from the controller rather than the CPU core. Without an I/O Memory Management Unit (IOMMU) translating and filtering peripheral bus requests, a rogue or untrusted device descriptor can target any physical memory frame. Complete mediation requires mapping device control registers as supervisor-only and routing all DMA through an IOMMU programmed strictly with validated kernel-owned physical frame buffers.

Exercise 1.1.2: Random Identifier Authorization vs Protection

Verification application

A cloud storage service assigns each tenant a virtual disk, queues requests in a single FIFO queue, and uses 128-bit random tokens as handles without per-operation permission checks. Analyze whether this system achieves abstraction,

multiplexing, and protection.

Evaluate whether unguessable handles establish unforgeability, revocation, and complete mediation.

The system achieves abstraction (presenting disk-block operations) and multiplexing (sharing the physical storage path via the FIFO queue). However, it does not achieve true protection. A 128-bit random token provides obscurity, but tokens can be leaked, intercepted, or replayed. A sound capability protection system requires unforgeability, controlled delegation, explicit revocation semantics, and complete mediation at every access point.

Exercise 1.1.3: Shared Memory Invariants and Authority Verification

Formalisation transfer

For a kernel system call `map_shared(frame_handle, va, writable)`, formulate a mathematical safety invariant that permits intentional inter-process memory sharing while strictly preventing accidental or unauthorized mutable aliasing.

Formulate a condition relating physical frames to protection domains and authorized sharing groups.

A sound invariant is: For every physical frame F, either (1) at most one protection domain P_i holds a mapping with write permissions, or (2) all protection domains holding write permissions belong to an explicit sharing group G authorized by the frame owner. The authorization check must verify the caller identity, validate that `frame_handle` conveys mapping and writing rights, ensure the requested virtual range $[va, va + \text{len})$ is unmapped, and check group authorization before modifying the page table.

Exercise 1.1.4: Asynchronous Cancellation and Object Lifetime

Construction transfer

A user process invokes an asynchronous block read. While the request is in flight in the storage controller, another thread issues a cancellation request. Detail the state machine and reference counting rules required to prevent a use-after-free or buffer corruption hazard upon completion.

Track request lifecycle explicitly through state transitions rather than deallocating on cancel.

The request must follow an explicit lifecycle such as `QUEUED -> IN_FLIGHT -> COMPLETING -> DONE`. When cancellation is requested while `IN_FLIGHT`, the kernel marks the state `CANCEL_REQUESTED` but retains all buffer allocations and DMA mappings. When the hardware interrupt fires, the completion handler observes `CANCEL_REQUESTED`, tears down the DMA mapping, records an aborted status, and releases the final reference. Deallocating memory immediately on cancellation allows the pending DMA write to corrupt reallocated memory.

Exercise 1.1.5: Capacity Limits vs Feasible Scheduling Guarantees

Verification synthesis

Two tasks run on a uniprocessor: Task A requires 8 ms of CPU every 20 ms, and Task B requires 15 ms of CPU every 25 ms. Compute the total utilization demand and evaluate whether satisfying the capacity bound guarantees that all deadlines are met.

Sum utilization fractions and evaluate whether nominal capacity implies schedulability without scheduling assumptions.

Utilization is $U = \frac{8}{20} + \frac{15}{25} = 0.40 + 0.60 = 1.00\,(100\%)$. While nominal demand equals total uniprocessor capacity, this does not guarantee deadline satisfaction. Zero slack exists for context switching overhead, timer interrupt latency, or cache refill penalties. Furthermore, under rate-monotonic or earliest-deadline-first scheduling, phase alignment and non-preemptive regions can cause immediate deadline misses. Capacity is a necessary precondition, not a proof of liveness.

Exercise 1.1.6: Complete Mechanism Map and Safety Invariant Defense

Construction synthesis

Construct a structured mechanism map for a microkernel IPC subsystem transferring messages between mutually untrusted processes. Specify the state tuple, controlled entry, authorization check, safety invariant, and a falsifiable performance measurement plan.

Decompose the IPC subsystem into abstract state, authority checks, invariants, and measurable tail latency metrics.

State tuple: $S = (K_{\mathrm{c}}^{\mathrm{ip}}, P_{\mathrm{r}}^{\mathrm{sende}}, P_{\mathrm{r}}^{\mathrm{receive}}, K_{\mathrm{c}}^{\mathrm{ip}})$, where $K_{\mathrm{c}}^{\mathrm{ip}}$ tracks channel endpoints, message queues, and capability tokens. Controlled entry: system call `ipc_send(cap_endpoint, msg_ptr, len)`. Authority check: verifies caller owns the endpoint capability with send rights, validates $len \leq \mathrm{MAX_MSG}$, and verifies that `msg_ptr` references valid user memory. Safety invariant: messages are delivered strictly to the intended recipient endpoint; unmapped physical frames are never exposed to peer address spaces; and failed deliveries leave endpoints consistent. Performance measurement: benchmark end-to-end round-trip latency and throughput under varying message sizes while controlling for cache warm-up, context-switch cycles, and CPU frequency pinning.

References & Further Reading

1. RISC-V International, *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*, especially privilege modes and supervisor trap state.
2. MIT PDOS, *xv6: a simple, Unix-like teaching operating system*, Chapters 1–2. The xv6 code and book are distributed under permissive notices retained by upstream.
3. The Open Group, POSIX.1–2024 `open()` and `read()`, for the visible file-descriptor and I/O contracts used in the worked trace.
4. MIT PDOS, official xv6 RISC-V rev5 source, pinned at commit `7d7adbb1b0acbd67c9766a20d0f9900fef2789fa` for the implementation spine.

SECTION DETAIL

Traps, Interrupts, and System Calls

A process calls `write(fd, user_buf, 4096)`. The descriptor is valid, but `user_buf + 4096` wraps around the machine word and the kernel checks only the starting address. A naive kernel then copies across an unintended mapping while executing with supervisor privilege. The trap instruction worked exactly as designed; the security failure occurred because privilege transfer is not argument validation.

A controlled entry must solve three different problems:

1. transfer control without trusting a user-supplied destination;
2. preserve enough interrupted state to resume or terminate correctly; and
3. convert untrusted register values into validated kernel objects.

Prediction — what must already be safe?

At the first supervisor instruction after a user `ecall`, decide whether each item has already been established by hardware: kernel stack, kernel page table, complete general-register save, previous program counter, trap cause, interrupts disabled, trusted handler address.

Separate architectural CSRs from conventions implemented by entry assembly.

Hardware has written the interrupted PC to `sepc`, written the cause to `scause`, recorded the previous privilege and interrupt-enable state in `sstatus`, disabled supervisor interrupts, and set the PC from the protected `stvec` configuration. It has not selected a kernel stack, switched `satp`, or saved all integer registers. xv6's first assembly instructions must preserve user registers while arranging those software-owned transitions. Treating a trap like a high-level function call misses the dangerous interval before a normal C environment exists.

Event taxonomy: source, timing, and restart point

"Trap" is the umbrella term for a control transfer caused by an exception or interrupt. The exact vocabulary varies across architectures; the causal distinction does not.

> **DEFINITION**
>
> Synchronous events are attributable to the current instruction; asynchronous interrupts are selected between architectural instruction effects and need not be caused by that instruction.

Event	Relation to current instruction	RISC-V example	Typical continuation
Requested synchronous exception	Caused deliberately by the current instruction	`ecall` from U-mode	Advance past `ecall` after servicing.
Fault-like synchronous exception	Current instruction cannot complete as requested	load page fault	Repair cause and retry the same instruction, or terminate.
Terminal synchronous exception	Current execution cannot safely continue	illegal instruction with no emulator	Report or terminate; retry is not useful.
Asynchronous interrupt	Raised independently of the current instruction boundary	timer or external-device interrupt	Service event, then resume the interrupted instruction stream.

The classification determines the return PC. xv6 adds four bytes to its saved user PC for an `ecall`, because that instruction is four bytes and `sepc` points to it. A recoverable page fault leaves the PC unchanged so the load or store can retry. An interrupt resumes the interrupted stream without pretending the instruction requested the event.

Cause classification from raw state

For each snapshot, state the event class and correct PC action. Assume RV64 and a direct trap vector.

Snapshot	`scause`	`sepc`	`stval`
A	`8`	`0x1040`	`0`
B	`13`	`0x2088`	`0x4000_1000`
C	high interrupt bit + `5`	`0x30ac`	unspecified

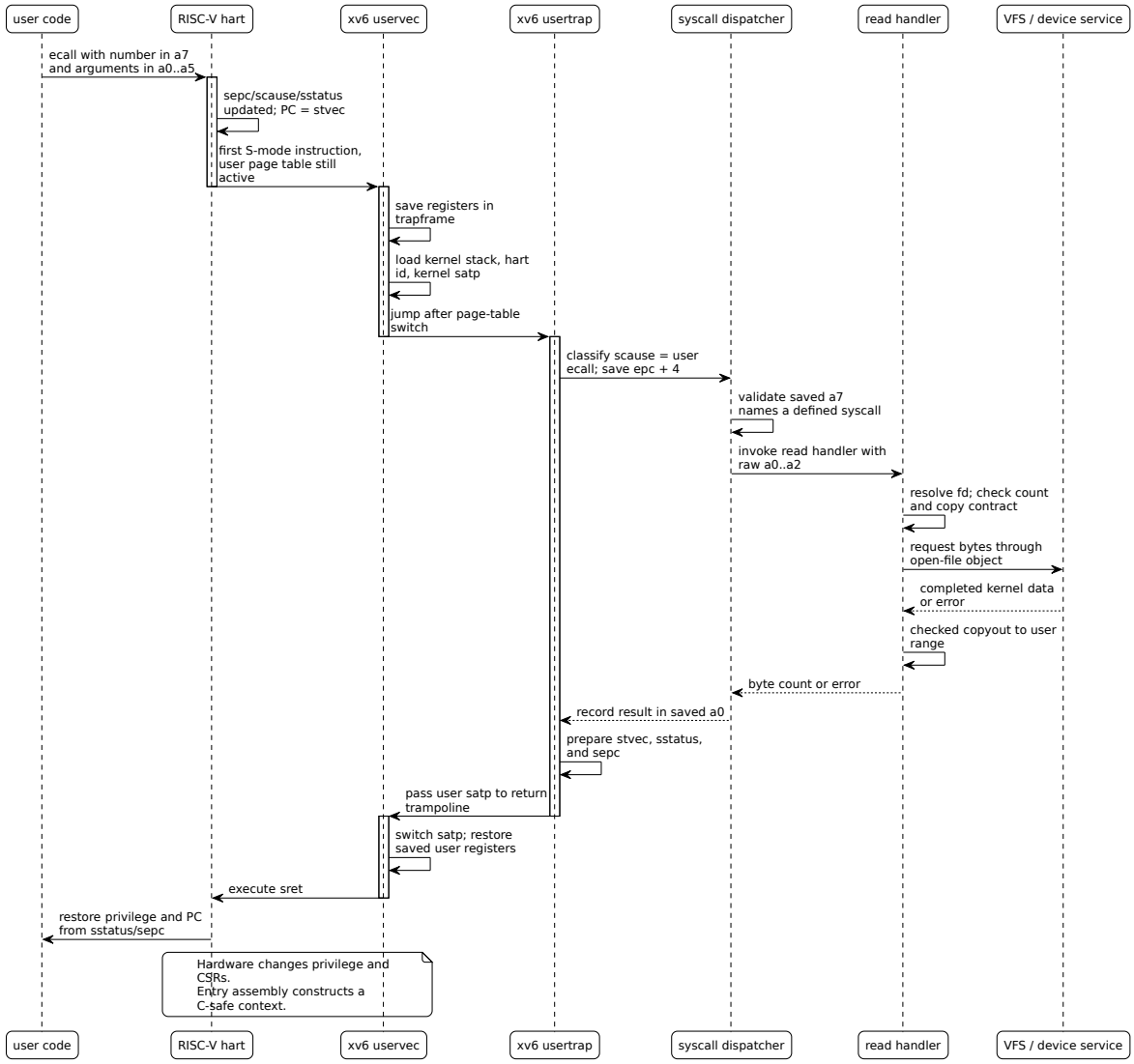
A is an environment call from U-mode: dispatch it and, on normal return, resume at `0x1044`. B is a load page fault: `stval` supplies the faulting virtual address when the implementation reports it; resolve the mapping and retry at `0x2088`, or terminate. C is a supervisor timer interrupt: acknowledge/rearm it as required and resume at `0x30ac`, unless scheduling deliberately changes which saved context runs. Advancing every `sepc` by four would skip the faulting load and corrupt program semantics.

The architectural transition into supervisor mode

For a trap delegated to S-mode, RISC-V supplies a small, protected landing contract:

- `sepc` receives the address of the interrupted or exception-causing instruction;
- `scause` records interrupt-versus-exception and a cause code;
- `stval` may record cause-specific information such as a faulting address;
- `sstatus.SPP` records the prior privilege, `SPIE` receives the old `SIE`, and `SIE` is cleared;
- the PC is loaded from `stvec` according to its direct or vectored mode.

The architecture intentionally does **not** prescribe a process structure, syscall ABI, kernel stack, page-table switch, or complete context frame. Software chooses those mechanisms.



The two diagrams are intentionally not equivalent: the rendered sequence above names the ownership transfers and saved state. Any generic arrow chain that omits the user page table at first entry conceals xv6's hardest constraint.

Architectural boundary check

Why can xv6 not set `stvec` directly to an ordinary C function compiled into the kernel text?

On a trap from user space, RISC-V has not switched `satp`; the user page table remains active. The first instruction address must therefore be mapped there, and C additionally assumes a valid stack and calling-convention state that do not yet exist. xv6 maps trampoline code at the same virtual address in user and kernel page tables, saves registers to a per-process trapframe, installs the kernel stack and page table, then enters C. “C cannot run in supervisor mode” is wrong; C cannot run safely until its software conventions are established.

Scrub the complete request path

Before moving the control, predict the first stage at which `fd`, byte count, and the user address can all be interpreted in the caller’s process context. Then find the first stage that may block.

SCRUB THROUGH A READ SYSTEM CALL

USER #1
Application

USER #2
Library wrapper

HARDWARE #3
RISC-V trap hardware

KERNEL #4
xv6 trampoline entry

KERNEL #5
Dispatcher and read handler

KERNEL #6
VFS and device path

KERNEL #7
Kernel completion

HARDWARE #8
Trampoline and sret

At kernel dispatch, the call number, descriptor, byte count, and user address are validated before the kernel acts on the request.

The model fixes one `read` path and suppresses cache hits, short reads, signals, partial device completion, and concurrent descriptor mutation. Its point is control ownership. In particular, the library wrapper is not the privilege boundary, and trap entry is not proof that the request is legal.

The eight-stage model deliberately combines two source-level cuts in "Dispatcher and read handler." Generic syscall dispatch validates only that saved `a7` selects a defined handler. The `read` handler and lower file/copy helpers collectively decode and check the descriptor, byte count, and destination buffer in the caller's context. Attributing those object/range checks to the dispatcher alone hides where their authority and address-space information becomes available.

The return occupies two stages. "Kernel completion" copies completed bytes to the checked destination, records the result in saved `a0`, and has C prepare `sepc` / `sstatus` for the chosen continuation. "Trampoline + sret" first uses assembly to switch mappings and restore the saved register file; the `sret` instruction then applies the prepared privilege and PC state. Hardware does not copy file bytes, choose the result, or restore registers from xv6's trapframe.

Reconstruct the hardware/trampoline handoff

At the component's "RISC-V trap hardware" and "xv6 trampoline entry" stages, draw a table with rows for `a0`, `a7`, user SP, user PC, kernel SP, `satp`, `stvec`, and privilege. Fill columns "before `ecall`," "after RISC-V trap hardware," and "after xv6 trampoline entry." Hardware changes privilege and protected trap CSRs; only trampoline software selects the kernel stack and `satp`.

Hardware preserves some values by not changing them; that is different from copying them into durable storage.

State	Before `ecall`	After RISC-V trap hardware	After xv6 trampoline entry
`a0`	first argument	still first argument, not yet durably saved	saved in trapframe; register reused/restored later
`a7`	syscall number	unchanged	saved in trapframe
user SP	in `sp`	still in `sp`	saved; `sp` is kernel-stack top
user PC	current `ecall`	copied to `sepc`; PC is trap vector	copied into process trapframe by C path
kernel SP	stored in trapframe metadata	not selected	loaded into `sp`
`satp`	user page table	user page table	kernel page table
`stvec`	trampoline address	source of new PC	changed to kernel vector before kernel traps are enabled
privilege	U	S	S

The likely reasoning gap is confusing "register not overwritten yet" with "safe across arbitrary handler code." The assembly must save it before reuse.

Dispatch and validation are separate proof steps

An ABI converts source arguments into raw machine words. In rev5, the syscall number is in saved `a7`, six raw arguments are available from saved `a0`–`a5`, and the result is written to saved `a0`. `syscall()` verifies only that the number indexes a defined handler. For `read`, `sys_read` and lower file/copy routines collectively interpret the descriptor, count, and user address under type-, authority-, and process-specific contracts; pointer legality is established by checked copying, not by `argaddr()` alone. The interactive's combined stage reflects that collective proof without pretending all checks live in `syscall.c`.

For a range `[u, u+n)`, a check such as `u < process_size` is insufficient. At minimum, reasoning must reject arithmetic overflow and every inaccessible page touched by the copy. A robust pattern either checks `n <= limit - u` after establishing `u <= limit`, or walks/copies in bounded chunks with explicit failure at each mapping.

```
/* Schematic boundary logic, not an xv6 patch. */
int copy_from_user_checked(void *dst, uint64 u, uint64 n, uint64 limit)
{
    if (u > limit || n > limit - u)
        return -1;
    return copy_mapped_chunks(dst, u, n);
}
```

The subtraction form avoids constructing `u+n` before proving that addition is representable. It still delegates page permissions and mapping continuity to `copy_mapped_chunks`.

Application — kill four validators

For `write(fd, u, n)`, give one input that defeats each implementation.

1. checks only `u < size`;
2. checks `u+n <= size` using wrapping unsigned addition;
3. validates the user range, then passes `fd` as a direct pointer to a kernel file object;
4. validates everything, but holds a spinlock while copying a page that may fault or block.

1. Choose `u=size-1`, `n=2`; the start is legal and the range is not.
2. Choose `u` near `UINT64_MAX` and `n` large enough that `u+n` wraps to a small value.
3. Supply an integer whose bit pattern names arbitrary kernel memory; descriptors must be resolved through the caller's descriptor table, not cast.
4. Arrange a missing or reclaim-contended user page. If copying can sleep, sleeping while holding a spinlock can deadlock or violate lock rules. If this particular kernel's copy path never faults, the design must state that invariant rather than rely on wishful behavior.

The four defects concern range, arithmetic, authority, and execution context. One "validate inputs" slogan does not diagnose them.

SYSTEM CALL ADDRESS RANGE VALIDATION MATRIX

Evaluate each statement regarding how kernel entry code validates a user address range [u, u+n) before copying:

Answer: Checking u <= limit followed by n <= limit - u prevents arithmetic wraparound without forming u + n.; Validating user memory requires verifying page-table presence and write permissions across every page.

Directly evaluating u + n can overflow to a small value and bypass bounds checks. Subtraction bounds (n <= limit - u) prevent wraparound, and page permissions must be validated across the entire span.

Worked trace: a recoverable load page fault

Assume a user load at PC `0x2088` reads virtual address `0x40001000`. The address lies in a lazily allocated region but has no present leaf mapping.

EXAMPLE

Recovery keeps `sepc` at the faulting load because the repaired mapping makes retry—not instruction skipping—the semantics-preserving action.

1. The load raises a synchronous exception. `sepc=0x2088`; `scause` identifies a load page fault; `stval` reports the attempted address on the modeled platform.
2. Entry assembly preserves the process state exactly as for a syscall. The common mechanism does not imply a common policy.
3. The handler verifies that the address belongs to a region eligible for lazy allocation and that the requested access is readable.
4. It allocates and initializes a frame, installs a user-readable mapping, and performs required translation-cache synchronization.
5. Return leaves the saved PC at `0x2088`. The load retries and may now complete.

If allocation fails, the kernel must not loop forever retrying an impossible instruction. It reports or records a fatal process condition and returns control through a termination path.

Transfer — same cause number, different trust

A load page fault occurs while supervisor code dereferences a kernel pointer. Why is "allocate a page and retry" generally unsafe even though that response was correct for the user fault?

The faulting protection domain and address-space contract differ. An unexpected kernel fault can indicate a dangling pointer, corrupted page table, stack overflow, or violation of a no-fault region; allocating could hide corruption and continue with supervisor authority. A kernel may deliberately support recoverable faults during bounded user-copy routines, but that requires an explicit exception table or recovery protocol. Cause code alone does not determine policy; origin, expected region, and current lock/interrupt context matter.

Worked trace: a timer interrupt through a context switch

Assume process A runs in user mode when the timer fires.

EXAMPLE

The timer path contains two suspended continuations: the user trapframe and the kernel handler's scheduler context.

Moment	Saved continuation	Active stack	Scheduling consequence
interrupt entry	A's user PC in trap state	transition to A's kernel stack	none yet
kernel handler	A's full trapframe plus kernel frames	A's kernel stack	handler may call `yield()`
scheduler switch	A's kernel continuation saved in scheduler context	per-CPU scheduler stack/context	another runnable process B may run
eventual resume	switch returns inside A's suspended handler	A's kernel stack	handler restores trap CSRs
`sret`	A's user continuation restored	user stack	interrupted user execution resumes

There are two nested resumptions: scheduler-to-kernel-handler and kernel-handler-to-user. Saving only the user trapframe cannot resume the suspended C call chain; saving only the kernel context cannot reconstruct all user registers.

Counterexample — enabling interrupts too early

Suppose `usertrap()` enables supervisor interrupts before changing `stvec` away from `uservec` and before installing a valid kernel stack. Construct the failure.

A timer can arrive while supervisor code is running but `stvec` still names the user-entry trampoline. That entry assumes a user trapframe and transition context; it can overwrite saved user state or load stale process metadata while already on a kernel path. The safe design disables interrupts during the ambiguous interval, points kernel-origin traps at a handler that assumes a kernel page table and stack, and enables interrupts only after trap CSRs needed for classification have been consumed or saved.

REMARK

Reusing trap entry does not unify event policy: system calls advance, recoverable faults retry, and interrupts resume unless a scheduler selects another saved context.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 1.2.1: Hardware Trap Transition and Software Invariants

Description application

Identify what architectural state transitions are performed automatically by RISC-V hardware on an `ecall` trap into supervisor mode, and contrast them with the responsibilities that must be performed by software in `trampoline.S`.

Distinguish hardware CSR updates from software stack selection and register saving.

Hardware automatically saves the faulting/call PC to `sepc`, writes the trap cause code to `scause`, captures previous privilege mode and interrupt enables in `sstatus`, disables interrupts in supervisor mode, and sets PC to `stvec`. Hardware does NOT switch the stack pointer (`sp`), does NOT switch the page table (`satp`), and does NOT save general-purpose registers (`x1..x31`). Software entry code (`trampoline.S`) must exchange `sp` with the per-process trapframe, save all integer registers, configure the kernel stack pointer, and switch `satp` to the kernel page table before entering C code.

Exercise 1.2.2: Subtraction-Based Range Validation Against Wraparound

Verification application

A kernel system call receives user address `u` and length `n`. Prove that the condition `u + n <= limit` fails to detect integer wraparound in modular arithmetic, and explain why the ordered check `u <= limit && n <= limit - u` is sound.

Consider boundary values where u + n exceeds the maximum 64-bit integer.

In fixed-width unsigned arithmetic modulo 2^{64}, if `u = 0xFFFFFFFFFFFFFF00` and `n = 0x200`, the sum wraps around to `0x100`. Testing `u + n <= limit` succeeds because $256 \leq \text{limit}$, even though the buffer spans across memory into invalid or kernel space. By contrast, testing `u <= limit` first establishes that `limit - u` is non-negative and mathematically bounded without overflow. The subsequent check `n <= limit - u` ensures that the entire span $[u, u + n)$ fits below `limit` without ever computing the wrapping sum.

Exercise 1.2.3: Synchronous Recoverable Page Fault Execution Trace

Description transfer

Trace the control flow, CSR state, and page-table modifications when a process triggers a recoverable demand-paging load fault at instruction PC `0x2088` accessing virtual address `0x40001000`.

Trace why sepc remains at the faulting load instruction rather than incrementing by 4.

The load instruction triggers a synchronous exception. Hardware writes `sepc = 0x2088`, sets `scause` to load page fault (cause 13), writes `stval = 0x40001000`, and jumps to `stvec`. The kernel trapframe preserves user registers. The VM fault handler verifies that `0x40001000` lies within a valid VMA, allocates a physical frame, maps it with read permissions, and flushes the TLB (`sfence.vma`). Crucially, `sepc` is NOT incremented; on `sret`, execution returns to `0x2088` to re-execute the load instruction, which now hits the valid translation.

Exercise 1.2.4: Stale Trampoline Interrupts and Trap Vector Swapping

Construction transfer

Explain the failure mode if supervisor interrupts are re-enabled inside kernel mode while `stvec` still points to the user-mode trampoline entry vector.

Consider what trapframe stvec dereferences when an interrupt occurs while already on the kernel stack.

If an asynchronous timer or device interrupt arrives while `stvec` points to the user trampoline, the CPU executes the user entry code while already in supervisor mode. The trampoline attempts to read `sscratch` (which now points to the kernel or contains user register data), swaps the stack pointer to an invalid state, and overwrites the active user trapframe with kernel register values. To prevent state corruption, kernel entry must switch `stvec` to `kernelvec` before enabling interrupts, and disable interrupts before switching `stvec` back to `uservec` on exit.

Exercise 1.2.5: Complete Boundary Protocol and Snapshot Semantics for copy_digest

Construction synthesis

Design the complete validation sequence for a system call `copy_digest(in_ptr, length, out_ptr, flags)`. Specify the argument decoding order, memory bounds validation, chunked copy strategy, and error reporting.

Reject invalid flags first, validate non-wrapping bounds, copy into bounded kernel buffers, and protect against concurrent modification.

1. Decode and reject unknown flags immediately with `-EINVAL`.
2. Check range bounds: verify `in_ptr <= USER_MAX && length <= USER_MAX - in_ptr`, and verify `out_ptr <= USER_MAX && 32 <= USER_MAX - out_ptr`.
3. Accept zero length without reading `in_ptr`, but still verify that `out_ptr` is writable.
4. Process input in bounded kernel-allocated chunks (e.g., 4 KiB buffers) via `copyin`, feeding each chunk into a kernel cryptographic digest context. This bounds kernel memory usage and prevents unbounded locks.
5. Once hashing completes, write the 32-byte digest to `out_ptr` via `copyout`. If a fault occurs during `copyout`, return `-EFAULT`.
6. Advance `sepc += 4` before executing `sret` so the user process resumes at the instruction following `ecall`.

References & Further Reading

1. RISC-V International, *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*, Sections 3.3.1 and 12.1.
2. MIT PDOS, *xv6: a simple, Unix-like teaching operating system*, Chapter 4.
3. MIT PDOS, xv6 rev5 `kernel/trampoline.S`, `kernel/trap.c`, `kernel/syscall.c`, `kernel/sysfile.c`, `kernel/file.c`, and `kernel/vm.c`.
4. The Open Group, POSIX.1-2024 `read()`, for short-read, interruption, and user-visible I/O semantics used in the examples.

SECTION DETAIL

Boot and Kernel Organization

Hart 0 builds the kernel page table, writes `started = 1`, and enables paging. Hart 1 observes `started == 1` and immediately loads a stale page-table root because the compiler or CPU made the flag visible before the page-table writes. The source order looks correct; `volatile` would still not create inter-hart publication. Early boot is a concurrent protocol with fewer diagnostics than ordinary code, not a magical phase exempt from memory ordering.

Boot has one governing invariant:

> ***Activation invariant.*** *No execution context may use a facility until the facility's state is constructed, its dependencies are live, and publication makes those writes visible to that context.*

Failure localization before terminology

For the stale-page-table scenario, separate the failure into three questions: Was the table constructed? Was its address published? Was the consuming hart activated only after an acquire operation? State one observation that distinguishes each failure.

A ready flag can have the correct value while the object it announces is not safely visible.

Construction is tested by inspecting the producing hart's table memory and root value before publication. Publication is tested by verifying a release operation occurs after all table writes and that the announced root is the one later loaded. Activation is tested by verifying each secondary hart performs a matching acquire before reading the table or enabling translation. A zero or malformed PTE suggests construction; a wrong root suggests publication of identity; correct producer memory but stale consumer data suggests missing synchronization. A debugger that halts all harts may accidentally add ordering and hide the defect.

Reset, handoff, entry, and activation

These stages must be named by contract rather than by a universal product sequence.

> **DEFINITION**
>
> A boot stage is an ownership transfer with explicit postconditions; a product name such as firmware or bootloader does not supply those postconditions by itself.

Stage	Producer's obligation	Kernel may assume	Kernel must not assume
Reset/firmware	Establish architecture-defined reset state and platform firmware behavior	Only the documented platform contract	A C stack, a page table, or discovered RAM unless promised
Loader/firmware handoff	Place the image and provide entry parameters	Entry PC, image layout, register/CSR state explicitly specified	UEFI, BIOS, or device tree on every platform
Assembly entry	Establish stack and minimal execution environment	Linker-defined symbols and handoff registers	Allocator, interrupts, or global constructors are ready
Privilege setup	Configure delegation, protection, timer access, and return target	CSRs written by the boot code	Supervisor mode is active before `mret`
Global construction	Initialize shared allocators, tables, drivers, and the first process	Single bootstrap owner where the protocol guarantees it	Secondary harts see writes before publication
Per-hart activation	Install hart-local translation, trap vector, and interrupt routing	Published global state	Another hart configured this hart's local CSRs

"Bootloader" is therefore a role, not one mandatory binary. In the xv6 rev5 QEMU configuration, QEMU loads the kernel at physical `0x80000000` with `-bios none` and starts each hart at `_entry`. A production RISC-V Linux kernel expects a different documented handoff, including hart ID and device-tree address registers and `satp=0`.

Retrieval — identify the contract, not the brand

A platform jumps to the kernel with translation disabled, `a0=hartid`, `a1=device_tree_pa`, and interrupts masked. Which facts belong in a portable kernel-entry contract, and which would be unjustified additions?

The four stated facts belong in the contract: entry PC by implication, translation disabled, register arguments, and interrupt state. It is unjustified to infer UEFI remains callable, RAM begins at a particular address, all harts enter simultaneously, `.bss` is zeroed, or a stack exists. Those may be supplied by a specific image format or loader, but each needs separate evidence. The reasoning gap is replacing an explicit handoff with a familiar PC boot story.

Worked trace: one xv6 hart reaches C

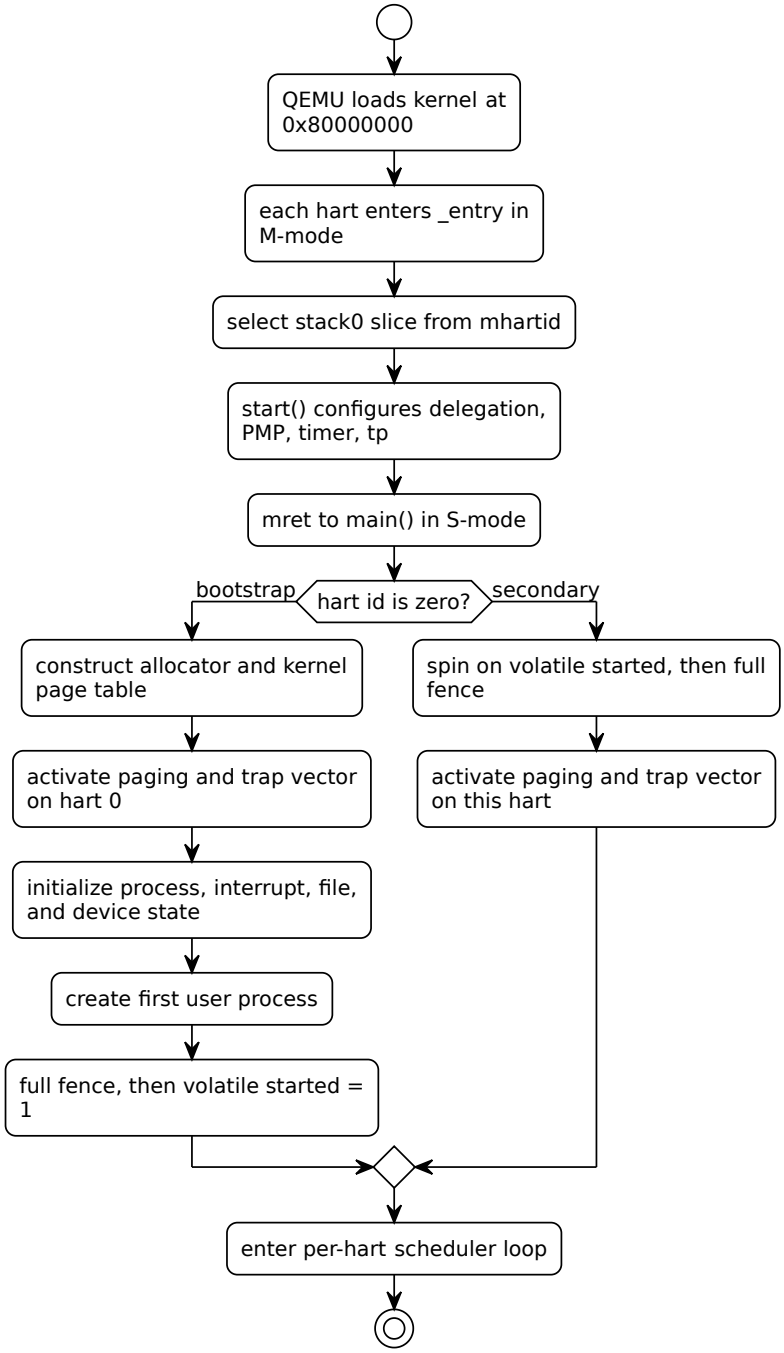
Xv6 rev5 fixes enough details for an exact trace.

Cut	Mode	PC / operation	Stack and translation	Established fact
0	M	QEMU jumps to `_entry` at `0x80000000`	no xv6 stack; paging off	image placement matches `kernel.ld`
1	M	`_entry` reads `mhartid`	selects one 4096-byte slice of `stack0`	C calls cannot collide across harts
2	M	`call start`	valid aligned stack; paging off	C machine-mode setup can run
3	M	`start` writes `mstatus.MPP=S`, `mepc=main`, `satp=0`	same stack	`mret` target and next privilege fixed
4	M	delegates traps, configures PMP, timer access, and `tp=hartid`	same stack	supervisor kernel receives intended events/resources
5	S	`mret` transfers to `main`	paging still off	C supervisor initialization begins

The stack formula is

$$sp = \mathrm{addr}(stack0) + (hartid + 1) \cdot 4096.$$

The `+1` makes `sp` point just above the hart's downward-growing stack slice. If hart 0 used `stack0 + 0`, its first push would write before the allocation.



The diagram encodes a partial order, not a duration chart. Global construction occurs once; CSR activation occurs per hart. Rev5 implements publication with `__sync_synchronize()` before the bootstrap hart's volatile flag store and another `__sync_synchronize()` after a secondary observes the flag. Those full compiler/hardware fences play the ordering role; the volatile access alone would not publish the preceding object initialization. This is the concrete rev5 protocol, not a claim that the source uses C11 release/acquire atomics.

Application — corrupt the entry stack

Assume `NCPU=8` but `stack0` reserves only four 4096-byte slices. Explain two distinct failure signatures: hart 5 enters C, and hart 5 never enters C because only four harts are started.

If hart 5 enters, its computed SP points beyond `stack0`; a call or spill corrupts adjacent static storage, often producing delayed, symbol-dependent failures. If the platform starts only harts 0–3, the undersized array is latent: no current access is out of bounds, but the binary violates its declared `NCPU` configuration. A correct audit checks both allocated capacity and the actual set of entering hart IDs; "it boots on my four-core QEMU command" does not prove the formula safe.

MULTICORE BOOT PUBLICATION MATRIX

Hart 0 builds the kernel page table and sets started = 1. Evaluate each statement regarding multicore memory publication without memory fences:

Answer: Without a release fence on hart 0, preceding page-table stores may not be visible when hart 1 reads started.; Hart 1 requires an acquire fence or data dependency to order its read of started before reading page tables.

volatile restricts compiler optimizations on that single variable but creates no inter-hart memory ordering. Publication requires a release fence on hart 0 and an acquire fence on hart 1.

From machine mode to a supervisor kernel

`mret` uses `mstatus.MPP` to choose the next privilege and `mepc` as the next PC. xv6 also arranges that supervisor mode can access RAM through PMP, receives delegated exceptions and interrupts, can read the time counter, and receives timer events. These are independent permissions. Setting the return privilege does not automatically delegate traps or grant physical-memory access.

> **WARNING**
>
> Successful control transfer into S-mode proves neither RAM authority nor interrupt delegation; each CSR establishes a separate part of the execution contract.

The trap delegation values in a teaching kernel are a platform choice, not a portable recipe. A firmware environment may retain machine-mode responsibilities. A secure monitor may deliberately withhold regions. A robust explanation lists the authority each CSR grants rather than presenting a block of writes as ceremony.

Transfer — one missing machine-mode permission

Supervisor code reaches `main` but faults on its first RAM access. `satp=0`, the physical address is correct, and `mstatus.MPP` was set to S. Which machine-mode setup should be investigated first, and why does successful `mret` not exonerate it?

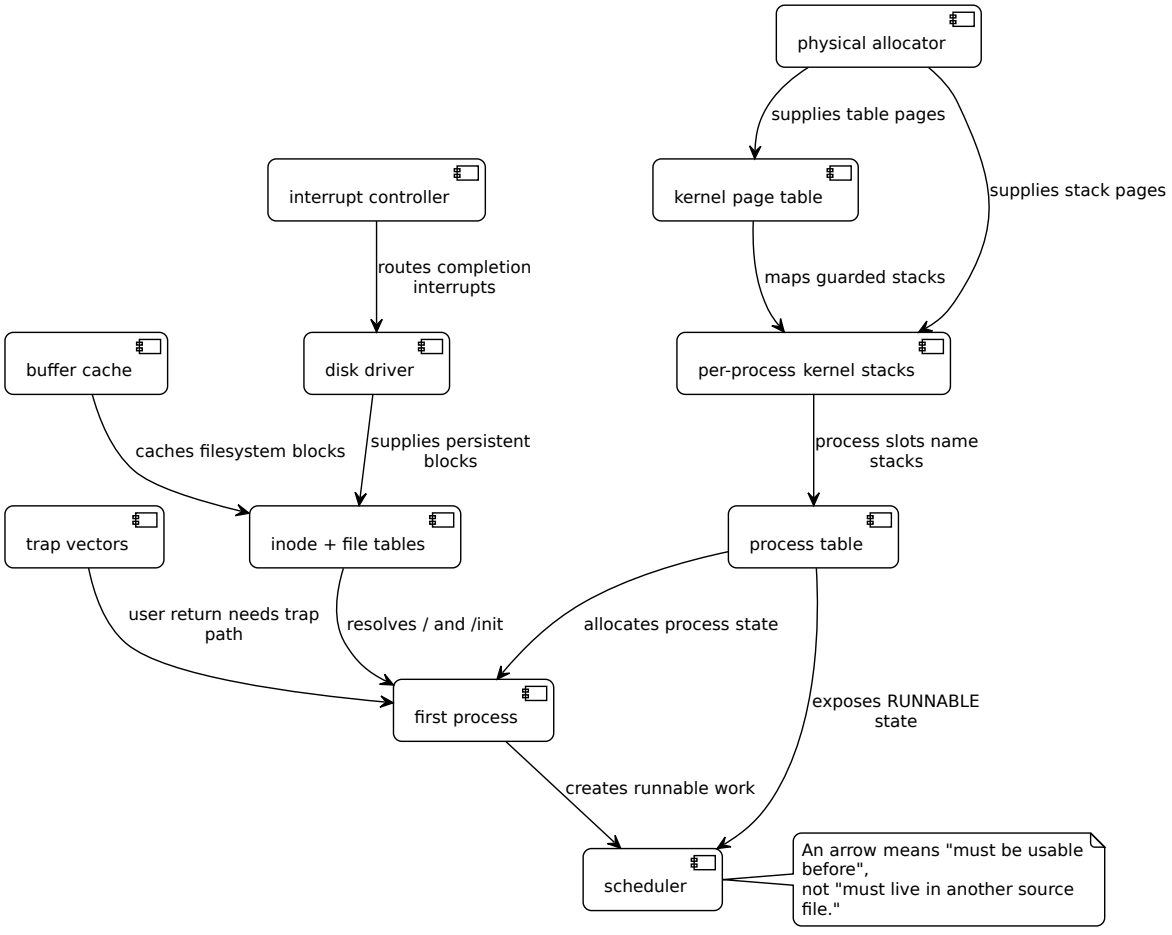
Investigate PMP configuration. The privilege return and physical-memory authorization are separate controls: `mret` can enter S-mode while PMP denies S-mode access to the RAM region. Check `pmpaddr0` / `pmpcfg0` and firmware-reserved ranges before blaming the page table, which is disabled. The failure reveals a missing authority grant, not an incorrect control-transfer target.

Global construction is a dependency graph

On the bootstrap hart, xv6 rev5 performs this high-level order:

```
console → physical allocator → kernel page table → enable paging
        → process table → trap state → interrupt controller
        → buffer cache → inode/file tables → disk → first process
        → publish started
```

The real graph branches. Console output can precede the allocator. Kernel stacks require physical pages and mappings. Per-hart trap-vector installation requires a valid vector address on that hart. The first user process needs process-table and filesystem-facing state. The scheduler needs a process table and a valid per-hart context, but it need not wait for every optional driver in a different design.



The graph is more useful than memorizing calls because it supports counterfactual design. A RAM-only first process could remove the disk/filesystem dependency. A demand-loaded driver could move activation later. Every moved node needs a revised error and concurrency contract.

Dependency exercise — parallelize without guessing

A team wants to initialize the buffer cache, inode table, and interrupt controller on secondary harts before `started` is published. Give a safe plan or reject the proposal. State the ownership and publication needed for each parallel task.

A safe plan needs more than graph independence. The allocator and kernel mappings must already be published to workers. Each initialization task must have exclusive ownership of its state or internal synchronization; the bootstrap hart needs a join/acquire before any consumer runs. Buffer-cache construction and inode-table construction may contend for allocator state and may have an ordering edge if inode initialization assumes cache locks exist. Interrupt-controller setup often has both one-time global and per-hart parts, so assigning it wholesale to one worker is incorrect. Reject the proposal unless these contracts are exposed. Parallelizing a short teaching-kernel boot is unlikely to justify the proof burden, but the decision should follow dependencies and measurements, not instinct.

Worked failure trace: publishing before the first process

Suppose rev5's `__sync_synchronize(); started = 1` publication sequence is moved immediately after the page table is initialized, before `procinit()` and `userinit()`.

EXAMPLE

Correct fence/flag publication can announce the wrong readiness predicate perfectly; visibility does not strengthen what the predicate means.

1. Hart 1 observes nonzero `started`, crosses rev5's consumer-side full fence, safely observes the page table, installs translation and its trap vector, then calls `scheduler()`.
2. The scheduler scans `proc[]` while hart 0 is still initializing process locks and states.
3. Hart 1 may acquire an uninitialized lock, observe zero-filled state that happens to resemble an enum member, or race with `userinit()` publishing the first runnable process.
4. The fence/flag/fence ordering has worked; the published predicate did not imply that process state was initialized.

The repair is not "use stronger memory ordering." Publish a condition whose truth implies every stated dependency of the consumer, or use separate phase barriers. Synchronization orders writes; it does not invent the right invariant.

Counterexample — a successful boot that is still wrong

Hart 1's scheduler sees no runnable process, executes `wfi`, and later wakes on a timer after hart 0 has created the first process. The system appears to work. Why must the early publication still be rejected?

The path relies on accidental zero-state interpretations, interrupt timing, and unsynchronized concurrent initialization. A different compiler, hart count, timer phase, or lock representation can expose corruption. Eventual wakeup does not prove process-lock construction or state publication was race-free. The correct criterion is the activation invariant across all allowed executions, not one observed boot transcript.

Kernel organization: call boundary, protection boundary, fault boundary

xv6 is monolithic: process management, virtual memory, filesystems, and drivers execute in one supervisor address space. Source files and C interfaces improve reasoning, but a wild write in one subsystem can mutate another subsystem's state.

DEFINITION

A module boundary organizes code; a protection boundary mediates authority; a fault boundary limits propagation. One design edge may provide any subset of the three.

Compare three kinds of boundary:

Boundary	Enforcement mechanism	Failure contained?	Typical cost
C module/API	language and build conventions	not against memory corruption in privileged C	ordinary call
in-kernel compartment	memory protection, language isolation, or control-flow enforcement	depends on mechanism and shared authority	checked call / mapping change
user-space service	separate address space plus IPC authority	direct memory faults usually contained; protocol faults may propagate	trap, scheduling, copying or mapping

Moving a driver to user space can reduce direct kernel-memory authority, but it adds IPC, lifecycle, and recovery paths. A monolithic call can be fast and simple, but every privileged parser joins the TCB. "Microkernel is secure" and "monolithic is fast" are incomplete claims until the workload, hardware, and authority graph are specified.

Synthesis — isolate a fallible parser

A new kernel feature parses an untrusted, variable-length device manifest during boot. Design two placements: privileged in-process and isolated service. For each, state the earliest boot phase, required authority, failure recovery, and evidence needed to choose.

The privileged design can run after a bounded input buffer, allocator, and diagnostics exist. Give it read-only access to the manifest and a narrow builder API rather than raw global pointers; parse into temporary state, validate references and lengths, then commit. Failure can reject the device or stop boot, but memory corruption remains in the TCB. The isolated design requires address spaces, IPC, scheduling, and a way to hand over a bounded buffer, so it must run later or use a pre-kernel compartment mechanism. It can be restarted on a fault, but the kernel must validate the returned object graph and handle service absence. Measure boot latency and memory cost, fuzz both protocol boundaries, and audit authority. Isolation is valuable only if the trusted consumer does not blindly accept malicious parsed output.

REMARK

Omitted boot facilities (such as ACPI, device trees, or secure boot) may not be assumed in the xv6 traces unless added to the handoff contract explicitly.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 1.3.1: Per-Hart Entry Stack Allocation and Bounds Corruption

Description application

In RISC-V xv6 `entry.S`, initial stack pointers are computed as `sp = stack0 + (hartid + 1) * 4096`. Explain the failure modes that occur if `NCPU` is configured to 8 but `stack0` in static data allocates only four 4096-byte slices.

Trace the memory layout when hart 5 writes to its computed stack pointer.

Each hart computes an offset based on its hardware ID. If only 4 slices (16 KiB) are allocated in the BSS/data segment, harts 0 through 3 occupy valid bounds. When hart 4 or 5 boots, its computed `sp` points beyond the `stack0` buffer into adjacent static kernel variables (such as kernel locks or page tables). Any subsequent function call or register spill corrupts those symbols, causing delayed, non-deterministic system panics. If the hardware platform only powers on harts 0 through 3, the bug remains latent until tested on an 8-core CPU.

Exercise 1.3.2: Memory Ordering Fences in Secondary Hart Bootstrapping

Verification application

Hart 0 constructs the kernel page table and initializes subsystems before setting a volatile flag `started = 1`. Secondary harts spin on `while (started == 0)`. Explain why removing memory fences (`__sync_synchronize()`) causes secondary harts to crash even though `started` is declared `volatile`.

Contrast compiler volatile semantics with hardware store-buffer visibility across CPU cores.

In C, `volatile` prevents compiler register caching and dead-code elimination on the variable itself, but it does NOT emit CPU memory barrier instructions. Modern out-of-order or weakly-ordered architectures (like RISC-V) buffer stores in local write buffers. Without a release fence (`__sync_synchronize()`) on hart 0, the write `started = 1` can become globally visible to hart 1 before the stores populating the page tables and kernel data structures drain to cache. Hart 1 then exits its spin loop and loads uninitialized, corrupt page tables into `satp`.

Exercise 1.3.3: Machine-Mode Handoff, PMP, and Exception Delegation

Description transfer

Detail why setting `mstatus.MPP` to supervisor mode and jumping to `mret` is insufficient to establish a functioning supervisor kernel without configuring Physical Memory Protection (PMP) and trap delegation CSRs (`medeleg` and `mideleg`).

Recall that machine mode owns all physical memory by default and traps trap to machine mode unless delegated.

By default on RISC-V, all traps (exceptions and interrupts) trap to machine mode (M-mode), and supervisor mode (S-mode) has zero physical memory access rights. If M-mode executes `mret` into S-mode without programming `pmpcfg0` and `pmpaddr0` to grant supervisor read/write/execute permissions to physical RAM, the first instruction fetch in S-mode immediately triggers an access fault. Furthermore, without setting bits in `medeleg` (system calls, page faults) and `mideleg` (timer and external interrupts), any subsequent user trap or timer interrupt traps back into M-mode rather than reaching the kernel's supervisor trap handler.

Exercise 1.3.4: Subsystem Construction Graph and Premature Interrupt Hazards

Formalisation transfer

Construct the strict partial order of initialization in `main()` for the following kernel subsystems: console, physical page allocator, kernel page table, process table, trap vector, and PLIC interrupt controller. Identify the failure that occurs if interrupts are enabled before the process table is initialized.

Map the prerequisite resources required by each subsystem constructor.

Strict initialization order:

1. Console and print locks (enables diagnostic output).
2. Physical page allocator `kinit()` (provides free physical frames).
3. Kernel page table `kvminit()` and activation `kvminithart()` (enables MMU translation).
4. Process table `procinit()` (allocates proc locks and kernel stacks).
5. Trap vector `trapinithart()` (installs `kernelvec` into `stvec`).
6. PLIC / timer interrupt controller `plicinit()` and `plicinithart()`. If device interrupts are enabled before `procinit()`, an incoming device interrupt invokes `trap()`, which attempts to acquire proc locks, inspect the current CPU's process pointer, or wake waiting processes. Dereferencing uninitialized process table locks or pointers causes an immediate kernel panic or deadlock.

Exercise 1.3.5: Monolithic vs Compartmentalized Boot-Time Device Manifest Parsers

Construction synthesis

Compare placing a complex device-tree or ACPI hardware manifest parser directly inside the early monolithic boot sequence versus executing it inside an isolated user-space or pre-kernel compartment.

Analyze the trade-offs between trusted computing base (TCB) size, crash recovery, and early boot resource availability.

Monolithic placement: The parser runs in early supervisor mode with direct access to physical memory. Benefit: minimal resource requirements (runs before scheduler or address spaces exist) and zero IPC overhead. Cost: complex C string and tree parsing runs with full kernel privilege; a memory corruption vulnerability compromises the entire kernel. Isolated compartment placement: The parser runs in a restricted address space with only read access to the raw manifest buffer and communicates parsed records via IPC. Benefit: parser crashes or exploits cannot corrupt kernel memory; faults can be logged and contained. Cost: requires deferring device parsing until memory management, scheduling, and IPC are functional, complicating bootstrap of boot-critical storage or interrupt controllers.

References & Further Reading

1. MIT PDOS, xv6 rev5 `kernel/entry.S`, `kernel/start.c`, `kernel/main.c`, `kernel/kernel.ld`, and `Makefile`.
2. MIT PDOS, *xv6: a simple, Unix-like teaching operating system*, Chapters 1–2.
3. RISC-V International, *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*, machine/supervisor privilege, delegation, PMP, and return-from-trap sections.
4. Linux kernel documentation, RISC-V Kernel Boot Requirements and Constraints, for a production handoff comparison rather than an xv6 requirement.

Processes and CPU time

SECTION DETAIL

Processes, Threads, and Context Switches

A kernel saves every user register before preemption, chooses another process, and restores that process's user registers. It still crashes. The missing state is the suspended kernel call chain: stack pointer, return address, and preserved registers needed to resume inside the first process's trap handler. A user trapframe answers "how does this thread return to user code?" A scheduler context answers "how does this kernel thread resume its suspended C execution?" Neither subsumes the other.

Prediction — two continuations

Process A enters the kernel on a timer interrupt and calls the scheduler, which runs B. Name the exact continuation that must be stored for each arrow:

```
A user → A kernel → per-CPU scheduler → B kernel → B user
```

A privilege return and a C return consume different saved program counters.

`A user → A kernel` stores A's user PC in trap state/trapframe and all user-visible registers in A's trapframe. `A kernel → scheduler` stores the return address into the suspended kernel call chain plus A's kernel SP and callee-saved registers in A's scheduler context. `scheduler → B kernel` restores B's kernel SP/return path, resuming where B last yielded or at its first-run trampoline. `B kernel → B user` restores B's trapframe and uses `sret`. If the answer names one generic PC, it misses two independently nested stacks and return mechanisms.

Ownership model: process is not thread

A useful decomposition is:

DEFINITION

A process groups resources and authority; a thread is one schedulable flow of control through those resources.

Entity	Owns or denotes	Shared by threads in one process?	Required to resume one thread?
Process	resource container: address space, descriptor table, credentials, limits, process relationships	yes	identity only
Thread	one flow of control and scheduling state	no	yes
Address space	virtual-to-physical mappings and permissions	normally yes	select or confirm translation context
Protection domain	authority boundary enforced by hardware/software	normally yes	identify access rights
User stack	user call chain and automatic storage for one thread	no	yes, via saved user SP
Kernel stack	privileged call chain while servicing one thread	no	yes, via saved kernel SP
Trapframe	full user architectural state at a controlled entry	no	yes for kernel-to-user return
Scheduler context	kernel callee-saved continuation across `swtch`	no	yes for scheduler-to-kernel return

POSIX defines threads in one process as separate flows of control with per-thread scheduling attributes while memory reachable by address is accessible to all live threads in that process. Thus threads provide concurrency, not mutual protection. A process may contain one thread; the concepts remain distinct because resource and execution lifetimes differ.

Retrieval — classify state after a thread switch

Threads T_1 and T_2 belong to one process. The kernel switches from T_1 to T_2 on the same CPU. Classify each item as necessarily changed, possibly changed, or normally shared: page-table root, user SP, kernel SP, descriptor table, signal mask, CPU register file, current-working directory.

User SP, kernel SP, and the active CPU register file necessarily change because they encode each thread's continuation. The page-table root normally remains shared, though an implementation may reload it or use thread-specific mappings. Descriptor table and current-working directory are normally process resources. A POSIX signal mask is per-thread, so it may change. The diagnostic mistake is using "context switch" as shorthand for "replace every process resource."

CONTEXT SWITCH SAVE SETS CLASSIFICATION MATRIX

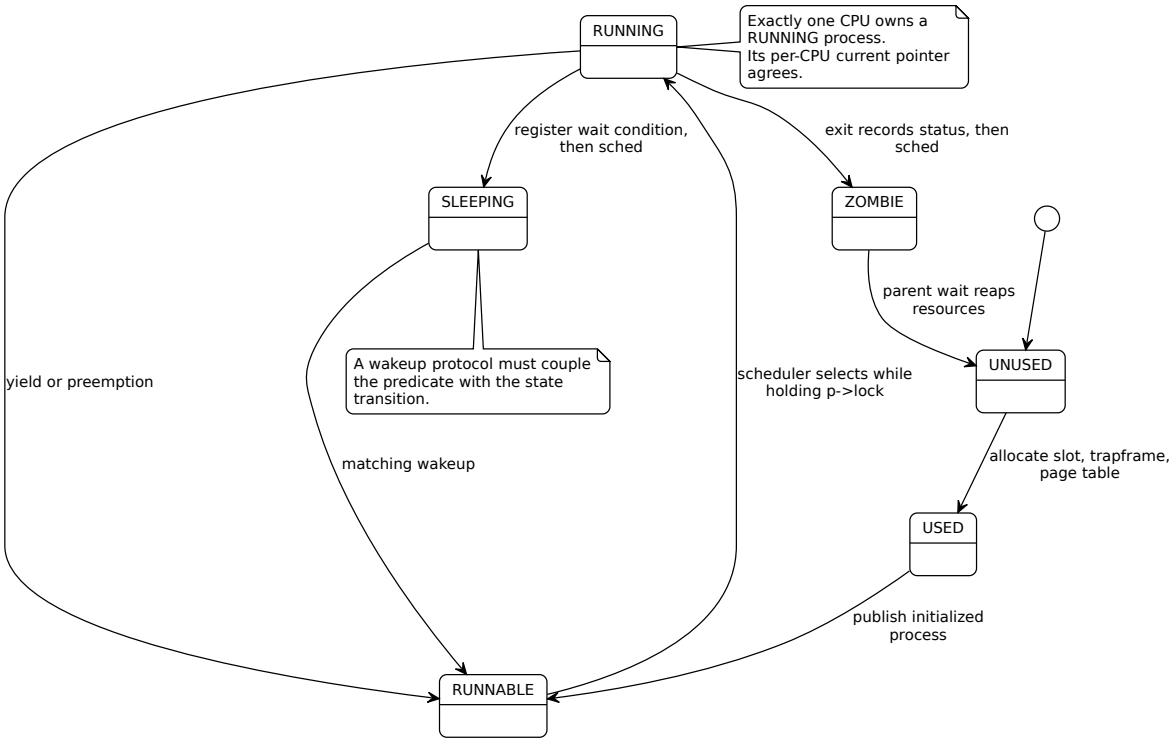
Evaluate each statement regarding the differences between the user trapframe and the kernel scheduler context in RISC-V:

Answer: The user trapframe preserves all 31 general registers because asynchronous traps can interrupt any live register.; The kernel swtch() function preserves only callee-saved registers because it conforms to the standard C calling ABI.

User traps can interrupt arbitrary live registers requiring complete trapframe saves. In contrast, swtch() is invoked as a C function call where the compiler already preserved caller-saved registers on the kernel stack.

Process state is a global invariant

Xv6 rev5 uses `UNUSED`, `USED`, `SLEEPING`, `RUNNABLE`, `RUNNING`, and `ZOMBIE`. These names encode both lifetime and scheduling eligibility.



Three invariants constrain every transition:

1. **Exclusive execution:** a process is `RUNNING` on at most one CPU, and that CPU's current-process pointer names it.
2. **Continuation ownership:** a non-running process has a saved kernel context sufficient for its next scheduler entry; a running process's live registers/stack are owned by its CPU.
3. **Queue/state agreement:** every selection data structure agrees with the process state; a sleeping or zombie process is not selectable.

xv6 protects process state and context with `p->lock`. The scheduler acquires that lock before observing `RUNNABLE`, changes the state to `RUNNING`, installs `c->proc`, and switches while the lock remains part of the handoff protocol. This unusual lock transfer prevents another CPU from selecting

the same process during the interval when its execution ownership is moving.

Application — the double-run interleaving

Two CPUs scan the process table. Rewrite the following broken selection as a minimal interleaving that runs process P twice:

```
if (p->state == RUNNABLE) {
    p->state = RUNNING;
    switch_to(p);
}
```

Then state the linearization point in a repaired design.

CPU 0 reads `RUNNABLE`; CPU 1 reads `RUNNABLE`; CPU 0 writes `RUNNING`; CPU 1 writes `RUNNING`; both install and restore P's context. Concurrent stacks/registers then corrupt one logical continuation. In the repaired xv6-style design, acquisition of `p->lock` serializes the check-and-transition; the successful write `RUNNABLE → RUNNING` while holding the lock is the selection linearization point. An atomic state compare-and-swap could serialize that field, but it would still need a protocol for context and CPU-pointer ownership.

Why there are two save sets

At user trap entry, arbitrary user code can have live values in every allocatable register. The kernel did not call that code under a C ABI that permits clobbering caller-saved registers. Entry assembly therefore saves the complete user register state needed for transparent resumption.

At `swtch(old, new)`, the compiler emitted an ordinary function call. Under the RISC-V calling convention, `s0` – `s11` and `sp` must be preserved across calls; the return address determines where the suspended call resumes. Caller-saved temporaries and argument registers need not be saved by `swtch`: if the C caller needs one after the call, the compiler has already preserved it according to the ABI, often on the kernel stack. Xv6 rev5 stores:

```
context = { ra, sp, s0, s1, ..., s11 }
```

`swtch` has no privileged instruction. It stores the old set through `a0`, loads the new set through `a1`, then executes `ret`. Because `ra` came from the new context, `ret` continues the other kernel thread.

Register audit — kill two plausible bugs

For each mutation, predict the earliest reliable failure mechanism rather than a single output string.

1. omit `s3` from both save and restore;
2. save `ra` correctly but restore `sp` from the old context rather than the new.

1. A suspended C function with a live value allocated to `s3` resumes with the next thread's value. The symptom depends on compiler allocation; force it with assembly or a function that keeps distinct sentinel values live across repeated yields. The failure is an ABI-preservation violation.
2. Control returns to the new thread's `ra` while using the old thread's kernel stack. Returns, spills, and local accesses combine two call chains, quickly corrupting frames or returning through attacker/unrelated data. A test must give stacks distinct canaries and alternate contexts; merely checking that a switch occurred may pass once.

Worked trace: timer preemption of process A

Assumptions: one hart; A is in user mode; B is `RUNNABLE`; the timer event is recognized; scheduling policy selects B.

EXAMPLE

The trace crosses user state, A's kernel continuation, the per-CPU scheduler, and B's continuation; no single save record owns all four cuts.

Cut	Active execution	Durable user state	Durable kernel continuation	Process state
0	A user registers/stack	none required yet	none; A not in kernel	A `RUNNING`, B `RUNNABLE`
1	A trap-entry assembly	A trapframe being filled	A kernel stack selected	unchanged
2	A timer handler	complete A trapframe	live on A kernel stack	unchanged
3	`yield()`	complete	handler call chain + `swtch` caller live	A: `RUNNING→RUNNABLE` under lock
4	per-hart scheduler	complete	A context now stores SP/RA/s-registers	A runnable; CPU owns scheduler context
5	B kernel continuation	B trapframe	B context restored; B kernel stack active	B: `RUNNABLE→RUNNING`
6	B user continuation	B registers/stack restored	kernel continuation remains latent	B running in user mode

There is no direct `swtch(A, B)` in xv6. A switches to the per-CPU scheduler context; the scheduler later switches to B. This coroutine structure centralizes policy and maintains a scheduler stack not owned by either process.

Trace calculation — count transitions, not names

Starting at cut 0, count (a) privilege transitions, (b) `swtch` calls, and (c) address-space changes needed before B reaches user mode in xv6 rev5. State assumptions.

A context switch and a privilege transition are independent events.

There is one U-to-S transition on A's timer trap and one S-to-U transition on B's eventual `sret`: two privilege transitions. There are two `swtch` executions: A-to-scheduler and scheduler-to-B. xv6's user trap entry switches from A's user page table to the kernel page table, and B's return switches from the kernel page table to B's user page table: two `satp` installations, with required fences in the trampoline. A path that resumes B already inside the kernel still needs only the first page-table switch until B returns. The count depends on the explicit starting/ending cuts and xv6's mapping design.

Worked trace: blocking read and wakeup

Assumptions: A calls a pipe read when the pipe is empty but still has a writer; B later writes data.

> **EXAMPLE**
>
> Blocking preserves a suspended kernel continuation, while wakeup changes eligibility rather than transferring execution immediately.

1. A holds `pi->lock` and observes `nread == nwrite && writeopen` in rev5 `piperead`.
2. A calls the combined `sleep(&pi->nread, &pi->lock)`. Inside `sleep(chan, lk)`, it acquires A's `p->lock` **before** releasing `pi->lock`, then records `chan` and changes `RUNNING→SLEEPING`. A writer cannot complete a matching `wakeup` through the gap: after acquiring `pi->lock`, it must also acquire A's `p->lock` before changing A's state.
3. `sleep` calls `sched()` while holding A's `p->lock`. The user trapframe remains unchanged; the saved kernel context now resumes after that `sched()` call.
4. B acquires `pi->lock`, writes bytes, and calls `wakeup(&pi->nread)`. `wakeup` acquires A's `p->lock`, changes `SLEEPING→RUNNABLE`, and releases it. Wakeup establishes eligibility; it does not run A.
5. A scheduler eventually selects A. `swtch` returns inside `sleep`; A clears `chan`, releases its `p->lock`, reacquires `pi->lock`, returns to `piperead`, and reevaluates the `while` predicate.
6. The read copies available bytes through rev5's checked `copyout`, releases the pipe lock, and the trap-return path resumes A after the system call.

The context switch preserves control; the predicate protocol preserves meaning. A perfect register save cannot repair a lost wakeup.

Transfer — wakeup is not execution

Why is changing `SLEEPING → RUNNING` in `wakeup()` unsafe on a multicore kernel, even if an idle CPU exists?

`RUNNING` asserts ownership by one CPU and requires a restored live context and consistent per-CPU current pointer. `wakeup()` only establishes eligibility; it has not selected a CPU, acquired a scheduler context, or performed a switch. The correct transition is to `RUNNABLE`, after which the scheduler's serialized selection establishes execution ownership. Direct dispatch is possible in another design, but it must perform the full ownership protocol rather than relabel state.

Boundary cases that expose false equivalences

Claim	Counterexample	Correct distinction
"Every system call context-switches."	A fast `getpid` can enter and return to the same thread without scheduling another.	mode switch is not scheduling switch
"Every context switch changes processes."	Two threads in one process share its address space/resources.	thread switch need not replace process state
"Every process switch flushes all translation state."	Tagged TLBs/ASIDs can retain entries; kernel designs differ.	logical address-space switch is not necessarily a full hardware flush
"A blocked thread consumes no resources."	It retains stacks, descriptors, mappings, and wait metadata.	not running is not nonexistent
"A zombie is still schedulable because it has a process slot."	A zombie retains exit status for its parent but has no executable continuation.	lifetime state is not eligibility

WARNING

A mode switch, context switch, process switch, and address-space switch can coincide, but none logically implies all the others.

Counterexample — copying a process structure

A kernel implements `fork` by byte-copying its entire process structure, including the spinlock, kernel stack pointer, and scheduler context. Explain at least three independent failures.

The copied lock may appear held by a nonexistent owner or duplicate internal wait state. The kernel-stack pointer aliases the parent's stack, so parent and child corrupt one call chain. The scheduler context resumes at the parent's suspended kernel PC with parent-specific stack/register assumptions rather than a controlled child-first-return path. Descriptor objects also need reference-counted duplication rather than arbitrary pointer ownership. Correct creation copies selected semantics and constructs fresh synchronization/execution state.

REMARK

Any architectural extension that introduces vector registers, floating-point units, or inter-core thread migration must explicitly expand the kernel continuation ownership table before altering the assembly switch sequence.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 2.1.1: Classifying Thread-Private Continuations vs Process-Shared Resources

Description application

In a multi-threaded Unix environment, classify which architectural and operating-system resources must be swapped during a thread switch within the same process versus a full process context switch across distinct address spaces.

Determine which registers and memory structures define an execution continuation versus an address space.

During a thread switch within the same process, the kernel swaps only thread-private execution state: the kernel stack pointer (`sp`), user stack pointer, architectural CPU register file (callee-saved registers in cooperative switches, or full trapframe in preemptive switches), thread-local storage base pointers, and thread-specific signal masks. The virtual address space (`satp` root page table), open file descriptor table, current working directory, and shared memory segments remain unchanged. During a full process context switch, the kernel additionally reloads `satp` (flushing or tagging TLBs with ASIDs), switches process credential contexts, and points to a distinct descriptor table.

Exercise 2.1.2: C ABI Invariants in Context Switching vs Asynchronous Traps

Verification application

Explain why the kernel `swtch(&old->context, &new->context)` assembly routine on RISC-V saves only callee-saved registers (`ra` , `sp` , `s0..s11`), whereas the user-to-kernel `trapframe` must save all 31 general-purpose registers.

Contrast the caller/callee contract of a standard C function call with an arbitrary asynchronous interrupt.

The `swtch()` routine is invoked as a standard C function call by the kernel scheduler. According to the RISC-V calling convention (ABI), the compiler automatically generates code to spill any caller-saved temporary registers (`t0..t6` , `a0..a7`) that contain live values across the call onto the thread kernel stack. Thus, `swtch()` only needs to preserve callee-saved registers to restore the C execution context. In contrast, an asynchronous user trap (such as a timer interrupt) can strike at any arbitrary instruction boundary where any or all of the 31 user registers hold live values; the kernel has no compiler guarantees and must save every user register to preserve transparency.

Exercise 2.1.3: Multicore Mutual Exclusion and Double-Execution Prevention

Formalisation transfer

Consider a multicore system where two idle CPU cores run `scheduler()` loops concurrently. State the exact locking invariant and state transition linearization point that prevents both cores from selecting the same `RUNNABLE` process and executing it simultaneously on two CPUs.

Analyze the atomic state check and lock acquisition in xv6 proc.c scheduler loop.

The invariant is: No process struct can have state `RUNNING` on more than one CPU, and a thread kernel stack can be active on at most one CPU core simultaneously. To linearize selection, the scheduler on each CPU acquires the candidate process lock (`acquire(&p->lock)`), checks `if (p->state == RUNNABLE)` , updates `p->state = RUNNING` , and calls `swtch()` while still holding `p->lock` . The newly scheduled process releases `p->lock` only after running on the new CPU core. Any competing CPU scheduler attempting to acquire `p->lock` spins until the state transition to `RUNNING` is complete, whereupon it observes `p->state != RUNNABLE` and skips it.

Exercise 2.1.4: Execution Trace of Blocking Pipe Read and Waiter Wakeup

Description transfer

Trace the process state transitions, queue manipulations, and context switches when Process A blocks reading an empty pipe, Process B writes data to the pipe, and Process A is subsequently scheduled and completes its read.

Trace the lock handoff between pipe lock and process lock during sleep() and wakeup().

1. Process A acquires the pipe lock, discovers the pipe buffer is empty, and calls `sleep(&pipe->nread, &pipe->lock)`.
2. Inside `sleep()`, A acquires `a->lock`, releases `pipe->lock`, updates `a->state = SLEEPING`, records the channel `a->chan = &pipe->nread`, and calls `sched()` to switch context to the per-CPU scheduler.
3. The CPU scheduler releases `a->lock` and schedules another process.
4. Process B runs, acquires `pipe->lock`, copies bytes into the pipe buffer, and calls `wakeup(&pipe->nread)`.
5. `wakeup()` scans the process table, acquires `p->lock` for each sleeping process, and upon matching `p->chan == &pipe->nread`, updates `p->state = RUNNABLE` and releases `p->lock`.
6. Later, an idle CPU scheduler acquires `a->lock`, observes `a->state == RUNNABLE`, sets `a->state = RUNNING`, and switches context to A.
7. A resumes inside `sleep()`, re-acquires `pipe->lock`, releases `a->lock`, reads the newly written bytes from the buffer, releases `pipe->lock`, and returns to user space.

Exercise 2.1.5: Failure Modes of Direct Struct Byte-Copying in Fork Implementation

Construction synthesis

A junior operating systems engineer attempts to optimize `fork()` by executing a raw memory copy (`memcpy(child_proc, parent_proc, sizeof(struct proc))`). Detail three catastrophic failure modes that manifest during execution.

Identify synchronization objects, stack pointers, and kernel continuations that cannot be aliased.

1. Lock corruption and deadlock: The copied spinlock (`child->lock`) inherits the locked/unlocked state and holder ID of the parent. If copied while held or contended, the child lock is poisoned, leading to deadlocks or assertion panics in the scheduler.
2. Kernel stack aliasing: The child struct inherits the parent kernel stack pointer (`sp`). When the child runs, its push and pop operations overwrite active stack frames of the parent, resulting in immediate kernel call chain corruption.
3. Scheduler continuation aliasing: The child inherits the parent scheduler context (`context.ra` and `context.sp`). Rather than returning through the dedicated `forkret` entry point that prepares a fresh user trapframe return, the child resumes at the parent suspended kernel continuation with parent-specific local variables.

Exercise 2.1.6: Designing Context Switching for Multi-Threaded Processes with Migration

Construction synthesis

Design the context switch protocol for a multi-threaded kernel supporting multiple kernel threads per address space and core migration. Detail when `satp` must be reloaded, how thread-local register state is managed, and how migration across cores preserves cache and TLB consistency.

Check whether incoming and outgoing threads share the same page table root to optimize MMU reloads.

1. Address space check: The context switch routine inspects `prev->proc == next->proc`. If both threads belong to the same process, reloading `satp` and issuing `sfence.vma` is bypassed, eliminating unnecessary TLB invalidation overhead. If switching across distinct processes, `satp` is updated to `next->proc->pagetable` and an address-space-tagged TLB sync is executed.
2. Thread-local state: The user thread-pointer register (`tp` or thread-local base CSR) is restored from `next->trapframe->tp`.
3. Migration safety: If a thread migrates from CPU i to CPU j, CPU j acquires `thread->lock` before switching. The release fence in the lock ensures all stores from CPU i are visible to CPU j. Lazy floating-point/vector registers must be explicitly saved from CPU i hardware registers before allowing CPU j to activate the thread.

References & Further Reading

1. MIT PDOS, *xv6: a simple, Unix-like teaching operating system*, Chapter 7.
2. MIT PDOS, xv6 rev5 `kernel/proc.h`, `kernel/proc.c` combined sleep/wakeup path, `kernel/pipe.c` read path, `kernel/swtch.S`, and `kernel/trap.c`.
3. RISC-V International, RISC-V ABIs Specification, integer register convention (CC BY 4.0).
4. The Open Group, POSIX.1–2024 Base Definitions, definition of a live thread.
5. The Open Group, POSIX.1–2024 `read()`, empty-pipe blocking and EOF behavior.

SECTION DETAIL

Process API, IPC, and Signals

A shell runs `generate | compress`. Both programs finish their useful work, but `compress` waits forever for EOF. The parent retained one copy of the pipe's write descriptor. No process will write another byte, yet the kernel must treat writing as still possible while any write end remains open. The defect is not in `read`; it is an ownership leak created during process construction.

Process APIs are easiest to reason about as transformations of four graphs:

1. process and parent-child relationships;
2. process images and threads;
3. descriptor edges to kernel open-file objects; and
4. asynchronous event state: mask, pending set, and disposition.

Prediction — who still owns the write end?

Start with parent descriptors `r` and `w` from `pipe()`. The parent forks a producer and consumer. No close calls occur. The producer exits after writing; the consumer reads until EOF. List every process that retains `w` after both forks and explain why producer exit is insufficient.

`fork` duplicates descriptor-table edges; it does not move them.

After both forks, parent, producer, and consumer each retain a descriptor referring to the pipe's write-end open file description. Producer exit closes only its copy. Parent and consumer still keep the write end open, so an empty-pipe read blocks instead of returning zero. The consumer must close its inherited write descriptor, the producer must close its read descriptor, and the parent must close both ends once it no longer needs them. The missed concept is reachability/lifetime, not byte buffering.

`fork`, `exec`, and `wait` form a constructor protocol

The operations do not all "create a process." Their state transformations differ.

> **DEFINITION**
>
> Process construction is a protocol because identity creation, image replacement, descriptor transformation, and lifecycle observation occur in separate operations with separate failure cuts.

Operation	Process identity	Address space / image	Threads	Descriptor effect	Return behavior
`fork()`	creates a child with a new PID and parent link	child begins as a specified copy of parent state	child contains only the calling thread in a multi-threaded parent	copied descriptors refer to the same corresponding open file descriptions	returns child PID in parent, `0` in child, `-1` on failure
`execve()` family	preserves process identity	replaces current process image	successful call leaves the initial thread of new image; other old threads end	descriptors normally survive unless close-on-exec applies	does not return on success
`_exit(status)`	enters terminal lifecycle state	image no longer executes	no live execution remains	descriptors close as process resources are released	never returns
`waitpid()`	parent selects eligible child status	does not replace parent image	caller may block	no descriptor transformation by itself	reports child/status or error; consumes waitable status as specified

The classic shell sequence is `fork` to create a distinct process identity, modify descriptors in the child, then `exec` to replace only that child's image. Performing descriptor redirection before `fork` would also redirect the parent. Calling `exec` without `fork` would replace the shell itself.

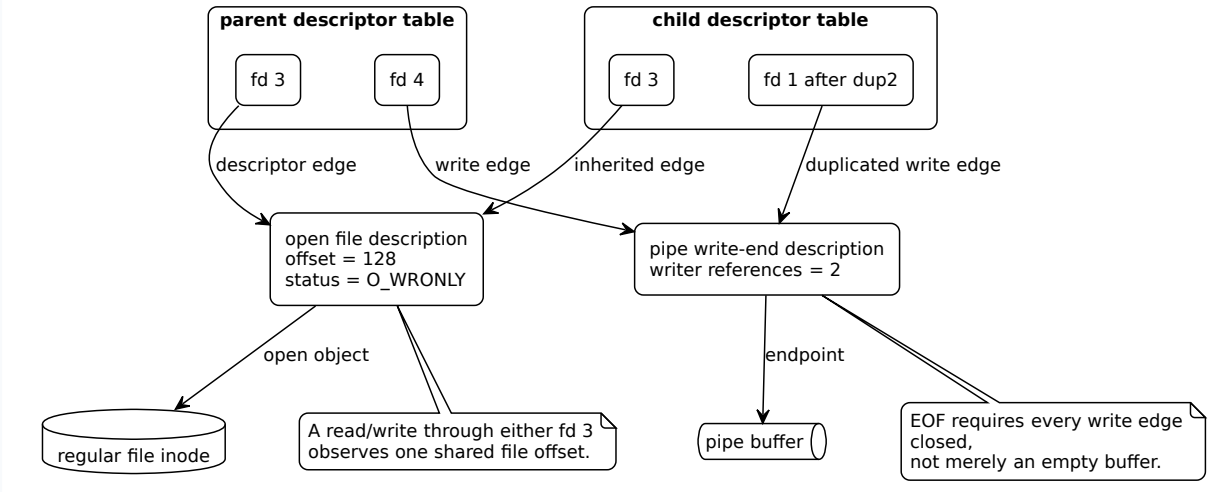
Retrieval — return paths are evidence

Why must code after a successful `execve()` be treated as an error path, while code after `fork()` must branch on three outcomes?

Successful `execve()` overlays the image and does not return to the old call site; reaching the following statement means failure and `errno` is meaningful. `fork()` returns in two processes with different values and can fail in the original process: negative for failure, zero in the child, positive child PID in the parent. A branch that handles only parent/child silently treats resource exhaustion as a child path or continues with an invalid PID.

Descriptor number is not open-file identity

`open()` creates an **open file description** containing status flags and file offset, then creates a process-local **file descriptor** that refers to it. `dup2(old, new)` makes `new` refer to the same open file description as `old`; `fork()` creates child descriptors referring to the corresponding same descriptions. Descriptor flags such as close-on-exec belong to descriptor edges, while file status and offset belong to the shared open-file description.



Closing a descriptor removes one edge. The kernel object remains alive while another edge or internal reference exists. Reusing the same small integer later does not resurrect the old object.

Application — shared offset trace

Parent opens a regular file at offset 0, forks, then parent reads 4 bytes and child reads 4 bytes through the inherited descriptor. Assume the reads occur in that order and both succeed. What offsets can each operation use? How would reopening the pathname in the child change the answer?

The inherited descriptors refer to one open file description. Parent consumes bytes at offsets 0–3 and advances the shared offset to 4; child then consumes offsets 4–7 and advances it to 8. If the child independently calls `open`, it receives a new open file description with its own initial offset, normally 0, so both may read bytes 0–3. A pathname identifies a file for lookup; it is not the runtime open-file object.

FILE DESCRIPTOR AND DESCRIPTION SHARING MATRIX

A parent opens a regular file, forks a child, and both read via the inherited descriptor. Evaluate each statement regarding file descriptor sharing:

Answer: Both child and parent descriptors refer to the same open file description, sharing one advancing file offset.; If the child independently calls open() on the pathname, it obtains a distinct file description and private offset.

File offsets reside in the system-wide open file description object, which child descriptors inherit by reference on fork(). Reopening via open() constructs a new description with an independent offset.

Worked trace: `sort < raw | uniq -c > counts`

Let pipe ends be `p[0]` (read) and `p[1]` (write). Let `infd` and `outfd` denote opened files. The target post-`exec` descriptors are:

Process image	fd 0	fd 1	Descriptors that must be closed before `exec`
`sort`	`infd`'s open file description	pipe write end	`infd`, `outfd`, `p[0]`, original `p[1]` after `dup2`
`uniq -c`	pipe read end	`outfd`'s open file description	`infd`, `outfd`, original `p[0]` after `dup2`, `p[1]`
parent shell	unchanged	unchanged	`infd`, `outfd`, `p[0]`, `p[1]` after both children are created

One robust order is:

1. create the pipe and open both redirection files using close-on-exec where available;
2. fork the `sort` child; in it, atomically duplicate `infd→0` and `p[1]→1`, close all originals and unrelated ends, then `exec`;
3. fork the `uniq` child; in it, duplicate `p[0]→0` and `outfd→1`, close all originals and unrelated ends, then `exec`;
4. parent closes every pipeline descriptor and waits for both child PIDs, retrying waits interrupted before status is obtained as required by its policy.



The construction must handle failures between forks. If the second fork fails, the parent still closes its pipe ends, terminates or drains policy-consistently, and reaps the first child. Error paths are part of descriptor ownership.

Failure injection — close one edge at a time

Predict behavior for each mutation.

1. consumer retains `p[1]`;

2. parent closes `p[0]` before forking the consumer;
3. producer forgets to close original `p[1]` after `dup2(p[1],1)`;
4. parent waits for producer before creating consumer while producer writes more than pipe capacity.

1. Consumer waits forever for EOF after draining data because it is itself a writer reference.
2. The consumer cannot inherit the read end; `dup2` fails or an accidentally reused descriptor is redirected.
3. This is usually harmless for EOF after producer exit because both edges close together, but it leaks a descriptor during the image lifetime and becomes harmful if fd 1 closes earlier while the extra remains. Tests must cover staged closes, not only process exit.
4. Producer can block on a full pipe while parent waits for it; consumer does not yet exist to drain, producing a construction deadlock. Create all pipeline stages before waiting.

Pipe semantics are state-dependent

For a blocking read on an empty pipe:

DEFINITION

Pipe EOF means the buffer is empty and no write-end reference remains; emptiness alone is not EOF.

Writer references	Buffer	Result
at least one	empty	block until data, signal interruption, or all writers close
zero	empty	return `0` (EOF)
any	non-empty	return available bytes subject to interface rules

EOF is therefore a statement about future possibility, not a byte stored in the pipe. A leaked descriptor changes the kernel's proof about future writes.

Transfer — request/reply over two pipes

Two processes use pipe A for requests and pipe B for replies. Each writes a request larger than pipe capacity before reading the reply. Construct the deadlock and give two redesigns with different trade-offs.

If both large writes fill their outbound pipe and block waiting for readers, neither reaches its read. Redesign 1 uses an event loop with nonblocking descriptors and bounded output queues; it adds readiness and partial-write state. Redesign 2 assigns a reader thread or process per direction; it adds concurrency and synchronization. A protocol-level redesign can bound each frame below available capacity and enforce alternating request/reply, but that couples correctness to capacity and excludes streaming. "Pipes are full duplex" is also wrong here: each POSIX pipe is one directed channel in the model.

`waitpid` closes the lifecycle loop

An exited child may retain minimal status and identity so its parent can observe termination. This zombie state is not runnable. `waitpid` selects status from an eligible child set; a parent that never waits can accumulate unreaped children until another system policy intervenes.

WARNING

Completion order and pipeline-stage order are different; status must be attributed by returned PID rather than by wait-call position.

Robust supervisors record PIDs and associate status with the correct stage. Waiting "for any child" without a mapping can attribute failure to the wrong pipeline command. A pipeline's overall exit policy —last stage, first failure, or aggregate—is an application decision, not a fact supplied by `waitpid`.

Application — status attribution

Three children exit in order C, A, B with statuses 0, 7, 0. The parent calls `waitpid(-1, ...)` three times. What extra state is necessary to report "stage A failed with 7," and what race does the negative PID argument **not** create?

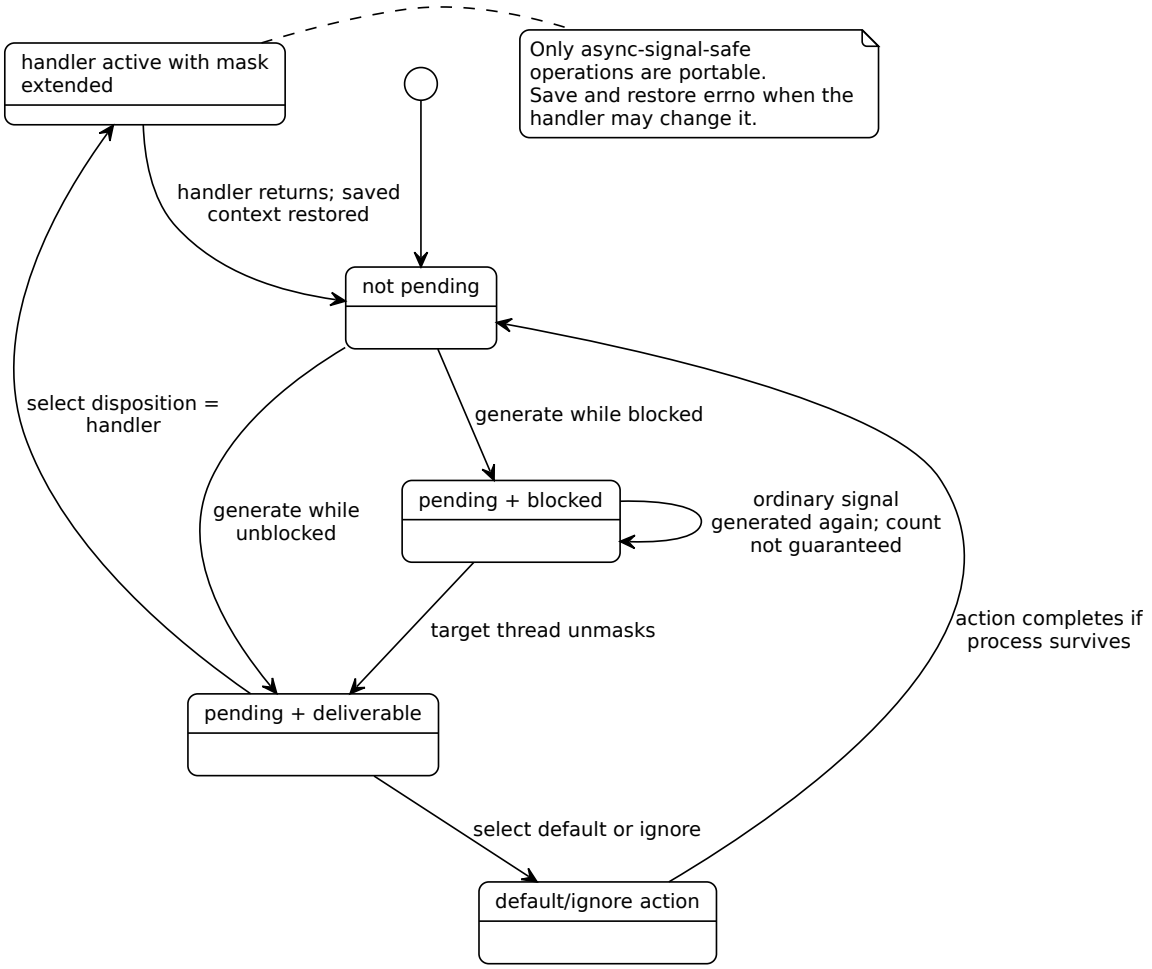
The parent needs a PID-to-stage mapping recorded at each successful fork and must decode each returned status. `waitpid(-1, ...)` can return any eligible child in completion order; it does not lose identity because it returns the PID. The race is in application bookkeeping if the mapping is missing or reused, not in the kernel confusing children. Waiting each specific PID simplifies attribution but may delay observing already-finished siblings.

Signals are asynchronous state transitions, not messages with queues by default

Separate three moments:

1. **generation:** an event targets a process or thread;
2. **pending:** the signal has been generated but is blocked or not yet selected for delivery;
3. **delivery/acceptance:** the kernel applies the current disposition to a selected thread or a waiting interface.

A per-thread signal mask blocks delivery, not necessarily generation. A disposition can be default action, ignore, or a handler installed with `sigaction`. For ordinary non-realtime signals, repeated generation while already pending need not produce one delivery per occurrence; using a handler invocation as an event counter is therefore unsound. Realtime signals have different queuing rules and are deliberately outside this core pattern.



Counterexample — `printf` in a handler

The main thread is inside `printf`, holding an internal stdio lock, when a signal handler calls `printf` again. The handler can deadlock or corrupt library state. The same reasoning applies to `malloc` and arbitrary application data structures. `volatile` does not make a general library operation reentrant and is not thread synchronization.

The safest handler often records the smallest possible notification through an async-signal-safe operation, then lets ordinary code perform work. A self-pipe pattern is one option:

```c
static int notify_fd;

static void on_sigchld(int signo) {
    int saved_errno = errno;
    unsigned char byte = (unsigned char)signo;
    (void)write(notify_fd, &byte, 1); /* write end configured
nonblocking */
    errno = saved_errno;
}
```

The ordinary event loop drains the pipe and repeatedly calls `waitpid(-1, &status, WNOHANG)` until no more completed children are available. If the nonblocking pipe is full, dropping a notification byte is acceptable only because the loop treats it as a level-triggered prompt to reap **all** available children; signal counts are not inferred from bytes.

Signal trace — two children, one pending `SIGCHLD`

1. Parent blocks `SIGCHLD` while installing its handler and constructing the notification pipe.
2. Child A exits; `SIGCHLD` becomes pending.
3. Child B exits before the signal is delivered. The implementation need not queue a second ordinary `SIGCHLD` occurrence.
4. Parent unblocks; handler runs once and writes one byte.
5. Event loop drains the byte and loops `waitpid(..., WNOHANG)`, reaping A and B.

The loop is correct because child status is the durable source of truth; signal delivery is only a wakeup hint.

Transfer — handler installation race

A supervisor forks a fast child, then installs its `SIGCHLD` handler. Sometimes the child exits first. Give a race-free ordering and explain why installing the handler earlier may still be incomplete in a multi-threaded process.

Block `SIGCHLD` in the constructing thread before creating the child, install the disposition and notification machinery, record the child's PID, then unblock or use a synchronous signal-wait design. In a multi-threaded process, masks are per-thread and another unblocked thread may receive the process-directed signal; establish the mask policy in all relevant threads, commonly by blocking before creating worker threads. Even with a lost notification, a supervisor can perform an initial nonblocking reap after registration, but that must be part of the protocol.

The multi-threaded `fork` boundary

POSIX specifies that a child of a multi-threaded `fork()` contains only the calling thread but inherits an address-space image that may contain mutexes held by vanished threads. Until a successful `exec`, the child is sharply constrained; calling code that acquires such a lock can deadlock. The portable construction should use only permitted async-signal-safe operations in that interval, or use `posix_spawn` under a separately studied contract.

> REMARK
>
> Each omitted IPC or process-control interface changes at least one graph—authority, descriptors, lifecycle, or asynchronous delivery—and therefore needs a new ownership trace.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 2.2.1: Distinguishing Integer File Descriptors from Open File Descriptions

Description application

Explain the architectural distinction between a per-process integer file descriptor, a system-wide open file description (`struct file`), and an underlying filesystem inode. Trace how file offset updates are shared across processes after `fork()`.

Trace the three-tier pointer chain from fd table index to struct file to struct inode.

An integer file descriptor is merely an index into the per-process file descriptor table (`proc->ofile[fd]`). That entry points to a system-wide open file description (`struct file`), which holds the current read/write byte offset, access mode flags (`O_RDONLY`, `O_WRONLY`), and a reference count. The open file description in turn points to an `inode` representing the physical file on storage. When a process calls `fork()`, the child inherits a copy of the parent's descriptor table; corresponding entries point to the *same* `struct file` instances with incremented reference counts. When either parent or child reads or writes, it updates the offset stored inside the shared `struct file`, so sequential operations naturally interleave across the single advancing stream.

Exercise 2.2.2: Descriptor Closing Invariants for Pipeline EOF Propagation

Construction application

In a Unix pipeline `cmd1 | cmd2`, the shell creates a pipe `int p[2]`, forks both children, duplicates descriptors, and executes both commands. Explain why the shell itself must close `p[1]` after forking both children, and identify the failure mode if it remains open in the shell.

Recall that read() on a pipe returns EOF (0 bytes) only when the write reference count reaches zero.

A pipe read end (`p[0]`) returns end-of-file (`read() == 0`) if and only if the pipe buffer is empty AND the kernel reference count for the write end of the pipe reaches zero. If the parent shell fails to close its inherited write descriptor `p[1]`, the write reference count remains at least 1 even after `cmd1` completes and closes its write end. Consequently, `cmd2` blocks indefinitely waiting for further input on `p[0]`, resulting in a pipeline hang / deadlock.

Exercise 2.2.3: Bidirectional Pipe Deadlocks and SIGPIPE Generation

Verification transfer

Two processes A and B communicate over two unidirectional pipes P_1 ($A \to B$) and P_2 ($B \to A$). Construct a deadlock schedule where both processes block, and explain what happens if process B crashes while process A attempts to write to P_1.

Consider finite pipe buffer capacities and write operations to broken pipes without readers.

Deadlock occurs if both processes fill their outgoing pipe buffers without draining their incoming pipes. For example, A writes 64 KiB to P_1 (filling the kernel buffer) and then blocks attempting to write more before reading from P_2. Concurrently, B writes 64 KiB to P_2 and blocks attempting to write before reading from P_1. Both processes sleep waiting for write buffer space that will never be drained. If process B crashes, all its descriptors (including the read end of P_1) are closed by the kernel. When process A attempts to write to P_1 with zero active readers, the kernel generates a `SIGPIPE` signal to A and sets `errno = EPIPE`. If `SIGPIPE` is ignored, the `write()` returns -1 with `EPIPE`.

Exercise 2.2.4: Standard Signal Coalescing and Non-Blocking Waitpid Loops

Formalisation transfer

A parent process spawns 5 child processes. All 5 children exit almost simultaneously. Explain why a signal handler that executes `wait(NULL)` exactly once per `SIGCHLD` delivery can orphan child processes, and formulate the correct reaping loop.

Standard POSIX signals represent pending bits that coalesce multiple arrivals into a single flag.

Standard POSIX signals (signals 1–31) are represented in the process struct as a single bitmask of pending signals without a FIFO queue. If multiple children exit while `SIGCHLD` is blocked or pending, the bit is set once; additional terminations are coalesced and lost. If the handler executes `wait(NULL)` only once, only the first child is reaped, leaving the remaining 4 children as permanent zombies until process termination. The correct handler implementation must reap in a non-blocking loop:

```
void sigchld_handler(int sig) {
    int saved_errno = errno;
    while (waitpid(-1, NULL, WNOHANG) > 0) {}
    errno = saved_errno;
}
```

This reaps all exited children until `waitpid()` returns 0 (children still running) or -1 with `ECHILD` (no children left).

Exercise 2.2.5: Async-Signal-Safety Constraints in Multi-Threaded Fork

Description synthesis

POSIX specifies that after `fork()` in a multi-threaded process, the child process contains only the calling thread, but inherits all user-space heap memory and lock states. Explain why calling `malloc()` or `printf()` in the child before `exec()` can cause an unrecoverable deadlock.

Consider a background thread that was inside a heap allocator mutex at the instant fork() was called.

In a multi-threaded process, standard library functions like `malloc()` and `printf()` acquire internal mutexes protecting the heap allocator free lists and stdio output buffers. If another thread in the parent process happened to hold the allocator lock at the precise instant `fork()` executed, the child process inherits that memory image with the lock marked held. Because other parent threads do not exist in the child, no thread will ever release that lock. If the child calls `malloc()`, it attempts to acquire the held lock and deadlocks permanently. Between `fork()` and `exec()`, the child must only execute async-signal-safe system calls (such as `close()`, `dup2()`, and `execve()`).

Exercise 2.2.6: Multi-Stage Pipeline Launcher with Diagnostic Failure Handling

Construction synthesis

Design a robust Unix pipeline launcher for `cmd1 | cmd2 | cmd3` that detects and cleanly recovers from pre-`exec` redirection errors in child processes without leaking open file descriptors or leaving zombie processes.

Use unidirectional diagnostic pipes with O_CLOEXEC to communicate exec errors back to the parent.

1. Create inter-process data pipes `p1[2]` and `p2[2]` with `pipe2(..., O_CLOEXEC)`.
2. Create an auxiliary diagnostic pipe `diag[2]` with `O_CLOEXEC`.
3. In each child:
   - Use `dup2()` to bind `p[0]` to `STDIN_FILENO` and `p[1]` to `STDOUT_FILENO`.
   - Call `execvp()`. If `execvp()` returns, an error occurred. The child writes the `errno` integer to `diag[1]` and immediately invokes `_exit(127)`.
4. If `execvp()` succeeds, the kernel automatically closes `diag[1]` due to `O_CLOEXEC`, delivering EOF to the parent on `diag[0]`.
5. The parent closes all data pipe write ends and reads from `diag[0]`. If it reads a non-zero integer, a child failed pre-`exec`; the parent terminates surviving children via `kill(SIGTERM)` and loops over `waitpid()` to reap all children cleanly.

References & Further Reading

1. The Open Group, POSIX.1–2024 `fork()`, `exec`, and `wait()/waitpid()`.
2. The Open Group, POSIX.1–2024 `open()`, `dup()/dup2()`, `pipe()`, and `read()`.
3. The Open Group, POSIX.1–2024 `sigaction()` and General Information §2.4, for signal generation, delivery, masks, and async-signal-safety.
4. MIT PDOS, *xv6: a simple, Unix-like teaching operating system*, Chapters 1 and 7, plus rev5 `kernel/proc.c`, `kernel/file.c`, and `kernel/pipe.c`, as a compact implementation contrast rather than a complete POSIX implementation.

SECTION DETAIL

CPU Scheduling as Policy

Four jobs arrive at a single CPU. A long batch job arrives first; three short interactive jobs arrive immediately after it. First-come, first-served keeps the CPU busy and completes every job, yet users wait behind work that is irrelevant to their requests. “100% utilization” has concealed a latency failure.

A scheduling policy chooses among runnable execution contexts. Timer interrupts, context save/restore, and run-queue operations are mechanisms. The choice is defensible only relative to a workload and objective.

Prediction — one trace, three objectives

Jobs $A(0, 8), B(0, 2), C(1, 1)$, and $D(2, 2)$ are written as `(arrival, CPU service)`. With no preemption or switch cost, predict which first job minimizes each objective: completion time of A, mean response time, and maximum waiting time. One choice need not optimize all three.

An objective about A is not an objective about the population.

Running A first minimizes A’s completion time. Choosing a short ready job such as B improves mean response relative to A-first. Maximum waiting depends on the full order; among simultaneous known jobs, short-first tends to delay A substantially, while long-first delays every short job. The point is not a unique answer before constructing complete schedules: the objectives are different functions and cannot be replaced by “fast.”

Workload model and metrics

For job i, define:

DEFINITION

A scheduling metric is a function of explicit arrival, execution, blocking, and completion cuts; its name does not determine which intervals are charged.

- arrival a_i;
- first dispatch f_i;
- completion c_i;
- total required CPU service b_i; and
- total blocked time d_i during the observation.

Then

$$\begin{aligned} \text{response}_i &= f_i - a_i, \\ \text{turnaround}_i &= c_i - a_i, \\ \text{ready-wait}_i &= c_i - a_i - b_i - d_i. \end{aligned}$$

In the teaching traces, $d_i = 0$, so waiting is turnaround minus service. Under preemption, response counts only until the **first** dispatch; later ready intervals contribute waiting but not response.

For an interval of length L:

$$\text{throughput} = \frac{\text{completed jobs}}{L}, \qquad U_{useful} = \frac{\text{time executing job instructions}}{L}.$$

Scheduler and switch work may keep the CPU physically busy while reducing U_{useful}. Report both if the distinction matters.

Worked trace — metric reconstruction for a preempted job

Job P arrives at 2, runs on intervals $[4, 6)$ and $[10, 13)$, blocks on I/O during $[6, 8)$, and completes at 13. Compute response, turnaround, service, blocked time, and ready-wait.

Response is $4 - 2 = 2$. Turnaround is $13 - 2 = 11$. CPU service is $2 + 3 = 5$. Blocked time is 2. Ready-wait is $11 - 5 - 2 = 4$: two units before first dispatch and two from wakeup at 8 until redispatch at 10. Using turnaround minus service alone would report 6 and incorrectly charge blocked time to the CPU scheduler.

CPU SCHEDULING METRICS CLASSIFICATION MATRIX

Job Q arrives at 3, is first dispatched at 5, blocks on I/O during [7,9), resumes at 10, and completes at 14 with 6 units of CPU service. Evaluate each statement:

Answer: Ready-waiting time is 3 units, calculated by subtracting 6 service units and 2 blocked units from 11 turnaround units.; Response time is 2 units, measured as the initial latency from arrival at time 3 until first CPU dispatch at time 5.

Ready-waiting time measures only time spent in the runnable state waiting for a core (here 2 units before first dispatch plus 1 unit from wakeup at 10 to completion at 14, totaling 3 units). Blocked I/O time must not be charged to the CPU scheduler.

FCFS and the convoy effect

First-come, first-served orders ready work by arrival, with an explicit tie rule. Under non-preemptive FCFS and tie order A, B, C, D, the motivating workload becomes:

```
time  0        8  10 11  13
CPU   |   A    | B |C| D |
```

Job	First run	Completion	Response	Turnaround	Waiting
A	0	8	0	8	0
B	8	10	8	10	8
C	10	11	9	10	9
D	11	13	9	11	9
mean	—	—	**6.50**	**9.75**	**6.50**

The CPU is usefully busy for all 13 units, but short jobs inherit the long job's service time. This is the convoy effect. FCFS remains valuable when order itself is the contract, overhead must be minimal, or jobs have similar service demands. A counterexample is not a universal rejection.

Transfer — arrival order versus queue order

At time 5, a running FCFS job blocks. Two jobs already wait; a third arrives while the CPU is selecting. What must the policy specify to remain deterministic, and why is "earliest PID" not FCFS?

Specify the linearization of runnable insertion and the tie rule for equal logical arrival times. Existing waiters should precede the newly runnable/arriving job under FIFO semantics. PID is identity, not arrival order; reuse and allocation patterns can reorder service. A real concurrent queue needs synchronization around insertion/selection, but the policy contract must exist before implementation.

SJF: an optimum inside a narrow theorem

Assume n non-preemptive jobs all arrive at time zero, service times are known exactly, and the objective is mean waiting time. Consider adjacent jobs i then j after prefix duration T, with $b_i > b_j$. Their contribution to total waiting is

$$T + (T + b_i) = 2T + b_i.$$

Swapping them contributes

$$T + (T + b_j) = 2T + b_j,$$

which is smaller by $b_i - b_j > 0$. Repeatedly removing every inversion yields nondecreasing service order. Therefore shortest-job-first minimizes total—and hence mean—waiting under these assumptions.

This proof says nothing about unknown bursts, unequal arrivals, preemption, deadlines, weights, or maximum waiting.

For the motivating workload, non-preemptive SJF can choose B at time 0; by time 2 C and D have arrived:

```
time  0  2 3   5        13
CPU   | B|C| D |    A    |
```

Responses are A=5, B=0, C=1, D=1; mean response and waiting are 1.75. Mean turnaround is $(13 + 2 + 2 + 3)/4 = 5.00$. The improvement is large because exact service times were supplied by the exercise.

Proof boundary — construct starvation

Suppose a long job L with service 100 is ready. Can an arrival stream of short jobs (service 1) starve L indefinitely? Distinguish non-preemptive SJF from preemptive SRTF:

1. **Non-preemptive SJF with L alone at time 0:** If L is the sole ready job at $t = 0$, non-preemptive SJF immediately dispatches L. Because the policy is non-preemptive, subsequent arrivals cannot displace L; it runs uninterrupted from $t = 0$ to $t = 100$. Starvation does **not** occur.
2. **True starvation under non-preemptive SJF:** Requires that at $t = 0$, another job S_0 is already running or ready, and before each short job finishes, a new short job arrives. At every scheduling decision boundary, the scheduler sees at least one shorter job ready, so L is never selected.
3. **Starvation under preemptive SRTF:** If short jobs arrive with remaining service $< \mathrm{remaining}(L)$, L is repeatedly preempted or delayed, suffering unbounded waiting.

Under non-preemptive SJF, a long job ready at time 0 cannot starve if it is selected first; once running, non-preemption protects it. Unbounded starvation requires either: (a) under non-preemptive SJF: at time 0, a short job S_0 is dispatched, and before each S_k completes, S_1 arrives so the ready queue is never empty of shorter jobs at any decision point; or (b) under preemptive SRTF: frequent short arrivals repeatedly preempt L.

The simultaneous-arrival batch theorem ($b_i \leq b_j$) does not prevent starvation because its foundational assumption—a finite, closed batch arriving simultaneously at $t = 0$—is violated by continuous dynamic arrivals. Real systems prevent this via dynamic priority aging.

SRTF and the value of preemption

Shortest-remaining-time-first re-evaluates the runnable job with least remaining service whenever the eligible set changes. Let A arrive at 0 with service 10 and B arrive at 1 with service 1.

EXAMPLE

New arrival information can make preemption reduce mean turnaround even when non-preemptive SJF made the best decision available at time zero.

- Non-preemptive SJF starts A because it is the only job at time 0: A runs 0–10, B 10–11. Mean turnaround is $(10 + 10)/2 = 10$.
- SRTF runs A 0–1, B 1–2, then A 2–11. Turnarounds are 11 and 1; mean is 6.

Preemption improves this trace because new information arrives. It also adds context-switch cost and can starve large jobs under sustained short arrivals. Exact remaining time is usually unavailable; systems infer behavior from history or use application-provided classes under an authority policy.

Application — preemption is not free

In the A/B trace, each change between jobs costs 0.75 time units and no useful work occurs during a switch. Recompute both completion times for SRTF if the scheduler charges A→B and B→A, but not initial dispatch.

A runs 0–1; switch to B takes 1–1.75; B runs 1.75–2.75; switch to A takes 2.75–3.5; A's remaining 9 units finish at 12.5. Turnarounds are A=12.5 and B=1.75, mean 7.125. This still beats the modeled non-preemptive mean 10, but the benefit shrinks. If switch cost exceeded the latency saved, preemption would lose. The scheduler must charge overhead to the schedule rather than treating it as an annotation.

Round robin: bound service gaps, choose a quantum

Round robin places runnable jobs in FIFO order and runs each for at most quantum q. A job that blocks or finishes early gives up the CPU; an unfinished job returns to the tail. With m continuously runnable jobs, zero switch cost, and no higher-priority interference, a job waits at most roughly $(m-1)q$ between its quanta. That is a model bound, not a production latency guarantee.

For the motivating workload with $q = 2$ and zero switch cost:

```
time  0  2  4 5  7  9  11 13
CPU   | A| B|C| D| A| A | A |
```

The queue after A's first quantum is B, C, D, A. Responses are A=0, B=2, C=3, D=3; mean response is 2.00. Turnarounds are A=13, B=4, C=4, D=5; mean turnaround is 6.50. Mean waiting is 3.25.

If every quantum causes a switch costing s, the saturated upper-bound overhead fraction is

$$\frac{s}{q+s}.$$

As $q \to \infty$, round robin approaches FCFS for finite jobs. As $q \to 0$, response opportunities become frequent while switching consumes the machine. A quantum is therefore a workload-timescale decision.

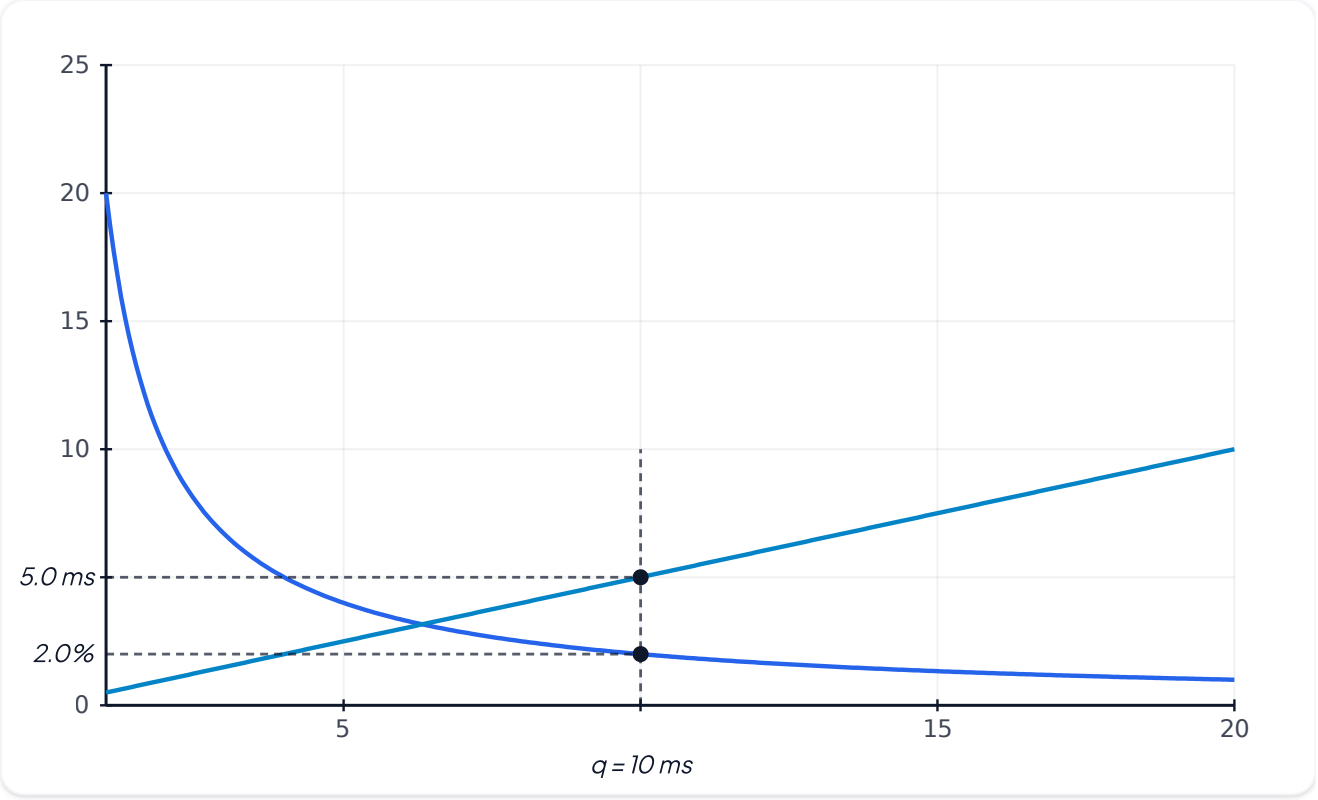


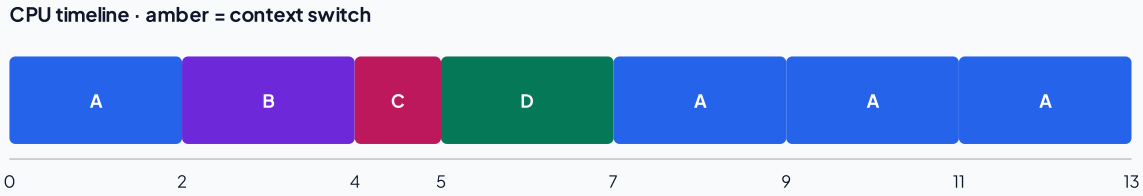
Figure 2.3.1: Round-Robin trade-off: context-switch overhead δ/q (decaying as $O(1/q)$) vs. interactive queue response latency (increasing linearly with quantum size q).

Prediction before interaction

For the convoy workload, predict the direction—not exact magnitude—of these changes:

1. FCFS → RR with zero switch cost: mean response and mean turnaround;
2. RR quantum 8 → quantum 1 with switch cost 2: response opportunity and useful utilization;
3. convoy → staggered workload: idle time.

SCHEDULE THE SAME JOBS UNDER COMPETING POLICIES



Job	Arrival	Burst	Response	Waiting	Turnaround
A	0	8	0	5	13
B	0	2	2	2	4
C	1	1	3	3	4
D	2	2	3	3	5

Static state: convoy workload, round robin, quantum 2, zero switch cost. Response improves over FCFS, but real systems must account for switch overhead and workload uncertainty.

The simulator charges switch cost only when the CPU changes jobs, assumes one CPU and CPU-only jobs, and gives SJF exact burst knowledge. It is a trace calculator for those assumptions. It does not model cache warming, I/O wakeups, priority, multicore migration, or production Linux.

Simulator investigation — falsify “smaller quantum is better”

Use the convoy workload and RR. Record mean response, turnaround, and useful utilization for at least four `(quantum, switch cost)` pairs. Find one pair where a smaller quantum improves response but makes turnaround and utilization worse. Explain the causal sequence.

Increase switch cost enough that extra boundaries become visible.

One valid investigation compares zero cost with cost 1 or 2 while lowering the quantum. Exact displayed means are the evidence. The causal claim must mention that more preemption points let short jobs begin earlier, while each inter-job boundary inserts non-service time and delays every completion. If the selected run has only one job remaining, consecutive quanta of that job do not incur inter-job switch cost in this model; count actual amber segments rather than multiplying all quanta blindly.

Unknown bursts and feedback policies

Exact SJF is an oracle policy. A practical scheduler can estimate the next CPU burst using an exponentially weighted history

WARNING

A burst predictor changes the information available to a policy; it does not turn a workload estimate into a service-time guarantee.

$$\hat{b}_{k+1} = \alpha b_k + (1-\alpha)\hat{b}_k, \qquad 0 \le \alpha \le 1.$$

Large α reacts quickly and forgets history; small α smooths noise and adapts slowly. This is a predictor, not a guarantee. Adversarial or phase-changing workloads defeat it.

A multilevel feedback queue (MLFQ) can favor tasks that yield quickly and demote tasks consuming full quanta. To be a complete policy it must specify:

- number of queues and quantum per queue;
- insertion on creation and wakeup;
- demotion/accounting across voluntary yields;
- priority boost or aging rule;
- preemption between queues; and
- treatment of multiple threads or process weights.

Without cumulative accounting, a CPU-bound job can yield just before each quantum expires and avoid demotion. Without aging, lower queues can starve.

Transfer — predictor phase change

A task alternates 100 bursts of length 1 with 100 bursts of length 20. Compare qualitatively $\alpha = 0.1$ and $\alpha = 0.9$ at each phase boundary. Which is "better"?

$\alpha = 0.9$ moves the estimate near the new phase within a few samples but follows individual noise; $\alpha = 0.1$ retains the old phase for many bursts and is stable within a stationary phase. Neither is universally better. If misclassifying a newly interactive phase causes severe response harm, rapid adaptation may matter; if burst measurements are noisy and classes drive costly migration, smoothing may win. Report prediction error by phase rather than one global mean.

Fairness, tails, and overload

Mean latency can hide one victim. A policy evaluation should include:

WARNING

An average can improve while one class starves; report a per-job or per-class bound whenever starvation freedom is part of the claim.

- median and high-percentile response/turnaround;
- maximum or bounded waiting under stated admission assumptions;
- per-class service share versus assigned weight;
- deadline-miss or service-level violation rate when relevant;
- useful utilization and switch/migration overhead; and
- starvation and gaming traces.

For continuously runnable tasks with positive weights w_i, a proportional-share target over a long interval is

$$\text{share}_i \approx \frac{w_i}{\sum_j w_j}.$$

Short intervals, wakeups, minimum granularity, and hierarchy create deviation. Fairness is always measured over a window and subject set.

Counterexample — the mean approves starvation

One policy completes 99 one-unit jobs immediately and delays one 100-unit job for 10,000 units. Another delays every short job by 5 but begins the long job promptly. Why can mean response alone select the first policy, and what constraint repairs the objective?

The 99 near-zero responses dominate the average, so one extreme victim contributes only 1% of samples. Add a maximum-wait or starvation-freedom constraint, report tail distribution, or optimize a weighted objective that reflects the long job's class. The repair is not to discard means; it is to prevent aggregation from erasing a required per-job guarantee.

Production comparison: virtual time and EEVDF

Current Linux scheduler documentation describes a transition from the older CFS design toward Earliest Eligible Virtual Deadline First (EEVDF). The documented model tracks virtual runtime and lag: a task with nonnegative lag is eligible, and the eligible task with earliest virtual deadline is selected. Shorter requested slices can improve latency while lag accounts for fair service.

This is not SJF: a virtual deadline is not an oracle for total job length. It is not the fixed-quantum RR simulator either: eligibility, weights, lag, wakeup behavior, and scheduler hierarchy matter. Production comparison should preserve those distinctions and cite the running kernel/version measured.

> **REMARK**
>
> A scheduler extension earns a place only with a new workload model, invariant, and falsifying trace—not by adding policy names to a comparison list.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 2.3.1: Convoy Effect and Head-of-Line Blocking in FCFS

Description application

Analyze the convoy effect under First-Come, First-Served (FCFS) scheduling when one CPU-bound task with an 80 ms burst arrives immediately ahead of ten interactive I/O-bound tasks with 2 ms bursts. Compute the average waiting time and describe the utilization of the CPU and I/O devices over time.

Contrast the FCFS schedule with an interleaved round-robin or shortest-job-first order.

Under FCFS, the 80 ms CPU job blocks all ten 2 ms interactive jobs. The first interactive job waits 80 ms, the second waits 82 ms, and the tenth waits 98 ms, yielding a mean waiting time of $\frac{80+82+\cdots+98}{11} \approx 80.9\ \text{ms}$. During this initial 80 ms window, the I/O devices sit completely idle. Once the CPU job finishes, all ten interactive jobs quickly execute for 2 ms and issue I/O, leaving the CPU idle while disk or network transfers complete concurrently. FCFS creates head-of-line blocking that penalizes interactive response and degrades device concurrency.

Exercise 2.3.2: Starvation in Shortest Job First Under Continuous Short Job Arrival

Proof application

Prove that non-preemptive Shortest Job First (SJF) and preemptive Shortest Remaining Time First (SRTF) can cause indefinite starvation for a long job, and formulate an aging mechanism that restores bounded turnaround.

Construct an adversarial workload where new short jobs arrive faster than they can be cleared.

Suppose a long job J_{long} with burst length $T = 100$ arrives at time 0. At every time interval $t = 5k$ (for $k = 0, 1, 2, \ldots$), a new short job J_k with burst length 4 arrives. Because $4 < 100$ (and $4 < \text{remaining}(J_{\text{long}})$), the SJF/SRTF dispatch rule perpetually prioritizes the newly arrived short job J_k over J_{long}. Thus, J_{long} is never scheduled and starves indefinitely. An aging mechanism dynamically decrements the effective burst estimate or increments an effective priority by $\alpha \cdot \text{waiting_time}$, guaranteeing that J_{long}'s priority eventually dominates every fresh arrival.

Exercise 2.3.3: Round-Robin Quantum Sizing and Switch Overhead Trade-off

Verification transfer

In a Round-Robin scheduler with context switch cost $\delta = 0.2$ ms, evaluate the fractional CPU overhead $\frac{\delta}{q+\delta}$ and maximum interactive response latency $(m-1)(q+\delta)$ across $m = 10$ runnable tasks for quantum lengths $q = 1$ ms, $q = 10$ ms, and $q = 100$ ms.

Apply the saturation overhead formula and queue cycle bounds.

1. $q = 1$ ms:
   - Overhead: $\frac{0.2}{1.0+0.2} = \frac{0.2}{1.2} \approx 16.67\%$.
   - Max latency between quanta: $(10-1)(1.0+0.2) = 9 \times 1.2 = 10.8$ ms.
2. $q = 10$ ms:
   - Overhead: $\frac{0.2}{10.0+0.2} = \frac{0.2}{10.2} \approx 1.96\%$.
   - Max latency between quanta: $9 \times 10.2 = 91.8$ ms.
3. $q = 100$ ms:
   - Overhead: $\frac{0.2}{100.0+0.2} = \frac{0.2}{100.2} \approx 0.20\%$.
   - Max latency between quanta: $9 \times 100.2 = 901.8$ ms. Small quanta guarantee snappy interactive response (10.8 ms) but waste 16.7% of CPU cycles in switching overhead; large quanta minimize overhead (0.2%) but degrade interactive responsiveness toward FCFS.

Exercise 2.3.4: Exponential Moving Average CPU Burst Estimation

Verification transfer

A process exhibits measured CPU bursts $t_1 = 12, t_2 = 4, t_3 = 6, t_4 = 8$. Using an exponential moving average $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$ with weighting factor $\alpha = 0.5$ and initial guess $\tau_1 = 10$, compute the predicted bursts $\tau_2, \tau_3, \tau_4, \tau_5$.

Apply the recursive formula step by step.

1. $\tau_2 = 0.5 \cdot t_1 + 0.5 \cdot \tau_1 = 0.5(12) + 0.5(10) = 6 + 5 = 11.0$.
2. $\tau_3 = 0.5 \cdot t_2 + 0.5 \cdot \tau_2 = 0.5(4) + 0.5(11.0) = 2 + 5.5 = 7.5$.
3. $\tau_4 = 0.5 \cdot t_3 + 0.5 \cdot \tau_3 = 0.5(6) + 0.5(7.5) = 3 + 3.75 = 6.75$.
4. $\tau_5 = 0.5 \cdot t_4 + 0.5 \cdot \tau_4 = 0.5(8) + 0.5(6.75) = 4 + 3.375 = 7.375$. The estimator dampens transient spikes while gradually tracking shifts in process behavior.

Exercise 2.3.5: Virtual Runtime, Lag, and Eligibility in EEVDF Scheduling

Description synthesis

Explain the concepts of virtual runtime, lag, and virtual deadlines in Linux Earliest Eligible Virtual Deadline First (EEVDF) scheduling. Describe how EEVDF prevents a short interactive task from being starved by compute-heavy threads while preserving weighted proportional-share fairness.

Contrast the eligibility check (lag >= 0) with the selection criterion (earliest virtual deadline).

In EEVDF, each entity has a weight w_i. Fluid fair-share time advances at rate $w_i / \sum w_j$. An entity's lag is the difference between the service it *should* have received in an ideal fluid model and the service it actually received: $\text{lag}_i = V - v_i$. An entity is defined as **eligible** if and only if $\text{lag}_i \geq 0$ (it has not received more than its fair share). Among eligible tasks, EEVDF selects the task with the earliest **virtual deadline** $d_i = v_i + q_i / w_i$, where q_i is the task's requested slice. An interactive task requests a small slice q_i, yielding an early virtual deadline that grants it immediate preemption upon becoming eligible, while compute threads with large slices are bounded to their proportional share without starving the interactive task.

Exercise 2.3.6: Scheduling Mixed Real-Time, Interactive, and Batch Workloads

Construction synthesis

Design a multi-queue scheduling policy for a single-core embedded controller running a hard real-time control loop (0.5 ms CPU every 10 ms), an interactive console daemon, and a background batch data compression worker. Specify admission checks, preemption triggers, quantum budgets, and overload shed policies.

Combine fixed-priority real-time scheduling with rate-monotonic analysis and round-robin background queues.

1. Queue hierarchy:
   - Priority 1 (Highest, Real-Time FIFO): Control loop C. Admitted only after rate-monotonic feasibility check: $U_C = 0.5/10 = 5\% \leq 69.3\%$. Preempts all lower queues immediately upon timer release.
   - Priority 2 (Interactive Round-Robin): Console daemon I. Allocated a small quantum $q = 2\ \text{ms}$ for low response latency.
   - Priority 3 (Lowest, Batch Background): Compression worker B. Runs with a large quantum $q = 20\ \text{ms}$ to maximize L1/L2 cache locality and reduce switch overhead.
2. Overload protection: If C overruns its 0.5 ms budget (detected via hardware watchdog or CPU timer), it is quarantined and an alarm logged. If I floods the system, Priority 2 is throttled to at most 20% CPU time, preserving residual bandwidth for B.

References & Further Reading

1. The Open Group, POSIX.1–2024 General Information §2.8.4: Process Scheduling, FIFO/RR conceptual semantics.
2. Linux kernel documentation, EEVDF Scheduler and CFS Scheduler, for current production virtual-time context.
3. MIT PDOS, xv6 rev5 `kernel/proc.c` and `kernel/trap.c`, for the teaching scheduler and timer-yield mechanism.
4. MIT PDOS, *xv6: a simple, Unix-like teaching operating system*, Chapter 7.

SECTION DETAIL

Scheduler Lab: Weighted Virtual Time in xv6

A scheduler can pass `usertests`, print plausible shares, and remain wrong. Placing a child below both its parent's charged debt and the scheduler floor rewards sequential fork/wait chains. Resetting virtual time after sleep lets an interactive task monopolize the CPU after every wakeup. Copying statistics to a bad user pointer while holding `p->lock` can freeze the process table. Selecting a candidate from an unlocked scan can run one process on two harts.

The lab treats a scheduling patch as a concurrent mechanism with evidence, not as a changed `if` statement.

> ***Original lab brief.*** *The task, milestones, interface, workloads, wrong-implementation matrix, and verification criteria below are original LibreUni material. They were designed against the permissively licensed upstream xv6 source and do not reproduce MIT 6.1810 lab instructions or solutions.*

Implementation contract and invariants

Implement a scheduler named **WVT** (weighted virtual time) in a detached checkout of xv6 rev5. Each runnable process has weight $w \in \{1, 2, 4, 8\}$ and virtual runtime v. When a process leaves `RUNNING` after consuming measured processor time Δt,

DEFINITION

WVT orders runnable processes by charged virtual service; weight changes the rate of debt accumulation, not the amount of physical time in a timer interval.

$$v \leftarrow v + \max\left(1, \left\lfloor \frac{\Delta t}{w} \right\rfloor\right).$$

On one hart, select a `RUNNABLE` process with minimum v; break exact ties by PID. Larger weight accumulates virtual time more slowly and therefore receives a larger long-run share while continuously runnable.

The complete implementation package contains:

- the patch, based on the exact commit below;
- `schedtests`, a user-space test binary with machine-checkable summaries;
- optional test-only kernel hooks isolated behind `#ifdef SCHED_TEST`;
- raw CSV or TSV measurements and the script/notebook used to summarize them;
- a design memo of at most 2,500 words plus appendices; and
- answers to the diagnostic analysis questions.

No full implementation appears here. Pseudocode exposes proof obligations, not line-by-line placement.

Entry diagnostic — explain before editing

Why does "pick the smallest `vruntime`" fail as a complete specification? List at least five missing rules.

Creation, sleeping, time measurement, ties, concurrency, and wraparound all change the order relation.

Missing rules include initialization, child placement at $\max(v_{parent}, floor)$, wakeup placement, permitted weights, when elapsed time is charged, how voluntary yield is charged, deterministic ties, candidate revalidation on multiple harts, lock order, overflow/renormalization, exit/reap accounting, and what "fair" means over a measurement window. Any five with consequences constitute a sound analysis. A data-field definition is not a scheduling policy.

VIRTUAL TIME DEBT ACCUMULATION MATRIX

Under Weighted Virtual Time (WVT), a process with weight w = 4 runs for delta_t = 6 timer ticks. Evaluate each statement regarding vruntime updating:

Answer: The update formula v = v + max(1, floor(delta_t / w)) adds exactly 1 unit of virtual runtime debt.; Higher weight processes accumulate virtual debt at a lower rate per unit of physical execution time.

Weight determines the rate of virtual debt accumulation. Here floor(6 / 4) = 1, so 1 tick of virtual runtime is added. Higher weights accumulate debt more slowly, receiving proportionally more physical CPU time.

Reproducible setup

The source authority is the upstream MIT PDOS repository. Preserve its copyright and permission notice.

```
git clone https://github.com/mit-pdos/xv6-riscv.git xv6-wvt
cd xv6-wvt
git switch --detach xv6-riscv-rev5
git rev-parse HEAD
git tag --points-at HEAD
git status --short
make clean
make CPUS=1 qemu
```

Expected `git rev-parse HEAD` output is `7d7adbb1b0acbd67c9766a20d0f9900fef2789fa`, the tag output is `xv6-riscv-rev5`, and status is empty. Upstream's rev5 Makefile requires a RISC-V GCC/binutils toolchain and QEMU 7.2 or newer. Record exact versions:

```
riscv64-unknown-elf-gcc --version || riscv64-linux-gnu-gcc --version
qemu-system-riscv64 --version
make --version
```

Inside xv6, run `usertests` and `forktest`; save complete console output. Repeat the baseline with `CPUS=3`. If the unmodified baseline fails, stop. A policy comparison against a broken or unpinned baseline is uninterpretable.

> ***Revision guard.*** *Rev5 declares* `sleep(void *chan, struct spinlock *lk)` *and* `copyout(pagetable_t, uint64, char *, uint64)`*, and defines* `r_time()` *in* `kernel/riscv.h`*. A checkout exposing the branch-head* `sleep_prepare` *split or an extra process-size argument to* `copyout` *is not this lab's baseline. Adapt the lab only to the rev5 signatures; do not combine source fragments from different revisions.*

Baseline record

Record this table before implementation:

Evidence	CPUS=1	CPUS=3
commit hash		
compiler/QEMU versions		
`usertests` result and duration		
`forktest` result		
first ten scheduler selections with PID/state		
unexpected console output		

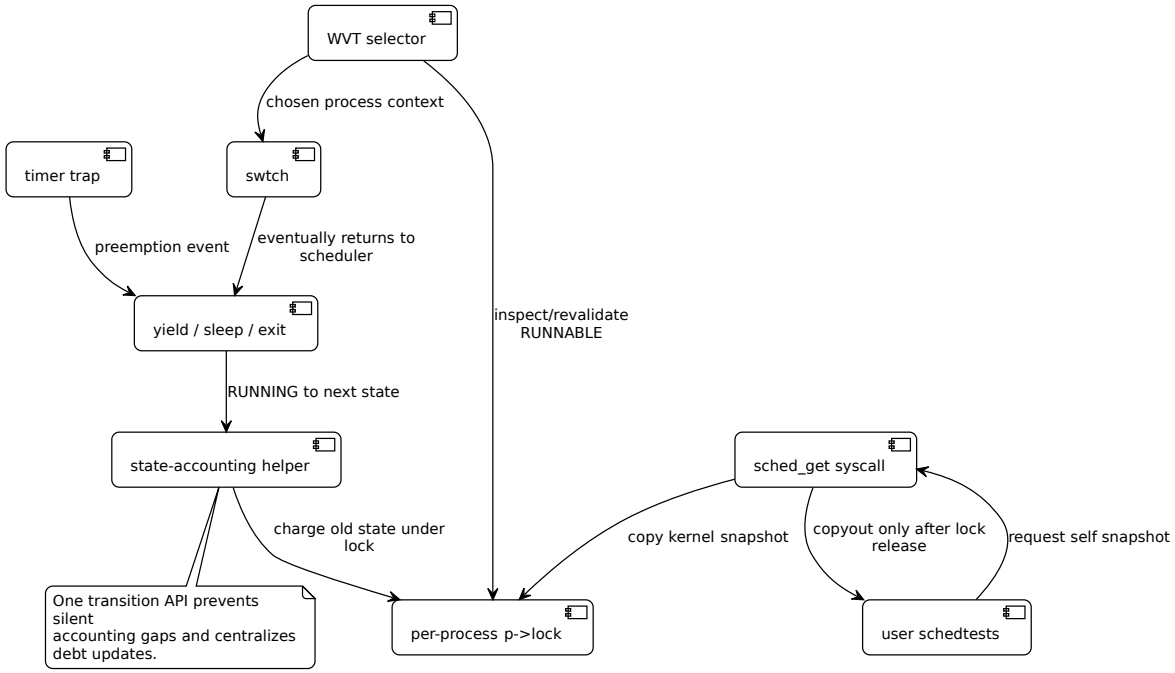
The only fixed value is the commit hash. Times and PID traces are environment observations and must not be invented. A missing or dirty baseline blocks causal comparison because later failures cannot be attributed to WVT. "Tests passed" without preserved output and configuration is incomplete evidence.

Baseline mechanism map

Read, annotate, and cite these files in the memo:

File	Question to answer
`kernel/proc.h`	Which fields require `p->lock`, and what do trapframe/context own?
`kernel/proc.c`	Where does every state transition occur; where is the scheduler lock handoff?
`kernel/swtch.S`	Which registers cross a scheduler switch, and why exactly those?
`kernel/trap.c`	Which timer path calls `yield`, and where is the next timer requested?
`kernel/start.c`	Which time source and timer interval are configured?
`kernel/syscall.c`, `syscall.h`, `sysproc.c`	How are numbers, raw arguments, handlers, and checked copies connected?
`user/user.h`, `user/usys.pl`, `Makefile`	How does a new syscall become callable and a test binary enter the image?

Run a repository search for every assignment to process state. Classify each as creation, selection, yield/preemption, sleep, wakeup, exit, or reap. A transition-based accountant is wrong if one assignment bypasses it.



Trace the unmodified switch

Starting in user process A at a timer interrupt, list each lock/state/context cut until process B reaches user mode. Mark where A's runtime should be charged and where B's dispatch count should increase.

A traps to its kernel stack; the timer path eventually calls `yield`. `yield` acquires A's lock and changes A from `RUNNING` to `RUNNABLE`; the transition helper should charge elapsed A runtime here using the dispatch timestamp and old weight. `sched()` switches A's kernel context to the per-CPU scheduler while A's lock participates in the handoff. On return to the scheduler, it clears current-process ownership and releases A's lock. The selector scans and revalidates B under B's lock, changes B `RUNNABLE→RUNNING`, increments B's dispatch count, records dispatch start, installs `c->proc`, and calls `swtch`. B resumes in its kernel continuation, releases its lock according to xv6's protocol, and eventually returns through its trapframe. Charging at timer entry alone misses voluntary block/exit intervals; incrementing dispatch after `swtch` returns charges the wrong time.

Contract: state accounting and user ABI

Use RISC-V's monotonic `time` counter as the lab's raw unit. Do not call the xv6 `ticks` lock while holding `p->lock`; the timer path can hold `tickslock` and wake processes, producing an inverse lock order. Reading `r_time()` avoids that dependency in this platform model.

Add per-process fields equivalent to:

```
uint64 state_since;   /* raw time at last state-accounted transition */
uint64 run_time;
uint64 ready_time;
uint64 sleep_time;
uint64 dispatches;
uint64 vruntime;
uint64 dispatch_start;
int weight;           /* exactly 1, 2, 4, or 8 */
```

Expose only current-process control and observation:

```
struct sched_sample {
    int pid;
    int state;
    int weight;
    uint64 sample_time;
    uint64 run_time, ready_time, sleep_time;
    uint64 dispatches, vruntime;
};

int sched_set_weight(int weight);
int sched_get(struct sched_sample *out);
```

`sched_set_weight` rejects all other values and updates elapsed `RUNNING` time under the old weight before changing it. `sched_get` acquires `p->lock` and performs exactly one `now = r_time()` read. Into a local kernel `sched_sample`, it sets `sample_time = now`, copies the stored counters, and adds `now-state_since` to the one counter selected by the current `RUNNING`, `RUNNABLE`, or `SLEEPING` state. It does not mutate the stored counters or `state_since`. It then releases the lock and performs rev5's four-argument `copyout`. The same `now` therefore anchors the reported state, all current-state accrual, and the sample timestamp. An invalid user destination returns `-1` and leaves the kernel usable.

The self-only API avoids inventing a credential model xv6 does not have. Children report samples to parents through ordinary pipes before exit.

Accounting invariant at two sample cuts

Let sample 1 and sample 2 come from the same process. If that process exists continuously between the cuts and every instant in the half-open interval belongs to exactly one of `RUNNING`, `RUNNABLE`, or `SLEEPING`, require

$$\Delta sample_time = \Delta run_time + \Delta ready_time + \Delta sleep_time.$$

This is an exact integer partition, not a statistical tolerance: each transition and each snapshot cuts the same raw `time` counter once, and snapshot work after that cut is outside the interval. The claim excludes intervals containing `USED`, `ZOMBIE`, reuse of a PID as a different process, or raw-counter wrap. `sample_time` is the guest RISC-V counter value, not host wall time or xv6 `ticks`. Keep raw integers in test output; decimal conversion can manufacture a residual that the kernel did not produce.

Milestone A — one transition helper

Create one helper whose precondition is "`p->lock` held." It reads time once, charges `now - state_since` to the old state, updates virtual runtime when leaving `RUNNING`, records the new state and timestamp, and performs new-state bookkeeping. Replace every relevant raw assignment or justify why construction/reap remains outside it.

Pseudocode obligation:

```
transition(p, new_state):
    assert p->lock held
    now = time()
    delta = now - p.state_since
    charge delta to p.old_state
    if old_state == RUNNING:
        p.vruntime += weighted(delta, p.weight) with overflow policy
    p.state = new_state
    p.state_since = now
```

Do not paste this pseudocode into C without handling initialization, same-state calls, overflow, exit, and fork creation placement.

COMPLETE THE COHERENT SNAPSHOT AND COPYOUT BOUNDARY IN SCHED_GET

```
uint64 sys_sched_get(void) {
  uint64 uaddr;
  argaddr(0, &uaddr);
  struct proc *p = myproc();
  struct sched_sample snap;

  acquire(&p->lock);
  uint64 now = r_time();
  snap.sample_time = now;
  snap.run_time = p->run_time + (p->state == RUNNING ? now - p->state_since : 0);
  release(&p->lock);

  if (copyout(p->pagetable, uaddr, (char *)&snap, sizeof(snap)) < 0)
      return -1;
  return 0;
}
```

Holding `p->lock` while reading `r_time()` once guarantees that `p->state`, `p->state_since`, and the stored accumulators share a single linearization cut. Releasing `p->lock` before `copyout` prevents holding a spin lock across a fallible page-table walk or user-memory copy.

SIMULATE WVT VIRTUAL-RUNTIME CHARGING AND FORK FLOOR PLACEMENT

```
def wvt_charge(vruntime: int, delta_t: int, weight: int) -> int:
  return vruntime + max(1, delta_t // weight)

def fork_vruntime(parent_v: int, floor_v: int) -> int:
  return max(parent_v, floor_v)

procs = {
  "hog_w1": {"w": 1, "v": 0},
  "worker_w4": {"w": 4, "v": 0},
}
for tick in range(1, 5):
  for name, p in procs.items():
      p["v"] = wvt_charge(p["v"], delta_t=8, weight=p["w"])
  floor_v = min(p["v"] for p in procs.values())
  print(f"After quantum {tick}: hog_w1 v={procs['hog_w1']['v']}, worker_w4 v=
{procs['worker_w4']['v']}, floor={floor_v}")

print("New child from sleeping parent (v=0) placed at:", fork_vruntime(0, floor_v))
```

Milestone B — checked ABI

Wire declarations and numbers through all kernel/user layers; add a tiny test that changes each legal weight, rejects `0,3,16,-1`, and snapshots itself. Take two samples around known live-state activity and verify monotonic `sample_time` plus the exact three-state partition. Pass an invalid pointer and prove a subsequent `getpid` / `fork` still works. Inspect the disassembly or generated syscall stub once; a header declaration alone does not create an ABI.

The dangerous operation is not reading the stats; it is copying them across a fallible boundary while owning scheduler state.

Verified correct behavior: legal weights round-trip; illegal values return `-1` without mutation; `sample_time` and current-state accrual derive from one `r_time()` read; paired live-state samples satisfy the exact partition; invalid `copyout` returns `-1`; no process lock remains held; baseline tests still pass. A kernel that reads time separately for each field, validates only the start address, or writes the user pointer directly fails the boundary contract. The implementation remains intentionally undisclosed.

Contract: WVT selection, creation, and wakeup

DEFINITION

The WVT policy includes selection, debt charging, creation placement, wakeup placement, tie behavior, and overflow; omitting any one leaves observable behavior undefined.

Continuously runnable selection

On `CPUS=1`, choose the minimum `vruntime` among `RUNNABLE` processes, then PID. Revalidate state and ordering assumptions while holding the candidate's lock before changing it to `RUNNING`. Record a dispatch before `swtch`, not after it returns.

On `CPUS>1`, exact global minimum is not required: concurrent selectors observe a moving runnable set. Safety is required. No process may be selected by two harts; no stale candidate may be switched after it ceases to be runnable. Document the lock/revalidation protocol and quantify that fairness results are single-hart only.

Fork creation placement

A new child inherits the parent's weight. At the creation cut, first materialize the parent's elapsed `RUNNING` charge under the old weight. Then place the child at exactly the greater of that charged parent value and the current scheduler floor:

$$v_{child} = \max(v_{parent}, floor).$$

Neither `v_child = 0` nor `v_child = v_parent` without the floor clamp satisfies the contract. The floor term matters when a parent sleeps in `wait` while a sequence of children consumes CPU: the sleeping parent's own debt no longer tracks that descendant work, but the global service frontier advances. Placement at $\max(v_{parent}, floor)$ prevents each new child from re-entering behind that frontier. It still does not provide group fairness when several descendants are runnable at once. State the lock order used to read/update parent debt, read the floor, and initialize the locked child.

Wakeup placement

Maintain a nondecreasing scheduler floor representing recently selected virtual time. When a process becomes runnable after sleeping, preserve its old v but clamp excessive credit:

$$v \leftarrow \max(v,\ \max(0, floor - C)),$$

where C is a documented wakeup-credit bound, initially two baseline timer intervals in virtual-time units. A short sleeper can receive bounded latency credit; a process asleep for minutes cannot return at ancient virtual time and monopolize the CPU. All floor access needs a lock order with no reverse path.

Overflow

Choose and defend one strategy:

- renormalize all live virtual runtimes by subtracting a common floor before the 64-bit danger region, under a quiescent/single-hart test protocol; or
- use a proven modular comparison and bound that guarantees no pair differs by 2^{63} or more.

"It takes a long time" is not an integer-correctness argument. A test-only hook must place values near the boundary without waiting years.

Milestone C — policy and creation placement

Implement minimum selection, exact tie behavior, weight charging, inherited weight, and child creation at $v_{child} = \max(v_{parent}, floor)$ after charging the parent to the creation cut. Implement floor maintenance and a documented lock order for that cut. Keep a compile-time baseline policy if useful for controlled comparison; do not let two policy implementations silently use different accounting.

Milestone D — bounded wakeup and overflow

Add wakeup clamping and the test-only overflow path. Record why the chosen C is a policy value, not a theorem. Run all functional tests before collecting performance data.

Public tests that kill named wrong implementations

Every test prints one line beginning `SCHEDTEST name PASS` or `... FAIL` and exits nonzero on failure. Calibrate tolerances for statistical service/latency tests from at least 10 baseline repetitions; do not widen them until a known-bad implementation passes. W6 is not statistical: its in-scope integer residual must be zero.

> **WARNING**
>
> A test that passes only the intended implementation but has never rejected its target defect has not demonstrated discrimination.

ID	Named wrong implementation	Test construction	Required discriminator
W1 `FIRST_SLOT`	restarts at the first runnable table slot after every dispatch	two equal CPU-bound children created in reverse slot/PID patterns	both receive service; ordering does not pin one indefinitely
W2 `WEIGHT_IGNORED`	selects by v but charges raw time equally for all weights	weight-1 and weight-4 CPU loops, barrier start, fixed wall window, CPUS=1	median run-time ratio in a justified band around 4; swapped weights swap share
W3 `FORK_BELOW_FRONTIER`	child starts at zero, or copies only stale parent debt below the floor	persistent competitor versus a sequential fork/wait chain with at most one attacker descendant runnable; run both known-bad variants	both known-bad builds fail; the competitor retains a substantial near-half share under the correct $\max(v_{parent}, floor)$ placement; fork rate does not buy unbounded service
W4 `ANCIENT_WAKEUP`	sleeper returns with unbounded old credit	CPU hog plus repeated long-sleep/short-run task	wakeup latency remains bounded **and** post-wakeup burst does not monopolize multiple quanta
W5 `TICK_ONLY_CHARGE`	charges only timer-preempted work	one task repeatedly yields after calibrated sub-tick work; one CPU-bound peer	yielding task cannot obtain arbitrarily more useful work for equal weight
W6 `STATE_GAP`	one raw state assignment bypasses accounting, or snapshot fields use different clock reads	force yield, timed sleep, and wake paths between paired self-snapshots	`sample_time` is monotonic and $\Delta sample_time - \Delta run_time - \Delta ready_time - \Delta sleep_time = 0$ for every in-scope pair
W7 `LOCKED_COPYOUT`	holds `p->lock` across user copy	call `sched_get` with an invalid/unmapped pointer, then run fork/yield tests	returns error without deadlock, leaked lock, or later scheduler stall
W8 `STALE_CANDIDATE`	switches a candidate after unlocked scan without recheck	CPUS=3 scheduler stress plus debug `on_cpu` invariant	no double-running assertion, stack-canary fault, or non-runnable dispatch
W9 `WRAP_COMPARE`	unsigned minimum reverses after wrap	test-only virtual runtimes straddle the chosen boundary	selected order matches documented normalization/modular proof

The weight-share test needs two directions. A hard-coded preference for one PID can accidentally produce a 4:1 ratio once; swapping weights and creation order kills that counterfeit.

DEFEATING THE SEQUENTIAL FORK/WAIT STARVATION ATTACK (W3)

Process P1 (weight 1) sleeps in `wait()` at `vruntime = 10` while spawning a chain of short-lived children C1, C2, ..., C50 that each run for one quantum and exit before the next is forked. Meanwhile, a long-running competitor P2 (weight 1) has advanced the scheduler floor to `vruntime = 400`.

What happens if each new child initializes `v_child = v_parent` without clamping to the scheduler floor?

Reference decision: Every new child Ck starts at `vruntime = 10`, far behind P2's `vruntime = 400`, allowing the sequential fork/wait chain to monopolize the CPU and starve P2 until the attacker stops forking

While P1 sleeps in `wait()`, P1 is not `RUNNING` and therefore accumulates no virtual-runtime debt for the CPU time consumed by its children. Without clamping `v_child = max(v_parent, floor)`, every newly forked child inherits P1's ancient `vruntime = 10` and preempts P2 (`vruntime = 400`) indefinitely.

Test oracle details

Barrier start. Parent creates report pipes, forks children, then releases both through a start pipe so fork/setup time is outside the measurement. Each child sets its own weight, waits for the token, burns deterministic integer work until a common `uptime` deadline, snapshots itself, and writes the sample.

Useful-work calibration. Report both scheduler `run_time` and loop iterations guarded against optimization. A share claim needs agreement between service accounting and useful work; divergence exposes instrumentation or cache/compiler effects.

Snapshot cut. `sched_get` must return the one raw `sample_time` used to accrue the current state. For each pair, assert identity continuity and record both sampled states. Apply the exact conservation oracle only when the process remained inside the three live states; do not hide a nonzero residual behind timing tolerance. Invalid-pointer tests are separate because no sample is returned.

Latency distribution. The sleeper records elapsed time around each bounded sleep and subtracts the requested sleep interval under the lab clock model. Report median, p95, maximum, and raw samples. A mean alone does not kill one extreme wakeup.

Safety assertion. Under `SCHED_TEST`, maintain `on_cpu` under `p->lock`: assert zero before `RUNNABLE→RUNNING`, set to hart ID plus one, clear only when scheduler regains ownership. The assertion is instrumentation, not a substitute for the lock proof.

Test-design checkpoint

Why does a single 4:1 run-time observation fail to distinguish WVT from a scheduler that gives PID 2 four quanta for every one quantum of any other PID?

The observed ratio may be caused by PID bias, not weight. Swap weights between the same processes, reverse creation order, and repeat with a third equal-weight competitor. WVT predicts service follows weight under the controlled window; PID bias follows identity/slot. A discriminating test changes one causal variable while controlling alternatives.

Failure injection matrix

Run these after each milestone, not only at the end.

WARNING

Failure injection is evaluated at the first invariant violation and recovery behavior, not by whether the kernel eventually prints a panic.

Injection	Expected behavior	Forbidden behavior
`sched_get((void *)MAXVA)` or equivalent invalid destination	`-1`; process and scheduler continue	panic, lock leak, partial kernel-memory exposure
illegal weight during active CPU use	reject atomically; old weight/debt retained	partial counter reset or free priority
exhaust process slots with repeated `fork`	failures returned; existing tasks continue; no half-initialized runnable slot	crash, leaked lock/page, child placed below $\max(v_{parent}, floor)$
kill a sleeping process near wakeup	process becomes runnable/terminates according to xv6 policy; accounting closes exactly once	permanent sleep, double charge, double run
force virtual-runtime boundary with `SCHED_TEST`	documented ordering survives	wrap to permanent priority or starvation
CPUS=3 rapid fork/yield/sleep	baseline safety tests pass; no `on_cpu` assertion	exact-share claim without evidence, stale dispatch

For every observed failure, save the seed/configuration, first invariant violation, smallest reproducer, and corrected test. Do not patch from the final panic line without reconstructing the earlier state cut.

Milestone E — adversarial suite

Demonstrate each public test against the intended implementation and at least four deliberately injected wrong variants, including W3, W4, W7, and W8. A test suite that has never failed is unvalidated.

Measurement protocol

Functional correctness precedes claims about fairness or performance.

Controlled factors

Record commit, host CPU, host load policy, QEMU version, compiler, optimization flags, `CPUS`, timer interval, workload seed, warmup, run duration, and repetitions. Run policy variants in randomized or counterbalanced order. Keep console logging outside measured windows; scheduler tracing can dominate the scheduler.

Required workloads

1. **Equal CPU:** two and eight continuously runnable weight-1 jobs.

2. **Weighted CPU:** weights 1:2, 1:4, and 1:8; repeat with identity/order swapped.
3. **Interactive:** one CPU hog plus sleepers with two sleep/run duty cycles.
4. **Fork adversary:** persistent job plus a sequential fork/wait chain; compare correct $\max(v_{parent}, floor)$ placement with zero-start and parent-only known-bad variants.
5. **Mixed:** CPU, sleeper, and short-lived process creation together.
6. **SMP safety:** CPUS=3 stress; no exact ratio claim.

Required metrics

- per-process `sample_time`, run, ready, and sleep time from each coherent cut;
- dispatches and virtual-runtime trajectory sampled at bounded frequency;
- useful-work count;
- response/wakeup median, p95, and maximum;
- service-share error $|observed - target|$ over at least three windows;
- wall completion and context switches per unit of useful work; and
- exact live-state residual $\Delta sample_time - \Delta run_time - \Delta ready_time - \Delta sleep_time$, retained as a signed integer.

For weights w_i, target share is $w_i / \sum_j w_j$ only while the measured set remains continuously runnable. Remove sleeping intervals or model them; otherwise the denominator is false.

Causal comparison

Compare pinned baseline and WVT using the **same accounting instrumentation**. If WVT improves weighted-share error but increases dispatches or scan cost, report both. If response improves only because the timer interval changed, the scheduler-policy claim is confounded. If useful work falls while run time is unchanged, inspect cache/branch/compiler effects before blaming context switching.

Milestone F — results and uncertainty

Provide raw observations and a plot/table that retains distributions. For each major result write:

```
claim → model prediction → measurement → uncertainty → competing
explanation → follow-up
```

“WVT is fair” is not a result. “With two continuously runnable processes at weights 1:4, CPUS=1, 12 randomized 200-tick runs produced median high-weight share X with range Y; swapped identity changed it by Z” is auditable.

Design memo

The memo must contain:

> REMARK
>
> The memo is a compact proof and evidence index; source narration without invariants, counterexamples, and measurements does not satisfy it.

1. **Baseline and scope:** exact revision/toolchain; single-hart fairness boundary; unchanged upstream behavior.
2. **State and invariants:** field ownership, state-transition API, lock order, selection linearization/revalidation.
3. **Policy:** charging equation, tie rule, fork placement at $\max(v_{parent}, floor)$, wakeup clamp C, weight-change semantics, overflow strategy.
4. **ABI:** argument validation, single-read `sample_time` cut, exact live-state conservation, copyout ordering, failure behavior.
5. **Test discrimination:** W1–W9 mapping, known-bad demonstrations, statistical-tolerance derivation, and the zero-tolerance W6 oracle.
6. **Measurements:** workload controls, raw-data link/path, distributions, overhead, anomalies.
7. **Failure analysis:** at least one injected defect from symptom to first invariant violation.
8. **Limitations:** QEMU, time source, coarse preemption, cache effects, SMP fairness, missing production features.

Every design claim cites source lines by file/function or primary reference. Do not paste large source fragments.

Diagnostic analysis questions

Answer without running the code, then use traces to check the answers.

1. Why is `dispatches++` immediately after `swtch` returns attributed to the wrong event?

2. A process changes weight from 1 to 8 halfway through a long uninterrupted run. Which weight charges the first half?
3. Why is child `vruntime=parent.vruntime` insufficient when the parent waits for each short-lived child, and what placement closes that reset path?
4. What behavior changes as wakeup credit C moves from 0 to unbounded?
5. Why can exact global minimum be ill-defined for concurrent CPU selectors?
6. Which lock order could deadlock if accounting reads `ticks` while holding `p->lock`?
7. How does W5 exploit timer-only charging, and which measurement exposes it?
8. Why must `sched_get` use one time read under `p->lock`, then copy a local snapshot after releasing the lock?
9. What does the single-hart SJF-style minimum scan cost asymptotically in `NPROC`, and which measurement could reveal it?
10. Which result would falsify the claim that observed 1:4 share follows weight rather than PID or creation order?

1. Return occurs when the process later yields back; the dispatch happened before entering it.
2. Charge elapsed time to the change point under weight 1, reset the timestamp, then charge subsequent time under 8.
3. The waiting parent's virtual runtime stops advancing while each child consumes service, so every new child can inherit stale debt. Initialize the child at `max(parent.vruntime, floor)` to keep it at the current service frontier. Multiple simultaneous descendants still create multiple schedulable entities; group scheduling needs a hierarchy.
4. Zero removes latency credit and may penalize sleepers; unbounded lets ancient virtual time dominate after wakeup. A finite C bounds both effects.
5. Runnable state and virtual time change while selectors operate; only a serialized global run queue defines one total decision point. Safety can remain exact with approximate concurrent policy.
6. Timer code can hold `tickslock` then acquire process locks during wakeup; `p->lock→tickslock` creates the reverse cycle.
7. Yield before each timer tick to consume CPU without virtual debt. Compare useful-work and `r_time`-based service against virtual-runtime increments.
8. One time read gives the state and all accrued counters a common linearization cut, making the three-state conservation equation exact. User memory access can fail and may invoke mechanisms forbidden while holding scheduler state; a local kernel copy preserves that cut without extending lock lifetime through `copyout`.
9. A full scan is $O(NPROC)$ per selection. Measure scheduler CPU time or useful throughput as runnable count grows while holding work constant.
10. Swapping weights while holding PID/order fixed fails to swap observed share, or reversing creation changes share more than weight does.

Verification checklist and defect conditions

> **WARNING**
>
> Safety blockers cap the result regardless of throughput because a fast double-running or lock-leaking scheduler is not an acceptable scheduler.

Dimension	Invariant criterion	Defect condition
Baseline reproducibility and source map	Clean diff against pinned commit `7d7adbb1…`	Wrong/unrecorded commit or broken baseline
State/accounting invariants	Conservation of sample time across all process states	Bypassed transition, double charge, missing/split-read `sample_time`, or nonzero in-scope residual
WVT policy completeness	Correct debt updates, floor tracking, bounded wakeup	Child below $\max(v_{parent}, floor)$, no wakeup bound, undefined ties/overflow
Locking and SMP safety	Total ordering across locks without deadlock	Double-running, stale dispatch, or lock-order deadlock
ABI and failure behavior	Safe parameter validation and unheld copyout	Unchecked pointer, lock-held copy-out, partial mutation
Test discrimination	Falsification of flawed mutants W1–W9	Tests do not kill W3/W4/W7/W8, statistical tolerances are post-hoc, or W6 tolerates a nonzero residual
Measurements and causal interpretation	Reproducible trial outputs and controlled sweeps	No raw data, confounded policy comparison, invented values
Architectural understanding	Complete derivation of state and lock invariants	Cannot explain own state/lock protocol

A verified implementation must satisfy all invariant criteria with zero defect conditions. Performance throughput cannot compensate for a safety or isolation defect. Every diagnostic question should be falsifiable against execution traces.

Handoff checklist

REMARK

Every checked item must name retained evidence; an unchecked limitation remains explicit rather than being converted into a claim.

Required lab evidence checklist:

Engineering artifact	Verification criterion	Required evidence artifact
Pinned patch diff	Formatted against pinned commit `7d7adbb1…`	Git diff file with zero unrelated changes
Upstream test suite	Passes full regression suite on CPUS=1 and CPUS=3	Raw console logs with zero assertion failures
Discriminator tests	Passes all W1–W9 tests and rejects flawed builds	Output logs demonstrating rejection of W1–W9 mutants
W3 placement check	Rejects both `v_child=0` and un-floored placement	Trace confirming $v_{child} = \max(v_{parent}, floor)$
Accounting invariant	Monotonic single-read sample time	Verified log where $\Delta\text{sample} - \sum \Delta\text{states} = 0$
Clean release binary	Zero debug prints and hooks in production build	Binary symbol check confirming hooks compiled out

REMARK

The implementation may be correct inside this contract while remaining unsuitable for production; the memo must preserve that distinction.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 2.4.1: Monotonic State Accounting and Time Sampling

Verification application

Explain the sampling rule that prevents a process from having its runtime attributed to multiple states or missed entirely during a state transition in xv6. State the algebraic invariant linking delta sample time to state accumulators.

Sample the hardware monotonic timer once at state transition and charge delta time atomically.

The accounting invariant requires: $\Delta \text{sample} - \sum \Delta \text{states} = 0$. At every state transition (e.g., inside `sleep()`, `yield()`, or `exit()`), the kernel reads the hardware monotonic clock exactly once into a local variable t_{now}. The elapsed interval $\Delta t = t_{\text{now}} - p \to \text{last_sample}$ is calculated and credited to the old state accumulator before updating $p \to \text{state}$ and setting $p \to \text{last_sample} = t_{\text{now}}$. Re-reading the hardware timer twice or outside locks creates non-monotonic intervals or double-counts time.

Exercise 2.4.2: Checked User-Space System Call ABI for Priority Weights

Construction application

Design the system call `sys_setweight(int weight)` validating user-supplied scheduling weights against the course policy $[1, 16]$. State the error handling and lock acquisition order required.

Check bounds before acquiring the process lock to prevent lock holding on error.

1. Read integer argument using `argint(0, &weight)`.
2. Validate bounds: if `weight < 1 || weight > 16`, return `-1` with `EINVAL` without modifying any state.
3. Acquire `p->lock` protecting the calling process struct.
4. Update `p->weight = weight`.
5. Release `p->lock` and return 0 for success.

Exercise 2.4.3: Child Fork Placement at Max of Parent and Queue Floor

Formalisation transfer

In Weighted Virtual Time (WVT), explain why initializing a newly forked child with $v_{\text{child}} = 0$ enables a fork-bomb denial-of-service attack, and prove why setting $v_{\text{child}} = \max(v_{\text{parent}}, \text{queue_floor})$ preserves fairness.

Analyze how zero virtual runtime grants newly created processes an unfair monopoly over the CPU.

If a child is initialized with $v = 0$ while existing processes have accumulated virtual runtime $v \gg 0$, the scheduler strictly selects the process with the smallest $vruntime$. A parent process can repeatedly call `fork()`, generating an infinite stream of children with $v = 0$ that starve all pre-existing processes on the system. Initializing $v_{\text{child}} = \max(v_{\text{parent}}, \text{queue_floor})$ ensures the child cannot gain an unfair head-start over either its parent or the currently active runnable cohort.

Exercise 2.4.4: Sleeping Wakeup Placement and Integer Wraparound Invariants

Construction transfer

A process sleeps on a long I/O operation while competing processes accumulate millions of virtual runtime ticks. Explain the starvation hazard upon wakeup if its $vruntime$ is unchanged, and detail how clamping $v \leftarrow \max(v, \text{queue_floor} - \text{latency_target})$ protects against starvation while bounding interactive latency.

Prevent a sleeping process from waking up with an ancient virtual runtime that monopolizes the CPU for seconds.

If a process sleeps for hours, its stored $vruntime$ remains frozen at an ancient small value. When it wakes, it would monopolize the CPU for millions of consecutive ticks before competing processes ever get scheduled, causing massive starvation. Conversely, setting $v = \text{queue_floor}$ erases all interactive credit. Clamping $v \leftarrow \max(v, \text{queue_floor} - \Delta_{\text{bonus}})$ allows the interactive task to preempt and run briefly ahead of batch tasks while strictly bounding its unfair monopolization.

Exercise 2.4.5: Adversarial Test Suite Rejecting Broken Schedulers (W1–W9)

Verification synthesis

Describe the design of an automated test oracle that differentiates a correct WVT implementation from mutant W3 (child initialized with $v = 0$) and mutant W5 (weights multiplied instead of divided).

Construct workloads whose output completion ratios differ quantitatively under mutant logic.

1. Mutant W3 detector: Launch a long-running batch process that runs for 500 ticks so its $vruntime$ reaches hundreds. Then, fork a series of short child jobs. Under correct placement ($v = \max(v_{\text{parent}}, floor)$), the children and parent share CPU time proportionally. Under Mutant W3, the children starve the parent completely until their runtimes catch up. The test measures parent progress during child execution and asserts parent makes non-zero forward progress.
2. Mutant W5 detector: Run two CPU-bound tasks with weights $w_A = 1$ and $w_B = 4$. Under correct division, task B receives approximately $4\times$ the CPU time of task A. Under Mutant W5 (multiplication), task A accumulates debt slower and receives $4\times$ the CPU time of task B. The test measures ticks elapsed and asserts $\text{ticks}(B) > 3 \times \text{ticks}(A)$.

References & Further Reading

1. MIT PDOS, official xv6 RISC-V rev5 tag and license.
2. MIT PDOS, rev5 `kernel/proc.c`, `kernel/proc.h`, `kernel/swtch.S`, `kernel/trap.c`, and `kernel/start.c`.
3. MIT PDOS, *xv6: a simple, Unix-like teaching operating system*, Chapters 4 and 7.
4. RISC-V International, *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*, time counter, timer, and supervisor trap architecture.
5. RISC-V International, RISC-V ABIs Specification, register preservation and syscall calling-convention context (CC BY 4.0).
6. Linux kernel documentation, EEVDF Scheduler, used only to distinguish the teaching policy from current production virtual-time scheduling.

Concurrency and liveness

SECTION DETAIL

Interleavings and Atomicity

Two workers dequeue jobs from a shared ring. The queue contains one job. Each worker reads `count == 1`, each reads the same head slot, and each decrements `count`. Both return the same job; one real job is executed twice. No instruction failed. The failure exists only in the order formed by individually ordinary instructions.

Concurrency turns **time** into program input. Correctness must therefore quantify over every execution the scheduler and hardware may permit, not only the order seen in one debugger run.

Freeze the execution model before tracing

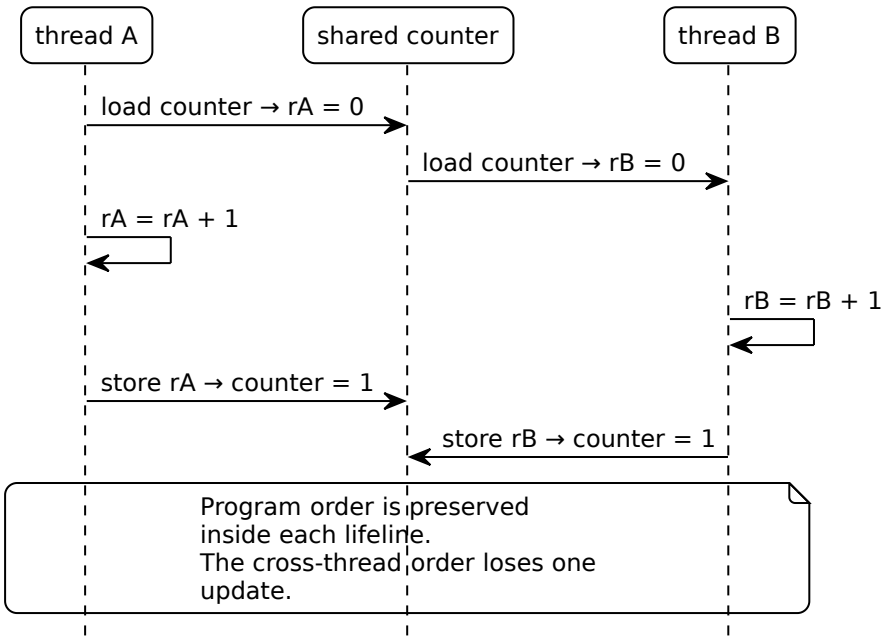
DEFINITION 3.1.1 History and program order

A **history** is a finite sequence of invocation, internal, and response events from one or more threads. For each thread T, **program order** $\prec_T$ orders that thread's events as the abstract machine requires. An **interleaving** is a total order of events that preserves every $\prec_T$.

This lesson begins with a deliberately strong machine model:

- one processor instruction completes before the next event in the history;
- every load sees the most recent preceding store to that location in the total order;
- a context switch may occur between the primitive events shown in the trace; and
- no compiler or processor reorders those events.

That is a **sequentially consistent trace model**, not a claim about C or RISC-V execution. It isolates the scheduling problem. The next lesson adds language-level data-race rules and weak hardware ordering.



Retrieval — answer before expanding. If thread A has three ordered events and thread B has three ordered events, how many six-event interleavings preserve both threads' program order?

Choose which three of the six positions belong to A; B occupies the rest. The count is

$$\binom{6}{3} = \frac{6!}{3!3!} = 20.$$

This counts schedules, not distinct final states. Several schedules may produce the same state, and synchronization can make many schedules observationally equivalent.

For event sequences of lengths m and n, the same argument gives $\binom{m+n}{m}$. With three threads, the multinomial coefficient grows quickly. Exhaustive reasoning is still practical for a reduced state machine; it is not practical if every source statement is treated as one indivisible event without justification.

An increment is a protocol, not a glyph

Suppose `counter` initially equals 0 and each thread evaluates `counter++`. In the trace model, expand the expression into load, local add, and store:

Event	Thread A state	Thread B state	Shared `counter`
initial	`rA = –`	`rB = –`	0
A loads	`rA = 0`	`rB = –`	0
B loads	`rA = 0`	`rB = 0`	0
A adds	`rA = 1`	`rB = 0`	0
B adds	`rA = 1`	`rB = 1`	0
A stores	`rA = 1`	`rB = 1`	1
B stores	`rA = 1`	`rB = 1`	1

The intended postcondition is `counter == 2`. The first five events leave a recoverable state: a remaining store might still produce 2 only if it were derived from 1. The final store commits a stale value and makes the intended postcondition unreachable in this history. That store is the first event at which the completed-operation count and shared counter cannot be reconciled.

Prediction — do not move the slider yet. In the lost-update schedule, after both loads and both local additions but before either store, is the invariant already false? State which invariant you are using.

FIND THE LOST UPDATE IN AN INTERLEAVING

Thread A

```
rA = counter;
rA = rA + 1;
counter = rA;
atomic_fetch_add(&counter, 1);
```

Register rA: **1**

Thread B

```
rB = counter;
rB = rB + 1;
counter = rB;
atomic_fetch_add(&counter, 1);
```

Register rB: **1**

SHARED MEMORY

1

counter

Invariant fails: one increment was lost

The unsafe schedule finishes both increments with counter = 1 because each thread stores a value derived from the same stale read.

Transfer. Change the initial counter to 9 and replace A's increment with `counter += 2` while B still adds 1. Construct a legal six-event history ending in 10, one ending in 11, and the serialized history ending in 12.

- Final 10: A and B both load 9; A computes 11; B computes 10; A stores 11; B stores 10.
- Final 11: use the same stale reads but let B store 10 before A stores 11.
- Final 12: A completes load/add/store first, then B loads 11 and stores 12; or B completes first and A subsequently loads 10 and stores 12.

The first answer to the prediction depends on the invariant. `counter == completed increments` still holds before either operation responds because neither increment has completed. A stronger implementation invariant such as "each local pending update was derived from a distinct logical version" is already violated when both loads observe version 0. State which layer of correctness the invariant describes.

Name the failure precisely

Term	Required evidence	What it does not imply
race condition	correctness depends on relative timing or ordering	that two accesses target the same byte
atomicity violation	a multi-event operation is observed in a state inconsistent with its required indivisible effect	that the implementation language declares undefined behavior
ISO C data race	conflicting actions in different threads, at least one non-atomic, without a happens-before relation	that the only possible symptom is a lost update
determinism failure	the specified observable result varies across allowed executions	that the result is incorrect; nondeterminism may be intentional

In ISO C, a data race has undefined behavior. The sequentially consistent lost-update table is therefore a useful abstract diagnosis, not a list of the only outcomes a real racy C program may exhibit. The compiler is not required to preserve an intuitive interleaving for a program outside the language contract.

`volatile` does not repair the program. In C it constrains accesses to volatile objects for implementation-defined observable behavior such as device registers; it neither combines load/add/store into one atomic action nor creates inter-thread synchronization. A `volatile int`

`counter` can still lose updates, and a C data race on it still has undefined behavior.

Application. Classify each case and justify the minimum claim:

1. Two threads use an atomic fetch-add; the final count is correct, but which thread receives ticket 0 varies.
2. Two C threads write a plain `int flag` without synchronization, even though both write 1.
3. A thread checks `access(path, W_OK)` and later opens the pathname after another process replaces the directory entry.

1. The result may be nondeterministic without an atomicity violation or data race. Each fetch-add can still have one indivisible place in the order.
2. The writes conflict, are non-atomic, and lack happens-before; this is an ISO C data race even if a particular machine appears to store the same value twice.
3. This is a time-of-check/time-of-use race. The pathname can name a different object at use time. The relevant repair is an interface that binds checking and use to one kernel object or directory-relative handle—not a mutex private to one process.

CONCURRENT ACCESS CLASSIFICATION MATRIX

Thread T loads a shared counter with atomic_load while Thread U concurrently executes atomic_fetch_add on the same variable. Evaluate each classification:

Answer: Each atomic access executes indivisibly at a single linearization point, producing a legally serialized history.; Reading either the pre-increment or post-increment value reflects valid concurrency nondeterminism, not an error.

Atomic accesses do not race under ISO C, and each atomic operation occupies a single indivisible point in time. Observing either pre- or post-increment values is legitimate nondeterminism.

Invariants turn schedules into proof obligations

An invariant is a predicate that must hold at designated observable states. It is stronger than "the test passed" and more useful than "the code is thread-safe."

For a bounded queue with capacity N, useful clauses include:

$$0 \le \mathit{count} \le N, \qquad \mathit{tail} \equiv \mathit{head} + \mathit{count} \pmod{N},$$

plus ownership clauses stating which slots contain initialized jobs and when a returned job stops belonging to the queue. The arithmetic catches overflow and duplicate removal; the ownership clause catches returning a pointer whose slot another producer may overwrite.

Worked trace: duplicate dequeue. Let `head = 4`, `count = 1`, and slot 4 contain job J. Workers A and B each perform `if (count > 0) { j = ring[head]; head++; count--; }` without synchronization.

1. A and B both test `count > 0`.
2. Both read slot 4 and obtain J.
3. A updates `head = 5`, `count = 0`.
4. B uses stale local values and also publishes `head = 5`, `count = 0`.

The numeric invariant still holds. The abstraction invariant fails because one abstract queue element produced two successful dequeue responses. A correct invariant must connect representation state to operation history, not merely bound integers.

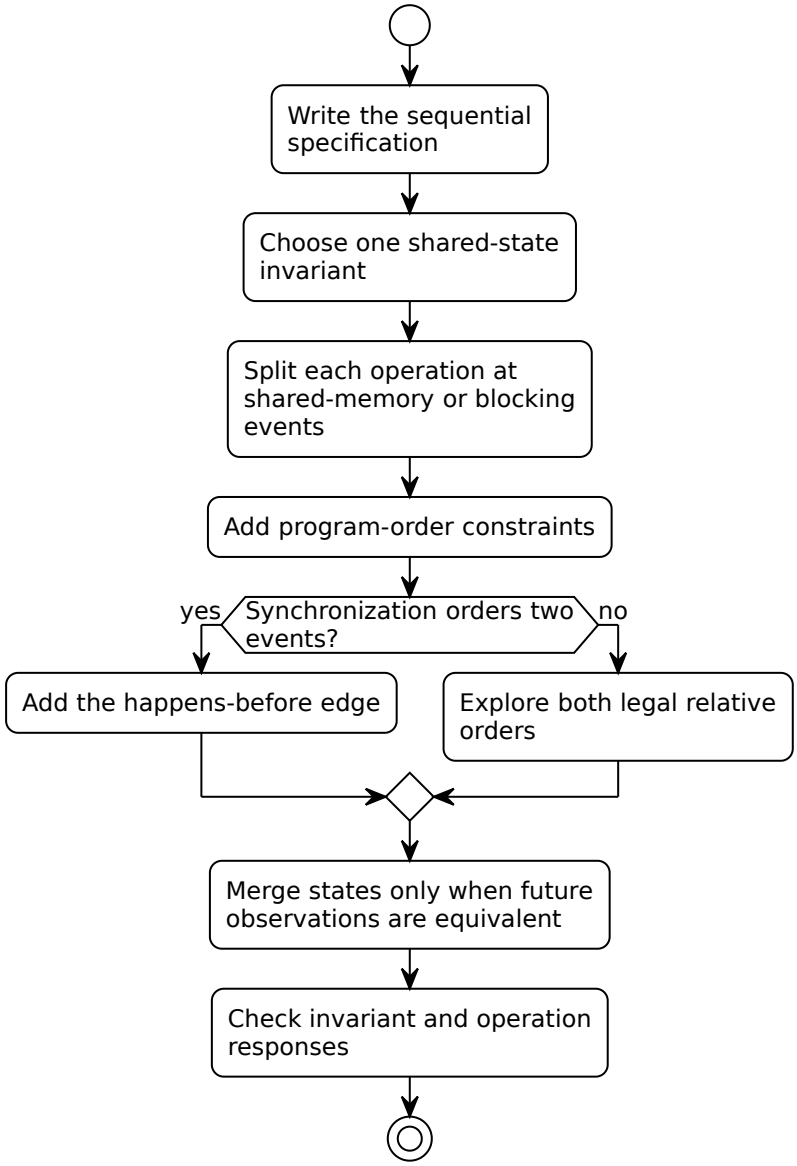
> **DEFINITION 3.1.2** **Linearizability**
>
> A concurrent history is **linearizable** when each completed operation can be assigned one point between its invocation and response so that the resulting sequential order (1) obeys the object's sequential specification and (2) preserves real-time order for non-overlapping operations.

The assigned point is a **linearization point**. For an atomic fetch-add, it is the successful atomic read-modify-write. For a lock-protected enqueue, it may be the pointer or count update that makes the new item reachable while the lock excludes observers. "Somewhere inside the function" is not enough: name an event and prove every allowed observer sees a history consistent with it.

Counterexample: the response is not always the point. A queue operation may link a node, then spend time reclaiming metadata before returning. A concurrent dequeue can observe the node during that interval. Linearizing enqueue at its response would put dequeue before enqueue in the

sequential order even though dequeue returned the new node. The publication store—not the later return—is the candidate point.

Reduce the state space without deleting behavior



Treat local arithmetic with no shared observation as one event. Split at loads, stores, atomic operations, lock operations, waits, wakeups, and externally visible responses. This reduction is valid only when the merged local steps cannot block, fault in a relevant way, call unknown code, or expose state through a signal or callback.

Worked search. Two one-slot exchanger operations each execute `publish(value)` then `take(peer)`. If `publish` and `take` are atomic abstract events, there are six interleavings of two two-event threads. Four schedules let both publish before either take and can succeed. Two schedules attempt a take before the peer publication. Whether those block, fail, or read "empty" is part of the sequential specification; deleting that choice from the model would assume the result.

Random stress can discover a violating schedule but cannot establish that no schedule exists. A useful concurrency test harness controls one uncertainty at a time:

- barriers place threads at a chosen boundary;
- a deterministic scheduler selects the next event;
- an invariant checker runs after every event in the reduced model;
- a history recorder captures invocation and response order; and
- a test oracle checks whether the observed history admits a legal sequentialization.

Dynamic race detectors answer a different question: whether an execution contains unsynchronized conflicting memory accesses under their model. They can miss unexecuted races and cannot prove a higher-level queue or filesystem protocol correct.

> REMARK
>
> A concurrency mechanism is justified only by the bad histories it excludes and the progress histories it preserves. Naming a primitive is not a correctness argument.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 3.1.1: Counting Legal Interleavings for Concurrent Instructions

Verification application

Two threads each execute an instruction sequence: Thread 1 executes m sequential instructions, and Thread 2 executes n sequential instructions. Derive the total number of distinct interleaved instruction execution histories, and evaluate this count for $m = 3$ and $n = 3$.

Recall combinations of m steps chosen from m + n total positions.

Any valid interleaving is a sequence of length $m + n$ where m steps belong to Thread 1 and n steps belong to Thread 2, preserving their respective intra-thread program orders. The total number of distinct interleavings is given by the binomial coefficient:

$$\binom{m+n}{m} = \frac{(m+n)!}{m!\,n!}.$$

For $m = 3$ and $n = 3$:

$$\binom{6}{3} = \frac{6 \times 5 \times 4}{3 \times 2 \times 1} = 20 \text{ distinct interleavings.}$$

Even for very short code sequences, testing a few arbitrary thread orders explores only a small fraction of the possible interleaving space.

Exercise 3.1.2: Lost-Update History for Non-Atomic Fetch-Add

Construction application

Two threads concurrently execute `counter++` on a shared integer initialized to 0. In assembly, each increment consists of: `L: load r, [counter]`; `A: add r, r, 1`; `S: store [counter], r`. Construct an interleaving schedule of these 6 operations that produces a final counter value of 1 instead of 2.

Interleave the loads so both threads read the initial value 0 before either stores.

Schedule:

1. $T1$: `L` $\implies r1 = 0$.
2. $T2$: `L` $\implies r2 = 0$.
3. $T1$: `A` $\implies r1 = 1$.
4. $T1$: `S` $\implies \text{counter} = 1$.
5. $T2$: `A` $\implies r2 = 1$.
6. $T2$: `S` $\implies \text{counter} = 1$. Both threads read the initial value 0. $T2$'s store overwrites $T1$'s update with the identical value 1, resulting in a classic lost update anomaly.

Exercise 3.1.3: Check-Then-Act Race Condition in Naive Lock-Free Allocator

Description transfer

An engineer replaces a mutex-protected free list with atomic operations: `if (atomic_load(&head) != NULL) { block = atomic_load(&head); atomic_store(&head, block->next); return block; }`. Demonstrate an interleaving where two concurrent threads allocate and return the exact same memory block.

Trace two threads executing the atomic loads concurrently before either performs the atomic store.

Assume the free list contains two blocks: $X \to Y \to \text{NULL}$.

1. Thread A executes `atomic_load(&head)`: observes $X \neq \text{NULL}$.
2. Thread B executes `atomic_load(&head)`: observes $X \neq \text{NULL}$.
3. Thread A executes `block_A = atomic_load(&head)`: receives pointer to X.
4. Thread B executes `block_B = atomic_load(&head)`: receives pointer to X.
5. Thread A executes `atomic_store(&head, block_A->next)`: updates `head` to Y and returns X.
6. Thread B executes `atomic_store(&head, block_B->next)`: updates `head` to Y and returns X. Both threads return block X simultaneously, violating allocator uniqueness and corrupting application memory. Making individual loads and stores atomic does not make the compound multi-step protocol atomic.

Exercise 3.1.4: Time-of-Check to Time-of-Use Pathname Races

Verification transfer

A setuid root utility checks file permissions: `if (access(path, W_OK) == 0) { fd = open(path, O_WRONLY); write(fd, data, len); }`. Construct an adversarial race condition that allows an unprivileged user to overwrite `/etc/passwd`.

Exploit the time gap between access() and open() by substituting a symlink.

The time gap between `access()` (check) and `open()` (act) creates a Time-of-Check to Time-of-Use (TOCTOU) vulnerability.

1. The unprivileged user creates a harmless temporary file `/tmp/target` that they own.
2. The user runs the privileged program passing `path = "/tmp/target"`.
3. The kernel executes `access("/tmp/target", W_OK)`. Because the user owns the file, `access()` returns 0 (permission granted).
4. Before `open()` executes, the user process rapidly replaces `/tmp/target` with a symbolic link: `unlink("/tmp/target"); symlink("/etc/passwd", "/tmp/target");`.
5. The privileged utility calls `open("/tmp/target", O_WRONLY)`. The kernel follows the symlink with root privileges, opening `/etc/passwd` and truncating or overwriting critical system data. Repair: Avoid path-based check-then-act sequences; use file descriptor handles and `open(..., O_NOFOLLOW | O_CREAT)` with dropped privileges.

Exercise 3.1.5: Designing Compare-and-Swap Loops with Linearization Points

Construction synthesis

Design a correct concurrent stack `push()` operation using an atomic compare-and-swap primitive (`atomic_compare_exchange_weak`). Identify the exact linearization point of the operation and explain how failure retries preserve safety.

Snapshot the current top pointer and retry if another thread modified it before the swap.

```c
struct Node {
    void *data;
    struct Node *next;
};

void push(atomic_ptr *top, struct Node *new_node) {
    struct Node *old_top = atomic_load(top);
    do {
        new_node->next = old_top;
    } while (!atomic_compare_exchange_weak(top, &old_top,
new_node));
}
```

Linearization point: The single clock cycle where `atomic_compare_exchange_weak` succeeds. At that precise instant, `top` transitions from `old_top` to `new_node`, making `new_node` atomically visible to all concurrent push and pop operations. If another thread prepends a node in the interim, the CAS returns false, updates `old_top` with the new value, and repeats the loop without modifying shared memory, guaranteeing safety.

References & Further Reading

- Maurice Herlihy and Jeannette Wing, “Linearizability: A Correctness Condition for Concurrent Objects”, *ACM Transactions on Programming Languages and Systems*, 1990.
- ISO/IEC JTC1/SC22/WG14, N1570: ISO C11 committee draft, especially §§5.1.2.4 and 6.7.3.
- Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, concurrency chapters.

- MIT PDOS, *xv6: a simple, Unix-like teaching operating system*, Chapter 6/7 “Locking” in the linked revision.

SECTION DETAIL

Locks, Atomics, and Memory Ordering

A driver appends a request to a queue, then sets `ready = 1`. Another core observes `ready == 1` but follows an old queue tail and submits the wrong descriptor to hardware. No two cores wrote the same queue field at the same instant; the publication still failed because the consumer observed the flag without an ordering guarantee for the data it announced.

Mutual exclusion answers **who may mutate an invariant now**. Memory ordering answers **which earlier effects another observer must see**. An atomic instruction can contribute to both, but the words are not interchangeable.

Start from the invariant, not the lock variable

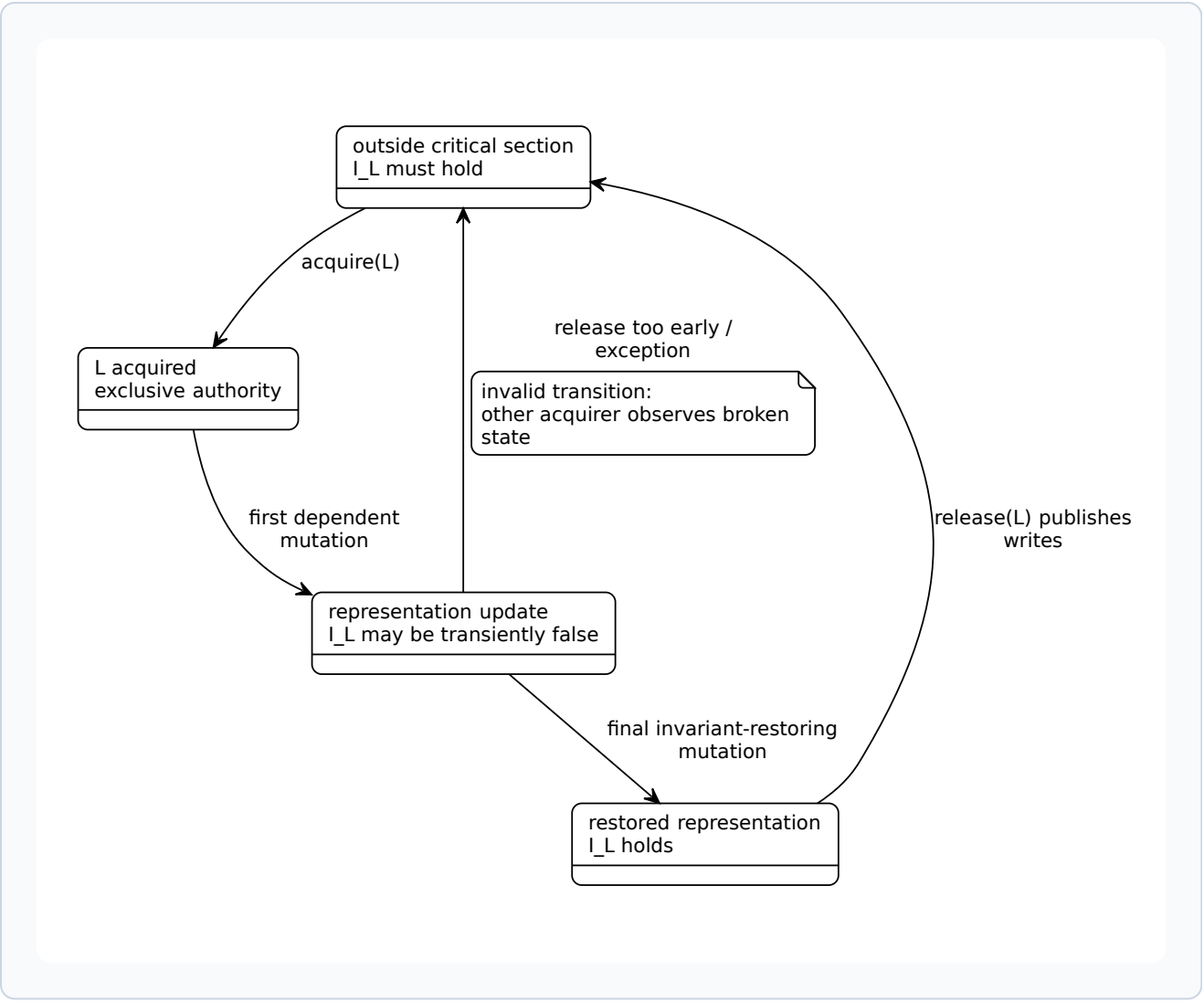
DEFINITION 3.2.1 Protected invariant

For a lock L, the **protected invariant** I_L is a predicate over shared state that may be temporarily false only while one thread owns L. Every acquisition may assume I_L; every release must publish a state satisfying I_L.

Consider a queue represented by `head`, `tail`, and `count` with capacity N:

$$0 \le count \le N, \qquad tail \equiv head + count \pmod{N}.$$

The mutex is not part of the abstract queue. It is a protocol granting temporary exclusive authority to violate and restore those equations. A field is "protected by L" only if **every** access that depends on the invariant follows that protocol. Locking writers while readers inspect fields unlocked is not protection.



Retrieval. Why is "only one thread executes this function" weaker than a queue correctness proof?

The function may call another path that accesses the same fields under a different lock, readers may bypass the function, callbacks may re-enter it, and an early return may release before restoring the representation. The proof must identify all shared state covered by I_L, all access paths, acquisition and release boundaries, and preservation of the abstraction—not one function name.

Worked trace and preservation derivation for one enqueue

Assume the pre-state satisfies $0 \le count < N$ and `tail == (head + count) mod N`. Under exclusive ownership:

1. write the item into slot `tail`;
2. set `tail' = (tail + 1) mod N`;
3. set `count' = count + 1`;
4. release only after the item initialization and both metadata updates are complete.

Then $0 < count' \leq N$, and

$$tail' \equiv tail + 1 \equiv head + count + 1 \equiv head + count' \pmod{N}.$$

The arithmetic proof is incomplete without an ownership clause: the new slot must contain a fully initialized item before another thread may observe the increased count. That publication obligation leads directly to memory ordering.

MEMORY PUBLICATION AND LOCK ORDERING MATRIX

A queue protects state with lock L. A producer initializes buffer[tail] and increments count while holding L. A consumer reads count without L. Evaluate each statement:

Answer: Without acquiring lock L, the consumer can observe updated count > 0 before buffer initialization stores arrive.; Acquire and release memory semantics establish visibility edges only between threads synchronizing on the same lock.

Acquire/release memory ordering establishes synchronization only across paired acquire and release operations on the same lock variable. An unsynchronized read has no acquire edge and can observe out-of-order writes on weakly ordered memory architectures.

A spin lock needs one indivisible winner

Two threads cannot implement a lock with plain “check then set”:

```
while locked == true:
    wait
locked = true
```

Both can read false before either store. A lock acquisition needs one atomic state transition that says, in effect, "change false to true and tell me whether I was the thread that changed it." Compare-and-exchange, atomic exchange, load-reserved/store-conditional, and architecture-specific atomic memory operations can supply that transition.

```
#include <stdatomic.h>
#include <stdbool.h>

typedef struct {
    atomic_flag held;
} spinlock;

void spin_lock(spinlock *lock) {
    while (atomic_flag_test_and_set_explicit(
        &lock->held, memory_order_acquire)) {
        /* architecture-specific pause/backoff may reduce contention */
    }
}

void spin_unlock(spinlock *lock) {
    atomic_flag_clear_explicit(&lock->held, memory_order_release);
}
```

This fragment establishes one owner and acquire/release ordering under the C abstract machine when the flag is initialized correctly. It does **not** promise FIFO admission, bounded waiting, reentrancy, interrupt safety, adaptive backoff, or good performance. A production lock also needs an ownership policy, debug behavior, and architecture/runtime integration.

Application — kill a plausible wrong lock. Replace `atomic_flag_test_and_set` with an atomic load followed by an atomic store. Construct a schedule with two owners.

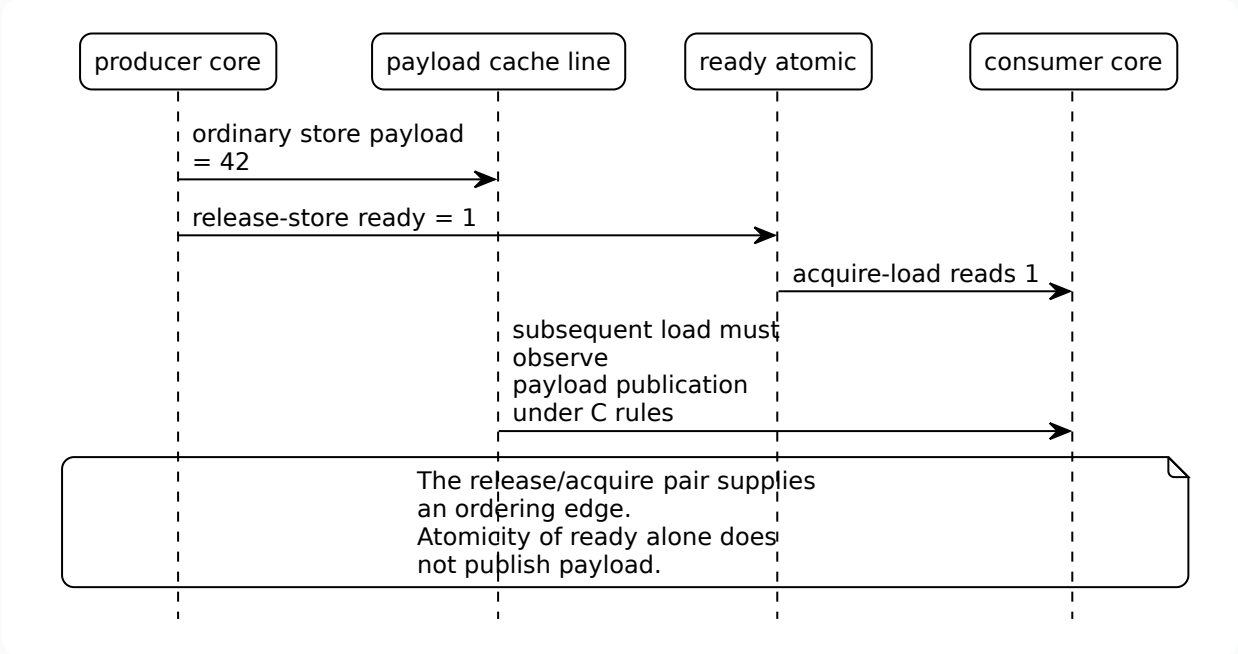
The atomicity of each access is insufficient:

1. A atomically loads `held == false`.
2. B atomically loads `held == false`.
3. A atomically stores true and enters.
4. B atomically stores true and enters.

The winner election must itself be one read-modify-write operation, or a compare-and-exchange loop whose failed contender retries from a fresh observation.

Acquire and release publish a relation

Coherence normally gives each atomic location a consistent modification order. It does not, by itself, require a core that observes a new flag to observe earlier writes to other locations. The language and ISA memory models define the required cross-location order.



In ISO C, the standard publication idiom is:

```c
#include <assert.h>
#include <stdatomic.h>

static int payload;
static atomic_int ready = 0;

void publish(void) {
    payload = 42;
    atomic_store_explicit(&ready, 1, memory_order_release);
}

void consume(void) {
    if (atomic_load_explicit(&ready, memory_order_acquire) == 1) {
        assert(payload == 42);
    }
}
```

If the acquire load reads the value from the release sequence, the release synchronizes with it. The payload store is sequenced before the release; the payload load is sequenced after the acquire; transitivity places the store before the load in the happens-before relation. The plain `payload` accesses therefore do not form a data race in that successful branch.

Replacing both operations on `ready` with `memory_order_relaxed` preserves atomic accesses to `ready` but removes the publication edge for `payload`. Relaxed order is suitable for a counter whose numeric value is independent, not for announcing that an object graph is initialized.

ESTABLISH AN ACQUIRE/RELEASE PUBLICATION EDGE IN C11

```
void publish(void) {
  payload = 42;
  atomic_store_explicit(&ready, 1, memory_order_release);
}

void consume(void) {
  if (atomic_load_explicit(&ready, memory_order_acquire) == 1) {
      assert(payload == 42);
  }
}
```

A release store on `ready` paired with an acquire load on `ready` creates a synchronizes-with edge when the load observes the stored value, placing `payload = 42` in happens-before order prior to `assert(payload == 42)`.

INTERRUPT-CONTEXT SELF-DEADLOCK ON A SINGLE HART

A kernel thread on hart 0 acquires spin lock `rx_lock` to dequeue a packet from a driver ring buffer. While `rx_lock` is held with local interrupts still enabled, the network interface raises an interrupt on hart 0. The interrupt handler runs on hart 0 and calls `spin_lock(&rx_lock)` to append the newly arrived descriptor.

What is the exact failure mode on hart 0, and which locking protocol prevents it?

Reference decision: Hart 0 spins forever in the interrupt handler because the interrupted thread holding `rx_lock` cannot resume on hart 0 until the handler returns; disable local interrupts on the hart before acquiring any spin lock shared with an interrupt handler

An interrupt handler preempts the current thread on the same hart and must run to completion before the interrupted context can execute another instruction. If the handler spins on a lock held by the interrupted thread on that hart, neither can progress. Any spin lock shared with an interrupt handler must mask local interrupts prior to acquisition (such as xv6 `push_off()` / Linux `spin_lock_irqsave()`).

Transfer. A statistics counter is incremented with relaxed atomics. A shutdown thread observes the expected count and then frees every worker's result object. Is the count observation enough to publish those objects?

No. A relaxed fetch-add makes the counter update atomic and gives that atomic object a modification order, but it does not make preceding ordinary writes to each result happen before the shutdown thread's reads or frees. Use a joining operation, a release/acquire completion protocol, a mutex/condition predicate, or another mechanism with the required lifetime and visibility guarantees. The exact repair depends on who owns each object and whether workers can still access it.

RISC-V's RVWMO permits observers on different harts to see memory operations in orders not implied by source order unless preserved-program-order rules, dependencies, acquire/release annotations, atomics, or fences constrain them. The A extension provides atomic read-modify-write

and load-reserved/store-conditional instructions. Kernel code should use the kernel's established primitives; hand-inserting `fence` instructions without connecting ISA semantics to compiler semantics is not a portable repair.

Spin, block, or redesign

DEFINITION 3.2.2 Lock acquisition cost model

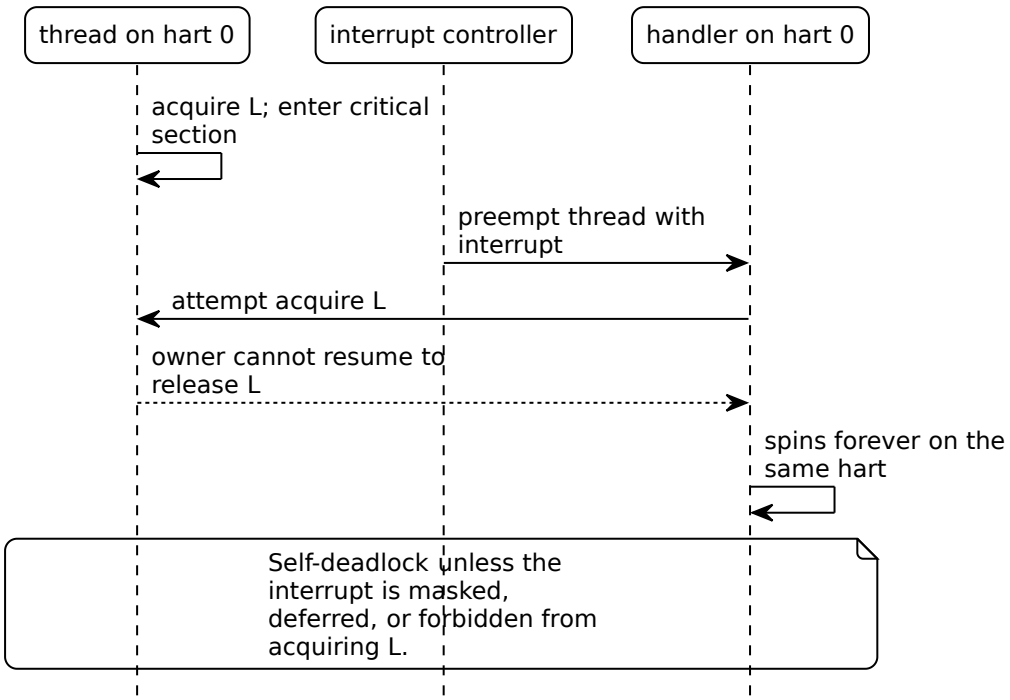
Let W be expected wait time, S the cost to deschedule and later resume, H the lock-hold time, and P the number of runnable contenders. Spinning is plausible only when W is reliably small, the owner can run, and consuming a CPU for W does not violate a stronger objective. Blocking is plausible when W may exceed S or CPU time must be available to other work.

Context	Candidate	Reason	Failure boundary
short non-sleeping kernel section on another running core	spin lock	avoids scheduler path	collapses under long hold or oversubscription
user thread waiting on I/O-protected state	blocking mutex	yields CPU while owner/device progresses	sleep/wakeup and priority policy add cost
read-mostly immutable snapshot	versioning, RCU, or copy-on-write	readers avoid a global owner	reclamation and update publication become harder
high-contention counter	per-CPU/per-shard state	removes one shared cache line	aggregation may be approximate or delayed

Worked cost comparison. A critical section normally holds a lock for 200 ns but once per 10,000 entries blocks for a 2 ms device operation. The mean hold time is roughly $0.9999(200\ \text{ns}) + 0.0001(2\ \text{ms}) \approx 400\ \text{ns}$, which looks small. A contender arriving during the rare device wait can still burn 2 ms of CPU. The bug is not the average; sleeping while holding a spin lock makes the tail unbounded relative to the intended mechanism. Move the blocking operation outside the section or change the synchronization design.

Spin locks also generate coherence traffic. A naïve exchange loop writes the lock line on every failed attempt, invalidating peer copies. Test-and-test-and-set first spins on read-only loads and attempts the write only when the lock appears free; queue locks arrange local spinning. These designs change traffic and fairness, not the protected invariant.

Kernel context is part of the lock type



If an interrupt handler on one hart can acquire `L`, ordinary code on that hart cannot hold `L` while the handler is allowed to preempt it. The handler would spin waiting for an owner that cannot resume. One design disables the relevant local interrupts before acquisition and restores the previous interrupt state after release. Another ensures the handler never takes that lock and defers work to a context that may block.

xv6's spin-lock path tracks interrupt-disable nesting for this reason. That is an xv6 mechanism under its own assumptions, not evidence that every lock acquisition should globally disable interrupts. Scope, nesting, non-maskable events, preemption, and architecture rules matter.

Counterexample: "one CPU means no race." A uniprocessor kernel can be preempted by an interrupt, fault, signal delivery path, or scheduler transition between shared-state operations. Disabling preemption may exclude thread interleavings on that CPU but not necessarily device DMA, non-maskable events, or another CPU. State exactly which actors the exclusion covers.

Granularity changes both proof and performance

EXAMPLE

Splitting one buffer-cache lock into bucket locks removes false conflicts between independent buckets but creates a two-lock protocol for moves and resize. Parallelism and proof complexity change together.

Suppose a buffer cache has 1,024 buckets. One global lock gives a simple invariant but serializes independent buckets. One lock per bucket permits parallel lookups, but moving an entry from bucket 3 to bucket 8 now touches two protected regions.

Worked two-lock update. Impose the order `min(bucket_id)` before `max(bucket_id)`. To move 8 → 3, acquire 3 then 8 despite the source being 8. Revalidate that the entry is still in bucket 8 after both locks are held; another thread may have changed it between the initial lookup and acquisition. Unlink, relink, restore both bucket invariants, then release in either documented order.

Fine-grained locks can introduce:

- a larger proof surface and lock-order graph;
- cross-region operations that need rollback or revalidation;
- false sharing when distinct lock words occupy one cache line;
- memory overhead and initialization/lifetime hazards; and
- starvation if a hot region repeatedly admits local contenders.

The right comparison measures throughput and tail latency against a named workload while retaining invariant checks. “More locks scales better” is not a theorem.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 3.2.1: Ring Buffer Invariant Derivation and Pointer Wraparound

Formalisation application

Consider a bounded FIFO queue of capacity N protected by mutex L. Formulate the mathematical invariants governing `head`, `tail`, and `count` modulo N. Show why updates to `count` and buffer entries require mutual exclusion.

Relate head, tail, and count with modulo arithmetic and state slot ownership boundaries.

The queue invariant I_L requires:

1. $0 \leq count \leq N$.
2. $tail \equiv (head + count) \pmod{N}$.
3. When $count > 0$, buffer slots at indices $(head + i) \pmod{N}$ for $0 \leq i < count$ contain valid initialized elements; all other $N - count$ slots are available for allocation. If a thread enqueues without holding L, concurrent readers can observe $count > 0$ before the buffer slot is written, or concurrent writers can compute the same `tail` offset and overwrite each other's data, corrupting the modulo relationship and lost elements.

Exercise 3.2.2: Test-and-Set Spinlock Implementation and Interrupt Disabling

Construction application

Write the C/assembly implementation of `acquire()` and `release()` for a basic spinlock using the atomic exchange primitive `__sync_lock_test_and_set`. Explain why kernel spinlocks must disable local CPU interrupts before spinning.

Trace the deadlock that occurs if an interrupt handler on the same CPU attempts to acquire the lock.

```c
void acquire(struct spinlock *lk) {
    push_off(); // disable local CPU interrupts
    if (holding(lk))
        panic("acquire");
    while (__sync_lock_test_and_set(&lk->locked, 1) != 0)
        ; // spin
    __sync_synchronize(); // memory fence (acquire)
    lk->cpu = mycpu();
}

void release(struct spinlock *lk) {
    if (!holding(lk))
        panic("release");
    lk->cpu = 0;
    __sync_synchronize(); // memory fence (release)
    __sync_lock_release(&lk->locked);
    pop_off(); // restore interrupt state
}
```

If a CPU core acquires lock L with interrupts enabled, a device or timer interrupt can fire on that same CPU. If the interrupt service routine attempts to acquire L, it will spin waiting for L to be released. However, because the interrupted thread holding L is frozen on the same CPU, the lock is never released, causing an instantaneous, unrecoverable deadlock.

Exercise 3.2.3: Message Passing Publication with Acquire-Release Fences

Description transfer

A producer thread initializes payload data in array `data[0..1023]` and sets `flag = 1`. A consumer thread spins waiting for `flag == 1` and then reads `data[0..1023]`. Describe how modern relaxed hardware can violate correctness without memory barriers, and state the exact placement of release and acquire barriers.

Explain store buffering and speculative load reordering across memory locations.

On weakly-ordered architectures (such as ARM or RISC-V RVWMO), memory stores to independent addresses may drain from the CPU write buffer out of program order. Without barriers, the store `flag = 1` may become globally visible in cache while stores to `data` are still buffered locally. Concurrently, the consumer CPU may speculatively load `data` before reading `flag == 1`. Placement:

- Producer: Store payload into `data`; execute a **release barrier** (`atomic_store_explicit(..., memory_order_release)` or `fence rw, w`); store `flag = 1`.
- Consumer: Load `flag` with an **acquire barrier** (`atomic_load_explicit(..., memory_order_acquire)` or `fence r, rw`); read payload from `data`. This guarantees that all stores preceding the release are visible to any thread that executes the matching acquire.

Exercise 3.2.4: Deadlock Hazard When Acquiring Held Spinlocks in Interrupt Handlers

Verification transfer

Identify three independent kernel operations that are strictly prohibited while holding a spinlock in an operating system kernel.

Consider operations that yield the CPU, sleep, allocate unbounded memory, or cross privilege modes.

1. Sleeping or blocking: A thread holding a spinlock must never call `sleep()`, wait on a condition variable, or perform blocking disk I/O. If the thread sleeps, other CPU cores spinning on that lock waste processor cycles waiting for a sleeping thread, and if a waiting thread runs on the same CPU, deadlocks occur.
2. Dereferencing unpinned user-space pointers: Accessing unpinned user memory can trigger a demand-paging page fault. If the page fault handler sleeps waiting for disk reads, it violates the rule against sleeping with spinlocks held.
3. Enabling interrupts without saving depth: Arbitrarily re-enabling interrupts while inside a nested spinlock critical section exposes the CPU to nested interrupt deadlocks.

Exercise 3.2.5: Replacing Contended Global Spinlocks with Sharded Table Structures

Construction synthesis

A multi-tenant database server runs on a 64-core system. A global process/connection lookup table protected by a single spinlock experiences severe cache-line bouncing and lock contention under high query rates. Propose a scalable architectural redesign.

Consider hash partitioning, fine-grained bucket locks, and read-copy-update mechanisms.

1. Hash sharding: Partition the single global table into K independent buckets (e.g., $K = 256$, where $K \geq \text{number of cores}$). Each bucket has its own dedicated spinlock and cache-aligned memory boundary to prevent false sharing.
2. Read-dominated optimization: If lookups vastly outnumber insertions/deletions, employ a Read-Copy-Update (RCU) or versioned pointer architecture. Readers traverse pointers lock-free without atomic operations or write invalidations. Writers allocate a new node copy, update pointers atomically with release semantics, and defer memory reclamation until a grace period passes.
3. Measurable validation: Benchmark throughput and 99th-percentile latency under uniform and skewed zipfian lookup distributions, measuring L1/L2 cache-miss invalidation rates with hardware performance counters.

References & Further Reading

- ISO/IEC JTC1/SC22/WG14, N1570: ISO C11 committee draft, §§5.1.2.4 and 7.17.
- RISC-V International, RVWMO Memory Consistency Model, Version 2.0.
- The Open Group, POSIX.1–2024 `pthread_mutex_lock`, `pthread_mutex_trylock`, and `pthread_mutex_unlock`.
- Linux kernel documentation, Linux kernel memory barriers.
- MIT PDOS, *xv6: a simple, Unix-like teaching operating system*, locking chapter.

SECTION DETAIL

Condition Variables and Semaphores

A consumer checks an empty queue, releases its mutex, and prepares to sleep. In that gap a producer enqueues work and sends a notification. No waiter is registered, so the notification changes nothing. The consumer then sleeps forever beside a non-empty queue.

Mutual exclusion protected the queue while code was running. It did not solve **waiting until a state becomes true**. That requires an atomic handoff from "I hold the mutex and the predicate is false" to "I am registered as a waiter and no longer hold the mutex."

A condition variable is not the condition

DEFINITION 3.3.1 **Condition predicate**

A **condition predicate** $P(S)$ is a Boolean expression over shared state S, evaluated while holding the mutex that protects every field on which P depends. A condition variable is a wait set used to block and make waiters eligible; it does not store the truth of P.

For a bounded queue, a consumer's predicate may be `count > 0`; a producer's may be `count < capacity`. The queue fields are durable program state. A notification is only a scheduling hint that some waiter should re-evaluate state.

```
int queue_take(queue *q, item *out) {
    int rc = pthread_mutex_lock(&q->mutex);
    if (rc != 0) return rc;

    while (q->count == 0 && !q->closed) {
        rc = pthread_cond_wait(&q->not_empty, &q->mutex);
        if (rc != 0) {
            pthread_mutex_unlock(&q->mutex);
            return rc;
        }
    }

    if (q->count == 0 && q->closed) {
        pthread_mutex_unlock(&q->mutex);
        return QUEUE_CLOSED;
    }

    *out = q->slots[q->head];
    q->head = (q->head + 1) % q->capacity;
    q->count -= 1;
    pthread_cond_signal(&q->not_full);
    return pthread_mutex_unlock(&q->mutex);
}
```

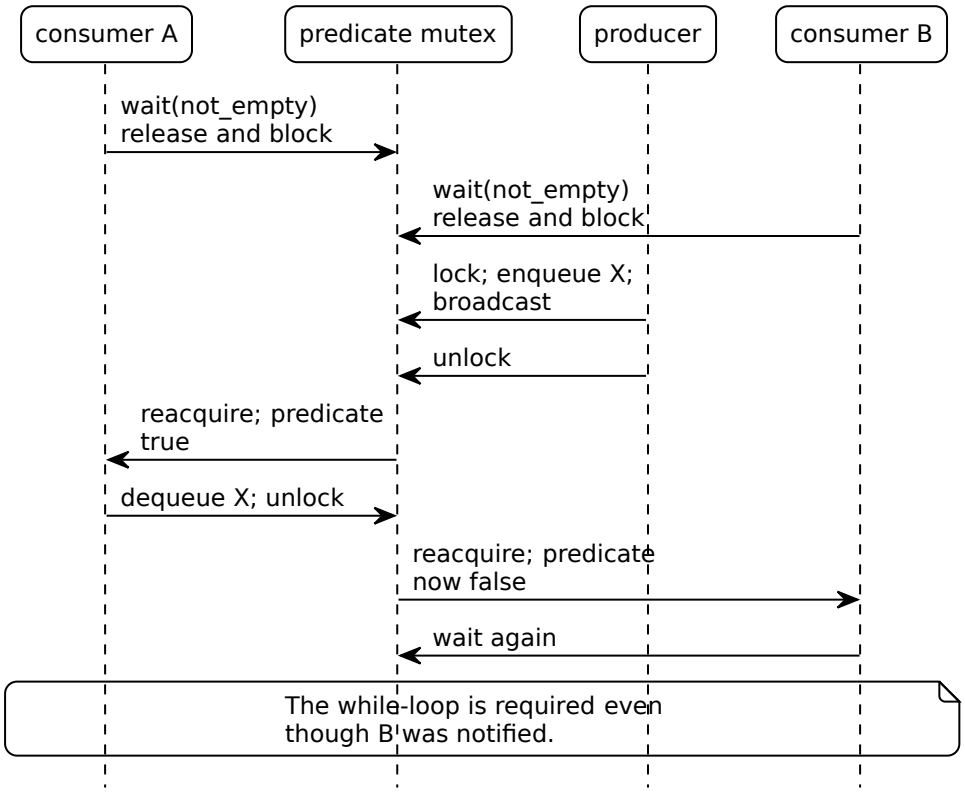
The wait operation atomically releases the mutex and blocks with respect to another thread acquiring that mutex and signaling the condition. Before returning, it reacquires the mutex. The loop re-evaluates the predicate after reacquisition.

Retrieval. Why does `pthread_cond_wait` require the caller to pass the mutex rather than asking the caller to unlock immediately before waiting?

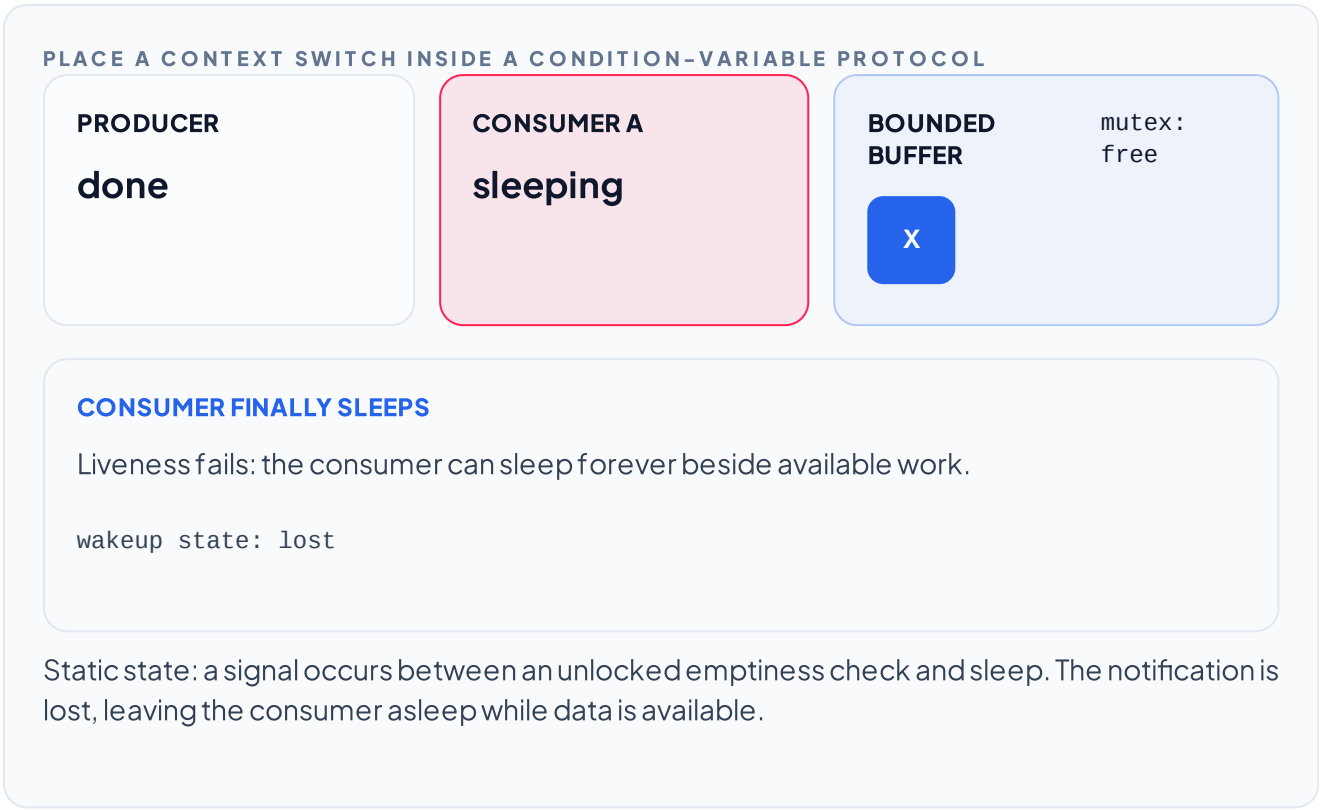
Separate unlock and wait operations create an observable gap. A producer can acquire the mutex, make the predicate true, signal when no waiter is registered, and finish before the consumer blocks. The combined wait protocol closes that gap: once another thread can acquire the predicate mutex, the waiter is treated as blocked for a subsequent signal.

Wakeup means “compete again,” not “proceed”

Most practical condition-variable APIs use **Mesa-style** semantics: signaling makes a waiter ready, but the signaler keeps running until it releases the mutex. By the time a waiter reacquires, another thread may have consumed the resource. Implementations may also permit wakeups even when no program-level event made the predicate true.



Prediction — fix an invariant in writing before moving the trace. In the broadcast schedule, which event makes an `if (count == 0) wait(...)` consumer unsafe: the broadcast, consumer A’s dequeue, or consumer B’s wakeup?



Transfer. The queue can be closed. A close operation sets `closed = true` and no further producer may enqueue. Should close signal one waiter or broadcast? What predicates do waiting producers and consumers use?

Close changes a state relevant to every waiter, so it normally broadcasts on each condition wait set. A consumer waits while `count == 0 && !closed`; after wakeup, `count > 0` permits draining existing work, while `count == 0 && closed` returns end-of-stream. A producer waits while `count == capacity && !closed`; any observation of `closed` rejects the enqueue. Both predicates and the close transition use the same queue mutex.

Consumer A's dequeue is the event that makes the predicate false before B reacquires. Broadcast is legal; B's wakeup is not permission to skip the predicate.

Signal one waiter when one unit of progress became possible. Broadcast when a transition can satisfy several different predicates, when all waiters must observe shutdown/configuration change, or when the implementation cannot identify the relevant waiter. Broadcast can create a **thundering herd**: many threads wake, contend for one mutex, discover false predicates, and sleep again. That is a performance failure, not permission to weaken the loop.

CONDITION VARIABLE MESA SEMANTICS MATRIX

Under Mesa semantics, a producer thread signals not_empty and releases the mutex. Evaluate each statement regarding why waiters must loop on wait():

Answer: Another thread can acquire the mutex and consume the resource before the woken waiter reacquires the lock.; Operating systems and threading runtimes may permit spurious wakeups even when no thread signaled the variable.

Under Mesa monitor semantics, signal() only transitions a waiter to runnable; it does not atomically hand off the CPU or lock. An intervening thread or spurious wakeup can invalidate the predicate before reacquisition.

Predicate updates define the notification protocol

WARNING

A notification cannot repair an update performed outside the predicate's synchronization protocol. First make the state transition visible under the mutex; then decide whom to wake.

The safe sequence is derived from state ownership:

1. lock the predicate mutex;
2. change the fields on which the predicate depends;
3. signal or broadcast as required by the transition;
4. unlock, publishing the new state to a waiter that later acquires.

POSIX does not make "signal only while holding the mutex" a universal syntactic law. Some programs can signal after unlocking without losing correctness because the predicate remains authoritative and all state transitions are synchronized. Holding the mutex through the state change and notification is easier to audit: the waiter cannot interleave between publication and notification, and the review can connect one transition to one wake decision.

Counterexample: notifications as remembered events. A thread calls `signal` before any waiter exists. A later waiter does not consume a stored signal; condition variables do not generally count notifications. If the application requires one future operation to consume one earlier permit, represent that fact in state—a counter, queue entry, generation number, latch, or semaphore.

Application: generation wait. A configuration service increments `generation` after each reload. A worker holds `seen`. Its predicate is `generation != seen || shutting_down`; after waking it copies the current generation to `seen`. A bare boolean `changed` can merge several reloads, which may be correct for "refresh to latest" but wrong if each reload requires one audit action. The state representation decides the semantics.

Timeouts add another state transition. A timed wait returning does not establish that the predicate is false or true; reacquire the mutex and test it. When elapsed duration matters, use a monotonic clock-capable API so wall-clock adjustment does not change the deadline semantics.

Semaphores store permits

DEFINITION 3.3.2 **Counting semaphore**

A counting semaphore has a nonnegative logical permit count. `wait` / `P` consumes one permit or blocks until one can be consumed. `post` / `V` adds one permit and may wake a waiter. The operation that changes the count and chooses whether to block is atomic with respect to other semaphore operations.

A semaphore post before a wait is remembered as a permit. That makes a semaphore suitable for a pool of interchangeable resources or queued units of availability. A condition variable is suitable when a thread must re-evaluate an arbitrary predicate over separately protected state.

Mechanism	Persistent state	Ownership on release	Natural question
mutex	locked/unlocked plus owner	owner releases	who may mutate this invariant?
condition variable	no application truth value	none	when should waiters recheck $P(S)$?
counting semaphore	permit count	usually no mutex-style owner	how many units may proceed?
event/generation counter	monotonically changing value	application-defined	has state advanced since my observation?

A binary semaphore can have values 0/1, but that does not automatically make it a mutex. Mutex ownership enables errors such as non-owner unlock to be rejected or defined; mutex protocols often carry priority inheritance and memory synchronization semantics. A semaphore can be posted by a different thread and often represents handoff rather than ownership.

Derive the bounded-buffer semaphore protocol

Let capacity be N. Initialize:

```
empty = N      // permits to reserve an empty slot
full  = 0      // permits to reserve a filled slot
mutex = 1      // exclusive queue metadata access
```

Producer:

```
P(empty)
P(mutex)
enqueue(item)
V(mutex)
V(full)
```

Consumer:

```
P(full)
P(mutex)
item = dequeue()
V(mutex)
V(empty)
```

Taking `empty` before `mutex` prevents a producer from sleeping for capacity while holding the metadata lock needed by a consumer. The consumer follows the symmetric order.

At quiescent points, `empty + full == N`. During concurrent operations that equality omits reservations. A stronger accounting invariant is

$$empty + full + R_p + R_c = N,$$

where R_p counts producers that consumed an empty permit but have not posted a full permit, and R_c counts consumers that consumed a full permit but have not posted an empty permit. Queue occupancy and permit counts are related by the protocol phase, not by one unconditional two-variable equation.

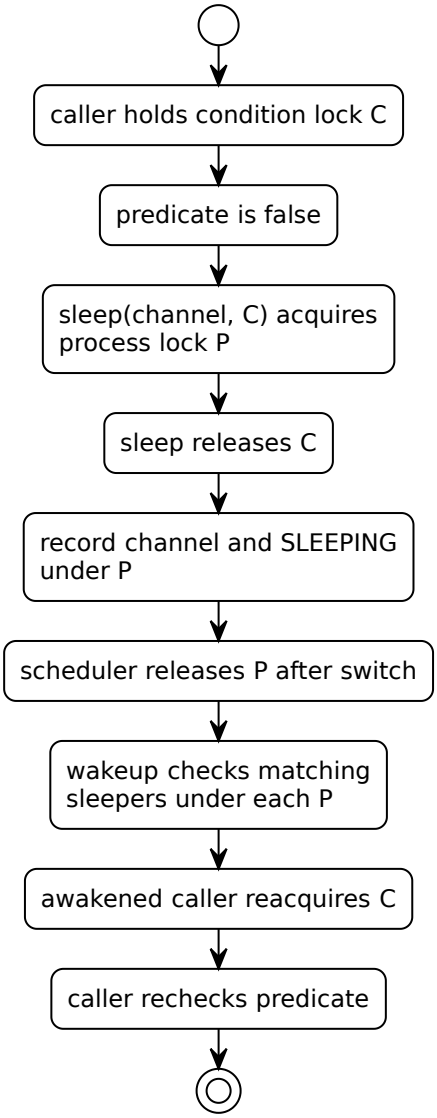
Worked trace, $N = 2$. Start `empty=2, full=0`. Producers A and B each perform `P(empty)`, so `empty=0, R_p=2`. A acquires `mutex`, enqueues X, releases, and posts `full`: `full=1, R_p=1`. A consumer consumes that full permit (`full=0, R_c=1`) before B has enqueued. It may dequeue only after acquiring `mutex`; B's reserved slot does not count as a filled item. The mutex and phase ordering preserve representation safety while permits limit capacity.

Counterexample: take the mutex first. With a full buffer, a producer acquires `mutex` then blocks in `P(empty)`. A consumer can consume a `full` permit but blocks trying to acquire `mutex`, so it cannot dequeue and post `empty`. The system deadlocks. This schedule is one reason resource-acquisition order belongs in the protocol proof.

xv6 sleep and wakeup expose the same handoff

xv6 kernel code waits on a **channel**, an arbitrary identifier associated with a predicate. `sleep(chan, lock)` acquires the process lock before releasing the caller's condition lock, records the channel and sleeping state, and enters the scheduler. The overlapping lock ownership prevents a concurrent `wakeup(chan)` from passing between predicate inspection and visible sleep registration.

The channel is not the predicate and wakeup is not a stored permit. Callers inspect their condition in a loop under the condition lock. `wakeup` makes matching sleepers runnable; by the time one runs, another process may have changed the condition.



Holding C while entering `sleep` forever would block the producer that needs C to make the predicate true. Releasing C before the sleep protocol owns P would reopen the lost-wakeup gap. The handoff must ensure the waiter owns one lock or the other across the vulnerable interval.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 3.3.1: Lost-Wakeup Race Condition When Releasing Mutex Before Sleeping

Description application

Explain why an implementation of `cond_wait(cv, mutex)` that executes `mutex_unlock(mutex)` and then separately executes `thread_sleep(cv)` possesses a fatal lost-wakeup race condition. Trace the exact interleaving with a concurrent signaler.

Trace what happens if a signal arrives between mutex_unlock and thread_sleep.

If unlocking and sleeping are not atomic:

1. Waiter acquires `mutex`, tests condition `count == 0`, and decides to sleep.
2. Waiter executes `mutex_unlock(mutex)`.
3. Before the waiter executes `thread_sleep(cv)`, a context switch occurs.
4. Producer acquires `mutex`, inserts an item into the queue, executes `cond_signal(cv)`, and releases `mutex`.
5. Because the waiter is not yet enqueued on `cv`'s wait set, the signal is lost into the void.
6. Waiter resumes and executes `thread_sleep(cv)`, blocking indefinitely even though the queue is non-empty. `cond_wait` must atomically release the mutex and enroll the thread into the wait queue.

Exercise 3.3.2: Mesa Semantics and Spurious Wakeup Invariants

Verification application

Contrast Hoare monitor semantics with Mesa monitor semantics. Explain why Mesa semantics require condition wait calls to be placed inside a `while (!predicate)` loop rather than an `if (!predicate)` block.

Analyze whether signaling hands off lock ownership immediately or merely marks the waiter runnable.

Under Hoare semantics, signaling immediately transfers the monitor lock and CPU execution directly to the awakened waiter; thus, the predicate is guaranteed to remain true when the waiter resumes. Under Mesa semantics, `cond_signal` merely transitions a waiting thread from sleeping to runnable; the signaler retains the lock until it explicitly releases it. Between the signal and the moment the awakened waiter reacquires the lock, a third intervening thread can execute, acquire the lock, and invalidate the predicate. Furthermore, POSIX permits spurious wakeups (e.g., due to kernel interrupt signals). Wrapping `pthread_cond_wait` inside a `while` loop ensures the predicate is perpetually verified under mutual exclusion before proceeding.

Exercise 3.3.3: Bounded Buffer Synchronization Using Counting Semaphores

Construction transfer

Write the pseudocode for a concurrent bounded FIFO queue of capacity N using counting semaphores `empty_slots`, `full_items`, and a binary semaphore `mutex`. Explain why `sem_wait(&empty_slots)` must precede `sem_wait(&mutex)`.

Inverting the semaphore acquisition order creates an instant deadlock when the queue is full.

```
semaphore empty_slots = N;
semaphore full_items = 0;
semaphore mutex = 1;

void enqueue(item):
    sem_wait(&empty_slots);  // wait for space
    sem_wait(&mutex);        // acquire buffer access
    buffer[tail] = item;
    tail = (tail + 1) % N;
    sem_post(&mutex);
    sem_post(&full_items);

item dequeue():
    sem_wait(&full_items);   // wait for an item
    sem_wait(&mutex);        // acquire buffer access
    item = buffer[head];
    head = (head + 1) % N;
    sem_post(&mutex);
    sem_post(&empty_slots);
    return item;
```

If a producer executes `sem_wait(&mutex)` before `sem_wait(&empty_slots)` when the buffer is completely full (`empty_slots == 0`), the producer acquires `mutex` and then blocks on `empty_slots`. A consumer attempting to dequeue blocks immediately on `mutex`, creating an instant deadlock where the consumer cannot free a slot and the producer will never release the mutex.

Exercise 3.3.4: Thundering Herd Performance Degradation vs Targeted Signaling

Description transfer

Explain the performance phenomenon known as the “thundering herd” problem when `pthread_cond_broadcast()` is invoked on a condition variable with 100 waiting worker threads, only one of which can successfully claim the newly available work item.

Trace the lock contention and context-switching overhead among the 99 unsuccessful waiters.

When a producer broadcasts on a condition variable with 100 waiting threads, the kernel wakes all 100 threads, placing them on the runnable queue. However, all 100 threads immediately contend for the single mutex protecting the queue. One winner acquires the lock, drains the work item, and releases the lock. The remaining 99 threads sequentially acquire the lock, recheck their `while (count == 0)` loop, observe the predicate is false, and re-enter `cond_wait()`. This results in 99 redundant context switches, severe CPU cache-line bouncing, and high scheduling latency with zero forward progress. Targeted `cond_signal()` should be preferred when exactly one waiter can make progress.

Exercise 3.3.5: Designing Graceful Queue Shutdown and Waiter Cancellation

Construction synthesis

Design a thread-safe multi-producer multi-consumer work queue supporting a `close()` operation that gracefully unblocks all current and future waiting consumers with an `END_OF_STREAM` status without leaking memory or dropping enqueued items.

Add a closed boolean flag under the mutex and broadcast to all waiters upon shutdown.

1. Shared state: Add `bool closed = false` under the queue mutex, alongside `cond_var not_empty` and `cond_var not_full`.
2. Enqueue: Producers wait `while (count == N && !closed)`. If `closed`, enqueue immediately returns `-ESHUTDOWN`.
3. Dequeue: Consumers wait `while (count == 0 && !closed)`. Upon waking:
   - If `count > 0`: dequeue item and return success (draining pre-existing work).
   - If `count == 0 && closed`: return `-EPIPE` / `END_OF_STREAM`.
4. Close operation:

```
void queue_close(struct Queue *q) {
    pthread_mutex_lock(&q->mutex);
    q->closed = true;
    pthread_cond_broadcast(&q->not_empty);
    pthread_cond_broadcast(&q->not_full);
    pthread_mutex_unlock(&q->mutex);
}
```

   Broadcasting ensures every blocked consumer and producer awakens, observes `closed == true`, and exits without blocking forever.

References & Further Reading

- The Open Group, POSIX.1–2024 `pthread_cond_wait`, `pthread_cond_timedwait`, and `pthread_cond_clockwait`.
- The Open Group, POSIX.1–2024 semaphore functions.

- Butler Lampson and David Redell, “Experience with Processes and Monitors in Mesa”, *Communications of the ACM*, 1980.
- MIT PDOS, *xv6: a simple, Unix-like teaching operating system*, sleep and wakeup discussion.
- Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, condition-variable and semaphore chapters.

SECTION DETAIL

Deadlock, Liveness, and Priority Inversion

Thread A holds the process-table lock and requests the filesystem-cache lock. Thread B holds the filesystem-cache lock and calls a helper that requests the process-table lock. Both locks work exactly as designed: each excludes the other thread forever.

Safety mechanisms can destroy progress when their acquisition dependencies form a cycle. Liveness analysis asks not only whether state remains valid, but whether required operations retain a path to completion.

Classify the stalled system before repairing it

DEFINITION 3.4.1 **Four distinct progress failures**

- **Deadlock:** a nonempty set of participants is permanently blocked because each waits for an event that only another participant in the set can cause.
- **Starvation:** one participant can remain unable to make progress while other participants continue completing work.
- **Livelock:** participants continue taking steps and changing protocol state, but the required operation never completes.
- **Priority inversion:** a higher-priority task is blocked, directly or transitively, by work owned by a lower-priority task; the inversion is **unbounded** when unrelated intermediate-priority work can extend that delay without a protocol bound.

These categories overlap in symptoms, not definitions. “The service hung” supplies no diagnosis.

Trace	Classification	Decisive evidence
A waits for L2 held by B; B waits for L1 held by A	deadlock	closed wait-for cycle; neither owner can release
reader-preferred lock admits a stream of readers while one writer waits	possible writer starvation	system completes reads; one enabled class is postponed indefinitely
two polite workers repeatedly release and retry in synchrony	livelock	both run and mutate state; neither commits
high-priority H waits for a mutex held by low-priority L while medium M preempts L	priority inversion	M delays the owner that could unblock H

Prediction — answer before expanding. A low-priority thread is preempted for 2 ms while holding a mutex needed by a high-priority thread. Is that fact alone deadlock, starvation, or priority inversion? Predict the classification and required evidence before revealing the answer.

It is priority inversion during that interval. A finite delay does not establish deadlock or starvation. To claim an unbounded inversion, show that unrelated work can keep the owner from running without a protocol bound. To claim starvation, show an allowed infinite execution in which the waiting thread never completes while others do.

CLASSIFYING PROGRESS FAILURES

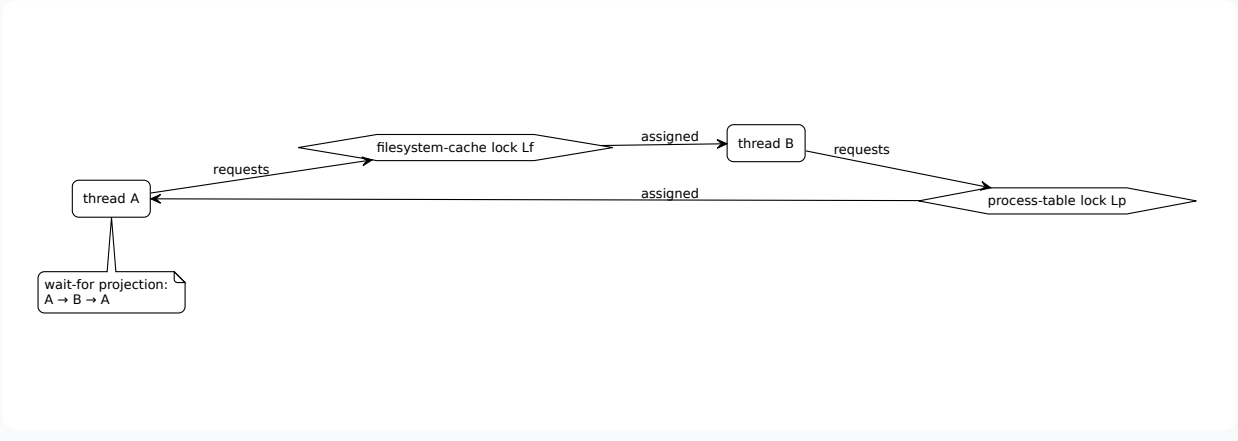
Evaluate each statement regarding concurrency, deadlocks, and liveness failures:

Answer: Livelock occurs when threads repeatedly change state and execute instructions without completing their operation.; Priority inheritance temporarily raises the scheduling priority of a lock holder to match its highest waiter.

Livelock involves active execution without progress. Priority inheritance bounds inversion by elevating the owner priority to the highest waiter priority. Fixed sleep intervals cause synchronized livelock rather than breaking deadlocks, and cycles in multi-unit resource graphs are necessary but not sufficient for deadlock.

Resource graphs expose ownership and demand

A resource-allocation graph has thread nodes and resource nodes. An edge $T \to R$ means T requests R; an edge $R \to T$ means an instance of R is assigned to T. For single-instance reusable resources, collapse each request through the current owner into a wait-for edge $T_i \to T_j$.



In a wait-for graph where every resource has one instance and blocked requests cannot be withdrawn, a directed cycle is necessary and sufficient for deadlock among the cycle's threads. With multiple interchangeable instances, a cycle is necessary but may not be sufficient: a thread outside the cycle can release another instance and satisfy a request. State the resource model before interpreting a cycle.

Worked trace: multi-instance wait-for graph. A holds one of two database connections and waits for a log slot. B holds the log slot and waits for a database connection. The graph contains A → B and B → A if edges are projected only through the currently observed connection owner. Yet a third thread C can return the second connection, allowing B to proceed. The projected cycle did not prove deadlock because the resource had another instance and another release path.

The four classical conditions describe a resource regime in which deadlock can arise:

1. **mutual exclusion:** at least one resource cannot be shared simultaneously;
2. **hold and wait:** a participant holds resources while requesting more;
3. **no forced preemption:** resources are released by their owners rather than safely taken away; and
4. **circular wait:** a closed chain of participants each waits for a resource held by the next.

All four are necessary for deadlock in that model. Their availability as system policies is not proof that a particular execution is deadlocked; construct the current ownership and wait relation.

A lock order is a proof, not a comment

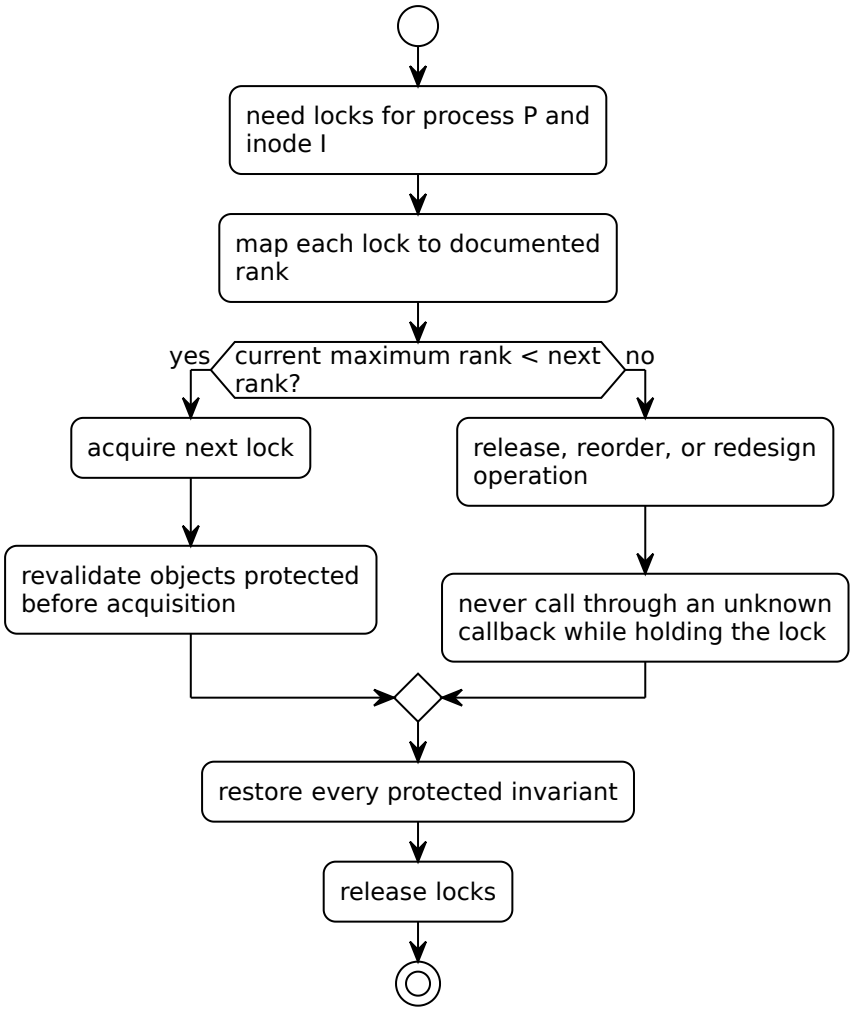
Assign every lock a rank in a strict total order and require a thread to acquire locks only in increasing rank. Suppose a deadlock cycle exists:

$$T_1 \to L_2 \to T_2 \to \cdots \to T_k \to L_1 \to T_1.$$

Each thread holds one lock and requests the next, so the order rule implies

$$rank(L_1) < rank(L_2) < \cdots < rank(L_k) < rank(L_1),$$

which is impossible for a strict order. The policy eliminates circular wait.



Worked transfer operation. To move a page from source cache bucket 9 to destination bucket 3, the semantic order "source then destination" conflicts with numeric rank. Acquire bucket 3 then bucket 9, regardless of transfer direction. Because the source was inspected before both locks were held, revalidate that the page remains in bucket 9 and that the destination does not already contain it. Lock order prevents deadlock; revalidation preserves correctness.

Counterexample: hidden acquisition. Code holds rank-20 lock L and invokes a callback supplied by another subsystem. The callback acquires rank-10 lock K. The local source file shows no descending acquisition, but the dynamic call graph violates the proof. Solutions include forbidding callbacks under L, splitting prepare/commit phases, passing immutable data, or making the callback's lock contract explicit and mechanically checked.

Application. A recursive tree traversal acquires each child lock while retaining its parent lock. Does "parent before child" define a global order when another operation can rotate the tree?

Only if parent/child relationships remain stable for the entire acquisition and no operation needs child-before-parent. Rotation changes topology, so a relation derived from mutable structure may cease to be acyclic or consistently observed. Stable object IDs, address/rank order, lock coupling with revalidation, or a coarser tree lock can supply a defensible order. The answer must include object lifetime: comparing addresses is unsafe if objects can be freed and reused while a contender waits.

Prevention, avoidance, and recovery solve different contracts

REMARK

Prevention constrains every execution before a cycle forms; avoidance rejects grants that would remove all safe completion sequences; detection accepts risk and requires a credible rollback or victim policy.

Strategy	Required knowledge or power	Useful case	Boundary
prevent hold-and-wait	request all resources together or release before requesting more	small known resource set	lowers concurrency; all-or-nothing acquisition may be impossible
prevent circular wait	stable global rank	kernel lock hierarchies	hidden callbacks and dynamic resources complicate ranking
avoidance	declared maximum future demand; grant only if state remains safe	controlled batch/resource systems	inaccurate maximum claims make policy unusable or unsafe
detection and recovery	observable allocation state plus a victim/rollback mechanism	transactions, killable jobs	kernel locks and irreversible I/O may not be safely revoked
timeout	clock and abort path	bounds one caller's wait when abort is valid	does not prove the cycle is gone; synchronized retries can livelock

Avoidance asks whether there exists a **safe completion sequence** after a proposed grant. A state can be safe even when not every thread could complete next; at least one sequence lets all finish within declared maximum demands. Banker's-style reasoning is meaningful only when maximum claims and resource counts are trustworthy.

Worked safe-state check. Two processes share three identical buffers. A holds one and may need at most one more; B holds one and may need at most two more. One buffer is free. Grant it to A: A can finish and release two, then B can obtain its remaining demand. Grant it to B: B would hold two and need one, A would hold one and need one, and none would be free; under the declared maxima, no completion sequence remains. The second grant is unsafe, though no one is deadlocked at the instant before either makes its next request.

Detection does not imply recovery. Killing a process that holds a user-space mutex may be acceptable if robust-mutex recovery marks protected state inconsistent. Killing a kernel thread halfway through device programming may leave hardware and memory in an unrepairable protocol state. Define the rollback boundary before choosing "detect and abort."

Backoff can replace deadlock with livelock

Two transfer threads each `trylock` one account, fail on the second, release, wait exactly 1 ms, and retry. Their lockstep behavior can continue forever. There is no blocked cycle because each releases; there is no progress because both repeat the same collision.

> **WARNING**
>
> Random delay reduces the probability of synchronized retry under a stated adversary; it is not a deterministic liveness proof. Fair queueing, asymmetric rank, a bounded retry followed by serialization, or scheduler-assisted parking can supply a stronger contract.

Starvation requires a fairness argument. A test-and-set spin lock can repeatedly admit a nearby core while a remote core loses cache-line races. A reader-preferred reader/writer lock can admit an endless stream of readers ahead of a writer. FIFO admission bounds overtaking but can create convoying when the head waiter is descheduled. Liveness and throughput objectives can conflict.

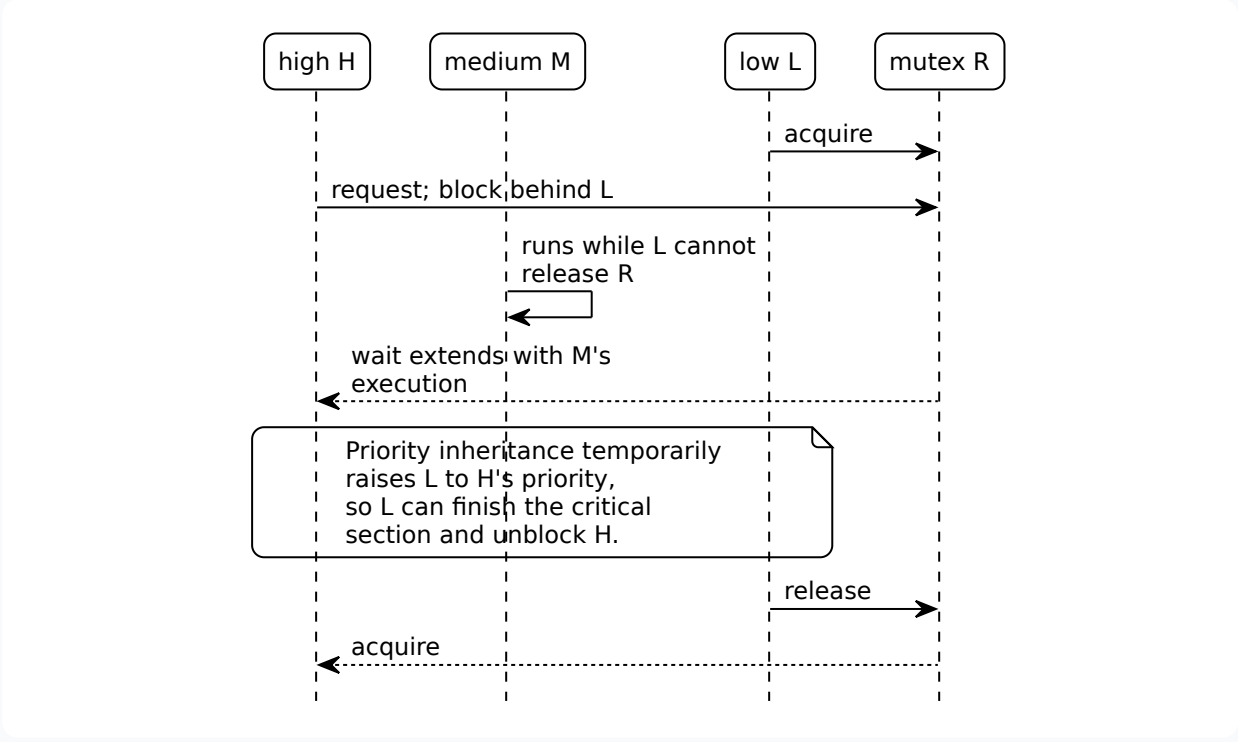
Transfer. Define a test that distinguishes deadlock from livelock without declaring every slow run a failure.

Instrument protocol state and completed operations. In a controlled schedule, deadlock reaches a stable wait-for cycle with no enabled participant that can release a required resource. Livelock continues producing state transitions or retries while a completion counter remains unchanged. Use a finite watchdog only to trigger capture; classify from the captured wait-for and transition evidence. A wall-clock timeout alone cannot distinguish load, preemption, deadlock, and livelock.

Priority inversion is a dependency–scheduling problem

Let H have high priority, M medium priority, and L low priority:

1. L acquires mutex R.
2. H becomes runnable, preempts L, then blocks on R.
3. M becomes runnable. Because M outranks L, M runs.
4. H is effectively delayed by M even though H and M share no lock.



Under **priority inheritance**, L temporarily inherits the highest priority of a waiter on R. If L itself blocks on a mutex owned by another task, the boost may propagate along that dependency chain. The boost is removed when the relevant dependency ends. This bounds interference from unrelated medium-priority work under the protocol's scheduler assumptions; it does not shorten L's critical section or cure a deadlock cycle.

A **priority-ceiling** protocol associates a ceiling with each resource based on the highest-priority task that may lock it. Variants can prevent certain deadlocks and bound a task's blocking by restricting acquisitions according to active ceilings. Those guarantees assume fixed-priority scheduling, declared resource use, bounded critical sections, and the specified protocol. They are not inherited by an arbitrary mutex with a "priority" field.

Worked nested inversion. H waits for R1 held by L. L waits for R2 held by X. Boosting only L does not help because X remains preemptible by M. A transitive PI implementation propagates H's effective priority through L to X; X releases R2, L resumes and releases R1, then H proceeds. The chain also shows why PI implementation needs cycle detection and careful priority restoration.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 3.4.1: Evaluating the Four Coffman Necessary Conditions

Verification application

State the four Coffman conditions necessary for a system deadlock to arise. For each condition, describe how a concrete operating system mechanism directly invalidates that condition.

Recall mutual exclusion, hold and wait, no preemption, and circular wait.

The four necessary conditions established by Coffman, Elphick, and Shoshani (1971) are:

1. **Mutual exclusion:** at least one resource must be held in a non-shareable mode. *Invalidation:* Use lock-free data structures (e.g., atomic compare-and-swap) or read-only shared access where multiple threads proceed concurrently without mutual exclusion.
2. **Hold and wait:** a thread currently holding at least one resource may request additional resources. *Invalidation:* Require a thread to request all needed resources simultaneously in a single atomic operation, or release all held resources before requesting new ones.
3. **No preemption:** resources cannot be forcibly confiscated from a thread; they can only be released voluntarily by the holding thread. *Invalidation:* Implement preemption protocols where the OS kernel or runtime revokes a resource (or aborts and restarts a transaction) if an additional request cannot be immediately satisfied.
4. **Circular wait:** a closed chain of threads $\{T_0, T_1, \ldots, T_{k-1}\}$ exists such that T_i waits for a resource held by $T_{(i+1) \bmod k}$. *Invalidation:* Impose a strict global partial or total ordering on all resource classes, mandating that resources are acquired only in increasing order.

Breaking any single condition renders deadlock mathematically impossible.

Exercise 3.4.2: Cycle Detection in Multi-Unit Resource Graphs

Construction application

Construct a concrete resource-allocation graph with two thread nodes (T_1, T_2) and two resource nodes (R_1, R_2) where R_1 has 2 available units and R_2 has 1 available unit. Show that a directed cycle exists in the graph, yet the system is not deadlocked.

Give R1 two units held by different threads or an unallocated unit that allows one thread to finish.

Let resource R_1 possess 2 units and resource R_2 possess 1 unit. Introduce a third thread T_3 to illustrate the distinction between single-unit and multi-unit resource graphs:

- R_1 allocation edges: $R_1 \to T_1$ and $R_1 \to T_3$ (both units allocated).
- R_2 allocation edge: $R_2 \to T_2$ (its single unit allocated).
- Request edges: $T_1 \to R_2$ (T_1 requests R_2) and $T_2 \to R_1$ (T_2 requests R_1).

Notice the directed cycle: $T_1 \to R_2 \to T_2 \to R_1 \to T_1$.

However, thread T_3 holds a unit of R_1 and has no outstanding requests. Thread T_3 executes to completion and releases its unit of R_1. The newly freed unit of R_1 is allocated to T_2, satisfying T_2's request. T_2 completes and releases both its unit of R_1 and R_2. Finally, R_2 is allocated to T_1, allowing T_1 to complete.

Hence, while a cycle is necessary and sufficient for deadlock in single-unit resource graphs, in multi-unit graphs a cycle is merely necessary, not sufficient. A knot (or absence of reducible sinks) is required for deadlock in general graphs.

Exercise 3.4.3: Proving Deadlock Freedom via Global Lock Hierarchies

Proof synthesis

Let $\mathcal{L} = \{L_1, L_2, \ldots, L_m\}$ be a finite set of mutex locks with a strict total order $\prec$ defined over them. Suppose every thread acquires locks strictly in ascending order: if a thread holds lock L_i and requests lock L_j, then $L_i \prec L_j$. Prove that

circular wait cannot occur.

Assume a cycle of waiters exists and examine the ordering of the held locks.

Assume for contradiction that a circular wait exists among $k \geq 2$ threads $\{T_0, T_1, \ldots, T_{k-1}\}$.

For each thread T_i, let $H_i \subseteq \mathcal{L}$ be the set of locks currently held by T_i, and let $\max(H_i)$ denote the maximum lock held by T_i with respect to $\prec$. Suppose thread T_i is blocked requesting lock R_i, which is held by thread $T_{(i+1) \bmod k}$.

By the strict ordering protocol, because T_i requests R_i while holding $\max(H_i)$, we must have:

$$\max(H_i) \prec R_i.$$

Furthermore, since $R_i \in H_{(i+1) \bmod k}$, by definition of the maximum element:

$$R_i \preceq \max(H_{(i+1) \bmod k}).$$

Combining these inequalities yields:

$$\max(H_i) \prec \max(H_{(i+1) \bmod k}) \quad \text{for all } i \in \{0, 1, \ldots, k-1\}.$$

Chaining this strict inequality along the entire cycle from T_0 to $T_k = T_0$:

$$\max(H_0) \prec \max(H_1) \prec \max(H_2) \prec \cdots \prec \max(H_{k-1}) \prec \max(H_0).$$

By transitivity of the strict total order $\prec$, this implies $\max(H_0) \prec \max(H_0)$, which violates the irreflexivity of strict orders.

Thus, no circular wait can form, proving that the system is guaranteed deadlock-free.

Exercise 3.4.4: Deterministic Retry Traps versus Randomized Exponential Backoff

Description application

Explain why two threads using non-blocking `try_lock()` with identical, deterministic sleep intervals can enter livelock. Explain how randomized exponential backoff breaks the synchronization lockstep.

Contrast deterministic phase alignment with probabilistic symmetry breaking.

When two threads attempt to acquire two locks L_1 and L_2 using `try_lock()`:

1. Thread A acquires L_1, and Thread B acquires L_2.
2. Both attempt to `try_lock()` the second lock and fail.
3. Both release their held lock to avoid deadlock and immediately execute a fixed deterministic sleep (e.g., `sleep(1ms)`).
4. Because both threads wake up at identical offsets, they repeat the acquisition in perfect lockstep: Thread A gets L_1, Thread B gets L_2, both fail on the second lock, both release, and both sleep again.

This is livelock: CPU cycles and state transitions occur continuously, but the aggregate operation makes zero forward progress.

Randomized exponential backoff breaks this deterministic symmetry by selecting a sleep duration uniformly at random from a dynamically doubling window: $t_{\text{sleep}} \sim U[0, 2^k \cdot \tau]$ on the k-th failure. With high probability, one thread draws a significantly shorter delay, awakens earlier, and acquires both locks uncontested before the other thread wakes up.

Exercise 3.4.5: Priority Inheritance Protocol Bounds on Priority Inversion Chains

Formalisation transfer

Consider three threads with priorities $\pi(H) > \pi(M) > \pi(L)$. Thread L acquires mutex R, after which thread H arrives and blocks requesting R. Thread M becomes runnable and does not need R.

1. Formalize the duration of priority inversion experienced by H without priority inheritance.
2. Formalize the maximum blocking time under the basic priority inheritance protocol.

Without inheritance, M can preempt L indefinitely. With inheritance, L runs at effective priority pi(H).

1. **Without priority inheritance:** Thread H is blocked awaiting mutex R held by L. However, thread M has higher priority than L ($\pi(M) > \pi(L)$) and is ready to run. The scheduler preempts L to execute M. If a set of medium-priority threads $\mathcal{M}$ arrives, thread L cannot execute until all threads in $\mathcal{M}$ complete or block. The blocking time experienced by H is:

$$T_{\text{block}}(H) = t_{\text{crit}}(L) + \sum_{M_j \in \mathcal{M}} t_{\text{exec}}(M_j),$$

   where $\sum t_{\text{exec}}(M_j)$ can be unbounded if new medium-priority tasks continue arriving. This is unbounded priority inversion (the classic Mars Pathfinder hazard).

2. **Under basic priority inheritance protocol (PIP):** When H blocks on mutex R held by L, the kernel elevates L's effective priority:

$$\pi_{\text{eff}}(L) = \max(\pi(L), \pi(H)) = \pi(H).$$

   Now $\pi_{\text{eff}}(L) > \pi(M)$ for all $M \in \mathcal{M}$. The scheduler cannot preempt L with medium-priority work. L executes its critical section without interference from $\mathcal{M}$, releases R, restores its original priority $\pi(L)$, and immediately unblocks H. The maximum blocking duration for H is strictly bounded by the execution time of L's critical section:

$$T_{\text{block}}(H) \leq t_{\text{crit}}(L).$$

References & Further Reading

- Edward G. Coffman Jr., M. J. Elphick, and Arie Shoshani, “System Deadlocks”, *ACM Computing Surveys*, 1971.
- Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky, “Priority Inheritance Protocols: An Approach to Real-Time Synchronization”, *IEEE Transactions on Computers*, 1990.
- Linux kernel documentation, RT-mutex implementation design.
- The Open Group, POSIX.1–2024 mutex priority protocols.
- Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, deadlock chapter.

Virtual memory

SECTION DETAIL

Address Spaces and Translation

Two independently linked programs both place code at virtual address `0x1000`, grow a heap upward, and grow a stack downward. Loading both programs into one physical memory creates an immediate conflict if an address names a physical byte directly. Relocating every pointer when a process moves is not sufficient: pointers can be stored in files, registers, data structures, or generated while the program runs. A faulty process could also manufacture the physical address of another process's data.

The missing mechanism is a controlled naming layer. A process issues a **virtual address**; the processor interprets that address through the process's current **address space**; translation either yields an authorized **physical address** or raises an exception before the access completes.

> ***Isolation invariant.*** *A user-mode memory access may affect or reveal a physical byte only if the active address space contains a mapping for the requested virtual address and that mapping authorizes the requested operation.*

The invariant contains three independent questions: *Which address-space context is active? Which physical storage does the name select? Which operations does the mapping permit?* Omitting any one turns translation into relocation without protection.

Names require a context

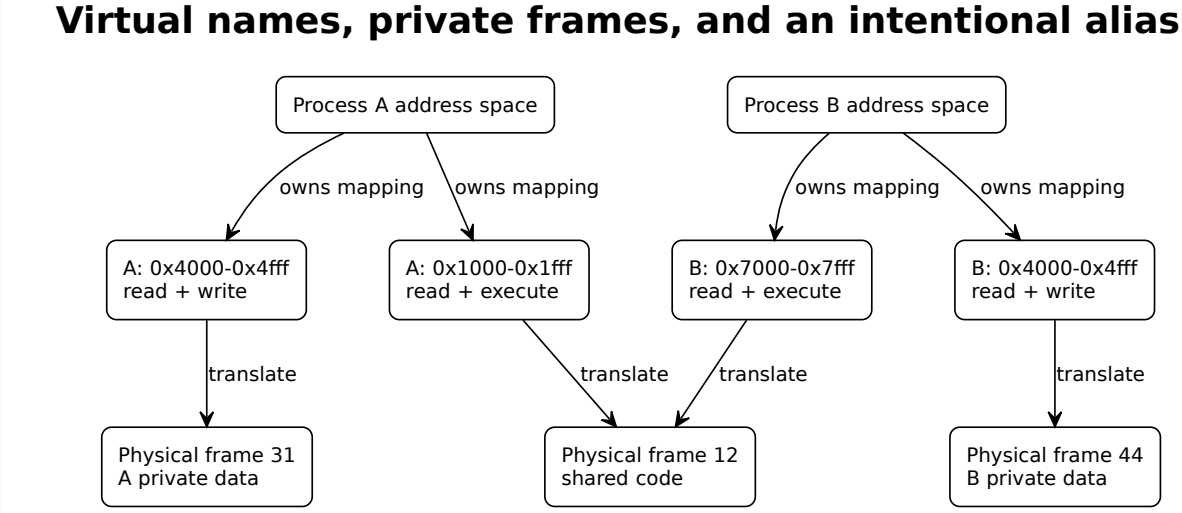
A virtual address is not globally meaningful. Model an address space as a partial function

$$T_A : (v, a) \mapsto \begin{cases} (p, \text{allow}) & \text{if address space } A \text{ maps } v \text{ and permits access } a, \\ \text{fault} & \text{otherwise.} \end{cases}$$

Here v is a virtual byte address, $a \in \{\text{read}, \text{write}, \text{execute}\}$, and p is a physical byte address. The same bit pattern can produce different results under different contexts:

Context and access	Mapping	Result
process A reads `0x1800`	private data → frame 9, read/write	physical byte in frame 9
process B reads `0x1800`	private data → frame 27, read/write	a different physical byte
process B executes `0x1800`	frame 27, read/write only	protection fault
process C reads `0x1800`	no mapping	translation fault

Two virtual names may also intentionally denote one physical object. Shared libraries, shared-memory regions, and file-backed mappings rely on this aliasing. Isolation therefore does **not** require one-to-one translation. It requires that every alias be deliberate, authorized, and revoked safely.



Retrieval — close the prose before answering. Why is the pair `(virtual address, process identity)` still insufficient to decide whether a store may proceed?

Identity and virtual address select a candidate mapping, but authority is operation-specific. A read-only code mapping may authorize a load and instruction fetch while rejecting a store. An answer that mentions only different physical locations has recovered relocation but not protection.

ADDRESS TRANSLATION AND OPERATION AUTHORITY

Evaluate each statement regarding virtual address translation, memory isolation, and paging invariants:

Answer: Address translation checks operation permissions so read-only segments permit loads but trigger faults on stores.; Two distinct virtual page numbers in different address spaces can intentionally map to the same physical page frame.

Address translation validates operation permissions independently from address validity. Shared memory allows distinct virtual pages to reference identical physical frames. Paging preserves the exact intra-page offset, and valid entries may still reject user stores if write permissions or supervisor restrictions intervene.

Relocation by base and bound

The smallest useful translation mechanism assigns one contiguous physical region to a process. Let B be the physical base and L the length of the process's virtual region. For an unsigned virtual address v:

$$T(v) = \begin{cases} B + v & 0 \le v < L, \\ \text{fault} & \text{otherwise.} \end{cases}$$

The bounds check must occur **before** the addition. If a fixed-width addition wraps, checking the physical result can authorize an address outside the allocated region.

Worked trace: a valid load and an overflow attack

Assume 16-bit addresses, `B = 0x4000`, and `L = 0x1800`.

1. A load from `v = 0x02F0` passes because `0x02F0 < 0x1800`.
2. Translation yields `0x4000 + 0x02F0 = 0x42F0`.
3. A load from `v = 0xF100` fails the virtual bound and must fault.

An incorrect implementation first computes a 16-bit sum and then tests whether the result lies below `B + L`:

```
// Incorrect: wrapped addition occurs before authority is established.
uint16_t physical = base + virtual;
if (physical < base + length)
    return physical;
```

For `base = 0x4000` and `virtual = 0xF100`, the sum wraps to `0x3100`. A weak upper-bound-only test may accept the wrapped result. The safe form compares `virtual` with `length`, using an unsigned type whose width is part of the machine contract, and adds only after acceptance.

Worked trace: relocation without pointer rewriting

Process P executes a load from virtual `0x0710` while `B = 0x2000`; the physical access is `0x2710`. The scheduler later runs P from a region beginning at `0x9000`. Updating the privileged base register changes the same instruction's physical access to `0x9710`. P's register and memory values remain virtual; no embedded pointer is rewritten.

Base-and-bound provides cheap relocation, but it imposes one contiguous extent. A 1 MiB heap and a 1 MiB stack separated by 1 GiB of unused virtual space either reserve that physical hole or cannot be represented by one pair. Growing the region can require physical relocation. Sharing one read-only library while keeping data private also needs more than one extent.

Application — execute before revealing. A 20-bit machine gives a process `B = 0x62000` and `L = 0x2A000`. Classify virtual accesses `0x01FFC`, `0x29FFF`, and `0x2A000`; compute every valid physical address.

- `0x01FFC < 0x2A000`, so it maps to `0x63FFC`.
- `0x29FFF < 0x2A000`, so it maps to `0x8BFFF`, the final byte in the region.
- `0x2A000` equals the length and is outside the half-open interval `[0, L)`; it faults.

Accepting the last address indicates an inclusive-upper-bound error. Subtracting one from every physical result indicates confusion between a region length and its final index.

Paging derives from sparse allocation

Paging divides virtual memory and physical memory into equal-size units. Let page size $P = 2^k$ bytes. For virtual address v,

$$\mathrm{VPN}(v) = \left\lfloor \frac{v}{P} \right\rfloor, \qquad \mathrm{offset}(v) = v \bmod P.$$

A page-table entry maps a virtual page number (VPN) to a physical frame number (PFN) and carries control state such as validity and permissions. If the entry is valid and authorizes the access,

$$p = \mathrm{PFN}(\mathrm{VPN}(v)) \cdot P + \mathrm{offset}(v).$$

The offset is preserved because pages and frames have the same size. Translation changes the container, not the byte's position within that container.

Worked trace: noncontiguous physical placement

Take a 16-bit virtual address, 1 KiB pages, and these entries:

VPN	PFN	Permission	Present
`0x00`	`0x12`	read + execute	yes
`0x01`	`0x2B`	read + write	yes
`0x02`	—	—	no
`0x3F`	`0x07`	read + write	yes

Because $P = 1024 = 2^{10}$, the low ten bits are the offset.

- Virtual `0x0634` has VPN `1` and offset `0x234`. A read maps to `0x2B × 0x400 + 0x234 = 0xAE34`.
- Executing `0x0634` reaches a present page but violates its `read + write` permission; the result is a protection fault, not a translation.
- Reading `0x0A10` selects VPN `2`, whose entry is absent; the result is a not-present fault.
- Writing `0xFE20` selects VPN `0x3F`, offset `0x220`, and maps to `0x1E20`.

The final case demonstrates non-monotonic placement: a high virtual address can map to a low physical frame. Sorting virtual addresses does not sort physical addresses.

Fragmentation is relocated, not eliminated

Paging removes **external fragmentation** from allocation of individual frames: any free frame can hold any virtual page. It introduces **internal fragmentation** in the partially used final page of an allocation. An object of n bytes consumes

$$\left\lceil \frac{n}{P} \right\rceil P$$

bytes of mapped capacity, wasting that quantity minus n when it cannot share the final page.

A 10,500-byte isolated region with 4 KiB pages needs three pages: 12,288 bytes, of which 1,788 bytes are unused. The waste is a property of allocation granularity, not evidence that those physical frames are adjacent.

Transfer — change the granularity. The same 10,500-byte region uses 2 KiB pages. Compute mapped capacity and waste, then state one cost that the smaller pages increase.

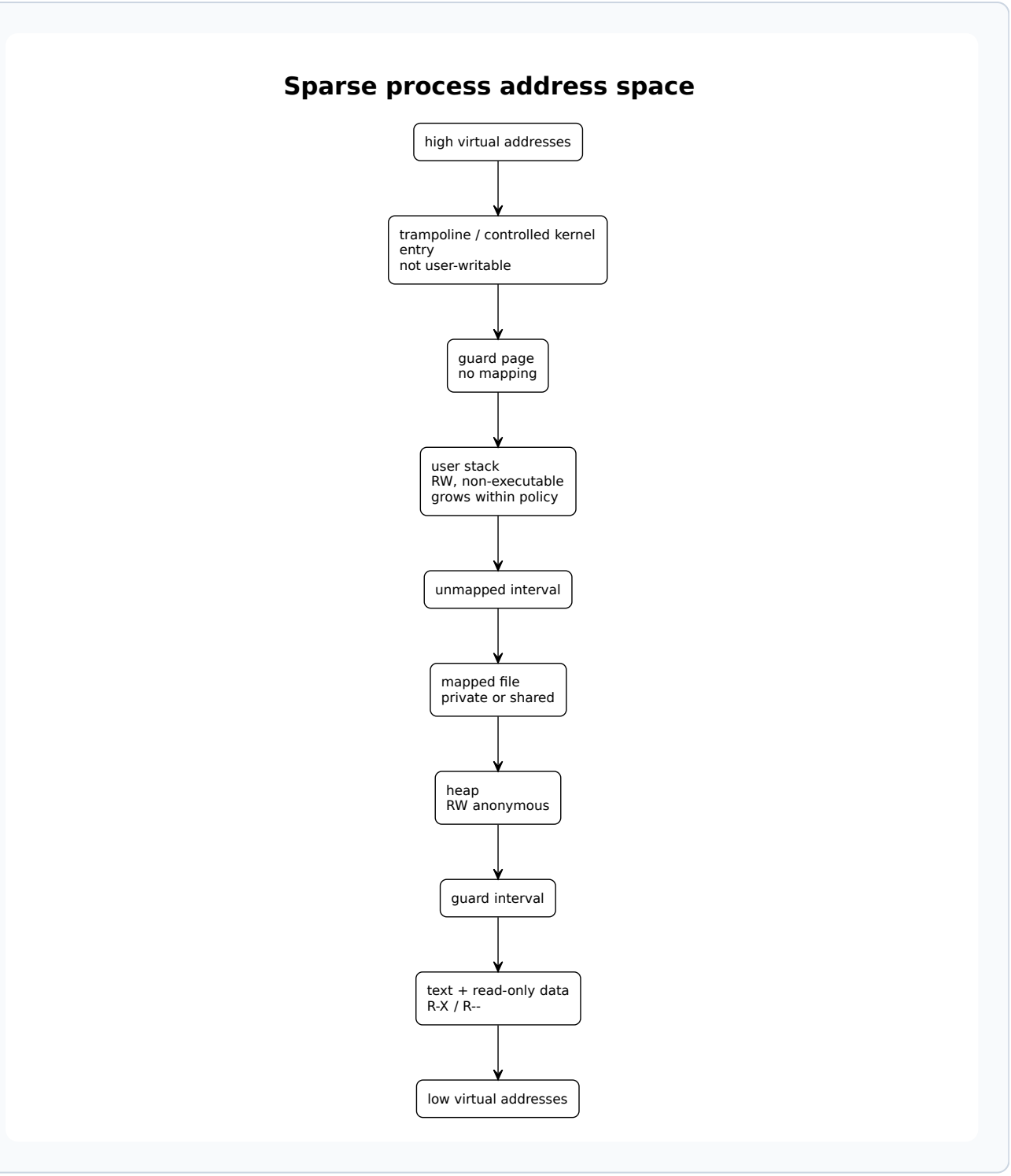
Six 2 KiB pages provide 12,288 bytes, so this particular size still wastes 1,788 bytes. Smaller pages do not guarantee less waste for every object. They increase the number of mappings and therefore page-table memory and translation-cache pressure. A response that claims the waste must halve has treated an average tendency as a per-object theorem.

Constructing a process address space

An address space is a set of regions with different backing objects and authorities, not a promise that every address between the lowest and highest mapping is valid.

Region	Typical backing	Initial materialization	Common authority
executable text	executable file	file page on demand or at load	read + execute
read-only data	executable file	file page	read
writable data	private file image	file page, then private on write	read + write
heap	anonymous memory	zero-filled page on first touch	read + write
memory-mapped file	file object	file page on demand	mapping-dependent
guard interval	none	never	no access
user stack	anonymous memory	policy-bounded growth	read + write, non-execute

The address-space manager records region-level intent; page tables record the translations currently installed. A valid region need not have a resident page yet. Conversely, a page-table entry left behind after its region is removed is stale authority.



Worked trace: stack growth or invalid access?

Assume a stack region may grow into one designated unmapped page below its current lowest mapped page, down to a fixed maximum size. Its current mapped interval is `[0x7FFF_D000, 0x8000_0000)`; `[0x7FFF_C000, 0x7FFF_D000)` is the current growth page, and the page below that is a non-growable guard.

1. A store at `0x7FFF_CFF8` lands eight bytes below the current mapping, in the permitted one-page growth interval. The kernel may allocate a zeroed frame, install a non-executable user mapping, and retry.
2. A store at `0x7FFE_0000` is far below the growth frontier. Treating every lower address as stack growth would let a stray pointer consume arbitrary memory and collide with another region. The access must fail.
3. An instruction fetch from `0x7FFF_E100` targets a stack page. Presence is not enough; non-execute authority causes a protection fault.

The growth policy belongs to region metadata. A page-table miss alone cannot distinguish an intentional lazy page from an invalid pointer.

Application — design a map. Two worker processes need private 64 KiB heaps, one 12 KiB read-only lookup table shared at different virtual addresses, and one writable 8 KiB communication ring shared at the same virtual address. Draw the VPN → PFN relations for 4 KiB pages and state the permission invariant for each object.

A complete design has sixteen distinct heap frames per process, three table frames mapped into both address spaces without write permission, and two ring frames mapped read/write into both address spaces. The virtual addresses of the table may differ because sharing is defined by common PFNs, not common VPNs. The ring's equal virtual address is a convenience, not a requirement. Complete verification requires separate PTE permissions in each address space and an explicit rule that no private heap PFN appears in the other process's user mappings.

Common defects:

- allocating six table frames loses sharing;
- mapping the table writable loses its integrity contract;
- requiring equal VPNs confuses virtual naming with physical identity;
- sharing page-table entries between processes without context separation can expose unrelated mappings.

Boundary failures and adversarial cases

A pointer is not a physical address

Passing a user pointer to a device for direct memory access without pinning and mapping its pages gives the device an address in the wrong namespace. A device may use a bus or I/O virtual address, and the process mapping may change while the operation is in flight. Translation must occur in the correct agent's context.

Presence does not imply authority

Marking code present and user-accessible while accidentally retaining write permission defeats write-xor-execute policy. The access does not fault because the mapping itself grants the wrong authority. Fault-handler testing cannot detect every over-permissive mapping; mapping construction needs invariant checks.

Unmapping is a concurrent revocation

Removing a page-table entry does not prove that no processor can use a cached translation. The kernel must invalidate stale translation state on every relevant processor before reusing the physical frame for a new security domain. Detailed invalidation mechanics are outside this scope, but the authority rule is immediate: frame reuse is forbidden until revocation is globally complete.

Aliases can disagree about policy

Mapping one physical frame writable through one alias and executable through another can defeat a policy checked per mapping. A system that requires write-xor-execute must reason about all aliases to the frame, not each PTE in isolation.

Retrieval plus boundary check. Classify each statement as invariant, common implementation choice, or false claim:

1. Page translation preserves the offset within a page.
2. Every process must map its shared library at the same virtual address.
3. A valid PTE guarantees a user store may proceed.
4. Any free physical frame can hold a virtual page of the same configured page size.

1. **Invariant** for equal-size paging: the mapping substitutes the frame number and preserves the offset.
2. **False claim:** different VPNs may map the same library frames; position-independent code makes this practical.
3. **False claim:** user/supervisor and read/write/execute permissions still govern the operation.
4. **Implementation property under the stated paging model:** equal-size frames eliminate the need for contiguous placement, although devices, huge pages, coloring, or physical constraints may narrow the eligible set.

REMARK

Address translation is modeled as an architecturally precise function $\mathrm{VA} \mapsto \mathrm{PA} \cup \{\mathrm{fault}\}$, cleanly separating abstract $\mathrm{VPN} \mapsto \mathrm{PFN}$ mappings from hardware radix-tree encoding (covered in Page Tables and TLBs) and microarchitectural transient execution.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 4.1.1: Virtual Page Number and Offset Bitwise Decomposition

Verification application

A processor uses 32-bit virtual addresses with a 4 KiB page size (2^{12} bytes). Given the virtual address `0x0045A8C4`, determine:

1. The page offset in hexadecimal.
2. The Virtual Page Number (VPN) in hexadecimal.
3. The resulting 32-bit physical address if the page table maps this VPN to Physical Frame Number (PFN) `0x1B02`.

With 4 KiB pages, the lower 12 bits form the offset, while the upper 20 bits form the VPN.

1. **Page offset:** With a 4 KiB page size, the offset width is $\log_2(4096) = 12$ bits (the lowest 3 hex nybbles). For `0x0045A8C4`, the lowest 12 bits are `0x8C4`.
2. **Virtual Page Number (VPN):** Shifting right by 12 bits ($\mathrm{VA} \gg 12$) extracts the upper 20 bits:

$$\mathrm{VPN} = \lfloor \texttt{0x0045A8C4} / 4096 \rfloor = \texttt{0x0045A}.$$

3. **Physical address calculation:** Paging preserves the page offset exactly. Substituting PFN `0x1B02` into the upper 20 bits gives:

$$\mathrm{PA} = (\mathrm{PFN} \ll 12) \mid \mathrm{offset} = (\texttt{0x1B02} \ll 12) \mid \texttt{0x8C4} = \texttt{0x01B028C4}.$$

Exercise 4.1.2: Address Translation under Base-and-Bound Protection Registers

Construction application

A system implements hardware base-and-bound memory relocation. A process is assigned a base register of `0x00300000` and a bound (limit) register of `0x00028000`.

1. Compute the translated physical address for a load instruction at virtual address `0x00014A00`.
2. Determine the hardware response for a load instruction at virtual address `0x00028000`.
3. Explain why the boundary check must use strict inequality ($v < \text{Bound}$) rather than weak inequality ($v \leq \text{Bound}$).

The bound defines the total length of the segment, so valid offsets range from 0 to Bound - 1.

1. **Valid load:** The virtual offset $v = \texttt{0x00014A00}$ is strictly less than the bound `0x00028000`. Hardware translates the address by addition:

$$\text{PA} = \text{Base} + v = \texttt{0x00300000} + \texttt{0x00014A00} = \texttt{0x00314A00}.$$

2. **Boundary check:** The access at $v = \texttt{0x00028000}$ satisfies $v = \text{Bound}$, so $v \not< \text{Bound}$. The MMU hardware raises a segmentation fault or memory protection exception prior to executing the bus cycle.
3. **Strict inequality rationale:** A segment of length $L = \text{Bound}$ bytes occupies offsets $0, 1, 2, \ldots, L - 1$. An access at offset L falls exactly one byte beyond the authorized allocated range. Using $v \leq \text{Bound}$ would expose an unallocated or neighboring byte, violating memory isolation.

Exercise 4.1.3: Constructing Dual Virtual Mappings to a Shared Physical Frame

Construction transfer

Two processes P_1 and P_2 share an inter-process communication ring buffer allocated at physical frame `0x04A0`. P_1 maps the buffer as read-write at virtual page `0x100`, while P_2 maps the buffer as read-only at virtual page `0x500`.

1. Construct the respective page table entry (PTE) fields (VPN, PFN, Read, Write, User) for both processes.
2. If P_1 writes the value `0xDEADBEEF` at virtual address `0x100040`, what virtual address does P_2 read to observe this update?
3. What occurs if P_2 executes a store instruction to its mapped address?

The two processes use different virtual page numbers that translate to the identical physical frame.

1. **PTE configuration:**
   - For P_1: $\mathrm{VPN} = \texttt{0x100} \mapsto \mathrm{PFN} = \texttt{0x04A0}$, with permissions $\mathrm{R} = 1, \mathrm{W} = 1, \mathrm{X} = 0, \mathrm{U} = 1$.
   - For P_2: $\mathrm{VPN} = \texttt{0x500} \mapsto \mathrm{PFN} = \texttt{0x04A0}$, with permissions $\mathrm{R} = 1, \mathrm{W} = 0, \mathrm{X} = 0, \mathrm{U} = 1$.

2. **Aliased read:** The write from P_1 occurs at intra-page offset `0x040` (since `0x100040` has VPN `0x100` and offset `0x040`), modifying physical address $(\texttt{0x04A0} \ll 12) + \texttt{0x040} = \texttt{0x004A0040}$. To read the same physical byte, P_2 accesses VPN `0x500` with the identical offset `0x040`:

$$\mathrm{VA}_{P2} = (\texttt{0x500} \ll 12) + \texttt{0x040} = \texttt{0x00500040}.$$

3. **Protection enforcement:** When P_2 attempts a store to VA_{P2}, the MMU validates the operation against P_2's PTE. Because $W = 0$, the MMU aborts the store and raises a page-protection fault exception to the kernel, preserving the read-only contract.

Exercise 4.1.4: Isolation Invariant Defense in Sparse Address Space Layouts

Verification synthesis

A 64-bit operating system places a 4 KiB non-mapped guard page immediately below each thread stack and immediately above the process heap.

1. Formulate the isolation invariant that explains how unmapped guard pages detect stack overflow and heap overrun.
2. Suppose an application bug causes an unchecked pointer increment of 64 KiB in a single step, jumping over the 4 KiB guard page into adjacent valid data. Explain why guard pages cannot prevent this corruption.

A guard page traps accesses whose stride is strictly smaller than the guard page thickness.

1. **Isolation invariant for guard pages:** Let virtual page g be unmapped ($\mathrm{valid} = 0$). Any instruction accessing $v \in [\mathrm{page_start}(g), \mathrm{page_end}(g)]$ immediately triggers an unhandled page fault exception before modifying memory. When stack frames grow sequentially by small increments ($< 4\ \mathrm{KiB}$), the stack pointer inevitably references page g on overflow, allowing the kernel to terminate the offending thread or expand the stack safely without silent memory corruption.
2. **Large-stride failure:** Guard pages rely on continuous, localized traversal. If an allocation or array access strides by 64 KiB, the calculated target virtual address $v_{\mathrm{target}} = v_{\mathrm{stack}} - 65536$ skips entirely past the 4 KiB range of g and lands directly on an active, mapped virtual page belonging to another segment. Because v_{target} possesses a valid PTE with write permissions, the hardware executes the store without faulting, silently corrupting unrelated data.

Exercise 4.1.5: Contrasting Paging Internal Fragmentation and Contiguous External Fragmentation

Description application

Contrast the origin and manifestation of internal fragmentation in fixed-size paging with external fragmentation in variable-size contiguous memory allocation. State why modern operating systems accept internal fragmentation in exchange for eliminating external fragmentation.

Consider unused space inside allocated units versus unallocatable scattered free space between units.

- **Internal fragmentation:** Occurs when memory is allocated in fixed-size discrete units (e.g., 4 KiB pages). If a process requests 5 KiB of heap space, the operating system must allocate two 4 KiB pages (8 KiB total). The remaining 3 KiB within the second page is allocated to the process but unused, representing internal waste. The average internal fragmentation per mapped segment is half a page ($P/2$).
- **External fragmentation:** Occurs in variable-size contiguous allocation schemes (such as pure segmentation or unpaged physical placement). As processes allocate, deallocate, and terminate at varying lifetimes, free physical memory becomes checkerboarded into small non-contiguous blocks. Total free memory may exceed a request, but no single contiguous chunk is large enough to satisfy it, requiring expensive memory compaction.
- **Architectural tradeoff:** Paging completely eliminates external fragmentation at the physical level because any free physical frame can satisfy any virtual page request, regardless of physical location. Operating systems accept bounded internal fragmentation (at most a fraction of a page per region) because it avoids compaction overhead and enables non-contiguous, demand-paged allocation.

References & Further Reading

- RISC-V Instruction Set Manual, Volume II: Privileged Architecture, supervisor address translation
- POSIX.1–2024 `mmap()`
- POSIX.1–2024 `fork()`
- xv6: a simple, Unix-like teaching operating system, RISC-V revision 5
- xv6–riscv revision 5 source, commit `7d7adbb`

SECTION DETAIL

Page Tables and TLBs

A process removes write permission from a page, updates the page-table entry, and hands the physical frame to a read-only consumer. Another core writes through a translation cached before the update. The page table in memory is correct; the system is not.

Page tables are authoritative data structures, but processors cache their results. Correctness therefore has two layers:

1. a page-table walk must derive the right frame and authority from the current entries; and
2. every translation cache that could hold an older result must stop using it before the old authority is considered revoked.

> ***Translation-cache invariant.*** *Every usable TLB entry must be equivalent to a currently authorized page-table translation, or the kernel must prevent the entry from being used until invalidation completes.*

Sv39 is a radix-tree lookup

The baseline RISC-V Sv39 scheme uses 4 KiB pages and three page-table levels. A page-table page contains 512 eight-byte entries, so each level consumes nine virtual-page-number bits. The 64-bit virtual address is interpreted as:

Field	Bits	Meaning
sign extension	`63:39`	must equal bit 38 for a canonical Sv39 address
`VPN[2]`	`38:30`	root-page index
`VPN[1]`	`29:21`	middle-level index
`VPN[0]`	`20:12`	final-level index
page offset	`11:0`	byte within a 4 KiB page

The `satp` control register identifies the translation mode, root page-table physical page number, and—when implemented and used—an address-space identifier (ASID). An Sv39 page table is a sparse radix tree: a non-leaf PTE points to the next page-table page; a leaf PTE supplies the final physical page number (PPN) and permissions.

PTE meaning before arithmetic

For the baseline Sv39 PTE, the relevant low bits are:

Bit	Name	Architectural role in this model
0	`V`	entry is valid
1	`R`	loads permitted at a leaf
2	`W`	stores permitted at a leaf
3	`X`	instruction fetch permitted at a leaf
4	`U`	user-mode access permitted
5	`G`	mapping is global across address spaces
6	`A`	page has been accessed, according to the implemented A/D scheme
7	`D`	page has been written, according to the implemented A/D scheme
`9:8`	`RSW`	reserved for supervisor software
`53:10`	`PPN`	next table or mapped physical page

An entry with `V=0` is invalid. The combination `R=0, W=1` is also reserved as invalid. If any of `R`, `W`, or `X` is set in a valid entry, the entry is a leaf. If all three are clear, the entry is a pointer to the next level. User access additionally requires the appropriate operation permission and `U=1`.

Leaf entries may appear above level zero to create superpages. A level-1 leaf maps 2 MiB; a level-2 leaf maps 1 GiB. The unused low PPN fields must be zero so the physical superpage is aligned. A high-level leaf with misaligned physical bits faults rather than rounding the address.

Retrieval — reconstruct the discriminator. Why can the walker not decide “leaf versus next table” from `V` alone?

`V` establishes only that the entry has an architecturally valid form. A valid entry with `R=W=X=0` points to a lower-level table; a valid entry with any of `R`, `W`, or `X` set is a leaf. Treating every valid PTE as a leaf would interpret a page-table page's PPN as application data.

SV39 PAGE TABLES AND TLB MANAGEMENT

Evaluate each statement regarding RISC-V Sv39 radix page tables and TLB coherence:

Answer: A level-1 leaf PTE mapping a 2 MiB superpage triggers a page fault if its low physical page bits PPN[0] are non-zero.; Multi-core systems must execute an inter-processor shootdown before freeing a physical frame to eliminate stale translations.

In Sv39, superpages must be naturally aligned (PPN[0] must be zero for 2 MiB leaves). Stale translations cached in remote TLBs must be explicitly invalidated before physical frame reuse. Valid entries with R=W=X=0 are table pointers rather than leaf translations, and memory updates do not automatically invalidate cached translations on remote cores.

Execute a complete page-table walk

For a memory access at virtual address v, a conceptual Sv39 walker performs this state transition:

```
require canonical(v)
table_pa := satp.PPN × 4096
for level i := 2 down to 0:
    pte_pa := table_pa + VPN[i](v) × 8
    pte := physical_memory[pte_pa]
    reject invalid encodings
    if pte is a leaf:
        check superpage alignment, privilege, and access permission
        combine leaf PPN, any inherited VPN bits, and offset
        return physical address
    if i = 0:
        fault: no leaf exists
    table_pa := pte.PPN × 4096
```

The pseudocode omits extension-dependent details but preserves the control dependencies: validity before dereference, leaf classification before permission checks, and permission checks before the data access completes.

Worked trace: a three-level 4 KiB mapping

Translate a user read from virtual address `0x0000000123456789`. It is canonical because bit 38 and bits `63:39` are all zero.

Field extraction gives:

Field	Decimal	Hex
`VPN[2]`	4	`0x004`
`VPN[1]`	282	`0x11A`
`VPN[0]`	86	`0x056`
offset	1929	`0x789`

Assume `satp.PPN` selects root table physical address `0x81000000`.

1. Root PTE address: `0x81000000 + 0x004 × 8 = 0x81000020`. Its valid non-leaf PTE points to table `0x81200000`.

2. Middle PTE address: `0x81200000 + 0x11A × 8 = 0x812008D0`. Its valid non-leaf PTE points to table `0x81300000`.
3. Final PTE address: `0x81300000 + 0x056 × 8 = 0x813002B0`. It is a leaf with `V=1, R=1, W=0, X=0, U=1` and PPN `0x2A35F`.
4. A user read is authorized. The physical address is `(0x2A35F << 12) | 0x789 = 0x2A35F789`.

Changing the operation to a store produces a protection fault at step 3; the walker must not issue a store to `0x2A35F789`. Changing `U` to zero also rejects the user access even though `R=1`.

Worked trace: a level-1 superpage

Take virtual address fields `VPN[2]=7`, `VPN[1]=12`, `VPN[0]=301`, offset `0x2A0`. At level 1 the walker finds an `R-X` leaf. A 2 MiB mapping replaces only the high physical page-number fields; the low nine PPN bits represented by `VPN[0]` come from the virtual address. The physical construction is conceptually:

$$PA = \mathrm{PTE.PPN}[2{:}1] \;||\; \mathrm{VPN}[0] \;||\; \mathrm{offset}.$$

If `PTE.PPN[0]` is nonzero, the leaf is misaligned and the access faults. Copying the full PTE PPN into the physical address would map a 4 KiB page and discard the very property that made the entry a level-1 leaf.

Application — audit a walk. A trace reaches level 0 with `V=1, R=0, W=0, X=0`. The implementation returns `PTE.PPN << 12 | offset`. Identify the defect and the correct architectural outcome.

At level 0, a valid non-leaf entry asks the walker to descend when no level remains. No leaf translation exists, so the access faults. Returning the PPN exposes the page-table page itself as user data and can turn metadata into writable authority if subsequent checks are also wrong.

A TLB caches a proof result, not only a number

A translation lookaside buffer (TLB) caches a key such as `(ASID, VPN, page size)` and a result containing the PPN plus permission and status state. A TLB hit bypasses page-table memory references, but it does not bypass the access check. A cached read-only translation must still reject a store.

A **TLB miss** means the translation is absent from the cache. A **page fault** means translation cannot complete under the current page-table and access rules. A TLB miss can end in a successful walk; a TLB hit can end in a protection fault.

Translation cost under an explicit serial model

Assume:

- a TLB lookup costs t_T;
- each memory access costs t_M;
- a miss performs an L-level serial page-table walk and then the requested data access;
- the walk is not served by caches and does not overlap the data-cache lookup; and
- page faults are excluded.

With TLB hit rate h,

$$E[T] = h(t_T + t_M) + (1 - h)(t_T + (L + 1)t_M).$$

For $h = 0.98$, $t_T = 1$ ns, $t_M = 60$ ns, and $L = 3$:

$$E[T] = 0.98(61) + 0.02(241) = 64.6 \text{ ns}.$$

The 2% miss rate adds 3.6 ns over the 61 ns hit path under this model. Real processors overlap lookups and cache page-table entries, so this equation is a derivation exercise, not a hardware benchmark.

Application — expose the hidden assumption. A report uses `1 ns + 3 × 60 ns + 60 ns = 241 ns` as the cost of *every* load on the machine above. Which observation would falsify the report fastest?

Reach and page size

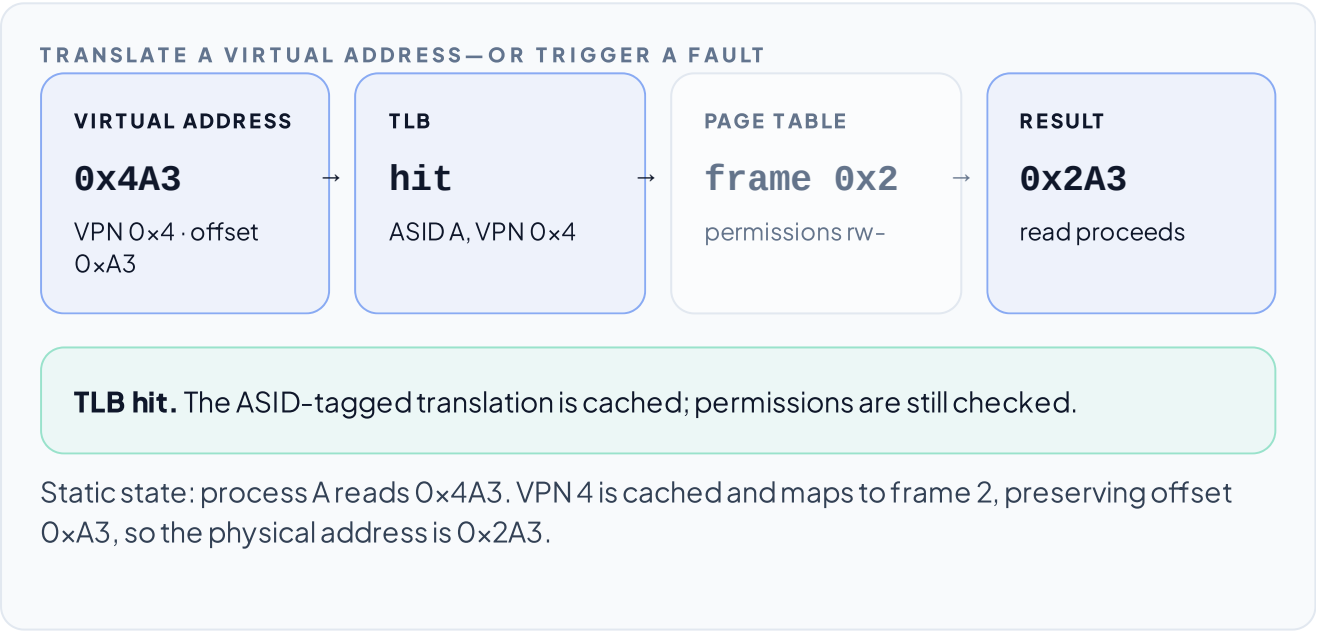
TLB **reach** is the mapped byte span represented by useful cached entries. Sixty-four 4 KiB entries cover at most 256 KiB; sixty-four 2 MiB entries cover 128 MiB. Larger pages increase reach but coarsen allocation, sharing, protection, and fault granularity. Reach is an upper bound: conflicting

sets, mixed page sizes, duplicate aliases, and context changes can reduce useful coverage.

Experiment: translation, permission, and cache state

The investigation below uses a deliberately small machine: 12-bit virtual addresses, 256-byte pages, one page-table level, 4-bit physical frame numbers, and a fixed ASID-tagged TLB snapshot. It models address decomposition and permission decisions, not Sv39's three memory references, replacement behavior, A/D updates, or invalidation races.

Prediction — record an answer before changing a control. Process A reads virtual `0x4A3`. Which bits are preserved in the physical address, and can a TLB hit make a later write legal if the PTE is read-only?



Static state: process A reads 0×4A3. VPN 4 is cached and maps to frame 2, preserving offset 0×A3, so the physical address is 0×2A3.

Transfer — force three distinct outcomes. Find one setting that produces a TLB hit, one that performs a successful page-table walk, and one that faults despite the page being present. For each, write `(process, access, VA, VPN, offset, result)` and identify the first decision that distinguishes it from the other two.

Examples from the fixed model:

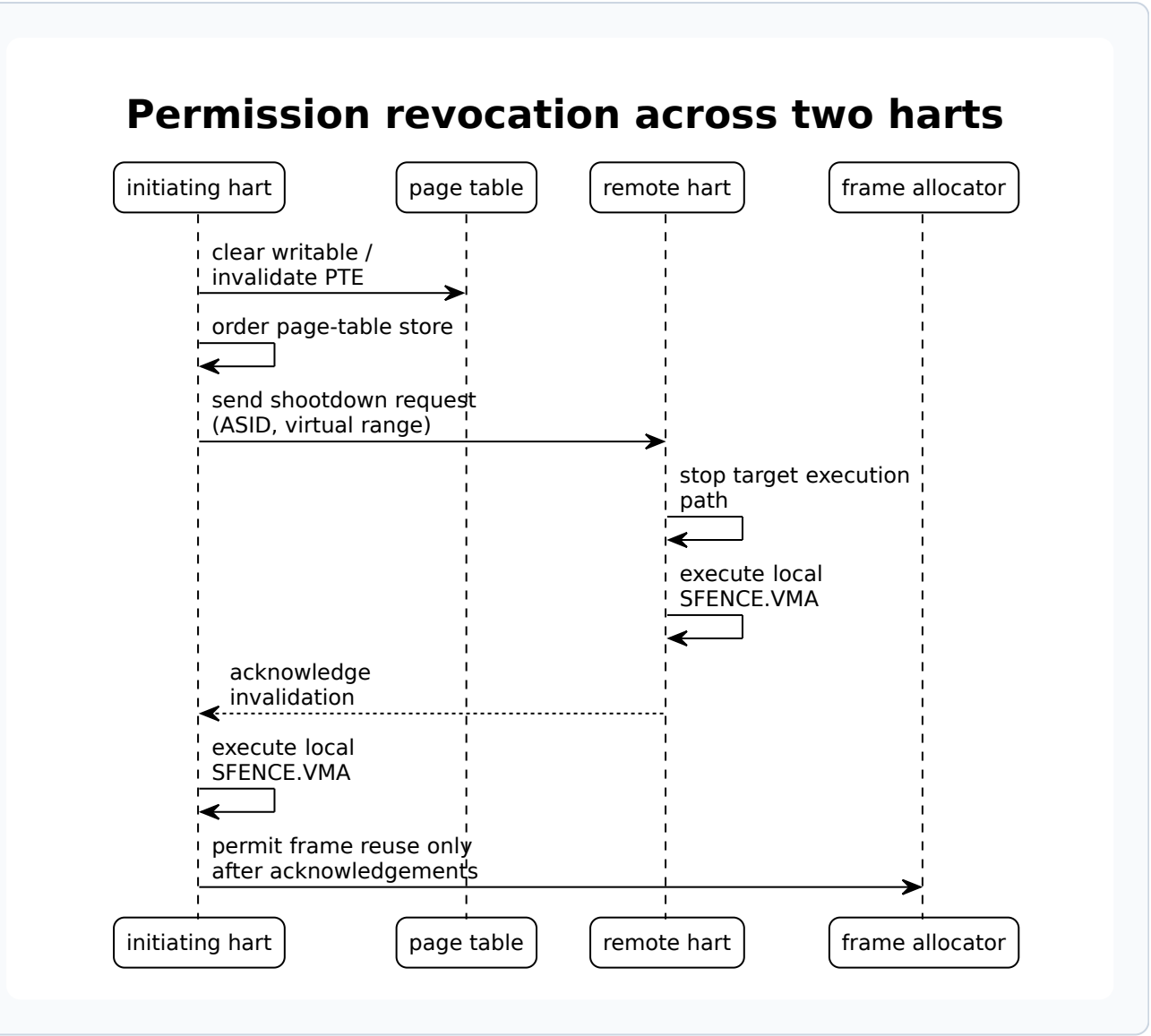
- `(A, read, 0x4A3)` selects VPN `0x4`, offset `0xA3`; A's TLB contains the mapping, so the result is frame `0x2`, physical `0x2A3`.
- `(B, read, 0x8A3)` selects VPN `0x8`, offset `0xA3`; the page is present but absent from B's fixed TLB, so a walk maps it to frame `0x6`, physical `0x6A3`.
- `(A, write, 0xCA3)` selects present VPN `0xC`, but its `r--` permission rejects the store.

Thus the prediction preserves the low eight bits `0xA3`; a TLB hit cannot make a write legal when the cached translation is read-only because permission state is part of the cached result. Equivalent states are fully valid. The decisive evidence is not the widget label: the answer must show key lookup, presence, and operation-specific authority as separate checks.

Context switches, ASIDs, and invalidation

Without ASIDs, a context switch that installs a new address space may require discarding untagged user translations. With ASIDs, translations from multiple address spaces can coexist and be selected by `(ASID, VPN)`. ASIDs optimize identity; they do not remove revocation. Reusing an ASID for a different page table requires invalidating translations associated with the old meaning.

RISC-V `SFENCE.VMA` orders the issuing hart's explicit page-table updates with that hart's later translations and invalidates translations selected by its operands. It is a **local-hart** operation. Revoking a mapping on a multicore system requires coordination with every hart that could use the address space.



The protocol's safety point is the final arrow. Reusing the frame after changing the PTE but before remote acknowledgement can expose the new owner's bytes through an old translation. Sending an interprocessor interrupt without waiting for completion is also insufficient: delivery is not invalidation.

Worked race: stale writable authority

1. Hart 1 runs process A and caches `(ASID 9, VPN 4) → PFN 21, RW`.
2. Hart 0 changes A's PTE to read-only and maps PFN 21 into process B as private data.
3. Hart 0 performs a local fence but sends no shootdown.
4. Hart 1 resumes A and stores through its cached `RW` entry.

The page-table snapshot appears secure, yet A modifies B's data. The defect is not "the TLB contains old data" in an abstract sense; it retains **old authority**. The repair must cover every hart on which A could have run, or prove that a hart never cached A's translations.

Transfer — optimize without weakening safety. A process has run only on harts 0 and 2 of a 64-hart machine. State a valid targeted shootdown and the evidence the kernel must maintain to avoid sending 62 unnecessary invalidations.

Track an address-space CPU mask that conservatively includes every hart that may hold its translations. Update the PTE, send range/ASID invalidations to harts 0 and 2, execute the corresponding local invalidation on the initiating hart when applicable, wait for acknowledgements, and only then complete revocation or frame reuse. The optimization is sound only if mask updates are synchronized with scheduling; a false negative is a security bug, while a false positive costs performance.

Counterexamples that separate the mechanisms

WARNING

Never equate a TLB miss with a page fault or a PTE write with immediate access revocation: a TLB miss is resolved in hardware or firmware when the leaf PTE is valid, and revoking permissions in memory leaves stale writable authority cached in remote harts' TLBs until shootdown completes.

Claim	Counterexample	Correct distinction
“TLB miss means the page is on storage.”	present PTE absent from TLB	cache miss versus not-present fault
“A valid PTE is a leaf.”	valid `R=W=X=0` entry	next-table pointer versus leaf
“ASIDs eliminate flushes.”	ASID reuse or permission downgrade	identity tagging versus revocation
“A larger page always improves performance.”	sparse writes dirty an entire coarse mapping and waste memory	reach versus granularity
“Updating the PTE revokes access.”	remote hart retains writable TLB entry	authoritative table versus cached authority
“More TLB entries give exactly proportional reach.”	set conflicts and mixed mappings	capacity bound versus effective coverage

REMARK

Baseline Sv39 radix trees and ASID-tagged TLBs provide the concrete hardware mechanism for process isolation; when a leaf PTE is marked invalid ($V = 0$) or lacks write permission ($W = 0$) for a legal region, control transfers to the trap handler studied in Page Faults, Copy-on-Write, and Memory-Mapped Files.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 4.2.1: Extracting Sv39 Three-Level Virtual Page Number Indices

Verification application

Given a 64-bit canonical virtual address `0x0000000123456789` in RISC-V Sv39:

1. Verify whether the address is canonical.
2. Extract the 12-bit intra-page byte offset.
3. Extract the 9-bit indices for VPN[2] (root level), VPN[1] (middle level), and VPN[0] (leaf level) in hexadecimal and decimal.

Sv39 requires bits 63:39 to equal bit 38, followed by three 9-bit VPN fields and a 12-bit offset.

1. **Canonical check:** Bit 38 is the most significant bit of VPN[2]. For `0x0000000123456789`, bit 38 is 0. Bits 63:39 are all 0, matching bit 38. Hence, the address is canonical.
2. **Page offset:** The lowest 12 bits (bits 11:0) are `0x789` (decimal 1929).
3. **VPN decomposition:** Shift right by 12 bits to discard the offset: `0x0000000123456`.
   - $\text{VPN}[0] = \text{bits } 20:12 = \texttt{0x123456} \;\&\; \texttt{0x1FF} = \texttt{0x056}$ (decimal 86).
   - $\text{VPN}[1] = \text{bits } 29:21 = (\texttt{0x123456} \gg 9) \;\&\; \texttt{0x1FF} = \texttt{0x11A}$ (decimal 282).
   - $\text{VPN}[2] = \text{bits } 38:30 = (\texttt{0x123456} \gg 18) \;\&\; \texttt{0x1FF} = \texttt{0x004}$ (decimal 4).

Exercise 4.2.2: Distinguishing Leaf Translations from Intermediate Radix Pointers

Verification application

An operating system kernel traverses an Sv39 page table. At level 1, the MMU encounters a PTE with bit pattern `0x0000000080000A0B`.

1. Decode the flag bits V, R, W, X, U.
2. Determine whether this PTE is an invalid entry, a pointer to a level-0 page table, or a superpage leaf translation.
3. If it is a leaf, state the page size mapped and verify if the physical alignment requirement is satisfied.

Inspect bits 0 (V), 1 (R), 2 (W), and 3 (X). For a leaf at level 1, PPN[0] must be zero.

1. **Flag decoding:** The lowest byte is `0x0B` (00001011_2).
   - Bit 0 (V) = 1 (valid).
   - Bit 1 (R) = 1 (readable).
   - Bit 2 (W) = 0 (not writable).
   - Bit 3 (X) = 1 (executable).
   - Bit 4 (U) = 0 (supervisor only).
2. **Classification:** Since $V = 1$ and at least one of R, W, X is set ($R = 1, X = 1$), this PTE is a leaf translation, not a next-level pointer (which would require $R = W = X = 0$).
3. **Superpage alignment:** At level 1, a leaf maps a 2 MiB superpage ($2^9 \times 4\ \mathrm{KiB}$). The physical page number occupies bits 53:10. Shift right by 10 bits: $\texttt{0x0000000080000A0B} \gg 10 = \texttt{0x200002}$. The lowest 9 bits of PPN ($\mathrm{PPN}[0]$) are $\texttt{0x200002}\ \&\ \texttt{0x1FF} = \texttt{0x002} \neq 0$. Because $\mathrm{PPN}[0] \neq 0$, the superpage is misaligned. Hardware will raise a page-fault exception on access rather than translating.

Exercise 4.2.3: Calculating Effective Memory Access Time with Multi-Level Walks

Construction application

A system features a single-level TLB with hit access time $t_{\text{TLB}} = 1.2$ ns and main memory access time $t_{\text{mem}} = 60$ ns. Address translation requires a 4-level page table walk in hardware on a TLB miss.

1. Formulate the equation for Effective Memory Access Time (EMAT) given TLB hit rate h.
2. Calculate EMAT for $h = 0.98$.
3. Calculate the minimum TLB hit rate required to keep translation overhead within 10% of bare memory access time (66 ns).

On a TLB hit, time is t_TLB + t_mem. On a miss, time is t_TLB + 4 * t_mem + t_mem.

1. **EMAT formulation:**
   - TLB hit: $T_{\text{hit}} = t_{\text{TLB}} + t_{\text{mem}}$.
   - TLB miss (4 memory accesses for the walk plus 1 for the data): $T_{\text{miss}} = t_{\text{TLB}} + 4 \cdot t_{\text{mem}} + t_{\text{mem}} = t_{\text{TLB}} + 5 \cdot t_{\text{mem}}$.
   - $\text{EMAT} = h \cdot T_{\text{hit}} + (1 - h) \cdot T_{\text{miss}} = t_{\text{TLB}} + t_{\text{mem}} + 4(1 - h)t_{\text{mem}}$.

2. **Numeric evaluation for $h = 0.98$:**

   $$\text{EMAT} = 1.2 + 60 + 4(1 - 0.98)(60) = 61.2 + 4(0.02)(60) = 61.2 + 4.8 =$$

3. **Threshold hit rate:** We require $\text{EMAT} \leq 66$ ns:

   $$61.2 + 240(1 - h) \leq 66 \implies 240(1 - h) \leq 4.8 \implies 1 - h \leq 0.02 \implies h$$

   A hit rate of at least 98.0% is required.

Exercise 4.2.4: TLB Reach Expansion via Megapages and Gigapages

Construction transfer

A processor TLB contains 512 entries.

1. Compute the TLB reach using standard 4 KiB base pages.
2. If an application database allocates 256 entries for 2 MiB superpages and 256 entries for standard 4 KiB pages, compute the new total TLB reach.
3. Explain why using 2 MiB superpages for an irregular 12 KiB data structure is counterproductive.

TLB reach is the number of entries multiplied by page size. Consider internal fragmentation for small structures.

1. **Base page reach:**

$$\text{Reach}_{\text{base}} = 512 \times 4 \text{ KiB} = 2048 \text{ KiB} = 2 \text{ MiB}.$$

2. **Mixed superpage reach:**

$$\text{Reach}_{\text{mixed}} = (256 \times 4 \text{ KiB}) + (256 \times 2 \text{ MiB}) = 1 \text{ MiB} + 512 \text{ MiB} = 513$$

   Using superpages expands the virtual address reach by more than 250$\times$ with identical hardware entry counts.

3. **Fragmentation drawback:** Mapping a 12 KiB structure with a 2 MiB superpage wastes $2048 \text{ KiB} - 12 \text{ KiB} = 2036 \text{ KiB} (> 99.4\%)$ of contiguous physical RAM to internal fragmentation. Furthermore, page-out or dirty-tracking operations must move or mark the entire 2 MiB chunk, wasting memory bandwidth.

Exercise 4.2.5: Proving Correct Ordering in Multi-Core TLB Invalidation Protocols

Proof synthesis

Let Core 0 update a PTE to revoke write permissions on physical frame F, intending to reassign F to an isolated process on Core 1. Prove that Core 0 must not allocate F until all remote cores have completed and acknowledged their TLB shootdown.

Show that a window exists where Core 1 writes through stale TLB authority if reuse precedes shootdown completion.

Assume for contradiction that Core 0 reallocates frame F to Process B before Core 1 acknowledges completion of the TLB invalidation.

1. Prior to revocation, Core 1 executes Process A and caches a valid, writable TLB entry: $(A, \mathrm{VPN}_A) \mapsto (F, \mathrm{RW})$.
2. Core 0 modifies the memory PTE for (A, VPN_A) to set $W = 0$ (or $V = 0$), and issues an asynchronous Inter-Processor Interrupt (IPI) to Core 1.
3. Without awaiting acknowledgment, Core 0 assigns frame F to Process B and begins initializing confidential data into F.
4. Before Core 1 handles the IPI, Process A on Core 1 executes a store instruction to VPN_A.
5. The MMU on Core 1 queries its local TLB, finds the cached translation with $W = 1$, and commits the store directly into physical frame F without walking the memory page table.
6. Process A has thus corrupted or observed Process B's confidential memory, violating the fundamental memory isolation invariant.

Therefore, the happens-before edge:

$$\text{Remote TLB Invalidation} \prec \text{Remote Acknowledgment} \prec \text{Frame Reallocation}$$

is strictly necessary for safety.

References & Further Reading

- RISC-V Privileged Architecture: Supervisor-Level ISA and Sv39
- RISC-V Privileged Architecture PDF, ratified snapshot 2024-04-11
- xv6-riscv revision 5 book, Chapters 3-4
- xv6-riscv `kernel/vm.c` at revision 5
- Linux kernel: Cache and TLB Flushing Under Linux

SECTION DETAIL

Page Faults, Copy-on-Write, and mmap

A 1 GiB process calls `fork()`, and the child immediately replaces its image with a small program. Eagerly copying 1 GiB delays the child and consumes memory for bytes it never reads. Pre-materializing every valid page avoids its initial not-present fault but pays work before demand exists. Deferring all allocation without recording region intent lets an invalid pointer masquerade as legitimate demand.

A page fault resolves that tension. It is a synchronous exception raised by the current instruction when address translation or access checking cannot complete. Some faults are policy requests—materialize a valid lazy page or make a private copy. Others are violations—unmapped address, forbidden operation, or access beyond the backing object. The same hardware event can represent either class; region metadata and access intent decide which.

> ***Fault-resolution invariant.*** *Retrying the faulting instruction is safe only after the kernel has installed a mapping that (1) belongs to the faulting region, (2) represents the correct backing object and byte offset, (3) grants no more authority than the region allows, and (4) is visible after required translation invalidation.*

A fault is evidence, not a diagnosis

The assessed machine uses precise, restartable RISC-V page faults. `sepc` identifies the faulting instruction; `scause` identifies the access class; `stval` supplies the faulting virtual address when the implementation reports it. Relevant synchronous cause codes are:

`scause` code	Access that failed	Typical policy distinction
12	instruction page fault	executable mapping, code loading, or execute denial
13	load page fault	readable mapping, demand-zero, or file read
15	store/AMO page fault	writable mapping, dirtying, or copy-on-write

Hardware does not know whether an absent PTE means "grow a heap," "load a file page," "guard page," "mapping removed," or "process bug." The kernel must look up the virtual-memory region that contains the address and apply its policy.

A useful classification key is the tuple

$$(\text{region},\ \text{access},\ \text{PTE state},\ \text{backing state}).$$

Region/PTE condition	Load	Store	Instruction fetch
anonymous, authorized, absent	allocate zeroed frame	allocate zeroed frame	reject unless region executable
private file, authorized, absent	load/cache file page	load then preserve private-write semantics	load if executable
present COW mapping	read shared frame	create or promote private writable frame	follow execute permission
present genuine read-only mapping	read	reject	follow execute permission
no containing region	reject	reject	reject
guard region	reject	reject	reject

Retrieval — state the missing evidence. A store page fault arrives at an address whose PTE has `W=0`. Why is "make it writable" an invalid generic handler?

`W=0` could encode a genuine read-only mapping, a COW mapping, a guard, or an invalid state. The handler needs region permission and a COW-specific marker or equivalent metadata. Promoting every read-only PTE grants write authority to code, constants, and shared read-only objects.

PAGE FAULT CLASSIFICATION AND COPY-ON-WRITE INVARIANTS

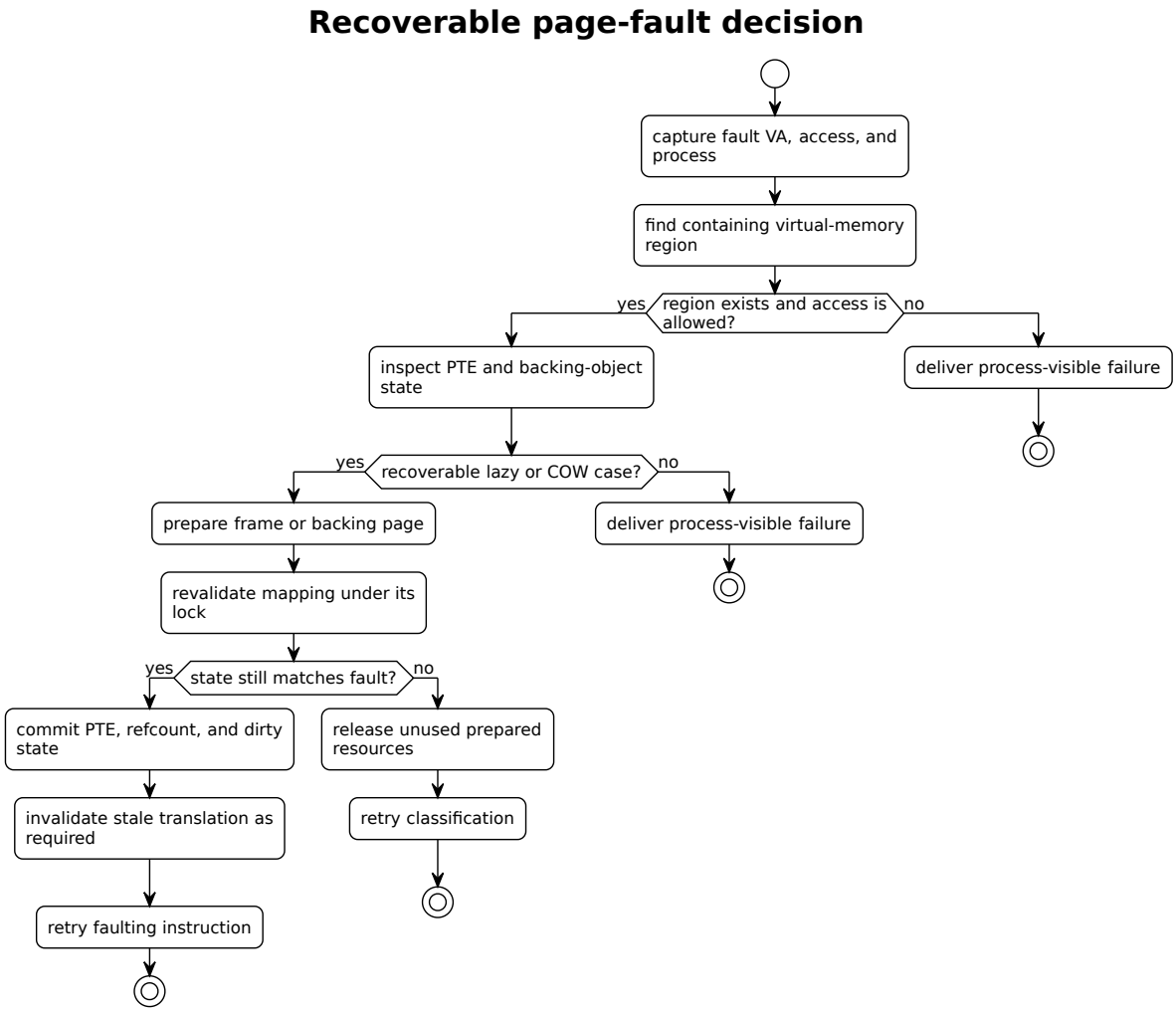
Evaluate each statement regarding page faults, copy-on-write fork semantics, and memory-mapped files:

Answer: When a sole surviving process writes to a copy-on-write page with refcount one, the kernel promotes the frame in place.; Fresh physical frames allocated for demand zero pages must be zeroed before publishing page table mappings to user space.

When refcount is 1, COW pages are promoted in place without copying. Zeroing before PTE publication prevents data disclosure. Accesses outside valid regions are fatal segment violations rather than demand-allocated pages, and MAP_PRIVATE explicitly discards modifications upon unmapping without altering the underlying file.

Derive a restartable fault protocol

Fault resolution crosses several ownership domains: process metadata, page tables, physical allocator, file/page cache, and scheduler. A robust protocol separates **validation**, **preparation**, **commit**, and **retry**.



The prepare-before-commit shape matters under memory pressure. If allocation or file I/O fails, the old mapping remains valid and no partial authority is exposed. Revalidation matters under concurrency: another thread may resolve, unmap, protect, or replace the region while the handler sleeps for memory or storage.

The handler also needs a bounded lock discipline. Holding a spinlock across file I/O or a sleeping allocator deadlocks or stalls unrelated CPUs. Dropping all protection without recording a generation permits committing a page into a region that has been unmapped and reused. A production kernel

solves this with layered locks, reference counts, region versions, or retry protocols; the assessed invariant is that commit rechecks the state it relied on.

Worked trace: anonymous demand-zero allocation

Region `[0x400000, 0x404000)` is anonymous, user read/write, non-executable. No pages are resident. A process stores `7` at `0x4012A0`.

1. `scause=15` identifies a store page fault; `stval=0x4012A0` identifies the address.
2. Region lookup succeeds and authorizes a store.
3. The absent PTE is an expected lazy state, not a COW or file-backed state.
4. The kernel allocates a 4 KiB frame and zeroes **all** bytes before user exposure. Zeroing prevents disclosure of a prior owner's data.
5. Under the mapping lock, the kernel rechecks that the region still covers the address and the PTE is still absent.
6. It installs a user read/write, non-executable leaf for page `0x401000`, performs architecture-required translation synchronization, and retries the store.
7. The retried store changes only offset `0x2A0`; every other byte remains zero.

If frame allocation fails at step 4, the PTE stays absent. Mapping an uninitialized frame and planning to zero it later violates confidentiality as soon as user mode can load from it.

Worked trace: a competing unmap

Thread T0 faults at `0x4012A0` and drops the region lock to allocate. T1 concurrently unmaps `[0x400000, 0x404000)` and maps a read-only file at the same virtual interval. T0 resumes with a zeroed frame.

Committing from the stale region snapshot would overwrite the new file mapping with a writable anonymous page. Revalidation detects that the region generation or backing object changed; T0 releases its frame and restarts classification. The restarted store now fails against the read-only file region.

Application — order the operations. Place these actions into a safe demand-zero sequence: `publish writable PTE`, `zero frame`, `allocate frame`, `validate region`, `revalidate region/PTE`, `retry instruction`, `invalidate stale translation if needed`.

Validate region → allocate frame → zero frame → revalidate region/PTE → publish writable PTE → invalidate any stale translation required by the architecture → retry. Allocation and zeroing may occur before the commit lock; publication cannot. Failure before publication releases the candidate frame. Failure after publication must use a defined rollback that first removes authority and completes invalidation; avoiding that complex path is why commit should contain no fallible work.

Copy-on-write preserves fork semantics lazily

POSIX requires a child's private mappings to behave as private after `fork()`: writes performed by either process after the fork are not visible through the other's private mapping. It does not require an eager physical copy. Copy-on-write (COW) implements the semantic contract with temporary physical sharing plus write faults.

For each writable private page at fork time:

1. map the same PFN into parent and child;
2. clear writable permission in **both** PTEs;
3. mark both mappings as COW using software metadata distinguishable from genuine read-only state;
4. increment the physical frame's reference count for the additional mapping; and
5. invalidate any parent translation that may still cache write authority before either process resumes.

The COW invariant is:

> *A COW frame may have multiple readers, but a process receives writable authority only to a frame that is private to the mapping or is being atomically converted to private ownership.*

Worked trace: parent and child diverge

Before `fork`, parent VPN 6 maps PFN 40 `RW`; frame 40 contains bytes `[A, B, C, …]` and has reference count 1.

After `fork`:

Mapping	PTE	Physical frame	Refcount
parent VPN 6	`R--, COW`	40	2
child VPN 6	`R--, COW`	40	2

The child stores `Z` at offset 1:

1. Hardware raises a store page fault because the child PTE is not writable.
2. Region and software metadata prove this was originally writable private memory and is now COW.
3. Refcount 2 means another mapping still needs frame 40. Before relying on its bytes outside the mapping lock, the handler acquires a temporary reference/pin to frame 40. It allocates frame 91 and copies all 4 KiB from the pinned source.
4. Under the relevant locks, it revalidates the child PTE, establishes frame 91's mapping reference, and maps child VPN 6 to frame 91 as `RW` without COW. It retains both the child's old mapping reference to frame 40 and the temporary pin while any stale child translation can still name frame 40.
5. It completes the required invalidation on every hart that could use the child's old translation. Only after those acknowledgements may it drop the child's old mapping reference and then the temporary pin. Frame 40 now has one mapping reference, held by the parent.
6. The store retries and changes frame 91 offset 1 to `Z`. Parent frame 40 remains `[A, B, C, …]`.

The parent later writes the same page. Since frame 40 now has refcount 1 and the parent's PTE is still COW, the kernel may remove the COW marker and restore write permission in place. No copy is needed, but translation invalidation still is.

The original permission must survive

Suppose a text page was read/execute before `fork`. Marking every shared page COW would let a store fault convert executable text into writable memory. The design needs to distinguish:

- **private writable, temporarily read-only because of COW**; and
- **intrinsically read-only or execute-only**.

Software-reserved PTE bits can carry a COW marker on an architecture that permits it, or region/page metadata can. The encoding choice is subordinate to the semantic proof: only a mapping whose policy includes write may resolve a COW store.

Concurrent COW faults

Two threads in one process can fault on the same COW page concurrently. Both may allocate candidate frames before either commits. At commit, one wins and installs its private frame. The other must detect that the PTE no longer matches the old COW state, release its unused candidate, and retry the store against the new writable mapping. Installing both candidates in sequence can leak one frame or lose a completed write.

A failure-atomic skeleton is:

```
lock mapping and old-frame references
snapshot := require current COW PTE
acquire a temporary pin while snapshot's mapping reference is valid
unlock
candidate := allocate and copy old frame
lock mapping and old-frame references
if current PTE != snapshot:
    unlock; free candidate; drop temporary pin; retry classification
establish candidate's mapping reference
install candidate as private writable
retain old mapping reference and temporary pin
unlock
invalidate every target translation and wait for completion
drop old mapping reference, then drop temporary pin
retry instruction
```

The two retained old-frame references have different meanings: one accounts for the PTE that was replaced, and one pins the snapshot while the handler works. Neither may be released in a way that makes the old frame reusable before stale translations are gone. A kernel may instead invalidate the old PTE and complete shootdown before publishing the candidate, but it must serialize competing faults through that transient state. The real implementation must define lock ordering, shootdown waiting, and the exact reference-count operations. The skeleton intentionally omits a complete kernel patch.

Transfer — allocation failure. A child faults on a shared COW page with refcount 2, but no frame is available. Which three states must remain unchanged, and why is “temporarily grant write to the shared frame” invalid?

The child's PTE must remain read-only COW, the parent's mapping must remain unchanged, and the shared frame's reference count/content must remain valid. Granting the child write authority would make its modification visible to the parent, violating private mapping semantics. The kernel may report an out-of-memory failure or apply a documented reclamation policy; it cannot weaken isolation to make the instruction complete.

mmap binds regions to memory objects

`mmap()` establishes a region whose bytes correspond to an anonymous object or to a file object's byte range. `prot` describes permitted accesses. `MAP_PRIVATE` and `MAP_SHARED` describe the disposition of writes, not whether initial physical pages are shared.

Mapping	Read fault	First authorized write	Persistence relation
anonymous private	supply zero page/frame	private writable frame	no file object
file `MAP_PRIVATE`	obtain file-backed page	COW to process-private frame	modification does not change underlying object
file `MAP_SHARED`	obtain shared file page	dirty shared page under VM/filesystem rules	modification targets underlying object

A private mapping can initially reference the same page-cache frame as other mappings. "Private" constrains the visibility of modifications; it does not demand a unique frame for every read.

Worked trace: one file, two mapping policies

A file contains 6,000 bytes. Page size is 4,096 bytes. Process P maps 8,192 bytes `MAP_PRIVATE | PROT_READ | PROT_WRITE`; process Q maps the same range `MAP_SHARED | PROT_READ | PROT_WRITE`.

1. P and Q read offset 100. Both may map the same page-cache frame; no private copy is required.
2. P writes offset 100. A COW fault gives P a private frame for the first file page. Q and the file object do not observe P's byte.
3. Q writes offset 5,000. Q modifies the shared second file page and marks it dirty according to the OS/filesystem contract.

4. Bytes from file offset 6,000 to the end of the last partial page are zero-filled under POSIX mapping semantics, and modifications there are not written to the file.
5. If a longer mapping included a whole page beyond the object and the process referenced that page, POSIX requires `SIGBUS` for the out-of-object whole-page access.

Closing the file descriptor after a successful mapping does not remove the mapping: POSIX specifies that the mapping holds an additional reference to the file until the mappings are gone.

Truncation is a concurrent object change

If another process changes the mapped file's size after `mmap`, POSIX leaves the effect on added or removed portions unspecified. A robust application cannot treat a concurrent truncation as an ordinary stable-memory event. Kernel internals still need coherent invalidation and fault behavior, but portable application reasoning must stay within the interface contract.

Application — classify without the slogan "fault equals disk." For each event, state whether disk I/O is required under the information given:

1. first write to a COW page whose source frame is resident;
2. first read of an anonymous demand-zero page;
3. read fault on a file page already in the page cache;
4. read fault on a nonresident file page;
5. store to a genuine read-only mapping.

1. No file read is required; memory allocation and a RAM copy may be required.
2. No file read is required; a zeroed frame or protected shared-zero-page mechanism is sufficient.
3. No storage read is required; install a translation to the resident cached page subject to permissions.
4. Storage I/O is required unless another cache/device layer already supplies the data.
5. No recovery I/O is justified; the access is rejected.

The diagnostic gap is conflating a translation exception with one particular backing-store action.

Fault-handler boundaries that prevent security defects

> **WARNING**
>
> Every page-fault handler is an authority gate between untrusted user execution and physical memory: publishing a PTE before zeroing a freshly allocated frame discloses residual kernel/tenant data, and resolving a kernel `copyin` / `copyout` fault without enforcing user-region permissions turns a syscall buffer pointer into an arbitrary kernel write primitive.

Zero before exposure

A newly allocated user frame may contain bytes from a previous process or kernel object. Zeroing after PTE publication creates a disclosure interval. The zeroing operation must complete before any user-readable mapping exists.

Validate arithmetic and object offsets

For file-backed mapping start v_0, fault address v, and file offset f_0, the backing byte begins at

$$f = f_0 + \mathrm{page_floor}(v - v_0).$$

Every subtraction, alignment, addition, and length check must be overflow-safe. An overflow that wraps into the beginning of a file can map an authorized virtual address to the wrong object bytes.

A timeout or kill does not revoke DMA or I/O

If a process is killed while a fault waits on storage, the in-flight I/O may still own a kernel buffer. The process mapping can be abandoned only through a path that preserves buffer and request lifetimes. Process cancellation is not device cancellation.

Kernel accesses to user memory need a policy

Copying syscall data through `copyin` or `copyout` may encounter lazy pages without arriving through the ordinary user-trap path. The kernel must either invoke the same validated fault mechanism or reject the copy according to its contract. Bypassing region checks because the CPU is privileged can turn a user pointer into kernel-chosen memory corruption.

REMARK

This lesson establishes the state transitions for demand paging, copy-on-write, and file-backed mappings when free frames are available; policy decisions for choosing an eviction victim when physical memory is exhausted follow in Replacement, Working Sets, and Thrashing.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 4.3.1: Demand-Zero Anonymous Page Fault State Transitions

Verification application

A user process invokes `sbrk(4096)` to expand its heap by one 4 KiB page. The kernel records the virtual allocation in its region descriptor list but leaves the hardware PTE invalid ($V = 0$).

1. State the hardware register values (`scause`, `sepc`, `stval`) when the user code first reads from the new virtual page address `0x00405000`.
2. Trace the kernel page-fault handler actions to resolve this fault cleanly.
3. Why must the newly allocated physical frame be zeroed before the PTE is set to valid?

scause 13 indicates a load page fault; stval holds the faulting address; sepc holds the faulting instruction PC.

1. **Hardware trap state:**
   - `scause`: 13 (Load page fault exception).
   - `stval`: `0x00405000` (the faulting virtual memory address).
   - `sepc`: points to the specific load instruction that triggered the fault.
2. **Kernel resolution steps:**
   - Trap entry saves user registers and queries `stval` and `scause`.
   - The handler searches the process virtual memory region tree and verifies that `0x00405000` falls within an authorized, readable anonymous heap region.
   - The kernel allocates a free physical frame F from the physical memory pool.
   - The kernel zeroes all 4096 bytes of frame F.
   - The kernel installs a leaf PTE mapping $\mathrm{VPN}(0x00405000) \mapsto F$ with $V = 1$, $R = 1, W = 1, U = 1$.
   - The kernel executes `SFENCE.VMA` (or issues appropriate TLB invalidation if required).
   - Trap return restores registers and executes `sret`, re-executing the faulting instruction at `sepc`, which now succeeds.
3. **Information disclosure defense:** If the frame is mapped into user space before being zeroed, a concurrent thread or immediate load instruction can read residual bytes left behind by prior processes or the kernel itself, creating a critical information leak vulnerability.

Exercise 4.3.2: Tracing Copy-on-Write Reference Counts and Private In-Place Promotion

Construction application

A parent process owns a private writable page mapped to physical frame F_1. The parent forks child C_1, which later forks child C_2. No process writes until after all forks complete.

1. State the PTE permissions (R, W) and physical frame reference count for F_1 immediately after the second fork.
2. Child C_1 executes a store to the page. Describe the fault handling, frame allocation (F_2), and updated reference counts.
3. Parent terminates. Child C_2 then executes a store to the page. Describe how the kernel handles this second write without copying.

On the second write, check if the remaining reference count is equal to one.

1. **State after two forks:**
   - Each process (Parent, C_1, C_2) maps F_1 with permissions $R = 1, W = 0$ (read-only COW).
   - The kernel reference count for frame F_1 is $\text{refcount}(F_1) = 3$.

2. **First store by C_1:**
   - C_1 triggers a store page fault (`scause` 15).
   - Handler verifies C_1 has write authority in its region descriptor, but $\text{refcount}(F_1) = 3 > 1$.
   - Handler allocates a fresh physical frame F_2, copies 4096 bytes from F_1 to F_2, and sets $\text{refcount}(F_2) = 1$.
   - Handler updates C_1's PTE to map to F_2 with $R = 1, W = 1$.
   - Handler decrements $\text{refcount}(F_1)$ from 3 to 2, flushes C_1's local TLB, and returns to user space.

3. **Termination and second store by C_2:**
   - When Parent terminates, its page table is unmapped, decrementing $\text{refcount}(F_1)$ from 2 to 1.
   - When C_2 writes, it triggers a store page fault on F_1.
   - Handler checks $\text{refcount}(F_1)$ and discovers it equals 1.
   - Because C_2 is the sole remaining owner of F_1, no memory copy is needed. The kernel directly promotes C_2's PTE in place by setting $W = 1$.
   - After invalidating C_2's TLB, the store completes on F_1 without allocating memory.

Exercise 4.3.3: Comparing MAP_PRIVATE and MAP_SHARED Write Semantics

Verification transfer

A program opens a 1 MiB file and maps it twice: region R_{priv} uses `MAP_PRIVATE`, and region R_{share} uses `MAP_SHARED`. Both regions specify read-write permissions (`PROT_READ | PROT_WRITE`).

1. A store is executed to byte offset `0x100` within R_{priv}. What are the effects on the underlying file and on other processes mapping the same file?
2. A store is executed to byte offset `0x100` within R_{share}. What are the effects on other processes mapping the same file and on disk storage?

MAP_PRIVATE creates an isolated copy-on-write page; MAP_SHARED writes to the shared page cache frame.

1. **MAP_PRIVATE store:** The store triggers a copy-on-write page fault. The kernel allocates a private anonymous frame, copies the file page contents into it, and updates the process PTE to point to the private frame with write permissions.
   - The underlying file is never modified.
   - Other processes mapping the file (whether private or shared) do not observe the store; the modification remains strictly private to the calling process and is discarded when unmapped.
2. **MAP_SHARED store:** The store completes directly on the shared page in the operating system page cache.
   - All processes that map the same file region (using `MAP_SHARED`) immediately observe the modified byte (subject to CPU cache coherency).
   - The kernel marks the underlying page cache page as dirty. The modified page will eventually be written back to the backing block device during periodic flushing (`sync`, `msync`, or dirty page writeback).

Exercise 4.3.4: Handling Memory Exhaustion during Copy-on-Write Allocation

Construction synthesis

Suppose two processes share a COW page frame F. Process A executes a store instruction, triggering a COW fault, but the physical frame allocator returns `NULL` because physical memory and swap are completely exhausted.

1. Why must the kernel NOT set $W = 1$ directly on frame F to let the store proceed?
2. Why can the kernel NOT ignore the fault and resume user execution at `sepc + 4`?
3. What is the correct, safe kernel failure behavior?

Setting W=1 destroys process isolation; skipping the instruction silently skips user computation.

1. **Violation of memory isolation:** If the kernel set $W = 1$ on the shared frame F without allocating a private frame, Process A's store would overwrite memory visibly belonging to Process B. This would destroy process memory isolation, leading to arbitrary data corruption or privilege escalation.
2. **Violation of execution correctness:** If the kernel skipped the store by resuming at `sepc + 4`, the user program would proceed under the false assumption that its memory update completed, causing silent mathematical and data structure corruption.
3. **Correct failure recovery:** The kernel must treat allocation failure as an unrecoverable resource exhaustion condition for the faulting thread or process. It either:
   - Blocks the thread until memory can be reclaimed by eviction or page out (if asynchronous reclamation is configured), or
   - Delivers a `SIGBUS` or `SIGSEGV` signal to Process A, or terminates Process A via the Out-Of-Memory (OOM) killer, leaving Process B's state and frame F completely intact.

Exercise 4.3.5: Proving Safe Kernel Copyin Execution over Lazy Virtual Memory

Proof synthesis

When a user program passes a pointer `buf` to the `read(fd, buf, count)` system call, the target buffer may reside on a valid but unmaterialized lazy anonymous page. Prove that the kernel must validate user region bounds and permissions rather than relying on raw supervisor loads and stores.

In supervisor mode, page fault protections may differ; passing an arbitrary kernel address must be rejected.

Assume for contradiction that the kernel implements `copyout` by directly dereferencing the user pointer in supervisor mode without explicit user-address validation:

1. In supervisor mode, the MMU may allow stores to supervisor pages unless restricted by hardware features like SMAP (Supervisor Mode Access Prevention).
2. If an adversarial user process passes a kernel data structure address (e.g., `buf = 0xFFFFFFFF80000000`), a naive supervisor dereference will read or write directly into kernel memory, allowing an unprivileged process to overwrite page tables or kernel text.
3. Furthermore, if `buf` points to an unmapped user address or a read-only code region (`PROT_READ`), a blind kernel write would either trigger an unexpected kernel-mode trap or overwrite read-only binaries.

Therefore, the kernel must execute `copyin` / `copyout` using specialized routines that:

- Explicitly verify that `[buf, buf + count)` lies strictly within the user virtual address range.
- Validate that the containing virtual memory region grants the requested access ($W = 1$ for `copyout`, $R = 1$ for `copyin`).
- Install a trap recovery fixup table so that any unexpected page fault inside the copy routine returns `-EFAULT` cleanly to the system call caller rather than panicking the kernel.

References & Further Reading

- RISC-V Privileged Architecture: trap causes and supervisor address translation

- POSIX.1–2024 `fork()`
- POSIX.1–2024 `mmap()`
- xv6–riscv revision 5 book, virtual memory and traps
- xv6–riscv `kernel/trap.c` at revision 5
- xv6–riscv `kernel/vm.c` at revision 5

SECTION DETAIL

Replacement, Working Sets, and Thrashing

Demand paging can admit more virtual memory than physical memory. Once every free frame is occupied, satisfying one not-present fault requires reclaiming or evicting something else. A poor victim may be referenced again immediately, causing another fault that evicts another hot page. Under sufficient pressure, CPUs spend their time servicing faults and storage traffic while application throughput collapses.

Replacement is not "choose the oldest page." It is an online decision under incomplete knowledge. The kernel must estimate which resident page has the lowest future value while accounting for dirty writeback, sharing, fairness, and the allocation domain from which the victim may be taken.

> ***Reclaim objective.*** *Free enough usable memory while minimizing future stall cost and preserving allocation isolation. A policy must be evaluated against a reference workload and cost model, not by its name.*

Reference strings make policy claims falsifiable

Let a **reference string** $r_1, r_2, \ldots, r_n$ record virtual page numbers in access order. A **resident set** R_t is the set of pages occupying the assessed frames after reference t. A reference is a **hit** if $r_t \in R_{t-1}$ and a **fault** otherwise. On a fault with no free frame, the policy selects a victim $v \in R_{t-1}$.

The small trace model assumes:

- every reference accesses one page;
- all pages have equal size;
- any evicted page can later be restored;
- replacement decision time is ignored;
- faults are counted equally unless dirty cost is introduced explicitly; and
- prefetching, sharing, NUMA, and concurrent references are absent.

These assumptions isolate victim choice. They do not model end-to-end program time.

Four comparison policies

Policy	Victim rule	Knowledge required	Principal boundary
FIFO	page loaded earliest	arrival order	can evict a frequently used old page
exact LRU	page referenced least recently	total recency order	metadata/coordination cost is high
OPT/MIN	page whose next use is farthest in the future	future trace	offline lower bound, not deployable online
CLOCK/second chance	first candidate whose reference bit is clear	approximate recent-use bit	scan rate and bit sampling affect quality

OPT is useful because it separates "the workload inherently needs many faults" from "the online policy chose poorly." It is not an implementation proposal.

Retrieval — recover the online boundary. Why can OPT be used in an evaluation even though a kernel cannot implement it for an arbitrary future workload?

An offline experiment already has the complete recorded trace, so it can compute the minimum achievable fault count for that trace and frame capacity. The gap between an online policy and OPT is evidence about victim decisions. Deploying OPT would require knowledge of future references that the running kernel does not possess.

FIFO can get worse when memory increases

Consider the reference string

```
1 2 3 4 1 2 5 1 2 3 4 5
```

Worked trace: FIFO with three frames

The queue is written oldest → newest.

Ref	Frames/queue after access	Result	Evicted	Faults
1	`1`	fault	—	1
2	`1 2`	fault	—	2
3	`1 2 3`	fault	—	3
4	`2 3 4`	fault	1	4
1	`3 4 1`	fault	2	5
2	`4 1 2`	fault	3	6
5	`1 2 5`	fault	4	7
1	`1 2 5`	hit	—	7
2	`1 2 5`	hit	—	7
3	`2 5 3`	fault	1	8
4	`5 3 4`	fault	2	9
5	`5 3 4`	hit	—	9

With four frames, FIFO faults **10** times on the same trace. The larger resident set changes queue order so that pages 1–5 repeatedly displace one another near the end. This is Bélády's anomaly: increasing allocated frames can increase faults for a non-stack policy.

> ***Stack property.*** *A replacement algorithm has the stack property if, after every common reference prefix, the resident set with k frames is a subset of the resident set with $k+1$ frames.*

Exact LRU has this property. Maintain all referenced pages in one recency order. The k-frame resident set is the first k pages of that order; the $(k+1)$-frame set is the first $k+1$. Therefore every hit with k frames is also a hit with $k+1$, so adding a frame cannot increase exact-LRU faults. FIFO's resident sets are not prefixes of one shared order, so the proof does not apply.

PAGE REPLACEMENT AND STACK PROPERTIES

Evaluate each statement regarding page replacement policies, stack properties, and thrashing:

Answer: Replacement algorithms satisfying the stack property guarantee that increasing physical frames cannot increase faults.; The CLOCK second-chance algorithm uses hardware accessed bits to approximate LRU with bounded pointer traversal cost.

Stack algorithms obey the inclusion property, which mathematically prevents Belady's anomaly. CLOCK approximates LRU using access bits. FIFO violates the stack property and can produce Belady's anomaly, while thrashing occurs when aggregate working sets exceed physical memory capacity, requiring admission control rather than mere victim policy tweaks.

Experiment: force a non-monotonic fault curve

The simulator implements FIFO, exact LRU, and offline OPT for fixed reference strings. It counts all faults equally, assumes clean pages, and uses two to five frames in one global pool. Its “changing working set” and “cyclic scan” traces are deterministic examples, not workload measurements.

Prediction — write the number before moving the frame control. For the Bélády trace under FIFO, will four frames produce fewer, equal, or more faults than three? State which invariant would rule out your answer for exact LRU.

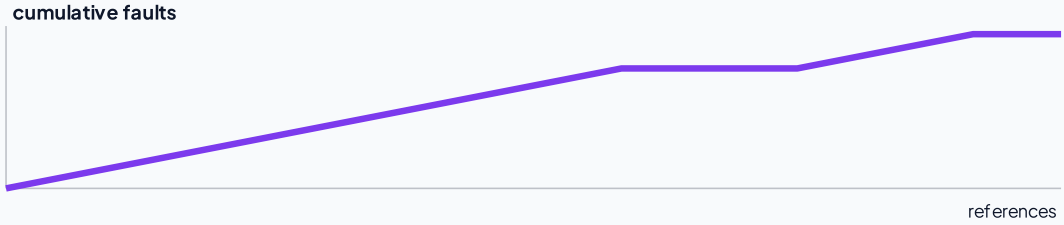
RUN A PAGE-REPLACEMENT POLICY AGAINST LOCALITY

FRAMES AFTER 12 REFERENCES

frame 0	5
frame 1	3
frame 2	4

Hit: frame contents unchanged

1 2 3 4 1 2 5 1 2 3 4 5



Static state: FIFO with three frames on the Belady trace produces 9 faults. Compare four frames interactively; stack algorithms such as LRU do not exhibit this anomaly.

Transfer — construct a policy challenge. Select the cyclic scan and compare exact LRU with OPT at two and five frames. Explain why recency performs poorly even though its stack property still holds. Then describe one trace shape where LRU should approach OPT.

For the prediction, FIFO rises from 9 faults with three frames to 10 with four; exact LRU's stack property rules out such an increase. On the six-page cyclic scan, exact LRU faults 12 times at both two and five frames. OPT faults 10 times at two frames and 7 times at five. With five frames, LRU's least-recent page is the one needed next after the current insertion, so every reference misses even though adding capacity has not made the count worse. A trace with a compact repeatedly reused set plus rare one-time pages gives recency useful evidence and can make LRU approach OPT.

Locality produces a working-set demand

Programs usually move through phases: a loop touches a compact set of code and data; a later phase touches another set. A **working set** estimates the pages in active use over a recent window. For window length Δ references,

$$W(t, \Delta) = \{r_i \mid t - \Delta < i \le t\}, \qquad w(t, \Delta) = |W(t, \Delta)|.$$

The window is a model parameter. Too small a Δ omits pages whose reuse interval is slightly longer; too large a Δ mixes phases and overestimates current demand.

Worked trace: a phase boundary

Use references

```
1 2 1 3 2 1  |  8 9 8 10 9 8
```

At $t = 6$ with $\Delta = 5$, the window is `2 1 3 2 1`, so $W(6, 5) = \{1, 2, 3\}$ and $w = 3$. At $t = 9$, the window is `2 1 8 9 8`, so $W(9, 5) = \{1, 2, 8, 9\}$ and $w = 4$: the window straddles the transition. At $t = 12$, the window is `9 8 10 9 8`, so $W(12, 5) = \{8, 9, 10\}$ and $w = 3$.

Allocating only two frames to either steady phase forces recurrent faults. Allocating four because of the transient boundary may be reasonable if phase changes are frequent, but it is not evidence that every phase needs four.

Application — compare windows. For the same trace at $t = 12$, compute $W(t, 3)$ and $W(t, 8)$. Which estimate responds faster to a phase change, and which retains stale pages longer?

The final three references are `10 9 8`, so $W(12, 3) = \{8, 9, 10\}$. The final eight are `2 1 8 9 8 10 9 8`, so $W(12, 8) = \{1, 2, 8, 9, 10\}$. The shorter window adapts faster but may miss a page with a reuse distance greater than three; the longer window retains pages 1 and 2 from the old phase and therefore reacts more slowly.

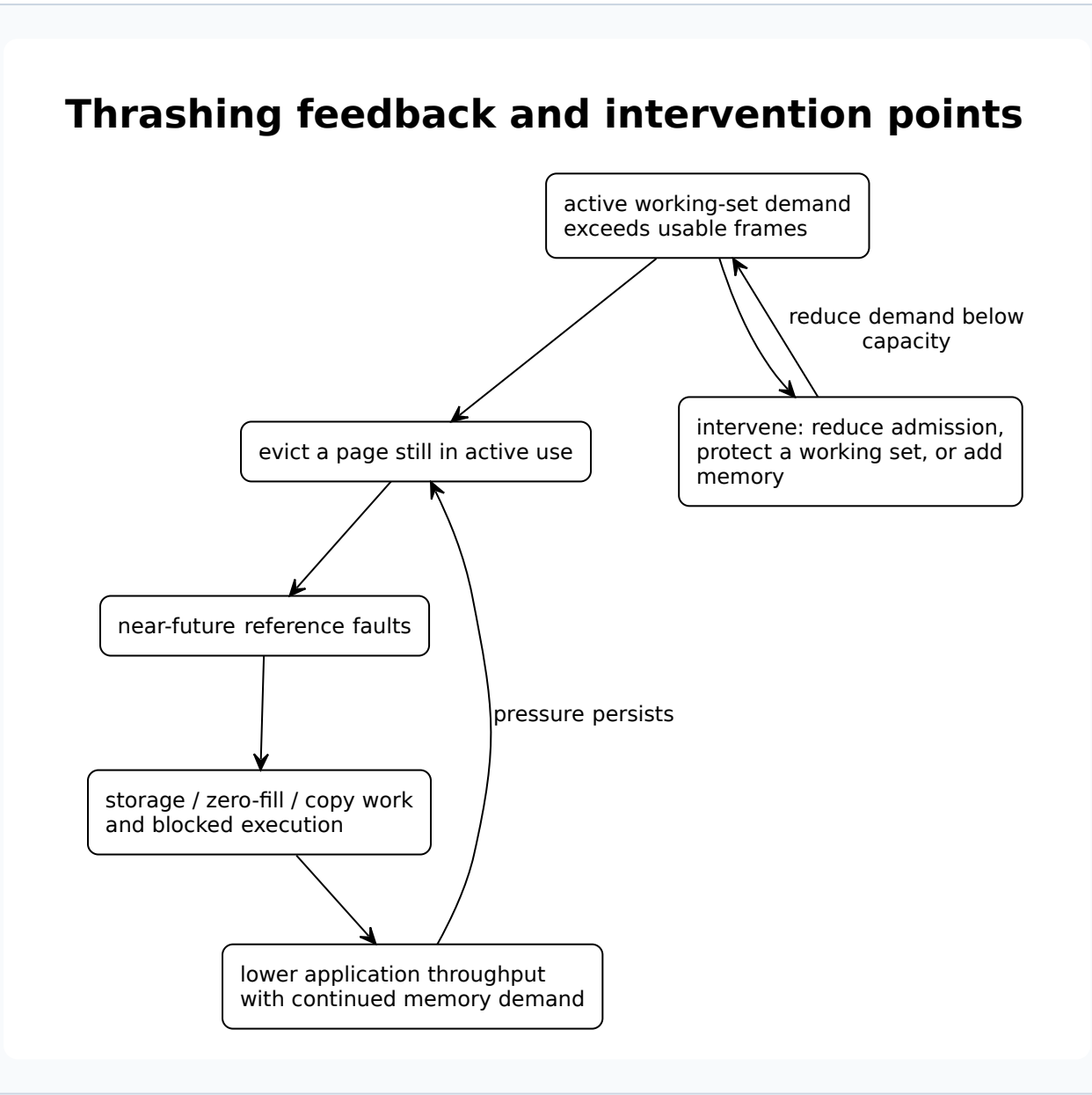
Thrashing is a system state, not a large fault count alone

Suppose m processes have estimated working-set sizes $w_i(t, \Delta)$ and the allocation domain has F usable frames. If

$$D(t, \Delta) = \sum_{i=1}^{m} w_i(t, \Delta) > F,$$

the domain cannot keep all estimated working sets resident. Repeatedly stealing a needed page from one process to satisfy another can create a positive feedback loop: faults increase, blocked processes release CPUs, runnable processes touch more nonresident pages, and I/O queues grow.

This inequality is a diagnostic model, not a universal threshold. Shared pages may reduce physical demand; inaccurate windows distort D; clean file pages and dirty anonymous pages have different reclaim costs. The key signal is loss of useful progress per unit of fault/reclaim work.



Worked diagnosis: multiprogramming overload

Three jobs have stable working-set estimates of 40, 35, and 30 frames. The cgroup has 90 usable frames. Their combined demand is 105. Equal allocation gives 30 frames each: the third job fits, while the first two continually lose ten and five active pages. Giving 40/35/15 protects two jobs but makes the third fault heavily. No replacement ordering creates 15 missing frames.

Admitting only the first two jobs uses 75 frames and leaves 15 for kernel/reclaim variation. Throughput may increase even though fewer jobs run concurrently. The defensible claim is not "suspend the largest process"; it is "reduce simultaneous working-set demand below measured capacity, then compare completed work and tail latency."

Transfer — distinguish pressure from thrashing. Name two measurements that show productive progress and two that show reclaim cost. Explain why CPU utilization alone is ambiguous.

Productive measures include completed transactions, retired application work units, or useful bytes processed. Cost measures include major/minor fault rate, reclaim scans, swap/file reads, dirty writeback, and time blocked on faults. CPU utilization can stay high because kernel reclaim and fault handling consume CPU even while application throughput falls; it can also fall because tasks block on storage. A diagnosis needs both progress and cost over the same interval.

CLOCK approximates recency with bounded metadata

Exact LRU needs a total recency update on every reference, which is expensive on multicore systems. CLOCK arranges candidate frames in a circle and maintains an accessed/reference bit `A`:

1. inspect the frame under the hand;
2. if `A=1`, clear it and advance, granting a second chance;
3. if `A=0` and the page is eligible, select it;
4. if the page is dirty, either schedule writeback and continue or account for its higher cost;
5. advance the hand after selection.

RISC-V permits implementation-dependent A/D update schemes within architectural rules; a kernel must follow the architecture's contract when sampling or clearing those bits. A software-managed bit or fault-assisted scheme can replace hardware updates, but it changes overhead.

Worked trace: second chances and dirty cost

Four frames under the hand contain:

Frame	Page	`A`	Dirty	Eligible
0	7	1	no	yes
1	2	0	yes	yes
2	9	0	no	no, pinned for I/O
3	4	0	no	yes

Starting at frame 0, CLOCK clears page 7's `A` and advances. Page 2 is unreferenced but dirty. If the policy minimizes immediate stall, it can schedule page 2 for writeback and continue. Page 9 is ineligible. Page 4 is clean, unreferenced, and eligible, so it is the immediate victim. A later pass may reclaim page 2 after writeback completes.

Calling page 2 "the correct victim" from recency alone ignores the assessed objective. If clean and dirty evictions have different latency, policy needs a cost rule in addition to recency.

Application — mutate one bit. If page 4 has `A=1` instead, trace the hand until a victim is found under the rule "schedule dirty writeback but do not block on it; pinned pages are skipped."

Frame 0: clear page 7 and advance. Frame 1: schedule page 2 writeback and advance. Frame 2: skip pinned page 9. Frame 3: clear page 4 and advance. On the next revolution, page 7 now has `A=0`, is clean and eligible, so page 7 is selected—unless a real reference set its bit again before the hand returned. Omitting that concurrency caveat treats CLOCK as a fixed-reference batch algorithm.

Production reclaim adds allocation and page classes

WARNING

Evicting a dirty page is not a single-step free operation: reclaiming a dirty anonymous or file-backed frame requires asynchronous writeback to swap or the filesystem before the frame can be reassigned, whereas clean file-cache pages can be reclaimed immediately if unmodified.

Real kernels cannot place every page in one undifferentiated queue:

- anonymous pages may require swap before eviction;
- clean file-cache pages can often be discarded and reread;
- dirty file pages require writeback and ordering;
- shared pages serve several mappings;
- pinned or unevictable pages are not candidates;

- memory cgroups and NUMA nodes constrain where reclaim occurs; and
- refault observations can reveal that recently evicted pages were still hot.

Linux's documented Multi-Gen LRU groups pages by estimated recency generations and provides working-set/refault-oriented controls. Its existence is evidence that production reclaim is an adaptive subsystem, not evidence that one named policy is universally optimal.

Counterexample: global efficiency can violate isolation

A global policy may evict pages from a latency-sensitive service because a batch job generates more faults. Total faults might fall while the service misses its latency objective. A local or cgroup-bounded policy can preserve allocation isolation at some cost to global utilization. The correct scope follows the resource contract and workload objective.

> **REMARK**
>
> Working-set theory $\sum_i |W_i(t,\tau)| \le M$ explains why local replacement policy changes cannot cure thrashing when aggregate active-page demand exceeds physical memory M; resolving thrashing requires admission control, working-set reduction, or per-cgroup memory isolation.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 4.4.1: Tracing Belady's Anomaly Under FIFO Replacement

Verification application

Consider the classic reference sequence $\sigma = 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5$.

1. Compute the total page fault count under FIFO replacement with 3 physical frames.
2. Compute the total page fault count under FIFO replacement with 4 physical frames.
3. State the formal definition of Belady’s anomaly demonstrated by this reference trace.

Step through arrival orders in a FIFO queue. With 3 frames, faults occur 9 times; with 4 frames, faults occur 10 times.

1. **FIFO with 3 frames:** Tracing the queue (oldest to newest):
    - Ref 1: [1], Fault 1.
    - Ref 2: [1, 2], Fault 2.
    - Ref 3: [1, 2, 3], Fault 3.
    - Ref 4: [2, 3, 4], Fault 4 (evicts 1).
    - Ref 1: [3, 4, 1], Fault 5 (evicts 2).
    - Ref 2: [4, 1, 2], Fault 6 (evicts 3).
    - Ref 5: [1, 2, 5], Fault 7 (evicts 4).
    - Ref 1: [1, 2, 5], Hit (1 resident).
    - Ref 2: [1, 2, 5], Hit (2 resident).
    - Ref 3: [2, 5, 3], Fault 8 (evicts 1).
    - Ref 4: [5, 3, 4], Fault 9 (evicts 2).
    - Ref 5: [5, 3, 4], Hit (5 resident). Total faults with 3 frames = 9.
2. **FIFO with 4 frames:**
    - Ref 1: [1], Fault 1.
    - Ref 2: [1, 2], Fault 2.
    - Ref 3: [1, 2, 3], Fault 3.
    - Ref 4: [1, 2, 3, 4], Fault 4.
    - Ref 1: [1, 2, 3, 4], Hit.
    - Ref 2: [1, 2, 3, 4], Hit.
    - Ref 5: [2, 3, 4, 5], Fault 5 (evicts 1).
    - Ref 1: [3, 4, 5, 1], Fault 6 (evicts 2).
    - Ref 2: [4, 5, 1, 2], Fault 7 (evicts 3).
    - Ref 3: [5, 1, 2, 3], Fault 8 (evicts 4).
    - Ref 4: [1, 2, 3, 4], Fault 9 (evicts 5).
    - Ref 5: [2, 3, 4, 5], Fault 10 (evicts 1). Total faults with 4 frames = 10.

3. **Belady's anomaly definition:** Belady's anomaly is the counter-intuitive phenomenon whereby increasing the physical page frame allocation for a given reference string results in a strictly greater number of page faults under a non-stack replacement algorithm.

Exercise 4.4.2: Proving Stack Inclusion and Anomaly Immunity for Exact LRU

Proof synthesis

Let $R_k(t)$ denote the resident set of pages in physical memory after reference t using exact Least Recently Used (LRU) with k frames.

1. Prove by induction on t that $R_k(t) \subseteq R_{k+1}(t)$ for all $t \geq 1$ (the inclusion / stack property).
2. Deduce that exact LRU can never exhibit Belady's anomaly for any reference string or capacity.

Express R_k(t) as the set of the k most recently referenced unique pages.

1. **Proof of the stack inclusion property:** At any time t, let $S(t) = \langle p_1, p_2, \ldots, p_u \rangle$ be the sequence of unique pages referenced up to time t, ordered strictly by decreasing recency of their most recent reference (so $p_1 = r_t$ is the most recently referenced page). By definition of exact LRU:

   - For a memory of capacity k, the resident set consists of the first $\min(k, u)$ elements of $S(t)$:

   $$R_k(t) = \{p_1, p_2, \ldots, p_{\min(k,u)}\}.$$

   - For a memory of capacity $k+1$, the resident set consists of the first $\min(k+1, u)$ elements of $S(t)$:

   $$R_{k+1}(t) = \{p_1, p_2, \ldots, p_{\min(k+1,u)}\}.$$

   Since $\min(k, u) \leq \min(k+1, u)$, it follows immediately from the definition of subset that:

   $$R_k(t) \subseteq R_{k+1}(t) \quad \text{for all } t \geq 1.$$

2. **Deduction of anomaly immunity:** Suppose page reference r_t results in a cache hit under k frames. Then $r_t \in R_k(t-1)$. By the inclusion property proved above, $R_k(t-1) \subseteq R_{k+1}(t-1)$, which immediately implies $r_t \in R_{k+1}(t-1)$. Thus, any reference that is a hit with k frames is guaranteed to also be a hit with $k+1$ frames. The total fault count with $k+1$ frames must therefore be less than or equal to the fault count with k frames ($F_{k+1} \leq F_k$). Consequently, exact LRU can never suffer Belady's anomaly.

Exercise 4.4.3: Simulating CLOCK Hand Traversals with Dirty Page Writeback

Construction application

A circular buffer of 4 frames contains pages $P = [7, 2, 9, 4]$ at indices $0, 1, 2, 3$. The CLOCK pointer currently points to index 0. The hardware access (A) and dirty (D) bits are:

- Index 0 (page 7): $A = 1, D = 0$.
- Index 1 (page 2): $A = 0, D = 1$.
- Index 2 (page 9): $A = 1, D = 1$ (pinned for active DMA).
- Index 3 (page 4): $A = 0, D = 0$.

Trace the CLOCK algorithm to select a victim to satisfy an incoming fault for page 5 under the policy: "advance hand, clear accessed bits, do not evict pinned pages, schedule dirty pages for writeback without blocking, and evict the first clean unreferenced eligible page."

Check each frame in order from the current hand position.

1. **Index 0 (page 7):** $A = 1$. The algorithm clears the accessed bit ($A \leftarrow 0$) and advances hand to index 1.
2. **Index 1 (page 2):** $A = 0, D = 1$. The page has not been recently accessed, but it is dirty. Under the non-blocking policy, the kernel issues an asynchronous disk writeback request for page 2 and advances the hand to index 2 (without immediately stalling on writeback).
3. **Index 2 (page 9):** Pinned for active DMA. The algorithm skips pinned frames and advances hand to index 3.
4. **Index 3 (page 4):** $A = 0, D = 0$, unpinned. The page has $A = 0$ and is clean ($D = 0$). The algorithm selects page 4 as the immediate victim frame. The pointer is left pointing to index 0 (circular advance: $(3 + 1) \bmod 4 = 0$), and page 5 is loaded into frame 3 with $A = 1, D = 0$.

Exercise 4.4.4: Computing Denning Working-Set Sizes Across Reference Windows

Construction transfer

Let an operating system process reference virtual pages according to the sequence:

$$\sigma = \langle 1, 2, 1, 3, 4, 1, 2, 3, 2, 4, 5, 1 \rangle.$$

1. Using Denning's working-set model with window size $\Delta = 4$, compute the working set $W(t, 4)$ and its cardinality $|W(t, 4)|$ for time steps $t = 4, 8, 12$.
2. If the operating system allocates only 2 physical frames to this process during the interval $t \in [8, 12]$, explain why thrashing is inevitable.

The working set W(t, Delta) is the set of distinct pages referenced in the interval [t - Delta + 1, t].

1. **Working-set evaluations:**
   - At $t = 4$, the window is the last 4 references (indices 1 to 4): $\{1, 2, 1, 3\}$. The distinct pages are $W(4, 4) = \{1, 2, 3\}$, so $|W(4, 4)| = 3$.
   - At $t = 8$, the window is references at indices 5 to 8: $\{4, 1, 2, 3\}$. The distinct pages are $W(8, 4) = \{1, 2, 3, 4\}$, so $|W(8, 4)| = 4$.
   - At $t = 12$, the window is references at indices 9 to 12: $\{2, 4, 5, 1\}$. The distinct pages are $W(12, 4) = \{1, 2, 4, 5\}$, so $|W(12, 4)| = 4$.

2. **Thrashing onset:** During $t \in [8, 12]$, the process actively demands 4 distinct pages ($|W(t, 4)| = 4$) to execute its phase loop. If the OS allocates only 2 frames ($M = 2 < 4$), every memory access requiring one of the unrepresented working-set pages triggers a page fault. Every fault necessarily evicts a page that will be demanded again within the next 1–3 instructions. The process spends virtually 100% of its execution time waiting on storage I/O, which is the exact definition of thrashing.

Exercise 4.4.5: Diagnosing System-Wide Thrashing and Sizing Allocation Domains

Description synthesis

Explain why aggregate working-set overload $\sum_{i=1}^{N} |W_i(t, \Delta)| > M_{\text{tota}}$ causes a sudden cliff in system throughput rather than a gradual linear degradation. Describe how memory cgroups and process suspension (swapping) restore equilibrium.

Consider the feedback loop between page-fault handling latency, queue buildup, and CPU utilization collapse.

- **The non-linear thrashing cliff:** When $\sum |W_i| \leq M_{\text{total}}$, page fault rates are low, and the swap/storage device queue is largely idle. When aggregate working-set demand crosses the physical threshold M_{total}, processes continuously evict each other's active working-set pages. Each evicted page triggers a synchronous disk read latency ($50\ \mu\text{s}$ to $10\ \text{ms}$). Because threads block waiting on I/O, the operating system scheduler attempts to compensate by dispatching other runnable threads, which immediately fault as well. Storage request queues explode, I/O latency compounds, and CPU utilization collapses to near zero. Throughput drops precipitously from near 100% to near 0% across a tiny increase in working-set demand.
- **Remediation mechanisms:**
  1. **Admission control / suspension:** The kernel suspends one or more processes entirely (moving their active threads to swapped state) until $\sum_{\text{active}} |W_i| \leq M_{\text{total}}$. The remaining processes run to completion at full speed without thrashing, after which suspended tasks are resumed.
  2. **Memory cgroups (resource isolation):** By placing processes into bounded cgroups with `memory.max` and `memory.high` thresholds, the kernel isolates memory pressure. A runaway batch job thrashes only within its own private quota without evicting pages belonging to critical interactive or production services.

References & Further Reading

- L. A. Bélády, R. A. Nelson, and G. S. Shedler, "An anomaly in space-time characteristics of certain programs running in a paging machine" (1969)
- Peter J. Denning, "The Working Set Model for Program Behavior" (1968)
- Peter J. Denning, working-set primary-publication index
- Linux kernel documentation: Multi-Gen LRU
- RISC-V Privileged Architecture: accessed and dirty PTE state

Devices and I/O

SECTION DETAIL

Device I/O, Interrupts, and DMA

A driver fills a descriptor with a buffer address and length, advances a producer index, and rings a device doorbell. The CPU reordered the producer-index store before the descriptor stores. The device observes new work, reads the old length and address, and transfers data into a frame that has already been reassigned.

The defect is not "DMA is unsafe." The driver published ownership before the work description was complete. Device I/O is a concurrent protocol between agents with different clocks, caches, and failure modes. Correctness comes from explicit state transitions and ordering edges.

> ***Buffer-ownership invariant.*** *At every instant, the protocol identifies which agent may read or write each descriptor and data buffer. Ownership changes only after all prior writes required by the new owner are visible, and reclamation occurs only after the old owner can no longer access the object.*

A device contract precedes driver code

A driver needs more than register addresses. Its contract must define:

Contract field	Questions that must have one answer
register semantics	width, alignment, reset value, read/write side effects, reserved bits
command publication	which fields are written first; what store or I/O barrier orders them
device ownership	when the device may read descriptors or write buffers
completion	status encoding, interrupt condition, acknowledgment sequence, batching
queue capacity	full/empty test, wrap rule, maximum outstanding work
failure	malformed status, timeout meaning, reset/quiescence, late completion
concurrency	which locks protect software metadata; which fields the device changes independently

Memory-mapped I/O (MMIO) gives device registers addresses in a processor-visible range. Register access is not ordinary RAM access: reads and writes can have side effects, widths can be constrained, and the platform defines ordering and caching attributes. In C, `volatile` can stop the compiler from removing or merging an access, but it does not create a cross-CPU lock, a device memory barrier, or a DMA ownership protocol. Kernel code should use architecture/platform MMIO accessors and barriers whose contract matches the device.

Programmed I/O (PIO) has the CPU copy data through registers or memory. **Direct memory access (DMA)** lets a device transfer between device-visible memory and itself after the CPU supplies descriptors. DMA removes a CPU copy loop from the data path; it adds another memory-accessing agent whose authority and lifetime must be controlled.

Retrieval — separate three guarantees. For an MMIO command register, state what `volatile`, a spinlock, and a device write barrier each establish. None may be described as a substitute for the other two.

- `volatile` constrains compiler treatment of the access according to the C implementation; it does not by itself order another core or device.
- A spinlock serializes participating CPU threads and usually carries CPU memory-ordering semantics; the device does not acquire it.
- The architecture/platform's device barrier orders specified CPU memory operations with respect to device observation.

Confusing these layers often produces code that passes on one machine because its memory model is stronger than the driver's stated assumptions.

DEVICE MEMORY ACCESS AND DMA ORDERING INVARIANTS

Evaluate each statement regarding memory-mapped IO registers, volatile qualifiers, and DMA synchronization:

Answer: The C volatile qualifier prevents compiler reordering but does not issue hardware memory barriers to devices.; Drivers must execute a device write memory barrier before writing the doorbell to publish DMA descriptors.

volatile informs only compiler optimizations, not CPU-device bus synchronization. A hardware write barrier ensures descriptors are completely visible in RAM before the doorbell is rung. Cache coherence keeps caches in sync but does not enforce operation ordering across out-of-order execution, and reusing a buffer before device quiescence invites memory corruption.

Polling and interrupts trade work for notification latency

Polling repeatedly reads status. It can give bounded detection delay when a core polls at a controlled period, but consumes CPU work even when no event arrives. An **interrupt** transfers control when a device signals an event; it avoids idle polling but pays entry, dispatch, acknowledgment, and cache-disruption costs per notification.

For a periodic poll every P seconds, uniformly distributed event arrival gives expected detection delay $P/2$ and worst-case delay near P. If one poll costs C_p CPU seconds, polling occupancy is approximately

$$U_p = \frac{C_p}{P}.$$

For event rate λ and interrupt-handling CPU time C_i per event,

$$U_i = \lambda C_i,$$

until queueing, coalescing, or saturation invalidates the independent-event model. Equating the two gives a first-order crossover $\lambda^* = C_p/(PC_i)$. The result is meaningful only if the latency constraint permits period P and the measured costs match the platform.

Worked example: sparse control events

A device produces 200 events/s. One status poll costs 120 ns. The deadline requires detection within 50 µs, so choose $P = 50$ µs for worst-case compliance. Polling consumes

$$\frac{120\ \text{ns}}{50\ \mu\text{s}} = 0.0024 = 0.24\%$$

of one CPU. If one interrupt consumes 2.5 µs of CPU time, interrupts consume

$$200\ \text{s}^{-1} \times 2.5\ \mu\text{s} = 0.0005 = 0.05\%.$$

Interrupts use less CPU under these assumptions. Polling may still be chosen if its tighter jitter matters, if a dedicated core is available, or if event bursts would produce interrupt storms. The model does not decide the objective.

Coalescing changes the event unit

If the device posts 64 completions and sends one interrupt, the fixed notification cost is amortized across the batch. Coalescing reduces interrupt rate but delays early completions. The control variable is a count or timer threshold; evaluate both CPU occupancy and tail completion latency.

Application — choose a coalescing bound. Events arrive at 80,000/s. One interrupt costs 1.5 µs of CPU time. Batching 16 completions per interrupt adds at most 200 µs by policy. Calculate notification CPU occupancy without and with full batches; state why the latency claim still needs a burst/idle analysis.

Without batching, occupancy is `80,000 × 1.5 µs = 0.12`, or 12% of one CPU. With full batches of 16, interrupt rate is 5,000/s and occupancy is 0.75%. Full batches do not always form: after a sparse arrival, a timer must release a partial batch, and the first completion may wait near the timer limit. The 200 µs policy bound also excludes interrupt dispatch and scheduling delay unless those are included explicitly.

DMA changes who moves bytes

EXAMPLE

Under a linear CPU-cost model where programmed I/O costs $C_{\mathrm{PIO}}(n) = C_{\mathrm{setup}} + C_{\mathrm{irq}} + n \cdot c_{\mathrm{copy}}$ and DMA costs $C_{\mathrm{DMA}}(n) = C_{\mathrm{dma_setup}} + C_{\mathrm{completion}}$, the byte-count crossover is $n^* = (C_{\mathrm{dma_setup}} + C_{\mathrm{completion}} - C_{\mathrm{setup}} - C_{\mathrm{irq}})/c_{\mathrm{copy}}$. Below n^*, setting up and tearing down the DMA mapping consumes more CPU cycles than copying the payload directly through registers.

For a transfer of n bytes, compare two CPU-busy models:

$$C_{PIO}(n) = C_{setup} + C_{interrupt} + nc_{copy},$$

$$C_{DMA}(n) = C_{dma_setup} + C_{completion}.$$

DMA is advantageous for CPU occupancy when its fixed setup cost is less than the avoided CPU copy cost. It does not make the device latency or bus transfer disappear. Small transfers can favor PIO because DMA setup dominates.

Worked crossover under the simulator's constants

The investigation below uses a 3 GHz CPU, 500 setup cycles for programmed I/O, 1,800 for DMA, 1,200 per interrupt, two CPU cycles per copied byte, and 100 MB/s transfer rate. Equating interrupt-driven PIO and DMA CPU work:

$$500 + 1200 + 2n = 1800 + 1200 \quad \Rightarrow \quad n = 650 \text{ bytes}.$$

Under these invented constants, DMA has lower modeled CPU occupancy above 650 bytes; interrupt-driven PIO has lower fixed CPU work below it.

For 4 KiB, 1 ms latency, and 100 operations/s:

Mechanism	CPU busy per operation	Elapsed per operation	CPU occupancy
busy polling + PIO	1.0439 ms	1.0439 ms	10.44%
interrupt + CPU copy	0.00330 ms	1.0443 ms	0.033%
DMA + completion interrupt	0.00100 ms	1.0420 ms	0.010%

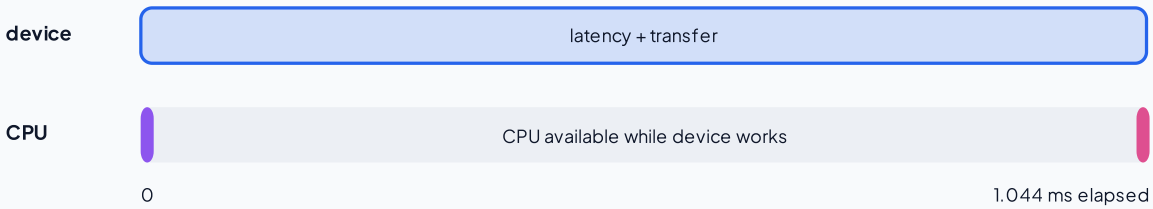
DMA has both the lowest CPU occupancy and, by a small margin, the lowest elapsed time for this 4 KiB state: its larger setup is outweighed by eliminating the modeled CPU copy. That elapsed-time result can reverse for transfers below the 650-byte crossover. Latency and CPU demand remain different objectives.

Experiment: change bytes, events, and device latency

The simulator assumes one independent operation, no queueing, a fixed 100 MB/s transfer rate, and the numeric constants listed above. Polling means continuous busy waiting for the modeled device latency, not periodic polling. CPU occupancy is capped at 100%; once offered work exceeds capacity, the widget does not model the resulting queue or latency growth.

Prediction — quantify before moving a slider. At 64 bytes, 0.1 ms device latency, and 2,000 operations/s, which has lower modeled CPU occupancy: interrupt-driven copying or DMA? Use the fixed-cycle equations, then set the widget to that state.

CHOOSE POLLING, INTERRUPTS, OR DMA FROM WORKLOAD EVIDENCE



Static state: 4 KiB transfers, 1 ms device latency, 100 operations/s, interrupt-driven CPU copying. The CPU is free during device latency but pays setup, interrupt, and copy work.

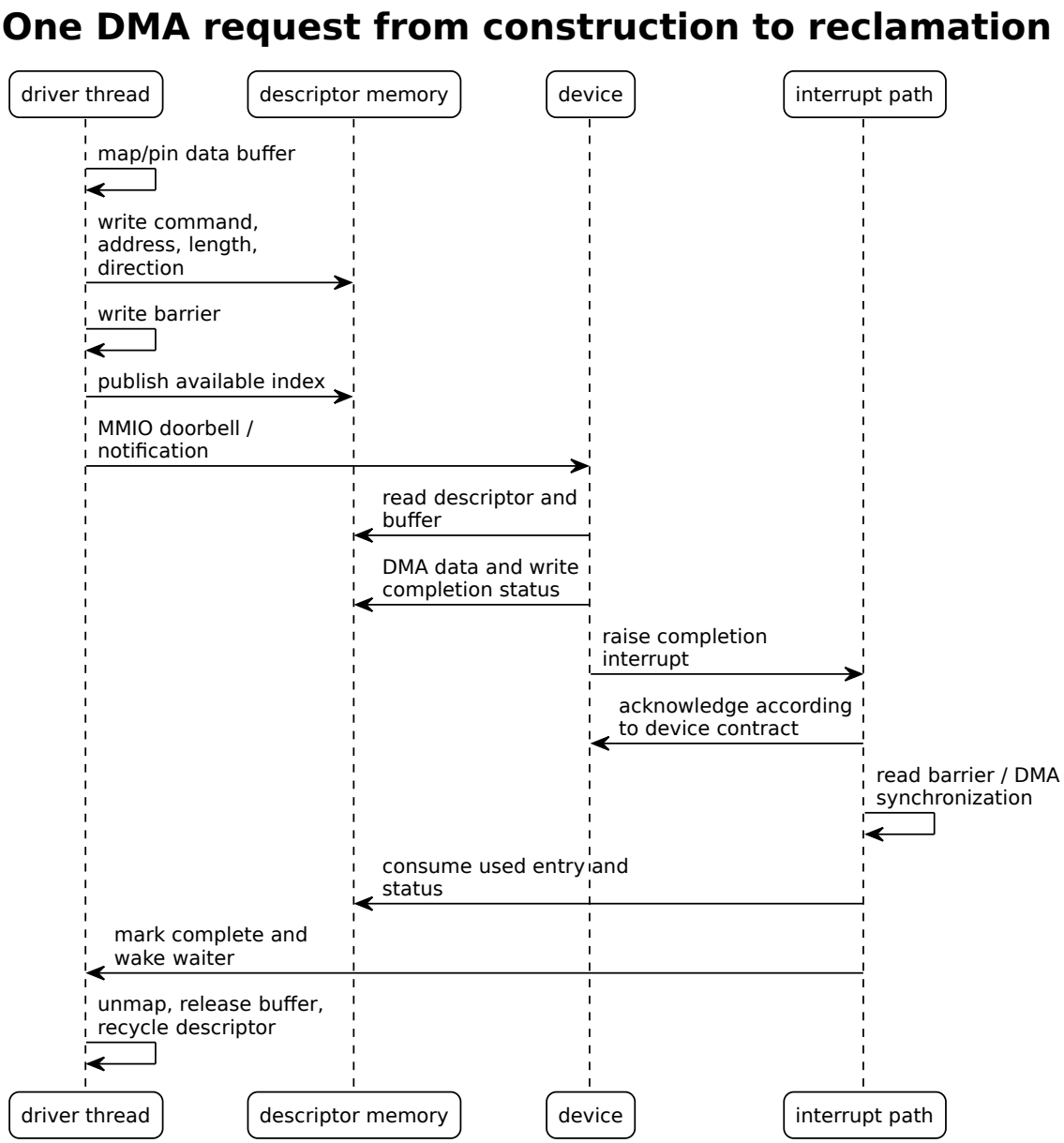
Interrupt-driven copying consumes `(500 + 1200 + 128)/3 GHz = 0.609 µs` per operation, or about 0.122% of a CPU at 2,000/s. DMA consumes `3,000/3 GHz = 1 µs` per operation, or 0.200%. The small transfer favors CPU copy under this CPU-work model; device latency changes elapsed time but not these two modeled CPU-work equations.

Transfer — invalidate one assumption. Replace the constant 100 MB/s device with a device that sustains four transfers concurrently but slows each transfer by 30% at queue depth four. State what additional state the model needs before it can predict throughput and tail latency.

The model needs at least queue depth, arrival process, number of service lanes, service-time dependence on concurrency, completion order, and a latency distribution rather than one constant. CPU occupancy can still be computed per submission/completion, but end-to-end latency requires waiting time plus concurrency-dependent service. A single-operation bar cannot support a queueing claim.

Descriptor rings encode ownership

High-throughput devices use shared-memory descriptors. A descriptor typically contains a device-visible address, length, direction, command, and status. A split ring such as Virtio has driver-written available state and device-written used state.



For ring capacity N and monotonic producer/consumer counters, the software invariant is

$$0 \le producer - consumer \le N.$$

Slots are indexed modulo N, but counters should retain enough generation information to distinguish full from empty after wrap. Publication follows a release pattern:

```
write descriptor fields
write barrier for device visibility
write ring slot containing descriptor head
write barrier required by transport
advance published producer index
notify device if required
```

The device may access the descriptor immediately after the publication point. Editing a submitted descriptor before completion violates ownership even if the CPU still has a pointer to it.

Worked wrap trace

An eight-entry ring has monotonic `producer=14`, `consumer=10`, so occupancy is four. Published requests occupy logical sequence numbers 10–13 and physical slots 2–5. The producer's next physical slot is `14 mod 8 = 6`.

Publishing three requests uses slots 6, 7, and 0, then advances producer to 17. Occupancy becomes seven. One slot remains. Comparing only `producer mod 8 = 1` with `consumer mod 8 = 2` does not directly reveal occupancy; generation-bearing counters do.

Application — kill publish-first. A mutation advances `producer` before writing the descriptor's length. Construct an interleaving in which the device reads a valid old address and old length. State which single happens-before edge is missing.

CPU publishes producer 15; device observes the new index and reads slot 6; slot 6 still contains the previous generation's descriptor; device begins DMA; CPU writes the new address and length too late. The missing edge is "all descriptor and ring-slot stores happen before the publication-index store as observed by the device." A CPU-only lock does not create that edge unless the device participates in its memory-ordering protocol, which it does not.

Coherence, synchronization, and IOMMU authority

> **WARNING**
>
> Neither cache-coherent DMA nor an IOMMU prevents use-after-free corruption when a driver reclaims an in-flight buffer on timeout: if the physical frame is reallocated before the device is quiesced and its IOMMU mapping invalidated, a late device completion will overwrite the new owner's memory.

A coherent DMA allocation lets CPU and device observe each other's writes without explicit cache-line cleaning for coherence, but coherent memory still needs barriers to order fields. A streaming DMA mapping may require explicit synchronization when ownership passes between CPU and device. Linux's DMA API documents separate `dma_sync_*_for_device` and `dma_sync_*_for_cpu` operations for reused streaming mappings.

An IOMMU translates a device-visible I/O virtual address and enforces device permissions. It can restrict a device to the pages assigned to its DMA context, but it cannot repair a driver that reuses a still-owned buffer. Teardown requires both:

1. stop or quiesce old device transactions; and
2. revoke/invalidate device translations before the physical pages enter a new ownership domain.

Counterexample: timeout is not cancellation

A request exceeds its deadline. The driver returns an error, unmaps the DMA address, and allocates the frame to another process. The device completes late and writes into the new process's memory. The timeout proved only that completion was not observed by a deadline. Safe recovery must retain the buffer or complete a documented cancel/reset/quiesce protocol before reuse.

Interrupt handling closes the protocol

An interrupt handler must drain enough device state that the interrupt condition is understood and acknowledged according to the device contract. It must also preserve concurrency invariants:

- validate completion IDs before indexing software arrays;
- synchronize before reading device-written status or data;
- transition each request to completed at most once;
- detach ownership before waking a waiter that may free resources;

- avoid blocking work that cannot run in interrupt context; and
- handle more than one completion per interrupt when the device batches.

Worked race: wake before state

Waiter code uses `while (req.state != DONE) sleep(req, lock)`. The interrupt handler calls `wakeup(req)` and then writes `req.state = DONE`. A waiter can wake, reacquire the lock before the state store, observe `IN_FLIGHT`, and sleep again after the one wakeup has passed. The fix is to update completion state while holding the protocol lock and wake after the state transition is visible. The loop remains necessary because wakeups can be spurious or another event can change scheduling.

Duplicate and malformed completions

A duplicate completion must not decrement a reference twice or recycle a descriptor already assigned to another generation. Track request identity and lifecycle, validate `id < ring_size`, require `state == IN_FLIGHT`, and pair a reusable slot with a generation or sequence number when delayed events can cross reuse. A malformed status is a device/transport failure to report; treating every completion as success converts device error into silent data corruption.

Transfer — teardown defense. A hot-unplug event arrives with four requests in flight. Write a state sequence that permits their buffers to be reused. “Wait 100 ms” is not an admissible transition.

Stop admitting new submissions; mark the device stopping; mask or route interrupts according to the platform contract; request and verify device/queue reset or another specified quiescence mechanism; drain or fail completions while keeping buffers pinned; revoke DMA mappings and invalidate IOMMU state after quiescence; transition every request exactly once to completed/failed; wake waiters; then release descriptors and buffers. If the platform cannot prove quiescence, retaining/quarantining the memory is safer than reuse.

REMARK

This lesson establishes the fundamental producer-consumer ring and DMA ownership invariants required by the Device Driver Lab, deliberately deferring multi-queue interrupt steering (MSI-X), adaptive polling/interrupt switching (NAPI), and PCIe transaction-layer packet ordering to specialized systems references.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 5.1.1: Calculating the Polling versus Interrupt CPU Occupancy Crossover

Verification application

A peripheral device produces I/O events at mean rate λ events per second.

- Polling implementation: The CPU polls every $P = 20\ \mu\text{s}$, with each poll requiring $C_p = 100\ \text{ns}$ of CPU execution time.
- Interrupt-driven implementation: Each event triggers a hardware interrupt requiring $C_i = 4\ \mu\text{s}$ of CPU service time (entry, context save, driver handler, EOI acknowledgment, context restore).

1. Calculate the steady-state CPU occupancy of the polling driver ($U_{\text{l}}^{\text{pol}}$).
2. Compute the event arrival rate λ^* at which the interrupt-driven driver consumes identical CPU occupancy ($U_{\text{r}}^{\text{int}} = U_{\text{l}}^{\text{pol}}$).

3. If the actual arrival rate is $\lambda = 2500$ events/s, determine which notification strategy minimizes CPU overhead.

Occupancy for polling is C_p / P; occupancy for interrupts is lambda * C_i.

1. **Polling occupancy:**

$$U_{\text{poll}} = \frac{C_p}{P} = \frac{100 \text{ ns}}{20\ \mu\text{s}} = \frac{0.1\ \mu\text{s}}{20\ \mu\text{s}} = 0.005 = 0.5\%.$$

2. **Crossover arrival rate** λ^*: Set $U_{\text{r}}^{\text{int}} = \lambda^* C_i = U_{\text{l}}^{\text{pol}}$:

$$\lambda^* \times 4\ \mu\text{s} = 0.005 \implies \lambda^* = \frac{0.005}{4 \times 10^{-6}\ \text{s}} = 1250 \text{ events/s}.$$

3. **Evaluation at** $\lambda = 2500$ events/s:

$$U_{\text{intr}} = 2500 \times (4 \times 10^{-6}\ \text{s}) = 0.010 = 1.0\%.$$

Because $\lambda = 2500 > \lambda^* = 1250$, polling consumes 0.5% CPU whereas interrupts consume 1.0%. Polling minimizes CPU occupancy at this arrival rate.

Exercise 5.1.2: Memory Barrier Placement in Producer–Consumer Descriptor Rings

Construction application

A driver submits a DMA packet by populating a ring descriptor `desc[idx]` in host RAM (fields: `buffer_pa`, `length`, `flags`) and subsequently ringing an MMIO doorbell register `DOORBELL = idx` to notify the network device.

1. Explain the race condition that occurs if the CPU or compiler reorders the doorbell store before the descriptor field stores.
2. Specify the exact memory barrier or fence required between the descriptor initialization and the MMIO doorbell write on weakly ordered architectures.

The device starts fetching memory immediately upon receiving the doorbell write.

1. **Reordering race condition:** If the MMIO store to `DOORBELL` reaches the peripheral before the writes to `desc[idx]` retire to coherent RAM or cache, the device immediately reads `desc[idx]`. The device observes uninitialized or stale descriptor values (e.g., an old buffer address or invalid length), causing the device to read or overwrite unauthorized host memory.
2. **Barrier specification:** The driver must place a release memory barrier or write memory barrier (`smp_wmb()` or `atomic_thread_fence(memory_order_release)`) immediately before the doorbell store:

```
desc[idx].buffer_pa = dma_addr;
desc[idx].length = packet_len;
desc[idx].flags = DESC_VALID;
dma_wmb(); /* guarantees memory writes precede MMIO writes
*/
mmio_write32(&dev->doorbell, idx);
```

This enforces the happens-before relationship:

$$\text{Write}(\text{Descriptor Fields}) \prec \text{Write}(\text{MMIO Doorbell}).$$

Exercise 5.1.3: Deriving Transfer-Size Thresholds for DMA versus Programmed I/O

Construction transfer

A 2 GHz CPU executes programmed I/O (PIO) or direct memory access (DMA):

- PIO setup is negligible, but copying requires 3 CPU cycles per transferred byte.
- DMA setup requires pinning pages, setting up descriptors, and cache flushing/invalidation, costing a fixed $T_{\mathrm{p}}^{\mathrm{setu}} = 1800\ \mathrm{cycles}$. The DMA engine transfers data without CPU intervention.
- Both schemes pay a fixed $T_{\mathrm{r}}^{\mathrm{int}} = 1200\ \mathrm{cycles}$ for completion interrupt processing.

1. Write the CPU cycle cost functions $C_{\mathrm{O}}^{\mathrm{PI}}(B)$ and $C_{\mathrm{A}}^{\mathrm{DM}}(B)$ for a transfer of B bytes.
2. Find the breakeven transfer size B^{*} where DMA becomes strictly more efficient for CPU occupancy than PIO.

Set C_PIO(B) = C_DMA(B) and solve for B.

1. **Cost functions:**
   - For Programmed I/O: $C_{\text{O}}^{\text{PI}}(B) = T_{\text{r}}^{\text{int}} + 3 \cdot B = 1200 + 3B$ cycles.
   - For DMA: $C_{\text{A}}^{\text{DM}}(B) = T_{\text{p}}^{\text{setu}} + T_{\text{r}}^{\text{int}} = 1800 + 1200 = 3000$ cycles.

2. **Breakeven size B^*:**

$$1200 + 3B^* = 3000 \implies 3B^* = 1800 \implies B^* = 600 \text{ bytes.}$$

For transfers strictly greater than 600 bytes, DMA is more CPU-efficient; for transfers smaller than 600 bytes (e.g., 64–byte control packets), PIO avoids the setup tax and reduces total CPU load.

Exercise 5.1.4: Proving Buffer Invariant Violations on Premature Timeout Reclamation

Proof synthesis

Suppose a disk driver submits a DMA read request for physical buffer B. After 5 seconds, the request has not completed, and a software timeout fires. Prove that unmapping B and immediately reallocating it to an unrelated process violates the buffer-ownership invariant unless the device is proven quiesced.

A timeout measures only software wait time, not device bus state; the device may commit transfer cycles later.

Assume for contradiction that the driver frees and reallocates buffer B to Process P_2 immediately upon timeout expiration without issuing a hardware reset or awaiting device quiescence:

1. The expiration of a software timer proves only that no completion interrupt was processed; it does not prove the peripheral hardware aborted its bus mastering activity.
2. The device controller may be delayed due to internal queueing, bus contention, or high controller latency.
3. Once buffer B is reassigned to Process P_2, Process P_2 begins writing sensitive data into B.
4. At $t = 5.2\ \text{seconds}$, the delayed device DMA engine initiates or finishes writing its delayed disk block into physical address B.
5. The device overwrites P_2's active data, destroying memory contents and violating the buffer-ownership invariant:

$$\forall t, \quad \text{Owner}(B, t) \in \{\text{CPU}\} \oplus \{\text{Device}\}.$$

Therefore, buffer B cannot be freed or reused until the driver executes a verified hardware reset, aborts the in-flight command on the controller, and confirms bus quiescence.

Exercise 5.1.5: Eliminating Lost Wakeup Races in Asynchronous Driver Waiters

Description synthesis

Analyze the following buggy driver completion sequence between an interrupt handler and a sleeping process:

- Waiter: `while (req->status == PENDING) { sleep_on(&req->wq); }`
- Handler: `wake_up(&req->wq); req->status = DONE;`

Show how this order can cause an indefinite deadlock, and formulate the correct synchronized sequence.

Show that the waiter can evaluate the condition right before the handler executes both lines.

- **Failure schedule:**
  1. Waiter reads `req->status`, observes `PENDING`.
  2. Before the waiter enters the sleep queue, a hardware interrupt fires on the CPU.
  3. The interrupt handler executes: calls `wake_up(&req->wq)`, but the wait queue is currently empty.
  4. The interrupt handler then sets `req->status = DONE` and returns from interrupt.
  5. The preempted waiter resumes and executes `sleep_on(&req->wq)`.
  6. The waiter is now suspended indefinitely because the wakeup occurred in the past, causing a lost wakeup and indefinite thread hang.

- **Correct synchronization protocol:** The status transition must happen before the wakeup, and both operations must be protected by the queue lock:
  - In handler:

```
acquire(&req->lock);
req->status = DONE;
wake_up_locked(&req->wq);
release(&req->lock);
```

  - In waiter:

```
acquire(&req->lock);
while (req->status == PENDING) {
    sleep_locked(&req->wq, &req->lock);
}
release(&req->lock);
```

  Holding `req->lock` ensures that checking the condition and sleeping is atomic with respect to the state modification and wakeup.

References & Further Reading

- OASIS Virtual I/O Device (Virtio) Version 1.3
- Linux kernel: Dynamic DMA Mapping Guide
- RISC-V Privileged Architecture
- xv6–riscv revision 5 book, interrupts and device drivers
- xv6–riscv `kernel/virtio_disk.c` at revision 5
- xv6–riscv `kernel/trap.c` at revision 5

SECTION DETAIL

Storage Devices and I/O Scheduling

A scheduler sorts every block request by logical block address because that reduced seek distance on a magnetic disk. The same policy is deployed on an NVMe SSD. Requests from one latency-sensitive process wait behind a long adjacent run, while the SSD's independent internal channels remain underfilled. "Sequential is faster" was promoted from a device-specific observation to a universal rule.

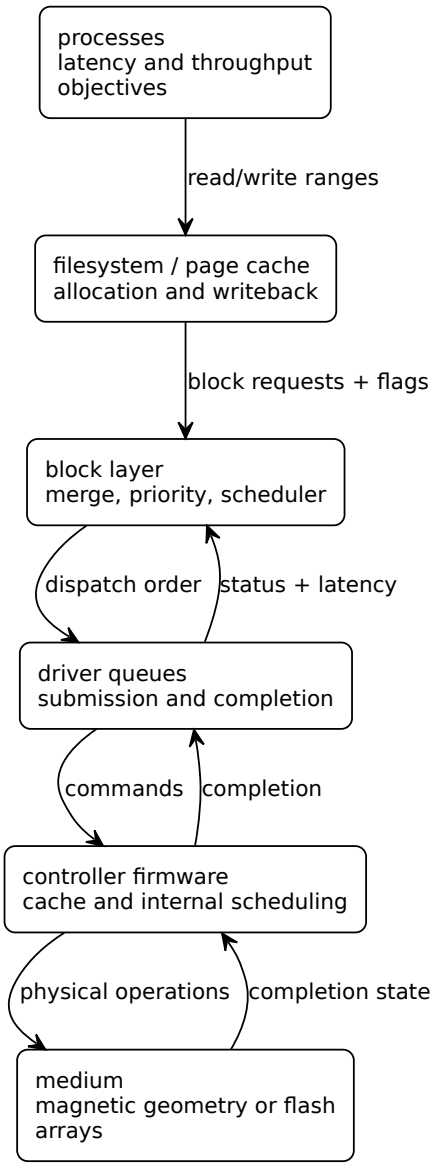
Storage scheduling sits between workload intent and a device whose service function may be mechanical, solid-state, virtualized, or remote. A policy is defensible only when its cost model matches the medium and its objective includes more than aggregate throughput.

> ***Media-aware scheduling contract.*** *Request order must be justified by measured or specified device behavior, queue semantics, fairness boundaries, and durability constraints. Logical address proximity alone is not a universal service-time metric.*

The block layer does not erase the medium

Applications and filesystems issue operations on logical byte or block ranges. The storage stack may merge, split, queue, prioritize, and dispatch them. A controller or device can reorder again, serve requests in parallel, cache writes, or translate logical addresses internally.

Where one logical block request can be reordered



The observable latency of request i can be decomposed as

$$Ti = Tqueue,i + Tcontroller,i + Tmedium,i + Ttransfer,i + Tcompletion,i.$$

The decomposition prevents one useful optimization—such as reducing seek—from being mistaken for the entire response time.

Retrieval — identify hidden reordering. A filesystem submits writes A then B, and the driver rings one queue doorbell. Name two lower layers that may still reorder work and the interface feature needed when A must become durable before B.

Controller firmware and the medium's internal scheduler/translation layer may reorder. The host needs an ordering/durability mechanism defined by the device protocol, such as an ordered dependency, flush, or force-unit-access semantics as appropriate. Submission order alone is not a durability proof. Exact commands are protocol-specific and are outside the assessed filesystem-recovery scope.

Magnetic disks pay mechanical positioning cost

A simplified magnetic-disk access has

$$T_{medium} = T_{seek} + T_{rotation},$$

followed by transfer. Seek moves the actuator to the target track/cylinder; rotational latency waits for the desired sector to pass under the head. With rotation rate R revolutions/minute and uniformly distributed angular position,

$$E[T_{rotation}] = \frac{1}{2}\frac{60}{R} \text{ seconds}.$$

At 7,200 RPM, one revolution is 8.33 ms and average rotational latency under the uniform-angle model is 4.17 ms.

Worked trace: service-time estimate

Assume a magnetic disk with measured 8 ms seek for a particular distance, 7,200 RPM, sustained transfer 200 MB/s, no queueing, and no controller-cache hit. A 64 KiB request has

$$T_{transfer} = \frac{65{,}536}{200{,}000{,}000}\ \text{s} = 0.328\ \text{ms}.$$

The estimate is

$$8 + 4.17 + 0.328 = 12.498\ \text{ms}.$$

This is not a constant per-request law. Seek is nonlinear in distance, rotational position is not always uniform, adjacent sectors can stream, controller caches can satisfy a request, and zone geometry changes transfer rates.

Cylinder movement is a proxy, not time

Classical scheduling traces use total cylinder movement because nearby requests often reduce mechanical positioning. For initial head 53 and queue

```
98, 183, 37, 122, 14, 124, 65, 67
```

the supplied model produces:

Policy	Service path	Movement
FCFS	`53→98→183→37→122→14→124→65→67`	640 cylinders
SSTF	`53→65→67→37→14→98→122→124→183`	236 cylinders
SCAN upward	`53→65→67→98→122→124→183→199→37→14`	331 cylinders

SSTF wins this movement total, but repeatedly arriving nearby requests can postpone a distant request indefinitely. SCAN trades some movement for directional progress. The model's SCAN travels to cylinder 199 before reversing; LOOK would reverse at the final request in the direction, producing a different path.

Application — expose starvation. The head starts at 50. A request at 190 waits while one request arrives every service interval, alternating between cylinders 49 and 51. Trace the first six SSTF choices and state whether any finite waiting bound follows from the policy.

At each choice, a newly arrived request one cylinder away is closer than 190, so SSTF can alternate around 50 and postpone 190 for all six choices—and indefinitely if arrivals continue. No finite waiting bound follows from pure nearest-request selection. Adding an age/deadline rule changes the policy and can bound postponement under stated dispatch assumptions.

STORAGE DEVICE CHARACTERISTICS AND SCHEDULING INVARIANTS

Evaluate each statement regarding magnetic hard drives, solid-state drives, and IO scheduling algorithms:

Answer: At 7200 RPM with uniform angular arrival, the expected rotational latency is exactly 4.17 milliseconds.; Shortest Seek Time First (SSTF) minimizes head movement but can starve requests located at distant cylinders.

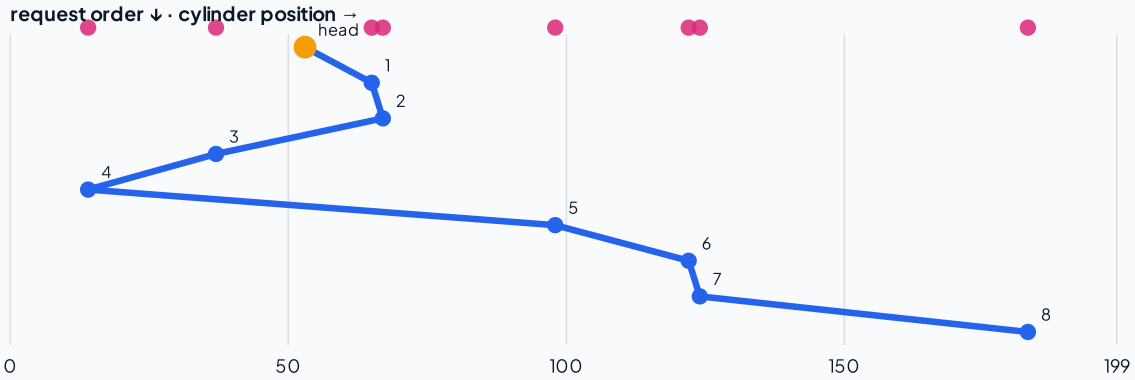
Expected rotational delay at 7200 RPM is half a revolution (4.17 ms). SSTF can indefinitely starve requests far from the active head track. SSDs have no moving mechanical heads so cylinder seeking is meaningless, and Little's law shows that queue depth equals throughput times latency, meaning excessive queueing inflates latency once device capacity is saturated.

Experiment: compare movement and fairness

The simulator uses cylinders `0..199`, a single magnetic head, a fixed request batch, and movement equal to absolute cylinder-distance sums. FCFS preserves listed arrival order. SSTF chooses the nearest pending cylinder, breaking equal distances by the workload's current list order. SCAN moves upward, reaches cylinder 199 when lower requests remain, then reverses. It omits rotation, transfer length, deadlines, writes, caches, and new arrivals.

Prediction — choose an objective before a policy. On the mixed queue above, which policy has the least movement? Which policy gives the strongest service-order argument to the oldest request from the information provided?

MOVE THE DISK HEAD UNDER COMPETING QUEUE POLICIES



Static state: SSTF starts at cylinder 53 and travels 236 cylinders. It minimizes the next seek greedily but can postpone distant requests indefinitely.

Transfer — change the medium. Reinterpret the same request numbers as SSD logical block addresses. Identify every visual quantity that loses physical meaning, then name three measurements that would replace cylinder movement in an SSD scheduler evaluation.

For the prediction, SSTF has the least movement on the fixed mixed batch, while FCFS gives the strongest argument about the oldest request because it preserves listed arrival order. After changing the medium, head position, path slope, cylinder distance, direction, and endpoint reversal lose their mechanical meaning. Logical adjacency may still help merging or sequential prefetch, but it does not reveal flash-channel placement or FTL physical location. Useful SSD measurements include throughput/IOPS, mean and tail completion latency, queue occupancy, device utilization, read/write mix, write amplification or bytes written, fairness, and per-queue/core contention.

FCFS, SSTF, SCAN, and deadlines optimize different contracts

EXAMPLE

Suppose read requests at cylinders `52, 54, 56` arrive continuously while a read at cylinder `180` has been queued for 45 ms. Pure SSTF services `52, 54, 56` first and starves `180`; a deadline scheduler maintains both a sector-sorted tree and a FIFO expiration list (e.g., 50 ms read expiration), switching from sector-ordered dispatch to the expired FIFO head as soon as `180`'s age crosses the deadline threshold.

Policy	Useful property	Failure boundary	Repair direction
FCFS	transparent arrival order	long seeks and convoy effects	merge/batch without erasing age
SSTF	low next seek under cylinder proxy	starvation of distant requests	age or deadline override
SCAN	directional progress	endpoint travel; unequal waiting by position	LOOK/C-SCAN or deadlines
deadline hybrid	limits age while batching locality	tuning couples throughput and latency	measure read/write classes and tails

A deadline scheduler can keep requests in both address order and age order. It dispatches batches by address for locality but checks whether an old request must be served. The guarantee is only as strong as the precise timer, batch, and lower-layer assumptions. Linux's documented deadline scheduler, for example, exposes separate expiration and batch controls rather than claiming exact completion deadlines.

Worked policy decision: interactive reads and bulk writes

A magnetic drive receives 4 KiB interactive reads with a 20 ms start-service objective and 1 MiB sequential background writes. Pure FCFS lets a long write run delay reads. Pure read priority can starve writeback and eventually block the filesystem. A hybrid may:

- merge adjacent writes into bounded batches;
- dispatch reads preferentially;
- force any request whose age exceeds its class threshold;

- cap consecutive reads so dirty writeback progresses; and
- measure **start–service** and completion separately.

The design is incomplete without queue-depth and arrival assumptions. A deadline cannot be guaranteed if admitted work exceeds device capacity and no request can be rejected or degraded.

SSDs replace a head with translation and parallelism

DEFINITION

A Flash Translation Layer (FTL) maintains a dynamic mapping $\mathrm{LBA} \mapsto \mathrm{PPA}$ from host logical block addresses to physical flash pages, writing updated blocks out-of-place into erased pages, marking old physical pages stale, and asynchronously compacting valid pages across erase blocks via garbage collection.

NAND flash cannot overwrite a programmed page in place. A flash translation layer (FTL) maps host logical blocks to physical flash pages, writes updates out of place, and reclaims erase blocks through garbage collection. Thus consecutive logical addresses need not remain physically adjacent, and background firmware work can produce variable latency.

An NVMe interface exposes memory-based submission and completion queues and supports substantial parallelism. Host software can map queues to CPUs and keep multiple commands outstanding. The important scheduling dimensions shift toward:

- enough queue depth to expose device parallelism without unbounded waiting;
- avoiding one global software lock or queue bottleneck;
- read/write and latency-class isolation;
- merge/split overhead and transfer size;
- firmware garbage-collection tail latency;
- flush/order semantics; and
- fairness across processes, cgroups, or virtual machines.

Linux's blk-mq documentation explicitly contrasts mechanical disks—where head-position ordering addressed random-access cost—with solid-state devices that have no mechanical parts and can process requests in parallel.

Counterexample: LBA sorting can reduce parallelism

Suppose SSD requests at LBAs 100–107 map internally across eight channels, while LBAs 108–115 happen to contend on one flash die after FTL remapping. A host policy that assumes the second run is "more sequential" has no evidence for physical service cost. Issuing enough independent work and observing completions may exploit parallelism better than serializing by LBA. The exact mapping is usually controller-private, so the claim must be measurement-driven.

Queue depth, throughput, and latency obey a conservation relation

For a stable observed system, Little's law relates long-run average outstanding requests N, effective completion rate λ, and average time in the system W:

$$N = \lambda W.$$

At 100,000 completed operations/s and 400 µs mean end-to-end storage time,

$$N = 100{,}000 \times 0.0004 = 40$$

requests are outstanding on average. A queue cap of 16 cannot coexist with both measured values in a stable closed domain; at least one of rate, latency, scope, or measurement is different. Little's law does not say that setting queue depth to 40 will cause 100,000 IOPS. Service capacity and arrival behavior determine whether that operating point is reachable.

Mean latency does not protect the tail

Two devices can both average 400 µs while one completes every request near 400 µs and the other pauses during garbage collection. Report p50, p95/p99, maximum within a defined run, and the full workload conditions. A scheduler that improves mean throughput while multiplying p99 latency may violate an interactive-service objective.

Application — audit a benchmark. A report claims 150,000 IOPS at 600 µs mean latency with "queue depth 32." Use Little's law to identify the inconsistency and list two possible scope explanations.

The two measured values imply average outstanding work `150,000 × 0.0006 = 90`, not 32. Possible explanations include 32 queues each with more than one outstanding request, a per-thread queue-depth label while 90 requests exist system-wide, latency measured over a broader layer than the reported queue, or throughput and latency taken from different intervals. The result may be valid, but the term “queue depth 32” is underspecified.

Boundary cases that defeat one-number policies

WARNING

Optimizing a storage scheduler solely for aggregate IOPS or cylinder distance breaks correctness or tail-latency contracts when upper layers require write-barrier durability (`REQ_PREFLUSH` / `REQ_FUA`), when virtual-disk layers hide an underlying SSD or network store, or when aggressive request merging delays small latency-critical reads.

Writes have ordering and durability semantics

A scheduler may reorder independent writes but must preserve barriers and dependencies expressed by upper layers and supported by the device. Completion to software can mean accepted, cached, or durable depending on the protocol and flags. A filesystem cannot infer power-loss durability from queue order alone.

Virtual disks conceal another scheduler

A guest sees logical blocks presented by a virtual device. The host may map them to a file, network store, SSD, or thin-provisioned volume. Guest cylinder optimization can fight host scheduling. The guest should rely on advertised features and measurement rather than inventing physical geometry.

Shingled and zoned media alter random-write rules

Some magnetic and solid-state devices expose zones with sequential-write constraints. They are not modeled as ordinary random-write disks. Naming zoned storage without teaching its command/state model would not establish mastery, so it remains outside scope.

Queue merging can violate latency isolation

Merging adjacent requests lowers per-command overhead, but waiting to form a large merge can delay a small urgent request. Merge windows need bounds tied to service objectives.

REMARK

This lesson bridges the low-level device ring protocols of Device I/O, Interrupts, and DMA with the persistent data layouts and crash-recovery logs of the filesystem sequence: the scheduler may reorder independent requests for throughput or fairness, but must never violate explicit persistence barriers required by write-ahead logging.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 5.2.1: Decomposing Hard Disk Drive Access Latency and Service Time

Verification application

A magnetic disk operates with the following parameters:

- Rotational speed: 10,000 RPM.
- Average seek time: $T_{\text{k}}^{\text{see}} = 4.2$ ms.
- Sustained media transfer rate: 150 MB/s $(150 \times 10^6$ bytes/s).
- Controller and bus overhead: $T_{\text{d}}^{\text{overhea}} = 0.1$ ms.

1. Calculate the rotational period and the expected rotational latency T_{t}^{ro} assuming uniform angular arrival.

$^{\text{xfe}}$

2. Calculate the transfer time T_{r} for a 64 KiB ($65{,}536$ byte) block request.
3. Compute the total expected service time T_{e} for an isolated random 64 KiB read request.

servic

One revolution at 10,000 RPM is 60 / 10000 seconds. Expected rotational delay is half a revolution.

1. **Rotational latency:** The duration of one complete revolution is:

$$T_{\mathrm{rev}} = \frac{60\ \mathrm{s}}{10{,}000} = 0.006\ \mathrm{s} = 6.0\ \mathrm{ms}.$$

   Assuming uniform angular position of the desired sector upon head arrival at the cylinder:

$$E[T_{\mathrm{rot}}] = \frac{1}{2} T_{\mathrm{rev}} = 3.0\ \mathrm{ms}.$$

2. **Transfer time:**

$$T_{\mathrm{xfer}} = \frac{65{,}536\ \mathrm{bytes}}{150 \times 10^{6}\ \mathrm{bytes/s}} \approx 0.000437\ \mathrm{s} \approx 0.437\ \mathrm{ms}.$$

3. **Total expected service time:**

$$T_{\mathrm{service}} = T_{\mathrm{seek}} + E[T_{\mathrm{rot}}] + T_{\mathrm{xfer}} + T_{\mathrm{overhead}} = 4.2 + 3.0 + 0.437 + 0.$$

   Notice that mechanical positioning (seek + rotation = $7.2\ \mathrm{ms}$) accounts for over 93% of the total access time.

Exercise 5.2.2: Comparing FCFS, SSTF, and SCAN Traversal Distances

Construction application

A disk queue holds requests for cylinders $\langle 95, 180, 34, 119, 11, 123, 62, 64 \rangle$ in arrival order. The read/write head is currently positioned at cylinder 50 and moved from cylinder 40 (moving upward toward higher cylinders).

1. Trace the head movement sequence and compute the total cylinder distance for First-Come First-Served (FCFS).
2. Trace the sequence and total cylinder distance for Shortest Seek Time First (SSTF).
3. Trace the sequence and total cylinder distance for SCAN (Elevator algorithm, reversing at cylinder 199).

SSTF selects the closest pending cylinder; SCAN sweeps upward to the boundary then sweeps downward.

1. **FCFS:** Path: $50 \to 95 \to 180 \to 34 \to 119 \to 11 \to 123 \to 62 \to 64$. Distances: $|95 - 50| + |180 - 95| + |34 - 180| + |119 - 34| + |11 - 119| + |123 - 11| + |62 - 123| + |64 - 62| = 45 + 85 + 146 + 85 + 108 + 112 + 61 + 2 = 644$ cylinders.

2. **SSTF:** Starting at 50, find nearest remaining:

   - Nearest to 50 is 62 ($|62 - 50| = 12$).
   - Nearest to 62 is 64 ($|64 - 62| = 2$).
   - Nearest to 64 is 34 ($|34 - 64| = 30$, vs 95 which is 31).
   - Nearest to 34 is 11 ($|11 - 34| = 23$).
   - Nearest to 11 is 95 ($|95 - 11| = 84$).
   - Nearest to 95 is 119 ($|119 - 95| = 24$).
   - Nearest to 119 is 123 ($|123 - 119| = 4$).
   - Nearest to 123 is 180 ($|180 - 123| = 57$). Path: $50 \to 62 \to 64 \to 34 \to 11 \to 95 \to 119 \to 123 \to 180$. Total distance: $12 + 2 + 30 + 23 + 84 + 24 + 4 + 57 = 236$ cylinders.

3. **SCAN (Elevator upward):** Services all higher cylinders in ascending order up to 199, then reverses downward:

   - Upward pass: $50 \to 62 \to 64 \to 95 \to 119 \to 123 \to 180 \to 199$.
   - Reversal downward: $199 \to 34 \to 11$. Total distance: $(199 - 50) + (199 - 11) = 149 + 188 = 337$ cylinders.

Exercise 5.2.3: Applying Little's Law to Storage Queue Depth Sizing

Construction transfer

An enterprise NVMe SSD benchmark reports a sustained random read throughput of $\lambda = 500{,}000\ \text{IOPS}$ at an average request latency of $W = 160\ \mu\text{s}$.

1. Use Little's law ($L = \lambda W$) to calculate the average number of concurrent requests L in flight across the host driver and SSD controller queues.
2. If the host application uses a single submission queue with a maximum depth limit of 32, what is the maximum achievable throughput if latency remains at $160\ \mu\text{s}$?
3. What happens to request latency if the application increases queue depth to 256 while device service capacity is capped at 500,000 IOPS?

L = lambda * W. If L is capped, lambda = L / W. If lambda is capped, W = L / lambda.

1. **Average in-flight requests L:**

$$L = \lambda \cdot W = (500{,}000 \text{ requests/s}) \times (160 \times 10^{-6} \text{ s}) = 80 \text{ requests}.$$

2. **Throughput at queue depth limit 32:** With $L \leq 32$ and mean service latency $W = 160\ \mu$s:

$$\lambda_{\max} = \frac{L_{\max}}{W} = \frac{32}{160 \times 10^{-6} \text{ s}} = 200{,}000 \text{ IOPS}.$$

   A queue depth of 32 leaves the SSD under-utilized, achieving only 40% of its rated 500,000 IOPS capacity.

3. **Queue inflation and latency explosion:** Once device throughput saturates at $\lambda_{\text{sat}} = 500{,}000$ IOPS, raising queue depth to $L = 256$ cannot increase throughput. Instead, requests accumulate in queues:

$$W_{\text{inflated}} = \frac{L}{\lambda_{\text{sat}}} = \frac{256}{500{,}000 \text{ s}^{-1}} = 0.000512 \text{ s} = 512\ \mu\text{s}.$$

   Average latency increases by more than $3.2\times$ solely due to queueing delay (bufferbloat).

Exercise 5.2.4: Internal Flash Hierarchy and Multi-Queue NVMe Concurrency

Description application

Contrast the physical architecture of an NVMe solid-state drive with a magnetic hard drive. Explain why traditional single-queue elevator schedulers degrade SSD performance, whereas NVMe multi-queue (blk-mq) architectures optimize it.

Consider mechanical actuators versus multiple parallel flash channels, chips, dies, and planes.

- **Physical contrast:**
  - Hard disk drives have a single mechanical arm with heads that can read or write only one cylinder/track at any given millisecond. Performance depends strictly on minimizing physical seek distance and rotational delay.
  - Solid-state drives contain no moving parts. Flash storage is organized into multiple independent flash channels (e.g., 8–16 channels), each connected to multiple flash chips, dies, and planes. A modern SSD can execute 32–64 flash page reads simultaneously in parallel across independent dies.
- **Why elevator scheduling degrades SSDs:** Sorting block requests by LBA in an elevator queue serializes requests into a single bottlenecked thread, pays CPU lock contention overhead, and suppresses the queue depth needed to keep multiple internal flash channels active simultaneously. Furthermore, logical block addresses do not correspond to physical flash die locations due to Flash Translation Layer (FTL) wear-leveling mappings.
- **NVMe Multi-Queue (blk-mq) architecture:** NVMe provides up to 64,000 separate submission/completion queues, each supporting 64,000 commands. By dedicating per-CPU-core queues that bypass cross-core lock synchronization, the OS dispatches deep concurrent streams of requests directly to controller hardware, maximizing internal flash die parallelism.

Exercise 5.2.5: Proving Write Ordering Failures Under Reordered Storage Caches

Proof synthesis

A database system appends a transaction record to block $B_{\log}$ and then updates user balance in block B_{data}.

1. Prove that if the drive controller possesses a volatile write cache and reorders writes, a power failure can produce unrecoverable database corruption unless an explicit cache flush or FUA command is issued.
2. Contrast the semantics of `REQ_PREFLUSH` and `REQ_FUA` in modern operating system block layers.

Show an interleaving where data reaches stable flash before log data is persisted.

1. **Proof of corruption under cache reordering:**

    - The database relies on the Write-Ahead Logging (WAL) invariant:

$$\text{Persist}(B_{\text{log}}) \prec \text{Persist}(B_{\text{data}}).$$

    - The operating system submits B_{log} followed by B_{data}.
    - The disk controller acknowledges both writes immediately upon placing them into its volatile write-back DRAM cache.
    - The disk controller firmware's internal scheduler writes B_{data} to persistent magnetic platters or NAND flash first, because B_{data} is closer to the current head or open flash block.
    - Before B_{log} is drained from the volatile cache, a power failure occurs.
    - Upon recovery, B_{data} contains the committed transaction modifications, but B_{log} has vanished. The recovery engine cannot undo or redo the transaction, leaving database state permanently inconsistent. Hence, software submission order alone is insufficient; explicit hardware barriers are strictly required.

2. **PREFLUSH versus FUA semantics:**

    - `REQ_PREFLUSH`: Instructs the storage controller to synchronously flush all dirty blocks currently resident in its volatile write-back cache to non-volatile media before executing the current command.
    - `REQ_FUA` (Force Unit Access): Instructs the controller that the specific I/O block carried by this command must be written directly to persistent non-volatile media before completing and returning success to the host, bypassing or write-through flushing the volatile cache for this request.

References & Further Reading

- Linux kernel: Multi-Queue Block I/O Queueing Mechanism
- Linux kernel: Deadline I/O scheduler tunables

- NVM Express Base Specification landing page and current ratified revision
- NVM Express Base Specification, Revision 2.3 (archived primary specification)
- H. Kim et al., “BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage” (FAST 2008)
- Seagate Savvio 15K.2 SAS Product Manual, seek and access-time definitions
- John D. C. Little, “A Proof for the Queuing Formula: L = λW” (1961)

SECTION DETAIL

Device-Driver Lab: A Defensible Virtio Queue

xv6-riscv revision 5 uses an eight-entry Virtio split queue. Each block request consumes a chain of three direct descriptors: command header, data buffer, and one-byte status. At most two requests can hold complete chains simultaneously because a third chain would require nine descriptors. The Virtio indirect-descriptor feature can represent those three elements behind one main-ring descriptor, raising the queue's descriptor-limited ceiling from two requests to eight when the device offers the feature.

The implementation target is not "make the benchmark number larger." The target is a driver whose feature negotiation, publication order, request lifetime, completion validation, fault behavior, and measurements can be defended from traces.

This is an original LibreUni lab. It uses the licensed upstream teaching kernel and the public Virtio specification; it does not reproduce another course's assignment or solution.

Implementation contract

Maintain a repository branch with the pinned upstream commit as a verified ancestor, plus generated evidence outside the source diff. Learners studying the operating systems sequence cumulatively retain earlier labs; independent learners may start directly from the pinned baseline.

1. the driver and minimal, non-conflicting syscall/user-test support needed by the design, preserving earlier cumulative interfaces;
2. a test manifest mapping each invariant to an oracle and named mutant;
3. raw measurement output and a script or exact command sequence that reproduces every table;
4. a design memo of 1,500–2,500 words;
5. a driver-lab patch-range diff against the recorded lab-start commit, plus the ancestry evidence back to the pinned baseline; and
6. a diagnostic self-verification trace with the code and execution logs available.

A diff that cannot be built from the declared commit is invalid. Performance claims are meaningless until correctness invariants pass. The 720-minute workload is a planning estimate, not permission to stop with an unverified state.

Non-negotiable invariants

Label these `I1` – `I10` in the memo and cite the code and test that establish each one.

ID	Invariant
I1	A feature is used only if the device offered it and the driver accepted it before `FEATURES_OK`.
I2	One request owns either one indirect head descriptor or one complete direct chain, never a partial mixture.
I3	Descriptor contents and the available-ring slot become device-visible before the published available index.
I4	The data-buffer direction bit matches the operation: device-writable for reads, device-readable for writes.
I5	`0 ≤ in_flight ≤ negotiated_queue_capacity`; full/empty remains correct across 16-bit index wrap.
I6	A used-ring ID is bounds-checked before indexing driver metadata.
I7	A request transitions to terminal state once; duplicate completion before reap is detected, not completed twice.
I8	Completion state and status become visible before a waiter is woken; waiters recheck a predicate in a loop.
I9	A descriptor, indirect table, status byte, and data buffer are not reused until device ownership ended and the waiter/reaper transferred ownership.
I10	Timeout or process death does not imply device cancellation; no timed-out buffer is reused without specified quiescence/reset evidence.

> ***Defense boundary.*** *Standard split-ring used entries return a descriptor-chain head ID, not the driver's private generation counter. A software generation detects many lifecycle defects, but it cannot distinguish a malicious stale completion that arrives after the same head ID has been legitimately reused unless the protocol returns equivalent identity. Safety after reset therefore depends on verified device/queue quiescence, not generation numbers alone.*

Retrieval — before opening source. Explain why eight main-ring entries permit only two baseline requests but as many as eight indirect requests. State the resource that still may reduce observed concurrency below eight.

Baseline requests each require three direct descriptors, so `floor(8/3)=2` complete chains fit. An indirect request consumes one main descriptor that points to its private three-entry indirect table, so eight heads can fit if all other resources are available. Buffer-cache locking, process scheduling, available workload, device service time, and driver metadata can reduce observed concurrency; eight is a descriptor-capacity ceiling, not a throughput promise.

Pin and verify the baseline

The required baseline is the official tag `xv6-riscv-rev5`, commit `7d7adbb1b0acbd67c9766a20d0f9900fef2789fa`. Preserve its `LICENSE` and upstream notices. Do not reset a cumulative branch to this commit: verify ancestry, record the reviewed exit commit from the preceding lab as the driver-lab start, and branch from that state.

Standalone setup only:

```
git clone https://github.com/mit-pdos/xv6-riscv.git libreuni-xv6-driver
cd libreuni-xv6-driver
git checkout 7d7adbb1b0acbd67c9766a20d0f9900fef2789fa
git switch -c libreuni/virtio-indirect-lifecycle
git rev-parse HEAD
make qemu
```

For a cumulative branch, run the following from the accepted prior-lab state instead of cloning again:

```
git merge-base --is-ancestor 7d7adbb1b0acbd67c9766a20d0f9900fef2789fa
HEAD
git merge-base HEAD 7d7adbb1b0acbd67c9766a20d0f9900fef2789fa
git rev-parse HEAD
git switch -c libreuni/virtio-indirect-lifecycle
```

The first command must exit zero; the second must print the pinned baseline; the third is the lab-start commit to preserve in the implementation record. In standalone setup, `rev-parse` instead prints the baseline itself. The revision-5 Makefile requires a RISC-V toolchain and QEMU 7.2 or newer. Capture regression evidence before changing the driver; on a cumulative branch, this also proves the earlier labs still build and run:

```
$ usertests -q
$ stressfs
```

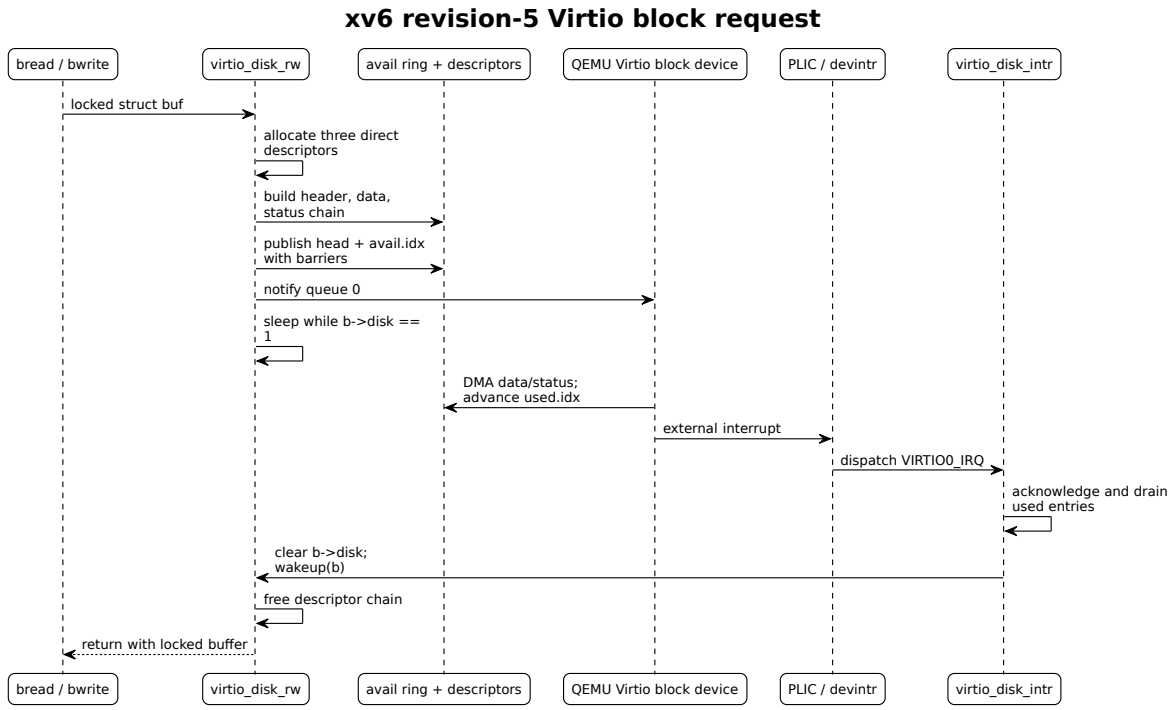
Save command output before editing. If the baseline fails, repair the environment rather than changing kernel code to hide the failure.

Read the path as one protocol

Inspect these files and record the named state each contributes:

File	Required observation
`kernel/bio.c`, `kernel/buf.h`	buffer lock, `valid`, `disk`, and caller ownership
`kernel/virtio_disk.c`	feature negotiation, descriptor allocation, publication, wait, completion, free
`kernel/virtio.h`	register offsets, queue layouts, flags, `NUM=8`
`kernel/trap.c`, `kernel/plic.c`	external-interrupt claim, dispatch, acknowledgment, PLIC completion
`kernel/riscv.h`	memory-order primitive and `r_time()` source
`kernel/syscall.c`, `kernel/sysproc.c`, `user/user.h`, `user/usys.pl`	test-only control/stat syscall path
`Makefile`	kernel/user objects, programs, CPU count, QEMU invocation

The baseline worked trace of a single block request is:



Baseline evidence checkpoint

Add temporary trace points or use GDB to capture one read and one write. The trace must identify:

- all three descriptor indices and flags;
- sector and block-number relation;
- `avail.idx` before/after publication;
- `used_idx` and device `used->idx` at completion;
- the interrupting hart;

- `b->disk` before sleep and before wake; and
- the point at which each descriptor becomes free.

Remove ad hoc print flooding before the final patch. The permanent trace facility is bounded and structured.

A complete derivation requires a causally ordered trace, not a list of function names. A read's data descriptor has `VRING_DESC_F_WRITE` because the **device** writes into memory; a write's data descriptor lacks that flag because the device reads memory. The request head is the first descriptor ID placed in the available ring and later returned in the used ring. Descriptor reclamation occurs in the waiting thread after completion wakes it in revision 5.

VIRTIO DESCRIPTOR FLAGS AND QUEUE INVARIANTS

Evaluate each statement regarding Virtio descriptor flags, queue operations, and buffer directions in xv6:

Answer: In block read operations the data descriptor sets the device-write flag because peripheral hardware fills memory.; Indirect descriptors allow an 8-entry virtqueue to support up to 8 in-flight requests instead of the 2-request limit.

On disk reads, the device writes to host RAM so VRING_DESC_F_WRITE must be set; on writes, the device reads RAM so the flag is clear. Indirect descriptors raise capacity from floor(8/3)=2 to 8 requests. Used ring entries return descriptor chain head IDs rather than software generation counters.

Milestone A — isolate negotiation from queue layout

The baseline explicitly clears `VIRTIO_RING_F_INDIRECT_DESC` from the offered feature word. Refactor feature selection into a small testable decision whose output records whether indirect descriptors were accepted. Requirements:

1. read offered features;
2. reject features the driver does not implement;
3. retain `VIRTIO_RING_F_INDIRECT_DESC` only when the indirect path is compiled and offered;
4. write the accepted set;
5. set `FEATURES_OK` and re-read status as the baseline does; and
6. select direct fallback if the feature is absent.

The test harness must call the pure decision logic with synthetic feature words. Do not depend on undocumented QEMU command-line properties to simulate absence.

Interface sketch, not a solution

```
struct vdisk_features {
  uint32_t accepted;
  int indirect;
};

// Pure decision: no MMIO inside this function.
struct vdisk_features choose_vdisk_features(uint32_t offered,
                                            int indirect_build_enabled);
```

Tests must cover offered+enabled, offered+disabled, absent+enabled, and unknown offered bits. The MMIO initialization code remains responsible for status transitions.

Application — reject force-indirect. A mutant sets `VRING_DESC_F_INDIRECT` in descriptors whenever the build supports it, even when the device did not offer the feature. Which invariant fails before the first request, and what deterministic unit input exposes it?

I1 fails. Call the decision function with an offered word whose indirect bit is clear and build support enabled. The result must select direct fallback and must not set the indirect bit in `accepted`. A test that boots only one default QEMU instance may never exercise this case.

Milestone B — implement one-head indirect requests

Add `VRING_DESC_F_INDIRECT` using the value defined by the Virtio specification. For each main descriptor slot, provide a persistent, device-visible, suitably aligned three-entry indirect table. It cannot live on a stack that disappears while the device owns it.

The indirect entries describe:

Indirect entry	Buffer	Device direction
0	`struct virtio_blk_req`	device reads
1	`b->data`	device writes for block read; device reads for block write
2	one-byte status	device writes

The main descriptor points to the indirect table, has length `3 × sizeof(struct virtq_desc)`, and sets `VRING_DESC_F_INDIRECT`. It must not also chain direct descriptors with `NEXT`. The used-ring ID remains the main descriptor index.

The xv6 kernel directly maps the relevant physical RAM, but the memo must still justify why each address given to the device is a valid DMA address in this teaching machine. “The C pointer compiled” is not evidence.

Request resource model

Define resource acquisition as one transaction:

```
reserve head slot
reserve per-head request metadata and indirect table
initialize command, data, and status entries
publish exactly one head
```

If any required resource is unavailable, release everything acquired in that attempt before sleeping/retrying. Partial descriptor ownership is forbidden by I2.

Direct fallback remains first-class

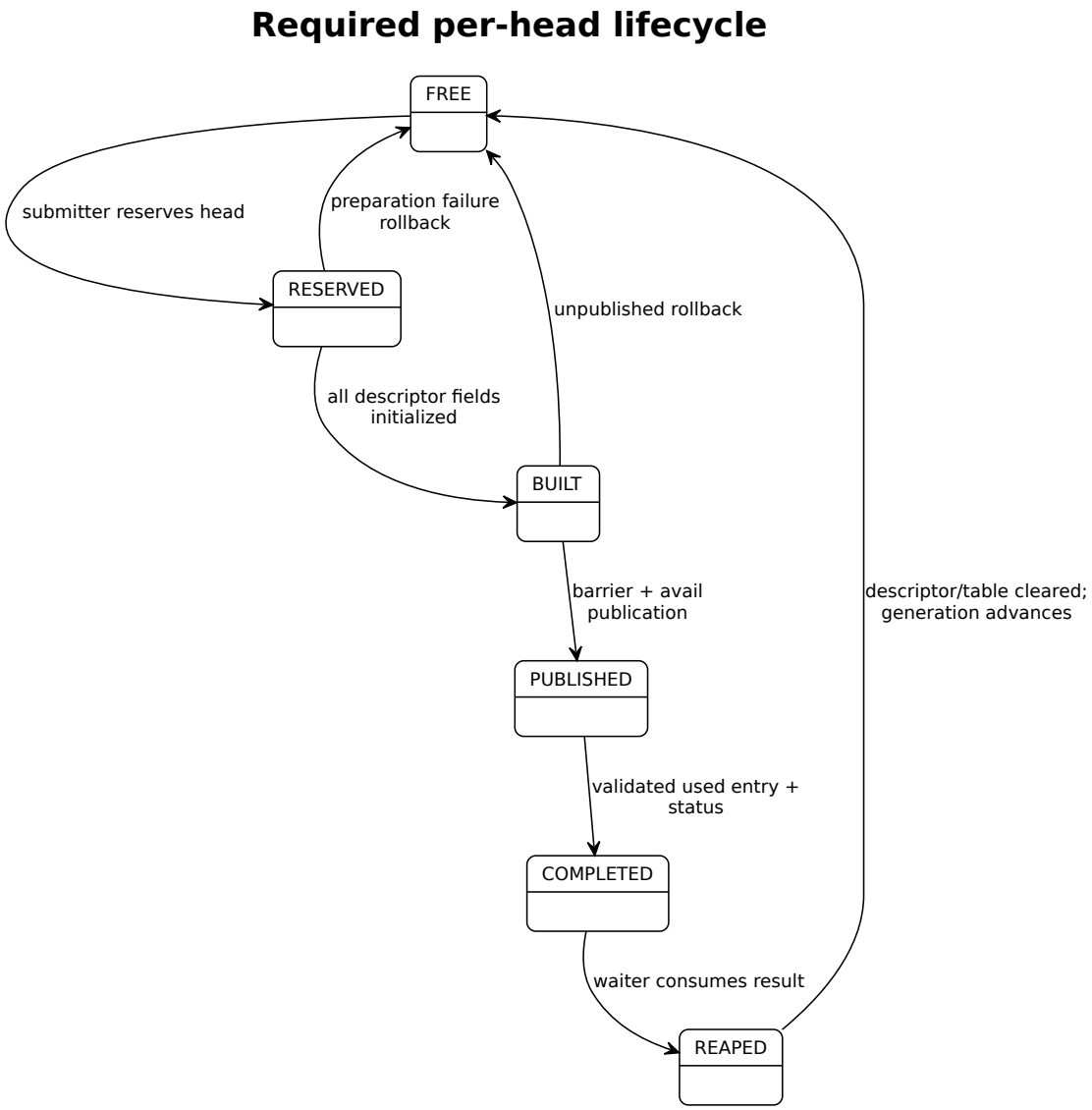
Do not delete the direct-chain implementation. Factor common command/status setup while retaining two publication shapes. Both must satisfy the same lifecycle, status, wait, and trace interfaces. Run correctness and fault tests in both modes.

Application — direction audit. For an xv6 `bread`, write the flags for all three indirect entries. Repeat for `bwrite`. Explain the flag name from the device's perspective.

Entry 0 is device-readable and chains to entry 1. For `bread`, entry 1 sets device-write plus next because the device fills `b->data`; for `bwrite`, entry 1 sets next without device-write because the device consumes `b->data`. Entry 2 sets device-write because the device writes status. The main indirect descriptor uses the indirect flag and no direct-chain next flag.

Milestone C — make request lifetime explicit

Replace implicit combinations of `free[]`, `info[].b`, status, and `b->disk` with one auditable lifecycle. The exact enum names may differ, but every transition needs a lock, actor, and test.



State `PUBLISHED` is the device-ownership boundary. Rollback from it is prohibited without a device-defined cancel/reset/quiesce operation. A timeout may mark diagnostic state, stop admission, or fail-stop the teaching kernel; it may not transition directly to `FREE`.

Metadata sketch

```
enum req_state { REQ_FREE, REQ_RESERVED, REQ_BUILT,
                 REQ_PUBLISHED, REQ_COMPLETED, REQ_REAPED };

struct req_meta {
  enum req_state state;
  uint32_t generation;
  struct buf *buffer;
  uint64_t submit_time;
  uint64_t complete_time;
  unsigned char status;
};
```

This sketch is not a full implementation. Decide which fields are device-written, which lock protects transitions, when generation advances, and whether timing reads occur inside or outside the lock. Do not expose kernel pointers through a user-visible stats syscall.

Completion validation order

The interrupt path must establish, in order:

1. interrupt status belongs to the device and is acknowledged as specified;
2. used-ring writes are visible before entries are consumed;
3. `id < NUM` before any `info[id]` access;
4. request state is `PUBLISHED`;
5. status is success before the buffer is declared valid to the caller;
6. state becomes `COMPLETED` and `b->disk` becomes false under the protocol lock;
7. waiters are woken after the predicate transition; and
8. `used_idx` advances exactly once per consumed used entry.

For an invalid ID, duplicate completion, or error status, a revision-5 teaching kernel may fail-stop with a stable diagnostic rather than implement full filesystem error propagation. It must not index out of bounds, report success, free owned DMA memory, or continue in an unknowable queue state.

Transfer — generation limit. A duplicate used entry for head 3 arrives after head 3 was completed, reaped, incremented to generation 18, and republished. The hardware used entry carries only ID 3. Can private `generation=18` prove that this completion belongs to generation 18?

No. The standard split-ring used entry does not return the private generation. The driver can detect a duplicate before reuse because state is no longer `PUBLISHED`, and test hooks can carry synthetic generation metadata, but a post-reuse stale ID is indistinguishable from the current request at this interface. The Virtio device contract and verified reset/quiescence prevent such late completions. Claiming generation alone solves the case overstates the protocol.

Milestone D — publish with a provable happens-before edge

Virtio's split-ring sequence requires the driver to write descriptor contents and the available-ring entry, execute a suitable memory barrier, then update `avail.idx`; further ordering applies before notification checks/doorbells. Preserve the baseline's architecture-appropriate synchronization or replace it only with an equally justified primitive.

Create a test-only **device observer hook** at the publication boundary. When enabled, it snapshots the head descriptor, indirect entries, status poison, generation, and ring slot at the instant the driver is about to expose or has exposed `avail.idx`, according to the tested phase. The hook must verify that:

- every address and length is initialized;
- indirect chaining terminates correctly;
- status begins with a non-success poison value;
- direction flags match operation;
- head/generation metadata agrees with the ring slot; and
- no post-publication mutation occurs before completion.

The hook makes an early-publication mutant deterministic in QEMU, where accidental strong ordering might otherwise hide it. It supplements, rather than replaces, the source-level barrier argument.

Ring-wrap proof obligation

`avail.idx` and `used.idx` are 16-bit monotonically increasing counters that wrap naturally. Physical ring slots use `% NUM`. Do not reduce the stored counters modulo `NUM` or compare only physical indices for full/empty.

Provide a pure ring arithmetic test that advances at least 65,540 logical submissions/completions without performing 65,540 disk operations. Check occupancy, slot selection, and empty/full transitions across both the eight-entry wrap and the 16-bit counter wrap.

For unsigned 16-bit producer and consumer counters with at most `NUM` outstanding, `(uint16) (producer - consumer)` yields occupancy across natural wrap. The harness should place counters near `65535`, submit and complete while never exceeding eight, and compare against a wider reference model. Any implementation that stores only `index % 8` cannot distinguish full from empty when the physical indices are equal.

Milestone E — bounded observability without print storms

Add a fixed-size trace ring, for example 256 events, whose writer never allocates or sleeps in interrupt context. Each event should include only what is needed to reconstruct causality:

```
sequence, r_time, cpu, event,
head, generation, operation, block,
avail_idx, used_idx, in_flight, status
```

Required event types: reserve, build, publish, sleep, interrupt, complete, wake, reap, rollback, and fault-detected. If the trace overwrites old entries, count drops/overwrites explicitly. Sequence numbers must let the reader detect missing events.

Add a snapshot interface for aggregate statistics:

- submitted/completed reads and writes;
- direct versus indirect submissions;
- descriptor-wait count and accumulated wait time;
- maximum in-flight requests;
- interrupts and completions drained per interrupt;
- request-time histogram or bounded sample sufficient for median/p95;
- error/fault-injection counters; and

- trace-overwrite count.

Copy a consistent snapshot to user space without holding the driver spinlock across `copyout`. One admissible pattern copies counters to a local kernel struct under the lock, releases the lock, then copies the local snapshot to user memory.

Time units

Use raw `r_time()` deltas or calibrate them against the revision-5 timer interval. Do not label raw counter differences "nanoseconds" without a verified timebase. QEMU measurements describe the emulator and host run, not physical Virtio hardware.

Application — trace completeness. A trace contains `publish(g=7)`, `complete(g=7)`, and `publish(g=8)` for head 2, but no `reap(g=7)` or overwrite marker. Which invariant cannot be audited, even if all file data is correct?

I9 cannot be audited: the trace does not show that generation 7's descriptor/table/buffer ownership returned to the driver before reuse for generation 8. Correct output is insufficient because a timing change could expose early reuse. Either the reap event is missing due to trace loss—which must be marked—or the lifecycle is wrong.

Milestone F — deterministic fault injection

WARNING

Fault-injection hooks must be compiled strictly under an explicit build guard (`VDISK_FAULT_TEST`) and must never free or recycle DMA-owned descriptors on a simulated timeout while the device may still write to the physical buffer.

Compile fault controls only in an explicit test build such as `VDISK_FAULT_TEST`. A normal build must not expose a user-controlled kernel fault surface. A test-only control call should arm one mode for the nth matching event, report whether it fired, and reset to a known disabled state at boot.

Required modes:

Mode	Injection point	Correct observable outcome
`SPURIOUS_WAKE`	while a request remains `PUBLISHED`	waiter wakes, rechecks, and sleeps again; no early return
`HOLD_COMPLETIONS`	before consuming selected used entries	request resources remain owned; other controller process can release hold
`BAD_USED_ID`	completion decoder	stable fail-stop before metadata indexing
`DUPLICATE_PRE_REAP`	feed one accepted ID again before reap	duplicate-state detection; no second wake/free/refcount update
`STATUS_ERROR`	completion status validation	no success/valid declaration; stable fail-stop or explicit propagated error
`PUBLISH_OBSERVER`	descriptor publication boundary	deterministic snapshot proves all fields precede publication
`DESCRIPTOR_PRESSURE`	hold completions while many readers submit	submitters block without partial ownership; release drains and wakes
`LATE_AFTER_TIMEOUT`	hold request past a diagnostic deadline	timeout never frees/reuses DMA-owned resources

`HOLD_COMPLETIONS` needs a separate controlling process because the faulting worker may be asleep. Release must make forward progress without requiring the held worker to run first.

Timeout policy for the core lab

Full Virtio reset is outside the core implementation. When a diagnostic deadline expires, choose one documented safe behavior:

- mark the device failed, stop new submissions, retain all in-flight resources, emit a bounded trace, and fail-stop; or
- retain ownership and wait for the test controller to release the held completion.

Returning an error while freeing the descriptor or buffer is forbidden. Recovery with queue reset is an extension only if it follows the negotiated transport procedure and proves that the device can no longer use old descriptors before release.

Tests must kill named wrong implementations

For each mutant below, include a one-line code transformation, the test that fails, the expected failure, and evidence that the unmutated driver passes. An automated test harness or adversarial reviewer may apply equivalent mutations rather than your exact macro.

Mutant name	Deliberate defect	Test that must kill it
`M_FORCE_INDIRECT`	use indirect descriptors when feature absent	synthetic negotiation matrix
`M_PUBLISH_INDEX_FIRST`	advance `avail.idx` before descriptor visibility	publication observer
`M_REVERSE_DIRECTION`	invert device-write flag on data entry	known-block read/write checksum test
`M_PARTIAL_RESERVE`	keep one descriptor after multi-resource failure	pressure/rollback inventory test
`M_IF_WAIT`	replace predicate `while` with `if`	spurious wake test
`M_WAKE_BEFORE_STATE`	wake before terminal predicate update	controlled wait/interrupt interleaving
`M_TRUST_USED_ID`	index `info[id]` before range check	bad-ID injection
`M_ACCEPT_DUPLICATE`	omit `PUBLISHED` state check	duplicate-pre-reap injection
`M_IGNORE_STATUS`	set `b->disk=0` on nonzero status	status-error injection
`M_FREE_IN_INTERRUPT`	recycle head/table before waiter reaps	hold waiter + pressure + trace check
`M_MODULO_COUNTERS`	store only indices modulo eight	pure 16-bit wrap test
`M_TIMEOUT_CANCELS`	free a held request when deadline expires	late-after-timeout ownership test

Required test groups

Upstream regression
Run in direct fallback and indirect modes:

```
usertests -q
stressfs
grind
```

`usertests -q` and `stressfs` must reach their normal completion markers. Revision-5 `grind` runs forever by design: drive it from an external harness for a declared duration or progress count, require continued `A` / `B` progress with no panic or failure diagnostic, then terminate that QEMU run from the harness. Record the exact command, CPU count, build mode, QEMU version, completion markers for finite tests, and the bound plus termination method for `grind`. An invented `grind` exit marker is invalid evidence.

On a cumulative branch, run every earlier lab's accepted exit suite before and after this driver change. A passing upstream suite cannot substitute for a scheduler or synchronization invariant that the cumulative branch added.

Data integrity

Write deterministic per-block patterns to a temporary file larger than `NBUF`, wait for every write call to return, close the descriptors, terminate QEMU normally, and reboot the same image before reading/verifying in sequential and permuted block order. Use multiple child processes on disjoint files to create concurrent misses. A checksum mismatch must print file, logical block, expected generation, and observed checksum without dumping arbitrary data. This is a clean-restart integrity test, not a sudden-power-loss durability claim: revision 5 exposes no user-level flush contract for proving the latter.

Deterministic queue pressure

Arm `HOLD_COMPLETIONS`, launch at least eight disjoint cold reads, wait until the trace proves the queue reached its resource ceiling, release completions, and require every worker to finish with correct data. The direct build should expose its two-request descriptor ceiling; the indirect build should admit more than two and may reach eight. If another subsystem limits it, report that trace rather than fabricating a result.

Fault isolation

Run each fail-stop injection in a fresh throwaway `fs.img`/QEMU process. Match a stable diagnostic and verify that no later success marker appears. Rebuild the image for the next destructive case. Do not infer safety from a hung emulator.

Wrap and lifecycle unit tests

Exercise pure negotiation/ring functions without device I/O. Check every legal state edge and reject every illegal edge. Advance the reference counter across 65,535 and compare against a wider oracle.

Test-oracle table

Your manifest must have this form:

Test	Precondition	Trigger	Observable oracle	Mutant killed
`vdisk_spurious_wake`	one worker asleep in `PUBLISHED`	wake without completion	no return; later real completion returns once	`M_IF_WAIT`
`vdisk_bad_id`	fault-test build	used ID = `NUM`	`vdisk fault: bad-id` before any array access	`M_TRUST_USED_ID`

Add every required test. “Kernel did not crash” is not a sufficient oracle for data integrity or ordering.

Measurements must explain, not decorate

Compare at least these builds:

- baseline-compatible direct chains;
- indirect descriptors enabled and offered; and
- indirect-capable build with synthetic negotiation selecting direct fallback for unit tests.

For runtime measurements, hold constant:

- recorded lab-start commit and final code except mode;
- QEMU and host version;
- `CPUS=1` and `CPUS=3` as separate conditions;
- image state and file contents;
- operation count, read/write mix, access order, and concurrency;
- warm-up rule and buffer-cache treatment; and
- number of repetitions.

Use at least three workloads:

1. one process, cold sequential reads over a file larger than `NBUF`;
2. four processes, disjoint cold reads to expose queue concurrency;
3. controlled writes followed by reboot and checksum verification.

Report:

Metric	Why it is needed
completed operations / raw-time interval	throughput under declared time units
median and p95 request time	distribution, not only average
max in-flight	whether indirect descriptors changed concurrency
descriptor waits and wait time	whether main-ring capacity is binding
completions / interrupt	batching behavior
CPU count and workload	reproducibility and contention context
data-integrity result	performance never substitutes for correctness

Do not claim real-device speedup from QEMU. A valid result may show no throughput improvement because the filesystem, emulator, host scheduler, or workload is the bottleneck. The memo must connect observed max-in-flight and descriptor waits to any performance claim.

Transfer — interpret a null result. Indirect mode reaches six in-flight requests instead of two, but throughput is unchanged and p95 latency rises. Give three plausible mechanisms and one measurement for each.

- Device/emulator service is already saturated: inspect device utilization or completion rate versus queue depth.
- Extra queueing increases waiting without parallel service: split submit-to-dispatch and dispatch-to-complete times.
- A shared kernel/host lock or CPU is the bottleneck: compare `CPUS=1/3`, lock-wait instrumentation, and host CPU saturation.
- The workload hits another cache/path limit: inspect descriptor-wait count and actual block-driver submissions versus user operations.

The conclusion should identify which evidence discriminates mechanisms, not select one from intuition.

Design memo

> REMARK
>
> Every claim in the design memo must point to a concrete state transition, memory barrier, or deterministic fault-injection log; narrative summaries without happens-before edges or trace evidence do not satisfy the proof obligation.

Use these headings; each section must cite code, trace events, or test output.

Baseline and threat model

State the pinned ancestor, recorded lab-start commit, inherited cumulative changes, queue shape, device assumptions, trusted Virtio guarantees, and failures you inject. Distinguish a buggy driver, a spec-compliant device, and a device that violates the protocol.

State and ownership proof

Define every request state, owner, lock, and legal edge. Prove I1–I10. Include the point after which rollback is forbidden and the point before which reuse is forbidden.

Direct and indirect layouts

Draw both descriptor layouts, feature fallback, address lifetime, flags, alignment, and maximum in-flight calculation. Explain why the indirect table is persistent and device-visible.

Ordering argument

Identify descriptor stores, available-ring store, barriers, index publication, notification, used-ring synchronization, terminal state, wake, and reap. A source line list without happens-before claims is incomplete.

Failure analysis

For every injection, state detected symptom, contained state, resources intentionally retained, and why no unauthorized reuse occurs. Disclose that full reset/recovery is omitted.

Evaluation

Present raw-data link/path, environment, repetitions, summary statistics, uncertainty, and competing explanations. Separate QEMU evidence from hardware claims.

Residual risk

Name at least three untested assumptions, including post-reuse stale IDs, device reset, and filesystem error propagation. State the experiment or design work needed to reduce each risk.

Self-audit and diagnostic verification questions

> **EXAMPLE**
>
> For Question 9 (16-bit index wrap): if `avail_idx = 65535` (`0xFFFF`) and `used_idx = 65527` (`0xFFF7`), the in-flight count in unsigned 16-bit modular arithmetic is `(uint16_t)(65535 - 65527) = 8` (full 8-slot ring). Publishing one more request wraps `avail_idx` to `0`, and `(uint16_t)(0 - 65528) = 8`, whereas an empty ring has `avail_idx == used_idx` (`0`). Storing only indices modulo 8 (`M_MODULO_COUNTERS`) collapses both `0` and `8` to `0`, making full and empty indistinguishable.

Audit your implementation against these questions, verifying each against runtime traces or source invariants:

1. Why does one indirect descriptor increase request capacity without increasing main-ring size?
2. Which exact store publishes device ownership, and what must happen-before it?
3. Why is the data descriptor device-writable for a read but not a write?
4. What prevents a spurious wakeup from returning an incomplete buffer?
5. Where is a used ID checked, and what state remains safe when it is invalid?
6. Why can a private generation counter not identify a stale used entry after head reuse?
7. Which resources remain pinned after your diagnostic timeout, and what event would permit release?
8. How does the direct fallback prove that feature negotiation is not ceremonial?
9. Show the counter values immediately before and after 16-bit wrap; why is full distinguishable from empty?

10. Why can a descriptor be completed but not yet reusable?
11. Which test fails if completion state is written after `wakeup`?
12. What evidence shows indirect descriptors affected the bottleneck rather than only max in-flight?
13. If p95 worsens while throughput improves, which objective decides acceptance?
14. Which claim in the memo would become invalid on a noncoherent DMA platform?
15. What additional protocol is required to recover rather than fail-stop after a device timeout?
16. Which earlier-lab invariant is most exposed when a submitter sleeps for descriptors, and what regression trace proves this driver preserved it?

An answer may consult source and traces. Guessing an undocumented guarantee is worse than identifying the boundary.

Verification checklist and defect conditions

> **WARNING**
>
> Any implementation that reuses a descriptor or buffer while DMA ownership remains active, or reports success on a nonzero Virtio status byte, is a critical safety defect that invalidates verification.

Dimension	Invariant criterion	Verification evidence
Baseline reconstruction	Clean build from pinned commit	Exact commit, passing baseline, causal read/write traces
Negotiation and fallback	Compliance with Virtio feature negotiation	Synthetic feature matrix and zero unsupported feature use
Indirect implementation	Memory-safe chained descriptor allocation	Correct persistent layout, directions, fallback, and rollback
Lifecycle and ordering	Invariants I1–I10 preserved across all transitions	I1–I10 mapped to code, legal transitions, barriers, completion order
Fault injection	Deterministic fault handling and recovery	Deterministic failure modes with bounded, documented outcomes
Mutation-killing tests	Falsification of flawed driver variants	Every named mutant fails for the intended reason; correct builds pass
Data integrity/regression	Filesystem consistency after reboot	Upstream and reboot/checksum evidence in both modes
Measurement and analysis	Controlled performance evaluation	Controlled runs, raw data, tails, concurrency, uncertainty
Architectural understanding	Complete derivation of ownership and race boundaries	Precise residual-risk disclosure and falsifiable explanations

Critical defect conditions that invalidate verification:

- device uses an unoffered feature;
- user-visible success on nonzero device status;
- descriptor or buffer reuse while device ownership may remain;
- out-of-bounds metadata access from an invalid used ID;
- data corruption hidden by rebuilding the image;
- fabricated timing units or unrun tests; or
- a copied full solution or assignment.

Suggested work allocation and exit evidence

> **REMARK**
>
> Treat the six work units as a strict dependency chain: do not begin indirect-descriptor allocation until the synthetic feature-negotiation matrix passes in direct fallback mode.

Work unit	Planned minutes	Exit evidence
environment and baseline trace	75	commit/build/test record; one read and write trace
feature decision and direct fallback	75	synthetic negotiation tests
indirect descriptor path	180	direct+indirect data-integrity tests; capacity trace
lifecycle, trace ring, and stats	135	state-edge tests and consistent snapshot
fault hooks and mutation matrix	150	failing mutant logs and passing correct logs
measurements, memo, and self-audit	105	raw runs, analysis, final memo
Total	**720**	

Exit evidence, not elapsed time, completes a work unit. A failed invariant returns the implementation to the relevant milestone.

Extension work

> **REMARK**
>
> Optional extensions are evaluated by the same mutation-killing standard as the core lab: every added capability (such as queue reset or `VIRTIO_RING_F_EVENT_IDX`) must include a dedicated fault mode and a mutant that fails when its safety check is removed.

Extensions are considered only after the core implementation passes all verification invariants:

- implement Virtio queue reset when the feature/transport supports it, with proof that old DMA ownership ended;
- propagate block errors through buffer cache and filesystem without panic;
- negotiate event-index notification suppression and measure interrupt batching;
- implement packed virtqueues;
- add multiple queues with per-CPU submission while preserving fairness; or
- port the protocol to a noncoherent DMA model with explicit cache synchronization.

Each extension needs new mutants and failure oracles. Feature count is not quality evidence.

> **REMARK**
>
> This lab bounds its scope to split virtqueues and MMIO transport in xv6-riscv rev5 under QEMU so every descriptor state transition is directly inspectable; production drivers must additionally coordinate IOMMU page tables, MSI-X per-queue interrupts, and transport-level bus reset protocols.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 5.3.1: Verifying Features-OK in Virtio MMIO Negotiation

Verification application

Trace the feature negotiation sequence mandated by the OASIS Virtio 1.3 specification for MMIO transport. Explain why reading the `STATUS` register immediately after setting `FEATURES_OK` is required before asserting `DRIVER_OK`.

The device clears FEATURES_OK if the subset of negotiated features is unacceptable.

The negotiation protocol requires:

1. Driver resets device by writing 0 to `STATUS`.
2. Driver sets the `ACKNOWLEDGE` status bit (1).
3. Driver sets the `DRIVER` status bit (2).
4. Driver reads the device-offered feature bits from `DEVICE_FEATURES`.
5. Driver writes the accepted feature subset (e.g., enabling `VIRTIO_RING_F_INDIRECT_DESC` only if offered and supported) into `DRIVER_FEATURES`.
6. Driver sets the `FEATURES_OK` status bit (8).
7. **Verification step:** Driver immediately reads `STATUS`. The specification dictates that if the device does not accept the requested feature combination, it clears `FEATURES_OK`. If the read shows `FEATURES_OK` is clear, the driver must abort initialization, as the device refuses to operate with that feature set.
8. Only if `FEATURES_OK` remains set does the driver proceed to configure queues and write `DRIVER_OK` (4).

Exercise 5.3.2: Constructing 3-Entry Indirect Descriptor Tables

Construction application

Construct the 3-entry indirect descriptor table used for a Virtio block read in xv6. Specify the fields (`addr`, `len`, `flags`, `next`) for each entry:

1. Entry 0: Request header (`struct virtio_blk_req`).
2. Entry 1: Data payload buffer (512 bytes).
3. Entry 2: One-byte status response (`uint8 status`).

Header and status are device-writable/readable according to protocol; chained with VRING_DESC_F_NEXT.

The indirect table is a contiguous array of three `struct virtq_desc`:

1. **Entry 0 (Header):**
   - `addr`: Physical address of `struct virtio_blk_req` (specifying type `VIRTIO_BLK_T_IN` and sector number).
   - `len`: `sizeof(struct virtio_blk_req)` (16 bytes).
   - `flags`: `VRING_DESC_F_NEXT` (device reads the header, points to next entry).
   - `next`: 1.
2. **Entry 1 (Data payload):**
   - `addr`: Physical address of target disk buffer `b->data`.
   - `len`: 512 bytes (BSIZE).
   - `flags`: `VRING_DESC_F_WRITE | VRING_DESC_F_NEXT` (on a read, the device writes data into memory; points to next).
   - `next`: 2.
3. **Entry 2 (Status byte):**
   - `addr`: Physical address of 1-byte completion status `info[idx].status`.
   - `len`: 1 byte.
   - `flags`: `VRING_DESC_F_WRITE` (device writes completion status; no `NEXT` flag, terminating chain).
   - `next`: 0.

The single descriptor placed in the main virtqueue table has `flags = VRING_DESC_F_INDIRECT`, pointing to this 3-entry table.

Exercise 5.3.3: Proving Safe Available Ring Memory Barrier Publication

Proof synthesis

Let `avail->ring[avail->idx % N] = head_id` and `avail->idx++`. Prove that an architecture memory fence (`__sync_synchronize()` or `dma_wmb()`) is strictly necessary between populating the descriptor/ring entry and incrementing `avail->idx`.

Show that out-of-order execution allows the peripheral to read uncommitted descriptor data.

Assume for contradiction that the memory barrier is omitted:

1. On modern superscalar or weakly ordered processors (such as multi-core RISC-V or ARM), independent memory writes can retire out of program order to memory.
2. The store updating `avail->idx` (which signals to the device that a new request exists) may become visible to the Virtio MMIO peripheral before the stores writing the descriptor contents or `avail->ring[...]` entry commit to RAM.
3. The device observes the incremented `avail->idx`, reads `avail->ring[...]`, and fetches the referenced descriptor table.
4. Because the descriptor table writes are still buffered in processor store queues, the device reads uninitialized memory or stale pointers from a prior request.
5. The device executes DMA against an invalid host memory address, causing silent memory corruption or bus faults.

Therefore, an explicit release barrier:

$$\text{Write}(\text{Descriptor Table}) \prec \text{Write}(\text{Avail Ring}) \prec \text{wmb}() \prec \text{Write}(\text{Avail Idx})$$

is strictly mandatory.

Exercise 5.3.4: Formalizing the Driver Request Lifecycle State Machine

Formalisation transfer

Formalize the state machine for driver request slots:

$$S \in \{\text{FREE}, \text{PREPARED}, \text{IN_FLIGHT}, \text{COMPLETED}, \text{REAPED}\}.$$

Specify the legal transitions, triggering events, and the invariant that prevents duplicate interrupt completions from corrupting driver state.

Define transitions: allocate -> prepare -> submit -> interrupt/done -> wait_reap -> free.

- **State set:** $\mathcal{S} = \{\text{FREE}, \text{PREPARED}, \text{IN_FLIGHT}, \text{COMPLETED}, \text{REAPED}\}$.
- **Transitions:**
  1. $\text{FREE} \xrightarrow{\text{alloc}} \text{PREPARED}$: Driver allocates a free request index and populates indirect descriptors.
  2. $\text{PREPARED} \xrightarrow{\text{submit}} \text{IN_FLIGHT}$: Descriptors published to available ring, doorbell notified.
  3. $\text{IN_FLIGHT} \xrightarrow{\text{interrupt}} \text{COMPLETED}$: Device places ID in used ring and triggers interrupt; handler verifies `state == IN_FLIGHT`, records status, marks `COMPLETED`, and wakes waiter.
  4. $\text{COMPLETED} \xrightarrow{\text{reap}} \text{REAPED}$: Sleeping waiter awakens, reads status, reclaims buffer, and marks `REAPED`.
  5. $\text{REAPED} \xrightarrow{\text{free}} \text{FREE}$: Slot returned to allocator pool with incremented generation.
- **Duplicate completion defense:** If an interrupt arrives with descriptor ID i when $\text{state}[i] \neq \text{IN_FLIGHT}$ (for instance, if i is already `COMPLETED` or `FREE`), the driver detects a protocol violation or duplicate interrupt. It ignores the spurious notification and flags an error without waking waiters or double-releasing buffers.

Exercise 5.3.5: Establishing Hardware Quiescence Before Buffer Reclaim

Description synthesis

Explain why a driver cannot safely free a DMA buffer when a timeout expires in `virtio_disk_rw()`. Detail the sequence of hardware reset operations required to achieve verified device quiescence before memory reclamation.

The device engine may still access host memory unless reset to 0 and verified.

- **The hazard:** A software timeout indicates that the host has waited longer than its deadline. It provides zero guarantees regarding device controller state. If the driver frees the kernel buffer `struct buf *b` or returns it to the buffer cache while the device DMA engine is still active, the device may write 512 bytes into the buffer milliseconds later, corrupting whatever new file block or page table now occupies that memory.
- **Quiescence protocol:** To safely reclaim memory after a timeout or catastrophic device hang:
  1. Driver writes `0` to the MMIO `STATUS` register to initiate a full device reset.
  2. Driver polls `STATUS` until reading `0`, confirming the hardware reset has completed and all internal DMA pipelines are aborted.
  3. Driver unmaps or revokes all IOMMU mappings associated with outstanding virtqueue requests.
  4. Only after reset is acknowledged as complete is it safe to mark in-flight request slots as failed and release data buffers.

References & Further Reading

- xv6–riscv revision 5 source, pinned commit
- xv6–riscv revision 5 `kernel/virtio_disk.c`
- xv6–riscv revision 5 `kernel/virtio.h`
- xv6–riscv revision 5 book, interrupts and device drivers

- xv6–riscv license
- OASIS Virtual I/O Device (Virtio) Version 1.3
- RISC-V Privileged Architecture

Persistence

SECTION DETAIL

Filesystem Interface and Names

A configuration daemon opens `/srv/app/config`, validates its contents, and keeps the returned descriptor. An administrator writes a replacement file and atomically renames it over `config`. New calls to `open("/srv/app/config", ...)` reach the replacement; the daemon's existing descriptor still reaches the old object. If the old name was its last directory entry, the old object has no pathname at all, yet the daemon can continue reading it.

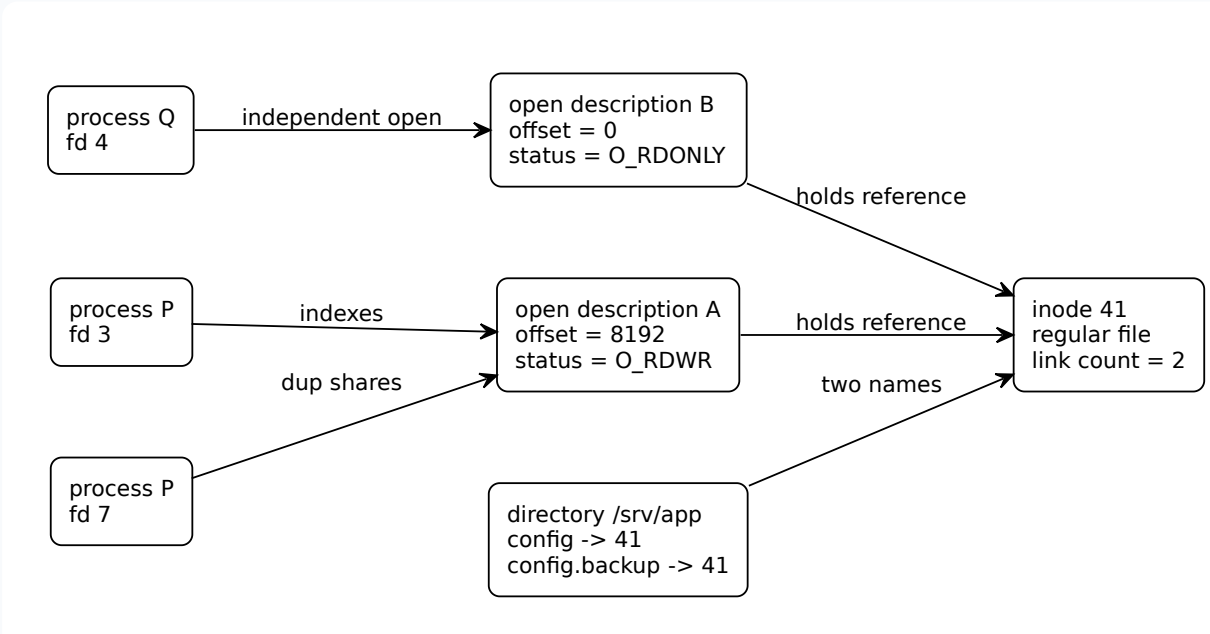
The apparent contradiction disappears when a pathname and an open file are modeled as different references. A pathname is a request to traverse a mutable namespace. A descriptor is a process-local handle to an already opened description. Treating both as "the file" causes stale-read bugs, authority races, and incorrect durability claims.

Five objects hidden behind one integer

Assume a local POSIX-like filesystem, one mount namespace, no I/O error, and no concurrent namespace mutation until a trace explicitly introduces one. The model contains five distinct objects:

Object	Identity	Mutable state relevant here	Typical lifetime
pathname string	bytes supplied by a caller	components may be interpreted relative to a directory	one operation
directory entry	pair `(name, object-id)` inside a directory	name and target mapping	until unlink or rename
filesystem object	inode-like identity	type, size, link count, permissions, block map	while linked or referenced
open file description	kernel record for an open instance	current offset and file-status flags	while any descriptor refers to it
file descriptor	small integer in one process	descriptor flags such as close-on-exec	until close or table replacement

A symbolic link is itself a filesystem object whose payload is another pathname. A hard link is another directory entry for an existing object; it is not a pointer to another name. A process may have several descriptors for one open file description, and several independently opened descriptions may refer to one object.



The diagram separates two graphs. Directory entries form the **namespace graph**. Descriptors and open descriptions form the **open-reference graph**. Unlinking an edge in the first graph does not revoke edges in the second.

INVARIANT

A descriptor operation first resolves the descriptor to a live open file description; it does not resolve the original pathname again. The open description keeps the underlying object alive even when its link count becomes zero.

Retrieval checkpoint: what does `dup` copy?

Without consulting the answer, record which state is shared after `b = dup(a)`: pathname, descriptor number, descriptor flags, current offset, file-status flags, and inode identity.

`a` and `b` are different descriptor numbers in one table. They refer to the same open file description, so they share its current offset and file-status flags; they consequently reach the same inode. Descriptor flags are per descriptor and are not thereby shared. No pathname is stored as the descriptor's operative identity. If the process calls `open` a second time instead of `dup`, the new descriptor normally receives a distinct open file description and therefore a distinct offset.

A response that says "everything is copied" has collapsed three layers: descriptor entry, open description, and inode. A response that gives each descriptor an independent offset has confused `dup` with a second `open`.

FILE DESCRIPTORS, OPEN DESCRIPTIONS, AND NAMESPACE INVARIANTS

Evaluate each statement regarding POSIX file descriptors, directory entries, and file lifecycles:

Answer: Unlinking a file removes its directory entry while existing open file descriptors continue reading its data.; Duplicating a descriptor with dup creates a new descriptor that shares the exact file offset and status flags.

Unlinking removes the directory name, but the inode and data blocks remain allocated until the last open description is closed. Descriptors duplicated with dup share the identical underlying open file description (and therefore share offset). Atomic rename replaces names in the namespace but does not force volatile caches to disk without fsync, and independent open calls create separate open file descriptions with independent offsets.

Worked trace: shared and independent offsets

File `records` contains sixteen bytes `abcdefghijklmnop`. Reads below are successful and no signal interrupts them.

```
int a = open("records", O_RDONLY);  // open description X, offset 0
int b = dup(a);                     // also X
int c = open("records", O_RDONLY);  // open description Y, offset 0
```

Execute `read(a, p, 4)`, `read(b, q, 3)`, `read(c, r, 2)`, then `lseek(a, 1, SEEK_SET)` and `read(b, s, 3)`.

Event	bytes returned	offset of X after event	offset of Y after event
initial state	—	0	0
`read(a, ..., 4)`	`abcd`	4	0
`read(b, ..., 3)`	`efg`	7	0
`read(c, ..., 2)`	`ab`	7	2
`lseek(a, 1, SEEK_SET)`	—	1	2
`read(b, ..., 3)`	`bcd`	4	2

The descriptor used for an operation does not determine the offset object. `a` and `b` both mutate X. `c` reaches the same inode through Y but cannot observe X's offset changes.

Application checkpoint: inherited descriptors

A parent opens an output file, calls `fork`, and both processes write fixed-size records through the inherited descriptor without seeking. State the property that descriptor sharing supplies and the property it does not supply.

The inherited descriptors refer to one open file description, so they share one current offset. POSIX specifies relevant descriptor operations atomically with respect to the effects on regular files, preventing both writes from independently using the same old offset under the specified interface semantics. This does **not** impose an application-level order between parent and child records, combine several writes into one transaction, make buffered library I/O safe after `fork`, or make either record durable. If record order is part of correctness, the processes need an explicit protocol.

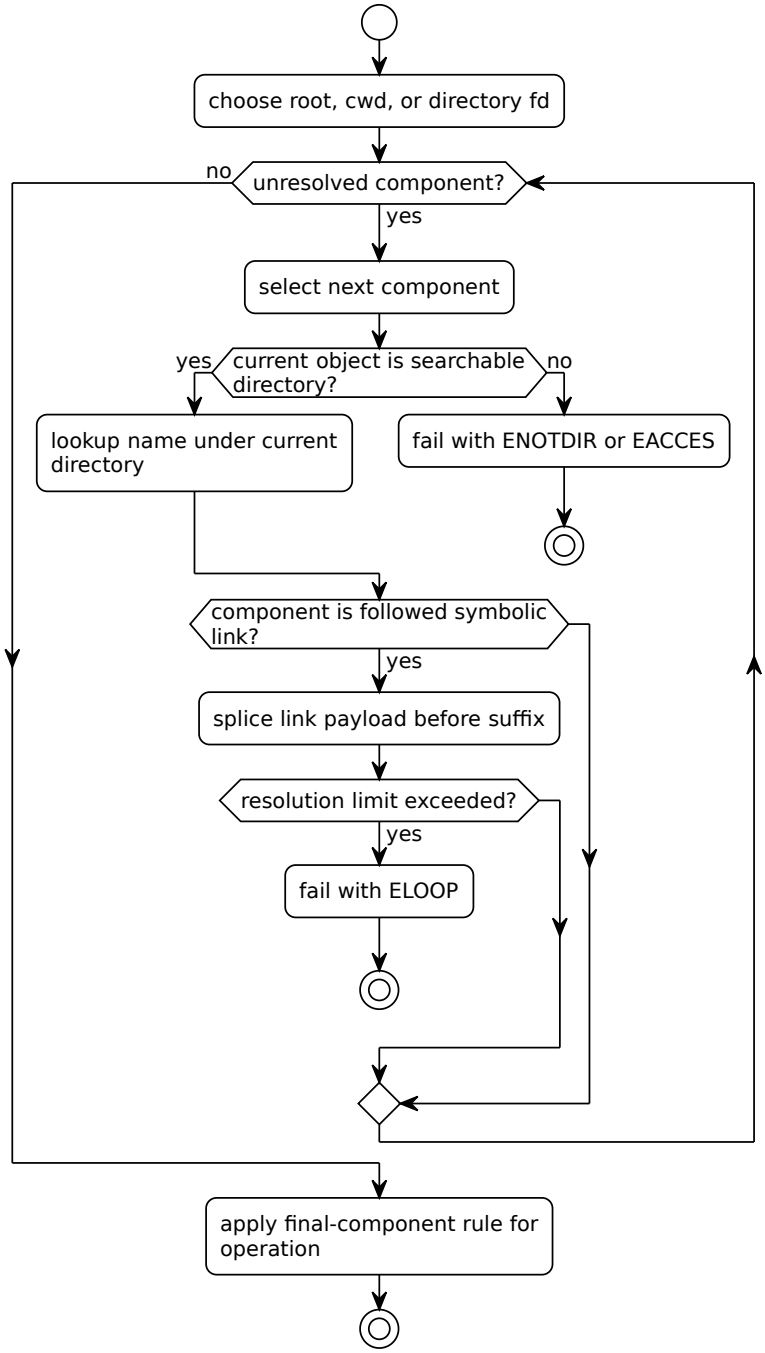
Path resolution is a stateful walk

For an absolute pathname, resolution begins at the process's root directory. For a relative pathname, it begins at the current working directory or at a directory descriptor supplied to an `*at` interface. For each non-final component, the kernel must:

1. verify that the current object is a directory and that search is authorized;

2. look up the component in that directory;
3. follow a symbolic link when the operation's rules require it, splicing the link payload into the unresolved suffix; and
4. continue from the resulting directory, crossing a mount point if the namespace contains one.

`.` selects the current directory. `..` selects its parent subject to root and mount semantics. The final component is operation-specific: **`open`** normally follows a final symbolic link, **`lstat`** examines the link itself, and **`rename`** renames a final link rather than its target. A create operation resolves the parent and leaves the absent final name for insertion.



Worked trace: a symbolic-link pivot

Assume this initial namespace and a process whose current directory is `/work`:

```
/work/input        -> symlink payload "../safe/report"
/safe/report       -> inode 80
/attacker/report   -> inode 99
```

The process first calls `stat("input", ...)`, validates inode 80's owner, and later calls `open("input", O_RDONLY)`. Between the calls, an attacker who can modify `/work` replaces `input` with a symlink to `../attacker/report`. The validation and use are two independent pathname walks:

Time	operation	resolved object
t_0	`stat("input")`	inode 80
t_1	attacker replaces directory entry	namespace now targets inode 99
t_2	`open("input")`	inode 99

No corruption of the kernel's lookup algorithm occurred. The application incorrectly treated a pathname result as a stable capability. Opening first and validating with `fstat` binds validation and subsequent I/O to the same open object. When policy requires confinement beneath a trusted directory on Linux, a trusted directory descriptor plus `openat2` resolution constraints can also reject symlink, mount, or escape behavior; those flags are Linux interfaces, not portable POSIX guarantees.

Transfer checkpoint: close the authority race

A privileged image service must open a caller-supplied relative name beneath an already opened directory `tenant_dir`. The caller can concurrently rename entries and create symbolic links. Specify a safe ordering and state what remains unproved.

Keep `tenant_dir` as an open directory descriptor acquired from trusted configuration. Resolve relative to that descriptor, not the process-wide current directory. On Linux, use `openat2` with a policy-appropriate combination such as `RESOLVE_BENEATH` or `RESOLVE_IN_ROOT`, plus restrictions on symbolic links or mount crossings when the service's contract forbids them. Open with the least required access, then inspect the returned descriptor with `fstat` and perform I/O through that descriptor.

The protocol binds later I/O to the object actually opened. It does not prove that file contents are benign, that the trusted directory itself was acquired securely, that every filesystem implements identical remote semantics, or that a kernel vulnerability cannot violate the boundary. Listing flags without deriving them from the service's path policy is not a complete answer.

Links, removal, and replacement

For a regular file, the link count records directory entries that name the object. Open descriptions contribute references but not hard links. Storage may be reclaimed only after both the namespace link count and the relevant in-memory references reach zero.

Worked trace: unlink while open

Initial state: `/tmp/job.out -> inode 17`, `nlink(17)=1`, and process A has descriptor 5 open on inode 17.

Step	namespace	link count	open references	result
0	`job.out -> 17`	1	1	named and open
1: `unlink("/tmp/job.out")`	no entry	0	1	new lookup fails; fd 5 still works
2: A writes through fd 5	no entry	0	1	inode 17 grows but remains unnamed
3: A closes fd 5	no entry	0	0	object becomes reclaimable

Creating a new `/tmp/job.out` after step 1 creates or links some other object. It does not reattach the old descriptor by name.

`rename(old, new)` changes namespace edges. Within the scope where the operation is supported, another thread must not observe `new` absent during replacement: it refers to either the prior target or the renamed object. If `new` existed and its last link is removed, open references to that displaced object still keep it alive. Cross-filesystem rename may fail with `EXDEV` because one atomic metadata transaction cannot generally span unrelated implementations.

> ***Boundary.*** *Namespace atomicity is not storage durability. A successful `rename` establishes an allowed concurrent-observation result. It does not by itself prove which names or file contents survive power loss.*

Application checkpoint: enumerate legal observations

`old` names inode A and `new` names inode B. Process R repeatedly resolves `new` while process W executes successful `rename(old, new)`. No other namespace changes occur. Which inode identities may R obtain? What happens to an already open descriptor for B?

Each completed resolution of `new` may obtain B before the rename linearization point or A after it. The specified replacement does not expose an interval in which `new` is missing. A descriptor opened on B before replacement continues to refer to B; B's content is reclaimed only after its last link and open reference are gone. An answer claiming that R's existing descriptor switches from B to A has made descriptors resolve names repeatedly. An answer claiming that the result necessarily survives a crash has imported a durability guarantee that the premise does not provide.

Deriving a crash-aware replacement protocol

Consider a local filesystem whose documentation states that `fsync(file)` makes the file's required data and metadata persistent and `fsync(directory)` persists directory-entry updates. Under those explicit assumptions, an application replacing `config` can seek either of two recovery states: the complete old object or the complete new object.

```
1. create a temporary file in the target directory without reusing an
attacker-controlled name
2. write the complete new representation
3. fsync the temporary file and check the result
4. rename the temporary name over config and check the result
5. fsync the containing directory and check the result
```

Step 3 prevents a durable new name from exposing unwritten or stale content under the assumed interface. Step 4 provides atomic namespace replacement to concurrent readers. Step 5 requests persistence of the namespace update. Every failure path must preserve the old file or report that recovery is uncertain; ignoring an `fsync` error destroys the protocol's evidence.

This recipe is not universal folklore. Filesystem, device-cache, network, and platform semantics can narrow or alter it. It does not make several files into one atomic transaction. It also does not update long-lived reader descriptors; readers that require fresh configuration must reopen or use an explicit version-notification protocol.

REMARK

The three-level POSIX state model—per-process file descriptors, system-wide open file descriptions (holding offset and status flags), and underlying inodes (holding link count and data)—is a mandatory API contract that every storage backend in the subsequent implementation and journaling lessons must satisfy.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 6.1.1: Tracing Shared versus Independent File Offsets Across dup and open

Verification application

A file `log.txt` contains 20 bytes `0123456789abcdefghij`. A process executes:

```
int fd1 = open("log.txt", O_RDONLY);
int fd2 = dup(fd1);
int fd3 = open("log.txt", O_RDONLY);
char buf[5];
read(fd1, buf, 4); // Op A
read(fd2, buf, 4); // Op B
read(fd3, buf, 4); // Op C
```

1. State the file offset for `fd1`, `fd2`, and `fd3` after Op A.
2. State the byte string read into `buf` by Op B and Op C.

3. Draw the relationship between descriptor table entries, open file descriptions, and the in-memory inode.

dup shares the open file description and offset; separate open calls create new open file descriptions.

1. **Offsets after Op A:** `fd1` and `fd2` point to the identical open file description (OFD 1). When `read(fd1, buf, 4)` consumes 4 bytes, OFD 1's offset advances from 0 to 4. Therefore, both `fd1` and `fd2` now observe offset 4. `fd3` refers to independent OFD 2, whose offset remains 0.
2. **Read results:**
   - Op B (`read(fd2, buf, 4)`): Reads 4 bytes from OFD 1 starting at offset 4, returning `"4567"`. OFD 1's offset advances to 8.
   - Op C (`read(fd3, buf, 4)`): Reads 4 bytes from OFD 2 starting at its initial offset 0, returning `"0123"`. OFD 2's offset advances to 4.
3. **Architecture graph:**
   - `fd1` (slot in proc table) → Open File Description 1 (offset = 8) → Inode (`log.txt`).
   - `fd2` (slot in proc table) → Open File Description 1 (offset = 8).
   - `fd3` (slot in proc table) → Open File Description 2 (offset = 4) → Inode (`log.txt`).

Exercise 6.1.2: State Transitions for Unlinking Open Files with Active Descriptors

Construction application

Process P_1 opens `/var/run/worker.pid` with descriptor 3. While P_1 runs, Process P_2 invokes `unlink("/var/run/worker.pid")`.

1. State the inode link count (`i_nlink`) and in-core reference count (`i_count`) before and after the `unlink`.
2. Can P_1 continue to read and write through descriptor 3 after the `unlink` succeeds?
3. When are the file's data blocks and inode metadata physically deallocated by the filesystem?

The directory link count drops to 0, but the kernel in-core reference count remains 1 while open.

1. **Reference counts:**
   - Before `unlink`: On-disk `i_nlink = 1` (directory entry exists); in-core `i_count = 1` (P_1's open file description holds an active reference).
   - After `unlink`: The directory entry is removed from the directory block, so on-disk `i_nlink = 0`. However, in-core `i_count` remains 1 because P_1's descriptor 3 is still active.
2. **Access persistence:** Yes. POSIX dictates that descriptor operations operate through the open file description, not pathname resolution. P_1 can continue reading, writing, and seeking through descriptor 3 without error. Any new attempt by other processes to `open("/var/run/worker.pid")` will return `ENOENT` (unless a new file of that name is created).
3. **Deallocation trigger:** When P_1 executes `close(3)` (or when P_1 terminates), the kernel decrements `i_count` from 1 to 0. Finding `i_count == 0` and `i_nlink == 0`, the VFS/filesystem driver deallocates all mapped data blocks to the free bitmap and marks the inode free on disk.

Exercise 6.1.3: Linearization Points and Concurrent Reader Observations Across rename

Verification transfer

Process W prepares an updated database file by creating temporary file `db.tmp` and executing `rename("db.tmp", "db.live")`. Concurrently, reader processes $R1, R2, \ldots$ repeatedly call `open("db.live", O_RDONLY)`.

1. Prove that no reader process can ever observe a "file not found" (`ENOENT`) error during the rename operation.
2. If reader R_1 opened `db.live` prior to the rename linearization point, what content does R_1 observe after the rename commits?
3. Explain why `rename` alone does not guarantee that the new database file survives a power failure.

POSIX rename is atomic in namespace; readers either get the old inode or new inode; durability needs fsync.

1. **Namespace atomicity invariant:** POSIX specifies that `rename(old, new)` is atomic with respect to namespace lookups. The filesystem directory implementation updates or replaces the directory entry in a single atomic directory transaction. At any point in time, a pathname resolution traversing `"db.live"` either resolves to the old inode I_{old} (before the directory commit) or the new inode I_{new} (after the commit). There is no intermediate window where the directory entry is empty or absent.
2. **Reader observation:** R_1's descriptor is bound to the open file description of the old inode I_{old}. Modifying the directory pointer does not change existing open descriptors. R_1 continues reading the original database contents until it closes its descriptor.
3. **Durability boundary:** `rename` guarantees directory namespace atomicity in RAM/dcache. It does not force dirty directory blocks or file data from volatile OS page cache to disk platters/flash. If power fails immediately after `rename`, storage may restore to an older state, write back out-of-order blocks, or expose a 0-byte file unless preceded by `fsync(fd_tmp)` and followed by `fsync(fd_dir)`.

Exercise 6.1.4: Designing a Crash-Resilient and Race-Free Atomic Publication Protocol

Construction synthesis

Specify the exact, crash-resilient 5-step sequence required to replace a configuration file `/etc/daemon.conf` using POSIX system calls, ensuring that crash recovery either yields the complete old file or complete new file without corrupt half-writes.

Create temp in same filesystem -> write -> fsync file -> rename -> fsync directory.

The atomic, crash-resilient publication protocol proceeds as follows:

1. **Create temporary file in same directory:** Call `int fd = open("/etc/daemon.conf.tmp.PID", O_WRONLY | O_CREAT | O_EXCL, 0644)`. The temporary file must reside in the same filesystem/mount as `/etc` so that subsequent `rename` is a local directory update rather than a cross-device `EXDEV` copy.
2. **Write complete contents:** Execute a write loop ensuring all buffer bytes are written, handling partial writes and interrupted system calls (`EINTR`).
3. **Sync file data and metadata:** Call `fsync(fd)` and check the return code. This guarantees all file data blocks and the inode are durable on non-volatile media before publication. Close `fd`.
4. **Atomic rename:** Call `rename("/etc/daemon.conf.tmp.PID", "/etc/daemon.conf")` and check the return code. This atomically swings the directory pointer from old to new.
5. **Sync containing directory:** Open the directory `int dfd = open("/etc", O_RDONLY | O_DIRECTORY)`, execute `fsync(dfd)`, and close `dfd`. This commits the updated directory entry to durable storage.

Exercise 6.1.5: Symbolic Link Traversal Limits and Detection of Cyclic References

Description application

Explain how an operating system path resolution routine traverses symbolic links. Describe how a cyclic reference (e.g., `ln -s /a/b /a/c` and `ln -s /a/c /a/b`) is detected and handled to prevent infinite kernel loops.

Track symlink traversal count against SYMLOOP_MAX (typically 32 to 40) and return ELOOP.

- **Traversal mechanism:** When `namei` or path walk encounters a directory entry pointing to an inode of type `S_IFLNK`, the kernel reads the symlink's payload target string into a buffer. If the target begins with `'/'`, traversal restarts from the root directory; otherwise, it resolves relative to the current directory. The remaining components of the original pathname are appended to the link path, and resolution continues.
- **Cycle detection and loop bounds:** Filesystems do not detect cycles at creation time (since creating a dangling or cyclic link is permissible). Instead, the path resolver maintains an integer counter tracking the number of nested symbolic links traversed during a single resolution call. POSIX defines `SYMLOOP_MAX` (typically 32 in modern Linux / BSD). If the counter exceeds `SYMLOOP_MAX`, the kernel halts traversal and returns `-ELOOP` (Too many levels of symbolic links), protecting the kernel stack from overflow and terminating the infinite loop.

References & Further Reading

- The Open Group, POSIX.1–2024 Base Definitions, Chapter 4: General Concepts — pathname resolution and filesystem-operation semantics.
- The Open Group, POSIX.1–2024 `open()` and `openat()` — descriptor creation and final-component behavior.
- The Open Group, POSIX.1–2024 `rename()` and `renameat()` — atomic namespace replacement and boundary conditions.
- Linux man-pages, `openat2(2)` — Linux-specific resolution constraints for race-resistant path handling.
- Linux kernel documentation, Overview of the Virtual File System — dentries, inodes, open file objects, and operation dispatch.

- MIT PDOS, xv6 RISC-V book and immutable xv6-rev5 source tree — the compact descriptor, inode, directory, and pathname implementation used for teaching-kernel examples.

SECTION DETAIL

Filesystem Implementation

Appending one byte can require more than one byte of work. If a file has exhausted its direct block pointers, the append may allocate an indirect block, allocate a data block, set a bitmap bit, install two pointers, extend the inode size, dirty several cache entries, and arrange for all of those changes to become recoverable. A design that models `write` as "copy bytes to disk" cannot explain either its cost or its failure modes.

The implementation problem is to maintain one connected graph across several representations:

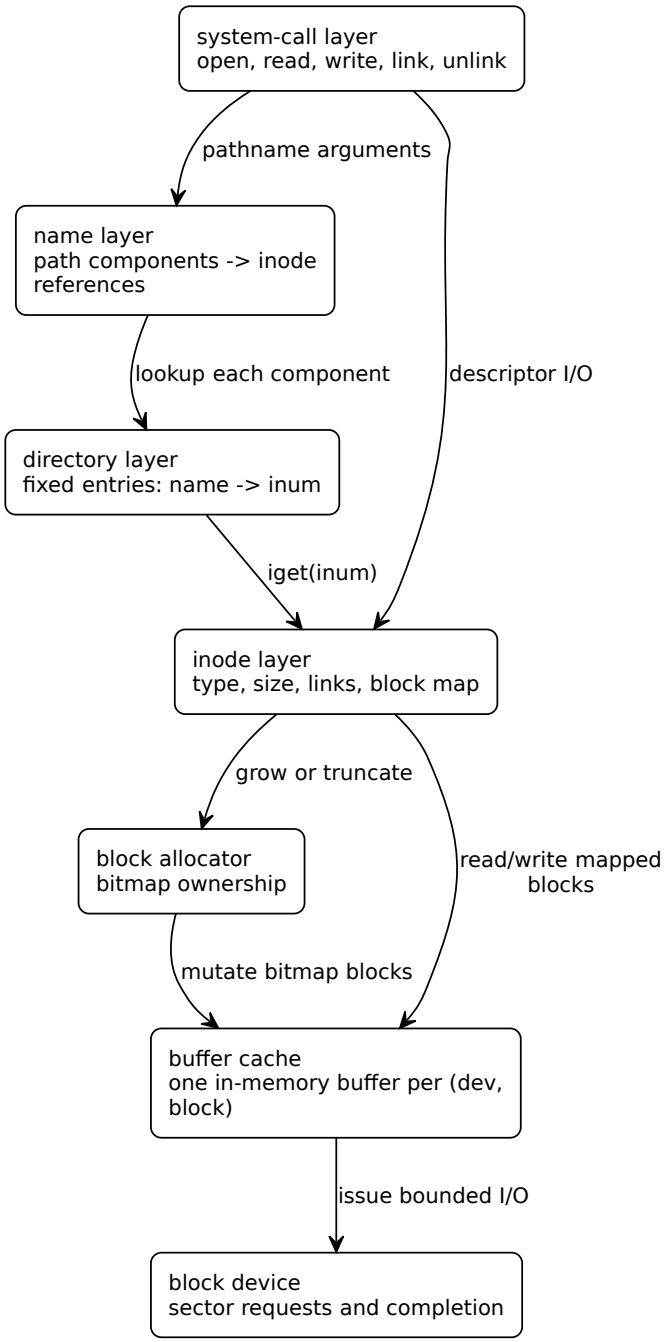
$$\text{name} \rightarrow \text{inode number} \rightarrow \text{block map} \rightarrow \text{device blocks}.$$

Every cache duplicates part of that graph in memory. Every allocator mutates its ownership relation. Correctness therefore depends less on individual helper functions than on invariants spanning layers.

Every xv6-specific constant and control-path trace below is pinned to official tag `xv6-riscv-rev5`, commit `7d7adbb1b0acbd67c9766a20d0f9900fef2789fa`. A moving branch may change the format or helper behavior and is not evidence for these derivations.

A layered contract, not a call-stack accident

The concrete model uses one block device, fixed-size blocks, one filesystem, inodes, fixed-size directory entries, a bitmap allocator, and a write-back buffer cache. System calls may execute concurrently. Device completion can sleep and wake a process. The model makes no crash-ordering assumption.



Each arrow is a contract boundary:

- the name layer returns a referenced inode identity, not an unprotected pointer into an on-disk block;
- the inode layer maps a logical byte range to blocks without exposing allocator internals;
- the allocator owns the rule that one block cannot belong to two live allocations;
- the buffer cache serializes access to a cached block and prevents two divergent in-memory copies of the same device block; and

- the device layer reports completion or error rather than deciding filesystem policy.

Production kernels add virtual-filesystem dispatch, page-cache integration, writeback policy, extents, per-CPU structures, and many caches. The compact layers remain useful because their invariants can be enumerated completely.

Deriving the disk map

Freeze these parameters for the traces:

parameter	value	consequence
block size B	1024 bytes	every region is addressed in 1 KiB units
disk-block number	32-bit unsigned	one indirect block stores 256 pointers
on-disk inode size	64 bytes	16 inodes fit in one block
direct addresses	12	logical blocks 0 through 11 are direct
directory name bytes	14	one entry is a 2-byte inode number plus 14 name bytes

The disk regions are ordered:

```
block 0       boot
block 1       superblock
next nlog     recovery log
next ...      inode blocks
next ...      free-space bitmap blocks
remainder     file and directory data blocks
```

The superblock records region starts and sizes. Derived constants are

$$IPB = \frac{B}{\operatorname{sizeof}(dinode)} = \frac{1024}{64} = 16,$$

$$BPB = 8B = 8192 \text{ allocation bits per bitmap block},$$

and

$$N_{indirect} = \frac{B}{4} = 256.$$

For inode number i and data block b:

$$inodeBlock(i) = inodeStart + \left\lfloor \frac{i}{IPB} \right\rfloor, \qquad inodeSlot(i) = i \bmod IPB,$$

$$bitmapBlock(b) = bitmapStart + \left\lfloor \frac{b}{BPB} \right\rfloor, \qquad bitmapBit(b) = b \bmod BPB.$$

Worked trace: locate two pieces of metadata

Let `inodeStart = 32` and `bitmapStart = 48`.

- Inode 37 is in block $32 + \lfloor 37/16 \rfloor = 34$, slot $37 \bmod 16 = 5$.
- Data block 9000 is represented in bitmap block $48 + \lfloor 9000/8192 \rfloor = 49$, bit 808 of that block. Byte $\lfloor 808/8 \rfloor = 101$ contains the bit; mask $1 << (808 \bmod 8) = 1$ selects it.

The formulas reveal a boundary condition: inode 16 starts a new inode block, and data block 8192 starts a new bitmap block. Tests that use only zero and small identifiers will not exercise those transitions.

Derivation checkpoint: resize the model

The block size becomes 4096 bytes, inode size remains 64 bytes, and block numbers remain 32 bits. Compute `IPB`, `BPB`, and `NINDIRECT`. Then state which values are format commitments rather than values the kernel may change while mounting an existing image.

$IPB = 4096/64 = 64$, $BPB = 8 \cdot 4096 = 32768$, and $NINDIRECT = 4096/4 = 1024$. Block size, on-disk inode layout, pointer width, region locations, and interpretation of bitmap bits are format commitments: changing them without a versioned migration changes the meaning of existing bytes. An in-memory cache capacity can change without rewriting the disk format, provided all observable behavior and invariants remain intact.

A response giving `BPB=4096` has counted bytes rather than bits. A response treating `sizeof(struct dinode)` as automatically stable across compilers has missed why on-disk layouts need fixed-width fields, layout checks, and versioning.

ON-DISK LAYOUT ARITHMETIC AND INODE INVARIANTS

With 1024-byte blocks, 64-byte inodes, inodeStart = 32, and bitmapStart = 48, evaluate each structural mapping claim:

Answer: Inode 20 resides in disk block 33 at slot index 4 within that block because sixteen inodes fit per block.; Data block 8192 is tracked by bitmap block 49 at bit offset 0 because each 1024-byte bitmap block tracks 8192 bits.

With 16 inodes per block (1024/64), inode 20 is at block 32 + floor(20/16) = 33, slot 20 mod 16 = 4. With 8192 bits per block (8 x 1024), block 8192 is tracked by bitmap block 48 + floor(8192/8192) = 49, bit 0.

Inodes split identity from residency

An on-disk inode stores persistent metadata: object type, device identifiers where applicable, link count, byte size, and block addresses. An in-memory inode adds transient state: device and inode identity, reference count, validity, and a sleepable lock protecting loaded metadata.

These states must not be conflated:

- `ref > 0` means kernel code holds an in-memory reference; it does not mean a directory names the inode.
- `nlink > 0` means namespace entries name the object; it does not count open descriptors.
- "valid in cache" means metadata has been read from disk; it does not mean a caller currently holds the inode lock.
- the inode number identifies the object within a filesystem; the address of an in-memory cache slot is not persistent identity.

The inode cache should return the same in-memory entry for a live `(device, inode-number)` pair. Otherwise two locks could protect two copies of one object's size and block map, making both updates appear locally synchronized while losing one globally.

> ***Cache-identity invariant.*** *At most one active in-memory inode entry represents a given* `(dev, inum)`*. Its reference count prevents reuse; its inode lock serializes metadata inspection and mutation.*

Worked trace: why reference and link counts differ

Object 52 has two names, `a` and `b`, and one open descriptor. Its state is `nlink=2`, with at least one in-memory reference from the open file.

1. `unlink("a")` removes one directory entry and decrements `nlink` to 1.
2. `unlink("b")` removes the other and decrements `nlink` to 0.
3. The open descriptor still holds an inode reference, so truncation and inode reuse are postponed.
4. Closing the descriptor drops the final reference. The implementation may then free data blocks and mark the inode unused inside an appropriate recoverable operation.

Freeing at step 2 would let the allocator hand a still-open block to another file. Deferring forever because `nlink==0` but ignoring the reference transition would leak it.

Boundary checkpoint: detect the alias bug

An inode cache searches without a lock, finds no entry for `(1,52)`, and two CPUs each install that identity into a different free slot. Each slot has its own sleep lock. Explain why adding locks around `writei` cannot repair the design.

The first failure is cache identity, before either `writei` begins. The two locks guard different memory objects, so both writers can hold a lock while independently reading and overwriting inode 52's size or block pointers. The lookup-and-reserve transition must be atomic under a cache-level lock or equivalent uniqueness mechanism. Per-inode locking is meaningful only after identity uniqueness has been established.

Mapping byte offsets to blocks

With 12 direct pointers and one single-indirect pointer, logical block number l maps as follows:

$$map(l) = \begin{cases} inode.addrs[l], & 0 \le l < 12, \\ indirect[l-12], & 12 \le l < 12 + 256, \\ \text{out of range}, & l \ge 268. \end{cases}$$

The maximum representable file is $268 \cdot 1024 = 274{,}432$ bytes in this model. The maximum valid byte offset is one less than that value. A zero pointer denotes a hole or an unallocated block according to the filesystem's policy; a growing write may allocate it, while a read from a sparse hole may return zeros in a filesystem that supports sparse files. Compact xv6 allocation eagerly allocates blocks reached by writes and is not a complete sparse-file implementation.

Worked trace: offset through the indirect block

Translate byte offset $13 \cdot 1024 + 37 = 13{,}349$.

1. Logical block is $\lfloor 13{,}349 / 1024 \rfloor = 13$.
2. In-block offset is $13{,}349 \bmod 1024 = 37$.
3. Since $13 \ge 12$, consult indirect entry $13 - 12 = 1$.
4. If the inode's indirect pointer names block 700 and word 1 of block 700 contains `912`, the requested byte is byte 37 of device block 912.

This access can require two cache lookups: one for indirect block 700 and one for data block 912. Caching the inode does not cache the indirect block's contents inside the inode.

Application checkpoint: boundary test set

Choose the smallest set of byte offsets that distinguishes direct mapping, the direct-to-indirect transition, the last representable byte, and the first rejected byte.

One suitable set is:

- `0`: first byte of direct logical block 0;
- `12*1024 - 1 = 12287`: final byte reachable through the last direct pointer;
- `12*1024 = 12288`: first byte requiring indirect entry 0;
- `268*1024 - 1 = 274431`: final representable byte;
- `268*1024 = 274432`: first out-of-range byte.

Adding offsets around an inode-block and bitmap-block boundary tests different mechanisms and is valuable, but it is not required to distinguish the block-map branches themselves.

Directory files implement the namespace edge

In the compact format, a directory is a file whose data is an array of fixed-size entries:

```
struct dirent {
    uint16_t inum;
    char name[14];
};
```

An entry with inode number zero is free. Lookup scans entries, compares bounded names, and returns a referenced inode for the matched number. Insertion must reject duplicate names, find or append a free slot, and update the directory inode's size if it grows. Removal clears an entry and adjusts the target's link count under the operation's locking and recovery protocol.

Fixed entries make traversal easy to inspect, but impose a severe name limit and linear lookup. Production directories commonly use hashed or tree indexes and variable-length records. Those structures optimize lookup; they do not change the abstract requirement that each live directory

edge targets a valid object under the filesystem's rules.

Worked trace: `/usr/log` to a data byte

Assume root inode 1, and directory contents:

```
inode 1: "." -> 1, ".." -> 1, "usr" -> 18
inode 18: "." -> 18, ".." -> 1, "log" -> 37
inode 37: regular file, size 1500, direct[0] = 620, direct[1] = 777
```

Resolving byte 1200 of `/usr/log` performs this conceptual trace:

stage	state transition
root selection	acquire reference to inode 1
component `usr`	lock inode 1, scan its directory blocks, obtain referenced inode 18, release inode 1
component `log`	lock inode 18, scan entries, obtain referenced inode 37, release inode 18
type and range check	lock inode 37; verify regular file and `1200 < size`
block map	logical block 1, in-block offset 176, direct pointer 777
data access	acquire buffer `(device,777)` and copy from byte 176

The trace elides permission checks and mount traversal by assumption. It still exposes every identity and lock handoff needed to inspect a bug.

Allocation and the buffer cache

Allocation scans bitmap blocks for a zero bit, sets one bit while synchronized, and normally zeroes the selected data block before exposing it to a file. Zeroing is a confidentiality and determinism obligation: otherwise a new file could read bytes left by a previous owner.

The core allocation invariants are:

1. each allocated block has its bitmap bit set;

2. no live metadata path assigns one block to two owners unless the format explicitly supports sharing;
3. a block is initialized before user-visible reads can reach it;
4. freeing removes all live metadata references before clearing ownership; and
5. allocator changes and pointer changes belong to one recoverable update protocol.

The buffer cache maps `(device, block-number)` to one in-memory buffer. A cache lock protects the identity map and replacement metadata; a per-buffer sleep lock protects the bytes. Holding a global spinlock across device I/O would block unrelated CPUs and may sleep in an invalid context. Releasing all identity protection before marking a victim reserved can create duplicate buffers.

Worked mutation: crossing into indirect addressing

A file is exactly 12 KiB long, with all direct blocks allocated and no indirect block. One byte is written at offset 12 KiB. Ignoring recovery order, a correct growth path must make these logical changes:

1. allocate and zero an indirect block;
2. store its block number in the inode's indirect slot;
3. allocate and zero a data block;
4. place the data-block number in indirect entry 0;
5. copy the byte into that data block;
6. set inode size to 12 KiB + 1.

On rev5, `FSSIZE=2000` is smaller than `BPB=8192`, so both allocations necessarily dirty the filesystem's single bitmap block; the remaining unique blocks are the indirect block, data block, and inode block. A larger image using the same bitmap format could place the two allocations in different `BPB` ranges and dirty two bitmap blocks. Rewriting the same buffer twice should not consume two logical log slots in a log that absorbs duplicate block updates.

Ordering these writes directly to home locations is unsafe: publishing the inode pointer before initializing its target exposes stale data; publishing a data allocation but never linking it leaks a block; growing size without a block pointer exposes an unmapped logical range. A transaction or copy-on-write publication mechanism must constrain the crash-visible states.

Transfer checkpoint: construct a killing test

A faulty allocator sets the bitmap bit and returns the block but does not zero it. A prior file filled that block with `SECRET`, then was removed. Design a deterministic test that distinguishes the faulty allocator from a correct one without assuming which numeric block will be reused.

Fill the filesystem sufficiently to control the free set, create a file whose allocated blocks contain a repeated nonzero pattern, delete it, then allocate a new file into the now-constrained free space. Extend the new file in a way that makes newly allocated bytes readable without overwriting the full block—if the filesystem supports sparse extension—or instrument the teaching kernel immediately after allocation and before caller writes. Assert that every exposed byte in the new allocation is zero.

On compact xv6, ordinary writes copy caller bytes into the written range, and its interface does not provide a rich sparse-file operation. A pure user-space test may therefore be unable to observe unwritten bytes deterministically. The honest solution is a kernel test hook or a scoped sparse-extension feature, not a claim that one lucky block reuse proves zeroing. The test must also restore or rebuild the image so free-space history is controlled.

REMARK

Although production filesystems replace xv6's fixed direct/indirect block array and linear directory scans with extent trees (e.g., ext4, XFS) and B-tree/HTree directories, they must preserve the exact same structural invariants between namespace edges, link counts, open-inode references, and allocation bitmaps.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 6.2.1: Calculating Inode and Bitmap Block Locations

Verification application

A filesystem layout defines:

- Block size: $B = 1024$ bytes.
- On-disk inode size: 64 bytes.
- Superblock specifies: `inodestart = 32`, `bmapstart = 48`.

1. Calculate the number of inodes per block (IPB) and the number of bits tracked per bitmap block (BPB).
2. Determine the exact disk block number and intra-block slot index for inode 45.
3. Determine the exact bitmap disk block number and intra-block bit offset for data block 18,432.

Use integer division and modulo with IPB and BPB.

1. **Parameters:**
   - Inodes per block: $\text{IPB} = \frac{1024}{64} = 16$.
   - Bits per bitmap block: $\text{BPB} = 8 \times 1024 = 8192 \text{ bits}$.
2. **Inode 45 location:**
   - Inode block number:

$$\text{block} = \text{inodestart} + \left\lfloor \frac{45}{16} \right\rfloor = 32 + 2 = 34.$$

   - Inode slot index within block 34:

$$\text{slot} = 45 \bmod 16 = 13.$$

3. **Data block 18,432 bitmap location:**
   - Bitmap block number:

$$\text{block} = \text{bmapstart} + \left\lfloor \frac{18432}{8192} \right\rfloor = 48 + 2 = 50.$$

   - Bit offset within block 50:

$$\text{bit} = 18432 \bmod 8192 = 2048.$$

Exercise 6.2.2: Maximum Representable File Size Under Direct and Indirect Pointers

Construction application

An on-disk inode structure contains 12 direct block pointers and 1 singly-indirect block pointer. Block size is $B = 1024$ bytes, and disk block addresses are 32-bit (4 bytes) unsigned integers.

1. Calculate the maximum number of data blocks that can be addressed by a single file.
2. Calculate the maximum file size in KiB and bytes.
3. If an engineer adds 1 doubly-indirect block pointer to the inode, calculate the new maximum file size.

One indirect block contains B / 4 pointers = 256 block addresses.

1. **Maximum blocks with singly-indirect pointer:**
   - Direct pointers: 12 blocks.
   - Singly-indirect pointers: $\frac{1024 \text{ bytes}}{4 \text{ bytes/pointer}} = 256$ blocks.
   - Total blocks: $12 + 256 = 268 \text{ blocks}$.
2. **Maximum file size:**

$$\text{Size} = 268 \times 1024 \text{ bytes} = 274{,}432 \text{ bytes} = 268 \text{ KiB}.$$

3. **With doubly-indirect pointer added:**
   - Doubly-indirect pointer addresses $256 \times 256 = 65{,}536$ data blocks.
   - Total blocks = $12 + 256 + 65{,}536 = 65{,}804 \text{ blocks}$.
   - New maximum file size:

$$65{,}804 \times 1024 \text{ bytes} = 67{,}383{,}296 \text{ bytes} \approx 64.26 \text{ MiB}.$$

Exercise 6.2.3: Tracing Logical File Offsets to Physical Disk Blocks via bmap

Construction transfer

A process writes to file offset byte 15,000 on an xv6 filesystem with 1024-byte blocks. The inode contains direct pointers $[500, 501, \ldots, 511]$ and indirect block pointer 700. Indirect block 700 contains entries $[800, 801, 802, \ldots]$.

1. Determine the logical block index and intra-block byte offset corresponding to byte offset 15,000.
2. State whether this logical block is direct or indirect, and identify its entry index.
3. Compute the physical disk block number where this byte will be written.

Divide byte offset by 1024. Direct blocks cover indices 0 to 11.

1. **Logical block and offset:**
   - Logical block index:

$$bn = \left\lfloor \frac{15000}{1024} \right\rfloor = 14.$$

   - In-block byte offset:

$$\text{offset} = 15000 \bmod 1024 = 15000 - (14 \times 1024) = 15000 - 14336 =$$

2. **Direct vs Indirect lookup:**
   - Indices $0 \ldots 11$ (12 blocks) are direct.
   - Since $bn = 14 \geq 12$, the block resides in the indirect range.
   - Indirect entry index: $14 - 12 = 2$.

3. **Physical block translation:**
   - The kernel fetches indirect block 700.
   - It reads entry at index 2, which contains physical block pointer 802.
   - The byte is written to physical block 802 at byte offset 664.

Exercise 6.2.4: Proving Buffer Cache Uniqueness and Preventing Split-Brain Aliasing

Proof synthesis

Prove that the operating system buffer cache must enforce the invariant:

$$\forall (dev, blk), \quad \text{count}(\text{resident buffers matching } (dev, blk)) \leq 1.$$

Show how allowing two concurrent in-memory `struct buf` instances for the same disk block causes split-brain data loss.

Show that two threads modifying different fields in separate buffer copies overwrite each other on writeback.

Assume for contradiction that two distinct in-memory buffer instances B_1 and B_2 exist simultaneously in the buffer cache for the identical disk block (dev, blk):

1. Suppose Thread 1 acquires B_1 to update directory entry D_1 within block (dev, blk) and writes its changes into B_1's memory.
2. Concurrently, Thread 2 acquires B_2 to update directory entry D_2 in the same block (dev, blk) and writes its changes into B_2's memory.
3. Thread 1 commits its modification: the buffer cache writes B_1's 1024 bytes to disk. The disk block now contains updated D_1 but stale D_2.
4. Subsequently, Thread 2 commits its modification: the buffer cache writes B_2's 1024 bytes to disk.
5. Because B_2 was modified independently without observing B_1's changes, B_2's writeback completely overwrites the physical disk block, silently destroying Thread 1's update to D_1.

Therefore, the buffer cache must maintain a global lock across lookup and allocation (`bcache.lock`) ensuring that any request for (dev, blk) returns a pointer to the single existing in-memory buffer, incrementing its reference count rather than instantiating a duplicate.

Exercise 6.2.5: Data Block Zeroing and Elimination of Residual Data Leaks

Description application

Explain why the block allocator `balloc()` must synchronously zero newly allocated disk blocks before attaching them to an inode block map. Describe the security vulnerability that arises if a newly allocated block is published to user space with uninitialized contents.

Consider prior file deletions leaving unencrypted sensitive data on deallocated blocks.

- **The confidentiality vulnerability:** When a file is deleted, the filesystem marks its blocks free in the bitmap (`bfree()`) but does not immediately wipe the physical bytes on magnetic media or flash chips to avoid disk write latency. As a result, freed blocks retain residual data (e.g., plaintext passwords, private cryptographic keys, user emails, or database records). If a newly created file allocates a block and attaches it to its inode without zeroing:
  1. An unprivileged user can create a large file, immediately extend its length or read its contents without writing, and read the raw residual bytes left behind by prior processes or administrative files.
  2. This constitutes an information disclosure vulnerability (residual data leakage across security domains).
- **Enforcement:** `balloc()` acquires a buffer for the allocated block, executes `memset(bp->data, 0, BSIZE)`, dirties the buffer in the active log transaction, and commits it. This guarantees that user-mode readers observe only zeroes until authorized writes occur.

References & Further Reading

- MIT PDOS, xv6 RISC-V source and book — official entry point for the licensed teaching kernel and text.
- MIT PDOS, xv6-rev5 `kernel/fs.h` — immutable on-disk format and mapping constants used by the concrete model.

- MIT PDOS, xv6-rev5 `kernel/fs.c` — immutable block allocation, inode cache, block mapping, directories, and pathname traversal source.
- MIT PDOS, xv6-rev5 `kernel/sysfile.c` — immutable filesystem system-call validation and transaction boundaries.
- MIT PDOS, xv6-rev5 `kernel/param.h` — immutable `FSSIZE`, per-operation log reservation, and cache-capacity constants used to bound rev5-specific traces.
- Linux kernel documentation, Overview of the Virtual File System — production VFS object and operation boundaries.
- Linux kernel documentation, ext4 data structures — a production-format contrast for superblocks, block groups, inodes, extents, and directories.

SECTION DETAIL

Crash Consistency and Journaling

A file grows from one block to two. The kernel must initialize a new data block, mark it allocated, and publish its address and the new size in the inode. Power fails after the inode reaches stable storage but before the data and bitmap. After reboot, the inode exposes a block the allocator considers free; another file can receive the same block, and the first read may disclose stale bytes.

Locks cannot repair this failure. They constrain live interleavings, but a crash erases lock state and volatile memory. Recovery correctness requires a protocol over **persistence order**: which subsets of writes may survive, which subset becomes authoritative, and how reboot code distinguishes complete from incomplete updates.

The crash model is part of the theorem

DEFINITION

Under the bounded atomic-sector crash model, a storage state transition from pre-crash write set $W_{\mathrm{d}}^{\mathrm{issue}}$ to post-crash durable state $S_{\mathrm{h}}^{\mathrm{cras}}$ permits any subset of unbarried writes to be present or absent in arbitrary order, subject to two axioms: (1) every write preceding an acknowledged barrier is present, and (2) each individual sector holds either its complete old value or its complete new value.

Use a bounded block-device model unless a problem states otherwise:

- a block write may be delayed or reordered before stable acknowledgement;
- after an acknowledged flush or equivalent barrier, preceding writes in the specified domain are stable;
- a sector write is atomic: recovery observes its old or new value, not a torn mixture;
- volatile kernel and controller state disappears at the crash;
- the device does not silently lose a write it acknowledged as stable; and
- recovery itself may crash and run again.

These are assumptions, not universal hardware facts. A controller without protected write-back power, a filesystem mounted with weaker barriers, or a device that violates flush semantics invalidates the proof. Checksums and duplicated headers can detect some torn writes; they cannot manufacture durability from a device that lies about persistence.

Define four properties separately:

property	question
runtime atomicity	can another live operation observe an intermediate name-space state?
crash atomicity	after recovery, is the update entirely absent or entirely present?
durability	after the documented success/barrier point, must the up-date survive the modeled crash?
recovery idempotence	does running recovery twice produce the same authorita-tive state as running it once?

A filesystem may provide one without all four. An atomic `rename` constrains concurrent name visibility; it does not, by itself, state that the renamed data or directory entry is durable.

CRASH CONSISTENCY, WRITE ORDERING, AND JOURNALING INVARIANTS

Evaluate each statement regarding filesystem crash consistency, journaling, and recovery idempotence:

Answer: Write-ahead logging mandates that transaction log blocks must be durable on media before the commit record is written.; Physical block redo logging is strictly idempotent because installing full-block images repeatedly yields identical disk state.

Write-ahead logging requires all logged blocks to be durable on disk before writing the commit record. Physical block replay is idempotent because writing target block images multiple times produces the identical outcome. Rename provides only namespace atomicity without disk durability, and writing to home locations prior to log commit destroys crash recovery guarantees.

Derive invariants before ordering writes

Model one append with Boolean state (D, B, I):

- D: new block contains the intended bytes;
- B: allocation bitmap marks the block owned;
- I: inode size and block map expose the block.

The minimum safety invariants are

$$I \Rightarrow B \qquad \text{and} \qquad I \Rightarrow D.$$

$B \wedge \neg I$ is unreachable allocated space: a leak, but not cross-file aliasing. $D \wedge \neg B \wedge \neg I$ is an unreachable overwrite of a free block and can destroy data if the allocator had concurrently assigned it; the single-operation model assumes the allocator reserved the block in volatile state while the operation ran. $I \wedge \neg B$ is dangerous because the same block can be reallocated. $I \wedge \neg D$ exposes stale or partial content.

Worked crash trace: metadata-first append

Suppose home-location writes reach stable storage in this order: inode, bitmap, data. Crashes produce:

crash cut	durable tuple	diagnosis
before any write	`(0,0,0)`	complete pre-state
after inode	`(0,0,1)`	dangling reference and stale-data exposure
after bitmap	`(0,1,1)`	inode exposes uninitialized content
after data	`(1,1,1)`	complete post-state

Reversing the writes to data, bitmap, inode avoids an exposed uninitialized block in this one append:

crash cut	durable tuple	diagnosis
after data	`(1,0,0)`	namespace remains old; new bytes unreachable
after bitmap	`(1,1,0)`	leaked allocation
after inode	`(1,1,1)`	complete post-state

Ordering reduces damage but does not give crash atomicity: the leak is neither the complete pre-state nor post-state. More complex operations have cyclic dependencies. Replacing a file can require the new directory edge before removing the old to avoid loss, while link counts and replacement targets require another order. A general mechanism needs a commit point separate from home locations.

Application checkpoint: derive, do not memorize

A directory insertion requires a new directory data block. The operation changes bitmap bit B, directory-inode pointer P, directory size S, and entry bytes E. State necessary implications for a visible new name, then propose a data-before-metadata order and identify its remaining non-atomic state.

If lookup can reach the new entry, then the directory's block pointer and size must expose the block, the block must be allocated, and the entry bytes must be initialized. One compact set is

$$(P \wedge S) \Rightarrow B, \qquad (P \wedge S) \Rightarrow E.$$

A conservative home-write order is initialize E, persist B, then publish P and S together if they occupy one atomically written inode block. Crashing after B but before publication leaks the block. If P and S can tear or occupy different atomic units, more intermediate states exist. The correct diagnosis is "ordered but not atomic," not "safe because metadata was last."

Prediction before the recovery model

The journal contains a complete redo image for every block in transaction 9. Its commit record is stable. None of transaction 9's home-location writes is stable when power fails. Predict the recovered state and name the ordering rule your answer relies on.

PULL THE POWER DURING A FILE-SYSTEM UPDATE

ON DISK WHEN POWER FAILS	VISIBLE AFTER RECOVERY
DATA BLOCK 2 `old / absent`	DATA BLOCK 2 `B written`
ALLOCATION BITMAP `free`	ALLOCATION BITMAP `allocated`
INODE / ACTIVE ROOT `old file only`	INODE / ACTIVE ROOT `references block 2`

Recovery result: Committed journal transaction is replayed: post-state is restored.

Static state: power fails after a durable journal commit but before home-location writes. Recovery replays the committed transaction and restores the complete post-state.

The modeled answer is the complete post-state: recovery recognizes the durable commit and replays every logged block to its home location. The conclusion depends on all transaction records becoming stable before the commit record and on the commit record being atomically recognizable. Move the crash point to one step before commit; the same records are ignored because no authoritative commit exists.

Transfer after changing one assumption

Replace the atomic-sector assumption with “a commit sector may tear into a mixture of old and new bytes.” What additional evidence must recovery require before replay?

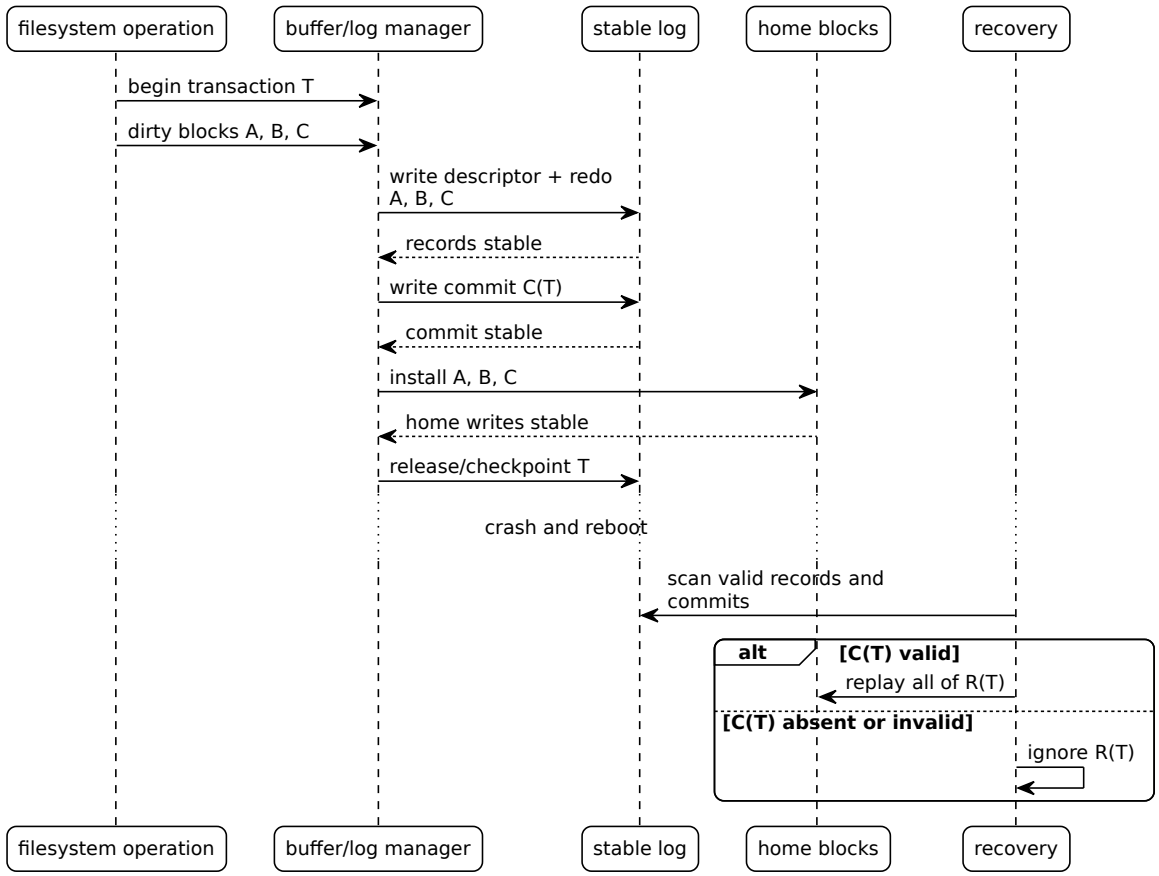
Recovery needs a commit representation whose validity can be distinguished from every relevant torn state—for example a checksummed, versioned record with redundancy or an atomic write unit guaranteed by the actual device. The checksum domain must cover transaction identity and the record set it commits; otherwise a valid old commit could be paired with unrelated new records. Recovery must reject an invalid or incomplete commit and preserve the pre-state.

A checksum makes corruption detectable with a quantified residual collision risk; it does not force preceding records to stable storage. Ordering/barrier semantics and commit integrity solve different obligations.

Write-ahead logging from three obligations

A physical redo log stores intended new block images in a reserved region before installing them at home locations. Let $R(T)$ be all records for transaction T, and $C(T)$ its commit record. The protocol follows from three obligations:

1. **Write ahead:** every member of $R(T)$ is stable before $C(T)$ may become stable.
2. **Publish once:** $C(T)$ is the sole recovery-visible decision that T is committed.
3. **Do not overwrite evidence early:** log space for T is reusable only after its committed records are installed at home and that checkpoint state is stable.



The home installation may write A, B, and C in any order after commit because the log retains a complete post-state. If power fails midway, recovery replays all three. Replaying a physical block image twice is idempotent: writing the same final bytes again does not change the result.

> ***Recovery proof obligation.*** *For every crash cut, recovery returns a state observationally equivalent to the pre-state of every uncommitted transaction and the post-state of every committed transaction, while preserving filesystem invariants.*

This statement requires a transaction-serialization model. If two committed transactions update the same block, recovery must respect their log order or replay the latest committed image. If a log record is reused before all earlier home writes are stable, the only repair evidence may be destroyed.

Worked trace: atomic name replacement

Transaction 12 replaces directory entry `config -> inode 40` with `config -> inode 71` and decrements inode 40's link count from 1 to 0. Its physical log contains new images for directory block D and inode block I.

durable log/home state at crash	recovery action	visible result
D record stable, I record absent, no commit	ignore transaction 12	old name and old link count
both records stable, no commit	ignore transaction 12	old name and old link count
both records and commit stable, no home writes	replay D and I	new name; old inode unlinked
commit stable, only home D installed	replay both	new name; old inode unlinked
commit stable, both home blocks installed, log not cleared	replay both harmlessly	same complete post-state

Replaying only the missing-looking inode block would require recovery to infer which home bytes are current, a harder protocol. Whole-transaction redo keeps the decision local: valid commit means install every record.

Proof checkpoint: idempotence is not optional

Recovery replays transaction 12's directory block, then power fails before it replays the inode block. Explain why restarting recovery must replay the directory block again and why an increment-style log record would be dangerous.

The commit is still present, so the restarted scan cannot assume that any subset reached home. A physical redo record says "set block D to these bytes"; applying it twice yields the same D. An operation record saying "decrement inode 40's link count" is not idempotent: replaying it twice decrements twice unless the protocol carries transaction identity and per-object application state. Logical logging can be correct, but it needs a stronger replay design than blind repetition.

Concurrency changes transaction admission

Compact xv6 brackets filesystem system calls with `begin_op` and `end_op`. It reserves enough log capacity for worst-case operations before admitting more work. Modified buffers are registered with `log_write`; repeated modifications to one block are absorbed into one transaction entry. A commit occurs only when no filesystem operations remain active, preventing a commit from installing a block that still contains another operation's uncommitted update.

This design is intentionally restrictive:

- several system calls may share one log transaction;
- long operations must fit the bounded log reservation model;
- one outstanding commit simplifies ordering but limits throughput; and
- a syscall joining other active operations may return before their shared transaction commits, depending on the exact implementation path and completion timing.

Therefore a user-visible "write returned" claim must not be silently upgraded to "power-safe." The documented synchronization interface and the implementation's actual commit semantics determine that boundary.

Application checkpoint: reservation arithmetic

A log has 30 data slots. Each admitted operation reserves a worst-case 10 slots, although duplicate-block absorption often makes actual use smaller. Why may `begin_op` block the fourth concurrent operation even when the first three currently occupy only six unique slots?

Admission must remain safe if every active operation reaches its worst case after the check. Three operations can require all 30 slots. Admitting a fourth based on current use can deadlock or overflow later, when none can surrender partial work without violating atomicity. Conservative reservation trades concurrency for a simple capacity proof. A more concurrent design needs finer-grained preclaims, multiple transactions, abort support, or dynamically extensible log space; observed average use is not a proof.

Production contrast: ext4 modes are different contracts

> **EXAMPLE**
>
> When a process appends 4 KiB to a file and the system crashes immediately after the JBD2 metadata transaction commits: under `data=writeback`, the updated inode size is replayed even if the new data block never reached disk, exposing uninitialized or stale disk contents to post-reboot readers; under `data=ordered`, JBD2 forces the dirty data block to stable storage before writing the metadata commit record, preventing stale-block disclosure without journaling the data bytes twice.

Linux ext4 uses JBD2 to journal metadata updates. The journal contains descriptors, logged blocks, and a commit record; replay applies transactions with valid commits. The documented data modes differ:

mode	journaled content	relevant promise boundary
`data=journal`	file data and metadata	strongest of these modes, with additional write cost
`data=ordered`	metadata; associated new data is ordered before metadata commit	default documented mode, avoiding some stale-data exposure
`data=writeback`	metadata; data need not precede metadata	weaker post-crash file-content expectations

The table is not a complete ext4 guarantee. Delayed allocation, direct I/O, `fsync`, directory persistence, device flushes, fast commits, checksums, and kernel versions affect a real claim. A teaching log and JBD2 share write-ahead reasoning; their scale, concurrency, record formats, and user-visible contracts are not interchangeable.

Failure boundaries that invalidate easy proofs

- **Torn sectors:** an apparently present header may be half old and half new.
- **Reordered caches:** issuing records before commit is insufficient unless stable order is enforced.
- **Lost acknowledgement:** a device claiming a flush completed but losing it violates the model.

- **Log checksum omission:** recovery may replay corrupted or mismatched records.
- **Log reuse before checkpoint:** a second transaction can overwrite the only redo image for the first.
- **Recovery side effects:** a non-idempotent replay operation can corrupt more on each boot.
- **Unbounded transaction:** a rename or allocation path that exceeds reserved log capacity cannot be made atomic by wishing it fit.
- **Interface confusion:** runtime atomicity and a successful return are not equivalent to post-crash durability.

REMARK

Physical redo logging in xv6 establishes the core write-ahead invariant—log records stable before commit, commit stable before home installation, and home installation stable before log reuse—while leaving physiological redo/undo logging (ARIES), copy-on-write tree publication, and distributed consensus to advanced storage and database courses.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 6.3.1: The Three Write-Ahead Logging Ordering Rules

Verification application

State the three fundamental write-ordering rules governing physical block redo journaling. For each rule, state the catastrophic crash failure mode that occurs if that rule is violated.

Rule 1: Log before Commit. Rule 2: Commit before Home. Rule 3: Home before Log Reuse.

The three write-ahead logging (WAL) invariants are:

1. **Rule 1 (Log records before Commit):** All transaction data and metadata blocks written into the journal must reach stable storage before the commit record is written.
   - *Failure if violated:* If the commit record arrives on disk first and power fails while journal blocks are still inflight, recovery will observe a "committed" transaction header and replay garbage or unwritten bytes into home filesystem structures, corrupting the filesystem.

2. **Rule 2 (Commit before Home installation):** The commit record must reach stable storage before any modified block is written back to its permanent home location.
   - *Failure if violated:* If a block is written directly to its home location before commit and power fails, home data is modified but the transaction is uncommitted. Recovery discards uncommitted transactions, leaving home structures permanently inconsistent or partially updated.

3. **Rule 3 (Home installation before Log reuse / checkpoint):** All home-location writes for a transaction must be completely flushed to disk before the log header is updated to free that transaction's journal slots.
   - *Failure if violated:* If log space is reclaimed while home blocks remain buffered in volatile RAM and a crash occurs, the journal record is gone and home blocks never reached disk. Updates are silently and permanently lost.

Exercise 6.3.2: Crash Cut State Enumeration and Consistency Analysis

Construction application

A filesystem appends a block to an existing file by updating three distinct structures:

- D: newly written data block.
- B: block allocation bitmap.
- I: inode size and pointer.

Suppose a naive filesystem writes these blocks directly to home disk locations in the order $I \to B \to D$ without journaling.

1. Enumerate the durable disk state $(D, B, I) \in \{0, 1\}^3$ across all possible crash points.
2. For each state, determine whether filesystem consistency and user data security are preserved or violated.

Check invariants: I implies B (no free block aliasing) and I implies D (no stale data exposure).

The sequence of writes is: (1) Inode I, (2) Bitmap B, (3) Data D.

- **Cut 0 (Crash before I is written):** Durable state $(D, B, I) = (0, 0, 0)$. *Evaluation:* Consistent. The file retains its original size; no blocks are leaked or corrupted.
- **Cut 1 (Crash after I, before B):** Durable state $(D, B, I) = (0, 0, 1)$. *Evaluation:* **Critical violation.** The inode claims the block, but the bitmap marks it free ($I \wedge \neg B$). A subsequent file creation will allocate the identical block, causing cross-file aliasing and corruption. Furthermore, reading the block exposes uninitialized garbage ($I \wedge \neg D$).
- **Cut 2 (Crash after B, before D):** Durable state $(D, B, I) = (0, 1, 1)$. *Evaluation:* **Security violation.** The block is marked allocated in the bitmap and mapped in the inode, but data write D never reached disk. Reading the block exposes stale disk contents left behind by prior deleted files.
- **Cut 3 (Crash after D):** Durable state $(D, B, I) = (1, 1, 1)$. *Evaluation:* Consistent. All metadata and data updates completed successfully.

Exercise 6.3.3: Proving Idempotence in Physical Redo Journaling

Proof synthesis

Let R denote a physical redo recovery procedure that reads a valid committed log transaction containing blocks $\{(b_1, \text{data}_1), \ldots, (b_k, \text{data}_k)\}$ and writes each data_i into disk block b_i.

1. Prove that $R \circ R = R$ (idempotence).
2. Explain why a logical log record of the form "increment link count of inode 12" is NOT idempotent, and show how a crash during recovery causes catastrophic corruption under logical logging.

Physical logging writes absolute block contents; logical logging applies relative operations.

1. **Proof of physical redo idempotence:** Let S denote the disk state. Physical redo logging defines the state transition:

   $$R(S)[b_i] = \text{data}_i \quad \text{for all } i \in \{1, \ldots, k\}.$$

   Applying the recovery procedure a second time ($R(R(S))$):

   $$(R \circ R)(S)[b_i] = \text{data}_i = R(S)[b_i].$$

   For all other blocks $b \notin \{b_1, \ldots, b_k\}$, R does not alter $S[b]$. Because setting fixed byte contents is an idempotent assignment ($x \leftarrow c$), executing recovery n times produces the identical disk state as executing it once:

   $$R^n = R \quad \text{for all } n \geq 1.$$

2. **Failure of logical logging without version numbers:** If the log records a relative transition $\text{link_count}(12) \leftarrow \text{link_count}(12) + 1$:
   - Recovery starts: executes link increment, writing $\text{link_count}(12) = 2$.
   - Power fails before recovery completes the transaction checkpoint.
   - Upon reboot, recovery starts again from the beginning of the log. It executes the link increment again, producing $\text{link_count}(12) = 3$.
   - The file now has a phantom link count, preventing it from ever being deallocated when unlinked and corrupting filesystem reference integrity.

Exercise 6.3.4: Contrasting ext4 Journal, Ordered, and Writeback Data Modes

Description application

Contrast the write ordering and durability guarantees provided by the three ext4 JBD2 journaling modes: `data=journal`, `data=ordered`, and `data=writeback`. Explain why `data=ordered` represents the default industry choice.

Consider performance cost of writing data twice versus stale data disclosure risks.

- `data=journal`: Both file payload data and filesystem metadata are written to the JBD2 journal before being flushed to home locations.
  - *Guarantee:* Full crash atomicity for data and metadata. No stale data exposure; data modifications never roll back if metadata commits.
  - *Drawback:* Severe performance penalty (every byte of file data is written to disk twice: once to the log, then to home).
- `data=ordered` **(Default):** Only metadata is written to the journal. However, the kernel enforces a strict write-ordering barrier: dirty file data blocks must be flushed to their permanent home locations before the corresponding metadata transaction commit record is written to the journal.
  - *Guarantee:* Prevents stale data leakage. If the system crashes, recovery never replays an inode pointing to unwritten blocks. File data is written only once, providing excellent performance while preserving data integrity.
- `data=writeback`: Only metadata is journaled. Data blocks are written back asynchronously whenever the OS page cache flushes, with no ordering dependency with respect to the metadata commit record.
  - *Guarantee:* Highest throughput, but vulnerable to data disclosure. A crash after metadata commit but before data flush leaves inodes pointing to unwritten blocks containing previous file data.

Exercise 6.3.5: Log Reservation Sizing and Concurrency Control

Construction transfer

An xv6 filesystem log has a fixed capacity of $L = 30$ disk blocks. Each system call must reserve its maximum worst-case block consumption M_{p} in `begin_op()` before modifying any buffers.

1. If a file append can touch at most 10 unique blocks (bitmap, indirect block, data block, inode, etc.), compute the maximum number of concurrent operations that `begin_op()` can safely admit.

2. Explain why admitting a 4th operation when the first 3 operations have currently dirtied only 2 blocks each would risk deadlock.

A thread holding an active transaction cannot abort or surrender partial work on log overflow.

1. **Admissible operations:** To guarantee that the log never overflows even if every admitted operation simultaneously consumes its worst-case reservation:

$$N_{\max} = \left\lfloor \frac{L}{M_{\text{op}}} \right\rfloor = \left\lfloor \frac{30}{10} \right\rfloor = 3 \text{ operations.}$$

2. **Deadlock hazard of optimistic admission:** Suppose the kernel optimistically admits a 4th operation because the first 3 operations have currently modified only $3 \times 2 = 6$ unique blocks:
   - All 4 operations continue executing concurrently.
   - Workload conditions demand that each of the 4 operations now allocates additional blocks, requiring their full 10 blocks: total demand = $4 \times 10 = 40$ blocks.
   - The log fills completely to 30 blocks.
   - All 4 operations stall waiting for additional log space to become free in `log_write()`.
   - However, in xv6, a log transaction cannot commit until all active operations call `end_op()`.
   - None of the 4 operations can finish without more log space, and the log cannot commit and free space until the operations finish. The system enters an unrecoverable deadlock. Conservative reservation strictly prevents this state.

References & Further Reading

- MIT PDOS, xv6 RISC-V source and book — official teaching-kernel source and Chapter 10 filesystem/log explanation.
- MIT PDOS, xv6-rev5 `kernel/log.c` — immutable transaction admission, log absorption, commit, install, and recovery implementation.
- MIT PDOS, xv6-rev5 `kernel/fs.c` — immutable metadata-update paths protected by the log.
- Linux kernel documentation, ext4 Journal (JBD2) — transaction format, commits, replay, checkpoints, and data modes.
- The Open Group, POSIX.1-2024 `fsync()` — application-visible synchronization interface.
- The Open Group, POSIX.1-2024 `rename()` — runtime namespace atomicity that must not be confused with persistence.

SECTION DETAIL

Filesystem Lab: Recoverable Rename

The naive implementation is four lines: locate `old`, add `new`, remove `old`, return success. It loses data when target insertion runs out of space, leaks the replaced inode when its link count is not decremented, deadlocks when opposite cross-directory renames acquire parents in argument order, and exposes a mixed namespace after a crash when either directory write bypasses the log.

The project is complete only when tests distinguish those implementations from a correct one. A successful happy-path shell transcript is not evidence for atomicity or recovery.

This brief is original to LibreUni. It uses official xv6 RISC-V tag `xv6-riscv-rev5`, commit `7d7adbb1b0acbd67c9766a20d0f9900fef2789fa`, as the licensed course root and does not reproduce another institution's lab text or private solution.

Freeze the experimental object

Every implementation builds from the same immutable **course root**:

```
repository:  https://github.com/mit-pdos/xv6-riscv
tag:         xv6-riscv-rev5
course root: 7d7adbb1b0acbd67c9766a20d0f9900fef2789fa
```

Choose and declare one start mode:

- **clean baseline:** branch directly from the course root and implement only the rename extension;
- **cumulative checkpoint stack:** apply the learner's accepted course checkpoint patches in the documented course order on top of the same root, record every patch identifier and checksum, then branch the rename work from that stack tip.

The cumulative stack does not create a second upstream baseline. It creates a reproducible pre-lab start point whose ancestry must include the exact course root. The lab-only treatment is the diff from that recorded start point. This separation permits a clean implementation for isolation or an integrated cumulative kernel without attributing earlier checkpoint effects to rename.

Preserve upstream license and notices. Record, in `REPORT.md`:

```
course-root URL:
course-root tag and commit:
start mode (`clean-rev5` or `cumulative-stack`):
ordered checkpoint patch IDs and checksums (`none` for clean mode):
pre-lab start-point commit or tree checksum:
rename treatment commit or patch checksum:
toolchain version:
QEMU version:
host architecture and OS:
build command:
upstream and checkpoint regression commands:
rename test command:
```

Before modifying code, save evidence for:

1. the tag resolving to the exact course-root hash;
2. a clean course-root build and its upstream test result;
3. a clean start-point boot plus upstream and applicable checkpoint regression results;
4. `usertests` or the start point's equivalent result; and
5. a filesystem image hash after a fresh start-point build.

If either the course root or recorded start point does not build, implementation stops. Changing implementation, checkpoint stack, and toolchain simultaneously destroys the comparison.

Baseline checkpoint

Explain why recording only `xv6-riscv-rev5` is insufficient for a cumulative implementation even though the tag itself is immutable.

The tag and hash fix only the course root. A cumulative start point also contains ordered checkpoint patches whose contents, order, conflict resolutions, and configuration can change behavior. Record their immutable identifiers/checksums and the resulting pre-lab tree, then express rename as a lab-only diff from that tree. Toolchain and QEMU versions fix major translation and emulation variables; commands make the build path inspectable. A container image without its source, digest, and build instructions merely packages the ambiguity.

Rev5 compatibility audit

The tag was chosen because the filesystem constants and paths used by the course are concrete: `BSIZE=1024`, `NDIRECT=12`, `DIRSIZ=14`, `MAXOPBLOCKS=10`, and `LOGBLOCKS=30`; `begin_op`, `end_op`, `log_write`, `dirlookup`, `dirlink`, `writei`, and recovery are present. Four integration deltas must remain visible:

rev5 boundary	consequence for the lab
no `rename` syscall or user ABI exists	adding the bounded syscall path is required work, not a missing setup patch
`skipelem` accepts at least 14 bytes by retaining only `DIRSIZ` bytes	the lab's overlength-final-component rejection requires validation from the untruncated path, before aliasing can occur
the filesystem has no general transaction abort; `bmap` / `writei` can allocate before a later ENOSPC result	all expected failures must precede publication, or a proved rollback/reservation mechanism must restore allocation and metadata exactly
log admission reserves `MAXOPBLOCKS=10` unique blocks per operation	the worst-case rename dirty set must be proved `<= 10`; exceeding it requires an explicitly reviewed narrow log-capacity change, not a silent constant edit

Cumulative checkpoint patches may change a touched syscall table, inode path, lock order, log bound, disk format, or test ABI. Treat any failed patch application, changed invariant, expanded dirty set, or altered source line as an integration incompatibility: preserve the conflict, record it in `REPORT.md`, adapt the lab-only patch deliberately, and rerun every affected checkpoint test. Do not resolve a conflict by copying a clean-tree file over cumulative work.

LOG CAPACITY AND UNIQUE DIRTY BLOCK BUDGETING

Evaluate each statement regarding log capacity, buffer absorption, and rename invariants in xv6:

Answer: Multiple log_write calls modifying the identical disk block consume only a single physical log slot.; Acquiring two parent directory locks in ascending inum order mathematically prevents cross-directory deadlock.

xv6 log_write absorbs repeated writes to the same block into a single transaction slot, so the budget depends on unique dirty blocks rather than call counts. Sorting parent inode locks by inum eliminates circular wait deadlocks, and writing directly to home blocks before commit violates write-ahead logging.

Required interface and bounded semantics

Add a user-visible call with this contract:

```
int rename(const char *old_path, const char *new_path);
```

It returns `0` on success and `-1` on failure. The required domain is narrower than POSIX and must be documented as such:

- `old_path` resolves to an existing regular file;
- `new_path` has an existing directory parent on the same xv6 filesystem;
- source and target parents may differ;
- the target may be absent or may name a regular file;
- replacing an existing target removes one link from the displaced target;
- if both names already identify the same inode, the operation succeeds without changing either link count;
- final components `.` and `..`, directory sources, directory targets, overlength final components, and invalid paths fail without namespace change; and

- no symbolic-link behavior is required because the pinned teaching baseline has no required symbolic-link interface for this project.

The success-state namespace is:

```
before: old -> S, new -> D or absent
after:  old absent, new -> S
```

If `D` existed and differed from `S`, its link count decreases by one. `S` keeps the same link count: one directory edge is moved, not created or destroyed. Open descriptors for `S` and `D` retain their inode references.

> ***No-change-on-error contract.*** *Every expected validation or capacity failure must leave the visible namespace and persistent link counts at their pre-call values. Kernel panic is not an accepted error path.*

Worked trace: replacement with open descriptors

Initial state:

```
/a/old -> inode 20, nlink(20)=1, process P has fd 3 open on inode 20
/b/new -> inode 44, nlink(44)=1, process Q has fd 5 open on inode 44
```

After successful `rename("/a/old", "/b/new")`:

```
/a/old absent
/b/new -> inode 20, nlink(20)=1
inode 44 has nlink=0 but remains live while Q holds fd 5
P's fd 3 and a fresh open of /b/new both reach inode 20
Q's fd 5 still reaches inode 44
```

Closing Q's descriptor permits inode 44 reclamation. Redirecting Q's open descriptor to inode 20 would violate open-file identity; leaving `nlink(44)=1` would leak persistent ownership.

Contract checkpoint: two names, one inode

`/a/old` and `/b/new` are hard links to inode 20, whose link count is 2. What must rename do?

The operation succeeds as a no-op under the stated contract. Both names remain and `nlink(20)` remains 2. Treating `new` as a displaced different object and decrementing its link count corrupts the relationship between on-disk link count and directory references. This case must be detected by `(dev, inum)` identity, not pathname-string equality.

Invariants to write before code

Your design memo must state and use these invariants:

N1 — Directory validity. Every nonzero directory entry names an allocated inode of the same filesystem.

N2 — Regular-file link accounting. After recovery with no live process state, each regular inode's `nlink` equals the number of directory entries that name it.

N3 — Move conservation. Renaming `S` from one name to another does not change `nlink(S)`.

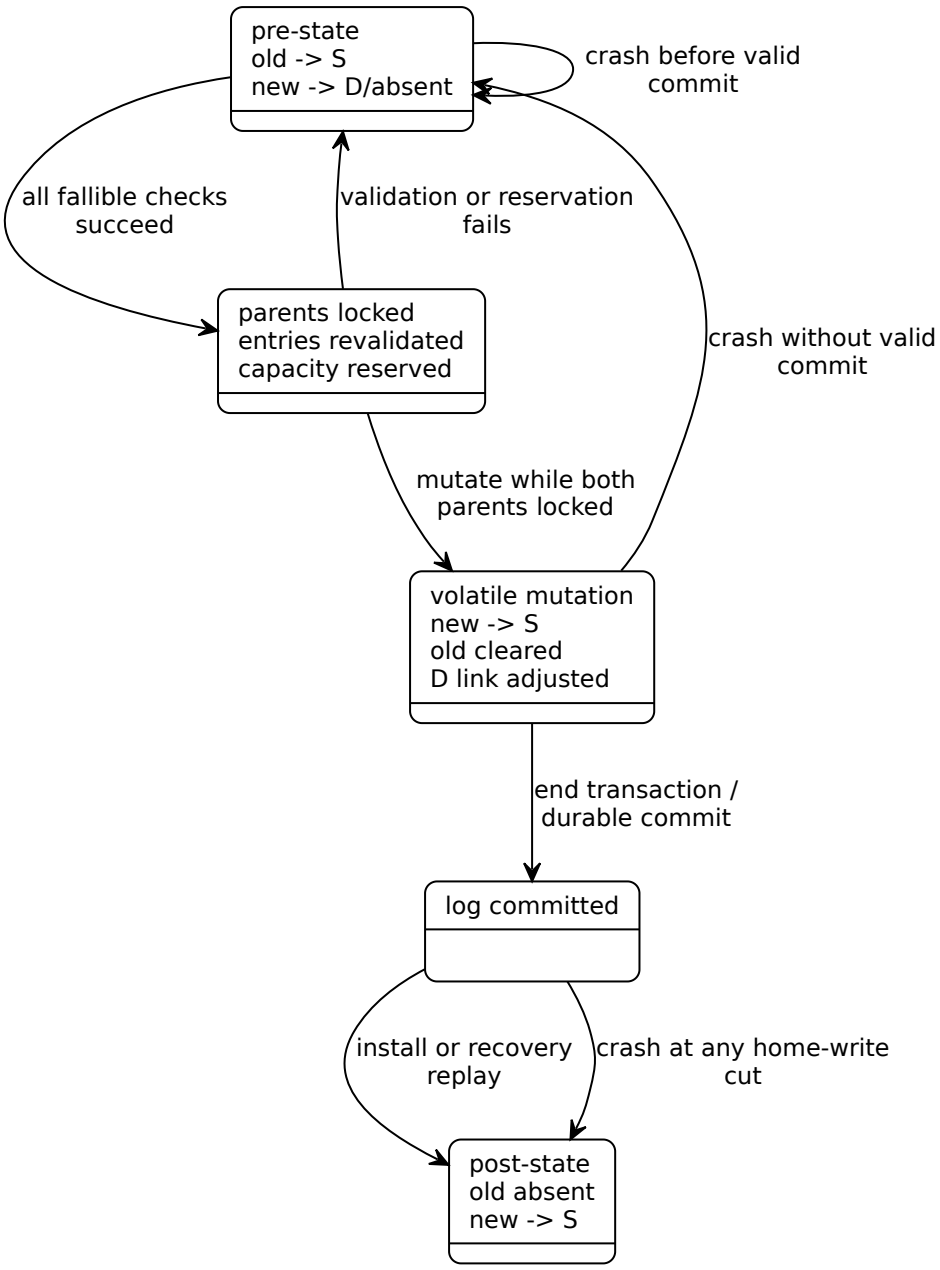
N4 — Replacement accounting. Replacing distinct `D` removes exactly one namespace edge to `D`; reclamation waits for both zero links and zero in-memory references.

N5 — Runtime namespace atomicity. A lookup beginning after the operation's linearization point observes `new -> S` and no `old`; a lookup completed before that point may observe the pre-state. The implementation does not release a modified parent lock while the other parent still exposes an intermediate state.

N6 — Crash state set. After log recovery, the pair of names and link counts is exactly the pre-state or exactly the post-state—not a mixture.

N7 — Lock-order acyclicity. Every operation that simultaneously locks two parent inodes acquires distinct parents by ascending `(dev, inum)` order and releases them in reverse order.

N8 — Log boundedness. One rename never dirties more unique blocks than the transaction reservation permits; repeated updates to one block count once.



The diagram is a proof skeleton. Your code may use different helper boundaries, but every failure and crash cut must map to one allowed edge.

Design the mutation in three phases

Do not use the following as copyable implementation code; it is a review decomposition.

Resolve and retain identities

Within one filesystem transaction:

1. copy both user path strings into bounded kernel buffers;
2. resolve and retain references to both parent inodes and final components;
3. reject unrepresentable final names, `.` and `..`;
4. acquire one or two parent inode locks in canonical `(dev, inum)` order; and
5. re-run source and destination entry lookup while those locks are held.

The second lookup is not redundant. Path traversal produced references before both parent locks were acquired; another namespace operation may have changed an entry. All decisions used for mutation must be based on state protected by the locks held during mutation.

Validate and reserve before publication

While parent locks exclude entry changes:

- obtain referenced source and optional destination inodes;
- reject non-regular types;
- detect same-inode no-op;
- verify link counts are positive and within representable range; and
- determine a writable destination entry slot.

If inserting `new` requires extending the destination directory, perform every allocation that can fail before clearing `old`. Account for the destination directory data block, its bitmap block, its inode block, source directory block, optional displaced-inode block, and any overlaps. A harmless reserved empty slot is not permission to exceed log capacity; report its treatment explicitly.

Compact xv6 does not offer general transaction abort. Once the first visible edge is changed, the remaining path should contain no expected recoverable failure. An implementation that calls an allocation routine after clearing `old` has already violated the no-change-on-error contract.

Publish under locks and log coverage

With capacity secured and identities revalidated:

1. install or replace the destination entry with source inode number;
2. clear the source entry when it is a distinct slot;
3. decrement and persist the displaced inode's link count when destination identity differs;
4. keep source link count unchanged;

5. keep both parent locks until all namespace mutations are complete; and
6. route every modified buffer through the existing log rather than writing a home block directly.

The order of steps 1 and 2 is not the crash protocol—the log supplies crash atomicity—but destination-first avoids losing the only name if an internal assertion reveals an unexpected failure. Runtime readers cannot observe the temporary duplicate because the relevant parent locks remain held.

Lock-order checkpoint

Thread A renames `/x/a` to `/y/a`; thread B simultaneously renames `/y/b` to `/x/b`. Both acquire the source parent first. Construct the wait-for cycle, then remove it using canonical order.

Argument-order acquisition permits:

```
A holds inode(x), waits for inode(y)
B holds inode(y), waits for inode(x)
```

The wait-for graph has a cycle. If both compare `(dev, inum)` and lock the lower identity first, they request the same first lock; one waits before holding the second resource, so the two-node cycle cannot form. The same-parent case locks once. A timeout only detects the bug; it does not establish acyclic acquisition.

Crash matrix, not random hope

Add deterministic failpoints around the existing log protocol. A test-only failpoint may halt QEMU or force reboot after a named event. It must be disabled in the normal production build and documented so it cannot trigger accidentally.

At minimum, inject after:

cut	durable event	expected recovered state
C0	transaction admitted, no log record stable	pre-state
C1	some rename block images stable, no valid commit	pre-state
C2	all block images stable, no valid commit	pre-state
C3	commit record stable, no home block installed	post-state
C4–k	commit stable, exactly `k` home blocks installed	post-state for every `k`
C5	all home blocks installed, log not released	post-state
C6	log released/checkpoint complete	post-state

Rebuild or restore an identical filesystem image before each cut, and make rename the only outstanding filesystem operation so rev5's group commit cannot fold unrelated mutations into the crash oracle. After boot, run both behavioral checks and an offline or boot-time invariant checker. A workload that keeps running after a failpoint is not testing power-loss semantics.

Crash-oracle checkpoint

Why is checking only whether `/b/new` exists insufficient after C4–k?

The name can exist while its inode is unallocated, its source name also remains, the displaced target's link count is wrong, or bitmap ownership is inconsistent. The oracle must inspect the complete pre/post tuple and global invariants: directory entries, inode types and link counts, block references, and bitmap ownership. Namespace appearance alone cannot detect a leaked or multiply owned object.

Required invariant checker

DEFINITION

A quiescent filesystem image satisfies the rename consistency predicate $\mathrm{ValidFS}(S)$ if and only if (1) every allocated regular inode i has $\mathrm{nlink}(i) = |\{(d, e) \in \mathrm{DirEntries}(S) : e.\mathrm{inum} = i\}|$, (2) the block-ownership mapping from allocated inodes to data/indirect blocks is injective and matches the bitmap, and (3) the namespace edge set equals either the exact pre-rename state or the exact post-rename state.

Write a separate checker or a bounded kernel diagnostic that operates on a quiescent image. It must report evidence rather than repair silently.

Required checks:

1. every nonzero directory entry has an in-range inode number whose on-disk type is allocated;
2. counted regular-file directory references equal on-disk `nlink`;
3. every direct and indirect data-block reference is in range and marked allocated;
4. no data or indirect block has two owners in this no-sharing format;
5. every allocated non-reserved block is reachable from exactly one inode metadata path;
6. `old` / `new` states match exactly one row of the operation's pre/post oracle; and
7. the log header is structurally valid under the pinned baseline's format.

The checker must distinguish reserved filesystem blocks from leaked data blocks. It must not count `.` as a regular-file hard link or apply regular-file link rules blindly to directory parent links.

Public behavior matrix

EXAMPLE

In case `P3` (replacing an open regular file `D`), `rename("old", "new")` unlinks `D`'s directory entry and decrements `D`'s on-disk `nlink` from 1 to 0, while `D`'s in-memory reference count `ref` remains 1 because a process still holds an open file descriptor to `D`. Consequently, `new` immediately resolves to `S`'s inode, existing reads on the open descriptor continue returning `D`'s displaced blocks, and `iunlockput` / `iput` frees `D`'s blocks and inode only when that descriptor is closed.

Implement deterministic tests for all rows. Each test starts from a fresh image or creates and removes its own fixtures without depending on test order.

ID	setup	required result
P1	same directory, target absent	old absent; new has source bytes and inode identity
P2	different directories, target absent	same post-state across parent boundary
P3	target is distinct regular file and remains open	new reaches source; old target descriptor still reads displaced bytes
P4	old and new are hard links to one inode	success no-op; both names and link count unchanged
P5	old missing	`-1`; target and all counts unchanged
P6	source or target is a directory	`-1`; entire tree unchanged
P7	final name is `.` or `..`, empty, or overlength	`-1`; no truncated alias created
P8	destination directory requires a new data block	success within measured log bound
P9	no free block for required directory growth	`-1`; old name preserved and no leak
P10	two opposite cross-directory renames repeated under watchdog	completion without deadlock; invariant checker clean
P11	concurrent lookup loops around successful rename	each lookup result belongs to allowed pre/post observation set
P12	every crash cut C0–C6 for absent and replacing targets	exact pre- or post-state; checker clean after recovery

Tests may expose inode numbers through a test-only `stat` field or checker; do not infer identity only from equal bytes.

Mutation tests against named wrong implementations

> **WARNING**
>
> A test suite that passes a mutated kernel (`M1` – `M8`) provides zero evidence for the invariant that mutant violates; every mutant must fail deterministically with the predicted invariant-checker or behavioral oracle diagnostic.

Create temporary faulty branches or patch toggles. The correct test suite must kill each mutant for the stated reason:

mutant	defect	test that must fail
M1 `clear-old-first`	removes source before target capacity is known	P9 observes lost source after ENOSPC
M2 `home-write-directory`	bypasses `log_write` for one directory block	C1/C2 or C4-k produces mixed pre/post namespace
M3 `keep-displaced-link`	never decrements replaced target	P3 checker reports link-count mismatch/leak
M4 `source-parent-first`	locks parents in argument order	P10 watchdog captures ABBA wait-for cycle
M5 `same-inode-is-replacement`	decrements link count when identities match	P4 reports two names but link count one
M6 `trust-first-lookup`	does not revalidate under parent locks	targeted concurrent unlink/link schedule changes wrong entry
M7 `allocate-after-clear`	destination growth can fail after publication begins	P9 leaves neither exact pre-state nor post-state
M8 `count-log-calls`	confuses helper calls with unique dirty blocks	P8 either rejects a fitting transaction or overruns reservation under boundary layout

Do not keep faulty code in the final normal path. Preserve the mutation procedure and failing output as test evidence.

Worked test design: kill M6 deterministically

A probabilistic loop may run thousands of times without hitting the race. Introduce a test-only barrier after the first source lookup but before acquiring/revalidating both parents. The test thread pauses there; a second thread replaces the source directory entry with a hard link to a different inode; the first resumes. A correct implementation either observes the changed identity during locked revalidation and applies the defined policy or fails without changing the wrong object. M6 mutates using stale identity and the checker exposes the mismatch.

The barrier belongs at a named state transition and is compiled out of release code. A sleep delay is not a deterministic schedule.

Evidence package

The complete implementation package contains:

```
README.md                build, boot, and 10-minute reproduction path
REPORT.md                contract, invariants, design, limits, and
results
BASELINE.md              course root, start mode, ordered
patch/checksum stack
PATCH.diff               rename-only change relative to the recorded
start point
tests/                   public behavior, concurrency, and mutation
drivers
tools/fscheck-rename.*   invariant checker source
scripts/crash-matrix.*   deterministic image restore and failpoint
runner
results/raw/             machine-readable logs for every P, C, and M
case
results/summary.md       compact table generated from named raw files
LICENSES/                retained upstream notices and added dependency
notices
```

`README.md` must identify any command that can overwrite an image or terminate a virtual machine. The crash runner may operate only on an explicit lab image path; it must refuse a missing or root-like target.

Design memo questions

Answer with code locations and traces:

1. What is the runtime linearization interval, and which locks exclude partial lookup observations?
2. How are two parents ordered? List every path that holds more than one inode lock.
3. Which operations can still fail after the first directory entry mutation, and why?
4. What is the worst-case unique dirty-block set for P8? Show overlaps and the log-capacity inequality.
5. Why is source `nlink` unchanged? When does displaced storage become reclaimable?
6. How does each crash cut map to log commit state?
7. Which test kills each named mutant, and what wrong state does it observe?
8. What behavior remains outside the contract?

Invariant and verification checklist

WARNING

Any implementation that bypasses `log_write` for a metadata update, acquires parent locks in caller-argument order without canonical `(dev, inum)` sorting, or leaves a mixed pre/post crash state triggers a non-compensable failure regardless of passing normal-execution tests.

Dimension	Invariant criterion	Verification evidence
Interface and namespace behavior	Correct error codes and atomic state changes	All P1–P9 results and exact no-change-on-error states
Locking and concurrency	Deadlock-free two-parent locking order	Canonical-order argument, deterministic stale-lookup schedule, P10–P11 traces
Crash consistency	All-or-nothing recovery across failures	Complete C0–C6 matrix for absent and replacing targets; exact pre/post recovery
Persistent invariants	Structural integrity of disk metadata	Checker source and clean reports; deliberately corrupted image is detected
Adversarial test power	Falsification of flawed rename variants	Every M1–M8 mutant killed by the predicted oracle
Implementation quality	Safe memory and error handling	Bounded helpers, checked errors, no bypass of log or license, readable diff
Reproducibility and evidence	Single-command automated replication	Pinned baseline, scripted clean run, raw data tied to summary
Synthesis analysis	Invariant reasoning under altered assumptions	Learner derives an unseen trace and explains limits without relying on patch text

Critical defect conditions

The implementation is invalid if its start point does not descend reproducibly from the exact course root, its rename-only patch cannot be separated from the checkpoint stack, it fails applicable upstream/checkpoint tests, corrupts an unrelated filesystem image, lacks source attribution, includes a copied full solution, or permits a crash state outside the declared pre/post set. Lengthy explanations cannot compensate for those failures.

Diagnostic analysis prompts

EXAMPLE

For the `MAXOPBLOCKS = 10` transaction capacity audit: if destination directory growth crosses from direct block 11 into indirect block 0, the operation can dirty the indirect map block, the new directory data block, the bitmap block, the destination directory inode block, the source directory data block, the source directory inode block, and the displaced target's inode block—7 unique blocks in the worst case, which satisfies $7 \leq \mathrm{MAXOPBLOCKS} = 10$ only because `log_write` absorbs multiple updates to the same buffer into a single log slot.

Self-audit against these changed scenarios:

- Change destination from absent to a second hard link of the source. Predict every count before running code.
- Move one `log_write` to a direct home write. Select the first crash cut that can expose it.
- Reverse two parent inode numbers while preserving pathname order. Show that acquisition order remains unchanged.
- Force destination directory growth from direct block 11 to indirect entry 0. Recompute the unique dirty set and its `MAXOPBLOCKS=10` inequality.
- Hold an open descriptor on the displaced target through recovery. Separate on-disk recovery state from lost process state.
- Remove the atomic-sector assumption for the commit header. State the additional mechanism required.

These questions are diagnostic self-checks. An engineer should be able to explain the lock order and crash oracle without relying on test logs alone.

REMARK

Restricting the core lab to regular-file rename isolates two-parent canonical locking, worst-case log-slot accounting, and deferred open-inode reclamation; directory rename adds tree-cycle detection (verifying the source directory is not an ancestor of the destination parent) and `..` link-count maintenance.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 6.4.1: Proving Deadlock Freedom in Cross-Directory Rename via Inode Sorting

Proof synthesis

Suppose Process 1 executes `rename("dirA/f", "dirB/f")` while Process 2 concurrently executes `rename("dirB/g", "dirA/g")`.

1. Explain how locking parent directories in caller argument order creates an immediate circular wait deadlock.
2. Prove that sorting parent directory acquisitions strictly by `(dev, inum)` mathematically eliminates deadlocks.

Establish a strict total order on locks: lock(min(inumA, inumB)) before lock(max(inumA, inumB)).

1. **Deadlock in caller order:**
   - Process 1 looks up `dirA` and `dirB`. Following argument order (`old_parent` then `new_parent`), Process 1 acquires `ilock(dirA)`.
   - Concurrently, Process 2 looks up `dirB` and `dirA`. Following its argument order, Process 2 acquires `ilock(dirB)`.
   - Process 1 attempts to acquire `ilock(dirB)` and blocks behind Process 2.
   - Process 2 attempts to acquire `ilock(dirA)` and blocks behind Process 1. Both processes hold one lock and wait for the other, forming a closed circular wait cycle ($T_1 \to \mathrm{dirB} \to T_2 \to \mathrm{dirA} \to T_1$)—an unrecoverable deadlock.

2. **Proof of deadlock freedom via canonical sorting:** Define a strict total order $\prec$ on directory inodes:

   $$I_1 \prec I_2 \iff (I_1.\mathrm{dev} < I_2.\mathrm{dev}) \vee (I_1.\mathrm{dev} == I_2.\mathrm{dev} \wedge I_1.\mathrm{inum} < I_2.\mathrm{inum}$$

   Mandate that whenever two parent directories must be locked simultaneously:
   - If $I_{\mathrm{d}}^{\mathrm{ol}} == I_{\mathrm{w}}^{\mathrm{ne}}$ (same-directory rename), lock the parent once.
   - If $I_{\mathrm{d}}^{\mathrm{ol}} \neq I_{\mathrm{w}}^{\mathrm{ne}}$, let $I_{\mathrm{t}}^{\mathrm{firs}} = \min_{\prec}(I_{\mathrm{d}}^{\mathrm{ol}}, I_{\mathrm{w}}^{\mathrm{ne}})$ and $I_{\mathrm{d}}^{\mathrm{secon}} = \max_{\prec}(I_{\mathrm{d}}, I_{\mathrm{w}})$.
   - The thread acquires `ilock(first)` then `ilock(second)`. Because every thread acquires locks strictly in ascending order under total order $\prec$, no cycle of waiters can form. Circular wait is eliminated.

Exercise 6.4.2: Computing Worst-Case Unique Dirty Blocks Against MAXOPBLOCKS

Verification application

In xv6 revision 5, `MAXOPBLOCKS = 10`. An adversarial cross-directory rename replaces an existing file in `dirB`, causing `dirB` to allocate an indirect block. Enumerate all distinct disk blocks dirtied by this operation and verify that the total unique block count does not exceed `MAXOPBLOCKS`.

Count unique disk blocks: bitmap, indirect, new directory block, destination inode, source directory block, source inode, target inode.

The worst-case cross-directory replacing rename dirties the following unique disk blocks:

1. Destination directory inode block (`dinode` for `dirB`).
2. Destination directory new data block (added to hold the new directory entry).
3. Destination directory indirect block (if directory size crossed from 12 direct blocks to indirect entry 0).
4. Free-space bitmap block (allocating the new directory block and indirect block; on xv6 `FSSIZE=2000` , one bitmap block covers all data blocks).
5. Source directory data block (clearing the old directory entry).
6. Source directory inode block (if in a different inode block from `dirB`).
7. Displaced target inode block (decrementing `nlink` to 0).
8. Target block bitmap block (if target blocks are freed; in xv6, block freeing is deferred until `iput()` when `ref` drops to 0).

Summing the unique touched blocks yields at most 7 (or 8 with separate bitmap blocks). Because $7 \leq \mathrm{MAXOPBLOCKS} = 10$, the operation fits strictly within the transaction log reservation. Repeated updates to `dirB` 's inode or bitmap are absorbed into single log slots by `log_write()` .

Exercise 6.4.3: Decreasing Link Count and Deferring Inode Block Reclamation

Construction application

Suppose `rename("dirA/f", "dirB/g")` overwrites an existing file `g` that is currently open by a background logging daemon with file descriptor 5.

1. Trace the modifications to `g`'s on-disk and in-core state inside the rename transaction.
2. Why must the rename transaction NOT immediately deallocate `g`'s data blocks?
3. When and where are `g`'s data blocks finally returned to the free bitmap?

Decrement nlink inside the transaction, but block deallocation occurs only when refcount drops to 0 in iput().

1. **Modifications inside the transaction:**
   - The rename operation overwrites `dirB`'s directory entry for `"g"` to point to `f`'s inode number.
   - The kernel acquires `ilock(g)`. It decrements `g->nlink` from 1 to 0 and logs the updated inode block via `iupdate(g)`.
   - The transaction commits.
2. **Deferred deallocation requirement:** The logging daemon holds descriptor 5, which references `g` via an open file description (`g->ref > 0`). Deallocating `g`'s blocks immediately would cause the daemon's subsequent writes to overwrite blocks that might be reallocated to unrelated files, violating file isolation.
3. **Final deallocation path:** When the logging daemon calls `close(5)` (or terminates), the VFS executes `iput(g)`. Finding that `g->ref == 0` and `g->nlink == 0`, `iput()` begins a new transaction, invokes `itrunc(g)` to free all direct and indirect data blocks to the bitmap, clears `g->type`, and writes the free inode to disk.

Exercise 6.4.4: Mapping Deterministic Crash Cuts to Log Commit States

Verification transfer

An adversarial test harness injects a power failure during a cross-directory rename.

- Cut C_1: Power fails while writing journal block 3 (before the log commit record is written).
- Cut C_2: Power fails after the log commit record is written, while installing the source directory block to its home disk location.

1. State the post-reboot filesystem state observed for Cut C_1.
2. State the post-reboot filesystem state observed for Cut C_2.

Uncommitted transactions are discarded; committed transactions are completely replayed.

1. **Cut C_1 (Pre-commit failure):** The log header on disk does not reflect a valid commit record for this transaction. During recovery upon reboot, `initlog()` inspects the log header, finds `n == 0` or uncommitted state, and skips replay. All home disk blocks remain in their exact pre-crash state: the source file remains in `dirA`, the destination file in `dirB` is untouched, and no partial or corrupt rename is visible.
2. **Cut C_2 (Post-commit failure):** The log header contains a valid committed transaction describing all modified blocks. During recovery, `install_trans()` reads every logged block from the journal and writes it to its home location, completing the installation. Upon mount, the filesystem reflects the complete post-rename state: `dirB/g` points to the new file, `dirA/f` is removed, and the displaced target's link count is decremented.

Exercise 6.4.5: Cycle Detection and Ancestor Validation in Directory Renaming

Description application

Explain why renaming a directory (`rename("dirA/sub", "dirB/target")`) requires an ancestor traversal check that regular-file renaming does not. Show how omitting this check can detach a subtree from the root directory.

Check that new_parent is not inside the subtree rooted at old_dir to avoid creating an unreachable cycle.

- **The subtree cycle hazard:** If an application executes `rename("dirA", "dirA/sub/nested")` :
  - `dirA` is moved to become a child of `dirA/sub` .
  - However, `sub` is already a child of `dirA` .
  - Modifying the directory pointers without checking creates a closed directed cycle in the directory tree: `dirA` points to `sub` , and `sub` points to `dirA` .
  - The entire directory subtree is now detached from the root directory `/` . No path from `/` can ever reach `dirA` or any file inside it, yet all inodes retain positive link counts. The entire subtree becomes orphaned, permanently leaking storage.
- **The ancestor validation check:** Before permitting a directory rename, the kernel must walk upward through parent pointers (`..`) starting from the destination parent directory `dirB` up to the root `/` . If `dirA` is encountered along this upward traversal path, the destination lies inside the source directory's own subtree. The kernel must abort the operation and return `-EINVAL` .

References & Further Reading

- MIT PDOS, xv6 RISC-V source and book — official text and upstream entry point; the implementation target remains the immutable rev5 commit below.
- MIT PDOS, xv6–rev5 `kernel/sysfile.c` — immutable link, unlink, create, and open transaction paths against which rename is integrated.
- MIT PDOS, xv6–rev5 `kernel/fs.c` — immutable inode, directory, allocation, path, and ENOSPC helper contracts.
- MIT PDOS, xv6–rev5 `kernel/log.c` and `kernel/param.h` — immutable log admission, capacity, commit, installation, and recovery behavior.
- MIT PDOS, xv6–rev5 license — notice that must remain with upstream-derived work.
- The Open Group, POSIX.1–2024 `rename()` — production interface reference used to identify both adopted and deliberately excluded semantics.
- Linux kernel documentation, ext4 Journal (JBD2) — production contrast for commit records, replay, checksums, and checkpointing.

Protection, isolation, and synthesis

SECTION DETAIL

Protection and Authority

A privileged document converter receives an input descriptor and a caller-supplied output pathname. It also needs write access to `/var/lib/converter/accounting`. A malicious caller chooses that accounting pathname as the output. The converter faithfully opens the requested name with its own authority and overwrites the file it was supposed to protect.

No memory-safety violation is required. Authentication succeeded, permissions behaved as configured, and the converter performed a permitted write. The failure is **authority confusion**: the converter could not tell whether a write request exercised authority delegated by the caller or ambient authority held for its own bookkeeping.

Name the actors and the authority

Protection answers whether a subject may perform an operation on an object in a state. The vocabulary must remain separate:

term	precise role	example
principal	entity to which accountability or policy attaches	user, service identity, tenant
subject	active execution context acting for one or more principals	process, thread, protection domain
object	protected resource with stable identity	inode, endpoint, address space, device
operation	requested state transition or observation	read, map, signal, send, delegate
permission	policy relation allowing one operation	`read` on object 17
authority	usable power to affect an object, including powers obtained by delegation	descriptor granting write to object 17
credential	evidence used to establish a principal or attributes	UID, signed token, certificate

Authentication establishes or supports a claim about a principal. Authorization decides whether the acting subject has the required authority. Auditing records a decision or action. None substitutes for the others.

For subjects S, objects O, and operation set R, an access matrix is

$$A : S \times O \to \mathcal{P}(R).$$

$A[s, o]$ is the set of operations subject s may invoke on object o. The matrix is a specification, not usually a dense implementation. An access-control list stores a column by object: which subjects have which rights. A capability list stores a row by subject: which object references and rights the subject possesses.

> ***Object capability.*** *An unforgeable reference that designates an object and carries authority to invoke a bounded set of operations on it. Possession supplies both designation and authorization.*

Unforgeability can come from kernel-managed table entries, tagged hardware, cryptographic construction under a stated threat model, or another protected representation. A pathname string is not an object capability: it is forgeable data requiring a fresh namespace lookup and separate authorization.

Worked representation: one matrix, two indexes

Suppose:

subject / object	upload U	cache directory C	audit endpoint L
parser P	`read`	—	`send`
renderer R	`read`	`create`	`send`
janitor J	—	`list`, `remove`	—

Object C's ACL is `{R:create, J:list+remove}`. R's capability list is `{U:read, C:create, L:send}`. Both encode cells from the same abstract matrix, but their engineering costs differ:

- answering "who can remove from C?" is local in C's ACL;
- handing R attenuated `create` authority to one worker is local with a delegable capability;
- revoking all access to C is direct at a central ACL check;
- revoking one copied capability requires indirection, versioning, a revocation tree, object destruction, or another explicit mechanism.

The words ACL and capability do not decide policy quality. A capability carrying broad directory mutation can exceed least authority; an ACL can be narrowly scoped and completely mediated.

Retrieval checkpoint: designation versus authorization

Why can `open("/tenant/report", O_WRONLY)` be vulnerable even when every pathname component passes a permission check?

The caller supplied a **designation**—a string interpreted in a mutable namespace. Permission checks answer whether the converter's subject may traverse and write the object that the string resolves to. They do not establish that the caller was authorized to choose that object for this operation. A confused deputy exploits the converter's broader authority. A sound protocol must bind caller-conferred authority to the designated object, for example by accepting an already opened descriptor with bounded rights or resolving beneath a directory capability supplied for this request.

ACCESS CONTROL MATRICES, CAPABILITIES, AND CONFUSED DEPUTIES

Evaluate each statement regarding access control models, capabilities, and system authority:

Answer: Access control lists represent columns of an access matrix, whereas capability lists represent rows held by subjects.; Passing explicit unforgeable object descriptors over IPC eliminates confused-deputy ambient authority vulnerabilities.

ACLs store authorization by object (matrix columns); capability lists store rights by subject (matrix rows). Passing explicit descriptors binds designation to authority, preventing confused deputies. Linux capabilities grant ambient coarse-grained privilege bypasses rather than referencing single objects, and access() followed by open() creates a classic TOCTOU window.

Derive the reference monitor obligations

DEFINITION

A reference monitor mediates an access request (s, o, r) soundly only if it satisfies three properties simultaneously: **complete mediation** (no access path bypasses the check or uses stale unrevoked cached authority), **tamper resistance** (untrusted subjects cannot modify policy state or forge references), and **verifiability** (the trusted enforcement boundary is small and analyzable).

A protection decision is trustworthy only if every relevant access path passes through a mechanism that policy controls.

Reference monitor obligations and failure modes:

Obligation	Architectural implementation mechanism	Concrete failure mode when violated
Complete mediation	Hardware MMU checks every memory access; syscall trap checks all requests	Peripheral DMA bypasses CPU MMU; or stale cached capability permits revoked access
Tamper resistance	Hardware privilege rings (U-mode vs S-mode); kernel memory protected	Stack buffer overflow overwrites kernel PC; or writable mapping aliases page table
Verifiability	Minimal trusted computing base with formalized state invariants	Tens of millions of lines in a monolithic kernel preclude formal correctness proofs

Complete mediation does not require a slow policy database lookup on every byte. A successful `open` can create a kernel-protected descriptor that embodies the authorized object and access mode; subsequent `read` operations mediate possession and rights of that descriptor. The cached decision is valid only while the policy contract permits it. Revocation semantics must say whether existing descriptors continue working.

Saltzer and Schroeder's related design principles sharpen the mechanism:

- **fail-safe defaults:** authority is absent unless granted;
- **least privilege/least authority:** a subject receives only powers needed for the current task and duration;
- **least common mechanism:** minimize shared mechanism on which mutually distrusting subjects depend;
- **separation of privilege:** require independent conditions when one condition is too dangerous; and
- **economy of mechanism/open design:** make the trusted mechanism analyzable rather than relying on design secrecy.

These are design tests, not slogans. "Use least privilege" is incomplete until the object set, operations, duration, delegation, and failure state are enumerated.

Worked exploit: the pathname deputy

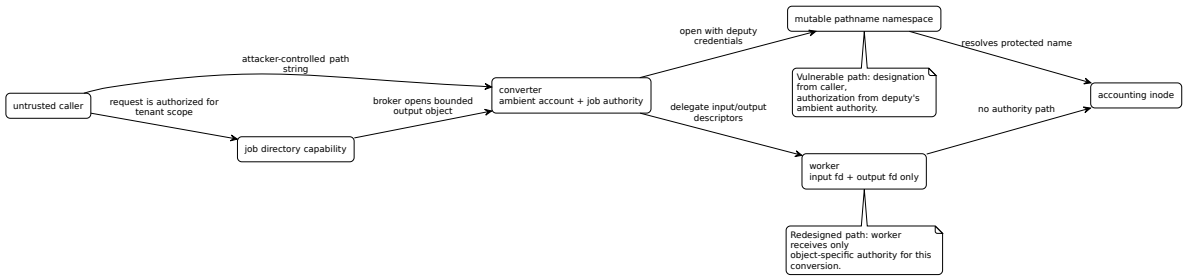
The converter runs with write authority to:

```
/var/lib/converter/accounting
/srv/jobs/<tenant-output>
```

Its API is `convert(input_fd, output_path)`. It validates document syntax, then opens `output_path` itself. The attack trace is:

step	subject and authority	transition
1	caller controls string but cannot write accounting	passes `/var/lib/converter/accounting`
2	converter has ambient write authority	resolves string under converter credentials
3	kernel authorizes converter	returns writable descriptor to accounting object
4	converter writes output	accounting object is corrupted

The permission system did exactly what its input requested. The API discarded provenance: it could not express “use only write authority delegated by this caller for this job.”



Repair A: descriptor-oriented delegation

An authorized broker opens the output under the tenant's approved directory, verifies object type, and passes the descriptor to a worker. The worker receives input-read, output-write, and bounded audit-send authority. It never receives a pathname-based ability to open arbitrary files.

The API becomes conceptually `convert(input_fd, output_fd, audit_endpoint)`. The worker can corrupt the designated output if its conversion logic is faulty; it cannot select accounting unless the broker delegated that object.

Repair B: attenuated directory authority

If the worker must create a name, give it a descriptor for one tenant directory and use relative, confinement-aware resolution. Restrict mutation operations, final-component policy, symbolic links, and mount crossings according to the service contract. On capability systems, derive a child capability with reduced rights; on POSIX-like systems, compose directory descriptors, process credentials, namespaces, and kernel resolution controls.

Neither repair proves content safety, kernel correctness, or durability. It removes a specific authority path.

Application checkpoint: verify least authority

The worker needs to read one input, create one output in a tenant directory, and emit one audit record. It receives a read-write descriptor for the entire tenant root and a network socket able to contact any address. Identify excess authority and propose an attenuated set.

Excess powers include reading or overwriting unrelated tenant objects, traversing subdirectories, deleting or renaming names if the descriptor and APIs permit it, and sending arbitrary network traffic. A narrower set is: an input descriptor with read-only access; either an already opened output descriptor with write-only/truncate semantics or a create-only authority scoped to one directory and validated name; and a connected or brokered audit endpoint that accepts only the expected message form. Close or revoke each after the conversion.

The answer must discuss operations as well as object scope. Replacing "tenant root" with "tenant output directory" while retaining remove, rename, and read authority may still exceed the task.

Delegation, attenuation, and revocation

Capabilities make delegation explicit: subject A transfers or derives a reference for B. Safe derivation is **monotonic attenuation**—the child authority is no greater than the parent in object scope, operation set, or an enforced constraint such as time or quota.

Let a capability be modeled as $c = (o, R, g)$, where o is an object, R rights, and g an optional guard. A derived capability $c' = (o', R', g')$ is attenuated only if the kernel-enforced meaning satisfies

$$Authority(c') \subseteq Authority(c).$$

Matching object identifiers and checking $R' \subseteq R$ is insufficient if a new guard broadens name resolution or if the derived reference reaches an alias outside the parent's object scope.

Revocation is not automatic:

- an ACL check can reject future opens after an entry is removed, while already open descriptors may remain valid by policy;
- a direct object capability copied to many subjects may require deleting the object, traversing a derivation tree, or routing through a revocable proxy;
- time-limited signed capabilities rely on clock and key assumptions; and

- revocation that races an in-flight operation needs a defined linearization point.

Worked trace: delegated read with revocation proxy

Subject A owns object X and creates proxy P. B receives capability `read(P)`, while P holds `read(X)`.

1. Before revocation, B invokes P; P checks its live state and forwards read to X.
2. A marks P revoked under P's lock.
3. A later read beginning after that transition is rejected.
4. A read already forwarded to X may complete unless the contract includes cancellation.

The proxy adds an indirection and a failure/availability dependency. It supplies a crisp future-use revocation point that an unmediated copied capability lacks. Destroying X revokes more broadly but may violate other clients' availability.

Transfer checkpoint: choose the revocation contract

A signing service delegates "sign at most ten artifacts before 18:00" to a build worker. List the kernel or trusted-service state required to enforce both constraints, and explain why a bearer token containing those words is insufficient by itself.

The operation count needs non-forgeable, race-free mutable state shared by all copies or a trusted online mediator that decrements a counter atomically. The deadline needs a trusted clock and a policy for clock rollback and in-flight operations. The object and key must remain bound to the authority; the worker must not possess an unrestricted signing-key copy. A bearer token is only serialized policy input until a trusted component verifies it and controls every signing path. Copying a ten-use token to two workers must not create twenty uses.

ACLs and capabilities fail in different ways

> **WARNING**
>
> Storing permissions by object (ACLs) simplifies central revocation and auditing of who can access an object, but encourages ambient authority and confused-deputy bugs when callers pass forgeable names; storing unforgeable references by subject (capabilities) binds designation to authority, but requires explicit indirection or revocation trees to claw back delegated access.

Consider an object store with one million objects and short-lived workers:

question	ACL-centered pressure	capability-centered pressure
enumerate all users of one object	local to object policy	may require tracking distributed copies
delegate one object to a worker	update policy or mint scoped credential	transfer/derive object reference
revoke all future access	update central policy	revoke proxy/version/object or derivation subtree
avoid ambient object naming	naming and authorization remain separable	designation and authority can be one reference
audit principal intent	identity policy is direct	possession may need explicit principal/audit binding

A system can combine them. The relevant review question is which component mediates which path and what state survives delegation, caching, process creation, and revocation.

Linux capabilities are not object capabilities

Linux divides powers traditionally associated with superuser into per-thread **capability** units such as `CAP_CHOWN` or `CAP_NET_ADMIN`. Permitted, effective, inheritable, ambient, and bounding sets govern their availability across operations and `execve` under detailed rules.

Despite the name, `CAP_DAC_OVERRIDE` is not an unforgeable reference to one file. It is broad privilege to bypass a class of discretionary access checks within the relevant namespace context. Granting it may increase ambient authority over many objects. By contrast, a descriptor opened to one file is closer to an object-specific capability: it designates one kernel object and carries bounded access mode, though surrounding APIs, `ioctl`s, delegation, and revocation semantics still require review.

Similarly, a UID is a credential attribute, not a reference to every object that policy might allow that UID to access. "Use capabilities" is ambiguous until the mechanism is named.

Boundary checkpoint: seccomp and authority

A worker may call only `read`, `write`, `close`, and `exit` under a system-call filter. It inherits a writable descriptor for the host's audit database. Is it least-authority with respect to that database?

No. `write` remains permitted and the inherited descriptor designates the database with write authority. Reducing syscall surface does not attenuate the descriptor. The worker needs the descriptor closed or replaced with a narrower endpoint. Conversely, closing the descriptor does not eliminate every kernel attack path; syscall filtering and object authority address different dimensions.

REMARK

Object capabilities and access-control lists are dual representations of the abstract access matrix $A : S \times O \to \mathcal{P}(R)$; subsequent lessons on containers and virtual machines examine how the operating system enforces resource and fault boundaries when subjects also distrust the shared runtime or kernel.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 7.1.1: Access Matrix Row and Column Duality for ACLs and Capabilities

Verification application

An access matrix A defines permissions for three subjects $\{S_1, S_2, S_3\}$ across three objects $\{O_1, O_2, O_3\}$:

- $A[S_1, O_1] = \{\text{read}, \text{write}\}$, $A[S_1, O_3] = \{\text{read}\}$
- $A[S_2, O_2] = \{\text{read}, \text{execute}\}$, $A[S_2, O_3] = \{\text{read}, \text{write}\}$
- $A[S_3, O_1] = \{\text{read}\}$, $A[S_3, O_2] = \{\text{read}\}$

1. Write the Access Control Lists (ACLs) associated with objects O_1, O_2, O_3.
2. Write the Capability Lists associated with subjects S_1, S_2, S_3.
3. Explain which representation is more efficient when revoking all access to O_3, versus revoking all access held by S_2.

ACLs partition matrix A by columns; capability lists partition matrix A by rows.

1. **Access Control Lists (Columns of A):**
   - $\mathrm{ACL}(O_1) = \{(S_1, \{\mathrm{read}, \mathrm{write}\}), (S_3, \{\mathrm{read}\})\}$.
   - $\mathrm{ACL}(O_2) = \{(S_2, \{\mathrm{read}, \mathrm{execute}\}), (S_3, \{\mathrm{read}\})\}$.
   - $\mathrm{ACL}(O_3) = \{(S_1, \{\mathrm{read}\}), (S_2, \{\mathrm{read}, \mathrm{write}\})\}$.

2. **Capability Lists (Rows of A):**
   - $\mathrm{CapList}(S_1) = \{(O_1, \{\mathrm{read}, \mathrm{write}\}), (O_3, \{\mathrm{read}\})\}$.
   - $\mathrm{CapList}(S_2) = \{(O_2, \{\mathrm{read}, \mathrm{execute}\}), (O_3, \{\mathrm{read}, \mathrm{write}\})\}$.
   - $\mathrm{CapList}(S_3) = \{(O_1, \{\mathrm{read}\}), (O_2, \{\mathrm{read}\})\}$.

3. **Revocation efficiency:**
   - **Revoking all access to object O_3**: Trivial with ACLs: the kernel modifies the single entry $\mathrm{ACL}(O_3)$ directly at the object. With capabilities, finding all references to O_3 requires traversing the capability lists of all subjects in the system (unless indirect capability indirection or generation counts are maintained).
   - **Revoking all access held by subject S_2**: Trivial with capabilities: destroying S_2 's capability list revokes all its access in one step. With ACLs, the kernel must scan every object's ACL in the entire filesystem to purge entries naming S_2.

Exercise 7.1.2: Tracing and Repairing a Confused-Deputy Attack Scenario

Construction application

A compiler daemon runs as root (`uid 0`) to compile user programs and write billing metrics to `/var/log/billing.log`. The user invokes the compiler with: `compile --src main.c --out /var/log/billing.log`.

1. Trace how the compiler becomes a confused deputy and overwrites the billing log.

2. Explain why checking the caller's permissions via `access(out_path, W_OK)` before opening fails to solve the architectural problem.
3. Formulate the capability-based architectural repair that eliminates the vulnerability.

The deputy combines client-chosen designation with server ambient authority. Pass an open descriptor instead.

1. **Exploitation trace:**
   - User inputs destination path `/var/log/billing.log`.
   - The compiler exercises its ambient `root` authority and calls `open("/var/log/billing.log", O_WRONLY | O_CREAT | O_TRUNC)`.
   - Because `root` possesses write authority over `/var/log`, the open succeeds.
   - The compiler writes compiled machine code over the billing log, destroying audit history. The compiler was confused: it used its own ambient authority to access an object designated by an untrusted client.
2. **Failure of `access()` check:**
   - First, `access()` creates a time-of-check to time-of-use (TOCTOU) race condition: an attacker can replace the path with a symlink between `access()` and `open()`.
   - Second, it relies on global ambient path lookup rather than conferring explicit authority.
3. **Capability-based repair:** The client must open the destination file in its own user context and pass the resulting open file descriptor to the compiler service over a Unix domain socket (`SCM_RIGHTS`). The compiler writes exclusively through the provided descriptor. If an unprivileged user specifies `/var/log/billing.log`, the user's initial `open()` fails with `EACCES` before the compiler is ever invoked. Designation and authority are bundled into an unforgeable descriptor.

Exercise 7.1.3: Eliminating Time-of-Check to Time-of-Use (TOCTOU) Races via openat

Construction transfer

A setuid-root program verifies that a target file in `/tmp/scratch` belongs to the user before writing to it:

```
struct stat st;
stat("/tmp/scratch/file", &st);
if (st.st_uid == getuid()) {
    // [ATTACK WINDOW]
    int fd = open("/tmp/scratch/file", O_WRONLY);
    write(fd, payload, size);
}
```

1. Construct the concurrent race schedule whereby an attacker replaces `/tmp/scratch/file` with a symlink to `/etc/shadow` to overwrite password hashes.
2. Rewrite the code using `open()` flags (`O_NOFOLLOW`) or `fstat()` on an opened descriptor to completely eliminate the TOCTOU race.

fstat on the opened descriptor inspects the exact inode that was opened; O_NOFOLLOW refuses symlinks.

1. **Attack schedule:**
   - Attacker creates regular file `/tmp/scratch/file` owned by their user UID.
   - Setuid-root process calls `stat("/tmp/scratch/file", &st)`. The check `st.st_uid == getuid()` passes.
   - In the attack window before `open()`, attacker executes: `unlink("/tmp/scratch/file"); symlink("/etc/shadow", "/tmp/scratch/file");`
   - Setuid-root process resumes and calls `open("/tmp/scratch/file", O_WRONLY)`. The kernel resolves the symlink and opens `/etc/shadow` with root privileges.
   - Setuid-root writes user payload into `/etc/shadow`, corrupting system credentials.

2. **Race-free repair:** Open the file first without following symlinks and inspect the resulting descriptor directly:

```
int fd = open("/tmp/scratch/file", O_WRONLY | O_NOFOLLOW);
if (fd < 0) return -1;
struct stat st;
if (fstat(fd, &st) < 0 || st.st_uid != getuid() ||
!S_ISREG(st.st_mode)) {
    close(fd);
    return -1;
}
write(fd, payload, size);
close(fd);
```

   `fstat(fd, ...)` inspects the identical in-memory inode that was opened. Any subsequent rename or symlink modification in `/tmp` has zero effect on the already opened file description.

Exercise 7.1.4: Contrasting Linux Thread Privileges with Fine-Grained Object Capabilities

Description application

Contrast Linux thread capability bits (`CAP_SYS_ADMIN`, `CAP_NET_RAW`, `CAP_DAC_OVERRIDE`) with object-capability systems (such as Capsicum or seL4). Explain why granting `CAP_DAC_OVERRIDE` to a service increases ambient authority rather than enforcing least authority.

Linux capabilities are coarse privilege flags; object capabilities are unforgeable handles to specific resources.

- **Architectural distinction:**
  - **Linux capabilities (`capabilities(7)`):** Partition the monolithic `root` UID into approximately 40 coarse-grained privilege categories associated with a thread credential. They act as global permission flags checked by the kernel when an unprivileged operation is attempted.
  - **Object capabilities (Capsicum / seL4):** Unforgeable, fine-grained references that pair the identity of a specific resource (e.g., file descriptor 4, endpoint `0x10A`) with a restricted bitmask of allowed methods (`CAP_READ`, `CAP_SEEK`). A thread can invoke operations only on resources for which it holds an explicit capability.
- **Ambient authority hazard of `CAP_DAC_OVERRIDE`:** `CAP_DAC_OVERRIDE` instructs the VFS to bypass all discretionary file read, write, and execute permission checks across the entire filesystem hierarchy. Granting `CAP_DAC_OVERRIDE` to a service that needs to modify a single configuration file does not restrict it to that file; it grants the ambient power to read or overwrite every file, shadow password, and device node on the system. It maximizes ambient authority rather than enforcing least authority.

Exercise 7.1.5: Proving Confinement and Attenuation in Capability Sandboxes

Proof synthesis

Let a parent process hold capability $C = (O, \{\text{read}, \text{write}, \text{grant}\})$.

1. Define the principle of capability attenuation and prove that a sandboxed child created with derived capability $C' = (O, \{\text{read}\})$ cannot forge or recover write authority over O.
2. Formulate Lampson's confinement problem: why can a child restricted to read-only capabilities still leak data to an external collaborator?

Capability derivation allows monotonically decreasing rights; confinement must prevent covert channels.

1. **Proof of attenuation safety:**
   - **Attenuation rule:** A subject holding capability $C = (O, R)$ can derive a new capability $C' = (O, R')$ if and only if $R' \subseteq R$. The derivation operation is monotonic: rights can be surrendered, never added ($R' \not\supset R$).
   - Because capability registers or tables are maintained exclusively in kernel space (or protected by hardware tags), user-mode code cannot alter the rights bitmask directly.
   - When the parent derives $C' = (O, \{\text{read}\})$ and executes the child in a sandbox lacking the original capability C and lacking `grant` authority, the child's capability table contains only C'.
   - The kernel reference monitor validates every system call against the supplied capability descriptor. Any attempt by the child to issue a write request on C' is rejected because $\text{write} \notin \{\text{read}\}$. Since the child has no interface to mint or expand capabilities, write authority cannot be recovered.
2. **Lampson's confinement problem (1973):** Even if the child is completely deprived of write and network capabilities, it can transmit confidential data read from O to an unconfined conspirator process via **covert channels** (e.g., modulating CPU scheduling bursts, varying memory footprint to induce page faults, lock contention on shared hardware resources, or thermal throttling). Preventing unauthorized data release requires eliminating all observable side-channel modulations, a much harder problem than discretionary access control.

References & Further Reading

- Saltzer and Schroeder, "The Protection of Information in Computer Systems" (1975) — primary statement of design principles and protection mechanisms.
- Norman Hardy, "The Confused Deputy (or why capabilities might have been invented)" — primary confused-deputy formulation and capability-oriented diagnosis.
- Dennis and Van Horn, "Programming Semantics for Multiprogrammed Computations" (MIT-LCS-TR-023) — primary capability-oriented protection model.

- Watson et al., “Capsicum: Practical Capabilities for UNIX” — object-capability application compartmentalization on a UNIX interface.
- seL4 Reference Manual and capability tutorial — kernel-object capability semantics and rights.
- Linux kernel documentation, Credentials in Linux — task, file, UID/GID, key, and capability credential structures.
- Linux man-pages, `capabilities(7)` — Linux privilege-unit semantics and capability sets.
- Linux man-pages, `openat2(2)` — resolution controls relevant to descriptor-relative confinement.

SECTION DETAIL

Virtual Machines

A guest kernel believes it controls page tables, interrupt masking, and device registers. Running it as an ordinary host process cannot preserve that illusion: an unmodified privileged instruction may fail, silently expose host state, or alter resources belonging to another guest. Running it with unrestricted hardware privilege is worse—the first guest bug becomes a host compromise.

A virtual-machine monitor (VMM) resolves the contradiction by giving most guest instructions direct hardware execution while retaining control of sensitive state. Hardware establishes a guest execution mode, records controlled exits, and inserts another authorization/translation stage below guest decisions. The abstraction is an entire machine, not merely another virtual address space.

The VMM contract

Use these terms precisely:

term	role
host	execution environment that owns physical resources and runs the VMM stack
guest	software stack presented with a virtual hardware interface
VMM/hypervisor	trusted mechanism that multiplexes and controls virtual machines
vCPU	schedulable virtual processor state: guest registers, virtual control state, pending events
VM entry	transfer from VMM control into guest execution
VM exit	hardware-mediated transfer from guest execution to VMM handling
emulated device	software state machine implementing a guest-visible device contract
paravirtual interface	guest-aware interface designed to avoid expensive hardware emulation paths

Popek and Goldberg's model gives three requirements:

1. **Equivalence/fidelity:** software in the virtual environment has behavior equivalent to the real-machine contract, apart from permitted differences such as timing and resource availability.
2. **Resource control/safety:** the VMM retains control of the resources it virtualizes.
3. **Efficiency:** a statistically dominant subset of guest instructions executes without VMM intervention.

For their abstract machine, a sufficient condition is that every sensitive instruction be privileged—one that traps when executed without the required privilege. Modern extensions supply explicit guest modes and intercept controls where older instruction sets made classical trap-and-emulate difficult. The theorem is model-specific; it is not a claim that every modern device, timing source, or microarchitectural effect is perfectly virtualized.

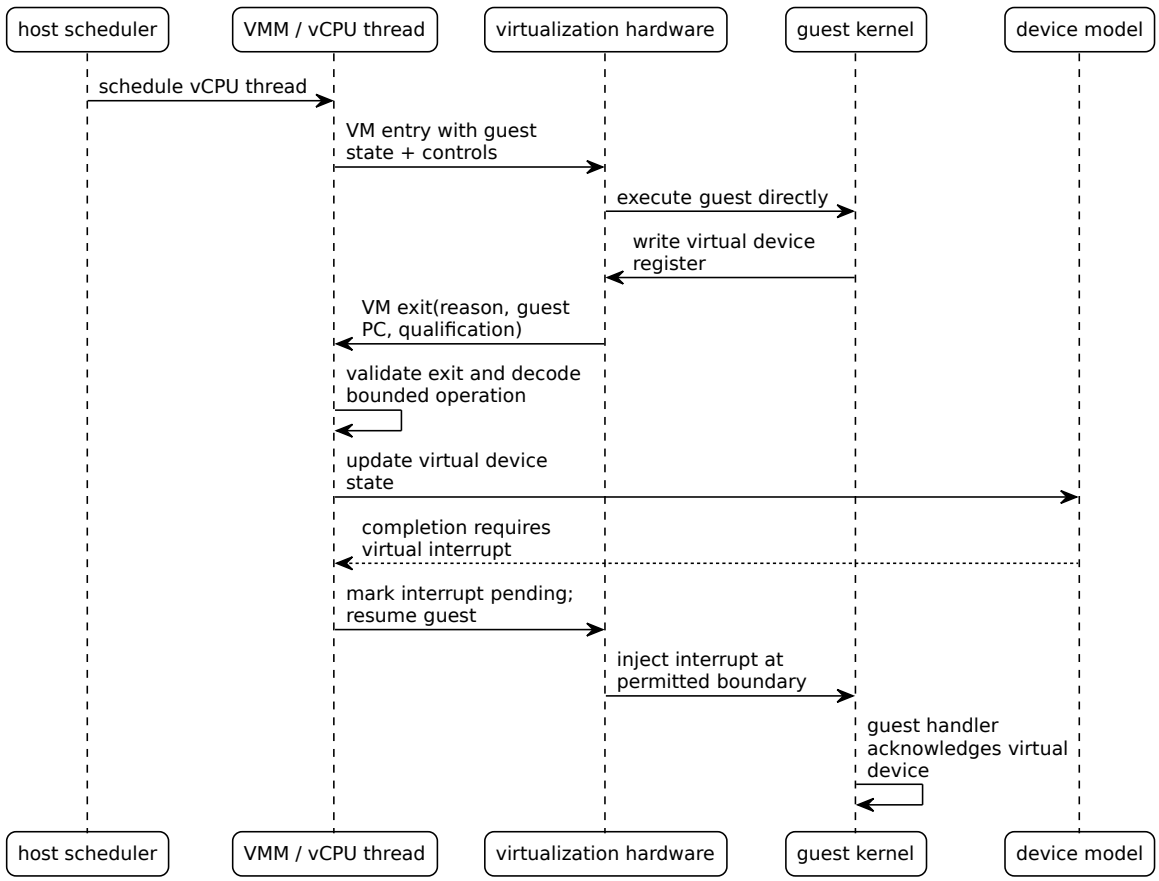
Retrieval checkpoint: why not emulate everything?

State one correctness benefit and two costs of interpreting every guest instruction in software instead of using controlled direct execution.

Interpretation gives the monitor an opportunity to inspect and define every instruction's effect, which can support architectures unlike the host or detailed instrumentation. It adds dispatch/translation work to every instruction and prevents the hardware from directly exploiting its normal execution pipeline and many caches. Controlled direct execution keeps VMM intervention concentrated on selected exits. It still requires the VMM to validate every exit path and preserve the guest-visible contract.

Controlled direct execution

Before VM entry, the VMM loads or identifies guest register state, virtual interrupt state, intercept policy, and the root of second-stage translation. Hardware executes guest instructions until an event configured to exit, a second-stage fault, an external interrupt requiring host handling, or an explicit hypercall transfers control to the VMM.



The exit record is untrusted input in the security sense: a malicious guest can deliberately trigger unusual exit reasons and operands. The VMM must validate addresses, lengths, device state, and transitions before touching host resources.

Worked trace: emulated MMIO completion

Assumptions: a paravirtual block device exposes a queue-notify register in guest-physical MMIO space; that page is configured to exit on write; descriptors reside in guest memory; no device assignment is active.

1. The guest driver places a request descriptor chain in guest memory.
2. It executes a store to the notify register's guest virtual address.
3. Guest page tables translate that address to the MMIO guest-physical page.

4. The second-stage mapping marks the access for VMM handling, so hardware records a VM exit rather than issuing a host physical store.
5. The VMM verifies the exit reason and guest-physical address, then the device model validates descriptor bounds and translates guest memory references through VMM-controlled mappings.
6. After host I/O completes, the VMM marks a virtual interrupt pending.
7. Hardware injects it only when guest interrupt state permits; the guest handler consumes the completion.

This trace is one design, not universal VM I/O. Device assignment can let hardware DMA directly under IOMMU control; userspace device models can add process crossings; polling can replace interrupts. The guest-visible device contract and trust boundary must be stated before performance claims.

Application checkpoint: classify four events

For each event, state whether it necessarily exits under all VMM designs, may execute directly, or depends on configured interception: ordinary integer addition, guest page-table update, second-stage permission failure, and virtual-device MMIO write.

- Ordinary integer addition is intended to execute directly when it touches no intercepted state.
- A guest page-table memory write may execute as an ordinary store, while later TLB-management or page-table use is virtualized; a VMM may instead write-protect and intercept page-table pages in a shadow-paging design.
- A second-stage permission failure necessarily transfers to a more privileged handler because the guest cannot authorize access beyond the VMM mapping.
- A virtual-device MMIO write depends on the device path: emulation usually arranges an exit, while a directly assigned device may be mapped for hardware access under IOMMU and interrupt-remapping controls.

Answers that label every privileged-looking guest operation an exit have ignored hardware guest modes and configuration. Answers that let a second-stage fault remain entirely under guest control violate resource control.

TWO-STAGE TRANSLATION AND VM EXECUTION INVARIANTS

Evaluate each statement regarding hardware-assisted virtualization and two-stage address translation:

Answer: Effective memory permissions in virtual machines are the strict intersection of stage-1 and stage-2 permissions.; A stage-2 translation fault raises a VM exit to the hypervisor rather than delivering a trap directly to the guest.

Effective permissions are the logical intersection of both stages: if either stage forbids an access, hardware faults. Stage-2 translation faults exit to the hypervisor to preserve host resource control. Guests cannot override host stage-2 restrictions, and nested paging replaces software shadow page tables while retaining VM exits for I/O and unmapped pages.

Two-stage address translation

The guest controls a mapping from guest virtual address (GVA) to guest physical address (GPA). The VMM controls a second mapping from GPA to host physical address (HPA):

$$GVA \xrightarrow{\text{guest page tables}} GPA \xrightarrow{\text{VMM-controlled stage}} HPA.$$

The effective permission is the intersection of both stages. A guest cannot grant itself host memory by installing a permissive guest PTE when the second stage lacks that mapping.

RISC-V's ratified H extension names the first guest-controlled translation VS-stage and the second G-stage. Other architectures use different names and control structures. The causal model is the same: guest ownership claims are subordinate to VMM resource mappings.

Worked translation: permission stops the write

Assume 4 KiB pages:

```
GVA 0x00004123
guest PTE: GVPN 0x4 -> GPFN 0x2a, permissions R/W
second-stage PTE: GPFN 0x2a -> HPFN 0x91, permission R only
```

Translation preserves the 12-bit offset `0x123`:

$$GPA = (0x2a \ll 12) + 0x123 = 0x2a123,$$

$$HPA = (0x91 \ll 12) + 0x123 = 0x91123.$$

A read may reach HPA `0x91123`. A write passes the guest PTE but fails the second-stage write check before host memory is modified. Hardware reports a guest-page or second-stage fault to the VMM with architecture-defined state; the VMM may inject a guest-visible fault, populate/mutate a mapping according to policy, or terminate the VM.

TLB composition and invalidation

Walking guest page tables can itself require GPA-to-HPA translations for each page-table memory reference. Hardware therefore caches combined translations in TLB structures and may cache intermediate walk results. Changing either stage creates invalidation obligations scoped by virtual address, guest identifier, address-space identifier, and processor.

A stale combined TLB entry after second-stage revocation can bypass the new VMM policy until invalidated. Correctness requires this ordering pattern:

```
remove or restrict second-stage PTE
perform architecture-required synchronization
invalidate relevant guest/combined translations on every vCPU that may
cache them
wait for required completion
only then reuse the physical page for another protection domain
```

The exact fence and shootdown instructions are architecture-specific. Reusing the page before remote invalidation completes is a cross-VM confidentiality failure, not a performance defect.

Transfer checkpoint: ballooning and stale translation

A VMM reclaims guest frame `0x2a` through a balloon driver and immediately assigns its HPA to another VM. One vCPU still has a combined GVA-to-HPA TLB entry. Construct the violation and name the missing synchronization.

The old vCPU issues a load to GVA `0x4123`; its combined TLB entry bypasses both page-table walks and reaches the reassigned HPA. It reads the other VM's data. The VMM must revoke the second-stage mapping and complete architecture-required invalidation on every vCPU that can retain the translation before reassignment. Guest cooperation by the balloon driver is not sufficient evidence that hardware caches are empty.

CPU state, time, and interrupts

WARNING

Virtualizing a CPU requires virtualizing time and interrupt delivery together: if a vCPU is descheduled by the host while holding a guest spinlock or waiting for a timer deadline, injecting timer interrupts without accounting for stolen host time can trigger false guest lockup panics or livelock spin-waiters.

A vCPU has more state than general registers: control registers, pending exceptions, virtual interrupt controller state, timers, debug state, and architecture features. The VMM chooses which features to expose and must preserve migration and scheduling invariants for that virtual contract.

Virtual time has at least three meanings:

- execution time consumed while the vCPU runs;
- wall or monotonic time expected by guest software; and
- virtual timer deadlines delivered despite vCPU descheduling.

Pausing a VM while exposing an unadjusted wall clock can produce a large apparent jump. Freezing wall time can violate external protocol expectations. The policy is part of the virtual platform contract.

Interrupts add a three-way race: a host interrupt arrives, a virtual interrupt becomes pending, and the guest changes its interrupt-enable state. Hardware/VMM state must ensure no required event is lost and no interrupt is injected at a guest-forbidden point. Coalescing can improve throughput while worsening latency.

Worked scheduling trace: pending timer

Guest timer deadline is virtual time 100 ms. At virtual time 96 ms, the host deschedules the vCPU for 20 ms of host wall time.

- Under execution-time semantics, virtual time may remain 96 ms while descheduled; the timer fires after four more milliseconds of vCPU execution.
- Under wall-correlated semantics, the deadline passes during descheduling; the VMM should mark the timer pending and inject it when the vCPU can receive interrupts.

Neither behavior is universally correct. A real-time guest, benchmark, and general-purpose VM can require different contracts. Reporting only "timer latency 20 ms" without defining the clock is uninterpretable.

A cost model that refuses to impersonate data

For a fixed workload, a diagnostic decomposition is

$$T_{vm} = T_{direct} + N_{exit}C_{exit} + N_{walk}C_{walk} + C_{emulation} + C_{schedule} + C_{noise}.$$

This is an accounting model. It predicts questions:

- Does reducing exits lower time when exit handling dominates?
- Do huge pages or better locality reduce second-stage walk cost?
- Does batching device notifications trade completion latency for fewer exits?
- Does vCPU overcommit add host scheduling delay?

The symbols are not constants transferable across processors. `N_exit` must be counted for the actual workload; costs vary by exit reason, caches, mitigations, VMM path, device model, and hardware. Measure baseline and changed system on a pinned environment. A modeled reduction from 10,000 to 1,000 exits is not a measured 10× speedup.

Quantitative checkpoint

An illustrative model assigns 1.2 µs to each of 20,000 exits and 0.15 µs to each of 40,000 expensive translation events; direct execution and all other costs total 70 ms. Compute modeled total. If an optimization halves exits but increases translation events to 100,000, compute the new modeled total and state what cannot be concluded.

Baseline overhead is $20{,}000(1.2\ \mu s) + 40{,}000(0.15\ \mu s) = 24 + 6 = 30$ ms, so modeled total is 100 ms. Changed overhead is $10{,}000(1.2\ \mu s) + 100{,}000(0.15\ \mu s) = 12 + 15 = 27$ ms, so modeled total is 97 ms.

The model predicts a 3% reduction under fixed costs. It does not establish real exit costs, causal independence, cache effects, tail latency, throughput under contention, or statistical significance. Measurement could reverse the result if the new translation events alter locality or the assumed event costs are wrong.

Isolation is a dependency chain

WARNING

Second-stage page tables (G-stage / EPT) isolate guest memory accesses from the host, but the VM isolation boundary is only as strong as the VMM's exit handlers, userspace device emulators, and IOMMU DMA mappings: a single out-of-bounds buffer write in an emulated Virtio or PCI device compromises the VMM process despite hardware page-table enforcement.

A guest-kernel compromise gives the attacker guest-kernel authority. Containment then depends on:

1. CPU virtualization controls and second-stage translation;
2. VMM exit validation and memory-management code;
3. virtual device models, often a large parser surface;
4. host kernel and process isolation when device models run in userspace;

5. IOMMU and interrupt remapping for assigned devices;
6. firmware and hardware correctness; and
7. the accepted side-channel model.

A VM has a separate guest kernel, unlike a container. It does not remove the VMM or hardware from the trusted computing base. Shared caches, speculative state, memory deduplication, shared devices, and host resource exhaustion can create channels or denial of service even when architectural memory isolation holds.

> ***Counterexample.*** *"The guest cannot map host memory through its page tables" proves only that the guest-controlled translation stage is subordinate. It does not prove that an emulated device lacks an out-of-bounds write into VMM memory.*

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 7.2.1: Evaluating the Popek–Goldberg Virtualizability Criterion

Verification application

State the classical Popek–Goldberg theorem regarding instruction set virtualizability. Explain why early x86 architectures (e.g., IA-32 with non-faulting `POPF`, `PUSHF`, and `SGDT` in user mode) were not classically virtualizable without binary translation or paravirtualization.

Sensitive instructions must be a subset of privileged instructions.

- **Popek–Goldberg Virtualizability Theorem (1974):** An architecture is strictly virtualizable under trap-and-emulate if and only if all **sensitive instructions** are a subset of **privileged instructions**:

$$I_{\text{sensitive}} \subseteq I_{\text{privileged}}.$$

  - *Sensitive instructions:* Instructions that manipulate hardware configuration (control registers, interrupt masks, MMU state) or behave differently depending on the execution privilege mode.
  - *Privileged instructions:* Instructions that trap to the supervisor/monitor when executed in user mode.

- **The x86 virtualization hole:** On early 32-bit x86, approximately 17 sensitive instructions failed this criterion. For instance, `POPF` (Push/Pop to Interrupt Flags) silently ignored modifications to the interrupt flag when executed in unprivileged Ring 1 without trapping. `SGDT` (Store Global Descriptor Table) allowed unprivileged code to read the CPU's global descriptor table base address without trapping, exposing host hardware state. Because these instructions failed silently instead of trapping, a VMM could not intercept and emulate them, necessitating expensive dynamic binary translation (pioneered by VMware) or paravirtualization (Xen) until hardware virtualization extensions (Intel VT-x, AMD-V) were introduced.

Exercise 7.2.2: Calculating Address Translation Overhead in Two-Stage Paging

Construction application

Consider hardware nested paging (two-stage address translation) on a 64-bit architecture where both guest and host use 4-level page tables with standard 4 KiB pages:

$$\text{Guest: GVA} \to \text{GPA (4 levels)}, \quad \text{Host: GPA} \to \text{HPA (4 levels)}.$$

1. In the worst case where the TLB is completely cold (no GVA or GPA translations cached), compute the total number of physical memory accesses required to translate a single guest virtual address.
2. If host physical memory access latency is $t_{\mathrm{mem}} = 50\ \mathrm{ns}$, calculate the total translation latency.
3. Explain how configuring 1 GiB host superpages for the second stage drastically reduces this page-walk multiplier.

Each guest page table access is a GPA that itself requires a full 4–level host translation walk.

1. **Total physical accesses:** A 4–level guest walk accesses 4 guest page table entries (level 4, 3, 2, 1) plus the final guest data access (1 access). However, every guest page table page resides in guest-physical memory (GPA), which itself must be translated through the 4–level host page table (HPA).
   - Guest Level 4 PTE lookup: requires 4 host memory accesses + 1 memory read = 5 accesses.
   - Guest Level 3 PTE lookup: requires 4 host accesses + 1 read = 5 accesses.
   - Guest Level 2 PTE lookup: requires 4 host accesses + 1 read = 5 accesses.
   - Guest Level 1 PTE lookup: requires 4 host accesses + 1 read = 5 accesses.
   - Final Guest Data lookup: requires 4 host accesses to resolve the data HPA.

     Total physical memory accesses:

$$4 \times (4 + 1) + 4 = 4 \times 5 + 4 = 24 \text{ memory accesses.}$$

   In general, for n-level guest and m-level host tables, total accesses $= n \cdot (m + 1) + m = 4(5) + 4 = 24$.
2. **Translation latency:**

$$T_{\text{walk}} = 24 \times 50 \text{ ns} = 1200 \text{ ns} = 1.2 \ \mu\text{s}.$$

3. **Superpage reduction:** If the host maps guest memory with 1 GiB superpages (level 2 leaves), the host walk requires only 2 levels ($m = 2$) instead of 4. Total accesses drop to $4 \times (2 + 1) + 2 = 12 + 2 = 14$ accesses, cutting memory bus latency by more than 40%.

Exercise 7.2.3: Comparing Emulated MMIO Device Exits with Virtio Paravirtualization

Description application

Contrast the runtime overhead of an emulated legacy IDE controller with a paravirtualized Virtio block device. Detail why Virtio reduces VM exits by orders of magnitude during high-throughput I/O.

Legacy devices trap on every register access; Virtio batches requests into shared ring buffers with single doorbells.

- **Legacy hardware emulation overhead:** An emulated IDE or realtek network controller reproduces physical register-level hardware behavior. Every read and write to an I/O port or MMIO register triggers a VM exit. For a single block transfer:
  1. The guest issues 8–16 register writes to setup sector, cylinder, head, and count (8–16 VM exits).
  2. If using programmed I/O, the CPU reads data words through port `0x1F0`, generating 256 VM exits per 512-byte sector. Each VM exit costs $1000–3000\ \text{CPU cycles}$ for hardware context saving and VMM trap handling, causing massive CPU overhead.
- **Virtio paravirtualization efficiency:** Virtio uses cooperative shared-memory queues (`virtqueues`) agreed between guest and hypervisor:
  1. The guest populates descriptors in shared RAM without any VM exits.
  2. Multiple I/O requests (e.g., 64 disk blocks) are enqueued in batch.
  3. The guest writes to a single doorbell MMIO register once to notify the hypervisor (1 VM exit for the entire batch).
  4. The hypervisor processes all 64 requests and injects a single virtual interrupt. By amortizing the exit cost across large batches, Virtio achieves near-native I/O throughput.

Exercise 7.2.4: Formalizing Virtual Time and Clock Skew Under vCPU Descheduling

Formalisation transfer

Let t_{host} represent host monotonic wall time, and let $\tau(t_{\text{host}})$ represent the virtual time experienced by a vCPU.

1. Formulate the relationship between τ and t_{host} when a vCPU is descheduled by the host scheduler during time interval $[t_1, t_2]$.
2. Explain why a guest operating system using an unadjusted TSC (Time Stamp Counter) experiences spurious network timeouts or incorrect TCP retransmissions after a descheduling event.

During descheduling, tau stops while t_host advances; TSC jumps forward relative to instructions executed.

1. **Virtual time formulation:** Let $S(t) \in \{0, 1\}$ indicate whether the vCPU thread is actively scheduled on a physical CPU core at host time t. Virtual execution time τ is the integral of scheduled execution:

$$\tau(t_{\text{host}}) = \int_0^{t_{\text{host}}} S(u)\, du.$$

If the vCPU is descheduled during $[t_1, t_2]$ ($S(u) = 0$ for $u \in [t_1, t_2]$):

$$\tau(t_2) = \tau(t_1), \quad \text{while } t_{\text{host}} \text{ has advanced by } \Delta t = t_2 - t_1.$$

2. **Clock skew and protocol failure:** If the guest reads the raw physical hardware TSC, it observes an abrupt leap of $\Delta t = 200$ ms across zero guest instructions.
   - TCP retransmission timers fire spuriously, assuming round-trip packet drop.
   - Internal scheduler runqueues observe apparent massive task CPU starvation.
   - Watchdog timers (e.g., lockup detectors) trigger false kernel panics. To prevent this, paravirtual clock protocols (such as `kvm-clock`) publish explicit stolen-time offsets and paused-state notifications so the guest adjusts time-accounting structures.

Exercise 7.2.5: Proving Isolation Boundaries Against Device-Model Buffer Overflows

Proof synthesis

Suppose a guest kernel exploits a buffer overflow in an emulated USB device model running within the hypervisor userspace process (e.g., QEMU).

1. Prove that hardware nested paging (EPT/NPT) cannot prevent the compromised device model from writing to host memory.

2. Formulate the principle of privilege separation applied to VMM architecture (e.g., modern micro-VMMs like Cloud Hypervisor or Firecracker) to contain device model compromises.

EPT restricts guest CPU execution; device models run with host process privileges on the host.

1. **Failure of EPT against userspace device bugs:** Hardware nested paging (EPT/NPT) applies strictly to guest-mode memory accesses executed by the vCPU hardware pipeline. When a device model runs in the host hypervisor process:
   - The device model executes in host user mode (or host kernel mode) on host physical CPU cores.
   - Its memory accesses are governed by host page tables, NOT guest EPT page tables.
   - If an adversarial guest triggers an out-of-bounds write inside the QEMU device emulator, the exploit writes directly into host process heap or stack space.
   - EPT hardware does not intercept these host-level stores. The attacker achieves arbitrary code execution within the host VMM process, breaking out of the guest VM.

2. **Architectural privilege separation defense:** To contain device vulnerabilities:
   - **Minimalist attack surface:** Eliminate legacy emulated devices entirely, replacing them with minimal Virtio implementations written in memory-safe languages (e.g., Rust in Firecracker/crosvm).
   - **Process sandboxing:** Run each device backend in a separate, unprivileged host process confined with `seccomp-bpf` (forbidding all unauthorized host syscalls), separate mount/PID namespaces, and read-only host filesystem views.
   - **Restricted shared memory:** Grant the device process access only to the guest's allocated RAM via explicit file descriptor passing, preventing access to the hypervisor's control structures.

References & Further Reading

- Popek and Goldberg, “Formal Requirements for Virtualizable Third Generation Architectures” — primary VMM model and virtualizability condition.
- RISC-V Ratified Specifications, Hypervisor Extension — guest privilege, VS-stage, G-stage, and hypervisor control semantics.
- Intel 64 and IA-32 Software Developer’s Manuals — official VMX, memory, interrupt, and protection architecture reference.
- Linux kernel documentation, KVM — official KVM userspace API and architecture documentation.
- Linux kernel documentation, KVM API — virtual machine, vCPU, memory, and device control interface.
- seL4 Reference Manual — a contrasting capability and virtualization-support boundary where applicable.

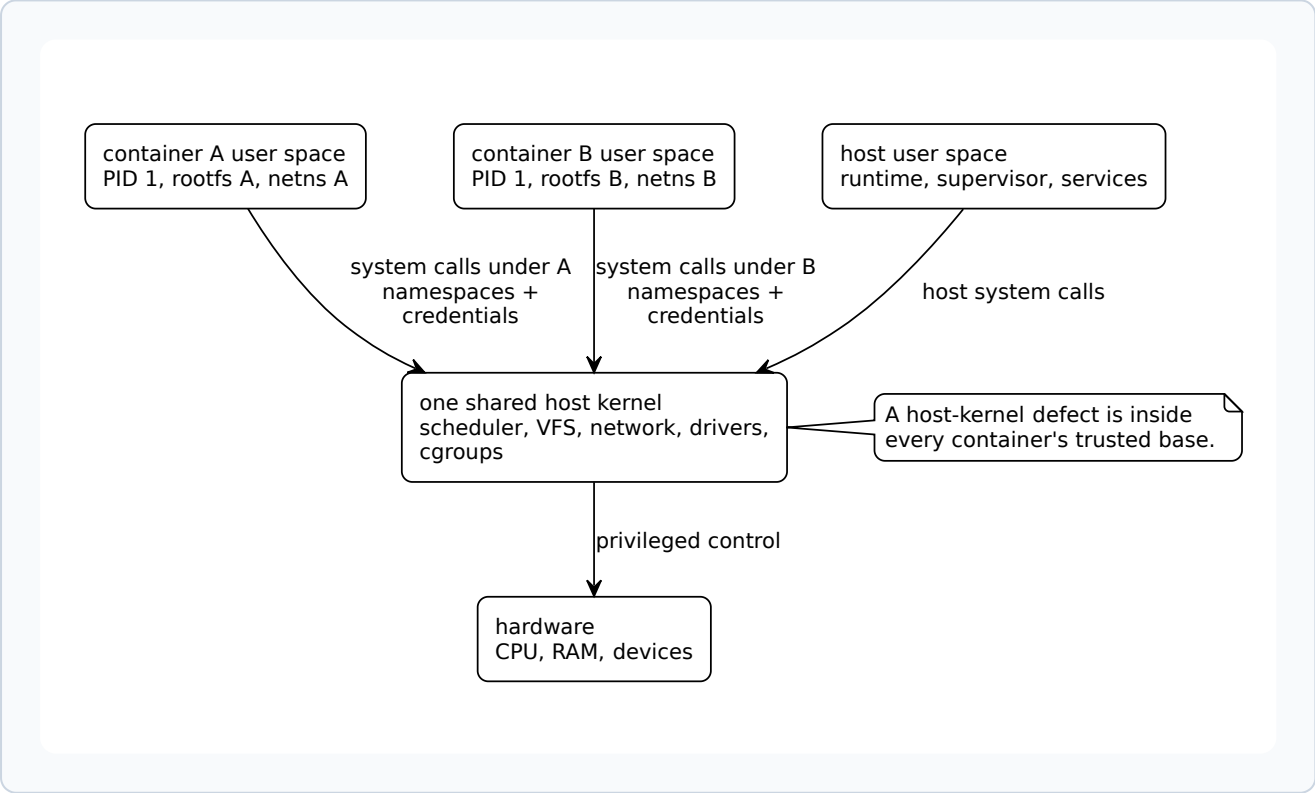
SECTION DETAIL

Containers and Resource Control

Two services each report PID 1 and each sees `/` as its own image root. One service forks until the host cannot create another process. Another triggers a vulnerability in the filesystem parser and executes in kernel context. The first failure is a missing resource limit; the second crosses every container boundary that relies on that kernel.

Names and resource accounting create convincing private views, but they do not create another kernel. A Linux container is a set of ordinary host tasks whose namespace memberships, cgroup memberships, credentials, filesystem view, and security policy are configured to constrain what those tasks can name and consume.

The boundary stack



The runtime creates and joins kernel objects; it is not on the path of every later system call. Enforcement resides primarily in the host kernel and configured security modules. The image supplies user-space bytes. It does not supply a guest kernel.

Contrast a virtual machine:

```
container: application -> shared host kernel -> hardware
VM:        application -> guest kernel -> VMM/hypervisor -> hardware
```

A VM's guest kernel is inside the VM boundary, but the VMM and hardware remain trusted dependencies. A container's kernel is the host kernel itself.

NAMESPACES, CGROUPS, AND CONTAINER BOUNDARIES

Evaluate each statement regarding Linux namespaces, cgroups v2, and container isolation models:

Answer: Without cgroup process limits, a fork bomb in a private PID namespace can exhaust host kernel task capacity.; In cgroups v2, memory.high acts as a throttle triggering background reclaim before hard memory.max termination.

PID namespaces isolate only process identifier visibility, so without pids.max, forks exhaust global kernel task tables. In cgroups v2, memory.high throttles processes and triggers asynchronous reclaim while memory.max triggers OOM kills. Classic chroot isolates only pathname lookups, and user namespaces map UIDs on the shared host kernel rather than running a guest kernel.

Namespaces alter views and identities

A namespace wraps a class of global resources so member tasks observe a namespace-specific instance or mapping. It does not grant arbitrary permission over the underlying host resource.

namespace	selected virtualized view	important boundary
PID	process identifiers and visible process hierarchy	host still schedules the same tasks; namespace PID 1 has special lifecycle duties
mount	mount table and resulting filesystem tree	filesystem implementations execute in the shared kernel
network	interfaces, routes, ports, protocol-stack view	packets and drivers still cross host-kernel code
IPC	System V IPC and POSIX message-queue namespace	does not isolate every file-descriptor IPC channel
UTS	hostname and domain name	presentation identity, not authority boundary by itself
user	UID/GID mappings and namespaced capabilities	UID 0 inside may map to an unprivi-leged host UID
cgroup	visible cgroup root	resource enforcement comes from controller state, not this view alone
time	selected boot/monotonic clock offsets	not a separate scheduler or hardware clock

A task is a member of one namespace of each type. Namespace lifetimes can outlive their last member when file descriptors or bind-mounted namespace handles retain references.

Worked trace: one task, two PID and UID views

Host state:

```
host PID: 4242
host UID: 100000
child user namespace mapping: inside UIDs 0..65535 -> host UIDs
100000..165535
child PID namespace: task is namespace init
```

Inside the container, the task sees PID 1 and UID 0. From the initial host namespaces, an administrator sees PID 4242 and UID 100000. A file created with inside UID 0 is owned by host UID 100000 under this mapping.

“Root in the container” therefore describes authority evaluated in a user-namespace context. Operations reserved to the initial user namespace—such as loading arbitrary host kernel modules under the relevant Linux policy—do not become permitted merely because the task’s inside UID is zero. Conversely, namespaced root can still exercise substantial authority over objects owned by that user namespace and can reach host-kernel attack surface through allowed system calls.

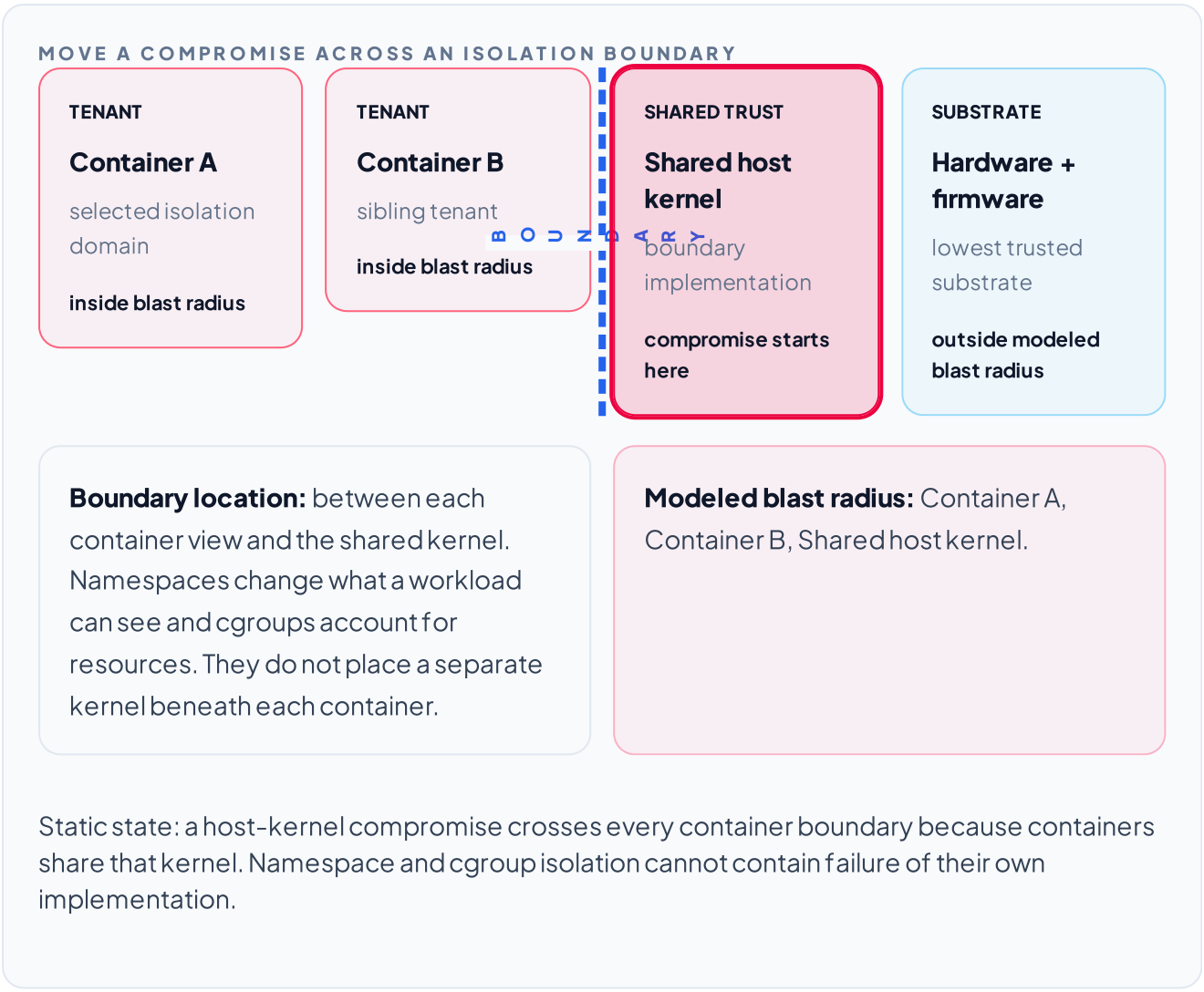
Application checkpoint: `/proc` shows the wrong tasks

A process creates a PID namespace but reuses a `/proc` mount created in the parent namespace. Inside, `ps` lists host-visible PIDs rather than the intended child view. Explain the mismatch and repair.

The process joined a new PID-numbering domain, but the mounted procfs instance reflects the PID namespace of the process that mounted it. Create or use an appropriate mount namespace and mount a fresh procfs from the child PID namespace. A PID namespace alone does not rewrite bytes supplied by an already mounted procfs view.

Predict the blast radius before moving the control

A hostile process has already obtained arbitrary code execution in the **host kernel** through an allowed filesystem system call. Predict which container namespace, cgroup, or seccomp rule can still mediate that code as an untrusted user process.



For the stated compromise, none of those user-process controls remains a sound boundary: the code executes inside the mechanism that implements them. This shared-kernel failure mode is a concrete counterexample to treating a container as a hardware-isolated virtual machine. The interaction is a trust-dependency model, not an exploitability score. Its four boxes represent the selected tenant, a peer tenant, their shared implementation, and hardware/firmware; red should be read as "this selected layer or a relied-on lower layer is no longer trustworthy," not as a measured probability or a complete attack graph.

Transfer after changing the boundary

Place the workload in a VM. A guest-kernel compromise occurs, while the VMM, its device models, second-stage translation, IOMMU configuration, firmware, and hardware remain correct under the threat model. What changes? Then move the compromise into the VMM.

The guest-kernel compromise controls that guest and its applications, but second-stage translation and the VMM can contain architectural access to host and other-VM memory under the assumptions. If the VMM is compromised, the layer responsible for VM isolation is lost; guest boundaries cannot constrain it. The stronger first result depends on the VMM/hardware assumptions and does not prove absence of side channels or resource denial.

Cgroups control resources, not names

Cgroup v2 organizes tasks in a hierarchy. Controllers distribute or limit resources among children. Namespace membership answers "what identifiers and objects can this task name?" Cgroup membership answers "how is this task's resource use accounted and constrained?" A secure deployment usually needs both.

CPU weights are work-conserving shares

Assume two sibling cgroups A and B are continuously runnable under one parent and no other constraints apply. With `cpu.weight` values 100 and 300, the idealized share is

$$share_A = \frac{100}{100 + 300} = 0.25, \qquad share_B = 0.75.$$

If the parent receives four cores for a 100 ms interval, its capacity is 400 core-ms. The model allocates about 100 core-ms to A and 300 to B while both can use them. If A is idle, a work-conserving scheduler may let B consume the available capacity rather than reserve 25% unused.

This is a policy model, not an exact short-window deadline guarantee. Runnable patterns, CPU affinity, per-CPU queues, priorities, bandwidth limits, topology, throttling, and measurement windows affect observed service. A weight does not promise a maximum latency.

Memory and process limits define overload states

For a batch worker cgroup:

```
memory.high = 384 MiB
memory.max  = 512 MiB
pids.max    = 128
```

`memory.high` is a pressure boundary that drives reclaim/throttling behavior rather than an immediate hard allocation failure in the ordinary model. `memory.max` is the hard containment boundary; if usage cannot be reduced, the kernel invokes cgroup-scoped out-of-memory handling according to controller policy. `pids.max` rejects creation when the task count would exceed the configured limit.

Limits interact. Reclaim can consume CPU and produce I/O; a process unable to fork may fail a supervisor protocol; killing one task may leave shared state inconsistent. Resource isolation needs an application failure contract, not only numeric files.

Worked overload trace: decompression service

Container A expands an adversarial archive. Container B serves latency-sensitive requests.

time	A state	kernel/controller action	impact to test
t_0	300 MiB, 20 tasks	below configured boundaries	baseline throughput and B latency
t_1	crosses 384 MiB	A experiences reclaim/throttling pressure	A throughput falls; host I/O may rise
t_2	approaches 512 MiB with unreclaimable pages	allocations stall/fail or cgroup OOM policy acts	verify victim scope and supervisor response
t_3	tries task 129	creation rejected by `pids.max`	verify bounded error, no host-wide PID exhaustion

The target claim is not "B is unaffected." Shared last-level cache, memory bandwidth, storage, kernel locks, and unaccounted resources can still interfere. Measure B's tail latency and host pressure, then add controls or admission policy for observed channels.

Quantitative checkpoint: idle sibling

A and B have equal CPU weights. A is runnable for only 20 ms of a 100 ms interval; B is runnable throughout. Under an ideal work-conserving model on one core, how much CPU can B receive? Why is "exactly 50 ms" wrong?

During the 20 ms overlap, equal weights suggest roughly 10 ms each. During the remaining 80 ms, B can use the idle capacity. B therefore can receive about 90 ms, not a fixed 50 ms reservation. Exact short-window service depends on scheduler details and workload timing; the calculation states the simplified model. A hard bandwidth cap would be a different controller mechanism.

Layer the security policy

No single switch defines a container sandbox. A defensible policy composes:

1. **user namespace and credentials** — map container identities to least host authority;
2. **mount namespace and root filesystem** — expose required paths, make immutable inputs read-only, avoid unnecessary host mounts;
3. **capability sets** — remove broad Linux privilege units, especially from the bounding and ambient paths where applicable;
4. **no-new-privileges and exec policy** — prevent unexpected privilege gain across execution;
5. **seccomp filtering** — reduce reachable syscall surface for the actual architecture and workload;
6. **LSM/mandatory policy** — constrain object operations beyond discretionary mode bits where required;
7. **device policy** — do not expose host devices whose `ioctl` or DMA authority exceeds the task;
8. **cgroup limits** — bound CPU, memory, process, and I/O failure domains; and
9. **runtime/supervisor policy** — validate configuration, retain minimal authority, and handle abnormal exit.

Seccomp filters inspect syscall number, architecture, and argument words under defined constraints. They cannot dereference arbitrary user pointers in the in-kernel filter. They reduce exposed kernel entry points; they do not validate application-level information flow, remove inherited descriptors, or repair a vulnerable allowed syscall.

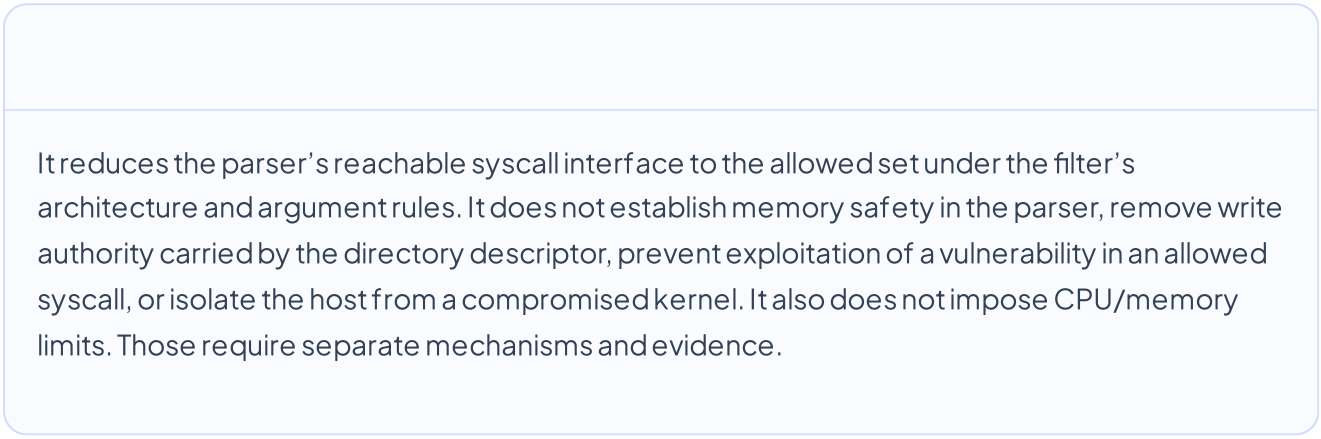
Worked threat trace: inherited host descriptor

A runtime opens `/var/run/admin.sock`, then starts a container without closing or marking the descriptor close-on-exec. The container has a private mount namespace and cannot name that path. It nevertheless inherits the connected descriptor.

The mount namespace removed a pathname, not the open-reference edge. If `send` is allowed, the process may use the descriptor. Repair requires an explicit descriptor allowlist/close sweep at process creation and least-authority supervisor design. Adding another bind-mount rule cannot revoke an inherited object reference.

Boundary checkpoint: seccomp-only sandbox

A parser receives untrusted images. Its filter allows `read`, `mmap`, `munmap`, `futex`, `close`, and `exit`; it shares the host kernel and has a writable descriptor to a tenant directory. State one property the filter supports and three it does not establish.



It reduces the parser's reachable syscall interface to the allowed set under the filter's architecture and argument rules. It does not establish memory safety in the parser, remove write authority carried by the directory descriptor, prevent exploitation of a vulnerability in an allowed syscall, or isolate the host from a compromised kernel. It also does not impose CPU/memory limits. Those require separate mechanisms and evidence.

Containers are not resource guarantees by default

WARNING

Because all containers on a host execute as tasks inside a single shared kernel, namespace visibility, cgroup accounting, and syscall filtering are orthogonal controls: omitting any one leaves an unmediated resource or authority surface, and a vulnerability in any reachable kernel path compromises the host and every co-tenant container.

Failure modes of orthogonal container mechanisms:

Mechanism	What it isolates	What it leaves unisolated (failure mode)
`chroot`	Pathname lookup root directory	Supplies no PID, IPC, network, or resource isolation; root processes can escape via file-descriptor `fchdir`
Namespaces alone	Identifier views (PIDs, mounts, net interfaces)	Provides zero resource bounds; an unconstrained fork bomb or anonymous allocation exhausts the host
Cgroups v2 alone	Resource consumption (CPU shares, memory bytes)	Provides zero naming isolation; processes see and signal arbitrary host PIDs and mounts
User namespaces alone	Maps container UID 0 to unprivileged host UID	Container processes still possess namespaced capabilities and expose reachable kernel attack surfaces
Seccomp BPF alone	Restricts permitted syscall numbers and registers	Cannot validate deep pointer payloads, unmediated open descriptors, or CPU/memory quotas

---

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 7.3.1: Process Namespace Hierarchies and PID 1 Orphan Reaping

Verification application

In a Linux container created with `CLONE_NEWPID`, the entrypoint process is assigned virtual PID 1 inside the namespace while holding PID 14,200 on the host.

1. Explain the special lifecycle responsibilities of PID 1 inside a PID namespace regarding orphaned child processes.
2. What occurs if PID 1 inside the container exits or is terminated by `SIGKILL`?

PID 1 must reap orphaned children; terminating PID 1 destroys all namespace members.

1. **Orphan reaping duties:** In Unix systems, when a parent process terminates before its children, those children are automatically reparented to PID 1 (`init`) of the containing PID namespace. PID 1 must continuously call `waitpid()` in a loop to reap zombies and collect their exit status codes. If PID 1 does not implement orphan reaping (e.g., if a naive application binary runs directly as container PID 1), zombie processes accumulate in the host kernel's process table, eventually exhausting host process slots (`pids.max`).
2. **Termination cascade:** POSIX and Linux specify that if PID 1 in a PID namespace terminates for any reason, the kernel automatically sends `SIGKILL` to all remaining processes belonging to that PID namespace. The entire container process tree is dismantled, and the PID namespace is destroyed.

Exercise 7.3.2: Sizing cgroup v2 Memory and CPU Controller Quotas

Construction application

A container group is configured under `/sys/fs/cgroup/build_worker` :

- Hard memory limit: 2 GiB ($2{,}147{,}483{,}648$ bytes).
- Soft throttle threshold: 1.8 GiB ($1{,}932{,}735{,}283$ bytes).
- CPU bandwidth: capped at 2.5 CPU cores per 100 ms scheduling period.

1. State the exact cgroup v2 controller filenames and parameter values required to enforce these three constraints.
2. Contrast the kernel reaction when consumption crosses the soft throttle threshold versus the hard limit.

Use memory.high for throttle, memory.max for OOM kill, and cpu.max for quota and period.

1. **Cgroup v2 configuration files:**
   - Soft memory threshold: `echo 1932735283 > /sys/fs/cgroup/build_worker/memory.high`
   - Hard memory ceiling: `echo 2147483648 > /sys/fs/cgroup/build_worker/memory.max`
   - CPU bandwidth quota ($2.5 \times 100{,}000\ \mu\text{s} = 250{,}000\ \mu\text{s}$ per $100{,}000\ \mu\text{s}$ period): `echo "250000 100000" > /sys/fs/cgroup/build_worker/cpu.max`
2. **Enforcement behavior:**
   - **Crossing `memory.high`:** The kernel does not kill the process. Instead, it throttles the container's threads (delaying syscall return) and activates asynchronous direct page reclaim to force memory usage back below 1.8 GiB without abrupt process termination.
   - **Crossing `memory.max`:** The kernel executes synchronous reclaim. If reclaim fails to free memory below 2 GiB, the kernel invokes the cgroup-scoped Out-Of-Memory (OOM) killer to terminate one or more container processes (`oom_score`).

Exercise 7.3.3: Mapping Container Root to Unprivileged Host UIDs

Construction transfer

A container uses user namespaces (`CLONE_NEWUSER`). The container runtime configures `/proc/[pid]/uid_map` with: `0 100000 65536`

1. Explain the meaning of the three numbers in this UID mapping entry.

2. What host UID owns a file created inside the container by `root` (UID 0)?
3. If container root executes `mknod /dev/sda1 b 8 1`, explain why the host kernel rejects the system call even though the process has effective `UID 0` and `CAP_MKNOD` in its namespace.

The mapping format is: container_start host_start length. Device node creation requires global root authority.

1. **Mapping specification:**
   - `0`: The starting UID inside the container namespace.
   - `100000`: The starting host UID on the underlying system.
   - `65536`: The contiguous range of mapped UIDs. Container UID $k \in [0, 65535]$ maps to host UID $100000 + k$.
2. **File creation ownership:** When container `root` (UID 0) creates a file, the host kernel stores the physical on-disk inode owner as host UID 100,000. On the host, the file is owned by unprivileged user 100,000, possessing zero superuser access to host resources.
3. **Rejection of `mknod`:** In Linux user namespaces, capabilities are namespaced. The container process possesses `CAP_MKNOD` only within its private user namespace. Creating real block device nodes exposes raw host disk storage; therefore, the kernel's VFS explicitly requires `CAP_MKNOD` in the *initial root namespace* (host namespace). Since host UID 100,000 lacks global superuser capabilities, `mknod()` returns `-EPERM`.

Exercise 7.3.4: Proving Confinement Failure Under Inherited Host File Descriptors

Proof synthesis

A container is confined using a strict seccomp-BPF filter that allows only `read()`, `write()`, `close()`, and `exit()`. A bug in the container launcher leaves host file descriptor 3 open, connected to `/etc/shadow` with read permissions.

1. Prove that seccomp-BPF cannot prevent the sandboxed container process from reading password hashes through descriptor 3.
2. Specify the process creation invariant required to prevent descriptor leakage into containers.

Seccomp checks syscall numbers and register arguments, not whether an open descriptor refers to a sensitive file.

1. **Proof of confinement failure:**
   - Seccomp-BPF operates as an instruction filter on system call invocation: it inspects the system call number (`__NR_read`), architecture, and scalar register values (file descriptor integer `3` , user buffer pointer `buf` , length `count`).
   - Seccomp cannot inspect the underlying kernel object to which descriptor 3 points; it only observes the integer `3` .
   - Because `read()` is explicitly allowed by the policy filter, the seccomp filter returns `SECCOMP_RET_ALLOW` .
   - The kernel VFS executes `vfs_read(file_table[3], buf, count)` . Because the descriptor was opened on `/etc/shadow` before container creation, the read succeeds, disclosing confidential password hashes. Seccomp restricts syscall APIs, not ambient object capabilities carried by open descriptors.

2. **Process creation invariant:** The container runtime must enforce:

$$\forall \mathrm{fd} > 2, \quad \mathrm{close}(\mathrm{fd}) \vee \mathrm{fcntl}(\mathrm{fd}, \mathrm{F_SETFD}, \mathrm{FD_CLOEXEC})$$

   prior to invoking `execve()` , ensuring that only explicitly whitelisted standard I/O descriptors (0, 1, 2) cross the containment boundary.

Exercise 7.3.5: Evaluating the Shared-Kernel Threat Boundary in Multi-Tenant Containers

Description synthesis

Compare the isolation boundary of multi-tenant containers with hardware-isolated virtual machines. Explain why a single local kernel privilege escalation exploit (e.g., a vulnerability in an allowed ioctl, socket protocol, or eBPF verifier) breaks all co-tenant containers on the host.

Containers share a single kernel address space and monolithic execution path.

- **The shared-kernel vulnerability:** In containerized environments, every container executes system calls directly against the same running host kernel. There is only one kernel text, one page table mapping supervisor memory, and one set of kernel locks. If an attacker in Container A triggers an unpatched kernel bug (such as a use-after-free in `netfilter`, a race condition in `io_uring`, or an integer overflow in a filesystem driver):
  1. The exploit executes arbitrary shellcode in CPU supervisor/kernel mode.
  2. Once in supervisor mode, the exploit completely overrides all namespace data structures, cgroup accounting limits, and seccomp filters.
  3. The attacker can read and write all host physical memory, inspect processes in Container B, access host disks, and achieve full system takeover.
- **Contrast with Virtual Machines:** In virtual machines, the guest operating system kernel runs in a separate hardware virtualization mode (Guest Supervisor) below the hypervisor (Host Root). A guest kernel vulnerability gives the attacker control over only that guest's virtual machine. Escaping to the host requires finding and chaining a secondary hypervisor breakout exploit (e.g., in QEMU device emulation or KVM exit handling). VMs provide two independent layers of defense-in-depth.

References & Further Reading

- Linux man-pages, `namespaces(7)` — namespace types, APIs, handles, and lifetimes.
- Linux man-pages, `pid_namespaces(7)` — PID visibility, namespace init, and procfs coupling.
- Linux man-pages, `mount_namespaces(7)` — mount-table views and propagation restrictions.
- Linux man-pages, `user_namespaces(7)` — UID/GID mappings and namespaced capability scope.

- Linux kernel documentation, Control Group v2 — hierarchy, controllers, weights, limits, and resource-distribution models.
- Linux kernel documentation, Seccomp BPF — syscall-filter mechanism, requirements, and sandbox boundary.
- Open Container Initiative, Runtime Specification: Linux — runtime mapping for namespaces, cgroups, devices, and process configuration.
- Linux kernel documentation, Credentials in Linux — task credentials and Linux capability sets.

SECTION DETAIL

Multicore Scalability and Observability

A kernel service completes 1.0 million operations per second on one core, 3.4 million on four, 4.2 million on eight, and 4.1 million on sixteen. All sixteen cores are busy. The team concludes that the operation is "CPU-bound" and adds cores.

Busy cores do not imply useful parallel work. They may spin on one lock, migrate a cache line, retry an atomic operation, handle interrupts, or run instrumentation. Scalability analysis needs a causal chain from workload to shared state to hardware event to observable delay.

Define the quantities before drawing a curve

For fixed work W:

- throughput $X(N)$ is completed operations per unit time on N cores;
- latency is elapsed time per operation, whose distribution includes tails;
- speedup is $S(N) = T(1)/T(N)$ for the same work and defined environment;
- efficiency is $E(N) = S(N)/N$;
- scalability is how a metric changes as a named resource or workload dimension changes;
- contention is delay or retry caused by competition for a shared resource; and
- serialization is work that the model permits only one participant to perform at once.

Strong scaling fixes total work. Weak scaling grows work with resources. A server saturation study may fix offered load, increase it to capacity, or keep per-core load fixed; those are different experiments.

Derive Amdahl's bound

Normalize one-core time to 1. If fraction s remains serial and fraction $1 - s$ parallelizes perfectly over N cores, then

$$T(N) = s + \frac{1 - s}{N},$$

so

$$S(N) = \frac{1}{s + (1 - s)/N}, \qquad \lim_{N \to \infty} S(N) = \frac{1}{s}.$$

This is an upper bound under the model. Real multicore costs can grow with N: lock handoff, coherence messages, remote memory, scheduler migration, interrupts, and queueing. Treating s as a universal constant can hide the mechanism being diagnosed.

Worked calculation: eight percent serialized

With $s = 0.08$ and $N = 16$:

$$S(16) = \frac{1}{0.08 + 0.92/16} = \frac{1}{0.1375} \approx 7.27,$$

$$E(16) = 7.27/16 \approx 45.5\%.$$

The asymptotic bound is $12.5\times$. If measured speedup is 5×, Amdahl's model does not prove that the serial fraction is exactly 16.7%; growing overhead can produce the same curve.

Retrieval checkpoint: invert the model carefully

A fixed-work experiment measures $S(8) = 4$. Under the ideal Amdahl model, solve for s. Then state why this inferred value is not yet a measured lock critical-section fraction.

Solve

$$\frac{1}{s + (1 - s)/8} = 4.$$

Thus $s + (1 - s)/8 = 1/4$, so $8s + 1 - s = 2, 7s = 1$, and $s = 1/7 \approx 14.3\%$. This is an **effective serial fraction in the model**. It can absorb load imbalance, memory stalls, scheduling, coherence, and other overhead. Only instrumentation tying time to a lock path can identify a lock critical-section fraction.

AMDAHL SCALABILITY BOUNDS AND MULTICORE CONTENTION

Evaluate each statement regarding Amdahl scalability bounds, lock contention, and multicore kernel behavior:

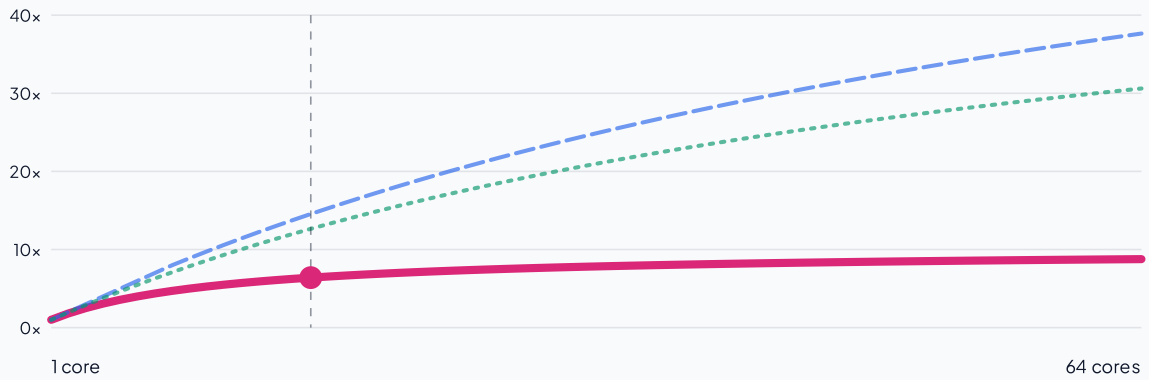
Answer: Amdahl model establishes that the maximum speedup is strictly bounded by the inverse of the serialized fraction.; Independent CPU cores writing to distinct variables located on the same cache line incur cache invalidation stalls.

Amdahl limits speedup to 1/s. False sharing occurs when independent variables occupy the same 64–byte cache line, causing inter-core bounce. High CPU utilization often reflects lock spin contention rather than productive parallel work, and RCU eliminates reader locks but writers still serialize via locks.

Predict the analytical curves

At 16 cores and 10% serialized critical work, predict the global-lock relative throughput and its infinite-core limit before changing any control. All designs will be divided by the **one-core global-lock** throughput, not by their own one-core result. Given a fixed RCU management/grace cost, predict whether the RCU curve may begin above, at, or below 1× on one core. Then predict its direction as the read share falls from 90% to 10%.

WATCH ONE LOCK FLATTEN A MULTICORE SPEEDUP CURVE



Static state: with 10% serialized work, one global lock yields 6.4× modeled speedup at 16 cores; adding cores cannot remove the serial bottleneck.

Normalize the one-core global-lock time to 1. With critical-work fraction f, read share r within that work, core count N, and $K = \min(8, N)$ active shards, the displayed relative-throughput hypotheses are

$$R_G(N) = \frac{1}{f + (1 - f)/N},$$

$$R_S(N) = \frac{1}{f/K + (1 - f)/N},$$

and

$$R_R(N) = \frac{1}{f(1 - r) + 0.08fr + \big((1 - f) + fr\big)/N}.$$

For the global lock, $R_G(16) = 1/(0.10 + 0.90/16) = 6.4$ and the limit is 10×. The RCU expression does **not** delete read-side critical work: fr moves into the parallel term. Updates $f(1 - r)$ remain serialized, and the illustrative $0.08fr$ management/grace term is fixed with respect to core count. Therefore $R_R(1) = 1/(1 + 0.08fr)$ can be slightly below the common 1× baseline. Lowering read share erodes the modeled RCU opportunity because more critical work becomes serialized update work. The sharded curve assumes balanced requests across at most eight shards.

The graph labels all three designs; color is supplementary. Every value is relative throughput against the same one-core global-lock baseline, so it is not a within-design speedup when the selected design is sharding or RCU. The 8% RCU term is an illustrative model assumption, not a measured grace-period cost; the sharded curve omits skew. None of the curves includes cache-coherence traffic, NUMA, allocator contention, interrupt placement, or correctness cost.

Transfer after breaking balance

Keep eight shards but send 90% of operations to one shard. Explain why the displayed sharded curve becomes optimistic and derive a lower-level bottleneck hypothesis.

The model divides critical work evenly by the number of active shards. Under 90% hot-shard traffic, one lock serializes most operations while seven locks are lightly used. Aggregate shard count is not effective parallelism. Hypotheses include hot-key distribution, a shared allocator beneath all shards, or one cache line containing multiple shard locks. Measure per-shard acquisition/wait distributions and address placement before attributing the curve to the high-level design alone.

Lock contention is a queueing mechanism

A contended lock has at least four times:

```
request -> wait/spin -> acquire -> critical work -> release/handoff
```

Reducing critical-section instruction count can help, but a lock's cost also includes cache-line ownership transfer and wakeup/scheduling. A spinlock is appropriate only when sleeping is forbidden or expected waits are shorter than sleep/wakeup cost under the stated preemption model. A mutex avoids burning CPU during long waits but can add scheduler transitions and convoy behavior.

Worked trace: one global inode-cache lock

Four cores repeatedly look up distinct inodes. The cache uses one lock for lookup, reference increment, and a slow eviction scan.

time	core 0	core 1	core 2	core 3	lock line
t_0	acquires; scans	requests	requests	requests	owned by core 0
t_1	releases	acquires	spins/waits	spins/waits	migrates to core 1
t_2	requests again	scans	waits	waits	remains core 1
t_3	waits behind earlier callers	releases	acquires	waits	migrates to core 2

Distinct inode identities do not help because lookup and eviction share one lock. Core count increases both queue length and ownership migration. Candidate redesigns include partitioning lookup buckets, separating eviction policy from fast lookup, per-CPU references with safe aggregation, or an RCU-protected read path. Each changes invariants and must be proven before performance testing.

Application checkpoint: shorten or split?

Profiling shows 70% of lock hold time is a linear scan for an eviction victim; 20% is hash lookup; 10% is reference update. Give one local optimization and one structural optimization. State a correctness risk for each.

A local optimization maintains an eviction data structure so victim selection does not scan the whole cache under the lock. Its risk is stale or inconsistent replacement metadata under concurrent hits. A structural optimization shards the identity map and separates global eviction coordination; its risk is duplicate cache identities, cross-shard lock cycles, or an eviction selecting an entry whose shard/reference state changed. “Use a faster lock” does not remove the long serialized scan.

False sharing without a logical lock

Cache coherence commonly tracks fixed-size lines. If core 0 increments counter A and core 1 increments adjacent counter B in the same line, each write requires ownership of the line even though the variables are logically independent.

```
64-byte cache line
+----------------------+----------------------+-------------------+
| counter[0] (core 0)  | counter[1] (core 1)  | unused / others   |
+----------------------+----------------------+-------------------+
```

The ownership trace alternates:

```
core 0 obtains write ownership -> invalidates core 1 copy -> writes A
core 1 obtains write ownership -> invalidates core 0 copy -> writes B
repeat
```

No source-level data race exists if each counter has one writer. Performance still collapses through coherence. Padding or per-CPU allocation can place counters on separate lines, followed by occasional aggregation. The trade-off is footprint, cache pressure, aggregation cost, and topology dependence.

Boundary checkpoint: padding did not help

After aligning counters to separate lines, throughput remains flat and lock-wait time remains high. What conclusion is justified?

The tested padding did not remove the dominant measured bottleneck under that workload. It does not prove false sharing was absent, because the event attribution or layout may be wrong; nor does it prove the global lock is causal solely because wait time correlates with flattening. Inspect addresses/layout, coherence or cache events where supported, lock traces, and a controlled lock-structure change.

Sharding changes the proof

LEMMA

If every operation that acquires multiple shard locks from a partitioned table $\{L_0, L_1, \ldots, L_{K-1}\}$ acquires its required subset in strictly ascending shard-index order $i_1 < i_2 < \cdots < i_m$, then the shard wait-for graph is a directed acyclic graph and no deadlock cycle can form among shard locks.

Partition a table by $h(key) \bmod K$. A single-key operation locks one shard. A two-key operation may need two. Canonical shard order prevents deadlock, but atomic multi-key semantics may still require a version, global sequence, or retry protocol.

The critical invariants are:

- each key has exactly one authoritative shard under the current mapping;
- resize or rehash does not make a key absent or duplicated to readers beyond the API's allowed state;
- operations touching multiple shards acquire them in one global order; and
- per-shard accounting sums to the global quantity without lost updates or double counting.

Worked cross-shard transfer

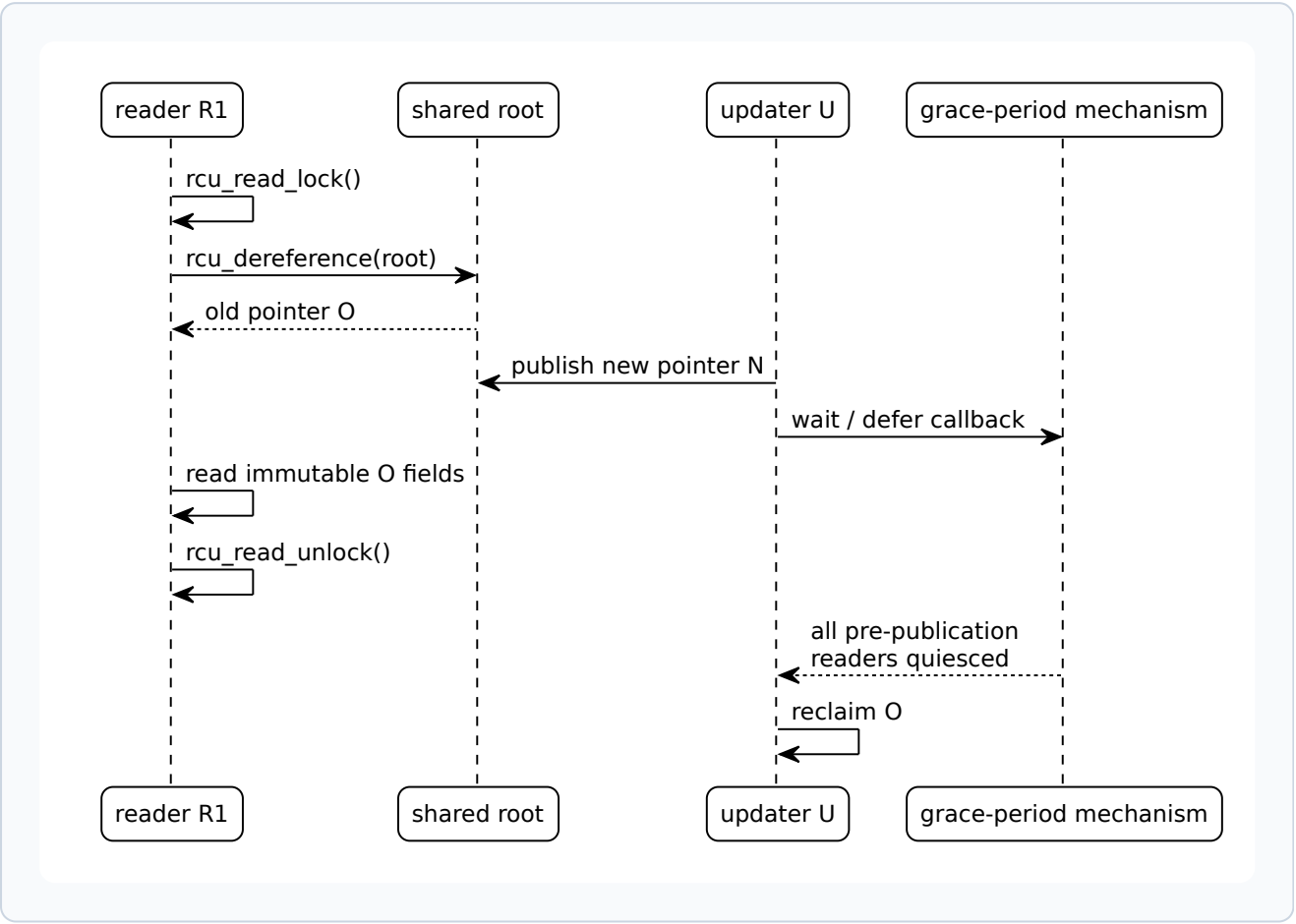
Moving one resource from key A in shard 7 to key B in shard 2 must atomically preserve total ownership.

1. Compute both shard identities before locking.
2. Lock shard 2, then shard 7 because $2 < 7$, independent of transfer direction.
3. Revalidate source ownership and destination eligibility.
4. Remove A and add B while both locks are held.
5. Release shard 7, then shard 2.

If the reverse transfer locks source first, concurrent opposite transfers can deadlock. If each operation releases one shard between remove and add, readers can observe missing or duplicated ownership.

RCU separates removal from reclamation

Read-copy update targets read-mostly structures. Readers enter an RCU read-side critical section, obtain a protected pointer, and traverse an immutable or carefully updated version. An updater publishes a new pointer or removes the old object, then waits for a grace period before reclaiming memory that pre-existing readers could still hold.



RCU does not mean "reads need no rules." The pointer must be published and dereferenced with the required primitives; referenced data must remain valid through the read-side critical section; updates requiring atomic multi-field visibility need an appropriate publication design; and reclamation cannot precede the grace period. Writers commonly still serialize with locks.

Worked failure: reclaim before grace

R1 reads pointer O and is preempted. U removes O from the shared list and immediately frees it. The allocator reuses O's memory for object X. R1 resumes and interprets X bytes as O. No stale-list link is needed; lifetime alone is violated. A grace period delays reuse until every reader that could have obtained O has completed.

Transfer checkpoint: RCU under write-heavy load

Why can an RCU conversion worsen a write-heavy workload even if read-side lock acquisition disappears?

Writers still need synchronization, allocate or copy versions, publish pointers with ordering, track callbacks, and delay reclamation. Frequent updates can build callback backlog, increase memory footprint, and spend more work copying than the removed reader lock cost. RCU is a correctness protocol with a read-mostly performance opportunity, not a universally faster lock replacement.

Observe mechanisms at the right resolution

Different evidence answers different questions:

evidence	strength	common ambiguity
aggregate counter	low overhead count/rate	loses temporal and call-path context
sampled event	attributes recurring events to code	sampling bias, skid, lost samples, multiplexing
tracepoint/event trace	ordered state transitions with timestamps	instrumentation overhead and finite buffers
lock statistics	acquisition, contention, wait/hold distributions	configuration overhead; correlation is not redesign proof
pressure signal	time workloads stall for CPU/memory/I/O	identifies resource pressure, not the exact code path
controlled intervention	tests a causal prediction	change may alter several mechanisms at once

Linux `perf_event_open` supports counting and sampled hardware, software, cache, tracepoint, and implementation-specific events. Availability and meaning vary by platform. `time_enabled` and `time_running` reveal multiplexing. Ftrace supplies function, function-graph, latency, and event tracing. Lockstat requires a configured kernel and reports lock behavior. PSI reports time in which some or all non-idle work is stalled on CPU, memory, or I/O pressure.

Illustrative commands—not portable promises—might include:

```
perf stat -r 15 -e cycles,instructions,context-switches,cpu-migrations
./workload
perf lock record ./workload
perf lock report
```

Record whether events exist, whether kernel access is permitted, whether counters were multiplexed, and what scope was measured. Raw cache-event names are processor-specific; a command succeeding does not make two machines' event semantics identical.

Worked evidence bundle

The following values are **illustrative data**, not results measured for LibreUni:

cores	throughput (Mop/s)	p99 (µs)	global-lock wait share	migrations/op
1	1.0	12	0.5%	0.00
2	1.9	14	3%	0.01
4	3.4	23	12%	0.03
8	4.2	71	35%	0.08
16	4.1	190	58%	0.15

The bundle supports a hypothesis: a global lock and associated migration become dominant as concurrency rises. It does not isolate whether hold time, handoff, scheduler placement, or a shared cache line beneath the lock causes the wait. Next evidence should include hold/wait distributions by call path, owner/waiter CPU, critical-section samples, and a controlled implementation that removes or partitions the suspected shared state while preserving workload and correctness tests.

Application checkpoint: choose a discriminating measurement

Two hypotheses fit the flat curve: H1, long critical sections queue on one lock; H2, per-CPU counters falsely share a line outside that lock. Choose one observation and one intervention that distinguish them.

Observe lock hold/wait distributions tied to call paths and inspect counter addresses/cache-line layout with coherence/cache events where the platform supports meaningful ones. Intervene separately: first remove the counters from shared lines without changing lock code; then, in another build, shorten or shard the lock path without changing counter layout. H1 predicts lock wait and throughput respond primarily to the lock intervention; H2 predicts the layout intervention reduces coherence traffic and improves scaling even if lock statistics are unchanged. Run correctness tests for both builds.

Reproducible scaling experiment

> **WARNING**
>
> Reporting single-run throughput or hardware PMU counts without checking `time_enabled` versus `time_running` (counter multiplexing) and tracing perturbation overhead can attribute measurement artifacts to kernel scalability.

A defensible experiment records:

1. source revision, configuration, compiler, kernel, firmware, and microcode where relevant;
2. processor topology, SMT state, CPU affinity, NUMA placement, frequency/turbo policy, and interrupt placement;
3. workload generator, offered-load model, dataset, operation mix, seeds, warm-up, and stopping rule;
4. baseline and changed implementation built from the same environment;
5. repetitions at each core count in randomized or counterbalanced order;
6. throughput plus latency distributions, not throughput alone;
7. raw counter/trace output with lost-sample and multiplex information;
8. uncertainty intervals or complete replicate distributions; and
9. threats to construct, internal, external, and conclusion validity.

Instrumentation can perturb the schedule. First quantify no-trace versus trace overhead. High-rate tracing that drops events cannot support complete event counts unless loss is modeled.

REMARK

Amdahl model and lock-contention counters bound the maximum speedup of a serialized critical section, while hardware cache-coherence invalidations explain why even lock-free or sharded structures fail to scale when independent cores write to the same cache line. Synchronization defines correctness; cache coherence and topology define communication costs; scheduling defines waiter and interrupt placement; filesystems and networking supply shared kernel objects; containers and VMs change accounting and observation scope. Observability connects these mechanisms to evidence but cannot replace the mechanism argument.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 7.4.1: Inverting Amdahl Model from Measured Multicore Speedup

Verification application

A kernel benchmark executes on an 8-core symmetric multiprocessing system and achieves a measured speedup of $S(8) = 5.0$ compared to single-core execution on identical fixed work. Assuming the ideal Amdahl scalability equation $S(N) = \frac{1}{s + (1-s)/N}$, derive the implied serial fraction s. Determine the theoretical maximum speedup attainable on an infinite number of cores ($N \to \infty$) under this model.

Equate S(8) to 5.0 and solve for s algebraically before evaluating the limit 1/s.

Equate the observed speedup to the Amdahl equation:

$$\frac{1}{s + \frac{1-s}{8}} = 5.0 \implies s + \frac{1-s}{8} = \frac{1}{5} = 0.20$$

Multiply through by 8 to clear the denominator:

$$8s + 1 - s = 1.6 \implies 7s = 0.6 \implies s = \frac{0.6}{7} = \frac{3}{35} \approx 0.0857\ (8.57\%)$$

Under the ideal Amdahl model with $s = \frac{3}{35}$, the infinite-core asymptotic speedup limit is:

$$S(\infty) = \lim_{N \to \infty} \frac{1}{s + \frac{1-s}{N}} = \frac{1}{s} = \frac{35}{3} \approx 11.67\times$$

Even with an infinite number of cores, execution time cannot fall below 8.57% of the original single-core execution time because the serialized component cannot be parallelized.

Exercise 7.4.2: Eliminating False Sharing via Cache-Line Alignment and Padding

Construction application

Consider an array of per-core statistics counters defined as `struct core_stats { uint64_t rx_packets; uint64_t tx_packets; } stats[64];` running on a 64-core system with standard 64-byte hardware cache lines. Explain why concurrent

updates by different cores cause severe cache-line bouncing (false sharing). Rewrite the structure definition using cache-line alignment attributes or explicit padding to ensure each core writes to an isolated cache line.

Calculate sizeof(struct core_stats) and compare to 64 bytes; apply __attribute__((aligned(64))).

In the initial definition, `sizeof(struct core_stats) = 16` bytes (two 8-byte integers). Consequently, $64/16 = 4$ distinct core structures pack into a single 64-byte cache line. When Core 0 increments `stats[0].rx_packets` and Core 1 concurrently increments `stats[1].rx_packets`, both cores execute read-for-ownership (RFO) requests, invalidating each other's L1/L2 cache lines repeatedly despite accessing distinct variables.

To eliminate false sharing, pad or align each entry to 64 bytes:

```
struct alignas(64) core_stats {
    uint64_t rx_packets;
    uint64_t tx_packets;
    uint8_t  _pad[48]; /* 16 bytes payload + 48 bytes padding =
64 bytes */
};

struct core_stats stats[64];
```

Or using GCC/Clang attribute syntax:

```
struct core_stats {
    uint64_t rx_packets;
    uint64_t tx_packets;
} __attribute__((aligned(64)));

struct core_stats stats[64];
```

Each element occupies exactly one cache line (64 bytes), ensuring concurrent updates by distinct cores modify disjoint cache lines without generating coherence invalidations.

Exercise 7.4.3: Tracing RCU Removal, Grace Period Detection, and Memory Reclamation

Construction transfer

Explain how Read-Copy Update (RCU) decouples readers from writers in a linked list lookup table. Trace the exact sequence of steps when a writer deletes a node `N`:

1. Pointer removal (`list_del_rcu`),
2. Waiting for a grace period (`synchronize_rcu` or `call_rcu`), and
3. Node deallocation (`kfree`). State what condition must be met on every active CPU core before the grace period ends and `kfree(N)` is safe.

Recall that readers execute within rcu_read_lock/rcu_read_unlock without blocking writers; grace periods expire once all cores pass a quiescent state.

The sequence proceeds as follows:

1. **Removal step:** The writer executes `list_del_rcu(N)` (or updates the preceding node's `next` pointer using an atomic store with release semantics). Node `N` is unlinked from the active list. Concurrent readers that entered their critical section before or during the update either see the old pointer or the new pointer; readers seeing `N` continue traversing it safely because `N` remains allocated in memory.
2. **Grace period detection:** The writer initiates a grace period via `synchronize_rcu()` (blocking) or `call_rcu()` (asynchronous callback). The kernel tracks quiescent states across all CPU cores. In non-preemptible kernels, a quiescent state occurs whenever a core undergoes a context switch, enters the idle loop, or executes in user mode. Once every core has passed through at least one quiescent state, all readers that were inside `rcu_read_lock()` at the time of unlinking are guaranteed to have exited their read-side critical sections.
3. **Reclamation step:** When the grace period completes, no reader retains a reference to node `N`. The callback or returning thread invokes `kfree(N)` safely without locks or memory corruption.

Exercise 7.4.4: Diagnosing Lock Contention Bottlenecks with lockstat and perf

Description application

A 32-core database server experiences degraded transaction throughput accompanied by 85% system CPU utilization. Profiling with Linux kernel observability tools reveals that the time is spent in `native_queued_spin_lock_slowpath`. Describe how you would utilize `/proc/lock_stat` and `perf record -g -e lock:lock_acquire` to identify:

1. The exact lock symbol exhibiting the highest contention,
2. The ratio of contention events (`con-bounces`) to total acquisitions, and

3. The specific kernel call chains responsible for holding the lock for the longest hold durations.

Explain how lockstat tracks acquisition count, hold time, and wait time, and how perf record samples call stacks on lock events.

1. **Enabling and inspecting lockstat:** Enable lock statistics via `echo 1 > /proc/sys/kernel/lock_stat`. Run the workload, then inspect `/proc/lock_stat`. The output sorts locks by contention counts and wait times. Look for the top entry in the `contentious:` table to identify the lock data structure and symbolic name (for example, `&sb->s_umount` or `inode_hash_lock`).
2. **Evaluating contention metrics:** Examine the acquisition columns: `acquisitions` (successful acquires), `contentions` (acquisitions that failed immediately and had to spin/block), and `waittime` (total and maximum nanoseconds spent waiting). If `contentions / acquisitions` approaches a substantial fraction (for example, > 20%), the lock is a primary scalability bottleneck.
3. **Attributing hold time call stacks with perf:** To discover who holds the lock while others wait, record lock events with stack traces:

   ```
   perf record -g -e lock:lock_acquire -e lock:lock_acquired -
   - sleep 10
   perf report
   ```

   Inspect call graphs under the lock hold events to pinpoint the kernel function executing inside the critical section. If hold time is dominated by heavy memory allocations or iterative scans under the lock, refactoring the algorithm outside the lock or sharding the protected state resolves the contention.

Exercise 7.4.5: Proving Consistency Invariants in Sharded Multicore Counters

Proof synthesis

To prevent global atomic contention, an operating system implements a sharded statistics counter with K per-core slots: `int64_t slots[K]`. Updates execute locally on core i: `slots[i] += delta`. A global reader reads the total count by summing all slots: $T = \sum_{i=0}^{K-1} \text{slots}[i]$.

1. Prove that without global locking, T does not guarantee a linearizable snapshot across concurrent increments.
2. Construct a concrete race trace where two increments of $+1$ on Core 0 and Core 1 occur concurrently with a reader summation, such that the reader returns a value that reflects neither the initial state nor the final state, yet satisfies monotonicity if increments are strictly non-negative.
3. Formulate the correctness invariant guaranteed by this sharded counter design under relaxed reader execution.

Consider the order in which the reader loop scans slot indices relative to concurrent writes; prove conservation of increments.

1. **Non-linearizability proof:** Linearizability requires each read operation to appear to take effect instantaneously at some point between its invocation and its return. Suppose Core 0 has completed incrementing `slots[0] = 1`, and the reader reads `slots[0] = 1`. Before the reader reads `slots[1]`, Core 1 transfers a count from `slots[1]` to `slots[0]` by subtracting 1 from `slots[1]` and adding 1 to `slots[0]`. The reader then reads `slots[1] = 0`. The reader computes total $T = 1 + 0 = 1$. However, the true total after the transfer was 2, and before the transfer was 2. The read observes an impossible intermediate state where one component was read post-transfer and the other pre-transfer, violating linearizable atomic snapshot semantics.
2. **Concrete race trace:** Let initial state be `slots[0] = 0, slots[1] = 0`.
   - Reader begins summation loop: reads `slots[0] = 0`.
   - Core 0 executes `slots[0] += 1` (now `slots[0] = 1`).
   - Core 1 executes `slots[1] += 1` (now `slots[1] = 1`).
   - Reader advances loop to index 1: reads `slots[1] = 1`.
   - Reader computes $T = 0 + 1 = 1$. The true final state has total 2. The initial state had total 0. The reader observes total 1. If all updates are non-negative increments, the reader never reads a value exceeding the true final count or less than the state prior to invocation, preserving monotonic progress.
3. **Guaranteed invariant (eventual conservation):** If all updates cease at time t_0, any reader whose execution begins strictly after t_0 evaluates:

$$T = \sum_{i=0}^{K-1} \text{slots}[i] = \sum \Delta_{ij}$$

which equals the exact sum of all committed deltas. Under concurrent execution with monotonically increasing deltas, the observed sum satisfies $T_{\text{c}}^{\text{invo}} \leq T_{\text{d}}^{\text{observe}} \leq T^{\text{completio}}_{\text{n}}$, providing an admissible bounded approximation without serializing writers.

References & Further Reading

- Amdahl, “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities” — primary serialized-work argument.
- Linux kernel documentation, RCU Concepts and What is RCU? — removal, grace periods, reclamation, and API obligations.
- Linux man-pages, `perf_event_open(2)` — event types, counting, sampling, scope, multiplexing, and permissions.
- Linux kernel documentation, ftrace — function, event, and latency tracing.
- Linux kernel documentation, Lock Statistics — lock acquisition and contention instrumentation.
- Linux kernel documentation, Pressure Stall Information — CPU, memory, and I/O pressure signals.
- Clements et al., “The Scalable Commutativity Rule” project — workload/interface reasoning about kernel scalability.

SECTION DETAIL

Comparative Kernel Design

"Choose a microkernel for security" and "choose a monolithic kernel for speed" are not engineering decisions. Neither statement identifies a workload, deployed component boundary, authority path, device model, cache state, failure, or measurement. A Linux system with a user-space service can isolate that service from the kernel address space; a carelessly configured microkernel system can grant a driver DMA authority over all memory. Labels constrain possibilities but do not instantiate a design.

The unit of comparison is a **concrete system configuration under an explicit workload**.

Freeze four reference configurations

The comparison uses these bounded configurations, not every possible build or extension:

configuration	privileged deployment boundary	service/authority mechanisms relevant here	stated purpose in comparison
xv6 RISC-V rev5 (`7d7adbb1…`) teaching baseline	compact kernel containing scheduler, VM, filesystem, pipes, and drivers	process page tables, simple Unix-style descriptors/permissions, one filesystem log	expose complete mechanism traces at inspectable scale
Linux general-purpose build	one kernel address space containing core subsystems and selected drivers; user processes outside	VFS objects, namespaces/cgroups, credentials/LSMs, descriptors, RCU and many locking schemes	production mechanism and scalability evidence
XNU open-source tree	Mach, BSD, IOKit, VM, and security subsystems composed in the XNU kernel source; user tasks outside	Mach ports/messages, BSD vnode/file interfaces, task/address-space mechanisms, IOKit driver framework	hybrid composition and dual Mach/BSD interfaces
seL4-based component system	minimal seL4 kernel; selected services and drivers placed in separate user address spaces	kernel-object capabilities, synchronous IPC endpoints, explicit address spaces, IRQ objects	capability-mediated component isolation

“Hybrid,” “monolithic,” and “microkernel” are summaries of these deployed boundaries. They do not specify which optional driver runs where, whether an IOMMU exists, or how a user-level filesystem is built.

The seL4 kernel does not itself supply a POSIX filesystem or full operating environment. A seL4 **system** must choose user-level servers and protocols. Comparing “Linux open” to “seL4 open” without defining those servers invents a path that the kernel does not prescribe.

Retrieval checkpoint: compare a kernel or a system?

A report claims that seL4 has slower pathname lookup than Linux but gives only seL4 kernel IPC timings. Identify the category error and the missing configuration.

Pathname lookup is a whole-system service, not an seL4 kernel primitive. The report must define the user-level VFS/filesystem server, namespace and cache placement, client/server protocol, page sharing or copying, storage backend, scheduling, and cache state. Kernel IPC timing is one possible component cost, not pathname latency. Linux must also be pinned by filesystem, mount options, cache state, kernel build, and workload.

KERNEL ARCHITECTURAL BOUNDARIES AND HARDWARE ISOLATION

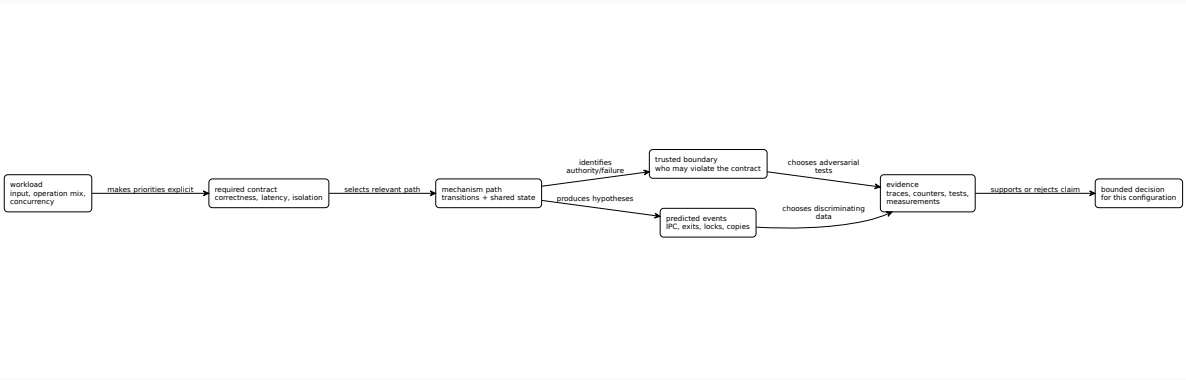
Evaluate each statement regarding kernel architectural models, device isolation, and hardware authority:

Answer: User-space driver isolation prevents illegal CPU writes but requires an IOMMU to prevent malicious peripheral DMA writes.; An seL4 microkernel system enforces all spatial authority and resource allocation through unforgeable capability tokens.

User-space drivers protect CPU memory via MMUs but need IOMMUs to restrict peripheral DMA. seL4 mediates all resources via capabilities. Monolithic kernels scale path lookups effectively via RCU, and formal kernel verification guarantees kernel correctness but does not prevent user server logic errors.

A comparison protocol

For every claim, fill this chain:



Skipping from system name to decision is a missing argument. Counting transitions without defining what each transition does is also weak: one IPC can batch a page, while one syscall can take several blocking device operations.

Workload A: open and read a hot 4 KiB file

Assumptions:

- one process opens `/data/x`, reads 4096 bytes, and closes it;
- pathname metadata and file data are resident in memory;
- permissions allow the operation;
- no concurrent rename or write occurs;
- no storage I/O or page fault is required; and
- each configuration uses its normal cached path.

These assumptions isolate naming, open-state creation, cache lookup, and copying. They do not represent cold-cache storage performance.

xv6 trace

```
user ecall
  -> syscall dispatch
  -> namei walks fixed directory entries under inode locks
  -> filealloc + process descriptor table
  -> read resolves struct file -> inode
  -> readi maps block and obtains buffer-cache block
  -> copyout 4096 bytes
  -> close drops file/inode references
```

The path is small enough to inspect completely. Linear directory scans, fixed cache capacities, coarse teaching choices, and copy behavior are not production baselines.

Linux trace at the stable abstraction level

```
open-family syscall
  -> VFS pathname walk through dentry/inode objects and filesystem
methods
  -> allocate/install struct file in descriptor table
  -> read-family syscall resolves struct file
  -> filesystem/page-cache read path supplies resident bytes
  -> copy to user and advance open-file position
```

Linux's exact functions vary by filesystem, kernel version, flags, and I/O interface. The VFS documents dentries as in-memory pathname components, inodes as filesystem objects, and `struct file` as the open-file object. RCU-walk and lock/ref-walk can take different pathname paths. A responsible trace records which path occurred rather than treating "VFS" as one instruction.

XNU trace at the open-source boundary

```
BSD syscall interface
  -> pathname/vnode and file-descriptor mechanisms in BSD subsystem
  -> filesystem vnode operations and resident-data path
  -> VM/unified-buffer integration where configured
  -> copy/result to user task
```

The XNU source tree composes BSD, Mach VM/IPC, IOKit, and security code in one kernel. “Mach-based” does not imply that a POSIX file read becomes a user-space filesystem IPC. A source-backed analysis must follow the selected XNU build and filesystem.

seL4 component-system trace

No unique trace exists until the architecture is supplied. One candidate is:

```
client invokes VFS endpoint capability
  -> user-level VFS server resolves cached components
  -> filesystem server or shared cache supplies page/bytes
  -> reply carries data, mapped page, or shared-buffer completion
```

The seL4 kernel mediates endpoint capability and IPC, scheduling, and mappings. User-level servers define pathname, descriptor, cache, and storage semantics. Copy count and context switches depend on protocol, passive-server/scheduling design, shared-memory policy, and whether VFS and filesystem are one component.

Worked comparison: what can be concluded without timing?

property	source-backed conclusion	unsupported conclusion
xv6 inspectability	the whole teaching path and fixed structures can be traced	xv6 is representative of production lookup scale
Linux implementation	VFS and filesystem/page-cache paths are in the kernel configuration	one kernel crossing means a fixed latency
XNU composition	BSD file mechanisms coexist with Mach/VM/IOKit in XNU	every file operation is a Mach message
seL4 placement	file service semantics live in selected user components	microkernel structure determines exact copies or speed

Application checkpoint: cold-cache reversal

Change only one assumption: data requires storage I/O. Which new mechanisms must enter every comparison, and why can the hot-cache result no longer rank the systems?

Add block mapping, request queueing, device driver placement, DMA, interrupts, scheduler blocking/wakeup, read-ahead, filesystem recovery state, and the storage medium. The I/O latency may dominate naming and IPC costs; batching and asynchronous interfaces can reverse a hot-path ranking. Each system's cache and driver architecture must be specified. A hot-cache measurement cannot estimate cold-device behavior by adding one constant unless that model is validated.

Workload B: a malformed device request

Assume a network or storage driver parses an adversarial descriptor and executes one out-of-bounds write. Compare the first protection boundary that can contain architectural memory corruption.

configuration	driver placement in bounded configuration	first-order result
xv6	kernel address space	kernel state can be corrupted; system integrity is lost
Linux with in-kernel driver	kernel address space	corruption is within kernel TCB; process isolation does not contain it
XNU with in-kernel IOKit driver	kernel address space	hybrid label supplies no memory boundary around that driver
seL4 system with user driver	separate address space with bounded capabilities	process memory fault may be contained; service can fail; delegated authority still matters

The seL4 row depends on page mappings, endpoint capabilities, IRQ authority, and DMA. Without an IOMMU or a trusted device that respects bounded DMA, a user-space driver can program a device to overwrite memory outside its address space. Moving code to user space changes CPU page-table protection; it does not automatically constrain devices.

Linux and XNU can also place selected device logic in user space. The right conclusion is that **placement and authority**, not kernel family name, determine this fault boundary.

Worked design: multi-tenant device gateway

Requirements:

- untrusted clients submit variable-length commands;
- one parser crash must not corrupt the scheduler or other tenants;
- device DMA must reach only per-request buffers;
- restart within 100 ms is required;
- steady-state p99 is important but not yet measured.

A defensible initial architecture places parser and policy in an unprivileged component, gives it no direct DMA-programming authority, and uses a small broker/driver with IOMMU-enforced mappings and a restart protocol. seL4 capabilities can express endpoint, frame, and IRQ authority explicitly; a Linux design can use process isolation, descriptors, an IOMMU-aware kernel driver, seccomp/LSM, and a narrow broker protocol.

The choice between them cannot be made from "microkernel" alone. Evidence must include malformed-message tests, component restart state, stale DMA mapping tests, broker TCB review, and p99 measurements under the actual protocol. If formal assurance of the kernel mechanism is a

hard requirement and the target board lies in a verified seL4 configuration, that requirement can dominate. If a required production driver exists only in Linux, implementation and verification cost can dominate.

Transfer checkpoint: remove the IOMMU

The same gateway runs on hardware with no IOMMU, and the device accepts arbitrary physical DMA addresses. What authority boundary remains credible?

CPU page tables still isolate ordinary loads/stores by the user driver, but they do not constrain device DMA programmed with arbitrary physical addresses. The trusted driver or broker must validate every descriptor and be trusted not to program out-of-range DMA; physical bounce buffers or a device with its own enforced bounds may reduce authority. The user-space address-space boundary alone no longer establishes memory containment for device actions.

Workload C: a 64-byte control message

Assume two mutually distrustful services send a request and receive a response. No payload page transfer is needed. Required semantics are one request boundary, authenticated endpoint authority, bounded queueing, and cancellation on peer death.

Mechanism comparison

system	concrete primitive	semantic questions before timing
xv6	pipe byte stream	who frames messages, handles partial reads/writes, and detects peer close?
Linux	Unix-domain socket or pipe	stream vs datagram/sequence packet, credential passing, descriptor transfer, buffer limits
XNU	Mach port/message	send/receive rights, message structure, port death notification, queue limits
seL4	endpoint capability and IPC	synchronous rendezvous, capability transfer, badge/authority, scheduling-context behavior

A count of "two context switches" is not portable evidence. A blocked sender may donate or not donate scheduling context; a socket may buffer without scheduling the receiver immediately; a pipe requires application framing; a Mach or seL4 message can carry protected references with different semantics.

Worked protocol selection

The server must never accept requests from a client lacking authority to one object. In a seL4 configuration, give each client an endpoint capability or badged derivation whose possession identifies the authorized channel; validate message length and object-specific protocol state. In Linux, accept over a protected Unix-domain socket, authenticate peer credentials where that policy is meaningful, and pass an already authorized object descriptor rather than a pathname. In XNU, bind authority to appropriate port rights and validate message descriptors. In xv6, the application must build framing and authority atop inherited pipe descriptors and the teaching kernel's limited policy.

The protocols can implement equivalent high-level policy, but their trusted brokers, delegation semantics, revocation, and failure paths differ.

Boundary checkpoint: shortest path is not safest path

One design removes a broker IPC by mapping the device command queue writable into every client. It measures lower median latency. Identify the authority change and the missing evidence.

Every client now holds direct mutation authority over shared queue state and possibly device-visible descriptors. A malformed or malicious client can corrupt peers' requests, violate queue invariants, or program excess DMA unless hardware and layout enforce separation. Required evidence includes per-client ring isolation, descriptor validation, IOMMU mappings, memory-order proof, overload behavior, and adversarial tests. Lower median latency does not compensate for violating the stated authority contract.

Workload D: read-mostly metadata on 64 cores

Assume 99% lookups, 1% insertion/removal, a skewed key distribution, and a target of stable p99 latency through 64 cores.

- xv6's small teaching caches and lock structure provide a readable baseline, not a scaling claim.
- Linux uses several techniques, including RCU in pathname/dentry-related read-mostly paths, fine-grained locks, per-CPU state, and specialized caches; the actual path must be traced.
- XNU's current source contains subsystem-specific locks and scalable-memory-reclamation structures; a claim requires tracing the selected path rather than extrapolating from "hybrid."
- a seL4 system chooses whether metadata is in one server, sharded servers, shared memory with capabilities, or another arrangement; the kernel does not decide the data structure.

EXAMPLE

Suppose 64 cores repeatedly resolve `/usr/bin/env`. If each lookup acquires even a shared reader-writer lock, every core atomically increments and decrements a shared reader counter, bouncing that cache line across all 64 cores in Modified/Invalid states. Under an RCU read-side critical section (`rcu_read_lock()` / `rcu_read_unlock()`), readers issue no shared-memory writes at all during traversal, allowing the dentry cache lines to remain Shared in every core's L1 cache while writers publish replacement nodes and defer reclamation until a grace period elapses.

Worked experiment design

For each configuration:

1. define identical abstract operations and key distribution;
2. verify semantic equivalence, including update visibility and failure;
3. pin core affinity, topology, memory placement, and interrupt handling;
4. measure 1, 2, 4, 8, 16, 32, and 64 cores with repeated trials;
5. report throughput, latency distribution, CPU time, migrations, cache/coherence indicators where valid, and IPC/lock traces;
6. inspect the hottest serialized state; and
7. run a controlled redesign or workload-shape change predicted to affect it.

If one system batches 100 lookups per IPC while another issues one syscall per lookup, report both operation and transition counts. “IPC count” is not a workload-normalized metric.

Architecture labels fail as predictors

> **WARNING**
>
> High-level taxonomy labels (“monolithic”, “microkernel”, “hybrid”) do not determine either the effective Trusted Computing Base (TCB) or multicore scalability. A system’s fault blast radius and latency distribution are functions of the concrete authority graph, DMA isolation, and shared-cache-line write set along the executed path.

Counterexamples to architectural taxonomy:

Taxonomy label assumption	Concrete counterexample	Underlying mechanism causing the reversal
“Microkernels guarantee driver fault isolation”	User-space driver without an IOMMU corrupts kernel RAM	Unrestricted peripheral DMA bypasses CPU MMU page tables
“Monolithic kernels cannot scale pathname lookup”	Linux VFS pathname walk scales across 64 cores	Lockless RCU traversal avoids writes to shared cache lines
“Microkernel IPC is always the dominant latency bottleneck”	Batched IPC server outperforms single-syscall path	Amortizing message-passing cost across multi-block transfers
“Formally verified kernels guarantee total application safety”	Verified kernel running misconfigured user servers	Proof assumes correct hardware, configuration, and user capability policy
“Monolithic kernels expose every subsystem bug to every process”	Local unprivileged container cannot reach network parser	Namespace filtering and seccomp remove vulnerable syscall entry paths

The comparison target is always the end-to-end trusted and executed path.

REMARK

This lesson restricts comparative claims to four inspectable reference architectures (xv6–riscv rev5, Linux, XNU, and seL4) under fixed workload models so every mechanism claim can be traced to concrete source files or formal specifications rather than marketing taxonomies. Chronology and product popularity are excluded because neither explains a mechanism; alternative kernel paradigms deserve the same source-backed configuration treatment.

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 7.5.1: Contrasting Monolithic and Microkernel Control Transfer Paths

Verification application

A user application invokes a file read operation (`read(fd, buf, 4096)`). Compare the sequence of privilege transitions, context switches, and cache effects under:

1. A monolithic kernel (such as Linux or xv6), and
2. A classical microkernel (such as seL4 or Mach) where the file system executes in an independent user-space server task. Count the minimum number of user-kernel boundary crossings and address-space switches incurred by each architecture.

In a monolithic kernel, the syscall handler executes within the kernel address space. In a microkernel, client IPC to server and server reply to client each pass through the kernel.

1. **Monolithic path:**
    - Client executes a trap instruction (`ecall` / `syscall`): User → Kernel transition.
    - The kernel resolves the descriptor, accesses the page cache / VFS in kernel mode, copies 4096 bytes directly to user buffer `buf` .
    - Kernel executes `sret` / `sysret` : Kernel → User transition.
    - **Metrics:** Exactly 2 privilege transitions (`U -> K -> U`), 0 address-space context switches (page table root register `satp` / `CR3` does not change), minimal TLB invalidations.
2. **Microkernel path:**
    - Client sends synchronous IPC to VFS server: User → Kernel trap (Transition 1).
    - Microkernel transfers IPC message, switches address space from Client to VFS Server: `satp` / `CR3` switch (Switch 1), resumes VFS server in user space (Transition 2).
    - VFS server processes request, accesses buffer/disk, and sends IPC reply: User → Kernel trap (Transition 3).
    - Microkernel transfers reply data, switches address space from VFS Server back to Client: `satp` / `CR3` switch (Switch 2), resumes Client in user space (Transition 4).
    - **Metrics:** At least 4 privilege transitions (`U -> K -> U -> K -> U`), 2 full address-space context switches, and potential TLB/cache eviction overhead depending on tagged TLB hardware support.

Exercise 7.5.2: Tracing seL4 Capability Invocation and IPC Message Transfer

Construction application

In seL4, all kernel services are invoked via capabilities stored in per-thread C-nodes. Describe how a client thread initiates a synchronous rendezvous IPC with a server thread listening on an Endpoint object:

1. The capability token checked by the kernel,
2. The transfer of small message payloads via hardware CPU registers, and
3. The generation and transmission of an unforgeable Reply capability to the server.

Explain how the kernel validates the Endpoint capability rights, copies register-sized arguments directly, and creates a single-use Reply capability.

1. **Capability validation:** The client loads an Endpoint capability index from its C-space into a CPU register and invokes `seL4_Call(ep_cap, msg_info)`. The kernel trap handler decodes the capability: it verifies that `ep_cap` points to a valid kernel Endpoint object and possesses the `Write` (or `Call`) right. If invalid or rights are missing, the kernel faults or returns an error.
2. **Direct register message passing:** If the server is already blocked waiting on the Endpoint (`seL4_Recv`), the kernel performs a direct thread switch. For short messages (up to 4–8 words depending on ISA), the kernel transfers the message words directly across CPU argument registers (`a2` - `a7` on RISC-V) without copying through kernel memory buffers.
3. **Reply capability issuance:** Because the client invoked `seL4_Call` (a call-reply pattern), the kernel transitions the client to the `BlockedOnReply` state and automatically mints a transient, single-use `Reply` capability placed into the server thread's designated C-slot. When the server finishes processing, it invokes `seL4_Send` on that Reply capability, waking the client, returning the result, and invalidating the Reply capability.

Exercise 7.5.3: Comparing Trusted Computing Base Footprints and Verification Tractability

Description application

Evaluate the trade-off between Trusted Computing Base (TCB) size and system verification feasibility:

1. Compare the lines of code (LoC) of the seL4 microkernel (~10,000 lines) with a general-purpose Linux kernel (~30+ million lines).
2. Explain why end-to-end formal verification (such as the Isabelle/HOL functional correctness, integrity, and confidentiality proofs achieved for seL4) is tractable for seL4 but intractable for monolithic production kernels.
3. Discuss what remains outside the verified boundary in a complete running seL4–based operating system.

Focus on state space explosion, concurrency complexity, device driver volume, and user-space policy servers.

1. **TCB scale difference:** seL4 comprises approximately 10,000 lines of verified C code and assembly, maintaining a strictly minimized kernel state space. In contrast, a typical Linux distribution kernel exceeds 30 million lines of C, where drivers, filesystems, network protocols, and scheduling policies all reside in supervisor mode.
2. **Verification tractability:**
   - Formal verification requires constructing mathematical proofs of refinement from abstract specification to executable binary. State space explosion makes automated and interactive theorem proving scale non-linearly with lines of code, concurrency interleavings, and unstructured pointers.
   - seL4 is deliberately single-threaded in the kernel (with event-driven preemption points or fine-grained multicore models), restricts dynamic memory allocation to capability retyping of user-provided untyped memory, and avoids in-kernel complexity.
   - In Linux, complex preemptible RCU subsystems, dynamic locking graphs, heterogeneous device drivers, and continuous code churn render complete formal proof mathematically intractable with current proof technology.
3. **Unverified residual boundary:** Formal proof of seL4 guarantees only that the microkernel binary adheres to its specification under the assumed hardware model. It does not verify: hardware errata / CPU microarchitectural side-channels, board firmware / bootloaders, device firmware, or the application/server logic running in user space (for example, a buggy user-space file server can still leak its own files).

Exercise 7.5.4: Analyzing Unikernel Attack Surfaces and Single-Address-Space Boundaries

Formalisation transfer

A unikernel (such as MirageOS or OSv) compiles a single specialized application together with minimal operating system library routines into a single binary running in a single address space at Ring 0 inside a hardware virtual machine (VM).

1. Formalize the isolation boundary: what protects two distinct unikernel instances running on the same physical host?
2. Compare the attack surface of an application running as a Linux container versus the same application running as a unikernel within a type-1 hypervisor.
3. State one major operational disadvantage of unikernels regarding observability and debugging.

Contrast hardware hypervisor VM isolation (EPT/NPT) against Linux kernel system call attack surfaces, and note the absence of user/kernel boundary tools like gdb or ptrace.

1. **Formal isolation boundary:** Within the unikernel instance, there is no spatial or privilege boundary: application logic and kernel library code execute at the identical privilege level (e.g., Ring 0 / S-mode) sharing one page table. Isolation between distinct unikernel instances is enforced entirely by the underlying hardware hypervisor (Type-1 or Type-2) using Second-Level Address Translation (Intel EPT / AMD NPT) and VMCS/VMCB hardware virtualization controls.
2. **Attack surface comparison:**
   - In a Linux container, all containers share a single host kernel image. A vulnerability in any of the ~400+ host system calls, ioctl handlers, or network stack parsers allows container breakout to the host kernel.
   - In a unikernel, the guest cannot attack host kernel system calls because no host system call interface exists inside the guest. The hypervisor exposes a drastically narrower hypercall interface and virtual device emulation layer, substantially reducing the breakout attack surface.
3. **Operational disadvantage:** Because a unikernel lacks multi-process abstractions, interactive shells (`bash`), and standard debugging infrastructure (`ptrace` , `gdb` , `ps` , `/proc`), inspecting runtime state, attaching debuggers, or applying live hot-patches is extremely difficult. Diagnosing production crashes requires external hypervisor memory inspection or rebuilding the whole binary.

Exercise 7.5.5: Proving Subsystem Fault Isolation in Microkernel Architectures

Proof synthesis

Let a system consist of microkernel K, user application process A, file system server F, and device driver D, each residing in distinct address spaces S_K, S_A, S_F, S_D. Communication occurs solely via message passing across kernel endpoints E_F and E_D. Hardware includes an MMU and an IOMMU configured such that peripheral P controlled by D can only DMA into memory mapped within S_D.

1. Prove that an arbitrary control-flow hijack or fatal memory fault in driver D cannot alter the persistent file metadata state maintained in S_F without transmitting a semantically valid IPC message across E_D.
2. Specify the exact failure mode that occurs if the IOMMU is absent or disabled.

Formulate the invariants enforced by the MMU page tables, IOMMU DMA translation, and kernel capability enforcement.

1. **Proof of subsystem isolation:**
   - **Spatial CPU isolation:** By MMU construction, the CPU page table for S_D maps only memory frames allocated to D. Frames backing S_F and S_K are either unmapped or marked invalid in D's page table. Therefore, no CPU instruction executed by D (even under arbitrary control-flow hijacking) can read or write memory in S_F directly via load/store instructions.
   - **DMA peripheral isolation:** Because the IOMMU translation tables for peripheral P map only the DMA buffer frames assigned to S_D, peripheral P cannot write to S_F frames via direct memory access over the system bus.
   - **Kernel control isolation:** Control transfers between S_D and S_F are mediated exclusively by microkernel K via endpoint $E_{D \to F}$. The kernel delivers messages to F only if D executes a valid kernel trap with a valid capability to $E_{D \to F}$.
   - **Conclusion:** Because D's instructions cannot access S_F's memory directly and D's peripheral cannot DMA into S_F's memory, the internal state of F can change solely in response to messages received through $E_{D \to F}$. If F's protocol rejects malformed or unauthenticated requests, corrupted state in D cannot arbitrarily corrupt F's metadata.
2. **Absence of IOMMU failure mode:** If the IOMMU is absent, peripheral P issues DMA writes with raw physical bus addresses. A compromised or malfunctioning driver D can program P's DMA target register to point directly to physical frames holding S_F or S_K memory, completely bypassing CPU MMU protections and corrupting file metadata or kernel code without sending any IPC message.

References & Further Reading

- MIT PDOS, xv6 RISC-V book and immutable xv6-rev5 source tree — inspectable teaching-kernel mechanisms fixed for the comparison.
- Linux kernel documentation, Overview of the Virtual File System — dentry, inode, file, superblock, and operation interfaces.
- Linux kernel documentation, RCU Concepts — read-mostly synchronization and reclamation.

- Apple open-source XNU README and source tree — official Mach/BSD/IOKit/security composition and build structure.
- seL4 Reference Manual — kernel-object, address-space, scheduling, IRQ, IPC, and capability interfaces.
- seL4 capability tutorial and IPC tutorial — concrete authority and message mechanisms.
- Linux kernel documentation, Credentials in Linux — Linux subject/object credential mechanisms used in the authority comparison.

SECTION DETAIL

Operating Systems Capstone

A patch passes the upstream tests and reports “18% faster.” The claim still fails review. The author changed the benchmark while changing the scheduler, retained only five favorable runs, did not record the baseline commit, and never tested a blocking workload. Under memory pressure, one error path leaves a process runnable on two queues. The benchmark result is neither reproducible nor relevant to the broken invariant.

The capstone is a bounded engineering argument:

> *Under a stated workload and failure model, this mechanism preserves these invariants and changes this observable relative to this pinned baseline; these tests and measurements support the claim, subject to these limitations.*

Every noun in that sentence must resolve to an artifact. A build log is not a correctness argument. A passing test is not proof that no bug exists. A simulation is not a measurement. A fast median does not excuse corruption, deadlock, authority expansion, or an unexplained tail.

The project contract

Select exactly one option from the bounded set below. A project may extend an earlier checkpoint only when it introduces a new mechanism, invariant, adversarial failure, and evaluation question. Repackaging a lab patch with more prose does not satisfy the contract.

All work descends from official xv6 RISC-V tag `xv6-riscv-rev5`, commit `7d7adbb1b0acbd67c9766a20d0f9900fef2789fa`. Use either a clean rev5 tree or the course’s ordered cumulative checkpoint patch stack rooted at rev5. Record the chosen start point and keep the capstone treatment separable from every earlier patch. The project must remain small enough that one engineer can explain and trace every changed control path.

Definitions that make the claim testable

term	operational meaning
course root	official xv6-riscv-rev5 commit `7d7adbb1b0acbd67c9766a20d0f9900fef2789fa`; every allowed project history descends from this object
start point	either the clean course root or a reproducible, ordered cumulative checkpoint stack applied to that root
baseline	the immutable chosen start point plus configuration, toolchain, emulator or hardware target, and command sequence against which the capstone treatment is compared
treatment	the baseline plus only the isolated capstone change; earlier checkpoints, upgrades, and unrelated refactors are not treatment
contract	externally visible allowed behavior, error behavior, and explicitly unsupported behavior
invariant	a property required after every relevant transition, not merely at function return
failure model	the faults, interleavings, resource limits, crashes, and adversarial inputs the design claims to handle
oracle	a rule that maps an observation to pass, fail, or inconclusive without consulting the implementation's intended answer
workload	operations, input distribution, concurrency, machine state, duration, and units that generate an observation
trial	one independently identified execution with recorded configuration, seed, order, and raw result
claim	a falsifiable relation among mechanism, workload, observable, and baseline
threat	a plausible reason the evidence could support the claim even though the claim is false, or reject it even though it is true

Assume failures can occur at any documented injection point, CPUs can interleave at every synchronization boundary, allocations can fail where the kernel permits failure, and user inputs can take every contract-valid boundary value. Do not assume sequential consistency for unsynchronized shared memory, a particular disk write order without an enforced persistence mechanism, or FIFO device completion unless the device contract provides it.

Retrieval checkpoint: what is missing?

The statement "the new lock-free table is correct because it completed one million lookups without crashing" contains an observation. Name four missing parts of an engineering claim.

At minimum it lacks:

1. a contract defining which lookup results are legal under concurrent insert or delete;
2. invariants covering publication, object lifetime, and reclamation;
3. a pinned baseline and treatment boundary;
4. a workload and failure model, including CPU count and interleavings;
5. an independent result oracle; and
6. adversarial evidence capable of exposing stale references or forbidden results.

Completion without a crash demonstrates only that one execution reached its end. It cannot establish linearizability, memory safety, progress, or superiority to a baseline.

CAPSTONE EVALUATION RIGOR AND TREATMENT BOUNDARIES

Evaluate each statement regarding experimental controls, treatment isolation, and invariant testing in the capstone project:

Answer: The treatment diff must contain solely the isolated mechanism change applied directly on top of the pinned baseline.; A rigorous evaluation requires adversarial stress tests capable of failing if core invariants are deliberately violated.

The treatment must isolate only the evaluated mechanism. Correctness and invariant preservation strictly supersede throughput metrics. Confounding unrelated fixes corrupts causal inference, and adversarial tests must be capable of falsifying broken implementations.

Choose one bounded option

> **WARNING**
>
> Selecting a single option and keeping its treatment diff strictly separated from earlier lab checkpoints is mandatory: combining two treatments (for example, modifying the scheduler while redesigning the buffer cache) confounds causal attribution in paired trials.

The hard exclusions are part of each option. A proposal that silently absorbs an excluded production feature is too broad.

option	required mechanism and contract	mandatory adversarial evidence	hard exclusions
A — scheduler policy and accounting	add one policy or one bounded accounting mechanism; define runnable-state ownership, queue membership, wakeup, migration, and idle behavior	CPU-bound plus sleeping tasks, multicore migration, repeated wakeups, starvation probe, and accounting conservation	hard real-time guarantees, energy policy, full priority inheritance, and claims about every workload
B — virtual-memory state transition	implement either copy-on-write fork or a bounded anonymous/file-backed mapping subset; define PTE permissions, frame reference counts, fault outcomes, and teardown	parent/child writes, fork trees, memory exhaustion, concurrent exit, invalid addresses, and stale-translation probe	complete POSIX `mmap`, swapping, NUMA placement, huge pages, and production-grade overcommit
C — interrupt-driven device queue	implement a bounded queue for one emulated device or teaching device; define descriptor ownership, publication, completion, retry, and shutdown	wraparound, full and empty queues, duplicate/spurious completion, delayed interrupt, malformed length, and injected allocation failure	universal driver framework, real-device certification, DMA isolation proof, and unmodeled hardware claims
D — recoverable filesystem mutation	add one persistent operation distinct from the rename lab, such as atomic temporary-file publication or a bounded durability call; define namespace and recovery semantics	transaction exhaustion, allocation failure, every named crash cut, recovery idempotence, and concurrent conflicting operations	full POSIX durability, directory trees, cross-filesystem operations, ACLs, and storage-device firmware guarantees
E — multicore kernel-state redesign	shard or redesign one contended kernel table while preserving its lookup/update contract; define lock order, object lifetime, and progress	skewed keys, CPU-count sweep, create/delete races, exhaustion, hotplug exclusion, and contention tracing	a general RCU library, universal scalability, lock-freedom claims without proof, and production NUMA tuning

Combining two options usually weakens the evidence. A queue plus a new scheduler creates two treatments and makes a causal result difficult to interpret. Focus on a single bounded mechanism to preserve clear causal attribution.

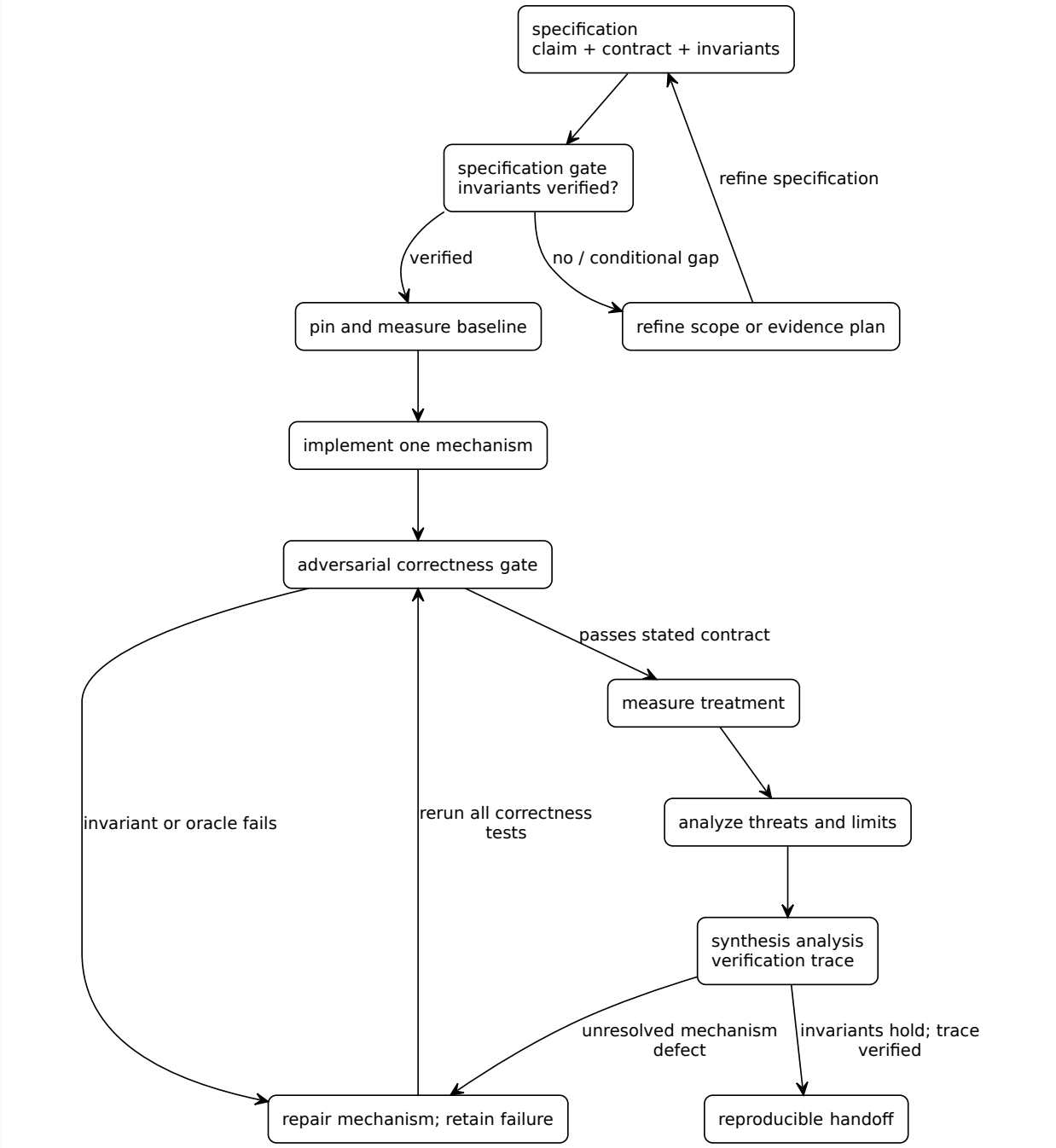
Scope budget: 1,140 minutes

phase	minutes	exit evidence
specification, source audit, invariants, and threat model	120	falsifiable specification, invariant inventory, and threat model
pinned baseline and experimental harness	180	reproducible clean build, upstream tests, calibration data
mechanism implementation and design record	360	reviewable commits or patch series, transition traces, invariant assertions
adversarial, concurrency, and fault tests	240	test suite, wrong-implementation kill matrix, minimized failures
evaluation and threats-to-validity analysis	120	raw trials, analysis script, figures/tables, bounded interpretation
design memo, verification trace, and handoff	120	reproducible package verifiable from a fresh checkout
total	**1,140**	**19 hours**

The time box bounds scope, not rigor. If the chosen mechanism cannot be made safe within it, preserve the failing evidence, reduce the contract, and document the residual gap. Do not hide an unsafe path to make the project appear complete.

Specification and verification gate

Implementation begins only after establishing a precise specification. Verify that the specification defines explicit invariants, an observable baseline, and an adversary. A disposable feasibility probe is allowed when declared, but none of its code enters the treatment until the specification is verified and its provenance is clear.



The gate rejects proposals that promise “better performance,” “more security,” or “POSIX support” without an observable, adversary, and boundary.

Proposal packet

Keep the packet short enough to review but precise enough to falsify. It must contain:

1. **Claim.** One sentence naming workload, treatment, baseline, preserved contract, observable, expected direction, mechanism, and limitation.

2. **Pinned object.** Course-root tag/hash, clean or cumulative start mode, ordered checkpoint patch checksums, resulting start-point identity, license, configuration, compiler, emulator or hardware, host facts, and commands.
3. **Contract and exclusions.** Inputs, outputs, error behavior, concurrency semantics, persistence point if relevant, and non-goals.
4. **State model.** State variables, initial state, legal transitions, invariant after each transition, and ownership or authority.
5. **Failure model.** Relevant crashes, races, exhaustion, malformed inputs, interrupt behavior, and explicitly unhandled faults.
6. **Mechanism sketch.** A control-path diagram plus a lock/authority/resource-order table. No implementation listing is required.
7. **Evidence map.** At least four named wrong implementations and a test or observation that should kill each one.
8. **Evaluation plan.** Baseline, workloads, units, trial order, repetitions, raw-data format, analysis rule, and stop conditions.
9. **Threat preview.** One plausible construct, internal, external, and conclusion-validity threat.
10. **Provenance plan.** Primary sources, borrowed interfaces or code, licenses, collaborators, tools, and assistance-disclosure method.

Use this claim grammar when the scope drifts:

```
Under [workload W and failure model F], treatment [T] relative to pinned
baseline [B] preserves [contract/invariants C] and changes [observable
M]
in [direction or bounded relation D] because [mechanism Z]. Evidence [E]
can reject the claim; the result does not cover [limitation L].
```

Worked trace 1: repair an unreviewable scheduler proposal

Initial proposal:

> *Make xv6 scheduling fairer and faster.*

The terms “fairer” and “faster” have no units; the workload is absent; simultaneous improvement may be impossible; no invariant prevents a lost runnable process; and the claim could not be rejected by a specific result.

A bounded replacement:

```
Under a two-CPU QEMU workload containing three CPU-bound processes and
one
process that sleeps for 10 ms after each 1 ms burst, weighted round-
robin T
relative to rev5-rooted start point B preserves single-queue membership
and eventual
dispatch for continuously runnable positive-weight processes. T is
expected
to reduce the interactive process's p99 runnable-to-running delay, while
reporting aggregate completed bursts and CPU time as possible trade-
offs.
Queue-transition assertions, deterministic wakeup tests, randomized
stress,
and paired trials E can reject the claim. The result does not establish
a
hard latency bound or superiority for other workload mixes.
```

The revised proposal names a fairness proxy instead of treating fairness as a mood. It permits throughput to fall and makes that trade-off visible rather than defining success as "every metric improved."

Application checkpoint: gate two claims

Classify each claim as **approve**, **revise**, or **reject**, and give the smallest repair.

1. "COW fork uses less memory than eager copying." The plan pins a commit, measures resident frames after a 64-page fork tree, tests write faults and OOM, records raw trials, and excludes page cache effects. It does not define when a shared frame may be freed.
2. "My driver works on all VirtIO devices." The implementation targets xv6's emulated block device, tests one QEMU version, and assumes completions arrive in descriptor order although the proposed contract does not guarantee that order.

1. **Revise.** The performance/memory observable is bounded, but correctness lacks a frame-lifetime invariant. Add: the frame reference count includes mapping references and temporary pins; at least one such reference remains while any valid or stale translation, copy, or in-flight fault can access the frame. The allocator may reclaim the frame only when the final reference is dropped after PTE removal and required TLB invalidation complete. Define atomicity/locking for those transitions and add a concurrent exit/write test that can expose premature reuse.
2. **Reject as written.** Evidence from one emulated device cannot support "all VirtIO devices," and the completion-order assumption can corrupt descriptor ownership. Repair the claim to the pinned QEMU device/configuration, identify the normative queue contract, key completions by descriptor identity rather than assumed order, and inject reordered completions if the model permits them.

Freeze a reproducible baseline

The baseline is experimental data, not a nostalgic description of upstream behavior.

Create `REPRODUCE.md` with values rather than placeholders:

```
course-root URL, tag, hash, and license:
start mode (`clean-rev5` or `cumulative-stack`):
ordered checkpoint patch IDs/checksums (`none` for clean mode):
baseline start-point commit or tree checksum:
treatment commit or patch checksum:
submodules/dependencies and checksums:
compiler, assembler, linker, and make versions:
QEMU or hardware model and version:
host OS, architecture, CPU allocation, and memory allocation:
kernel/build configuration:
build command:
upstream test command:
project correctness-test command:
measurement command:
trial seeds and randomized execution order:
raw-data path and schema:
analysis command and expected output checksums:
known nondeterminism and unsupported environments:
```

Baseline procedure

1. Verify that tag `xv6-riscv-rev5` resolves to the exact course-root hash; never substitute a moving branch.
2. Choose clean mode or reconstruct the cumulative checkpoint stack in course order, recording every patch/checksum and conflict resolution.
3. Record license and notices before importing or adapting any code.
4. Build the clean course root, then build the chosen start point with the exact documented commands from fresh trees.
5. Run the upstream suite on both; run every applicable checkpoint regression on a cumulative start point; preserve complete output plus exit status.
6. Run the proposed harness against the **unchanged start point**. A harness that fails only because the capstone feature is absent must distinguish "unsupported" from "baseline defect."
7. Execute a no-change calibration: baseline A versus an independently rebuilt baseline A. Large apparent differences expose harness or host noise before they are mistaken for a treatment effect.
8. Save the machine-readable environment, raw data, and checksums in the implementation package.

The clean mode isolates the capstone mechanism. The cumulative mode tests integration with accepted checkpoint work. Results from the two modes are not interchangeable: an earlier scheduler, VM, driver, or filesystem patch can change both correctness paths and performance. If the capstone applies only to one mode, state that boundary rather than merging results.

Integration incompatibilities

Any cumulative patch conflict, changed syscall number, altered disk format, different lock order, expanded log bound, test-hook collision, or failed checkpoint regression is an incompatibility requiring a recorded resolution and reviewer review. Preserve the conflicting patch and explain which invariant or measurement it affects. Do not replace a cumulative file with its clean rev5 version or fold earlier changes into the capstone diff.

Rev5 does not already implement most option mechanisms: it has no COW fork, production mapping interface, general device framework, or scalable shared-table abstraction. Those absences define implementation work, not permission to import a later branch. A later upstream fix may be evaluated only as a separately labeled sensitivity comparison.

A container image can help but is not sufficient by itself. Record its source definition and immutable digest; otherwise the opaque image merely packages the missing provenance.

Transfer checkpoint: the baseline moved

Upstream fixes a wakeup bug after proposal approval. The fix seems relevant to the scheduler project. Should the treatment silently rebase?

No. Preserve the rev5–rooted approved baseline so the course comparison remains interpretable. If the fix matters, create a separate sensitivity branch, document the source delta, and rerun both its no-capstone and capstone variants as a new labeled pair. It cannot replace the course root or be mixed with rev5 measurements: source change and proposed mechanism would be confounded.

Derive the mechanism from invariants

DEFINITION

A kernel state transition $S \xrightarrow{o} S'$ is invariant-preserving if, for every concurrent interleaving and every failure cut permitted by the project's failure model, either all externally visible effects of o satisfy the post-transition invariant $I(S')$ or cleanup restores an allowed pre-transition state satisfying $I(S)$.

Do not organize the design memo around files changed. Organize it around state and transitions.

For every mutable object, provide a table with this shape:

state or resource	owner / protecting mechanism	legal writers	invariant after publication	reclamation or persistence rule
example: run-queue node	one queue lock	enqueue, dequeue, migration	runnable process appears in exactly one queue	remove under source lock before reuse

Then trace each operation through five questions:

1. What state is read before validation?
2. Which transition establishes ownership?
3. At what point can another CPU, interrupt, process, or recovery pass observe it?
4. Which failure can occur after each visible transition?
5. What restores or completes the invariant on every exit path?

Assertions should encode cheap local consequences of invariants. They supplement adversarial tests; an assertion that repeats the same mistaken model as the implementation is not an independent oracle.

Worked trace 2: copy-on-write frame lifetime

Assume parent `P` maps virtual page `v` to frame `F`, and `fork` creates child `C`. Both PTEs become read-only COW mappings. The reference count is protected by `ref_lock`.

Initial post-fork state:

object	state
`P[v]`	`F`, valid, user, read-only, COW
`C[v]`	`F`, valid, user, read-only, COW
`ref[F]`	2 mapping references, no temporary pin

`P` writes `v` while `C` exits on another CPU:

step	CPU 0: parent fault	CPU 1: child exit	required invariant
1	under page-table/ref synchronization, revalidate `P[v] = F` and acquire a temporary pin; `ref[F]: 2 → 3`	read `C[v] = F`	mapping references plus temporary pins keep every accessible frame non-allocatable
2	allocate private frame `G` with one fault-owned temporary reference, `ref[G]=1`	clear `C[v]`; complete the required child TLB invalidation; drop C's mapping reference, `ref[F]: 3 → 2`	P's mapping reference and the fault pin still protect `F`; G cannot return to the allocator during the copy
3	copy bytes `F → G` while holding the temporary pin	finish child teardown	an in-flight copy owns a counted reference even if another mapping disappears
4	under page-table/ref synchronization, replace `P[v]` with writable `G`, convert G's temporary reference into its mapping reference (`ref[G]` remains 1), and drop P's old mapping reference; `ref[F]: 2 → 1`	—	the F pin substitutes for the removed mapping while a stale P translation may remain; G has exactly one published owner
5	complete P's local invalidation and any required remote shootdown; then drop the temporary pin, `ref[F]: 1 → 0`, and return `F` to the allocator	—	transition to zero occurs only after no PTE, stale translation, or in-flight copy can access `F`

A wrong implementation omits the temporary pin and drops P's mapping reference before copying. Child exit can then reduce `ref[F]` to zero, the allocator fills `F` with another page, and the parent copies unrelated bytes into `G`. A second wrong implementation takes the pin but drops it immediately after PTE replacement, before P's stale TLB entry is invalidated; a stale write can then

target allocator-owned `F`. Killing tests pause at each boundary, force reuse with a known poison page, and check both copied bytes and post-shootdown allocator ownership. A loop that merely forks and exits may never schedule either dangerous interleaving.

This trace does not prove a complete COW implementation. It requires the selected rev5–rooted design to define whether invalidation is local or requires remote shootdown, how fault retry and page-table locks interact, and how a temporary pin is represented without holding a global reference lock across the copy.

Build evidence that can kill the design

Correctness testing precedes performance measurement. If an invariant fails, treatment timing is a measurement of a different or invalid contract.

Public test contract

Publish tests and oracles for all of these applicable categories:

category	minimum evidence
upstream regression	unchanged upstream suite, complete output, and exit status
happy path	one smallest valid operation with exact post-state
boundary	zero, one, maximum supported value, wraparound, and just-over-limit behavior
invalid input	malformed identifiers, addresses, lengths, states, or permissions rejected without partial effect
exhaustion	allocator, queue, log, descriptor, or process limit reached at each fallible stage
concurrency	barriers or hooks force a named interleaving; stress is additional evidence, not the only method
failure injection	every documented injection point maps to an allowed recovered state or explicit unsupported outcome
observability	counters conserve events and do not alter the contract they measure
repeatability	fixed seeds reproduce each deterministic failure; randomized seeds are recorded on failure

Tests must inspect externally visible state or independently reconstructed invariants. Do not call a private helper solely to ask whether that helper believes its own update succeeded.

Wrong-implementation kill matrix

Name at least four plausible defects specific to the chosen mechanism. The generic examples below are prompts, not a substitute for project-specific analysis.

defect ID	wrong design	observation that can kill it
W1	publishes an object before all fields are initialized	reader barrier at publication; validate every field and authority bit
W2	assumes FIFO completion or wakeup	reorder two distinguishable operations and check identity-based completion
W3	drops a lifetime reference before the last copy, stale translation, or shootdown completes	pause at the release boundary, force reclamation, and detect poison/reuse or stale access
W4	updates two persistent locations outside one recovery unit	crash at each write boundary; accept only enumerated recovered states
W5	retries after partial effect without idempotence	inject duplicate completion or recovery replay; count effect exactly once
W6	improves a counter by omitting expensive work	compare end-state and independent conserved-event totals, not the reported counter alone
W7	acquires two resources in caller-dependent order	run opposite operations behind a barrier; detect cycle with bounded timeout and trace
W8	treats allocation failure as impossible	fail the nth allocation at every site; compare pre/post ownership and leak counts

For each row selected, record the test identifier, forced schedule or fault, oracle, result on the wrong implementation, result on the treatment, and residual blind spot. A test that passes both versions does not kill that defect.

Hidden tests and human review

An adversarial test suite may withhold exact inputs so an implementation cannot specialize to a finite public test suite. The **categories** are explicit:

- contract boundaries and just-over-limit values;
- resource exhaustion and cleanup;
- forced races and alternate CPU counts;
- crash or interrupt cuts at declared publication points;
- malformed user-controlled state;
- repeated operation, replay, and teardown;
- independent invariant reconstruction.

Hidden tests contribute evidence; they do not create secret requirements. A failure outside the published contract triggers clarification, not retroactive scope. A pass does not establish absence of bugs or authorship.

Human review examines the state model, control paths, lock/authority order, error cleanup, source provenance, meaningful test oracles, interpretation of results, and whether the author can derive an unfamiliar trace. The site and harness must not claim to automate that judgment.

Diagnostic checkpoint: a test that rewards the bug

A scheduler test passes when a new `dispatches` counter is at least 20% lower. The treatment increments the counter only on CPU 0, so it passes on two CPUs. Repair the oracle.

First define the intended claim. If the goal is fewer context switches for the same completed work, independently count dispatch events from a trace point common to baseline and treatment, and verify conservation per CPU. Hold workload completion constant, record CPU count, and compare trace-derived dispatches per completed operation. Also assert that every actual transition to running has exactly one event and that the sum of per-CPU events equals the global total. A treatment-owned counter can remain an observability output, but it cannot be its own sole oracle.

Measure after correctness gates pass

Performance evidence must identify what changed, where the expected event reduction occurs, and which cost may have moved elsewhere.

Prediction before measurement

Write a causal prediction before collecting treatment results:

```
mechanism change
  -> event expected to decrease/increase
  -> counter or trace that observes that event
  -> end-to-end metric affected
  -> competing cost or trade-off
  -> result that would reject the mechanism explanation
```

Examples of rejecting evidence include unchanged lock wait despite a claimed lock redesign, lower reported allocations with unchanged allocator traces, or throughput improvement that disappears in no-change calibration.

Trial protocol

- Preserve every valid trial, including slow and failed runs. Mark invalid trials by a rule written before seeing treatment results.
- Randomize or counterbalance baseline/treatment order so thermal drift and host load do not always favor one side.
- Use identical workload generation, build mode, CPU/memory allocation, cache-state policy, and measurement interval.
- Separate warmup from measured intervals and justify both.
- Report the raw distribution, not only a mean. Include units, trial count, central tendency, dispersion or interval, and tail metric when latency matters.
- Use paired comparisons when each treatment run has a matching baseline run under the same block/seed.
- Repeat across the independent variable the claim names, such as CPU count or queue depth. One favorable point cannot support a scaling claim.
- Record observation overhead. Tracing can serialize or perturb the path it measures.

Illustrative result trace: throughput rises, tail latency fails

The numbers below are a **modeled review example**, not measurements from xv6 or any real scheduler. Each row pairs equal workload seeds; throughput is completed bursts per second.

pair	baseline throughput	treatment throughput	baseline p99 delay (ms)	treatment p99 delay (ms)
1	101	118	4.1	10.9
2	99	117	4.3	11.4
3	100	120	4.2	11.8
4	102	119	4.0	12.1
5	98	45	4.5	97.0
6	100	118	4.2	11.6
7	100	119	4.1	11.9

The median throughput changes from 100 to 118 bursts/s, an 18% increase in this modeled sample. That does not justify "the scheduler is 18% faster":

- every treatment p99 delay is worse;
- pair 5 is a severe collapse and cannot be discarded merely because it is inconvenient;
- seven pairs provide weak information about rare stalls;
- a burst/s result does not establish CPU efficiency, fairness, or behavior under another mix.

The next action is causal diagnosis of pair 5 using preserved seed, runnable-queue trace, lock wait, and host metadata—not editing the chart. A defensible claim might eventually be "higher median throughput for this fixed CPU-bound-heavy workload, with substantially worse interactive tail latency," if repeated evidence supports it. The proposal's latency goal would still be rejected.

Application checkpoint: separate model, observation, and claim

Classify each statement.

1. "Splitting one lock into four shards permits at most four independent critical sections in the simplified lock model."
2. "Across 30 recorded trials at eight virtual CPUs, the treatment median was 1.31 times the baseline median."
3. "Sharding makes the kernel scalable."

1. **Model result.** It follows from a simplified structural model and omits key skew, cache, and remaining-lock costs.
2. **Measurement statement**, provided the raw trials, units, pairing, and configurations are available. It reports the sample without generalizing beyond it.
3. **Unsupported broad claim.** "Scalable" needs an independent variable, range, workload, metric, correctness contract, and comparison. Repair it by naming the tested CPU range and key distribution, reporting the observed curve, and limiting the conclusion to that configuration.

Analyze threats to validity

Do not append a ceremonial "more testing is needed." Connect each threat to a claim and mitigation.

threat class	capstone question	example	mitigation or bounded interpretation
construct	does the observable represent the concept named?	dispatch count is called fairness	report wait distribution or defined fairness index; narrow the claim
internal	could another difference cause the result?	treatment uses a different compiler or benchmark seed	freeze treatment boundary; pair/randomize trials; no-change calibration
external	where may the result fail to transfer?	one QEMU model stands for physical multicore hardware	state emulation boundary; test an approved second setting or restrict conclusion
conclusion	is the sample/analysis strong enough for the relation?	median reported from three unstable runs	preserve raw data, increase independent trials, report uncertainty and instability

Also audit instrumentation validity: event loss, counter overflow, timestamp source, tracing perturbation, sampling period, CPU migration, and whether the observer shares the lock it is diagnosing.

Counterexample: a clean graph with no valid comparison

Suppose the treatment benchmark is run after the filesystem cache is warmed, while the baseline begins after a fresh image boot. Thirty perfectly formatted repetitions can yield narrow intervals around a confounded difference. Repetition reduces random uncertainty; it does not remove systematic bias. The correct response is to repair the protocol and rerun both sides, not to attach a more sophisticated statistical test.

Transfer checkpoint: negative result

The mechanism preserves all tested invariants, but paired trials show no meaningful throughput change and lock traces show that contention moved to the allocator. Is the project a failure?

The performance hypothesis is rejected, but the project can still be rigorous. Preserve the result; show that the intended lock wait fell while allocator wait rose; bound the workloads and instrumentation; explain the causal bottleneck migration; and avoid claiming improvement. Correct mechanism work plus a reproducible negative evaluation and credible diagnosis can score strongly. Fabricating a win or silently changing the metric cannot.

Design memo

REMARK

The design memo is evaluated as a falsifiable engineering argument: every claimed invariant must map to an assertion or test oracle, and every performance claim must pair pre-registered event predictions with raw trial distributions.

Structure a concise memo whose structure follows the argument rather than the chronology of coding:

1. **Claim and scope:** approved claim, option, contract, exclusions, and reviewer conditions.
2. **Baseline and provenance:** immutable source/configuration, license, dependencies, assistance, and treatment boundary.
3. **State and invariants:** state variables, legal transitions, ownership, authority, lifetime, persistence, and error states.

4. **Mechanism:** control path, synchronization or transaction order, resource bounds, and alternatives rejected.
5. **Failure analysis:** one concrete race/crash/exhaustion trace, cleanup path, and residual unsupported fault.
6. **Tests:** independent oracles, public categories, wrong-implementation kill matrix, hidden-category expectations, and coverage limits.
7. **Evaluation:** pre-registered prediction, workloads, raw-data schema, analysis, observed distributions, and modeled-versus-measured labels.
8. **Threats and interpretation:** construct/internal/external/conclusion threats, mitigations, negative evidence, and bounded conclusion.
9. **Handoff:** build/test/measurement commands, patch organization, known issues, and safe extension points.

Include diagrams or state tables where they replace ambiguous prose. A screenshot of passing output without the command, exit status, and artifact identity is not useful evidence.

Verification trace and changed assumptions

> **EXAMPLE**
>
> In the interrupt-window verification prompt, if an interrupt arrives after the driver writes `avail->idx` but before the submitter sets `req.state = IN_FLIGHT`, the interrupt handler observes a completion for a slot still marked `PREPARED`. The invariant repair requires transitioning `req.state = IN_FLIGHT` while holding the queue lock *before* publishing `avail->idx`.

The verification demonstration is a controlled falsification attempt, not a feature tour. In your verification demonstration:

1. identify the exact baseline and treatment;
2. state the contract and one non-goal;
3. draw or trace the hardest state transition;
4. run one ordinary case and one adversarial case;
5. inject one fault or forced interleaving and explain the oracle before revealing the result;
6. show raw baseline/treatment evidence and one threat that limits interpretation;
7. reproduce from a fresh checkout or clean build state.

To audit robustness, evaluate system behavior under an altered assumption: CPU count, queue order, allocation outcome, crash cut, authority, or workload skew. Reason from invariants rather than recall a memorized execution sequence.

Representative analysis prompt:

> *Your queue test assumes the interrupt cannot arrive between publishing the descriptor and recording the request as in flight. Move the interrupt to that boundary. Which state can the handler observe, which invariant applies, and what code or contract change is required?*

A strong answer enumerates the concrete states, identifies the publication edge and synchronization, predicts the observable failure, and either derives why the state is safe or acknowledges a defect and proposes a discriminating test. Confidence without a trace is invalid.

Reproducible handoff

The final archive or repository state must contain:

```
README.md                 project purpose, contract, exclusions
REPRODUCE.md              pinned environment and exact commands
DESIGN.md or report.pdf   design memo and diagrams
PATCHES/ or named commits reviewable treatment boundary
tests/                    public tests, fixtures, fault hooks, oracles
results/raw/              immutable machine-readable trial outputs
results/analysis/         scripts/notebook plus generated tables
results/manifest.json     checksums, seeds, configurations, status
LICENSES/ or notices      upstream and imported-code obligations
KNOWN_ISSUES.md           residual defects, unsupported states, next
probes
ASSISTANCE.md             people, tools, models, sources, and adapted
material
```

Do not include generated binaries as the only way to reproduce a result. Do not require credentials, an undisclosed local path, or an environment-specific script. An independent engineer or automated runner should be able to determine within ten minutes how to build and start the public tests, even if a full measurement campaign takes longer.

The handoff must explain where a future engineer can safely add a state, where lock or transaction order is defined, and which tests must change with the contract. “Read the code” is not a maintenance boundary.

Handoff checkpoint: checksum mismatch

The analysis script produces the published table, but one raw file’s checksum differs from `manifest.json`. May the table remain in the final memo?

Not as verified evidence. Determine whether the raw file, manifest, or table came from a different run; preserve all versions with provenance; regenerate the table only from a clearly identified immutable dataset; and update the manifest through the documented process. If provenance cannot be recovered, label the result unverifiable and exclude it from the supported claim. A visually identical table does not repair a broken evidence chain.

Provenance and assistance policy

WARNING

Automated test passes never substitute for architectural understanding: every external source, tool, or generative model consultation must be logged in `ASSISTANCE.md` with its verification method, and you must be able to re-derive any implementation control path under a modified assumption.

Architectural mastery means being able to explain and verify every mechanism, test oracle, result, and source decision. It does not mean working without tools or feedback.

Sample ASSISTANCE.md disclosure record:

Tool / external source	Purpose of consultation	Artifact affected	What was accepted vs rejected	Independent verification method
LLM code assistant	Drafted initial Virtio descriptor chain	`kernel/virtio_disk.c`	Kept descriptor struct; rejected unbarried ring write	Verified memory fence ordering against Virtio v1.2 spec §2.7
xv6 commit log	Clarified same-inode rename semantics	`kernel/sysfile.c`	Adopted no-op return with link count 2	Wrote adversarial test asserting nlink unchanged

Allowed when disclosed:

- reading and citing primary documentation and licensed source;
- discussing mechanisms and experimental design without exchanging project patches;
- using debuggers, tracers, static analysis, fuzzers, code completion, and generative systems;
- adapting licensed code when the option permits it, attribution is explicit, and you can defend the adaptation;
- receiving review comments and recording which changes they caused.

Not allowed:

- incorporating unverified third-party patches, tests, memo, or results as original work;
- asking for, reconstructing, probing, or special-casing hidden-test inputs;
- fabricating, deleting, or relabeling trials to improve a conclusion;
- concealing assistance, copied interfaces, incompatible licenses, or known failing states;
- using generated text without verifying the underlying mechanism.

`ASSISTANCE.md` records source/tool, purpose, affected artifact, what was accepted or rejected, and how it was verified. A long undifferentiated tool transcript is not a provenance record. You should be able to rederive any assisted portion under a changed assumption.

Provenance requires falsifiable evidence. Token overlap, style, or an automated detector alone is not a finding. Conversely, passing automated checks cannot certify understanding.

Invariant and quality checklist

Verify evidence against objective engineering standards, not patch size or presentation polish alone.

dimension	invariant criteria	verification evidence
baseline, scope, and provenance	immutable baseline; licensed treatment boundary; bounded claim and exclusions	clean diff, verified git ancestry, explicit exclusions
contract, state model, and invariants	precise transitions, ownership/lifetime/persistence rules, and error states	formal state machine, invariant table covering all transitions
mechanism correctness	control paths preserve stated invariants across normal, boundary, and cleanup paths	zero contract-breaking defects, clean failure logs
adversarial tests and oracles	deterministic races/faults/exhaustion plus regression tests	test suite and wrong-implementation kill matrix
design quality	synchronization/transaction/authority order is justified	lock/interrupt ordering graph, bounded resources
reproducible evaluation	prediction, calibrated baseline, raw paired or randomized trials, units, uncertainty	automated scripts, machine-readable trial outputs
failure analysis and validity	minimized failure trace, negative evidence, four threat classes	documented failure injected and minimized
handoff quality	fresh-build path, organized patch, tests/data/checksums, known issues	single-command fresh checkout build and test
synthesis analysis	concise causal argument; adversarial verification; reasoning under changed assumptions	trace-backed analysis under altered parameters

The following are critical failure conditions that invalidate verification:

- the chosen start point does not descend reproducibly from rev5, or the capstone treatment does not build as a separable change from it;
- the start-point regression suite or contract fails without a documented scope change;
- a test exposes corruption, authority escape, use-after-free, deadlock, or recovery violation within scope;
- claimed measurements cannot be tied to raw data and configuration;
- required license/provenance is missing;
- the central implementation control path or oracle cannot be derived or explained;
- the evidence contains fabricated data or unattributed work.

A correct but negative performance result is fully valid engineering evidence. A fast implementation that violates its contract cannot trade benchmark throughput for correctness.

Synthesis analysis

Analyze system behavior under this changed requirement:

> *The accepted design was evaluated on two CPUs with uniform keys. Deployment now uses eight CPUs and 90% of operations target one key. Which part of the claim survives, what new bottleneck or race becomes plausible, and what evidence would you gather before recommending deployment?*

A complete answer separates correctness from performance. Contract and invariants may survive if they are independent of CPU count and skew, but that must be traced through all shared state; uniform-key performance does not transfer. The hot key can collapse sharding onto one lock/cache line, increase queueing, expose lifetime races at higher concurrency, or move contention to allocation/reclamation. Gather an eight-CPU correctness stress run with forced create/delete interleavings, per-shard operation and wait distributions, cache/coherence indicators available in the environment, end-to-end latency/throughput across a skew sweep, no-change calibration, and raw paired trials. Recommend deployment only within the resulting evidence boundary.

REMARK

By anchoring every project to pinned `xv6-riscv-rev5` commit `7d7adbb1b0acbd67c9766a20d0f9900fef2789fa`, the capstone trades production hardware breadth for complete mechanism inspectability across syscalls, synchronization, virtual memory, I/O, persistence, and observability.

The chosen change must connect at least three mechanism layers without expanding into three projects:

local mechanism	connection to trace explicitly
syscall or fault boundary	validation, privilege transition, user-visible error, and restart behavior
synchronization	publication edge, lock/interrupt order, sleep/wakeup, lifetime, and progress
virtual memory	translation, permission, frame ownership, TLB state, copy/pin behavior
device path	descriptor ownership, interrupt ordering, DMA/model assumptions, and completion identity
persistence	transaction boundary, write ordering, crash cut, replay, and recovered invariant
protection	principal, authority conveyed by handle, confused-deputy risk, and revocation/lifetime
observability	event definition, collection overhead, conservation, and causal link to metric

SECTION EXERCISES

Deep Practice

Solve these problems to test and consolidate your understanding. Complete worked solutions and diagnostic hints are available for each problem.

Exercise 7.6.1: Synthesizing Privilege, Memory, and File System Subsystems

Verification application

A process executes `mmap` to map an 8 KiB file into its address space with `PROT_READ | PROT_WRITE` and `MAP_SHARED`. It then modifies the first byte of the second page (`offset 4096`) and calls `msync(MS_SYNC)`. Trace how the operating system coordinates privilege transitions, virtual memory faults, the buffer cache/page cache, and disk write transactions to guarantee durability.

Trace: initial VMA creation (unmapped page table entry), first access triggering page fault, allocation of physical frame and disk block read, PTE update with dirty bit, user-mode write, and msync dirty-page flush through journal transaction.

1. **VMA registration:** During `mmap`, the kernel validates arguments and allocates a Virtual Memory Area (VMA) recording range `[addr, addr + 8192)`, permissions (`PROT_READ | PROT_WRITE`), flag `MAP_SHARED`, and backing file/inode reference. The page table entries (PTEs) remain invalid (demand paging).
2. **Page fault and cache lookup:** Accessing `offset 4096` triggers a hardware page fault trap (`scause = 13` or `15`). The kernel trap handler searches the process VMA list, discovers a valid shared mapping, and checks the page cache for file offset 4096.
3. **PTE population:** If not in page cache, a frame is allocated and populated via device driver DMA. The kernel constructs a leaf PTE pointing to this frame with `R=1, W=1, U=1, V=1` and installs it in the process page table, followed by `sfence.vma`.
4. **User-mode write:** The faulting instruction restarts in user mode, successfully modifying byte 4096 in memory. The hardware or OS tracks the dirty state on the page.
5. **Durability via msync:** When the application calls `msync(..., MS_SYNC)`, the kernel traverses the VMA page table entries, identifies dirty pages, begins a file system log/journal transaction, copies the modified block into the buffer cache, issues barrier-ordered disk writes to persistent storage, and commits the transaction before returning to user space.

Exercise 7.6.2: Tracing Cross-Subsystem Invariant Races Between VM and VFS

Construction application

Consider an asynchronous Direct I/O write (`aio_write` or `io_uring`) reading from a user memory buffer `B` and writing to disk, while another thread in the same process concurrently invokes `munmap(B, len)` or `fork()` .

1. Identify the race condition if the kernel merely validates the user virtual address `B` at submission time without pinning the underlying physical pages.
2. Describe the synchronization mechanism used by modern kernels (`get_user_pages` / page pinning and refcounting) to maintain memory safety throughout the asynchronous I/O lifecycle.

Without page pinning, munmap can free physical frames back to the page allocator, allowing disk controllers to DMA stale or reassigned physical memory.

1. **The unpinned race condition:** If the kernel validates virtual address `B` at submission and starts an asynchronous DMA transfer, but does not pin the physical frames:
   - Thread 2 executes `munmap(B, len)`. The kernel tears down the PTEs and returns the physical frames to the global page allocator.
   - The allocator reassigns those frames to an unrelated sensitive process (e.g., storing cryptographic keys or kernel stacks).
   - The storage controller executes DMA from the reassigned physical frames into the target file on disk, leaking sensitive foreign memory to persistent storage.
2. **Resolution via page pinning (`GUP`):**
   - At asynchronous I/O submission, the kernel calls `get_user_pages()`, which walks the page table, increments the reference count (`struct page->_refcount`) on each physical frame, and locks the pages against eviction.
   - Even if `munmap` destroys the user page table mappings, the physical frames cannot be reclaimed by the page allocator because their reference count remains greater than zero.
   - Once the storage controller signals DMA completion via hardware interrupt, the kernel decrements the frame reference counts (`put_page()`), allowing normal deallocation only after the hardware transfer completes.

Exercise 7.6.3: Proving End-to-End File Durability from Page Fault to Disk Platter

Proof synthesis

Let a persistent storage subsystem guarantee durability under Write-Ahead Logging (WAL) with storage cache flush commands.

1. Formulate the two write-ordering invariants required to prevent corruption across sudden power loss:
    - Invariant 1 (Log Record Order): Log descriptor and block writes must reach non-volatile media before the commit block.
    - Invariant 2 (Checkpoint Order): The commit block must reach non-volatile media before in-place data blocks are overwritten.
2. Prove that if the disk controller reorders writes across the commit block boundary (violating Invariant 1), a crash during checkpointing can leave the file system in an unrecoverable corrupted state.

Show that an out-of-order write of the commit block followed by a power cut leaves an incomplete transaction marked as committed, causing replay of corrupt or partial blocks.

1. **Formal ordering invariants:**
   - **Invariant 1** ($W_{\text{log}} \prec W_{\text{commit}}$): For any transaction T_i, all journal log block writes $L_{i,1}, \ldots, L_{i,k}$ must be completely flushed to persistent media before the commit record C_i is persisted:

$$\forall k,\ \text{flushed}(L_{i,k}) \prec \text{flushed}(C_i)$$

   - **Invariant 2** ($W_{\text{commit}} \prec W_{\text{checkpoint}}$): The commit record C_i must be completely flushed before any dirty metadata or data block is written in-place to its permanent location P_j :

$$\text{flushed}(C_i) \prec \text{flushed}(P_{i,j})$$

2. **Proof of failure under Invariant 1 violation:**
   - Suppose the disk controller reorders writes due to write-back volatile caching, writing C_i to persistent media while log payload block $L_{i,1}$ is still volatile in controller RAM.
   - Power fails immediately.
   - Upon reboot, crash recovery scans the journal log. It inspects transaction T_i and detects a valid C_i on disk, concluding that T_i was successfully committed.
   - The recovery routine replays the transaction, copying the contents of $L_{i,1}$ into permanent location P_1 .
   - However, because $L_{i,1}$ was never persisted, the sector on disk contains random garbage or stale data from an earlier epoch.
   - The file system overwrites valid in-place file system metadata (such as the root inode or allocation bitmap) with garbage, resulting in catastrophic metadata corruption that journal replay cannot rectify.

Exercise 7.6.4: Evaluating End-to-End Scalability Across Kernel Boundaries

Construction transfer

An engineer optimizes an operating system kernel by replacing a coarse-grained central spinlock on the virtual memory subsystem with fine-grained per-VMA range locks. In a single-threaded benchmark, throughput drops by 4%. In a 64-core multi-threaded benchmark with disjoint memory mappings, throughput scales 38× higher.

1. Account for the 4% single-threaded throughput regression in terms of instruction count, memory barriers, and lock acquisition overhead.
2. Explain why the multi-threaded benchmark achieves near-linear speedup, identifying the hardware cache coherence mechanisms involved.
3. Formulate the decision rule for when this kernel redesign should be accepted into production.

Contrast lock instruction overhead on uncontended paths against cache-line bouncing (MESI invalidation) on contended global locks.

1. **Single-threaded overhead:** Acquiring and releasing multiple fine-grained locks requires executing atomic instructions (e.g., `lr` / `sc` or `cmpxchg`) and memory fence instructions for each individual VMA accessed, whereas a single coarse lock executed those atomic instructions once. On an uncontended single-core execution path, these extra instructions and pipeline flushes increase execution cycles, manifesting as a 4% latency overhead.
2. **Multicore scaling mechanism:** With a global lock, all 64 cores compete for the identical 64-byte cache line containing the lock state. Under the MESI coherence protocol, every `lock acquire` issues a Read-For-Ownership (RFO) bus transaction, invalidating the cache line across all 63 other cores and stalling them on interconnect latency. With fine-grained per-VMA locks accessed by disjoint threads, each core modifies disjoint memory addresses residing in distinct cache lines in local `Exclusive` or `Modified` state, eliminating cross-core cache invalidations and allowing parallel progress.
3. **Production decision rule:** Accept the redesign if the targeted deployment environments are predominantly concurrent multi-threaded workloads on multicore hardware where the throughput gains (up to 38×) vastly outweigh the modest 4% single-thread penalty, and where memory consumption overhead from per-VMA lock structures remains within bounded system limits.

Exercise 7.6.5: Auditing the Final Operating System Trusted Computing Base

Description synthesis

At the completion of an operating system design, an architectural audit evaluates the Trusted Computing Base (TCB).

1. Define the TCB of a computer system.
2. For an xv6-based kernel running user processes, enumerate every hardware and software component included in the TCB.

3. If an attacker discovers a vulnerability in the xv6 virtio disk driver, state the maximum privilege the attacker can achieve and explain how architectural redesign (such as driver privilege separation) could restrict that blast radius.

The TCB includes all hardware, firmware, and software elements whose misbehavior can break the system security policy. In monolithic xv6, all kernel code executes in supervisor mode.

1. **TCB definition:** The Trusted Computing Base comprises the complete set of hardware, firmware, and software components that are critical to system security. If any component within the TCB contains a vulnerability or behaves maliciously, the security policy (confidentiality, integrity, availability) of the entire system can be compromised.
2. **xv6 TCB components:**
    - **Hardware:** Physical CPU core and MMU translation hardware, physical RAM, system bus, interrupt controllers (PLIC/CLINT), timer, and storage devices.
    - **Firmware:** RISC-V machine-mode bootloader (OpenSBI / BIOS) executing at M-mode.
    - **Software:** The entire xv6 kernel binary executing in supervisor mode (S-mode) —including the trap handler, scheduler, physical memory allocator, page table manager, system call dispatcher, VFS and file system, and device drivers (UART, VirtIO disk).
3. **Driver exploit blast radius and containment:**
    - In xv6, device drivers execute directly within supervisor mode with full access to kernel address space and page tables. A vulnerability in the virtio driver allows arbitrary code execution in S-mode, granting complete authority to alter kernel code, disable page protection, inspect arbitrary process memory, or bypass authentication entirely.
    - To contain this blast radius, the architecture can isolate device drivers in unprivileged user-mode address spaces (as in microkernels) mediated by an IOMMU. Under this redesign, a compromised driver can corrupt only its own user space and device buffers; it cannot access kernel memory or other processes without passing through capability-checked IPC endpoints.

References & Further Reading

- Russ Cox, Frans Kaashoek, Robert Morris, et al., xv6: a simple, Unix-like teaching operating system, MIT PDOS. Use the book with the immutable course source and license below.

- MIT PDOS, xv6-riscv-rev5 immutable source tree and license. Commit `7d7adbb1b0acbd67c9766a20d0f9900fef2789fa` is the course root; cumulative checkpoints and the capstone remain separately identified layers above it.
- USENIX, 2025 USENIX Annual Technical Conference: Call for Artifacts. Its availability, functionality, and reproducibility distinctions motivate the artifact questions; this course does not award USENIX badges.
- Tomas Kalibera and Richard Jones, Rigorous Benchmarking in Reasonable Time, ISMM 2013. Use it to reason about experimental dimensions, repetitions, effect uncertainty, and analysis—not as a ritual that rescues a confounded workload.
- Google, syzkaller: kernel fuzzer and its linked official documentation. Its crash-reproduction workflow is useful background for preserving seeds, programs, logs, and minimized failures; use only if it fits the project's kernel and scope.

These sources define baselines and methods, not a capstone solution. Cite the exact primary source beside any imported interface, code fragment, hardware contract, filesystem guarantee, or measurement method that materially shapes the verified design.

Course Practice

SECTION DETAIL

Operating Systems Practice Workshop

Problem Workshop

This workshop generates algorithmic practice problems spanning operating systems: Round-Robin scheduling timelines, multi-level page table index decomposition, inode capacity arithmetic, and Amdahl multicore scalability limits.

Recommended Workflow

1. Derive scheduling Gantt charts, virtual address bit-slices, and storage capacities independently before inspecting hints.
2. Consult progressive hints when reaching an address translation or scheduling impasse.
3. Verify derivations against the complete worked solution, checking boundary invariants and state transitions.

INTERACTIVE PRACTICE WORKSHOP

Randomized parameter generation, progressive hints, and worked derivations are available in the interactive web edition. Covered topics:

- CPU Scheduling
- Virtual Memory & Page Tables
- Concurrency & Deadlock
- Storage & File Systems

References & Further Reading

- Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau (2018). *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books.
- Russ Cox, Frans Kaashoek, Robert Morris (2025). *xv6: a simple, Unix-like teaching operating system*. MIT PDOS.
- Abraham Silberschatz, Peter B. Galvin, and Greg Gagne (2018). *Operating System Concepts* (10th ed.). Wiley.