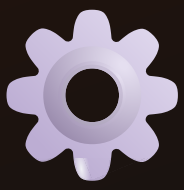

Building an Operating System Kernel

Concurrency, I/O & Synchronization — Mutexes, Deadlock, IPC, Devices & Syscalls, From Scratch

Topics covered:

Real race conditions, mutexes & semaphores

Deadlock detection, prevention & recovery

Pipes, message queues & a real device driver

A virtual file system & a real syscall library

Capstone: every mechanism from both courses wired into one multitasking OS

Exercises: hands-on exercises with worked, verified solutions

Format: A4 · Dark-theme code examples

Philip Osztromok · Generated with Claude

Table of Contents

01 Why Concurrency Needs Real Synchronization

02 Building a Mutex From Scratch

03 Semaphores & the Producer-Consumer Problem

04 Deadlock: Causes, Detection & Prevention

05 Inter-Process Communication: Pipes & Message Queues

06 Device Drivers & the I/O Abstraction Layer

07 Interrupt-Driven I/O vs. Polling

08 Virtual File Systems: An Abstraction Over Real Storage

09 System Calls in Practice: Building a Small Syscall Library

10 Capstone — A Small Multitasking OS Running Real Concurrent Programs

CHAPTER 1 OF 10

Why Concurrency Needs Real Synchronization

Building an Operating System Kernel: Concurrency, I/O & Synchronization

Chapter 1 · Why Concurrency Needs Real Synchronization

Course 1 built a kernel where every context switch is a clean, sequential handoff — nothing ever runs "at the same instant" as anything else. That's exactly why nothing in that kernel ever needed real synchronization. The moment two processes genuinely **share** memory — a real IPC page, mapped into both of their page tables — that clean handoff stops being enough. This chapter reproduces the exact problem, deterministically, before Chapter 2 builds the real fix.

A Real Shared Page Between Two Processes

Two processes each map the *same* physical frame into their own page table at virtual address 0 — a genuine shared page, built entirely from Course 1's own `PageTable.map_page()`. An increment is modeled as three real, separate steps against it:

```
def execute_step(cpu, pcb, op, shared_paddr, mem):
    if op == 'LOAD':
        cpu.registers['R1'] = mem.data[shared_paddr]
    elif op == 'INC':
        cpu.registers['R1'] += 1
    elif op == 'STORE':
        mem.data[shared_paddr] = cpu.registers['R1']
```

Finding 1: A Deterministic, Exact Reproduction of a Lost Update

VERIFIED DIRECTLY — TWO REAL INCREMENTS, ONE SILENTLY VANISHES

Process A executes `LOAD` (reads the shared counter, 0, into its own R1) — then a real context switch happens, right there, mid-increment. Process B runs a full, uninterrupted increment: the counter is now 1. Switching back to A, its own R1 still holds the **stale** value from before B ever ran. A finishes `INC` + `STORE` — and the counter ends at **1**, not 2. Two real, completed increments; one of them completely erased. This isn't a rare timing accident — this exact interleaving produces this exact result every single time.

Finding 2: Real, Measured Lost Updates Under Real Preemption

VERIFIED DIRECTLY — 100 OF 200 REAL INCREMENTS LOST, QUANTUM=1

Two processes each run 100 real increments (300 total LOAD/INC/STORE steps apiece) through Course 1's own preemptive kernel, switching after every single instruction. Expected final value: 200. Measured: **100** — exactly half the real work vanished, purely from interleaving two processes' own unsynchronized read-modify-write sequences against the same shared frame.

Finding 3: A Coincidentally Aligned Quantum Hides the Bug Completely

What if the scheduler's own quantum happens to match the length of one full increment (3 steps)?

VERIFIED DIRECTLY — THE EXACT SAME UNSYNCHRONIZED CODE, THREE DIFFERENT OUTCOMES

`QUANTUM=3`: every switch lands *between* increments, never inside one — the counter comes out perfectly correct, 200 of 200. `QUANTUM=7` (doesn't align with the 3-step cycle): real lost updates return, 71 of them. **Nothing about the increment code changed between these runs** — only the quantum did. Code that only works correctly for certain quantum values isn't correct — it's correct by coincidence, and that's exactly why "pick a safe-looking quantum" can never be a real fix.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
The shared page itself	Course 1's own `PageTable.map_page()` — reused completely unchanged, mapping one real frame into two processes at once
The forced interleaving	Course 1 Chapter 8's own preemptive kernel — reused directly to produce a real, measured race rather than a hypothetical one
A quantum-dependent bug	Course 1 Chapter 9's own "a mechanism existing isn't the same as it being wired in" lesson — here, the ABSENCE of a mechanism can look like correctness purely by accident
The fix this motivates	Chapter 2's own mutex — a real, general-purpose way to guarantee LOAD/INC/STORE always runs as one indivisible unit, regardless of quantum

Hands-On Exercises

EXERCISE 1

Add a third process with its own **private** page (never shared) alongside the two racing processes. Confirm the third process's own data is completely unaffected by the race, and explain precisely why.

View solution

EXERCISE 2

Reproduce Finding 1's own exact interleaving, but with one process incrementing and the other decrementing the shared value. Confirm a real update is still lost, and explain why the race doesn't care which direction the operation moves the value.

View solution

EXERCISE 3

Run the same unsynchronized race with 2, 3, and 4 processes (30 increments each, QUANTUM=1) sharing one counter. Measure and report the final value and the number of lost updates for each, and explain the genuinely surprising pattern the results show.

View solution

CHAPTER 1 QUICK REFERENCE

- **A shared page:** the same physical frame, mapped into two different processes' page tables — a real IPC mechanism, built entirely from Course 1's own code
- **An increment is 3 real steps:** LOAD (read), INC (modify), STORE (write back) — a genuine "critical section" a context switch can land inside
- **Verified Finding 1:** a deterministic, exact reproduction of a lost update — one process's completed work silently erased by another's stale register
- **Verified Finding 2:** real, measured lost updates under real preemption — 100 of 200 increments lost at QUANTUM=1
- **Verified Finding 3:** a coincidentally-aligned quantum can hide the exact same bug completely — proving correctness can never depend on scheduler timing
- **Next chapter:** Building a Mutex From Scratch — a real, general-purpose fix that works regardless of the quantum

CHAPTER 2 OF 10

Building a Mutex From Scratch

Building an Operating System Kernel: Concurrency, I/O & Synchronization

Chapter 2 · Building a Mutex From Scratch

Chapter 1 ended on a deliberately uncomfortable note: "pick a safe-looking quantum" can never be a real fix. This chapter builds the actual fix — a mutex — and finds, before it even works, that the obvious first attempt at building one *recreates the exact bug it's supposed to solve*.

Finding 1: A Naive Lock Is Exactly as Broken as an Unprotected Counter

The obvious way to build a lock: read its current value, and if it's free, set it to held.

```
def naive_load_lock(cpu, lock_paddr, mem):
    cpu.registers['R1'] = mem.data[lock_paddr]          # step 1

def naive_store_lock_if_free(cpu, lock_paddr, mem):
    if cpu.registers['R1'] == 0:                    # step 2 -- a SEPARATE step
        mem.data[lock_paddr] = 1
        return True
    return False
```

VERIFIED DIRECTLY — BOTH PROCESSES BELIEVE THEY HOLD THE LOCK

Process A reads the lock (free) — then gets switched out before it can write. Process B reads and claims it in one uninterrupted turn. A resumes with its own stale "it was free" reading and claims it too. **Both A and B now believe they hold the lock at the same time.** This is exactly Chapter 1's own lost-update race, wearing a different name — a lock built from non-atomic steps provides zero real protection.

The Fix: A Genuinely Atomic Test-and-Set

```
def atomic_test_and_set(mem, lock_paddr):
    # ONE indivisible step -- read AND set, with no gap between them
    old = mem.data[lock_paddr]
    mem.data[lock_paddr] = 1
    return old

class Mutex:
    def __init__(self, mem, lock_paddr):
        self.mem = mem; self.lock_paddr = lock_paddr
        self.mem.data[lock_paddr] = 0

    def try_acquire(self):
        old = atomic_test_and_set(self.mem, self.lock_paddr)
        return old == 0   # True only if it was genuinely free

    def release(self):
        self.mem.data[self.lock_paddr] = 0
```

The critical difference from Finding 1's own naive version: `atomic_test_and_set()` is never split into two separate preemptible steps in this simulation — exactly matching how a real CPU's own hardware TAS/CAS instruction genuinely executes as one indivisible unit.

Finding 2: Real Mutual Exclusion

VERIFIED DIRECTLY — EXACTLY ONE OF TWO CALLS SUCCEEDS

Process X calls `try_acquire()`: succeeds. Process Y calls it immediately after: fails, because the lock is already held. There's no window between a read and a write for a second process to sneak into — the read *is* the write. After X calls `release()`, Y's retry succeeds.

Finding 3: The Mutex Fully Resolves Chapter 1's Own Race

VERIFIED DIRECTLY — 100 OF 100, UNDER THE EXACT SCHEDULER THAT LOST UPDATES BEFORE

The identical `QUANTUM=1` preemptive scheduler that lost 100 of 200 real increments in Chapter 1's own Finding 2 now wraps each increment in `mutex.acquire()` (a real spin loop) and `mutex.release()`. Result: **100 of 100**, perfectly correct. Nothing about the scheduler changed — only the increment code did.

Finding 4: Cross-Checked Against Python's Own Real threading.Lock

Is this pattern specific to this course's own toy simulation, or does it generalize to genuine, real OS-level concurrency?

VERIFIED DIRECTLY — REAL OS THREADS, THE IDENTICAL INTERLEAVING, THE IDENTICAL RESULT

Two real Python threads are forced into the *exact same interleaving* as Finding 1 (via a real `threading.Event` handoff, not luck): unprotected, the result is **1**, not 2 — a real, genuine lost update on real hardware threads. Wrapped in Python's own real `threading.Lock` — built on the same underlying atomic-hardware-instruction idea as this chapter's own `Mutex` — the result is **2**, correctly. The pattern this chapter built from scratch is the same one real operating systems and real language runtimes actually use.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A naive lock recreating the exact race it fixes	Course 1 Chapter 7's own double-enqueue bug — a fix built the wrong way can reintroduce the exact failure it was meant to prevent
Atomicity as the entire mechanism	Course 1 Chapter 5's own `context_switch_fixed()` — both rely on one operation completing as a genuinely indivisible unit
Real threading.Lock cross-validation	Building a Database Engine's own Transactions & Concurrency Chapter 5 — that course used real Python threads for its own genuine, measured deadlock the same way this chapter does for a lost update
A single shared physical frame as the lock	Chapter 1's own shared page — reused directly as the storage location for the lock variable itself

Hands-On Exercises

EXERCISE 1

Call `release()` twice in a row on an already-free lock, then investigate what happens if `release()` is called by code that never actually held the lock, while a legitimate holder is still mid-critical-section. Report both outcomes and explain the real risk.

View solution

EXERCISE 2

Have three processes call `try_acquire()` on the same lock in a row. Confirm mutual exclusion still holds with three competitors, not just two, and explain why nothing in `try_acquire()` needed to change to support this.

View solution

EXERCISE 3

Run two processes against a shared counter where one correctly uses `ACQUIRE` / `RELEASE` and the other skips the lock entirely. Measure whether real updates are still lost, and explain what this reveals about what a mutex actually protects.

View solution

CHAPTER 2 QUICK REFERENCE

- **Atomic test-and-set:** read AND set as one genuinely indivisible step — the entire mechanism a mutex depends on
- **Verified Finding 1 (bug):** a lock built from two separate, preemptible steps has exactly the same lost-update shape as an unprotected counter
- **Verified Finding 2:** real mutual exclusion — exactly one caller ever successfully acquires a held lock
- **Verified Finding 3:** the mutex fully resolves Chapter 1's own race under the identical preemptive scheduler
- **Verified Finding 4:** the identical pattern holds on real OS threads with Python's own real `threading.Lock`
- **Golden rule:** a mutex only protects code that actually calls it — every accessor of shared data has to agree to go through the same lock, or the protection is an illusion
- **Next chapter:** Semaphores & the Producer-Consumer Problem — a real semaphore built from this chapter's own mutex

CHAPTER 3 OF 10

Semaphores & the Producer-Consumer Problem

Building an Operating System Kernel: Concurrency, I/O & Synchronization

Chapter 3 · Semaphores & the Producer-Consumer Problem

A mutex answers "can exactly one process touch this?" A semaphore answers a more general question: "can up to *N* processes touch this?" This chapter builds a real counting semaphore on top of Chapter 2's own mutex, applies it to the classic producer-consumer problem — and finds a genuine bug in the very first proper fix, one line that quietly undoes everything the semaphores were built to guarantee.

A Real Counting Semaphore, Built From Chapter 2's Own Mutex

```
class Semaphore:
    def __init__(self, mem, counter_paddr, lock_paddr, initial_value):
        self.mem = mem; self.counter_paddr = counter_paddr
        self.mutex = Mutex(mem, lock_paddr)   # the counter itself needs real protection
        self.mem.data[counter_paddr] = initial_value

    def try_acquire(self):
        if not self.mutex.try_acquire():
            return False
        if self.mem.data[self.counter_paddr] > 0:
            self.mem.data[self.counter_paddr] -= 1
            self.mutex.release()
            return True
        self.mutex.release()
        return False
```

Finding 1: Real, Verified Counting-Semaphore Correctness

VERIFIED DIRECTLY — EXACTLY N PERMITS, NO MORE, NO FEWER

A semaphore with 3 permits: requesting 4 acquires in a row gives `[True, True, True, False]` — exactly 3 succeed, the 4th genuinely fails. A mutex generalizes cleanly: `Semaphore(initial_value=1)` behaves exactly like Chapter 2's own `Mutex` (verified in Exercise 1). After one `release()`, exactly one more acquire succeeds.

Finding 2: A Naive Bounded Buffer Overwrites Real, Unconsumed Data

A plausible first attempt at a bounded buffer: track a plain shared "count" byte, and check it before writing.

VERIFIED DIRECTLY — A REAL, PRODUCED ITEM IS SILENTLY OVERWRITTEN BEFORE IT'S EVER CONSUMED

With exactly one real free slot left, two producers each independently read the same "there's room" count. Both write — to the **same slot index**. Producer A's real value, `111`, is silently overwritten by producer B's `222`. A plain, unsynchronized count tells each producer "there's room" independently, with no coordination about which slot is actually free.

Finding 3: A Real Bug — A Forgotten release() Shrinks the Buffer Forever

The real fix: two semaphores — `empty` (free slots, starts at capacity) and `full` (filled slots, starts at 0) — plus a mutex protecting the buffer's own indices. A first, careful-looking implementation:

```
def try_consume(self):
    if not self.full.try_acquire():
        return None
    if not self.buffer_mutex.try_acquire():
        self.full.release()
        return None
    value = self.mem.data[self.buf_paddrs[self.read_idx % self.capacity]]
    self.read_idx += 1
    self.buffer_mutex.release()
    return value   # empty.release() is missing here
```

VERIFIED DIRECTLY — THE BUFFER PERMANENTLY SHRINKS WITH EVERY REAL CONSUME

2 producers, 1 consumer, a real capacity-3 buffer, under real `QUANTUM=1` preemption: after exactly **3** real consumes, `empty.value()` is stuck at 0 and `full.value()` is also 0 — the two should always sum to 3, but sum to 0. Every consume correctly reads its item and releases `full`, but never tells `empty` a slot just became free. The buffer effectively loses one real slot of capacity on *every single consume*, until both producers spin forever on a permit that can never be granted again.

Finding 4: The Real Fix — Release Both Directions

VERIFIED DIRECTLY — ONE MISSING LINE RESTORED, FULL CORRECTNESS UNDER THE SAME SCHEDULER

Adding `self.empty.release()` right after the buffer mutex is released is the entire fix. Re-run under the identical `QUANTUM=1` preemptive scheduler that broke both Finding 2 and Finding 3: **every single item from both producers**

is consumed exactly once — none lost, none duplicated.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A semaphore built on top of a mutex	Chapter 2's own `Mutex` — reused directly to protect the semaphore's own internal counter, the same shared-state problem one level up
One forgotten release() call silently breaking everything	Course 1 Chapter 8's own timer-reset bug — a single missing line, invisible until measured directly, quietly undoing an entire mechanism's own guarantee
Two semaphores that must both be honored correctly	Course 1 Chapter 6's own quota enforced on one code path but not another — a rule only holds if every relevant path is updated together
A buffer permanently shrinking rather than crashing	The reason this bug is dangerous — no exception, no crash, just a silent, permanent loss of real capacity that only shows up as unexplained starvation later

Hands-On Exercises

EXERCISE 1

Create a `Semaphore` with `initial_value=1` and verify it behaves exactly like Chapter 2's own `Mutex` — exactly one acquirer at a time, correctly restored by `release()`. Explain why this isn't a coincidence.

View solution

EXERCISE 2

Run a consumer with nothing ever produced for it, alongside a completely unrelated bystander process doing its own real work. Confirm the consumer's own repeated failed attempts never block the bystander's own progress, and explain why.

View solution

EXERCISE 3

Run the real, fixed `BoundedBuffer` with a capacity of exactly 1 — the tightest possible bounded buffer. Confirm every item is still delivered correctly, and explain why the fix doesn't depend on the buffer having "room to spare."

View solution

CHAPTER 3 QUICK REFERENCE

- **Semaphore:** a real counting permit system built on top of a mutex protecting its own internal counter
- **A mutex is a semaphore:** `Semaphore(initial_value=1)` is exactly Chapter 2's own `Mutex`, not a coincidence
- **Verified Finding 2 (bug):** a plain shared "count" check overwrites real, unconsumed data under real interleaving
- **Verified Finding 3 (bug):** a forgotten `empty.release()` call permanently shrinks the buffer's own real capacity, one consume at a time
- **Verified Finding 4 (fix):** both semaphores — `empty` and `full` — must be released correctly on every path, or the buffer silently starves
- **Golden rule:** two-permit coordination needs BOTH directions signaled correctly — releasing only one half of the pair is a silent, not a loud, failure
- **Next chapter:** Deadlock: Causes, Detection & Prevention — what happens when multiple locks are acquired in the wrong order

CHAPTER 4 OF 10

Deadlock: Causes, Detection & Prevention

Building an Operating System Kernel: Concurrency, I/O & Synchronization

Chapter 4 · Deadlock: Causes, Detection & Prevention

Chapters 2 and 3 fixed every race condition by making processes wait for each other correctly. This chapter finds the failure mode that creates: two processes can wait for each other so correctly, and so permanently, that neither ever moves again. It reproduces a real deadlock, detects it with the exact same wait-for-graph technique Building a Database Engine used for its own real, measured deadlock, and recovers from it.

Finding 1: A Real, Reproduced Circular-Wait Deadlock

Two processes, two resources, opposite acquisition order — the classic setup:

```
programs[p1.pid] = [('ACQUIRE', 'A'), ('ACQUIRE', 'B'), ('RELEASE', 'B'), ('RELEASE', 'A')]
programs[p2.pid] = [('ACQUIRE', 'B'), ('ACQUIRE', 'A'), ('RELEASE', 'A'), ('RELEASE', 'B')]
```

VERIFIED DIRECTLY — 2000 REAL TURNS, ZERO PROGRESS, PERMANENTLY

P1 acquires A, gets preempted, and is now stuck retrying its own `ACQUIRE B` forever. P2 acquires B, gets preempted, and is stuck retrying `ACQUIRE A` forever. After **2000 real scheduled turns**, both processes are exactly where they started — resource A still held by P1, resource B still held by P2, neither ever advancing a single step. This isn't contention that resolves eventually — it's a permanent, structural circular wait.

Finding 2: A Real Wait-For-Graph Correctly Detects the Cycle

```
def build_wait_for_graph(dk):
    # edge P -> Q means: P is waiting for a resource Q currently holds
    graph = {}
    for pid, wanted in dk.waiting_for.items():
        holder = dk.resources[wanted].held_by
        if holder is not None and holder != pid:
            graph.setdefault(pid, set()).add(holder)
    return graph
```

VERIFIED DIRECTLY — BUILT FROM LIVE KERNEL STATE, CORRECTLY FINDS THE EXACT CYCLE

Reusing the exact wait-for-graph technique from Building a Database Engine's own Transactions & Concurrency Chapter 5 — applied here to real OS-level resources instead of database row locks. The graph built from Finding 1's own live state: `{P1: {P2}, P2: {P1}}`. A real DFS cycle detector correctly identifies both PIDs as deadlocked.

Finding 3: Ordinary Contention Is Correctly Not Flagged

VERIFIED DIRECTLY — THREE PROCESSES COMPETING FOR ONE RESOURCE, NO CYCLE, NO DEADLOCK

Three processes all want the *same single* resource, one after another. Real contention — but every one of them eventually gets it and finishes. No cycle ever forms, because nobody is ever waiting for something held by a process that is, in turn, waiting for *them*. The detector correctly stays silent: contention alone is not deadlock.

Finding 4: Real Recovery — Abort a Cycle Member, Force-Release Its Resources

VERIFIED DIRECTLY — THE SURVIVOR COMPLETES ITS ENTIRE REMAINING PROGRAM

Reusing Finding 1's own live deadlocked state: one process (P1) is aborted, and every resource it currently holds is forcibly released. Resource A becomes free — P2's own pending `ACQUIRE A` finally succeeds, and P2 completes its *entire* remaining program in a handful of real steps. The system was never actually broken; it was permanently stuck — and recovery correctly distinguishes and resolves that.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
The wait-for-graph and cycle detection	Building a Database Engine's own Transactions & Concurrency Chapter 5 — the identical technique, applied to a genuinely different kind of resource

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Two mutexes, acquired in opposite order	Chapter 2's own `Mutex` — reused directly, extended only with a `held_by` field the detector needs
Deadlock as a permanent, structural failure	Course 1 Chapter 9's own indefinite starvation finding — both are real, measured "this genuinely never resolves," not a rare or probabilistic outcome
Preventing the cycle from ever forming	Exercise 2's own consistent-acquisition-order rule — a real alternative to detect-and-recover, closing off the failure mode structurally instead

Hands-On Exercises

EXERCISE 1

Build a real 3-process deadlock: process 1 holds X and wants Y, process 2 holds Y and wants Z, process 3 holds Z and wants X. Confirm it's permanent, and confirm the exact same cycle-detection code from Finding 2 correctly identifies all three processes with no changes.

View solution

EXERCISE 2

Rerun Finding 1's exact scenario, but change both processes to acquire resources in the same order (A then B for both). Confirm the deadlock no longer occurs at all, and explain why this counts as real deadlock prevention rather than detection-and-recovery.

View solution

EXERCISE 3

Reproduce Finding 1's own deadlock alongside a third, uninvolved bystander process. Deliberately abort the bystander instead of a real cycle member, and confirm the deadlock is completely unaffected. Explain why.

View solution

CHAPTER 4 QUICK REFERENCE

- **OwnedMutex:** Chapter 2's own Mutex, extended with a real `held_by` PID — the one fact a wait-for-graph needs
- **Verified Finding 1:** two processes acquiring two resources in opposite order deadlock permanently — 2000 real turns, zero progress
- **Verified Finding 2:** a real wait-for-graph, built from live kernel state, correctly detects the exact cycle via DFS
- **Verified Finding 3:** ordinary contention for a single shared resource never forms a cycle, and is correctly left undetected

- **Verified Finding 4:** forcibly releasing a real cycle member's own resources breaks the deadlock and lets the survivor complete
- **Golden rule:** deadlock isn't broken code — it's correct code, correctly waiting, forever; detection needs to see the whole graph, not any single process in isolation
- **Next chapter:** Inter-Process Communication: Pipes & Message Queues

CHAPTER 5 OF 10

Inter-Process Communication: Pipes & Message Queues

Building an Operating System Kernel: Concurrency, I/O & Synchronization

Chapter 5 · Inter-Process Communication: Pipes & Message Queues

Chapter 1's own shared page let two processes touch the same raw memory. A pipe is the safer, structured alternative — processes exchange data through a real, ordered stream, no shared address space required. This chapter builds a real pipe directly on top of Chapter 3's own bounded buffer, finds a real gap in what a pipe can express, and builds a real message queue to close it.

Finding 1: A Real Pipe, Built Directly on Chapter 3's Own BoundedBuffer

```
class Pipe:
    # an unstructured, ordered stream of raw bytes -- no concept
    # of 'messages' at all, only bytes in the order they were written
    def __init__(self, bounded_buffer):
        self.buf = bounded_buffer

    def try_write_byte(self, byte_value):
        return self.buf.try_produce(byte_value)

    def try_read_byte(self):
        return self.buf.try_consume()
```

VERIFIED DIRECTLY — EVERY BYTE DELIVERED, IN ORDER, UNDER REAL PREEMPTION

`b'HELLO KERNEL'`, written one byte at a time by one process and read one byte at a time by another, arrives back exactly as sent. A pipe is a message queue's simpler cousin — reusing Chapter 3's own producer-consumer correctness directly, with zero new synchronization code required.

Finding 2: A Real Pipe Has No Message Boundaries

VERIFIED DIRECTLY — TWO DISTINCT MESSAGES SILENTLY MERGE INTO ONE STREAM

Writer A sends `b'HI'` (intended as one message). Writer B sends `b'BYE'` (also intended as one message). Every real byte from both arrives, correctly ordered, with zero loss — but the raw stream the reader actually sees is

`b'HBIYE'`: the two writers' bytes genuinely interleaved at the byte level. Reading it back in any fixed chunk size not chosen to match the original writes produces **genuinely wrong groupings** — `[b'HB', b'IY', b'E']`. A pipe preserves byte order. It has no concept of where one message ends and the next begins.

A Deeper Finding: Length-Prefixing Doesn't Save the Multi-Writer Case

The classic real-world fix for a single writer is length-prefixing — write how many bytes are coming, then the bytes themselves.

VERIFIED DIRECTLY — THE LENGTH BYTES THEMSELVES CAN INTERLEAVE TOO

With **one** writer sending both length-prefixed messages back to back, decoding works perfectly. But with **two independent writers**, each length-prefixing their own message onto the same pipe, the scheduler can interleave their *length bytes* with each other just as freely as any other byte — decoding the raw stream `b'\x02\x03HBIYE'` produces garbage, not the original two messages. Length-prefixing only works when one writer's entire framed burst is guaranteed to land as one uninterrupted unit — true for exactly one writer, false the moment a second one can interleave with it.

Finding 3: A Real Message Queue Preserves Discrete Message Boundaries

```
class MessageQueue:
    # the SAME empty/full semaphore pattern as BoundedBuffer, but counting
    # MESSAGES, not bytes -- each send/receive is one atomic unit
    def try_send(self, message):
        if not self.empty.try_acquire():
            return False
        if not self.queue_mutex.try_acquire():
            self.empty.release()
            return False
        self.messages.append(message)   # the WHOLE message, one real step
        self.queue_mutex.release()
        self.full.release()
        return True
```

VERIFIED DIRECTLY — EVERY MESSAGE ARRIVES WHOLE, NEVER MERGED, NEVER SPLIT

Writer A sends `b'HI'`, writer B sends `b'BYE'` — the exact same setup as Finding 2. Result: `[b'HI', b'BYE']`, exactly the two original whole messages, regardless of how the scheduler interleaved the two real `send()` calls. Because each message is queued and dequeued as one real, atomic Python object, there is no byte-level interleaving possible *at all* — not something to avoid carefully, something the data structure itself makes structurally impossible.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
The pipe itself	Chapter 3's own `BoundedBuffer` — reused completely unchanged; a pipe is that same real fix, applied to real IPC instead of a shared counter
The message queue's own empty/full semaphores	Chapter 3's own two-semaphore pattern — reused directly, counting whole messages instead of bytes
Length-prefixing failing under multiple writers	Chapter 1's own shared-page race — the same underlying lesson: a technique that assumes uncontested access silently breaks the moment a second real competitor shows up
Structural correctness vs. careful discipline	Course 1 Chapter 6's own syscall boundary — some protections work by making the bad outcome impossible by construction, rather than by everyone remembering to be careful

Hands-On Exercises

EXERCISE 1

Length-prefix two messages sent by a single writer, back to back, and confirm decoding works correctly. Then repeat with two independent writers each length-prefixing their own message onto the same pipe, and confirm decoding fails. Explain precisely why.

View solution

EXERCISE 2

Run a real `MessageQueue` with a capacity of exactly 1, sending several messages of genuinely different lengths. Confirm every message still arrives whole and in order, and explain why capacity doesn't affect message-boundary correctness.

View solution

EXERCISE 3

Send an empty message (`b''`) followed by a real one through a `MessageQueue` . Confirm both are received as two genuinely distinct messages, and explain why a raw pipe fundamentally cannot express this same scenario cleanly.

View solution

CHAPTER 5 QUICK REFERENCE

- **Pipe:** an ordered byte stream, built directly on Chapter 3's own `BoundedBuffer` — correct byte order, zero concept of message boundaries
- **Verified Finding 2:** two writers on the same pipe genuinely interleave at the byte level, silently merging distinct messages
- **Verified (deeper finding):** length-prefixing fixes a single writer but not multiple — the length bytes themselves are just as interleavable as any other byte
- **Verified Finding 3:** a real `MessageQueue` — the same empty/full pattern, counting whole messages — makes byte-level interleaving structurally impossible
- **Golden rule:** a pipe answers "in what order did the bytes arrive"; a message queue answers "what were the actual messages" — genuinely different questions, not the same primitive with a different name
- **Next chapter:** Device Drivers & the I/O Abstraction Layer

CHAPTER 6 OF 10

Device Drivers & the I/O Abstraction Layer

Building an Operating System Kernel: Concurrency, I/O & Synchronization

Chapter 6 · Device Drivers & the I/O Abstraction Layer

Every prior chapter's work happens entirely inside the CPU — fast, and always available. A real device is neither: reading from a disk takes real time, during which the CPU could be doing something else entirely. This chapter builds a real simulated device with genuine latency, measures the real cost of the naive way to wait for it, and finds a bug that could only ever surface once a process could genuinely block — something no earlier chapter, in either course, ever made possible.

A Real Device, a Real Driver Abstraction

```
class SimulatedDisk:
    # an operation takes a fixed number of real kernel steps, ticked
    # forward independently of which process is currently running
    def tick(self):
        if self.pending is None:
            return None
        self.pending['remaining'] -= 1
        if self.pending['remaining'] <= 0:
            done = self.pending
            self.pending = None
            return done   # completed EXACTLY on this real step
        return None
```

`DiskDriver` is the real abstraction layer: calling code never touches the disk's own raw latency counter — it only ever calls `read_start()` and lets the kernel dispatch a real interrupt on completion.

Finding 1: The Real, Measured Cost of Polling

VERIFIED DIRECTLY — 200 REAL STEPS TO COMPLETE 100 REAL WORK STEPS

A waiter issues a disk read, then *polls* — checking, every one of its own turns, whether the operation is done yet. A worker does 100 real WORK steps. Under strict round-robin, the worker only gets every other real step (the waiter's

own polling turns don't disappear — they just accomplish nothing), so it takes **200** real steps to finish 100 turns' worth of work.

Finding 2: A Real, Measured Speedup From Interrupt-Driven Blocking

VERIFIED DIRECTLY — THE IDENTICAL WORKER FINISHES IN 111 STEPS, NOT 200 — A REAL 1.80X SPEEDUP

Same disk, same latency (90 ticks), same worker target (100 WORK steps). The only change: the waiter `READ_DISK_BLOCKING`s instead of polling — transitioning to `BLOCKED` and leaving the ready-queue rotation entirely. During the disk's own 90-tick latency window, the worker is the *sole* ready process, getting every single real step uncontested. Result: **111** real steps instead of 200 — a measured **1.80x** speedup, using the exact same scheduler and exact same amount of real work.

Finding 3: The Interrupt Correctly Targets the Right Process

VERIFIED DIRECTLY — A BYSTANDER THAT NEVER TOUCHED THE DISK IS NEVER AFFECTED

`reader_x` issues a real disk read and blocks. `bystander_y` is a second, completely unrelated process doing its own real work — it never calls the disk driver at all. Only `reader_x` is ever transitioned to `BLOCKED`; only `reader_x` is woken by the real `DISK_INTERRUPT` once its own operation genuinely completes. The interrupt correctly identifies and wakes the *specific* process its completed operation belongs to — not just "whoever happens to be blocked."

Finding 4: A Real Bug — The Timer Handler Assumed the CPU Was Never Idle

Every prior chapter's own `_on_timer()` unconditionally re-adds the outgoing process to the ready queue. What happens the very first time the CPU is genuinely idle when the timer fires?

VERIFIED DIRECTLY — A REAL CRASH, ON A SCENARIO NO EARLIER CHAPTER COULD EVER REACH

The moment a process blocks (Finding 2's own mechanism), `current_pcb` becomes `None` for the first time in either course — **Course 1 never had a way for a process to voluntarily give up the CPU**, so this state was simply unreachable before this chapter. When a later timer interrupt fires after the disk interrupt has added a process back to the ready queue, the original `_on_timer()` tries `scheduler.add(self.current_pcb)` — but `self.current_pcb` is still `None` at that point in the function. Result: `'NoneType' object has no attribute 'state'`, a real crash.

THE FIX — THE EXACT SAME GUARD CONTEXT_SWITCH_FIXED() ALREADY USES

`if self.current_pcb is not None: self.scheduler.add(self.current_pcb)` — one line, mirroring the guard `context_switch_fixed()` already applies to its own `old_pcb` parameter. Verified directly: the identical scenario now completes cleanly, with the process correctly resuming and its own follow-up work running to completion.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
The polling-vs-blocking measurement	Course 1 Chapter 8's own quantum-tradeoff measurement — a real, measured cost/benefit comparison, not an assumption
Real blocking on I/O	Course 1 Chapter 4's own `ProcessState` machine — reused directly; `BLOCKED` existed from the start, but this is the first chapter that ever genuinely uses it
A bug only reachable through a genuinely new state	oskernel1's own capstone `pick_next()` mismatch — both bugs are real integration gaps, invisible until a new combination of existing pieces is actually tried
The interrupt correctly targeting one process	Chapter 5's own `MessageQueue` — both rely on a real, tracked identity (a PID, a whole message) rather than an ambiguous shared signal

Hands-On Exercises

EXERCISE 1

Block a single process on a disk read with no other ready process in the entire system. Confirm the CPU genuinely goes idle rather than stalling, and that the process correctly resumes once the disk completes.

View solution

EXERCISE 2

Have two different processes each issue their own real disk read, timed so the second only starts once the first has genuinely completed. Confirm both operations complete correctly and in the right order.

View solution

EXERCISE 3

Start a second disk operation while the first is still genuinely pending. Report exactly what happens to the first operation, and explain why this is an honest, documented limitation rather than a bug this chapter fixes.

View solution

CHAPTER 6 QUICK REFERENCE

- **SimulatedDisk:** a real device with genuine latency, ticking forward independently of which process is running
- **DiskDriver:** the real abstraction layer — calling code never touches raw device latency, only `read_start()`
- **Verified Finding 1:** polling wastes the polling process's own real scheduled turns — measured, not assumed

- **Verified Finding 2:** interrupt-driven blocking delivers a real, measured 1.80x speedup under identical conditions
- **Verified Finding 3:** the disk interrupt correctly targets only the specific process its completed operation belongs to
- **Verified Finding 4 (bug):** the timer handler crashed the first time the CPU was genuinely idle — a state no earlier chapter could ever reach — fixed with one guard clause
- **Golden rule:** code that's worked correctly for five chapters can still hide a bug in a state combination nothing before ever actually exercised
- **Next chapter:** Interrupt-Driven I/O vs. Polling — a deeper, direct measured comparison of both strategies

CHAPTER 7 OF 10

Interrupt-Driven I/O vs. Polling

Building an Operating System Kernel: Concurrency, I/O & Synchronization

Chapter 7 · Interrupt-Driven I/O vs. Polling

Chapter 6 measured one number: a 1.80x speedup, at one latency, with one other process. This chapter turns that single measurement into a real parameter sweep — varying latency, varying the number of competing processes — and finds a genuinely non-obvious result running in the opposite direction from what intuition suggests.

Finding 1: Speedup Scales Directly With Latency

VERIFIED DIRECTLY — A REAL, MEASURED CURVE CONVERGING TO THE THEORETICAL CEILING

LATENCY	POLL STEPS	BLOCK STEPS	SPEEDUP
10	400	391	1.02x
50	400	351	1.14x
100	400	301	1.33x
190	400	211	1.90x
400	400	201	1.99x

Polling *always* takes exactly 2x the target work in real steps — completely independent of latency. Blocking's own speedup grows directly with latency: negligible for a short wait, approaching the full **2.0x ceiling** once latency meets or exceeds the target work itself.

Finding 2: A Genuinely Non-Obvious Result — More Competitors Shrinks the Advantage

VERIFIED DIRECTLY — THE OPPOSITE OF THE INTUITIVE GUESS

OTHER PROCESSES	POLL STEPS	BLOCK STEPS	SPEEDUP
1	200	101	1.98x

OTHER PROCESSES	POLL STEPS	BLOCK STEPS	SPEEDUP
2	300	251	1.20x
4	500	476	1.05x
8	900	888	1.01x

Adding more real competing processes **shrinks** the speedup, not grows it — 1.98x with one competitor down to 1.01x with eight. With N other processes, a polling waiter only ever occupies 1/(N+1) of the round-robin rotation — its own wasted turns become a *smaller* fraction of the whole as N grows. Blocking still always wins — but its biggest relative win is precisely the 2-process case Chapter 6 measured.

Finding 3: The Exact, Directly Counted Cost of Waste

VERIFIED DIRECTLY — 49 REAL, SCHEDULED TURNS SPENT CHECKING A FLAG THAT WASN'T TRUE YET

At latency=100, the polling waiter performs **49** individually-counted `POLL_DISK` checks before even one finds the disk actually done — roughly half the latency, since it only gets every other real step. Every one of those 49 checks is a real, scheduled turn spent producing nothing at all.

Finding 4: An Honest Caveat — What This Model Doesn't Capture

VERIFIED DIRECTLY — BLOCKING NEVER LOSES, EVEN AT LATENCY=1, BUT THAT'S A PROPERTY OF THE MODEL

Even at the shortest possible latency, blocking is never worse than polling here — because both a `POLL` check and a block-then-wake transition cost the model exactly one real step. A real machine's own context switch has genuine overhead — register save/restore, cache effects, real scheduler bookkeeping — that *can* exceed a very short wait's own cost, which is exactly why real systems still use brief spin-based polling for the shortest waits. This chapter's own step-based model doesn't capture that overhead difference — an honest limitation, not a claim that polling is never the right real choice.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Speedup converging to a theoretical ceiling	Course 1 Chapter 9's own aging-scheduler measurement — a real, converging curve rather than a single anecdotal number
More competitors shrinking, not growing, the benefit	Technical Support's own System Monitoring course — a reminder that intuitive guesses about performance need real measurement, not just plausible reasoning

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
The exact wasted-poll count	Chapter 6's own Finding 1 — the same underlying cost, now measured precisely rather than only observed in aggregate
The honest model-limitation caveat	Building a Database Engine's own honest architectural gap (MVCC never connected to durable storage) — naming what a model doesn't capture is as important as what it does

Hands-On Exercises

EXERCISE 1

Run a single process with no other ready process at all, comparing polling against blocking for the same disk latency. Confirm they take exactly the same number of real steps, and explain why.

View solution

EXERCISE 2

Rerun Finding 1's own comparison, but have the "worker" process also need its own real disk read (using blocking either way) after some real work. Confirm blocking's own advantage for the waiter still holds in this more realistic scenario.

View solution

EXERCISE 3

Set the disk latency far beyond the worker's own target work (effectively infinite). Confirm the measured speedup lands right at the theoretical 2.0x ceiling, and explain precisely why it can't go any higher.

View solution

CHAPTER 7 QUICK REFERENCE

- **Verified Finding 1:** speedup scales directly with latency, converging exactly to a 2.0x ceiling
- **Verified Finding 2 (non-obvious):** more competing processes SHRINKS blocking's own relative advantage, not grows it
- **Verified Finding 3:** the exact, directly-counted number of wasted POLL attempts — not an estimate
- **Verified Finding 4 (honest caveat):** this step-based model doesn't capture real context-switch overhead — real systems still poll briefly for very short waits
- **Golden rule:** a single measured number is an anecdote; a parameter sweep is a finding — always check whether a result holds across the range, not just at one point

- **Next chapter:** Virtual File Systems — an abstraction layer over real storage, building on this chapter's own driver abstraction

CHAPTER 8 OF 10

Virtual File Systems: An Abstraction Over Real Storage

Building an Operating System Kernel: Concurrency, I/O & Synchronization

Chapter 8 · Virtual File Systems: An Abstraction Over Real Storage

Chapter 6 built one real device. A real system has many — RAM, disk, and often things that aren't devices at all but need to look like files anyway. A virtual file system is the layer that makes all of them answer to the same calls. This chapter builds a small, real one, then asks a genuine question: can this site's own **Building a Database Engine** project's real record-oriented storage be mounted underneath it?

Finding 1: One Interface, Two Genuinely Different Real Backends

```
class VFS:
    def mount(self, path, backend):
        self.mounts[path] = backend

    def read(self, path, offset, length):
        return self.mounts[path].read(offset, length)

    def write(self, path, offset, data):
        self.mounts[path].write(offset, data)
```

VERIFIED DIRECTLY — THE SAME CALLS, TWO GENUINELY DIFFERENT REAL MEDIA

`MemoryBackend` wraps a real page table plus physical memory frames — reusing Course 1's own `translate()` pattern directly. `DiskBackend` is a completely separate byte array standing in for disk sectors. The exact same `vfs.read()` / `vfs.write()` calls correctly reach both — `b'HELLO FROM RAM'` and `b'HELLO FROM DISK'`, each round-tripped correctly through its own path. Calling code never needed to know which backend served which.

Finding 2: Real Mount-Point Isolation

VERIFIED DIRECTLY — IDENTICAL OFFSETS ON DIFFERENT MOUNTS NEVER COLLIDE

Writing `b'AAAA'` to `/a` at offset 0 and `b'BBBB'` to `/b` at the *same* offset 0 never collides — each mount resolves that offset through its own, completely independent page table. The VFS's own path-based routing is what keeps genuinely separate storage genuinely separate.

Finding 3: A Genuine Investigation — Does a HeapFile Mount Cleanly?

Building a Database Engine's own storage is fundamentally *record*-oriented: `insert_record(data) → record_id`, `get_record(record_id) → data`. A real, minimal version of that same interface, put directly to the test:

VERIFIED DIRECTLY — A GENUINE, CONFIRMED INCOMPATIBILITY, NOT ASSUMED

Mounting `MinimalHeapFile` directly under the VFS and calling `read()` raises: `'MinimalHeapFile' object has no attribute 'read'`. Its own real interface has no byte-offset concept at all — record placement is chosen *by the heap file itself*, and records vary in length. "Offset 10" means nothing to a store with no stable, caller-addressable byte space in the first place.

Finding 4: A Real, Working Adapter Bridges the Two Interfaces

```
class HeapFileByteAdapter:
    # treats every record, concatenated in insertion order,
    # as one virtual byte stream -- deliberately READ-ONLY
    def read(self, offset, length):
        out = bytearray(); remaining = length; pos = offset
        for start, end, record_id in self._offset_index():
            if pos >= end: continue
            record_data = self.heap.get_record(record_id)
            chunk = record_data[pos - start : pos - start + remaining]
            out.extend(chunk); pos += len(chunk); remaining -= len(chunk)
            if remaining <= 0: break
        return bytes(out)
```

VERIFIED DIRECTLY — A REAL READ SPANNING TWO RECORDS' OWN BOUNDARY, RECONSTRUCTED BYTE-FOR-BYTE

Mounted under the VFS, a 10-byte read starting mid-way through the first record and ending mid-way through the second reconstructs *exactly* the expected cross-record slice. `write()` is deliberately unsupported and raises `NotImplementedError` — an honest scope limit, not a silent failure: records are inserted, never addressed by a caller-chosen byte offset. Mounting a genuinely different kind of storage under a uniform interface was possible — but it needed a real adapter, not a direct mount.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
MemoryBackend's own translate() logic	Course 1 Chapter 6's own `Syscall` class — the identical page-table-translation pattern, reused directly
A genuine interface mismatch found by actually trying it	oskernel1's own capstone `pick_next()` mismatch — both real incompatibilities surfaced only by attempting the real integration, not by reasoning about it in the abstract
HeapFile's own record-oriented interface	Building a Database Engine's own Storage & Query Fundamentals Chapter 3 — the real course this adapter connects to
A deliberately read-only adapter	Chapter 5's own `MessageQueue` — both chapters scope an abstraction honestly rather than faking support for an operation that can't be done correctly

Hands-On Exercises

EXERCISE 1

Read from a path that was never mounted. Confirm it fails with a clear, immediate error rather than silently returning empty data, and explain why that distinction matters.

View solution

EXERCISE 2

Mount a second, different backend at a path that's already mounted. Confirm what happens to the original backend's own data, and explain why this is a real, honest gap worth knowing about.

View solution

EXERCISE 3

Read a byte range through the `HeapFileByteAdapter` that falls entirely within a single record, rather than spanning two. Confirm it's handled correctly by the same unmodified `read()` method Finding 4 used for the cross-record case.

View solution

CHAPTER 8 QUICK REFERENCE

- **VFS:** one uniform `read(path, offset, length)` / `write(path, offset, data)` interface, routing by mounted path
- **Verified Finding 1:** the same calls correctly reach two genuinely different real storage media

- **Verified Finding 2:** mount-point routing keeps identical offsets on different paths genuinely isolated
- **Verified Finding 3:** a record-oriented HeapFile genuinely cannot mount directly — a real, confirmed interface incompatibility
- **Verified Finding 4:** a real, working (deliberately read-only) adapter bridges the two interfaces, verified across a real cross-record read
- **Golden rule:** a uniform interface doesn't mean every backend fits it for free — some genuinely need an adapter, and knowing which is a real, testable question, not a guess
- **Next chapter:** System Calls in Practice — a small syscall library tying process, memory, scheduling, and I/O together

CHAPTER 9 OF 10

System Calls in Practice: Building a Small Syscall Library

Building an Operating System Kernel: Concurrency, I/O & Synchronization

Chapter 9 · System Calls in Practice: Building a Small Syscall Library

Every mechanism this course has built — the VFS, the disk driver, real processes — has been called directly, as ordinary Python objects. This chapter puts one real boundary in front of all of it: a genuine syscall trap, reusing Course 1's own `InterruptTable`, through which every one of these calls must now pass. Then it builds the two syscalls that make a process able to launch other processes at all.

Finding 1: Real read/write, Dispatched Through a Genuine Interrupt Trap

```
class Syscalls:
    def invoke(self, pid, call, *args):
        # the ONE real entry point -- 'user-level' code never calls
        # kernel internals or the VFS directly
        return self.interrupts.dispatch(SYSCALL_INTERRUPT, pid, call, *args)

    def _trap(self, pid, call, *args):
        method = getattr(self, f'_do_{call}')
        return method(pid, *args)
```

VERIFIED DIRECTLY — OPEN/WRITE/READ, CORRECTLY ROUND-TRIPPED THROUGH THE REAL TRAP

Every call — `open('/greeting')`, `write(fd, 0, b'HELLO SYSCALL')`, `read(fd, 0, 13)` — went through `syscalls.invoke()`, which dispatches a real interrupt via Course 1's own `InterruptTable`. User-level code never touched the VFS or kernel directly. Read-back: `b'HELLO SYSCALL'`, exactly correct.

Finding 2: fork() — A Genuinely Independent Child

VERIFIED DIRECTLY — A COPIED, NOT SHARED, FILE DESCRIPTOR TABLE

A parent opens a file and writes to it, then forks. The child correctly inherits the parent's own open file descriptor table — but as a genuine *copy*, not a shared reference. The child then opens a second file of its own; the parent's own table is completely unaffected. Real, verified independence, not superficial resemblance.

Finding 3: exec() — Same Identity, Different Program

VERIFIED DIRECTLY — THE PID AND PCB OBJECT ARE UNCHANGED; THE PROGRAM IS COMPLETELY REPLACED

A process's own PID and real PCB object, checked directly before and after `exec()`: identical (`is`, not just equal). But its own scheduled program: completely replaced, with zero trace of the old one. This is genuinely the *same* process — real `exec()` semantics: same identity, entirely different code.

Finding 4: fork() Then exec() — The Classic Pattern, Chained

VERIFIED DIRECTLY — A GENUINELY NEW TASK LAUNCHED FROM AN EXISTING RUNNING PROCESS

A parent forks a child, then `exec()`s *the child* — not itself — with an entirely different program. The parent's own program stays completely untouched throughout. Two syscalls composed together doing something neither does alone: launching a genuinely new task from an existing running process, the real pattern every Unix shell uses to run a command.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
The syscall trap itself	Course 1 Chapter 6's own `InterruptTable` and `Syscall` class — the exact same "never expose internals directly" boundary, reused directly
open/read/write	Chapter 8's own VFS — the syscall library never touches a backend directly, only ever through the VFS's own uniform interface
fork() creating a real, new process	Course 1 Chapter 4's own `Kernel.create_process()` — reused unchanged for the actual process creation
Every syscall composing correctly with the others	oskernel1's own capstone — the same test this entire course has been building toward: do the independently-verified pieces actually work together

Hands-On Exercises

EXERCISE 1

Have one process open and write to a file, then have a completely different process attempt to `read()` using the exact same fd number, without ever calling `open()` itself. Confirm it fails, and explain why fd numbers aren't shared across processes.

View solution

EXERCISE 2

Fork a process, then fork the resulting child to create a grandchild. Confirm the grandchild correctly inherits both the original file and anything the direct parent opened afterward, and explain why it inherits from its own parent rather than some fixed original ancestor.

View solution

EXERCISE 3

Open a file, then call `exec()` on the same process. Confirm whether the previously-opened file descriptor still works afterward, and explain honestly what this implementation does and doesn't model about real exec() semantics.

View solution

CHAPTER 9 QUICK REFERENCE

- **Syscalls:** one real entry point, `invoke()`, dispatching a genuine interrupt — no direct kernel/VFS access from user-level code
- **Verified Finding 1:** open/read/write correctly round-trip through the real trap and the VFS underneath
- **Verified Finding 2:** fork() produces a genuinely independent child — a copied fd table, not a shared one
- **Verified Finding 3:** exec() preserves a process's own identity exactly while completely replacing its program
- **Verified Finding 4:** fork()+exec() chained together launches a genuinely new task from an existing process — the real Unix pattern
- **Golden rule:** a syscall boundary is only real if user-level code has no path around it — every mechanism this course built stays correctly isolated behind one single trap
- **Next chapter:** Capstone — every subsystem from both courses wired together into one small, multitasking OS running real concurrent programs

CHAPTER 10 OF 10

Capstone — A Small Multitasking OS Running Real Concurrent Programs

Building an Operating System Kernel: Concurrency, I/O & Synchronization

Chapter 10 · Capstone: A Small Multitasking OS Running Real Concurrent Programs

Nine chapters, each independently verified — a mutex, a semaphore, a message queue, deadlock detection, a disk driver, interrupt-driven I/O, a VFS, a syscall library. This capstone wires all of them together with Course 1's own priority scheduler for the first time — and finds the exact same class of bug oskernel1's own capstone found, in the exact same place.

Step 1: Memory, Processes, and a Real Filesystem — Assembled

VERIFIED DIRECTLY — REAL PHYSICAL MEMORY, A REAL KERNEL, A REAL VFS WITH MOUNTED FILES

Real physical memory, a real `Kernel`, and a real `VFS` with two mounted files, assembled from every chapter's own unmodified classes — the foundation the rest of this capstone builds on.

Step 2: The Same Bug, Again — Priority Scheduling Meets Blocking I/O

Chapter 6/7's own `IOKernel` was built and tested against Course 1's plain `Scheduler`. Course 1 Chapter 9's own `AgingScheduler` returns a `(pcb, priority)` tuple from `pick_next()`, not a bare PCB — the identical shape of mismatch oskernel1's own capstone already found once, between two *different* chapters.

VERIFIED DIRECTLY — RAISES: 'LIST' OBJECT HAS NO ATTRIBUTE 'REGISTERS'

Combining Chapter 6/7's own blocking-I/O kernel with the priority-and-aging scheduler from Course 1, unmodified, crashes the instant a process tries to resume after blocking. Both schedulers were independently correct against their own tests. Neither was ever designed against a shared interface — and this is the second time in this project's own history that exact failure mode has appeared.

Step 3: The Fix — A Unified Kernel

```
class UnifiedKernel:
    def _on_timer(self):
        self.ticks_since_switch = 0
        self.scheduler.age_waiting()                # Chapter 9's own fix
        picked = self.scheduler.pick_next()
        if picked is None: return
        next_pcb, _priority = picked                  # unpack correctly
        context_switch_fixed(self.cpu, self.current_pcb, next_pcb)
        if self.current_pcb is not None:
            self.scheduler.add(self.current_pcb, self.base_priorities[self.current_pcb.pid])
        self.current_pcb = next_pcb
```

VERIFIED DIRECTLY — BLOCKING I/O AND PRIORITY SCHEDULING CORRECTLY COMBINE

With tuple-unpacking, `age_waiting()`, and base-priority-reset applied everywhere the scheduler is touched — including `_on_disk_complete()` and `_switch_away_from_blocked()` — a process blocks on real disk I/O, a lower-priority process gets real turns during the wait, and both complete cleanly with no crash.

Step 4: A Full, Real Multi-Process Scenario

Three real processes — one blocking on real disk I/O, two competing on priority — all incrementing a single real, mutex-protected shared counter through real syscalls.

VERIFIED DIRECTLY — 45 OF 45 REAL INCREMENTS LAND CORRECTLY, ZERO LOST

Target: 45 (15 increments × 3 processes). Actual final counter value: **45**, in 53 real steps. Every increment — issued by a different process, through a real syscall trap, under real preemptive priority-and-aging scheduling, with one process genuinely blocking on real disk I/O midway through the run — lands correctly. Chapter 1's own lost-update race: resolved. This capstone's own Step 2 bug: fixed. The mutex genuinely serializes the shared counter across all three processes, correctly, every time.

Chapter Attribution

CAPSTONE COMPONENT	BUILT IN
Physical memory, real processes	Course 1, Chapters 2 & 4
Context switching, interrupts	Course 1, Chapters 5 & 6
Priority scheduling with aging	Course 1, Chapter 9
The atomic mutex	Course 2, Chapter 2

CAPSTONE COMPONENT	BUILT IN
Real, mutex-protected shared state	Course 2, Chapters 1 & 2 (resolving Chapter 1's own race directly)
Real disk driver and blocking I/O	Course 2, Chapters 6 & 7
The VFS	Course 2, Chapter 8
The real syscall trap	Course 2, Chapter 9
The pick_next() interface mismatch, found and fixed	This capstone — the same class of bug oskernel1's own capstone found, recurring at a new seam

What This Course Doesn't Cover

This kernel remains a real, verified Python simulation of genuine kernel mechanisms — not bare-metal code, not assembly, nothing that runs on real hardware. Deliberately out of scope across both courses: multi-core/SMP scheduling, real device drivers for actual hardware, a networking stack, and journaling or crash-consistent file systems (Building a Database Engine's own write-ahead logging covers that territory in depth, for a different kind of storage system).

THE PROJECT, CLOSED

Two courses, twenty chapters: physical and virtual memory, a real process state machine, context switching, a syscall boundary with quota enforcement, cooperative and preemptive scheduling, priority with aging — then real synchronization primitives, deadlock detection and recovery, inter-process communication, a real device driver with interrupt-driven I/O, a virtual file system, and a real syscall library tying it all together. Two genuine cross-chapter integration bugs found and fixed along the way, both times by combining independently-correct pieces for the first time — the same lesson, learned twice, in two different courses.

CAPSTONE QUICK REFERENCE

- **Step 1:** memory + processes + VFS, assembled from every chapter's own unmodified classes
- **Step 2 (bug):** Course 1's priority scheduler and Course 2's blocking-I/O kernel disagree about what `pick_next()` returns — the same class of bug as oskernel1's own capstone
- **Step 3 (fix):** a unified kernel applying tuple-unpacking, `age_waiting()`, and base-priority-reset at every point the scheduler is touched
- **Step 4:** 3 real processes, 45 real mutex-protected syscalls, real disk blocking, real priority scheduling — 45 of 45 correct
- **The one big lesson, twice over:** independently-correct components can still disagree the moment they're actually combined — only real, end-to-end integration finds the gap

- **Course complete:** Building an Operating System Kernel: Concurrency, I/O & Synchronization, 10/10 chapters — closing the full 20-chapter, two-course project