

The Engineer's Loom

A loom weaves hundreds of threads into one cloth by passing each through the same narrow space in turn. An operating system does the same with programs. It gives every process the illusion of a machine of its own while one set of cores, one memory and one disk are shared between them all. This volume takes that illusion apart: processes and system calls, scheduling, the bugs that come from sharing, virtual memory, file systems that survive a crash, and the containers built from all of it. Every mechanism is shown working, in Python that makes real system calls or simulates what can't be shown directly.

CHAPTERS

20

CODE EXAMPLES

20

EXERCISES

60

STAGE BUDGET

150 h

Contents

How to use this volume	4
Part I · The process	6
Chapter 1 · Processes and address spaces	7
Chapter 2 · System calls and the kernel boundary	13
Chapter 3 · Signals, pipes and the shell	19
Part II · Sharing the CPU	26
Chapter 4 · Threads and context switches	27
Chapter 5 · Scheduling	33
Chapter 6 · Fair shares and many cores	41
Part III · Concurrency	47
Chapter 7 · Race conditions and atomicity	48
Chapter 8 · Locks	54
Chapter 9 · Condition variables and semaphores	61
Chapter 10 · Deadlock, livelock and starvation	68
Chapter 11 · Event loops, actors and channels	75
Part IV · Memory	82
Chapter 12 · Virtual memory and paging	83
Chapter 13 · TLBs and page faults	89
Chapter 14 · Swapping and page replacement	95
Chapter 15 · Memory allocation and garbage collection	101
Part V · Files and I/O	108
Chapter 16 · Files and file systems	109
Chapter 17 · Crash consistency, journaling and fsync	115
Chapter 18 · I/O models: blocking, multiplexed and asynchronous	122
Part VI · Between processes	128
Chapter 19 · Interprocess communication	129

Chapter 20 · Virtual machines and containers	135
The deliverable	142
The twenty rules	145
Reading list	147

This is where the machine becomes shared. Stage 04 explained one program running on one processor. Real machines run hundreds of programs at once, and every production incident that involves a hung process, a full disk, a memory leak, a lost write or a deadlock is an argument with the operating system that shares the machine between them. The table shows where each of the stage's twelve topics is taught here, and what to read or build alongside it.

STAGE 05 TOPIC	CHAPTERS	READ OR BUILD ALONGSIDE
Processes: address space, fork and exec, process states, signals	Ch. 1, 3	OSTEP ch. 4–5; xv6 util lab
Threads, context switching, kernel and user mode, system calls	Ch. 2, 4	OSTEP ch. 6, 26–27; xv6 syscall and traps labs
Scheduling: round robin, priority, multi-level feedback, fairness	Ch. 5–6	OSTEP ch. 7–10
Concurrency primitives: mutexes, semaphores, condition variables, atomics	Ch. 7–9	OSTEP ch. 28–31; The Little Book of Semaphores
Race conditions, deadlock, livelock, starvation, the Coffman conditions	Ch. 7, 10	OSTEP ch. 32
Event loops, async/await, actors and channels	Ch. 11	OSTEP ch. 33
Virtual memory: paging, page tables, the TLB, page faults, swapping	Ch. 12–14	OSTEP ch. 13–23; xv6 pgtbl and cow labs
Memory allocation, fragmentation, garbage collection	Ch. 15	OSTEP ch. 14, 17; xv6 lock lab
File systems: inodes, directories, journalling, durability and fsync	Ch. 16–17	OSTEP ch. 39–42; xv6 fs lab
I/O models: blocking, non-blocking, multiplexed, asynchronous	Ch. 18	OSTEP ch. 33; the epoll and kqueue manual pages
Interprocess communication: pipes, sockets, shared memory, queues	Ch. 19	The Linux Programming Interface, ch. 43–61
Virtualisation and containers: namespaces,	Ch. 20	OSTEP appendix B; the

Run it, then build it. Most examples make real system calls through Python's `os` module: they fork processes, connect them with pipes, send signals, link files and share memory. They run on Linux and macOS; on Windows, run them under WSL. Where the real thing can't be shown the same way twice, because it depends on timing, the example simulates it instead. The concurrency chapters go further and check every possible interleaving of a small program, which is how the bugs that appear once a month in production can be made to appear every time. The two examples that time real hardware say so, and show one recorded run. Every other output shown is what the program printed when this volume was built.

The deliverable is a shell, a thread pool and a kernel. The stage asks for a shell with pipes, redirection and job control; a thread pool with a work queue; and the xv6 labs from MIT's operating systems course, including at least one system call of your own. The appendix maps every lab to the chapters that prepare for it.

THE ONE HABIT

Everything an operating system manages is shared, and every kind of sharing needs a rule: who runs next, who may see which memory, and what happens if someone stops halfway through.

A thread can be interrupted between any two instructions, a process can be killed between any two system calls, and the power can fail between any two disk writes. So for every piece of shared state, **ask what happens if this is interrupted at the worst possible moment.** Most of this volume is the answer to that question for one resource after another.

PART I

The process

Three chapters on the unit an operating system manages: the process, a running program with an address space and an identity of its own; the system call, the one door through which a process asks the kernel for anything; and the signals and pipes with which a shell starts, connects and controls processes. By the end of this part you can build the core of a Unix shell.

Processes and address spaces

3 exercises

A program is a file on disk. A process is that program running, together with everything the operating system keeps for it: its own memory, its open files, its saved registers and an identity. The first thing an operating system does is give every process the illusion of a machine to itself. This chapter shows what a process is made of, creates processes with `fork` and `exec` the way every Unix shell does, and follows one through its life until its parent collects its exit status.

What a process is

A process has three kinds of state. Its **address space** is the memory it can see: the program's code, its global data, a heap that grows as it allocates, a stack for each thread, and any files or libraries mapped in (chapter 17 of *The Engineer's Mill*). Its **CPU state** is the program counter, the stack pointer and the other registers; while the process isn't running, these are saved in the kernel. Its **kernel state** is the rest: a process ID, its parent's ID, the user and group it runs as, a table of open files, a current directory, how it handles each signal, and its resource limits. The kernel keeps all of this in one record per process, the process control block. On Linux it's a structure called `task_struct`, several kilobytes long.

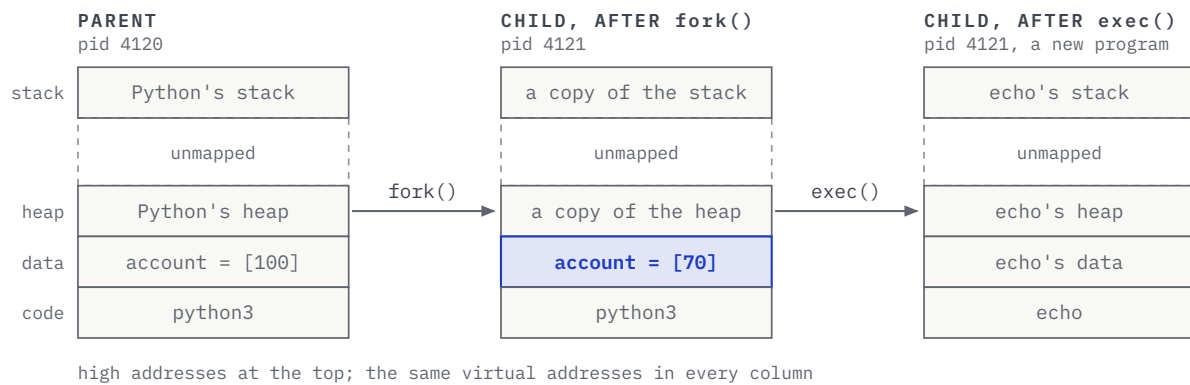
On Linux you can read much of this directly. `/proc/<pid>/maps` lists a process's address space region by region, `/proc/<pid>/fd` lists its open files, and `/proc/<pid>/status` shows its state, memory use and owner. `ps` and `top` are mostly ways of reading `/proc`. On macOS, `ps`, `lsopf` and `vmmap` show the same things.

Each process sees its own memory

Addresses in a program are **virtual**. When two processes both use address 0x601000, each reaches its own memory, because the kernel gives each process its own map from virtual addresses to physical memory. Part IV explains how that map works and what it costs. For now, the consequence is the one that matters: one process can't read or overwrite another's memory by accident, and a crash in one leaves the rest running.

fork, exec and wait

Unix creates processes in two steps. `fork()` makes a copy of the calling process. The copy, the **child**, has the same code, a copy of the same memory, the same open files and the same position in the program. It differs in its process ID and in one value: `fork()` returns 0 in the child and the child's ID in the parent, which is how each knows which it is. `exec()` then replaces the program running in a process with a new one, keeping the process ID and the open files. The parent calls `wait()` to learn when the child has finished, and how.



What `fork` and `exec` do to memory. After `fork` the child has its own copy of every region, at the same virtual addresses, so its change to `account` is invisible to the parent. After `exec` the same process runs a different program.

Splitting creation into two calls looks odd, but it's what makes the shell simple. Between `fork` and `exec`, the child is still running the shell's code, so it can rearrange its own file descriptors, change directory, set limits or drop privileges, and the program it then runs inherits the result. That's how `ls > out.txt` works (chapter 3). Windows takes the other approach: `CreateProcess` creates a process and starts a program in one call, with parameters for everything a Unix child would set up itself. POSIX also offers `posix_spawn`, which does the same.

The example forks a child that changes a value, shows that the parent's copy is untouched although the address is the same, and collects the child's exit status. It then runs `echo` through `fork` and `exec`, and reads the exit statuses of four commands, one of which doesn't exist.

EXAMPLE

process.py · fork, separate memory at the same addresses, exec, and exit statuses collected with `waitpid`

```
import os, sys

# 1. fork() returns twice: 0 in the child, the child's process ID in the parent
account = [100]                                # one object, at one virtual address
address = id(account)
reader, writer = os.pipe()                      # a pipe, so the child can report back
sys.stdout.flush()                             # empty Python's buffer, or the child inherits
# a copy of it
pid = os.fork()
if pid == 0:                                   # ---- the child ----
    account[0] -= 30                           # changes the child's copy only
    os.write(writer, f"{account[0]} {id(account)}".encode())
    os._exit(3)                                # exit status 3, without running the parent's
    # cleanup
_, status = os.waitpid(pid, 0)                 # ---- the parent waits for it ----
child_value, child_address = map(int, os.read(reader, 100).split())
print(f"fork returned the child's pid to the parent: {pid > 0}")
print(f"child's account: {child_value}; parent's account: {account[0]}")
print(f"same virtual address in both processes: {child_address == address}")
print(f"child's exit status, collected by waitpid: {os.waitstatus_to_exitcode(status)}")
```

```

# 2. exec() replaces the running program; the process, and its pid, stay the same
sys.stdout.flush()
pid = os.fork()
if pid == 0:
    os.execvp("echo", ["echo", "now running echo, inside the process that fork made"])
    os._exit(127)                                # only reached if exec failed
os.waitpid(pid, 0)

# 3. Exit statuses: 0 means success; anything else is an error code for the parent to read
for cmd in (["true"], ["false"], ["sh", "-c", "exit 7"], ["no-such-program"]):
    sys.stdout.flush()
    pid = os.fork()
    if pid == 0:
        try:
            os.execvp(cmd[0], cmd)
        except FileNotFoundError:
            os._exit(127)                        # the shell's convention for "command not
                                                found"
    _, status = os.waitpid(pid, 0)
    print(f'{ " ".join(cmd):16} exited with status {os.waitstatus_to_exitcode(status)}')

```

OUTPUT

```

fork returned the child's pid to the parent: True
child's account: 70; parent's account: 100
same virtual address in both processes: True
child's exit status, collected by waitpid: 3
now running echo, inside the process that fork made
true           exited with status 0
false          exited with status 1
sh -c exit 7   exited with status 7
no-such-program exited with status 127

```

Two details in the code matter in any language. The child ends with `os._exit`, which leaves at once, instead of returning into the parent's code or running the parent's clean-up handlers twice. And the parent empties Python's output buffer before each `fork`, for a reason exercise 1.2 explains.

`fork` has critics. Copying a process with a large address space is slow even with the tricks of chapter 13, and forking a process that runs several threads copies only the thread that called it, which can leave locks held forever by threads that don't exist in the child. Baumann and colleagues argued in 2019 that `fork` no longer belongs in operating systems. Python 3.14 stopped using it by default for `multiprocessing` on Linux for this reason. Shells and servers still use it everywhere, so it's worth understanding precisely.

The life of a process

A process moves between a few states. It's **running** while a core is executing it, **ready** while it could run but is waiting for a core, and **blocked** while it waits for something else: a disk read, a network packet, a timer, a lock. Chapters 4 to 6 are about the move from ready to running. A process that has exited but whose parent hasn't yet called `wait` is a **zombie**. Its memory and files are gone, but the kernel keeps its process ID and exit status until the parent collects them. In `ps` output these states appear as R, S (sleeping, blocked), D (blocked in a way signals can't interrupt, usually disk I/O), T (stopped) and Z.

Two rules follow. A parent should wait for every child, or tell the kernel it doesn't care, by setting `SIGCHLD` to be ignored, so that children are reaped automatically. Otherwise zombies accumulate until no process IDs are left. And if a parent exits first, its children are **orphans**: the kernel hands them to process 1, `init` or `systemd`, which waits for them. Exit statuses follow a convention shells rely on: 0 means success, small numbers are errors chosen by the program, 126 means a file couldn't be executed, 127 that the command wasn't found, and 128 plus n that the process was killed by signal n .

COMMON TRAPS

Never calling wait, so finished children pile up as zombies.

Forking with unflushed output, which then appears twice.

Forking a multithreaded program and calling anything but `exec` in the child.

Treating a non-zero exit status as the only failure. A process killed by a signal has no exit status at all.

Assuming process IDs are unique forever. They're reused, so a stale PID file can name an unrelated process.

EXERCISES

EXERCISE 1.1 Counting processes

A program runs `for i in range(3): os.fork()` and then prints one line. How many lines are printed? How many processes call `fork` in the third iteration?

SOLUTION

Every process that reaches a `fork` becomes two, so the number of processes doubles in each iteration: 2 after the first, 4 after the second and 8 after the third. Eight processes print, so there are 8 lines. In the third iteration all 4 existing processes call `fork`, the original included. In general n calls in a loop like this create 2^n processes, which is why a loop that forks by mistake, a **fork bomb**, can exhaust a machine in seconds. Per-user process limits, `ulimit -u`, exist largely for that reason.

EXERCISE 1.2 The line that appears twice

A program runs `print("starting")` and then `os.fork()`, and both processes then exit normally. Run in a terminal, it prints "starting" once. With its output redirected to a file, the file contains "starting" twice. Why?

SOLUTION

`print` doesn't make a system call straight away. It writes into a buffer inside the process, which is emptied when it fills, when the program exits, or, for a terminal, at the end of each line. On a terminal the line was written out before the `fork`, so only one copy exists. When output goes to a file, the buffer is flushed only when full or at exit, so at the moment of the `fork` "starting" is still in the buffer, part of the process's memory. The child gets a copy, and both processes flush their copies at exit. The fix is to flush before forking, as the example does, or to have the child end with `os._exit`, which exits without flushing. C's `printf` and `exit` behave the same way.

EXERCISE 1.3 A server that never waits

A server forks a child to handle each request and never calls `wait`. Each child exits within a second. What builds up, what eventually goes wrong, and what are two ways to fix it?

SOLUTION

Every finished child stays a zombie, holding a process ID and a small kernel record, until the parent waits for it, and this parent never does. Zombies use almost no memory, but process IDs are limited: by the system-wide maximum, 32,768 by default on older Linux systems and over 4 million on newer ones, and by the per-user limit, which is usually much lower. When the limit is reached, `fork` fails with `EAGAIN` for the server and for every other process the same user runs. One fix is to handle `SIGCHLD`, which the kernel sends when a child exits, and call `waitpid(-1, WNOHANG)` in a loop until there are none left to collect. The other is to set `SIGCHLD` to be ignored, which on Linux and macOS tells the kernel to reap children automatically. If the server itself exits, its zombies are adopted by process 1 and reaped.

CHECKLIST

- ☐ Can you list what a process consists of: its address space, CPU state and kernel state?

- ☐ Can you explain what `fork` returns, and why creation is split into `fork` and `exec`?
- ☐ Can you name the states a process passes through, including zombie?
- ☐ Can you explain how a parent learns how its child ended, and the exit status conventions?
- ☐ Can you find a running process's memory map and open files on your system?

GO DEEPER

BOOK Operating Systems: Three Easy Pieces, chapters 4 and 5: processes and the process API Arpaci-Dusseau & Arpaci-Dusseau

PAPER A fork() in the road Baumann, Appavoo, Krieger & Roscoe, HotOS 2019

BOOK The Linux Programming Interface, chapters 24 to 28: process creation and termination Michael Kerrisk

System calls and the kernel boundary

3 exercises

A process can compute on its own at full speed, but anything that touches the world outside its memory goes through the kernel: reading a file, sending a packet, starting a process, allocating more memory. The processor enforces that boundary with privilege modes, and system calls are the one controlled way across it. This chapter follows a system call from a line of Python to a device driver and back, measures what the crossing costs, and draws the rule that follows: make fewer, bigger system calls.

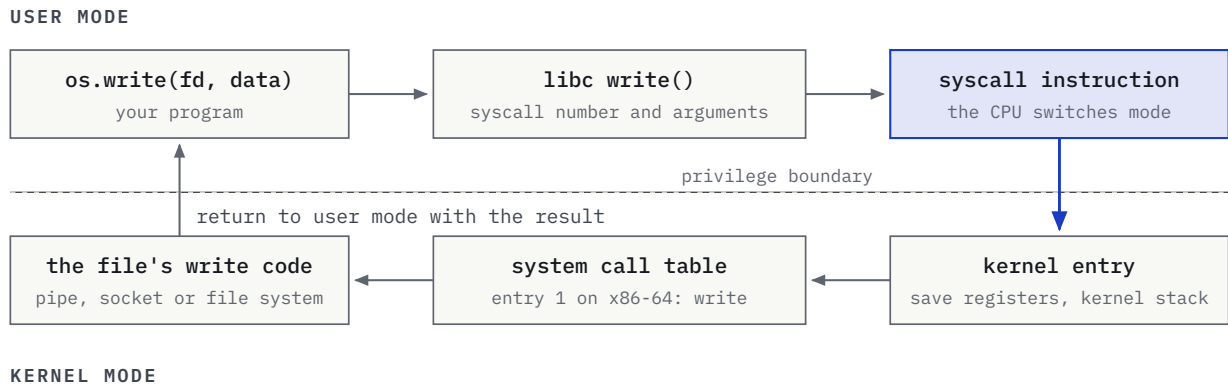
Two modes

Every modern processor runs in at least two modes. In **user mode**, the mode programs run in, some instructions are forbidden: those that talk to devices, change the page tables, switch off interrupts or halt the machine. In **kernel mode** everything is allowed. On x86 the modes are called rings 3 and 0; on ARM they're exception levels EL0 and EL1. A user program that tries a privileged instruction doesn't get to run it. The processor traps into the kernel, which normally kills the program.

This is what lets the operating system run programs directly on the hardware, at full speed, and still stay in charge. OSTEP calls it **limited direct execution**. The program's instructions run on the real processor with no interpreter in the way, but there are only three ways back into the kernel: the program asks, with a system call; the program does something it can't, such as dividing by zero or touching unmapped memory, which raises an exception; or a device or the timer raises an interrupt. The timer interrupt is the one that guarantees the kernel can always take the processor back, even from a program stuck in a loop (chapter 5).

Crossing the boundary

From Python, `os.write(fd, data)` calls the C library's `write` function, which is a thin wrapper. On Linux on x86-64, it puts the system call's number, 1 for `write`, in the `rax` register and the three arguments in `rdi`, `rsi` and `rdx`, then executes the `syscall` instruction. On 64-bit ARM the instruction is `svc #0` and the number goes in `x8`. The processor switches to kernel mode and jumps to an entry point the kernel registered at boot. Nothing else about the jump is up to the program.



One `write`, from Python to the kernel and back. Only the highlighted step crosses the boundary, and the kernel decides where it lands.

The kernel's entry code saves the program's registers, switches to a stack that belongs to the kernel, and looks the number up in the **system call table**, about 450 entries on Linux. The handler checks its arguments before trusting them. A pointer from user space must point into the caller's own memory, and the kernel copies data in and out with special functions instead of following the pointer directly, because a malicious program could pass the address of kernel memory. When the work is done, the result goes back in a register and the processor returns to user mode. On failure the kernel returns a negative error number, and the C library turns it into a return value of `-1` with the number in `errno`. Python turns it into an exception: `ENOENT` becomes `FileNotFoundError`, `EACCES` becomes `PermissionError`, `EAGAIN` becomes `BlockingIOError`.

What a crossing costs

A system call costs far more than a function call. The mode switch, saving and restoring registers, the checks, and on some processors the mitigations added after the Meltdown and Spectre attacks of 2018 (chapter 13 of *The Engineer's Mill*) all take time, and the kernel's code and data push the program's out of the caches. The example times the smallest system call there is, `getppid`, then one that does real work, and then writes a million bytes three ways. The timings are one recorded run on an Apple M1 Pro.

EXAMPLE

`syscalls.py` · the cost of a system call, and writing a million bytes with one call per byte, through a buffer, and all at once

```

import io, os, time

def per_call(fn, n):
    best = float("inf")
    for _ in range(5):
        t0 = time.perf_counter()
        fn(n)
        best = min(best, time.perf_counter() - t0)
    return best / n * 1e9

def nothing(): return 0
  
```

best of five runs, to skip noise

nanoseconds per call

```

def calls(n):
    for _ in range(n): nothing()           # an ordinary function call: no kernel
def getppid(n):
    for _ in range(n): os.getppid()        # the smallest real system call
def stat(n):
    for _ in range(n): os.stat("/")        # a system call that does some work

N = 200_000
base = per_call(calls, N)
print(f"Python function call      {base:6.0f} ns")
print(f"os.getppid() system call  {per_call(getppid, N):6.0f} ns")
print(f"os.stat('/') system call  {per_call(stat, N):6.0f} ns")

# Writing a million bytes to /dev/null: one write per byte, a buffered file, or one big write
fd = os.open(os.devnull, os.O_WRONLY)
SIZE = 1_000_000
def unbuffered(_):
    for _ in range(SIZE): os.write(fd, b"x")    # a million system calls
def buffered(_):
    with open(os.devnull, "wb") as f:           # Python collects bytes in a buffer...
        for _ in range(SIZE): f.write(b"x")     # ...and calls write() when it fills
def one_write(_):
    os.write(fd, b"x" * SIZE)                   # one system call
print(f"\nwriting {SIZE:,} bytes (Python's buffer is {io.DEFAULT_BUFFER_SIZE:,} bytes):")
for name, fn, syscalls in (("one write() per byte", unbuffered, SIZE),
                           ("buffered file object", buffered, -(-SIZE //
                           io.DEFAULT_BUFFER_SIZE)),
                           ("one write() of everything", one_write, 1)):
    ms = per_call(fn, 1) / 1e6
    print(f"    {name:26} {ms:9.3f} ms    {syscalls:>9,} system call{'s' if syscalls > 1 else ''}")

```

OUTPUT

```

Python function call      29 ns
os.getppid() system call  161 ns
os.stat('/') system call  1301 ns

writing 1,000,000 bytes (Python's buffer is 131,072 bytes):
  one write() per byte      573.625 ms    1,000,000 system calls
  buffered file object      39.286 ms        8 system calls
  one write() of everything   0.037 ms        1 system call

```

Even from Python, where an ordinary function call costs about 29 ns, the cheapest system call costs over five times as much, and from C the ratio is closer to a hundred. Writing a byte at a time made a million system calls and took over half a second. A buffered file object made 8, because Python collects bytes in a 128 KiB buffer and calls `write` only when it fills; its 39 ms are almost all the cost of a million calls to `f.write` in Python, not of the kernel. A single `write` of the whole megabyte took 0.04 ms. The per-call overhead, not the bytes, was the cost.

Fewer, bigger crossings

That result is behind a whole family of techniques. **Buffering**, in C's `stdio` and in every language's file objects, turns many small writes into a few large ones. `readv` and `writew` move several buffers in one call. `sendfile` and `copy_file_range` move data from one file descriptor to another without bringing it into the process at all. `mmap` maps a file into

memory so that reading it needs no system calls, only page faults (chapter 13). `epoll` returns many ready connections from one call (chapter 18), and Linux's `io_uring` lets a program queue hundreds of operations and collect their results through memory shared with the kernel.

Some calls don't need the kernel at all. Reading the clock is so common that Linux maps a small read-only page of kernel data and code, the `vDSO`, into every process. `clock_gettime` runs there, in user mode, and costs about as much as a function call. macOS does the same with its `commpage`.

Watching system calls

On Linux, `strace` prints every system call a program makes, with its arguments and result. `strace -c` counts them and totals their time, and `strace -e trace=openat` shows only the files a program opens, which is often the fastest way to find out why it can't find its configuration. `perf trace` does the same with less overhead. On macOS, `dtruss` and Instruments do the same job, though system integrity protection limits what they can trace. A program that's slow without using much CPU is often blocked in a system call, and these tools show which one.

COMMON TRAPS

Unbuffered output in a loop: one system call per byte or per line.

Ignoring short reads and writes. A read or write on a pipe or socket can move fewer bytes than asked.

Not retrying after `EINTR` in C, when a signal interrupts a slow call. Python retries for you.

Assuming a call that looks cheap is cheap. Each `os.stat` here cost over a microsecond.

Trusting pointers or lengths that come from user space, if you ever write kernel or driver code.

EXERCISES

EXERCISE 2.1 The short write

`os.write(sock_fd, data)` is called with 10,000 bytes on a socket and returns 3,000. What happened, and what must the caller do? Write a function that writes all of `data`.

SOLUTION

The kernel accepted only what fit in the socket's send buffer and reported how much that was. That's not an error: `write` returns the number of bytes it actually took, and on pipes, sockets and terminals that can be fewer than asked. The caller must send the rest:

```
def write_all(fd, data):
    view = memoryview(data)
    while view:
        n = os.write(fd, view)      # may take only part of it
        view = view[n:]
```

Python's `socket.sendall` and buffered file objects do this loop for you; `os.write` and C's `write` don't. Reads behave the same way: `read` returns what's available, which may be less than requested even before the end of the data.

EXERCISE 2.2 A logger's budget

A service logs 50,000 lines a second, 80 bytes each, with one unbuffered `write` per line costing 1.5 μ s. What share of a core does that take? What if lines are collected in a 64 KiB buffer? What does buffering risk, and how do logging libraries limit the risk?

SOLUTION

Unbuffered: $50,000 \times 1.5 \mu\text{s}$ is 75 ms of every second, 7.5% of a core spent on system calls alone. Buffered: the service produces 4 MB a second, so a 64 KiB buffer fills about 61 times a second, and 61 larger writes cost well under a millisecond. The risk is that lines still in the buffer are lost if the process crashes, which is exactly when you most want them. Logging libraries flush on a timer, every few hundred milliseconds, and flush immediately for errors, and many write logs from a separate thread or hand them to a local agent, so the service never waits for the disk.

EXERCISE 2.3 Why the clock needs no kernel

Explain why reading the time can run entirely in user mode through the vDSO, while `write` can't.

SOLUTION

Reading the time needs no privileged instruction and changes nothing. The kernel publishes the clock's current parameters, such as a base time and a scale for the processor's cycle counter, in a page mapped read-only into every process, and updates them as needed. The vDSO

code reads those values and the counter, which user mode is allowed to read, and computes the time. Nothing it does could harm another process. `write` is different: it must check that the process may write to that file, change shared state such as file contents, pipe buffers or a device, and possibly wake other processes. All of that needs kernel mode and the kernel's data structures, so it has to cross the boundary.

CHECKLIST

- ☐ Can you explain user and kernel mode, and the three ways back into the kernel?
- ☐ Can you describe the steps of a system call, from the library wrapper to the return?
- ☐ Can you estimate the cost of a system call relative to a function call?
- ☐ Can you name four ways to make fewer system calls?
- ☐ Can you use `strace` or an equivalent to see what a program is asking the kernel to do?

GO DEEPER

BOOK Operating Systems: Three Easy Pieces, chapter 6: limited direct execution
Arpaci-Dusseau & Arpaci-Dusseau

REFERENCE `syscalls(2)`: the Linux system call list [The Linux man-pages project](#)

PAPER FlexSC: exception-less system calls Soares & Stumm, OSDI 2010

Signals, pipes and the shell

3 exercises

A Unix shell is a short program built from about a dozen system calls: `fork`, `exec` and `wait` from chapter 1, plus `pipe`, `dup2`, `open`, `setpgid` and `kill`. Knowing how it uses them explains pipelines, redirection, exit statuses, Ctrl-C and Ctrl-Z, and why `cmd > file 2>&1` and `cmd 2>&1 > file` do different things. This chapter builds the core of a shell in about forty lines and then controls a running pipeline with signals.

File descriptors 0, 1 and 2

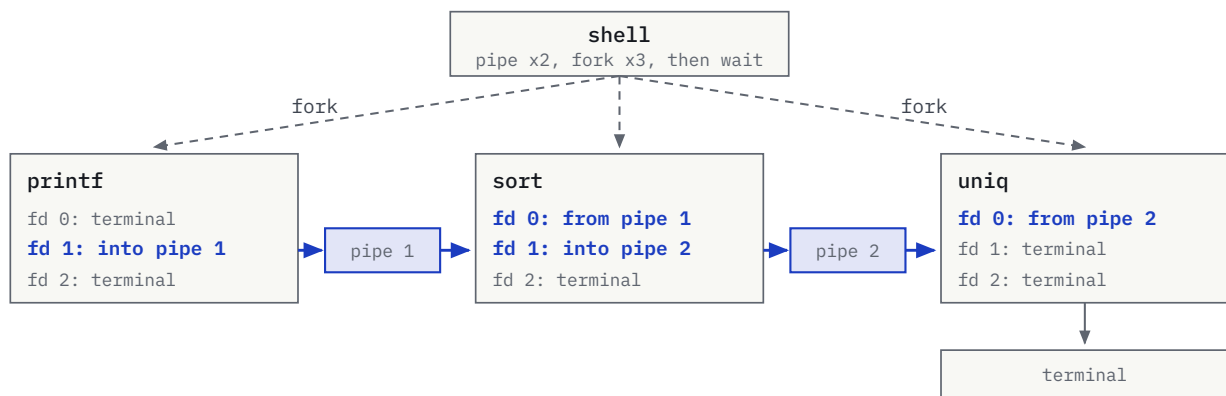
Every process starts with three open file descriptors: 0 for standard input, 1 for standard output and 2 for standard error. A program writes to descriptor 1 without knowing whether it leads to a terminal, a file or another program. Whoever started the program decided that, and that's the trick behind redirection. `open` always returns the lowest free descriptor, and `dup2(old, new)` makes descriptor `new` refer to the same open file as `old`, closing whatever `new` referred to before. To run `ls > out.txt`, the shell forks, and the child opens `out.txt`, calls `dup2` to put it on descriptor 1, closes the original descriptor, and execs `ls`. `ls` writes to descriptor 1 as always and never learns the difference.

Redirections are applied left to right, each one a `dup2`. `2>&1` means "make descriptor 2 a copy of whatever descriptor 1 is now". So `cmd > file 2>&1` first points 1 at the file and then copies that to 2, sending both to the file. `cmd 2>&1 > file` copies 1, still the terminal, to 2 first, and only then points 1 at the file.

Pipes

`pipe()` creates a buffer in the kernel and returns two descriptors: bytes written to one can be read from the other, in order. The buffer is small, 64 KiB by default on Linux, so a fast writer blocks until the reader catches up, and a reader blocks until there's something to read. That keeps a pipeline's memory use bounded however much data flows through it. A read returns end-of-file only when every write end of the pipe is closed, and a write to a pipe with no readers left sends the writer `SIGPIPE`, which kills it by default. That's how `yes | head -1` ends: `head` exits after one line and `yes` dies on its next write.

For a pipeline the shell creates one pipe between each pair of stages and forks one child per stage. Each child moves the right pipe ends onto its descriptors 0 and 1 with `dup2`, closes every other pipe descriptor it inherited, and execs its program. The parent closes its copies too. Each of those closes matters. A reader sees end-of-file only when *every* copy of the write end is closed, including copies left open by mistake in the parent or a sibling.



The descriptors of `printf ... | sort | uniq` after the shell has wired it up. None of the three programs knows it's in a pipeline: each reads descriptor 0 and writes descriptor 1.

A shell in forty lines

The example's `start` function does what a real shell does for a pipeline with redirections, and `wait_all` waits for every stage and reports the status of the last, which is the status a shell reports for a pipeline. It runs the pipeline in the figure, writes a file through a redirection and reads it back through another, and shows that `true | false` fails while `false | true` succeeds. The second half starts `sleep 30 | sleep 30`, suspends it, resumes it and interrupts it, the way Ctrl-Z, `fg` and Ctrl-C would, and then catches a signal sent to itself.

EXAMPLE

shell.py · a pipeline launcher with redirection, exit statuses, and job control with signals

```

import os, sys, shlex, signal, tempfile

def start(line):
    """Start a pipeline like 'a | b < in > out' in its own process group; return the child
    pids."""
    stages = [shlex.split(part) for part in line.split("|")]
    pids, pgid, prev_read = [], 0, None
    exec_r, exec_w = os.pipe() # close-on-exec: every child's copy closes when it
    execs
    for i, argv in enumerate(stages):
        infile = outfile = None
        if "<" in argv: k = argv.index("<"); infile = argv[k + 1]; del argv[k:k + 2]
        if ">" in argv: k = argv.index(">"); outfile = argv[k + 1]; del argv[k:k + 2]
        read_end, write_end = os.pipe() if i < len(stages) - 1 else (None, None)
        sys.stdout.flush()
        pid = os.fork()
        if pid == 0: # the child: wire up fds, then
            exec
            os.setpgid(0, pgid) # join the pipeline's process
            signal.signal(signal.SIGINT, signal.SIG_DFL) # undo Python's Ctrl-C handler,
            as shells do
            if prev_read is not None: os.dup2(prev_read, 0) # stdin from the previous stage
            if write_end is not None: os.dup2(write_end, 1) # stdout into the next stage
  
```

```

        if infile: os.dup2(os.open(infile, os.O_RDONLY), 0)
        if outfile: os.dup2(os.open(outfile, os.O_WRONLY | os.O_CREAT | os.O_TRUNC, 0o644),
                               1)
        for fd in (prev_read, read_end, write_end, exec_r): # close the originals, or
            readers
            if fd is not None: os.close(fd) # never see end-of-file
        try: os.execvp(argv[0], argv)
        except OSError: os._exit(127) # never fall back into the
            shell's code
    pgid = pgid or pid # the first child leads the
        group
    try: os.setpgid(pid, pgid) # set it here too, to avoid a
        race
    except PermissionError: pass # (the child got there first
        and exec'd)
    if prev_read is not None: os.close(prev_read)
    if write_end is not None: os.close(write_end)
    prev_read = read_end
    pids.append(pid)
    os.close(exec_w)
    os.read(exec_r, 1) # returns only when every child has exec'd (or
        died)
    os.close(exec_r)
    return pids

def wait_all(pids):
    """Wait for every stage; like a shell, report the status of the last one."""
    statuses = [os.waitpid(pid, 0)[1] for pid in pids]
    last = statuses[-1]
    if os.WIFSIGNALED(last): return f"killed by {signal.Signals(os.WTERMSIG(last)).name}"
    return f"exit status {os.waitstatus_to_exitcode(last)}"

def run(line):
    print(f"$ {line}"); return wait_all(start(line))

work = tempfile.mkdtemp()
os.chdir(work)
run("printf 'pear\\napple\\nfig\\napple\\n' | sort | uniq")
run("printf 'beta\\nalpha\\n' | sort > sorted.txt")
run("tr a-z A-Z < sorted.txt")
for line in ("true | false", "false | true"):
    print(f"$ {line:13} -> {wait_all(start(line))}")

# Signals: a whole pipeline in one process group can be stopped, resumed or killed together
pids = start("sleep 30 | sleep 30")
os.killpg(pids[0], signal.SIGSTOP) # what Ctrl-Z sends: suspend
    the job
os.killpg(pids[0], signal.SIGCONT) # what 'fg' sends: resume it
os.killpg(pids[0], signal.SIGINT) # what Ctrl-C sends: interrupt
    it
print(f"\nsleep 30 | sleep 30, after SIGSTOP, SIGCONT and SIGINT to its group:
    {wait_all(pids)}")

# A process can catch most signals and decide what to do; SIGKILL and SIGSTOP can't be caught
signal.signal(signal.SIGTERM, lambda signum, frame: print("caught SIGTERM: cleaning up before
    exit"))
os.kill(os.getpid(), signal.SIGTERM)
try:
    signal.signal(signal.SIGKILL, lambda signum, frame: None)
except OSError as e:
    print(f"installing a SIGKILL handler fails: {e.strerror}")

```

OUTPUT

```

$ printf 'pear\napple\nfig\napple\n' | sort | uniq
apple

```

```
fig
pear
$ printf 'beta\nalpha\n' | sort > sorted.txt
$ tr a-z A-Z < sorted.txt
ALPHA
BETA
$ true | false -> exit status 1
$ false | true -> exit status 0
```

```
sleep 30 | sleep 30, after SIGSTOP, SIGCONT and SIGINT to its group: killed by SIGINT
caught SIGTERM: cleaning up before exit
installing a SIGKILL handler fails: Invalid argument
```

One detail took longer to get right than the rest. The parent must not signal the pipeline until both children have called `exec`: a child interrupted while still running Python would handle the signal as Python, not as `sleep`. The example waits on an extra pipe that Python creates with the `close-on-exec` flag. Each child's copy closes automatically when that child execs, so the parent's read returns only when all of them have. Shells solve the same race in their own ways, and it's a good example of how much of systems programming is about the order in which things happen.

Signals

A **signal** is a small asynchronous notification sent to a process by the kernel or by another process. The common ones: `SIGINT`, sent by Ctrl-C; `SIGTERM`, the polite request to exit that `kill` sends by default; `SIGKILL`, which ends the process at once; `SIGTSTP`, sent by Ctrl-Z, and `SIGSTOP`, which suspend it; `SIGCONT`, which resumes it; `SIGCHLD`, sent to a parent when a child exits or stops; `SIGPIPE`; `SIGSEGV`, for an invalid memory access; and `SIGHUP`, sent when the terminal goes away and, by convention, used to ask a daemon to reload its configuration.

For each signal a process chooses the default action, which for most signals is to terminate, to ignore the signal, or to run a handler. `SIGKILL` and `SIGSTOP` can't be caught or ignored, which is what makes them reliable; the example's attempt to install a `SIGKILL` handler fails. A handler can run between any two instructions of the program, so in C it may only call functions that are safe at any moment: not `malloc` and not `printf`, which may be halfway through updating their own state. The usual pattern is for the handler to set a flag, or write a byte to a pipe that the main loop watches, and let the main program do the work. Python does this for you: its C-level handler only records the signal, and your Python handler runs in the main thread at the next safe point.

Job control

A shell puts each pipeline in its own **process group**, with `setpgid`, so the whole job can be signalled at once with `killpg`. The terminal driver tracks one foreground process group. When you press Ctrl-C, the terminal sends `SIGINT` to every process in that group, and the shell itself, which is in another group, is untouched. Ctrl-Z sends `SIGTSTP`; the shell notices that the job has stopped (through `waitpid` with the `WUNTRACED` flag), takes the terminal back, and prints a prompt. `fg` hands the terminal to the job again and sends `SIGCONT`; `bg`

sends `SIGCONT` without handing the terminal over. The same mechanisms run production systems. When a container or a systemd service is stopped, its main process gets `SIGTERM`, a grace period, 10 seconds by default in Docker and 30 in Kubernetes, and then `SIGKILL`. A service that handles `SIGTERM` by finishing its current requests and exiting shuts down cleanly. One that doesn't is killed in the middle of them.

COMMON TRAPS

Leaving a pipe's write end open in the parent or a sibling, so the reader never sees end-of-file.

Real work inside a C signal handler, which can deadlock in `malloc` or corrupt state.

A server killed by `SIGPIPE` when a client disconnects. Ignore the signal and handle `EPIPE` instead.

Writing `2>&1 > file` and expecting errors in the file.

Trusting a pipeline's exit status: it's the last stage's. Use `set -o pipefail` in scripts.

A container's main process that ignores `SIGTERM`, so every stop waits out the grace period.

EXERCISES

EXERCISE 3.1 Redirection, as system calls

Write the sequence of `open` and `dup2` calls a shell's child makes for `cmd > out.txt 2>&1` and for `cmd 2>&1 > out.txt`, and say where standard error ends up in each.

SOLUTION

For `cmd > out.txt 2>&1`: `fd = open("out.txt", O_WRONLY|O_CREAT|O_TRUNC)`, `dup2(fd, 1)`, `close(fd)`, then `dup2(1, 2)`. Descriptor 1 is the file when 2 is copied from it, so both go to the file. For `cmd 2>&1 > out.txt`: `dup2(1, 2)` first, while 1 is still the terminal, so 2 becomes the terminal; then `open`, `dup2(fd, 1)` and `close(fd)` point 1 at the file. Output goes to the file and errors to the terminal. Each redirection copies a descriptor's current value. It doesn't create a lasting link between descriptors.

EXERCISE 3.2 The pipeline that never ends

A student's shell runs `printf 'hi\n' | cat . cat` prints "hi" and then the pipeline hangs forever. `printf` has exited. What's the most likely bug?

SOLUTION

`cat` reads until end-of-file, and a pipe reports end-of-file only when every descriptor for its write end is closed. `printf`'s copy closed when it exited, so another copy must still be open: most likely the shell process never closed its own write end after forking, or the `cat` child inherited the write end and didn't close it before exec. `cat` holding a write end to its own input pipe is the subtlest case, because it can then never see end-of-file however long it waits. The fix is to close every pipe descriptor a process doesn't use, in the parent right after forking and in each child before exec. Creating pipes with the close-on-exec flag, as Python does by default, removes most chances to get this wrong.

EXERCISE 3.3 The ten-second deploy

Every time a containerised service is stopped, it takes exactly 10 seconds, and its logs show it was killed rather than exiting. The Dockerfile ends with `CMD python server.py`. Explain the delay and give two fixes.

SOLUTION

Written that way, the "shell form", Docker runs `/bin/sh -c "python server.py"`, so the shell is process 1 in the container and Python is its child. `docker stop` sends `SIGTERM` to process 1 only. Process 1 is also special: the kernel doesn't apply the default "terminate" action to it, so a signal it has no handler for is simply ignored. Either way the server never hears about the stop, and after the 10-second grace period Docker sends `SIGKILL`, which can't be ignored. Fixes: use the exec form, `CMD ["python", "server.py"]`, so Python is process 1, and give it a `SIGTERM` handler that stops accepting work, finishes current requests and exits; or run a minimal init such as `tini`, with `docker run --init`, which forwards signals to the child and reaps zombies.

CHECKLIST

- ☐ Can you explain how a shell implements `>`, `<` and `2>&1` with `open` and `dup2`?

- ☐ Can you wire a three-stage pipeline and say which descriptors each process must close?
- ☐ Can you name the common signals, what each one does by default, and which two can't be caught?
- ☐ Can you explain what a C signal handler may safely do?
- ☐ Can you describe how Ctrl-C, Ctrl-Z, `fg` and `bg` work with process groups?

GO DEEPER

REFERENCE Implementing a job control shell [The GNU C Library manual](#)

REFERENCE `signal(7)`: an overview of signals [The Linux man-pages project](#)

TUTORIAL Write a shell in C [Stephen Brennan](#)

PART II

Sharing the CPU

Three chapters on how many programs share a few cores. Threads run several lines of execution inside one process's memory, and a context switch moves a core from one thread to another. The classic scheduling policies trade finishing work quickly against responding quickly. Real kernels add fair shares, weights and many cores to those policies.

Threads and context switches

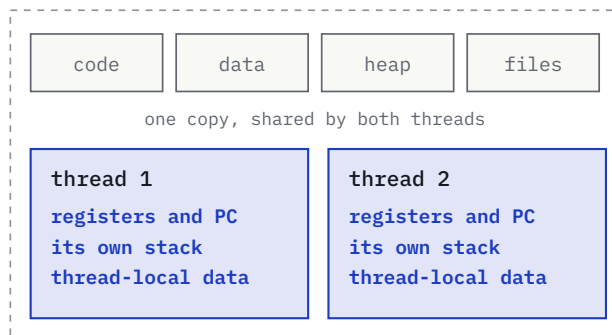
3 exercises

A thread is another line of execution inside the same process. It has its own registers and its own stack but shares everything else: the memory, the open files, the process ID. That makes sharing data between threads trivially easy, which is both why threads are useful and why Part III exists. This chapter compares threads with processes, shows what a context switch saves and restores, measures what creating each costs and how long a handover between them takes, and explains how runtimes from Go to Python map their own threads onto the kernel's.

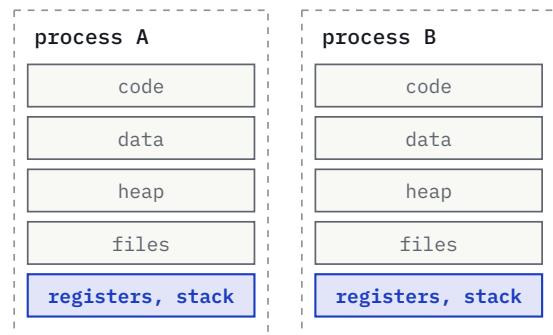
Processes and threads

Each thread has its own program counter and registers, its own stack, its own thread-local variables (in C, `errno` is one) and its own signal mask. Everything else belongs to the process and is shared by all its threads: the code, the global variables, the heap, the open file descriptors, the current directory, the user ID and the process ID. A pointer created in one thread works in every other. On Linux a thread is simply a task, like a process, created with `clone` and flags that ask it to share the parent's memory, files and signal handlers. Each thread has its own thread ID, and `ps -L` or `top -H` lists them.

ONE PROCESS, TWO THREADS



TWO PROCESSES



What's shared and what's private. Threads share everything except their registers and stacks, so one thread's change to the heap is instantly visible to the other, for better and for worse. Processes share nothing unless they arrange it (chapter 19).

Threads are cheaper than processes to create and to switch between, because there's no address space to copy or change, and they communicate through ordinary memory. Processes are safer: a crash, a memory leak or a corrupted heap in one process leaves the others alone, while a bad pointer in any thread brings down the whole process. Browsers

run each site in its own process for that reason, and databases such as PostgreSQL use a process per connection. Most servers written in Java, Go or C++ run many threads in one process.

What a context switch does

A **context switch** moves a core from one thread to another. It starts in the kernel, either because the running thread made a system call that blocks, such as a read with no data waiting, or because a timer interrupt fired and the scheduler decided the thread's turn was over. The kernel saves the running thread's registers and stack pointer, picks the next thread (chapters 5 and 6), loads that thread's stack pointer and registers, and returns to it as if from an interrupt. If the next thread belongs to a different process, the kernel also switches address spaces by loading a new page-table root, `CR3` on x86 or `TTBR0` on ARM. Tagging TLB entries with an address-space ID, PCID on x86 and ASID on ARM, means the switch doesn't have to throw away every cached translation (chapter 13).

The direct cost is a microsecond or two. The indirect cost is often larger: the new thread finds the caches and the TLB full of the old thread's data, and runs slowly until it has warmed them up again. That's why a scheduler doesn't switch as often as it could, and why a server that keeps thousands of threads busy can spend a surprising share of its time switching.

Measured

The example first shows the difference in sharing: four threads appending to one list fill it, while four forked processes each append to their own copy. It then times creating a thread against forking a process, and measures a handover with a ping-pong over two pipes, in which each side blocks until the other replies, so every round trip forces two wake-ups and two switches. The timings are one recorded run on an Apple M1 Pro.

EXAMPLE

threads.py · shared memory in threads but not processes, creation costs, and a handover measured by ping-pong

```
import os, sys, threading, time

# 1. Threads share one address space; forked processes each get a copy
results = []
def work(name):
    for i in range(1000): results.append((name, i))
threads = [threading.Thread(target=work, args=(n,)) for n in range(4)]
for t in threads: t.start()
for t in threads: t.join()
print(f"4 threads appended to one list: it now holds {len(results):,} items")
results.clear()
pids = []
for n in range(4):
    sys.stdout.flush()
    pid = os.fork()
    if pid == 0: work(n); os._exit(0)          # appends to the child's own copy
    pids.append(pid)
```

```

for pid in pids: os.waitpid(pid, 0)
print(f"4 forked processes appended to 'the same' list: the parent's holds {len(results):,}")

# 2. What creating each costs
def timed(fn, n):
    t0 = time.perf_counter(); fn(n); return (time.perf_counter() - t0) / n * 1e6
def make_threads(n):
    for _ in range(n):
        t = threading.Thread(target=lambda: None); t.start(); t.join()
def make_processes(n):
    for _ in range(n):
        pid = os.fork()
        if pid == 0: os._exit(0)
        os.waitpid(pid, 0)
print(f"\nstart and join a thread:      {timed(make_threads, 2000):6.1f} μs")
print(f"fork and wait for a process:  {timed(make_processes, 500):6.1f} μs")

# 3. A context switch, measured by ping-pong: each side blocks until the other answers
def ping_pong(n, use_threads):
    a_r, a_w = os.pipe(); b_r, b_w = os.pipe()
    def echo():
        for _ in range(n): os.write(b_w, os.read(a_r, 1))
    if use_threads:
        other = threading.Thread(target=echo); other.start()
    else:
        sys.stdout.flush()
        pid = os.fork()
        if pid == 0: echo(); os._exit(0)
    t0 = time.perf_counter()
    for _ in range(n):
        os.write(a_w, b"x"); os.read(b_r, 1)      # hand over, then wait to be handed back
    elapsed = time.perf_counter() - t0
    other.join() if use_threads else os.waitpid(pid, 0)
    for fd in (a_r, a_w, b_r, b_w): os.close(fd)
    return elapsed / n / 2 * 1e6                  # two handovers per round trip
N = 20_000
print(f"\none handover between two processes: {ping_pong(N, False):5.1f} μs")
print(f"one handover between two threads:    {ping_pong(N, True):5.1f} μs")
print(f"(this machine has {os.cpu_count()} cores; a handover includes a wake-up, a switch and a pipe)")

```

OUTPUT

```

4 threads appended to one list: it now holds 4,000 items
4 forked processes appended to 'the same' list: the parent's holds 0

start and join a thread:      41.8 μs
fork and wait for a process:  904.5 μs

one handover between two processes:  2.1 μs
one handover between two threads:    1.7 μs
(this machine has 8 cores; a handover includes a wake-up, a switch and a pipe)

```

Starting and joining a thread took about 42 μs, and forking and waiting for a process about 0.9 ms, over twenty times as long, because the kernel had to duplicate the page tables of a whole Python process. Each handover took about 2 μs, and that includes waking a thread blocked on a pipe, scheduling it, and the pipe read and write, so the switch itself is only part of it. Linux on a typical server gives similar orders of magnitude, with a faster `fork` .

The practical lessons: create threads ahead of time and reuse them, which is what a thread pool does, and don't design a system that needs a context switch for every small piece of work.

Kernel threads and user threads

The threads above are **kernel threads**: the kernel knows about each one and schedules it. That's the model of POSIX threads in C, of Java's platform threads and of Rust's standard library, one language thread per kernel thread. Each needs a kernel stack and a user stack, 8 MiB of reserved address space by default on Linux, of which typically only tens of kilobytes are ever touched, and every switch between them goes through the kernel.

Many runtimes instead multiplex many lightweight **user threads** onto a few kernel threads, switching between them in user space for a few hundred nanoseconds. Go's goroutines start with 2 KiB stacks that grow as needed, and the Go runtime runs them on roughly one kernel thread per core. Java added virtual threads on the same model in JDK 21, and Erlang has always worked this way. The difficulty is blocking: when a user thread makes a system call that blocks, it blocks the kernel thread under it, and every user thread scheduled there. These runtimes therefore use non-blocking I/O internally (chapter 18) and hand blocking calls to spare kernel threads.

Python's threads

CPython's threads are real kernel threads, but in the standard build a global interpreter lock, the **GIL**, lets only one of them run Python code at a time. A thread releases it while waiting for I/O, so threads work well for programs that spend their time waiting on networks and disks, and do nothing for programs that spend it computing in Python. The example's thread handover includes passing the GIL. For CPU-bound work, use processes (`multiprocessing` or `concurrent.futures.ProcessPoolExecutor`), or libraries such as NumPy whose C code releases the lock. A free-threaded build without the GIL became an officially supported option in Python 3.14, which makes the data races of chapter 7 real for Python code too.

COMMON TRAPS

One thread per request with no limit, until memory or the scheduler gives out under load.

.....

Expecting Python threads to speed up CPU-bound code on a build with the GIL.

.....

Creating threads in a hot path instead of reusing them from a pool.

.....

Thread-local state left over in a pooled thread from the previous task.

.....

Forgetting that one crashing thread kills the whole process, including every other thread's work

EXERCISES

EXERCISE 4.1 Mine or ours?

For each, say whether every thread has its own or all threads share one: a global variable; a local variable in a function both threads are running; an object on the heap; an open file descriptor and its file offset; the program counter; C's `errno`.

SOLUTION

Shared: the global variable, the heap object, and the file descriptor together with its offset, so two threads reading the same descriptor take turns consuming the file. Private: local variables, which live on each thread's own stack, so two threads running the same function have separate copies; the program counter and the other registers; and `errno`, which modern C libraries make thread-local precisely so that one thread's failed call doesn't overwrite another's error. A local variable stops being private as soon as a pointer to it is handed to another thread, which is one of the easier ways to create a race.

EXERCISE 4.2 Sizing a thread pool

Each request spends 5 ms computing and 45 ms waiting for a database. The machine has 8 cores. How many threads keep all the cores busy? How does the answer change for a Python service on a build with the GIL?

SOLUTION

Each thread computes for 5 ms out of every 50, so it keeps a core busy 10% of the time. To keep 8 cores busy you need $8 / 0.1 = 80$ threads, which is Little's law: the work in progress equals the throughput times the time each piece spends in the system. With fewer, cores sit idle while every thread waits on the database; with many more, extra threads only queue and add switches. Under the GIL, Python code runs on one core at a time however many cores there are, so about $1 / 0.1 = 10$ threads saturate it, and the way to use 8 cores is 8 processes of about 10 threads each. In both cases, check the database can take that many concurrent queries, because the pool often moves the bottleneck there.

EXERCISE 4.3 The switching tax

A server on 8 cores performs 200,000 context switches a second, each costing about 2 μ s directly. What share of the machine is that? Give two ways to reduce it.

SOLUTION

200,000 $\times$ 2 μ s is 0.4 seconds of CPU time per second, 0.4 of a core, or 5% of the 8. The indirect cost of cold caches and TLBs can easily double that, so 10% is a fair estimate. To reduce it: do more work per wake-up, for example by batching queue items or handling every ready connection in one pass (chapter 18); use fewer threads than there are blocking points, with non-blocking I/O, so threads rarely sleep and wake; and pin hot threads to cores so they keep their caches warm.

`vmstat 's cs` column and `pidstat -w` show the switch rate on Linux.

CHECKLIST

- ☐ Can you list what threads of one process share and what each has of its own?
- ☐ Can you weigh threads against processes for isolation and cost?
- ☐ Can you describe the steps of a context switch, and its indirect costs?
- ☐ Can you explain how goroutines or virtual threads differ from kernel threads?
- ☐ Can you say when Python threads help and when they don't?

GO DEEPER

BOOK Operating Systems: Three Easy Pieces, chapters 26 and 27: threads Arpaci-Dusseau & Arpaci-Dusseau

DOC Scalable Go scheduler design Dmitry Vyukov

PROPOSAL PEP 703: making the global interpreter lock optional Sam Gross

Scheduling

3 exercises

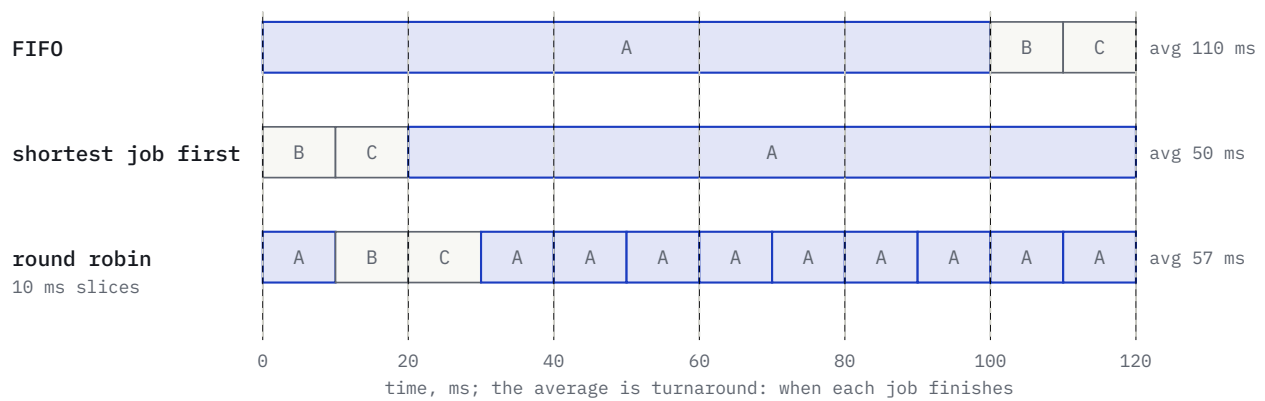
When more threads are ready to run than there are cores, the scheduler decides which run. Every policy trades one goal for another: finishing jobs soon, responding to them soon, using the processor fully, treating everyone fairly. This chapter simulates the classic policies on small workloads: first come first served, shortest job first, shortest time to completion and round robin. It then builds the multi-level feedback queue, which responds quickly to interactive programs without knowing in advance how long anything will run, and shows how a program can game it and how the scheduler stops that.

What a scheduler optimises

Chapters 2 and 4 covered the **mechanism**: the timer interrupt that hands control back to the kernel, and the context switch that moves a core to another thread. This chapter is about **policy**, the rule for which thread runs next. Two measures dominate. A job's **turnaround time** is how long it takes from arriving to finishing, which matters for batch work such as a build or a report. Its **response time** is how long it waits before it first runs, which is what a person at a keyboard or a client waiting on a request feels. Throughput, the work finished per second, and fairness, whether each job gets its share, complete the list. No policy wins on all of them.

First come, first served, and the convoy

The simplest policy runs jobs in the order they arrive, each to completion. Its weakness shows as soon as a long job arrives first. With A needing 100 ms and B and C 10 ms each, B and C wait behind A like cars behind a truck, the **convoy effect**, and the average turnaround is 110 ms. Running the shortest job first brings it down to 50 ms, and when all jobs arrive together, shortest job first is provably the best possible order for average turnaround.



The same three jobs under three policies, to scale. Putting the long job last halves the average turnaround; round robin comes close to that without knowing how long anything will take, and gives every job its first turn within 20 ms.

Shortest job first has two problems. It needs to know how long each job will run, which a general-purpose operating system never does. And if it's **non-preemptive**, letting each job run until it finishes, it's helpless when short jobs arrive just after a long one has started: in the example's second workload B and C arrive 10 ms after A and wait 90 ms for it, and the average turnaround goes back up to 103 ms. The preemptive version, **shortest time to completion first**, interrupts A when they arrive and brings the average back to 50 ms.

Round robin and response time

Round robin gives each ready job a fixed time slice in turn and puts it at the back of the queue when the slice ends. It needs to know nothing about the jobs, and it's excellent for response time: with 1 ms slices every job runs within 2 ms of arriving. It's poor for turnaround, because it stretches every job out to near the end. The slice length is the dial between the two. Short slices give quick responses but spend more time switching, and with 20 CPU-bound threads ready, a 100 ms slice means each waits up to 1.9 seconds for its turn. Desktop and server kernels use slices of a few milliseconds.

EXAMPLE

scheduling.py · FIFO, SJF, STCF and round robin on two workloads, then a multi-level feedback queue, gamed and fixed

```
# A scheduler simulator in 1 ms ticks. Each job: (name, arrival time, CPU time needed).
def simulate(jobs, policy, quantum=None):
    left = {n: run for n, _, run in jobs}
    arrive = {n: at for n, at, _ in jobs}
    first, done, queue, t, current, used = {}, {}, [], 0, None, 0
    while len(done) < len(jobs):
        queue += [n for n, at, _ in jobs if at == t]           # newly arrived jobs join
        if current and policy == "STCF":                      # preempt for a shorter
            job                                               job
            queue.append(current); current = None
        if current and policy == "RR" and used == quantum:   # time slice over:
            requeue
```

```

        queue.append(current); current = None
    if current is None and queue:
        if policy in ("SJF", "STCF"): queue.sort(key=lambda n: left[n]) # shortest first
        current, used = queue.pop(0), 0
        first.setdefault(current, t)
    if current:
        left[current] -= 1; used += 1
        if left[current] == 0: done[current] = t + 1; current = None
    t += 1
turnaround = sum(done[n] - arrive[n] for n in done) / len(jobs)
response = sum(first[n] - arrive[n] for n in first) / len(jobs)
return turnaround, response

def table(title, jobs, policies):
    print(title)
    for label, policy, q in policies:
        ta, rs = simulate(jobs, policy, q)
        print(f"    {label:28} average turnaround {ta:6.1f} ms    average response {rs:5.1f} ms")

table("A needs 100 ms, B and C 10 ms each, all arriving at time 0:",
      [("A", 0, 100), ("B", 0, 10), ("C", 0, 10)],
      [("FIFO", "FIFO", None), ("shortest job first", "SJF", None), ("round robin, 1 ms slice",
        "RR", 1)])

table("\nA arrives at 0; B and C arrive at 10 ms:",
      [("A", 0, 100), ("B", 10, 10), ("C", 10, 10)],
      [("shortest job first", "SJF", None), ("shortest time to completion", "STCF", None),
        ("round robin, 10 ms slice", "RR", 10)])

# A multi-level feedback queue: a 10 ms time slice, demotion after 10 ms of CPU at a level
def mlfq(jobs, ticks=3000, levels=3, boost_every=None, charge_blocked=True):
    """jobs: name -> (burst, io): run `burst` ms, then wait `io` ms for I/O, forever (io=0:
        CPU-bound)."""
    level = dict.fromkeys(jobs, 0); used = dict.fromkeys(jobs, 0); burst = dict.fromkeys(jobs,
        0)
    wake = dict.fromkeys(jobs, 0); queued = dict.fromkeys(jobs, 0); cpu = dict.fromkeys(jobs,
        0)
    worst = dict.fromkeys(jobs, 0); waited = dict.fromkeys(jobs, 0); runs = dict.fromkeys(jobs,
        0)
    current, slice_used = None, 0
    for t in range(ticks):
        if boost_every and t and t % boost_every == 0:
            for n in jobs: level[n], used[n] = 0, 0 # everyone back to the
            top
        ready = [n for n in jobs if wake[n] <= t and n != current]
        if current and any(level[n] < level[current] for n in ready): # a higher-priority job
            woke up
            queued[current] = t; current = None
        if current is None and ready: # highest level, then
            longest waiting
            current = min(ready, key=lambda n: (level[n], max(queued[n], wake[n])))
            w = t - max(queued[current], wake[current])
            worst[current] = max(worst[current], w); waited[current] += w; runs[current] += 1
            slice_used = 0
        if current is None: continue
        n = current
        cpu[n] += 1; used[n] += 1; slice_used += 1; burst[n] += 1
        b, io = jobs[n]
        if io and burst[n] == b: # gives up the CPU for
            I/O
            burst[n] = 0; wake[n] = t + 1 + io; current = None
            if not charge_blocked: used[n] = 0 # the old rule: forget
            its usage
        if used[n] >= 10 and level[n] < levels - 1: # used up its allotment:
            demote
            level[n] += 1; used[n] = 0
            if current: queued[n] = t + 1; current = None

```

```

        elif current and slice_used == 10:                                # slice over: back of
            the queue
            queued[n] = t + 1; current = None
    for n in jobs:                                                         # still waiting at the
        end?
        if n != current and wake[n] <= ticks:
            worst[n] = max(worst[n], ticks - max(queued[n], wake[n]))
    return {n: (cpu[n] / ticks, waited[n] / runs[n], worst[n]) for n in jobs}

def show(title, result):
    print(title)
    for n, (share, mean, worst) in result.items():
        print(f"    {n:12} {share:6.1%} of the CPU    "
              f"waits to run: {mean:5.1f} ms on average, {worst:4d} ms at most")

mix = {"interactive": (1, 9), "batch-1": (100, 0), "batch-2": (100, 0)}
print()
show("Round robin, 10 ms slices: an interactive job (1 ms of CPU, then 9 ms of I/O) and two
    batch jobs:",
    mlfq(mix, levels=1))
show("\nThe same jobs under MLFQ with three levels and a boost every 100 ms:",
    mlfq(mix, boost_every=100))
game = {"gamer": (9, 1), "batch-1": (100, 0), "batch-2": (100, 0)}
show("\nA job that yields after 9 ms, every time, when blocking resets its allotment:",
    mlfq(game, boost_every=100, charge_blocked=False))
show("\nThe same job when the scheduler counts all the CPU it used:",
    mlfq(game, boost_every=100))
crowd = {f"chatty-{i}": (1, 4) for i in range(5)}, "batch": (100, 0)}
show("\nFive chatty jobs fill the top level (old rule, no boost):",
    mlfq(crowd, charge_blocked=False))
show("\nThe same mix with a boost every 100 ms:",
    mlfq(crowd, charge_blocked=False, boost_every=100))

```

OUTPUT

A needs 100 ms, B and C 10 ms each, all arriving at time 0:

FIFO	average turnaround	110.0 ms	average response	70.0 ms
shortest job first	average turnaround	50.0 ms	average response	10.0 ms
round robin, 1 ms slice	average turnaround	59.7 ms	average response	1.0 ms

A arrives at 0; B and C arrive at 10 ms:

shortest job first	average turnaround	103.3 ms	average response	63.3 ms
shortest time to completion	average turnaround	50.0 ms	average response	3.3 ms
round robin, 10 ms slice	average turnaround	50.0 ms	average response	3.3 ms

Round robin, 10 ms slices: an interactive job (1 ms of CPU, then 9 ms of I/O) and two batch jobs:

interactive	4.8% of the CPU	waits to run:	10.9 ms on average,	11 ms at most
batch-1	47.7% of the CPU	waits to run:	10.9 ms on average,	11 ms at most
batch-2	47.6% of the CPU	waits to run:	11.0 ms on average,	11 ms at most

The same jobs under MLFQ with three levels and a boost every 100 ms:

interactive	8.9% of the CPU	waits to run:	1.2 ms on average,	11 ms at most
batch-1	45.6% of the CPU	waits to run:	8.7 ms on average,	11 ms at most
batch-2	45.5% of the CPU	waits to run:	8.7 ms on average,	11 ms at most

A job that yields after 9 ms, every time, when blocking resets its allotment:

gamer	72.9% of the CPU	waits to run:	2.3 ms on average,	19 ms at most
batch-1	13.6% of the CPU	waits to run:	18.9 ms on average,	19 ms at most
batch-2	13.5% of the CPU	waits to run:	19.0 ms on average,	19 ms at most

The same job when the scheduler counts all the CPU it used:

gamer	27.0% of the CPU	waits to run:	13.9 ms on average,	20 ms at most
batch-1	36.5% of the CPU	waits to run:	14.1 ms on average,	19 ms at most
batch-2	36.5% of the CPU	waits to run:	14.1 ms on average,	19 ms at most

Five chatty jobs fill the top level (old rule, no boost):

chatty-0	19.9% of the CPU	waits to run:	0.0 ms on average,	10 ms at most
chatty-1	19.9% of the CPU	waits to run:	0.0 ms on average,	10 ms at most
chatty-2	19.9% of the CPU	waits to run:	0.0 ms on average,	10 ms at most
chatty-3	19.9% of the CPU	waits to run:	0.0 ms on average,	10 ms at most
chatty-4	19.9% of the CPU	waits to run:	0.0 ms on average,	10 ms at most
batch	0.3% of the CPU	waits to run:	5.0 ms on average,	2985 ms at most

The same mix with a boost every 100 ms:

chatty-0	18.0% of the CPU	waits to run:	0.6 ms on average,	10 ms at most
chatty-1	18.0% of the CPU	waits to run:	0.6 ms on average,	10 ms at most
chatty-2	18.0% of the CPU	waits to run:	0.6 ms on average,	10 ms at most
chatty-3	18.0% of the CPU	waits to run:	0.6 ms on average,	10 ms at most
chatty-4	18.0% of the CPU	waits to run:	0.6 ms on average,	10 ms at most
batch	10.0% of the CPU	waits to run:	87.0 ms on average,	90 ms at most

Not knowing the future: the multi-level feedback queue

A real scheduler has to favour short, interactive jobs without being told which they are. The **multi-level feedback queue** (MLFQ), introduced by Fernando Corbató for MIT's CTSS time-sharing system in 1962, learns it from behaviour. There are several priority levels. A job at a higher level always runs before one at a lower level, and jobs at the same level share it round robin. Every new job starts at the top. A job that uses up its allotment of CPU time at a level moves down a level, so CPU-bound jobs sink, while a job that runs briefly and then waits for I/O stays near the top.

The example shows the effect. An interactive job that wants 1 ms of CPU every 10 ms competes with two CPU-bound batch jobs. Under plain round robin it waits about 11 ms every time it wakes, behind the batch jobs' slices, and gets only 4.8% of the CPU instead of the 10% it needs. Under MLFQ it waits 1.2 ms on average, because by then the batch jobs have sunk below it, and it gets 8.9%.

The rules need two refinements. The first version of the demotion rule reset a job's allotment whenever it gave up the CPU for I/O. A program that does a tiny I/O just before its allotment runs out then never sinks. The example's "gamer" yields after 9 ms of every 10 and takes 72.9% of the CPU from two jobs that should share it equally. Counting all the CPU a job uses at a level, however it's split up, demotes the gamer like anyone else, and the three jobs then get roughly equal shares. The second refinement fixes **starvation**. Five chatty jobs that together keep the top level busy leave a batch job at the bottom with 0.3% of the CPU and a wait of almost 3 seconds. A periodic **priority boost** that moves every job back to the top fixes that: with a boost every 100 ms, the batch job gets 10% and never waits more than 90 ms. The boost also helps a job whose behaviour changes, such as a compiler that turns into an editor.

Variants of MLFQ schedule Windows threads and macOS threads, and FreeBSD's ULE scheduler scores each thread's interactivity from how much it sleeps. Linux took a different path, fair queuing, which chapter 6 covers.

Optimising the average when users feel the tail. Measure the 99th percentile of response time.

Lengthening time slices for throughput on a machine that also serves interactive requests.

Busy-waiting for something another thread will do, burning the slice that thread needs.

Using `sleep(0)` or `sched_yield` as synchronisation. They promise nothing about who runs next.

Measuring latency on an idle machine and expecting the same numbers under load.

EXERCISES

EXERCISE 5.1 Three policies by hand

Jobs A, B and C arrive together and need 30, 20 and 10 ms. Work out the average turnaround and response time under FIFO (in the order A, B, C), shortest job first, and round robin with 1 ms slices.

SOLUTION

FIFO: A finishes at 30, B at 50 and C at 60, an average turnaround of $140 / 3 \approx 46.7$ ms; they start at 0, 30 and 50, an average response of 26.7 ms. Shortest job first runs C, B, A: they finish at 10, 30 and 60, average 33.3 ms, and start at 0, 10 and 30, average 13.3 ms. Round robin: each job runs once every 3 ms until C has had its 10 ms, at 30 ms; then A and B alternate until B finishes at 50; A finishes at 60. The average turnaround is again 46.7 ms, but the average response is 1 ms, since the jobs start at 0, 1 and 2. The example's `simulate` function gives the same numbers.

EXERCISE 5.2 Choosing the slice

A context switch costs 50 μ s once its effect on the caches is counted. What share of the CPU goes to switching with slices of 1 ms, 10 ms and 100 ms, if every slice ends in a switch? With 20 CPU-bound threads ready, how long can each wait for its turn?

SOLUTION

Each slice of length q costs one switch, so the overhead is $50 \mu\text{s} / (q + 50 \mu\text{s})$: 4.8% at 1 ms, 0.5% at 10 ms and 0.05% at 100 ms. The wait is up to 19 other slices: 19 ms, 190 ms and 1.9 s. Somewhere around 10 ms keeps the overhead under 1% while a wait stays below a fifth of a second, and

that's roughly where general-purpose schedulers sit. They also shorten slices automatically when many threads are ready, so the time for everyone to get a turn stays bounded.

EXERCISE 5.3 Why accounting fixes gaming

Explain why charging a job for all the CPU it uses at a level stops the gamer, and what it costs a genuinely interactive job. Why does the priority boost make that cost acceptable?

SOLUTION

Under the old rule the allotment measured only the current run, so any job that blocked before its allotment ran out started afresh, and a job could stay at the top by blocking for a moment every 9 ms. Counting the total used at the level, across runs, means the gamer sinks after 10 ms of CPU however it splits them up. The cost is that an honest interactive job sinks too, just more slowly: at 1 ms of CPU per 10 ms it uses its allotment after about 100 ms and is demoted. The periodic boost restores it, and because it uses so little CPU it spends most of each boost period at or near the top, still above the CPU-bound jobs, which sink within their first few slices. The example's interactive job ran with an average wait of 1.2 ms under exactly these rules.

CHECKLIST

- ☐ Can you define turnaround and response time, and say which each policy favours?
- ☐ Can you explain the convoy effect and why shortest job first fixes it?
- ☐ Can you explain why preemption matters when jobs arrive at different times?
- ☐ Can you reason about the time-slice trade-off with numbers?
- ☐ Can you state the MLFQ rules, and explain gaming and starvation and their fixes?

GO DEEPER

BOOK Operating Systems: Three Easy Pieces, chapter 7: scheduling Arpaci-Dusseau & Arpaci-Dusseau

BOOK Operating Systems: Three Easy Pieces, chapter 8: the multi-level feedback queue Arpaci-Dusseau & Arpaci-Dusseau

Fair shares and many cores

3 exercises

A multi-level feedback queue guesses which jobs deserve priority. A fair-share scheduler instead promises each thread a defined share of the processor, in proportion to a weight. This chapter builds weighted fair queueing the way Linux does it, with a virtual clock per thread that runs faster or slower according to its weight. It compares that with random lottery scheduling and deterministic stride scheduling. It then turns to machines with many cores, where the scheduler also decides where each thread runs, and to the priority inversion that once kept resetting a spacecraft on Mars.

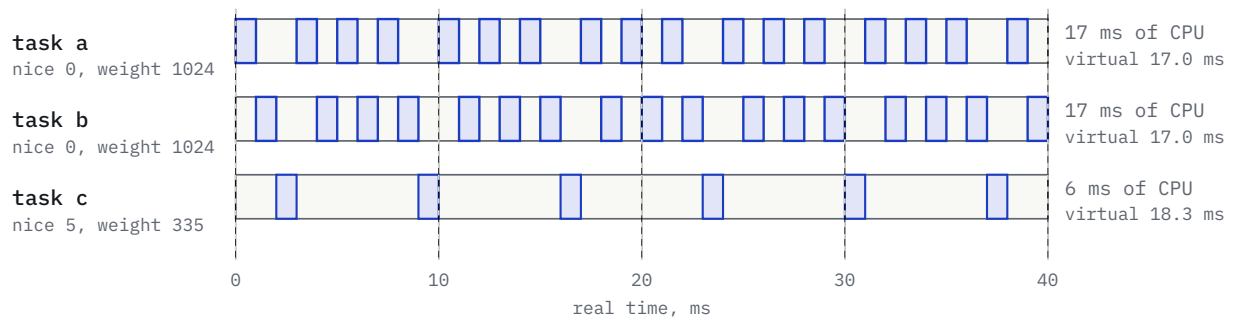
Proportional share

In proportional-share scheduling each thread holds a weight, and when threads compete each gets CPU time in proportion to its weight. Carl Waldspurger and William Weihl's **lottery scheduling** (1994) did it with tickets and chance: at every time slice, draw a ticket at random, and the holder runs. A thread with 75 of 100 tickets wins 75% of the draws in the long run. Lottery is simple, needs no global state, and handles new threads instantly, but it's only fair on average. Over short windows the shares wander, which is exactly when a user notices. Waldspurger's **stride scheduling** (1995) removes the randomness. Each thread has a "pass" value that grows by a stride, inversely proportional to its tickets, every time it runs, and the scheduler always picks the thread with the smallest pass.

Virtual runtime

Linux's Completely Fair Scheduler, CFS, written by Ingo Molnar and in use from 2007, applies the stride idea to time itself. Each thread has a **virtual runtime**: a clock that advances while the thread runs, at a rate inversely proportional to its weight. The scheduler always runs the thread with the smallest virtual runtime, kept in a red-black tree so the smallest is quick to find. A thread at the default weight's virtual clock runs at real speed; a thread with a third of the weight has a clock that runs three times as fast, so it reaches the front of the queue a third as often.

WHO RUNS IN EACH MILLISECOND



c's virtual clock runs $1024 / 335 = 3.06$ times faster, so it is picked about a third as often

Fair queueing over 40 ms. The scheduler doesn't count shares; it only ever runs the task whose virtual clock is furthest behind, and the weights make the shares come out right.

Weights come from a thread's **nice** value, from -20 to 19 , through a fixed table in the kernel: nice 0 is 1024 , and each step changes the weight by about a quarter, so that each step moves about 10% of the CPU between two competing threads. One detail matters for interactive programs. A thread that has been asleep has a small virtual runtime, and if it kept it, it would monopolise the processor on waking until it caught up. So on waking, its virtual runtime is moved up to just below the smallest of the running threads, a few milliseconds of credit and no more. In Linux 6.6, in 2023, Peter Zijlstra replaced CFS's selection rule with **EEVDF**, earliest eligible virtual deadline first. It keeps the virtual clocks but also gives each thread a deadline based on how far it's behind and how long a slice it asks for, so latency-sensitive threads can request short slices and run sooner without getting more CPU in total.

EXAMPLE

fairness.py · weighted fair queueing with nice values, lottery against stride, and five jobs on four cores

```
import random

# 1. Weighted fair queueing, as Linux's CFS does it: always run the task with the least virtual
# runtime, and charge each task's virtual clock at  $1024 / \text{weight}$  times real time.
WEIGHT = {-5: 3121, 0: 1024, 5: 335, 10: 110, 19: 15} # from the kernel's nice-to-
weight table

def fair(tasks, ms=3000, asleep_until=None):
    asleep_until = asleep_until or {}
    vruntime = dict.fromkeys(tasks, 0.0); cpu = dict.fromkeys(tasks, 0)
    for t in range(ms):
        awake = [n for n in tasks if asleep_until.get(n, 0) <= t]
        for n in awake:
            others = [vruntime[m] for m in awake if m != n]
            if t and asleep_until.get(n, 0) == t and others: # just woke up: no
                banked credit, # only a 3 ms head
                vruntime[n] = max(vruntime[n], min(others) - 3)
            start
            n = min(awake, key=lambda n: (vruntime[n], n))
            cpu[n] += 1
            vruntime[n] += 1024 / WEIGHT[tasks[n]]
    return {n: cpu[n] / ms for n in tasks}
```

```

print("three CPU-bound tasks, nice 0, 0 and 5:",
      ", ".join(f"{n} {s:.1%}" for n, s in fair({"a": 0, "b": 0, "c": 5}).items()))
print("expected from the weights 1024 : 1024 : 335:",
      ", ".join(f"w / 2383:.1%" for w in (1024, 1024, 335)))
print("nice -5 against nice 19:", ", ".join(f"{n} {s:.1%}" for n, s in fair({"x": -5, "y":
19}).items()))
woke = fair({"a": 0, "b": 0}, ms=2000, asleep_until={"b": 1000})
print(f"b sleeps for the first second, then competes: a {woke['a']:.1%}, b {woke['b']:.1%} "
      f"(once awake, b gets half, not a second of catch-up)")

# 2. Lottery against stride scheduling: 75 and 25 tickets, measured over short windows
random.seed(6)
def lottery(slices):
    return sum(random.random() < 0.75 for _ in range(slices)) / slices    # A's share of the
    window
def stride(slices):
    passes, a = {"A": 0.0, "B": 0.0}, 0
    for _ in range(slices):
        n = min(passes, key=lambda n: (passes[n], n))
        a += n == "A"
        passes[n] += 1 / (75 if n == "A" else 25)    # stride = 1 / tickets
    return a / slices
lot = [lottery(20) for _ in range(10)]
strd = [stride(20) for _ in range(10)]
print(f"\nA's share in ten windows of 20 time slices, lottery: {' '.join(f'{x:.0%}' for x in
lot)}")
print(f"A's share in ten windows of 20 time slices, stride: {' '.join(f'{x:.0%}' for x in
strd)}")

# 3. Many cores: five equal jobs on four cores, with and without migration between cores
def cores(jobs=5, n=4, ms=1000, migrate=False):
    cpu = [0] * jobs
    for t in range(ms):
        if migrate:
            running = sorted(range(jobs), key=lambda j: (cpu[j], j))[:n]    # balance: least-
            served first
        else:
            queues = [[j for j in range(jobs) if j % n == c] for c in range(n)]    # pinned at
            start
            running = [q[t % len(q)] for q in queues]    # round robin on each
            core
            for j in running: cpu[j] += 1
    return [c / ms for c in cpu]
print(f"\nfive jobs on four cores, never migrated: {' '.join(f'{s:.0%}' for s in cores())}")
print(f"five jobs on four cores, balanced: {' '.join(f'{s:.0%}' for s in
cores(migrate=True))}")

```

OUTPUT

```

three CPU-bound tasks, nice 0, 0 and 5: a 43.0%, b 43.0%, c 14.1%
expected from the weights 1024 : 1024 : 335: 43.0%, 43.0%, 14.1%
nice -5 against nice 19: x 99.5%, y 0.5%
b sleeps for the first second, then competes: a 75.0%, b 25.1% (once awake, b gets half, not a
second of catch-up)

A's share in ten windows of 20 time slices, lottery: 70% 65% 80% 70% 80% 75% 70% 80% 55% 60%
A's share in ten windows of 20 time slices, stride: 75% 75% 75% 75% 75% 75% 75% 75% 75% 75%

five jobs on four cores, never migrated: 50% 100% 100% 100% 50%
five jobs on four cores, balanced: 80% 80% 80% 80% 80%

```

The simulated shares match the weights to the decimal: 43.0%, 43.0% and 14.1% for nice 0, 0 and 5, and 99.5% to 0.5% for nice -5 against nice 19. The task that slept for a second got half the CPU once it woke, not a second of catch-up. The lottery's windows of 20 slices gave A anywhere from 55% to 80% of a share that should be 75%; stride gave exactly 75% every time.

Many cores

With several cores, the scheduler also decides where each thread runs. A single shared run queue would be fair but slow, because every core would contend for its lock, so kernels keep a run queue per core and balance them: periodically, and whenever a core goes idle and steals work from a busy one. Balancing fights with **cache affinity**. A thread that moves to another core leaves its warm caches behind, and on machines with several memory controllers, NUMA machines, it may end up far from its memory. The example's five equal jobs on four cores show what happens without balancing: the two jobs that share a core get 50% each while the other three get a whole core, and balancing gives all five 80%. Programs can restrict where a thread may run with `sched_setaffinity` or `taskset`, which helps for a latency-critical thread on a dedicated core and hurts when it overrides a balancer that knew better.

Priorities, real time and inversion

Above the fair classes, Linux has **real-time** classes. A `SCHED_FIFO` thread runs until it blocks or a higher-priority real-time thread wakes, and always ahead of every normal thread; `SCHED_RR` adds time slices among equals, and `SCHED_DEADLINE` schedules by declared runtimes and deadlines. A real-time thread stuck in a loop could freeze the machine, so by default Linux reserves 5% of each second for normal threads.

Strict priorities have a famous failure, **priority inversion**. In July 1997 the Mars Pathfinder lander began resetting itself every few days. A low-priority task held a mutex that a high-priority bus task needed. Medium-priority tasks, which needed neither, kept preempting the low-priority one, so it never released the mutex, and the high-priority task missed its deadline and a watchdog reset the system. The fix was **priority inheritance**: while a thread holds a lock that a higher-priority thread is waiting for, it runs at the waiter's priority. The engineers turned on the option, already in the operating system, VxWorks, by patching the software from Earth.

Containers apply the same ideas to groups of threads. A cgroup's `cpu.weight` divides CPU between groups in proportion, exactly like nice values between threads, and `cpu.max` sets a hard quota (chapter 20). In Kubernetes, a container's CPU request becomes its weight and its CPU limit becomes its quota.

COMMON TRAPS

Expecting `nice` to slow a job on an idle machine. Weights only matter when threads compete.

Pinning threads to cores by hand and leaving some cores overloaded while others idle.

Giving real-time priority to code that can spin, which can starve the rest of the machine.

Sharing a mutex between high- and low-priority threads without priority inheritance.

Reading Linux's load average as CPU use. It also counts threads blocked on disk I/O.

EXERCISES

EXERCISE 6.1 Shares from weights

Three CPU-bound threads run at nice 0, 0 and 10; the weight for nice 10 is 110. What share does each get? What share does the nice-10 thread get if the other two finish?

SOLUTION

The total weight is $1024 + 1024 + 110 = 2158$, so the shares are $1024 / 2158 = 47.5\%$ each for the nice-0 threads and $110 / 2158 = 5.1\%$ for the nice-10 thread. When it's alone, it gets 100%: a weight is a share of whatever is contended, not a cap. To cap a thread or a group at a fixed amount of CPU, use a quota such as a cgroup's `cpu.max`, with the throttling chapter 20 describes.

EXERCISE 6.2 How long until the lottery is fair?

A thread holds 75% of the tickets. Over a window of n slices, its share has a standard deviation of $\sqrt{(0.75 \times 0.25 / n)}$. How far off is a typical 20-slice window? How many slices until a typical window is within one percentage point, and how long is that with 10 ms slices?

SOLUTION

For $n = 20$ the standard deviation is $\sqrt{(0.1875 / 20)} \approx 0.097$, so a typical window is about 10 percentage points off, consistent with the example's 55% to 80%. For a standard deviation of 0.01 you need $0.1875 / n = 0.0001$, so $n \approx 1,875$ slices, which at 10 ms each is almost 19 seconds. A user notices unfairness over a fraction of a second, so random scheduling alone is too coarse, and deterministic schemes such as stride scheduling and virtual runtime are what real kernels use.

EXERCISE 6.3 Inversion, step by step

Threads H, M and L have high, medium and low priority on one core. L takes a mutex; H wakes and blocks on it; M wakes and runs a long computation. Describe what happens with strict priorities, and then with priority inheritance.

SOLUTION

With strict priorities: H blocks waiting for the mutex. M has higher priority than L, so M runs, for as long as its computation lasts. L can't run, so it can't release the mutex, and H, the most important thread, waits for M, which doesn't even use the mutex. Its priority has effectively been inverted below M's. With priority inheritance, when H blocks on L's mutex, L temporarily runs at H's priority. M can no longer preempt it, L finishes its critical section and releases the mutex, drops back to low priority, and H runs at once. POSIX mutexes offer this as the `PTHREAD_PRIO_INHERIT` protocol, and Linux's real-time mutexes use it.

CHECKLIST

- ☐ Can you explain lottery and stride scheduling and why stride is fairer over short windows?
- ☐ Can you explain virtual runtime and compute shares from nice weights?
- ☐ Can you say what happens to a sleeping thread's virtual runtime when it wakes, and why?
- ☐ Can you explain per-core run queues, load balancing and cache affinity?
- ☐ Can you describe priority inversion and priority inheritance?

GO DEEPER

BOOK Operating Systems: Three Easy Pieces, chapters 9 and 10: proportional share and multiprocessor scheduling Arpaci-Dusseau & Arpaci-Dusseau

DOCS The EEVDF scheduler The Linux kernel documentation

ARTICLE What really happened on Mars? Glenn Reeves, via Mike Jones

PART III

Concurrency

Five chapters on what goes wrong when threads share memory, and the tools that make sharing safe. They cover the race itself, and the atomic instructions and memory models underneath every fix; locks and how to build one; condition variables and semaphores for waiting until something is true; and deadlock, livelock and starvation. The last chapter covers the alternatives to shared-memory threads: event loops, async/await, actors and channels. The examples check every possible interleaving of small programs, so the bugs appear on every run.

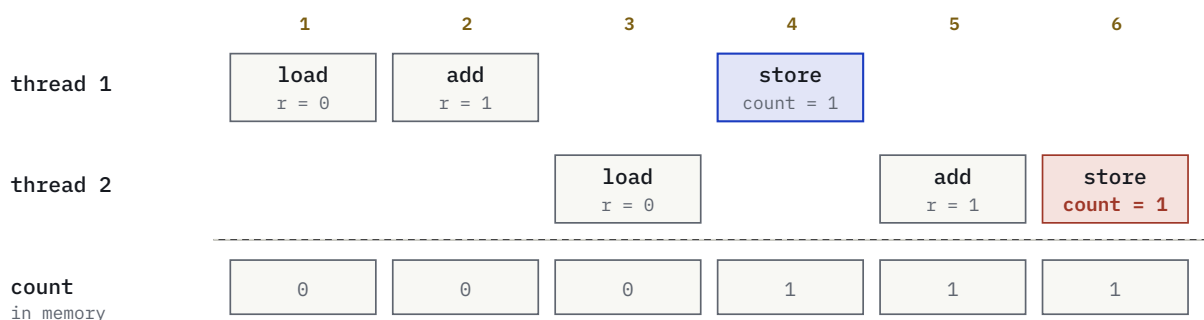
Race conditions and atomicity

3 exercises

Two threads that read and write the same memory without coordinating produce results that depend on timing. The trouble isn't that such code fails often. It fails rarely, differently each time, and almost never while you watch. This chapter makes races deterministic by trying every possible interleaving of tiny programs. It shows why `count += 1` loses an update in 18 of 20 orders, and how checking a balance before debiting it lets two withdrawals overdraw an account. It then covers the atomic instructions that fix these races, and the hardware memory models that make plain variables unsafe for coordination.

A race in three instructions

`count += 1` looks like one operation, but the processor does it in three: load `count` into a register, add one, store the register back. Each thread has its own registers, so if two threads both load before either stores, both store the same value and one increment is lost. Code that touches shared data and must not be interleaved with another thread's is a **critical section**, and a **race condition** is a bug whose outcome depends on how threads happen to interleave. When two threads access the same memory, at least one of them writes, and nothing orders the accesses, the program has a **data race**. In C and C++ that is undefined behaviour, so the compiler may assume it never happens and the program may do anything.



Two increments ran, but count ends at 1: thread 2 loaded 0 before thread 1 stored, then stored over it.

One of the 18 orders that lose an update. Each thread did everything right on its own; the bug is in the interleaving, which neither thread can see.

Every order at once

Running a racy program a thousand times proves little, because the bad interleavings are rare. The example instead treats each thread as a list of steps and tries every order in which the steps could run. Two threads of three steps each can interleave in 20 ways, and only the 2 that run one thread entirely before the other give the right answer. With two increments per thread there are 924 orders, and the count ends at 4 in only 6 of them. An atomic fetch-and-add, one indivisible instruction, gives 4 in every order. The last part of the example checks a balance before debiting it, and two withdrawals of 80 from 100 leave the account at -60 in 4 of the 6 orders.

EXAMPLE

aces.py · every interleaving of two threads: lost updates, an atomic fetch-and-add, and check-then-act

```
from collections import Counter

# A thread is a list of steps. Each step reads or writes shared memory or the thread's own
# registers,
# and the scheduler may switch threads between any two steps. We try every possible order.
def interleavings(threads, shared, regs=None, pcs=None, trace=()):
    regs = regs or [dict() for _ in threads]
    pcs = pcs or [0] * len(threads)
    if all(pc == len(t) for pc, t in zip(pcs, threads)):
        yield dict(shared), trace
        return
    for i, t in enumerate(threads):
        if pcs[i] < len(t):
            label, step = t[pcs[i]]
            s2, r2 = dict(shared), [dict(r) for r in regs]
            step(s2, r2[i])
            p2 = list(pcs); p2[i] += 1
            yield from interleavings(threads, s2, r2, p2, trace + (f"T{i + 1} {label}",))

def load(s, r): r["x"] = s["count"] # count += 1 compiles to three
                                instructions
def add(s, r): r["x"] += 1
def store(s, r): s["count"] = r["x"]
increment = [("load", load), ("add", add), ("store", store)]

runs = list(interleavings([increment, increment], {"count": 0}))
outcomes = Counter(final["count"] for final, _ in runs)
print(f"two threads each run count += 1 once: {len(runs)} possible orders")
for value, n in sorted(outcomes.items()):
    print(f"    final count {value}: {n:2} orders")
lost = next(trace for final, trace in runs if final["count"] == 1)
print("    one order that loses an update: " + ", ".join(lost))

runs = list(interleavings([increment * 2, increment * 2], {"count": 0}))
outcomes = Counter(final["count"] for final, _ in runs)
print(f"\ntwo threads each run count += 1 twice: {len(runs)} orders, final count "
      + ", ".join(f"{v} in {n}" for v, n in sorted(outcomes.items())))

def fetch_add(s, r): s["count"] += 1 # one indivisible instruction, like x86
    LOCK XADD
runs = list(interleavings([["fetch_add", fetch_add]] * 2 * 2, {"count": 0}))
finals = {final["count"] for final, _ in runs}
print(f"the same with an atomic fetch-and-add: {len(runs)} orders, final count always
      {finals.pop()}")

# Check-then-act: two withdrawals of 80 from a balance of 100, each checking first
def check(s, r): r["ok"] = s["balance"] >= 80
```

```
def debit(s, r):
    if r["ok"]: s["balance"] -= 80
withdraw = [("check", check), ("debit", debit)]
runs = list(interleavings([withdraw, withdraw], {"balance": 100}))
outcomes = Counter(final["balance"] for final, _ in runs)
print(f"\ntwo withdrawals of 80 from 100: " + ", ".join(f"balance {v} in {n} of {len(runs)}"
    orders"
                                                    for v, n in sorted(outcomes.items()))))
```

OUTPUT

```
two threads each run count += 1 once: 20 possible orders
    final count 1: 18 orders
    final count 2:  2 orders
    one order that loses an update: T1 load, T1 add, T2 load, T1 store, T2 add, T2 store

two threads each run count += 1 twice: 924 orders, final count 2 in 702, 3 in 216, 4 in 6
the same with an atomic fetch-and-add: 6 orders, final count always 4

two withdrawals of 80 from 100: balance -60 in 4 of 6 orders, balance 20 in 2 of 6 orders
```

In a real program the bad orders are rare because the window is tiny: a few nanoseconds between a load and a store, against time slices of milliseconds. A race can survive every test and still fire in production, under load, on more cores, or after an unrelated change shifts the timing. That's why races are found with tools, not luck. ThreadSanitizer, enabled with `-fsanitize=thread` in Clang and GCC, and Go's `-race` flag instrument memory accesses and report any two that aren't ordered, even when the run happened to produce the right answer. Model checkers go further and explore interleavings systematically, as the example does. Rust's `loom` crate does this for real concurrent code, and TLA+ does it for designs.

Check, then act

The withdrawal bug has a name, **time of check to time of use** (TOCTOU), and it's everywhere. "If the file doesn't exist, create it" races with another process creating it in between; `open` with `O_CREAT | O_EXCL` does both atomically and fails if the file exists. "If the user may access this path, open it" races with an attacker replacing the path with a symbolic link; open first and check the descriptor you got. In databases, reading a balance in one statement and updating it in another races with every other client; `UPDATE accounts SET balance = balance - 80 WHERE id = 7 AND balance >= 80` makes the check and the change a single atomic step, and the number of rows it changed says whether it succeeded.

Atomic instructions

Processors provide instructions that read, modify and write a memory location as one indivisible step, even with many cores. **Fetch-and-add** (`LOCK XADD` on x86) increments and returns the old value. **Compare-and-swap** (`LOCK CMPXCHG` on x86, `CAS` on recent ARM) writes a new value only if the location still holds an expected one, and reports whether it did. With compare-and-swap you can make any update atomic: read the value, compute the new one, and retry if someone else changed it in between. Languages expose these as atomic types: `std::atomic` in C++, `AtomicInteger` in Java, `sync/atomic` in Go, `AtomicU64`

in Rust. Uncontended, an atomic operation costs a few tens of cycles. When many cores hammer the same location, its cache line bounces between them and each operation can take a hundred nanoseconds or more.

Memory models

There's a second reason plain variables can't coordinate threads: neither the compiler nor the processor promises to perform memory operations in program order. A compiler may keep a variable in a register and never reload it, or move a store past a load. Processors reorder too. On x86 a store can wait in a **store buffer** while later loads go ahead, so another core sees the store late. ARM, used in Apple silicon and AWS Graviton, allows much more reordering. The **memory model** of a language or processor says which reorderings are allowed, and atomic operations carry ordering guarantees, such as acquire, release or sequentially consistent, that rule them out where it matters. Chapter 8's model checker shows the consequence: Peterson's algorithm, provably correct with plain loads and stores, fails once store buffers are modelled, and a fence fixes it.

Python is not exempt. In the standard build, the GIL means only one thread runs Python code at a time, but `count += 1` is several bytecodes and a thread switch can fall between them, so updates can still be lost. In the free-threaded build of Python 3.14 the threads really run in parallel. The rule is the same in every language: protect shared data with a lock, or use an operation documented to be atomic.

COMMON TRAPS

Testing for races by running the program a few times. Use ThreadSanitizer or a race detector.

A plain boolean flag between threads. The compiler may never reload it, and the hardware may reorder it.

Two atomic steps treated as one, such as `if key not in cache: cache[key] = value`.

Code tested only on x86 that breaks on ARM's weaker ordering.

Hand-written lock-free code without a model checker or a proof.

EXERCISES

EXERCISE 7.1 Counting orders

Why do two threads of 3 steps have exactly 20 interleavings, and two threads of 6 steps 924? Why are only 2 of the 20 correct for `count += 1`?

SOLUTION

An interleaving is fixed by choosing which of the 6 positions belong to thread 1, since each thread's own steps stay in order. That's $C(6, 3) = 20$. In general, threads of m and n steps interleave in $(m + n)! / (m! n!)$ ways, so two threads of 6 steps have $C(12, 6) = 924$, and the count explodes for more threads or longer code, which is why model checkers need clever pruning. For the increment, the result is right only if one thread's store happens before the other's load. If the second load comes before the first store, both threads computed from the same old value. That holds only when one thread runs all three steps before the other starts, which is 2 of the 20 orders.

EXERCISE 7.2 Three fixes for the withdrawal

Fix the check-then-act withdrawal three ways: with a lock, with compare-and-swap, and in a SQL database. Say what each one costs.

SOLUTION

With a lock, take it, check, debit, release; the critical section covers both steps, and the cost is that every withdrawal from any account waits for the lock, unless there's a lock per account. With compare-and-swap, loop: read the balance b ; if $b < 80$, fail; otherwise attempt $\text{CAS}(\text{balance}, b, b - 80)$, and if another thread changed the balance first, the CAS fails and you retry with the new value. No thread ever blocks, but under heavy contention the retries waste work. In SQL, the single statement `UPDATE accounts SET balance = balance - 80 WHERE id = ? AND balance >= 80` lets the database do the check and the change atomically; a row count of 0 means insufficient funds. That's usually the best choice, because the database already serialises updates to the row.

EXERCISE 7.3 The flag that doesn't work

In C, thread A runs `data = compute(); ready = 1;` and thread B runs `while (!ready) {} use(data);`, with both variables plain globals. Name two ways this can fail, and give a correct version.

SOLUTION

First, the compiler sees no write to `ready` inside B's loop, so it may load it once and spin forever. Second, the stores in A or the loads in B may be reordered, by the compiler or by an ARM processor, so B can see `ready == 1` and still read the old `data`. It's also a data race and therefore undefined behaviour. A correct version makes `ready` an

atomic: A does `atomic_store_explicit(&ready, 1, memory_order_release)` after writing `data`, and B loops on `atomic_load_explicit(&ready, memory_order_acquire)`. Release and acquire guarantee that everything A wrote before the store is visible to B after the load. Better still, don't spin: use a condition variable (chapter 9), or `threading.Event` in Python.

CHECKLIST

- ☐ Can you explain why `count += 1` isn't atomic, and define a data race?
- ☐ Can you count interleavings and say why races are rare in tests?
- ☐ Can you recognise and fix check-then-act bugs in memory, files and databases?
- ☐ Can you use fetch-and-add and compare-and-swap, and say what contention costs?
- ☐ Can you explain why plain variables can't coordinate threads under a weak memory model?

GO DEEPER

ARTICLE [Hardware memory models](#) Russ Cox

ARTICLE [An introduction to lock-free programming](#) Jeff Preshing

DOCS [ThreadSanitizer](#) The LLVM project

Locks

3 exercises

A lock lets only one thread at a time run a critical section. Building one correctly is harder than it looks. The obvious version doesn't work. A correct version using only ordinary loads and stores exists, and fails on real processors. The version everyone uses rests on an atomic instruction, plus help from the kernel for waiting. This chapter model-checks four lock algorithms, checks one of them again as real processors run it, explains why they need fences, and then covers spinning against sleeping, the futex, and how to stop a lock from becoming the bottleneck.

What a lock must do

A lock has two operations, acquire and release, and three requirements. **Mutual exclusion:** at most one thread is between acquire and release at any time. **Progress:** if the lock is free and threads want it, one of them gets it; they don't all wait forever. **Bounded waiting:** no thread is passed over indefinitely while others take the lock again and again. Then there's performance, which matters because locks sit on the hottest paths in a program. In code, the critical section should always be written so that the release can't be skipped: `with lock:` in Python, a scoped guard in C++ and Rust, `defer mu.Unlock()` in Go.

Building one from loads and stores

The obvious lock checks a variable and, if it's free, sets it. The example's model checker finds the flaw in four steps: both threads see the lock free before either sets it, and both walk in. Giving each thread its own flag, raised before waiting for the other's to drop, keeps them out of each other's way and introduces a new failure: if both raise their flags at once, each waits for the other forever. That's a deadlock. Gary Peterson's algorithm of 1981 adds one shared variable, `turn`. Each thread raises its flag, then gives the turn to the other, then waits while the other's flag is up and it's the other's turn. If both arrive together, whoever wrote `turn` last waits. The checker confirms it in all 26 reachable states: never two threads in the critical section, and never a state where neither can move.

EXAMPLE

`locks.py` · a model checker that tries every interleaving of four lock algorithms, then Peterson's with store buffers

```
from collections import deque

# A model checker. Two threads run a lock algorithm in a loop: entry code, critical section,
# exit code.
# Each instruction is (description, action); an action takes (i, read, write) and returns the
# next pc,
# or None to spin in place. We explore every reachable state and look for two threads in the
```

```

critical
# section at once, or a state where neither thread can ever move again.
def check(name, program, init, crit, store_buffers=False):
    def successors(state):
        pcs, mem, bufs = state
        for i in (0, 1):
            if store_buffers and bufs[i]:
                # the hardware drains a buffered store
                (var, val), rest = bufs[i][0], bufs[i][1:]
                m = dict(mem); m[var] = val
                drained = tuple(rest if k == i else bufs[k] for k in (0, 1))
                yield (pcs, tuple(sorted(m.items())), drained), f"T{i + 1}'s {var}={val} reaches memory"
            desc, act = program[pcs[i]]
            m, b = dict(mem), [list(x) for x in bufs]
        def read(var):
            # a thread sees its own buffered stores
            for v, val in reversed(b[i]):
                if v == var: return val
            return m[var]
        def write(var, val):
            if store_buffers: b[i].append((var, val))
            else: m[var] = val
        if desc == "fence" and b[i]: continue
        # a fence waits for the buffer to drain
        nxt = act(i, read, write)
        if nxt is None: continue
        # spinning: no change of state
        p = list(pcs); p[i] = nxt % len(program)
        yield (tuple(p), tuple(sorted(m.items())), tuple(tuple(x) for x in b)), f"T{i + 1}{desc}"
    start = ((0, 0), tuple(sorted(init.items())), (((), ()))
    parent, todo, stuck = {start: None}, deque([start]), None
    while todo:
        s = todo.popleft()
        if s[0] == (crit, crit):
            path = []
            while parent[s]: s, step = parent[s]; path.append(step)
            print(f"{name}: BOTH IN THE CRITICAL SECTION after {len(path)} steps "
                  f"({len(parent)} states searched)")
            for step in reversed(path): print(f"    {step}")
            return
        nexts = list(successors(s))
        if not nexts and stuck is None: stuck = s
        for n, step in nexts:
            if n not in parent: parent[n] = (s, step); todo.append(n)
    if stuck:
        path, s = [], stuck
        while parent[s]: s, step = parent[s]; path.append(step)
        print(f"{name}: mutual exclusion holds, but it can DEADLOCK: " + "
              ".join(reversed(path))")
    else:
        print(f"{name}: mutual exclusion holds and no deadlock, in all {len(parent)} reachable states")

other = lambda i: f"flag{1 - i}"
naive = [("sees lock free", lambda i, r, w: 1 if r("lock") == 0 else None),
         ("sets lock", lambda i, r, w: (w("lock", 1), 2)[1]),
         ("in critical section", lambda i, r, w: 3),
         ("clears lock", lambda i, r, w: (w("lock", 0), 0)[1])]
flags = [("raises its flag", lambda i, r, w: (w(f"flag{i}", 1), 1)[1]),
         ("passes the wait (other flag down)", lambda i, r, w: 2 if r(other(i)) == 0 else None),
         ("in critical section", lambda i, r, w: 3),
         ("lowers its flag", lambda i, r, w: (w(f"flag{i}", 0), 0)[1])]
peterson = [("raises its flag", lambda i, r, w: (w(f"flag{i}", 1), 1)[1]),

```

```

        ("gives the turn away", lambda i, r, w: (w("turn", 1 - i), 2)[1]),
        ("passes the wait (other flag down, or its turn)",
         lambda i, r, w: 3 if r(other(i)) == 0 or r("turn") == i else None),
        ("in critical section", lambda i, r, w: 4),
        ("lowers its flag", lambda i, r, w: (w(f"flag{i}", 0), 0)[1]))
def test_and_set(i, r, w):
    old = r("lock"); w("lock", 1)
    return 1 if old == 0 else None
tas = [("test-and-set", test_and_set),
       ("in critical section", lambda i, r, w: 2),
       ("clears lock", lambda i, r, w: (w("lock", 0), 0)[1])]
fenced = peterson[:2] + [("fence", lambda i, r, w: 3)] + peterson[2:]

check("check-then-set lock", naive, {"lock": 0}, crit=2)
check("two flags", flags, {"flag0": 0, "flag1": 0}, crit=2)
check("Peterson's algorithm", peterson, {"flag0": 0, "flag1": 0, "turn": 0}, crit=3)
check("atomic test-and-set", tas, {"lock": 0}, crit=1)
print()
both = {"flag0": 0, "flag1": 0, "turn": 0}
check("Peterson with store buffers", peterson, both, crit=3, store_buffers=True)
check("Peterson with buffers and a fence", fenced, both, crit=4, store_buffers=True)

```

OUTPUT

```

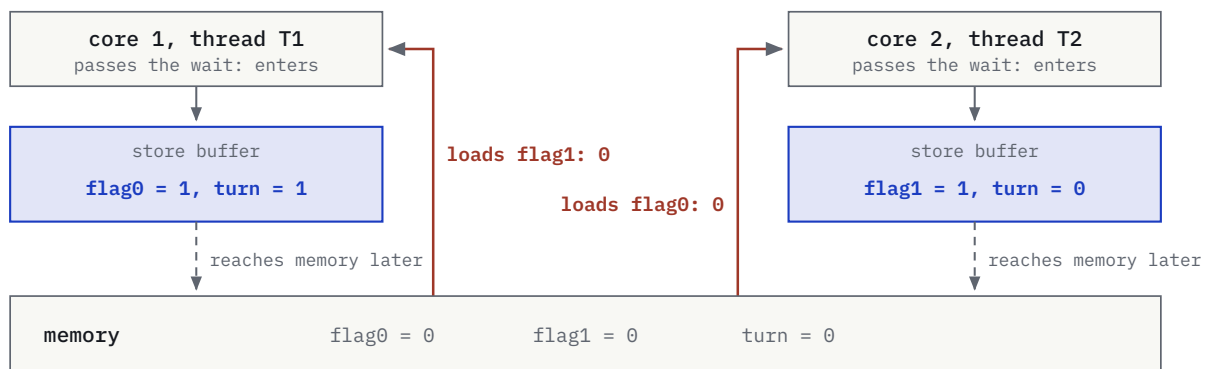
check-then-set lock: BOTH IN THE CRITICAL SECTION after 4 steps (14 states searched)
  T1 sees lock free
  T2 sees lock free
  T1 sets lock
  T2 sets lock
two flags: mutual exclusion holds, but it can DEADLOCK: T1 raises its flag; T2 raises its flag
Peterson's algorithm: mutual exclusion holds and no deadlock, in all 26 reachable states
atomic test-and-set: mutual exclusion holds and no deadlock, in all 5 reachable states

Peterson with store buffers: BOTH IN THE CRITICAL SECTION after 6 steps (97 states searched)
  T1 raises its flag
  T1 gives the turn away
  T1 passes the wait (other flag down, or its turn)
  T2 raises its flag
  T2 gives the turn away
  T2 passes the wait (other flag down, or its turn)
Peterson with buffers and a fence: mutual exclusion holds and no deadlock, in all 53 reachable
states

```

Real hardware breaks it

Peterson's proof assumes that a store is visible to the other thread as soon as it happens. On a real multicore processor it isn't. Each core puts its stores in a **store buffer** and carries on, and the stores reach the shared cache, and so the other core, a little later (chapter 7). The example models that, and the checker finds a run in six steps with no store reaching memory at all. Each thread raises its flag and gives the turn away into its own buffer, then reads the other's flag from memory, sees 0, and enters.



How store buffers break Peterson's algorithm. Each core reads its own buffered stores but the other core's old values, so each concludes the other isn't interested.

The fix is a **fence**, an instruction that makes a core wait until its buffered stores are visible before it performs later loads: **MFENCE** on x86, **DMB** on ARM. With a fence between giving the turn away and reading the other's flag, the checker again finds no bad state. In practice nobody writes Peterson's algorithm; it handles only two threads, and the atomic instructions of chapter 7 make locks much simpler. The lesson is the method: reasoning about concurrent code means reasoning about every interleaving and about the memory model, and a model checker is the only reliable way to do either for anything beyond a few lines.

Test-and-set and spinning

An atomic **test-and-set** writes 1 to the lock and returns the old value in one indivisible step, so exactly one thread sees the old 0 and gets the lock. The checker confirms it in 5 states. A lock that loops on test-and-set until it succeeds is a **spinlock**. Two refinements are standard. **Test and test-and-set** spins on an ordinary read and attempts the atomic only when the lock looks free, because every atomic write steals the cache line from the other cores, while plain reads let every waiting core spin on its own cached copy. And a **ticket lock** gives each arriving thread a number from a fetch-and-add and serves them in order, which guarantees bounded waiting. Spinlocks make sense when critical sections are a few hundred nanoseconds and the lock holder can't be descheduled, which is why kernels use them with preemption disabled. In user space, the holder can be preempted at any moment, and every spinner then burns its entire time slice waiting for a thread that isn't running.

Sleeping instead of spinning: the futex

A lock that puts waiters to sleep needs the kernel, but going to the kernel for every acquire would cost a system call each time (chapter 2). Linux's **futex**, a "fast user-space mutex", added in 2002, avoids that. The lock is an ordinary integer in the process's memory. An uncontended acquire or release is a single atomic instruction in user space, about 20 ns, with no system call. Only when a thread finds the lock taken does it call `futex(FUTEX_WAIT)` to sleep until the value changes, and the releasing thread calls

`futex(FUTEX_WAKE)` only if someone is waiting. glibc's `pthread_mutex`, Go's `sync.Mutex`, Rust's standard mutex and Python's `threading.Lock` all work this way on Linux; macOS has an equivalent primitive in `os_unfair_lock`. Most of them are **adaptive**: they spin briefly in case the holder is about to release, then sleep.

Locks at scale

A lock turns its critical section into serial code, and Amdahl's law (chapter 15 of *The Engineer's Mill*) then caps the speed-up. If 10% of the work happens under one lock, no number of cores makes the program more than 10 times faster, and contention makes it worse: the lock's cache line bounces between cores and waiters pile up in the kernel. The remedies are to hold locks for less time, never across I/O or calls into unknown code; to split one lock into many, such as one per hash bucket or per account, which is lock striping; to give each thread its own data and combine it only when needed, as per-thread counters do; and to choose structures built for concurrent reading, such as read-copy-update in the Linux kernel. Reader-writer locks look like an easy win and often aren't, because every reader still writes to the lock's shared counter. Finer locking also creates more ways to deadlock, the subject of chapter 10.

COMMON TRAPS

A release skipped on an error path. Use `with`, a scoped guard or `defer`.

Holding a lock during I/O or a callback, which turns a nanosecond section into a millisecond one.

Spinlocks in user-space code, where the holder can be preempted.

Assuming a reader-writer lock is faster without measuring it.

Double-checked locking with plain variables, which the memory model doesn't make safe.

EXERCISES

EXERCISE 8.1 What the turn variable buys

Explain the two-flags deadlock the checker found, and how Peterson's `turn` variable prevents it without letting both threads in.

SOLUTION

With flags alone, each thread raises its own flag and then waits for the other's to drop. If both raise their flags before either checks, each sees the other's flag up and waits, and neither will ever lower its flag,

because it's waiting: a deadlock. Peterson's algorithm lets a thread in if the other's flag is down or it's this thread's turn, and each thread gives the turn away before waiting. If both arrive together, both flags are up, and `turn` holds whichever value was written last. The thread that wrote last gave priority to the other, so it waits and the other enters. The value of `turn` breaks the tie exactly once, so one thread always goes ahead, which is progress, and the two can never both pass the wait at once, which is mutual exclusion.

EXERCISE 8.2 Spin or sleep?

A critical section takes 50 ns. Sleeping on a futex and being woken costs about 5 μ s in total. When should a waiting thread spin, and when should it sleep? What changes if the lock's holder can be preempted?

SOLUTION

If the expected wait is shorter than the cost of sleeping, spinning wins. With 50 ns sections a waiter is usually let in within a few hundred nanoseconds, so a short spin is far cheaper than 5 μ s of sleeping and waking. But if the holder is preempted in the middle of its section, it won't run again for a whole time slice, milliseconds, and a spinner wastes all of that. That's why user-space mutexes are adaptive: spin for roughly the cost of a sleep, a few microseconds, and then sleep. The worst case is then about twice the optimal cost, whatever the holder does.

EXERCISE 8.3 The hot counter

A service has 32 threads that increment one shared statistics counter, protected by a mutex, millions of times a second. Adding cores makes it slower. Why, and what are two better designs?

SOLUTION

Every increment needs the same lock and writes the same cache line, so the increments run one at a time, and each moves the line from one core's cache to another's, which takes far longer than the increment itself. More cores mean more contention, so throughput falls. Replacing the mutex with an atomic fetch-and-add helps but keeps the bouncing cache line. Better: give each thread its own counter, padded to a separate cache line so neighbours don't share one (false sharing, from chapter 11 of *The Engineer's Mill*), and add them up only when someone reads the statistic. Or have each thread count locally and add

its total to the shared counter every thousand events. Both make the hot path free of sharing, at the cost of a read that's slightly stale or slightly slower.

CHECKLIST

- ☐ Can you state the three requirements of a lock?
- ☐ Can you explain why check-then-set fails and why Peterson's algorithm works?
- ☐ Can you explain why store buffers break Peterson's algorithm and how a fence fixes it?
- ☐ Can you describe spinlocks, ticket locks and futexes, and when each fits?
- ☐ Can you name three ways to reduce lock contention?

GO DEEPER

BOOK Operating Systems: Three Easy Pieces, chapter 28: locks Arpaci-Dusseau & Arpaci-Dusseau

PAPER Futexes are tricky Ulrich Drepper

BOOK Is Parallel Programming Hard, And, If So, What Can You Do About It? Paul McKenney

Condition variables and semaphores

3 exercises

A lock stops threads interfering with each other, but it can't make a thread wait until something becomes true: until a queue has an item, a buffer has room, or a count reaches zero. Condition variables and semaphores do that. This chapter model-checks the bounded buffer, the classic producer-and-consumer problem. The checker finds both classic bugs, an `if` where a `while` belongs and one condition variable where two are needed, before the correct version runs with real threads. It ends with semaphores and the higher-level tools built on both.

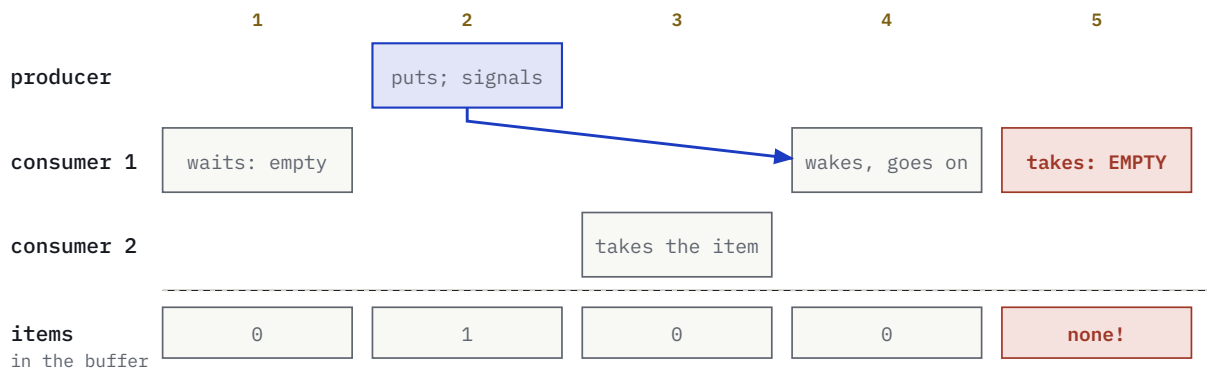
Waiting for a condition

A consumer that finds a queue empty has to wait. Spinning on the queue wastes a core; sleeping for 10 ms and checking again adds up to 10 ms of delay and still wastes wake-ups. A **condition variable** lets a thread sleep until another thread says the state may have changed. It's always used with a lock and a condition on shared state. A thread holding the lock checks the condition; if it's false, it calls `wait`, which atomically releases the lock and puts the thread to sleep, and reacquires the lock before returning. Another thread that changes the state, under the same lock, calls `signal` (`notify` in Python and Java) to wake one waiter, or `broadcast` (`notify_all`) to wake them all.

The atomic release-and-sleep is the whole point. If a thread checked the condition, released the lock and then went to sleep in three separate steps, another thread could change the state and signal in between, the signal would find no one waiting and be lost, and the first thread would sleep forever (exercise 9.1).

Mesa semantics, and why the check goes in a loop

When a waiting thread is signalled, it doesn't run immediately. It becomes ready, and it must win the lock back like any other thread. In between, another thread can take the lock and change the state again. This is called **Mesa semantics**, after the language in which Butler Lampson and David Redell described it in 1980, and it's what POSIX threads, Java, Python and Go all provide. The alternative, Hoare semantics, hands the lock straight to the woken thread, which is easier to reason about and slower to implement, and almost no real system uses it. Under Mesa semantics a signal is only a hint that the condition may now be true. POSIX even allows **spurious wake-ups**, from `wait` returning with no signal at all. So the condition must be checked again after every wake-up, in a loop: `while not condition: cv.wait()`.



The `if` bug, as the model checker found it. The signal at step 2 was true when sent, but consumer 2 got the lock first, and by step 4 the condition was false again.

Checked, then run

The example models one producer that puts two items and two consumers that take one each, through a buffer that holds one item, with a mutex and condition variables that follow Mesa semantics. With `if` and one condition variable, the checker finds the trace in the figure in 14 steps. With `while` and one condition variable it finds a different bug: a deadlock with every thread asleep. With `while` and two condition variables it checks all 139 reachable states and finds nothing. The correct version then runs for real with Python's `threading.Condition`, 30,000 items through a four-slot buffer to three consumers, and every item arrives exactly once.

EXAMPLE

condvars.py · a model checker for a bounded buffer with Mesa-style condition variables, then the correct version with real threads

```
import textwrap, threading
from collections import deque

# A bounded buffer holding one item: one producer puts 2 items, two consumers take one each.
# The checker models a mutex and condition variables with the semantics real systems use
# (Mesa semantics): signal() only wakes a waiter, which must then win the lock again.
def program(role, check, cvs):
    full, empty = ("cv", "cv") if cvs == 1 else ("fill", "space")
    if role == "producer":
        one = [("lock",), (check, lambda n: n == 1, empty), ("put",), ("signal", full),
              ("unlock",)]
        return one * 2
    return [("lock",), (check, lambda n: n == 0, full), ("get",), ("signal", empty),
          ("unlock",)]

def explore(check, cvs):
    label = f"{check:5} with {cvs} condition variable{'s' if cvs > 1 else ''}"
    def show(steps): print(textwrap.indent(textwrap.fill("".join(steps), 92), " "))
    progs = [program(role, check, cvs) for role in ("producer", "consumer", "consumer")]
    names = ["P", "C1", "C2"]
    # state: (pcs, status per thread, item count, lock holder, waiters per condition variable)
    start = ((0, 0, 0), ("run",) * 3, 0, None, ())
    parent, todo = {start: None}, deque([start])
```

```

def trace(s):
    path = []
    while parent[s]: s, step = parent[s]; path.append(step)
    return list(reversed(path))
while todo:
    s = todo.popleft()
    pcs, status, count, holder, waiters = s
    moves = []
    for i, prog in enumerate(progs):
        if pcs[i] == len(prog): continue
        op, st = prog[pcs[i]], status[i]
        def go(pc=pcs[i] + 1, stat="run", cnt=count, hold=holder, wait=waiters, note=""):
            p = list(pcs); p[i] = pc; t = list(status); t[i] = stat
            moves.append(((tuple(p), tuple(t), cnt, hold, wait), f"{names[i]} {note or
                op[0]}"))
            if st.startswith("waiting"): continue # asleep until
                signalled
            if st == "woken": # must win the
                lock back
                if holder is None:
                    again = pcs[i] if op[0] == "while" else pcs[i] + 1 # while: re-
                        check; if: go on
                    go(pc=again, hold=i, note="wakes and takes the lock")
                continue
            if op[0] == "lock" and holder is None: go(hold=i, note="takes the lock")
            elif op[0] == "unlock": go(hold=None, note="releases the lock")
            elif op[0] in ("if", "while"):
                if op[1](count): go(pc=pcs[i], stat="waiting", hold=None,
                    wait=tuple(sorted(waiters + ((op[2], i),))), note=f"waits
                        on {op[2]}")
                else: go(note="can go ahead")
            elif op[0] == "put": go(cnt=count + 1, note="puts an item")
            elif op[0] == "get":
                if count == 0:
                    print(f"{label}: BUG, a consumer took from an empty buffer")
                    show(trace(s) + [f"{names[i]} takes from the empty buffer"])
                    return
                go(cnt=count - 1, note="takes the item")
            elif op[0] == "signal":
                sleepers = [w for w in waiters if w[0] == op[1]]
                if not sleepers: go(note=f"signals {op[1]} (no one waiting)")
                for w in sleepers: # the system
                    picks any waiter
                    t = list(status); t[w[1]] = "woken"
                    p = list(pcs); p[i] += 1
                    moves.append(((tuple(p), tuple(t), count, holder, tuple(x for x in waiters
                        if x != w)),
                        f"{names[i]} signals {op[1]}, waking {names[w[1]]}"))
            if not moves and any(pc < len(pr) for pc, pr in zip(pcs, progs)):
                print(f"{label}: DEADLOCK, everyone left is asleep")
                show(trace(s))
                return
            for n, step in moves:
                if n not in parent: parent[n] = (s, step); todo.append(n)
    print(f"{label}: correct in all {len(parent):,} reachable states")

explore("if", 1)
explore("while", 1)
explore("while", 2)

# The correct version, run for real: Python's threading.Condition has the same Mesa semantics
buf, CAP = deque(), 4
lock = threading.Lock(); has_item = threading.Condition(lock); has_space =
    threading.Condition(lock)
def producer(items):
    for x in items:

```

```

        with lock:
            while len(buf) == CAP: has_space.wait()
            buf.append(x); has_item.notify()
totals = [0, 0, 0]
def consumer(k, n):
    for _ in range(n):
        with lock:
            while not buf: has_item.wait()
            totals[k] += buf.popleft(); has_space.notify()
threads = [threading.Thread(target=producer, args=(range(1, 30_001),))] + \
    [threading.Thread(target=consumer, args=(k, 10_000)) for k in range(3)]
for t in threads: t.start()
for t in threads: t.join()
print(f"\nreal threads, 3 consumers sharing 30,000 items through a 4-slot buffer: sum
      {sum(totals):,} "
      f"(expected {30_000 * 30_001 // 2:,:})")

```

OUTPUT

```

if    with 1 condition variable: BUG, a consumer took from an empty buffer
      C1 takes the lock; C1 waits on cv; P takes the lock; P can go ahead; P puts an item; P
      signals cv, waking C1; P releases the lock; C2 takes the lock; C2 can go ahead; C2 takes the
      item; C2 signals cv (no one waiting); C2 releases the lock; C1 wakes and takes the lock; C1
      takes from the empty buffer
while with 1 condition variable: DEADLOCK, everyone left is asleep
      C1 takes the lock; C1 waits on cv; C2 takes the lock; C2 waits on cv; P takes the lock; P
      can go ahead; P puts an item; P signals cv, waking C1; P releases the lock; P takes the
      lock; P waits on cv; C1 wakes and takes the lock; C1 can go ahead; C1 takes the item; C1
      signals cv, waking C2; C1 releases the lock; C2 wakes and takes the lock; C2 waits on cv
while with 2 condition variables: correct in all 139 reachable states

real threads, 3 consumers sharing 30,000 items through a 4-slot buffer: sum 450,015,000
      (expected 450,015,000)

```

One condition variable or two

The deadlock is subtler than the `if` bug. Producers wait for space and consumers wait for items, and with only one condition variable they all wait on the same one. In the trace, both consumers wait, the producer puts an item, signals, and then waits itself because the buffer is full. Consumer 1 takes the item and signals, intending to wake the producer, but the signal wakes consumer 2 instead, which finds the buffer empty and goes back to sleep. Now all three are asleep and nobody will ever signal. A signal must wake a thread that can act on it, so threads waiting for different conditions need different condition variables: one for "there's space", one for "there's an item". `notify_all` also fixes it, because it wakes everyone, but it does so at the cost of waking every waiter for every change, the **thundering herd**.

Semaphores

Edsger Dijkstra introduced the **semaphore** in 1965: an integer with two atomic operations, `P` (wait, or down), which decrements it and blocks while it would go negative, and `V` (signal, or up), which increments it and wakes a waiter. A semaphore that starts at 1 is a lock. One that starts at n admits n threads at a time, which is a natural limit on a pool of resources, such as ten database connections or four concurrent downloads. Unlike a condition variable, a semaphore remembers: a `V` with no one waiting isn't lost but raises

the count for the next `P`. A bounded buffer of n slots needs three semaphores: `slots` starting at n , `items` starting at 0, and a `mutex` starting at 1 for the buffer itself. The order of the waits matters, as exercise 9.2 shows.

In practice, write condition-variable code rarely, and reach for the tools built from these primitives: blocking queues such as Python's `queue.Queue` and Java's `BlockingQueue`; channels (chapter 11); barriers, which make a group of threads wait until all of them arrive, as chapter 10's example does; countdown latches; and futures. The thread pool in this stage's deliverable is a blocking queue of tasks and a fixed set of threads taking from it.

COMMON TRAPS

`if` instead of `while` around `wait`.

Changing the state or signalling without the lock, which can lose a wake-up.

One condition variable for two conditions with `notify`, which can wake the wrong thread.

`notify_all` everywhere, waking every waiter for every change.

Taking the buffer's mutex before waiting on a semaphore, which deadlocks.

A hand-written queue where the standard library already has a correct one.

EXERCISES

EXERCISE 9.1 The lost wake-up

Thread A runs `if not ready: cv.wait()` without holding the lock while it checks `ready`, and thread B runs `ready = True; cv.notify()`. Describe an interleaving in which A sleeps forever, and explain how holding the lock prevents it.

SOLUTION

A checks `ready` and finds it false. Before A has started waiting, B sets `ready` and calls `notify`, which finds no waiter and does nothing. A then calls `wait` and sleeps, and the only notification it would ever get has already happened. If both threads hold the lock, B can't change `ready` between A's check and A's sleep, because A holds the lock throughout, and `wait` releases it only at the moment A is safely asleep. B's update and notification then come either before A's check, so A sees

`ready` and doesn't wait, or after A is asleep, so the notification wakes it. Python actually requires it: calling `wait` or `notify` without holding the condition's lock raises `RuntimeError`.

EXERCISE 9.2 A bounded buffer from semaphores

Write the producer and consumer for an n -slot buffer using semaphores `slots` (n), `items` (0) and `mutex` (1). Then show how putting the producer's `P(mutex)` before `P(slots)` can deadlock.

SOLUTION

Producer: `P(slots); P(mutex); put(x); V(mutex); V(items)`.
Consumer: `P(items); P(mutex); x = take(); V(mutex); V(slots)`. A producer first reserves a free slot, possibly waiting, then takes the mutex only for the moment it touches the buffer. If the producer takes the mutex first, then when the buffer is full it holds the mutex while it waits on `slots`. A consumer that could free a slot needs the mutex to take an item, so it waits for the producer, which waits for the consumer: a deadlock. The rule: never block waiting for a resource while holding a lock that the thread that would free it needs.

EXERCISE 9.3 Limiting concurrency

A service must never run more than 10 database queries at once. Show how a semaphore enforces that, what happens if a query raises an exception, and how to make the code robust.

SOLUTION

Create `limit = threading.Semaphore(10)` and wrap each query in it: the eleventh caller blocks in `acquire` until one of the ten releases. If a query raises an exception between `acquire` and `release`, and the code calls them by hand, the release is skipped and the limit is permanently one lower; after ten such failures, every caller blocks forever. Using `with limit:` releases on every exit path, exceptions included. `threading.BoundedSemaphore` additionally raises an error if the code releases more times than it acquired, which catches the opposite bug. In asynchronous code, `asyncio.Semaphore` does the same for coroutines.

- ☐ Can you explain why `wait` must release the lock and sleep in one atomic step?
- ☐ Can you explain Mesa semantics and why the condition is re-checked in a loop?
- ☐ Can you explain when one condition variable isn't enough?
- ☐ Can you build a bounded buffer with semaphores, in the right order?
- ☐ Can you choose a higher-level tool such as a queue, barrier or semaphore for a given problem?

GO DEEPER

BOOK Operating Systems: Three Easy Pieces, chapters 30 and 31: condition variables and semaphores Arpaci-Dusseau & Arpaci-Dusseau

BOOK The Little Book of Semaphores Allen Downey

PAPER Experience with processes and monitors in Mesa Lampson & Redell, 1980

Deadlock, livelock and starvation

3 exercises

A deadlock is a set of threads, each waiting for something another one holds. Nothing moves, nothing fails, no error is logged: the program simply stops. This chapter model-checks the dining philosophers and finds their deadlock in a few dozen states. It sets out the four conditions every deadlock needs, with the fix that breaks each, detects a deadlock after the fact from the graph of who waits for whom, and reproduces one with real locks. It ends with livelock and starvation, the two ways a system can stay busy and still make no progress.

The dining philosophers

Dijkstra's puzzle from 1965: philosophers sit at a round table with one fork between each pair, and each needs both neighbouring forks to eat. If every philosopher picks up the left fork and then the right, there's a moment when all of them hold a left fork and wait for a right one that their neighbour is holding, and they wait forever. The example's checker explores every order in which the philosophers can act and finds that state after 14 states with three philosophers and 82 with five. Then it makes one change: every philosopher picks up the lower-numbered of their two forks first. Now the last philosopher, whose forks are numbered $n - 1$ and 0, reaches for fork 0 first, the circle is broken, and the checker finds no deadlock in any reachable state.

EXAMPLE

deadlock.py · dining philosophers, a wait-for cycle, a real deadlock between two locks, and a livelock broken by random backoff

```
import random, threading
from collections import deque

# 1. Dining philosophers: each picks up one fork, then the other, eats, and puts both down.
def dine(n, ordered):
    def forks(p):
        n)
        a, b = p, (p + 1) % n
        return (min(a, b), max(a, b)) if ordered else (a, b) # ordered: always the lower number
        first
    start = ((0,) * n, (None,) * n) # (step of each philosopher, fork
    owners)
    seen, todo = {start}, deque([start])
    while todo:
        steps, owner = todo.popleft()
        moves = []
        for p in range(n):
            first, second = forks(p)
            o = list(owner); s = list(steps)
            if steps[p] == 0 and owner[first] is None: o[first] = p; s[p] = 1 # pick up
            first fork
```

```

        elif steps[p] == 1 and owner[second] is None: o[second] = p; s[p] = 2      # pick up
            second
        elif steps[p] == 2: o[first] = o[second] = None; s[p] = 0                # eat, put
            both down
        else: continue
        moves.append((tuple(s), tuple(o)))
    if not moves:
        return f"DEADLOCK, found after {len(seen)} states: everyone holds one fork, waiting
            for another"
    for m in moves:
        if m not in seen: seen.add(m); todo.append(m)
    return f"no deadlock in any of the {len(seen)} reachable states"
for n in (3, 5):
    print(f"{n} philosophers, left fork first:  {dine(n, ordered=False)}")
    print(f"{n} philosophers, lower fork first:  {dine(n, ordered=True)}")

# 2. Detecting a deadlock after the fact: a cycle in the graph of who waits for whom
holds = {"L1": "T1", "L2": "T2", "L3": "T3", "L4": "T4"}
waits = {"T1": "L2", "T2": "L3", "T3": "L1", "T4": "L1"}
def find_cycle(start):
    path, t = [], start
    while t in waits and t not in path:
        path.append(t); t = holds[waits[t]]
    return path[path.index(t):] if t in path else None
cycle = next(c for c in (find_cycle(t) for t in waits) if c)
print("\nwait-for cycle: " + " -> ".join(f"{t} waits for {waits[t]} (held by
    {holds[waits[t]]}")
        for t in cycle))
print("T4 is stuck too, but only because it waits on the cycle; breaking the cycle frees it")

# 3. Real threads, real locks, opposite order. A barrier makes both take their first lock
    first.
A, B = threading.Lock(), threading.Lock()
both_hold_one, both_tried = threading.Barrier(2), threading.Barrier(2)
report = {}
def worker(name, first, second, align):
    with first:
        if align: both_hold_one.wait()                # make the bad timing certain
        got = second.acquire(timeout=0.5)             # give up after half a second
        report[name] = "got both locks" if got else "timed out waiting: deadlock"
        if got: second.release()
        if align: both_tried.wait()                   # neither lets go before both have
            tried
for label, orders, align in (("opposite order", ((A, B), (B, A)), True),
    ("same order", ((A, B), (A, B)), False)):
    ts = [threading.Thread(target=worker, args=(f"T{k} + 1", *orders[k], align)) for k in
        range(2)]
    for t in ts: t.start()
    for t in ts: t.join()
    print(f"\n{label}: " + " "; ".join(f"{k} {v}" for k, v in sorted(report.items()))))

# 4. Livelock: on a conflict each thread politely lets go and retries. In lockstep, forever.
def polite(backoff, rounds=50):
    random.seed(10)
    delay = [0, 0]
    for r in range(rounds):
        trying = [t for t in (0, 1) if delay[t] == 0]
        if len(trying) == 1: return f"thread {trying[0] + 1} got both locks in round {r + 1}"
        for t in (0, 1): delay[t] = max(0, delay[t] - 1)
        if len(trying) == 2 and backoff:                # both collided: each waits a
            random time
            for t in trying: delay[t] = random.randint(0, 3)
    return f"no one finished in {rounds} rounds: both busy, no progress"
print(f"\nlivelock, retrying at once:      {polite(False)}")
print(f"livelock, with random backoff:     {polite(True)}")

```

OUTPUT

```
3 philosophers, left fork first: DEADLOCK, found after 14 states: everyone holds one fork,
    waiting for another
3 philosophers, lower fork first: no deadlock in any of the 12 reachable states
5 philosophers, left fork first: DEADLOCK, found after 82 states: everyone holds one fork,
    waiting for another
5 philosophers, lower fork first: no deadlock in any of the 70 reachable states

wait-for cycle: T1 waits for L2 (held by T2) -> T2 waits for L3 (held by T3) -> T3 waits for L1
    (held by T1)
T4 is stuck too, but only because it waits on the cycle; breaking the cycle frees it

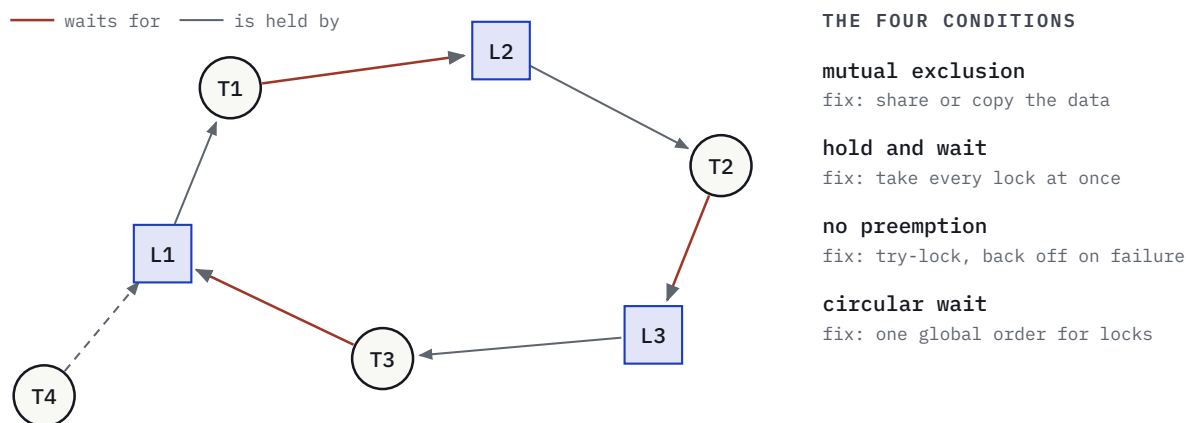
opposite order: T1 timed out waiting: deadlock; T2 timed out waiting: deadlock

same order: T1 got both locks; T2 got both locks

livelock, retrying at once: no one finished in 50 rounds: both busy, no progress
livelock, with random backoff: thread 1 got both locks in round 2
```

The four conditions

Edward Coffman and his colleagues showed in 1971 that a deadlock needs four conditions at once. **Mutual exclusion**: a resource can be held by only one thread. **Hold and wait**: a thread holds one resource while waiting for another. **No preemption**: a resource can't be taken away from the thread holding it. **Circular wait**: a cycle of threads, each waiting for a resource the next one holds. Remove any one and deadlock is impossible.



The example's wait-for graph. The cycle T1, T2, T3 is the deadlock; T4 is stuck only because it waits for a lock inside the cycle, and goes free when the cycle is broken.

In practice, circular wait is the condition to break. Give every lock a place in one global order and always acquire locks in that order, as the philosophers did with fork numbers. When the locks belong to objects, order them by something stable such as an ID: a transfer between two accounts locks the account with the lower ID first, whichever direction the money flows. Hold and wait can be broken by acquiring every lock a task needs in one step, which is hard when the set isn't known in advance. No preemption can be broken by trying for a lock with a timeout, or with a non-blocking try-lock, and releasing everything held on

failure before trying again. Dijkstra's banker's algorithm avoids deadlock by refusing any request that could lead to one, but it needs to know every thread's maximum demand in advance, so general-purpose systems don't use it.

Finding a deadlock

The alternative to preventing deadlocks is detecting them. Record which thread holds each lock and which lock each thread waits for, and look for a cycle, as the example's `find_cycle` does. Databases work this way, because transactions lock rows in whatever order the queries touch them. PostgreSQL waits a second, checks for a cycle, and aborts one transaction with "deadlock detected" so the others can continue. Runtimes help with diagnosis too: `jstack` reports a Java-level deadlock in a thread dump, Go's runtime stops with "all goroutines are asleep - deadlock!", and `py-spy dump` shows every Python thread's stack. The Linux kernel goes further with `lockdep`, which records the order in which locks are taken while the kernel runs and reports an order that *could* deadlock the first time it occurs, before any deadlock has happened.

The example also builds a real deadlock with two Python locks taken in opposite orders. A barrier makes sure each thread holds its first lock before either asks for its second, so the bad timing is certain instead of rare. Each thread asks for its second lock with a timeout and gives up after half a second, a simple form of detection. With both threads taking the locks in the same order, both finish.

Livelock and starvation

In a **livelock** the threads aren't blocked. They keep reacting to each other, and nothing gets done. Two threads that each take one lock, find the other taken, politely release theirs and retry at once can repeat that dance indefinitely if they stay in step, which the example's lockstep simulation does for 50 rounds. The fix is to break the symmetry with randomness: on a collision, each waits a random time before retrying, and doubles the range after each further collision. That's **exponential backoff with jitter**, the same rule Ethernet used for colliding transmissions and that well-behaved clients use for retrying failed requests. With it, one thread wins in the second round.

In **starvation**, the system makes progress but one thread never does, because the rules always favour someone else: readers that keep a reader-writer lock busy so a writer never gets in, an unfair lock that the same thread keeps winning, a low-priority job behind an endless stream of high-priority ones. The fixes are fairness and aging, such as serving waiters in order or boosting a waiter's priority as it waits, as the multi-level feedback queue's boost did in chapter 5.

COMMON TRAPS

Lock order that depends on the arguments, such as `transfer(a, b)` locking `a` first.

Calling unknown code while holding a lock: callbacks, logging handlers, plugins

Catching unknown code while holding a lock: callbacks, logging handlers, plugins.

Waiting with no timeout on locks and, especially, on network calls.

Retrying at once, or at fixed intervals, so colliding clients collide again. Add jitter.

Deadlocks across services, where each waits on the other's request; no local detector sees them.

EXERCISES

EXERCISE 10.1 The transfer deadlock

`transfer(a, b, amount)` locks account `a`, then account `b`, and moves the money. Show how two concurrent transfers deadlock, and write a version that can't.

SOLUTION

`transfer(x, y)` and `transfer(y, x)` run at once: the first locks `x`, the second locks `y`, and each then waits for the lock the other holds. That's circular wait with two threads. Fix it by always locking in a global order:

```
def transfer(a, b, amount):
    first, second = (a, b) if a.id < b.id else (b, a)
    with first.lock, second.lock:      # lower ID first, whichever way the money moves
        a.balance -= amount
        b.balance += amount
```

Now both transfers try to lock the lower-ID account first, so one of them gets both locks and the other waits for it, and no cycle can form. A transfer from an account to itself would try to take the same lock twice, so either reject it or use a reentrant lock.

EXERCISE 10.2 Which condition does it break?

For each design, say which of the four conditions it removes: one global lock around all shared state; a try-lock that releases everything and retries with backoff on failure; taking all the locks a task needs in sorted order before starting; giving each piece of data a single owner thread that other threads send requests to.

SOLUTION

One global lock: with only one lock there's never a second resource to wait for, so hold and wait, and with it circular wait, can't arise; the price is that nothing runs in parallel. Try-lock and back off: a thread that can't get a lock gives up the ones it has, which is preemption done voluntarily, so it breaks no preemption, at the risk of livelock without jitter. Sorted acquisition: a single global order makes circular wait impossible. A single owner per piece of data, the actor model of chapter 11, removes the locks, and mutual exclusion is guaranteed by the owner handling one request at a time. Actors can still deadlock if two of them each wait for a reply from the other, which is circular wait again at a higher level.

EXERCISE 10.3 Why retries need jitter

A thousand clients lose their connection to a server at the same moment and all retry after exactly one second, then two, then four. What does the server see, and how does jitter change it?

SOLUTION

Doubling the delay isn't enough when everyone doubles in step. The server sees all thousand clients reconnect in the same instant at 1, 3 and 7 seconds, a thundering herd at each of which it may be overloaded again, so the retries keep failing together. With jitter, each client waits a random time between zero and the current delay ("full jitter"), so the thousand retries spread evenly across the whole interval: the first retry wave arrives at about a thousand per second instead of a thousand at once, and later waves spread further. The total work is the same; the peak is a small fraction of it.

CHECKLIST

- ☐ Can you state the four conditions for deadlock and a way to break each?
- ☐ Can you impose a lock order on objects, such as accounts, that are locked in pairs?
- ☐ Can you detect a deadlock from a wait-for graph, and find one in a thread dump?
- ☐ Can you explain livelock and why random backoff fixes it?
- ☐ Can you recognise starvation and name a fairness fix?

GO DEEPER

BOOK	Operating Systems: Three Easy Pieces, chapter 32: common concurrency problems Arpaci-Dusseau & Arpaci-Dusseau
DOCS	Runtime locking correctness validator (lockdep) The Linux kernel documentation
ARTICLE	Exponential backoff and jitter Marc Brooker

Event loops, actors and channels

3 exercises

Threads are one way to do many things at once. Another is to do them all on one thread that never waits. An event loop asks the kernel what's ready and runs only code that can make progress. This chapter builds an event loop from Python generators in a dozen lines, shows `asyncio` doing the same, and shows what happens when one coroutine blocks. It then turns to two models that avoid shared memory altogether: actors, which own their state and handle one message at a time, and channels, which connect stages that share nothing.

Concurrency without threads

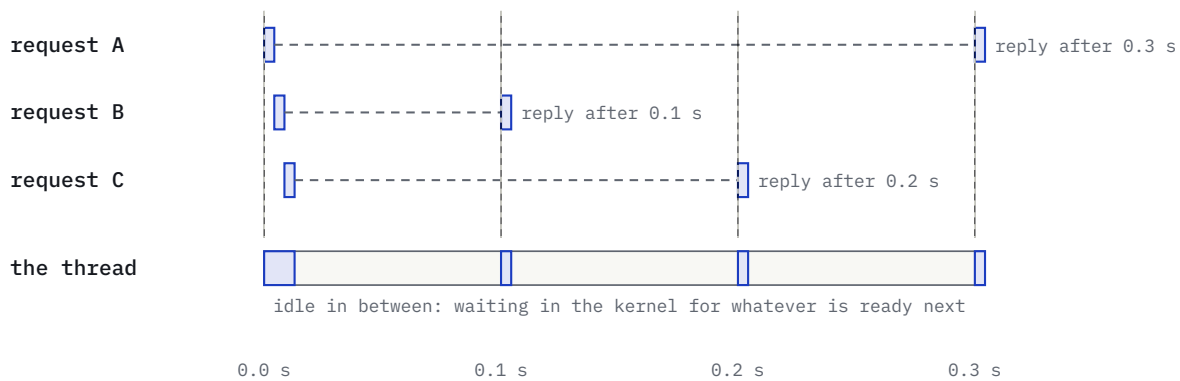
A server handling ten thousand connections spends almost all its time waiting: for a request to arrive, for a database to answer, for a slow client to accept its reply. With a thread per connection, each wait holds a thread, with its stack and its share of context switches (chapter 4). An **event loop** handles all of them on one thread. It keeps a queue of work that's ready to run and a list of timers, asks the kernel which connections have data with a single call such as `epoll_wait` or `kevent` (chapter 18), and runs the code waiting for each. No code ever blocks; code that would wait instead registers interest and returns to the loop. `nginx`, `Redis` and `Node.js` are built this way.

Early event-driven code was written as callbacks, functions to run when an event happens, and deeply nested callbacks are hard to read and harder to debug. Promises, or futures, made them composable, and `async/await` made them look like ordinary sequential code: C# added it in 2012, Python in 2015, JavaScript in 2017 and Rust in 2019. An `async` function is a coroutine, and each `await` is a point where it can pause and hand control back to the loop until the thing it's waiting for is ready.

An event loop in a dozen lines

The example's loop is the idea with nothing else in the way. Each task is a Python generator. When it would wait, it yields a request to the loop, here "wake me in 0.3 seconds", and the loop keeps a heap of timers ordered by when they're due. The loop pops the earliest, advances its clock to that time, and resumes that task until it yields again. Three requests that wait 0.3, 0.1 and 0.2 seconds all start at once and finish in order of their waits, and the whole thing takes 0.3 seconds, not 0.6.

ONE THREAD, THREE REQUESTS



The example's three requests on one thread, to scale. The waits overlap and the thread works only when something is ready, so the total is the longest wait, not the sum of them.

A real loop such as `asyncio` adds the other kind of waiting: instead of timers alone, it registers sockets with the kernel and asks which are readable or writable, then resumes the coroutines waiting on them. The example's second part runs a ticker and a worker under `asyncio`. When the worker waits with `await asyncio.sleep`, the ticker keeps ticking in between. When it calls `time.sleep` instead, which blocks the thread, nothing else runs until it returns.

EXAMPLE

eventloops.py · an event loop from generators, asyncio with a blocking call, an actor, and a pipeline of channels

```
import asyncio, heapq, queue, threading, time

# 1. An event loop from scratch. Tasks are generators; each 'yield' hands control back to the
#    loop
#    with a request: ("sleep", seconds). The loop keeps a clock and a heap of timers. No
#    threads.
now = 0.0
def loop(tasks):
    global now
    timers, n = [], 0
    for task in tasks: heapq.heappush(timers, (0.0, (n := n + 1), task))
    while timers:
        now, _, task = heapq.heappop(timers)          # jump the clock to the next thing
                                                       # that's due
        try:
            kind, arg = next(task)                    # run the task until it yields again
            heapq.heappush(timers, (now + arg, (n := n + 1), task))
        except StopIteration:
            pass
    return now
log = []
def fetch(name, seconds):
    log.append(f"  t = {now:.1f} s    {name} sends its request")
    yield ("sleep", seconds)                          # waiting for the network: let others
    log.append(f"  t = {now:.1f} s    {name} gets its reply")
total = loop([fetch("A", 0.3), fetch("B", 0.1), fetch("C", 0.2)])
```

```

print("three requests on one thread, waiting 0.3, 0.1 and 0.2 s for replies:")
print("\n".join(log))
print(f"finished at t = {total:.1f} s: the waits overlap, so the total is the longest wait, not
      their sum")

# 2. asyncio is the same idea. A blocking call inside a coroutine stalls every other task.
async def ticker(events):
    for i in range(3):
        events.append(f"tick {i}"); await asyncio.sleep(0.02)
async def worker(events, blocking):
    events.append("worker starts")
    if blocking: time.sleep(0.1)                # blocks the whole thread, and so the
        loop
    else: await asyncio.sleep(0.1)             # yields to the loop while it waits
    events.append("worker done")
async def main(blocking):
    events = []
    await asyncio.gather(ticker(events), worker(events, blocking))
    return "\n".join(events)
print(f"\nwith await asyncio.sleep: {asyncio.run(main(False))}")
print(f"with time.sleep: {asyncio.run(main(True))}")

# 3. An actor owns its state and handles one message at a time from its mailbox: no locks
needed
def account(mailbox, replies):
    balance = 0
    while (msg := mailbox.get()) != "stop":
        kind, amount = msg
        if kind == "deposit": balance += amount
        elif kind == "withdraw" and balance >= amount: balance -= amount
    replies.put(balance)
mailbox, replies = queue.Queue(), queue.Queue()
actor = threading.Thread(target=account, args=(mailbox, replies)); actor.start()
senders = [threading.Thread(target=lambda: [mailbox.put(("deposit", 1)) for _ in
range(10_000)])
            for _ in range(4)]
for s in senders: s.start()
for s in senders: s.join()
mailbox.put("stop"); actor.join()
print(f"\nfour threads sent 10,000 deposits each to one actor: balance {replies.get():,}")

# 4. Channels connect stages that share nothing: numbers -> squares -> a running total
async def pipeline():
    squares_in, totals_in = asyncio.Queue(maxsize=1), asyncio.Queue(maxsize=1)    # maxsize=1: a
    hand-off
    async def numbers():
        for x in range(1, 6): await squares_in.put(x)
        await squares_in.put(None)
    async def squarer():
        while (x := await squares_in.get()) is not None: await totals_in.put(x * x)
        await totals_in.put(None)
    async def summer():
        total, seen = 0, []
        while (x := await totals_in.get()) is not None: total += x; seen.append(total)
        return seen
    _, _, seen = await asyncio.gather(numbers(), squarer(), summer())
    return seen
print(f"running totals of squares through two channels: {asyncio.run(pipeline())}")

```

OUTPUT

```

three requests on one thread, waiting 0.3, 0.1 and 0.2 s for replies:
t = 0.0 s   A sends its request
t = 0.0 s   B sends its request
t = 0.0 s   C sends its request
t = 0.1 s   B gets its reply

```

```
t = 0.2 s    C gets its reply
t = 0.3 s    A gets its reply
finished at t = 0.3 s: the waits overlap, so the total is the longest wait, not their sum

with await asyncio.sleep: tick 0, worker starts, tick 1, tick 2, worker done
with time.sleep:           tick 0, worker starts, worker done, tick 1, tick 2

four threads sent 10,000 deposits each to one actor: balance 40,000
running totals of squares through two channels: [1, 5, 14, 30, 55]
```

The price: cooperation

An event loop is **cooperative**: a task runs until it chooses to yield, so one task that doesn't yield stops all the others. A blocking call, a slow library that does synchronous network or file I/O, or a long computation inside a coroutine freezes every connection on that loop, which is why Node.js documentation repeats "don't block the event loop". The fixes are to use asynchronous libraries throughout, and to move blocking or heavy work to a thread or process with `asyncio.to_thread` or `run_in_executor`. Async also spreads through a program: a function that awaits must itself be async, so its callers must be too. Bob Nystrom called this the problem of function colour.

In return, code between two `await` s runs without interruption, so there are no data races within a single loop. But interleaving hasn't gone away; it's just confined to the `await` points. A check, then an `await`, then an act is the chapter 7 race again (exercise 11.1).

Actors

An **actor** is a unit of computation that owns its state outright and communicates only by messages. It has a mailbox, handles one message at a time, and may change its state, send messages and create other actors in response. Carl Hewitt proposed the model in 1973. Erlang, designed at Ericsson in 1986 for telephone switches, made it practical, and Akka brought it to the JVM. Because only the actor touches its state, no lock is needed: the example's account actor takes 40,000 deposits from four threads, one at a time, and ends with exactly 40,000. Erlang adds the other half of the idea: actors are isolated, so one that crashes takes nothing else with it, and a **supervisor** actor restarts it in a known state. That's how Erlang systems stay up for years.

Channels

A **channel** is a typed pipe between concurrent tasks. Tony Hoare described communicating sequential processes, CSP, in 1978, and Go made channels a core feature, with the advice "don't communicate by sharing memory; share memory by communicating". An unbuffered channel is a rendezvous: the sender waits until a receiver takes the value. A buffered channel is a bounded queue. Either way, a fast stage blocks when the next one falls behind, which is **backpressure**: the whole pipeline runs at the speed of its slowest stage, with bounded memory. The example's pipeline passes numbers through two channels with

room for one item each, to a stage that squares them and on to one that keeps a running total. Go's `select` waits on several channels at once, and closing a channel tells its readers that no more values are coming, as the example's `None` does.

None of these models wins everywhere. Threads and locks suit shared in-memory structures and CPU-bound parallel work. Event loops suit many connections that mostly wait. Actors suit long-lived stateful entities that should fail independently, and channels suit pipelines. Real systems mix them: Go runs goroutines, which are threads, connected by channels; a Node.js server runs an event loop with a pool of worker threads behind it; a Python service runs `asyncio` with a thread pool for blocking calls.

COMMON TRAPS

A blocking call inside async code, which stalls every task on the loop.

Check, then `await`, then act: the race is back at the `await`.

Calling a coroutine without awaiting it. It never runs, and Python only warns.

Unbounded queues and mailboxes, which hide a slow consumer until memory runs out.

Tasks started and forgotten, whose exceptions disappear. Keep a reference and await them.

EXERCISES

EXERCISE 11.1 A race with one thread

Two `asyncio` tasks run `if account.balance >= 80: await audit_log("withdraw"); account.balance -= 80` on an account holding 100. Can the account be overdrawn, even though only one thread exists? How do you fix it?

SOLUTION

Yes. The first task checks, finds 100, and pauses at the `await` while the log write happens. The loop runs the second task, which also finds 100 and also pauses. Both then resume and debit, leaving -60: exactly chapter 7's check-then-act race, with `await` as the point where the interleaving happens. Fixes: do the check and the debit together with no `await` in between, and log afterwards, since code between awaits can't be interrupted; or hold an `asyncio.Lock` across the whole sequence. Async code is only free of races between its `await` points.

EXERCISE 11.2 One slow handler

An `asyncio` server handles a thousand requests a second, each needing 1 ms of work. One endpoint builds a report with 200 ms of pure-Python computation. What happens to everyone else while it runs, and what are the fixes?

SOLUTION

For 200 ms the loop can't run anything else, so about 200 other requests queue up, each delayed by up to 200 ms, and the server's latency percentiles jump every time someone asks for a report. Moving the computation to a thread with `asyncio.to_thread` keeps the loop responsive, but under the GIL the thread still competes with the loop for the interpreter. For CPU-heavy work, a process pool through `run_in_executor` runs it in parallel. Otherwise the report can be split into chunks with an `await asyncio.sleep(0)` between them, which lets other tasks in, at the cost of making the report itself slower.

EXERCISE 11.3 Why bounded channels

In the example's pipeline the channels hold one item each. What would happen if the first stage produced a million values a second and the last could handle only a thousand, with unbounded channels? With bounded ones?

SOLUTION

Unbounded, the channel in front of the slow stage grows by about 999,000 items a second, until memory runs out, and every item waits longer and longer to be processed, so latency grows without limit too. Bounded, the fast stage blocks as soon as the channel is full and runs only as fast as items are taken, so the whole pipeline settles at a thousand a second with a fixed amount of memory in flight. That's backpressure, and the same mechanism appears throughout systems: a full pipe blocks its writer (chapter 3), TCP's flow control stops a fast sender, and a well-designed service rejects work it can't queue rather than accepting it all.

CHECKLIST

- ☐ Can you explain how an event loop serves many tasks on one thread?
- ☐ Can you say what an `await` does, and why one blocking call stalls the loop?

- ☐ Can you find a race in async code that has only one thread?
- ☐ Can you describe actors and supervisors, and channels with backpressure?
- ☐ Can you choose between threads, an event loop, actors and channels for a given problem?

GO DEEPER

- | | |
|----------------|---|
| BOOK | Operating Systems: Three Easy Pieces, chapter 33: event-based concurrency Arpaci-Dusseau & Arpaci-Dusseau |
| <hr/> | |
| ARTICLE | What color is your function? Bob Nystrom |
| <hr/> | |
| THESIS | Making reliable distributed systems in the presence of software errors Joe Armstrong, 2003 |

PART IV

Memory

Four chapters on how an operating system shares physical memory between processes that each believe they have all of it. Page tables translate every address. The TLB makes translation fast, and page faults let the kernel fill memory only when it's touched. Replacement policies decide what to evict when memory runs out. Allocators and garbage collectors manage the memory a program gets.

Virtual memory and paging

3 exercises

Every address a program uses is virtual. On every memory access the hardware translates it, through tables that only the kernel can change. Translation is what lets every process use the same addresses, keeps each process out of the others' memory, and lets the kernel share libraries, map files and hand out more memory than the machine has. This chapter splits an x86-64 address into its parts, walks the four-level page table by hand, triggers a page fault and a protection fault, and counts what the tables themselves cost.

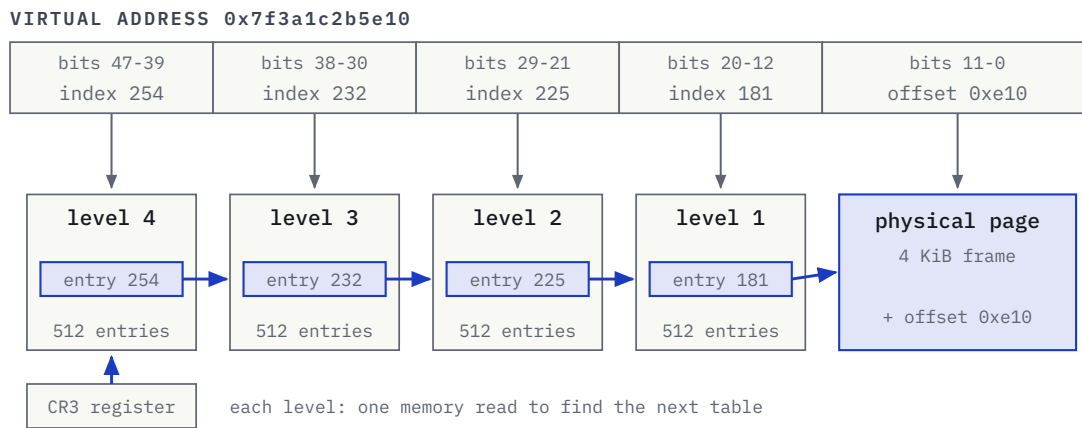
Why translate

Translation solves several problems at once. **Isolation:** a process can reach only the memory its tables map, so it can't read or corrupt another's. **Relocation:** every program can be linked to run at the same addresses, wherever its memory physically is. **Sparseness:** a 64-bit address space is huge, and a process uses a few scattered regions of it, code at the bottom and stacks near the top, without paying for the gaps. **Sharing:** the same physical memory can appear in many processes, as a shared library's code does, or as a parent's pages do after `fork` (chapter 13). And **overcommitment:** the kernel can promise memory and supply it only when it's touched, or take it back by moving it to disk.

Early machines translated with a single base and limit per process, which was simple but required each process's memory to be one contiguous piece. Segmentation used several such pieces and suffered external fragmentation, chapter 15's problem. **Paging**, used by every modern system, cuts both virtual and physical memory into fixed-size pages, 4 KiB on x86, and lets any virtual page map to any physical frame. ARM processors support 4, 16 and 64 KiB pages, and Apple silicon uses 16 KiB.

Splitting an address

On x86-64, a virtual address has 48 significant bits; newer processors and kernels can enable a fifth level and 57 bits. The low 12 bits are the **offset** within a 4 KiB page and pass through translation unchanged. The 36 bits above them are the virtual page number, and the hardware uses them as four 9-bit indices, one per level of a tree of tables. Each table is exactly one page: 512 entries of 8 bytes. An entry holds the physical frame number of the next table, or, at the last level, of the page itself, plus permission and status bits: present, writable, accessible from user mode, no-execute, and the accessed and dirty bits that the hardware sets when the page is used or written.



A page walk on x86-64. The **CR3** register holds the physical address of the top table, and each index chooses one entry, which points at the next table down. Four reads of memory find the page; the offset picks the byte within it.

Walking the tables

The kernel loads the physical address of a process's top-level table into **CR3** (on ARM, **TTBR0**) when it switches to that process, which is how a context switch changes address spaces (chapter 4). The memory management unit then walks the tree on every access, four dependent memory reads before the program's own read can even start. That would make every load five times slower, so processors cache recent translations in the TLB, the subject of chapter 13. The example builds the tree as nested dictionaries that exist only where something is mapped, maps three pages with different permissions, and translates five accesses.

EXAMPLE

paging.py · a four-level page table: splitting addresses, translating, page and protection faults, and what the tables cost

```

# x86-64 paging: a 48-bit virtual address is four 9-bit table indices and a 12-bit offset
PAGE = 4096
def split(va):
    return [(va >> shift) & 0x1FF for shift in (39, 30, 21, 12)], va & 0xFFF

class PageTables:
    """Four levels of 512-entry tables. Only the tables a process actually uses exist."""
    def __init__(self):
        self.root, self.tables, self.next_frame = {}, 1, 0x1000
    def map(self, va, frame, perms):
        (i1, i2, i3, i4), _ = split(va)
        node = self.root
        for i in (i1, i2, i3):
            if i not in node: node[i] = {}; self.tables += 1 # allocate a table on demand
            node = node[i]
        node[i4] = (frame, perms)
    def translate(self, va, access):
        (i1, i2, i3, i4), offset = split(va)
        node, reads = self.root, 0
        for i in (i1, i2, i3, i4):
            reads += 1 # each level is a memory

```

```

        read
        if i not in node: return f"page fault (no mapping; {reads} table reads)"
        node = node[i]
        frame, perms = node
        if access not in perms: return f"protection fault: {access!r} not allowed on a {perms}
        page"
        return f"physical 0x{frame * PAGE + offset:09x} ({reads} table reads)"

va = 0x7F3A1C2B5E10
idx, off = split(va)
print(f"0x{va:012x} -> table indices {idx}, offset 0x{off:03x}")

pt = PageTables()
pt.map(0x400000, 0x2A1, "r-x") # program code
pt.map(0x601000, 0x3F7, "rw-") # global data
pt.map(0x7FFFFFFDE000, 0x0B2, "rw-") # top of the stack
for va, access, what in ((0x400123, "x", "run code"), (0x601010, "w", "write data"),
                        (0x7FFFFFFDEFF8, "w", "push onto the stack"), (0x400123, "w", "write
                        to code"),
                        (0x0, "r", "read address 0 (a null pointer)")):
    print(f"    {what:32} {pt.translate(va, access)}")

# What the page tables themselves cost
print(f"\nthree mapped pages need {pt.tables} page tables: {pt.tables * PAGE // 1024} KiB")
flat = 2 ** 36 * 8 # one 8-byte entry for
every page
print(f"one flat table for all 48 bits: 2^36 entries x 8 bytes = {flat // 2 ** 30} GiB, for
every process")
big = PageTables()
for page in range(0x10000000 // PAGE, (0x10000000 + 2 ** 30) // PAGE): # map 1 GiB,
contiguous
    big.map(page * PAGE, page, "rw-")
print(f"mapping 1 GiB with 4 KiB pages: {2 ** 30 // PAGE:,} entries in {big.tables} tables "
      f"({big.tables * PAGE / 2 ** 20:.1f} MiB, {big.tables * PAGE / 2 ** 30:.2%} overhead)")
print(f"mapping 1 GiB with 2 MiB pages: {2 ** 30 // 2 ** 21} entries, one level fewer to walk")

```

OUTPUT

```

0x7f3a1c2b5e10 -> table indices [254, 232, 225, 181], offset 0xe10
run code          physical 0x0002a1123 (4 table reads)
write data        physical 0x0003f7010 (4 table reads)
push onto the stack physical 0x0000b2ff8 (4 table reads)
write to code      protection fault: 'w' not allowed on a r-x page
read address 0 (a null pointer) page fault (no mapping; 3 table reads)

three mapped pages need 8 page tables: 32 KiB
one flat table for all 48 bits: 2^36 entries x 8 bytes = 512 GiB, for every process
mapping 1 GiB with 4 KiB pages: 262,144 entries in 516 tables (2.0 MiB, 0.20% overhead)
mapping 1 GiB with 2 MiB pages: 512 entries, one level fewer to walk

```

Faults

When the walk finds an entry that isn't present, the processor raises a **page fault** and the kernel takes over, with the faulting address in a register. The kernel then decides what the fault means. If the address belongs to a region the process is entitled to, such as heap memory it allocated but never touched, a file it mapped, or a page that was moved out to disk, the kernel supplies the page, fixes the entry and resumes the instruction, and the program never knows. If the address belongs to nothing, the kernel sends `SIGSEGV`, the

segmentation fault. Page zero is left unmapped on purpose, so a null pointer is caught at once: the example's read of address 0 stops after three table reads, because no level 1 table exists for that region.

A **protection fault** is an access the entry forbids: a write to a read-only page, user-mode access to a kernel page, or an instruction fetch from a page marked no-execute. The example's write to its code page is one. Sometimes the kernel treats it as legitimate, which is how copy-on-write works (chapter 13), and otherwise the result is again `SIGSEGV`. Marking pages writable or executable but never both, a policy called W^X , is one of the defences against injected code from chapter 18 of *The Engineer's Mill*.

What the tables cost

A single flat table covering 48 bits would need 2^{36} entries of 8 bytes, 512 GiB for every process, which is why the table is a tree. The example's process maps three pages and needs only 8 tables, 32 KiB in all, because a table exists only where something below it is mapped. A densely mapped gigabyte needs 516 tables, 2 MiB, which is 0.2% overhead. Larger pages shrink this further: a 2 MiB page is mapped by a level 2 entry directly, skipping the last level, and a 1 GiB page by a level 3 entry. Chapter 13 shows the bigger benefit, for the TLB.

Every process's address space has two halves. The lower half holds the program; the upper half, which user mode can't access, maps the kernel, so that a system call or interrupt doesn't need to switch tables. Since the Meltdown attack of 2018, kernels on affected processors keep most of that upper half unmapped while user code runs, a technique called kernel page-table isolation, which is one reason system calls got slower. On Linux, `/proc/<pid>/maps` lists every region of a process's lower half, and `/proc/<pid>/smaps` shows how much of each is actually in memory.

COMMON TRAPS

Reading virtual size as memory use. Look at resident size (RSS), or proportional size (PSS) for shared pages.

Hard-coding 4096 as the page size. Apple silicon uses 16 KiB and some ARM servers 64 KiB; ask `sysconf`.

Trusting a successful allocation, when memory is only supplied on first touch and the out-of-memory killer may strike later.

Using memory after freeing it, which works until the page is reused or unmapped.

Turning on huge pages everywhere. Transparent huge pages can cause latency spikes, and some databases ask you to disable them.

EXERCISE 12.1 Split it yourself

Split the x86-64 address 0x00007fffffffe0a8, a typical address near the top of a process's stack, into its four table indices and its offset. What do the indices tell you about where in the address space it is?

SOLUTION

The offset is the low 12 bits, 0x0a8. Shifting right by 12, 21, 30 and 39 and keeping 9 bits each gives level 1 index 510, level 2 index 511, level 3 index 511 and level 4 index 255. The level 4 index 255 is the last entry of the lower half, since the indices from 256 to 511 belong to the kernel's upper half; and 511, 511, 510 below it put the address within the last few pages of that range. That's where Linux places the main thread's stack, which grows downward from just below the top of user space. You can check with `split` from the example, or with `cat /proc/self/maps` on Linux.

EXERCISE 12.2 Paging on a 32-bit machine

Classic 32-bit x86 paging splits an address 10 + 10 + 12, with 4-byte entries. How big would a flat, one-level table be per process? With two levels, what's the least a process with one code page near the bottom of memory and one stack page near the top needs?

SOLUTION

A flat table needs one entry per 4 KiB page of the 4 GiB address space, 2^{20} entries of 4 bytes, 4 MiB per process: affordable for a few processes in the 1980s, but wasteful for hundreds. With two levels, the top table, the page directory, has 1,024 entries each covering 4 MiB, and second-level tables exist only where needed. The code page and the stack page are in different 4 MiB regions, at opposite ends, so the process needs the directory plus two page tables: 3 pages, 12 KiB. The same idea, extended to four levels of 512 entries, is what makes 48-bit address spaces affordable.

EXERCISE 12.3 Why not bigger pages?

Bigger pages mean smaller page tables and fewer translations to cache. Give three costs that keep the base page size at 4 or 16 KiB.

SOLUTION

First, internal fragmentation: every region, such as each mapped file, each thread's stack and the end of the heap, wastes on average half a page, which adds up across thousands of regions when pages are 2 MiB. Second, coarser control: permissions, sharing and copy-on-write work on whole pages, so writing one byte to a shared 2 MiB page would copy 2 MiB instead of 4 KiB, and zeroing a fresh page costs more too. Third, paging in and out moves whole pages, so faults on sparse data read far more than was needed. That's why systems keep a small base page and offer large pages for regions that benefit, such as database buffer pools and big arrays, while Apple's 16 KiB is a middle point chosen for its own hardware.

CHECKLIST

- ☐ Can you list five things address translation makes possible?
- ☐ Can you split an x86-64 address into its indices and offset?
- ☐ Can you describe a page walk, and what a page-table entry contains?
- ☐ Can you explain how the kernel handles a page fault, and when it becomes SIGSEGV ?
- ☐ Can you estimate the memory a page table needs, and say how huge pages change it?

GO DEEPER

- | | |
|--------------|---|
| BOOK | Operating Systems: Three Easy Pieces, chapter 18: paging Arpaci-Dusseau & Arpaci-Dusseau |
| <hr/> | |
| BOOK | Operating Systems: Three Easy Pieces, chapter 20: smaller page tables Arpaci-Dusseau & Arpaci-Dusseau |
| <hr/> | |
| PAPER | What every programmer should know about memory, section 4: virtual memory Ulrich Drepper |

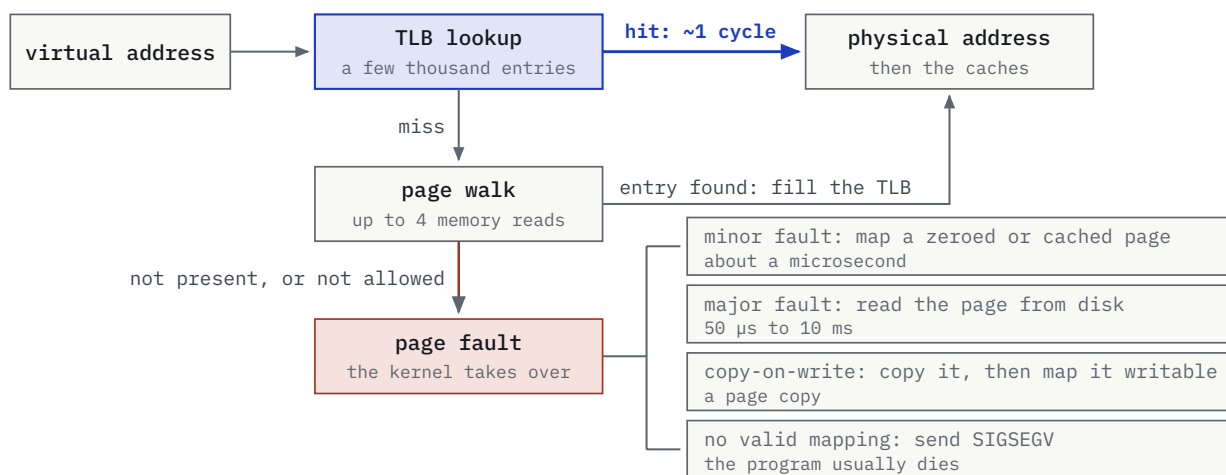
TLBs and page faults

3 exercises

A page walk on every access would make memory several times slower. Processors therefore cache recent translations in the translation lookaside buffer, and the kernel fills memory lazily, by handling page faults the first time each page is touched. This chapter simulates a TLB on four access patterns, with ordinary and huge pages. It shows copy-on-write reducing the copy that `fork` makes to almost nothing, and works out what a fault rate does to the average cost of a memory access.

The TLB

The **translation lookaside buffer** is a small cache of recent translations, each one a virtual page number, its physical frame and its permission bits. It's looked up on every memory access, in parallel with the first-level cache, and a hit costs about a cycle. Modern cores have two levels: a first-level data TLB of around 64 entries and a second-level TLB of 1,500 to 2,000 entries shared by code and data. A miss means a page walk. The processor does it in hardware on x86 and ARM, helped by caches of the upper levels of the tree, and it costs from a dozen cycles, when the tables are in cache, to hundreds, when they're not. How much memory the TLB can cover at once, its **reach**, is the number of entries times the page size.



From virtual address to data. The common case is the highlighted hit. Each step down the diagram costs roughly a hundred times more than the one above it.

Four access patterns

The example models a 1,536-entry TLB with least-recently-used replacement and runs 300,000 accesses in four patterns. Reading 8-byte values in order misses once per 512 accesses, 0.2%, because each 4 KiB page holds 512 of them. Touching one value every 4 KiB misses every time. Random reads in 64 MiB miss 91% of the time, because 64 MiB is 16,384 pages and the TLB holds 1,536; in 1 GiB, 99%. With 2 MiB pages every pattern misses at most 0.2% of the time, because the same TLB now reaches 3 GiB instead of 6 MiB.

EXAMPLE

tlb.py · a TLB simulator with 4 KiB and 2 MiB pages, copy-on-write fork, and the average cost of faults

```
import random
from collections import OrderedDict

# A TLB caches recent page translations. Model a 1,536-entry TLB with LRU replacement,
# the size of the second-level TLB on recent x86 cores.
def tlb_misses(addresses, page_size, entries=1536):
    tlb, misses = OrderedDict(), 0
    for a in addresses:
        page = a // page_size
        if page in tlb: tlb.move_to_end(page)
        else:
            misses += 1
            reads
            tlb[page] = True
            if len(tlb) > entries: tlb.popitem(last=False)
    return misses / len(addresses)

KiB, MiB, GiB = 2 ** 10, 2 ** 20, 2 ** 30
N = 300_000
random.seed(13)
patterns = {
    "sequential 8-byte reads": [i * 8 for i in range(N)],
    "one read every 4 KiB": [i * 4 * KiB for i in range(N)],
    "random reads in 64 MiB": [random.randrange(64 * MiB) for _ in range(N)],
    "random reads in 1 GiB": [random.randrange(GiB) for _ in range(N)],
}
print(f"{'TLB misses per access, {N:,} accesses each':>49} {'4 KiB pages':>11} {'2 MiB pages':>11}")
for name, addrs in patterns.items():
    print(f"    {name:46} {tlb_misses(addrs, 4 * KiB):11.2%} {tlb_misses(addrs, 2 * MiB):11.2%}")
print(f"TLB reach: 1,536 x 4 KiB = {1536 * 4 * KiB / MiB:.0f} MiB; "
      f"1,536 x 2 MiB = {1536 * 2 * MiB / GiB:.0f} GiB")

# Copy-on-write fork: share every page read-only, copy a page only when someone writes to it
def fork(parent_pages, written_by_child):
    child = dict(parent_pages)
    frames_shared = set(parent_pages.values())
    copies = 0
    for page in written_by_child:
        if child[page] in frames_shared:
            page
            child[page] = ("copy", page); copies += 1
    return copies

pages = {p: ("frame", p) for p in range(25_600)}
pages
print()
for label, writes in (("calls exec at once", []),
                     ("writes to 1% of its pages", range(0, 25_600, 100)),
                     ("rewrites everything", range(25_600))):
```

```

print(f"fork of a 100 MiB process, child {label:26} {fork(pages, writes):6,} of 25,600
      pages copied")

# What faults cost: the average access time with a 100 ns memory, when faults are served from
# storage
print(f"\n{'average time per memory access':38} {'faults from an SSD':>22} {'faults from a
      disk':>22}")
print(f"{'':38} {'(100 µs each)':>22} {'(5 ms each)':>22}")
for rate in (1e-6, 1e-4, 1e-3):
    ssd, disk = ((1 - rate) * 100 + rate * cost for cost in (100_000, 5_000_000))
    print(f"    one fault per {1 / rate:>9,.0f} accesses    {ssd:9,.1f} ns ({ssd / 100:5.2f}x) "
          f"{disk:11,.1f} ns ({disk / 100:5.2f}x)")

```

OUTPUT

TLB misses per access, 300,000 accesses each:	4 KiB pages	2 MiB pages
sequential 8-byte reads	0.20%	0.00%
one read every 4 KiB	100.00%	0.20%
random reads in 64 MiB	90.68%	0.01%
random reads in 1 GiB	99.44%	0.17%

TLB reach: 1,536 x 4 KiB = 6 MiB; 1,536 x 2 MiB = 3 GiB

fork of a 100 MiB process, child calls exec at once	0 of 25,600 pages copied
fork of a 100 MiB process, child writes to 1% of its pages	256 of 25,600 pages copied
fork of a 100 MiB process, child rewrites everything	25,600 of 25,600 pages copied

average time per memory access	faults from an SSD (100 µs each)	faults from a disk (5 ms each)
one fault per 1,000,000 accesses	100.1 ns (1.00x)	105.0 ns (1.05x)
one fault per 10,000 accesses	110.0 ns (1.10x)	600.0 ns (6.00x)
one fault per 1,000 accesses	199.9 ns (2.00x)	5,099.9 ns (51.00x)

Programs that jump around large amounts of memory, such as hash tables, graph algorithms, databases and garbage-collected heaps, can spend a large share of their time on TLB misses, and huge pages often speed them up by several to tens of percent. Linux offers them explicitly, through `hugetlbfs` or `mmap` with `MAP_HUGETLB`, and transparently, by promoting suitable regions automatically. Transparent huge pages have a cost of their own: assembling a 2 MiB physical region can require compacting memory at awkward moments, so latency-sensitive services such as Redis recommend turning them off.

The TLB across switches

TLB entries belong to one address space. When the kernel switches to another process, entries for the old one must not be used, so older processors flushed the whole TLB on every switch, and the new process started with every access missing. Tagging each entry with an address-space ID, called PCID on x86 and ASID on ARM, lets entries from several processes coexist and survive switches. The opposite problem arises when a mapping changes. If one thread unmaps memory, every core that might have cached the translation must drop it, so the kernel interrupts those cores with a **TLB shutdown**. Multithreaded programs that map and unmap memory constantly, as some allocators do, can spend real time on these.

Demand paging

`malloc` for a large block, and `mmap` of anonymous memory, only reserve virtual addresses. No physical memory is assigned until the program first touches each page. That first touch faults, and the kernel finds a free frame, zeroes it, maps it and resumes: a **minor fault**, costing around a microsecond. A file mapped with `mmap` works the same way. If the file's page is already in memory, in the kernel's page cache, the fault just maps it, another minor fault. If it isn't, the kernel reads it from storage, a **major fault**, which costs a disk access. This is why a process's resident size grows as it runs, why a freshly started program is slower than a warm one, and why `/usr/bin/time -v` and `perf stat -e page-faults` report fault counts.

Copy-on-write

`fork` would be ruinously slow if it copied a large process's memory, so it doesn't. It copies only the page tables, and marks every page in both parent and child read-only and shared. As long as both only read, they share the same physical pages. When either one writes to a page, the protection fault goes to the kernel, which sees a copy-on-write page, copies just that one, maps the copy writable for the writer, and resumes. In the example, a child that calls `exec` straight away copies none of a 100 MiB parent's 25,600 pages, and one that writes to 1% of them copies 256. Redis takes its snapshots this way: it forks, and the child writes the frozen view of memory to disk while the parent keeps serving. The memory cost depends on how many pages the parent writes during the snapshot (exercise 13.2).

What faults cost

The example's last table turns fault rates into an average memory access time, assuming memory takes 100 ns. One fault per million accesses costs almost nothing. One per ten thousand costs 10% when the faults are served from an SSD at 100 μ s, and makes memory six times slower when they're served from a disk at 5 ms. One per thousand, from disk, makes it 51 times slower. A machine in that state is barely running, which is the subject of chapter 14.

COMMON TRAPS

Random access over gigabytes with 4 KiB pages, where TLB misses dominate.

Benchmarking cold and counting first-touch faults as steady-state cost, or the reverse.

Forking a large process that keeps writing, which can nearly double its memory.

Constant `mmap` and `munmap` in multithreaded code, paying for TLB shootdowns.

Transparent huge pages on a latency-sensitive service without measuring.

EXERCISE 13.1 TLB reach

A core has a 64-entry first-level TLB and a 1,536-entry second-level TLB. A program's hot data is 24 MiB, accessed randomly. Roughly how often does it miss each level with 4 KiB pages? With 2 MiB pages?

SOLUTION

With 4 KiB pages, the first level reaches $64 \times 4 \text{ KiB} = 256 \text{ KiB}$, about 1% of 24 MiB, so it misses about 99% of random accesses; the second level reaches 6 MiB, a quarter of the data, so about 75% of accesses go on to a page walk. With 2 MiB pages, the 24 MiB is only 12 pages. They fit in the first-level TLB, which on most cores has a separate set of entries for 2 MiB pages, so after warm-up there are essentially no misses. A program like this can speed up noticeably just by putting its data on huge pages, without any change to its code.

EXERCISE 13.2 The cost of a snapshot

A 10 GiB Redis process forks to write a 60-second snapshot. Meanwhile clients modify 20,000 keys a second, each on a different, random 4 KiB page. How much extra memory can the snapshot cost? What if the memory used 2 MiB transparent huge pages?

SOLUTION

Each write to a page not yet copied makes the kernel copy that page. Over 60 seconds that's up to 1.2 million writes, against $10 \text{ GiB} / 4 \text{ KiB} \approx 2.6 \text{ million pages}$. Random writes repeat some pages, so roughly 1 million distinct pages are copied, about 4 GiB of extra memory, 40% more than the process's own size. With 2 MiB pages there are only 5,120 pages, and 1.2 million random writes touch almost every one of them within seconds; each copy is 2 MiB, so the snapshot costs close to the full 10 GiB again, and each first write stalls while 2 MiB is copied. That's why Redis's documentation tells you to disable transparent huge pages and to leave memory headroom for snapshots.

EXERCISE 13.3 Touch before you time

A benchmark allocates a 1 GiB array and times one pass over it, then a second pass. The first pass is much slower. Explain why, and say how to make the benchmark measure what you mean.

SOLUTION

The allocation only reserved address space. The first pass touches 262,144 pages of 4 KiB for the first time, and each touch is a minor fault in which the kernel finds a frame and zeroes it, so at around a microsecond each the first pass carries a quarter of a second of fault handling, plus the time to zero a gigabyte. The second pass runs over memory that's already mapped. To measure steady-state speed, touch the memory before the timed region, or allocate with `MAP_POPULATE`, and report the warm runs. To measure start-up cost, time the first pass on purpose and say so. Either way, count faults with `perf stat` so the difference is explained rather than guessed.

CHECKLIST

- ☐ Can you explain what the TLB caches, and compute its reach?
- ☐ Can you predict TLB behaviour for sequential, strided and random access?
- ☐ Can you explain address-space IDs and TLB shootdowns?
- ☐ Can you distinguish minor and major faults, and explain demand paging?
- ☐ Can you explain copy-on-write and estimate what a `fork` will copy?

GO DEEPER

BOOK Operating Systems: Three Easy Pieces, chapter 19: translation lookaside buffers Arpaci-Dusseau & Arpaci-Dusseau

DOCS Transparent hugepage support The Linux kernel documentation

DOCS Diagnosing latency issues: fork and transparent huge pages Redis

Swapping and page replacement

3 exercises

When memory is full and a process needs another page, the kernel must evict one, writing it to disk first if it has changed. Which page it chooses decides whether the machine runs at memory speed or disk speed. This chapter simulates the classic policies: first in first out, least recently used, the clock algorithm that approximates it, and the optimal policy no real system can implement. It meets Belady's anomaly, in which more memory causes more faults, and the looping access pattern that defeats recency completely. It ends with thrashing, and how modern systems avoid it.

Swapping

Physical memory is a cache for the pages processes have touched. When it's full, the kernel picks a victim page. A **clean** page, one that matches a copy on disk such as the code of a program or a file read through the page cache, can simply be dropped and read back if needed. A **dirty** page, such as modified heap memory, must first be written to **swap space**, a partition or file reserved for it, or for file data, back to its file. The entry for the page in the process's page table is then marked not present, and the next access faults and reads it back: a major fault (chapter 13). Linux reclaims memory in the background with a kernel thread, `kswapd`, which starts when free memory falls below a low watermark, so that most allocations find a free page without waiting.

Choosing a victim

The ideal policy, which László Belady described in 1966, evicts the page that won't be needed for the longest time. It's optimal and impossible, since it needs to know the future, but it's the yardstick: the example's `OPT` computes it from the whole reference string. **FIFO** evicts the page that has been in memory longest, which is simple and ignores how much the page is used. **LRU**, least recently used, evicts the page whose last use is furthest in the past, betting that the recent past predicts the near future. Exact LRU would mean updating an ordering on every memory access, far too expensive in hardware, so kernels approximate it.

The **clock algorithm** is the classic approximation. The processor sets a page's accessed bit in its page-table entry whenever the page is used. The kernel arranges the frames in a circle and moves a hand around it. If the page under the hand has its bit set, the kernel clears the bit and moves on, giving the page a second chance; the first page found with its bit clear is the victim. Pages in active use keep getting their bits set and survive, while pages nobody has touched since the last sweep go. Linux uses a refinement with two lists, active and inactive, and a page must be used twice to be promoted, so a single large scan doesn't flush everything useful.

EXAMPLE

replacement.py · FIFO, LRU, CLOCK and the optimal policy on Belady's string, a loop, and a workload with a working set

```
import random
from collections import OrderedDict

def faults(refs, frames, policy):
    memory, n, hand = OrderedDict(), 0, 0          # memory: page -> referenced bit (for CLOCK)
    order = []                                     # CLOCK's circular list of pages
    nxt, last = [len(refs)] * len(refs), {}       # OPT: when each reference's page is next
    used
    for i in range(len(refs) - 1, -1, -1):
        nxt[i] = last.get(refs[i], len(refs)); last[refs[i]] = i
    when = {}
    for i, page in enumerate(refs):
        when[page] = nxt[i]
        if page in memory:
            if policy == "LRU": memory.move_to_end(page)
            memory[page] = 1                       # hardware sets the referenced bit on every
            use
            continue
        n += 1                                     # a page fault
        if len(memory) == frames:
            if policy in ("FIFO", "LRU"):
                victim = next(iter(memory))         # oldest arrival, or least recently used
            elif policy == "OPT":
                # Belady's optimum: the page needed
                # furthest ahead
                victim = max(memory, key=lambda p: when[p])
            elif policy == "CLOCK":
                # sweep the hand, giving used pages a
                # second chance
                while memory[order[hand]]:
                    memory[order[hand]] = 0; hand = (hand + 1) % len(order)
                victim = order[hand]
            del memory[victim]
            if policy == "CLOCK": order[hand] = page; hand = (hand + 1) % len(order)
        elif policy == "CLOCK":
            order.append(page)
        memory[page] = 1
    return n

refs = [1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5]
print(f"reference string {' '.join(map(str, refs))}")
for policy in ("FIFO", "LRU", "OPT"):
    three, four = faults(refs, 3, policy), faults(refs, 4, policy)
    print(f"    {policy:5} faults with 3 frames: {three:2}    with 4 frames: {four:2}")
print("    FIFO faults more with more memory: Belady's anomaly. LRU and OPT never do.")

# A loop over 5 pages with 4 frames: recency is exactly the wrong guide
loop = [1, 2, 3, 4, 5] * 20
print(f"\nlooping over 5 pages with 4 frames, {len(loop)} references:")
for policy in ("FIFO", "LRU", "CLOCK", "OPT"):
    print(f"    {policy:5} {faults(loop, 4, policy):3} faults")

# Real programs have locality: most references go to a small working set that drifts slowly
random.seed(14)
refs, hot = [], list(range(20))
for _ in range(20_000):
    if random.random() < 0.02: hot[random.randrange(20)] = random.randrange(1000) # working
    set drifts
    refs.append(random.choice(hot) if random.random() < 0.9 else random.randrange(1000))
print("\n20,000 references with 90% going to a working set of 20 pages, out of 1,000:")
print("    frames " + "".join(f"{f:>8}" for f in (10, 20, 30, 40, 60)))
for policy in ("FIFO", "CLOCK", "LRU", "OPT):
```

```
rates = [faults(refs, f, policy) / len(refs) for f in (10, 20, 30, 40, 60)]
print(f"    {policy:6}" + " ".join(f"{r:8.1%}" for r in rates))
```

OUTPUT

```
reference string 1 2 3 4 1 2 5 1 2 3 4 5
FIFO  faults with 3 frames:  9   with 4 frames: 10
LRU   faults with 3 frames: 10   with 4 frames:  8
OPT   faults with 3 frames:  7   with 4 frames:  6
FIFO  faults more with more memory: Belady's anomaly. LRU and OPT never do.

looping over 5 pages with 4 frames, 100 references:
FIFO  100 faults
LRU   100 faults
CLOCK 100 faults
OPT   28 faults

20,000 references with 90% going to a working set of 20 pages, out of 1,000:
frames      10      20      30      40      60
FIFO        61.4%   34.8%   23.5%   18.8%   15.1%
CLOCK       60.9%   29.9%   15.7%   12.4%   11.6%
LRU         60.0%   27.0%   13.4%   11.8%   11.5%
OPT         32.5%   12.4%   10.6%    9.9%    9.1%
```

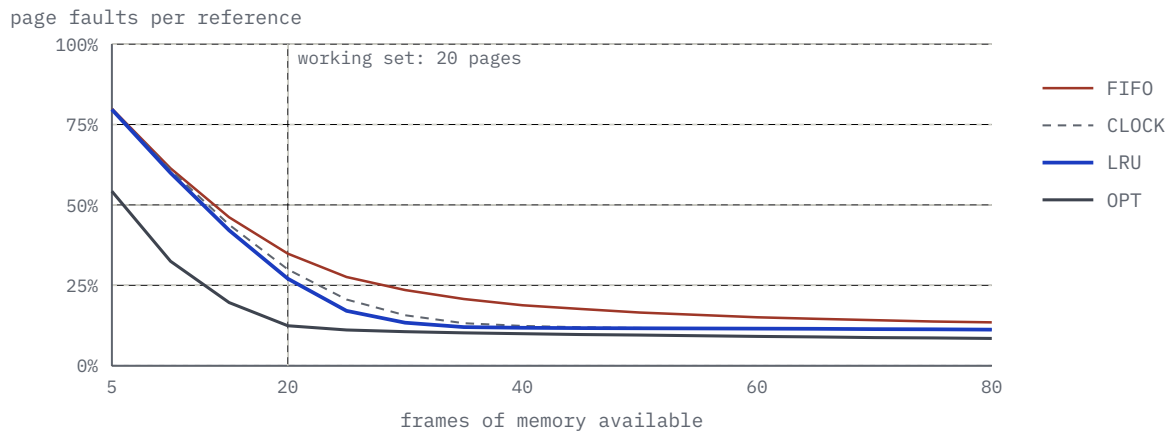
Belady's anomaly, and the loop

Belady also found something stranger. On the reference string 1 2 3 4 1 2 5 1 2 3 4 5, FIFO makes 9 faults with three frames and 10 with four: giving it more memory makes it worse. LRU and OPT can't behave this way. They're **stack algorithms**: the pages they'd keep with n frames are always a subset of those they'd keep with $n + 1$, so more memory never hurts. FIFO's choice depends on arrival order, which changes in unrelated ways when memory grows.

The second part of the example is the access pattern that defeats recency. A program loops over 5 pages with room for 4. Each page is evicted just before it's needed again, so FIFO, LRU and CLOCK fault on every one of the 100 references, while OPT faults on 28. The same pattern hit chapter 10 of *The Engineer's Mill* in caches, and it's common in practice: a sequential scan of a file larger than memory evicts everything else and gains nothing from it. Databases defend against it with scan-resistant buffer pools, and Linux's two-list design exists for the same reason.

Working sets

Real programs have locality. At any moment they use a small **working set** of pages heavily, and it drifts slowly as the program moves through its work, an idea Peter Denning formalised in 1968. The example's last workload sends 90% of its references to a working set of 20 pages out of 1,000, replacing one of them now and then. The figure plots its fault rate against memory size for all four policies.



Fault rate against memory size, computed with the example's policies. Below the working set every policy struggles; once memory holds the working set, the faults that remain are references nothing could have predicted. CLOCK tracks LRU closely for a fraction of its cost.

Two things stand out. The curves have a knee near the working-set size: memory below it is nearly useless, and memory beyond it buys little, because the remaining faults are the 10% of references to random pages, which no policy can avoid. And CLOCK, with one bit per page, stays close to exact LRU, while FIFO, which ignores use altogether, needs noticeably more memory for the same fault rate.

Thrashing

When the working sets of all running processes add up to more than physical memory, every process keeps faulting on pages another process's faults just evicted. The disk becomes the bottleneck, the processor sits idle waiting for it, and almost no work gets done. This is **thrashing**, and chapter 13's arithmetic shows why it's so severe: a fault per thousand accesses served from a disk makes memory 51 times slower. Early systems responded by reducing the number of programs running, suspending some until the rest fit, which is the right idea. Today, servers often run with little or no swap and let the out-of-memory killer end a process rather than letting the whole machine thrash. Linux's pressure stall information, in `/proc/pressure/memory`, reports how much time tasks spend waiting for memory, and tools such as `systemd-oomd` act on it before the machine locks up. Phones mostly avoid swapping to flash storage: iOS and Android compress memory instead, ask apps to release what they can, and then terminate background apps.

COMMON TRAPS

Treating swap as extra memory. It's a slow overflow; a working set in swap means thrashing.

A large sequential scan through a cache that only knows recency, evicting everything useful.

Sizing memory by average use rather than by the working set at peak.

Reading "free memory" as unused capacity. Linux fills spare memory with the page cache on purpose; look at "available".

Containers without memory limits, so one process's growth pushes every other into swap.

EXERCISES

EXERCISE 14.1 Belady by hand

Trace FIFO on 1 2 3 4 1 2 5 1 2 3 4 5 with three frames and with four, marking each fault. Where exactly does the extra frame start hurting?

SOLUTION

With three frames: 1, 2 and 3 fault; 4 faults and evicts 1; 1 faults and evicts 2; 2 faults and evicts 3; 5 faults and evicts 4; then 1 and 2 hit; 3 faults and evicts 1; 4 faults and evicts 2; 5 hits. That's 9 faults. With four frames: 1 to 4 fault; 1 and 2 hit; 5 faults and evicts 1; 1 faults and evicts 2; 2 faults and evicts 3; 3 faults and evicts 4; 4 faults and evicts 5; 5 faults and evicts 1. That's 10. The difference starts at the first 5. With three frames, the evictions forced by 1 and 2 just before it had cycled the old pages out, so 5 evicts 4 and leaves 1 and 2 in place for the next two references. With four frames, 1 and 2 hit earlier and stay at the front of the queue, so 5 evicts 1, then 1 evicts 2, and the queue keeps evicting exactly the page needed next.

EXERCISE 14.2 Why is OPT impossible, and why is it useful?

Explain why no operating system can implement the optimal policy, and give two practical uses for it anyway.

SOLUTION

OPT evicts the page whose next use is furthest in the future, so it must know every future reference, which depends on program inputs, timing and other processes that haven't happened yet. Offline, with a recorded trace, it can be computed exactly, which gives two uses. It's a lower bound: if LRU makes 12% faults and OPT 10% on the same trace, no policy could save more than that gap, which tells you whether a smarter policy is worth the effort or whether you need more memory instead. And it's a target for designs that do see the future in special cases, such as a database that knows the plan of the query it's running, or a video player that knows which frames come next.

EXERCISE 14.3 Sizing for the working set

Eight processes each have a working set of 3 GiB and run on a machine with 16 GiB of memory and a disk for swap. What happens, and what are the options?

SOLUTION

The working sets add up to 24 GiB, half as much again as the machine has. Whatever the policy, each process keeps faulting on pages the others evicted, the disk saturates, and throughput collapses: thrashing. Running all eight at once is slower than running fewer, because the time goes to moving pages instead of computing. The options: run at most five at a time, about 15 GiB of working sets, and queue the rest, which is what a batch scheduler or a worker-pool size does; add memory; reduce each working set, for example by processing data in smaller pieces or using more compact data structures; or split the work across machines. Adding swap space does nothing useful, because the problem is the speed of swap, not its size.

CHECKLIST

- ☐ Can you explain clean and dirty pages and what eviction does with each?
- ☐ Can you run FIFO, LRU and OPT on a reference string by hand?
- ☐ Can you explain the clock algorithm and why it approximates LRU?
- ☐ Can you explain Belady's anomaly and why stack algorithms avoid it?
- ☐ Can you recognise thrashing and name the ways to stop it?

GO DEEPER

BOOK Operating Systems: Three Easy Pieces, chapter 22: swapping policies
Arpaci-Dusseau & Arpaci-Dusseau

DOCS PSI: pressure stall information The Linux kernel documentation

PAPER The working set model for program behavior Peter Denning, 1968

Memory allocation and garbage collection

3 exercises

The kernel hands a process memory in pages. Programs want objects of every size, created and destroyed millions of times a second, and the allocator in between decides where each one goes. Its choices determine how much memory is wasted, in holes between objects and in rounding inside them. This chapter runs a free-list allocator under three placement policies, compares the rounding of a buddy allocator with that of size classes, and then turns to garbage collection: reference counting, the cycles it can't free, and the mark-and-sweep collector that can.

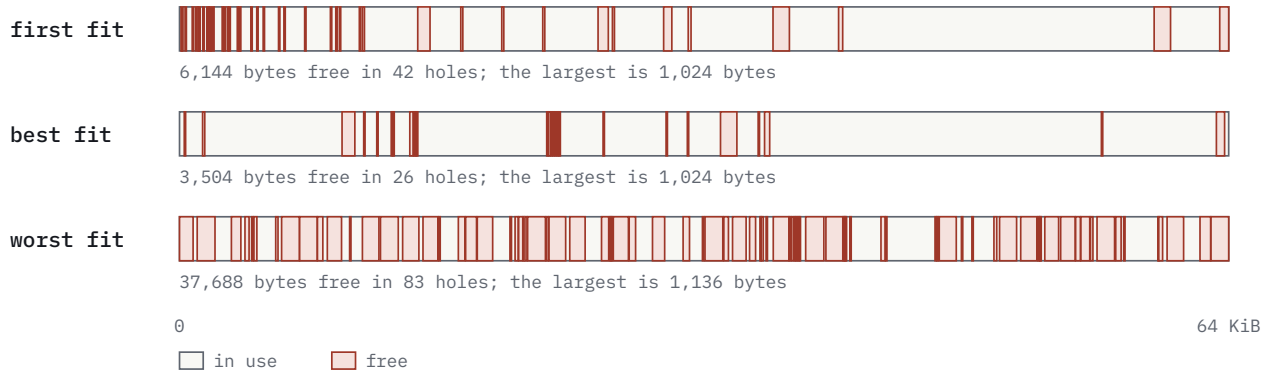
From pages to objects

A process gets memory from the kernel in large pieces: `brk` to grow the heap, and `mmap` for big blocks and for allocators that manage their own regions. The C library's `malloc` and `free`, or a replacement such as `jemalloc`, `tcmalloc` or `mimalloc`, carve those pieces into objects. The allocator must find space for each request quickly, return freed space for reuse, and keep two kinds of waste low. **External fragmentation** is free memory broken into holes too small for the requests that arrive. **Internal fragmentation** is space wasted inside allocated blocks, because a request was rounded up.

Free lists and placement

A simple allocator keeps a list of free blocks. To allocate, it finds a free block large enough, splits off what's needed and keeps the rest on the list. To free, it returns the block and **coalesces** it with free neighbours, so that small holes merge back into large ones. The placement policy decides which free block to use. **First fit** takes the first one that's large enough, which is fast. **Best fit** takes the smallest that fits, to keep large blocks intact. **Worst fit** takes the largest, on the theory that the leftover piece will be big enough to be useful.

The example runs 4,000 random allocations and frees, in sizes from 16 bytes to 2 KiB, through a 64 KiB heap with each policy. First fit and best fit fill the heap almost completely; their failures come when it's full, with a few kilobytes left in holes. Worst fit fails more than twice as often, with more than half the heap still free, because it has broken that space into 83 holes of which the largest is 1,136 bytes. It splits the large blocks first, which leaves no room for the large requests. Studies of real programs' allocation traces agree: best fit and first fit waste similar, small amounts, and worst fit does badly.



The three heaps after the same 4,000 requests, to scale. Worst fit has the most free memory and the least use for it: every hole is too small for a 2 KiB request.

EXAMPLE

allocators.py · first, best and worst fit, rounding in a buddy allocator and in size classes, reference counting, and mark and sweep

```
import gc, random, weakref

# 1. A free-list allocator over a 64 KiB heap, with coalescing of neighbouring free blocks
class Heap:
    def __init__(self, size, fit):
        self.free, self.fit, self.used = [(0, size)], fit, {} # free: sorted (start, length)
    def malloc(self, n):
        fits = [b for b in self.free if b[1] >= n]
        if not fits: return None
        start, length = {"first": fits[0], "best": min(fits, key=lambda b: b[1]), "worst": max(fits, key=lambda b: b[1])}[self.fit]
        self.free.remove((start, length))
        if length > n: self.free.append((start + n, length - n)) # split off the remainder
        self.free.sort(); self.used[start] = n
        return start
    def release(self, start):
        self.free.append((start, self.used.pop(start))); self.free.sort()
        merged = []
        for s, l in self.free: # coalesce adjacent free blocks
            if merged and merged[-1][0] + merged[-1][1] == s:
                merged[-1] = (merged[-1][0], merged[-1][1] + l)
            else: merged.append((s, l))
        self.free = merged

def workload(fit, seed=15):
    rng, heap, live, failed = random.Random(seed), Heap(65536, fit), [], 0
    for step in range(4000):
        if live and rng.random() < 0.48: # free a random live block
            heap.release(live.pop(rng.randrange(len(live))))
        else:
            size = rng.choice([16, 24, 32, 48, 64, 128, 256, 1024, 2048])
            p = heap.malloc(size)
            if p is None: failed += 1
            else: live.append(p)
    free = sum(l for _, l in heap.free)
    largest = max(l for _, l in heap.free)
    return failed, free, largest, len(heap.free)
```

```

print("4,000 random mallocs and frees in a 64 KiB heap:")
for fit in ("first", "best", "worst"):
    failed, free, largest, holes = workload(fit)
    print(f"    {fit:5} fit: {failed:3} requests failed; {free:6,} bytes free in {holes:3}
          holes, "
          f"largest {largest:6,} ({1 - largest / free:.0%} external fragmentation)")

# 2. Rounding up: a buddy allocator uses powers of two; size classes round much less
rng = random.Random(2)
sizes = [int(2 ** rng.uniform(4.1, 12)) for _ in range(10_000)] # small requests are the
                        most common
classes = [16, 32, 48, 64, 80, 96, 112, 128, 160, 192, 224, 256, 320, 384, 448, 512,
           640, 768, 896, 1024, 1280, 1536, 1792, 2048, 2560, 3072, 3584, 4096] # like
                        jemalloc's
buddy = sum(1 < (s - 1).bit_length() for s in sizes)
classed = sum(next(c for c in classes if c >= s) for s in sizes)
asked = sum(sizes)
print("\ninternal fragmentation on 10,000 requests between 17 bytes and 4 KiB:")
print(f"    powers of two (buddy):      {buddy / asked - 1:.1%} wasted")
print(f"    28 size classes:                {classed / asked - 1:.1%} wasted")

# 3. Garbage collection: reference counting frees at once, but not cycles
class Node:
    def __init__(self, name): self.name, self.other = name, None
freed = []
a = Node("a"); weakref.finalize(a, freed.append, "a")
del a
print(f"\nreference counting: after `del a`, freed {freed}")
b, c = Node("b"), Node("c"); b.other, c.other = c, b # a cycle: b -> c -> b
weakref.finalize(b, freed.append, "b"); weakref.finalize(c, freed.append, "c")
del b, c
print(f"a cycle b <-> c after `del b, c`: freed {freed} (each still has a reference from the
      other)")
gc.collect()
print(f"after the cycle collector runs: freed {sorted(freed)}")

# 4. Mark and sweep, from scratch: mark everything reachable from the roots, sweep the rest
heap = {"main": ["config", "users"], "config": [], "users": ["alice", "bob"], "alice": [],
        "bob": ["alice"],
        "temp1": ["temp2"], "temp2": ["temp1"], "old_cache": ["config"]}
roots, marked, todo = ["main"], set(), ["main"]
while todo:
    obj = todo.pop()
    if obj not in marked: marked.add(obj); todo.extend(heap[obj])
print(f"\nmark and sweep from root 'main': keeps {sorted(marked)}")
print(f"    frees {sorted(set(heap) - marked)}, the cycle included")

```

OUTPUT

```

4,000 random mallocs and frees in a 64 KiB heap:
  first fit:  42 requests failed;  6,144 bytes free in  42 holes, largest  1,024 (83% external
              fragmentation)
  best fit:   36 requests failed;  3,504 bytes free in  26 holes, largest  1,024 (71% external
              fragmentation)
  worst fit:  87 requests failed; 37,688 bytes free in  83 holes, largest  1,136 (97% external
              fragmentation)

internal fragmentation on 10,000 requests between 17 bytes and 4 KiB:
  powers of two (buddy):      39.3% wasted
  28 size classes:            9.2% wasted

reference counting: after `del a`, freed ['a']
a cycle b <-> c after `del b, c`: freed ['a'] (each still has a reference from the other)
after the cycle collector runs: freed ['a', 'b', 'c']

mark and sweep from root 'main': keeps ['alice', 'bob', 'config', 'main', 'users']

```

```
frees ['old_cache', 'temp1', 'temp2'], the cycle included
```

Rounding: buddies and size classes

A **buddy allocator** manages blocks whose sizes are powers of two. A request is rounded up to the next power, and a block is split in halves, its "buddies", until one of the right size appears; when both halves are free again they merge. Finding a buddy is a single bit operation on the address, so splitting and merging are fast, and the Linux kernel allocates physical pages this way. The cost is rounding. In the example, 10,000 requests between 17 bytes and 4 KiB, with small sizes the most common, waste 39% of the memory they're given.

User-space allocators instead use **size classes**: a fixed list of sizes spaced roughly a quarter apart, and a separate pool of same-sized slots for each, carved out of pages. A request goes to the smallest class that fits, so rounding waste is bounded, 9% in the example with 28 classes, and since every slot in a pool is the same size, allocating and freeing are a pop and a push on a list, with no searching and no external fragmentation within a pool. jemalloc, tcmalloc and mimalloc all work this way, add per-thread caches so that most allocations take no lock, and hand very large requests straight to `mmap`. The Linux kernel does the same for its own objects with its slab allocator.

Garbage collection: reference counting

In C and C++ the programmer calls `free`, or a destructor does, and the bugs are leaks, double frees and use after free. Garbage-collected languages instead free an object automatically when the program can no longer reach it. The simplest way is **reference counting**: every object counts the references to it, and when the count drops to zero, it's freed immediately. CPython works this way, as do Swift and Objective-C, and C++'s `shared_ptr` and Rust's `Rc`. It's predictable, and memory is reclaimed promptly. It has two costs. Every assignment updates counts, which in multithreaded code must be atomic. And it can't free a **cycle**: in the example, two objects that refer to each other keep each other's count above zero after the program drops its own references, so neither is freed until CPython's separate cycle collector runs.

Tracing collectors

A **tracing** collector instead starts from the **roots**, which are global variables, the stacks of every thread and the registers, and follows every reference to find everything reachable. **Mark and sweep** marks each reachable object, then sweeps the heap and frees everything unmarked, cycles included, as the example's last part does. A **copying** collector moves the live objects into a fresh region instead, which compacts the heap and makes allocation a pointer increment, at the cost of needing space to copy into. Most programs follow the **generational hypothesis**: most objects die young. So collectors in the JVM, .NET and JavaScript engines collect a small young generation often and cheaply, and the old generation rarely.

The price of tracing is the pause, or the work needed to avoid it. A simple collector stops the program while it traces. Concurrent collectors such as Java's ZGC and Go's collector do most of the work while the program runs, using write barriers to track changes, and keep pauses to around a millisecond or less in exchange for some throughput and extra memory. Tracing collectors also typically need more memory than the live data, often twice as much, to run efficiently.

COMMON TRAPS

Long-running services whose memory only grows because freed holes can't be reused. Measure fragmentation, not just leaks.

Cycles in reference-counted code, such as parent and child objects that point at each other. Use weak references for back-pointers.

Allocating in a hot loop what could be allocated once and reused.

Assuming `free` returns memory to the operating system. Allocators usually keep it for reuse.

Sizing a garbage-collected service by its live data without the collector's headroom.

EXERCISES

EXERCISE 15.1 Internal or external?

Classify each waste as internal or external fragmentation: a 33-byte request served from a 48-byte size class; 10 MiB free in a heap, but in pieces of at most 4 KiB, when a 64 KiB request arrives; a 5 KiB file stored in two 4 KiB disk blocks; a buddy allocator giving a 65 KiB request a 128 KiB block.

SOLUTION

The 33-byte request in a 48-byte slot wastes 15 bytes inside an allocated block: internal. Ten mebibytes free in small pieces, with no piece big enough, is the textbook case of external fragmentation. The file in two blocks wastes 3 KiB inside the second block, internal fragmentation on disk, which is why file systems pack small files or their tails specially. The buddy allocator's 128 KiB block for 65 KiB wastes 63 KiB inside the block, internal again, and the worst case for powers of two, just under half the block. Internal waste is bounded by the rounding rule; external waste depends on the history of allocations and frees, which is why it's harder to predict and to fix.

EXERCISE 15.2 The cycle that leaks

In a reference-counted language, a `Window` holds a list of its `Button`s and each button keeps a reference to its window so it can redraw it. What happens when the program drops its last reference to a window? Give two fixes.

SOLUTION

The window still has a reference from each of its buttons, and each button from the window's list, so every count stays above zero and none of them is freed: the whole window, with every button and everything they refer to, leaks. In Swift or with C++'s `shared_ptr`, nothing ever reclaims it. In CPython the cycle collector eventually does, but later and at a cost. Fixes: make the back-pointer weak, so a button's reference to its window doesn't count (Python's `weakref`, Swift's `weak`, C++'s `weak_ptr`); or break the cycle explicitly when the window closes, by clearing its button list. The general rule is that ownership should form a tree, with back-references weak.

EXERCISE 15.3 Memory that won't come back

A Python service's resident memory rises to 4 GB while it processes a large batch, then stays at 4 GB after the batch's objects have all been freed. Is that a leak? Explain what's likely happening and how to tell.

SOLUTION

Probably not a leak. Freed objects return to the allocator, not to the operating system. CPython's small-object allocator manages memory in arenas and can release an arena only when every object in it is free, and a single surviving object pins its arena. `malloc` likewise keeps freed memory for reuse and returns heap memory to the kernel only from the top. So resident size stays high, but the memory is available for the next batch. To tell the difference, run a second batch: a leak grows again, while reused memory stays near 4 GB. `tracemalloc` shows what Python objects are live, and `sys._debugmallocstats()` shows arenas. If the plateau matters, process batches in a worker process that exits afterwards, which returns everything at once.

- ☐ Can you distinguish internal and external fragmentation?
- ☐ Can you explain first, best and worst fit, and coalescing?
- ☐ Can you explain buddy allocation and size classes, and what each wastes?
- ☐ Can you explain reference counting, and why it can't free cycles?
- ☐ Can you describe mark and sweep, copying and generational collection, and their costs?

GO DEEPER

BOOK Operating Systems: Three Easy Pieces, chapter 17: free-space management
Arpaci-Dusseau & Arpaci-Dusseau

PAPER A scalable concurrent malloc implementation for FreeBSD Jason Evans, 2006

BOOK The Garbage Collection Handbook Jones, Hosking & Moss

PART V

Files and I/O

Three chapters on data that outlives a process. The first covers how a file system turns numbered disk blocks into named files and directories, through inodes. The second covers how it keeps that structure consistent when the power fails halfway through an update, and what `fsync` really promises. The third covers the ways a program can wait for I/O, from a blocking read to `epoll` and `io_uring`.

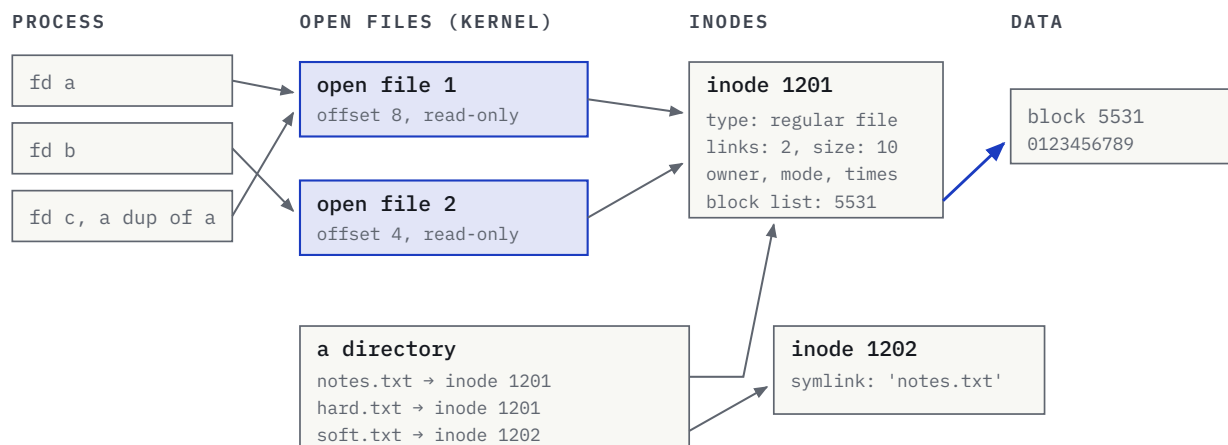
Files and file systems

3 exercises

A disk offers numbered blocks. A file system turns them into named files that grow and shrink, organised in directories, with owners and permissions. Its central idea is the *inode*, a record of a file's metadata and the location of its data, with the file's names kept separately in directories. This chapter traces `open`, `read`, `link` and `unlink` through real system calls and shows the three layers between a process and its data. It explains how an inode finds a file's blocks, why deleting a file frees no space while the file is open, and what extents and sparse files are.

Three layers

A **file descriptor** is a small integer, an index into the process's own table of open files. Each entry points to an **open file description** in the kernel, which holds the current offset and the mode the file was opened with. Each open file description points to an **inode**, the file itself. Two calls to `open` on the same file create two open file descriptions, each with its own offset. `dup` creates a second descriptor for the same description, and so does `fork`, for every descriptor the child inherits. Those share one offset, so a read through one moves the other.



Descriptors, open files, inodes and names, as they stand after the example's first three reads and its two links. The offset lives in the middle layer, which is why `dup` and `fork` share it and a second `open` doesn't. Names live in directories, not in the inode.

The example opens one file twice and duplicates one of the descriptors. Reading 4 bytes through each gives "0123" twice, from the two independent offsets, and then "4567" through the duplicate, which shares the first offset. A forked child reads one more byte through the

inherited descriptor, and the parent's next read starts after it.

EXAMPLE

files.py · descriptors and shared offsets, hard and symbolic links, unlinking an open file, a sparse file, and how far an inode reaches

```
import os, sys, tempfile

os.chdir(tempfile.mkdtemp())
with open("notes.txt", "w") as f: f.write("0123456789")

# 1. A file descriptor is an index into the process's table of open files; each open has an offset
a = os.open("notes.txt", os.O_RDONLY)
b = os.open("notes.txt", os.O_RDONLY)          # a second, independent open
c = os.dup(a)                                  # a second descriptor for the SAME open
print(f"read 4 via a: {os.read(a, 4)}, then 4 via b: {os.read(b, 4)}, then 4 via c: {os.read(c, 4)}")
sys.stdout.flush()
if os.fork() == 0:                             # a child inherits the parent's open files
    os.read(a, 1); os._exit(0)                  # ...and moves the shared offset
os.wait()
print(f"after a child read 1 byte through a, the parent's next read via a gets {os.read(a, 4)}")
for fd in (a, b, c): os.close(fd)

# 2. Names are just directory entries pointing at an inode, which holds the data and metadata
os.link("notes.txt", "hard.txt")                # a second name for the same inode
os.symlink("notes.txt", "soft.txt")             # a small file containing a path
n, h, s = os.stat("notes.txt"), os.stat("hard.txt"), os.lstat("soft.txt")
print(f"\nhard link: same inode {n.st_ino == h.st_ino}, link count {n.st_nlink}")
print(f"symbolic link: same inode {n.st_ino == s.st_ino}, its own content is the path "
      f"{os.readlink('soft.txt')!r}")
os.unlink("notes.txt")                         # removes one name; the inode survives
print(f"after unlinking notes.txt: hard.txt still reads {open('hard.txt').read()!r} "
      f"(link count {os.stat('hard.txt').st_nlink})")
print(f"soft.txt now dangles: exists {os.path.exists('soft.txt')}, the link itself exists "
      f"{os.path.lexists('soft.txt')}")

# 3. An open file outlives its last name: the space is freed only when the last descriptor closes
fd = os.open("hard.txt", os.O_RDONLY)
os.unlink("hard.txt")
print(f"\nunlinked while open: directory now holds {sorted(os.listdir('.'))}, "
      f"but the open descriptor still reads {os.read(fd, 10)!r}")
os.close(fd)

# 4. Sizes and blocks: a sparse file reports 1 GiB but stores almost nothing
with open("sparse.bin", "wb") as f:
    f.seek(2 ** 30 - 1); f.write(b"\0")
st = os.stat("sparse.bin")
print(f"sparse.bin: size {st.st_size:,} bytes; disk space used is under 1 MiB: "
      f"{st.st_blocks * 512 < 2 ** 20}")
os.remove("sparse.bin")

# 5. The classic Unix inode: 12 direct block pointers, then single, double and triple indirect blocks
block, ptrs = 4096, 4096 // 4                  # 4 KiB blocks, 4-byte block numbers
reach = [("12 direct pointers", 12 * block), ("single indirect", ptrs * block),
         ("double indirect", ptrs ** 2 * block), ("triple indirect", ptrs ** 3 * block)]
def human(n):
    for unit in ("bytes", "KiB", "MiB", "GiB", "TiB"):
```

```

        if n < 1024: return f"{n:.4g} {unit}"
        n /= 1024
total = 0
print("\nhow far each part of an ext2-style inode reaches, with 4 KiB blocks:")
for name, size in reach:
    total += size
    print(f"    {name:20} adds {human(size):>7}, for files up to {human(total)}")

```

OUTPUT

```

read 4 via a: b'0123', then 4 via b: b'0123', then 4 via c: b'4567'
after a child read 1 byte through a, the parent's next read via a gets b'9'

hard link: same inode True, link count 2
symbolic link: same inode False, its own content is the path 'notes.txt'
after unlinking notes.txt: hard.txt still reads '0123456789' (link count 1)
soft.txt now dangles: exists False, the link itself exists True

unlinked while open: directory now holds ['soft.txt'], but the open descriptor still reads
    b'0123456789'
sparse.bin: size 1,073,741,824 bytes; disk space used is under 1 MiB: True

how far each part of an ext2-style inode reaches, with 4 KiB blocks:
12 direct pointers    adds  48 KiB, for files up to 48 KiB
single indirect       adds   4 MiB, for files up to 4.047 MiB
double indirect       adds   4 GiB, for files up to 4.004 GiB
triple indirect       adds   4 TiB, for files up to 4.004 TiB

```

Names and inodes

An inode holds everything about a file except its name: its type, size, owner, permissions, timestamps, link count, and where its data is. A **directory** is a special file whose contents map names to inode numbers. That separation explains links. A **hard link** is a second directory entry for the same inode, so after `os.link` the two names are equally real and the link count is 2; `ls -li` shows the shared inode number. A **symbolic link** is a separate small file whose content is a path, followed when the link is opened, so it can point across file systems or at nothing at all. `unlink` removes a name and decrements the link count. The inode and its data are freed only when the count reaches zero *and* no process has the file open.

That last condition explains a common incident. A disk fills up, someone deletes a huge log file, and no space comes back, because the process writing the log still has it open and keeps writing to an inode with no name. The example does the same: after unlinking the last name, the directory is empty of it, but the open descriptor still reads the whole file. `ls -lsof +L1` lists open files with no links left, and the space returns when the process closes the file or exits.

Finding the blocks

The classic Unix inode, used by ext2 and ext3, holds 12 direct block pointers, enough for the first 48 KiB of a file in 4 KiB blocks. After them come a single indirect pointer, to a block full of 1,024 further pointers, which adds 4 MiB; a double indirect pointer, adding 4 GiB; and a triple indirect pointer, adding 4 TiB. Small files, which are most files, need no extra blocks

at all, and large files cost one extra read per level. Modern file systems, ext4, XFS, Btrfs and APFS, use **extents** instead: each entry describes a run of contiguous blocks with a start and a length, so a large file stored contiguously needs a handful of entries instead of millions of pointers.

Because an inode records where data is, not merely how big the file is, a file can have **holes**. The example seeks to just before 1 GiB and writes one byte. The file's size is 1 GiB, but the file system stores only the block that was written, and reads of the hole return zeros. Virtual machine images and database files are often sparse, which is why `ls -l` and `du` can disagree by gigabytes.

Layout, caching and the VFS

Where blocks go on the disk matters. The Berkeley Fast File System, in 1984, divided the disk into groups and placed each file's inode near its data and the files of a directory near each other, which multiplied throughput on spinning disks; ext4's block groups follow the same idea. Reads and writes don't go straight to the disk anyway. They go through the **page cache**, the kernel's cache of file data in otherwise unused memory: a read of cached data is a memory copy, and a write normally just updates the cache and marks the page dirty, to be written back seconds later. Chapter 17 is about what that delay means when the power fails.

On Linux, all of this sits behind the **virtual file system** layer, which gives ext4, XFS, Btrfs, network file systems and in-memory ones such as `tmpfs` the same `open`, `read` and `write`. It's also how files that aren't on any disk work: `/proc` and `/sys` generate their contents when read, which is what "everything is a file" means in practice.

COMMON TRAPS

Deleting a file a process still has open and expecting the space back.

Running out of inodes with free blocks to spare, on a volume of millions of tiny files. Check `df -i`.

Assuming `rename` works across file systems. It fails with `EXDEV`, and a copy isn't atomic.

Two processes appending to one file through separate opens without `O_APPEND`, overwriting each other.

Checking a path and then opening it, when a symbolic link can be swapped in between (chapter 7).

EXERCISES

EXERCISE 16.1 Two writers, one log

Two processes write lines to `app.log`. What ends up in the file if each opens the file separately with `O_WRONLY`? If each opens it with `O_WRONLY | O_APPEND`? If the parent opens it and the child inherits the descriptor through `fork`?

SOLUTION

Separate opens without `O_APPEND` give two offsets, each starting at 0, so each process overwrites the other's lines, and the file ends up as a mixture of fragments. With `O_APPEND`, every write first moves that description's offset to the current end of the file, as one atomic step on a local file system, so each write lands after everything before it and lines don't overwrite each other, though lines from the two processes interleave. With an inherited descriptor, both processes share one open file description and one offset, so writes also land one after another. In both of the last two cases, a line written with one `write` call stays intact; a line that a program writes in pieces can be split by the other process's writes.

EXERCISE 16.2 The full disk that isn't

`df` says a volume is 100% full, but `du` over all its files adds up to 40%. Give the most likely cause, how to confirm it, and how to get the space back without rebooting.

SOLUTION

`du` adds up the files it can reach by name, and `df` reports blocks in use, so the gap is space held by inodes with no names: almost always files deleted while a process still had them open, typically logs deleted by hand or by a misconfigured rotation. `ls -l +L1` lists them, with the process and descriptor, and on Linux they also appear in `/proc/<pid>/fd` marked "(deleted)". To free the space, restart or signal the process so it closes and reopens its logs, which is what log rotation's post-rotate hook is for. If it can't be restarted, truncating the file through `/proc/<pid>/fd/<n>` releases the blocks while leaving the process's descriptor valid.

EXERCISE 16.3 The largest file

An ext2-style file system uses 1 KiB blocks and 4-byte block numbers. With 12 direct pointers and single, double and triple indirect ones, what's the largest file it can hold? How many block reads does it take to find the last block of that file?

SOLUTION

A 1 KiB block holds 256 four-byte pointers. Direct pointers reach 12 KiB; the single indirect adds 256 blocks, 256 KiB; the double indirect 256^2 blocks, 64 MiB; and the triple indirect 256^3 blocks, 16 GiB. The largest file is therefore just over 16 GiB. In practice the real limit can be lower, for example 4 billion blocks if block numbers are 32 bits. Finding the last block goes through the triple indirect block, then a double indirect block, then a single indirect block, three reads of pointer blocks before the data block itself, unless they're already in the cache. With extents, a large contiguous file needs only a few entries, so finding any block is one lookup.

CHECKLIST

- ☐ Can you explain descriptors, open file descriptions and inodes, and which one holds the offset?
- ☐ Can you explain hard links, symbolic links and the link count?
- ☐ Can you say when unlinking a file actually frees its space?
- ☐ Can you work out how far direct and indirect pointers reach, and explain extents?
- ☐ Can you explain sparse files, the page cache and the VFS?

GO DEEPER

BOOK Operating Systems: Three Easy Pieces, chapter 39: files and directories
Arpaci-Dusseau & Arpaci-Dusseau

BOOK Operating Systems: Three Easy Pieces, chapter 40: file system
implementation Arpaci-Dusseau & Arpaci-Dusseau

PAPER A fast file system for UNIX McKusick, Joy, Leffler & Fabry, 1984

Crash consistency, journaling and fsync

3 exercises

Changing a file system means writing several blocks in different places, and the power can fail between any two of them; disks may even complete writes in a different order from the one they were issued in. This chapter tries every crash during a one-block append and classifies the damage. It shows how a journal makes the update all or nothing, and then moves up a level to applications: why overwriting a file in place can lose it, how writing a new file and renaming it fixes that, and what `fsync` does and doesn't promise.

One append, three writes

Appending one block to a file touches three structures on disk. The **bitmap** of free blocks must mark the new block used. The file's **inode** must record the new size and point to the block. And the **data block** itself must be written. They live in different places, so they're separate writes, and nothing guarantees that all three complete, or in which order. The example tries all eight combinations of which writes reached the disk before a crash, and asks what a file-system checker would conclude about each.

EXAMPLE

crashes.py · every crash during an append, with and without a journal, then saving a file safely with `fsync` and `rename`

```
import itertools, os, tempfile

# Appending one block to a file needs three writes to different places on the disk. The disk
# may
# complete them in any order, and the power can fail at any moment. Try every combination.
OLD = {"bitmap": {4}, "inode": {"size": 1, "blocks": [4]}, "block5": "????"} # before the
append
WRITES = {"bitmap": {4, 5}, "inode": {"size": 2, "blocks": [4, 5]}, "block5": "new data"}

def fsck(disk):
    """What a file-system checker would conclude about this disk."""
    pointed = set(disk["inode"]["blocks"]); used = disk["bitmap"]
    if pointed - used: return "INCONSISTENT: the inode points at a block marked free"
    if used - pointed: return "space leak: a block marked used belongs to no file"
    if 5 in pointed and disk["block5"] == "????": return "consistent, but the file now ends in
        garbage"
    if 5 in pointed: return "consistent: the append happened"
    return "consistent: as if the append never happened"

print("crash during an append, without a journal; which writes reached the disk:")
for k in range(4):
    for done in itertools.combinations(WRITES, k):
        disk = {**OLD, **{w: WRITES[w] for w in done}}
        print(f"    {k} + '.join(done) or 'none':24} {fsck(disk)}")
```

```

# With a journal: write the three blocks, then a commit record, to the log; then write them
# home.
# Recovery replays a transaction only if its commit record reached the disk. (A write barrier
# keeps the commit from landing before the blocks it covers.)
STEPS = ["log bitmap", "log inode", "log block5", "log commit",
         "home bitmap", "home inode", "home block5", "free the log"]
def journaled(crash_after):
    disk, log = {**OLD}, {}
    for step in STEPS[:crash_after]:
        kind, what = step.split(" ", 1)
        if kind == "log": log[what] = True
        elif kind == "home": disk[what] = WRITES[what]
        else: log = {}
    if log.get("commit"): disk = {**disk, **WRITES}          # recovery after reboot: replay
                                                                the log
    return fsck(disk)
print("\nwith a journal, crash after each step:")
for n in range(9):
    print(f"    {'(before any write)' if n == 0 else STEPS[n - 1]:24} {journaled(n)}")

# The same problem one level up: replacing a file's contents safely
def overwrite_in_place(crash_at):                             # open(path, "w") truncates first, then
    writes                                                    writes
    states = ["old contents", "", "new con", "new contents"]
    return states[min(crash_at, 3)]
def write_then_rename(crash_at):                             # write a temporary file, fsync it, rename
    over                                                     over
    return "new contents" if crash_at >= 3 else "old contents"
print("\nsaving a file, crash at every point:")
for name, save in (("overwrite in place:", overwrite_in_place),
                   ("write, fsync, then rename:", write_then_rename)):
    print(f"    {name:27} " + ", ".join(repr(x) for x in dict.fromkeys(save(c) for c in
                                range(4))))

# The rename pattern, for real
def atomic_write(path, data):
    tmp = path + ".tmp"
    with open(tmp, "wb") as f:
        f.write(data); f.flush()
        os.fsync(f.fileno())                                # the data is on the disk before the
                                                                rename
    os.replace(tmp, path)                                     # atomic: readers see the old file or the
                                                                new one
    dir_fd = os.open(os.path.dirname(path), os.O_RDONLY)
    os.fsync(dir_fd); os.close(dir_fd)                       # make the rename itself durable
path = os.path.join(tempfile.mkdtemp(), "settings.json")
atomic_write(path, b'{"theme": "dark"}')
atomic_write(path, b'{"theme": "light"}')
print(f"    after two atomic writes the file holds {open(path).read()}; "
      f"the directory holds {os.listdir(os.path.dirname(path))}")

```

OUTPUT

```

crash during an append, without a journal; which writes reached the disk:
none          consistent: as if the append never happened
bitmap        space leak: a block marked used belongs to no file
inode         INCONSISTENT: the inode points at a block marked free
block5        consistent: as if the append never happened
bitmap + inode consistent, but the file now ends in garbage
bitmap + block5 space leak: a block marked used belongs to no file
inode + block5 INCONSISTENT: the inode points at a block marked free
bitmap + inode + block5 consistent: the append happened

with a journal, crash after each step:
(before any write) consistent: as if the append never happened
log bitmap        consistent: as if the append never happened

```

log inode	consistent: as if the append never happened
log block5	consistent: as if the append never happened
log commit	consistent: the append happened
home bitmap	consistent: the append happened
home inode	consistent: the append happened
home block5	consistent: the append happened
free the log	consistent: the append happened

saving a file, crash at every point:

```

overwrite in place:      'old contents', '', 'new con', 'new contents'
write, fsync, then rename: 'old contents', 'new contents'
after two atomic writes the file holds {"theme": "light"}; the directory holds
    ['settings.json']

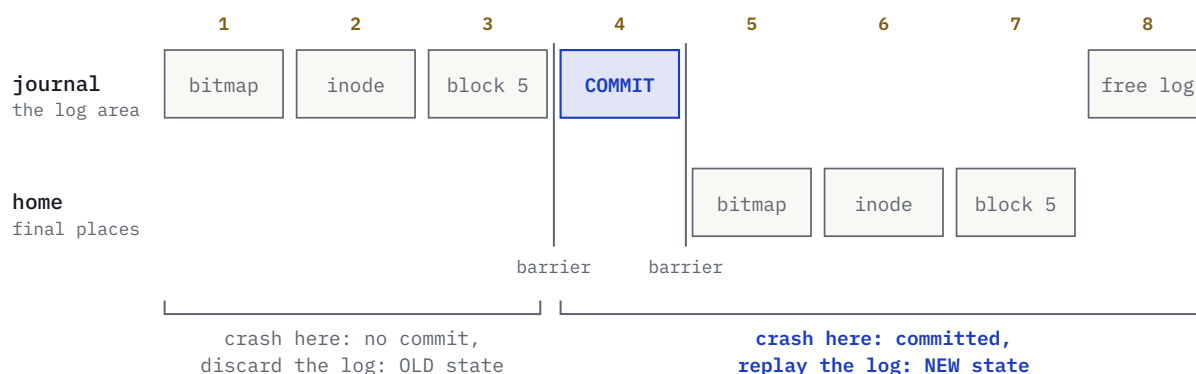
```

Only three outcomes are clean: all three writes, none of them, or only the data block, written into a block nobody points to. A bitmap update alone leaks a block that belongs to no file. An inode update without the bitmap points at a block the bitmap says is free, so a later allocation can give the same block to a second file, and the two files then overwrite each other. The bitmap and inode without the data leave a consistent file system with garbage at the end of the file: whatever was in that block before, possibly another user's deleted data. No checker can detect that last case, because the metadata is perfectly consistent.

Early Unix systems handled crashes by running `fsck` at boot: read every inode and bitmap, cross-check them, and repair what's inconsistent. It works, but it reads the whole file system, which on a large disk takes hours, and it can't recover data that was never written.

Journaling

A **journal**, or write-ahead log, makes the update atomic. Before touching the real structures, the file system writes everything it's about to change into a reserved log area, followed by a **commit** record. Only once the commit is on disk does it write the blocks to their home locations, the **checkpoint**, and finally it frees the log space. After a crash, recovery reads the log. A transaction with a commit record is replayed, which is safe to do twice, since replaying writes the same blocks again; a transaction without one is discarded. Either way the file system ends up entirely before or entirely after the update, which is what the example's second table shows for every crash point.



Journalling an append. The commit record is the single write that decides the outcome, and the barriers make sure it reaches the disk after the blocks it covers and before any of them is written home.

The order is enforced with **write barriers**, or cache flushes, because a disk with a write cache might otherwise write the commit before the blocks it describes. ext3, ext4, XFS and NTFS all journal. Writing every data block twice, once to the log and once home, halves write bandwidth, so ext4's default **ordered** mode journals only metadata, and writes a file's data blocks to their home locations before committing the metadata that points to them. That still rules out pointers to garbage. Copy-on-write file systems, ZFS, Btrfs and APFS, take another route: they never overwrite live blocks, write new versions elsewhere, and switch to them with a single atomic update of a root pointer.

What fsync promises

A `write` that returns has only copied the data into the page cache. Linux writes dirty pages back on its own schedule, typically within 30 seconds, and a crash before then loses them however consistent the file system stays. `fsync(fd)` blocks until the file's data and metadata are on stable storage, including telling the device to flush its own volatile cache; `fdatasync` skips metadata that isn't needed to read the data back, such as timestamps. On macOS, `fsync` does not force the drive to flush its cache, and durability across a power cut needs `fctl(fd, F_FULLFSYNC)`, which SQLite and other databases use there.

`fsync` is also where errors surface. In 2018 PostgreSQL's developers discovered that when Linux failed to write back dirty pages, `fsync` reported the error once and then marked the pages clean, so a retry would succeed and the data was simply gone. Since then PostgreSQL treats an `fsync` failure as fatal and recovers from its own log. The lesson for every program is that an `fsync` error means the data may be lost, not that it should try again. And `fsync` is slow: from around a hundred microseconds on a server SSD that can protect its cache against power loss, to several milliseconds on consumer drives and ten on spinning disks. That's why databases **group commit**, making many transactions durable with one `fsync` (exercise 17.2).

Saving a file safely

The same problem appears one level up. Opening a file with `"w"` truncates it first and then writes the new contents, so a crash in between leaves the old contents, an empty file, a partial one or the new one; the example lists all four. The safe pattern writes the new contents to a temporary file in the same directory, calls `fsync` on it, then uses `rename` to replace the original, and finally calls `fsync` on the directory so that the rename itself is durable. POSIX guarantees that `rename` replaces the target atomically, so every reader and every crash sees either the complete old file or the complete new one. The example's `atomic_write` does exactly this. When ext4 introduced delayed allocation in 2009, applications that skipped the `fsync` began leaving zero-length configuration files after crashes, and ext4 added a heuristic to protect them. Code that wants its data to survive shouldn't depend on heuristics.

COMMON TRAPS

Treating a successful `write` as durable. Until `fsync` returns, it's in memory.

Overwriting important files in place. Write a new file and rename it over the old one.

Forgetting to `fsync` the directory after creating or renaming a file.

Retrying a failed `fsync` and believing the second success.

One `fsync` per record in a hot path, instead of batching.

Relying on plain `fsync` on macOS, or on drives and RAID controllers that ignore flushes.

EXERCISES

EXERCISE 17.1 What fsck can and can't fix

For each damaged outcome in the example's first table, say whether `fsck` can repair it and whether any user-visible harm remains.

SOLUTION

A space leak, from the bitmap alone or the bitmap and data: `fsck` finds a block marked used that no inode points to and marks it free. The append is lost, but nothing else is harmed. An inode pointing at a block marked free: `fsck` can mark the block used, which prevents double allocation, but the block's contents were never written, so the file now ends in whatever the block held before, possibly another file's old data. Bitmap and inode without the data: `fsck` sees nothing wrong at all, and the file silently ends in garbage. So `fsck` restores the file

system's consistency, but it can't restore what the user wrote, and it can't even notice the worst case. Journaling, or ordered mode, prevents these states instead of repairing them.

EXERCISE 17.2 Group commit

A database must make every committed transaction durable, and an `fsync` of its log takes 2 ms. At most how many transactions a second can it commit with one `fsync` each? How does group commit raise that, and what does it cost each transaction?

SOLUTION

With one `fsync` per transaction, run one after another, the limit is $1 / 2 \text{ ms} = 500$ commits a second, however fast the processor is. With group commit, transactions append their log records to a shared buffer, and one `fsync` makes everything in the buffer durable. All the transactions that committed during the previous `fsync` share the next one, so at 5,000 commits a second each `fsync` covers about 10 transactions, and 500 `fsync`s a second suffice. The cost is latency: a transaction may wait for the `fsync` in progress to finish before its own starts, up to about two `fsync` times, 4 ms. Some systems also wait deliberately for a fraction of a millisecond to gather more commits into each group.

EXERCISE 17.3 Why fsync the directory?

A program writes a brand-new file `report.csv`, calls `fsync` on it and closes it. The power fails a second later, and after reboot the file doesn't exist. How is that possible, and what was missing?

SOLUTION

`fsync` on the file made its inode and data durable, but the file's name lives in a different file, the directory, which has its own dirty block in the page cache. If the crash came before that block was written back, the inode survived with no name pointing to it; after recovery it's an orphan, which `fsck` may move to `lost+found` or free. To make a new name durable, open the directory and call `fsync` on it after creating or renaming the file, as the example's `atomic_write` does. Some file systems happen to make this unnecessary in some modes, but portable code can't rely on that.

CHECKLIST

- ☐ Can you list the writes an append needs and what each partial crash leaves behind?
- ☐ Can you explain how a journal and its commit record make an update atomic?
- ☐ Can you explain ordered mode and copy-on-write file systems?
- ☐ Can you say exactly what `write`, `fsync` and `fdatasync` guarantee, including on macOS?
- ☐ Can you write the temporary-file, `fsync`, rename and directory-`fsync` pattern?

GO DEEPER

BOOK Operating Systems: Three Easy Pieces, chapter 42: crash consistency
Arpaci-Dusseau & Arpaci-Dusseau

ARTICLE Files are hard Dan Luu

PAPER All file systems are not created equal Pillai et al., OSDI 2014

I/O models: blocking, multiplexed and asynchronous

3 exercises

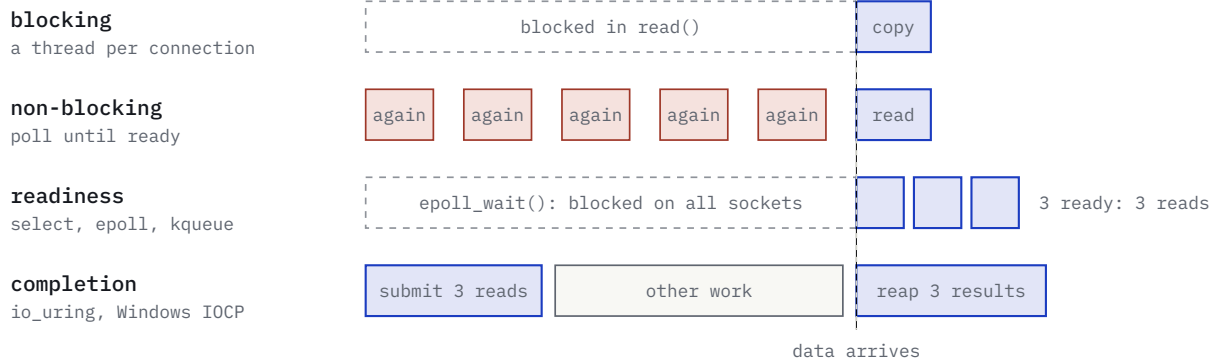
A read from a socket or a disk may have to wait, and how a program waits decides how many connections one thread can serve and how much processor time goes on waiting. This chapter compares four models: blocking calls, non-blocking calls with polling, readiness notification through `select`, `epoll` and `kqueue`, and completion-based I/O with `io_uring`. It runs a single-threaded server that handles a hundred connections by asking the kernel which ones are ready, and explains why regular files don't fit the readiness model.

Blocking

By default, a `read` on a socket with no data waiting puts the calling thread to sleep until data arrives. That's the simplest model to program: the code reads a request, handles it and writes a reply, in order, as if nothing else existed. To serve many clients, give each connection its own thread or process, which is how Apache's original servers and PostgreSQL work. Around 1999 Dan Kegel named the limit of that design the **C10K problem**: serving ten thousand connections at once with a thread each costs a great deal of memory, and every one of them needs scheduling. Most of those threads are idle, waiting for a slow client, and the waiting itself is what costs.

Non-blocking, and readiness

A descriptor in **non-blocking** mode, set with `O_NONBLOCK`, never sleeps: a read with nothing to read fails at once with `EAGAIN`, "try again". On its own that's only useful for checking, since a program that keeps retrying burns a core. Combined with a way to learn when a descriptor is ready, it becomes the basis of every high-performance server. `select`, from 4.2BSD in 1983, takes sets of descriptors and blocks until at least one is readable or writable. `poll` lifted `select`'s limit of 1,024 descriptors, but both still pass the whole set to the kernel on every call and scan it, so their cost grows with the number of connections, not with the number that are active. Linux's `epoll`, from 2002, and FreeBSD's `kqueue`, from 2000 and also used by macOS, fix that: a program registers each descriptor once, and each wait returns only the ones that are ready.



The four I/O models. Blocking ties up a thread per wait; polling wastes cycles; readiness lets one thread wait on everything at once; completion lets it keep working while the kernel does the I/O.

The example uses Python's `selectors` module, which picks `epoll` on Linux and `kqueue` on macOS. A non-blocking read on an empty socket raises `BlockingIOError`, Python's name for `EAGAIN`. Then a hundred socket pairs stand in for a hundred client connections. When ten clients send a request, one call reports exactly those ten as ready and one thread answers them; when all hundred send, one call reports a hundred. A client that hangs up is reported as ready too, and its read returns an empty result, end-of-file, which tells the server to close and unregister it.

EXAMPLE

iomodels.py · non-blocking reads, one thread serving a hundred connections through `epoll` or `kqueue`, a hang-up, and the memory per idle connection

```
import selectors, socket

# 1. Blocking and non-blocking reads. A socketpair is two connected sockets in one process.
client, server = socket.socketpair()
server.setblocking(False)
try:
    server.recv(100)
except BlockingIOError as e:
    print(f"non-blocking recv with no data: {type(e).__name__} (EAGAIN): try again later")
client.send(b"ping")
print(f"after the client sends: recv returns {server.recv(100)!r}")
client.close(); server.close()

# 2. One thread, many connections: ask the kernel which sockets are ready, then serve only those.
# DefaultSelector uses epoll on Linux and kqueue on macOS and the BSDs.
sel = selectors.DefaultSelector()
pairs = [socket.socketpair() for _ in range(100)]
for i, (c, s) in enumerate(pairs):
    s.setblocking(False)
    sel.register(s, selectors.EVENT_READ, data=i)

def serve_once():
    ready = sel.select(timeout=0)
    for key, _ in ready:
```

```

        msg = key.fileobj.recv(100)
        key.fileobj.send(msg.upper())           # small replies: the send buffer has room
    return len(ready)

for c, _ in pairs[:10]: c.send(b"hello")       # only 10 clients have said anything
print(f"\n100 connections, 10 clients sent a request: select() reports {serve_once()} ready")
got = [c.recv(100) for c, _ in pairs[:10]]
print(f"    replies: {len(got)} x {got[0]!r}")
print(f"    nothing pending now: select() reports {serve_once()} ready")
for i, (c, _) in enumerate(pairs): c.send(f"req {i}".encode())
print(f"all 100 clients send: select() reports {serve_once()} ready, all served by one thread")
replies = [c.recv(100) for c, _ in pairs]
print(f"    replies received: {len(replies)}, first {replies[0]!r}, last {replies[-1]!r}")

# 3. A closed connection is "ready" too: the read returns b"" (end of file)
c, s = pairs[0]
c.close()
for key, _ in sel.select(timeout=0):
    data = key.fileobj.recv(100)
    print(f"\nclient {key.data} hung up: select() reports it ready, and recv returns {data!r}")
    sel.unregister(key.fileobj); key.fileobj.close()
for c, s in pairs[1:]: c.close(); s.close()
sel.close()

# 4. Why not a thread per connection? Rough costs per idle connection.
print()
for n in (1_000, 10_000, 100_000):
    threads = n * 64 * 1024                    # ~64 KiB of stack and kernel state
        actually used
    loop = n * 4 * 1024                        # ~4 KiB of buffers and bookkeeping in an
        event loop
    print(f"{n:>7,} idle connections: thread per connection ~{threads / 2 ** 20:6,.0f} MiB, "
          f"event loop ~{loop / 2 ** 20:5,.0f} MiB")

```

OUTPUT

```

non-blocking recv with no data: BlockingIOError (EAGAIN): try again later
after the client sends: recv returns b'ping'

```

```

100 connections, 10 clients sent a request: select() reports 10 ready
replies: 10 x b'HELLO'
nothing pending now: select() reports 0 ready
all 100 clients send: select() reports 100 ready, all served by one thread
replies received: 100, first b'REQ 0', last b'REQ 99'

```

```

client 0 hung up: select() reports it ready, and recv returns b''

```

```

1,000 idle connections: thread per connection ~    62 MiB, event loop ~    4 MiB
10,000 idle connections: thread per connection ~   625 MiB, event loop ~   39 MiB
100,000 idle connections: thread per connection ~ 6,250 MiB, event loop ~  391 MiB

```

The details that bite

Readiness notification comes in two styles. **Level-triggered**, the default, reports a descriptor every time you ask, as long as it's readable. **Edge-triggered**, an option in `epoll`, reports it only when it becomes readable, which saves wake-ups but requires reading until `EAGAIN` every time, or the rest of the data sits unnoticed (exercise 18.1). Writes need the same care as reads: a send on a non-blocking socket can accept only part of the data when the buffer is full, so the server keeps the rest and asks to be told when the socket is writable

again. It should ask only while it has something to write, because an idle socket is almost always writable, and waiting for writability on every connection turns the event loop into a busy loop.

Regular files don't fit this model at all. A file is always reported readable, because the kernel's notion of readiness doesn't cover waiting for the disk, so a read can still block on a page-cache miss (chapter 13). Event-loop runtimes such as libuv, behind Node.js, therefore send file operations to a small pool of threads.

Completion: `io_uring`

Readiness tells a program when an operation won't block, and the program then makes the system call itself. **Completion-based** I/O inverts that: the program describes the operation, the kernel performs it, and the program collects the result later. Windows has worked this way since its I/O completion ports in the 1990s. Linux's `io_uring`, by Jens Axboe, added in 2019, uses two rings in memory shared between process and kernel: the program writes requests into the submission ring and reads results from the completion ring, so a single system call can submit and collect hundreds of operations, and in polling mode none at all. It handles files and sockets alike, which solves the file problem above. It has also had a long series of security flaws, and some environments, including Docker's default security profile and Google's production servers, disable it.

Choosing

The example's last lines put rough numbers on the trade-off. An idle connection costs perhaps 64 KiB of stack and kernel state with a thread each, and a few kilobytes of buffers with an event loop, so at 100,000 connections the difference is about 6 GiB against a few hundred MiB. Under load, readiness-based servers also avoid most context switches. Few programs use these interfaces directly, though, because language runtimes build on them. Go's network poller runs goroutines over `epoll` or `kqueue`, so code that looks blocking is multiplexed underneath. Java has NIO and Netty, Rust has tokio, and Python has `asyncio`. Knowing what's underneath explains their behaviour: why a blocking call inside them hurts (chapter 11), why file I/O goes to a thread pool, and why a server can hold far more connections than it has threads.

COMMON TRAPS

A blocking call inside a non-blocking server, which stalls every connection.

Edge-triggered `epoll` without reading to `EAGAIN`, leaving data unseen.

Asking for writability on every socket all the time, which spins at 100% CPU.

Using `select` with descriptor numbers of 1,024 or more, which overflows its fixed-size sets.

Assuming file reads never block because the file is "ready".

Closing a descriptor without unregistering it, when the number may be reused for a new connection.

EXERCISES

EXERCISE 18.1 Level and edge

A socket receives 10 KB at once. The handler reads 4 KB per event. What happens under level-triggered notification, and under edge-triggered notification?

SOLUTION

Level-triggered: after the first 4 KB read, the socket is still readable, so the next wait reports it again, and the handler gets all 10 KB in three events. It costs extra wake-ups but nothing is lost. Edge-triggered: the event fired once, when the socket went from empty to readable. After the handler reads 4 KB, 6 KB remain, but the socket hasn't become readable again, it simply stayed readable, so no new event comes. The 6 KB sits there until the client sends more, which may be never if it's waiting for a reply. With edge-triggered notification the handler must read in a loop until it gets `EAGAIN`.

EXERCISE 18.2 Why not select?

A server holds 50,000 mostly idle connections, of which about 20 have data at any moment. Compare the work of one `select` call and one `epoll_wait` call.

SOLUTION

`select` can't even do it with the standard fixed set size of 1,024 descriptors. `poll`, which has no limit, copies an array of 50,000 entries into the kernel on every call, the kernel checks all 50,000, and the program scans all 50,000 results to find the 20 that are ready: work proportional to the number of connections, repeated on every call, thousands of times a second. `epoll_wait` copies nothing in, because the interest list was registered once, and returns only the 20 ready entries: work proportional to the number of active connections. That difference is most of why one machine can now hold hundreds of thousands of connections.

EXERCISE 18.3 The slow client

A single-threaded event-loop server writes a 1 MB response to a client on a slow mobile connection. The socket's send buffer holds 256 KB. What happens if the server's write blocks? What should a non-blocking server do instead?

SOLUTION

A blocking write sleeps until the kernel has accepted the whole megabyte, which means until the slow client has drained most of it, possibly many seconds. The loop's only thread is stuck in that write, so every other connection waits. A non-blocking server writes what fits, 256 KB at first, keeps the remaining 768 KB in a per-connection buffer, asks to be told when the socket becomes writable, and goes back to serving others. Each writable event sends a little more, and when the buffer is empty it stops asking. It should also cap how much output it will buffer per connection, or many slow clients could exhaust its memory.

CHECKLIST

- ☐ Can you describe blocking, non-blocking, readiness and completion I/O?
- ☐ Can you explain why `epoll` and `kqueue` scale better than `select` and `poll`?
- ☐ Can you handle partial writes, hang-ups and edge-triggered events correctly?
- ☐ Can you explain why regular files don't fit the readiness model?
- ☐ Can you explain how `io_uring` works, and where language runtimes use these interfaces?

GO DEEPER

ARTICLE [The C10K problem](#) Dan Kegel

REFERENCE [epoll\(7\)](#) The Linux man-pages project

GUIDE [Lord of the io_uring](#) Shuveb Hussain

PART VI

Between processes

Two chapters on the boundaries between processes. The first covers how isolated processes deliberately talk to each other, through pipes, sockets, shared memory and the descriptors they can hand to each other. The second covers how the same isolation is pushed further, into virtual machines and containers, which are built from the mechanisms of every earlier chapter.

Interprocess communication

3 exercises

Processes share no memory by default, which is the point of having them. When they need to cooperate, the kernel offers several ways to talk, and this chapter runs most of them for real. It covers pipes and named pipes; stream and datagram sockets, and the framing a byte stream needs; shared memory, and the difference between shared and private mappings; and passing an open file descriptor from one process to another. It ends with how to choose between them.

Pipes and named pipes

A pipe (chapter 3) is a one-way byte stream through a kernel buffer, and a read returns end-of-file once every writer has closed its end. An anonymous pipe exists only as two descriptors, so it can connect only processes that inherited them from a common ancestor. A **named pipe**, or FIFO, created with `mkfifo`, has a name in the file system, so unrelated processes can find it, and permissions decide who may open it. Opening a FIFO blocks until the other end is opened too, which makes it a simple rendezvous between two programs.

Sockets

Unix domain sockets use the same API as network sockets, `socket`, `bind`, `connect`, `send` and `recv`, with an address that's a path in the file system instead of a host and port. They're bidirectional, faster than TCP over the loopback interface because they skip the network stack, and protected by file permissions; a server can also ask the kernel for the user ID of the process at the other end. The Docker daemon, PostgreSQL's local connections, systemd and the display servers of Linux desktops all listen on them.

A socket comes in kinds. A **stream** socket, `SOCK_STREAM`, carries bytes with no message boundaries: three sends can arrive as one read, as in the example, or one send as three reads. A **datagram** socket, `SOCK_DGRAM`, keeps each send as a separate message; for Unix sockets it's reliable and ordered, unlike UDP. A protocol on a stream must therefore mark where each message ends, which is **framing**. The example prefixes each message with its length in four bytes and reads exactly that many back, the method used by many binary protocols; text protocols such as HTTP and Redis's use delimiters instead (exercise 19.1).

EXAMPLE

`ipc.py` · a pipe, a FIFO, stream and datagram sockets and framing, shared and private mappings, and passing a file descriptor

```
import mmap, os, socket, struct, sys, tempfile

def child(fn):
```

```

    """Run fn in a forked child process and wait for it to finish."""
    sys.stdout.flush()
    pid = os.fork()
    if pid == 0:
        fn(); os._exit(0)
    os.waitpid(pid, 0)

# 1. A pipe: a one-way byte stream between related processes. Reading returns b"" at end of
    file.
    r, w = os.pipe()
    child(lambda: (os.close(r), os.write(w, b"rows 1-500 done\n"), os.write(w, b"rows 501-1000
        done\n")))
    os.close(w)
    chunks = []
    while (data := os.read(r, 4096)): chunks.append(data)
    print(f"pipe: the parent read {b''.join(chunks)!r}, then end of file")

# 2. A named pipe (FIFO) has a name in the file system, so unrelated processes can find it
    fifo = os.path.join(tempfile.mkdtemp(), "jobs.fifo")
    os.mkfifo(fifo)
    sys.stdout.flush()
    if os.fork() == 0:
        with open(fifo, "wb") as f: f.write(b"job 42")                # opening blocks until a reader
            arrives
        os._exit(0)
    with open(fifo, "rb") as f: print(f"FIFO at {os.path.basename(fifo)}: read {f.read()!r}")
    os.wait()

# 3. Streams have no message boundaries; datagram sockets keep them
    for kind, name in ((socket.SOCK_STREAM, "stream socket:"), (socket.SOCK_DGRAM, "datagram
        socket:")):
        a, b = socket.socketpair(socket.AF_UNIX, kind)
        for msg in (b"first", b"second", b"third"): a.send(msg)
        b.setblocking(False)
        got = []
        try:
            while True: got.append(b.recv(1024))                # read until nothing is left
        except BlockingIOError: pass
        print(f"{name:18} 3 sends arrive as {len(got)} read{'s' if len(got) > 1 else ''}: {got}")
        a.close(); b.close()
    def send_msg(sock, data): sock.sendall(struct.pack("!I", len(data)) + data)    # length-prefix
        framing
    def recv_msg(sock):
        size = struct.unpack("!I", sock.recv(4, socket.MSG_WAITALL))[0]
        return sock.recv(size, socket.MSG_WAITALL)
    a, b = socket.socketpair()
    for msg in (b"first", b"second", b"third"): send_msg(a, msg)
    print(f"{'stream + framing:':18} 3 sends arrive as 3 messages: {[recv_msg(b) for _ in
        range(3)]}")
    a.close(); b.close()

# 4. Shared memory: a MAP_SHARED mapping is the same physical pages in both processes
    shared = mmap.mmap(-1, 4096, flags=mmap.MAP_SHARED)
    private = mmap.mmap(-1, 4096, flags=mmap.MAP_PRIVATE)
    child(lambda: (shared.write(b"written by the child"), private.write(b"written by the child")))
    print(f"\nMAP_SHARED after the child wrote: {shared[:20]!r}")
    print(f"MAP_PRIVATE after the child wrote: {private[:20].rstrip(bytes(1))!r} "
        f"(nothing: the child wrote to its own copy-on-write copy)")

# 5. Passing an open file descriptor over a Unix socket: the receiver gets the open file itself
    secret = os.path.join(tempfile.mkdtemp(), "only-the-child-opens-this.txt")
    with open(secret, "w") as f: f.write("token 7f3a")
    parent_sock, child_sock = socket.socketpair(socket.AF_UNIX, socket.SOCK_STREAM)
    def give_fd():
        fd = os.open(secret, os.O_RDONLY)
        socket.send_fds(child_sock, [b"here"], [fd])

```

```

child(give_fd)
msg, fds, _, _ = socket.recv_fds(parent_sock, 100, 1)
print(f"\nreceived {msg!r} and a descriptor; reading it gives {os.read(fds[0], 100)!r}")

```

OUTPUT

```

pipe: the parent read b'rows 1-500 done\nrows 501-1000 done\n', then end of file
FIFO at jobs.fifo: read b'job 42'
stream socket:      3 sends arrive as 1 read: [b'firstsecondthird']
datagram socket:    3 sends arrive as 3 reads: [b'first', b'second', b'third']
stream + framing:   3 sends arrive as 3 messages: [b'first', b'second', b'third']

MAP_SHARED after the child wrote: b'written by the child'
MAP_PRIVATE after the child wrote: b'' (nothing: the child wrote to its own copy-on-write
copy)

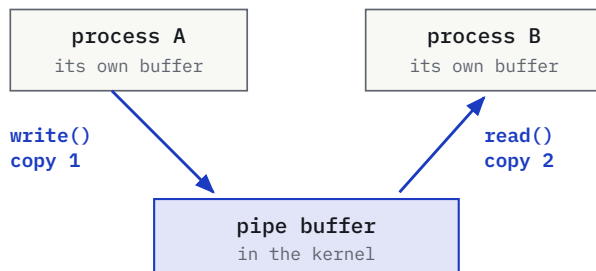
received b'here' and a descriptor; reading it gives b'token 7f3a'

```

Shared memory

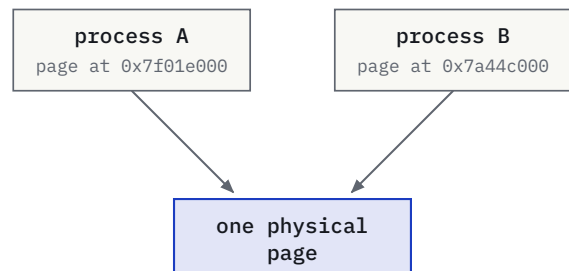
Every message through a pipe or socket is copied twice: from the sender into the kernel, and from the kernel into the receiver. **Shared memory** avoids both copies by mapping the same physical pages into both processes. A mapping created with `mmap` and `MAP_SHARED` before a `fork` is shared with the child; unrelated processes can share a named object from `shm_open` or a file mapped by both. In the example, the child writes into a shared mapping and the parent reads the child's words straight out of memory, while the same write to a `MAP_PRIVATE` mapping lands in the child's copy-on-write copy (chapter 13) and the parent sees nothing.

A PIPE OR SOCKET: TWO COPIES



every message crosses the kernel twice, but the kernel orders and synchronises it

SHARED MEMORY: NO COPIES



both see every write at once, with no system call; so they must synchronise with locks or atomics

Copying against sharing. Message passing costs two copies but gives you ordering and synchronisation for free; shared memory costs nothing to transfer, but every access is a potential race between processes, the subject of Part III.

Shared memory is the fastest form of communication and the most dangerous, because it brings back every race of chapters 7 to 10, now between processes. The usual answer is a structure designed for it, such as a ring buffer with atomic head and tail indices, or a mutex created as process-shared, which on Linux is a `futex` in the shared page. Databases put their

buffer caches in shared memory, as PostgreSQL does, and browsers use it to pass rendered frames between processes. Python's `multiprocessing.shared_memory` wraps the same mechanism.

Passing descriptors

A Unix socket can carry more than bytes. With an `SCM_RIGHTS` message, a process sends open file descriptors, and the receiver gets new descriptors for the same open files, sharing their offsets, as if it had inherited them. In the example the child opens a file and sends the descriptor, and the parent reads the file through it without ever opening it. The kernel checked permissions when the child opened the file, not when the parent used it, which makes this a tool for **privilege separation**: a small privileged helper opens what's needed and hands the descriptors to an unprivileged worker. It's also how servers restart without dropping connections. The old process passes its listening socket to the new one, which starts accepting while the old one finishes its current requests, and systemd's socket activation passes listening sockets to services in the same way.

Choosing

Other mechanisms fill the gaps: POSIX message queues, which keep message boundaries and priorities; the older System V IPC calls; `eventfd` for one process to wake another; and on Linux desktops, D-Bus, a message bus built on Unix sockets. In practice, the choice usually comes down to a few questions. A one-way stream between a parent and a child is a pipe. A local service with requests and replies is a Unix socket, or local TCP or HTTP if it may move to another machine one day, at a cost of tens of microseconds per request. Large volumes of data at a high rate, such as video frames or tensors, go in shared memory with a carefully designed queue. And the kernel's permission checks apply to all of them, so a socket that's writable by everyone is an open door: access to Docker's socket is equivalent to root.

COMMON TRAPS

Treating a stream socket as a stream of messages, with no framing and no handling of partial reads.

Shared memory without synchronisation, which reintroduces every race in Part III between processes.

Two processes that both write large amounts before reading, deadlocking on full buffers.

Named FIFOs, sockets and shared-memory objects left behind after a crash.

A sent descriptor not closed by the sender, which keeps the file open longer than intended.

EXERCISES

EXERCISE 19.1 Two ways to frame

Compare length-prefix framing with delimiter framing, such as one message per line. Give an advantage and a cost of each, and a protocol that uses each.

SOLUTION

A length prefix tells the receiver exactly how many bytes to read, so messages can contain any bytes and the reader never scans the data. It's compact and fast, and used by most binary protocols, such as PostgreSQL's wire protocol and gRPC. Its costs are that the sender must know the length before it starts sending, and that a corrupted length desynchronises everything after it. A delimiter, such as a newline, lets the sender stream without knowing the length, and is easy to read and debug by hand; HTTP/1.1 headers, Redis's protocol for commands, and JSON Lines use delimiters. Its cost is that the delimiter can't appear inside a message unless it's escaped, and the reader must scan every byte to find it. Many protocols mix the two: HTTP ends its headers with a blank line and gives the body's length in a header.

EXERCISE 19.2 The pipe deadlock

A parent starts a child with pipes to its standard input and output, writes 1 MB into the child's input, and then reads the child's output. The child echoes everything it reads. The program hangs. Why, and how do you fix it?

SOLUTION

The child reads input and writes output as it goes, but nobody reads its output until the parent has finished writing. The output pipe fills after 64 KiB, so the child blocks in `write` and stops reading its input. The input pipe then fills, so the parent blocks in its `write` too. Each waits for the other: a deadlock with two pipes. Fix it by writing and reading at the same time, with a thread for one direction or with readiness notification on both pipes. In Python, `subprocess.run(input=...)` and `Popen.communicate()` do exactly this, and the `subprocess` documentation warns against writing to `stdin` and reading `stdout` in sequence.

EXERCISE 19.3 Restarting without refusing anyone

Describe how a server can be replaced by a new version without refusing a single connection, using descriptor passing. What does the old process do with its existing connections?

SOLUTION

The listening socket is the key: as long as some process holds it open, the kernel keeps queueing incoming connections. The old process starts the new version and passes it the listening socket's descriptor over a Unix socket, or simply leaves it open across `fork` and `exec`. The new process starts calling `accept` on it. The old process then stops accepting, closes its copy of the listener, finishes the requests on its existing connections, and exits. Connections that arrived in between waited in the kernel's queue and are accepted by the new process, so none is refused. nginx and HAProxy reload this way. An alternative on Linux is `SO_REUSEPORT`, which lets both processes bind the same port for the overlap.

CHECKLIST

- ☐ Can you choose between a pipe, a FIFO and a Unix socket for a given pair of processes?
- ☐ Can you explain why stream sockets need framing, and implement it?
- ☐ Can you set up shared memory and say what synchronisation it then needs?
- ☐ Can you explain the difference between shared and private mappings after `fork`?
- ☐ Can you explain descriptor passing and two uses for it?

GO DEEPER

GUIDE Beej's guide to Unix interprocess communication Brian Hall

REFERENCE `unix(7)`: sockets for local interprocess communication The Linux man-pages project

BOOK The Linux Programming Interface, chapters 43 to 61: interprocess communication and sockets Michael Kerrisk

Virtual machines and containers

3 exercises

Virtual machines and containers both let many workloads share one machine as if each had it to itself, and both are built from the mechanisms of this volume. A virtual machine gives each guest its own kernel, running on virtual hardware with the processor's help. A container shares one kernel among all its guests and gives each group of processes its own view of it. This chapter explains how a hypervisor traps and emulates, and what nested paging adds to a TLB miss. It then takes a container apart: namespaces for what it sees, cgroups for what it may use, and layered images for its files. The example simulates an overlay file system and a CPU quota.

Virtual machines

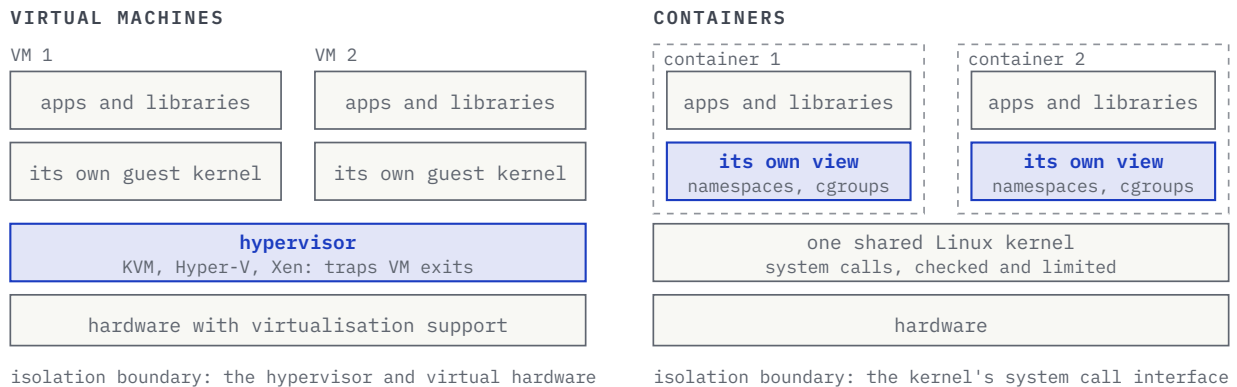
A **hypervisor**, or virtual machine monitor, runs operating systems as guests, each believing it controls real hardware. The classic technique is **trap and emulate**: run the guest kernel in user mode, so that every privileged instruction it executes traps to the hypervisor, which performs the instruction's effect on the guest's virtual state and resumes it. Gerald Popek and Robert Goldberg showed in 1974 that this works if every instruction that could reveal or change the machine's privileged state traps. For a long time x86 failed that test: a few instructions behaved differently in user mode without trapping. VMware worked around it in 1999 by rewriting guest kernel code on the fly, and Xen in 2003 by modifying guest kernels to call the hypervisor directly, which is **paravirtualisation**.

Since Intel's VT-x and AMD-V, in 2005 and 2006, the processor supports virtualisation directly. A guest runs in a special mode at full speed, kernel and all, and certain events cause a **VM exit** to the hypervisor: some privileged instructions, accesses to emulated devices, and interrupts. Linux's KVM, added to the kernel in 2007, turns Linux itself into a hypervisor, with QEMU emulating devices; most clouds run KVM or a close relative. A VM exit saves the guest's state, runs the hypervisor and resumes the guest, about a microsecond or two, and the example's last table shows why exits must be kept rare: 500,000 a second would take three quarters of a core. Paravirtual devices such as `virtio`, which batch requests through shared memory rings, and direct assignment of hardware devices to guests are the usual cures.

Memory for guests

A guest kernel has page tables that map its virtual addresses to what it believes are physical addresses. Those are really guest-physical addresses, which the hypervisor maps to the host's real memory with page tables of its own. With **nested paging**, Intel's EPT and AMD's NPT, the hardware walks both. Each of the guest's table pointers is itself a guest-physical address that needs a host walk, so a TLB miss walks a two-dimensional grid: up to $(4 + 1) \times$

$(4 + 1) - 1 = 24$ memory reads with four-level tables in both, against 4 natively, and 35 with five levels. TLBs and huge pages in the host matter even more for virtual machines than for processes, and hypervisors back guest memory with large pages whenever they can.



Where the boundary sits. A virtual machine's guest can only reach the hypervisor through a narrow interface of virtual hardware. A container's processes make system calls into the same kernel as everything else on the host, which is lighter and a much larger surface to defend.

A container is a process

The Linux kernel has no "container" object. A container is an ordinary set of processes that the kernel shows a restricted view and holds to limits, using features added separately over a decade. **Namespaces** give a group of processes its own instance of a global resource: a PID namespace with its own process 1, which the host sees under some other number; a mount namespace with its own file-system tree; a network namespace with its own interfaces, addresses and ports; and namespaces for host name, users, IPC and cgroups. **Cgroups** limit and account for what the group uses: CPU, memory, I/O and the number of processes. On top of these come dropped capabilities, so that root inside the container lacks most of root's powers, and a **seccomp** filter that blocks system calls the workload shouldn't need. Docker, containerd and Kubernetes assemble all this from a container image, following the Open Container Initiative's specifications.

Images and layers

A container image is a stack of file-system layers, each a set of changes on top of the one below: a base distribution, then installed packages, then the application. **overlayfs** presents them as one directory tree. Looking up a path takes the topmost layer that has it, so a container sees its image's files without any of them being copied. The layers are read-only and shared by every container started from the image, and each container adds one thin writable layer on top. Writing to a file from a lower layer first copies the whole file up into the writable layer, **copy-up**, and deleting one leaves a **whiteout** entry that hides it. In the example, a container changes a configuration file and deletes the shell, and its neighbour, started from the same image, still sees both untouched.

EXAMPLE

containers.py · an overlay file system with copy-up and whiteouts, a CPU quota throttling a parallel request, nested page walks, and the cost of VM exits

```
# 1. A container image is a stack of read-only layers; each container adds one writable layer
    on top.
#   A lookup takes the topmost layer that has the path. Writes copy the file up; deletes leave
    a
#   "whiteout" marker that hides the lower copy.
WHITEOUT = object()
base = {"/bin/sh": "shell", "/etc/os-release": "Debian 13", "/usr/lib/libc.so": "libc"}
app = {"/app/server.py": "print('hi')", "/etc/app.conf": "port=8080"}
class Container:
    def __init__(self, *layers): self.layers, self.upper = layers, {}
    def read(self, path):
        for layer in (self.upper, *reversed(self.layers)):
            if path in layer:
                return None if layer[path] is WHITEOUT else layer[path]
        return None
    def write(self, path, data): self.upper[path] = data # copy-up: the lower copy is
        untouched
    def delete(self, path): self.upper[path] = WHITEOUT
    def listing(self):
        paths = set().union(*self.layers, self.upper)
        return sorted(p for p in paths if self.read(p) is not None)

web1, web2 = Container(base, app), Container(base, app) # two containers, one image
web1.write("/etc/app.conf", "port=9090")
web1.delete("/bin/sh")
print(f"web1 reads /etc/os-release: {web1.read('/etc/os-release')}!r} (from the base layer)")
print(f"web1 reads /etc/app.conf: {web1.read('/etc/app.conf')}!r} (its own copy)")
print(f"web2 reads /etc/app.conf: {web2.read('/etc/app.conf')}!r} (the image is unchanged)")
print(f"web1 sees: {web1.listing()}")
print(f"web2 sees: {web2.listing()}")
print(f"web1's writable layer holds {len(web1.upper)} entries; both containers share the 2
    image layers")

# 2. A cgroup CPU limit: cpu.max = "50000 100000" allows 50 ms of CPU per 100 ms period.
#   A burst of parallel work uses the whole quota early in the period, then waits for the next
    one.
def finish_time(cpu_ms_needed, threads, quota=50, period=100):
    t, used, done = 0, 0, 0.0
    while done < cpu_ms_needed:
        if t % period == 0: used = 0 # a new period: the quota
            refills
        if used < quota:
            run = min(threads, quota - used, cpu_ms_needed - done) # ms of CPU in this 1 ms
                tick
            used += run; done += run
            t += 1
    return t
print("\na request needing 120 ms of CPU, in a container limited to half a CPU:")
for threads in (1, 4):
    label = f"{threads} thread{'s' if threads > 1 else ''}:"
    print(f"    {label:10} finishes after {finish_time(120, threads):3} ms "
        f"    {f'unlimited, on idle cores: {120 // threads} ms}")

# 3. Virtual machines and page tables: with nested paging, every guest table pointer must
    itself be
#   translated through the host's tables, so a TLB miss walks a two-dimensional grid of
    tables.
print()
for levels in (4, 5):
    nested = (levels + 1) * (levels + 1) - 1
```

```

print(f"{levels}-level tables in the guest and the host: a TLB miss reads up to {nested:2}
"
      f"table entries (native: {levels})")

# 4. What VM exits cost: each one saves the guest, runs the hypervisor and resumes (~1-2 µs)
print()
for exits_per_s in (1_000, 50_000, 500_000):
    share = exits_per_s * 1.5e-6
    print(f"    {exits_per_s:>7,} VM exits per second at 1.5 µs each: {share:6.2%} of a core")

```

OUTPUT

```

web1 reads /etc/os-release: 'Debian 13' (from the base layer)
web1 reads /etc/app.conf:   'port=9090' (its own copy)
web2 reads /etc/app.conf:   'port=8080' (the image is unchanged)
web1 sees: ['/app/server.py', '/etc/app.conf', '/etc/os-release', '/usr/lib/libc.so']
web2 sees: ['/app/server.py', '/bin/sh', '/etc/app.conf', '/etc/os-release',
            '/usr/lib/libc.so']
web1's writable layer holds 2 entries; both containers share the 2 image layers

a request needing 120 ms of CPU, in a container limited to half a CPU:
  1 thread: finishes after 220 ms (unlimited, on idle cores: 120 ms)
  4 threads: finishes after 205 ms (unlimited, on idle cores: 30 ms)

4-level tables in the guest and the host: a TLB miss reads up to 24 table entries (native: 4)
5-level tables in the guest and the host: a TLB miss reads up to 35 table entries (native: 5)

    1,000 VM exits per second at 1.5 µs each:  0.15% of a core
    50,000 VM exits per second at 1.5 µs each:  7.50% of a core
    500,000 VM exits per second at 1.5 µs each: 75.00% of a core

```

Limits: CPU quotas and throttling

Cgroups offer two kinds of CPU control. `cpu.weight` shares the processor in proportion when groups compete, like nice values (chapter 6). `cpu.max` sets a hard quota per period, such as 50 ms of CPU time every 100 ms, half a core. A quota counts CPU time across all of a container's threads, so a parallel burst can use the whole quota early in a period and then sit **throttled** until the next one starts. In the example, a request that needs 120 ms of CPU finishes in 220 ms on one thread, and still takes 205 ms on four threads, although four idle cores could finish it in 30. Throttling like this shows up as unexplained latency spikes on services that are nowhere near their average limit, which is why many teams running Kubernetes set CPU requests, which become weights, and leave CPU limits off. A related trap is a runtime that sizes its thread pool from the host's core count rather than the container's limit; the JVM has read cgroup limits since Java 10, and Go's `GOMAXPROCS` since Go 1.25. Memory limits are harder still: a container that exceeds `memory.max` is killed by the out-of-memory killer.

How strong is the wall?

A virtual machine's guest reaches the host only through the hypervisor and its virtual devices, a small and heavily scrutinised interface. A container's processes call straight into the host kernel, with its hundreds of system calls, and a single kernel bug can let one escape. Containers are therefore excellent for packaging and for running trusted workloads side by side, and need more care for untrusted code. Two families of projects bridge the

gap. gVisor intercepts a container's system calls and serves them from its own user-space kernel. Kata Containers and Firecracker run each container in a lightweight virtual machine: Firecracker, built for AWS Lambda, boots a microVM in about 125 ms with under 5 MiB of overhead, which gives virtual-machine isolation with container-like speed.

COMMON TRAPS

Treating a container as a VM-strength boundary for untrusted code, especially a privileged one or one with Docker's socket mounted.

CPU limits on latency-sensitive services, which throttle bursts even when the machine is idle.

Runtimes that size thread pools or heaps from the host rather than the container's limits.

Heavy writes into the container's writable layer instead of a volume.

A main process that neither handles signals nor reaps zombies (chapter 3).

EXERCISES

EXERCISE 20.1 Quota arithmetic

A container has `cpu.max` set to "200000 100000", two CPUs' worth. A request runs 16 threads in parallel and needs 1.6 s of CPU in total. How long does it take on an idle 16-core host with no limit, and with the limit? What would you change?

SOLUTION

Without a limit, 16 threads on 16 cores finish 1.6 s of work in 0.1 s. With the limit, each 100 ms period allows 200 ms of CPU, which 16 threads use up in 12.5 ms; then the container is throttled for the remaining 87.5 ms. The request needs $1.6 / 0.2 = 8$ periods' worth of quota, so it finishes 12.5 ms into the eighth period: $7 \times 100 + 12.5 \approx 713$ ms, seven times slower than unlimited, spent mostly waiting. Options: run 2 threads, matching the quota, which takes 800 ms but makes steady progress instead of short bursts and long stalls; raise the limit; or remove it and rely on `cpu.weight`, which only constrains the container when others compete.

EXERCISE 20.2 Appending one byte

A container's image includes a 2 GB data file in its base layer. The application appends one byte to it. What does overlayfs do, and what does it cost? What should the application do instead?

SOLUTION

overlayfs works at the level of whole files. Before the first write to a file from a lower layer, it copies the entire file into the container's writable layer, so the one-byte append first copies 2 GB, taking seconds and doubling the disk space used, and the copy belongs to that container alone: it disappears when the container is removed, and other containers from the same image don't see it. Data that changes belongs in a **volume**, a directory mounted from the host or from network storage, which bypasses the overlay entirely, is shared or persisted as needed, and doesn't depend on the container's lifetime. Images should hold code and read-only data.

EXERCISE 20.3 VM or container?

Choose virtual machines or containers for each, with a reason: running customers' untrusted code; fifty internal services built by one team; a Windows application on Linux hosts; a CI job that builds container images.

SOLUTION

Untrusted customer code: a virtual machine, or a microVM such as Firecracker or Kata, or at least gVisor, because a shared kernel is too large an attack surface to bet a multi-tenant service on. Fifty internal services: containers, which start in milliseconds, share memory for common layers and pack densely, and the services trust each other. A Windows application on Linux: a virtual machine, because a container shares the host's kernel and a Linux kernel can't run Windows binaries natively. A CI job that builds images: building images traditionally needed a privileged container, which is close to root on the host, so either run such jobs in short-lived virtual machines or use rootless build tools that need no special privileges.

CHECKLIST

- ☐ Can you explain trap and emulate, and what hardware virtualisation added?
- ☐ Can you explain nested paging and compute the cost of a nested TLB miss?
- ☐ Can you list the namespaces and cgroup controllers a container uses?

- ☐ Can you explain overlayfs layers, copy-up and whiteouts?
- ☐ Can you explain CPU quota throttling and choose between weights and limits?

GO DEEPER

BOOK Operating Systems: Three Easy Pieces, appendix B: virtual machine monitors Arpaci-Dusseau & Arpaci-Dusseau

REFERENCE namespaces(7): an overview of Linux namespaces The Linux man-pages project

PAPER Firecracker: lightweight virtualization for serverless applications Agache et al., NSDI 2020

The stage asks for three things: a shell that supports pipes, redirection and job control; a thread pool with a work queue; and the xv6 labs from MIT's operating systems course, including at least one system call of your own. The first two are programs you write on top of a real operating system. The third has you change one.

The shell

Chapter 3's example is the skeleton. Write the real one in C, where every system call is visible, and give it these features, in roughly this order:

- 01 Read a line, split it into words with quoting, and run a command with `fork`, `execvp` and `waitpid` (chapter 1).
 - 02 Report the exit status as `$?`, including 127 for a missing command and 128 plus *n* for a signal.
 - 03 Built-ins that must run in the shell itself: `cd`, which changes the shell's own directory, and `exit`.
 - 04 Redirection with `<`, `>`, `>>` and `2>&1`, applied left to right with `dup2`.
 - 05 Pipelines of any length, closing every unused pipe end in the parent and in each child.
 - 06 A process group per job, the terminal handed to the foreground job with `tcsetpgrp`, and Ctrl-C and Ctrl-Z reaching only that job.
 - 07 Background jobs with `&`, `jobs`, `fg` and `bg`, and finished children reaped through `SIGCHLD`.
 - 08 A test script that runs the shell on a file of commands and compares its output and statuses with those of `/bin/sh`.
-

The thread pool

A thread pool is a bounded blocking queue (chapter 9) and a fixed set of threads that take tasks from it. Build it in any language with threads and condition variables, without the standard library's executor, and then compare it with that executor. It should provide:

- 01 A fixed number of worker threads, created once (chapter 4), and a way to choose that number from the workload.
- 02 A queue with a capacity, so that `submit` blocks or fails when it's full rather than growing without limit.
- 03 `submit` returning a future whose result, or exception, the caller can wait for.
- 04 Shutdown in two forms: finish everything queued, or stop now and cancel what hasn't started.
- 05 No busy-waiting anywhere, and every wait on a condition inside a `while` loop.
- 06 A stress test with many submitters and tiny tasks, run under ThreadSanitizer or a race detector (chapter 7).

The xv6 labs

xv6 is a small teaching re-implementation of Unix for RISC-V, written at MIT, and its labs, from the course 6.1810, Operating System Engineering, have you extend it. Do them in order; each assumes the ones before. The table lists the labs of the 2026 course and the chapters that prepare for each.

LAB	WHAT YOU BUILD	CHAPTERS
util	user programs such as <code>sleep</code> and <code>find</code> , using <code>fork</code> , <code>exec</code> , pipes and directories	Ch. 1, 3, 16
syscall	debugging the kernel with gdb, new system calls that sandbox a process, and an attack on a kernel bug	Ch. 2, 20
pgtbl	printing page tables, detecting which pages were accessed, a page shared with the kernel to speed up a system call, and superpages	Ch. 2, 12, 13
traps	a kernel backtrace, and an alarm that interrupts a process and resumes it	Ch. 2, 3

cow	copy-on-write <code>fork</code>	Ch. 13
net	a driver for an emulated network card, with descriptor rings and interrupts	Ch. 18
lock	a memory allocator with per-CPU free lists, and a read-write lock	Ch. 8, 15
fs	doubly indirect blocks for large files, and symbolic links	Ch. 16
mmap	<code>mmap</code> and <code>munmap</code> for files, with pages loaded on demand	Ch. 13, 16, 19

The syscall lab already asks for new system calls, which meets the stage's requirement, but it's worth adding one more of your own design at the end, such as a call that reports a process's page-fault count, because designing the interface and checking every argument teaches more than following a specification. Chapter 19 of *The Engineer's Mill* covers the device side of the net lab. The xv6 book, free on the course site, is the companion to the code; read the chapter for each lab first.

One per chapter, each linked back to it. Read them before you write concurrent code, chase a memory problem, or trust that data has reached the disk.

-
- 01 Wait for every child you create, and flush your output before you fork.
 - 02 Cross into the kernel as rarely as you can: buffer and batch, and count the system calls.
 - 03 Close every pipe end you don't use, and answer `SIGTERM` by finishing your work and exiting.
 - 04 Create threads once and reuse them, and size the pool with Little's law, not by guesswork.
 - 05 Decide whether you need short turnaround or quick response before you choose a scheduling policy.
 - 06 Weights share a contended processor and quotas cap it; set hard limits only where you can afford the stalls.
 - 07 Shared data needs a lock or an atomic; running a test proves nothing about races, and a race detector does.
 - 08 Hold locks briefly and never across I/O, and let the library's mutex decide when to spin.
 - 09 Wait on a condition in a `while` loop, with the lock held, with one condition variable per condition.
 - 10 Take locks in one global order, and give every wait that can fail a timeout and jittered retries.
 - 11 Never block an event loop, and remember that every `await` is a point where the world can change.
 - 12 Every address is virtual: judge memory by what's resident, not by what's reserved.
-

-
- 13 Mind the TLB and the first touch: huge pages for large random-access data, and warm up before you time.
-
- 14 Size memory to the working set, and treat swapping as an alarm, not as capacity.
-
- 15 Fragmentation wastes memory as surely as leaks do; use pools and size classes, and weak references for back-pointers.
-
- 16 Names point to inodes, and a deleted file lives on until its last descriptor closes.
-
- 17 Nothing is durable until `fsync` returns: write a new file, sync it, rename it, sync the directory.
-
- 18 Wait for many things at once with `epoll` or `kqueue`, and read until `EAGAIN`.
-
- 19 A byte stream needs framing, and shared memory needs synchronisation.
-
- 20 A container is a set of processes with a restricted view of one shared kernel; trust it accordingly.
-

READING LIST

The books, courses and papers behind this volume, and the chapters each one deepens. If you read only one, make it *Operating Systems: Three Easy Pieces*, which is free and covers almost everything here in more depth. If you do only one thing, do the xv6 labs, because changing a real kernel turns every mechanism in this volume from a description into something you've built.

WORK	AUTHOR	DEEPENS
Operating Systems: Three Easy Pieces	Arpaci-Dusseau & Arpaci-Dusseau	Ch. 1–20
6.1810 Operating System Engineering, and the xv6 book	MIT: Cox, Kaashoek & Morris	Ch. 1–3, 12–16
The Linux Programming Interface	Michael Kerrisk	Ch. 1–4, 16–19
CS 162: Operating Systems and System Programming	UC Berkeley	Ch. 1–17
Operating Systems: Principles and Practice	Anderson & Dahlin	Ch. 4–10, 12–15
The Little Book of Semaphores	Allen Downey	Ch. 8–10
Is Parallel Programming Hard, And, If So, What Can You Do About It?	Paul McKenney	Ch. 7–8
Memory models	Russ Cox	Ch. 7–8
Systems Performance	Brendan Gregg	Ch. 2, 4, 13–14, 18
Files are hard	Dan Luu	Ch. 17
The Linux kernel documentation	The kernel's developers	Ch. 6, 10, 13–14

Compiled October 2026 as the companion to stage 05, Operating systems and concurrency, of The Engineer's Syllabus, alongside *The Engineer's Workbench*, *Compass*, *Mill*, *Exchange*, *Ledger*, *Craft*, *Shipyard*, *Fleet*, *Watch*, *Citadel* and *Foundry*. It follows the order of topics in the stage and draws on the works in the reading list, and on the original sources for its named results: Corbató, Merwin-Daggett and Daley (1962) on CTSS; Dijkstra (1965) on semaphores and the dining philosophers; Belady (1966) on replacement; Denning (1968) on working sets; Coffman, Elphick and Shoshani (1971) on deadlock; Hewitt (1973) on actors; Popek and Goldberg (1974) on virtualisation; Hoare (1978) on communicating sequential processes; Lampson and Redell (1980) on monitors; Peterson (1981) on mutual exclusion; McKusick, Joy, Leffler and Fabry (1984) on the Fast File System; Waldspurger and Weihl (1994) and Waldspurger (1995) on lottery and stride scheduling; Molnar (2007) on CFS and Zijlstra (2023) on EEVDF; Axboe (2019) on `io_uring`; Baumann, Appavoo, Krieger and Roscoe (2019) on `fork`; Pillai et al. (2014) on crash consistency in applications; and Agache et al. (2020) on Firecracker. Kernel versions, defaults and tool behaviour are described as of late 2026 and change; the mechanisms change far more slowly, and those are what to rely on.

Every example was executed before publication with Python 3.14 on macOS, and they're written to print the same output on Linux: most make real system calls through the `os`, `signal`, `socket`, `mmap` and `selectors` modules, avoid printing anything that varies between systems, such as process IDs or pipe capacities, and force any timing they depend on with barriers and pipes. The concurrency examples check every interleaving of small programs instead of hoping a race appears. Two examples time real hardware, chapter 2's system calls and chapter 4's threads and processes, and show one recorded run on an Apple M1 Pro. Re-run in October 2026 with Python 3.14.7, every example printed exactly the output shown, apart from the two recorded timings, which differ from run to run and support the same conclusions. Tested here: `fork`, `exec`, `waitpid` and exit statuses; the cost of system calls and of writing a byte at a time; a pipeline launcher with redirection and job control; sharing in threads and processes, and the cost of creating and switching them; FIFO, SJF, STCF, round robin and a multi-level feedback queue; weighted fair queueing, lottery and stride scheduling, and balancing across cores; every interleaving of a lost update and a check-then-act; a model checker for four lock algorithms, with and without store buffers; a bounded buffer with Mesa-style condition variables, then real threads; the dining philosophers, a wait-for graph, a real deadlock and a livelock; an event loop from generators, `asyncio`, an actor and channels; a four-level page table; a TLB with huge pages and copy-on-write `fork`; FIFO, LRU, CLOCK and the optimal policy, and Belady's anomaly; first, best and worst fit, size classes, reference counting and mark and sweep; descriptors, links, unlinked open files and sparse files; every crash during an append, journalling, and atomic file replacement; non-blocking I/O and one thread serving a hundred connections; pipes, FIFOs, framing, shared and private mappings and descriptor passing; and an overlay file system, a CPU quota and nested paging.