

Topic 07

University Computer Science

What this course is, and how to use it

This is the seventh course in University Computer Science: **operating systems** – the layer that turns one processor into an illusion of many machines: processes, scheduling, synchronisation, virtual memory, files, and the walls between users.

Scope limit, stated once: a study course, not a degree; the teaching kernel is an original small design, and every scheduling table, interleaving and page-table walk in this handout is deterministic – computed by the course suite, never simulated by hand-waving.

- Units 7.1–7.3 are the CPU’s illusions: processes, turns, and safe sharing.
- Units 7.4–7.5 are memory and storage illusions: address spaces and files.
- Unit 7.6 is the wall: protection and security.

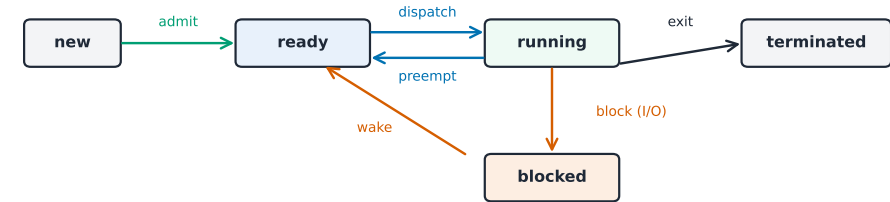
The OS habit: **an operating system is a resource accountant** – every mechanism in this course is a ledger of who owns what turn.

Processes and threads

Why it matters. A **process** is a program in motion: its own address space, registers, and open files. The OS’s first illusion is running hundreds of them on a handful of cores.

The idea, with a picture. A process lives through five **states**: **new**, **ready**, **running**, **blocked** (waiting on I/O), **terminated**. The scheduler moves processes between ready and running; devices wake the blocked. A **thread** is a execution flow inside a process: threads share the address space and files, but each owns its stack and registers.

five states, four transitions that matter



blocked waits on a device; ready waits only for the CPU — the PCB carries everything a suspend must save

Five states, four transitions that matter: dispatch and preempt between ready and running, block on I/O, wake on completion

The derivation (why the PCB exists). To suspend a process the OS must save everything that defines it – the **process control block** (PCB): register values, program counter, page-table pointer, open files. The **context switch** is exactly this save-and-restore: with our course timings, saving and restoring 2000 bytes of state at 2 GB/s costs 1 μ s – plus cache pollution that makes the next runs slower. Switching threads skips the address space (shared), so it is cheaper; switching processes pays the full PCB.

Worked example 1 – a page fault’s journey. Process A runs, faults on a page: running $\rightarrow$ blocked (disk), the scheduler dispatches B, the disk completes, A $\rightarrow$ ready, and later A $\rightarrow$ running again with the faulting instruction restarted. Four state changes, two processes, zero lost work – the restartable instruction is the condition that makes preemption safe at all.

Worked example 2 – thread versus process. A web server spawns a thread per connection: shared cache and configuration, 50 μ s create-and-switch. The same as processes costs 500 μ s (address space copy – well, with copy-on-write mostly the page-table setup) plus separate caches to fill. Sharing is the thread’s whole point – and, as unit 7.3 shows, its whole danger.

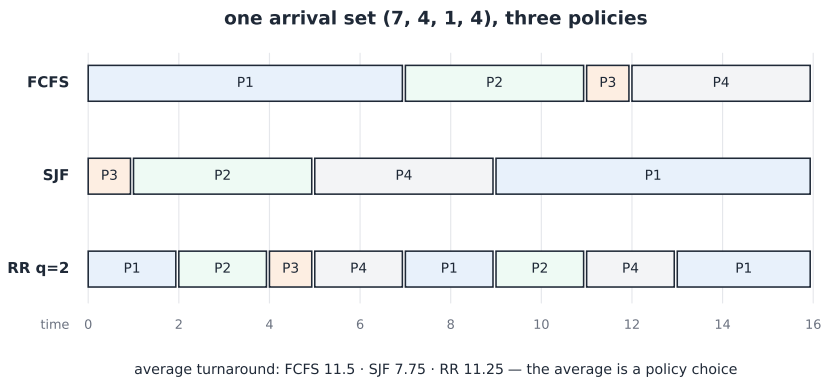
Common mistakes. Confusing blocked with ready (blocked waits on a device; ready waits only for the CPU). Believing a thread crash stays local (it takes the process – one address space). Counting the context-switch cost as pure CPU time (cache refill dominates).

Summary. Process = address space + PCB; thread = stack + registers inside it; five states with dispatch/preempt/block/wake; switching threads is cheaper because the space is shared.

Scheduling

Why it matters. More processes than cores means somebody decides the turns. The scheduler is that somebody, and its policy is a values statement: throughput, fairness, or latency – pick two.

The idea, with a picture. Our teaching queues: **FCFS** (first-come-first-served), **SJF** (shortest job first), **round robin** (RR, quantum $q = 2$), priority with aging. The **Gantt chart** shows who ran when.



One arrival set, three policies: SJF finishes the short jobs first, RR slices everyone in turn

The derivation (the turnaround formula). Turnaround time = finish – arrival; waiting time = turnaround – burst. For the course set – arrivals at 0 with bursts $P1 = 7, P2 = 4, P3 = 1, P4 = 4$ – FCFS finishes at 7, 11, 12, 16: turnarounds 7, 11, 12, 16 (all arrive at 0), average **11.5**. SJF order P3, P2, P4, P1 finishes at 1, 5, 9, 16: turnarounds 1, 5, 9, 16, average **7.75**. RR ($q = 2$) interleaves: the trace is P1 P1 | P2 P2 | P3 | P4 P4 | P1 P1 | P2 P2 | P4 P4 | P1 P1 | P1 – finishes P3 = 5, P2 = 11, P4 = 13, P1 = 16: average turnaround **11.25**. Same jobs, three policies, three answers – SJF wins the average, RR wins the worst-case wait.

Worked example 1 – convoy effect. Under FCFS a long job (P1, 7) arrives first and everything queues behind it: P3 waits 11 units for 1 unit of work. SJF would run P3 first (wait 0) – the convoy is FCFS’s signature failure with interactive users in the mix.

Worked example 2 – quantum choice. RR with $q = 2$: 9 run segments, so 8 preemptions in our trace; with $q = 7$: P1 finishes in one slice – fewer switches, but short P3 waits 6+ before its first run. The rule: q large degenerates to FCFS; q tiny spends the CPU on switching (our course bound: switch cost 1 μ s in a 2 ms quantum wastes 0.05%; at 20 μ s quantum, 5%).

Common mistakes. Averaging completion times instead of turnarounds. Forgetting

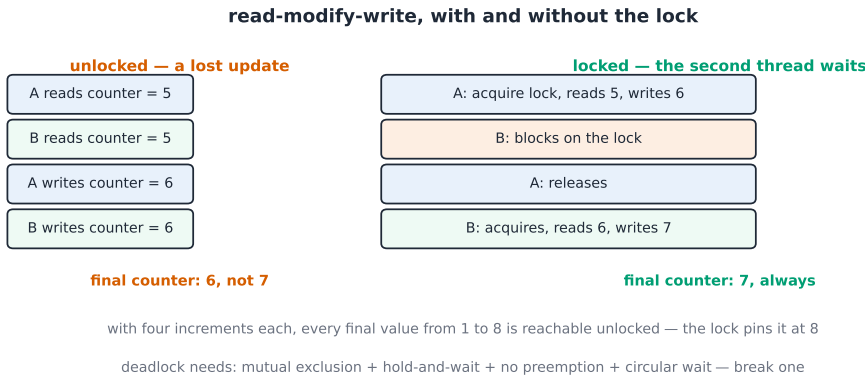
SJF needs burst foreknowledge (it is theoretical; MLFQ approximates it). Calling a small quantum faster (switches are work, not progress).

Summary. Turnaround = finish – arrival; SJF minimises average turnaround among non-preemptive policies; RR trades average for fairness and latency; the quantum prices the trade.

Synchronisation

Why it matters. Threads share memory, and unsynchronised sharing corrupts it: the lost update is the smallest and most expensive bug in computing.

The idea, with a picture. The classic race: two threads run `counter = counter + 1` on `counter = 5`. Each reads, adds, writes; the interleaving R1 R2 W1 W2 leaves 6, not 7 – one update is lost. The fix is **mutual exclusion**: a **lock** makes read-modify-write atomic.



Unlocked: the interleaving reads 5 twice and writes 6 twice – one increment vanishes; locked: the second thread waits

The derivation (the critical section bound). A correct synchronisation primitive for N threads must satisfy: mutual exclusion (one inside), progress (someone enters if free), bounded waiting (no thread starves). Peterson’s algorithm meets all three with two flags and a turn variable; hardware **test-and-set** meets them with one atomic instruction – but both **spin**, burning the CPU. Semaphores and mutexes add the OS queue: the blocked waiter sleeps instead, which is why real systems layer them.

Worked example 1 – counting deterministically. Two threads increment a shared counter 4 times each, unlocked. If every read of both threads happens before any write, all eight writes store $\text{read} + 1 = 1$, and the counter ends at **1**: seven of the eight increments vanish. The suite enumerates the interleavings with per-increment registers: every final value from 1 to 8 is reachable – the race does not corrupt loudly, it undercounts quietly. Locked: always 8.

Worked example 2 – deadlock’s four horsemen. Two threads, locks L1, L2: A holds L1 wants L2; B holds L2 wants L1. Deadlock needs mutual exclusion, hold-and-wait, no preemption, and circular wait – break any one and it cannot happen. The engineer’s favourite to break: circular wait, by ordering the locks (always L1 before L2).

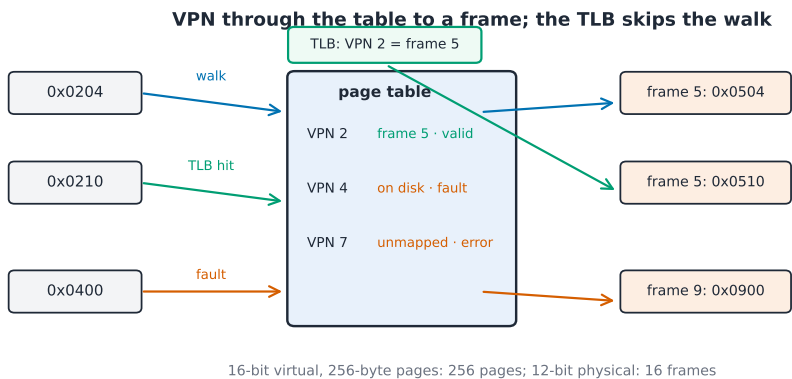
Common mistakes. Fixing races by inserting sleeps (narrows the window, closes nothing). Taking locks in different orders across threads (the deadlock recipe). Believing a single-core machine is safe (preemption between the read and the write still races).

Summary. Races come from non-atomic read-modify-write; locks make them atomic at the price of waiting; correctness is exclusion + progress + bounded waiting; break one of the four deadlock conditions – ordering locks is the cheap one.

Virtual memory

Why it matters. Every process believing it owns the whole address space – while sharing one physical RAM – is the OS’s grandest illusion and the source of both its speed and its worst failures.

The idea, with a picture. The **virtual address** splits into page number and offset; the **page table** maps virtual pages to **physical frames**, one entry per page; the **TLB** caches translations. Our course machine: virtual 16 bits, page 256 bytes $\Rightarrow$ 256 virtual pages; physical 12 bits $\Rightarrow$ 16 frames.



Virtual page numbers through the page table to frames: VPN 2 to frame 5, VPN 4 to frame 9, VPN 2 again hits the TLB

The derivation (the size of the illusion). Address width sets the space: 16-bit virtual, 256-byte pages $\Rightarrow$ VPN = 8 bits $\Rightarrow$ 256 pages; a process may address 64 KiB while physical frames number 16 (4 KiB). The page table is the ledger: 256 entries $\times$ (frame 4 bits + valid 1 + dirty 1) = 1536 bits. Two processes each get all 256 virtual

pages – overcommit is free until someone touches an unmapped page.

Worked example 1 – the walk, then the hit. Address 0x0204: offset 0x04 (low 8 bits), VPN 0x02. First access: TLB miss, page-table read: VPN 2 $\rightarrow$ frame 5, physical = $5 \times 256 + 4 = 0x0504$; the TLB caches VPN 2. Address 0x0210 next cycle: same VPN 2 – TLB hit, physical 0x0510, no table walk. The 10-entry TLB on our 256-page space: an 8-page loop of 8000 accesses misses 8 times (compulsory) – a 99.9% hit rate (suite-simulated, LRU).

Worked example 2 – the fault and its price. Address 0x0400: VPN 4, table entry invalid $\rightarrow$ **page fault**. The process blocks, the OS finds a victim (LRU: frame 9’s page, dirty – write it back), reads VPN 4 into frame 9, updates the table and TLB, restarts the instruction. Cost model: TLB hit 1 ns, table walk 10 ns, fault + disk 5 ms – one fault per million accesses still costs 5 ns per access on average ($5\text{ ms}/10^6$): the hierarchy tolerates rarity, not frequency.

Common mistakes. Computing the physical address without the frame multiply. Forgetting the dirty bit doubles a victim’s cost. Treating every page fault as either a disk fetch or an invalid access: a valid nonresident page may be loaded, a new page may be allocated, and a protection violation may terminate the process.

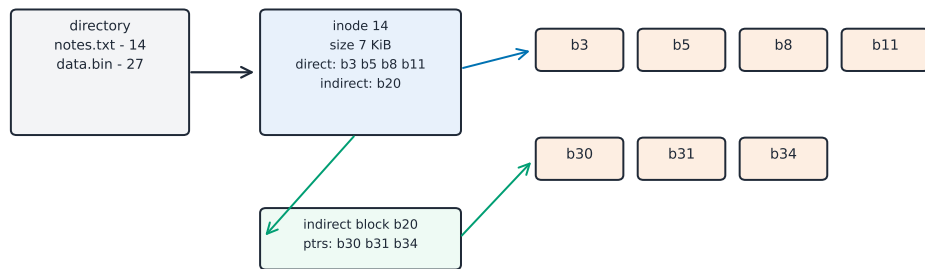
Summary. VPN + offset through the page table to frame + offset; the TLB makes walks rare; faults make overcommit safe at disk prices; the bits (valid, dirty) are the ledger’s footnotes.

File systems

Why it matters. Files are the illusion that outlives the process: the layout beneath decides speed, reliability, and what a crash eats.

The idea, with a picture. Our teaching disk: 32 blocks of 1 KiB. The **inode** holds the metadata and 4 direct pointers plus one indirect (pointing at a block of 256 pointers); the **directory** is a file of name $\rightarrow$ inode entries; **FAT** chains blocks instead.

name to inode to blocks — direct first, indirect for more



1 KiB blocks, 256 pointers per indirect block: max file $(4 + 256) \times 1 \text{ KiB} = 260 \text{ KiB}$

Directory entries to inodes to blocks: direct pointers first, the indirect block for big files, the free list on the side

The derivation (the file size bound). With 4 KiB blocks the arithmetic scales: 4 direct + one indirect block holding 1024 pointers $\Rightarrow$ max file $= (4 + 1024) \times 4 \text{ KiB} \approx 4 \text{ MiB}$. Our teaching size (1 KiB blocks, 256 pointers): $(4 + 256) \times 1 \text{ KiB} = 260 \text{ KiB}$. Access cost: block $n < 4$ costs one disk read (plus the inode, cached); block $n \geq 4$ adds the indirect block read. Fragmentation: chained layouts spread a file, so sequential reads become seeks – contiguity is a speed decision, not a correctness one.

Worked example 1 – read /notes/x, block 6. The directory lookup finds the inode (cached), block 6 is beyond the four direct pointers: read the indirect block (pointer $6 - 4 = 2$), then the data block – two disk reads for one logical block. Block 2 would have cost one. The suite counts: reading a 100-block file sequentially $= 100 + 1$ indirect reads, best case.

Worked example 2 – the crash question. Append writes the data block, then the free-list bit, then the inode size. Crash between writes: which order loses what? Data-then-inode loses nothing but space (orphan block); inode-then-data lies about size (reads return garbage). **Journaling** writes the intent first and replays – the reason modern file systems pay a small write tax for crash consistency.

Common mistakes. Forgetting the directory is itself a file (lookups are reads). Assuming the indirect pointer is free (it is a disk read). Believing deletion erases (it clears pointers – the data stays until reused).

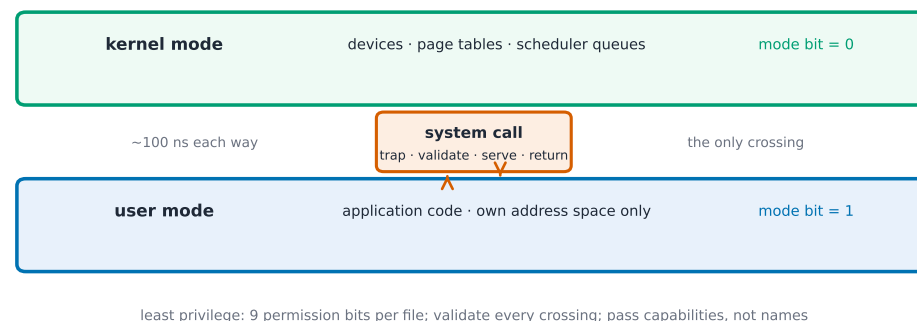
Summary. Directory $\rightarrow$ inode $\rightarrow$ direct/indirect pointers $\rightarrow$ blocks; size bound $= (\text{direct} + \text{pointers per block}) \times \text{block size}$; write order decides what a crash eats; journaling buys consistency.

Protection and security

Why it matters. Everything above assumed the programs cooperate. Protection is the hardware-plus-OS wall that makes the assumption unnecessary – and security is the practice of keeping that wall standing.

The idea, with a picture. Two rings : user mode runs application code; kernel mode runs the OS and can touch devices and page tables. The crossing points are **system calls** – the wall’s only doors – and the **mode bit** on the CPU enforces them.

two rings, one door



User code crosses to the kernel only through the syscall door; the kernel validates everything at the boundary

The derivation (the principle of least privilege). Every mechanism above is an attack surface: the scheduler’s queues, the page table’s bits, the file metadata. Least privilege gives each process only what it needs: the page tables already do it per process (unit 7.4), file permissions do it per user (read/write/execute $\times$ owner/group/other $= 9$ bits), and capabilities do it per object. The economics: a bug in a smaller privileged base is a smaller target – the reason microkernels and sandboxed browsers exist.

Worked example 1 – one syscall’s crossing. `open("/notes/x")` from user mode: the trap instruction switches the mode bit, saves user registers, jumps to a fixed vector; the kernel validates the path (permissions), fills a file descriptor, returns – and the mode bit flips back. Two crossings per syscall at $\approx 100 \text{ ns}$ each: a program doing 10^6 syscalls pays 0.2 s – the reason libraries buffer and batch.

Worked example 2 – the confused deputy. A privileged printer daemon asked to print `/salary.db` by an unprivileged user: the daemon’s authority, not the user’s, opens the file – a deputy confused about whose wishes to execute. The fix is to pass capabilities (open descriptors), not names: the deputy can only use what it was handed. The wall’s doors must check identity at every crossing, or the wall itself becomes the weapon.

Common mistakes. Calling security a layer (it is a property of every layer). Checking permissions at open but not at use (TOCTOU: time-of-check to time-of-use). Trusting the pointer because it came from user space (validate at the boundary, always).

Summary. The mode bit makes the wall, syscalls are its doors, least privilege sizes the rooms; validate every crossing; pass capabilities, not names.