Operating Systems

Tripos Revision Guide

CST Part IA

Complete coverage of Structures, CPU Scheduling, Memory & Paging,
Virtual Memory, I/O, Storage & Filesystems, and the UNIX/Linux Case Study

Topic Overview

Tripos patterns covered: 1997–2025 • Full worked numerical examples • Formula sheet • Common pitfalls

July 10, 2026

0 Contents

1 Introduction & System Organisation

1.1 What is an Operating System?

An OS is just a program that (efficiently) provides four things: **Control** over the execution of all other programs, **Multiplexing** of resources between programs, **Abstraction** over complexity and low-level detail, and **Extensibility** to meet changing demands. There is no universal agreement on exactly what "the OS" is (kernel? libc? runtime? browser?) – this course focuses on the **kernel**. The OS provides *mechanisms* which are used to implement *policies*; policies may be deliberately designed or accidents of implementation.

Exam Tip: Mechanism vs Policy

A classic Tripos distinction. **Mechanism** = *how* something is done (e.g., a timer interrupt). **Policy** = *what* is decided (e.g., which process runs next). Separating the two allows the same mechanism to support many different policies – always mention this when asked to justify a design.

1.2 Computer System Organisation

Hardware provides basic resources (CPU, memory, I/O); the OS controls and coordinates their use; application programs define how resources solve users' problems; users motivate the whole thing. The CPU executes programs using memory (a large byte-addressed array), devices for I/O, and a bus to move information between them. CPUs and devices are *concurrently active*, competing for memory cycles and bus access.

Buses

A bus is a shared communication medium: low cost and versatile, but a potential bottleneck. It comprises **address lines** (how many devices addressable), **data lines** (bits transferred at once), and **control lines** (target device, operation). Operation is *initiator–responder*: initiator asserts address and a read/write signal; responder replies. Buses form a **hierarchy** (processor bus → memory bus → PCI → legacy buses such as ISA), joined by **bridges** that forward requests from one side to the other.

1.3 System Operation

Booting

The bootstrap program (bootloader, historically in ROM/BIOS, now more often UEFI) initialises the system, then finds, loads and executes the kernel (possibly in stages). The kernel then enables process creation, device I/O and filesystem access; system processes start, beginning with `init` on UNIX.

Interrupts

I/O devices and the CPU execute concurrently. Each device controller has a local buffer; the CPU moves data between main memory and that buffer. The controller signals completion by raising an **interrupt** – the OS is fundamentally *interrupt driven*. Interrupts decouple the CPU's request from the device's (much slower) response: a 2ms disk read could otherwise waste 5×10^6 clock cycles of busy-waiting.

Interrupt Handling – Sequence

A raised interrupt transfers control via the **interrupt vector** (a table of Interrupt Service Routine addresses). The architecture saves the address of the interrupted instruction; a special instruction (e.g. `rti`) resumes it. Interrupts are typically deferred to an instruction boundary; the CPU saves (most) registers so the ISR cannot corrupt state. A **trap** (or exception) is a *software-generated* interrupt, caused by an error or a deliberate request (e.g., a system call).

Storage definitions

Bit → byte (8 bits) → word (architecture's native unit, e.g. 8 bytes on a 64-bit machine). 1 kB = 1024 bytes, 1 MB = 1024^2, 1 GB = 1024^3, 1 TB = 1024^4 (strictly the IEC *kibi-/mebi-/gibi-* prefixes; usage in practice is inconsistent, e.g. RAM vs. hard-disk marketing).

Storage hierarchy

Organised by speed, cost and volatility: registers → L1/L2/L3 cache → main memory (large, volatile, directly CPU-addressable) → secondary storage (very large, non-volatile: HDDs with tracks/sectors, or SSDs, faster but non-volatile). A **device driver** provides a uniform interface between each device controller and the kernel.

1.4 System Concepts

Key Vocabulary

Layering: manage complexity by restricting a layer X to use only layer $X-1$'s services and provide services only to layer $X+1$ (cf. OSI/Internet stack).
Multiplexing: one resource consumed by multiple consumers simultaneously.
Latency: how long something takes (e.g. "this read took 3ms").
Bandwidth: rate at which something occurs ($\sim$ throughput, e.g. "2Gb/s").
Jitter: statistical dispersal (variation) in latency.
Caching: small, fast storage masks a slower component's latency; relies on *locality* in time and space.
Buffering: memory placed between two components to absorb short-term rate mismatches – useless if the *long-term average* bandwidths differ.
Bottleneck: the most constrained resource; **80/20 rule**: 80% of time is spent in 20% of code, so measurement must precede tuning.

1.5 Resource Management Roles of the OS

- **CPU**: multiplex running threads over the CPU(s); lifecycle management, synchronisation, communication.
- **Memory**: track ownership; manage (de)allocation for code and data.
- **Storage**: abstract different media; create/delete/manipulate files, directories, free space.
- **I/O Subsystem**: abstract device peculiarities; provide drivers; manage buffering, caching, spooling.

Protecting the CPU, Memory and I/O

CPU: a (countdown) **timer** interrupts periodically so the OS regains control; only the OS may load the timer and its interrupt cannot be masked – this is also what enables time-sharing. **Memory**: a **base and limit register** pair define each program's legal address range; hardware checks every reference, raising an exception on violation; only the OS may update these registers. **I/O**: naive protection makes I/O instructions privileged, but some devices are memory-mapped, so **I/O protection ultimately relies on memory protection**.

TRAP: "Just make I/O instructions privileged"

This alone is insufficient! Memory-mapped device registers mean an unprivileged process could still read/write device state (and even rewrite interrupt vectors) unless memory protection also covers those addresses. A classic "true/false, explain why" Tripos question (2001).

2 Protection & OS Evolution

2.1 Evolution of Operating Systems

The Four Eras

Open shop (e.g. EDSAC, 1947–55): one machine, one user, one program; the user is the programmer is the operator.
Batch systems: tape drives collate and run batches of programs; **spooling** overlaps I/O with computation.
Multiprogramming: many programs loaded, one running at a time; **job scheduling** chooses which resident job runs.
Timesharing: switch jobs so fast users perceive concurrent execution; **CPU scheduling** chooses which *ready* job runs; enables interactive computing.

2.2 Dual-Mode Operation

Hardware **mode bit** distinguishes (at least) **user mode** (application code) from **kernel mode** (OS code); some instructions are *privileged*, executable only in kernel mode. Modern CPUs often support more modes still, e.g. a VMM/hypervisor mode for guest VMs, and "nested" rings (x86 rings 0–3) where an inner ring can do strictly more than an outer one.

2.3 Kernels

System calls

Since applications cannot do I/O directly, the kernel does it for them via a **trap** (software interrupt) that switches user → kernel mode. System calls are invoked by number, indexing a system call table; parameters may be (1) loaded into registers, (2) pushed on the stack, or (3) placed in memory with the address passed in a register – the latter two are usually preferred since registers are few and small. System calls are usually wrapped by a **runtime support library** (e.g. glibc) providing a language-specific API.

Microkernels

OS interfaces must be extremely stable, making them hard to extend or shrink. **Microkernels** move OS services into (sometimes privileged) user-space servers accessed via **message passing** rather than trapping, increasing modularity/extensibility at a performance cost (message passing must be efficient). Real systems blur the line: Linux has kernel modules *and* servers; Windows NT 3.5 was a microkernel but NT 4.0 moved services back into the kernel for performance.

Virtualisation

Virtual Machines encapsulate an entire OS and boot it over a **hypervisor** – Type 1 runs directly on hardware (possibly using extensions like VT-x); Type 2 runs above a host OS. **Containers** instead expose kernel functionality so each container looks like a separate system while sharing the underlying kernel (e.g. Linux Containers, FreeBSD Jails). VMs isolate more strongly; containers are lighter weight.

2.4 Security

Principle of least privilege

Objects (hardware or software) should be given *just enough* privilege to perform their task, limiting the damage from a bug or compromise. Privilege can be **static** or **dynamic** (domain switching, privilege escalation). **Compartmentalisation** takes this further, granularly restricting access to each component. **Covert channels** leak information via side effects not intended as a communication path (e.g. page-fault statistics, input-dependent timing, electromagnetic emission).

Domain of protection & the Access Matrix

An **access-right** is a pair $\langle$object-name, rights-set$\rangle$; a **domain** is a set of access-rights (in UNIX, a domain is a user ID). The **access matrix** has domains as rows and objects as columns; entry (d, o) is the set of operations domain d may invoke on object o. It is conceptually a table of triples $\langle$domain, object, rights-set$\rangle$, but is typically *huge and sparse*, so two compressed representations are used:

ACLs vs Capabilities

Access Control List (ACL) – store *by object*: each object keeps a list of ⟨domain, rights⟩ pairs (i.e. one *column* of the matrix). Natural for storage systems (e.g. inserted into the file naming path). Checking requires consulting the object, so on-disk ACLs need caching for high-duty-cycle checks.

Capability – store *by domain*: each domain keeps a list of ⟨object, rights⟩ pairs (i.e. one *row*). A capability is an unforgeable, OS-protected "secure pointer" – possession *is* authorisation. Hardware capabilities (e.g. CHERI) have special instructions to restrict capabilities and pass them across calls; software capabilities are protected by encryption, useful in distributed systems.

Exam Tip: Which representation for a laptop?

A modern personal laptop has relatively *few users* but *many files* ⇒ ACLs (per-object lists) are compact and natural, matching UNIX/Windows practice. Capabilities suit systems with many mutually-suspicious processes needing fine-grained, revocable, delegatable rights (distributed/microkernel systems). This exact reasoning was examined in 2025.

Authentication

User to system: typically a password/passphrase, hashed with a salt (easy to compute, hard to invert); multi-factor authentication adds a second factor; failed attempts are logged. **System to user**: avoid the user talking to an impostor program (e.g. Windows NT's Ctrl-Alt-Del, which applications cannot trap).

Common Pitfalls – Protection

- Confusing which structure is *by object* (ACL) vs *by domain* (capability) – easy to mix up under exam pressure.
- Forgetting that dual-mode operation alone is not enough – you need the timer (CPU), base/limit registers (memory) *and* their interaction with I/O protection.
- Capabilities are unforgeable *because the OS mediates them* – a common wrong answer is to say they're just like a Unix file descriptor with no distinction (see §12.4 for exactly how fds resemble, and differ from, capabilities).

3 Processes

3.1 Process vs Program

A **program** is static, sitting on disk; a **process** is dynamic – a program in execution. A process is the unit of **protection** and **resource allocation**: several processes may run from the same program. Each process has **Text** (code), **Data** (globals), **Heap** (runtime allocation), plus one or more **threads of execution**, each with its own program counter and stack.

3.2 Process Control Block (PCB)

The PCB holds: process ID, current state, CPU-scheduling info (priority, queue pointers), memory-management info, accounting info (CPU used, elapsed time, limits), and I/O status (allocated devices, open files). The **process context** – program counter and CPU registers – is the machine environment while running.

3.3 Threads and Context Switching

A **thread** is an individual execution context; a process may have many. Each thread has a **Thread Control Block (TCB)** (saved registers including stack pointer, scheduler info, PC). A **context switch** saves the context of the currently executing thread and restores that of the one being resumed – pure overhead, no useful work is done while switching. Cost ranges from "nothing" to full hardware task-switch support. If the new thread belongs to a different process, the *process* state (e.g. page tables) must switch too.

3.4 Process Lifecycle

Process States

New – being created → **Ready** – waiting for the CPU → **Running** – executing → **Waiting/Blocked** – waiting for an event → **Terminated** – finished. (Compare the more detailed 5-state Linux task lifecycle in §11.4.)

Process creation (UNIX model)

Processes form a hierarchy (parent creates child, forming a tree). Three design axes: **resource sharing** (all / subset / none), **child memory initialisation** (duplicate parent, or load a fresh program), **execution ordering** (concurrent, or parent waits). On UNIX: `fork()` clones a child from the parent; `execve()` then replaces the child's memory image with a new program; the parent typically `wait()`s for the child to `exit()`. Windows NT/2K/XP instead uses a single `CreateProcess` call naming the program directly.

Process termination

1. Illegal operation (unauthorised memory access, privileged instruction) → killed by OS.
2. Parent terminates child (`kill`/`abort`) – e.g. resource limits exceeded, task no longer needed, or **cascading termination** when the parent itself exits.
3. Normal exit – process calls `exit()`; parent `wait()`s to collect status.

Zombie vs Orphan – do not confuse these!

Zombie: child has exited but the parent has *not yet* called `wait()` – the child's exit status lingers in the process table. **Orphan**: the *parent* terminated *before* the child, without waiting – the child is re-parented (traditionally to `init`).

3.5 Inter-Process Communication (IPC)

All communication needs a protocol: agreed **syntax**, **semantics**, and **synchronisation**. Two fundamental models:

- **Shared memory**: communicating processes map a common region – fast, but requires relaxing memory protection and explicit synchronisation by the processes themselves.
- **Message passing**: kernel mediates; requires naming (direct vs indirect via mailbox), synchrony (blocking vs non-blocking), and buffering (zero / finite / unbounded).

UNIX IPC Mechanisms

Signals – asynchronous notification, mapped to a small integer (architecture dependent); `kill` sends one, `sigaction` sets a handler, `pause` blocks until one arrives. Key signals: `SIGHUP`(1), `SIGINT`(2), `SIGILL`(4), `SIGKILL`(9, *cannot* be caught/ignored), `SIGTERM`(15), `SIGSEGV`(11), `SIGUSR1/2`.

Pipes – `pipe()` returns a pair of file descriptors (fd[0], fd[1]); combined with `fork()` gives parent/child a shared read/write channel. **Named pipes (FIFOs)** extend this beyond parent/child, appearing as files in the filesystem.

Shared memory segments – `shmget` obtains a segment, `shmat` attaches it; read/write via pointers (needs explicit synchronisation); `shmdt` detaches, `shmctl` destroys.

Exam Tip: Comparing IPC mechanisms

When asked to compare signals / pipes / named pipes (2019), structure your answer around: (i) *what* information can be carried (signals carry none beyond their identity; pipes carry a byte stream); (ii) *who* can communicate (signals/pipes: related processes only, via inherited fds; named pipes: unrelated processes, via the filesystem namespace); (iii) *synchronisation* semantics (pipes block on full/empty; signals are asynchronous and can be lost/coalesced).

4 Scheduling: Concepts

4.1 Queues

Job queue: batch processes awaiting admission. **Ready queue**: processes in main memory, ready and waiting for the CPU. **Wait queue(s)**: processes waiting for I/O or other events (one queue per device/resource is typical).

CPU–I/O burst cycle

Process execution alternates CPU bursts with I/O waits. **I/O-bound** processes have many short CPU bursts; **CPU-bound** processes have fewer, longer bursts. The burst-length distribution (often (hyper-)exponential) parameterises scheduling decisions.

Schedulers

Short-term (CPU) scheduler – picks which ready process runs next; invoked frequently (ms), must be fast. **Long-term (job) scheduler** – controls the degree of multiprogramming, choosing which jobs enter the ready queue; invoked rarely (s/min), can afford to be slower; aims for a good CPU/I/O-bound process mix.

Idling

If nothing is ready to run: (1) **busy-wait** in the scheduler – fast response, but wasteful; (2) **halt the CPU** until interrupted – saves energy, adds latency; (3) run an **idle process** – clean uniform structure, can do housekeeping, but consumes resources and may slow interrupt response.

4.2 The Dispatcher

After the scheduler decides, the **dispatcher** gives the CPU to the chosen process: switches context, switches to user mode, and resumes execution at the correct location. **Dispatch latency** is the time this stop/start takes.

When to enter the scheduler?

Four possible trigger points: (1) running → waiting (blocks); (2) running → terminated; (3) running → ready (timer expiry); (4) waiting → ready (unblocks).

Pre-emptive vs Non-preemptive

If the scheduler is entered *only* under (1) and (2), it is **non-preemptive**: the running process itself decides when to yield (explicit `yield` or implicit yield on I/O). Simple, no timer required – but open to denial-of-service by a greedy process.
If the scheduler can also be entered under (3) or (4), it is **pre-emptive**: the OS can force entry via a timer interrupt. Precludes DoS, but is more complex (timer management, concurrency issues). **Almost all modern schedulers are pre-emptive.**

4.3 Scheduling Criteria

Definitions – learn these precisely

Turnaround time – total time from submission to completion (across *all* states).
Waiting time – time spent specifically in the *Ready* queue (the only interval the scheduler actually controls).
Response time – time until the process *starts* responding (matters for interactive/time-sharing use).
CPU utilisation – fraction of time the CPU is actively executing (maximise).
Throughput – rate of process completions (maximise).

TRAP: These criteria trade off against each other

Maximising CPU utilisation can penalise I/O-heavy processes (they leave the CPU "idle" from its perspective). Maximising throughput can starve long-running processes. There is rarely a single objective function – always state *which* metric you are optimising, and whether you mean the *average*, the *maximum* (worst case), or the *variance/distribution*.

4.4 Multiple-Processor Scheduling

Asymmetric multiprocessing: only one processor touches the scheduling data structures (simple, avoids sharing). **Symmetric multiprocessing (SMP)** – the common case today: each processor self-schedules, either from

one shared ready queue or from private per-processor queues. **Processor affinity**: a process may prefer (*soft* affinity) or be constrained to (*hard* affinity) a given CPU, since migrating loses cache state.

NUMA

Non-Uniform Memory Access: different CPUs have faster/slower access to different parts of memory (e.g. combined CPU+memory boards). Memory placement then affects affinity; switching CPU can be far more expensive than under uniform access.

Load balancing

Push migration: a periodic task checks load and pushes work off overloaded CPUs. **Pull migration**: idle CPUs pull work from busy ones. Modern complications: **multicore** (multiple cores per chip), **hyperthreading** (multiple hardware threads per core, so one can progress while another stalls on memory), and **virtualisation** (hypervisor and guest OS schedulers compete against each other).

5 Scheduling Algorithms

5.1 First-Come First-Served (FCFS)

Processes run strictly in arrival order. Simplest possible algorithm, but not robust to arrival order – suffers the **Convoy Effect**: short jobs queue up behind one long job, badly hurting average waiting time.

Worked Example: FCFS & the Convoy Effect

Processes $P_1(24), P_2(3), P_3(3)$ (burst times).

Order P_1, P_2, P_3: waiting times $0, 24, 27$. Average $= \dfrac{0+24+27}{3} = 17$.

Order P_2, P_3, P_1: waiting times $6, 0, 3$. Average $= \dfrac{6+0+3}{3} = 3$.

Same processes, wildly different average waiting time purely due to *arrival order* – this is the convoy effect (imagine one CPU-bound process stuck behind, or ahead of, many I/O-bound processes).

5.2 Shortest Job First (SJF) & Shortest Remaining Time First (SRTF)

SJF: always run the process with the shortest *next* CPU burst; **provably optimal** for minimising average waiting time over a *known, fixed* set of processes. **SRTF** is the pre-emptive version: a newly-arrived process pre-empts the running one if its burst is shorter than the *remaining* time of the current process.

Worked Example: SJF

$P_1(6), P_2(8), P_3(7), P_4(3)$, all arrive at $t = 0$. Run order by burst length: P_4, P_1, P_3, P_2. Waiting times: P_4=0, P_1=3, P_3=9, P_2=16. Average $= \dfrac{3+16+9+0}{4} = 7$.

TRAP: SRTF is *not* unconditionally optimal

Even though SJF is optimal for a known process set, SRTF is *not* automatically optimal once new processes can arrive: (1) context switches are not free – a stream of arriving short bursts can cause thrashing between processes with no useful work done; (2) more fundamentally, **you cannot know a future burst length exactly** – you can only predict it (see exponential averaging below). A favourite "why not?" Tripos question.

Predicting burst length: exponential averaging

Exponential Averaging

Let t_n be the measured length of the n-th burst, and τ_{n+1} the predicted length of burst $n+1$:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n, \qquad 0 \le \alpha \le 1$$

Expanding: $\tau_{n+1} = \alpha t_n + \alpha(1-\alpha)t_{n-1} + \cdots + (1-\alpha)^{n+1}\tau_0$ – each term is weighted less than its predecessor.

- $\alpha \approx 1$: past history irrelevant $\Rightarrow \tau_{n+1} \approx t_n$ (trust the most recent burst).
- $\alpha \approx 0$: recent history irrelevant $\Rightarrow \tau_{n+1} \approx \tau_n$ (trust the long-run prediction).

Works well when burst-length variance is low (and not changing too fast relative to the time-slice). Also consider system *load* – otherwise priorities can counter-intuitively increase as load increases.

5.3 Round Robin (RR)

Each process gets a **quantum** (time-slice, e.g. 10–100ms); on expiry it is pre-empted and appended to the ready queue; a timer interrupts every quantum. Too large a quantum degenerates towards FCFS; too small makes context-switch overhead dominate (context switches take $< 10\mu$s, so q is usually 10–100ms).

Round Robin – Fairness & Liveness

For quantum q and n ready processes: **Fair** – each gets $1/n$ of CPU time, in chunks of at most q. **Live** – no process ever waits more than $(n-1)q$ time units. RR typically gives *higher* average turnaround time than SRTF, but *better* average response time.

5.4 Priority Scheduling

Associate an integer priority with each process; run the highest priority (usually *lowest* number). SJF is a special case: priority = inverse of predicted burst length.

TRAP: Starvation under static priorities

Low-priority processes may *never* run under static priority scheduling – famously, processes submitted to MIT's IBM 7094 in 1967 were still unrun when the machine was shut down in 1973! Fix with **dynamic priorities**, e.g. **aging**: priority increases the longer a process waits.

Computed priority (classical UNIX scheduler)

Priorities 0–127; user processes start at base = 50; round-robin within a priority level, quantum 100ms; priority recomputed every 4 ticks (∼40ms). Kernel-mode scheduling is fixed-priority and non-preemptive (modified by wait reason, e.g. disk I/O < terminal input); user-mode scheduling is dynamically computed and pre-emptive.

$$CPU_j(i) = \frac{2 \times load_j}{2 \times load_j + 1} CPU_j(i-1), \qquad P_j(i) = base_j + \frac{CPU_j(i)}{4} + 2\, nice_j$$

where $load_j$ is the sampled (1-minute) run-queue length and $nice_j \in [-20, 20]$ (default 0) is a user-controllable adjustment.

5.5 Multilevel (Feedback) Queues

Multilevel queues partition the ready queue by process type (e.g. foreground/interactive vs background/batch); each queue has its own algorithm (e.g. RR for foreground, FCFS for background) and each process is *permanently* assigned to one queue. Scheduling *between* queues can be fixed-priority (risks starving lower queues) or time-sliced (e.g. 80% to foreground RR, 20% to background FCFS).

Multilevel feedback queues additionally let a process *move* between queues (e.g. implementing aging), defined by: number of queues, algorithm per queue, promotion rule, demotion rule, and initial-queue rule.

Worked Example: Multilevel Feedback Queue

Three queues: Q_0 RR quantum 8ms, Q_1 RR quantum 16ms, Q_2 FCFS. A new job enters Q_0; if it doesn't finish within 8ms it drops to Q_1 for a further 16ms; if still incomplete it drops to Q_2. Short jobs finish fast (favoured); long jobs gradually sink to cheaper-to-schedule FCFS.

5.6 Linux: the Completely Fair Scheduler (CFS)

Since kernel 2.6.23, Linux abandons fixed time-slices. Assume N runnable tasks should each get $1/N$ of the CPU, adjusted by `nice` value (smaller = higher weight): task j runs for a slice $t_j \propto w_j / \sum_i w_i$.

CFS – Target Latency & Minimum Granularity

Target latency: the interval during which every runnable task should run at least once (e.g. target latency 10ms, 2 tasks ⇒ each gets 5ms; 10 tasks ⇒ each gets 1ms).
Minimum granularity: floor on a task's slice to bound context-switch overhead as the number of runnable tasks grows unboundedly (e.g. with 1000 tasks you cannot give each target-latency/1000).
CFS tracks each task's **vruntime** (virtual runtime, decaying faster for lower-priority tasks) and always schedules the task with the *lowest* vruntime, implemented as a red-black tree with the left-most node cached.

Worked Example: RR vs CFS (Tripos 2024 style)

Five CPU-bound processes arrive together with demands 10, 20, 30, 40, 50 ms.

- **RR,** $q = 5$**ms**: many small slices round-robin across all still-runnable processes; low response time for all, but many context switches (each process needs multiple visits: $\lceil 10/5 \rceil, \lceil 20/5 \rceil, \ldots$ visits respectively).
- **RR,** $q = 20$**ms**: fewer, larger slices; the 10ms process finishes on its first (partial) slice; better throughput, worse worst-case response time for later processes.
- **CFS, target latency 60ms, min. granularity 2ms**: with 5 runnable tasks each initially gets $60/5 = 12$ms per round – larger than RR's small quantum, so *fewer* context switches, closer to batch-friendly behaviour.
- **CFS, target latency 20ms, min. granularity 5ms**: $20/5 = 4$ms < min. granularity, so each task

actually gets the 5ms floor – more like RR($q = 5$), more context switches but better interactivity.
Rule of thumb for the exam: larger effective slices ⇒ better for *batch* throughput (fewer switches); smaller effective slices ⇒ better for *interactive* response time.

Exam Technique for Scheduling Numericals

1. **Always draw the Gantt chart** – even a simple text/ASCII timeline avoids arithmetic slips.
2. State *waiting time* = turnaround time − burst time = (time process finishes) − (arrival time) − (burst time).
3. When arrival times are staggered (not all $t = 0$), carefully track which processes are actually *in the ready queue* at each decision point – a common source of lost marks.
4. For RR, remember the quantum applies per *visit*, not per process – a process may need several visits to complete.
5. If asked for a "fair" comparison across algorithms, always state *which* metric (avg. waiting / turnaround / response, or # context switches) you are comparing.

6 Memory Management

6.1 Memory Protection

The CPU can only access registers and main memory directly (register access is single-cycle; memory access takes many cycles, hidden by L1/L2/L3 caches). The memory unit sees only address+read or address+data+write requests, so protection must prevent buggy or malicious processes corrupting others (including the kernel).

Base & Limit Registers

Base = smallest legal address; Limit = size of the range. E.g. base = 300040, limit = 120900 ⇒ legal range [300040, 420940). The CPU checks *every* user-mode memory access against this range; violations raise an exception to the OS. Only the OS may modify the base/limit registers.

Address binding

Programs on disk must be placed *somewhere* in memory – but the code refers to memory locations. Binding happens at one of three points:

- **Compile time**: absolute code generated if the location is known a priori; needs recompilation if the base changes.
- **Load time**: relocatable code generated if the location is unknown at compile time; bound when loaded.
- **Execution time**: binding delayed until run time, allowing the process to move between memory segments *during* execution; needs hardware support (e.g. a relocation register).

Logical vs physical addresses & the MMU

Logical (virtual) address: generated by the CPU. **Physical address**: seen by the memory unit. They are identical under compile-time/load-time binding, but differ under execution-time binding. The **Memory Management Unit (MMU)** maps logical to physical addresses at run time; conceptually, add a **relocation register**'s value to every logical address before it reaches memory – user programs never see physical addresses.

Dynamic linking and loading

Static linking: all libraries combined into the binary. **Dynamic linking**: postponed to execution time – calls are replaced with a *stub* that locates (and then replaces itself with) the routine's address; useful for shared/system libraries needing version tracking. **Dynamic loading**: routines aren't loaded until called, improving memory usage for rarely-used code (e.g. error handlers), but requires relocatable compilation.

6.2 Memory Allocation

Swapping

When requested memory exceeds physical memory, processes are temporarily moved to storage. Transfer time is proportional to memory swapped (e.g. a 100MB process at 50MB/s transfer rate is expensive), so swapping is disabled by default and enabled only past some allocation threshold; whether a process must return to the *same* physical address depends on the binding scheme.

Contiguous allocation & fragmentation

Main memory is split into **holes** (free blocks of varying size). On process arrival, allocate from a hole large enough; on exit, free the partition and coalesce adjacent free space.

Allocation Strategies

First-fit: allocate the first big-enough hole.
Best-fit: allocate the smallest big-enough hole (search entire list unless size-ordered; leaves the smallest leftover hole).
Worst-fit: allocate the largest hole (leaves the largest leftover hole).
First-fit and best-fit generally beat worst-fit on both speed and utilisation.

External vs Internal Fragmentation

External: free memory exists but is *not contiguous*, so a request cannot be satisfied even though total free space suffices – can eventually block swap-in entirely. **Internal**: allocated memory is slightly *larger* than requested – the excess is wasted but unusable by anyone else. Analysis of first-fit shows roughly $0.5N$ blocks of the N allocated are lost to fragmentation (the "50-percent rule"). **Compaction** fixes external fragmentation by shuffling memory contents together, but requires dynamic, execution-time relocation, and pinning any process doing I/O (or restricting I/O to OS buffers).

6.3 Segmentation

A memory-management scheme matching the *user's view* of a program: a collection of logical units (the program, a procedure, an object, an array, ...). A logical address is a pair $\langle$segment-number, offset$\rangle$. The **segment table** maps each segment to a **base** (physical start) and **limit** (length); the **Segment-Table Base/Length Registers (STBR/STLR)** locate the table and bound the number of segments (segment s legal iff $s <$STLR). Per-entry **validation** and **R/W/X** bits provide protection; because segments vary in length, allocating them is a dynamic storage-allocation problem (same issues as above).

Sharing segments is subtle

A jump within shared code specifies $\langle$segment-number, offset$\rangle$ – but different processes may assign the shared segment a *different* number in their own table! Fix by using PC-relative or current-segment-relative branches, or by giving every segment a unique **System Segment Number (SSN)**, with each process's table mapping a local **Process Segment Number (PSN)** to the SSN.

7 Paging

7.1 Non-Contiguous Allocation

Paging divides **physical** memory into fixed-size **frames** and **logical** memory into equally-sized **pages**; a **page table** maps pages to frames. This eliminates external fragmentation (any free frame will do) at the cost of (average $\frac{1}{2}$-frame) internal fragmentation.

Address Translation

For a 2^m logical address space and 2^n page size, split each logical address into a **page number** p (the top $m-n$ bits, indexing the page table to find the frame's base address) and a **page offset** d (the bottom n bits, combined with that base to form the physical address).

$$\underbrace{p}_{m-n \text{ bits}} \quad \underbrace{d}_{n \text{ bits}}$$

Worked Example: Internal Fragmentation

Page size 2048 bytes, process size 72,766 bytes $\Rightarrow$ $72766/2048 = 35$ pages plus a remainder of $72766 - 35 \times 2048 = 1086$ bytes needing a 36th page. Internal fragmentation on that last page $= 2048 - 1086 = 962$ bytes.

7.2 Paging Implementation

Two memory accesses & the TLB

The **Page-Table Base/Length Registers (PTBR/PTLR)** locate the (in-memory) page table. Naively, *every* data/instruction access now needs *two* memory accesses: one for the page table, one for the actual data – a dramatic slowdown. The **Translation Lookaside Buffer (TLB)** fixes this: a small (64–1024 entry), fast, associative hardware cache of recent translations. On a hit, skip the page-table lookup; on a miss, do the full two-access lookup and insert the new entry (evicting via, typically, LRU). Because the TLB must be flushed (or tagged with an **ASID**) on a context switch, it also adds switch overhead.

Effective Access Time (EAT) with a TLB

$$\text{EAT} = h \times (t_{\text{TLB}} + t_{\text{mem}}) \; + \; (1-h) \times (t_{\text{TLB}} + 2\,t_{\text{mem}})$$

where h is the TLB hit ratio, t_{TLB} the TLB search time, and t_{mem} the memory access time (a hit still costs $t_{\text{TLB}} + t_{\text{mem}}$: check TLB, then fetch data; a miss costs t_{TLB} (failed check) + one access for the page table entry + one access for the data).

Worked Example: TLB Effective Access Time

TLB search time 20ns, memory access time 100ns, hit ratio 80%:

$$\text{EAT} = 0.8 \times (20 + 100) + 0.2 \times (20 + 2 \times 100) = 0.8 \times 120 + 0.2 \times 220 = 96 + 44 = 140\text{ns}$$

If the hit ratio rises to 98%:

$$\text{EAT} = 0.98 \times 120 + 0.02 \times 220 = 117.6 + 4.4 = 122\text{ns}$$

Only a ~13% improvement despite the hit ratio jumping from 80% to 98% – diminishing returns, because misses are so much more expensive than hits that even a few dominate the average.

TRAP: Multi-level page tables need MORE memory accesses per miss

With an L-level page table, a TLB *miss* costs L memory accesses to walk the levels, plus one for the data: $\text{EAT} = h\,t_{\text{mem}} + (1-h)(L+1)t_{\text{mem}}$ (ignoring TLB search time for simplicity, or add it as above). This is exactly the calculation behind 2020's and 2025's page-table-levels-vs-EAT questions – always count how many levels the address must walk on a miss.

Protection & sharing

Protection bits live in the PTE (kernel/user-only, R/W/X, valid/invalid) – this is *page*-granularity protection, not byte-granularity. An invalid access traps to the OS. **Sharing**: read-only (reentrant) code can have a single physical copy mapped into many processes' page tables (like threads sharing a process, or useful for read-write IPC); private code/data gets its own copy, placed anywhere in the logical address space.

7.3 Page Table Structure

TRAP: Naive page tables are enormous

32-bit address space, 4KB (2^{12}) pages $\Rightarrow 2^{32}/2^{12} = 2^{20} \approx 1$ million entries; at 4 bytes/entry that's 4MB *per process*, which you don't want to allocate contiguously. Solution: split the page table into **multiple levels** and page out all but the outermost level.

Two-Level (and Multi-Level) Paging

Given a 20-bit page number split into two 10-bit halves p_1, p_2 and a 12-bit offset d (4096-byte pages): PTBR $\rightarrow$ L1 table, indexed by p_1, gives the address of the relevant L2 page (a *forward-mapped* page table); p_2 indexes that L2 page to give the frame address; d then indexes into the frame. General k-level scheme: each level adds one extra memory access on a TLB miss, but only the outermost level need be resident at all times.

Worked Example: Sizing a Multi-Level Page Table (Tripos 2025 style)

64-bit machine, 57-bit virtual addressing, 5-level page table, 9 bits per level, 4096-byte (2^{12}) pages, 64-bit (8-byte) PTEs.

- Check: $5 \times 9 + 12 = 45 + 12 = 57$ bits – consistent with the given virtual address width.
- Each level table has $2^9 = 512$ entries, so is $512 \times 8 = 4096$ bytes = 4KB – conveniently exactly one page, so each page-table page is itself page-sized and easily allocated/paged.
- Addressable memory using the largest applicable SI unit: 2^{57} bytes $= 2^{57}/10^{18} \approx 144$ **petabytes** (using SI, not binary, prefixes as the question specifies).
- Address decomposition of a 57-bit address (from MSB to LSB): 9 bits for each of L1..L5 index, then 12 bits offset.

Larger address spaces (non-examinable detail, examinable at a high level)

For 64-bit spaces a simple hierarchy becomes impractical (either some level is huge, or too many accesses are needed). Alternatives: **hashed page tables** (hash the page number, follow a chain) and **inverted page tables** (one entry per *frame*, with a hash table limiting search – trades table size for lookup latency). Real examples: **Intel IA-32** hybridises segmentation (LDT/GDT) with two-level 4KB paging, or 4MB pages bypassing the inner level (PAE extends this to 36-bit addressing via a 3-level scheme). **x86-64** implements 48-bit addressing with four paging levels (page sizes 4kB/2MB/1GB); PAE variants give 52-bit physical addresses. **ARM** uses one-level paging for large 1MB/16MB "sections" and two-level for 4kB/16kB pages, with a two-tier TLB (micro TLBs supporting ASIDs, feeding a main TLB).

Exam Technique: Address-Translation Questions

1. Write down the bit-width equation first: (levels $\times$ bits/level) + offset bits = total virtual address bits – this catches most sizing errors immediately.
2. Compute each level's table size in *both* entries and bytes; check whether it happens to equal exactly one page (it often does, by design).
3. For EAT questions, always state explicitly whether a TLB *hit* needs 1 memory access (page already resident, no page-table walk) and a *miss* needs (levels)+1.
4. When comparing two systems (e.g. "should you accept this replacement hardware?"), compute EAT (or effective *bandwidth*, if asked) for both and compare like-for-like – don't just eyeball the individual numbers.

8 Virtual Memory

8.1 Virtual Memory Basics

Extend the valid/invalid page distinction with a **non-resident** designation: such pages live on non-volatile backing store, and processes access them exactly as if resident. This decouples logical from physical memory, letting the logical address space exceed physical memory, implemented via **demand paging** (and demand segmentation).

Benefits of Virtual Memory

Portability: programs run regardless of physical memory size, can exceed it, and can start before being fully loaded. **Convenience**: less must be resident at once ⇒ more multiprogramming, less I/O, easy large sparse structures. **Efficiency**: no memory wasted on rarely-used code/data (e.g. error handlers).

The virtual address space usually starts at 0 and grows contiguously; the stack starts at the top and grows down while the heap grows up, maximising use of the address space – the gap between them is a **hole** needing no physical memory until touched. This enables sparse address spaces, dynamically linked libraries, and (via `fork` / shared mappings) efficient sharing.

8.2 Page Faults

Referencing an invalid page traps to the OS (a **page fault**). The OS checks another table: if the reference is genuinely illegal, abort; if valid but non-resident, find a free frame and swap the page in, then mark the entry valid, then **restart the faulting instruction**.

TRAP: Instruction restart is not always simple

Simple case (e.g. `add`): re-fetch, re-decode, re-execute from scratch – fine, since locality makes multiple faults per instruction unlikely. Harder case: an instruction spanning several locations (e.g. a block move where source/destination overlap or straddle a page boundary) – the source may already be partially modified, so it *cannot* simply restart from the top; handle via microcode that touches every page first to guarantee no mid-instruction fault. A **double fault** (the fault handler itself faults) is simply fatal.

Locality of reference: in any short interval, references cluster into a few regions (current procedure, its sub-procedures, data/stack accesses) – this is precisely what makes demand paging (and caching generally) effective.

Demand Paging Performance – Effective Access Time

Let p be the page-fault rate ($0 \le p \le 1$), t_{mem} the (no-fault) memory access time, and t_{fault} the (large) page-fault service time:

$$\text{EAT} = (1-p)\,t_{\text{mem}} + p\,t_{\text{fault}}$$

Worked Example: Demand Paging EAT

$t_{\text{mem}} = 200\text{ns}$, $t_{\text{fault}} = 8\text{ms} = 8{,}000{,}000\text{ns}$.

$$\text{EAT} = (1-p)(200) + p(8{,}000{,}000) = 200 + 7{,}999{,}800\,p$$

If $p = 0.001$ (1 in 1000 accesses faults): EAT $= 200 + 7999.8 = 8199.8\text{ns} \approx 8.2\mu\text{s}$ – a **40**× slowdown versus 200ns!
To keep degradation under 10% requires EAT ≤ 220ns, i.e. $220 \ge 200 + 7{,}999{,}800\,p \Rightarrow p < 0.0000025$, i.e. **fewer than 1 fault per 400,000 accesses**. This dramatically illustrates why the page-fault rate must be kept vanishingly small.

Optimisations

Swap-space I/O can outperform filesystem I/O even on the same device; allocate it in large chunks and copy the whole process image at load time. **Copy-on-Write (COW)**: parent and child share pages after `fork`; a page is only copied when *written*, making process creation cheap. `vfork` takes this further: the child runs COW against the parent's address space, extremely efficient when the child immediately calls `exec`. Maintain a pool of **zero-fill-on-demand** pages so a fault never needs to *both* evict a victim *and* zero a fresh page.

8.3 Page Replacement

When no free frame exists, the fault handler must: locate the desired page on disk; select a victim frame (write it back if dirty, mark it invalid); read the desired page in; restart the process. No free frames roughly *doubles* the page-fault service time (one write-out plus one read-in).

Worked Example: FIFO, OPT, LRU (reference string $7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1$; 3 frames

FIFO: replace the longest-resident page ⇒ 15 faults. Exhibits **Bélády's Anomaly**: *adding more frames can increase* the fault count.
OPT: replace the page not needed for the longest future time ⇒ 9 faults (the theoretical best; needs an oracle, so useful only as a benchmark).
LRU: replace the least-recently-used page (approximates OPT by assuming the past predicts the future) ⇒ 12 faults – better than FIFO, worse than OPT.

TRAP: Which algorithms suffer Bélády's Anomaly?

FIFO **can** exhibit Bélády's Anomaly. LRU and OPT **cannot**, because both are **stack algorithms**: the set of pages held with k frames is always a *subset* of the set held with $k+1$ frames, so the replacement order is independent of the number of available frames. FIFO has no such subset property. This exact reasoning was asked directly in 2024.

Implementing LRU (exactly, and approximately)

Exact: a **counter** per PTE updated on every reference (replace smallest counter – needs a full table search plus a memory write per access), or a **stack** of page numbers moved-to-top on each reference (up to six pointer updates, tail is always the victim). Both are expensive in hardware.

Approximating LRU: reference bits, NRU, and Clock

Use one **reference bit** per PTE (0 initially, set to 1 on touch); periodically (e.g. every 20ms) clear all reference bits. **Not-Recently-Used (NRU)**: rank victims by (reference, dirty) bit pairs – *(0,0)* best to evict, *(0,1)* next, *(1,0)* probably in-use code, *(1,1)* worst choice. Can extend to an 8-bit shift-in history for finer granularity.
Second-Chance (Clock): a circular FIFO queue with a current pointer; if the pointed-at page's reference bit is 0, evict it; else clear the bit (give it a "second chance") and advance – guaranteed to terminate within one full cycle, degenerating to plain FIFO if every page is referenced.

Exam Tip: Emulating reference/dirty bits without hardware support

Mark the page *no-access*; any reference then traps, letting the OS software-update the reference bit and resume (checking the requested access mode against actual permissions distinguishes a read, indicating "referenced", from a write, indicating "dirty" too). This exact mechanism was asked in 2024.

Counting algorithms & page buffering

LFU (evict smallest reference count) ignores recency, so pages can wrongly stick around from initialisation (needs periodic decrement). **MFU** (evict largest count, on the theory a low count means recently brought in) – neither is common: both are expensive and approximate OPT poorly. **Page buffering**: keep a minimum pool of free frames ready (read in *before* choosing/evicting a victim); optionally keep evicted-but-unmodified frame contents around a little longer in case they're re-referenced.

8.4 Frame Allocation

Need a policy to distribute frames between processes after reserving OS and per-process minimums. Possible objectives: **fairness** (equal m/n split, or proportional to process size), **minimise system-wide fault rate** (may starve some processes of frames), or **maximise multiprogramming** (spread thin).

Global vs Local Replacement

Global: any page (from any process) may be a victim – higher throughput, but a process's own performance can depend entirely on what *other* processes are doing (it cannot control its own fault rate).
Local: a process may only evict its own pages – more predictable per-process performance, but risks under-utilising memory.

Thrashing & the Working Set

Thrashing: a process without "enough" frames faults constantly, evicting pages it will need again almost immediately. This can cascade: fault-handling time lowers CPU utilisation → the OS admits *more* processes to compensate → memory pressure rises further → the system collapses.

Working Set Model

The **working set** is the set of pages a process needs simultaneously resident to make progress. Over a window Δ of (say) 10,000 recent instruction references, WSS_i is process i's working-set size. Δ too small misses the whole locality; too large spans multiple localities (approaching the entire program). **Demand** $D = \sum_i WSS_i$; **thrashing occurs iff** $D > m$ (available frames), in which case suspend/swap out a process to relieve pressure. Approximate via an interval timer plus reference bits; **pre-paging** brings in the working set proactively when a process (re-)starts.

9 I/O Systems

9.1 The I/O Subsystem

An enormous variety of devices (human-readable, machine-readable, communications) differ in data rate, control complexity, transfer unit/direction, data representation, and error handling – yet share commonalities: a **port** connection point, interconnection via a **bus**, and a **controller** (host adapter) with its own processor/microcode/memory. Devices have registers (data-in, data-out, status, control) accessed either via privileged **direct I/O instructions** or via **memory-mapped I/O** (device registers appear in the normal address space – common for large-data devices like graphics cards).

Polling

For each byte: host busy-waits on device-busy, sets command + data, sets command-ready; device notices, sets device-busy, performs the operation, clears command-ready/device-busy. Simple, but step 1 (polling) wastes CPU if the device is slow, and risks lost/overwritten data if the CPU is switched away and misses a cycle.

Interrupts

More efficient when devices are accessed relatively infrequently. The device asserts an interrupt-request line, checked by the CPU after each instruction (aligning with instruction boundaries); the **interrupt vector** dispatches to the correct handler (with a context switch either side, priority ordering, and possibly non-maskable interrupts).

Top Half / Bottom Half Interrupt Handling

Top half (interrupt handler): processor-dependent; saves registers, establishes a language environment, demultiplexes in software to invoke the relevant device-specific **ISR**. **Bottom half** (the ISR itself): device-dependent; for programmed I/O, transfers data and clears the interrupt; for DMA, acknowledges the transfer, requests more if pending, signals waiting processes, and enters the scheduler or returns. Splitting this way keeps critical (top-half) sections short.

Direct Memory Access (DMA)

For high-speed devices, let the controller transfer blocks *directly* to/from main memory without CPU intervention, given a source address, sink address, and transfer size (each with increment/decrement/none). This generates only **one interrupt per block** rather than one per byte. Requires the device to act as a **bus master** (needs bus arbitration, since more than just the CPU now drives the bus) and involves **cycle stealing** (taking bus cycles away from the CPU). **Scatter/Gather DMA** chains multiple requests (e.g. several disk reads into a set of buffers) in one operation.

9.2 I/O Devices

Block devices (disks, CDs): commands include read/write/seek; accessed raw, via a filesystem ("cooked"), or memory-mapped. **Character devices** (keyboards, mice, serial): commands include get/put, with line-editing etc. layered on top. **Network devices**: different enough to warrant their own interface (e.g. Berkeley Sockets on UNIX/NT). **Miscellaneous**: clocks/timers; UNIX's `ioctl` covers everything else.

Blocking, Non-blocking, and Asynchronous I/O

Blocking: process suspended until I/O completes – simple, but insufficient for some needs.
Non-blocking: the call returns immediately with whatever data is available (possibly a 0-byte count); relies on multi-threading, often paired with `select`.
Asynchronous: the process continues running *while* I/O proceeds, and the I/O subsystem explicitly signals completion – most flexible and potentially most efficient, but most complex to use.

Synchronous I/O structure: control returns to the user program only on completion (at most one outstanding request; a wait loop contends for memory access). **Asynchronous I/O structure**: control returns immediately; a **device-status table** (type, address, state per device) lets the OS track outstanding requests and update state on interrupt.

9.3 Kernel Data Structures

The kernel must track open files, network connections, and character-device state – complex, performance-critical structures for buffers, allocation, and dirty blocks.

Vectored I/O & Buffering

Vectored I/O (e.g. UNIX `readv`/`writev`) lets one system call perform many I/O operations via a vector of buffers – reduces context-switch/syscall overhead versus many individual calls, and some variants provide atomicity.
Buffering strategies: **single** (one system buffer per request), **double** (consume from one buffer while the system fills the next), **circular** (best for bursty I/O). Buffering smooths short-term rate mismatches but cannot help if the *long-term average* demand exceeds supply (or vice versa) – and it can introduce **jitter**, bad for real-time/multimedia use.

Other cross-cutting issues: **caching** (fast copies for reads *and* writes – critical to performance), **scheduling** (per-device request ordering, sometimes with fairness guarantees), **spooling** (queue output for single-user devices like printers), **device reservation** (exclusive-access system calls), **error handling** (logged error codes), **protection** (all I/O instructions/memory-mapped regions must be privileged, forcing access via system calls), and raw **performance** (CPU time spent on drivers, kernel I/O code, interrupt-driven context switches, and data copying).

Worked Example: Buffering Adequacy (Tripos 2020 style)

An application consumes at a constant 1MB/s; the network delivers at a long-term average of 1MB/s but bursts to 5MB/s for up to 2s.

- **Single 1MB buffer, no simultaneous R/W**: during a burst, incoming data cannot be written while the application is still reading out the previous buffer $\Rightarrow$ data is dropped or the sender must stall.
- **Double-buffering, 512kB each**: while one buffer fills, the other drains – but during a 5MB/s burst a 512kB buffer fills in 512kB/5MB/s $\approx$ 102ms, likely faster than the consumer can drain the other, so buffers can still overrun on a sustained burst.
- **Circular buffer, 512kB chunks**: to survive a full 2s burst at the 4MB/s *excess* rate (5MB/s in $-$ 1MB/s out), need to absorb 4MB/s $\times$ 2s = 8MB, i.e. 8MB/512kB = 16 buffers, to avoid data loss.

This is exactly the reasoning examiners want: compute the *excess* rate during the burst, multiply by burst duration, divide by buffer chunk size.

10 Storage & File Management

10.1 Mass Storage

Hard disks: stacks of platters (historically 0.85"–14", commonly 1.8"–3.5"); real transfer rates ∼1Gb/s (theoretical ∼6Gb/s); seek time 3–12ms (∼9ms typical); rotation 7200 or 15,000 RPM. **SSDs**: non-volatile, no moving parts (no seek/rotational latency), faster and more reliable, but cells wear out with use, cost more per MB, and offer lower capacity.

Hard Disk Performance

$$\text{Average latency} = \tfrac{1}{2} \times \frac{60}{\text{RPM}} = \frac{30}{\text{RPM}} \quad \text{[seconds]}$$

$$\text{Access latency} = \text{Average seek time} + \text{Average latency}$$

$$\text{Average I/O time} = \text{Access latency} + \frac{\text{transfer amount}}{\text{transfer rate}} + \text{controller overhead}$$

Worked Example: Average I/O Time

4kB block, 7200 RPM, 5ms average seek, 1Gb/s transfer rate, 0.1ms controller overhead.

$$\text{Average latency} = \frac{30}{7200} = 4.17\text{ms}$$

$$\text{Transfer time} = \frac{4096 \times 8 \text{ bits}}{1024^3 \text{ bits/s}} = 0.031\text{ms}$$

$$\text{Average I/O time} = 5 + 4.17 + 0.031 + 0.1 = 9.30\text{ms}$$

10.2 Disk Scheduling

The disk controller must order a sequence of read/write requests – analogous to CPU scheduling but with very different mechanisms and policy goals (physical head movement, not abstract CPU time).

Disk Scheduling Algorithms

FCFS: service requests in arrival order – fair but potentially very inefficient (large head movements).
Shortest Seek-Time First (SSTF): service the request closest to the current head position (analogous to SJF) – a big improvement, but **can starve** far-away requests, and is not globally optimal (a locally-greedy choice can force a larger total movement later).
SCAN (elevator): sweep from one end of the disk to the other, servicing everything en route, then reverse.
C-SCAN: like SCAN, but always sweeps in one direction, then jumps back to the start without servicing on the return – gives more *uniform* wait times (density of requests just serviced when changing direction is high; furthest-away requests have waited longest, so C-SCAN avoids re-favouring the near end twice in a row).

10.3 Disk Management

Low-level (physical) formatting divides a disk into sectors (header + data + ECC, typically 512B of data). **Logical formatting** lays down filesystem structures; a disk is partitioned into groups of cylinders, each a logical disk; filesystems group blocks into **clusters** for efficiency (disk I/O is in blocks, file I/O is in clusters – though some applications, e.g. databases, prefer raw block access).

Booting from disk & on-disk structures

BIOS/UEFI is firm-coded to read a known first block containing the bootloader, which then reads the partition table and (possibly via chain-loading) enough code to parse the real filesystem, handling bad blocks via **sector sparing**. A disk = boot block + partitions; a partition (a UNIX filesystem, say) is laid out roughly as `|data blocks|` ≫ `|inode table|` ≫ `|superblock|`, where the **superblock** holds metadata (total/free blocks, free-block/free-inode list heads) – critical, so replicated to survive head crashes.

10.4 Files

The basic non-volatile storage abstraction, typically a single contiguous logical address space, with type (data/program/docume
and optional internal structure (none, simple records, or complex formats).

File Metadata & Operations

Metadata: type, location, size, protection, timestamps, owning user. The OS also tracks an **open-file table**: file pointer/cursor (per open handle), open count (to know when to evict the entry), on-disk location cache, and per-process access rights.
Directory operations: create (allocate blocks), open/close (handle + cursor), delete (return blocks to free list), stat (retrieve metadata).
File operations: write (at cursor), read (at cursor), truncate (clip to cursor).
Access pattern: random access permits `seek`; sequential access permits only `rewind`.

Reading a file resolves: **fd** (per-process open-file table index) → **system-wide open-file table** entry → **file control block** → actual data blocks on disk.

10.5 Directories

Must provide **grouping** (related files together), **naming** (convenience – distinct files with the same name, one file with many names), and **efficiency**. **Single-level**: simplest, but naming/grouping conflicts. **Two-level** (e.g. FAT): per-user namespaces, efficient search, but no grouping. **Tree-structured**: naming convenience, efficient search, *and* grouping, via a **current working directory (CWD)** and absolute/relative paths.

Acyclic-Graph Directories – what has to be managed

Generalising to a DAG allows shared subdirectories/files (**aliasing**: one file, two absolute names), but raises three problems: (1) *when to delete* – use back-references or **reference counting** (cf. UNIX hard- vs soft-links); (2) *who owns the storage* – for deletion and permissions; (3) *avoiding cycles* – typically by forbidding hard-links to directories.

Filesystem mounting: an unmounted filesystem (e.g. on another partition) is attached at a **mount point**, overlaying whatever was previously at that path in the existing tree.

10.6 Other Issues

Consistency

Even single-threaded, deleting a file (`unlink`, invoked by `rm`) must: check write permission on the file *and* its containing directory; remove the directory entry; decrement the inode's reference count; if zero, free the data blocks and inode. After a crash, `fsck` must check for unreferenced blocks (mark free) and double-referenced blocks (fix reference counts).

Efficiency & the buffer cache

Efficiency depends on allocation/directory algorithms (fragmentation, compaction – same challenges as memory), metadata types stored (creation vs. modification vs. access time), and pre-allocation vs. as-needed metadata. A **buffer cache** (separate memory region for frequently-used blocks) boosts performance; **synchronous writes** bypass the cache (must hit disk before acknowledgement – needed by some apps/OS invariants); **asynchronous writes** are buffered and faster.

Unified vs Non-unified Buffer Cache

Non-unified: a **page cache** (virtual-memory-addressed, for memory-mapped I/O) is separate from the **buffer (disk) cache** (for routine filesystem I/O) – risk of the same data being cached twice, inconsistently.
Unified: a single page cache serves both memory-mapped and normal disk I/O – the modern approach (see Linux, §12.5).

11 UNIX/Linux Case Study I: Structure & Processes

11.1 UNIX Key Features & History

UNIX separates the kernel from user space (editors, compilers etc. are just applications); processes are the unit of scheduling *and* protection (even the shell is just another process); and, famously, **"everything is a file"** – all I/O looks like file operations. Developed 1969 by Thompson & Ritchie at Bell Labs as a reaction to the bloated Multics; rewritten in (portable) C by 1973. From 1978, two main families diverged: **System V** (AT&T) and **BSD** (Berkeley); **POSIX** later attempted to re-unify them. **Linux** (Torvalds, 1991) is an original, UNIX-compatible kernel – now the most common UNIX-family OS – with distributions layering package management/tooling on top.

11.2 Components of a Linux System

Kernel: maintains core OS abstractions; runs in kernel mode with full hardware access; all kernel code/data share one address space. **System libraries**: standard functions apps use to talk to the kernel, implementing functionality that doesn't need kernel privilege. **System utilities**: the rich set of user-mode management programs.

Kernel Modules

Sections of kernel code compiled, loaded, and unloaded independently (device drivers, filesystems, network protocols); lets a system ship with a minimal kernel plus only the drivers it needs, and lets third parties distribute non-GPL components. Dynamic loading requires conflict resolution when modules compete for the same hardware.

11.3 Processes in Linux

UNIX separates process *creation* (`fork`) from *running a new program* (`exec`) – two distinct operations. Process properties fall into three groups: **Identity** (PID; credentials – UID and one or more GIDs; *personality*, for compatibility emulation; *namespace*, giving a specific filesystem-hierarchy view); **Environment** (argument vector and `NAME=VALUE` environment vector, both inherited from the parent); **Context** (the constantly-changing state of the running program).

Process Context

Scheduling context (most important) – lets the scheduler suspend/resume, plus accounting info. **File table** – an array of pointers into kernel file structures, indexed by **file descriptor**. **Filesystem context** – current root and default (working) directories for new searches. **Signal-handler table** – per-signal handling routine. **Virtual-memory context** – the full private address-space contents.

Processes and threads: `fork` vs `clone`

Internally, a thread is "just" a new process sharing its parent's address space – both are called **tasks**, distinguished only in how they're created. `fork` creates a task with an entirely new context; `clone` creates a task with its own identity but *sharing* chosen parent data structures (filesystem, memory space, signal handlers, open files) – giving fine-grained control over exactly what a "thread" shares.

11.4 Task Lifecycle

Five Linux task states: **Running/Runnable (R)**, **Uninterruptible Sleep (D)** (waiting for resources, can't be woken by a signal), **Interruptible Sleep (S)** (waiting, can be woken by a signal), **Stopped (T)** (`SIGSTOP` received, resumes on `SIGCONT`), **Zombie (Z)** (exited, awaiting parent's `wait()`).

Exam Tip: Transitions & the Scheduler (Tripos 2019 style)

A running *user-mode* process interrupted by the timer, having used its slice, transitions running(user) → ready/runnable – the pre-emptive scheduler is entered. A process invoking a *non-blocking asynchronous* I/O call typically transitions running(user) → running(kernel) → running(user) without ever leaving the CPU – the scheduler need *not* be entered. A *single-threaded blocking synchronous* I/O call transitions running(user) → running(kernel) → (interruptible) sleep – the scheduler *is* entered so something else can run.

11.5 Task Scheduling

Traditional UNIX used fixed 100ms time-slices with round-robin within priority levels – adequate for early time-sharing but poor for interactive performance at scale. Since kernel 2.6.23, Linux uses **CFS** (see §5.7): no fixed slices, per-task `vruntime`, red-black tree, target latency, minimum granularity.

11.6 Kernel Synchronisation

Kernel-mode execution is entered either via an explicit/implicit system call (e.g. page fault), or via a hardware interrupt invoking a driver's handler. Kernel critical sections must not be interrupted by *another* critical section. Pre-2.6: the kernel is non-preemptible, so a timer interrupt merely sets `need_resched` for later. Post-2.6: spin locks, or explicit enable/disable of pre-emption.

Top/bottom-half interrupt handlers

To avoid disabling interrupts for long critical sections, split ISRs: the **top half** runs as a normal ISR with recursive interrupts disabled; the **bottom half** runs later, with all interrupts enabled, scheduled by a mini-scheduler that guarantees bottom halves never self-interrupt (plus a mechanism to selectively disable bottom halves while running foreground kernel code).

11.7 Linux IPC

Signals: process-to-process, limited in number, carry no payload beyond which signal fired. **Wait queues** (kernel-internal): a process joins a queue for an event and tells the scheduler it's ineligible to run; all waiters wake when the event completes. **Pipes**: just another VFS inode type, with a pair of wait queues (reader/writer) for synchronisation. **Shared memory**: fast, persistent, but has *no* built-in synchronisation – must be added separately.

12 UNIX/Linux Case Study II: Memory, Filesystems & I/O

12.1 Physical Memory Management

Deals with allocating/freeing pages, groups of pages, and small blocks, split into hardware-characteristic **zones** (e.g. `DMA`, `DMA32`, `NORMAL`, `HIGHMEM` on x86_32 – some devices can only address the lowest 16MB; `HIGHMEM` is memory not mapped into kernel space). The **buddy-heap page allocator** pairs each allocatable region with an adjacent partner: two freed partners are combined into a larger region; a too-large free region can be recursively subdivided to satisfy a smaller request. For sub-page allocations, the kernel uses a **slab allocator**.

12.2 Virtual Memory in Linux

The VM manager maintains two views: a **logical view** (non-overlapping, page-aligned Virtual Memory Areas/regions) and a **physical view** (the actual hardware page tables). Each region has a **backing store** (a file, or "demand-zero" for nothing) and a **write reaction** (shared, or copy-on-write).

New Address Space Creation

Via `exec`: the process gets a completely empty address space, which program-loading routines then populate with VM regions.
Via `fork`: the child gets a *copy* of the parent's address space – VMA descriptors are copied, new page tables created, and the parent's page-table *entries* copied in with reference counts incremented, so parent and child *initially* share the same physical pages (until COW triggers a real copy on write).

Running a program (ELF loading)

The kernel's function table supports multiple binary formats, commonly **ELF**: a header plus page-aligned sections, mapped into VM regions and faulted in on demand. Unless statically linked, the dynamic linker resolves external symbols at load/run time; shared libraries are typically compiled **position-independent (PIC)** so they can load anywhere.

12.3 Linux File Systems

The user sees a single hierarchical directory tree; devices appear as special files, and `/proc` computes "file" contents on demand rather than storing data. The **Virtual File System (VFS)** abstracts over concrete filesystems via four object types: **inode** (a file), **file** (an open file), **superblock** (an entire filesystem), **dentry** (a directory entry) – each with an operations table (`open`, `read`, `write`, `mmap`, ...).

Inodes as File Control Blocks

UNIX filesystems use **inodes** as FCBs: a *combined* scheme where the inode holds direct pointers to data blocks, plus pointers to *indirection* blocks (which point to further blocks, or further indirection blocks). Alternative: purely linked schemes, where each index block points to data blocks and ends with either null or a pointer to the next index block.

Worked Example: Maximum File Size from Inode Structure (Tripos 2021 style)

An inode has 12 **direct** block pointers, plus **single**, **double**, and **triple** indirect pointers; each indirect block holds 256 addresses; disk blocks are 4KB.

$$\underbrace{12}_{\text{direct}} + \underbrace{256}_{\text{single indirect}} + \underbrace{256^2}_{\text{double indirect}} + \underbrace{256^3}_{\text{triple indirect}} \text{ blocks}$$

$$= 12 + 256 + 65{,}536 + 16{,}777{,}216 = 16{,}843{,}020 \text{ blocks}$$

$$\text{Max file size} = 16{,}843{,}020 \times 4\text{KB} \approx 64.3\,\text{GB}$$

Disk access counting: to open `/home/user1/test/test1.html` (4 data blocks), with `/`'s inode already cached in memory, you must read: `home`'s inode+data, `user1`'s inode+data, `test`'s inode+data, `test1.html`'s inode, then its 4 data blocks – carefully count *each* inode read *and* each directory-data-block read along the path. Reading a *second* file (`test2.html`) immediately after needs *fewer* accesses, since the intervening directories' inodes/data are now warm in the **buffer cache**.

Directories and links

A directory is just a file (pointed to by an inode) mapping names to inodes. A **hardlink** is an instance of a file in a directory, reference-counted in the inode, freed when the count hits zero; directories cannot have more than one hardlink (else cycles could form). A **softlink**/**symlink** is instead a normal file containing a filename, interpreted by the filesystem at traversal time.

In-memory table chain

Per-process **file descriptor** → per-process open-file table → system-wide open-file table (shared if multiple processes hold the same open file) → in-memory **inode table**.

12.4 Access Control

Every object (file, process, ...) uses **UID**/**GID** numeric identifiers. A process has a single UID but one or more GIDs; rights are resolved in order: owner UID match → user rights; else GID match → group rights; else → world rights. Root's UID has automatic access to everything. Rights can be *passed* between processes by forwarding file descriptors over a local socket (e.g. a print server receiving just the one fd it needs, without broader filesystem access).

File Permission Bits & `setuid/setgid`

Each inode stores 3 bits for each of owner/group/world: R/W/X for files; read-entry/write-entry/traverse for directories. **setuid**/**setgid** let a process temporarily assume the permissions of the file's owner/group while it runs (rather than the invoking user's own) – e.g. an exam application owned by the examiner with mode `0711`+setuid can write a separately-owned, otherwise-inaccessible `0600` scores file.

Worked Example: UNIX Groups & Permissions (Tripos 2021 style)

Users `user1, user2, user3`; groups `group1`={u1,u2}, `group2`={u2,u3}, `group3`={u3,u1}. Files:

Permissions	Owner:Group	File
`rw-rw--`	user1:group1	file1
`rw-r-r-`	user2:group3	file2
`rwxr---`	user3:group2	file3

user1 can read: file1 (owner rw), file2 (world r) – *not* file3 (not owner, not in group2, no world bits).
user2 can write: file1 (in group1, group has w) – *not* file2 (owner-only w, user2 *is* the owner so actually yes: owner write) – so file1 and file2; not file3.
Who can read file3: user3 (owner) and anyone in group2 (user2) – so {user2, user3}.
To let user2 additionally *execute* file3 while keeping everything else: since user2 ∈ group2, adding the group-execute bit (`rwxr-x--`) suffices. To let user2 execute it *as user3*: additionally set the **setuid** bit (`rwsr-x--`).

12.5 I/O in Linux

Two caches sit in front of disk storage: the **page cache** (unified with the VM system, caches data) and the **buffer cache** (caches metadata, indexed by physical disk block). Three device classes: **block** (random access, fixed-size blocks), **character** (most other devices; terminals use a *line discipline* to interpret data, e.g. the `tty` discipline gluing stdin/stdout to the terminal), **network** (via the kernel's networking subsystem, plus firewall/filtering/marking).

Buffer Cache – Performance vs Durability Trade-off

Reading: locate blocks via the inode, check the buffer cache, read from disk only on a miss, return from cache. Writing updates the cached copy immediately – this "typically" avoids around **85%** of the disk transfers that would otherwise be implied, but risks losing data that exists only in the cache if the system crashes before the periodic (~**30 second**) flush of dirty buffers.

Block devices use **Completely Fair Queueing (CFQ)** in the request manager to schedule buffer reads/writes to/from the driver.

12.6 Start of Day

The kernel (`/vmunix`) loads and mounts the root filesystem; PID 1 (`/etc/init`) is hand-crafted and reads `/etc/inittab`, forking a `tty` process per terminal entry. Each `tty` process initialises its terminal, prints `login:`, and on input `execve`s `/bin/login`, which prompts for a password, hashes and checks it against `/etc/passwd`, and on success sets the UID/GID and `execve`s the configured shell. When the shell exits, `init` resurrects the `tty` process, and the cycle repeats.

Shell operation & standard I/O

The shell is just another process, needing no special kernel understanding, using the CWD to avoid fully-qualified paths; command-line parsing handles wildcard expansion (globbing), tilde processing, and backgrounding (`&`). Every process is created with three inherited file descriptors: **stdin**, **stdout**, **stderr** – each independently redirectable (`>`, `>&`, `<`). A **pipeline** (`ls | wc`) is more efficient than redirecting through an intermediate file, since data streams directly between processes rather than round-tripping via disk.

Exam Tip: File Descriptors as "Capabilities"

FDs behave like capabilities in that possession grants access (an unforgeable, kernel-mediated handle that can be passed to another process, e.g. over a UNIX socket, without granting broader rights) – see the print-server example above. They *differ* from true capabilities in that: they are not freely nameable/transferable objects with fine-grained, per-operation rights; they are scoped to a single process's file table (indices, not global names); and the rights they encode were fixed at `open()` time by the ACL-based check, rather than being an independent, revocable, delegatable token. This exact comparison was examined in 2022.

12 Formula & Quick Reference Sheet

12.6 Scheduling

Scheduling

Waiting time = Turnaround time − Burst time = (Completion time) − (Arrival time) − (Burst time)
Exponential averaging (burst prediction): $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$
Round Robin: Fair share = $1/n$ per round; max wait = $(n-1)q$
Computed priority (classical UNIX): $CPU_j(i) = \dfrac{2\,load_j}{2\,load_j + 1} CPU_j(i-1), \quad P_j(i) = base_j + \dfrac{CPU_j(i)}{4} + 2\,nice_j$
CFS: slice $t_j \propto w_j / \sum_i w_i$; effective slice = max(target latency$/N$, min. granularity)

12.6 Memory, Paging & Virtual Memory

Memory & Paging

Logical address: $\langle p \mid d \rangle$, p is $(m-n)$ bits, d is n bits, for 2^m address space and 2^n page size.
TLB effective access time: EAT $= h(t_{\text{TLB}} + t_{\text{mem}}) + (1-h)(t_{\text{TLB}} + 2\,t_{\text{mem}})$ [generalises to $(L+1)\,t_{\text{mem}}$ on a miss for an L-level table]
Demand paging EAT: EAT $= (1-p)\,t_{\text{mem}} + p\,t_{\text{fault}}$
k-level page table: address bits = $k\times$(bits per level)+(offset bits); each level table size = $2^{\text{bits/level}}\times$(PTE size)
Thrashing condition: $D = \sum_i WSS_i > m$ (available frames)

12.6 Storage & Filesystems

Storage

Average rotational latency = $\dfrac{30}{\text{RPM}}$ [seconds]
Access latency = Average seek time + Average rotational latency
Average I/O time = Access latency + $\dfrac{\text{transfer amount}}{\text{transfer rate}}$ + controller overhead
Max file size (inode w/ direct + single/double/triple indirect, k addresses/indirect block, block size B):

$$\text{Max size} = \left(n_{\text{direct}} + k + k^2 + k^3\right) \times B$$

12.6 Protection & Access Control

Protection

Access-right = ⟨object-name, rights-set⟩; Domain = set of access-rights.
ACL = access matrix stored *by object* (column); Capability = access matrix stored *by domain* (row).
UNIX permission resolution order: owner UID match → group GID match → world. Root bypasses all checks.

12 Tripos Exam Strategy & Common Pitfalls

12.6 Paper Structure & High-Value Topics

Aspect	Detail
Format	Operating Systems is one question on Paper 2 (of several IB Paper 2 questions); allow ∼1 hour
Question structure	Multi-part (a)–(d)/(e); parts often build on each other, mixing short bookwork with a numerical/scenario part
High-yield topics (nearly every year)	CPU scheduling (esp. FCFS/SJF/SRTF/RR numericals), Paging & TLB effective-access-time calculations, Page replacement (FIFO/OPT/LRU, Bélády's anomaly)
Frequently examined	Protection (ACL vs capability, UNIX permissions), Address binding, Context switching, I/O buffering, Disk performance
Recurring "systems design" style	Given a novel proposal/scenario (e.g. a new buffering scheme, a modified idle process), argue costs/benefits – reward for structured, balanced reasoning over a single "correct" answer

12.6 General Exam Technique

1. **Read all parts of a question before starting.** Later parts often reveal what earlier parts are building towards (e.g. a scheduling scenario in part (b) that part (c) then asks you to critique).
2. **Show all working for numericals.** Method marks are awarded generously for correct formulas and structured steps, even if the final arithmetic slips.
3. **Draw a Gantt chart** for any scheduling question – it prevents ordering mistakes and makes waiting/turnaround time trivial to read off.
4. **For paging/TLB questions, write the bit-width equation first**: (levels × bits/level) + offset bits = total address bits. This single check catches most sizing errors before you compute anything else.
5. **For page replacement, tabulate the frame contents at every step** of the reference string – and always double check whether you're asked for OPT, LRU, or FIFO, since the running total differs.
6. **"Compare and contrast" or "advantages/disadvantages" questions want balance.** Give a genuine consideration on both sides before concluding, even if you have a preferred answer.
7. **For "true/false, explain why" questions**, no marks are given without a justification – always write at least one sentence of explanation, regardless of how obvious the answer seems.
8. **For UNIX permission/group questions**, write out the full access matrix or table of (owner, group, mode) explicitly before answering sub-questions – this avoids errors when the file's owner is themselves also a member of the relevant group.
9. **Systems-design / "propose a scheme" questions** (increasingly common since ∼2022) want you to (i) state the proposal precisely, (ii) identify the mechanism required, (iii) discuss at least one cost and one benefit, and (iv) where asked, compare quantitatively.

12.6 Top 10 Operating Systems Tripos Traps

Top 10 Tripos Traps

1. **"Dual-mode operation alone protects the system."** False – you also need the timer (for CPU), base/limit registers or paging (for memory), *and* the fact that I/O protection ultimately relies on memory protection (since some devices are memory-mapped).
2. **Confusing ACL and capability.** ACL is stored *by object* (a column of the access matrix); capability is stored *by domain* (a row). Getting this backwards is one of the most common errors.
3. **Zombie vs orphan.** Zombie = child exited, parent hasn't `wait()`ed yet. Orphan = parent exited first, child re-parented (traditionally to `init`). Do not swap these.
4. **TLB miss cost with multi-level page tables.** A miss costs $L+1$ memory accesses (one per level, plus the data), not just 2 – forgetting the extra levels is a very common numerical slip.
5. **Bélády's Anomaly.** Only FIFO can exhibit it (adding frames *increases* faults); LRU and OPT cannot, because they are *stack algorithms*.
6. **SRTF is not unconditionally optimal.** It's optimal only relative to a known process set; unpredictable

future arrivals and non-zero context-switch cost both break this in practice.

7. **Internal vs External fragmentation.** Internal = wasted space *within* an allocated unit (paging). External = free space that exists but isn't *contiguous* enough to satisfy a request (segmentation/contiguous allocation).
8. **Waiting time ≠ turnaround time.** Waiting time only counts time in the *Ready* queue; turnaround time counts the *entire* lifetime from submission to completion.
9. **Convoy effect is about arrival *order*, not the algorithm being "bad".** FCFS is fine in isolation; the pathology appears specifically when a long process happens to arrive ahead of several short ones.
10. **File descriptors resemble but are not true capabilities.** They're unforgeable and transferable (via socket passing) but scoped to a process's file table, with rights fixed at `open()` time by an ACL check rather than being an independently revocable/delegatable token.

12 Final Exam Checklist

Before submitting your answers, verify these things:

1. **Gantt charts:** For every scheduling numerical, have you drawn a chart and read waiting/turnaround times directly from it rather than trying to compute them purely algebraically?
2. **Bit-width check:** For paging questions, does (levels $\times$ bits/level) + offset bits actually equal the stated address width? If not, re-derive before proceeding.
3. **TLB miss cost:** Have you correctly counted $L + 1$ memory accesses per miss for an L-level page table, not just 2?
4. **Page replacement table:** For FIFO/OPT/LRU questions, have you tabulated frame contents at *every* reference, and double-checked which algorithm you're computing at each step?
5. **ACL vs Capability:** Did you correctly state which is stored *by object* (ACL) and which *by domain* (capability)?
6. **UNIX permissions:** Did you build the full owner/group/mode table before answering, and check whether a user is *both* owner *and* a group member (owner rights normally take precedence in your reasoning, but double-check the question's intent)?
7. **Balanced "compare and contrast" answers:** Did you give genuine points on *both* sides, not just a list favouring one option?
8. **Justification on every "true/false":** Is there a full sentence of reasoning behind every true/false answer, even the ones that felt obvious?
9. **Interpretation of every numerical result:** Did you state what the number *means* (e.g. "EAT = 140ns, a 40$\times$ slowdown versus the no-fault case") rather than leaving a bare figure?
10. **Mechanism vs Policy:** Where a question asks you to justify a design choice, have you distinguished the *mechanism* (how) from the *policy* (what is decided)?

Final Advice

Operating Systems rewards **precise definitions**, **careful numerical working**, and **balanced discussion** of trade-offs – there is very often no single "correct" design, only well- or poorly-justified ones. Don't rush the arithmetic on scheduling/paging/disk questions: set up the formula, define your notation, and show every step – method marks are generous. Two well-structured, fully-worked answers, each engaging honestly with the trade-offs the examiner is probing for, will comfortably earn a strong mark.

Good luck in your Tripos!