Goal of Operating Systems:

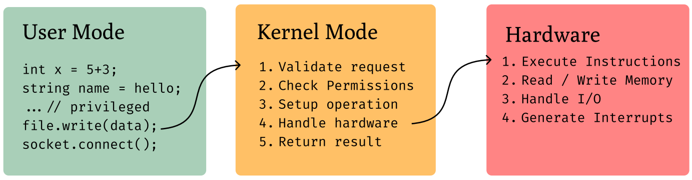
- Abstract complex hardware mechanisms into usable forms for applications, through the *virtualisation* of the CPU, memory, storage and I/O.
- Allow multiple programs to execute at the same time through *concurrency*.
- Retain data over time through *persistence*.

Processes: Programs in execution with an associated state.
Threads: Subsets of processes that share resources with other threads belonging to the same process.

Aspect	Process	Thread
Resources	Seperate	Shared
Memory	Seperate	Shared
Creation	High overhead. New memory space, resources and structures	Low Overhead. Only new stack and register set.
Context Switch	Longer. Save and restore full process.	Shorter. Only save registers stack.
Communication	Harder. Inter process comms necessary	Easier. Shared memory.
Isolation	Strong	Weak

CPU Virtualisation:

- Let programs *directly execute*. This is bad! Programs can access whatever they want or hog the CPU forever.
- Fix? Restrict programs from certain actions by making them defer to *kernel mode* via sys calls.
- System calls provide an interface between user programs and the Kernel.



Concurrency:

- Implement *process management* to give the illusion of continuous, concurrent execution. How?

Dispatching

Transfer control of the CPU to a process selected by the scheduler via *context switches*: Save the current process state, load the next process state in kernel mode, then switch to user mode to execute the process.

- Allow processes to *yield* control (can lead to malicious programs monopolising resources and never *yielding*)
- Allow OS to *interrupt* running processes and let others take control.
- Store *process states* related to underutilisation of the CPU due to slow I/O (blocked) and give control to other processes waiting on the CPU (ready) so that it can begin (running).

Scheduling

Strategies to determine fairly, which processes run next. Assessed using various time metrics:

- Turnaround Time (submission -> completion)
- Response Time (submission -> first response)
- Waiting Time (Total time spent waiting)
- Throughput (How many jobs completed in set time)
- Fairness (Equal CPU time distribution among processes)

Scheduling Algorithms:

First-In-First-Out: Executes processes to completion in the order they arrive before starting the next one.

- Processes are treated equally, so simple to implement with low overhead. Highly predictable.
- Convoy Issue: Short jobs wait for long jobs to finish.

Shortest Job First: Executes the shortest available job to completion next to minimise average waiting time.

- Sort processes based on estimated execution time and let them go first.
- Timing Issue: If a long job arrives before short jobs arrive, they're still stuck waiting. If short jobs keep arriving, long jobs are stuck waiting.
- Knowledge Issue: It's very hard to actually know how long processes will execute for.

Shortest-Time-To-Completion-First: Shortest job first with *pre-emption*.

- If a new job arrives with a shorter remaining time, stop the current process and let the new one start (pre-emption).
- Timing Issue: Removed first issue, but if short jobs keep arriving, long jobs are stuck waiting still.

Round Robin: Allocates CPU time to each process in equal, fixed size time slices in a cyclic order.

- Each process receives a fixed time slice (quantum) to execute, then is pre-empted to let another process execute.
- Order is based on arrival time. Once pre-empted, a process moves to the end of the queue.
- Time slice issues: Too small - excessive overhead. Too big - behaves like FIFO.

Multi-Level Feedback Queue: Add *priorities* (niceness) to processes based on their behaviour.

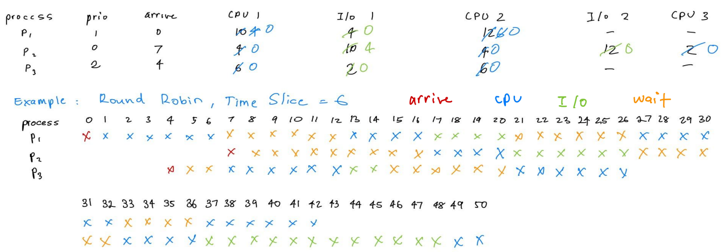
- Processes that yield alot (short CPU bursts) get high priority, processes that use their quantums up get low priority
- Run highest priority jobs first, then demote jobs that use up their time quantum.
- Periodically boost all jobs to highest priority to prevent starvation.
- Processes can 'game' the system by giving up it's time just before the time slice ends

Pre-emption (interrupting processes) and *priority* (assigning importance levels) are features that modify scheduling algorithms—not stand-alone algorithms themselves—to influence process execution order.

Lottery: Periodically hold a lottery where processes can be randomly selected to execute. Fair, stops gaming MLFQ.

Scheduling Tips:

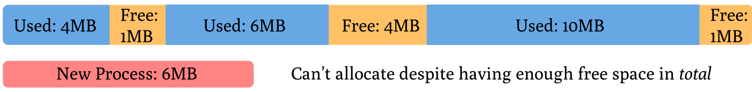
1. IO for processes can run simultaneously unless otherwise stated. CPU can only do one.
2. A process does not need to wait for the next time slice if a process finishes early.
3. If two processes arrive at the same time, choose via 1. whichever process executed last (if any), then 2. by priority (if both are brand new).



Memory Virtualisation: Programs expect unlimited contiguous memory but must safely share limited physical resources, so we give each process the illusion of its own private memory through *address translation*, turning shared physical memory into isolated virtual spaces.

Base & Bounds: Give each process a base register (where program starts in physical memory) and bounds register (how much it can use) - fails because *memory must be contiguous* and lacks protection between program segments.

Segmentation: Programs naturally divide into segments (code, stack, heap) that require different permissions and growth patterns. Segmentation provides per-segment base/bounds pairs and protection bits, allowing segments to be placed independently in memory and grow naturally. When a program accesses memory outside its segment bounds, hardware triggers a segmentation fault - protecting memory from buffer overflows and invalid accesses. However, this flexibility leads to external fragmentation: memory becomes riddled with variably-sized holes, making it impossible to allocate new segments despite having enough total free space.



Can't allocate despite having enough free space in *total*

Paging: Divides memory into fixed-size chunks (pages), eliminating external fragmentation. Every allocation is perfectly aligned - like dividing a bookshelf into identical slots. The MMU uses *page tables* to map each virtual page to its physical location (frame), tracking each page's status through bits:

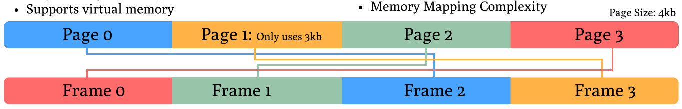
- Present bit: Is the book on shelf (RAM) or in storage (disk)?
- Dirty bit: Has the page been edited? Like a sticky note marking modified pages that need saving back to disk
- Referenced bit: Recently accessed? Like a bookmark for the Clock algorithm
- Protection bits: What's allowed? Like library cards marking read-only vs check-out privileges

Advantages:

- No external Fragmentation
- Flexible allocation (doesn't need to be contiguous
- Memory protection per page
- Easy sharing between processes
- Supports virtual memory

Disadvantages:

- Internal Fragmentation
- Page table overhead
- TLB Overhead
- Page fault handling overhead
- Memory Mapping Complexity



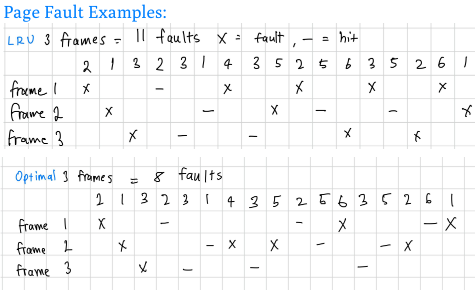
Optimisations: Tables can become enormous leading to memory waste and slow lookups. Optimise by:

- Inverted Page Tables: Reverse mapping (frame→virtual page) with single table per system, saves memory but slower lookups.
- Segmented page tables: Different page tables for each segment, good for sparse address spaces
- Multi-level page tables: Break page table into nested tables, only allocate needed pieces, saving memory for sparse address spaces
- Demand Paging: Only load pages when accessed, triggering *page faults*.

Page Faults + Replacements: When a program needs a non-present page, the page fault handler acts like a librarian fetching a book from storage.

1. Hardware triggers page fault when accessing non present page
2. OS saves current process state (like using a bookmark)
3. OS locates page on disk
4. If memory full, choose victim page to evict via:
    - FIFO: Oldest pages out first - simple but often wrong. Check the last n faults and replace the first one.
    - LRU: Least recently used - optimal but expensive to track. Check the last n faults + hits, skipping duplicates and replace the first one.
    - Clock Algorithm: Approximates LRU by checking referenced bits in a circular order, clearing each bit on first pass and evicting first page with cleared bit on second pass - like a librarian walking in circles, marking touched books and removing the first untouched one they see on their second round.
    - Optimal: Replace the page that won't be used for the longest time (not possible to implement, but theoretically the best). Look at the last 3 unique faults (1 for each frame) and replace the one that won't appear for the longest time.
5. If victim is dirty (has changes), write to disk
6. Read required page into memory
7. Update page table and store process state and execution (open book to bookmark and read)

Page Fault Examples:



Synchronization Examples:

```
// MUTEX
mutex m;
wait(m);          // Take mutex or block
do_work();
signal(m);        // Release mutex

// CONDITION VAR
mutex m; cond c;
wait(m);          // Take mutex
while(!ready)     // Need condition true
   wait(c);       // Give up mutex & wait
do_work();        // Have mutex & ready=true
signal(m);

// SEMAPHORE
sem = 2;          // Allow 2 threads
wait(sem);        // Decrement (block if 0)
use_resource();
signal(sem);      // Increment, wake one
```

Translation Lookaside Buffer: For speed, we cache recent virtual→physical translations in the TLB - like a librarian's shortlist of frequently requested books. This avoids expensive page table lookups. But this creates a context switch challenge: these cached locations belong to specific processes, so we must either clear the whole shortlist (flush TLB) or tag each cached entry with its owner's ID (Address Space ID/ASID). OS clears or tags TLB entries on context switches; hardware handles quick lookups. The hardware consults TLB first - if miss, then walks page table. Effectiveness relies on locality:

- Spatial: Programs tend to access nearby memory locations.
- Temporal: Recently accessed locations likely to be accessed again soon.

Extra Stuff:

- *Thrashing:* When a system spends more time resolving page faults than executing. Occurs when working set exceeds physical memory.
- *Copy-on-Write:* Share physical pages between parent and child processes after fork(), marking them read-only - only create copies when either process tries to modify a page, making process creation efficient as most pages are never modified.
- *Fast Fork:* Use COW to avoid copying entire address space, just duplicate page table.
- *Exec:* Since exec() typically follows fork() and replaces entire program, immediately discard unmodified COW pages instead of copying them

Concurrency: Sequential execution limits resource use and performance. Concurrency interleaves execution of threads or processes to improve utilization and responsiveness.

- *Race Conditions:* Occur when threads access shared resources simultaneously, causing unpredictable results due to interleaved operations.
- *Critical Sections:* Code segments accessing shared resources, requiring protection to prevent race conditions.

Synchronization Primitives: Tools to coordinate concurrent threads, ensuring orderly access to shared resources and preventing race conditions and deadlocks.

- *Locks:* Basic mutual exclusion allowing only one thread in a critical section.
  - Issue: High contention leads to inefficient waiting.
- *Spin Locks:* Threads "spin," repeatedly checking until the lock is free. Use for short critical sections with low contention, as they waste CPU during high contention. Issue: Spinning wastes CPU resources under high contention.
  - Disable Interrupts: Prevents context switches on uniprocessor systems. Fixes context switching but monopolizes the CPU.
  - Load/Store: Loads the lock's current value to check if it's free (0), then stores 1 to claim it. Issue: Not atomic, so race conditions can occur if multiple threads load and store simultaneously.
  - Test-and-Set: Atomically sets the lock's value to 1 and loads the previous value. If the previous value was 0 (unlocked), the thread acquires the lock; otherwise, it retries. Issue: Causes repeated retries (busy-waiting) when locked.
  - Compare-and-Swap (CAS): Atomically loads the lock's current value and compares it with 0. If 0, CAS stores 1 to claim the lock. If not, CAS does nothing, and the thread retries. Issue: Still involves busy-waiting, which wastes CPU.
- *Blocking Locks* (e.g., Mutexes): Threads block (sleep) if the lock is unavailable, reducing CPU waste. Suitable for long critical sections with high contention. Issue: Can cause deadlocks if threads hold resources while waiting for others; lacks conditional waiting.
- *Condition Variables:* Used alongside blocking locks (like mutexes) to allow threads to wait for specific conditions and signal others when conditions change.
  - `wait(mutex)`: Releases the mutex and waits, reacquiring it on wakeup.
  - `signal() / broadcast()`: Wakes up one or all waiting threads.
  - Issue: Spurious wakeups require rechecking the condition; doesn't manage access to multiple resources.
  - Bounded Buffer with Condition Variable: Uses two condition variables, `notEmpty` and `notFull`, for producer and consumer threads. Producers wait on `notFull` when the buffer is full; consumers wait on notEmpty when it's empty.
  - Producer/Consumer with Two Condition Variables: `notEmpty` signals consumers when items are available; `notFull` signals producers when there's space.
- *Semaphores:* Control access to multiple instances of a resource. A non-negative integer count represents available resources.
  - `acquire()`: Decrements count if positive; blocks if zero.
  - `release()`: Increments count; may wake waiting threads.
  - Bounded Buffer with Semaphores: Use a semaphore empty for free slots and full for filled slots, alongside a mutex to control access. Producers wait(empty) before adding; consumers wait(full) before removing.
- *Equivalence:* Synchronization primitives are functionally equivalent by adapting usage. A binary semaphore acts as a mutex by setting its count to 0 (acquire) and 1 (release) for mutual exclusion. Conversely, a mutex and condition variable mimic a counting semaphore by using a resource count and signalling waiting threads, enabling controlled access.

Deadlock: Occurs when threads are blocked, each waiting for resources held by others. 4 Conditions must be met to reach a deadlock, if any one condition is prevented, the deadlock is broken:

1. Mutual Exclusion: Only one thread can use a resource at a time.
2. Hold and Wait: Threads hold resources while waiting for others.
3. No Pre-emption: Resources cannot be forcibly taken away.
4. Circular Wait: Circular chain of threads each waiting for resources held by the next.

Avoidance:

- Wait-Free Algorithms: Ensure every operation completes in a finite number of steps, removing the need to wait.
- Atomic Lock Acquisition: Require threads to acquire all needed resources atomically.
- Pre-emption: Allow resources to be forcibly taken away.
- Resource Ordering: Impose an order on resource acquisition to prevent circular wait.
- *Banker's Algorithm:* Checks if granting a resource can be done safely without causing a deadlock. Parameters include:
  - Safe State: A state where the system can allocate resources to each process in some order and still avoid deadlock.
  - Available Resources: The number of available resources of each type.
  - Maximum Demand: The maximum number of resources a process may need.
  - Allocated Resources: The number of resources currently allocated to a process.
  - Need: The remaining resources that a process may still request.

  a. Check Need: Ensure the request doesn't exceed what the process still needs.
  b. Check Availability: Confirm enough resources are available to fulfill the request.
  c. Simulate Allocation: Temporarily grant the resources and verify if the system stays safe.
  d. Decision: If safe, grant the request. If not, deny and make the process wait.

```
Given: 3 people (Alice, Bob, Charlie), 3 fruits (Apples, Bananas, Oranges), Available: 2 1 0
Alice requests 1 1 0 (Need: 2 2 0, Allocated: 1 0 1) → Granted, no one starves
Charlie requests 1 0 1 (Need: 1 1 1, Allocated: 0 1 0) → Denied, exceeds Need
```

I/O Devices: Users need to be able interact with OS via external devices. OS Needs standardised way to interact with diverse hardware having different speeds and protocols.

- *Canonical devices* solve hardware diversity through standardized registers (status: reports device state, command: sends instructions, data: transfers information). Driver translates between standard interface and hardware. Device interaction happens through:
  - *Polling:* CPU repeatedly checks device status. Issue: Wastes CPU cycles - good for fast devices or predictable timing.
  - *Interrupts:* Device signals CPU when ready. Issue: Polling inefficient. Solution: CPU free to do other work while waiting.
  - *Direct Memory Access (DMA):* Issue: CPU wastes cycles copying data, so DMA transfers data between memory and devices independently. CPU only sets up transfer then continues work; gets interrupted when done.
- *Patterns*: What does the process do while waiting on I/O?
  - *Blocking*: Process waits until I/O completes, simple but idle CPU.
  - *Non Blocking*: Process starts I/O, keeps running, checks status periodically. Improves CPU use with added logic. Better CPU use but complex.
  - *Asynchronous*: Process initiates I/O, keeps running, receives notification on completion. Most efficient, highest complexity.

Device Drivers: Two level abstraction separates generic OS interface (top half) from hardware specific operations (bottom half). Issue: Different devices need consistent interface. Solution: Top half handles standard requests (read / write) while bottom half manages specific hardware details and interrupts. I/O Request Lifecycle:

1. Application sends request through top half
2. Driver converts to device commands in bottom half
3. Device executes operation
4. Device signals completion via interrupt
5. Bottom half processes interrupt, returns to top half
6. Result returns to application

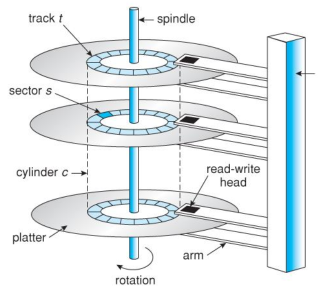
Persistence: Storage is essential for retaining data without power, unlike RAM. Ideal storage balances capacity, speed and cost. How do we persist data?

Disks: Store data in 512-byte sectors on spinning platters, with read/write heads detecting magnetic domains representing 0s and 1s. Access time is influenced by:

- Seek Time: Time for the read/write heads to reach the correct track.
- Rotational Latency: Delay as the disk rotates to align the target sector under the head.
- Transfer Time: Time to read/write data, depending on RPM and sector density.

Performance metrics:

- Random Access: Involves frequent seeks and rotations, focusing on average seek time and latency.
- Sequential Access: Reads/writes adjacent sectors, minimizing movement and maximizing transfer rate.



SSDs store data by trapping electric charges in flash memory cells, where each cell holds a binary value based on its charge state. Data is organized into pages (for reading) and blocks (for erasing), with a controller managing data placement and wear levelling to extend drive lifespan.

- Pros: Fast read/write speeds, low latency, durability, silent operation, low power, compact.
- Cons: Higher cost, limited write endurance, sudden failure risk, challenging data recovery, smaller max capacity.

I/O Scheduling: Determines the order in which I/O requests are processed to optimize device performance.

- *First-Come, First-Served (FCFS):* Processes requests in the order they arrive. Simple but can cause long waiting times if requests are scattered.
- *Shortest Positioning Time First (SPTF)*: Selects the request closest to the disk head, minimizing movement, though it may delay distant requests.
- *SCAN (Elevator)*: Moves the disk head in one direction, servicing requests in its path, then reverses. This balances efficiency and fairness by reducing back-and-forth movement.

Performance Metric: Work Conservation ensures the disk remains active when there are pending requests, maximizing utilization.

RAID combines multiple disks to improve storage, performance, and reliability.

- *RAID 0* - Striping: Splits data across disks, no redundancy. Increases speed but any disk failure results in data loss.
- *RAID 1* - Mirroring: Copies data to two disks, providing high fault tolerance and faster reads. Halves storage capacity.
- *RAID 4* - Striping with Dedicated Parity: Data is striped across disks, with one disk storing parity (error-checking data) for recovery if one disk fails. Slower writes due to single parity disk bottleneck.
- *RAID 5* - Distributed Parity: Data and parity are striped across all disks, allowing recovery from one disk failure and better write performance than RAID 4.

`N=#disks, C=capacity, S=sequential throughput, R=random throughput, D=latency of I/O operation`

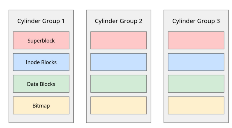
RAID Level	Capacity	Reliability	Read Latency	Write Latency	Seq Read	Seq Write	Rand Read	Rand Write
RAID 0	N * C	Lowest	D	D	N * S	N * S	N * R	N * R
RAID 1	N/2 * C	Highest	D	D	N/2 * S	N/2 * S	N * R	N/2 * R
RAID 4	(N-1) * C	Medium	D	2D	(N-1) * S	(N-1) * S	(N-1) * R	R/2
RAID 5	(N-1) * C	Medium	D	2D	(N-1) * S	(N-1) * S	N * R	N/4 * R

Unix File Systems: *Files* are abstractions representing persistent byte arrays in storage, enabling data retention beyond process lifetimes.

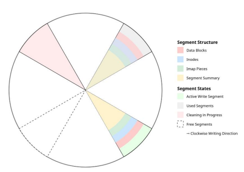
- *i-nodes:* Data structures holding metadata (size, permissions, block addresses) for a file, identified by an i-node number.
- *Directory Tree* (Unix Name Space): Hierarchical structure where directories map file names to i-node numbers, creating a human-readable path system.
- File Descriptors: Non-negative integers assigned by the kernel when a file is opened; used to access files without repeated directory traversal.
- *Hard Links*: Multiple directory entries that reference the same i-node, enabling a single file to have multiple names. Deleting one hard link does not affect others; all hard links must be deleted for the file data to be removed.
- *Symbolic Links*: Files that store a path to another file or directory, allowing links across filesystems and to directories. Symbolic links have different i-node numbers than the target file. If the target file is deleted, symbolic links break, as they rely on the original file path.
- *Permissions*: Control access to files through user/group/world permissions, specified as read (r), write (w), and execute (x).
- *Multi-level Block Indexing:* Uses a mix of direct and indirect pointers to manage large files by addressing blocks hierarchically.

File System Solutions: Locality issues cause slow access when related files are scattered across a disk, leading to frequent seek operations. Solutions focus on keeping related files physically close.

- *Fast File System (FFS)*: Uses clusters of consecutive blocks (cylinder groups) to minimize seek times by keeping related structures nearby:
  - Structures: Superblock (stores metadata), Allocation Bitmaps (track free/used blocks), Data Blocks (contain file contents), Directories (map filenames to i-node numbers).
  - Crash Recovery: FSCK (File System Consistency Check) scans across the entire file system to detect issues such as orphaned inodes, incorrect free block counts, and broken directory links. Issues: time-consuming, requires the file system to be unmounted, cannot always restore files to their original state, often resulting in data loss or orphaned files.



- *Log-Structured File System (LFS):* Writes data sequentially in a log to reduce seek time and optimize for write performance.
  - Structures: Segments (log-based storage units for files), Imap (maps i-nodes to storage locations in the log).
  - Crash Recovery: Uses checkpointing and roll-forward from the log to restore data after a crash.
  - Garbage Collection: Reclaims space by identifying "dead" blocks (stale or deleted data) in segments, compacting live data to free up entire segments for new sequential writes.



- *Journaling*: Maintains a journal to log changes before committing to disk, enabling efficient crash recovery. Faster than FSCK, checkpointing.
  - Structures: Journal (logs pending changes to ensure consistency).
  - Crash Recovery: Replays journal entries to recover data up to the last committed transaction, minimizing data loss. Checksums: Validates journal entries to detect and prevent corruption.