Computer Systems - CheatSheet

IN BA4 - Katerina Argyraki

Notes by Ali EL AZDI

A Computer Systems Cheatsheet has been authorized for the upcoming exam, and I'm sharing a copy of mine for anyone interested. It provides a concise summary of the key concepts and techniques covered in the course. For any updates or suggestions, feel free to reach out to me on Telegram at `elazdi_al` *or via EPFL email at* `ali.elazdi@epfl.ch`*.*

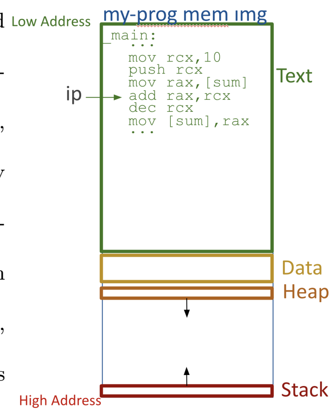
April 3rd, 2025

Program. Passive entity. A sequence of instructions stored in a file, not currently executing.

- **Storage:** Stored on persistent storage (e.g., disk). Loadable into main memory by the OS.
- **Access:** Static file. Contains instructions, data, and metadata used during execution.

Process. Active execution of a program. Managed by the OS. Each process has its own isolated virtual address space.

- **Storage:** Occupies main memory during execution.
- **Memory Image:** Text (code), data, heap, and one stack per thread.
- **Process ID (PID)**, assigned and tracked by the OS, unique system-wide.
- **Status:** OS tracks current status (e.g., running, waiting, terminated).
- **Virtualization:** Each process has the illusion of exclusive access to memory.
- **OS-allocated Resources:** File descriptors, sockets, I/O handles, etc.
- **Pointer to Page Table.** The **Kernel** tracks the process-specific page table pointer.
- **Access:** Operates in isolated virtual memory. No direct access to other processes. Interacts with hardware via system calls. OS handles scheduling and resource management.



If a process has a single thread, we may refer to the *thread's CPU context* as the *process's CPU context.*

Limited Direct Execution. OS design allowing user programs to execute instructions directly on the CPU, with restrictions.

- **Goal:** Maximize performance while maintaining control and protection.
- **User Code Execution:** CPU runs user programs natively (not emulated) in **user mode**.
- **OS Control:** OS retains control over hardware via **privileged instructions** and mode switching.

User Mode vs Kernel Mode. Two CPU execution modes stored as a state bit in a protected CPU register (0 = user mode, 1 = kernel mode) controlling access to hardware and instructions.

- **User Mode:** Restricted. User code cannot execute privileged instructions or directly access hardware/memory management.
- **Kernel Mode:** Mode in which is ran a central part of the operating system, the **Kernel**.
- **Switching:** A transition from user mode to kernel mode is triggered by:
  - System Calls (e.g., file I/O, memory allocation)
  - Hardware Interrupts (keyboard, timer,...)
  - Software Traps (divide-by-zero, invalid memory access),
    If an exception happens in kernel mode, an internal routine is called but no mode switch occurs.

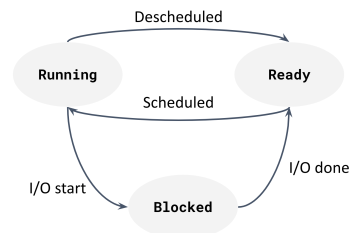
  After handling the event, the OS switches the CPU back to user mode to resume application execution.

Privileged Instructions. CPU instructions that can only execute in **kernel mode**. If attempted in user mode, the CPU raises a trap to the OS. Prevent user programs from interfering with other processes or the OS.
To do something that requires high privilege, a thread makes a syscall, invoking the kernel.

Exception/trap - synchronous signal. The CPU itself raises an exception/trap.
Interrupt - asynchronous signal. Some external entity raises an interrupt. If an external entity needs attention, it raises an **interrupt**, invokes the kernel.
Even if nothing external needs attention, the timer interrupt regularly invokes the kernel to run the OS scheduler.

Thread. The smallest unit of CPU execution within a process. Multiple threads can exist within one process.

- **Memory:** Shares the parent process's virtual address space:
  - Shared: Code (text), heap, global/static data, open files.
  - Private: Each thread has its own stack (for local variables and function calls).
- **Execution context (per thread), values of all the CPU registers at the *last moment* the thread was running:**
  - **Instruction Pointer (IP) / Program Counter (PC):** Physical register pointing to the next instruction.Private per thread.
  - **Stack Pointer (SP):** Points to the top of the thread's private stack.
  - **General-purpose Registers:** Include temporary registers (e.g., RAX, RBX), status, and flags. **All private per thread.**
  - **Thread ID (TID).** The Thread ID is unique within a process, but not necessarily system-wide, assigned and tracked by the OS.
  - **Status:** OS tracks each thread's status (Running, Ready, Blocked).
  - **Virtualization:** Each thread has the illusion it exclusively occupies the CPU.
  - **Context switching:** OS saves/restores full register set (IP, SP, general-purpose regs). The CPU switches from one thread to another. Thread switches are faster than process switches. It saves to memory the CPU context of the current thread; it restores from memory the CPU context of the new thread.
  - **Thread management:** A main thread may create and manage others. In some models, a dedicated manager thread exists.
  - **Managed by OS:** The OS kernel schedules threads individually and tracks them.



OS Components.

- **Kernel (core of the OS):** Loaded at boot, runs in **kernel mode**, and manages hardware, memory, processes, and system calls. It is the only part of the OS that runs in kernel mode.
  Three important routines inside of the kernel:
  - **Interrupt Handler or Interrupt Service Routine (ISR):** Handles interrupts from hardware devices.
  - **Exception / Trap Handler:** Handles exceptions and traps from user programs.
  - **System Call Handler:** Handles system calls from user programs.

  The three routines use two tables (**Trap/Interrupt Table** and **System Call Table**) to resolve the event type to a specific memory address of the handler routine managing the event.
- **Loader (part of the OS):** Prepares executables to run:
  - Loads program code/data into memory.
  - Maps required libraries (e.g., libc).
  - Sets up the process stack and heap.
  - Places command-line arguments and environment variables on the stack.
  - Sets the `%rsp` (stack pointer) and `%rdi` (argc), `%rsi` (argv) registers so the program can access arguments.
  - *Because the kernel manages processes and initiates execution, it is responsible for placing arguments in registers so the loader (running in user mode) can access them.*
  - Jumps to the program's entry point to begin execution in **user mode**.
- **User-level Programs:** Applications like shells, editors, browsers, etc. These are **part of the OS** but run entirely in **user mode**, relying on system calls to request kernel services.
- **System Libraries:** Shared libraries (e.g., libc) used by user programs. Loaded and linked by the loader, but executed in user mode. Provide wrappers around system calls.

---

fork() — *Clone the current process*

1. The operating system creates a new process (child) by duplicating the calling process.
2. The child initially shares all memory pages with the parent:
   - Pages are marked read-only and shared (Copy-On-Write).
   - If either process attempts to write to a page, it is copied privately for that process.
3. The child also receives:
   - Duplicated file descriptors (pointing to the same open files).
   - Identical program counter and stack pointer.
4. Returns:
   - In the parent process: the PID of the child.
   - In the child process: `0`.
5. Both processes resume execution at the instruction following `fork()`.

exec() — *Mutate the child into a new program*

- Typically called by the child process immediately after `fork()`.
- Replaces the child's memory space with a new program image.
- Stack, heap, code, and data segments are replaced.
- If successful: the new program starts execution from `main()`.
- If failed: `exec()` returns `-1`, child continues old code.

exit() — *Terminate a process cleanly*

- Frees system resources allocated to the process.
- Sends termination status to the parent.
- If main function of a program returns, `exit()` is implicitly called.

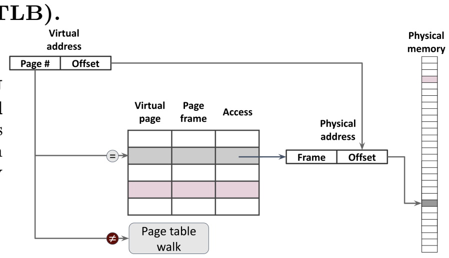
wait() — *Wait for a child process to terminate*

- Blocks the calling process until one of its child processes terminates.
- Returns the PID of the terminated child.
- If no child processes exist, it waits indefinitely.

Memory Management Unit (MMU) - Address Translation.
Hardware component managed by the **operating system** that translates **virtual addresses** into **physical addresses** using **per-process page tables**. **Each process has its own page table, and the currently active one is referenced in a special CPU register.**

Translation Lookaside Buffer (TLB).

A small, fast **cache inside the MMU** that stores recent **virtual-to-physical page mappings**. The TLB reduces the need to access the page table on every memory reference, significantly speeding up address translation.



Virtual Address Translation:

1. The **CPU executes an instruction** (e.g., `load`, `store`, or `fetch`) that references a memory location using a **virtual address**.
2. The **MMU extracts** the **virtual page number (VPN)** and **page offset** from the virtual address.
3. The MMU first checks the **TLB** for a cached translation of the VPN.
4. **If the translation is found in the TLB (TLB hit):**
   (a) The corresponding **physical frame number** is retrieved directly.
5. **If the translation is not found (TLB miss → page table walk):**
   (a) The MMU uses the active **page table** to look up the VPN and obtain the PFN.
   (b) The new mapping is inserted into the TLB for future accesses.
6. **If the page table entry is invalid or `present = 0` (page fault):**
   (a) The CPU triggers a **trap into the OS kernel**.
   (b) The **page fault handler** in the OS's **memory management subsystem** is invoked.
   (c) The handler checks whether the page is:
      i. **Never allocated before** → allocate a new physical page and zero-initialize it.
      ii. **Swapped out to disk** →
         A. Check the **swap cache** for the page.
         B. If found in the swap cache, use the cached page directly.
         C. If not in the cache, read the page from the **swap space** on disk into a free physical frame, and insert it into the swap cache.
   (d) The newly loaded or allocated page is mapped in the page table, and `present` is set to `1`.
   (e) The TLB entry for the VPN is updated if needed.
   (f) The faulting instruction is retried.
7. The final **physical address** is then used to **access RAM**.

Swapping
Transfer of memory pages between physical RAM and disk-based swap space to free up RAM.
- **Swap Space** Disk region reserved for evicted pages; holds non-resident memory to extend usable RAM.
- **Swap Cache** In-memory buffer of recently swapped-out pages; enables fast lookup and avoids redundant disk I/O during swap-in.

-Swap-Outs:
1. The OS detects that the number of free physical pages has fallen below a predefined threshold.
2. A background kernel thread is triggered to reclaim memory.
3. The kernel scans memory to identify candidate pages for eviction, typically using an aging or Least Recently Used algorithm.
4. A candidate page is selected if:
   (a) It is not currently in use (i.e., not recently accessed),
   (b) It is not locked, pinned, or shared with kernel-critical structures.
5. If the selected page is dirty (i.e., has been modified), its contents are written to the swap space on disk.
6. The page is inserted into the **swap cache** so it can be quickly retrieved if needed again.
7. The page table entry is updated: `present = 0`, and the swap location is recorded.
8. The physical frame is freed and returned to the pool of available memory.

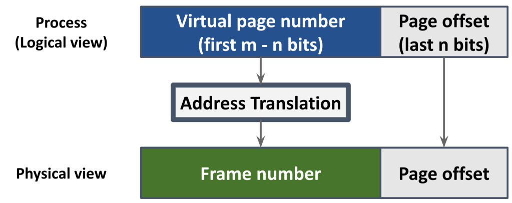
CPU Cache.
The **CPU cache** is a small, fast memory located on or near the processor that stores copies of frequently accessed data from main memory (RAM). Its purpose is to reduce memory access latency and improve overall performance by exploiting **temporal** and **spatial locality**.

Cache Levels:
- **L1 Cache (Level 1):** The smallest and fastest cache, located right on the CPU core. 64 KB, but takes $< 1nsec$ to access.
- **L2 Cache (Level 2):** Larger and slightly slower than L1, still located on the CPU core.256-512 KB, takes $< 4nsec$ to access.
- **L3 Cache (Level 3):** Shared among multiple cores, larger but slower than L1 and L2. Located on the CPU die. 6-32 MB, takes $10s\ of\ nsec$ to access.

Hierarchy Behavior:
Caches form a **hierarchy**: the CPU checks L1 first, then L2, then L3, and finally main memory if needed. This optimizes for latency and hit rate.

Paging.
Paging divides virtual memory into fixed-size **pages**, mapped to physical **frames** via a **page table**. It avoids **ext. fragmentation**, but may cause **internal fragmentation** if a page is only partially used.

Address Representation.
Virtual Addresss Size is not always equal to Physical Address Size!
Let m be the number of bits in the **virtual address** (i.e., $m = \log_2(\text{virtual address space size})$).
This also corresponds to the number of bits required to uniquely address every byte in the virtual memory. If the virtual memory consists of 2^k pages and each page is 2^n bytes, then the total virtual address space is $2^k \times 2^n = 2^m$ bytes, and thus $m = k + n$.



A virtual address is split into:
- **Offset** (n bits): identifies a byte within a page of size 2^n bytes, where $n = \log_2(\text{page size})$.
- **Virtual Page Number (VPN)** ($m - n$ bits): selects the page entry from the page table.

Page Table.
A process's **page table** maps each **Virtual Page Number (VPN)** to a **Physical Frame Number (PFN)**. The page table is indexed by the VPN and stores **Page Table Entries (PTEs)**.

Linear Page Table.
A single-level table with 2^{m-n} entries. Each VPN directly indexes a PTE. Simple but potentially large: $\Rightarrow$ Table size $= 2^{m-n} \times$ PTE size (may span multiple memory pages).

PTE Content (Typical Metadata Bits):
- **Present bit (P):** Valid address translation exists
- **Protection bits (R/W/X):** Read, write, execute permissions
- **User/Supervisor (U/S):** Access control (user vs. kernel)
- **Dirty bit (D):** Set if the page has been modified
- **Access/Reference bit (A):** Set on access; used in replacement policies

Multi-Level Page Tables.
Used to reduce memory overhead. Split the VPN into multiple parts to form a tree-like hierarchy. The **offset** remains the last $n = \log_2(\text{page size})$ bits of the address.

General Breakdown (Multi-Level Paging):

$$\text{Virtual Address Size} = m = \sum k_i + n$$

where:
- $n = \log_2(\text{page size})$: offset bits
- k_i: number of bits used at each level of the page table

Two-Level Page Table:

$$\text{VPN} = \underbrace{\text{Level 1 index}}_{k_1} \parallel \underbrace{\text{Level 2 index}}_{k_2}, \quad k_1 + k_2 = m - n$$

- Level 1 index (k_1 bits): selects the **page directory entry (PDE)**
- Level 2 index (k_2 bits): selects the **page table entry (PTE)** from the second-level table

Three-Level Page Table:

$$\text{VPN} = \underbrace{\text{L1 index}}_{k_1} \parallel \underbrace{\text{L2 index}}_{k_2} \parallel \underbrace{\text{L3 index}}_{k_3}, \quad k_1 + k_2 + k_3 = m - n$$

- Level 1 index (k_1 bits): selects **first-level page directory**
- Level 2 index (k_2 bits): selects **second-level directory/table**
- Level 3 index (k_3 bits): selects the final **PTE**

Bit Allocation in Multi-Level Page Tables:
The VPN portion ($m-n$ bits) is divided into L parts ($k_1+k_2+\cdots+k_L = m-n$), one per level.
- **Bit division is not necessarily even.** Systems may assign more bits to higher levels.
- If uneven, **last levels (closer to the leaf)** typically receive **fewer bits** (fewer entries).
- Each level i contains 2^{k_i} entries, indexing the next level or the final PTE.

Segmentation.
Segmentation divides memory into variable-sized **logical segments** (e.g., code, data, stack), each with a base and limit. It aligns with program structure and supports segment-level protection, but suffers from **ext. fragmentation** due to variable-sized allocations.

File System API. Kernel provides access to files and directories through system calls. Files are represented using **File Descriptors (FDs)** — integers indexing into a **per-process FD table**.

- **File Descriptor:** Non-negative int returned by `open()`. Index into the process's FD table.
- **Open File Description (OFD):** Kernel object holding metadata like file offset, mode, and inode reference.
- **Per-Process Table:** Each process has its own FD table mapping integers to OFDs.
- **Reserved FDs:** 0 = `stdin`, 1 = `stdout`, 2 = `stderr`.
- **FD Allocation:** `open()` returns the lowest unused FD.

Common FS System Calls. Core interface for file I/O and positioning.

- `open(path, flags, mode)`: Open file, return FD. Flags: `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_CREAT`, `O_TRUNC`.
- `read(fd, buf, count)`: Read up to `count` bytes from FD into buffer.
- `write(fd, buf, count)`: Write up to `count` bytes from buffer to FD.
- `lseek(fd, offset, whence)`: Move file offset. `SEEK_SET` = from start, `SEEK_CUR` = from current, `SEEK_END` = from end.
- `unlink(path)`: Remove (delete) a file. File is deleted once no processes have it open.
- `fsync(fd)`: Flush all modified file data and metadata to disk.
- `fstat(fd, &statbuf)`: Get metadata about file referred to by FD (size, mode, timestamps, etc.).

File Internals. A file consists of two main components: **data** and **metadata**.

- **Data:** Actual user content, stored in data blocks.
- **Metadata:** Stored in the inode:
  - Owner, permissions, timestamps
  - File size, type, and block pointers
  - Device ID and inode number
- **Inode:** Kernel-managed structure that holds file metadata and pointers to data blocks.
- **Filename:** Not stored in the inode. Stored in directory entries.
- **Uniqueness:** File is uniquely identified by (device, inode).
- **Allocation:** Inodes are created and managed by the file system.

Path Resolution, Step-by-Step. Converting a pathname to a target inode involves directory traversal.

1. Begin at root (/) or current working directory.
2. Parse each path component left to right.
3. For each component, look up entry in current directory: `filename → inode number`.
4. Follow inode to next directory or target file.
5. Final inode is cached in the file descriptor table for future access.

Directory Internals. A directory is a **special file** used to organize and reference files. - **Structure:** Stored like a regular file, but marked with a directory flag in its inode.
- **Content:** Contains a list of entries:
- Each entry is a `filename` → `inode number` mapping
- **Function:** Maps human-readable names to inodes.
- **Permissions:** Govern ability to read entries, traverse, or modify structure.
- **Traversal:** Used in path resolution to step through the hierarchy.
- **Isolation: Normal processes cannot write directly to directories as a file (writing arbitrary bytes is not allowed).**

Mount Point.
Directory where a filesystem is attached to the global namespace. Root filesystem is mounted at /.

mount command.
Attach filesystem to directory (mount point).
`mount [device] [dir]`
mount device at dir.

df command.
Display filesystem disk usage.
`df -- report filesystem disk usage.`

Partition.
Linear view of persistent storage: sequence of `N` blocks, indexed `0 to N-1`.

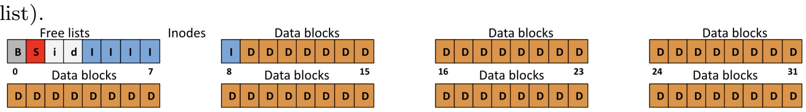
Inode.
Filesystem structure representing a file. Each inode stores:

- File type and permissions
- Owner UID / GID
- File size
- Timestamps (created, modified, accessed)
- Pointers to data blocks
- Link count (number of directory references)

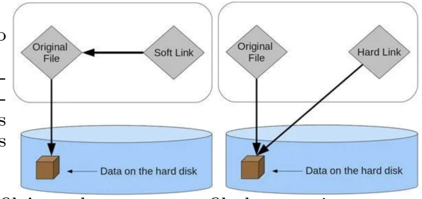
Block Usage.

- **Data blocks** — store file contents.
- **Metadata blocks** — store filesystem structures.
- **Boot block** — code for booting (block 0).
- **Superblock** — global File System metadata:
  – number of inodes, number of data blocks
  – Inode table location
  – Free inode/data block management

Block Types.
(I) **Inode Block** — contains array of inodes (e.g., 16 inodes/block at 256B each).
(D) **Data Block** — holds actual file content (user data).
(i)/(d) **Bitmap Blocks** — tracks used/free inodes and data blocks (free list).
(S) **Superblock** — global FS metadata:
– Total inodes and data blocks
– Inode table start block
– Bitmap locations
– FS state (clean, dirty)
(B) **Boot Block** — bootloader code; block 0 of partition.



Links. Multiple names can refer to files using links.
- **Hard Link:** Maps a file name directly to the **same inode** as the original file, deleting one's file name does not remove the actual data as long as another link still exists.



- **Symbolic Link:** Logically maps a file's path to a target file by creating an actual link to the original file with a **new inode number**, and becomes broken or invalid if the target file is removed or deleted.

File Allocation.
Strategy to map file data to disk blocks. Managed via pointers in inodes or allocation tables.

File Allocation Table.
Layout: A table holds block chains for files
How it works:
1. Inode contains pointer to first metadata block.
2. Each Metadata block contains pointer to next metadata block.
3. Read the corresponding data block in FAT table and repeat until end-of-file marker is reached.

Pros: No ext. fragmentation, avoids mixing data and metadata, only requires locating the first block.
Cons: Poor random access, limited metadata, FAT must remain in memory.

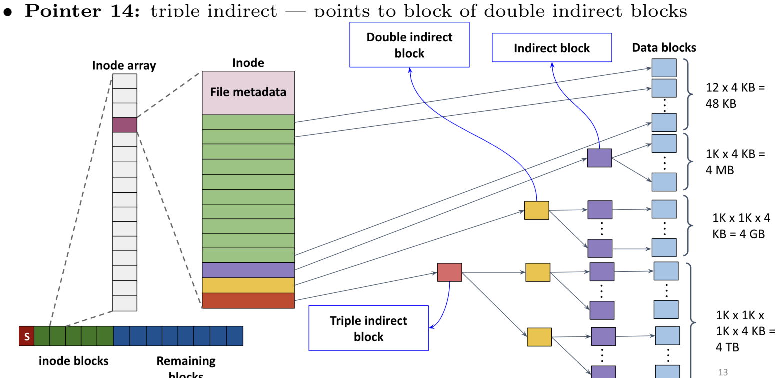
Contiguous Allocation.
Pros: fast access, simple offset computation
Cons: fragmentation, difficult resizing
Layout: A file is a sequence of consecutive blocks

Linked Allocation.
Pros: No fragmentation, simple - find the first block of a file.
Cons: slow random access, pointer overhead, mix data/metadata in the same block
Layout: Inode contains pointer to first block, each block contains pointer to next

Multi-Level Indexing.
Each inode holds pointers to data blocks directly or indirectly.
Pros: No ext. fragmentation, no conflating between data/metadata, Scales to large files, efficient for small files (via direct blocks), flexible block usage
Cons: Indirection overhead for large files, slower access for deep pointer chains.

Inode Block Structure.
Inode forms a fixed, asymmetric tree. Leaf nodes = fixed-size data blocks.

- **Pointers 0–11:** direct — point to data blocks
- **Pointer 12:** single indirect — points to block of data block pointers
- **Pointer 13:** double indirect — points to block of single indirect blocks
- **Pointer 14:** triple indirect — points to block of double indirect blocks



Direct Pointers:
`number_of_direct_pointers`
Single Indirect Pointer:
$N = \frac{\text{block\_size} - r}{\text{pointer\_size}}$
Double Indirect Pointer:
$N^2 = \left(\frac{\text{block\_size} - r}{\text{pointer\_size}}\right)^2$
Triple Indirect Pointer:
$N^3 = \left(\frac{\text{block\_size} - r}{\text{pointer\_size}}\right)^3$

Total Addressable Blocks:
$\text{number\_of\_direct\_pointers} + \sum_{i=1}^{3}\left(\frac{\text{block\_size} - r}{\text{pointer\_size}}\right)^i$
Maximum File Size (bytes):
`total_addressable_blocks × block_size`
- The count N represents the number of data blocks that can be reached through that single pointer.
- If no metadata is reserved in the indirect blocks, then set $r = 0$.

Reading.

	data bitmap	inode bitmap	root inode	cs202 inode	w07 inode	root data	cs202 data	w07 data[0]	w07 data[1]
open("cs202/w07")			read()						
						read()			
				read()					
							read()		
					read()				
read()					read()				
								read()	
					write()				

Writing.

	data bitmap	inode bitmap	root inode	cs202 inode	w07 inode	root data	cs02 data	w07 data[0]
open("cs202/w07")			read()					
						read()		
				read()				
							read()	
		read()						
		write()						
							write()	
					read()			
					write()			
				write()				
write()					read()			
	read()							
	write()							
								write()
					write()			

Performance Metrics.
- **Latency** — time per I/O (μs–ms)
- **Throughput** — data/sec (MB/s)
- **IOPS** — ops/sec

Caching.
Keep frequently accessed data in memory to reduce latency. Speeds up reads.

Batching.
Group multiple I/O operations to reduce syscall overhead and disk seeks.

Block Cache.
- In-memory cache of disk blocks
- **Good for reads** — avoids repeated disk access
- **Limited for writes** — consistency requires flushing

Write Caching.
- **Write-through** — data written to both cache and disk immediately (slower, but consistent)
- **Write-back** — data stays in cache, flushed later (faster, but risk of data loss on crash)
- `fsync()` — forces flush of dirty blocks to disk

Consistency Update Problem.
File system metadata may become inconsistent due to crashes during updates.

File System Checker (fsck).
Tool to scan and repair on-disk metadata inconsistencies.
Cons: Functionality - fix not always obvious or correct,
Performance - slow; may take hours.

fsck Fix Examples.
- **Link Count Inconsistency** — problem: inode's link count $\neq$ number of directory entries pointing to it; fix: correct link count to match actual references.
- **Lost Inodes** — problem: inode has link count > 0 but no directory entry points to it; fix: move file to `lost+found` for recovery.
- **Data Bitmap Errors** — problem: inode uses a block, but bitmap marks it free; fix: set corresponding bitmap bit to "used".
- **Duplicate Pointers** — problem: two inodes point to same data block; fix: duplicate the block and update one inode to preserve both files.
- **Invalid Pointers** — problem: inode points to a block beyond partition size; fix: remove invalid pointer to prevent access.

Journaling.
Crash-consistency technique. Metadata updates are first written to a log (journal) before applying to the main file system
Pros. Fast crash recovery (no full scan); Metadata consistency guaranteed; Safer than fsck in most cases.

How It Works.
1. Group changes into a **transaction**
2. Write to journal: `TxBegin | changes | TxEnd | Valid`
3. `Valid` block = transaction committed
4. Apply changes to file system (checkpoint)
5. Clear transaction from journal

Recovery.
1. For each uncleared transaction (stil in journal):
2. **If no `Valid`:** discard
3. **If `Valid` found:** replay transaction to disk

I/O Interrupt.
Device triggers an interrupt when it needs service.
Pro: efficient for unpredictable events.
Con: high overhead per interrupt.

I/O Polling.
CPU repeatedly checks device status register.
Pro: low overhead when frequent.
Con: wastes CPU cycles if idle or infrequent.

Livelock.
System busy handling I/O (e.g., interrupts) but makes no progress in real work. Spinning without forward progress.

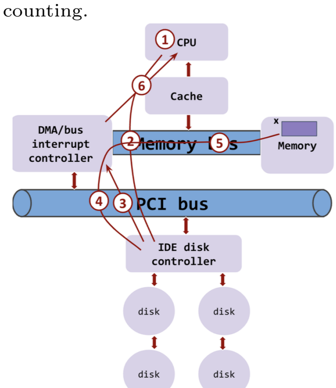
Real-World Strategy.
- Use **interrupts** for overlap (slow devices)
- Use **polling** for short bursts, small data, high performance
- **Coalescing**: delay, batch multiple responses for efficiency

PIO (Programmed I/O).
CPU tells the device **what data to read/write**.

DMA (Direct Memory Access).
CPU tells the device **where data is**.

DMA Controller.
Hardware unit that transfers data between device and memory without CPU involvement. Handles address incrementing and byte counting.

DMA Transfer Workflow. (device to memory transfer.)
1. The device driver receives an instruction to transfer disk data to a buffer at address X.
2. The driver commands the disk controller to transfer C bytes from disk to the buffer at address X.
3. The disk controller initiates the DMA transfer operation.
4. The disk controller sends each byte to the DMA controller.
5. The DMA controller transfers bytes to buffer X, incrementing the memory address and decrementing C until C = 0.
6. When C = 0, the DMA controller interrupts the CPU to signal completion of the transfer.



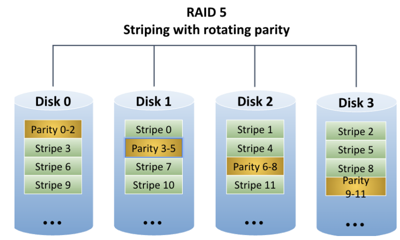
Disk Storage
Goal: fast, reliable, affordable persistent data access. Must return what was written—quickly and without loss.

RAID (Redundant Array of Inexpensive Disks)
Combines multiple physical disks into one logical unit for performance and fault tolerance:
Allows for a higher throughput and reliability.

RAID 0 (Striping)
data split across $N \geq 2$ disks
+ High throughput, full capacity
– No redundancy
Read: from corresponding disk
1 disk fails: all data lost

RAID 1 (Mirroring)
data duplicated on 2 disks
+ Survives 1-disk failure, fast reads
– 2× storage cost
Read: from either copy (load-balanced)
1 disk fails: use remaining mirror

RAID 5 (Striping + Parity) — block-level striping + distributed parity across $N \geq 3$ disks
+ Efficient redundancy, survives 1-disk failure
– Writes = read + modify parity (slow)
Read: from data disk if intact
1 disk fails: reconstruct via XOR:
Missing Block = Parity $\oplus \bigoplus_{i \neq \text{failed}} \text{Data}_i$
Or define parity as:
Parity = $\bigoplus_{i=1}^{N-1} \text{Data}_i$



Context Switch
Triggered by *timer interrupt* or blocking events. Used to enforce fairness and preempt misbehaving threads.

Procedure.
1. **Save state** of current thread (PC, registers, etc.) to PCB
2. **Choose next thread** via scheduler
3. **Restore state** of selected thread from its PCB
4. **Resume execution** via `return-from-trap`

Timer interrupts ensure control returns to the OS periodically, allowing preemption and preventing CPU hogging.

Scheduling Policy — strategy for choosing the next thread

Scheduling Metrics
- **CPU Utilization:** % time CPU is busy
- **Turnaround Time:** completion time − submission time
- **Response Time:** time from submission to first response (eg. submission to first time a thread is scheduled)

FIFO Scheduling (First-In, First-Out)
Non-preemptive (waits for completion before scheduling next thread). Runs threads in order of arrival.
1. **Enqueue** new threads at the tail of the ready queue
2. **Select** the thread at the head of the queue
3. **Run** it to completion or blocking
4. **Repeat** with the next thread in queue

Pros: Simple, fair (by arrival order), minimal overhead
Cons: Poor responsiveness, suffers from *convoy effect* (short jobs wait behind long ones)

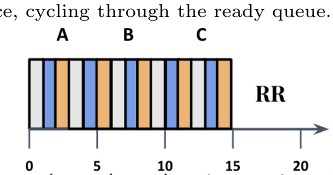
SJF Scheduling (Shortest Job First)
Non-preemptive. Picks the thread with the shortest remaining execution time.
1. **Estimate** or know job lengths in advance
2. **Select** the shortest job from the ready queue
3. **Run** it to completion or blocking
4. **Repeat** with the next shortest job

Pros Minimizes average turnaround time (optimal under perfect knowledge)
Cons Requires job length estimates, risk of starvation for long jobs

STCF Scheduling (Shortest Time to Completion First)
Preemptive SJF. Always runs the thread with the least remaining time.
1. **Track** remaining time for all ready/running threads
2. **On arrival** of a new thread, compare its time to current thread
3. **Preempt** if new thread has shorter remaining time
4. **Run** the shortest job until completion or preemption

Pros Optimal average turnaround time under preemption
Cons Requires accurate time estimates, may cause starvation of longer jobs, bad Response Time

Round Robin Scheduling
Preemptive. Each thread gets a fixed time slice, cycling through the ready queue.
1. **Enqueue** threads in arrival order
2. **Run** the thread at the head for one quantum
3. **Preempt** if not finished; move to tail of queue
4. **Repeat** with the next thread in queue



Pros Fair, responsive, avoids starvation
Cons High context switch overhead if time slice is too small; poor for short jobs if time slice is too large

MLFQ Scheduling (Multi-Level Feedback Queue)
Preemptive. Dynamically adjusts thread priority based on behavior.

Rules
1. If priority(A) > priority(B), **A runs**
2. If priority(A) = priority(B), **A, B run in Round Robin**
3. **New threads start at top priority**
4. If a thread uses up its time slice, **demote its priority**
5. Periodically **boost all threads to top priority** (prevents starvation)

Procedure
1. Maintain multiple ready queues by priority level
2. Insert new or boosted threads at highest priority
3. Select thread from highest non-empty queue
4. Run using Round Robin within queue
5. **Demote if thread uses full time slice; boost periodically** !!!!