Computer Systems - CheatSheet

IN BA4 - Katerina Argyraki

Notes by Ali EL AZDI

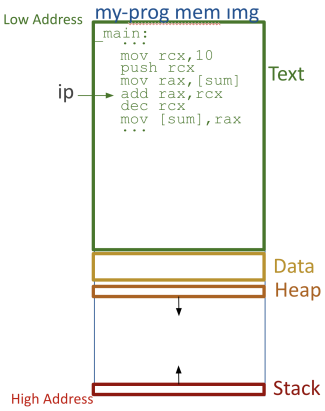
A Computer Systems Cheatsheet has been authorized for the upcoming exam, and I'm sharing a copy of mine for anyone interested. It provides a concise summary of the key concepts and techniques covered in the course. For any updates or suggestions, feel free to reach out to me on Telegram at `elazdi_al`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

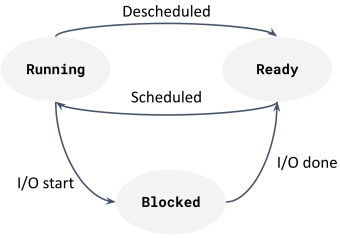
  If an exception happens in kernel mode, an internal routine is called but no mode switch occurs.

  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.

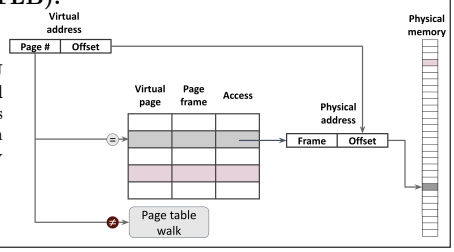
`exit()` — *Terminate a process cleanly*
- Frees system resources allocated to the process.
- Sends termination status to the parent.
- If main function of a program returns, `exit()` is implicitly called.

`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.

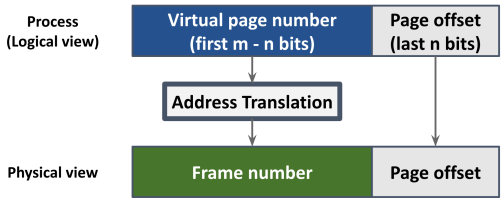
-Swap-Outs:
1. The OS detects that the number of free physical pages has fallen below a predefined threshold.
2. A background kernel thread is triggered to reclaim memory.
3. The kernel scans memory to identify candidate pages for eviction, typically using an aging or Least Recently Used algorithm.
4. 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` (read-only access), `O_WRONLY` (write-only access), `O_RDWR` (read-write access), `O_CREAT` (create if doesn't exist), `O_TRUNC` (truncate to zero length).
- `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.).
- `close(fd)`: Close the file descriptor.
  Trying to close an already closed FD results in an error.

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:

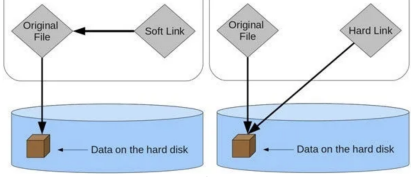
- File type and permissions
- Owner UID / GID
- File size
- Timestamps (created, modified, accessed)
- Pointers to data blocks
- Link count (number of directory references)

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.

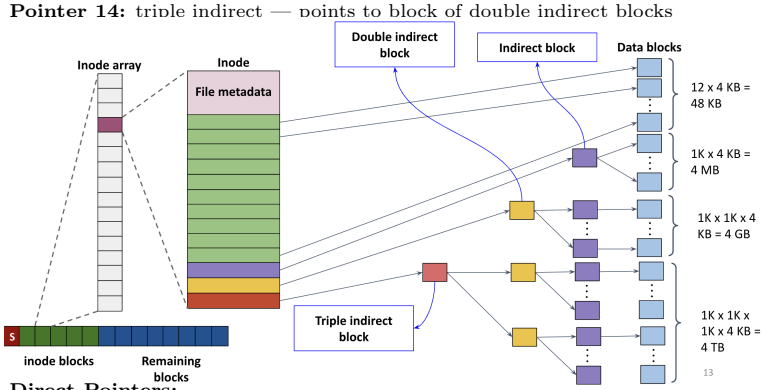
Pros: No ext. fragmentation, avoids mixing data and metadata, only requires locating the first block.
Cons: Poor random access, limited metadata, FAT must remain in memory.

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

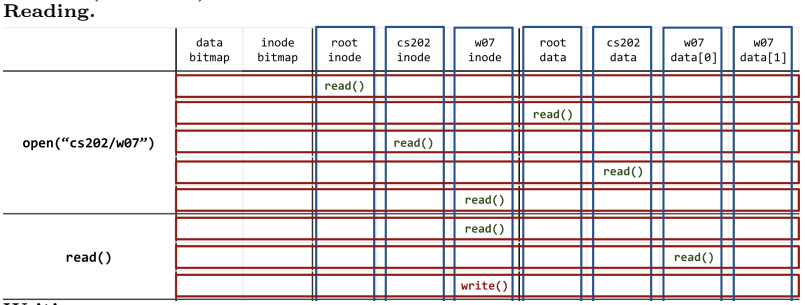


Direct Pointers:
`number_of_direct_pointers`
Single Indirect Pointer:
$N = \frac{\text{block\_size} - r}{\text{pointer\_size}}$
Double Indirect Pointer:
$N^2 = \left(\frac{\text{block\_size} - r}{\text{pointer\_size}}\right)^2$
Triple Indirect Pointer:
$N^3 = \left(\frac{\text{block\_size} - r}{\text{pointer\_size}}\right)^3$

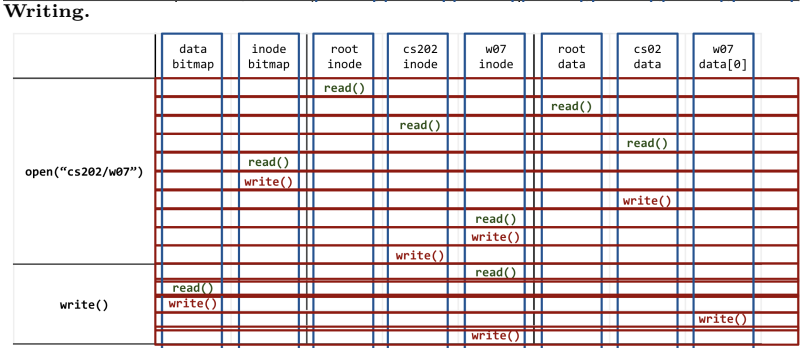
Total Addressable Blocks:
$\text{number\_of\_direct\_pointers} + \sum_{i=1}^{3} \left(\frac{\text{block\_size} - r}{\text{pointer\_size}}\right)^i$
Maximum File Size (bytes):
$\text{total\_addressable\_blocks} \times \text{block\_size}$

- 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.



Writing.



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

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

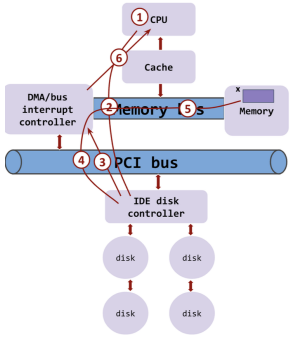
DMA (Direct Memory Access).
CPU tells the device **where data is**.

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.

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

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

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



`ls -l` command output.

```
pe     Usr  Grp  Oth  Ln   Owner       Group       Size     Mon    Day    Time/Year Name
/-/l]  ppp  ppp  ppp  [n]  [username]  [groupname] [bytes]  [Mon]  [DD]   [HH:MM / YYYY] [filename]
```

Permissions Breakdown:
`Type` — File type:
- `-` = regular file
- `d` = directory
- `l` = symbolic link

`Usr`, `Grp`, `Oth` — Permission triplets for:
- `r` = read
- `w` = write
- `x` = execute
- `-` = permission not granted

- `Ln` — Number of hard links
- `Owner` — Username of the file owner
- `Group` — Group associated with the file
- `Size` — File size in bytes
- `Mon, Day, Time/Year` — Last modification timestamp
- `Name` — File or directory name

`malloc(size_t size)` | Allocates `size` bytes. Returns `void*` to uninitialized memory.
Return: pointer to memory or `NULL` on failure
`calloc(size_t nmemb, size_t size)` | Allocates space for `nmemb` elements of `size` bytes each. Memory is zero-initialized.
Return: pointer to zeroed memory or `NULL`
`memset(void *ptr, int value, size_t num)` | Sets the first `num` bytes of `ptr` to `(unsigned char)value`.
Use: initialize or reset memory
`realloc(void *ptr, size_t new_size)` | Resizes previously allocated block to `new_size` bytes.
Return: pointer to new block (contents preserved), or `NULL`

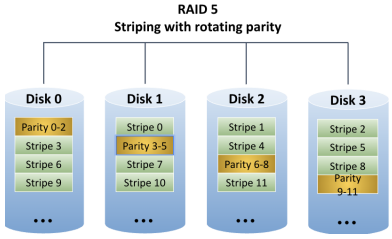
Disk Storage
Goal: fast, reliable, affordable persistent data access. Must return what was written—quickly and without loss.

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 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 = $\bigoplus_{i=1}^{N-1} \text{Data}_i$

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



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

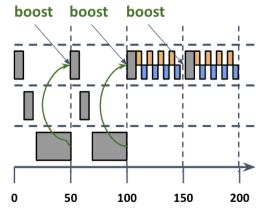
Round Robin Scheduling
Preemptive. Each thread gets a fixed time slice, cycling through the ready queue.
1. **Enqueue** threads in arrival order
2. **Run** the thread at the head for one quantum
3. **Preempt** if not finished; move to tail of queue
4. **Repeat** with the next thread in queue

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** !!!!



Recursive Struct.

```
typedef struct Node {
    int value;
    struct Node* next;
} Node;
```

Generic Pointers.

```
void apply(void* array, size_t len, size_t elem_size,
void (*func)(void*)) {
    for (size_t i = 0; i < len; i++) {
        void* elem = (char*)array + i * elem_size;
        func(elem);
    }
}
```

Network Addressing & Web Protocols:
- **IP Address:** 32-bit (IPv4) or 128-bit (IPv6) unique identifier for network devices. Format: dotted decimal (e.g., 192.168.1.1) or hexadecimal notation.
- **Port Number:** 16-bit identifier (0-65535) for specific application/service on a host. Well-known ports: HTTP=80, HTTPS=443, FTP=21, SSH=22.

HTTP(HyperText Transfer Protocol): Stateless application-layer protocol for web communication.
- **Stateless Protocol:** Server doesn't maintain client state between requests. Each request is independent and self-contained.
- **Web Objects:** Files identified by URLs (HTML pages, images, videos, CSS, JavaScript, etc.). Each object has a unique URL.
- **Web Page:** Base HTML file + referenced objects (images, stylesheets, scripts). **Single page may require multiple HTTP requests.**

HTTP Methods:
GET: Retrieve resource from server. Safe, idempotent.
HEAD: Like GET but returns only headers (no body). Used for metadata.
POST: Submit data to server (forms, uploads). Non-idempotent.
PUT: Update/create resource. Idempotent.
DELETE: Remove resource. Idempotent.

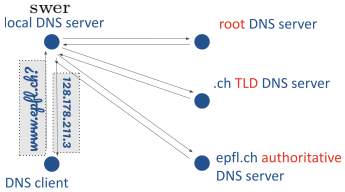
Common Response Codes:
200 OK: Request successful
301/302: Redirect (permanent/temporary)
404 Not Found: Resource doesn't exist
500 Internal Server Error: Server-side error

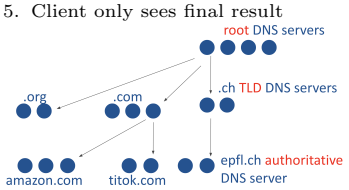
- **Cookies:** Small data stored by browser, sent with requests to maintain state across stateless HTTP. Used for authentication, preferences, tracking.
- **Web Caching:** Store frequently accessed objects closer to client to reduce latency and server load. Types: browser cache, proxy cache, and **Edge Caches** (part of a CDN) which are geographically distributed.
- **Max-Age:** Cache-Control directive specifying how long (in seconds) a resource can be cached before becoming stale. Example: `max-age=3600` (1 hour).

DNS (Domain Name System): Hierarchical distributed naming system that translates human-readable domain names to IP addresses.
- **DNS Name:** Hierarchical structure (e.g., www.epfl.ch). Read right-to-left: .ch (TLD), epfl (2nd level), www (subdomain).
- **Reserved Port:** UDP/TCP port 53.
- **DNS Server Hierarchy:**
  - **Root DNS Servers:** 13 logical servers (A-M) managing top-level domains
  - **TLD DNS Servers:** Manage specific top-level domains (.com, .org, .ch, etc.)
  - **Authoritative DNS Servers:** Store actual DNS records for specific domains
- **TTL (Time To Live):** Specifies how long DNS records can be cached (in seconds).
- **Protocol:** Primarily UDP for speed. TCP used for zone transfers or when response exceeds 512 bytes.
- **Client-side Syscalls (UDP):**
  - **socket():** Creates a new UDP socket endpoint.
  - **bind():** (Optional) Assigns a local port to the socket. If not called, an ephemeral port is assigned by the kernel on first 'sendto()'.
  - **sendto():** Sends the DNS query to the server's IP and port 53.
  - **recvfrom():** Waits to receive the DNS response from the server.

DNS Resolution Approaches:
Iterative Query:
1. Client queries local DNS
2. Local DNS queries root, gets TLD referral
3. Local DNS queries TLD, gets authoritative referral
4. Local DNS queries authoritative, gets answer
5. Each server returns **referral** or answer

Recursive Query:
1. Client queries local DNS
2. Local DNS recursively queries on behalf of client
3. Each server in chain does the work
4. Final answer returned through chain
5. Client only sees final result





Packet Switching & Network Delays:
- **Packet Switch:** Network device that forwards packets using store-and-forward switching. Receives entire packet before forwarding.
- **Transmission Delay:** Time to push packet onto link. $\mathbf{d_{trans}} = \frac{\mathbf{L}}{\mathbf{R}}$, where L is the packet size (bits), R is the transmission rate (bps).
- **Propagation Delay:** Time for signal to travel link. $\mathbf{d_{prop}} = \frac{\mathbf{d}}{\mathbf{s}}$, where d is the distance, s is the signal speed.
- **Queuing Delay:** Time packet waits in output queue before transmission. Depends on congestion level.
- **Total Packet Delay:** $\mathbf{d_{total}} = \mathbf{d_{proc}} + \mathbf{d_{queue}} + \mathbf{d_{trans}} + \mathbf{d_{prop}}$, where $\mathbf{d_{proc}}$ is the processing delay.
- **Packet Loss:** Occurs when packet arrives at full queue buffer. Packet is dropped.
- **Traffic Analysis:**
  - **Arrival Rate (λ):** Average rate at which packets arrive.
  - **Packet Inter-arrival Time:** Time between consecutive packet arrivals. Average inter-arrival time = $\frac{\mathbf{1}}{\lambda}$.
  - **Departure Rate (μ):** Rate at which packets are served/transmitted.
  - **If $\lambda > \mu$:** Queue grows indefinitely, delay increases, eventually packet loss.
  - **If $\lambda < \mu$:** Stable system, finite queuing delay.
  - **If $\lambda = \mu$:** System at capacity, high sensitivity to bursts.
- **Average Throughput:** min(transmission rates along path) - the bottleneck link determines end-to-end throughput.
- **Transfer Time (N packets through bottleneck):** Total time to send N packets of size L bits through a path with bottleneck link rate R_b.
  - Transfer Time = $\mathbf{d}_{\text{to bottleneck}} + \frac{\mathbf{N}\times\mathbf{L}}{\mathbf{R_b}} + \mathbf{d}_{\text{from bottleneck}}$
  - Where $d_{\text{to bottleneck}}$ and $d_{\text{from bottleneck}}$ include propagation and transmission delays on non-bottleneck links

Internet Structure:
- **ISP (Internet Service Provider):** Company that provides Internet access.
  - **Tier-1 ISP:** Global network, peers with all other Tier-1s. The backbone.
  - **Regional ISP:** Connects to one or more Tier-1 ISPs.
  - **Access ISP (Last-mile):** Connects end-users (homes, businesses).
- **IXP (Internet Exchange Point):** Physical infrastructure where multiple ISPs connect to exchange traffic directly, avoiding Tier-1 transit.

UDP (User Datagram Protocol): Connectionless, unreliable transport protocol.
- **Characteristics:** No connection establishment, no reliability guarantees, no flow/congestion control, minimal overhead.
- **Header Structure (8 bytes total):**
  - Source Port: 16 bits
  - Destination Port: 16 bits
  - Length: 16 bits (UDP header + data length)
  - Checksum: 16 bits (optional in IPv4, mandatory in IPv6)

TCP Protocol Essentials. Transmission Control Protocol provides reliable, ordered delivery over unreliable networks.
- **Three-Way Handshake:** Connection establishment process.
  - **Step 1:** Client sends SYN with initial sequence number (ISN)
  - **Step 2:** Server responds with SYN-ACK (server ISN + ACK of client ISN+1)
  - **Step 3:** Client sends ACK (acknowledges server ISN+1)
- **Sequence Number (SEQ):** 32-bit field identifying byte position in data stream. Initial value chosen randomly. Each byte of data increments SEQ by 1. Used for ordering and duplicate detection.
- **Acknowledgment Number (ACK):** 32-bit field indicating next expected sequence number from sender.

Congestion Window (cwnd): TCP sender's estimate of how many bytes can be sent without causing network congestion, dynamically adjusted based on network conditions.
- **Purpose:** Controls sending rate to prevent network overload and packet loss due to buffer overflow at intermediate routers.
- **Relationship:** Effective Window = min(cwnd, rwnd) where rwnd = receiver advertised window.

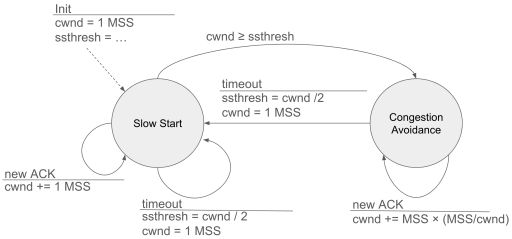
Maximum Segment Size (MSS): Largest amount of data bytes that TCP can send in a single segment, excluding TCP and IP headers.
Timeout Calculation: TCP adaptively calculates retransmission timeout to balance quick recovery from lost packets with avoiding unnecessary retransmissions due to network delay variations.
- **SampleRTT:** Measured time for a specific segment to be sent and acknowledged (not retransmitted segments)
- **EstimatedRTT:** Exponentially weighted moving average of RTT samples to smooth out fluctuations
  $\text{EstimatedRTT} = (\mathbf{1} - \alpha) \times \text{EstimatedRTT} + \alpha \times \text{SampleRTT}$
  where α = smoothing factor for RTT estimation (typically 0.125)
- **DevRTT:** Estimate of RTT variation to account for network jitter and delay variability
  $\text{DevRTT} = (\mathbf{1} - \beta) \times \text{DevRTT} + \beta \times |\text{SampleRTT} - \text{EstimatedRTT}||$
  where β = smoothing factor for deviation estimation (typically 0.25)

$\text{TimeoutInterval} = \text{EstimatedRTT} + \mathbf{4} \times \text{DevRTT}$
Checksum: 16-bit Internet checksum. Computed over TCP header, data, and pseudo-header (source IP, dest IP, protocol, TCP length). Uses 1's complement arithmetic.

TCP Congestion Control (Tahoe & Reno): Algorithms that dynamically adjust sending rate to prevent network congestion while maximizing throughput.
- **Common Phases (Both Algorithms):**
  - **Slow Start:** cwnd = cwnd + MSS for each ACK. Exponential growth until cwnd $\geq$ ssthresh.
  - **Congestion Avoidance:** cwnd = cwnd + $\frac{\text{MSS}^2}{\text{cwnd}}$ for each ACK. Linear growth.
  - **Fast Retransmit vs No Fast Retransmit:**
    - **Enabled:** Upon 3 duplicate ACKs, immediately retransmit lost segment. Faster recovery, reduced latency.
    - **Disabled:** Must wait for timeout expiration to detect and retransmit lost segments. Slower recovery, higher latency.

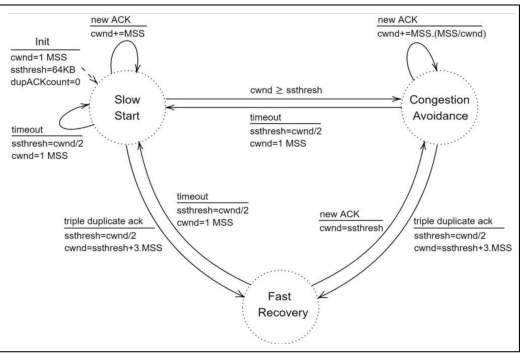
Tahoe
No fast-retransmit (Conservative):

Timeout
ssthresh = $\lfloor$cwnd/**2**$\rfloor$,
cwnd = MSS (slow start)
Behavior:
Always returns to slow start after ANY loss (more conservative).



Reno (Adaptive):

Timeout:
ssthresh = $\lfloor$cwnd/**2**$\rfloor$,
cwnd = MSS (slow start)
3 Dup ACKs:
ssthresh = $\lfloor$cwnd/**2**$\rfloor$,
cwnd = ssthresh + 3×MSS
(fast recovery)



Network Layers (1-5):
Layer 1 (Physical): Raw bit transmission over physical medium (cables, radio waves)
Layer 2 (Data Link): Frame transmission between directly connected nodes using MAC addresses
Layer 3 (Network): Packet routing between networks using IP addresses
Layer 4 (Transport): End-to-end delivery (TCP/UDP) with port numbers
Layer 5 (Application): User applications and protocols (HTTP, FTP, DNS)

Core Networking Concepts:
Network Interface (NIC): Hardware endpoint with an assigned IP address that sends/receives frames on a link; identified by a MAC address.
Subnet: Set of IP addresses sharing a common prefix; network interfaces with IPs in the same subnet can communicate directly without routing.
IP Prefix: First k bits of an IP address written as k (e.g., 192.168.1.0/24) defining a subnet.
Mask: 32-bit value with k leading 1s followed by 0s indicating prefix length (e.g., 255.255.255.0 for /24).
Router: Device that links subnets together; each router interface belongs to a different subnet and holds a forwarding table.
Forwarding Table: Router table mapping destination prefixes to outgoing interfaces / next-hop addresses. Built by routing algorithms (Dijkstra, Bellman-Ford) or manual configuration.

Dest IP Prefix	Output Interface
192.168.1.0/24	eth0
10.0.0.0/8	eth1
0.0.0.0/0	eth2 (default)

Prefix Merging: Multiple prefixes can be merged if:
(1) Same output interface,
(2) Contiguous address ranges,
(3) Count is power of 2.
Quick check: Group by interface → sort by IP → merge consecutive ranges with same prefix length.
Longest Prefix Match (LPM): Forwarding rule selecting the table entry with the longest matching prefix for a destination IP.
Routing vs Forwarding:
- **Routing:** Control plane process of computing paths and building forwarding tables using routing protocols.
- **Forwarding:** Data plane process of looking up destination in table and sending packet to appropriate output interface.

Switch vs Router:
- **Switch:** Layer-2 device that can be contained within a subnet, forwards frames using MAC addresses. Does not have IP addresses.
- **Router:** Layer-3 device that cannot be contained in a subnet; connects different subnets using IP addresses.

Packet Switch: General term for devices that receive, store, and forward packets (includes switches and routers).

Routing Algorithms (Building Forwarding Tables):
- **Dijkstra's Algorithm (Link-State):**
  1. **Base Table Template:**

Step	Nodes Visited	C(u), p(u)	C(v), p(v)	C(w), p(w)	...
0	$\emptyset$	∞, –	∞, –	∞, –	...
1	...	...	...	...	...
...	...	...	...	...	...

  2. **Initialize (Step 0):**
     – Nodes Visited = empty set
     – Set source node: C(S) = 0, p(S) = –
     – **For** each other node i:
     If direct link S↔i exists with cost c: set C(i)=c, p(i)=S
     Else: C(i)=∞, p(i)=–
  3. **At each Step k $\geq$ 1:**
     – Choose unvisited node u with smallest C(u)
     – Add u to Nodes Visited
     – **For** each neighbor v of u not yet visited:
     alt = C(u) + cost(u,v)
     If alt < C(v): update C(v)=alt and p(v)=u
     – Record new C(·), p(·) values in Step k row
  4. **Terminate:** When all nodes visited or remaining C(·)=∞
- **Bellman-Ford (Distance-Vector):**
  1. **Base Table Template:**

From \ To	u	v	w	z
v				
w				
z				

  2. **Initialize (Iteration 0):**
     – For each neighbor row: put direct link costs to each destination via that neighbor (∞ if no link)
     – For self row: put 0 to self, ∞ to all others
  3. **Each Iteration k $\geq$ 1:**
     – Copy neighbors' rows from their latest announcements
     – For each destination column:
       * Take your direct cost to neighbor + neighbor's cost to destination
       * Do this for every neighbor, pick the smallest total
       * Example: To reach node X, compare: (cost to v) + (v's cost to X) vs (cost to w) + (w's cost to X)
       * Put the smallest result in your row for destination X
     – Repeat for all destinations, update your entire row
  4. **Convergence:** When self row doesn't change between iterations

Algorithm Comparison:
- **Dijkstra:**
  Fast convergence ($O(n^2)$), but requires global topology flooding
- **Bellman-Ford:**
  No flooding (distributed), but slower (max n-1 iterations, where n = # nodes)

Powers of 2:
$2(2^1), 4(2^2), 8(2^3), 16(2^4), 32(2^5), 64(2^6), 128(2^7), 256(2^8), 512(2^9),$
$1024(2^{10}), 2048(2^{11}), 4096(2^{12}), 8192(2^{13}), 16384(2^{14}), 32768(2^{15})$

Network Trace Analysis Guide:
1. **(Optional) DNS resolution**
   (a) Do you already know the server's IP? If not: UDP DNS query (Src: your host, Dst: DNS server, Src Port: ephemeral 2000, Dst Port: 53)
   (b) **ARP-resolve your gateway's MAC before sending DNS packet**
2. **ARP for your default gateway**
   (a) Check ARP cache for gateway IP→MAC mapping
   (b) If missing: Broadcast ARP Request (Eth Dst: ff:ff:ff:ff:ff:ff, ARP target: gateway IP) → Receive Reply → cache MAC
3. **Encapsulate and send packet**
   (a) Ethernet: Src MAC (your NIC), Dst MAC (gateway)
   (b) IP: Src IP (your host), Dst IP (target server), TTL (e.g., 64), compute checksum
   (c) TCP SYN: Src Port (ephemeral 3000), Dst Port (80)
4. **Router processing** Strip Ethernet → verify IP checksum & TTL > 1 → decrement TTL → recalc checksum → routing table lookup
5. **Router ARP for next hop** If next-hop MAC not cached: ARP Request → Reply → store MAC
6. **Router re-encapsulation** New Ethernet header (Src: router interface MAC, Dst: next-hop MAC), same IP packet
7. **Multi-hop repeat** Steps 4-6 at each router: TTL decrement, routing lookup, ARP if needed
8. **Final-hop ARP** Router ARPs for server IP → gets server MAC
9. **Server receives SYN** Strips Ethernet → TCP handshake: server replies SYN-ACK
10. **Client completes handshake** Send ACK → TCP connection established.
    Note: Can send ACK and GET simultaneously
11. **HTTP exchange** Send GET over TCP → server processes → HTTP response (same encapsulation/routing)

Layer 2 Concepts:
MAC Address: 48-bit globally unique hardware identifier. First 24 bits = OUI (Organizationally Unique Identifier), last 24 bits = device-specific.
L2 vs L3 Forwarding:
- **Intra-subnet (L2):** Host A sends frame directly to Host B using B's MAC as destination. Switch receives frame → looks up dest MAC in table → forwards out appropriate port. If MAC unknown, floods to all ports.
- **Inter-subnet (L3):** Host A sends frame to router's MAC. Router strips frame, creates new frame with router's MAC as source, next-hop's MAC as destination. IP addresses stay unchanged. Process: Receives frame → extracts IP packet → looks up dest IP in routing table → encapsulates in new frame for next hop.

Self-Configuring Networks (MAC Learning):
- **Initial State:** Switch starts with empty MAC address table
- **Learning Process:** For every received frame on port X with source MAC = M: records "MAC M is reachable via port X" in table
- **Forwarding Decisions:**
  – **Known destination:** Looks up dest MAC in table → forwards frame out specific port
  – **Unknown destination:** Floods frame to ALL ports except incoming port
- **Table Management:** Entries age out (default 5 minutes) to handle device moves. Source learning happens on every frame, destination lookup on every frame.

Spanning Tree Protocol (STP):
- **Problem:** Redundant switch connections create broadcast loops
- **Solution:** Elects root bridge (lowest Bridge ID), calculates shortest path tree, blocks redundant ports
- **States:** Blocking (drops frames) → Listening → Learning → Forwarding
- Reconverges automatically when topology changes (link/switch failure)

Address Resolution & Ethernet Fundamentals:
- **Address Resolution Protocol (ARP):**
  – **Purpose:** Resolves IP address to MAC address (like DNS resolves domain to IP)
  – **Process:** Host broadcasts "Who has IP X?" → Target responds "IP X is at MAC Y" → Sender caches mapping
  – **ARP Table:**

IP Address	MAC Address	Timeout
192.168.1.1	aa:bb:cc:dd:ee:ff	300s
192.168.1.5	11:22:33:44:55:66	180s

- **Network Address Translation (NAT) Gateway:**
  – **Purpose:** Allows multiple private IP devices to share a single public IP address for Internet access
  – **Operation:** Translates private IP addresses (192.168.x.x, 10.x.x.x) to public IP when packets leave the network, reverse translation on return
  – **Translation Table:** Maps (private IP, private port) ↔ (public IP, public port) to track active connections
  – **Benefits:** IP address conservation, adds security layer (hides internal network), enables home/office Internet sharing

Internet Routing Architecture:
Autonomous System (AS): Administrative domain containing routers under single policy control (ISP, university, company). Each AS has unique number (ASN) and can choose internal routing protocols.
Two-Level Routing Hierarchy:
- **Intra-AS:** Fast convergence within AS using link-state or distance-vector.
- **Inter-AS:** Policy-based routing between ASes using BGP. Scalability through hierarchy.

Border Gateway Protocol (BGP):
- **Like Bellman-Ford:** Exchanges path vectors with neighbors, applies best path selection
- **Operation:** AS announces "I can reach prefix X via path [AS1, AS2, AS3]"
- **Enables:** Policy control (traffic engineering, business relationships), loop prevention (AS-path), Internet-scale routing

C Language Essentials:
- **const:** Read-only variable
- **size_t sizeof(type):** Returns size in bytes

```c
typedef struct { int x, y; } Point;
const char* str = "Hello"; // read-only

int arr[10] = {1,2,3}; // rest are 0
void func(int tab[]) { /* tab is pointer */ }
printf("Array size: %lu\n", sizeof(arr));

// Control structures
switch(i) {
    case 1: printf("One"); break;
    default: printf("Other");
}
for (int i = 0; i < 10; i++) { /* loop */ }
```

Memory Management Functions:
- **void* malloc(size_t size):** Allocates uninitialized heap memory
- **void* calloc(size_t count, size_t size):** Allocates zero-initialized memory
- **void* realloc(void* ptr, size_t size):** Resizes existing allocation
- **void free(void* ptr):** Deallocates memory (always call!)

```c
int *arr = malloc(5 * sizeof(int));
if (!arr) { perror("malloc"); exit(1); }

int *zeros = calloc(5, sizeof(int)); // all zeros
arr = realloc(arr, 10 * sizeof(int)); // may move
free(arr); free(zeros);
```

Common String Functions:
- **size_t strlen(const char* s):** String length (excludes null terminator)
- **char* strcpy(char* dest, const char* src):** Copy strings
- **char* strncpy(char* dest, const char* src, size_t n):** Copy strings (safer)
- **int strcmp(const char* s1, const char* s2):** Compare strings (returns < 0, 0, > 0)
- **int strncmp(const char* s1, const char* s2, size_t n):** Compare strings
- **char* strcat(char* dest, const char* src):** Concatenate strings
- **char* strncat(char* dest, const char* src, size_t n):** Concatenate strings
- **char* strchr(const char* s, int c):** Search for character
- **char* strstr(const char* haystack, const char* needle):** Search for substring

```c
char dest[20], src[] = "Hello";
strcpy(dest, src);           // Copy string
if (strcmp(dest, "Hello") == 0) { /* equal */ }
char *p = strchr(dest, 'l'); // Find 'l' in dest
memset(dest, 0, sizeof(dest)); // Zero out
```

Structs & Flexible Arrays:
- **Flexible array members:** For variable-length data

```c
typedef struct {
    int data;
    struct Node *next;
} Node;
// Flexible array member (C99)
typedef struct {
    size_t len;
    char data[];  // flexible array
} string_t;
string_t *s = malloc(sizeof(*s) + len + 1);
s->len = len;
memcpy(s->data, input, len + 1);
```

Function Pointers:

```c
int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }
int (*op)(int, int) = add;     // function pointer
int result = op(5, 3);         // calls add(5,3)
int (*ops[])(int,int) = {add, sub}; // array
result = ops[0](10, 5);        // calls add(10,5)
```

Parallelism & Concurrency:
- **Parallelism:** Doing two different tasks at the same time
- **Concurrency:** Doing a task while waiting on another (overlapping execution)
- **Benefits:** Increases throughput, reduces latency
- **Between processes:** Use `pid_t fork(void)`
- **Within process:** `int pthread_create(pthread_t* thread, const pthread_attr_t* attr, void* (*start_routine)(void*), void* arg)`

Thread Issues & Solutions:
- **Thread interleaving:** Uncontrolled scheduling → non-deterministic behavior
- **Race condition:** Timing/order affects program correctness
- **Data race:** One thread reads while another writes (no sync)
- **Solution:** Locks - only holder can modify, others wait

Lock Types:
- **Disable Interrupts:** Turn off interrupts during critical section
- **Test-and-Set Spinlock:** Busy wait until lock available
- **Mutex:** Sleepable locks with OS involvement

```c
// Disable Interrupts (single processor)
void lock(lock_t &l) { disable_interrupts(); }
void unlock(lock_t &l) { enable_interrupts(); }

// Test-and-Set Spinlock
int test_and_set(int *ptr, int val) {
    int old = *ptr; *ptr = val; return old;
}
bool lock1 = false;
void lock(bool *l) { while (test_and_set(l, true)); }
void unlock(bool *l) { *l = false; }

// Mutex (sleepable)
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
void critical_section(int *counter) {
    pthread_mutex_lock(&m);
    *counter += 1;
    pthread_mutex_unlock(&m);
}
```

Process System Calls:
- **pid_t fork(void):** Clone process, returns child PID or 0
- **void exit(int status):** Terminate process cleanly
- **pid_t wait(int* status):** Wait for child termination

```c
pid_t pid = fork();
if (pid == 0) {
    // Child: replace with new program
    execl("/bin/ls", "ls", "-l", NULL);
    exit(1); // only if exec fails
} else if (pid > 0) {
    // Parent: wait for child
    int status;
    wait(&status);
} else { perror("fork"); }
```

Thread System Calls:
- **int pthread_create(pthread_t* thread, const pthread_attr_t* attr, void* (*start_routine)(void*), void* arg):** Create new thread
- **int pthread_join(pthread_t thread, void** retval):** Wait for thread completion

```c
pthread_t thread_id;
void* action(void* arg) {
    printf("Thread running\n");
    return NULL;
}
pthread_create(&thread_id, NULL, action, NULL);
pthread_join(thread_id, NULL); // wait for completion
```

File I/O System Calls:
- **int open(const char* pathname, int flags, mode_t mode):** Open file, return FD
- **ssize_t read(int fd, void* buf, size_t count):** Read bytes from FD
- **ssize_t write(int fd, const void* buf, size_t count):** Write bytes to FD
- **off_t lseek(int fd, off_t offset, int whence):** Move file offset
- **int close(int fd):** Close file descriptor

File Streams (stdio.h):
- **FILE* fopen(const char* filename, const char* mode):** Open file stream
- **size_t fread(void* ptr, size_t size, size_t nmemb, FILE* stream):** Read data blocks
- **size_t fwrite(const void* ptr, size_t size, size_t nmemb, FILE* stream):** Write data blocks
- **int fflush(FILE* stream):** Force buffer write
- **int fclose(FILE* stream):** Close file stream

```c
int fd = open("file.txt", O_RDWR | O_CREAT, 0644);
if (fd == -1) { perror("open"); exit(1); }
char buf[100];
ssize_t n = read(fd, buf, sizeof(buf)-1);
if (n > 0) buf[n] = '\0';
write(fd, "Hello", 5);
lseek(fd, 0, SEEK_SET); // rewind to start
close(fd);
```

Socket System Calls:
- **int socket(int domain, int type, int protocol):** Create communication endpoint
- **int bind(int sockfd, const struct sockaddr* addr, socklen_t addrlen):** Assign address to socket
- **ssize_t sendto(int sockfd, const void* buf, size_t len, int flags, const struct sockaddr* dest_addr, socklen_t addrlen):** Send data
- **ssize_t recvfrom(int sockfd, void* buf, size_t len, int flags, struct sockaddr* src_addr, socklen_t* addrlen):** Receive data

```c
int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons(8080);
bind(sockfd, (struct sockaddr*)&addr, sizeof(addr));
```

General Exam Assumptions
- If a question asks to identify all subnets, include trivial subnets (those with no end-systems).
- If a question wants to ignore trivial subnets, it will say so explicitly.
- Reasonable alternative assumptions will not be penalized if clearly stated.
- SEQ number of the first segment carrying data is 1 (but other values are acceptable if consistently used).

Networking Assumptions
- Each router interface has an IP and MAC address.
- L2 switch interfaces have neither IP nor MAC addresses (even if unrealistic).
- Each subnet has a broadcast IP address (highest address in the range).
- Subnets have no reserved network address (e.g., 8.0.0.0 is assignable).
- Each end-system knows the IP address of its default gateway.
- Recursive or iterative DNS can be assumed; just state your assumption.
- If the ARP cache is empty, ARP requests must be made for:
- The default gateway's MAC (for routing outside the subnet).
- The local DNS server's MAC, if it is on the same subnet.
- Do not assume NAT is in use unless stated explicitly.
- Assume IPv4 address space is sufficient.

TCP Assumptions
- Use a single persistent TCP connection for multiple data exchanges.
- Second segment from A to B (3rd of handshake) carries MSS bytes of data and has SEQ = 1.
- First response from B to A carries both ACK and data, also with SEQ = 1.
- Transmission delays of ACKs are negligible.
- Timeout behavior:
- Once timeout occurs: congestion window reset to 1 MSS, and retransmission of oldest unacknowledged segment.
- Before timeout: sender may continue sending as long as congestion window allows.
- Congestion avoidance: use $\text{MSS}^2/2$ per ACK formula (not strictly 1 MSS per RTT).
- Assume Tahoe unless stated otherwise.
- Out-of-order segments are buffered, not dropped.

Memory Virtualization Assumptions
- A page table entry (PTE) contains the physical frame number (PFN), not a full physical address.
- Do not assume a minimum PTE size unless stated.
- Total physical memory = $2^{\#PFNs} \times$ frame size.

File Access Assumptions
- A filesystem cache (block cache) exists unless told otherwise.
- Read operations may be avoided due to caching; writes always require block access.
- A write() syscall:
- Updates the FS cache.
- Triggers disk writes for data and inode.
- A close() syscall does not trigger block accesses (you may assume it does, but it's unnecessary).

Scheduling Assumptions
- If a thread blocks or terminates mid-tick (e.g., after 0.99 ticks), nothing happens for the rest of the tick.
- The OS scheduler runs only at integer tick values (0, 1, 2, ...).

Timers in TCP
- The timer for a segment may be assumed to start either before or after transmission — both are valid.

All link-layer switches have just been rebooted, and all end-system caches/ARP tables are initially empty. All routers have populated their forwarding tables according to the intra-domain routing protocol.

The user of end-system A_1 visits web page `www.epfl.ch`, which contains no embedded objects (e.g., no images). Immediately after A_1's user views `www.epfl.ch`, the user of end-system B_1 visits the same web page.

State all the packets that are **received, forwarded, or transmitted by router R_3 as a result of B_1's actions and until B_1's user can view the web page**. For example, if router R_3 receives and forwards an IP packet, you should state that packet twice: once to state that R_3 received it, and once to state that R_3 forwarded it.

#	Source MAC	Dest MAC	Source IP	Dst IP	Transp. prot.	Src Port	Dst Port	Application & Purpose
1	k	broadcast	-	-	-	-	-	ARP request for g's MAC
2	g	k	-	-	-	-	-	ARP reply
3	k	g	b_1	d	UDP	2000	53	DNS request for w's IP
4	f	broadcast	-	-	-	-	-	ARP request for d's MAC
5	d	f	-	-	-	-	-	ARP reply
6	f	d	b_1	d	UDP	2000	53	DNS request for w's IP
7	d	f	d	b_1	UDP	53	2000	DNS reply
8	g	k	d	b_1	UDP	53	2000	DNS reply
9	k	g	b_1	w	TCP	4000	80	TCP SYN
10	h	v	b_1	w	TCP	4000	80	TCP SYN
11	v	h	w	b_1	TCP	80	4000	TCP SYN ACK
12	g	k	w	b_1	TCP	80	4000	TCP SYN ACK
13	k	g	b_1	w	TCP	4000	80	HTTP GET index
14	h	v	b_1	w	TCP	4000	80	HTTP GET index
15	v	h	w	b_1	TCP	80	4000	HTTP OK
16	g	k	w	b_1	TCP	80	4000	HTTP OK

Bellman-Ford applied on router Z:

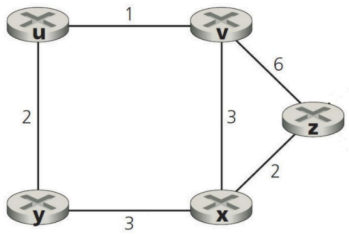


Figure 5: Network topology.

Initially (at step 0), node z has the following view of the network:

		To				
		u	v	x	y	z
From	v	∞	∞	∞	∞	∞
	x	∞	∞	∞	∞	∞
	z	∞	6	2	∞	0

At step 1:

		To				
		u	v	x	y	z
From	v	1	0	3	∞	6
	x	∞	3	0	3	2
	z	7	5	2	5	0

At step 2:

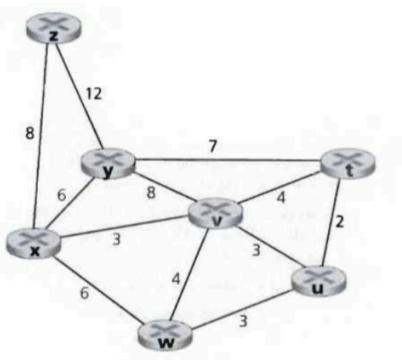
		To				
		u	v	x	y	z
From	v	1	0	3	3	5
	x	4	3	0	3	2
	z	6	5	2	5	0

At step 3:

		To				
		u	v	x	y	z
From	v	1	0	3	3	5
	x	4	3	0	3	2
	z	6	5	2	5	0

We see that from step 2 to step 3 the cost tables did not change, indicating that the algorithm has converged.

Dijktra for least cost path from x, v and t to the rest.



The least cost path from routers x, v, and t to all the other routers is displayed in the next table along with the execution steps of the link-state algorithm.

For each router i, C(i) stands for cost to i, and p(i) stands for predecessor to i.

1. Least-cost path from x to all network nodes.

step	nodes visited	C(t),p(t)	C(u),p(u)	C(v),p(v)	C(w),p(w)	C(y),p(y)	C(z),p(z)
0	x	∞	∞	3,x	6,x	6,x	8,x
1	x,v	7,v	6,v	3,x	6,x	6,x	8,x
2	x,v,u	7,v	6,v	3,x	6,x	6,x	8,x
3	x,v,u,w	7,v	6,v	3,x	6,x	6,x	8,x
4	x,v,u,w,y	7,v	6,v	3,x	6,x	6,x	8,x
5	x,v,u,w,y,t	7,v	6,v	3,x	6,x	6,x	8,x
6	x,v,u,w,y,t,z	7,v	6,v	3,x	6,x	6,x	8,x

2. Least-cost path from v to all network nodes.

step	nodes visited	C(t),p(t)	C(u),p(u)	C(w),p(w)	C(x),p(x)	C(y),p(y)	C(z),p(z)
0	v	4,v	3,v	4,v	3,v	8,v	∞
1	v,x	4,v	3,v	4,v	3,v	8,v	11,x
2	v,x,u	4,v	3,v	4,v	3,v	8,v	11,x
3	v,x,u,t	4,v	3,v	4,v	3,v	8,v	11,x
4	v,x,u,t,w	4,v	3,v	4,v	3,v	8,v	11,x
5	v,x,u,t,w,y	4,v	3,v	4,v	3,v	8,v	11,x
6	v,x,u,t,w,y,z	4,v	3,v	4,v	3,v	8,v	11,x

3. Least-cost path from t to all network nodes.

step	nodes visited	C(u),p(u)	C(v),p(v)	C(w),p(w)	C(x),p(x)	C(y),p(y)	C(z),p(z)
0	t	2,t	4,t	∞	∞	7,t	∞
1	t,u	2,t	4,t	5,u	∞	7,t	∞
2	t,u,v	2,t	4,t	5,u	7,v	7,t	∞
3	t,u,v,w	2,t	4,t	5,u	7,v	7,t	∞
4	t,u,v,w,x	2,t	4,t	5,u	7,v	7,t	15,x
5	t,u,v,w,x,y	2,t	4,t	5,u	7,v	7,t	15,x
6	t,u,v,w,x,y,z	2,t	4,t	5,u	7,v	7,t	15,x

Tahoe. ACKs are cumulative