1. Operating Systems & Networks

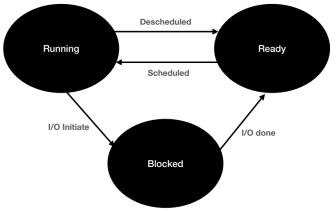
1.1 Introduction

1.1.1 Operating System

- ➔ Software that acts as an **intermediary between computer hardware and user apps**
- ➔ Manages computer's hardware resources (CPU, memory) and I/O devices (printers)
- ➔ Enables user programs to execute without worrying about hardware specifications.
- ➔ Three pillars of OS :
  - **Virtualisation** : Providing illusion of infinite memory and compute (CPU)
  - **Concurrency** : Running multiple processes at a time
  - **Persistence** : Managing storage on disk (hardware) using file systems (software)

1.1.2 Process

- ➔ A **program** is nothing but code; and executing programs are called **processes**.
- ➔ A process constitutes of a unique identifier (Process ID), memory image (static (code and data) and dynamic (stack and heap)), CPU context (registers, instruction pointer and program counter) and file descriptors (pointers to open files and devices for memory I/O).
- ➔ A process can be in one of the following **three states** :
  - **Blocked** : Waiting for some I/O call (not ready to run)
  - **Ready** : Waiting to be executed (ready to run)
  - **Running** : Executing on processor (running)



- ➔ Steps to **create a process** :
  - Load program into memory (lazy load from disk)
  - Runtime stack allocation (used for local variables, function parameters and return arguments)
  - Creation of program heap (used for dynamically allocated data)
  - Basic file setup (for STDIN, OUT, ERR)
  - Initialise CPU registers (setting PC to the first instruction)
  - Start the program

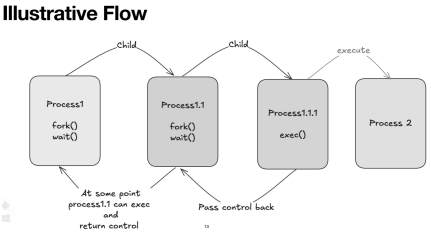
Time	Process 0	Process 1	What's happening
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process 0 initiates I/O
4	Blocked	Running	Process 0 is blocked, 1 runs
5	Blocked	Running	
6	Blocked	Running	I/O of process 0 is done
7	Ready	Running	Process 1 is done
8	Running	-	Process 0 is done

- ➔ OS uses process list to store metadata about processes, called **Process Control Block (PCB)**. It includes Process ID (identifier), process state and address space of the process (the register values).
- ➔ **init** process is the ancestor of all processes.

1.1.3 System Calls

- ➔ OS-provided function that allows user programs to interact with hardware.
- ➔ Two modes of execution : **User** and **Kernel** (higher privilege such as I/O)
  - Uses **Limited Direct Execution (LDE)** as a low level mechanism to separate user space from kernel space.
  - Kernel performs system calls on behalf of the user process. Uses a separate kernel stack and **Interrupt Descriptor Table** (IDT, aka Trap Table) to keep logs of different kernel functions addresses.
- ➔ **TRAP Instruction** : Special instruction to switch from user to kernel mode.
  - CPU to higher privilege level, save context (old PC, registers) on Kernel Stack, look up in IDT and jump to trap handler function in OS code.
  - Once done, the OS calls a special return-from-trap instruction which returns into the calling program, and back to user mode.
  - Different from interrupt (which are signals sent to CPU due to unexpected events, from either software or hardware), as it is a purely software generated interrupt caused by system calls or exceptions.

- ➔ **POSIX (Portable Operating Systems Interface)** : Standard set of system calls that an OS must implement to ensure portability. Programming languages abstract systems calls. Eg: printf() in C internally invokes write system call.
- ➔ Some important system calls :
  - **fork()** : Creates a new child process (with new PID), image copy of parent with independent memory. The new process is added to the list of processes and scheduled; and both start execution just after fork (with different return values).
  - **exec()** : Used to load a different executable to the memory of a child process and make it run a different program from the parent. We can also pass an executable in some variations.
  - **wait()** : Puts the parent in block state until the child terminates (options like waitpid() also exist). It also collects exit status of the terminated child process, providing some visibility to the parent process. wait allows the OS to reclaim the resources of the child and prevent **zombie processes**. init process adopts / reaps orphans.
  - **exit()** : Terminates a process



1.2 Process Virtualisation

- ➔ Multiple processes (even more than the number of processors) need to be executed with each of them having the illusion of exclusive access to resources.
- ➔ Requires a switching mechanism (hardware) along with some policies (software)

1.2.1 Switching

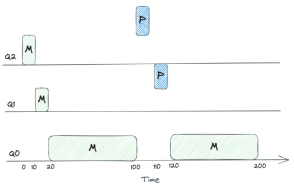
- ➔ **Context Switch** : A low-level piece of assembly code that saves a register value from executing process registers to kernel stack and restore values for the next process. Essentially return-from-trap will go to the new process.
- ➔ For switching (kicking the currently executing process), we have two approaches :
  - **Cooperative / Non-Preemptive** : OS trusts the processes to behave reasonably (Give control back - yield() call).
    - In case of a misbehaving process (eg: trying to do something they shouldn't), trap instruction transfers to the OS which terminates the process.
    - Problem : Reboot system, in cases of infinite execution of some process
  - **Non-Cooperative / Preemptive** : OS takes control with the help of interrupts.
    - Every X milliseconds, raise an interrupt -> halt the process -> invoke interrupt handler -> OS regains control (continue with the process or switch)

1.2.2 Scheduling

- ➔ Coming to scheduling policies, let's first define some **metrics** :
  - **Performance** : $T_{turnaround} = T_{completion} - T_{arrival}$ *and* $T_{response} = T_{firstrun} - T_{arrival}$
  - **Fairness** : Jains fairness index (fairness in scheduling)
- ➔ For I/O jobs, the scheduler simply moves the job to blocked state. Once I/O is done, an interrupt is raised and the OS moves the process back to ready state.
- ➔ Some assumptions, which we'll start with : All jobs arrive at the same time, Each job runs for the same amount of time, No I/O times for any job and Known run times.
- ➔ Various scheduling policies are as follows :
  - **First Come First Serve Policy** : Schedule the job that came first. As soon as it is done, schedule the job that comes next. Good when all above assumptions are held. But, if processes take different times, might lead to convoy effect (longest process hoarding and not letting smaller processes finish fast).
  - **Shortest Job First (SJF) Policy** : Assumes that all jobs come at the same time, and prioritises ones which will finish fast.
  - **Shortest Time to Completion First (STCF)** : Advanced SJF that does not assume jobs come at same time and switches whenever a faster completing job enters.
  - **Round Robin Scheduling** : Tries improving on the response time by running jobs for time slices (Run job for a time slice → switch to next job → while true) instead of individual job to completion. Time slice duration is also important, too

small wastes a lot of time on context switch; and too large leads to CPU hogging

- **Multi Level Feedback Queues (MLFQ)** : Assigns processes to different priority queues. If priority(A) > priority(B), A runs; if equal, they share CPU (round robin). Priorities are adjusted based on behavior, promoting interactive jobs and demoting long-running CPU-heavy ones. This reduces turnaround and response times but may lead to starvation (solved via periodic priority boosts) and gaming (processes faking I/O to stay high priority).



  - Determining periodic boost interval is hard (voo-doo constant). Too small might not give proper share to interactive jobs; and too big might starve long running jobs.

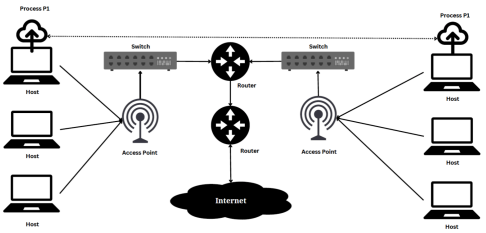
1.3 Networking

➔ Client sends a request to the server, to which the server provides a response.

➔ **Protocol** : Agreement between communicating parties on how to communicate

➔ Software components in the OS that support (using system calls) network calls are called **protocol stack**. Few hardware components :

- **Host** : Any device that send or receive traffic (can be client or server)
- **IP Address** : 32 bits, hierarchically assigned address used by host to send data
- **Repeater** : Allows regeneration of signals for long distance communication
- **Hub** : Multi-port repeaters (Key issue: everyone receives everyone's data)
- **Bridge** : Sits b/w two hubs and is aware about hosts on either side (for routing)
- **Switch** : Multi-port bridge. Devices connected to a switch are part of 1 network
- **Router** : Facilitate communication between networks. Act as traffic control point facilitating security, filtering and redirection.
  - All communications to / from out of the network goes through the router.



➔ Multiple types of networks exist :

- **Personal Area Network (PAN)** : Very short range. eg bluetooth (master-slave)
- **Local Area Network (LAN)** : Private network operating within / nearby a single building. Wireless LANs : 11 Mbps to 7 Gbps, wired LANs 100 Mbps to 40 Gbps. One large Physical LAN can be divided into smaller logical LANs (Virtual LANs)
- **Metropolitan Area Network (MAN)** : City wide networks
- **Wide Area Network (WAN)** : Span large geographical areas (country, continent, etc). Higher latency and lower transmission speeds. Internet is a large WAN (Dedicated WANs for large organisations also exist, costly tho)

➔ Networks are often organised as a stack of layers for abstraction. The set of layers along with protocols forms the **Network Architecture**.

- Layer n of one machine communicates with layer n of another using a protocol.
- Between each pair of adjacent layer there is an interface, which defines the primitive operations and services the lower layer makes available

1.3.1 The OSI Model

➔ Open System Interconnection (OSI), a conceptual framework used to understand how network communication works through different layers.

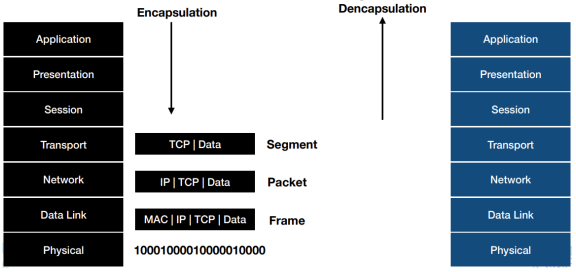
➔ Facilitate interoperability between different technologies

➔ Comprised on 7 layers :

- **Physical Layer (L1)** : Transmission of raw bits through physical medium. Comprises ethernet cables, optical fiber, coaxial cable, WiFi, hub, repeater, etc.
- **Data Link Layer (L2)** : Interacts with the physical medium using MAC address (12 hex digits). Ensures hop-to-hop communication by creating reliable links (error-correcting) between two directly connected (physically adjacent) nodes. Comprises NIC, WiFi access cards and switches (move data).
- **Network Layer (L3)** : Manages end-to-end communication (routing through different routes in a large network) using IP addressing (4 octets in IPv4).

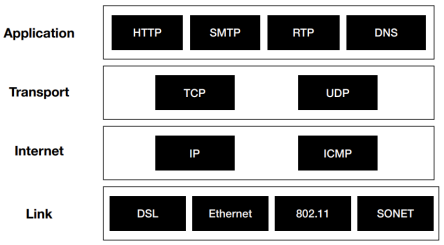
Performs logical addressing (IP), path selection and packet forwarding. Comprises routers, hosts, L3 switches, etc.

- **Transport Layer (L4)** : Service-to-service communication, ensuring that the right process receives the (reliable, sequential and free from others) data. Manages flow control and error correction. Uses ports (16-bit : 0-65535, privileged: 0-1023, registered: 1024 - 49151) to send / receive data, unique to each process. Comprises TCP and UDP.
- **Session Layer (L5)** : Manages (establish, maintain and terminate) connection between different devices.
- **Presentation Layer (L6)** : Data encryption & compression to ensure that data is in format that sender / receiver can understand.
- **Application Layer (L7)** : Provides support for end applications to format and manage data. In turn they make use of transport layer protocols. Comprises HTTP, DNS, SMTP, etc.



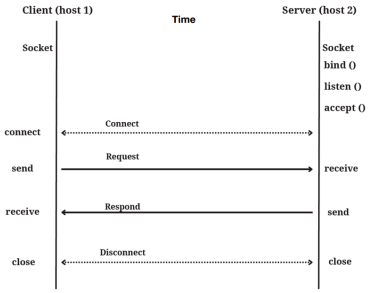
➔ We also have another model **Internet Model (TCP/IP Model)**, which, unlike OSI, is an actual practically used model with 4 layers :

- Application Layer : Corresponds to application, presentation and session
- Transport Layer : Transport layer of OSI
- Internet Layer : Network layer of OSI
- Network Layer : Physical and data link layers of OSI



1.3.2 Transport Layer

➔ **Socket API** : Simple abstraction, allowing applications to attach to the network at different ports. Socket establishing calls (like connect, accept, etc) are blocking calls, ie, raise trap instruction.



➔ Application process is identified by tuple (IP, Protocol, Port)

➔ We have many types of links :

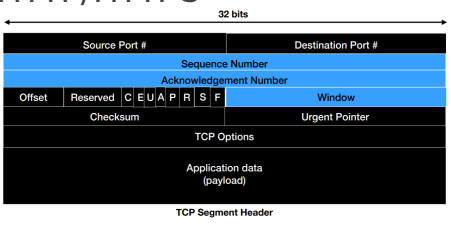
- **Full-duplex** : Bidirectional (both way at the same time)
- **Half-duplex** : Bidirectional (only one-way at a time)
- **Simplex** : Unidirectional

➔ Two step process :

- **Multiplexing** (sender) : Handle data from multiple sockets, add transport header
- **Demultiplexing** (receiver) : Use header info to deliver received segments to correct socket

➔ Two types of protocols :

- **User Datagram Protocol (UDP)** : Opposite of TCP (like not connection oriented, no flow control, retransmission, etc). Use case : VoIP, DNS queries, streaming.
  - UDP socket identified using destination IP and port. UDP segments with the same destination port are redirected to the same socket.
  - UDP segment header (32bit wide) comprises of source and destination port (16bit each) in the first row, length (in bytes, including header) and checksum (of UDP, header, payload and pseudo header from IP layer) in second row, followed by the application data (payload).
- **Transmission Control Protocol (TCP)** : Connection oriented (establishing connection before transmission), congestion control, reliable (order maintained, error detection using acknowledgement and retransmission), higher overhead (~20 bytes > ~8 bytes), and flow control (adjust transmission rate or message limit based on network). Use case : mail, file transfer, HTTP/HTTPS
  - TCP sockets are identified using source IP & port and destination IP port. A server can support multiple TCP sockets, each communicating with a different client.
  - A TCP packet comprises of sequence number



TCP Segment Header

(no. of sent bytes), acknowledgement number (next expected byte seq number), window (no. of bytes the receiver can accept), A (acknowledgement bit), R & S & F (connection management), C & E (congestion notification) and offset (length of the TCP header).