Unit 1

Definition of an Operating System

- A program that acts as an intermediary between the user and computer hardware.
- Provides a user-friendly environment for developing and executing programs.
- Eliminates the need for hardware knowledge for programming.

Goals of an Operating System

- Execute user programs and solve problems efficiently.
- Make the computer system easy to use.
- Efficiently manage resources like:
    - Memory
    - Processor(s)
    - I/O Devices

COMPUTER SYSTEM ARCHITECTURE

Components

1. **Hardware** → Provides basic computing resources (CPU, memory, I/O devices).
2. **Operating System** → Controls and coordinates hardware among applications and users.
3. **Application Programs** → Utilize system resources for user tasks (word processors, browsers, etc.).
4. **Users** → People, machines, or other computers.

What Does OS Do

- **User Perspective →** Convenience, ease of use, and good performance.
- **System Perspective →** Maximizes resource utilization and manages multiple users.
- **For Shared Systems (Mainframes, Servers) →**
    - Ensures fair resource allocation among users.
    - Maximizes CPU, memory, and I/O efficiency.
- **For Workstations (Personal Computers, Laptops) →**
    - Balances usability and resource management.
- **For Handheld & Embedded Systems →**
    - Optimized for usability and battery life.
    - Some systems have no user interface (e.g., car engine controllers).

ROLES OF OS

- **Resource Allocator** – Manages resources and resolves conflicts efficiently.
- **Control Program** – Controls program execution to prevent errors.
- OS brings together functions of controlling and allocating resources.
- The **Kernel** is the core program that always runs.
- Other components include:
    - **System Programs** (ships with OS)
    - **Application Programs** (installed by users)

COMPUTER SYSTEM ORGANISATION

- **Components →**
    - One or more CPUs
    - Device controllers connected via a common bus to shared memory

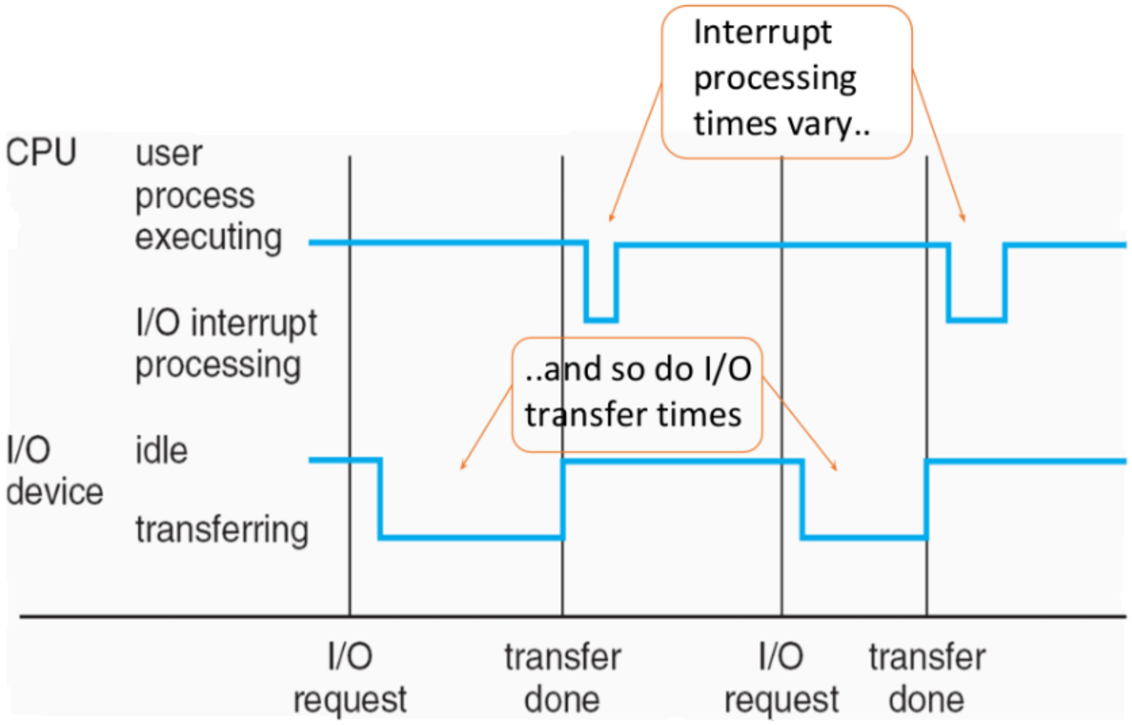
- Memory controller to manage access

- **Operation →**
  - CPU and device controllers execute concurrently.
  - Memory controller synchronizes access.

Bootstrap Program

- First program that runs when a system boots.
- Stored in **ROM** or **EEPROM** (firmware).
- Initializes CPU, memory, and device controllers.
- Loads and starts the **Operating System Kernel**.
- The first process initiated is **init**, which waits for system events.

INTERRUPTS IN OS



- **Interrupts →** Transfer control to an **Interrupt Service Routine (ISR)**.
- **Interrupt Vector →** Stores addresses of service routines.
- **Trap/Exception →** A software-generated interrupt due to an error or user request.

Interrupt Handling :

1. OS saves CPU state (registers, program counter).
2. Determines interrupt type →
    - **Polling** (checks devices in order).
    - **Vectored Interrupts** (device sends its interrupt vector).
3. Executes appropriate action based on interrupt type.

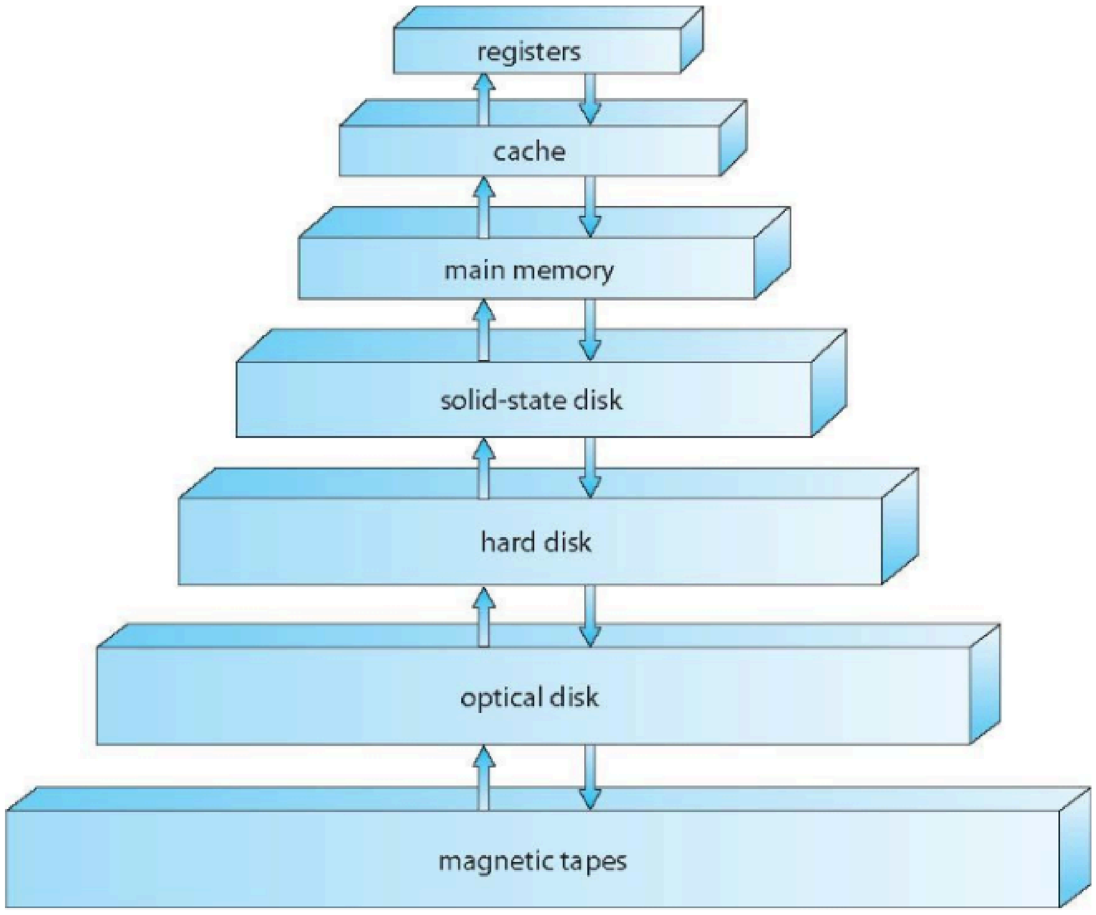
MEMORY STRUCTURE

- **Main Memory →**
    - Large, fast storage directly accessed by the CPU.
    - Typically **volatile** (RAM).
- **Other Memory Types →**
    - **ROM, EEPROM –** Used for firmware (e.g., system boot code).
    - **Smartphones** use EEPROM for factory-installed software.

PROGRAM EXECUTION MODEL

- Based on the ***Von Neumann Model →***
    1. ***Fetch*** instruction from memory.
    2. ***Decode*** the instruction.
    3. ***Execute*** the instruction.
    4. Repeat until the program ends.

STORAGE STRUCTURE



Types of Storage

- **Secondary Storage** – Non-volatile, large capacity.
    - **Hard Disks** – Magnetic storage with tracks and sectors.
    - **Solid-State Disks (SSD)** – Faster, non-volatile storage.
    - **Flash Memory** – Used in cameras, PDAs.

Storage Hierarchy

1. **Registers** (Fastest, expensive, smallest).
2. **Cache** (Small, temporary storage).

3. **Main Memory (RAM)**.
4. **Secondary Storage (HDD, SSD)**.
5. **Removable Storage (USB, CD/DVD)**.
6. **Cloud Storage (Slowest, cheapest, largest).**

Caching

- Frequently used data is stored in a **faster cache**.
- Improves access speed by reducing latency.
- Cache **management** is crucial (size, replacement policy).

I/O SRUCTURE

- **I/O Devices:** Storage is a form of I/O.
- **Device Controllers:** Manage specific types of devices.
- **Device Drivers:** Act as an interface between OS and hardware.

I/O Operations

- **Synchronous I/O:**
    - CPU waits for I/O completion.
    - Inefficient for large operations.
- **Asynchronous I/O:**
    - CPU continues execution while I/O happens.
    - Uses **interrupts** to signal completion.
- **Device-Status Table:**
    - OS maintains a table to track device type, address, and state.

DIRECT MEMORY ACCESS (DMA)

- Used for **high-speed** I/O devices.
- Transfers **large blocks of data** between memory and devices without CPU involvement.
- **Reduces interrupts**, improving efficiency.
- Only **one interrupt per block** instead of one per byte.

SINGLE PROCESSOR SYSTEMS

- Most systems use a **single general-purpose processor**.
- Other **special-purpose processors** are also present:
  - Examples: Disk controller, keyboard processor, graphics controller.
- Special-purpose processors:
  - Run a **limited set of instructions**.
  - Built into the hardware and managed by the OS.
  - Example: **Disk controller microprocessor** manages its own queue and scheduling, reducing CPU workload.

MULTIPROCESSOR SYSTEMS

- Also called **parallel systems** or **tightly-coupled systems**.
- Advantages →
  - **Increased throughput –** More tasks completed simultaneously.
  - **Economy of scale –** Shared components reduce cost.
  - **Increased reliability –** Supports graceful degradation and fault tolerance.

Types of Multiprocessors

1. **Asymmetric Multiprocessing (AMP) –** Each processor is assigned a specific task.
2. **Symmetric Multiprocessing (SMP) –** All processors share tasks equally.

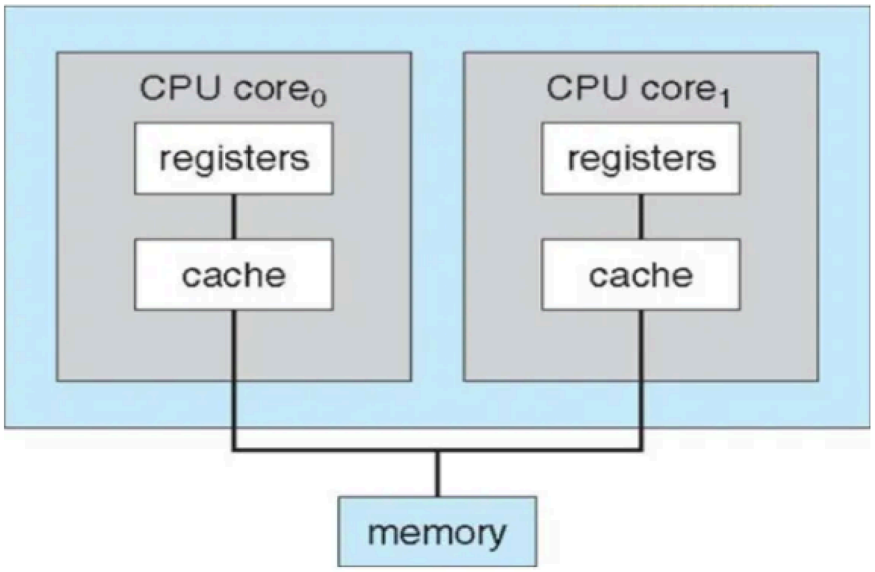
Symmetric Multiprocessing (SMP) Architecture

- No **boss-worker** relationship; all processors are **peers**.
- Each processor has its **own registers and cache**.
- All processors share **physical memory**.
- A **single chip** contains **multiple cores** (processors).
- More **efficient** than multiple separate chips due to **faster on-chip communication**.
- Consumes **less power** than multiple single-core chips.
- **Command to check cores and cache details:**

```
$ cat /proc/cpuinfo | more
```

Example:

- A **dual-core processor** has **two cores on the same chip**.



A dual-core design with two cores placed on the same chip

Blade Servers

- A **recent development** with multiple processor boards, I/O boards, and networking boards **in one chassis**.
- Each **blade-processor board boots independently** and runs its own OS.
- Some blade servers are **multiprocessor systems**, making them similar to clustered systems.

Clustered Systems

- Similar to multiprocessor systems but consist of **multiple computers working together**.
- Uses **shared storage** (Storage Area Network - SAN).
- Provides **high availability** (survives hardware failures).

Types of Clustering

1. **Asymmetric Clustering** – One machine is in **hot-standby mode** (backup).
2. **Symmetric Clustering** – All nodes run applications and monitor each other.
3. **High-Performance Computing (HPC) Clusters** – Used for scientific and technical computing.
4. **Clusters with Distributed Lock Manager (DLM)** – Prevents conflicts in shared storage access.

Multiprogramming (Batch Systems)

- Used to **improve efficiency** by keeping CPU and I/O devices busy.
- Organizes jobs in memory, and **CPU switches between jobs** to maximize usage.
- Uses **job scheduling** to select and execute jobs.

Multitasking (Time-Sharing Systems)

- **Logical extension of multiprogramming**.

- CPU **switches between jobs rapidly**, allowing user interaction.
- **Response time should be <1 second**.
- **CPU scheduling** is needed when multiple jobs are ready to run.
- **Swapping** moves processes in and out of memory when necessary.
- **Virtual memory** enables execution of processes **not completely in memory**.

OPERATIONS OF OS

- **Interrupt-driven** (hardware & software).
- **Hardware Interrupts:** Triggered by devices (e.g., keyboard input, I/O completion).
- **Software Interrupts (Traps/Exceptions):**
  - Division by zero.
  - System calls (requests to OS for services).
  - Infinite loops
  - Unauthorized access attempts.

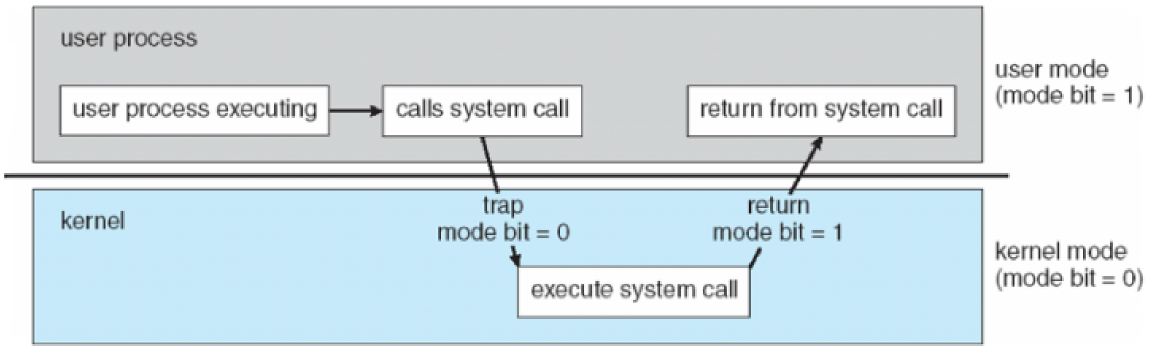
Dual-Mode & Multi-Mode Operations

- **Dual-mode operation** protects OS and system components.
- **Modes:**
  - **User Mode:** Runs application programs.
  - **Kernel Mode:** Executes system-level operations.
- **Mode Bit:** Hardware sets this bit to differentiate between user & kernel mode.
  - Kernel Mode - 0
  - User Mode - 1
- **Privileged instructions** can only run in kernel mode.
- System calls switch mode to kernel, returning resets to user mode.

- Modern CPUs support Multi-Mode operations (e.g., Virtual Machine Manager mode for guest VMs).

Mode Switching

- When an **interrupt/trap occurs**, hardware **switches to kernel mode**.
- Once the OS completes its task, it **returns to user mode**.
- The switching between modes occurs by changing the value of the mode bit.



Timer in Operating Systems

- Prevents infinite loops & resource hogging.
- **Timer Interrupt:** Occurs after a **fixed or variable time period**.
- **Working →**
  1. OS sets a **counter**.
  2. Each clock tick **decrements** the counter.
  3. When **counter reaches zero**, an **interrupt occurs**.
- Ensures fair resource allocation.
- Used to terminate long-running user processes.

KERNEL DATA STRUCTURES

Array

- A **simple data structure** where each element is accessed directly.
- **Main memory** is structured as an array.
- Data is accessed as: **item number × item size**.
- Challenges:
    - Storing **variable-sized** items.
    - Removing items **while preserving order**.

Lists

- **Used extensively** in OS for dynamic data storage.
- **Types:**
    1. **Singly Linked List:** Each item points to its successor.
    2. **Doubly Linked List:** Each item points to both predecessor and successor.
    3. **Circular Linked List:** The last element points to the first.

Advantages of Linked Lists

- Accommodates **variable-sized** items.
- Easy **insertion & deletion** of elements.

Disadvantages

- **Searching is slow** – O(n) time complexity.

Usage in OS

- Used in **kernel algorithms**.
- Constructing **stacks and queues**.

Stack

- **LIFO (Last In, First Out)** structure.

- Used in OS for:
  - **Function calls –** stores parameters, local variables, and return addresses.
  - **Recursion handling**.

Queue

- **FIFO (First In, First Out)** structure.
- Used in OS for:
  - **CPU scheduling –** tasks waiting for execution.
  - **Printer queues –** jobs are printed in submission order.

Trees

- **Hierarchical data structure**.
- **Binary Search Tree (BST):**
  - Left child ≤ Parent ≤ Right child.
  - **Search time: O(n)** (worst case).
- **Balanced Binary Search Tree:**
  - **Maximum height = log(n)**.
  - **Search time: O(log n)**.
  - Used in **CPU Scheduling** in Linux.

Hash Functions & Maps

- **Hash Function:** Maps inputs to a fixed-size value.
- **Hash Map:** Stores **key-value pairs** using a hash function.
- **Search performance: O(1)**.

Bitmaps

- **String of binary digits** representing the state of resources.
- Example: **001011101**
  - **0** → Resource **available**.
  - **1** → Resource **unavailable**.
- Used in OS to **track disk blocks, memory pages, and process availability**.

COMPUTING ENVIRONMENTS

Traditional Computing

- **Standalone general-purpose machines**.
- Increasing interconnection via **the Internet**.
- Examples:
  - **Thin clients** (Network computers).
  - **Home systems with firewalls** for security.

Mobile Computing

- Includes **smartphones, tablets, and laptops**.
- **Differences from traditional computing:**
  - **More OS features** (GPS, gyroscope).
  - Enables **augmented reality applications**.
- Uses **Wi-Fi (IEEE 802.11)** or **cellular networks**.
- **Popular OS:** Apple iOS, Google Android.

Distributed Computing

- **Network of multiple systems working together**.
- Uses **TCP/IP** as the primary communication protocol.

- Types of Networks →
    1. **LAN (Local Area Network)**
    2. **WAN (Wide Area Network)**
    3. **MAN (Metropolitan Area Network)**
    4. **PAN (Personal Area Network)**
- **Network Operating System (NOS)** enables communication between systems.

Client-Server Computing

- **Clients request services from servers.**
- Examples:
    - **Compute-server** – Provides processing services.
    - **File-server** – Stores and retrieves files.
- **Dumb terminals replaced by smart PCs**.

Peer-to-Peer (P2P) Computing

- **All nodes are equal (no dedicated server).**
- Nodes act as **both clients and servers**.
- Example: **Napster, Gnutella, VoIP (Skype)**.

Virtualization

- **Running multiple OSes on a single system.**
- **Emulation:** Runs OS on different CPU types (slowest method).
- **Virtual Machines (VMs):**
    - OS runs on a **Virtual Machine Manager (VMM)**.
    - Example: **VMware running Windows XP on macOS**.

Uses of Virtualization

- Running **multiple OSes** on the same device.
- App development for **different platforms**.
- **Data center management** and cloud computing.

Cloud Computing

- Delivers computing/storage services over a network.
- Based on virtualization.

Types

1. **Public Cloud** – Available to anyone (e.g., AWS, Google Cloud).
2. **Private Cloud** – Used by a single company.
3. **Hybrid Cloud** – Combination of public & private clouds.

Cloud Services

- **Software as a Service (SaaS):** Applications via the Internet (e.g., Google Docs).
- **Platform as a Service (PaaS):** Ready-to-use environments (e.g., database servers).
- **Infrastructure as a Service (IaaS):** Storage & servers over the Internet.

Real-Time Embedded Systems

- **Special-purpose systems** with real-time constraints.
- Found in **automobiles, industrial robots, medical devices, etc.**
- **Real-Time OS (RTOS)** must complete tasks **within a fixed time**.

OS SERVICES

- OS provides an **execution environment** and various **services** for programs and users.

- Services help with **user interaction, program execution, and resource management**.

User Services

User Interface (UI)

- Almost all OSes have a **UI** for interaction.
- Types of UI:
  - **Command-Line Interface (CLI) –** Uses command interpreters (e.g., Bash, PowerShell).
  - **Graphical User Interface (GUI) –** Uses windows, icons, and menus.
  - **Batch Processing –** Executes predefined commands without user interaction.

Program Execution

- OS must:
  - Load programs into memory.
  - Execute programs
  - Handle **normal or abnormal termination** (error handling).

I/O Operations

- Programs may need **I/O devices** (keyboard, disk, printer, etc.).
- OS manages I/O requests and operations.

File-System Manipulation

- OS provides file-related services:
  - Read, write, create, delete files and directories.
  - Search files, list file details, manage permissions.

Communication

- Processes communicate via:
    - **Shared Memory** (direct access to memory segments).
    - **Message Passing** (data packets managed by OS).
- Can be within a single system or across a **network**.

Error Detection

- OS monitors errors in:
    - CPU & memory hardware.
    - I/O devices.
    - User programs.
- Provides **debugging tools** for users and programmers.

System Resource Management Services

Resource Allocation

- OS **allocates resources** (CPU, memory, I/O devices) to multiple users and processes.

Accounting

- Tracks **resource usage** by users and processes.
- Helps in **billing, optimization, and system monitoring**.

Protection & Security

- **Protection:** Ensures controlled access to system resources.
- **Security:** Prevents unauthorized access via **user authentication** and **access controls**.
- Extends to **defending external I/O devices from attacks**.

SYSTEM CALLS

- System calls **provide an interface** for user programs to request OS services.
- Available in **higher-level languages** (C, Python) or **assembly language**.
- Example:
  - `printf()` in C invokes the `write()` system call to send output to a device.

Types of System Calls

1. **Process Control** – Create, terminate, and manage processes.
2. **File Manipulation** – Open, close, read, write files.
3. **Device Manipulation** – Request and release hardware resources.
4. **Information Maintenance** – Get system and process details.
5. **Communication** – Send and receive messages.
6. **Protection** – Set file permissions and access controls.

DESIGN and IMPLEMENTATION of OS

Design Goals

- OS design starts with defining **goals & specifications**.
- **User Goals:**
  - Easy to use, reliable, safe, fast.
- **System Goals:**
  - Simple to design, implement, maintain, and efficient.

Policy vs. Mechanism

- **Policy:** What will be done?
- **Mechanism:** How to do it?
- **Why separate them?**
  - Allows flexibility for future changes.

    - Example: A **timer** mechanism can implement different policies to prevent programs from running too long.

Implementation

- Early OSes were written in **assembly language**.
- Now implemented in **C, C++**, with scripts in **Perl, Python, Shell**.
- **Low-level parts** (like bootloader) use **assembly**.
- **High-level parts** use **C & scripting languages** for portability.
- **Emulation** allows an OS to run on non-native hardware.

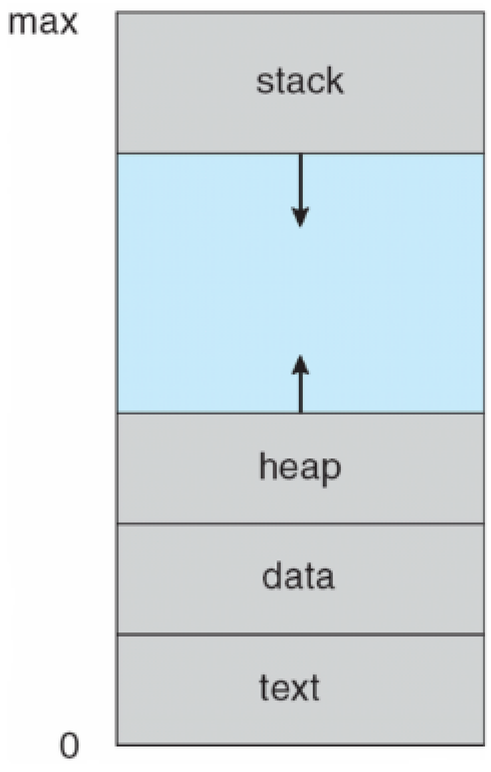
PROCESS CONCEPT

- An OS **executes multiple programs**:
    - **Batch systems** – Jobs.
    - **Time-shared systems** – User programs or tasks.
- **Process:** A program in execution.
- **Program (Passive) vs. Process (Active):**
    - A **program** is stored on disk as an executable file.
    - A **process** is created when the program is loaded into memory and executed.
- **One program can create multiple processes** (e.g., multiple users running the same application).

Process Identifiers

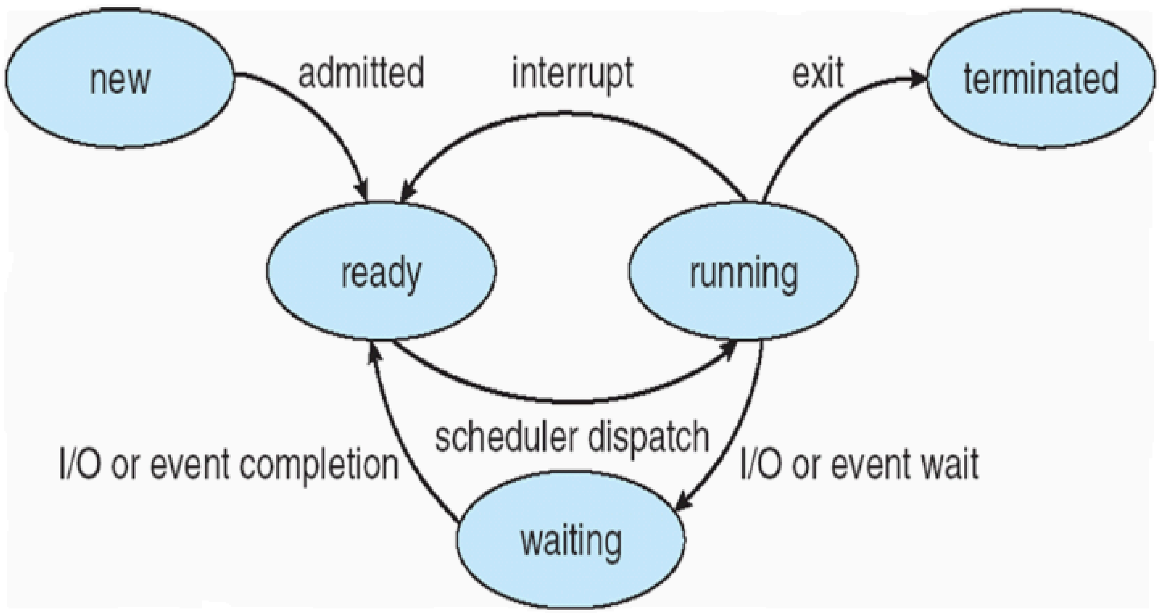
- Every process has a **unique non-negative integer** as its **process ID (PID)**.
- **PID is the only always-unique identifier** of a process.
- **Reused after termination –** but UNIX systems delay reuse to prevent confusion.

Process Structure in Memory



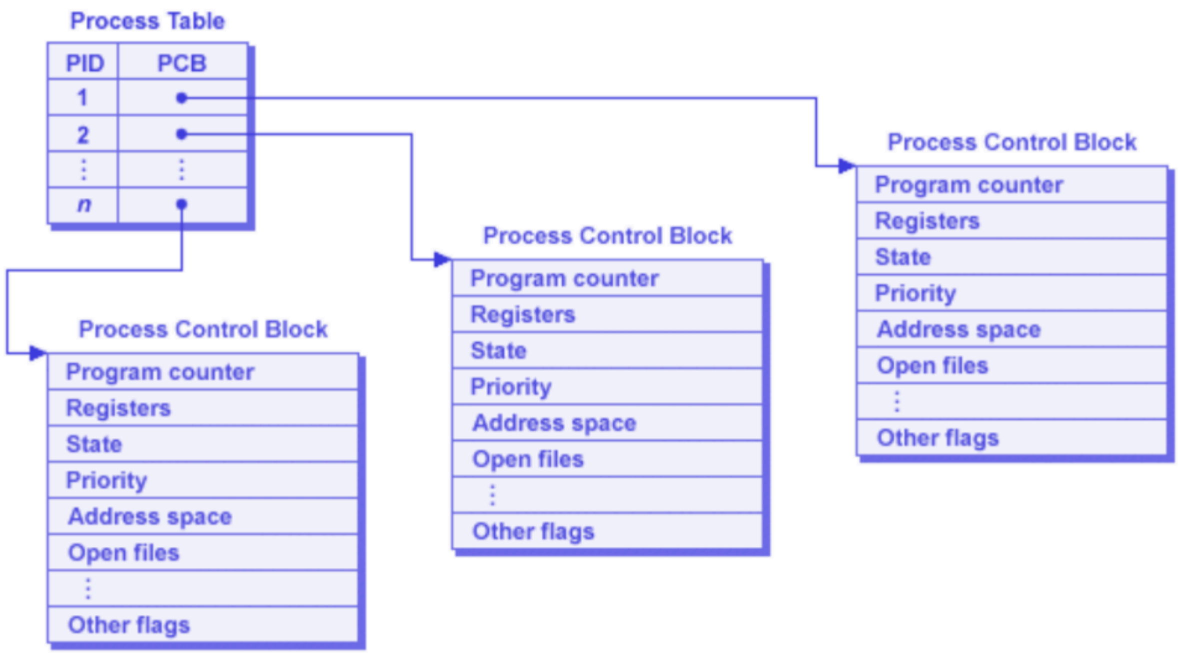
- **Text Section** – Contains the program code.
- **Program Counter** – Tracks the current instruction being executed.
- **Processor Registers** – Store process-related data.
- **Stack** – Stores function parameters, return addresses, and local variables.
- **Data Section** – Contains global variables.
- **Heap** – Stores dynamically allocated memory during runtime.

Process State



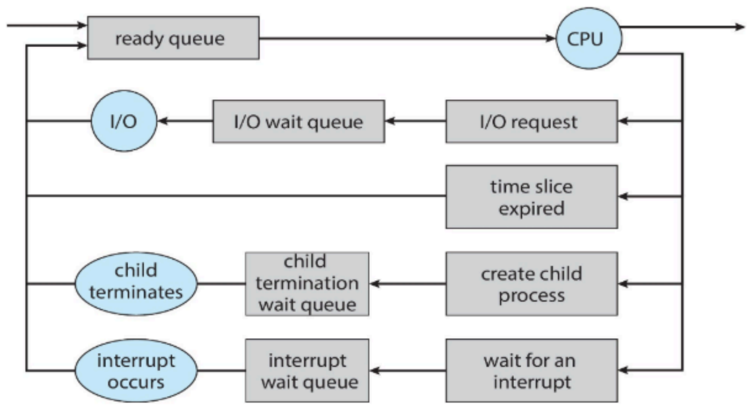
- A process transitions between different **states**:
  - **New** – Process is being created.
  - **Running** – Process instructions are being executed.
  - **Waiting** – Process is waiting for an event (e.g., I/O operation).
  - **Ready** – Process is ready to be assigned to a CPU.
  - **Terminated** – Process has finished execution.

Process Control Block (PCB)



- **Every process has a PCB** that stores essential information:
    - **Process State** – Running, waiting, etc.
    - **Program Counter** – Address of next instruction.
    - **CPU Registers** – Stores values for execution.
    - **CPU Scheduling Info** – Priorities, queue pointers.
    - **Memory Management Info** – Allocated memory.
    - **Accounting Info** – CPU usage, elapsed time.
    - **I/O Status** – Open files and assigned devices.

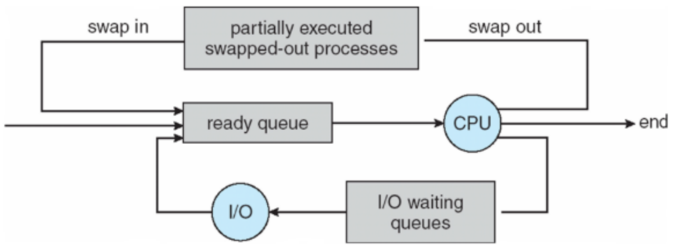
Process Scheduling



- **Objective:** Maximize CPU utilization by switching between processes efficiently.
- **Schedulers:**
  1. **Job Queue:** All processes in the system.
  2. **Ready Queue:** Processes in main memory, waiting for CPU.
  3. **Device Queue:** Processes waiting for I/O operations.
- **Process Migration:** Processes move between queues based on execution needs.

Types of Schedulers

1. **Short-Term Scheduler (CPU Scheduler)**
   - Selects which process runs next.
   - **Runs frequently** (every few milliseconds) → Must be fast.
2. **Long-Term Scheduler (Job Scheduler)**
   - Decides which processes enter the ready queue.
   - **Runs infrequently** → Controls **degree of multiprogramming**.
3. **Medium-Term Scheduler**
   - **Swaps processes** between memory and disk when necessary.
   - Used to **decrease multiprogramming** when memory is low.



Process Types

- **I/O-Bound Process:** More time spent on I/O, shorter CPU bursts.
- **CPU-Bound Process:** More time spent on computations, fewer but longer CPU bursts.
- **Good process mix** is necessary for efficient CPU scheduling.

Context Switching

- Switching between processes requires saving the old process state and loading the new one.
- **Process context (stored in PCB) must be restored during switch.**
- **Context-switching is overhead** (no useful work done during switch).
- **Faster switching depends on hardware support** (e.g., multiple CPU register sets).

Operations on Processes

Process Creation

- **Parent Process creates Child Processes**, forming a tree.
- **Process Identifier (PID)** uniquely identifies each process.
- **Resource Sharing Options:**
  1. Parent & child **share all** resources.
  2. Child **inherits a subset** of parent's resources.

3. Parent & child **have separate** resources.

- **Execution Options:**
  - Parent and child run concurrently.
  - Parent waits until child finishes.

Process Creation in UNIX

- **fork()** system call creates a new process.
- **exec()** replaces a child process's memory with a new program.

Process Termination

- A process terminates using **exit() system call**.
- **Parent retrieves termination status** using `wait()`.
- **OS deallocates resources** used by the process.

Reasons for Process Termination by Parent:

1. Child exceeds allocated resources.
2. Task assigned to child is no longer needed.
3. Parent is terminating, and OS doesn't allow orphan processes.

Orphan & Zombie Processes

- **Zombie Process:** Child has terminated, but parent hasn't called `wait()`.
- **Orphan Process:** Parent terminates before the child.

---

fork() – Creating a New Process

- A process can create a child process using `fork()`.
- **Return values:**
  - **Child process** gets **0**.
  - **Parent process** gets **PID of child**.

- **Error** returns **-1**.
- The child gets a **copy** of the parent's **data, heap, and stack**.

---

exit() – Normal Process Termination

A process can terminate normally via:

1. Returning from `main()`.
2. Calling `exit()` function.
3. Calling `_exit()` or `_Exit()` function.
4. Returning from the last thread's start routine.
5. Calling `pthread_exit()` from the last thread.

Abnormal Process Termination

1. Calling **abort()**.
2. Receiving **certain signals** (e.g., SIGKILL).
3. Last thread responds to a **cancellation request**.

Process Cleanup by OS

- **Releases memory** used by the process.
- Closes all open file descriptors.

Waiting for Process Termination

wait() & waitpid()

- A process calling `wait()` or `waitpid()` can:
    1. **Block** if its child processes are still running.
    2. **Return immediately** if a child has terminated.
    3. **Return error** if no child exists.
- **Returns:**

    - **PID of terminated child** (success).
    - **0** if state hasn't changed.
    - **1** on failure.
- If `wait()` is triggered by **SIGCHLD signal**, it returns immediately.

---

waitid()

- Used to **wait for specific child processes** based on PID or group ID.
- **Returns:**
    - **0 on success**.
    - **1 on error**.

Replacing a Process with a New Program

exec()

- **Replaces the calling process with a new program.**
- **Process ID remains the same**, but the **program code, data, heap, and stack are replaced**.
- **No return on success** (new program starts execution).
- **Returns** `1` **on error**.
- ***execl () →***
    - allows passing arguments as a variable length list
- ***execv () →***
    - passes arguments as an array of strings
- ***execle () →***
    - allows passing environment variables explicitly
- ***execve () →***
    - it takes both arguments and environment variable as arrays

- ***execlp () →***
  - searches for program in the directories listed in the path environment variables
- ***execvp () →***
  - it automatically searches the path environment variable to locate the program if its name is provided without a path
- ***fexecve () →***
  - instead of using a file path, it uses a file descriptor that refers to the executable file

---

Getting Process Information

getpid() & getppid()

- **getpid()** – Returns the **PID of the calling process**.
- **getppid()** – Returns the **PID of the parent process**.
  - If the parent has terminated, the process is **re-parented** to another process (usually `init`).

Race Condition

- Occurs when **multiple processes share data**, and the **execution order affects the final outcome**.
- **fork() can cause race conditions** if execution depends on whether the **parent or child runs first**.
- The execution order of parent and child **is unpredictable**.

Avoiding Race Conditions

- Use **wait() or waitpid()** to synchronize processes.
- Polling (looping to check parent/child termination) wastes CPU time.

- **Signaling mechanisms** (e.g., signals, semaphores) help **synchronize processes efficiently**.

CPU SCHEDULING

- In a **single-core CPU**, only **one process runs at a time**, while others wait.
- **Multiprogramming** maximizes **CPU utilization** by keeping processes running.
- Several processes are kept in **memory simultaneously**.
- When a process **waits for I/O**, the CPU is given to another process.

CPU–I/O Burst Cycle

- **Process execution alternates between:**
  1. **CPU bursts** (execution).
  2. **I/O bursts** (waiting for I/O).
- **Multiprogramming maximizes CPU utilization** by scheduling another process when one is waiting.

CPU Burst Characteristics

- **I/O-bound processes** → Many short CPU bursts.
- **CPU-bound processes** → Few long CPU bursts.
- CPU burst times follow a **specific frequency distribution** (histogram-based).

CPU Scheduling Mechanisms

CPU Scheduler (Short-Term Scheduler)

- Selects **processes from the ready queue** for CPU execution.
- Ready queue can be structured as:
  - **FIFO (First-In-First-Out)**
  - **Priority Queue**

    - **Tree-Based Structure**
    - **Unordered Linked List**
- **Process Control Blocks (PCBs)** store scheduling details.

Preemptive vs Non-Preemptive Scheduling

- **Scheduling occurs when a process:**
    - **Switches from running → waiting state** (e.g., I/O request).
    - **Switches from running → ready state** (e.g., preempted by another process).
    - **Switches from waiting → ready state** (e.g., I/O completion).
    - **Terminates.**

Preemptive vs Non-Preemptive

Scheduling Type	Description	Examples
Non-Preemptive	A running process **cannot be interrupted** until it finishes or waits	Used in Windows 3.x, old macOS versions
Preemptive	A running process **can be interrupted** by a higher-priority process	Used in Windows 95+, macOS, modern Linux

Disadvantages of Preemptive Scheduling

- **Race Conditions →**
    - A process updating shared data might be **interrupted**, leaving the data in an inconsistent state.
- **Solution →**
    - Use **mutex locks** to prevent **data corruption**.
- Most **modern OS kernels** support **full preemptive scheduling**.

DISPATCHER and LATENCY

Dispatcher Module

- Manages context switching between processes.
- ***Functions →***
    - **Switches context** (saves old process state, loads new process state).
    - **Switches to user mode**.
    - **Jumps to process execution point**.

Dispatch Latency

- **Time taken** to switch CPU control from one process to another.
- Lower latency = faster process switching = better performance.

Scheduling Performance Criteria

Metric	Definition	Goal
CPU Utilization	Percentage of time CPU is busy	**Maximize**
Throughput	Number of processes completed per time unit	**Maximize**
Turnaround Time	Total time from submission to completion	**Minimize**
Waiting Time	Time spent in the ready queue	**Minimize**
Response Time	Time from request submission to first response	**Minimize**

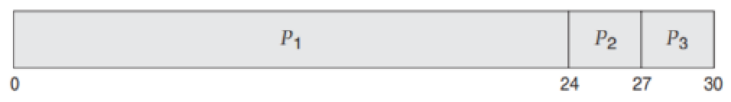
CPU SCHEDULING ALGORITHMS

First-Come, First-Served (FCFS) Scheduling

- Processes are scheduled in the **order of arrival**.
- **Non-preemptive** – Once a process gets CPU, it runs until completion.

Example 1 (Processes arrive in order P1, P2, P3)

The Gantt Chart for the schedule is:

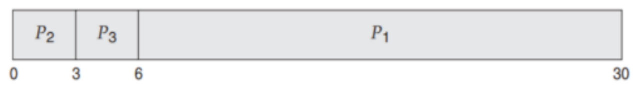


Process	Burst Time	Waiting Time
P1	24	0
P2	3	24
P3	3	27

- **Average Waiting Time:** (0 + 24 + 27) / 3 = **17 ms**

Example 2 (Processes arrive in order P2, P3, P1)

The Gantt Chart for the schedule is:



Process	Burst Time	Waiting Time
P2	3	0
P3	3	3
P1	24	6

- **Average Waiting Time:** (6 + 0 + 3) / 3 = **3 ms**

Disadvantages

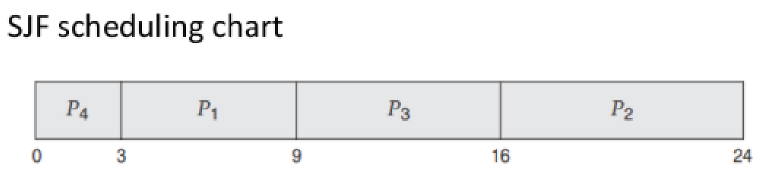
- **Convoy Effect →**
  - Short processes must wait behind long processes.

    - **Example:** A CPU-bound process delays I/O-bound processes, making the system inefficient.
- **Not suitable for time-sharing systems** (one process may keep CPU for too long).

Shortest Job First (SJF) Scheduling

- **Processes with the shortest burst time are scheduled first.**
- **Optimal** – Gives the minimum average waiting time.
- Can be preemptive or non-preemptive.

Non-Preemptive SJF Scheduling



Process	Burst Time	Waiting Time
P4	3	0
P1	6	3
P3	7	9
P2	8	16

- **Average Waiting Time:** (3 + 16 + 9 + 0) / 4 = **7 ms**
- **Comparison with FCFS:**
    - FCFS waiting time = **10.25 ms**
    - SJF waiting time = **7 ms** (better).

Determining the Length of the Next CPU Burst

- Estimated using **exponential averaging** formula:

$$Tn + 1 = \alpha Tn + (1 - \alpha)Tn$$

- ***Where →***
  - **Tn+1** = Predicted next burst time
  - **α** = Weight factor (commonly **0.5**)
  - **Tn** = Most recent actual burst time

Example Calculation (α = 0.5, previous bursts: 8, 7, 4, 16)

Step	Formula Used	Predicted Time (T)
1	T2 = 0.5(4) + 0.5(10)	**7**
2	T3 = 0.5(7) + 0.5(7)	**7**
3	T4 = 0.5(8) + 0.5(7)	**7.5**
4	T5 = 0.5(16) + 0.5(7.5)	**11.8**

- **Prediction for next burst (T5) = 11.8 ms**

Exponential Averaging in Burst Time Prediction

- If **α = 0**, the prediction does not consider recent CPU bursts.
- If **α = 1**, only the most recent burst is considered.
- Formula expansion:

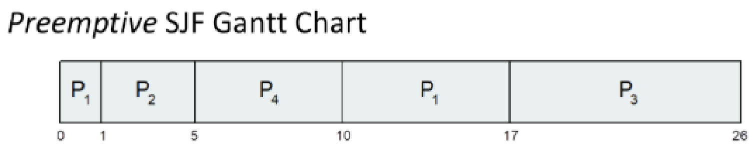
- Recent bursts have **higher weight**.
- Older bursts have **lower weight**.

Preemptive SJF (Shortest-Remaining-Time-First - SRTF)

- If a new process arrives with a shorter burst time, it preempts the current process.

- More responsive than non-preemptive SJF.

Preemptive SJF



Process	Arrival Time	Burst Time	Completion Time	Waiting Time
P1	1	10	10	9
P2	1	1	1	0
P3	2	17	19	15
P4	3	5	8	3

- **Average Waiting Time:** (9 + 0 + 15 + 3) / 4 = **6.5 ms**

Comparison of FCFS and SJF

Scheduling Algorithm	**Fairness**	**Efficiency**	**Best Used For**
FCFS	Poor (convoy effect)	Simple but inefficient	**Batch processing**
SJF (Non-preemptive)	Better than FCFS	Minimum waiting time but hard to predict	**Short predictable tasks**
SRTF (Preemptive SJF)	Best	More responsive	**Interactive systems**

Priority Scheduling

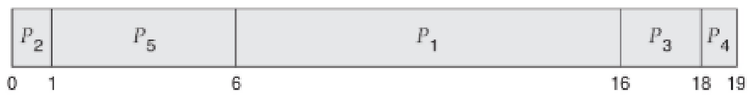
- Each process is assigned a **priority number** (integer).
- The **CPU is allocated to the process with the highest priority** (smallest integer = highest priority).

- Can be **preemptive** or **non-preemptive**.
- **SJF (Shortest Job First) is a type of priority scheduling** where the priority is the inverse of the CPU burst time.

Disadvantages

- **Starvation →**
    - Low-priority processes may never get CPU time.
- **Solution: Aging →**
    - Increase the priority of processes over time to prevent starvation.

Example



Process	Burst Time (ms)	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

- **Average Waiting Time:** (6 + 0 + 16 + 18 + 1) / 5 = **8.2 ms**

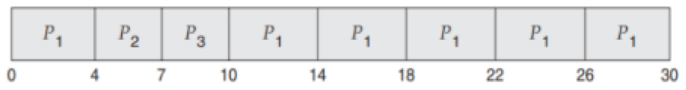
Round-Robin (RR) Scheduling

- Designed for **time-sharing systems**.
- Similar to **FCFS**, but **preemptive**.
- Uses a **time quantum (time slice)**, typically **10–100 ms**.
- The **ready queue is a circular queue**.

- The **CPU scheduler cycles through processes**, allocating CPU time for up to **1 time quantum** per turn.

Example

- **Processes arrive at time 0**.
- **Time quantum = 4 ms**.



Process	Burst Time (ms)	Waiting Time (ms)
P1	24	6
P2	3	4
P3	3	7

- **Average Waiting Time:** (6 + 4 + 7) / 3 = **5.66 ms**

Time Quantum Effects

- **If there are n processes and time quantum = q**
  - Each process gets **1/n of CPU time** in **chunks of at most q time units**.
  - Maximum wait time for a process: **(n - 1) × q**.
- **Example:**
  - 5 processes, **time quantum = 20 ms**.
  - Each process gets up to **20 ms** every **100 ms**.

Choosing the Right Time Quantum

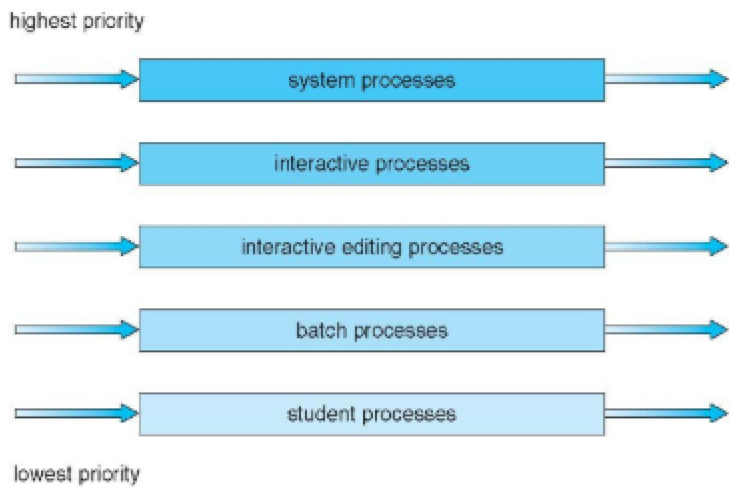
- Large time quantum → behaves like FCFS.
- Small time quantum → too many context switches → inefficient.
- Optimal time quantum minimizes waiting time without excessive context switching.

Example

Time Quantum (ms)	Context Switches	Impact
12	0	Process completes in one quantum.
6	1	Process requires 2 quanta, 1 switch.
1	9	Too many context switches, slow execution.

ADVANCED CPU SCHEDULING

Multilevel Queue Scheduling



- The **ready queue is divided into multiple queues** based on process type.
- Processes are permanently assigned to a queue.
- Each queue has its own scheduling algorithm:
    - **Foreground (interactive)** → **Round-Robin (RR)**.
    - **Background (batch)** → **First-Come, First-Served (FCFS)**.
- **Scheduling between queues:**

    - **Time slice allocation:**
        - CPU time is divided among queues (e.g., 80% for foreground, 20% for background).

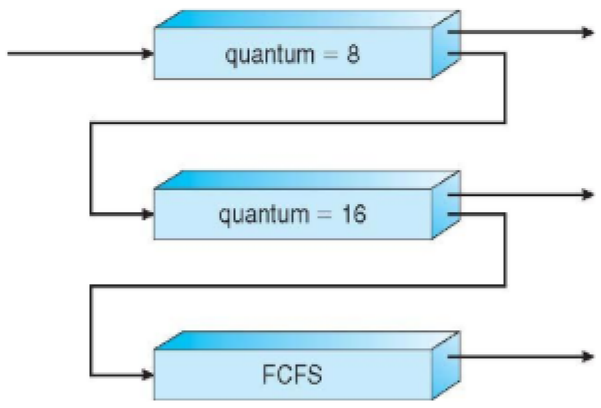
Characteristics

- **Absolute priority:** A process in a lower-priority queue will not run unless all higher-priority queues are empty.
- **Example:** If an interactive process arrives while a batch process is running, the batch process is preempted.

Multilevel Feedback Queue Scheduling

- Processes can move between queues.
- Aging can be implemented by gradually moving processes to higher-priority queues.
- ***Defined by the following parameters →***
    - Number of queues.
    - Scheduling algorithm for each queue.
    - Criteria for upgrading/demoting processes.
    - Method to determine queue assignment when a process enters the system.

Example



- **Three queues:**
  - Q0 → RR with time quantum 8 ms.
  - Q1 → RR with time quantum 16 ms.
  - Q2 → FCFS.
- **Scheduling process:**
  1. New jobs enter Q0 and get 8 ms CPU time.
  2. If not completed in 8 ms, moved to Q1.
  3. In Q1, gets 16 ms CPU time.
  4. If not completed in 16 ms, moved to Q2 (FCFS).
- **Priority Execution Order:**
  - Queue 0 (highest priority) executes first.
  - Queue 1 executes only if Q0 is empty.
  - Queue 2 executes only if Q0 and Q1 are empty.
- **Preemption Rules:**
  - A process arriving in **Q0 preempts a process in Q1 or Q2.**
  - A process arriving in **Q1 preempts a process in Q2.**

Multiple-Processor Scheduling

- If **multiple CPUs** are available, **load balancing** becomes important.

- Scheduling complexity increases with multiple processors.

Asymmetric Multiprocessing (AMP)

- One processor (master server) handles all scheduling and system tasks.
- Other processors execute only user code.
- ***Advantage →***
    - **Simple implementation** (only one processor manages system data).
- ***Disadvantage →***
    - Single point of failure.

Symmetric Multiprocessing (SMP)

- Each processor is self-scheduling.
- All processors share a common ready queue or each has a private queue.
- ***Advantage →***
    - More efficient than AMP (no single point of failure).
- Most modern operating systems support SMP.

Processor Affinity

- **Processor affinity** means a process should run on the same processor where it started.
- **Why?**
    - The processor **cache stores frequently accessed data.**
    - **Moving a process to another CPU** invalidates cache and **requires cache repopulation** (slow).

Types of Processor Affinity

- **Soft Affinity:**
    - OS **tries** to keep a process on the same processor but **allows migration** if necessary.

- **Hard Affinity:**
  - OS provides **system calls** to restrict a process to a subset of processors.

Load Balancing in Multiprocessor Systems

- Distribute workload evenly across processors.
- Required in systems where each processor has a private queue.
- Not necessary if all processors share a single queue.

Types of Load Balancing

- **Push Migration:**
  - A system task **periodically checks CPU loads** and moves processes **from overloaded to underloaded processors.**
- **Pull Migration:**
  - An **idle processor pulls** a process from a busy processor's queue.
- Both push and pull migration can work together.

LINUX and WINDOWS SCHEDULING

Linux Scheduling Through Version 2.5

- Used a variation of the standard UNIX scheduling algorithm.
- Did not support Symmetric Multiprocessing (SMP).
- **Performed poorly** for large-scale systems.

Linux Scheduling in Version 2.5

- Introduced **O(1) scheduling time** (constant time complexity).
- **Supported SMP** with **processor affinity and load balancing**.

- **Improved performance**, but had **poor response times for interactive processes**.

Linux Scheduling in Version 2.6.23+ (Completely Fair Scheduler - CFS)

- CFS became the default scheduling algorithm.
- Introduced **Scheduling Classes**:
  - Each class has a **specific priority**.
  - **Scheduler picks the highest-priority task** from the highest-priority class.
- Two default scheduling classes:
  - **CFS Scheduling Class**.
  - **Real-Time Scheduling Class**.

CPU Time Allocation in CFS

- Each task gets a **proportion of CPU time** based on its **nice value (-20 to +19)**.
- **Lower nice value = higher priority = more CPU time**.
- **Target Latency**: Defines the time interval within which every task should run at least once.
- More tasks increase target latency, reducing per-task CPU time.

Virtual Runtime (vruntime) in CFS

- CFS does not use direct priority values.
- **Each task has a virtual runtime →**
  - Higher priority → **Lower vruntime decay rate**.
  - Lower priority → **Higher vruntime decay rate**.
- Scheduler selects the task with the **lowest vruntime.**

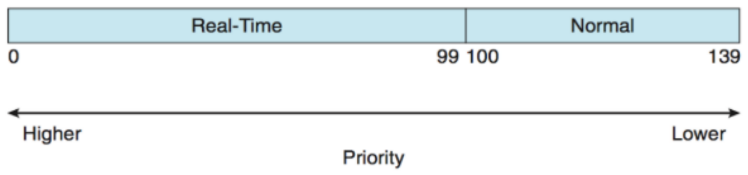
CFS and Balanced Binary Search Tree

- Each task is stored in a **balanced binary search tree (BST)**.
- **Key in BST:** The task's **vruntime**.
- **Operations:**
  - Task enters the queue → Added to BST.
  - Task is removed → Deleted from BST.
  - Scheduler selects the leftmost node (lowest vruntime) in O(log N) time.

Handling I/O-Bound vs. CPU-Bound Processes

- **I/O-bound processes have lower vruntime** → Higher priority.
- **CPU-bound processes accumulate vruntime faster** → Lower priority.
- If an I/O-bound process becomes runnable while a CPU-bound process is executing, it will preempt the CPU-bound process.

Linux Real-Time Scheduling



POSIX.1b Real-Time Scheduling

- Real-time tasks have **static priorities (0–99)**.
- **Real-time priority + normal priority mapped into a global priority scheme.**
- **Nice value mapping:**
  - **-20 maps to priority 100** (higher priority).
  - **+19 maps to priority 139** (lower priority).

Windows Scheduling

- Uses a **priority-based preemptive scheduling algorithm**.

- The **Windows kernel handles scheduling via the Dispatcher**.
- **The highest-priority thread always runs**.
- A thread continues execution until one of the following occurs →
    - A **higher-priority thread preempts it**.
    - It **terminates**.
    - Its **time quantum expires**.
    - It **calls a blocking system call**.

Windows Priority Scheme

32-Level Priority Scheme

- **Priority levels (0–31):**
    - **0** – Memory management thread.
    - **1–15** – Variable priority (for normal processes).
    - **16–31** – Real-time priority.
- Each priority level has a **separate queue**.
- If no runnable thread is found, the idle thread runs.

Windows API Priority Classes

Priority Class	Description
REALTIME_PRIORITY_CLASS	Highest priority, non-variable
HIGH_PRIORITY_CLASS	Above normal processes
ABOVE_NORMAL_PRIORITY_CLASS	Higher than normal
NORMAL_PRIORITY_CLASS	Default for most processes
BELOW_NORMAL_PRIORITY_CLASS	Lower than normal
IDLE_PRIORITY_CLASS	Lowest priority

- **Only REALTIME_PRIORITY_CLASS has a fixed priority**; all others are variable.

Windows Thread Priorities

Thread Priority in a Process

- A thread's priority is based on →
  - The priority class of its process.
  - Its relative priority within that class.

Thread Priority	Description
TIME_CRITICAL	Highest priority in class
HIGHEST	Very high priority
ABOVE_NORMAL	Slightly above normal
NORMAL	Default priority
BELOW_NORMAL	Slightly below normal
LOWEST	Low priority
IDLE	Lowest priority

Windows Foreground and Background Processes

- Windows distinguishes between **foreground and background processes**.
- Foreground processes get a 3x priority boost.
- This allows the active process to run longer before being preempted.

LINUX SHELL

- The **Linux shell** is a **command-line interpreter**.
- Provides an **interface** between the user and the **kernel**.
- Executes commands, applications, and scripts.
- Example: Entering `ls` runs the `ls` command.

Shell Command Syntax

- Commands follow this syntax:

  ```
  command [arg1] [arg2] ... [argn]
  ```

- Arguments (`arg1`, `arg2`, etc.) are **optional**.
- The shell parses the command and executes it if syntax is correct.
- Searches for the command in the **specified directory** or through the **$PATH** variable.

Types of Shells

- **sh** – Bourne Shell (original shell).
- **csh, tcsh** – C Shell and its extended version.
- **ksh** – Korn Shell (popular alternative).
- **bash** – Bourne Again Shell (default Linux shell, developed by GNU).

Check current shell:

```
echo $SHELL
```

Environment Variables

- **Defined for the current shell** and inherited by child shells.
- Used to pass information into spawned processes.
- **Shell variables** exist **only in the current shell**.

Common Environment Variables

Variable	Description
SHELL	Current shell interpreter.
TERM	Specifies terminal type.
USER	Currently logged-in user.

Variable	Description
PWD	Current working directory.
OLDPWD	Previous working directory.
PATH	List of directories checked for commands.
HOME	User's home directory.

View all environment variables:

```
env
```

or

```
printenv
```

Common Shell Variables

Variable	Description
BASHOPTS	List of options used when bash was executed.
BASH_VERSION	Human-readable Bash version.
BASH_VERSINFO	Machine-readable Bash version.

Shell Basics

Creating and Accessing Shell Variables

- Assign a variable:

  ```
  VAR_NAME="Hello"
  ```

- Access the variable:

  ```
  echo $VAR_NAME
  ```

Spawning a Child Shell Process

- A **child shell does not inherit shell variables** from the parent.
- Terminate a child shell by typing:

```
exit
```

- **Export variables to child shell:**

```
export VAR_NAME="Hello"
```

Removing a Shell Variable

- Unset a variable:

```
unset VAR_NAME
```

Setting Environment Variables at Login

- Edit the **.profile** file in the `$HOME` directory and add the `export` command.

Shell Control Flow

Conditional Statements (if)

- Example: Check if a file exists

```
if [ -f filename ]; then
   echo "File exists"
else
   echo "File does not exist"
fi
```

Loops

- **Nested Loop Example:**

```
for i in {1..3}; do
    for j in {1..3}; do
        echo "i=$i, j=$j"
    done
done
```

Continue and Break

- `break` **exits the loop.**
- `continue` **skips to the next iteration.**

  ```
  for i in {1..5}; do
      if [ $i -eq 3 ]; then
          continue
      fi
      echo "Iteration $i"
  done
  ```

Cron (Job Scheduling in Linux)

What is Cron?

- **Cron is a daemon** that runs **in the background** and executes scheduled tasks.
- **Used for job scheduling** in Unix/Linux systems.
- The `crond` daemon **reads the crontab (cron tables)** for predefined jobs.

Viewing Scheduled Jobs in Cron

- List scheduled jobs:

  ```
  crontab -l
  ```

Starting and Checking Cron Status

- Start the cron service:

```
sudo systemctl start crond
```

- Check cron service status:

```
sudo systemctl status crond
```

Adding Jobs to the Scheduler

- Edit crontab entries:

```
crontab -e
```

Cron Job Syntax

- Format:

```
* * * * * command_to_execute
┬ ┬ ┬ ┬ ┬
│ │ │ │ │
│ │ │ │ └── Day of the week (0-7, Sunday=0 or 7)
│ │ │ └──── Month (1-12)
│ │ └────── Day of the month (1-31)
│ └──────── Hour (0-23)
└────────── Minute (0-59)
```

Example Cron Jobs

Run a script every minute:

```
* * * * * /path/to/script.sh
```

Run a backup job at midnight daily:

```
0 0 * * * /path/to/backup.sh
```

Run a script every Monday at 3 AM:

```
0 3 * * 1 /path/to/weekly_report.sh
```