OPERATING SYSTEM AND SYSTEM PROGRAMMING

UNIT-I

Fundamentals of Operating Systems and Process Management

Introduction to Operating Systems:

Definition:

An **Operating System** is a System software that manages all the resources of the computing device.

- Acts as an interface between the software and different parts of the computer or the computer hardware.
- Manages the overall resources and operations of the computer.
- Controls and monitors the execution of all other programs that reside in the computer, which also includes application programs and other system software of the computer.
- Examples of Operating Systems are Windows, Linux, macOS, Android, iOS, etc.

Or

Definition: An Operating System (OS) is system software that manages computer hardware, software resources, and provides common services for computer programs. It acts as an intermediary between users and the computer hardware, enabling efficient execution of applications.

Basics :

Introduction to Operating System

An operating system acts as an intermediary between the user of a computer and computer hardware. In short its an interface between computer hardware and user.

- The purpose of an operating system is to provide an environment in which a user can execute programs conveniently and efficiently.
- An operating system is software that manages computer hardware and software. The hardware must provide appropriate mechanisms to ensure the correct operation of the computer system and to prevent user programs from interfering with the proper operation of the system.

Why do we need an Operating System?

Without an OS:

- You cannot run applications like browsers, games, or editors.
- Managing hardware like CPU, memory, and input/output devices will be complex for the user.
- The OS makes **computing easy, efficient, and secure**.

Main Functions of Operating System

✍a) Process Management

- A **process** is a running program.
- OS handles:
  - Creation and termination of processes.
  - Scheduling processes to use CPU efficiently.
  - Synchronization and communication between processes.

✍b) Memory Management

- OS manages **primary memory (RAM)**:
  - Allocates and deallocates memory to programs.
  - Keeps track of each byte of memory.
  - Ensures efficient use of memory without conflicts.

✍c) File Management

- OS manages **files and directories** on storage devices:
  - Creates, deletes, reads, writes, and organizes files.
  - Manages file permissions and security.

✍d) Device Management

- OS manages **input/output devices** like keyboard, mouse, printer, and disk drives using device drivers.
- Controls data transfer and error handling for devices.

✍e) Security and Protection

- Protects data from unauthorized access.
- Provides user authentication (passwords, biometrics).
- Controls access rights for files and resources.

GENERATIONS OF OPERATING SYSTEMS

1️⃣ First Generation (1940s – Early 1950s) – No Operating System

- **Computers used vacuum tubes and plugboards.**
- Programming done in **machine language (0s and 1s).**
- No OS; programs loaded manually using switches and punched cards.
- Execution was **slow and error-**.

✅**Example:** ENIAC, UNIVAC.

2️⃣ Second Generation (1955 – Mid 1960s) – Batch Processing Systems

- Computers used **transistors**.
- **Batch Operating Systems introduced.**
- Jobs collected into batches on punched cards and executed sequentially.
- **No user interaction during execution.**
- Used **job control languages (JCL)** for managing jobs.

✅**Features:**

- Automatic job sequencing.
- Reduced CPU idle time.

✅**Example:** IBM 7094.

3️⃣ Third Generation (Mid 1960s – 1970s) – Multiprogramming & Time-Sharing Systems

- Computers used **Integrated Circuits (ICs)**.
- **Multiprogramming introduced:** Multiple programs reside in memory, CPU switches between them to increase utilization.
- **Time-Sharing Systems:** Multiple users can interact with the computer simultaneously, each getting CPU time slices.
- Introduction of **spooling** for managing I/O efficiently.

✅**Features:**

- Better CPU utilization.
- Interactive computing started.
- User terminals connected to central system.

✅**Example:** IBM System/360.

Generation	Key Points

SUMMARY TABLE OF GENERATIONS IN OS

1 **First**	No OS, manual operation, machine language.

---

4 Fourth Generation (Late 1970s – Present) – Personal Computer (PC) Operating Systems

- Computers use **Very Large Scale Integration (VLSI) chips** (microprocessors).
- **Personal computers became popular**, requiring user-friendly OS.
- **Graphical User Interfaces (GUI)** introduced, making OS easier to use.
- Networking and distributed systems development started.

✓**Features:**

- Support for personal computing.
- Easy user interfaces (Windows, icons, menus).
- Networking and distributed computing support.

✓**Examples:** MS-DOS, Windows, MacOS, UNIX.

---

5 Fifth Generation (Present – Future) – Advanced & AI-Based Operating Systems

- 2Focus on **parallel processing, distributed systems, cloud OS, and AI integration**.
- Real-time OS for robotics and IoT devices.
- Increased security, stability, and efficiency.
- Virtualization and cloud computing support in OS.

✓**Features:**

- Intelligent operating systems using AI.
- Seamless cloud integration.
- Support for mobile and embedded systems.

✓**Examples:** Android, iOS, Windows 11, advanced Linux distributions, cloud-based OS systems.

---

2 **Second**	Batch OS, job sequencing, punched cards.
3 **Third**	Multiprogramming, time-sharing, spooling.
4 **Fourth**	GUI, PC OS, networking support.
5 **Fifth**	AI integration, cloud OS, real-time OS.

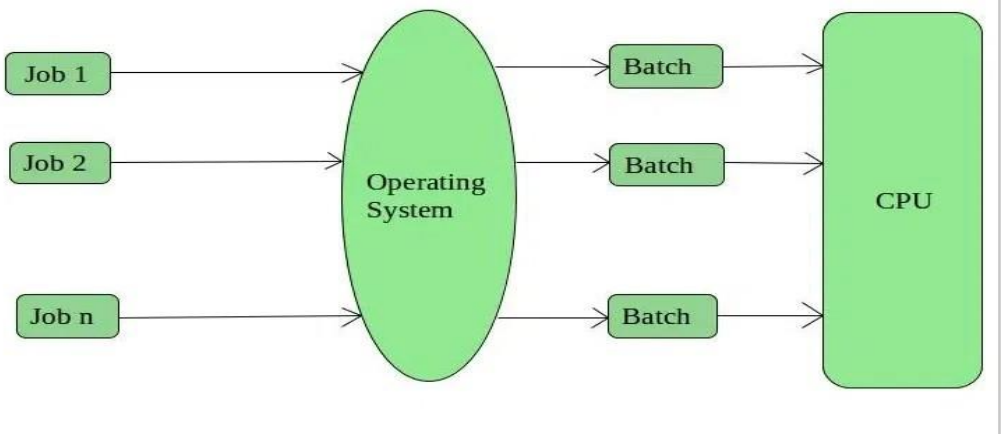
Operating System Types

Main Operating System Types

1. Batch Operating System

This type of operating system does not interact with the computer directly. There is an operator which takes similar jobs having the same requirements and groups them into batches. It is the responsibility of the operator to sort jobs with similar needs. Batch Operating System is designed to manage and execute a large number of jobs efficiently by processing them in groups.

Examples of Batch Operating Systems: Payroll Systems, Bank Statements, etc.



.

Advantages of Batch Operating System

- Multiple users can share the batch systems.
- The idle time for the batch system is very little.
- It is easy to manage large work repeatedly in batch systems. Ex: bank statements.

Disadvantages of Batch Operating System

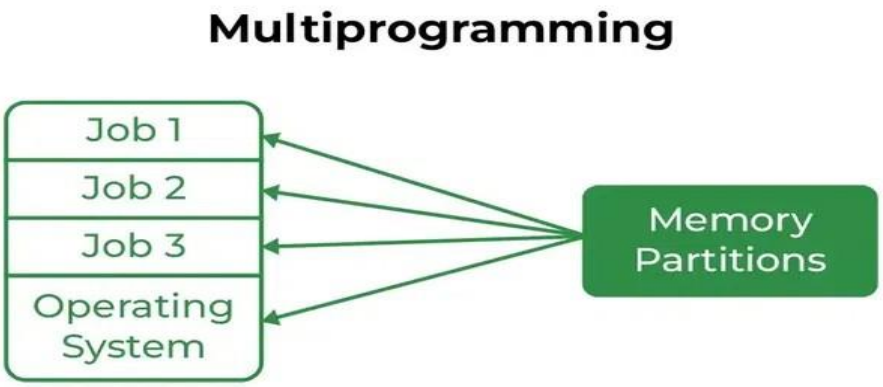
- CPU is not used efficiently. When the current process is doing IO, the CPU is free and could be utilized by other processes waiting. lk

k2. Multi-Programming Operating System

Multiprogramming Operating Systems can be simply illustrated as more than one program is present in the main memory and any one of them can be kept in execution. This is used for better utilization of resources.

Advantages of, Multi-Programming Operating System

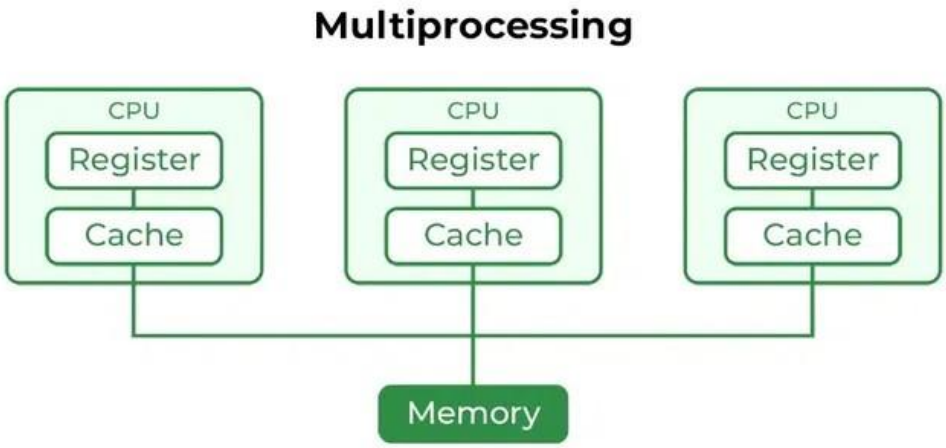
- CPU is better utilized, and the overall performance of the system improves.
- It helps in reducing the response time.



Example: desktop os,mobile os.applications os.' Media player

3. Multi-Processing Operating System

A Multi-Processing Operating Systemis a type of Operating System in which more than one CPU is used for the execution of resources. It betters the throughput of the System.

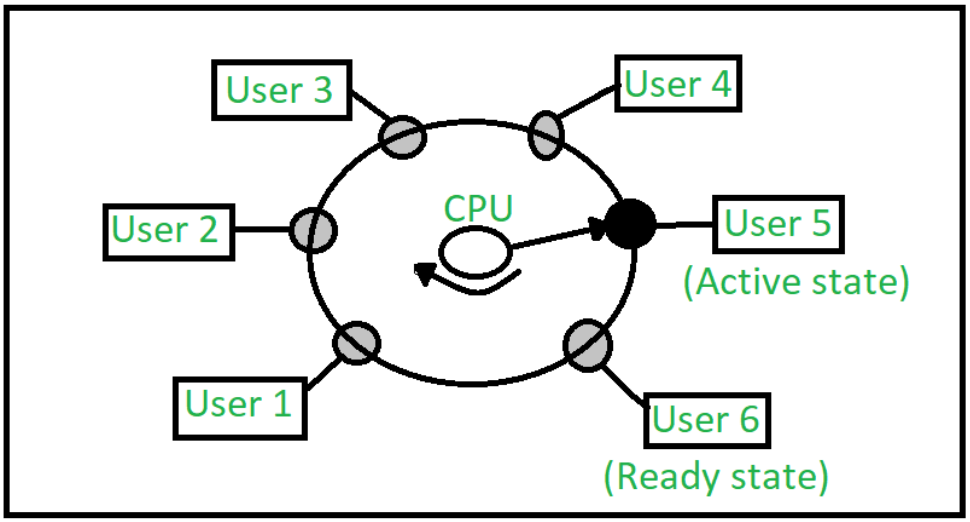


Examples:unix,linux.

- It increases the throughput of the system as processes can be parallelized.
- As it has several processors, so, if one processor fails, we can proceed with another processor

4. Multi-User Operating Systems

These systems allow multiple users to be active at the same time. This system can be either a multiprocessor or a single processor with interleaving



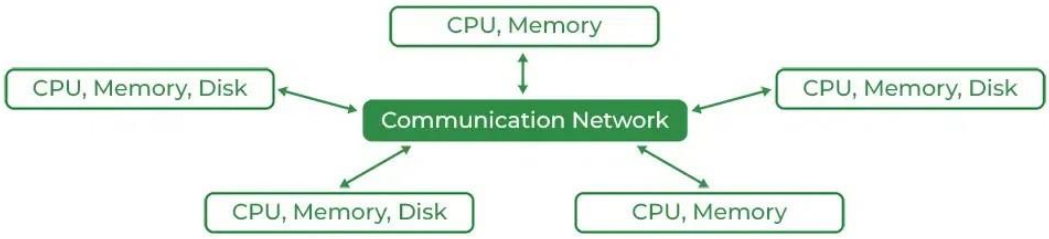
Advantages of a Multi-User Operating System

Examples:unix,macos.linus,

5. Distributed Operating System

These types of operating systems are a recent advancement in the world of computer technology and are being widely accepted all over the world and, that too, at a great pace. Various autonomous interconnected computers communicate with each other using a shared communication network. Independent systems possess their own memory unit and CPU. Systems. These systems' processors differ in size and function. The major benefit of working with these types of operating systems is that it is always possible that one user can access the files or software which are not present on his system but on some other system connected within this network, i.e., remote access is enabled within the devices connected to that network.

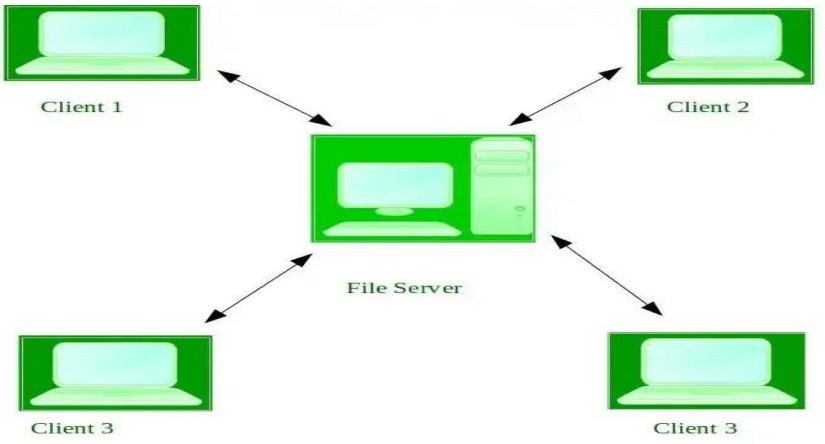
Architecture of Distributed OS



Examples: cloud computing, aws.social media platforms.

6. Network Operating System

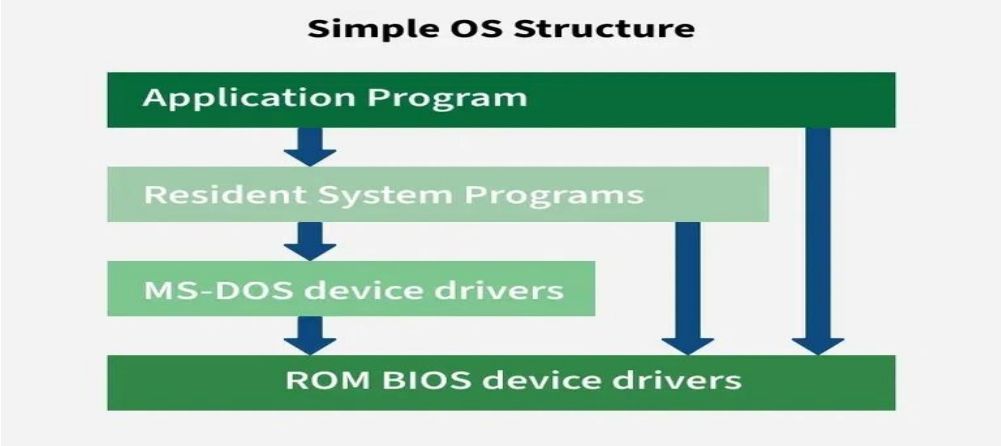
These systems run on a server and provide the capability to manage data, users, groups, security, applications, and other networking functions. These types of operating systems allow shared access to files, printers, security, applications, and other networking functions over a small private network. One more important aspect of Network Operating Systems is that all the users are well aware of the underlying configuration, of all other users within the network, their connections, etc., and that's why these computers are popularly known a tightly coupled systems.



Examples:server operating system.

Types of Operating System Structures

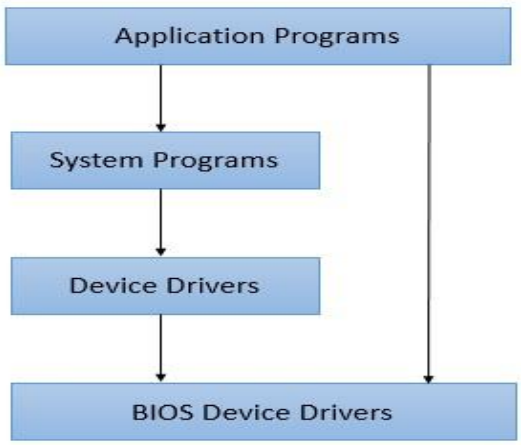
Simple Structure :Simple structure operating systems do not have well-defined structures and are small, simple, and limited. The interfaces and levels of functionality are not well separated. MS-DOS is an example of such an operating system. In MS-DOS, application programs are able to access the basic I/O routines. These types of operating systems cause the entire system to crash if one of the user programs fails.



OS STRUCTURE:

Simple Structure

There are many operating systems that have a rather simple structure. These started as small systems and rapidly expanded much further than their scope. A common example of this is MS-DOS. It was designed simply for amount for people. There was no indication that it would become so popular.



Advantages

Following are advantages of a simple operating system structure.

1. **Easy Development - In simple operation system, being very few interfaces, development is easy especially when only limited functionalities are to be delivered.**
2. **Better Performance - Such a sytem, as have few layers and directly interects with hardware, can provide a better performance as compared to other types of operating systems.**

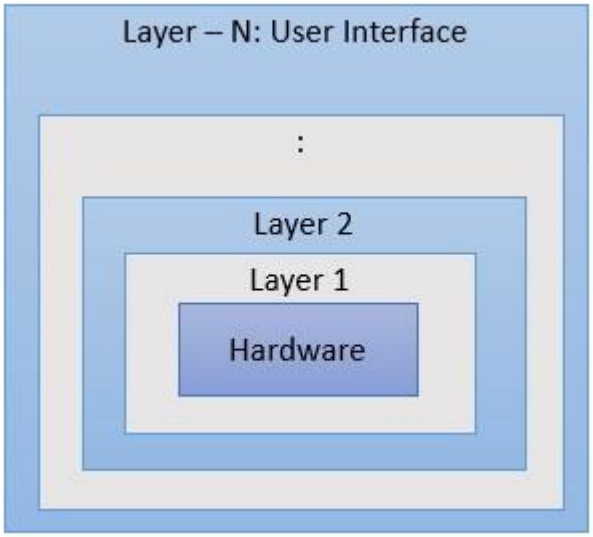
DIS ADVANTAGES

1. **Frequent System Failures**
2. **Poor Maintainability**

Layered Structure

One way to achieve modularity in the operating system is the layered approach. In this, the bottom layer is the hardware and the topmost layer is the user interface.

An image demonstrating the layered approach is as follows –



j

As seen from the image, each upper layer is built on the bottom layer. All the layers hide some structures, operations etc from their upper layers.

One problem with the layered structure is that each layer needs to be carefully defined. This is necessary because the upper layers can only use the functionalities of the layers below them.

Advantages

Following are advantages of a layered operating system structure.

- **High Customizable - Being layered, each layer implmentation can be customized easily. A new functionality can be added without impacting other modules as well.**
- **Verifiable - Being modular, each layer can be verified and debugged easily.**

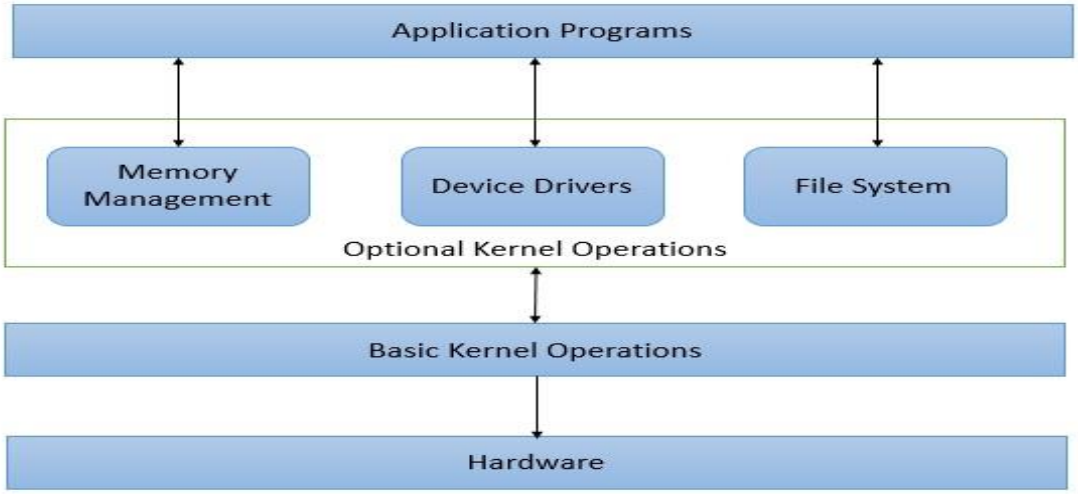
Disadvantages

Following are disadvantages of a layered operating system structure.

- **Less Performant - A layered structured operating system is less performant as compared to basic structured operating system.**
- **Complex designing - Each layer is to planned carefully as each layer communicates with lower layer only and a good design process is required to create a layered operating system.**

Micro-Kernel Structure

As in case monolith structure, there was single kernel, in micro-kernel, we have multiple kernels each one specilized in particular service. Each microkernel is developed independent to the other one and makes system more stable. If one kernel fails the operating sytem will keep working with other kernel's functionalities.



Advantages

Following are advantages of a microkernel operating system structure.

- **Reliable and Stable** - As multiple kernels are working simultaneously, chances of failure of operating sytem is very less. If one functionlity is down, operating system can still provide other functionalities using stable kernels.
- **Maintainability** - Being small sized kernels, code size is maintainable. One can enhance a microkernel code base without impacting other microkernel code base.

Disadvantages

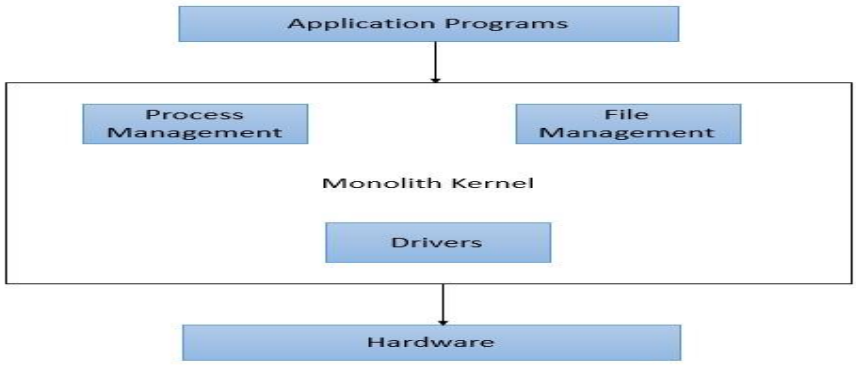
Following are disadvantages of a microkernel operating system structure.

- **Complex to Design** - Such a microkernel based architecture is difficult to design.
- **Performance Degradation** - Multi kernel, Multi-modular communication may hamper the performance as compared to monolith architecture.

Monolith Structure

In monolith structured operating system, a central piece of code called kernel is responsible for all major operations of an operating system. Such operations includes file management, memory management, device management and so on. The kernal is the main component of an operating system and it provides all the services of an operating system to the application programs and system programs.

The kernel has access to the all the resources and it acts as an interface with application programs and the underlying hardware. A monolithic kernel structure promotes timesharing, multiprogramming model and was used in old banking systems.



Advantages

Following are advantages of a monolith operating system structure.

1.Easy Development - As kernel is the only layer to develop with all major functionalities, it is easier to design and develop.

2.Performance - As Kernel is responsible for memory management, other operations and have direct access to the hardware, it performs better.

Disadvantages

Following are disadvantages of a monolith operating system structure.

- **Crash Prone - As Kernel is responsible for all functions, if one function fails entire operating system fails.**
- **Difficult to enhance - It is very difficult to add a new service without impacting other services of a monolith operating system.**

Operating System Services

Operating systems provide a set of essential **services** to:

✅**Users** – for ease of using the system
✅**Programs** – for execution and resource management
✅**System efficiency** – to manage hardware and software resources effectively.

1️⃣ User Services

These services help **users** use the computer system effectively:

- **Program Execution:** Load and run user programs, providing an environment for execution.
- **I/O Operations:** Handle input and output operations, hiding hardware details from the user.
- **File System Manipulation:** Create, delete, read, write, and manage files and directories.
- **Communication Services:** Allow processes to exchange information, either on the same system or over a network.
- **Error Detection:** Detect and report system and program errors for smooth operation.

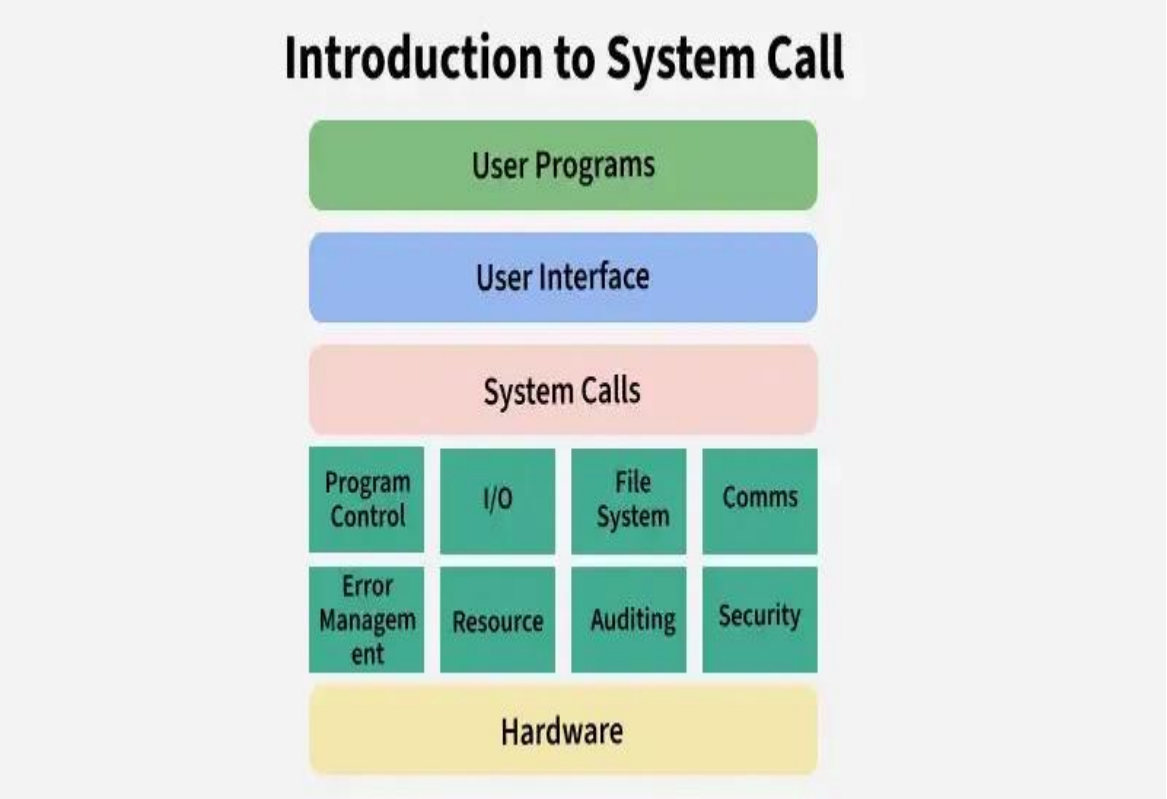
2️⃣ System Services

These services help **system management and efficiency**:

- **Resource Allocation:** Manage and allocate CPU time, memory, and I/O devices to ‘various programs and users.
- **Accounting:** Track resource usage (CPU time, memory usage, disk usage) for monitoring and billing (in multi-user systems).
- **Protection and Security:** Control access to system resources, ensuring that unauthorized users do not interfere with the system or other users.

System Calls

- **Interface between user programs and OS.**
- **Allow user-level processes to request OS services (e.g., file operations, process management).**
- **Example:87**
    - **fork() – create process**
    - **exec() – execute program**
    - **read(), write() – file operations**
    - **wait() – wait for processf**
    - **exit() – terminate process**



System Boot

- **Process of loading OS into memory after the computer is powered on.**
- **Uses bootstrap loader (bootloader) from ROM to load the OS kernel.**
- **Initializes system resources and prepares the system for user operations.**

Diagram:

[Power On]

|

[Bootstrap Loader in ROM]

|

[Load Kernel into Memory]

|

[OS Initialization]

|

[System Ready]

Types of Booting

There are two types of Booting available:

1. **Cold Booting/ Hard Booting: Cold booting is the process when our computer system moves from the shutdown state to the start by pressing the power button. The system reads the BIOS from ROM and will eventually load the Operating System.**
2. **Warm Booting/ Soft Booting: Warm booting is the process in which the computer gets restarted due to reasons like setting the configuration for newly installed software or hardware. Warm booting is called as *rebooting*.**

System Programs

- **System programs provide a convenient environment for program development and execution.**

Types of System Programs

1. **File Management**
   These system programs manage the operations related to files such as:

- **Create file**
- **Delete file**
- **Copy file**
- **Rename file**
- **Dump file**
- **List, etc.**

Status Information

These system programs are used to retrieve the information about the system such as:

- **Date**
- **Time**
- **Memory**
- **Number of users, etc.**

File Modification

These system programs are used for modifying the file stored on the hard disk or other storage devices. Other than modification these programs are also used to search some content on the file, or even to transform some content of the file.

Programming-Language Support

Some common system programs that support the programming languages like C, C++, Java, Visual Basic and Pearl come with the operating system. These system programs are:

- **Compiler**
- **Assembler**
- **Debuggers**
- **Interpreters, etc.**

Program Loading and Execution

Whenever we write a program we compile it and store it on the disk. Now when we want to execute it we have to load it to the main memory. For this, we require some system programs such as:

- **Loader**
- **Relocatable loader**
- **Linkage editor**
- **Overlay loader**

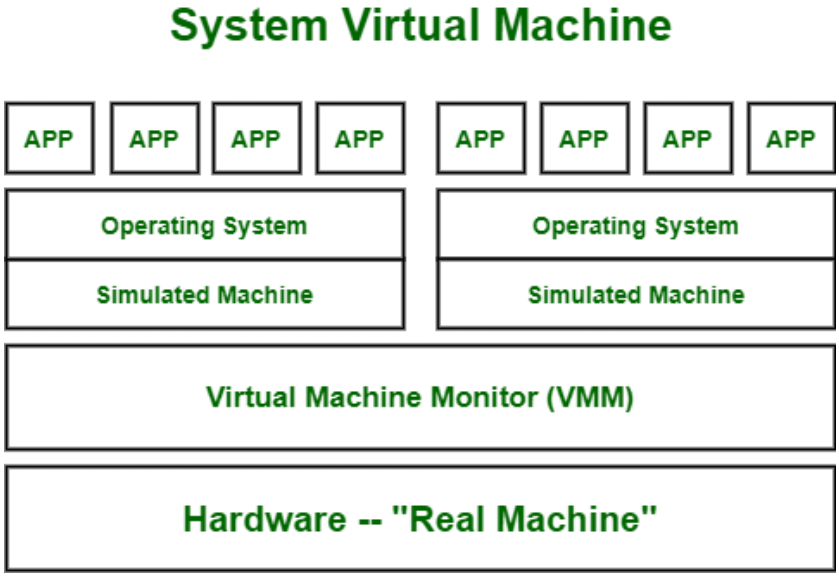
Communication

These kinds of system programs are used for connecting two communicating processes, users, and computer systems.

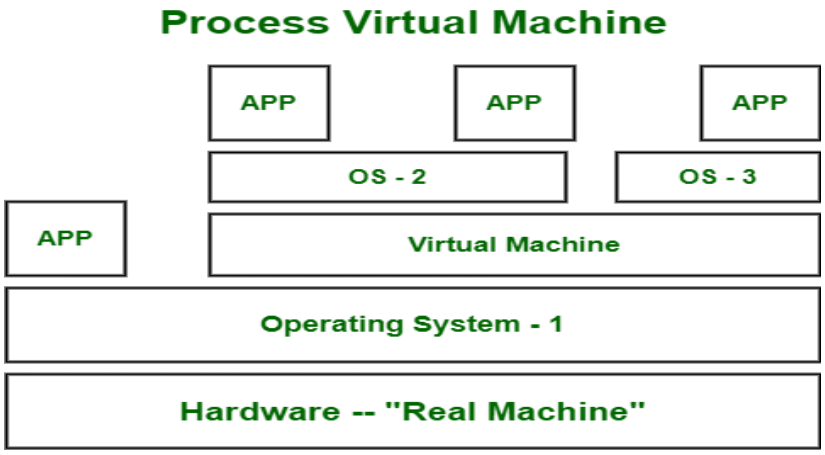
Virtual Machines

- **Virtual Machines allow multiple OS instances on the same hardware.**
- **Each VM behaves like a real computer with its own OS.**
- **Uses a Virtual Machine Monitor (VMM) to provide isolation between VMs.**

Types of Virtual Machines : You can classify virtual machines into two types:
1. System Virtual Machine: These types of virtual machines gives us complete system platform and gives the execution of the complete virtual operating system. Just like virtual box, system virtual machine is providing an environment for an OS to be installed completely. We can see in below image that our hardware of Real Machine is being distributed between two simulated operating systems by Virtual machine monitor. And then some programs, processes are going on in that distributed hardware of simulated machines separately.



2. Process Virtual Machine : While process virtual machines, unlike system virtual machine, does not provide us with the facility to install the virtual operating system completely. Rather it creates virtual environment of that OS while using some app or program and this environment will be destroyed as soon as we exit from that app. Like in below image, there are some apps running on main OS as well some virtual machines are created to run other apps. This shows that as those programs required different OS, process virtual machine provided them with that for the time being those programs are krunning. **Example -** Wine software in Linux helps to run Windows applications.



fg

Virtual Machine Language : It's type of language which can be understood by different operating systems. It is platform-independent. Just like to run any programming language (C, python, or java) we need specific compiler that actually converts that code into system understandable code (also known as byte code). The same virtual machine language works. If we want to use code that can be executed on different types of operating systems like (Windows, Linux, etc) then virtual machine language will be helpful.

Advantages:

✓Isolation

✓Resource sharing

✓Flexibility for OS development/testing

What is a Process?

Process is the execution of a program that performs the actions specified in that program. It can be defined as an execution unit where a program runs. The OS helps you to create, schedule, and terminates the processes which is used by CPU. A process created by the main process is called a child process.

Process operations can be easily controlled with the help of PCB(Process Control Block). You can consider it as the brain of the process, which contains all the crucial information related to processing like process id, priority, state, CPU registers, etc

What is Process Management?

Process management involves various tasks like creation, scheduling, termination of processes, and a dead lock. Process is a program that is under execution, which is an important part of modern-day operating systems. The OS must allocate resources that enable processes to share and exchange information. It also protects the resources of each process from other methods and allows synchronization among processes.

It is the job of OS to manage all the running processes of the system. It handles operations by performing tasks like process scheduling and such as resource allocation.

Process States

A process state is a condition of the process at a specific instant of time. It also defines the current position of the process.

There are mainly seven stages of a process which are:

- New: The new process is created when a specific program calls from secondary memory/ hard disk to primary memory/ RAM a
- Ready: In a ready state, the process should be loaded into the primary memory, which is ready for execution.
- Waiting: The process is waiting for the allocation of CPU time and other resources for execution.
- Executing: The process is an execution state.
- Blocked: It is a time interval when a process is waiting for an event like I/O operations to complete.

- Suspended: Suspended state defines the time when a process is ready for execution but has not been placed in the ready queue by OS.
- Terminated: Terminated state specifies the time when a process is terminated

Process Control Blocks

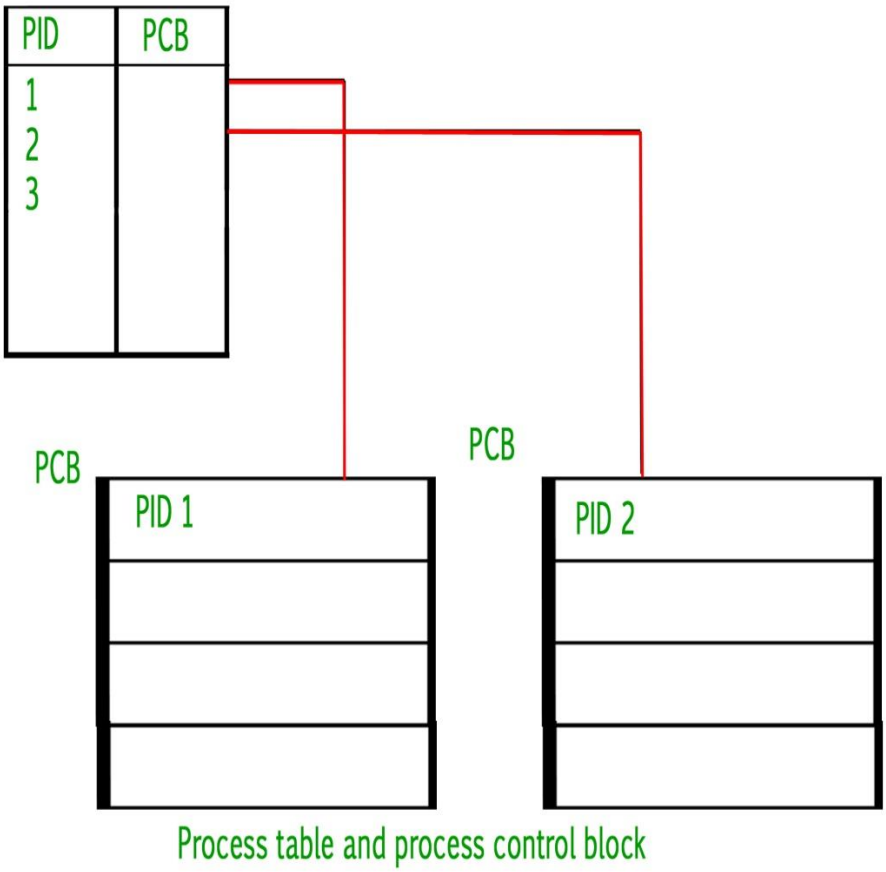
PCB stands for Process Control Block. It is a data structure that is maintained by the Operating System for every process. The PCB should be identified by an integer Process ID (PID). It helps you to store all the information required to keep track of all the running processes.

It is also accountable for storing the contents of processor registers. These are saved when the process moves from the running state and then returns back to it. The information is quickly updated in the PCB by the OS as soon as the process makes the state transition.

The diagram helps explain some of these key data items.

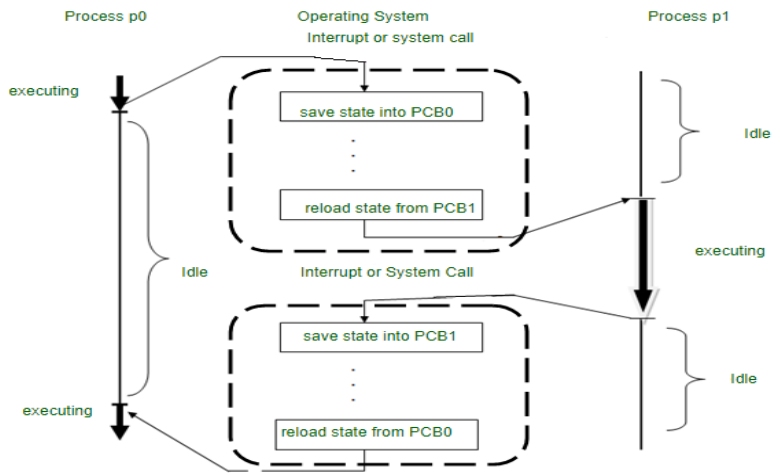
Process Control Block

- **Pointer:** It is a stack pointer that is required to be saved when the process is switched from one state to another to retain the current position of the process.
- **Process state:** It stores the respective state of the process.
- **Process number:** Every process is assigned a unique id known as process ID or PID which stores the process identifier.
- **Program counter:** Program Counter stores the counter, which contains the address of the next instruction that is to be executed for the process.
- **Register:** Registers in the PCB, it is a data structure. When a processes is running and it's time slice expires, the current value of process specific registers would be stored in the PCB and the process would be swapped out. When the process is scheduled to be run, the register values is read from the PCB and written to the CPU registers. This is the main purpose of the registers in the PCB.
- **Memory limits:** This field contains the information about memory management system used by the operating system. This may include page tables, segment tables, etc.
- **List of Open files:** This information includes the list of files opened for a process.



Working Process Context Switching

Context Switchhing in an operating system is a critical function that allows the CPU to efficiently manage multiple processes. By saving the state of a currently active process and loading the state of another, the system can handle various tasks simultaneously without losing progress. This switching mechanism ensures optimal use of the CPU, enhancing the system's ability to perform multitasking effectively.



State Diagram of Context Switching

In the context switching of two processes, the priority-based process occurs in the ready queue of the process control block. Following are the steps:

- The state of the current process must be saved for rescheduling.
- The process state contains records, credentials, and operating system-specific information stored on the PCB or switch.
- The PCB can be stored in a single layer in kernel memory or in a custom OS file.
- A handle has been added to the PCB to have the system ready to run.
- The operating system aborts the execution of the current process and selects a process from the waiting list by tuning its PCB.
- Load the PCB's program counter and continue execution in the selected process.
- Process/thread values can affect which processes are selected from the queue, this can be important.

Threads and Multithreading

A **thread** is a path that is followed during a program's execution. The majority of programs written nowadays run as a single thread. For example, a program is not capable of reading keystrokes while making drawings. These tasks cannot be executed by the program at the same time. This problem can be solved through multitasking so that two or more tasks can be executed simultaneously.

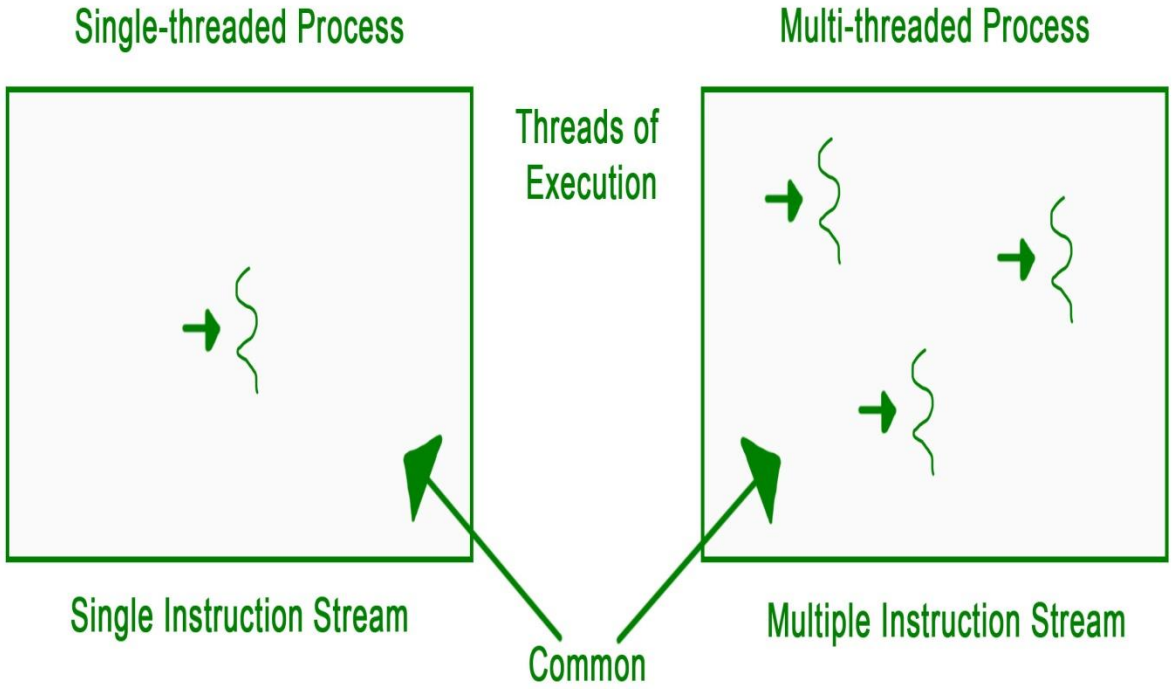
What is Multithreading?

Multithreading is a feature in operating systems that allows a program to do several tasks at the same time. Think of it like having multiple hands working together to complete different parts of a job faster. Each "hand" is called a thread, and they help make programs run more efficiently. Multithreading makes your computer work better by using its resources more effectively, leading to quicker and smoother performance for applications like web browsers, games, and many other programs you use every day.

How Does Multithreading Work?

Multithreading works by allowing a computer's processor to handle multiple tasks at the same time. Even though the processor can only do one thing at a time, it switches between different threads from various programs so quickly that it looks like everything is happening all at once. Here's how it simplifies:

- **Processor Handling** : The processor can execute only one instruction at a time, but it switches between different threads so fast that it gives the illusion of simultaneous execution.
- **Thread Synchronization** : Each thread is like a separate task within a program. They share resources and work together smoothly, ensuring programs run efficiently.
- **Efficient Execution** : Threads in a program can run independently or wait for their turn to process, making programs faster and more responsive.
- **Programming Considerations** : Programmers need to be careful about managing threads to avoid problems like conflicts or situations where threads get stuck waiting for each other.



Single Thread and Multi Thread Process

.

Operating System - Process Scheduling

Definition

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

Categories of Scheduling

There are two categories of scheduling:

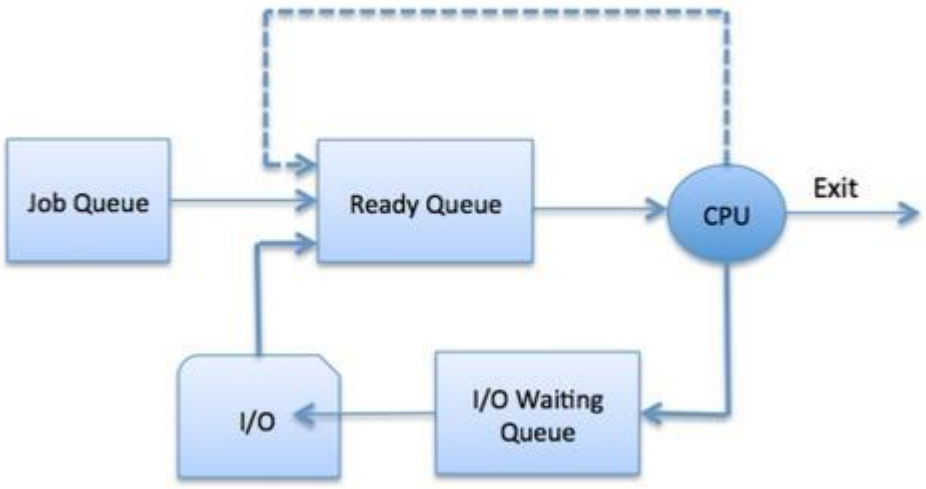
1. **Non-preemptive:** Here the resource cant be taken from a process until the process completes execution. The switching of resourcelos occurs when the running process terminates and moves to a waiting state.
2. **Preemptive:** Here the OS allocates the resources to a process for a fixed amount of time. During resource allocation, the process switches from running state to ready state or from waiting state to ready state. This switching occurs as the CPU may give priority to other processes and replace the process with higher priority with the running process.

   Process Scheduling Queues

   The OS maintains all Process Control Blocks (PCBs) in Process Scheduling Queues. The OS maintains a separate queue for each of tmhe process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

   The Operating System maintains the following important process scheduling queues −

- **Job queue** − This queue keeps all the processes in the system.
- **Ready queue** − This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
- **Device queues** − The processes which are blocked due to unavailability of an I/O device constitute this queue.
- .



Operating System Scheduling algorithms

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. There are six popular process scheduling algorithms which we are going to discuss in this chapter −

- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-Next (SJN) Scheduling
- Priority Scheduling
- Shortest Remaining Time
- Round Robin(RR) Scheduling
- Multiple-Level Queues Scheduling

These algorithms are either **non-preemptive or preemptive**. Non-preemptive algorithms are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time, whereas the preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

First Come First Serve (FCFS)

- Jobs are executed on first come, first serve basis.
- It is a non-preemptive, pre-emptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue.
- Poor in performance as average wait time is high.

Problem 1

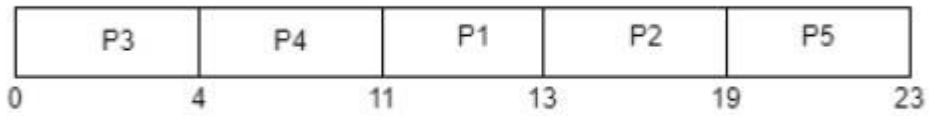
Consider the given table below and find Completion time (CT), Turn-around time (TAT), Waiting time (WT), Response time (RT), Average Turn-around time and Average Waiting time.

Process ID	Arrival time	Burst time
P1	2	2
P2	5	6
P3	0	4
P4	0	7

P5	7	4

Solution

Gantt chart



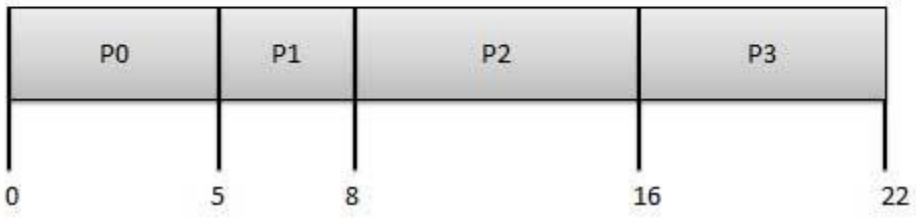
For this problem CT, TAT, WT, RT is shown in the given table

Process ID	Arrival time	Burst time	CT	TAT=CT-AT	WT=TAT-BT	RT
P1	2	2	13	13-2= 11	11-2= 9	9
P2	5	6	19	19-5= 14	14-6= 8	8
P3	0	4	4	4-0= 4	4-4= 0	0
P4	0	7	11	11-0= 11	11-7= 4	4
P5	7	4	23	23-7= 16	16-4= 12	12

Average Waiting time = (9+8+0+4+12)/5 = 33/5 = 6.6 time unit (time unit can be considered as milliseconds)

Average Turn-around time = (11+14+4+11+16)/5 = 56/5 = 11.2 time unit (time unit can be considered as milliseconds)

Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	8
P3	3	6	16



Wait time of each process is as follows –

Process	**Wait Time : Service Time - Arrival Time**
P0	0 - 0 = 0
P1	5 - 1 = 4
P2	8 - 2 = 6
P3	16 - 3 = 13

Average Wait Time: (0+4+6+13) / 4 = 5.75

SJF (SHORTEST JOB FIRST) Scheduling

In the Shortest Job First scheduling algorithm, the processes are scheduled in ascending order of their CPU burst times, i.e. the CPU is allocated to the process with the shortest execution time.

Examples of Non-Preemptive SJF Algorithm

Example 1

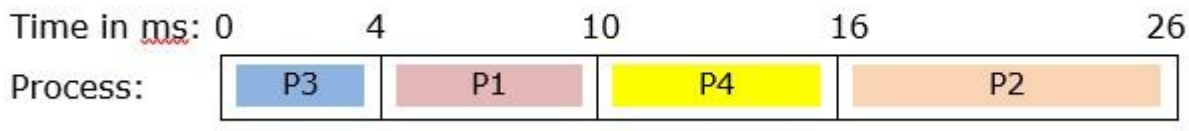
Suppose that we have a set of four processes that have arrived at the same time in the order P1, P2, P3 and P4. The burst time in milliseconds of each process is given by the following table −

Process	CPU Burst Time in ms
P1	6
P2	10
P3	4
P4	6

Let us draw the GANTT chart and find the average turnaround time and average waiting time using non-preemptive SJF algorithm.

GANTT Chart for the set of processes using SJF

Process P3 has the shortest burst time and so it executes first. Then we find that P1 and P4 have equal burst time of 6ms. Since P1 arrived before, CPU is allocated to P1 and then to P4. Finally P2 executes. Thus the order of execution is P3, P1, P4, P2 and is given by the following GANTT chart −



Let us compute the average turnaround time and average waiting time from the above chart.

Average Turnaround Time

=Sum of Turnaround Time of each Process / Number of Processes

= (TAT_{P1} + TAT_{P2} + TAT_{P3} + TAT_{P4}) / 4

= (10 + 26 + 4 + 16) / 4 = 14 ms

Average Waiting Time

= Sum of Waiting Time of Each Process / Number of processes

= (WT_{P1} + WT_{P2} + WT_{P3} + WT_{P4}) / 4

= (4 + 16 + 0 + 10) / 4 = 7.5 ms

Example 2

In the previous example, we had assumed that all the processes had arrived at the same time, a situation which is practically impossible. Here, we consider circumstance when the processes

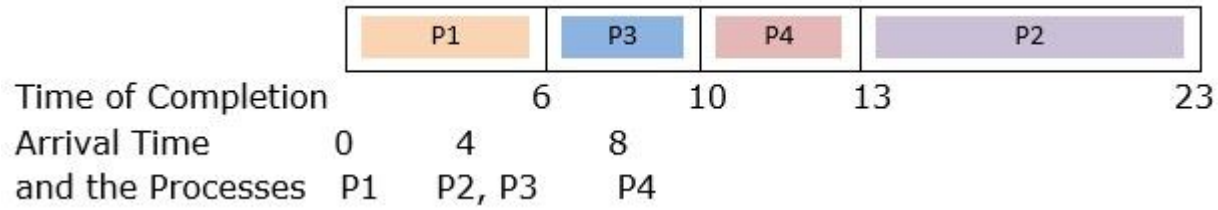
arrive at different times. Suppose we have set of four processes whose arrival times and CPU burst times are as follows −

Process	Arrival Time	CPU Burst Time
P1	0	6
P2	4	10
P3	4	4
P4	8	3

Let us draw the GANTT chart and find the average turnaround time and average waiting time using non-preemptive SJF algorithm.

GANTT Chart

While drawing the GANTT chart, we will consider which processes have arrived in the system when the scheduler is invoked. At time 0ms, only P1 is there and so it is assigned to CPU. P1 completes execution at 6ms and at that time P2 and P3 have arrived, but not P4. P3 is assigned to CPU since it has the shortest burst time among current processes. P3 completes execution at time 10ms. By that time P4 has arrived and so SJF algorithm is run on the processes P2 and P4. Hence, we find that the order of execution is P1, P3, P4, P2 as shown in the following GANTT chart −



Let us calculate the turnaround time of each process and hence the average.

Turnaround Time of a process = Completion Time Arrival Time

$TAT_{P1} = CT_{P1} - AT_{P1} = 6 - 0 = 6$ ms

$TAT_{P2} = CT_{P2} - AT_{P2} = 23 - 4 = 19$ ms

$TAT_{P3} = CT_{P3} - AT_{P3} = 10 - 4 = 6$ ms

$TAT_{P4} = CT_{P4} - AT_{P4} = 13 - 8 = 5$ ms

Average Turnaround Time

=Sum of Turnaround Time of each Process/ Number of Processes

= (6 + 19+ 6 + 5) / 4 = 9 ms

The waiting time is given by the time that each process waits in the ready queue. For a non-preemptive scheduling algorithm, waiting time of each process can be simply calculated as −

Waiting Time of any process = Time of admission to CPU Arrival Time

WT_{P1} = 0 - 0 = 0 ms

WT_{P2} = 13 - 4 = 9 ms

WT_{P3} = 6 - 4 = 2 ms

WT_{P4} = 10 - 8 = 2 ms

Average Waiting Time

= Sum of Waiting Time of Each Process/ Number of processes

= (WT_{P1} + WT_{P2} + WT_{P3} + WT_{P4}) / 4 = (0 + 9 + 2 + 2) / 4 = 3.25 ms

Priority Based Scheduling

- Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.
- Each process is assigned a priority. Process with highest priority is to be executed first and so on.
- Processes with same priority are executed on first come first served basis.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Round Robin Scheduling

- Round Robin is the preemptive process scheduling algorithm.
- Each process is provided a fix time to execute, it is called a **quantum**.
- Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
- Context switching is used to save states of preempted processes.

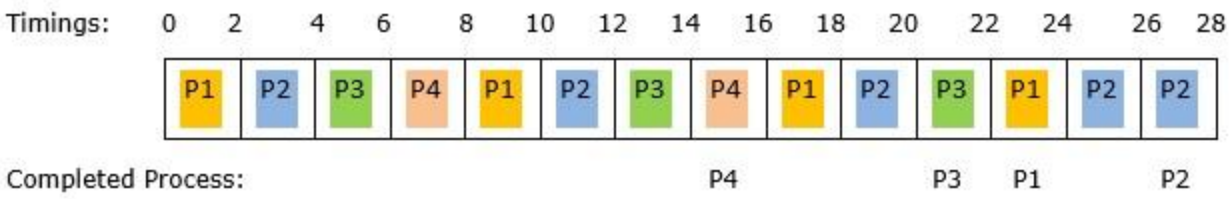
Example of Round Robin Scheduling

Let us consider a system that has four processes which have arrived at the same time in the order P1, P2, P3 and P4. The burst time in milliseconds of each process is given by the following table −

Process	CPU Burst Times in ms
P1	8
P2	10
P3	6
P4	4

Let us consider time quantum of 2ms and perform RR scheduling on this. We will draw GANTT chart and find the average turnaround time and average waiting time.

GANTT Chart with time quantum of 2ms



Average Turnaround Time

Average TAT = Sum of Turnaround Time of each Process / Number of Processes

$=(TAT_{P1}+TAT_{P2}+TAT_{P3}+TAT_{P4})/4$

= (24 + 28 + 22+ 16) / 4 = 22.5 ms

In order to calculate the waiting time of each process, we multiply the time quantum with the number of time slices the process was waiting in the ready queue.

Average Waiting Time

Average WT = Sum of Waiting Time of Each Process / Number of processes

$=(WT_{P1}+WT_{P2}+WT_{P3}+WT_{P4})/4$

= (8*2 + 9*2+ 8*2+ 6*2) / 4 = 15.5 ms

Multiple-Level Queues Scheduling

Multiple-level queues are not an independent scheduling algorithm. They make use of other existing algorithms to group and schedule jobs with common characteristics.

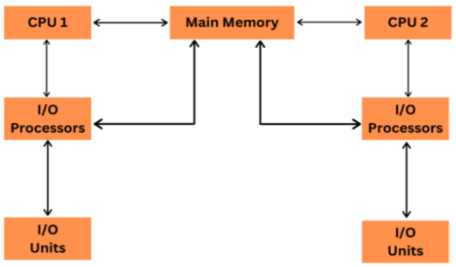
- Multiple queues are maintained for processes with common characteristics.
- Each queue can have its own scheduling algorithms.
- Priorities are assigned to each queue.

For example, CPU-bound jobs can be scheduled in one queue and all I/O-bound jobs in another queue. The Process Scheduler then alternately selects jobs from each queue and assigns them to the CPU based on the algorithm assigned to the queue.

Multiple-Processor Scheduling

The goal of multiple processor scheduling, also known as multiprocessor scheduling, is to create a system's scheduling function that utilizes several processors. In multiprocessor scheduling, multiple CPUs split the workload (load sharing) to enable concurrent execution of multiple processes. In comparison to single-processor scheduling, multiprocessor scheduling is generally more complicated. There are many identical processors in the multiprocessor scheduling system, allowing us to perform any process at any moment.

.



Approaches to Multiple Processor Scheduling

There are two different architectures utilized in multiprocessor systems: ?

Symmetric Multiprocessing

In an SMP system, each processor is comparable and has the same access to memory and I/O resources. The CPUs are not connected in a master-slave fashion, and they all use the same memory and I/O subsystems. This suggests that every memory location and I/O device are accessible to every processor without restriction. An operating system manages the task distribution among the processors in an SMP system, allowing every operation to be completed by any processor.

Asymmetric Multiprocessing

In the AMP asymmetric architecture, one processor, known as the master processor, has complete access to all of the system's resources, particularly memory and I/O devices. The master processor is in charge of allocating tasks to the other processors, also known as slave processors. Every slave processor is responsible for doing a certain set of tasks that the master processing has assigned to it. The master processor receives tasks from the operating system, which the master processor then distributes to the subordinate processors.
Types of Multiprocessor Scheduling Algorithms

Operating systems utilize a range of multiprocessor scheduling algorithms. Among the most typical types are ?

Round-Robin Scheduling ? The round-robin scheduling algorithm allocates a time quantum to each CPU and configures processes to run in a round-robin fashion on each processor. Since it ensures that each process gets an equivalent amount of CPU time, this strategy might be useful in systems wherein all programs have the same priority.

Priority Scheduling ? Processes are given levels of priority in this method, and those with greater priorities are scheduled to run first. This technique might be helpful in systems where some jobs, like real-time tasks, call for a higher priority.

Scheduling with the shortest job first (SJF) ? This algorithm schedules tasks according to how long they should take to complete. It is planned for the shortest work to run first, then the next smallest job, and so on. This technique can be helpful in systems with lots of quick processes since it can shorten the typical response time.

Fair-share scheduling ? In this technique, the number of processors and the priority of each process determine how much time is allotted to each. As it ensures that each process receives a fair share of processing time, this technique might be helpful in systems with a mix of long and short processes.

Earliest deadline first (EDF) scheduling ? Each process in this algorithm is given a deadline, and the process with the earliest deadline is the one that will execute first. In systems with real-time activities that have stringent deadlines, this approach can be helpful.

Scheduling using a multilevel feedback queue (MLFQ) ? Using a multilayer feedback queue (MLFQ), processes are given a range of priority levels and are able to move up or down the priority levels based on their behavior. This strategy might be useful in systems with a mix of short and long processes.

Process Synchronization and Deadlocks

1 Race Conditions

- A **race condition** occurs when **two or more processes access shared resources simultaneously, and the outcome depends on the order of execution**.
- It can lead to **inconsistent data or unexpected behavior** in concurrent programs.
- For example, two processes incrementing a shared counter without proper control may overwrite each other's values.
- Race conditions are difficult to detect and debug as they depend on the CPU scheduling.
- To avoid race conditions, **synchronization techniques (locks, semaphores) are used to control access to shared resources**.

L;;l

---

2 Critical Section

- A **critical section** is a part of the program where **a process accesses shared resources (like shared variables, files)**.
- To prevent data inconsistency, **only one process should execute in its critical section at a time**.
- It requires:
  - **M[utual exclusion:** Only one process in the critical section at a time.
  - **Progress:** If no process is in the critical section, others should be allowed to enter.
  - **[-Bounded waiting:** There should be a limit on how long a process waits to enter.
- Managing critical sections is essential for **process synchronization in concurrent systems**.

---

3 Mutual Exclusion

- **Mutual Exclusion** ensures **only one process can access the critical section at a time**.
- It prevents **race conditions** and maintains **data consistency**.
- Methods to achieve mutual exclusion include:
  - **Locks (mutexes)**
  - **Peterson’s solution**
  - **Semaphores**
  - **Monitors**
- Mutual exclusion is the foundation of **process synchronization in operating systems**.

4 Peterson's Solution

- **Peterson's solution** is a classical software-based solution for **achieving mutual exclusion between two processes**.
- Uses two shared variables:
  - `flag[2]`: indicates if a process wants to enter the critical section.
  - `turn`: indicates whose turn it is.
- Working:
  - Each process sets its `flag[i] = true` and `turn = j` before entering the critical section.
  - The process enters the critical section **only if `flag[j] == false` or `turn == i`.**
- Satisfies:
  - **Mutual Exclusion**
  - **Progress**
  - **;lBounded Waiting**

How it works:

Peterson's Solution utilizes two shared variables:

- `flag[2]`:

  A boolean array where `flag[i]` indicates if process `i` is interested in entering the critical section.

- `turn`:

  An integer variable indicating which process has priority to enter the critical section if both are interested.

Algorithm for Process `i` (the other process is `j`):

```
do {
    flag[i] = true; // Declare interest in entering
    turn = j;       // Give priority to the other process
    while (flag[j] && turn == j); // Wait if the other process is interested
AND it's their turn

    // Critical Section
    // Access shared resources here

    flag[i] = false; // Indicate no longer interested
    // Remainder Section
} while (true);
```

-

I

5 Semaphores

Semaphores are synchronization tools used to control access to shared resources in concurrent systems.

The process of using Semaphores provides two operations:

- **wait (P):** The wait operation decrements the value of the semaphore
- **signal (V):** The signal operation increments the value of the semaphore.

Uses of Semaphores

- **Mutual Exclusion** : Semaphore ensures that only one process accesses a shared resource at a time.
- **Process Synchronization** : Semaphore coordinates the execution order of multiple processes.
- **Resource Management** : Limits access to a finite set of resources, like printers, devices, etc.
- **';Reader-Writer Problem** : Allows multiple readers but restricts the writers until no reader is present.
- **Avoiding Deadlocks** : Prevents deadlocks by controlling the order of allocation of resources.

6 Monitors (Classical IP)

- A **Monitor** is a **high-level synchronization construct that provides a mechanism to achieve mutual exclusion and condition synchronization**.
- Encapsulates:
  - Shared variables.
  - Procedures to access those variables.
  - Synchronization mechanisms.
- Only **one process can execute a monitor procedure at a time**, ensuring **mutual exclusion automatically**.
- Provides **condition variables with `wait()` and `signal()` operations** for process synchronization.
- Simplifies **complex process synchronization problems** like **bounded-buffer and readers-writers**.

If you would like, I can also prepare **clear diagrams (memory model, critical section representation, semaphore flow) for your notes or slides** to aid your revision and presentations. Let me know!

Readers-Writers Problem

The Readers-Writers Problem is a classic synchronization issue in operating systems that involves managing access to shared data by multiple threads or processes. The problem addresses the scenario where:

- **Readers**: Multiple readers can access the shared data simultaneously without causing any issues because they are only reading and not modifying the data.
- **Writers**: Only one writer can access the shared data at a time to ensure data integrity, as writers modify the data, and concurrent modifications could lead to data corruption or inconsistencies.

Solution of the Reader-Writer Problem

There are two fundamental solutions to the Readers-Writers problem:

- **Readers Preference:** In this solution, readers are given preference over writers. That means that till readers are reading, writers will have to wait. The Writers can access the resource only when no reader is accessing it.
- **Writer's Preference:** Preference is given to the writers. It simply means that, after arrival, the writers can go ahead with their operations; though perhaps there are readers currently accessing the resource.

Solution When Reader Has The Priority Over Writer

Here priority means, no reader should wait if the share is currently open for reading. There are four types of cases that could happen here.

Case	Process 1	Process 2	Allowed/Not Allowed
Case 1	Writing	Writing	Not Allowed
Case 2	Writing	Reading	Not Allowed
Case 3	Reading	Writing	Not Allowed
Case 4	Reading	Reading	Allowed

Dining Philosophers Problem

Problem Statement

The **Dining Philosopher Problem** involves 'n' philosophers sitting around a circular table. Each philosopher alternates between two states: **thinking** and **eating**. To eat, a philosopher needs two chopsticks, one on their left and one on their right. However, the number of chopsticks is equal to the number of philosophers, and each chopstick is shared between two neighboring philosophers.

The standard problem considers the value of 'n' as 5 i.e. we deal with 5 Philosophers sitting around a circular table.

Constraints and Conditions for the Problem

- Every Philosopher needs two forks to eat.
- Every Philosopher may pick up the forks on the left or right but only one fork at once.
- Philosophers only eat when they have two forks. We have to design such a protocol i.e. pre and post protocol which ensures that a philosopher only eats if he or she has two forks.
- Each fork is either clean or dirty.

Solution

The correctness properties it needs to satisfy are:

- **Mutual Exclusion Principle:** No two Philosophers can have the two forks simultaneously.
- **Free from Deadlock:** Each philosopher can get the chance to eat in a certain finite time.
- **Free from Starvation:** When few Philosophers are waiting then one gets a chance to eat in a while.
- **No strict Alternation**
- **Proper utilization of time**

Algorithm

```
loop forever
    p1: think
    p2: preprotocol
    p3: eat
    p4: postprotocol
```

klDEADLOCKS:

Definition

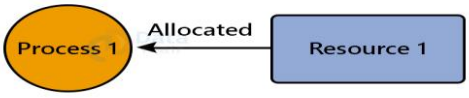
- A **deadlock** is a situation in which **two or more processes are unable to proceed because each is waiting for the other to release resources.**
- It causes **processes to be blocked indefinitely**, leading to system inefficiency.
- Example: Process A holds Resource X and waits for Resource Y held by Process B, while Process B waits for Resource X.

---

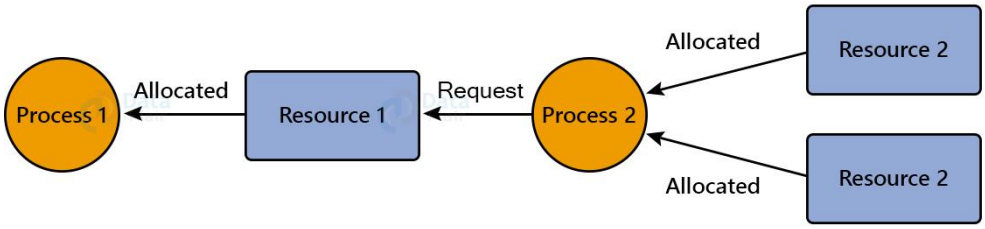
2 Characteristics (Coffman Conditions)

Deadlock can occur if **all the following four conditions hold simultaneously**:

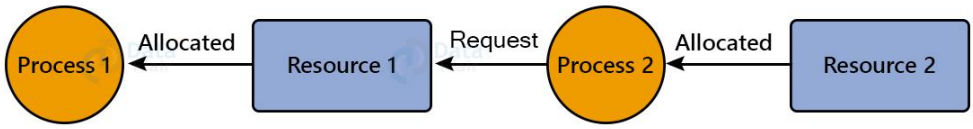
1 **Mutual Exclusion**: Mutex (Mutual Exclusion) is a type of binary semaphore that helps control access to the shared resources. It also has a priority inheritance mechanism that avoids extended priority inversion problems and allows tasks with higher priority to execute first. Shared resources don't llead to deadlocks, but resources, like printers and tape drives, need exclusive access..



2 **Hold and Wait**: A process holding at least one resource is waiting to acquire additional resources held by other processes.

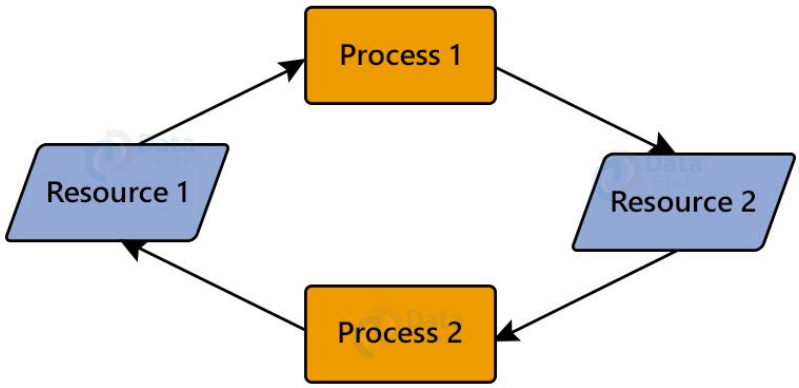


3 **No Preemption**: Resources cannot be forcibly taken away; they are released only by the process voluntarily.



;

4 **Circular Wait**: A circular chain of two or more processes exists, each waiting for a resource held by the next process in the chain.



Deadlock in Operating System

Deadlock Prevention

- **Prevent one of the Coffman conditions to avoid deadlock.**

Strategies:

- **Mutual Exclusion**: Make resources sharable if possible.
- **Hold and Wait**: Require processes to request all resources at once.
- **No Preemption**: If a process holding resources requests another unavailable resource, preempt the currently held resources.
- **Circular Wait**: Impose a **resource ordering** and require processes to request resources in increasing order.

4 Deadlock Avoidance

- Requires **prior knowledge of resource usage** and avoids deadlock dynamically.

Banker's Algorithm:

- Used for multiple instances of resources.
- The system checks the **safe state** before resource allocation.
- If granting a request leads to an unsafe state, it is denied.

5️⃣ Deadlock Detection

- Allows deadlocks to occur but **detects and resolves them.**

Detection Methods:

- For **single-instance resources**: Use **wait-for graphs** to detect cycles.
- For **multiple instances of resources**: Use a resource allocation graph with detection algorithms.

If a cycle is found, a deadlock exists.

6️⃣ Deadlock Recovery

Once deadlock is detected, the system must recover:

Methods:
✅**Process Termination**:

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock is resolved.

UNIT-II

UNIT - II: Memory, File, and Storage Management

Memory Management: Logical vs. Physical Address Mapping, Contiguous Memory Allocation, Internal and External Fragmentation, Compaction, Paging and Page Tables, Segmentation, Virtual Memory: Demand Paging, Page Faults, Page Replacement Algorithms, Thrashing and Working Set Model,

File System Management: File Concepts, Access Methods, File Types and Operations, Directory Structure, File System Structure, Allocation Methods, Free-Space Management, Directory Implementation.

Storage Management: Mass Storage: Disk Structure, RAID Levels, Disk Scheduling Algorithms, Swap Space Management, Stable Storage, Tertiary Storage Structure.

Memory Management:

Memory is a hardware component that stores data, instructions and information temporarily or permanently for processing. It consists of an array of bytes or words, each with a unique address.

- **Memory holds both input data and program instructions needed for the CPU to execute tasks.**
- **Memory works closely with the CPU to provide quick access to data being used.**
- **Memory management ensures efficient use of memory and supports multiprogramming.**

Logical vs. Physical Address Mapping

In computers, an address is used to identify a location in the **computer memory**. In **operating systems**, there are two types of addresses, namely, **logical address** and **physical address**. A logical address is the virtual address that is generated by the **CPU**. A user can view the logical address of a computer program. On the other hand, a physical address is one that represents a location in the computer memory. A user cannot view the physical address of a program.
Read this article to find out more about logical and physical address and how they are different from each other.

What is a Logical Address?

The **logical address** is a virtual address created by the CPU of the computer system. The logical address of a program is generated when the program is running. A group of several logical address is referred to a **logical address space**. The logical address is basically used as a reference to access the physical memory locations.
In computer systems, a hardware device named **memory management unit** (MMU) is used to map the logical address to its corresponding physical address. However, the logical address of a program is visible to the computer user.

What is a Physical Address?

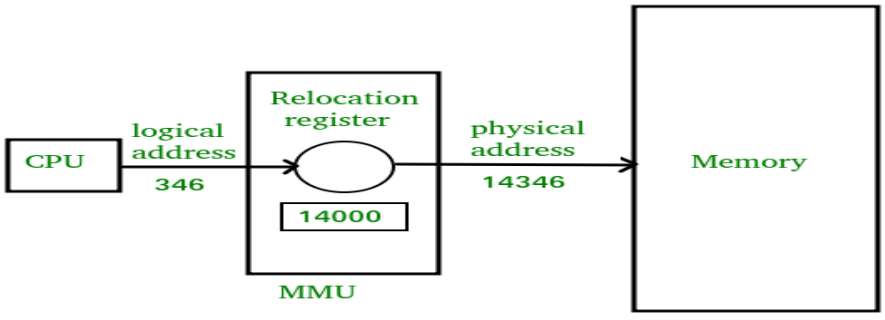
The **physical address** of a computer program is one that represents a location in the memory unit of the computer. The physical address is not visible to the computer user. The MMU of the system generates the physical address for the corresponding logical address.
The physical address is accessed through the corresponding logical address because a user cannot directly access the physical address. For running a computer program, it requires a physical memory space.

Therefore, the logical address has to be mapped with the physical address before the execution of the program.

Difference between Logical and Physical Address in Operating System

The following table highlights all the major differences between logical and physical address in operating system ?

S. No.	Logical Address	Physical Address
1.	This address is generated by the CPU.	This address is a location in the memory unit.
2.	The address space consists of the set of all logical addresses.	This address is a set of all physical addresses that are mapped to the corresponding logical addresses.
3.	These addresses are generated by CPU with reference to a specific program.	It is computed using Memory Management Unit (MMU).
4.	The user has the ability to view the logical address of a program.	The user can't view the physical address of program directly.
5.	The user can use the logical address in order to access the physical address.	The user can indirectly access the physical address.

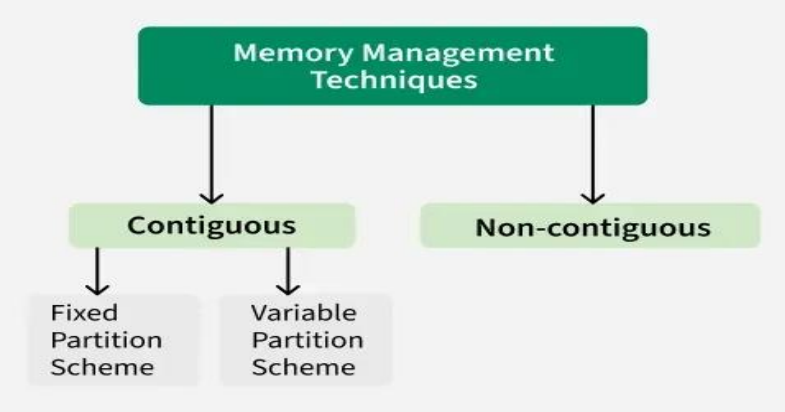


Contiguous Memory Allocation

Contiguous memory allocation is a memory allocation strategy. As the name implies, we utilize this technique to assign contiguous blocks of memory to each task. Thus, whenever a process asks to access the main memory, we allocate a continuous segment from the empty region to the process based on its size. In this technique, memory is allotted in a continuous way to the processes. Contiguous Memory Management has two types:

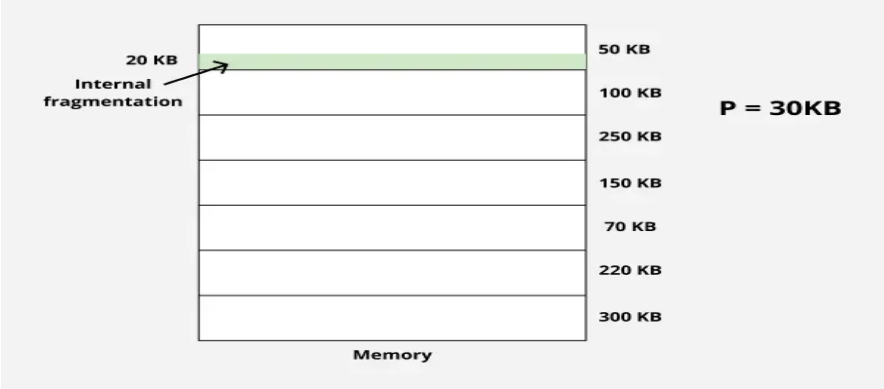
- Fixed (or Static) Partition

UNIT-II

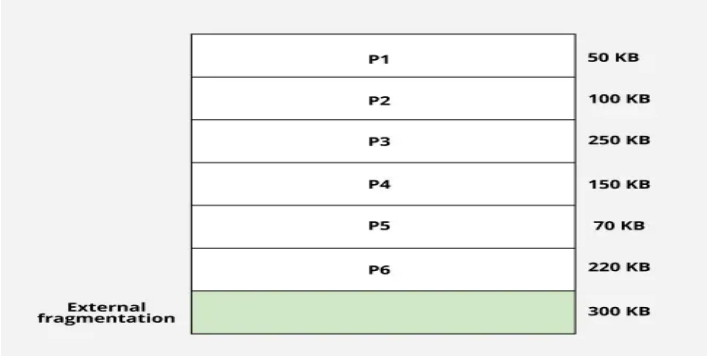


Types of Contiguous Allocation:

- **Fixed Partitioning**: In the fixed partition scheme, memory is divided into fixed number of partitions. Fixed means number of partitions are fixed in the memory. In the fixed partition, in every partition only one process will be accommodated. Degree of multi-programming is restricted by number of partitions in the memory. Maximum size of the process is restricted by maximum size of the partition. Every partition is associated with the limit registers.



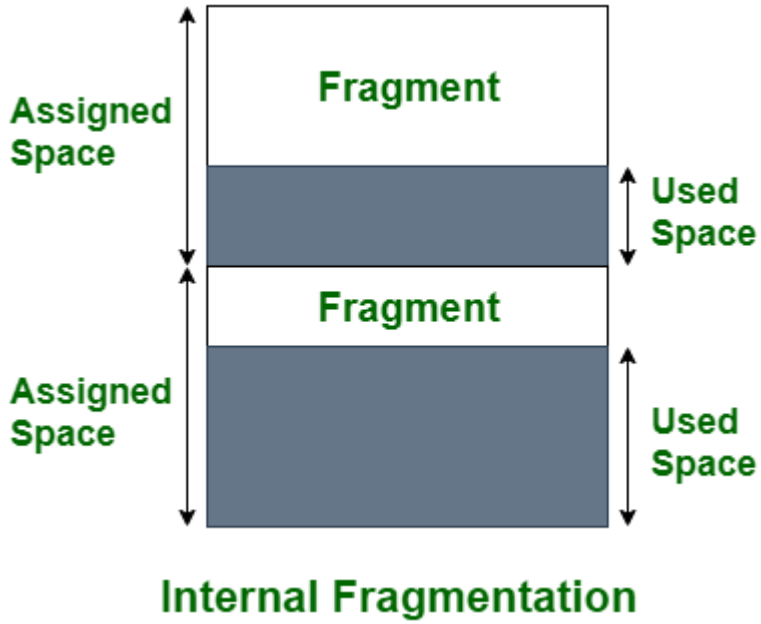
Variable Partitioning: In the variable partition scheme, initially memory will be single continuous free block. Whenever the request by the process arrives, accordingly partition will be made in the memory. If the smaller processes keep on coming then the larger partitions will be made into smaller partitions..



UNIT-II

What is Internal Fragmentation?

Internal fragmentation happens when the memory is split into mounted-sized blocks. Whenever a method is requested for the memory, the mounted-sized block is allotted to the method. In the case where the memory allotted to the method is somewhat larger than the memory requested, then the difference between allotted and requested memory is called internal fragmentation. We fixed the sizes of the memory blocks, which has caused this issue. If we use dynamic partitioning to allot space to the process, this issue can be solved.

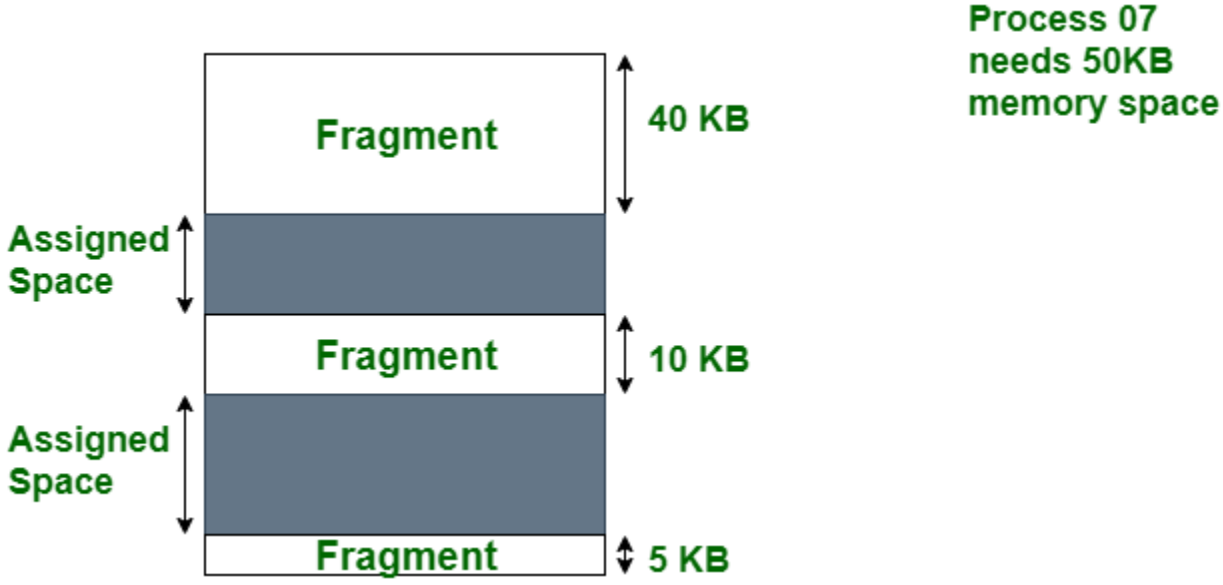


Internal Fragmentation

The above diagram clearly shows the internal fragmentation because the difference between memory allocated and required space or memory is called Internal fragmentation.

What is External Fragmentation?

External fragmentation happens when there's a sufficient quantity of area within the memory to satisfy the memory request of a method. However, the process's memory request cannot be fulfilled because the memory offered is in a non-contiguous manner. Whether you apply a first-fit or best-fit memory allocation strategy it'll cause external fragmentation.



External Fragmentation

In the above diagram, we can see that, there is enough space (55 KB) to run a process-07 (required 50 KB) but the memory (fragment) is not contiguous. Here, we use compaction, paging, or segmentation to use the free space ‘to run a process.

Compaction

Compaction is a technique to collect all the free memory present in the form of fragments into one large chunk of free memory, which can be used to run other processes.
It does that by moving all the processes towards one end of the memory and all the available free space towards the other end of the memory so that it becomes contiguous.
It is not always easy to do compaction. Compaction can be done only when the relocation is dynamic and done at execution time. Compaction can not be done when relocation is static and is performed at load time or assembly time.

Before Compaction

Before compaction, the main memory has some free space between occupied space. This condition is known as external fragmentation . Due to less free space between occupied spaces, large processes cannot be loaded into them.

Main Memory
Occupied space
Free space
Occupied space
Occupied space
Free space

After Compaction

After compaction, all the occupied space has been moved up and the free space at the bottom. This makes the space contiguous and removes external fragmentation. Processes with large memory requirements can be now loaded into the main memory.

Main Memory
Occupied space
Occupied space
Occupied space
Free space
Free space

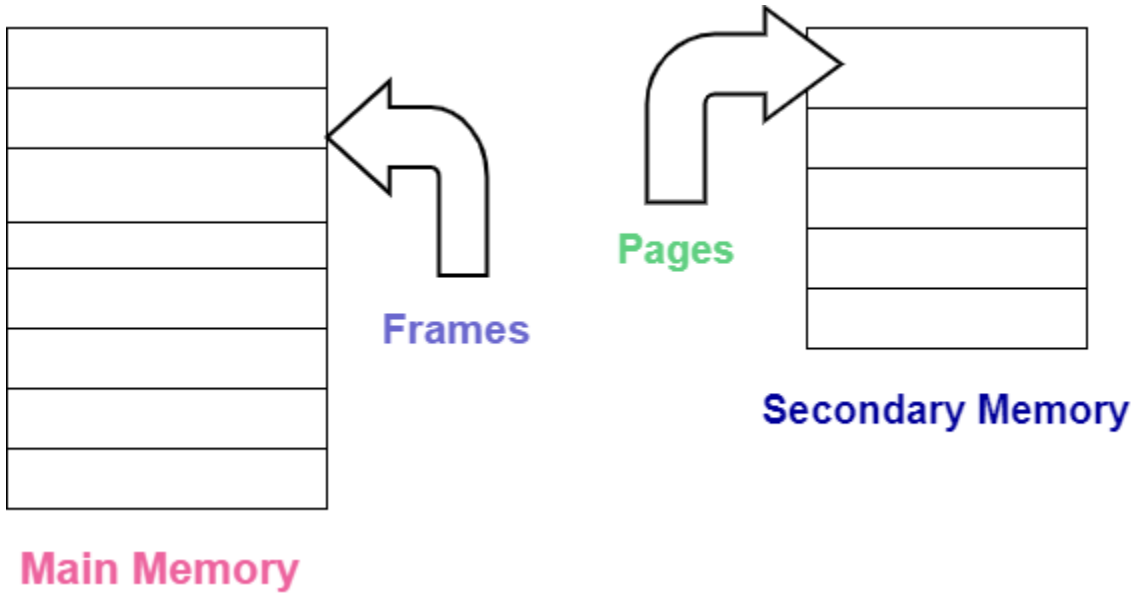
UNIT-II

Paging and Page Tables:

The paging technique divides the physical memory(main memory) into fixed-size blocks that are known as **Frames** and also divide the **logical memory(secondary memory) into blocks of the same size** that are known as **Pages.**

This technique keeps the track of all the free frames.

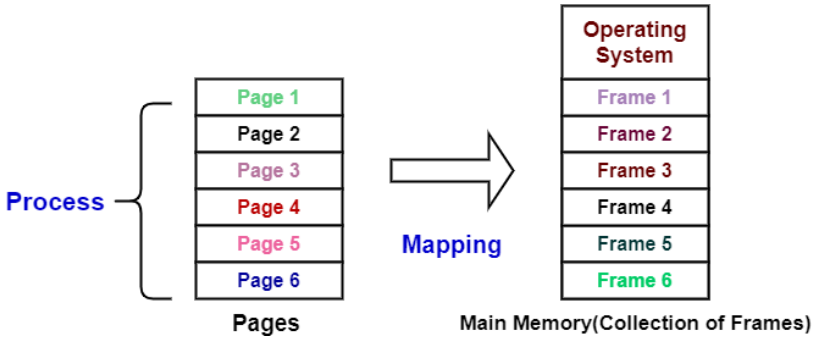
The Frame has the same size as that of a Page. A frame is basically a place where a (logical) page can be (physically) placed.



Each process is mainly divided into parts where the size of each part is the same as the page size.

There is a possibility that the size of the last part may be less than the page size.

- Pages of a process are brought into the main memory only when there is a requirement otherwise they reside in the secondary storage.
- One page of a process is mainly stored in one of the frames of the memory. Also, the pages can be stored at different locations of the memory but always the main priority is to find contiguous frames.



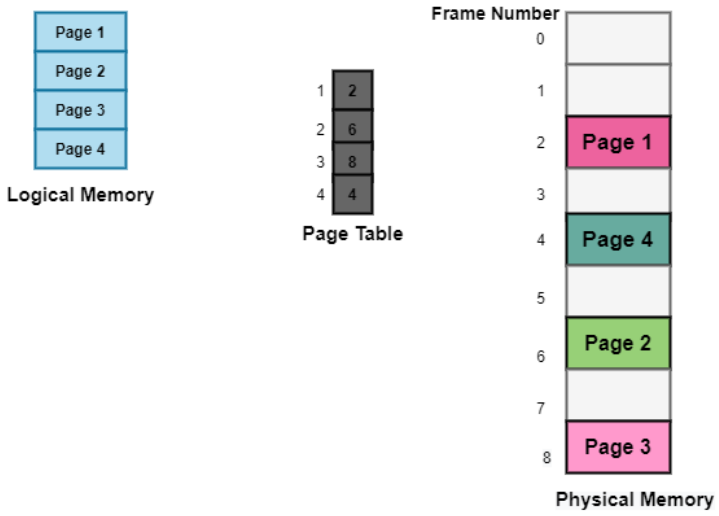
UNIT-II

PAGE TABLES:

The data structure that is used by the virtual memory system in the operating system of a computer in ordl.er to store the mapping between physical and logical addresses is commonly known as **Page Table**.

As we had already told you that the logical address that is generated by the CPU is translated into the physical address with the help of the page table.

- Thus page table mainly provides the corresponding frame number (base address of the frame) where that page is stored in the main memory.



Segmentation:

Segmentation is another way of dividing the addressable memory. It is another scheme of memory management and it generally supports the user view of memory. The Logical address space is basically the c.ollection of segments. Each segment has a name and a length.

Basically, a process is divided into segments. Like paging, segmentation divides or segments the memory. But there is a difference and that is while the **paging** divides the memory into a **fixed size** and on the other hand, segmentation divides the **memory into variable segments** these are then loaded into logical memory space.

A Program is basically a collection of segments. And a segment is a logical unit such as:

- main program
- procedure
- function
- method
- object
- local variable and global variables.
- symbol table
- common block
- stack

UNIT-II

Types of Segmentation

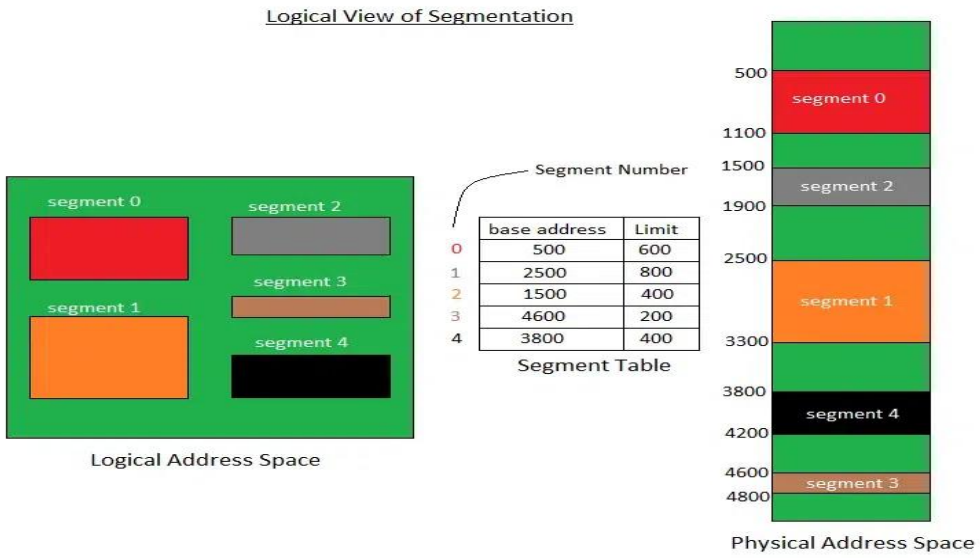
Given below are the types of Segmentation:

- **Virtual Memory Segmentation**
  With this type of segmentation, each process is segmented into n divisions and the most important thing is they are not segmented all at once.
- **Simple Segmentation**
  With the help of this type, each process is segmented into n divisions and they are all together segmented at once exactly but at the runtime and can be non-contiguous (that is they may be scattered in the memory).

What is Segment Table?

It maps a two-dimensional Logical address into a one-dimensional Physical address. It's each table entry has:

- **Base Address:** It contains the starting physical address where the segments reside in memory.
- **Segment Limit:** Also known as segment offset. It specifies the length of the segment.



virtual Memory in Operating System

Virtual memory is a memory management technique used by operating systems to give the appearance of a large, continuous block of memory to applications, even if the physical memory (RAM) is limited. It allows larger applications to run on systems with less RAM.

Objectives of Virtual Memory

- To support multiprogramming , it allows more than one program to run at the same time.
- A program doesn't need to be fully loaded in memory to run. Only the needed parts are loaded.
- Programs can be bigger than the physical memory available in the system.

UNIT-II

- Virtual memory creates the illusion of a large memory, even if the actual memory (RAM) is small.
- It uses both RAM and disk storage to manage memory, loading only parts of programs into RAM as needed.
- This allows the system to run more programs at once and manage memory more efficiently.

Demand Paging, Page Faults, Page Replacement Algorithms, Thrashing and Working Set Model,

Demand Paging:

Demand paging is a memory management scheme used in operating systems to improve memory usage and system performance. Let's understand demand paging with real life example Imagine you are reading a very thick book, but you don't want to carry the entire book around because it's too heavy. Instead, you decide to only bring the pages you need as you read through the book. When you finish with one page, you can put it away and grab the next page you need.

In a computer system, the book represents the entire program, and the pages are parts of the program called "pages" of memory. Demand paging works similarly: instead of loading the whole program into the computer's memory at once (which can be very large and take up a lot of space), the operating system only loads the necessary parts (pages) of the program when they are needed.

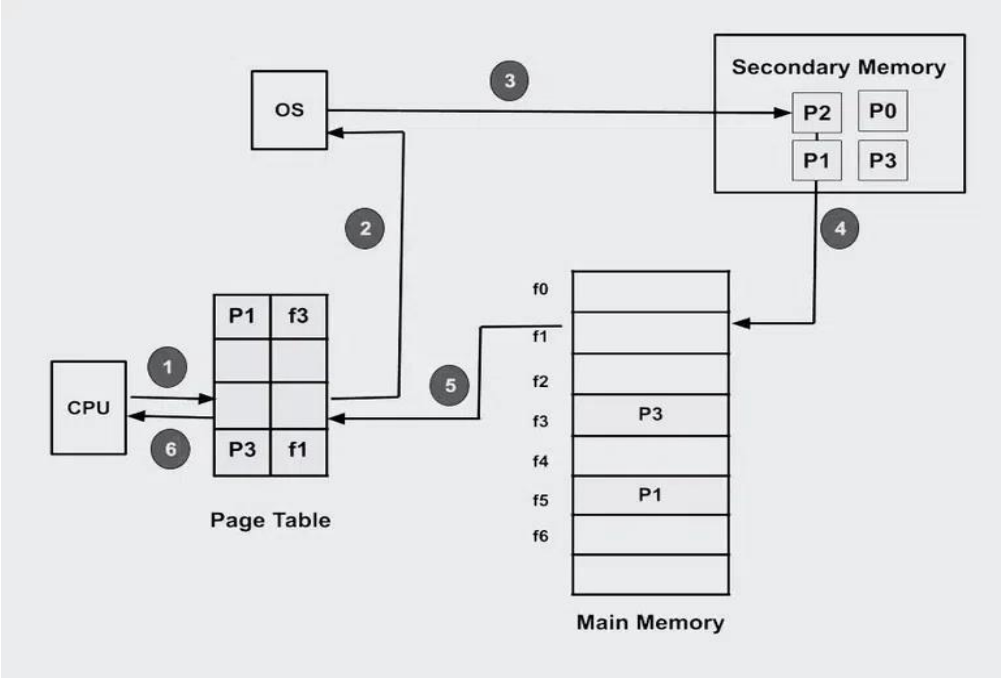
What is Page Fault?

The term "page miss" or "page fault" refers to a situation where a referenced page is not found in the main memory.
When a program tries to access a page, or fixed-size block of memory, that isn't currently loaded in physical memory (RAM), an exception known as a page fault happens. Before enabling the program to access a page that is required, the operating system must bring it into memory from secondary storage (such a hard drive) in order to handle a page fault.
In modern operating systems, page faults are a common component of virtual memory management. By enabling programs to operate with more data than can fit in physical memory at once, they enable the efficient use of physical memory. The operating system is responsible for coordinating the transfer of data between physical memory and secondary storage as needed.

Working Process of Demand Paging

Let us understand this with the help of an example. Suppose we want to run a process P which have four pages P0, P1, P2, and P3. Currently, in the page table, we have pages P1 and P3.

UNIT-II



The operating system's demand paging mechanism follows a few steps in its operation.

- **Program Execution:** Upon launching a program, the operating system allocates a certain amount of memory to the program and establishes a process for it.
- **Creating Page Tables:** To keep track of which program pages are currently in memory and which are on disk, the operating system makes page tables for each process.
- **Handling Page Fault:** When a program tries to access a page that isn't in memory at the moment, a page fault happens. In order to determine whether the necessary page is on disk, the operating system pauses the application and consults the page tables.
- **Page Fetch:** The operating system loads the necessary page into memory by retrieving it from the disk if it is there.
- The page's new location in memory is then reflected in the page table.
- **Resuming The Program:** The operating system picks up where it left off when the necessary pages are loaded into memory.
- **Page Replacement:** If there is not enough free memory to hold all the pages a program needs, the operating system may need to replace one or more pages currently in memory with pages currently in memory. on the disk. The page replacement algorithm used by the operating system determines which pages are selected for replacement.
- **Page Cleanup:** When a process terminates, the operating system frees the memory allocated to the process and cleans up the corresponding entries in the page tables.

UNIT-II

Common Algorithms Used for Demand Paging in OS

Demand paging is a memory management technique that loads parts of a program into memory only when needed. If a program needs a page that isn't currently in memory, the system fetches it from the hard disk. Several algorithms manage this process:

- **FIFO (First-In-First-Out):** Replaces the oldest page in memory with a new one. It's simple but can cause issues if pages are frequently swapped in and out, leading to thrashing.
- **LRU (Least Recently Used):** Replaces the page that hasn't been used for the longest time. It reduces thrashing more effectively than FIFO but is more complex to implement.
- **LFU (Least Frequently Used):** Replaces the page used the least number of times. It helps reduce thrashing but requires extra tracking of how often each page is used.
- **MRU (Most Recently Used):** Replaces the page that was most recently used. It's simpler than LRU but not as effective in reducing thrashing.
- **Random:** Randomly selects a page to replace. It's easy to implement but unpredictable in performance.

Reference string = `7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2`
Number of frames = `3`

Solution (FIFO works like a queue):

Step	Reference	Memory (Frames)	Page Fault?
1	7	[7]	Fault
2	0	[7, 0]	Fault
3	1	[7, 0, 1]	Fault
4	2	[0, 1, 2] (7 removed)	Fault
5	0	[0, 1, 2]	Hit
6	3	[1, 2, 3] (0 removed)	Fault
7	0	[2, 3, 0] (1 removed)	Fault
8	4	[3, 0, 4] (2 removed)	Fault
9	2	[0, 4, 2] (3 removed)	Fault
10	3	[4, 2, 3] (0 removed)	Fault
11	0	[2, 3, 0] (4 removed)	Fault
12	3	[2, 3, 0]	Hit
13	2	[2, 3, 0]	Hit

 Total Page Faults (FIFO) = 9

Problem 2: LRU Page Replacement

;;Same reference string: `7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2`
Frames = `3`

Solution (replace Least Recently Used):

Step	Reference	Memory (Frames)	Page Fault?
1	7	[7]	Fault
2	0	[7, 0]	Fault
3	1	[7, 0, 1]	Fault
4	2	[0, 1, 2] (7 removed)	Fault
5	0	[0, 1, 2]	Hit
6	3	[1, 2, 3] (0 removed)	Fault
7	0	[2, 3, 0] (1 removed)	Fault
8	4	[3, 0, 4] (2 removed)	Fault
9	2	[0, 4, 2] (3 removed)	Fault
10	3	[4, 2, 3] (0 removed)	Fault
11	0	[2, 3, 0] (4 removed)	Fault
12	3	[2, 3, 0]	Hit
13	2	[2, 3, 0]	Hit

 Total Page Faults (LRU) = 9

UNIT-II

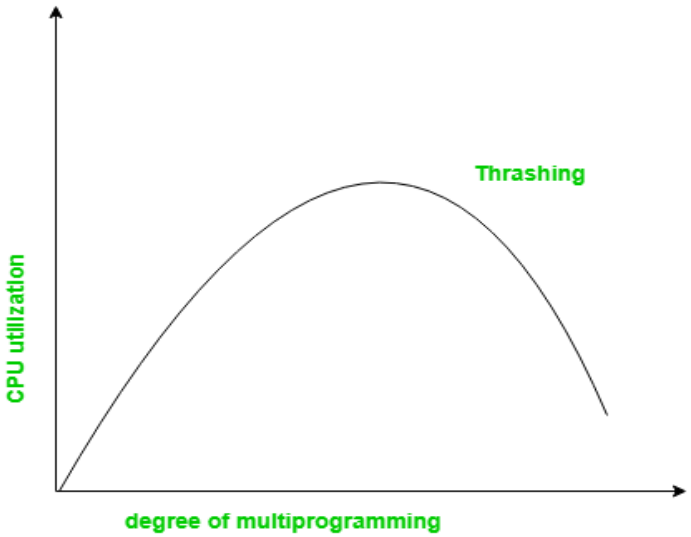
Thrashing is a condition or a situation when the system is spending a major portion of its time servicing the page faults, but the actual processing done is very negligible.

Causes of thrashing:

1. High degree of multiprogramming.
2. Lack of frames.
3. Page replacement policy.

We may also argue that as soon as the memory is full, the procedure begins to take a long time to swap in the required pages. Because most of the processes are waiting for pages, the CPU utilization drops again.

As a result, a high level of multi programming and a lack of frames are two of the most common reasons for thrashing in the operating system.



1. Working Set Model -

This model is based on the above-stated concept of the Locality Model.
The basic principle states that if we allocate enough frames to a process to accommodate its current locality, it will only fault whenever it moves to some new locality. But if the allocated frames are lesser than the size of the current locality, the process is bound to thrash.

According to this model, based on parameter A, the working set is defined as the set of pages in the most recent 'A' page references. Hence, all the actively used pages would always end up being a part of the working set.

The accuracy of the working set is dependent on the value of parameter A. If A is too large, then working sets may overlap. On the other hand, for smaller values of A, the locality might not be covered entirely.

If D is the total demand for frames and WSS_i is the working set size for process i,

$$D=\sum WSS_i$$

Now, if 'm' is the number of frames available in the memory, there are 2 possibilities:

- (i) D>m i.e. total demand exceeds the number of frames, then thrashing will occur as some processes would not get enough frames.
- (ii) D<=m, then there would be no thrashing.

File System Management

The **File System** manages how data is stored, retrieved, and organized on storage devices (like HDD, SSD, CD/DVD). It provides a structured way for users and programs to create, read, write, and delete files

1. File Concepts

- A **file** is a collection of related data stored on a storage device.
- It is the basic unit of storage and organization.
- **File Attributes (properties):**
  - **Name** – human-readable identifier.
  - **Type** – text, binary, executable, multimedia, etc.
  - **Location** – address on disk.
  - **Size** – length of file in bytes.
  - **Protection** – access rights (read, write, execute).
  - **Time & Date** – creation, last modified, last accessed.
  - **Owner/User ID** – identifies who owns the file.

2. Access Methods

Defines how data in a file can be accessed:

1. **Sequential Access**
   - Records are accessed one after another, in order.
   - Example: reading a text file line by line.
2. **Direct (Random) Access**
   - Access any block of file directly using an index or position.
   - Example: database records.
3. **Indexed Access**
   - A special index is maintained for fast search and access.
   - Example: library catalog or book index.

3. File Types and Operations

File Types:

- **Text file** – readable characters.
- **Binary file** – machine-readable (images, executables).
- **Executable file** – program code.
- **Multimedia files** – audio, video, images.

1. Text Files

- Contain **plain readable characters** (letters, numbers, symbols).
- Used for storing documents, notes, source code.
- Usually end with extensions like: `.txt`, `.c`, `.java`, `.py`.
- Example:
- `Hello, this is a text file.`

2. Binary Files

- Contain data in **binary (0s and 1s)** format.
- Not human-readable.
- Used by programs to store data efficiently.
- Examples: `.bin`, compiled files (`.exe`, `.class`, `.o`), images, audio.

3. Source Code Files

- Written by programmers in high-level languages.
- Need to be **compiled or interpreted** before execution.
- Extensions: `.c`, `.cpp`, `.java`, `.py`, `.js`.

4. Executable Files

- Contain **machine code** (ready-to-run program).
- Directly executed by the operating system.
- Examples: `.exe` (Windows), `.out` or no extension (Linux/Unix).

5. Multimedia Files

- **Image files**: `.jpg`, `.png`, `.gif`, `.bmp`
- **Audio files**: `.mp3`, `.wav`, `.aac`
- **Video files**: `.mp4`, `.avi`, `.mkv`

UNIT-II

File Operations:

- **Create** – make a new file.
- **Open / Close** – prepare file for use, then release it.
- **Read / Write** – transfer data to/from file.
- **Delete** – remove file permanently.
- **Append** – add data to the end.
- **Seek** – reposition the file pointer.

□ Basic File Operations

1. **File Creation**
   - A new file is created in the file system.
   - The OS allocates space for the file and updates the directory with:
     - File name
     - File type
     - Location (address on disk)
     - Other metadata (size, owner, permissions).

   Example: `create("myfile.txt")`

2. **File Writing**
   - Data is written into the file.
   - The OS manages writing from the program to the physical disk.
   - If the file already exists, data may overwrite or append (depending on mode).

   Example: `write("myfile.txt", "Hello World")`

3. **File Reading**
   - Data is read from the file into memory.
   - The OS keeps track of the **file pointer** (position of next read/write).

   Example: `read("myfile.txt") → "Hello World"`

4. **File Opening**
   - Before reading or writing, a file must be opened.

- The OS loads metadata into memory and provides a **file handle (descriptor)**.

Example: `open("myfile.txt")`

5. **File Closing**
   - After operations, the file must be closed.
   - The OS updates metadata (last modified time, size, etc.) and releases resources.

   Example: `close("myfile.txt")`

6. **File Deletion**
   - Removes a file from the directory structure.
   - The OS deallocates the disk space occupied by the file.

   Example: `delete("myfile.txt")`

7. **File Seeking**
   - Moves the **file pointer** to a specific location.
   - Allows random access (jumping to middle of a file instead of reading sequentially).

   Example: `seek("myfile.txt", position=50)`

4. Directory Structure

A **directory** is a collection of files that provides mapping between file names and storage locations.

UNIT-II

Types of Directory Structures:

1. **Single-Level Directory** – All files in one directory. (Simple but name conflicts).
2. **Two-Level Directory** – Each user has their own directory.
3. **Tree-Structured Directory** – Hierarchical (folders & subfolders).
4. **Acyclic Graph Directory** – Allows sharing of files/subdirectories.
5. **General Graph Directory** – More complex, may contain cycles.

1. Single-Level Directory

- **Structure:** All files are placed in the same directory.
- **Advantages:** Simple to implement and easy to understand.
- **Disadvantages:**
  - Naming conflicts (no two files can have the same name).
  - Difficult to group related files.

2. Two-Level Directory

- **Structure:** Each user has their own directory under the master directory.
- **Advantages:**
  - No naming conflict between users.
  - Easy user management.
- **Disadvantages:**
  - Still limited flexibility in organizing files.

3. Tree-Structured Directory

- **Structure:** A hierarchical (tree) structure with directories and subdirectories.
- **Advantages:**
  - Flexible and efficient for grouping files.
  - Easy navigation (pathnames).
- **Disadvantages:**
  - Searching can be slower compared to flat structures.

4. Acyclic Graph Directory

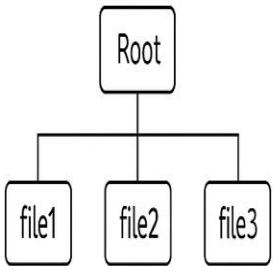
- **Structure:** Like a tree, but allows files or directories to be shared (links).
- **Advantages:**
  - Avoids file duplication (saves space).
  - Useful for software libraries.
- **Disadvantages:**
  - Complex to implement (must handle links and deletions).

5. General Graph Directory

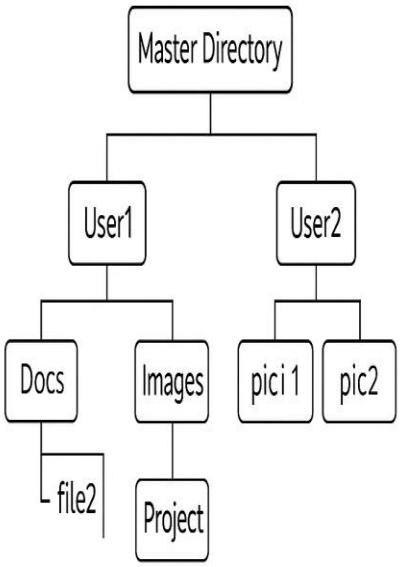
UNIT-II

- **Structure:** Extension of acyclic graph where cycles are allowed.
- **Advantages:**
  - Maximum flexibility (a directory can be shared multiple times).
- **Disadvantages:**
  - May lead to complexity and **infinite loops** while traversing.
  - Requires **garbage collection** to manage orphan files.

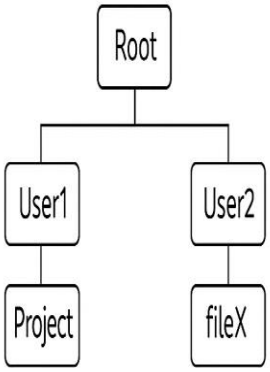
1. Single-Level Directory



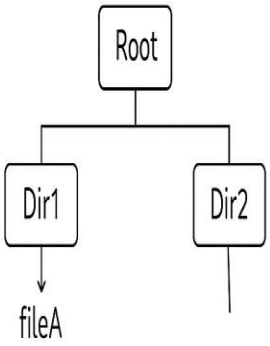
2. Two-Level Directory



3. Tree-Structured Directory



4. Acyclic Graph Directory



UNIT-II

5. File System Structure

The logical components of a file system are:

- **Boot Control Block** – Info required to boot OS.
- **Superblock** – Metadata (size, free blocks, free files).
- **File Control Block (FCB)** – Stores details of each file (name, size, permissions).
- **Directory Structure** – Maps file names to FCBs.
- **Data Blocks** – Actual storage for file contents.

6. Allocation Methods

Decides how disk space is assigned to files.

1. **Contiguous Allocation**
   - Files stored in consecutive disk blocks.
   - + Fast access, simple.
   - – External fragmentation, hard to grow file.
2. **Linked Allocation**
   - Each file is a linked list of blocks (each block points to next).
   - + No external fragmentation, file can grow easily.
   - – Slow random access, pointer overhead.
3. **Indexed Allocation**
   - A separate index block holds all addresses of a file's blocks.
   - + Direct access possible, no fragmentation.
   - – Extra overhead for index.

---

7. Free-Space Management

Keeps track of unused (free) disk blocks.

Methods:

1. **Bit Vector (Bitmap)** – 1 = allocated, 0 = free.
2. **Linked List** – Free blocks linked together.
3. **Grouping** – A free block stores addresses of other free blocks.
4. **Counting** – Store starting address + number of free blocks.

8. Directory Implementation

UNIT-II

How directories map file names to file information:

1. **Linear List**
   - Simple list of file names with pointers to FCBs.
   - **Disadvantage:** slow search (linear time).
2. **Hash Table**
   - Uses hashing to quickly locate file names.
   - **Advantage:** fast search (near constant time).

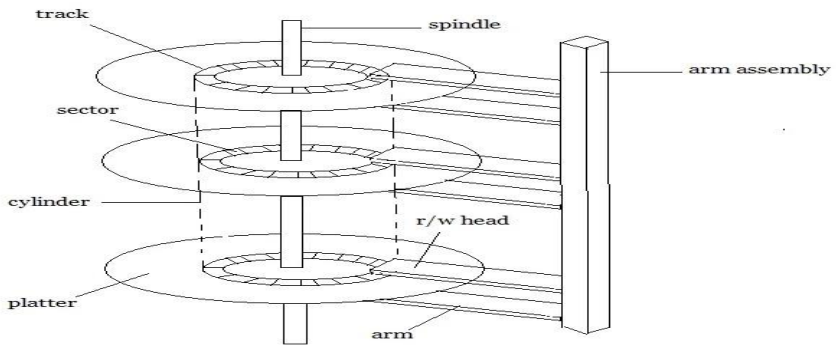
Storage Management

Storage management deals with how the operating system (OS) organizes and manages data on storage devices to ensure efficiency, reliability, and speed.

1. Mass Storage

- Refers to **large-capacity storage devices** like **Hard Disk Drives (HDDs)** and **Solid-State Drives (SSDs)**.
- Stores data permanently (non-volatile).
- Provides direct/random access to data blocks.
- Used for **OS, applications, and user files**.

2.Disk Structure



Disk Pack Structure

- A hard disk consists of:
  - **Platters** – circular disks coated with magnetic material.
  - **Tracks** – concentric circles on a platter.

  - **Sectors** – subdivisions of a track, smallest storage unit.
  - **Cylinders** – same track position across platters.
- **Disk Addressing** uses Cylinder, Head, Sector (CHS) or Logical Block Addressing (LBA).

---

3. RAID Levels (Redundant Array of Independent Disks)

RAID improves **performance** and/or **reliability** by combining multiple disks.

- **RAID 0 (Striping):**
  - Splits data across multiple disks.
  - + High speed, no redundancy.
- **RAID 1 (Mirroring):**
  - Duplicates data on two disks.
  - + High reliability, – costly (double space).
- **RAID 5 (Striping with Parity):**
  - Data + parity information distributed across disks.
  - + Reliability + performance, – slower writes.
- **RAID 6 (Double Parity):**
  - Stores two parity blocks for extra fault tolerance.
- **RAID 10 (Combination of RAID 1 + 0):**
  - Mirroring + striping, high speed & reliability.

---

4. Disk Scheduling Algorithms

Used to decide the order in which disk I/O requests are serviced (to reduce seek time).

1. **FCFS (First Come First Serve):** Serve requests in order of arrival.
2. **SSTF (Shortest Seek Time First):** Serve nearest track request.
3. **SCAN (Elevator Algorithm):** Head moves in one direction, servicing requests until end, then reverses.
4. **C-SCAN (Circular SCAN):** Similar to SCAN but only services one way; jumps back quickly.
5. **LOOK & C-LOOK:** Variants of SCAN/C-SCAN but stop at last request instead of disk end.

---

5. Swap Space Management

- **Swap Space:** Portion of disk used as an extension of RAM.
- Used in **virtual memory systems** to store inactive processes/pages.
- Located either in:
  - **Dedicated swap partition** (faster).
  - **Swap file** inside file system.
- Efficient swap management improves system performance.

---

6. Stable Storage

- Provides **reliable storage** even in case of failures (power loss, crash).
- Implemented using **redundancy techniques** (like mirroring & backups).
- Ensures data is not corrupted or lost during write operations.
- Used in **databases and critical systems**.

---

7. Tertiary Storage Structure

- Storage devices with **very large capacity but slower access time** compared to HDD/SSD.
- Examples: **Magnetic tapes, Optical disks (CD/DVD/Blu-ray), Cloud archival storage**.
- Characteristics:
  - Cheap and high capacity.
  - Mainly used for **backups and archival**.
  - Access is sequential (slow).

Disk Scheduling

- As mentioned earlier, disk transfer speeds are limited primarily by ***seek times*** and ***rotational latency.*** When multiple requests are to be processed there is also some inherent delay in waiting for other requests to be processed.
- ***Bandwidth*** is measured by the amount of data transferred divided by the total amount of time from the first request being made to the last transfer being completed, (for a series of disk requests.)
- Both bandwidth and access time can be improved by processing requests in a good order.

- Disk requests include the disk address, memory address, number of sectors to transfer, and whether the request is for reading or writing.

10.4.1 FCFS Scheduling

- ***First-Come First-Serve*** is simple and intrinsically fair, but not very efficient. Consider in the following sequence the wild swing from cylinder 122 to 14 and then back to 124:

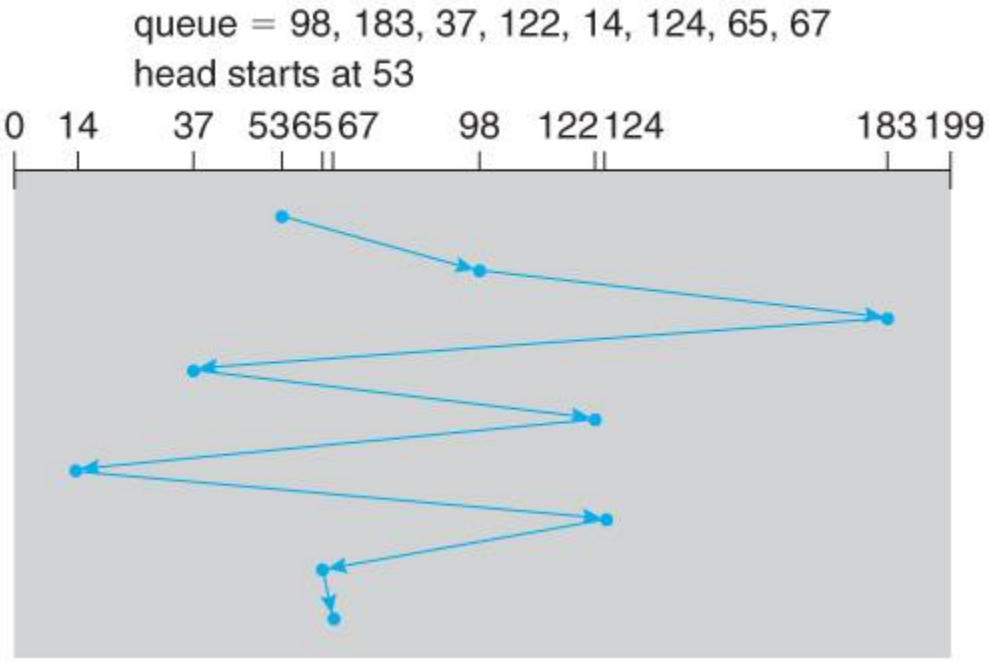


Figure 10.4 - FCFS disk scheduling.

10.4.2 SSTF Scheduling

- ***Shortest Seek Time First*** scheduling is more efficient, but may lead to starvation if a constant stream of requests arrives for the same general area of the disk.
- SSTF reduces the total head movement to 236 cylinders, down from 640 required for the same set of requests under FCFS. Note, however that the distance could be reduced still further to 208 by starting with 37 and then 14 first before processing the rest of the requests.

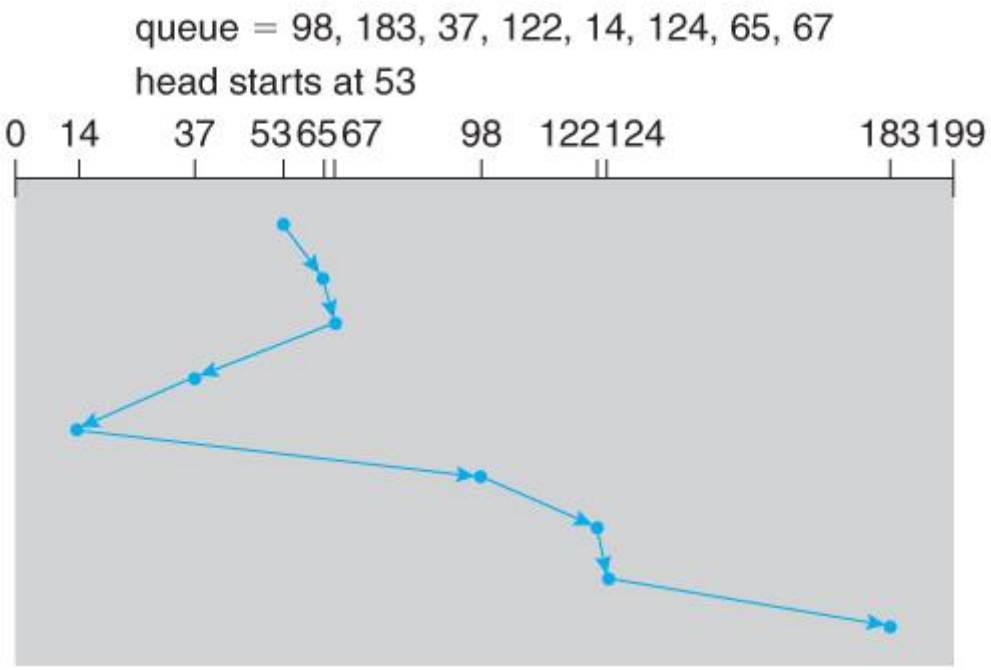


Figure 10.5 - SSTF disk scheduling.

10.4.3 SCAN Scheduling

- The ***SCAN*** algorithm, a.k.a. the ***elevator*** algorithm moves back and forth from one end of the disk to the other, similarly to an elevator processing requests in a tall building.

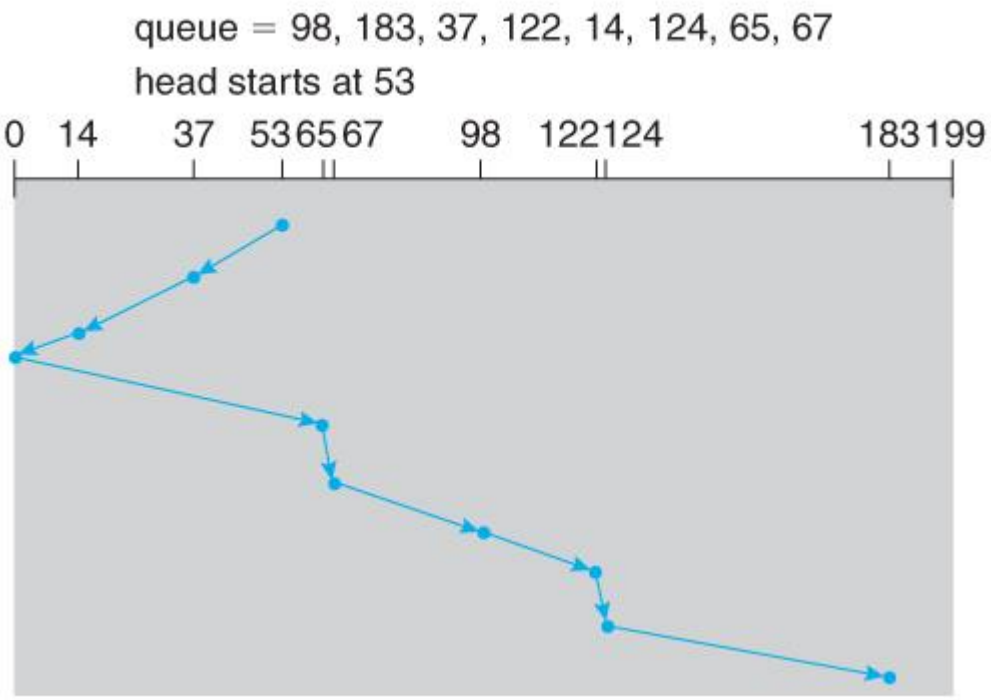


Figure 10.6 - SCAN disk scheduling.

- Under the SCAN algorithm, If a request arrives just ahead of the moving head then it will be processed right away, but if it arrives just after the head has passed, then it will have to wait for the head to pass going the other way on the return trip. This leads to a fairly wide variation in access times which can be improved upon.
- Consider, for example, when the head reaches the high end of the disk: Requests with high cylinder numbers just missed the passing head, which means they are all fairly recent requests, whereas requests with low numbers may have been waiting for a much longer time. Making the return scan from high to low then ends up accessing recent requests first and making older requests wait that much longer.

10.4.4 C-SCAN Scheduling

- The ***Circular-SCAN*** algorithm improves upon SCAN by treating all requests in a circular queue fashion - Once the head reaches the end of the disk, it returns to the other end without processing any requests, and then starts again from the beginning of the disk:

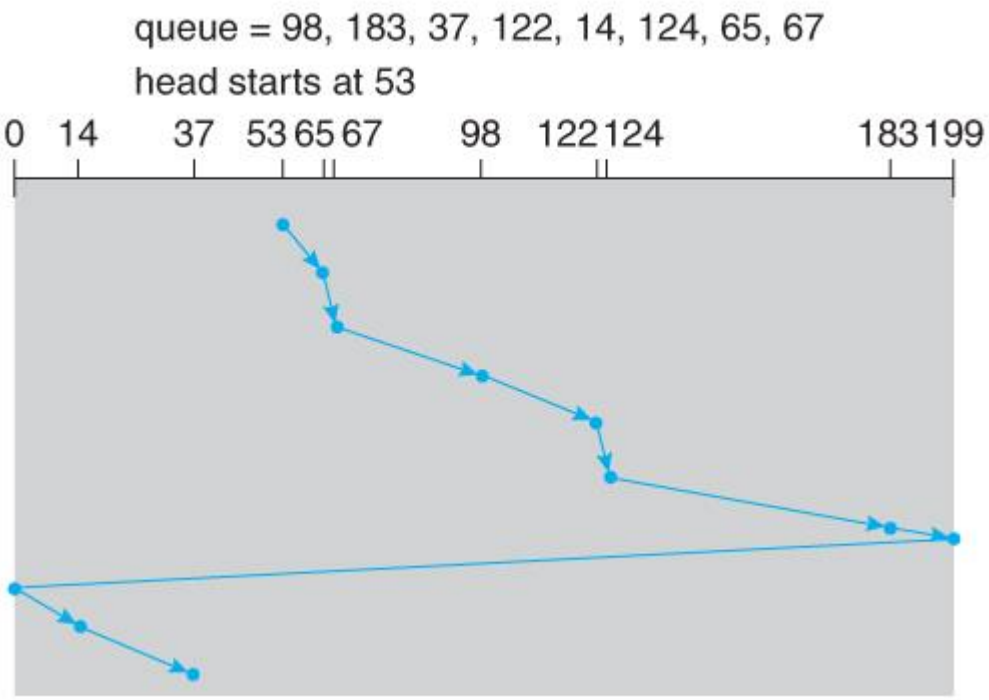


Figure 10.7 - C-SCAN disk scheduling.

12.4.5 LOOK Scheduling

- ***LOOK*** scheduling improves upon SCAN by looking ahead at the queue of pending requests, and not moving the heads any farther towards the end of the disk than is necessary. The following diagram illustrates the circular form of LOOK:

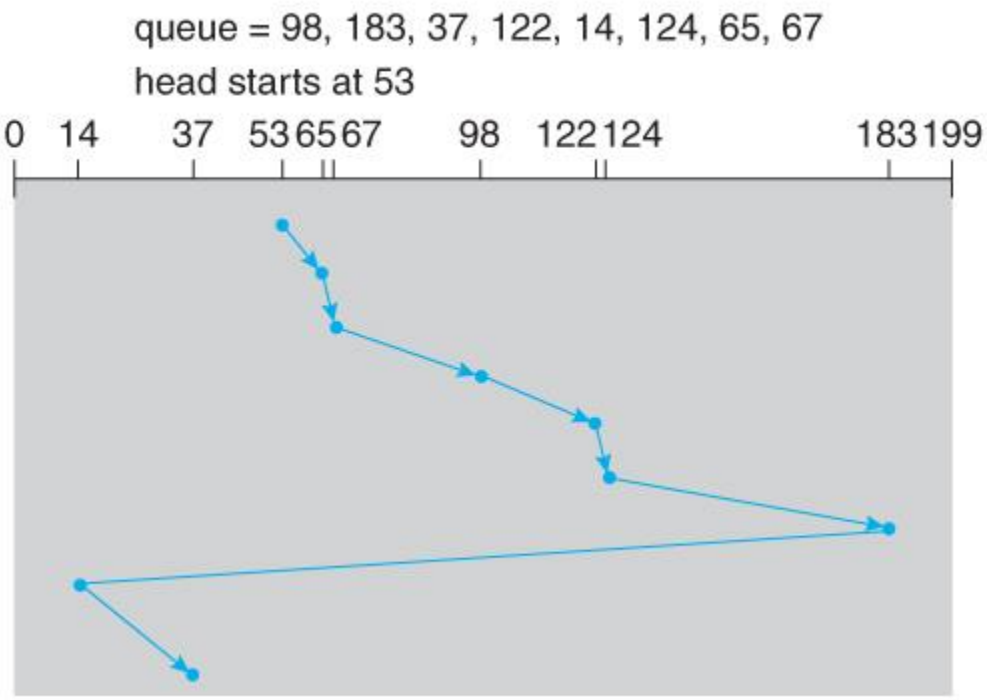


Figure 10.8 - C-LOOK disk scheduling.

UNIT-II

UNIT-II

UNIT - III: I/O Systems, Security, and Unix/Linux Overview

I/O System Management: I/O Hardware: Devices, Device Controllers, Direct Memory Access, I/O Software: Interrupt Handlers, Device Drivers, Device-Independent I/O Software,

System Protection and Security: Security Environment, Security Design Principles, User Authentication, Protection Mechanisms, Protection Domain, Access Control List,

Unix/Linux Overview & Case Studies: Development of Unix/Linux, Role of Kernel, System Calls, Elementary Linux Commands, Shell Programming, Directory Structure, System Administration.

I/O System Management:

I/O (Input/Output) System Management refers to the techniques and mechanisms an operating system uses to efficiently handle the flow of data between the computer's CPU and external hardware devices like keyboards, disks, and network interfaces. This involves managing device controllers, handling different types of I/O operations (character, block, network), employing buffering and caching, and implementing I/O scheduling algorithms to optimize performance and reduce wait times for applications.

(a) I/O Devices

- Devices are classified into two main categories:

1. **Block devices**
    - Store data in **fixed-size blocks** (sectors or clusters).
    - Data can be read/written in any order (random access).
    - Examples: Hard Disk, SSD, CD-ROM.
2. **Character devices**
    - Stream data **one character at a time** (sequential access).
    - No block structure.
    - Examples: Keyboard, Mouse, Serial ports.
3. **Network devices**
    - Transmit/receive data packets across a network.
    - Example: Ethernet card, Wi-Fi adapter.

(b) Device Controllers

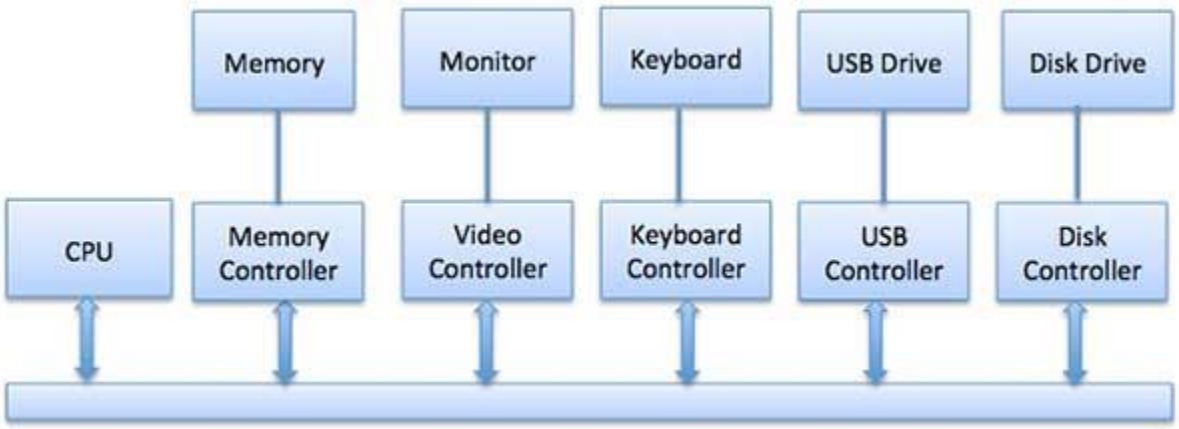
- Every I/O device is controlled by a **device controller** (electronic circuitry).
- Acts as an interface between **CPU ↔ Device**.
- Responsibilities:
    1. Receives commands from CPU.
    2. Converts commands into device-specific actions.

3. Stores data temporarily in **buffers** (in controller memory).
4. Reports device status via **status register**.

Registers in a device controller:

- **Data register** → holds data to be transferred.
- **Control register** → stores command from CPU (e.g., Read/Write).
- **Status register** → indicates device status (Ready/Busy/Error).

Example:
A **disk controller** manages reading/writing sectors from a hard disk. The OS sends "Read sector 5," and the controller handles actual head movement and data transfer.



Direct Memory Access (DMA)

Direct Memory Access is a feature that allows I/O devices **(like disk drives, network cards, sound cards) to directly transfer data to/from the** main memory (RAM) without continuous involvement of the CPU

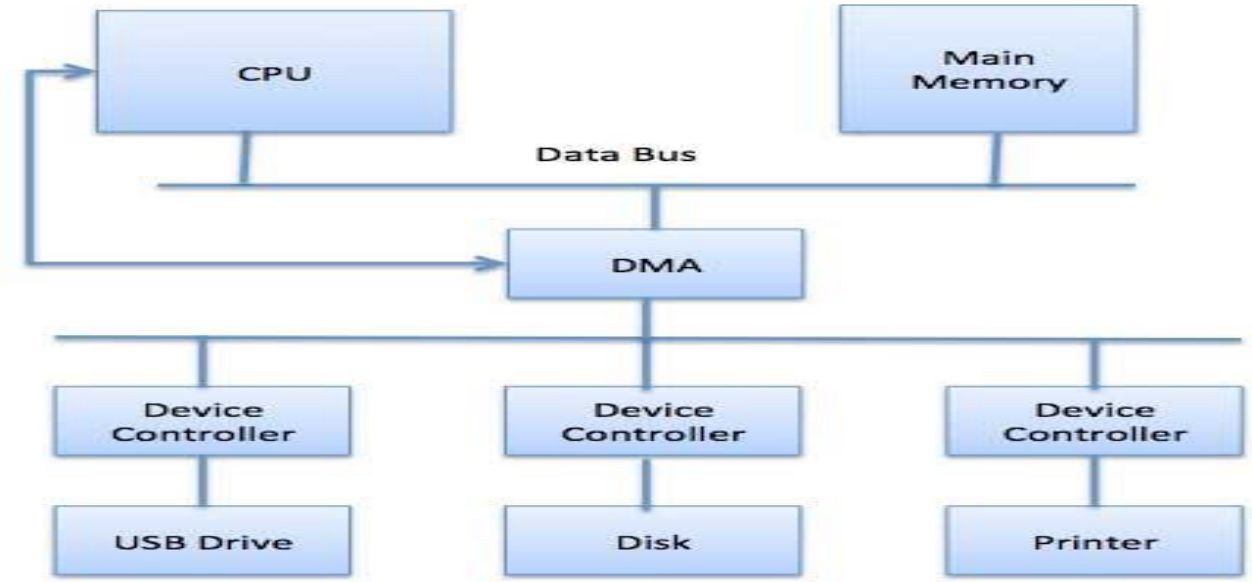
Steps in DMA transfer:

1. CPU programs DMA controller (gives source, destination, size).
2. DMA controller takes control of the **system bus**.
3. Transfers data **directly between device and main memory**.
4. When done, DMA sends an **interrupt** to notify CPU.

Advantages:

- Reduces CPU overhead.
- Increases throughput (CPU can execute other processes during I/O).

Example:
Copying a 4 GB movie file from HDD → RAM using DMA is faster than CPU doing each transfer.



I/O Software

A key concept in the design of I/O software is that it should be device independent where it should be possible to write programs that can access any I/O device without having to specify the device in advance. For example, a program that reads a file as input should be able to read a file on a floppy disk, on a hard disk, or on a CD-ROM, without having to modify the program for each different device.

OS hides hardware complexities using **I/O software layers**.

(a) Interrupt Handlers

- When a device completes an operation, it sends an **interrupt signal** to CPU.
- CPU pauses current execution, runs the **interrupt service routine (ISR)**.
- ISR responsibilities:
    - Save current process state.
    - Identify which device caused interrupt.
    - Service the device (e.g., fetch key pressed).
    - Restore process state.

Example:
When you press a key, the keyboard controller raises an interrupt → CPU runs **keyboard ISR** → character stored in buffer.

(b) Device Drivers

- Device-specific software inside OS.
- Role: Convert high-level OS commands → low-level hardware instructions.
- Each device type has a driver (keyboard driver, printer driver, NIC driver).
- Without drivers, the OS cannot communicate with hardware.

Example:
When you click "Print" → OS calls **printer driver** → driver sends instructions to printer controller → printer prints document.

(c) Device-Independent I/O Software

- Provides a **uniform interface** for all devices, hiding differences.
- Responsibilities:
    1. **Buffering** → Temporary storage to match speed difference between device and CPU.
        - Example: Printing characters stored in buffer before printing one-by-one.
    2. **Error reporting** → Detect & notify errors (disk failure, printer offline).
    3. **Naming** → Uniform naming of files/devices (e.g., `C:\file.txt` or `/dev/tty`).
    4. **Access control & protection** → Ensures only authorized users/processes access device.

(d) User-Level I/O (System Calls)

- Applications use **system calls** (like `read()`, `write()`) to request I/O operations.
- OS translates these requests into device-specific operations via drivers.

Example:
C program:

```
read(fd, buffer, 100);  // system call
```

- This requests OS to read 100 bytes from file/device linked with `fd`.

3. Layered I/O Structure

```
+--------------------------+  ← Application Programs
|       User Programs      |
+--------------------------+
|      System Calls        |  ← Interface for I/O
+--------------------------+
| Device-Independent I/O   |  ← Buffering, Naming, Error Handling
+--------------------------+
|     Device Drivers       |  ← Hardware-specific translation
+--------------------------+
|   Interrupt Handlers     |  ← Handle device interrupts
+--------------------------+
|   Hardware Controllers   |  ← Actual I/O hardware
+--------------------------+
```

System Protection and Security: Security Environment, Security Design Principles, User Authentication, Protection Mechanisms, Protection Domain, Access Control List,

System Protection and Security

1. Security Environment

The security environment in an operating system refers to a system of policies, mechanisms, and practices that protect hardware, software, and data from unauthorized access, modification, and disruption by malicious attacks or misconfigurationss

- Refers to the **overall framework** in which security policies, mechanisms, and tools operate.
- Includes:
  - **Threats**: Malware, unauthorized access, data leaks.
  - **Vulnerabilities**: Weak passwords, unpatched software, insecure networks.
  - **Attacks**: Phishing, DoS (Denial of Service), privilege escalation.

- **Security Goals**:
  - **Confidentiality** – only authorized users can access data.
  - **Integrity** – data cannot be modified without authorization.
  - **Availability** – resources are available when needed.

2. jSecurity Design Principles

1. Least Privilege

➡ Give minimum rights needed to perform a task.

The principle of least privilege is a security design principle that requires that users be given the bare minimum permissions necessary to perform their tasks. So, this principle is also sometimes referred to as the principle of least authority

 Example: In a college computer lab, students can **use lab PCs but cannot install new software**. Only the admin has that right.

2. Fail-Safe Defaults

➡ Default should deny access until explicitly allowed.

Fail-safe defaults are security settings that are configured to prevent unauthorized access or use of resources. By default, all users should have the least amount of privileges necessary to perform their job function.
 Example: When you join a new **Wi-Fi network**, it asks for a password. You **cannot connect by default** unless the admin allows you.

3. Economy of Mechanism

➡ Keep security design simple and small.

The principle of economy of mechanism states that a system should be designed to minimize the number of distinct components (Eg. processes, machines, nodes, etc.) that must interact to perform a given task. This principle is also known as the principle of least action. The design of a security system should be as simple and efficient as possible.

 Example: Instead of a complicated **10-step login**, a system just asks for a **username + password**. Simple → fewer chances of errors.

4. Complete Mediation

➡ Every access to a resource must be checked.

Security design principles should be comprehensive and address all potential security risks. It should be integrated into the overall design of the system and implemented in a way that minimizes the impact on performance and usability. It should be reviewed and updated on a regular basis.
 Example: Every time you open **WhatsApp**, it checks your **fingerprint/face lock** (not just once). It **re-checks** every time.

5. Open Design

➡ Security should depend on **keys/passwords**, not secret design.

Open design is a security design principle that advocates for the openness of security systems. The principle of open design states that security systems should be designed in such a way that they can be easily inspected, analyzed, and modified by anyone with the necessary skills and knowledge.
 Example: The **ATM system** is public knowledge, but security depends on your **ATM PIN** (not on keeping ATM's design secret).

6. Separation of Privilege

➡ Require more than one condition.

The principle of separation of privilege states that a user should not be able to access all areas of a system
 Example: For **online banking**, you need both:

- Password ✓
- OTP sent to your mobile ✓
  → Both must be correct.

7. Least Common Mechanism

The principle of least common mechanism states that security should be designed so that there is a minimum number of mechanisms that are shared by all users. This principle is important because it reduces the chances that a security flaw will be exploited by more than one user.

➡ Minimize sharing to reduce attacks.

 Example: In a **hostel mess card system**, each student has their **own card** instead of a **common password** for everyone. Shared access = higher risk.

8. Psychological Acceptability

➡ Security should not be too complex for users.

The psychological acceptability of security design principles refers to the extent to which users are willing to accept and comply with the security measures implemented in a system

 Example: Instead of asking you to **change password daily**, Gmail allows you to **stay signed in** but warns you when there's unusual activity. Easy for users.

- **ATM PIN**
  - A 4-digit PIN is easy to remember and quick to type → students actually use it.
  - If the bank forced a 16-digit PIN → students would write it down or forget it.
- **College Wi-Fi Login**
  - If Wi-Fi asks for your roll number + one-time password → easy to access and secure.
  - If it asked you to type 10 different codes every time → students won't use it.
- **Online Exam Portal**
  - If login requires just a username, password, and maybe OTP → acceptable.
  - If it required 5 layers of login (face scan + fingerprint + password + OTP + security questions) → too hard, students get frustrated.

User Authentication

User authentication is the process of establishing the identity of an individual who wants to have access to a particular system or service. It involves a process of ensuring that the user claiming to be a specific personality is substantial through proving credentials like passwords, biometric data, security tokens, or other authenticity factors. This verification step is essential in safeguarding systems and systems' components from access and use by unauthorized individuals and entities and in preventing misuse of information that is considered confidential or sensitive.

- **Process of verifying user identity.**
- Methods:
    1. **What you know** – Passwords, PINs.
    2. **What you have** – Smart card, OTP, security token.
    3. **What you are** – Biometrics (fingerprint, face recognition).
    4. **Where you are** – Location-based authentication (IP, GPS).
- Example: Online banking uses **multi-factor authentication** (password + OTP).

5. Protection Mechanisms

- **System protection** in an operating system refers to the mechanisms implemented by the operating system to ensure the security and integrity of the system. System protection involves various techniques to prevent unauthorized access, misuse, or modification of the operating system and its resources.
- There are several ways in which an operating system can provide system protection:
- **User authentication:** The operating system requires users to authenticate themselves before accessing the system. Usernames and passwords are commonly used for this purpose.
- **Access control:** The operating system uses access control lists (ACLs) to determine which users or processes have permission to access specific resources or perform specific actions.
- **Encryption:** The operating system can use encryption to protect sensitive data and prevent unauthorized access.
- **Firewall:** A firewall is a software program that monitors and controls incoming and outgoing network traffic based on predefined security rules.
- **Antivirus software:** Antivirus software is used to protect the system from viruses, malware, and other malicious software.
- **System updates and patches:** The operating system must be kept up-to-date with the latest security patches and updates to prevent known vulnerabilities from being exploited.

    - **Access Control** – Determines who can use what resources.
    - **Encryption** – Protects data in storage or during transmission.
    - **Sandboxing** – Restricts program execution in an isolated environment.
    - **Firewalls & Intrusion Detection** – Control and monitor network access.

5. Protection Domain

- Defines a **set of resources** (objects) and the **operations** (rights) that a process or user can perform.
- Each domain specifies:
    - Objects: files, printers, memory.
    - Rights: read, write, execute, delete.

- Example:
  - Domain A → Read/Write file1, Execute program1.
  - Domain B → Only Read file1.
- **Domain Switching**: A process can move between domains (e.g., user mode ↔ kernel mode).

6. Access Control List (ACL)

- **ACL = A table that specifies which users/processes have what access rights to an object.**
- Stored with each object (e.g., file, directory).
- Example (File.txt ACL):
- `User1: Read, Write`
- `User2: Read`
- `Admin: Read, Write, Execute`
- `Others: No Access`
- Advantages:
  - Fine-grained control. a security approach that allows for precise permissions on specific data or actions
  - Easy to audit.
- Disadvantage:
  - Can get large and hard to manage.

Unix/Linux Overview & Case Studies: Development of Unix/Linux, Role of Kernel, System Calls, Elementary Linux Commands, Shell Programming, Directory Structure, System Administration

 Unix/Linux Overview & Case Studies

1. Development of Unix/Linux

- **Unix**
  - Developed in **1969 at AT&T Bell Labs** by **Ken Thompson, Dennis Ritchie, and others**.
  - Originally written in **assembly**, later rewritten in **C** → portable across machines.
  - Key features: multitasking, multiuser, hierarchical file system.

Important Features Introduced

- **Multiuser**: Many users can log in and use at the same time.
- **Multitasking**: Multiple programs run together.
- **Hierarchical File System**: Files organized into directories and subdirectories.
- **Security**: User authentication and file permissions.

- **Portability**: Easy to adapt to new hardware.

- **Linux**
  - Created by **Linus Torvalds** in **1991** as a free, open-source Unix-like OS.
  - Uses the **GNU utilities** + **Linux kernel** → called **GNU/Linux**.
  - Popular distributions: Ubuntu, Fedora, Red Hat, Debian, Arch.
- Based on the principles of **Unix**, but available for everyone under the **GNU General Public License (GPL)**.

Key Features of Linux

- **Open Source** → Source code is free for anyone to use, modify, and distribute.
- **Multiuser** → Multiple users can work at the same time.
- **Multitasking** → Runs many programs simultaneously.
- **Security** → Provides strong user authentication and file permissions.
- **Portability** → Runs on desktops, servers, mobiles, IoT devices.
- **Stability & Reliability** → Rarely crashes, used in servers and supercomputers.

Difference between Unix and Linux

Point	Unix	Linux
Definition	A proprietary multitasking, multiuser OS developed in **1969 at Bell Labs**.	An **open-source Unix-like OS** developed by **Linus Torvalds in 1991**.
License	Mostly **proprietary (paid)**.	**Free and open-source** (GPL license).
Source Code	**Closed** (not freely available).	**Open** (anyone can view, modify, and share).
Development	Developed by **AT&T Bell Labs** and later companies (IBM, HP, etc.).	Developed by **Linus Torvalds** with worldwide community support.
Cost	Usually **expensive**.	**Free** (some enterprise editions may charge for support).
Portability	Limited to specific hardware.	Highly **portable** – runs on desktops, servers, mobiles, IoT devices.
Distributions	Various vendor versions: Solaris, AIX, HP-UX, BSD.	Many **distros**: Ubuntu, Fedora, Red Hat, Debian, Kali Linux, etc.
Usage Today	Mostly in large servers, mainframes, and	Widely used in **servers, desktops, Android**

Point	Unix	Linux
	special-purpose systems.	**phones, cloud, IoT, supercomputers**.

2. Role of Kernel

1. Definition of Kernel

- The **kernel** is the **core part of an operating system**.
- It works as a **bridge between hardware and software**.
- Without the kernel, applications cannot directly interact with hardware.

---

2. Main Roles of Kernel

1. **Process Management**
   - Creates, schedules, and terminates processes.
   - Handles multitasking (running many programs at the same time).
2. **Memory Management**
   - Allocates and frees memory for processes.
   - Ensures no process uses memory of another.
3. **Device Management**
   - Controls input/output devices (keyboard, disk, printer, etc.) using **device drivers**.
   - Acts as a mediator between software and hardware devices.
4. **File System Management**
   - Organizes data in files and directories.
   - Manages permissions (who can read/write/execute).
5. **System Calls / Communication**
   - Provides system calls for applications to request services (like read, write, open).
   - Ensures safe interaction between user programs and hardware.

1. Monolithic Kernel

- All OS services (memory, process, device drivers, file system) run in **one large kernel space**.
- **Advantages**: Fast execution (no switching overhead).
- **Disadvantages**: Large size, harder to maintain, one error may crash the whole system.
- **Example**: Linux, Unix.

2. Microkernel

- Only essential services (CPU scheduling, memory management, inter-process communication) are inside the kernel.
- Other services (drivers, file system, etc.) run in **user space**.
- **Advantages**: More secure, stable, easier to maintain.
- **Disadvantages**: Slower due to extra communication between kernel and user space.
- **Example**: Minix, QNX, Mach.

3. Hybrid Kernel

- Combination of **Monolithic** and **Microkernel** features.
- Core functions in kernel space, but some services run in user space.
- **Advantages**: Faster than microkernel, more stable than monolithic.
- **Disadvantages**: More complex design.
- **Example**: Windows NT/XP/10, macOS (XNU kernel).-

3. System Calls

- **Interface between user programs and the kernel.**
- A **system call** is a way for a program to **request a service from the operating system's kernel**.
- Since user programs **cannot directly access hardware**, they use system calls to communicate with the OS.

 Example: When you save a file, your program uses a **system call** to ask the OS to write data to disk.

Categories of System Calls

1. **Process Control**
   - Create, end, or manage processes.
   - Examples: `fork()`, `exit()`, `wait()`.
2. **File Management**
   - Create, open, read, write, close files.
   - Examples: `open()`, `read()`, `write()`, `close()`.
3. **Device Management**
   - Request and release devices, perform I/O operations.
   - Examples: `ioctl()`, `read()`, `write()`.
4. **Information Maintenance**
   - Get or set system data (date, time, system info).
   - Examples: `getpid()`, `alarm()`, `sleep()`.
5. **Communication**
   - Transfer information between processes.
   - Examples: `pipe()`, `shmget()`, `mmap()`, sockets.

Example:

```
int fd = open("file.txt", O_RDONLY);    // Open file
read(fd, buffer, 100);                  // Read 100 bytes
close(fd);                              // Close file
```

4. Elementary Linux Commands

Some basic commands (used daily):

- **File & Directory**
  - `ls` → list files
  - `pwd` → show current directory
  - `cd dir` → change directory
  - `mkdir dir` → create directory
  - `rm file` → delete file
- **File Operations**
  - `cp file1 file2` → copy
  - `mv file1 file2` → move/rename
  - `cat file` → display file content
  - `touch file` → create empty file
- **Process Control**
  - `ps` → show processes
  - `kill pid` → terminate process
  - `top` → real-time process monitor

6. Shell Programming

The shell acts as an interface between the user and the operating system. It interprets commands and translates them into actions the OS can understand. Examples include Bash (Bourne Again Shell), Zsh, Csh, and Ksh in Unix-like systems, and the Command Prompt or PowerShell in Windows.

- **Shell = Command interpreter** (e.g., `bash`, `sh`, `zsh`).
- **Shell script** = file containing a series of shell commands.

Example (simple script):

```
#!/bin/bash
echo "Enter your name: "
read name
```

```
echo "Hello, $name!"
```

Run using:

```
chmod +x script.sh
./script.sh
```

6. Directory Structure

Unix/Linux follows a **tree-structured hierarchy** with root (`/`) at the top.

Key directories:

- `/` → Root directory
- `/bin` → Essential binaries (ls, cp, mv)
- `/etc` → Configuration files
- `/home` → User home directories
- `/usr` → User programs, libraries
- `/var` → Logs, spool files
- `/tmp` → Temporary files
- `/dev` → Device files
- `/boot` → Boot loader & kernel

7. System Administration

Tasks performed by system administrators:

1. **User Management** – `adduser`, `passwd`, `deluser`.
2. **File System Management** – Mount/unmount disks (`mount`, `umount`).
3. **Backup & Recovery** – `tar`, `rsync`.
4. **Process & Resource Monitoring** – `ps`, `top`, `df`, `du`.
5. **Security Management** – Permissions (`chmod`, `chown`), firewall, updates.
6. **Networking** – Configuring IP (`ifconfig`, `ip`), services (`systemctl`).

UNIT IV: System Software and Language Processing

Overview of System Software: Software and Software Hierarchy, Systems Programming and Machine Structure, Interfaces, Address Space, and Computer Languages, System Software Development and Recent Trends,

Language Processors: Programming Languages and Language Processing, Symbol Tables and Data Structures for Language Processing, Search and Allocation Data Structures,

Assemblers and Macro Processors: Elements of Assembly Language Programming, Design and Types of Assemblers, Macro Definitions, Expansion, Nested Macros, and Advanced Macro Features, Design of Macro Assemblers and Macro Processors,

Linkers and Loaders: Concept of Linking and Relocation, Linking in MS-DOS, Dynamic Linking, Loading Schemes: Sequential, Direct, Absolute, Relocating, and Linking Loaders, Comparison of Linkers and Loaders.

Overview of System Software:

Software and Software Hierarchy:

Software

Software is a collection of programs, procedures, and documentation that perform specific tasks on a computer system. It provides instructions to the hardware and enables users to interact with the system. In simple terms, hardware is the body of the computer, and software is its brain that tells the hardware what to do. Without software, hardware is useless.

Types of Software / Software Hierarchy

Software is broadly divided into **two categories**:

1. System Software

- System software is designed to control and manage the hardware of a computer and provide a platform for running application software.
- It acts as an **interface between the user and hardware**.

Main components of system software:

1. **Operating System (OS):**
    - Manages computer resources such as CPU, memory, input/output devices.
    - Examples: Windows, Linux, macOS, Android.

2. **Device Drivers:**
    - Special programs that allow the operating system to communicate with hardware devices.
    - Example: Printer driver, Sound driver.
3. **Utility Programs:**
    - Perform maintenance tasks like antivirus scanning, file management, compression, and backup.
    - Example: WinRAR, Disk Cleanup, Antivirus software.

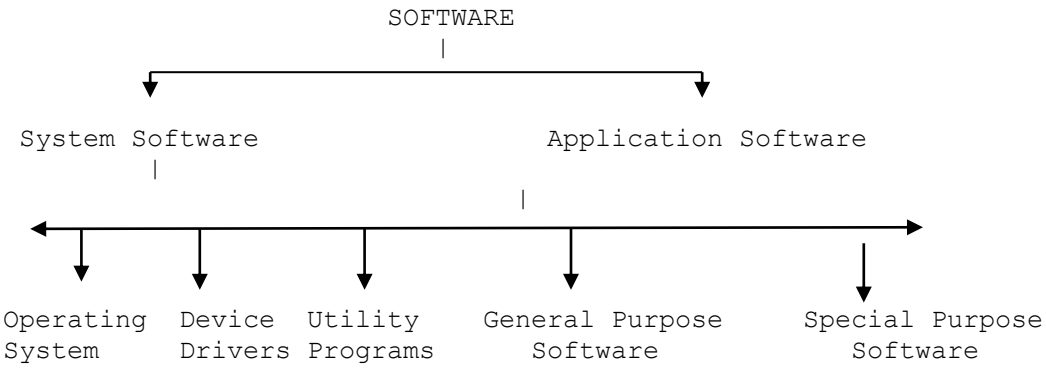
2. Application Software

- Application software is developed to perform **specific tasks for the user**.
- These are the programs we use in our daily life.

Types of application software:

1. **General Purpose Application Software:**
    - Used for common tasks such as typing documents, making presentations, browsing, or calculations.
    - Example: MS Word, Excel, PowerPoint, Google Chrome.
2. **Special Purpose Application Software:**
    - Developed for a particular purpose or organization.
    - Example: Banking software, Railway reservation system, Hospital management software.

Software Hierarchy Diagram



Systems Programming and Machine Structure

1. Introduction

A computer system is made up of **hardware** and **software**.

- **Hardware** consists of the physical components such as CPU, memory, input/output devices.
- **Software** consists of programs that make hardware useful.

To use the hardware effectively, special kinds of programs known as **system software** are required. The development of this software is studied under **Systems Programming**, while the physical and logical arrangement of hardware is described by **Machine Structure**.

2. Systems Programming

- **Definition:**
  Systems programming is the process of designing and implementing **system software** that controls the hardware and provides a platform for application programs.
- It deals with programs that **interact directly with computer hardware** and provide services to the user and application software.

Examples of System Software

1. **Assembler** – Converts assembly language into machine code.
2. **Compiler/Interpreter** – Converts high-level language into machine code.
3. **Loader & Linker** – Loads programs into memory and links different modules.
4. **Operating System** – Manages hardware resources and provides services.
5. **Device Drivers** – Enable communication with hardware devices like printers, keyboards.

Characteristics of Systems Programming

- Written in **low-level languages** like C and Assembly.
- Concerned with **efficiency, speed, and resource management**.
- Closer to **hardware** compared to application programming.
- Provides a **base for application programs**.

Machine Structure

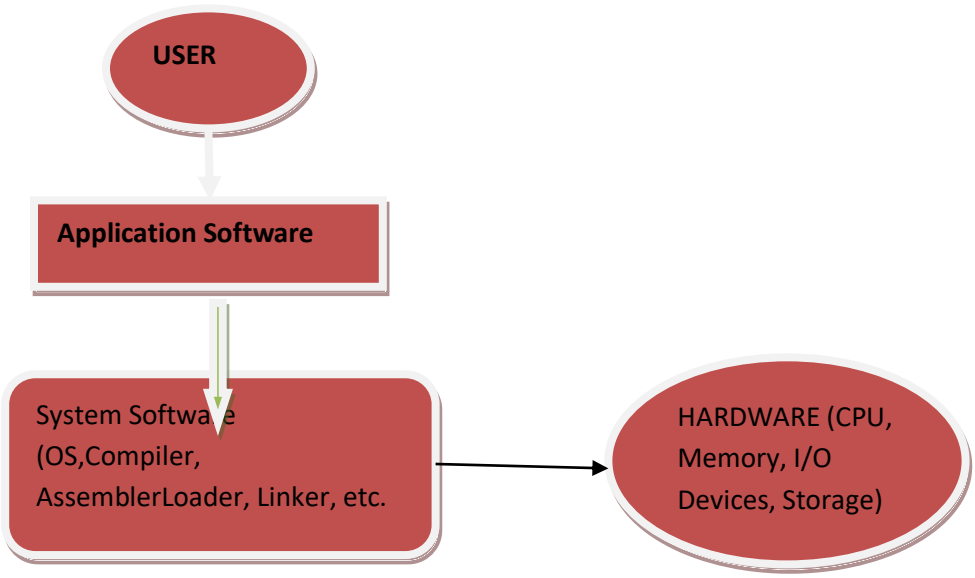
- **Definition:**
  Machine structure describes the **organization and architecture of a computer system**. It explains how hardware components (CPU, memory, I/O devices) interact with each other and how software communicates with hardware.

Components of Machine Structure

1. **Hardware:**
   - CPU, memory, input/output devices, and storage units.
   - Executes machine instructions.
2. **System Software:**
   - Works as a bridge between hardware and applications.
   - Examples: Operating system, compilers, assemblers.
3. **Application Software:**
   - Programs written for end-users to perform specific tasks.
   - Examples: MS Word, Photoshop, Web browsers.
4. **User:**
   - The person who interacts with the computer through application software.

4. Relationship Between Systems Programming and Machine Structure

- Systems programming **produces the software** that controls the hardware described by machine structure.
- Machine structure defines the **instruction set architecture (ISA)**, registers, and memory organization, which **system software must understand** to work correctly.
- Example:
  - An **assembler** (systems programming) translates assembly instructions into **binary machine instructions** (machine structure).
  - A **compiler** generates macjhine code based on the machine's architecture.
    Diagram



Interfaces

- **Definition:**
  An interface is a boundary where two systems, components, or software layers communicate with each other.
- In computer systems, interfaces ensure that **hardware, software, and users** can interact smoothly.

Types of Interfaces

1. **User Interface (UI):**
   - Allows the user to communicate with the computer.
   - Types: Command Line Interface (CLI), Graphical User Interface (GUI).
2. **Hardware Interface:**
   - The interaction between hardware components (CPU ↔ memory, CPU ↔ I/O devices).
3. **Software Interface (API):**
   - Defines how software components communicate with each other.
   - Example: Application Programming Interfaces (APIs).

2. Address Space

- **Definition:**
  Address space refers to the **range of memory addresses** that a process or program can use.

Types of Address Space

1. **Logical Address Space:**
   - Generated by the CPU (used in programs).
   - Independent of physical memory.
2. **Physical Address Space:**
   - Actual addresses in main memory (RAM).
   - Handled by **Memory Management Unit (MMU)**.

 Example: A program may think it uses addresses from `0x0000 to 0xFFFF` (logical), but physically those addresses may map to actual RAM locations.

3. Computer Languages

Computer languages allow humans to give instructions to computers.

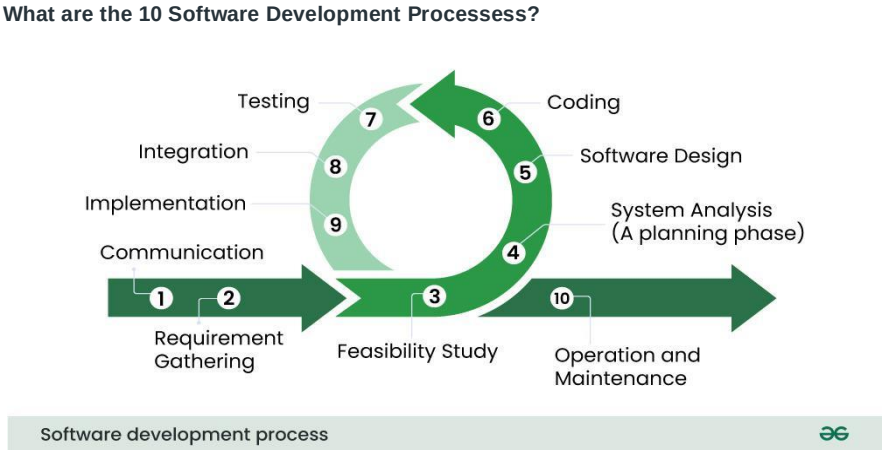
Levels of Languages

1. **Machine Language:**
   - Binary instructions (0s and 1s).
   - Directly understood by CPU.
   - Example: `10110000 01100001`
2. **Assembly Language:**
   - Symbolic representation of machine instructions.
   - Example: `MOV A, B`
   - Requires an **assembler** to convert into machine code.
3. **High-Level Languages (HLL):**
   - Closer to human language, easy to use.
   - Example: C, C++, Java, Python.
   - Requires a **compiler/interpreter**.

4. System Software Development

System software is developed to manage hardware and run application programs.

Steps in System Software Development



Software Developement Process

The software development process is the sequence of activities that leads to the production of a software product. The steps of software development process are as follows:

1. Communication

The first and foremost step is where the user contacts the service provider i.e. software organization and initiates the request for a desired software product. The software organization talks with the customer about its requirement and then work according to its needs.

2. Requirement Gathering

In this step, the team of software developers holds discussions with various stakeholders from the problem domain and provides as much as information possible for the requirement of the software product. The requirements can be of different forms like user requirements, system requirements, functional requirements, etc.

3. Feasibility Study

After requirement gathering, with the help of many algorithms, the team analyzes that if the software can be designed to fulfil all requirements of the user and also analyzes if the project is financially, practically and technologically feasible for the organization or not.

4. System Analysis(A planning phase)

Software developer decides on a roadmap for their plan and tries to bring up the best software model stable for the project. System analysis may also include understanding product limitations and identifying and addressing the impact of the project on the organization. The project analyzes the scope of the project and plans the resources accordingly.

5. Software Design

Software design whole knowledge of requirements and analyses are taken together to plan up design of software products. It takes input from the user and information gathered in the requirement-gathering phase. It gives output in the form of logical and physical design.

6. Coding

This step is also known as the programming phase. The implementation of software design starts in the form of writing code in suitable programming and developing error-free programs efficiently.

7. Testing

Software testing is done while coding by the testers' developing team members. Testing is done at various levels i.e. module testing, product testing, program testing and user-end testing.

8. Integration

After writing all the codes for the software such as frontend, backend, and databases, The software is integrated with libraries, databases and other programs.

9. Implementation

In this step, the software product is finally ready to be installed on the user's machine. Software is tested for profitability, integration, adaptability, etc.

10. Operation and Maintenance

This phase confirms the software operations in terms of more efficiency and fewer errors. If required, the users are trained or aided with the documentation on how to operate the software and how they keep the software operational. This software is maintained timely by updating the code according to the changes taking place in the user and environment or technology.

Examples of System Software Developed

- Operating Systems (Windows, Linux, Android)
- Assemblers, Compilers, Interpreters
- Loaders & Linkers
- Device Drivers
- Database Management Systems

5. Recent Trends in System Software

System software is continuously evolving with new technologies.

Recent Developments

1. **Virtualization & Cloud Computing:**
    - Virtual machines and cloud OS allow efficient resource sharing.
    - Example: VMware, Docker, Kubernetes.
2. **Mobile Operating Systems:**
    - Android, iOS designed for smartphones and IoT devices.
3. **Security-Oriented OS:**
    - Focus on cybersecurity, intrusion detection, and safe execution.
4. **Open-Source System Software:**
    - Linux, FreeBSD, and other open-source OS are widely used.
5. **AI & Machine Learning Integration:**
    - Intelligent system software for **self-healing, prediction, and automation**.
6. **Edge & IoT Computing:**
    - Lightweight operating systems designed for smart devices (Raspberry Pi OS, TinyOS).

Language Processors

1. Programming Languages and Language Processing

Programming Languages

- A **programming language** is a formal language used to write computer programs.
- It provides a way for humans to give instructions to the computer.

Types of Programming Languages:

1. **Machine Language** – Binary code (0s & 1s), directly executed by CPU.

2. **Assembly Language** – Uses mnemonics (e.g., MOV A, B). Needs an **assembler**.
3. **High-Level Language (HLL)** – Close to human language (e.g., C, C++, Java, Python). Needs **compiler/interpreter**.

Language Processing

- The process of translating high-level or assembly language into **machine code** that hardware can understand.
- Done by **language processors**.

Types of Language Processors:

1. **Assembler** – Converts assembly code → machine code.
2. **Compiler** – Translates entire high-level program → machine code.
3. **Interpreter** – Translates and executes high-level program line by line.

2. Symbol Tables and Data Structures for Language Processing

Symbol Table

- A **symbol table** is a data structure used by language processors (compiler/assembler) to **store information about identifiers** (variables, functions, objects, labels).

Information stored in Symbol Table:

- Name of variable/function
- Type (int, float, char, etc.)
- Scope (local, global)
- Memory location (address)
- Value (if constant)

Functions of Symbol Table:

1. Stores information about all identifiers.
2. Helps in semantic analysis.
3. Supports code generation by mapping identifiers to memory addresses.

Data Structures for Language Processing

Language processors need efficient data structures for **storing and retrieving information** quickly.

Criteria for Classification of Data Structure of LP:

• 1. Nature of Data Structure: whether a “linear” or “non linear”.

• 2. Purpose of Data Structure: whether a “search” DS or an “allocation” DS.

• 3. Lifetime of a data structure: whether used during language processing or during target program execution.

1. **Linear Data Structures:**
    - Arrays, Linked Lists → Used for intermediate code storage and token handling.
2. **Non-linear Data Structures:**
    - Trees → Used in syntax analysis (Parse Tree, Abstract Syntax Tree).
    - Graphs → Used in code optimization and flow analysis.
3. **Hash Tables:**
    - Widely used for **symbol tables** because of fast insertion and searching.

3. Search and Allocation Data Structures

Search Data Structures

Used to **find identifiers or instructions** quickly during compilation.

- **Linear Search** – Simple but slow (used for small tables).
- **Binary Search** – Faster, requires sorted data.
- **Hashing** – Best method for symbol tables (O(1) average time).

Allocation Data Structures

Used by language processors to manage **memory allocation** for programs.

1. **Stack Allocation:**
    - Used for local variables and function calls.
    - Memory is allocated/deallocated in **LIFO (Last-In-First-Out)** order.
2. **Heap Allocation:**
    - Used for dynamic memory (malloc/free in C, new/delete in C++).
    - Variables exist until explicitly freed.
3. **Static Allocation:**
    - Memory allocated at **compile-time**.
    - Example: Global variables.

Diagram: Role of Language Processor

ASSEMBLERS AND MACRO PROCESSORS

---

1. Elements of Assembly Language Programming

Definition:

Assembly language is a **low-level programming language** that uses symbolic names (mnemonics) to represent machine instructions. It provides a way for programmers to write instructions that are closer to the hardware but easier to read than pure binary or hexadecimal code.

Key Elements:

1. **Mnemonics:**
   These are symbolic names for machine instructions.
   Example:
   - `MOV` for move
   - `ADD` for addition
   - `SUB` for subtraction
2. **Operands:**
   These specify the data to be operated on or the memory locations involved.
   Example: `ADD AX, BX` → Adds the contents of register BX to AX.
3. **Labels:**
   Used to identify a line of code, usually for jump or branch instructions.
   Example:

```
4. START: MOV AX, BX
```

5. **Directives:**
   Instructions to the assembler (not executed by CPU).
   Example:
   - `DB` – Define Byte
   - `DW` – Define Word
   - `ORG` – Origin (starting address)
6. **Comments:**
   Used to describe the purpose of instructions for readability.
   Example:

```
7. MOV AX, 05H   ; Load 5 into AX
```

8. **Symbol Table:**
   During assembly, the assembler creates a **symbol table** that keeps track of all labels and variables with their corresponding memory addresses.

---

2. Design of an Assembler

What is an Assembler?

An **assembler** is a system program that converts an **assembly language program** into **machine code** (object code).

Functions of an Assembler:

1. **Translation:** Converts mnemonics into machine instructions.
2. **Symbol Resolution:** Assigns addresses to all symbols and labels.
3. **Program Relocation:** Adjusts address references for relocatable programs.
4. **Error Handling:** Detects and reports syntax or semantic errors.
5. **Listing and Output Generation:** Produces a listing file and an object file.

---

3. Types of Assemblers

Assemblers are classified based on how they process the source code:

(a) One-Pass Assembler

- The entire assembly process is completed in **one scan** of the source program.
- Symbols must be defined before they are used.
- Faster but **cannot handle forward references**.
- **Used in small programs**.

Advantages:

- Fast translation.
- Requires less memory.

Disadvantages:

- Cannot handle forward references.

(b) Two-Pass Assembler

- The assembler scans the source code **twice**.

Pass 1:

- Builds the **symbol table**.
- Assigns addresses to all labels.
- Handles directives and computes addresses.

Pass 2:

- Generates **machine code** using the symbol table.
- Replaces symbolic names with actual addresses.

Advantages:

- Can handle forward references.
- Produces more accurate code.

Disadvantages:

- Slightly slower because of two passes.

---

4. Macro and Macro Processor

Definition of Macro:

A **macro** is a single instruction or name that represents a **sequence of instructions**. It helps in **reusing code** and **reducing repetition**.

Example:

```
MACRO INCR X
    ADD X, 1
ENDM
```

When the assembler sees `INCR A`, it replaces it with `ADD A, 1`.

Macro Processor:

A **macro processor** is a system program that **recognizes macro definitions and replaces** (expands) macro calls with the corresponding sequence of instructions before actual assembly.

5. Macro Definitions and Expansion

Macro Definition:

A macro is defined using `MACRO` and `ENDM` directives.

```
MACRO NAME [parameters]
   ; body of macro
ENDM
```

Macro Call:

When a macro is invoked (called), the assembler **expands** it by inserting the macro body in place of the macro name.

Example:

```
MACRO MULT2 ARG
    MOV AX, ARG
    ADD AX, AX
ENDM

MULT2 NUM
```

→ During macro expansion, this becomes:

```
MOV AX, NUM
ADD AX, AX
```

Nested Macros

Definition:

A **nested macro** occurs when one macro **calls another macro** within its definition.

Example:

```
MACRO MAC1
   MAC2
ENDM

MACRO MAC2
   MOV AX, BX
ENDM
```

When `MAC1` is called, it also expands `MAC2`.
Assemblers must carefully handle nesting to avoid **infinite expansion** or **ambiguity**.

---

7. Advanced Macro Features

(a) Parameters:

Macros can accept parameters, making them reusable with different arguments.
Example:

```
MACRO ADDNUM X, Y
   MOV AX, X
   ADD AX, Y
ENDM
```

(b) Default Arguments:

If parameters are not given, macros can have default values.
Example:

```
MACRO INCX X=1
   ADD AX, X
ENDM
```

(c) Conditional Macros:

Conditional assembly can be done using directives like `IF`, `ELSE`, `ENDIF`.
Example:

```
IF FLAG
   MOV AX, BX
ENDIF
```

(d) Macro Expansion Control:

The assembler can limit how deep nested expansions go or allow selective expansion.

---

8. Design of Macro Assemblers and Macro Processors

A **macro assembler** combines the functions of an assembler and a macro processor. It first expands all macros and then translates the program into machine code.

Steps in Macro Processing:

1. Macro Definition Processing:

- The macro definitions are stored in a **Macro Definition Table (MDT)**.
- Each macro name and parameters are stored in the **Macro Name Table (MNT)**.
- Parameters and their values are stored in **Argument Lists (ALA)**.

2. Macro Expansion:

When a macro call is found:

- The macro name is searched in the MNT.
- The corresponding definition is fetched from MDT.
- Arguments are replaced using the ALA.
- The expanded code is inserted into the source code.

3. Assembly of Expanded Code:

After all macros are expanded, the assembler translates the expanded code into object code.

Tables Used in Macro Processing:

Table Name	Purpose
MNT (Macro Name Table)	Stores macro names and pointers to MDT entries
MDT (Macro Definition Table)	Stores macro body (actual instructions)
ALA (Argument List Array)	Stores parameters and their corresponding arguments
EVT (Expansion Variable Table)	Used for handling expansion-time variables

9. Advantages of Using Macros

- **Saves time:** Frequently used code can be written once and reused.
- **Reduces errors:** Common code does not need to be retyped each time.
- **Improves readability:** Replaces long code blocks with meaningful names.
- **Easier modification:** Changes in the macro definition automatically apply everywhere.

10. Difference Between Macro and Subroutine

Feature	Macro	Subroutine
Definition	Sequence of instructions inserted during assembly	Separate block of code executed by a call
Execution	Expanded inline before assembly	Called and executed at runtime
Speed	Faster (no call overhead)	Slightly slower
Code Size	Larger (code repeated each time)	Smaller (only one copy)
Parameters	Passed as arguments during expansion	Passed using registers or stack

LINKERS AND LOADERS

---

1. Introduction

When a program is written in a high-level language (like C, C++, or Java), it must go through several stages before execution.
The **compiler** translates the source code into **object code**, which still may not be directly executable.
To make it executable, the system uses **linkers** and **loaders**.

- **Linker:** Combines multiple object files and resolves references between them.
- **Loader:** Loads the linked program into main memory for execution.

Both are essential parts of the **program translation and execution process**.

---

2. Concept of Linking

Definition:

Linking is the process of **combining multiple object modules** (produced by the compiler or assembler) into a single executable program.

Each source file may contain functions, variables, or procedures that refer to other files.
The linker resolves these **external references**.

Example:

Let's say:

- `main.c` calls a function `sum()` defined in `math.c`
- The compiler generates:
  - `main.obj` (object file for main)
  - `math.obj` (object file for math)

The **linker** combines them:

```
main.obj + math.obj → program.exe
```

Functions of the Linker:

1. **Combines Object Modules:**
   Joins multiple `.obj` files into one executable.
2. **Symbol Resolution:**
   Matches function and variable names used across modules.
3. **Address Binding:**
   Assigns absolute memory addresses to program segments.
4. **Library Linking:**
   Links required system or user-defined libraries (like `stdio.h` or math libraries).
5. **Relocation:**
   Adjusts addresses when modules are moved in memory.

---

3. Concept of Relocation

Definition:

Relocation is the process of **modifying address-dependent code and data** in a program so it can be correctly executed from a different memory location than the one originally assumed.

Need for Relocation:

- When the exact memory location for execution is not known at compile time.
- When multiple programs share memory space.
- When dynamic memory allocation or paging is used.

Example:

If a program assumes it will start at address 1000 but the loader places it at address 3000, every instruction and data address must be increased by **(3000 − 1000) = 2000**.

This is done by the **relocation loader or linker**.

---

4. Linking in MS-DOS

In **MS-DOS**, the linking process is handled by the **LINK.EXE** utility.

Steps:

1. **Compilation:**
   Each `.c` file is compiled into an **object file (.OBJ)** using the compiler:

```
C:\> CL MAIN.C
```

3. **Linking:**
   The object file is linked using the linker:

```
C:\> LINK MAIN.OBJ
```

   The linker produces:

   - **Executable file (.EXE)** → the final runnable program.
   - **Map file (.MAP)** → lists memory layout and address assignments.

5. **Execution:**
   The executable file is loaded and executed in memory by the loader:

```
C:\> MAIN.EXE
```

Characteristics of Linking in MS-DOS:

- Static linking (functions combined at compile time).
- Relocation records maintained by linker.
- Symbol resolution done before execution.

---

5. Dynamic Linking

Definition:

Dynamic linking is a technique where linking of program modules is **performed at runtime**, rather than during compilation or assembly.

Features:

- Modules (like libraries) are linked **only when needed**.
- Saves memory because multiple programs can share the same library in memory.
- Common in modern operating systems like **Windows, Linux**, etc.

Example:

- Shared libraries in Windows → `.DLL` (Dynamic Link Library)
- Shared libraries in Linux → `.SO` (Shared Object)

Advantages:

1. **Memory Efficient:** Shared code is loaded only once.
2. **Easy Updates:** If a library is updated, all programs using it get the new version automatically.
3. **Faster Loading:** Initial loading time reduced since not all modules are loaded upfront.

Disadvantages:

- Slightly slower execution when functions are first called.
- Programs depend on the external shared libraries being present.

---

6. Loading Schemes

Loading means placing a program into memory for execution.
A **loader** is the system program responsible for this task.

There are various **loading schemes**, depending on how the addresses are managed.

(a) Sequential Loading

- The program is loaded into memory **from the first address available**.
- Instructions and data are placed sequentially.
- No relocation or linking is needed.

Use: Simple embedded or batch systems.

Example:
If memory starts at 1000, program is loaded at 1000, 1001, 1002, etc.

(b) Direct Loading

- The loader reads **absolute addresses** directly from the object file.
- The programmer or assembler must know where the program will be loaded.

- The program cannot be moved to another memory location.

Advantage: Fast loading.
Disadvantage: No flexibility, not relocatable.

(c) Absolute Loader

- The loader places the object code into **specific memory locations** as indicated in the object program.
- No relocation or linking is done.

Steps:

1. Read the object file.
2. Place instructions into specified memory addresses.
3. Transfer control to the start address.

Limitation:
Cannot handle programs that refer to external symbols or multiple modules.

Example:
Used in **simple microcontrollers** and early assembly systems.

(d) Relocating Loader

- Adjusts the addresses of instructions and data while loading to match the available memory space.
- Performs **relocation** using relocation information from the linker.

Features:

- Can load the same program at different memory locations.
- Supports **multiprogramming** (several programs in memory simultaneously).

Example:
If program expects to start at 1000 but is loaded at 2000, the loader adds 1000 to all address-sensitive instructions.

(e) Linking Loader

- Performs both **linking and loading** at the same time.
- It reads multiple object modules, links them, relocates them, and loads them into memory.

Steps:

1. Read all object files.
2. Resolve external symbols (linking).
3. Adjust addresses (relocation).
4. Load into memory (loading).
5. Start execution.

Advantages:

- No separate linker needed.
- More efficient memory management.

Example: Used in **modern operating systems** and **language runtime systems**.

---

7. Comparison of Linkers and Loaders

Feature	Linker	Loader
Definition	Combines multiple object modules into a single executable file	Loads the executable program into main memory
Main Function	Linking and relocation	Memory allocation and program execution
Input	Object files (.obj)	Executable file (.exe)
Output	Executable file (.exe)	Program in memory
Time of Operation	Before execution (compile time)	At execution time (runtime)
Relocation	Done by linker or passed to loader	Done by relocating loader
Symbol Resolution	Yes	No
Example Utility	LINK.EXE (MS-DOS)	LOAD.EXE, part of OS

---

8. Summary of All Concepts

Concept	Description
Linking	Combining multiple object modules into a single executable
Relocation	Adjusting addresses when the program is moved to a new memory location
Dynamic Linking	Linking done at runtime using shared libraries
Loader	Places executable program into memory for execution
Sequential Loader	Loads program sequentially in memory
Absolute Loader	Loads code at fixed addresses (no relocation)
Relocating Loader	Adjusts addresses based on new memory location
Linking Loader	Performs both linking and loading at runtime

---

9. Real-World Example

When you compile and run a program in C:

1. **Compilation:**

```
gcc main.c add.c -c
```

   → Produces object files `main.o` and `add.o`.

3. **Linking:**

```
gcc main.o add.o -o program
```

   → Linker combines them into an executable.

5. **Loading:**

```
./program
```

   → Loader loads it into memory, relocates addresses, and starts execution.

---

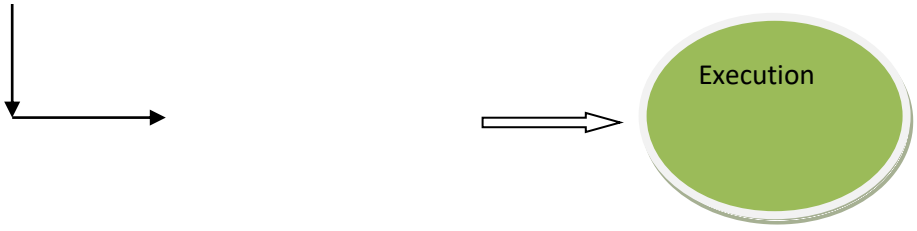
10. Advantages of Linkers and Loaders

Advantages of Linker:

- Simplifies program development by allowing modular programming.
- Supports use of libraries and reusable code.
- Handles symbol resolution and relocation efficiently.

Advantages of Loader:

- Provides flexibility in program placement.
- Supports dynamic memory management.
- Makes program execution easier and faster.



UNIT V: System Programming

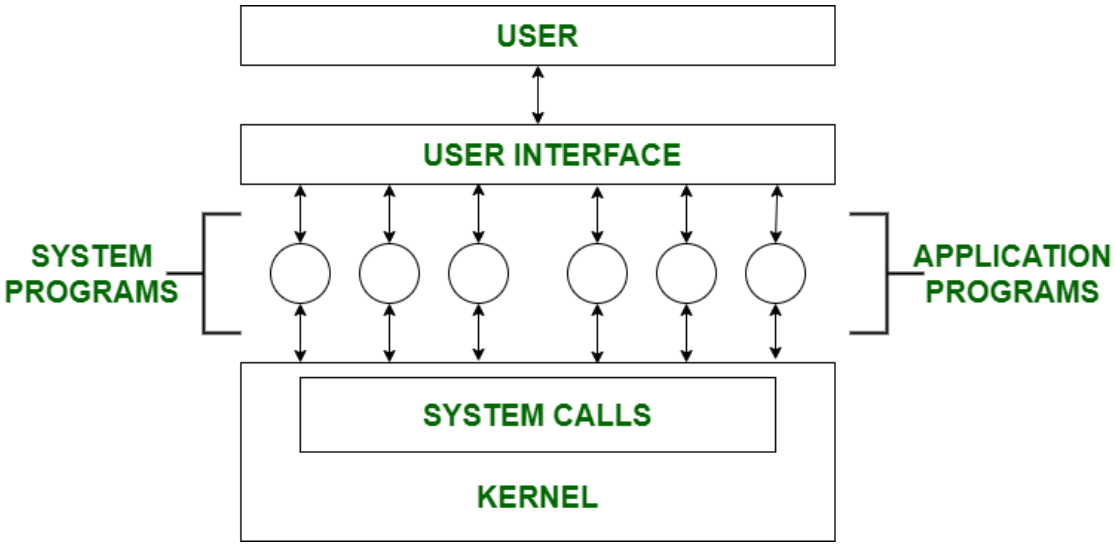
Scanning and Parsing: Programming Language Grammars and Classification, Ambiguity in Grammatic Specification, Scanning, Parsing,

Compilers and Interpreters: Compilation Process, Semantic Gap, Binding, and Scope Rules, Memory Allocation, Compilation of Expressions & Control Structures, Code Optimization, Overview of Interpreters and Debuggers,

Operating System Command & Shell Basics: C Development Tools, Machine-Level Representation of Data and Programs,

System-Level Programming and Concurrency: File I/O, Process Creation & Control (fork, exec), Pipes, Signals, and Basic Threading.

System Programming can be defined as the act of building Systems Software using System Programming Languages. According to Computer Hierarchy, Hardware comes first then is Operating System, System Programs, and finally Application Programs.
In the context of an operating system, system programs are nothing but a special software which give us facility to manage and control the computer's hardware and resources.
As we have mentioned earlier these programs work more closely with the operating system so it executes the operation fast and helpful in performing essential operation which can't be handled by application software.



UNIT: Scanning and Parsing

1. Programming Language Grammars and Classification

Definition:

A **grammar** is a set of rules that define the **syntactic structure** of a programming language. It tells how statements and expressions are formed.

Components of a Grammar (in compiler design):

A grammar **G** can be represented as a 4-tuple:
G = (V, T, P, S)
where,

- **V** – Variables (non-terminals)
- **T** – Terminals (symbols or tokens)
- **P** – Production rules
- **S** – Start symbol

Example Grammar:

```
E → E + T | T
T → T * F | F
;F → (E) | id
```

 This grammar defines arithmetic expressions.

Classification of Grammars (Chomsky Hierarchy):

Type	Name	Example Language	Description
Type 0	Unrestricted Grammar	Turing Machine languages	No restriction on production rules
Type 1	Context-Sensitive Grammar	$L = \{a^n b^n c^n\}$	Rules depend on context
Type 2	Context-Free Grammar (CFG)	Programming languages	Used for parsing and syntax analysis
Type 3	Regular Grammar	Regular expressions	Used in lexical analysis (scanning)

Use in Compiler:

- **Regular Grammar** → Used by **Scanner (Lexical Analyzer)**
- **Context-Free Grammar** → Used by **Parser**

2. Ambiguity in Grammatic Specification

Definition:

A grammar is said to be **ambiguous** if there exists **more than one parse tree** (or derivation) for the same string.

Example:

Grammar:

```
E → E + E | E * E | id
```

String: `id + id * id`

 Two possible parse trees:

1. `id + (id * id)`
2. `(id + id) * id`

Hence, **grammar is ambiguous**.

Problems Caused by Ambiguity:

- Confusing compiler behavior.
- Multiple interpretations of the same statement.
- Difficulty in syntax analysis.

Solution to Ambiguity:

- Modify grammar to make it **unambiguous**.
- Use **operator precedence** and **associativity rules**.

Example (Unambiguous Grammar):

```
E → E + T | T
T → T * F | F
F → id
```

Now `*` has higher precedence than `+`.

---

3. Scanning (Lexical Analysis)

Definition:

Scanning or **Lexical Analysis** is the **first phase** of a compiler.
It converts **source code** into a sequence of **tokens**.

Main Functions:

1. **Remove whitespaces and comments**
2. **Identify tokens** (keywords, identifiers, literals, operators, etc.)
3. **Generate symbol table entries**

Token Example:

For input:

```
sum = a + b;
```

Tokens generated are:
`<id, sum>, <=, assign>, <id, a>, <+, plus>, <id, b>, <;, semicolon>`

Lexical Analyzer Diagram:

```
Source Program → [Lexical Analyzer] → Tokens
                          ↓
                    Symbol Table
```

Tool Used:

Lex / Flex (Lexical Analyzer Generator)

---

4. Parsing (Syntax Analysis)

Definition:

Parsing is the **second phase** of the compiler.
It checks whether the sequence of tokens follows the **grammar rules** of the programming language.

Purpose:

- To ensure the syntax of the source code is correct.
- To build a **parse tree** or **syntax tree**.

Phases Relationship:

```
Source Code → Scanner → Tokens → Parser → Parse Tree
```

Types of Parsers:

Parser Type	Technique	Direction	Example

Parser Type	Technique	Direction	Example
Top-Down Parser	Start from root (start symbol) and try to reach input string	Left to right	Recursive Descent, LL Parser
Bottom-Up Parser	Start from input symbols and try to reach the start symbol	Left to right	Shift-Reduce, LR Parser

---

Parse Tree

Definition:

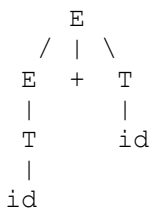
A **parse tree** is a tree representation showing how a string is derived from the start symbol using grammar rules.

Example Grammar:

```
E → E + T | T
T → id
```

For input: `id + id`

Parse Tree:



Unit: Compilers and Interpreters

1. Introduction

A **compiler** and an **interpreter** are both language processors that convert **high-level programs** (like C, Java, Python) into **machine code** that the computer can understand.
However, they differ in **how** and **when** they perform this translation.

2. Compiler vs Interpreter

Feature	Compiler	Interpreter
Translation	Converts the entire program into machine code at once	Translates one line at a time
Execution speed	Faster (code is already compiled)	Slower (translates while executing)
Output	Produces an executable file	No separate executable file
Error handling	Shows all errors after full compilation	Stops immediately when an error occurs
Example Languages	C, C++, Java	Python, BASIC, JavaScript

Example:

Code:

```
print("Hello World")
```

- In **compiler-based languages (C, C++)**, the entire program is compiled first before execution.
- In **interpreted languages (Python)**, each line is executed immediately.

---

3. Compilation Process

The **compilation process** involves several **phases**, each transforming the program step-by-step.

Phases of Compilation:

1. **Lexical Analysis (Scanning):**
   - Breaks source code into **tokens** (keywords, identifiers, symbols).
   - Example:
   - `a = b + 5;`

     Tokens → `a`, `=`, `b`, `+`, `5`, `;`
2. **Syntax Analysis (Parsing):**
   - Checks whether the tokens form a valid **syntax** according to grammar rules.
   - Example: Detects missing semicolon or wrong order of operators.
3. **Semantic Analysis:**

- Ensures **meaning** of the program is correct.
- Example: Checks variable types.
- `int a;`
- `a = "hello"; // Error: type mismatch`

4. **Intermediate Code Generation:**
   - Creates an **intermediate code** between source and machine code.
   - Example:
   - `t1 = b + 5`
   - `a = t1`
5. **Code Optimization:**
   - Improves the efficiency of the intermediate code (reduces redundancy).
   - Example:
     Before:
   - `t1 = b + 0`
   - `a = t1`

     After Optimization:

     `a = b`

6. **Code Generation:**
   - Converts optimized code into **machine code**.
7. **Code Linking and Loading:**
   - Combines all program parts and loads them into memory for execution.

---

4. Semantic Gap

Definition:

The **semantic gap** is the **difference between how humans think and write code** and how **machines understand instructions**.

- High-level languages (like C, Java) are **closer to human thinking**.
- Machine language is **binary**, hard for humans to understand.

Example:

```
sum = a + b;
```

- Human meaning: Add two variables.
- Machine view: Perform specific load, add, and store operations in registers.

Reducing the Gap:

Compilers and interpreters **bridge this gap** by translating human-friendly code into machine instructions.

---

5. Binding and Scope Rules

Binding:

Binding refers to **associating** program elements (like variables, functions) with their properties (like type, value, location).

Types of Binding:

1. **Static Binding (Early Binding):**
   Done at compile time.
   Example:

```
int x = 10;  // data type and value known at compile time
```

2. **Dynamic Binding (Late Binding):**
   Done at run time.
   Example:

```
Shape s = new Circle();  // Actual type known at runtime
```

Scope Rules:

Scope defines **where a variable can be accessed** in a program.

Types:

1. **Local Scope:** Variable declared inside a function or block.
   Example:

```
void fun() {
    int a = 10;  // local variable
}
```

2. **Global Scope:** Variable declared outside all functions.

```
int a = 20;  // global variable
```

Rule:
Local variables **override** global variables if they share the same name.

---

6. Memory Allocation

Memory allocation decides **where variables and data are stored** during program execution.

Types:

1. **Static Allocation:**
    - Done at **compile time**.
    - Example: Global variables, constants.
2. **Stack Allocation:**
    - Used for **local variables** and **function calls**.
    - Allocated when function is called, freed when function ends.
3. **Heap Allocation:**
    - Done **dynamically** at runtime using pointers.
    - Example (C language):
    - `int *p = (int*) malloc(sizeof(int));`

---

7. Compilation of Expressions and Control Structures

Expressions:

The compiler converts arithmetic expressions into machine instructions.

Example:

```
x = a + b * c;
```

Intermediate Code:

```
t1 = b * c
x = a + t1
```

Control Structures:

Compilers convert loops and conditionals into jumps and branches.

Example:

```
if (x > y)
    max = x;
else
    max = y;
```

Intermediate Code:

```
if x <= y goto L1
max = x
goto L2
L1: max = y
L2:
```

Code Optimization

Definition:

Code optimization makes the compiled program **run faster** or **use less memory** without changing its output.

Examples of Optimizations:

1. **Constant Folding:**
   Replace constant expressions with their results.
   Example:
   `a = 5 * 4;` → `a = 20;`
2. **Dead Code Elimination:**
   Remove code that never executes.
   Example:

   ```
   if (0) { ... }  // Removed
   ```

3. **Common Subexpression Elimination:**
   Avoid repeating calculations.
   Example:

   ```
   a = b * c;
   d = b * c + e;
   ```

   →

   ```
   t = b * c;
   a = t;
   d = t + e;
   ```

---

9. Overview of Interpreters and Debuggers

Interpreters:

- Translate **line by line** and **execute immediately**.

- Useful for **testing** and **rapid development**.
- Example: **Python Interpreter (IDLE)**

Debuggers:

- Help programmers **find and fix errors**.
- Features:
  - Breakpoints
  - Step-by-step execution
  - Variable watching

Example:
In C/C++ → **GDB (GNU Debugger)**
In Java → **Eclipse Debugger**

Diagram: Compilation Process

```
Source Code
     ↓
Lexical Analysis
     ↓
Syntax Analysis
     ↓
Semantic Analysis
     ↓
Intermediate Code Generation
     ↓
Code Optimization
     ↓
Code Generation
     ↓
Executable Code
```

□ UNIT: Operating System Command & Shell Basics

1. Introduction

An **Operating System (OS)** provides an interface between the **user and computer hardware**. One of the main ways users interact with an OS is through the **Command Line Interface (CLI)** or **Shell**.

A **Shell** is a command interpreter that executes commands entered by the user.

Examples of Shells:

- **Bash (Bourne Again Shell)** – Linux default shell
- **PowerShell** – Windows shell
- **Zsh, Ksh, Tcsh** – other Unix/Linux shells

---

2. Shell Basics

2.1 What is a Shell?

A **shell** acts as a **middle layer** between the user and the kernel.
It takes commands from the user and passes them to the kernel for execution.

Diagram:

```
User → Shell → Kernel → Hardware
```

k2.2 Common Shell Commands

Command	Meaning	Example
`pwd`	Show current working directory	`pwd`
`ls`	List files in directory	`ls -l`
`cd`	Change directory	`cd /home/user`
`mkdir`	Create new directory	`mkdir project`
`rm`	Remove files	`rm file.txt`
`cp`	Copy file	`cp a.txt b.txt`
`mv`	Move or rename file	`mv old.txt new.txt`
`cat`	Display contents of file	`cat file.txt`
`grep`	Search text in files	`grep "main" program.c`
`echo`	Print message	`echo Hello`

---

3. C Development Tools C Development Tools

---

1. Introduction

C Development Tools are the software components and utilities that help programmers **write, compile, debug, and run** C programs efficiently.

They provide everything required to convert human-readable C code into machine-executable programs.

---

2. Components of a C Development Environment

A **C development environment** generally includes:

Tool	Description	Example
Editor	Used to write C source code	Notepad, VS Code, Code::Blocks
Compiler	Translates C code to machine code	GCC, Turbo C, Clang
Assembler	Converts assembly code to object code	NASM, GAS
Linker	Combines multiple object files into a single executable	`ld` (GNU linker)
Loader	Loads the executable into memory for execution	OS Loader
Debugger	Helps to test and remove errors	GDB, Turbo Debugger

3. Diagram – Phases of C Program Development

| Source | ---> | Compiler | ---->| Assembler| ---->| Linker | ---->| Loader |
| Code (.c)| | (.obj/.o) | | (.obj) | | (.exe) | | (in RAM) |

4. Major C Development Tools

a) Text Editors

Used for writing and saving C source code files (`.c` extension).

Examples:

- Notepad, Sublime Text, Atom, VS Code, Vim

Features:

- Syntax highlighting
- Auto-completion
- Code indentation

---

b) Compiler

A **compiler** converts the C program into **object code** (machine-understandable format).

Popular C Compilers:

- **GCC (GNU Compiler Collection)** – Linux/Windows
- **Turbo C** – Legacy Windows IDE
- **Clang** – Fast modern compiler for macOS/Linux

Compilation Command (GCC Example):

```
gcc program.c -o program
```

➡ Creates an executable file named `program`.

---

c) Assembler

- The compiler first generates **assembly code**, which the **assembler** translates into **object code**.
- Converts **mnemonics** (assembly instructions) to **binary machine code**.

Example:
Assembly instruction `MOV AX, BX` → Machine code `89 D8`

---

d) Linker

- Combines **multiple object files** and required **library files** to produce a single executable program.
- Resolves **external references** (like `printf()` from standard libraries).

Example:
If your program uses `printf()`, the linker includes code from the **C standard library (libc)**.

---

e) Loader

- Loads the **executable file** into **main memory (RAM)**.
- Assigns memory to code and data segments and starts program execution.

---

f) Debugger

- Used to **detect and fix errors** (bugs) in programs.
- Allows step-by-step execution, setting breakpoints, and checking variable values.

Example Debugger:

- GDB (GNU Debugger)
- Turbo Debugger

Common GDB Commands:

```
gdb program
(gdb) break main
(gdb) run
(gdb) print x
(gdb) next
(gdb) quit
```

---

5. Popular Integrated Development Environments (IDEs)

IDE	Platform	Features
Turbo C/C++	Windows	Simple, used for learning
Code::Blocks	Cross-platform	Modern GUI, debugger, compiler integration
Dev-C++	Windows	Lightweight and fast

IDE	Platform	Features
Visual Studio Code (VS Code)	Cross-platform	Extensions for C/C++, Git integration
Eclipse CDT	Cross-platform	Industrial-grade IDE
Xcode	macOS	Used for Apple development

---

6. Workflow of a C Program

Example Program: `hello.c`

```
#include <stdio.h>

int main() {
    printf("Hello, System Programming!\n");
    return 0;
}
```

Compilation Steps:

Step	Command	Description
Preprocessing	`gcc -E hello.c -o hello.i`	Expands macros and includes header files
Compilation	`gcc -S hello.i -o hello.s`	Converts C code to assembly
Assembly	`gcc -c hello.s -o hello.o`	Converts assembly to object code
Linking	`gcc hello.o -o hello`	Creates final executable file

---

7. Errors in C Development

Error Type	Meaning	Example
Syntax Error	Mistake in code syntax	Missing semicolon
Linker Error	Missing function definition	Undefined reference to `printf`
Runtime Error	Occurs during program execution	Division by zero

Error Type	Meaning	Example
Logical Error	Incorrect logic	Wrong formula used

---

8. Advantages of Using Development Tools

- Faster program creation and debugging
- Efficient memory and resource management
- Easier code navigation
- Integrated testing and version control support

---

9. Summary Table

Tool	Function	Example
Editor	Write code	VS Code
Compiler	Translate code	GCC
Assembler	Convert assembly to object code	GAS
Linker	Combine object files	ld
Loader	Load program into memory	OS Loader
Debugger	Find and fix errors	GDB

---

10. Real-Time Example (GCC in Action)

```
# Step 1: Create source code
nano test.c

# Step 2: Compile
gcc test.c -o test

# Step 3: Run
./test
```

 Output:

```
Hello, System Programming!
```

---

11. Conclusion

C Development Tools form the foundation for **system-level and application-level programming**.
Understanding these tools helps students learn how **source code becomes executable**, manage **errors**, and optimize performance.

4. Machine-Level Representation of Data and Programs

4.1 Data Representation

All data (numbers, characters, instructions) are represented in **binary (0s and 1s)** inside the computer.

Data Type	Representation Example
Integer	`10` → `00001010` (binary)
Character	`'A'` → `65` → `01000001`
Float	Represented using IEEE-754 format

4.2 Program Representation

When a program is compiled:

1. **Source Code (.c)** → Human-readable
2. **Assembly Code (.s)** → Low-level instructions
3. **Object Code (.o)** → Machine-readable binary
4. **Executable File (.out)** → Ready for execution

Compilation Process:

```
Source Code → Compiler → Object Code → Linker → Executable
```

System-Level Programming

System-level programming involves writing programs that **interact directly with the operating system** using **system calls** (like file handling, process control, etc.).

6. File I/O in System Programming

System calls used for file operations are lower-level than standard library functions (`fopen`, `fclose`, etc.).

Important System Calls:

Function	Purpose
`open()`	Open a file
`read()`	Read data from a file
`write()`	Write data to a file
`close()`	Close the file

Example: File Write and Read

```
#include <fcntl.h>
#include <unistd.h>

int main() {
    int fd;
    char buffer[100];

    // Open or create file
    fd = open("demo.txt", O_CREAT | O_WRONLY, 0644);
    write(fd, "Hello, OS Programming!", 22);
    close(fd);

    // Read file
    fd = open("demo.txt", O_RDONLY);
    read(fd, buffer, 22);
    write(1, buffer, 22); // write to standard output
    close(fd);

    return 0;
}
```

Explanation:

- `O_CREAT` → Create file if not exist
- `O_WRONLY` → Open in write mode
- `0644` → File permission bits

7. Process Creation and Control

A **process** is an executing instance of a program.
Linux provides **system calls** like `fork()` and `exec()` to create and control processes.

7.1 fork() System Call

- Used to **create a new process (child process)**.
- After `fork()`, two processes run concurrently:
  - **Parent process**
  - **Child process**

Example:

```
#include <stdio.h>
#include <unistd.h>

int main() {
    int pid = fork();

    if (pid == 0)
        printf("Child Process\n");
    else
        printf("Parent Process\n");

    return 0;
}
```

Output:

```
Parent Process
Child Process
```

(order may vary)

7.2 exec() System Call

- Used to **replace** the current process with a new program.

Example:

```
#include <unistd.h>

int main() {
    char *args[] = {"ls", "-l", NULL};
    execvp("ls", args);
    return 0;
}
```

Explanation:
The current program is replaced by `ls -l`.

---

8. Pipes (Inter-Process Communication)

A **pipe** allows two processes to communicate by reading/writing through a shared buffer.

Diagram:

```
Process 1 → [Pipe] → Process 2
```

Example:

```
#include <unistd.h>
#include <string.h>

int main() {
    int fd[2];
    char msg[] = "Hello Pipe";
    char buffer[20];

    pipe(fd); // create pipe

    if (fork() == 0) {
        // Child reads
        read(fd[0], buffer, sizeof(buffer));
        printf("Child received: %s\n", buffer);
    } else {
        // Parent writes
        write(fd[1], msg, strlen(msg)+1);
    }
    return 0;
}
```

---

9. Signals

Signals are software interrupts sent to a process to notify it of an event.

Common Signals:

Signal	Meaning
`SIGINT`	Interrupt (Ctrl + C)
`SIGKILL`	Forcefully kill process
`SIGTERM`	Termination request
`SIGSEGV`	Segmentation fault

Example: Handling Signals

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void handler(int signum) {
    printf("Caught signal %d\n", signum);
}

int main() {
    signal(SIGINT, handler);
    while(1) {
        printf("Running...\n");
        sleep(1);
    }
    return 0;
}
```

Explanation:
When user presses **Ctrl + C**, it prints "Caught signal 2" instead of exiting.

---

10. Basic Threading

A **thread** is a lightweight sub-process that runs within the same memory space of a program.

Advantages of Threads:

- Faster context switching
- Shared memory
- Efficient for parallel tasks

Example:

```
#include <pthread.h>
#include <stdio.h>

void* printMsg(void* arg) {
    printf("Hello from thread!\n");
    return NULL;
}

int main() {
    pthread_t t1;
    pthread_create(&t1, NULL, printMsg, NULL);
    pthread_join(t1, NULL);
    printf("Thread finished.\n");
    return 0;
}
```

Output:

```
Hello from thread!
Thread finished.
```