Concepts of Operating Systems

- Vineela

Session 1 : Introduction to OS Lecture

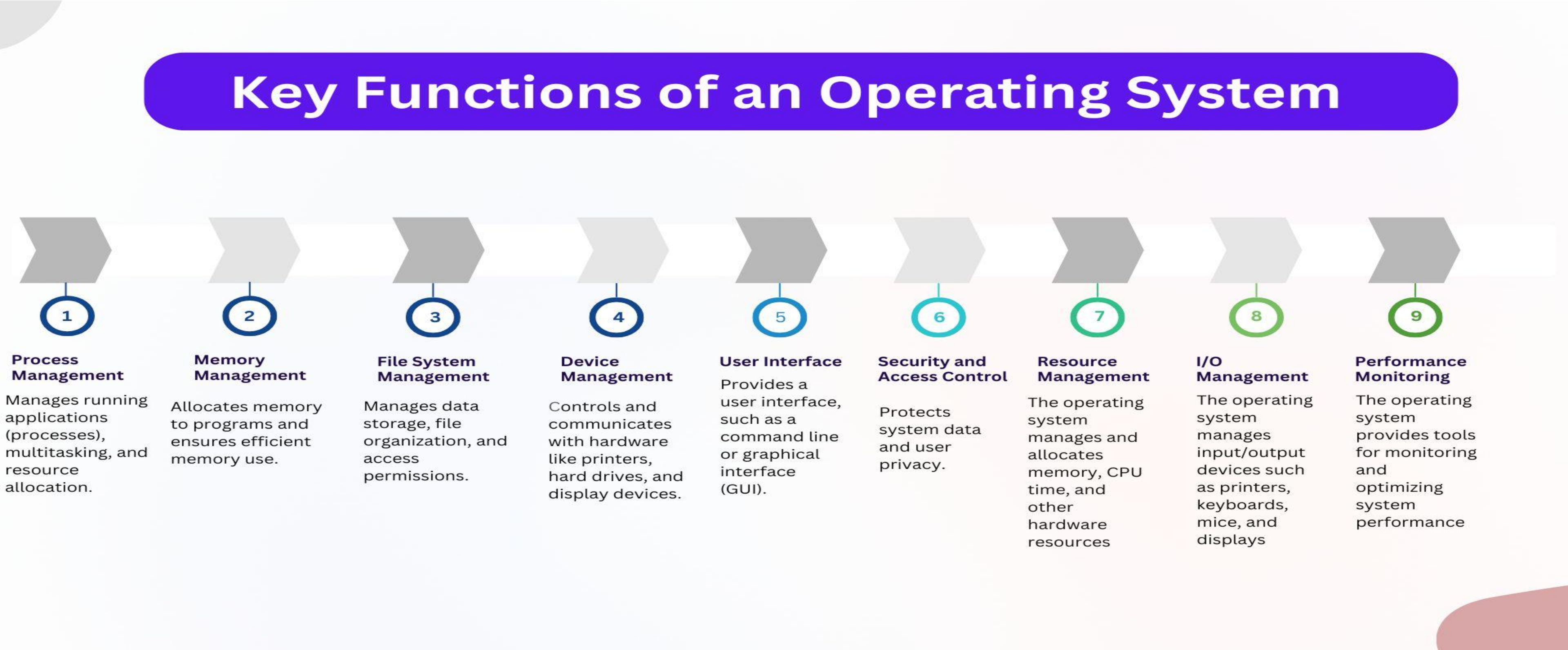
Lecture:

- What is OS; How is it different from other application software; Why is it hardware dependent?
- Different components of OS
- Basic computer organization required for OS.
- Examples of well-known OS including mobile OS, embedded system OS, Real Time OS, desktop OS server machine OS etc. ; How are these different from each other and why
- Functions of OS
- User and Kernel space and mode; Interrupts and system calls

What is OS; How is it different from other application software; Why is it hardware dependent?

- **What is OS?**

  It is **system software that manages computer hardware, software resources and provides common services for computer programs**.



Key Functions of an Operating System

1. **Process Management** – Manages running applications (processes), multitasking, and resource allocation.
2. **Memory Management** – Allocates memory to programs and ensures efficient memory use.
3. **File System Management** – Manages data storage, file organization, and access permissions.
4. **Device Management** – Controls and communicates with hardware like printers, hard drives, and display devices.
5. **User Interface** – Provides a user interface, such as a command line or graphical interface (GUI).
6. **Security and Access Control** – Protects system data and user privacy.
7. **Resource Management** – The operating system manages and allocates memory, CPU time, and other hardware resources
8. **I/O Management** – The operating system manages input/output devices such as printers, keyboards, mice, and displays
9. **Performance Monitoring** – The operating system provides tools for monitoring and optimizing system performance

How is OS different from other application software

Aspect	Operating System (OS)	Application Software
Purpose and Functionality:	An OS is system software designed to manage hardware and provide a platform for running other software applications. It controls and coordinates the use of hardware resources (like CPU, memory, and storage), ensuring the efficient and safe operation of a computer. **Examples** - Windows, macOS, Linux, Android, etc.	Application software is designed to perform specific tasks or solve particular problems for users. It **runs on top of an operating system** and interacts with it, but it does not manage hardware or control the system's core functions. **Examples** - Microsoft Word, Google Chrome, Photoshop etc.
Interaction with Hardware:	Directly interacts (**at a system level)** with the hardware and serves as an intermediary between the hardware and application software. It manages hardware resources like memory, processor, storage, and devices (e.g., printers, USB drives,task manager).	It operates **at the user level**, with a direct focus on the specific task or use case for the end-user. **For example**, when you open a document in Word, the application relies on the OS to handle memory and file management.

- **In short, an OS is like the foundation of a house, and application software is the furniture and decor placed on that foundation.**

Is OS is hardware dependant or not?

Is OS is hardware dependant or not?

Yes ,because it directly interacts and manages the hardware components of a computer or device

What are the Aspects of hardware dependency?

Hardware-Specific Drivers:

- Each type of hardware (such as the CPU, memory, storage devices, network interfaces, printer, and input devices (keyboard, mouse) needs **device drivers** tailored for it, so the OS must be designed to support the hardware it's running on.

Resource Management:

- OS allocates and manages **hardware resources** like CPU time, memory, and storage, so the OS must be aware of and adapt to the system's specific hardware resources.

CPU Architecture:

- A program written for an **x86** processor won't run on an **ARM** processor without specific adaptation, and vice versa. This means an **OS must be compiled for the particular type of CPU** used by the system.

Memory and Storage Management:

- Different systems can have different amounts and types of memory or storage (e.g., SSD vs. HDD), so It must be able to **detect, allocate, and deallocate memory** and ensure that different programs don't interfere with each other's memory spaces.

What are the Aspects of hardware dependency?

I/O Management

- The OS handles all **I/O operations** (like reading from or writing to files, input from a keyboard, or output to a monitor). and require specific interfaces and methods for communication, which the OS needs to manage in a hardware-dependent way.

Power Management:

- OS must control **how power is distributed to different components** (CPU, screen, wireless network, etc.) and this depends on the hardware capabilities of the device.

System Architecture and Customization:

- OS has to be **customized for each of the hardware environments of different devices** (like smartphones, desktops, and servers) and different hardware configurations, such as the number of cores in the CPU, types of sensors, and GPU types to take full advantage of the hardware features and optimize performance.

Different Components of OS

Process Management

- Handles the execution of processes.
- Manages CPU scheduling, process creation, and termination.
- Ensures smooth multitasking and prevents deadlocks.

Memory Management

- Allocates and deallocates memory for processes.
- Manages virtual memory and paging.
- Ensures efficient use of RAM.

File System Management

- Organizes and manages files and directories.
- Handles file permissions and access control.
- Provides storage management.

I/O Device Management

- Manages input and output devices like keyboards, printers, and displays.
- Uses device drivers to communicate with hardware.
- Ensures efficient data transfer.

Security & Access Control

- Protects data and system resources from unauthorized access.
- Implements authentication and encryption mechanisms.
- Prevents malware and cyber threats.

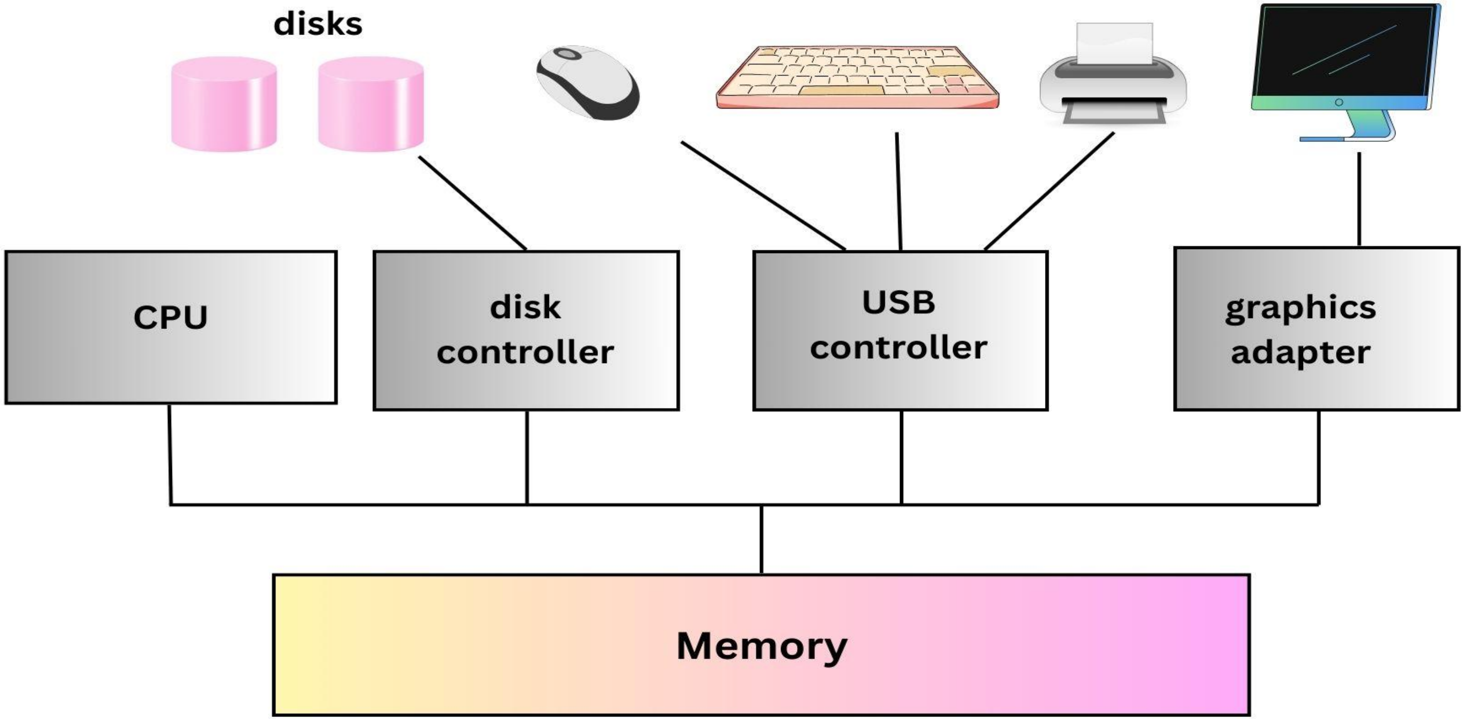
Network Management

- Manages communication between devices over a network.
- Handles protocols, data transmission, and connectivity.
- Supports internet and local networking.

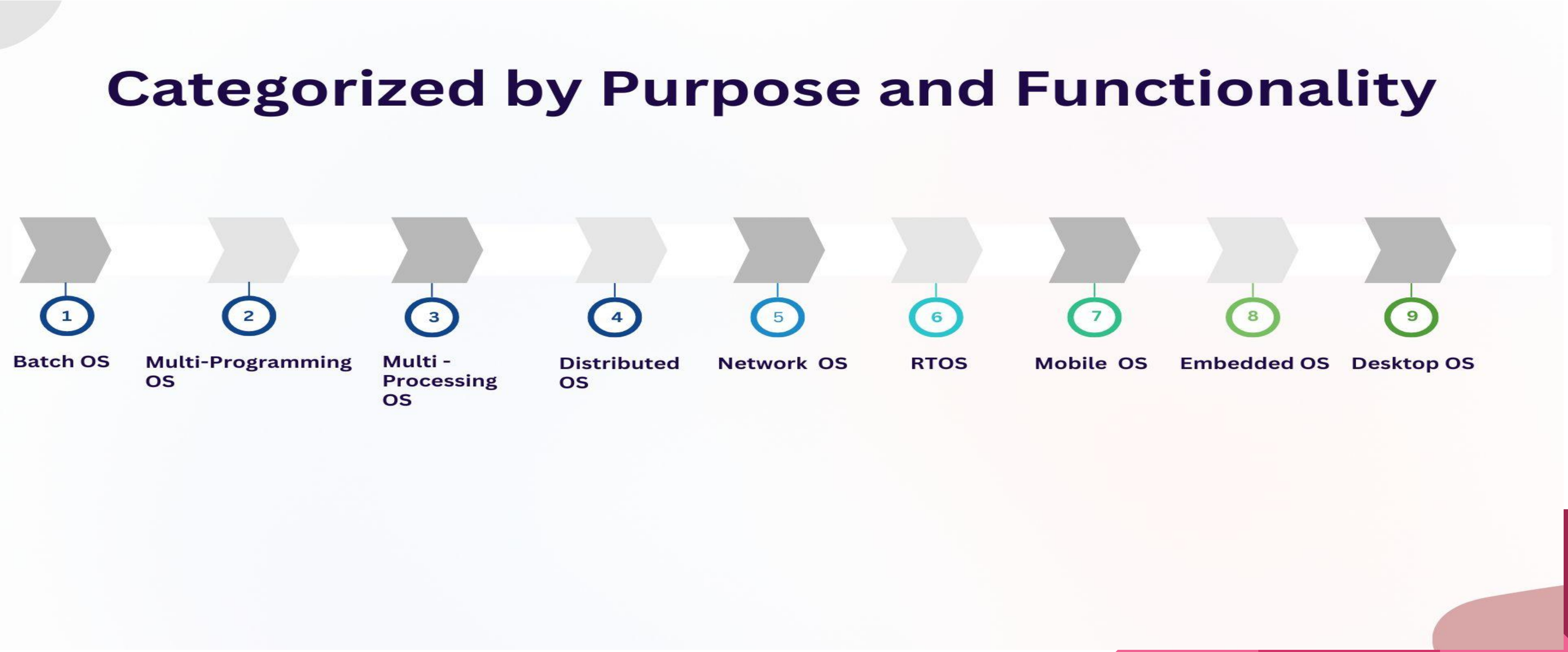
Command Interpreter (Shell)

- Provides an interface for users to interact with the OS.
- Executes commands and scripts.
- Can be graphical (GUI) or command-line (CLI).

Basic computer organization required for OS



Examples of well-known OS including mobile OS, embedded system OS, Real Time OS, desktop OS server machine OS etc. ; How are these different from each other and why



Categorized by Purpose and Functionality

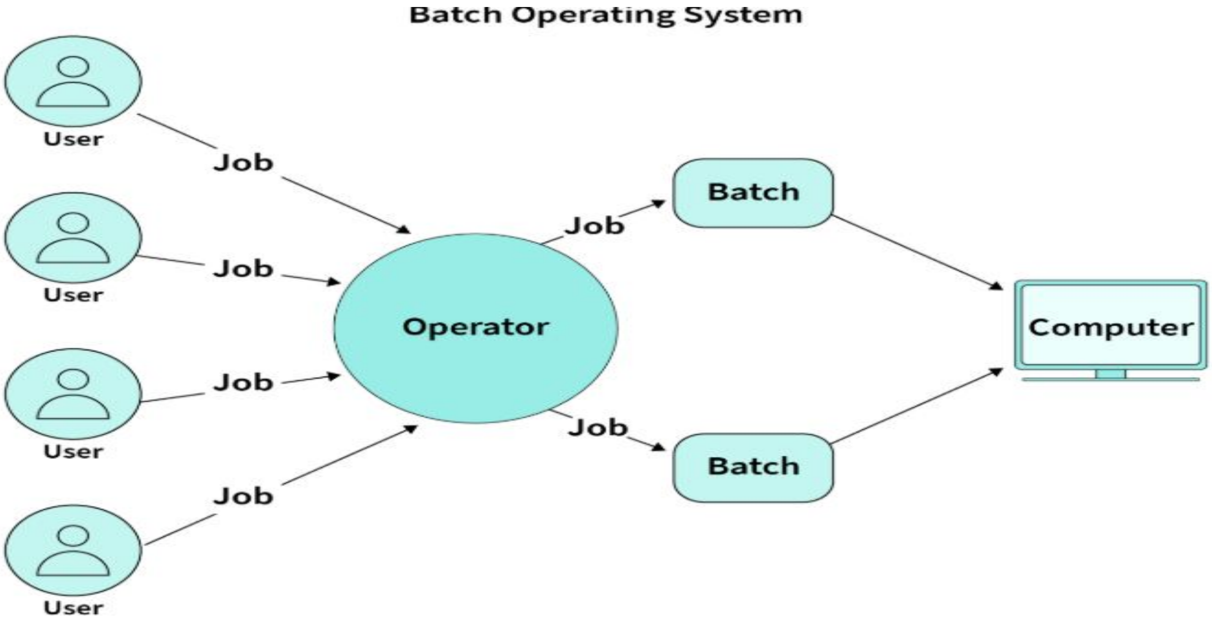
1. Batch OS
2. Multi-Programming OS
3. Multi - Processing OS
4. Distributed OS
5. Network OS
6. RTOS
7. Mobile OS
8. Embedded OS
9. Desktop OS

1) Batch Operating System

- It 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

- Payroll System
- Bank Invoice System
- Transactions Process
- Daily Report
- Research Segment
- Billing System



Advantages and Disadvantages of Batch Operating System

Advantages	Disadvantages
Multiple users can share the batch systems.	CPU is not used efficiently. When the current process is doing IO, the CPU is free and could be utilized by other processes waiting.
The idle time for the batch system is very little.	Other jobs will have to wait for an unknown time if any job fails.
It is easy to manage large work repeatedly in batch systems.	Average response time increases as all processes are processed one by one.

2) Multi - Programming OS

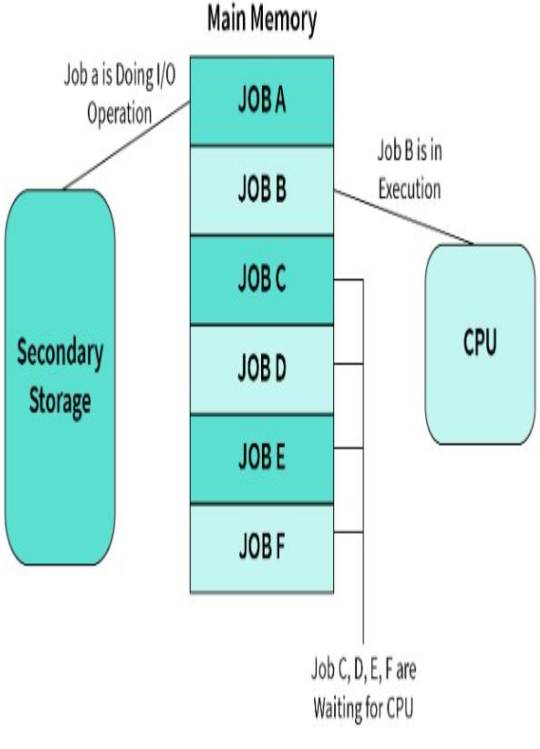
- Multiprogramming Operating Systems can be simply illustrated as, **more than one program is present in the main memory and any one of them can be kept in execution**, so this is used for better utilization of resources.

Examples

- Apps like office, chrome, etc.
- Microcomputers like MP/M, XENIX, and ESQview.
- Windows O/S
- UNIX O/S

A multiprogramming OS is of the following two types:

1. **Multitasking /Time-sharing OS**: Enables execution of multiple programs at the same time, by swapping each program in and out of memory one at a time.
2. **Multiuser Operating System**: This allows many users to share processing time on a powerful central computer from different terminals, by rapidly switching between terminals.



Multiprogramming Operating System

Advantages and Disadvantages of Multi - Programming OS

Advantages

- Great Reliability

  Reliability refers to the probability that the system will perform its intended functions correctly and without failure for a specified period under given conditions

- Improve Throughput

  Throughput refers to the amount of work or data processed within a specific time frame

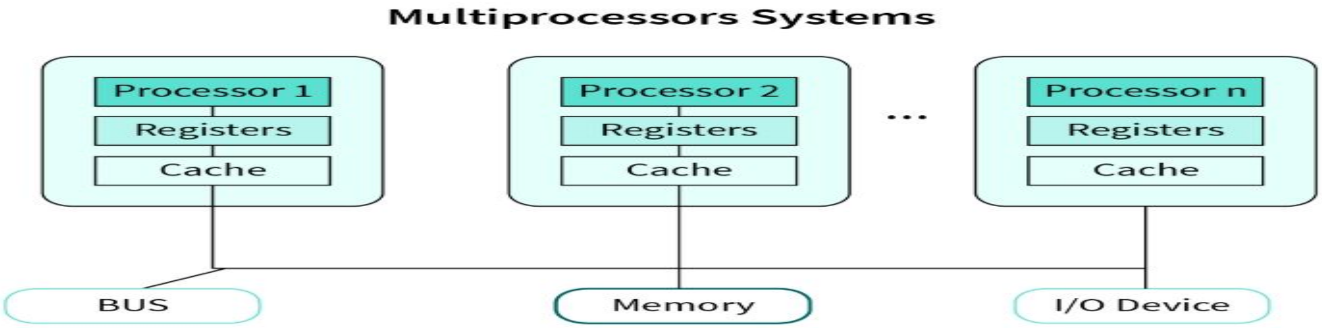
- Cost-Effective System
- Parallel Processing

Disadvantages

- It is more expensive due to its large architecture.
- Its speed can get degraded due to failing any one processor.
- It has more time delay when the processor receives the message and takes appropriate action.
- It has big challenges related to skew and determinism
- It needs context switching which can impact its performance.

3) Multi - Processor OS

- A Multi-Processing Operating System is 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.
- The following are four major components, used in the Multiprocessor Operating System:
  1. **CPU** – capable of accessing memories as well as controlling the entire I/O tasks.
  2. **Input Output Processor** – The I/P processor can access direct memories, and every I/O processor has to be responsible for controlling all input and output tasks.
  3. **Input/Output Devices** – These devices are used for inserting the input commands, and producing output after processing.
  4. **Memory Unit** – Multiprocessor system uses two types of memory modules - shared memory and distributed shared memory.



Advantages and Disadvantages of Multi - Processor OS

Advantages

- Failure of one processor does not affect the functioning of other processors.
- It divides all the workload equally to the available processors.
- Makes use of available resources efficiently.

Disadvantages

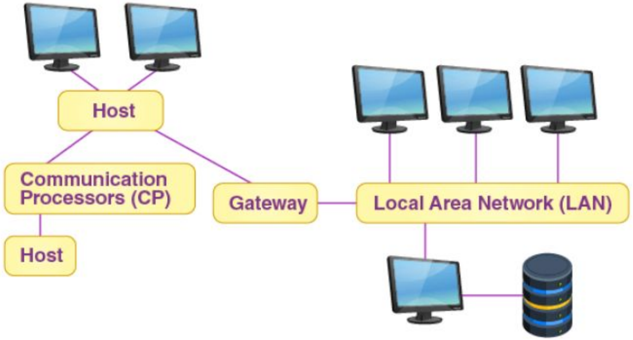
- Symmetrical multiprocessing OS are more complex.

Symmetric multiprocessing is a computer architecture where two or more identical processors share the same memory and input/output (I/O) devices, and are controlled by a single operating system

- They are more costlier.
- Synchronization between multiple processors is difficult.

4) Distributed Operating Systems

- The Distributed OS is separated into sections and **loaded on different machines rather than being placed on a single machine**
- All processors are **connected by valid communication mediums** such as high-speed buses and telephone lines, LAN/WAN lines and in which every processor contains its local memory along with other local processors



A typical view of a distributed System

Advantages and Disadvantages of Distributed OS

Advantages:

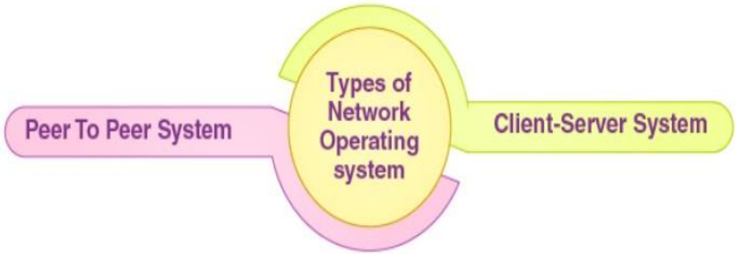
- Increased Reliability
- Scalability
- Resource Sharing
- Improved Performance

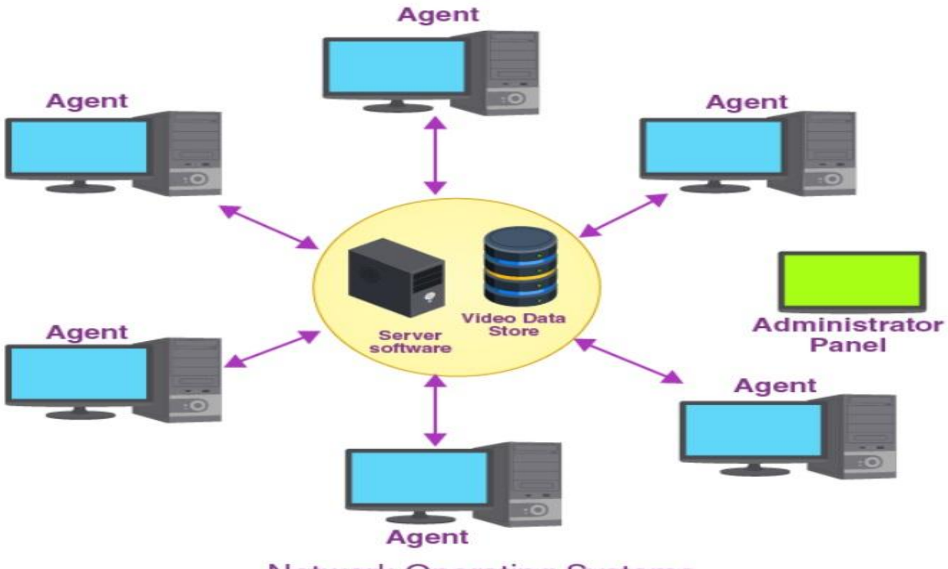
Disadvantages:

- Complex
- Security Concerns
- Network Dependency
- Consistency and Data Integrity
- Difficulty in Troubleshooting

5) Network Operating Systems

- Network Operating System has **special functions for connecting computers and devices into a local-area network or Inter-network.** Some popular network operating systems are Novell Netware, Linux, IBM OS/2, etc.
- There are two basic types of network operating systems:
  1. **Peer-to-Peer Network Operating Systems:** Allow users to **share network resources** saved in a common, accessible network location.
  2. **Client/Server Network Operating Systems**: Provide users with **access to resources through a server**.





Network Operating Systems

Advantages and Disadvantages Network Operating Systems

Advantages

- Centralized Management
- Enhanced Security
- Resource Sharing
- Cost-Effectiveness

Disadvantages

- Dependency and Potential for Failure
- High Setup and Maintenance Costs
- Performance Issues
- Security Risks

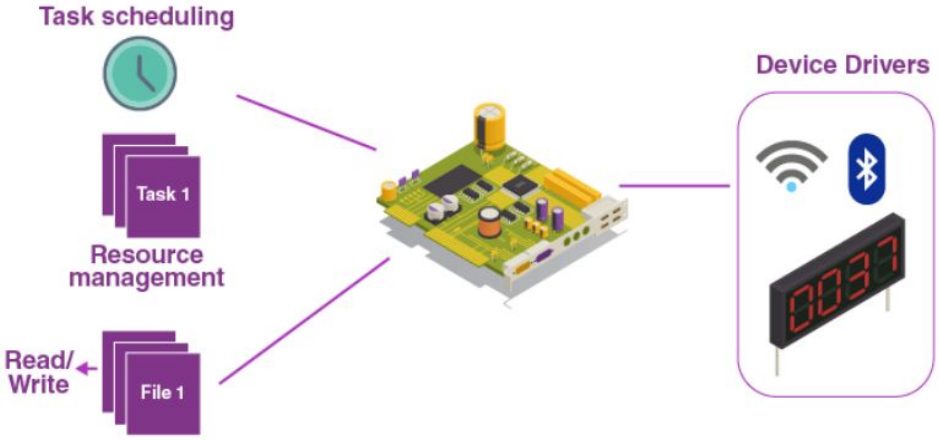
6) Real Time Operating Systems

- In this type of system, **each job has a deadline by which it must be completed; otherwise, there will be a significant loss**, or even if the output is provided, it will be utterly useless.

Example - In military applications, if you wish to drop a missile, the missile must be dropped with a specific degree of precision

Examples

- Airline traffic control systems
- Command Control Systems
- Airlines reservation system
- Heart Pacemaker
- Network Multimedia Systems
- Robotics



Real - Time Operating Systems (RTOS)

Advantages and Disadvantages of RTOS

Advantages

- Maximum utilization of devices and systems
- Error Free
- Best Memory allocation
- Time assigned for shifting tasks in these systems is very less

Disadvantages

- Usage of expensive system resources
- Complex Algorithms
- Device Driver And Interrupt Signals

7) Mobile Operating Systems

- It helps run application software on mobile devices
- The operating systems found on smartphones include Symbian OS, IOS, BlackBerryOS, Windows Mobile, Palm WebOS, Android, and Maemo
- Android, WebOS, and Maemo are all derived from Linux
- iPhone OS originated from BSD and NeXTSTEP, which are related to Unix

Examples

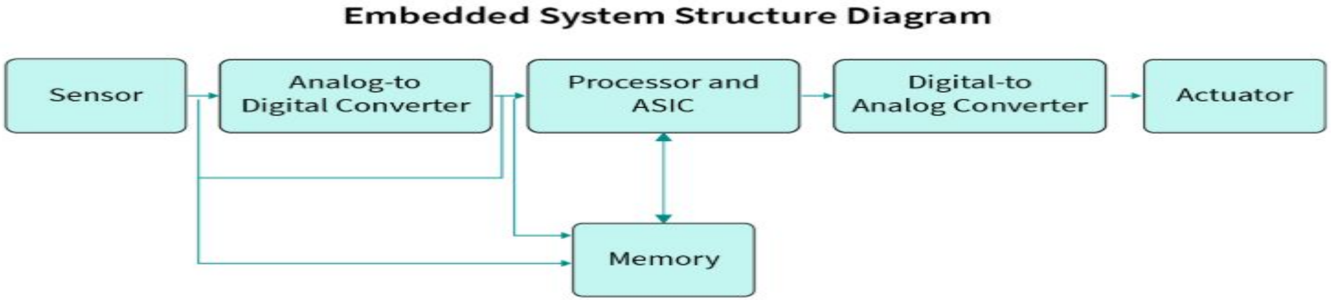
- Android
- IOS
- HarmonyOS
- PalmOS

8) Embedded Operating Systems

- It is built on Internet of Things devices
- It aims to perform with certainty specific tasks regularly that help the device operate.
- An embedded operating system often has limited features and functions.

Examples

- Windows Mobile/CE (handheld Personal Data Assistants)
- Symbian (cell phones)
- Linux-based OSes.


Embedded System Structure Diagram
Sensor
Analog-to Digital Converter
Processor and ASIC
Digital-to Analog Converter
Actuator
Memory

Advantages and Disadvantages of Embedded OS

Advantages

- The OS is often low-cost
- The OS tends to use few resources, including minimal power
- The performance is generally trouble-free.

Disadvantages

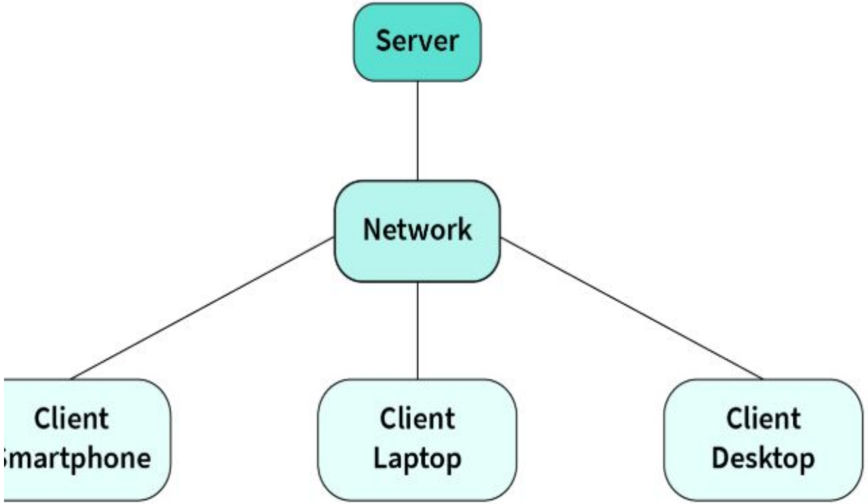
- Usually only run a single or very few applications.
- It is difficult to modify the OS
- Trouble-shooting can be difficult
- Inconsistent and timely execution of action
- Limited power and memory

9) Desktop Operating System

- The Client System can be said as a computer in a network where the user performs some task or activity over the network. Such OS **do not have complete control** over the resources but **use the network to access.**
- The processing power remains in the hands of the server OS, which is developed in such a way that it can fulfill all the requirements of the client or the desktop operating system.

Examples

- Windows
- Linux
- Unix
- MAC OS
- MS-DOS
- Solaris
- Ubuntu
- Fedora
- QNX



Advantages and Disadvantages of Desktop OS

Advantages

- Centralization of resources are present at a common location.
- Better management of resources as the files are stored in a single place.
- Remote access to the server gives processing power to every user.
- High security as only the server needs to be secured from threats and attacks.
- The server can play different roles for the different

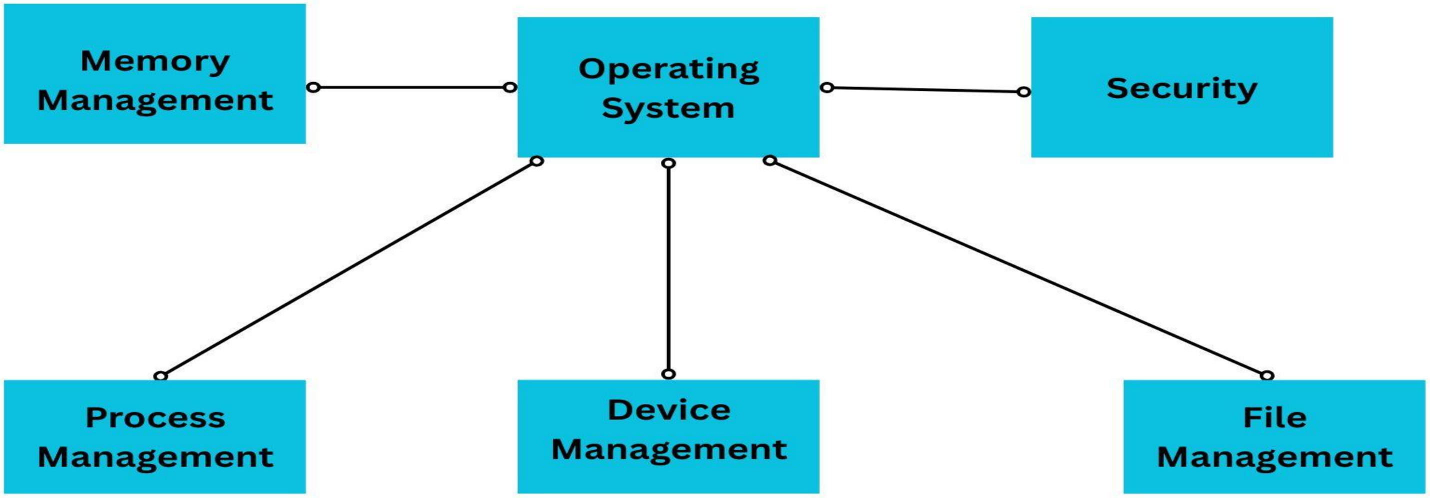
Disadvantages

- Network congestion as multiple requests from the clients can block the network traffic.
- The architecture of request and response is not robust enough for heavy processing.
- If the server fails, all the desktop systems connected over the network fail.
- If the service interrupts, the task has to be started from scratch. For instance, if a desktop system requests a file download that gets interrupted, the file becomes corrupt, and the entire process needs to be carried out from the start.
- The operating system architecture is highly costly.
- A professional IT personnel is needed to manage and maintain such an operating environment.

Functions of OS

- The main goal of an operating system is to make the computer environment more convenient to use and to utilize resources most efficiently.
- Operating System handles the following responsibilities:
  - Controls all the computer resources.
  - Provides valuable services to user programs.
  - Coordinates the execution of user programs.
  - Provides resources for user programs.
  - Provides an interface (virtual machine) to the user.
  - Hides the complexity of software.
  - Supports multiple execution modes.
  - Monitors the execution of user programs to prevent errors.

Functions of OS

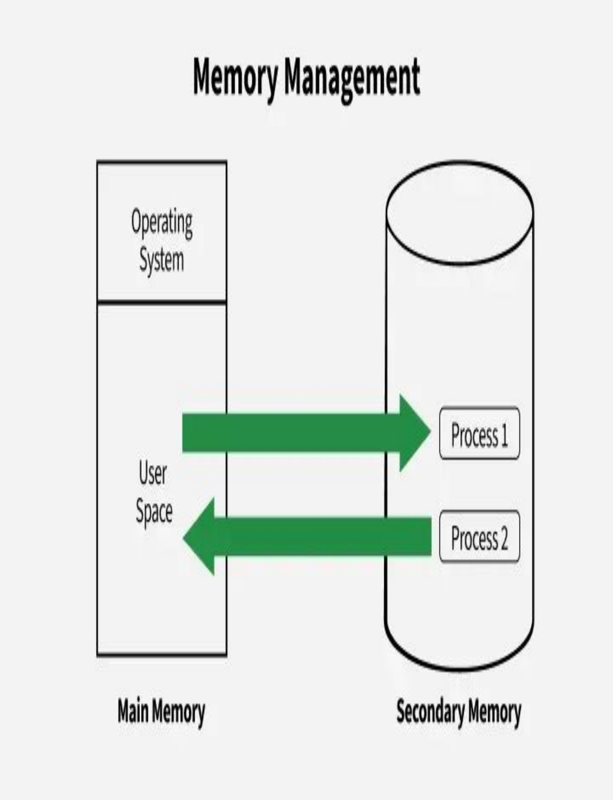


Memory Management

- OS handles the storage and organization of data in both main (primary) memory and secondary storage.

- It ensures that memory is allocated and deallocated properly to keep programs running smoothly.

- It also manages the interaction between volatile main memory and non-volatile secondary storage.



Key Activities in Memory Management:

Main Memory Management

- **Memory Allocation:** Assigns memory to processes using techniques like paging and segmentation.
- **Memory Deallocation:** Frees memory when no longer needed.
- **Memory Protection:** Prevents processes from accessing each other's memory.
- **Virtual Memory:** Uses disk space as extra memory to run larger processes.
- **Fragmentation:** Manages wasted memory space (internal/external) through compaction.

Secondary Memory Management

- **Disk Space Allocation**: Organizes how files are stored on the disk (contiguous, linked, indexed).
- **File System Management**: Manages files and directories for efficient data access.
- **Free Space Management**: Tracks available space on the disk.
- **Disk Scheduling**: Organizes the order of disk read/write requests.
- **Backup and Recovery**: Ensures data is backed up and can be restored after failure.

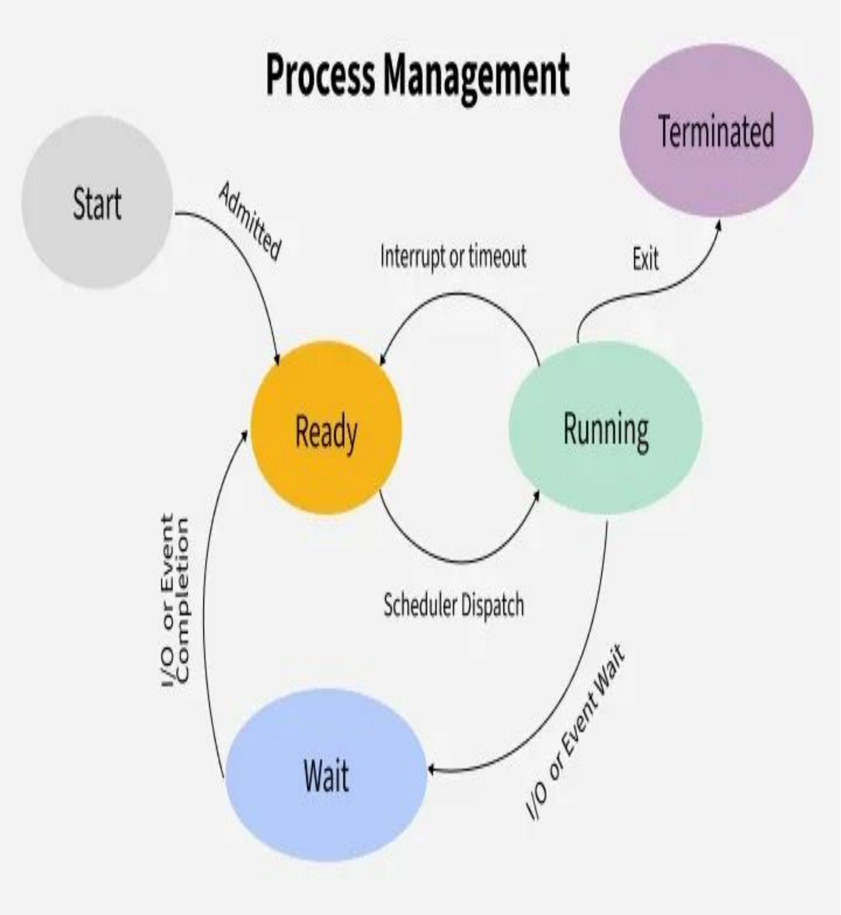
Process Management

- A Process is a running program from the moment program start and until it finishes.

  Operating system makes sure each process:

  - gets its turn to use the CPU
  - synchronized when needed
  - has access to the resources it needs, like memory, files, and input/output devices.
- It also handles issues like process coordination and communication, while preventing conflicts such as deadlocks. This way, the OS ensures smooth multitasking and efficient resource use.

- **Core Functions in Process Management:** Process Scheduling,Process Synchronization,Inter-Process Communication (IPC), Deadlock Handling

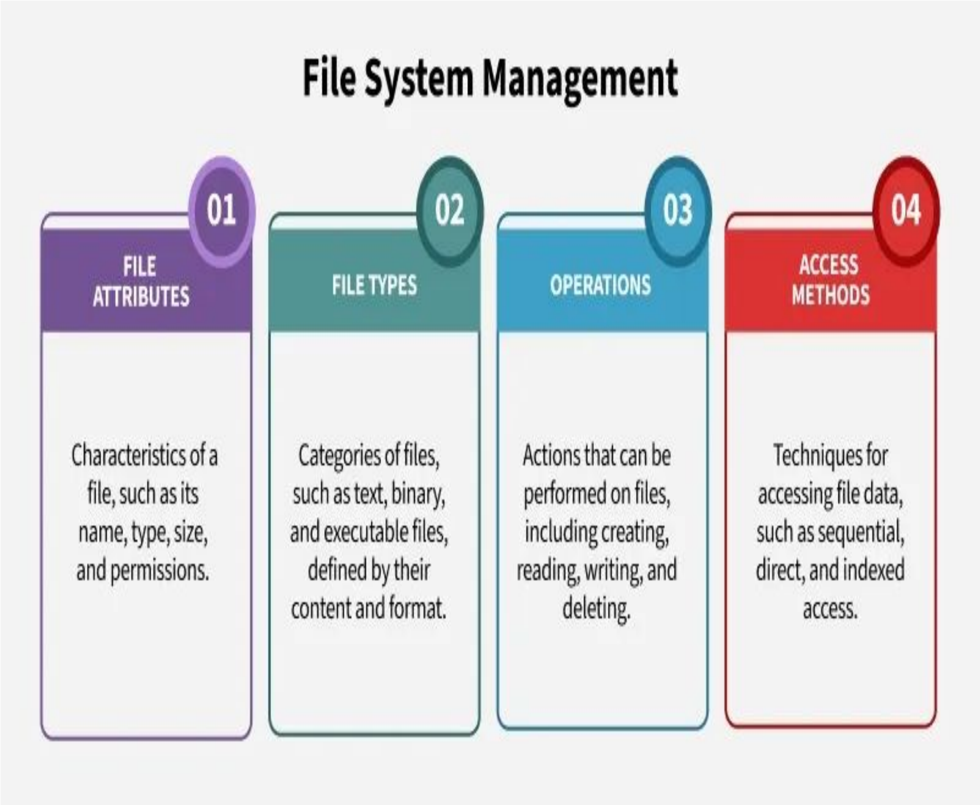


File System Management

- File management in the operating system ensures the organized storage, access and control of files.

- The OS **abstracts the physical storage details to present a logical view of files**, making it easier for users to work with data.

- It manages how files are stored on different types of storage devices (like hard drives or SSDs) and ensures smooth access through directories and permissions.



Characteristics of a file, such as its name, type, size, and permissions.

Categories of files, such as text, binary, and executable files, defined by their content and format.

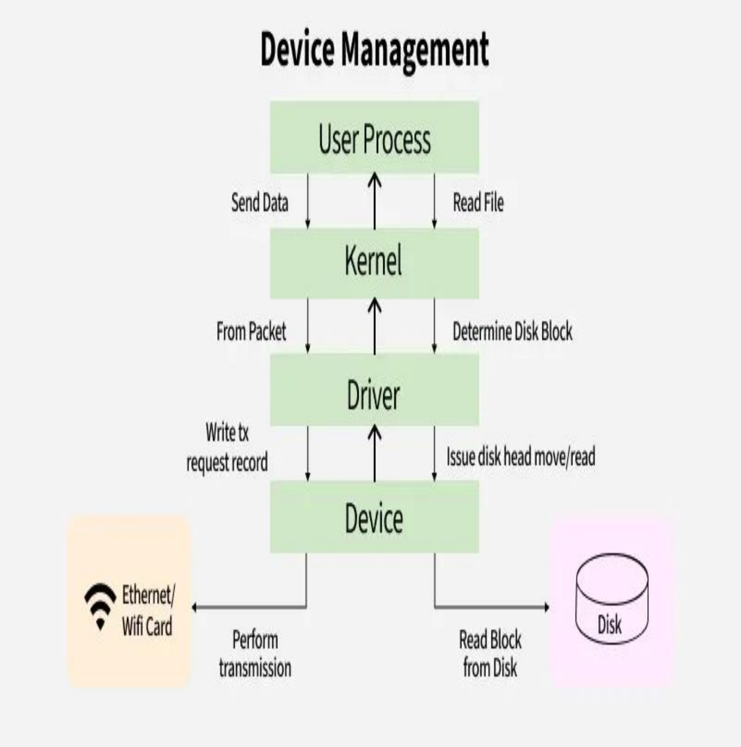
Actions that can be performed on files, including creating, reading, writing, and deleting.

Techniques for accessing file data, such as sequential, direct, and indexed access.

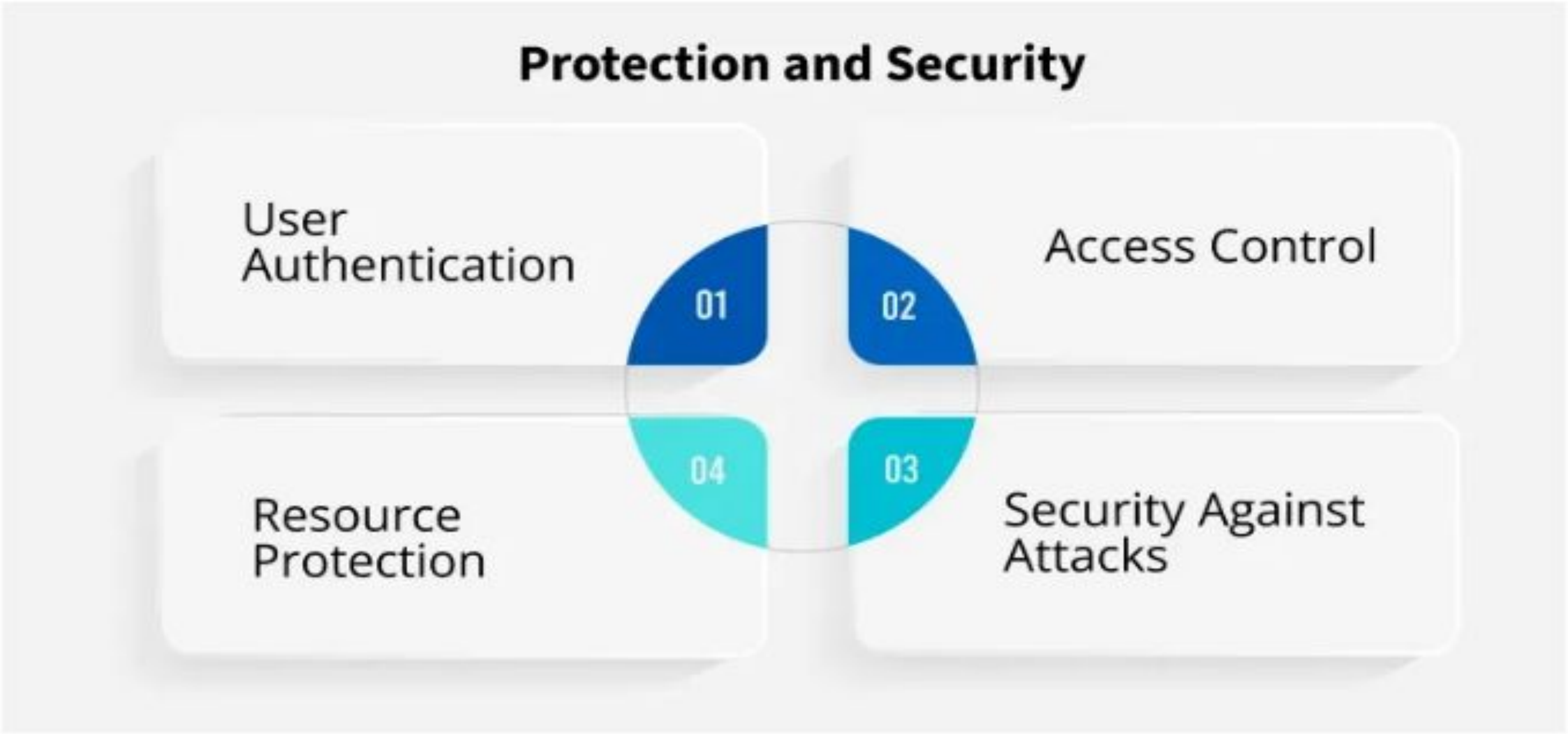
Device Management (I/O System)

- Device management of an operating system handles the communication between the system and its hardware devices, like printers, disks or network interfaces.

- OS provides device drivers to control these devices, using techniques like Direct Memory Access (DMA) for efficient data transfer and strategies like buffering and spooling to ensure smooth operation.



Protection and Security



Protection and Security

01 User Authentication

02 Access Control

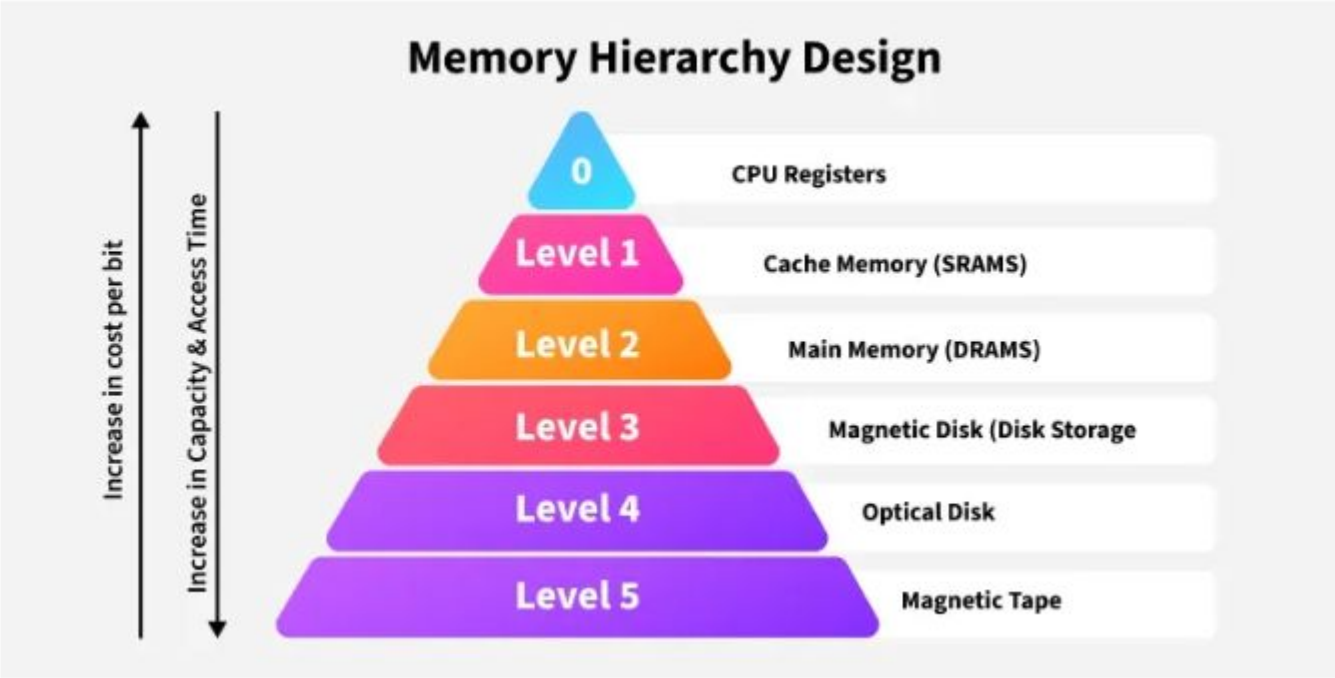
03 Security Against Attacks

04 Resource Protection

Security

Kernel

- **Core of the OS**, responsible for managing hardware and resources.
- Kernels are the **heart of operating systems** , managing how hardware and software communicate and ensuring everything runs smoothly



User and Kernel space and mode

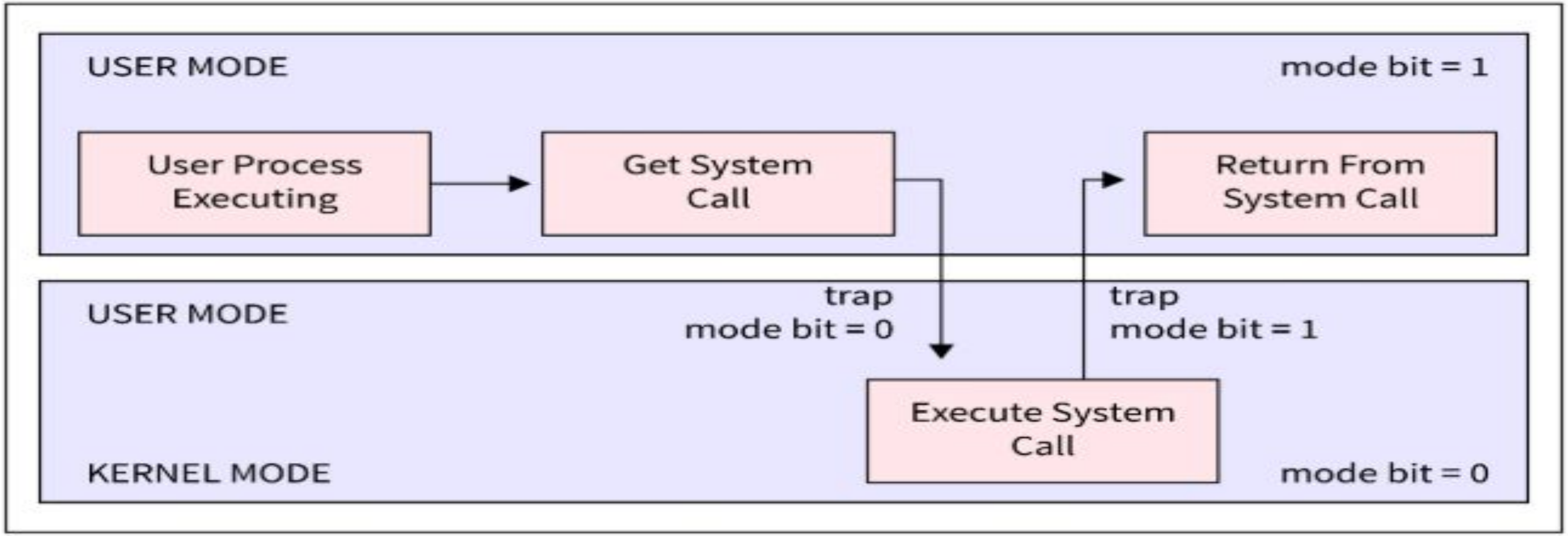
- **User mode and kernel mode are two working states** inside working system that determine the level of access and control.

  For example, if you are running MS Word, or watching some video using the VLC Player, all these software applications are running in the user mode.

- When opening the program in user mode, it is not allowed to access the RAM and hardware directly.
- To access the hardware and RAM in user mode, it sends a request to the kernel. That is the reason user mode is also known as **slave mode** or **restricted mode**.
- Running a program in user mode does not have its own address space and thus also it is unable to access the address space of the kernel.
- That is why, if there is any program failure in user mode, it does not affect the other processes. It only affects that particular process where an interrupt occurs.
- If an application running under user mode and it wants to access system resources and hardware, it will have to first go through the Operating system Kernel by using syscalls (system calls).
- The mode bit is set to 1 in user mode and while switching from user mode to kernel mode, this mode bit is set to 0.

What is Kernel Mode?

- **When we start our system, it boots in Kernel mode**. The kernel can access the hardware and RAM of the system directly but there are some privileged instructions that can run in the kernel mode only.
- These instructions are interrupt instructions, input-output management, etc. If these privileged instructions get executed under the user mode, it is not legal and it may **cause an illegal trap**.



What is Kernel Mode?

- In kernel mode, the mode bit is set to 0. And when switching from kernel mode to user mode, the bit mode is changed from 0 to 1. When the mode changes from user mode to kernel mode or vice-versa, it is known as **Context Switching**.

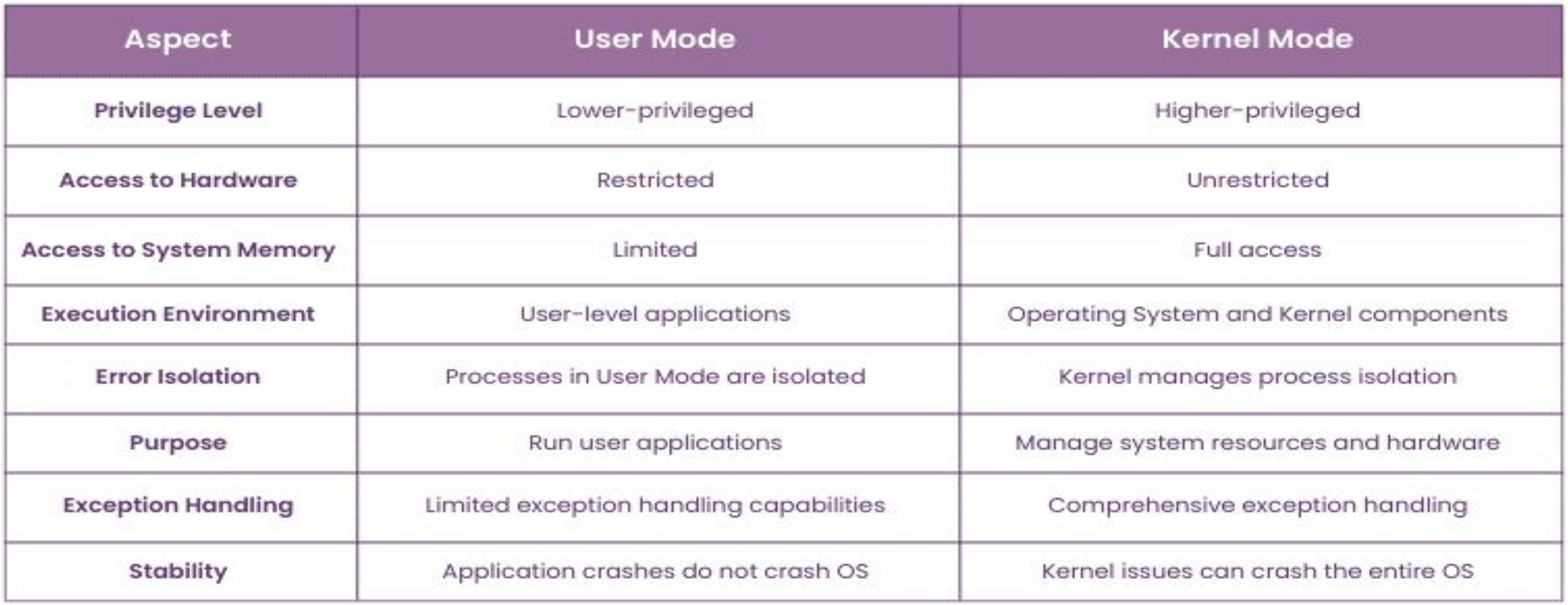
- Kernel is the **central module** of the operating system and it works as the **middle layer** between the operating system and the hardware of a system.

- OS maintain control over the computer system by utilizing the kernel of an operating system as a means of communication.

- The kernel handles the remaining system functions on behalf of the operating system, hence it is the **first software to load into memory** when a system boots up after the bootloader.

- The kernel is in the **memory of the system** until the operating system shuts down the system

Difference between User mode and Kernel mode



Aspect	User Mode	Kernel Mode
Privilege Level	Lower-privileged	Higher-privileged
Access to Hardware	Restricted	Unrestricted
Access to System Memory	Limited	Full access
Execution Environment	User-level applications	Operating System and Kernel components
Error Isolation	Processes in User Mode are isolated	Kernel manages process isolation
Purpose	Run user applications	Manage system resources and hardware
Exception Handling	Limited exception handling capabilities	Comprehensive exception handling
Stability	Application crashes do not crash OS	Kernel issues can crash the entire OS

Interrupts and System Calls

- The interrupt is a **signal emitted by hardware or software** when a process or an event needs immediate attention.
- In I/O devices one of the bus control lines is dedicated for this purpose and is called the **Interrupt Service Routine** (ISR).
- When a device raises an interrupt at let's say process i,e., the processor first completes the execution of instruction i.
- Then it loads the **Program Counter (PC)** with the address of the first instruction of the ISR. Before loading the Program Counter with the address, the address of the interrupted instruction is moved to a temporary location.
- Therefore, after handling the interrupt the processor can continue with process i+1.
- The amount of time between the generation of an interrupt and its handling is known as **Interrupt latency**

Types of Interrupt

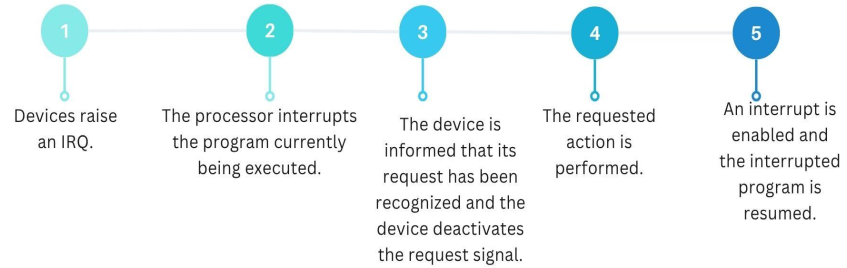
Software Interrupt

- Traps and exceptions are other names
- They serve as a signal to carry out a certain function or respond to an error condition.
- A particular instruction known as an "**Interrupt Instruction**" is used to create software interrupts
- Generated by programs or the operating system itself.
- Used to request OS services (via system calls).
- **Example**: A program may use a software interrupt to ask the OS to read from a file.

Hardware Interrupt

- Triggered by external devices (e.g., keyboard, mouse, timer).
- **Example**: Pressing a key sends a signal to the CPU,

SEQUENCE OF EVENTS


1
Devices raise an IRQ.
2
The processor interrupts the program currently being executed.
3
The device is informed that its request has been recognized and the device deactivates the request signal.
4
The requested action is performed.
5
An interrupt is enabled and the interrupted program is resumed.

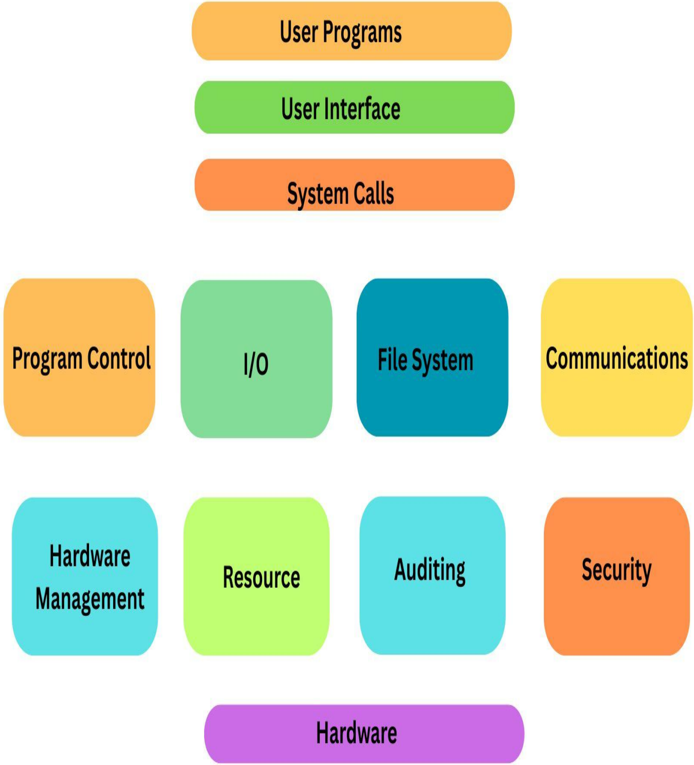
System Calls

- **A System Call** is a programmatic way in which a computer program requests a service from the kernel of the operating system on which it is executed.

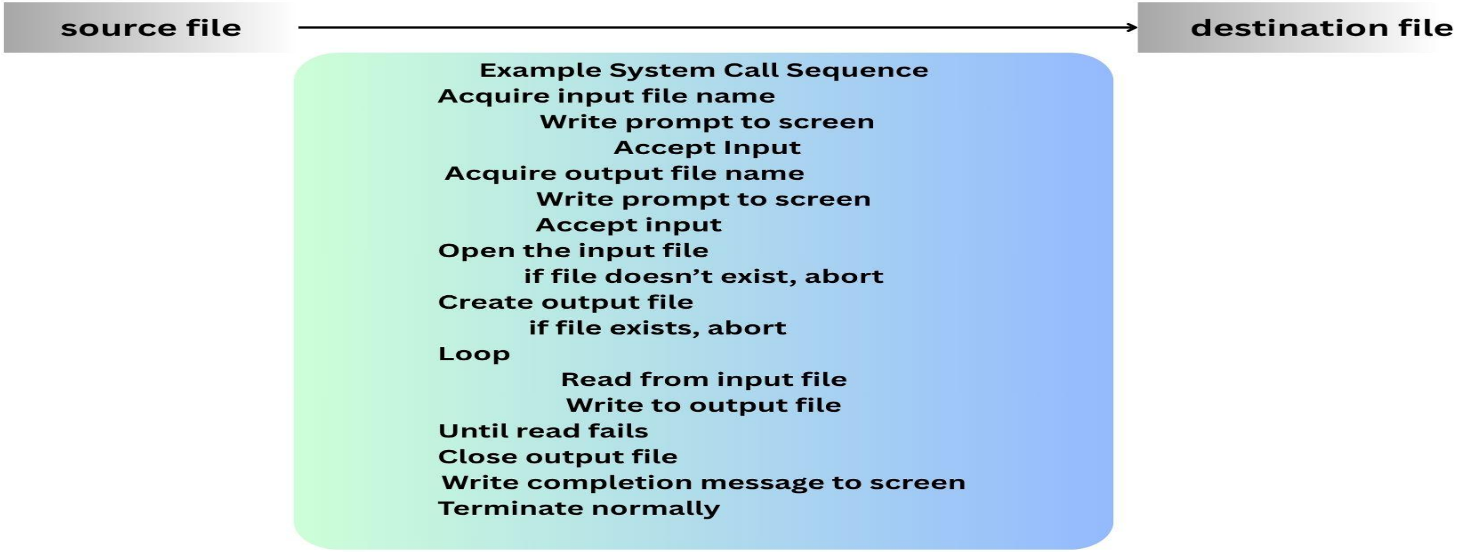
- A system call is a way for programs to **interact with the operating system**. A computer program makes a system call when it requests the operating system's kernel.

- System call **provides** the services of the operating system to the user programs via the Application Program Interface(API).

- **System calls are the only entry points into the kernel system and are executed in kernel mode.**



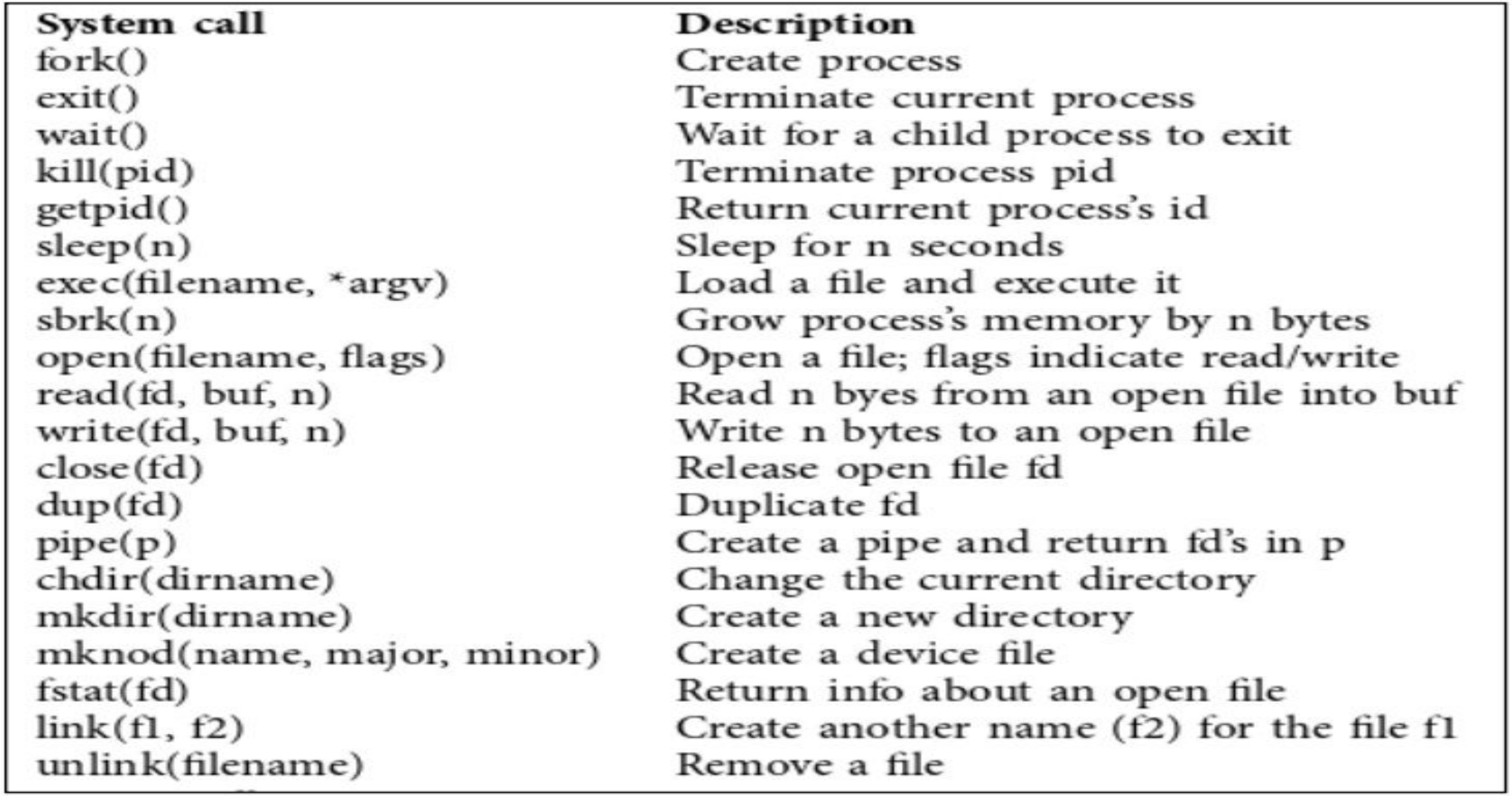
Example of System Call Sequence



source file

destination file

```
Example System Call Sequence
Acquire input file name
        Write prompt to screen
        Accept Input
Acquire output file name
        Write prompt to screen
        Accept input
Open the input file
        if file doesn’t exist, abort
Create output file
        if file exists, abort
Loop
        Read from input file
        Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```



System call	Description
fork()	Create process
exit()	Terminate current process
wait()	Wait for a child process to exit
kill(pid)	Terminate process pid
getpid()	Return current process's id
sleep(n)	Sleep for n seconds
exec(filename, *argv)	Load a file and execute it
sbrk(n)	Grow process's memory by n bytes
open(filename, flags)	Open a file; flags indicate read/write
read(fd, buf, n)	Read n byes from an open file into buf
write(fd, buf, n)	Write n bytes to an open file
close(fd)	Release open file fd
dup(fd)	Duplicate fd
pipe(p)	Create a pipe and return fd's in p
chdir(dirname)	Change the current directory
mkdir(dirname)	Create a new directory
mknod(name, major, minor)	Create a device file
fstat(fd)	Return info about an open file
link(f1, f2)	Create another name (f2) for the file f1
unlink(filename)	Remove a file

Session 2: Introduction to Linux

Lecture:

- Working basics of file system
- Commands associated with files/directories & other basic commands. Operators like redirection, pipe
- What are file permissions and how to set them?
- Permissions (chmod, chown, etc); access control list; network commands (telenet, ftp, ssh, sftp, finger)
- System variables like – PS1, PS2 etc. How to set them

Shell Programming

- What is shell; What are different shells in Linux?
- Shell variables; Wildcard symbols
- Shell meta characters; Command line arguments; Read, Echo

Introduction to Linux

- Linux is one of popular **version of UNIX**
- **Its** development began in 1991 by Linus Torvalds.

Basic Structure of a Linux System

- **Kernel** – Core of the system; manages CPU, memory,

  and devices.

- **Shell** – Interface that interprets user commands.
- **Libraries & Utilities** – Support programs for system

  functionality.

- **Applications** – End-user software (e.g., browsers,

  editors).

Popular Linux Distributions (Distros)

- A **distribution** is a complete Linux system including the kernel, tools, and applications.

Distro	**Description**
Ubuntu	Beginner-friendly, desktop and server
Debian	Very stable, used as a base for others
Fedora	Cutting-edge, developer-focused
Arch Linux	Lightweight, DIY approach
Kali Linux	For ethical hacking and cybersecurity
Red Hat/CentOS	Enterprise-grade, used in servers

Features of LINUX

Stable
-rarely crashes

Compatible
-large number of file formats

Portable
-work on different types of hardware

Open Source
-free to use

Multi-use OS
-multiple users can access the system

Secure
-provide encryption

Multi Programming
-multiple application can be run at same time

Working basics of File System

- A **File System** is a method for storing and organizing data on a computer

- One of the key features of the Linux Operating System is that everything in Linux is a file, including devices, programs, and system information

- Linux File System is a **hierarchical, tree-like and organized structure** that starts with the root directory (/), which contains all other directories and files and branches out into subdirectories as needed and makes complicated systems can be structured logically and organized

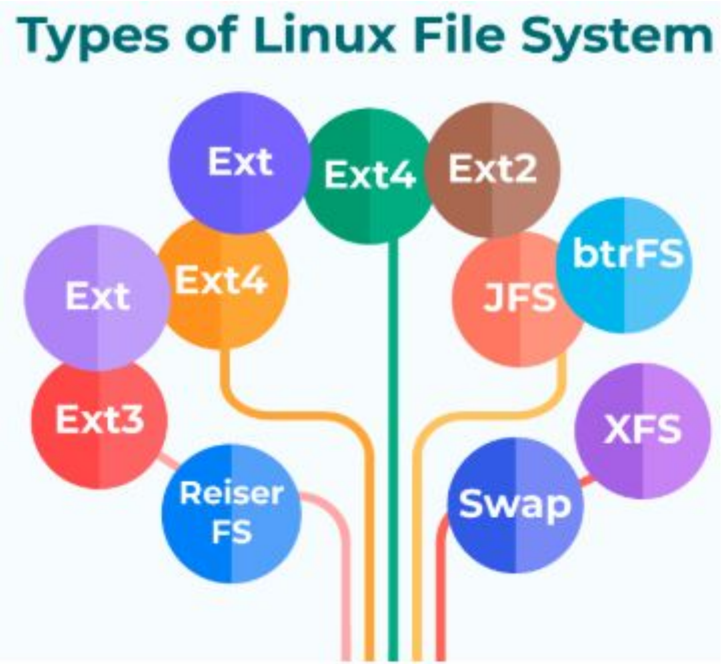
- Even the most **basic commands** such as ls and cat are also files, which lies inside the /bin directory, which **itself is also a file**

- The Linux file system structure also **provides an API (Application Programming Interface)** that allows applications to interact with the file system.

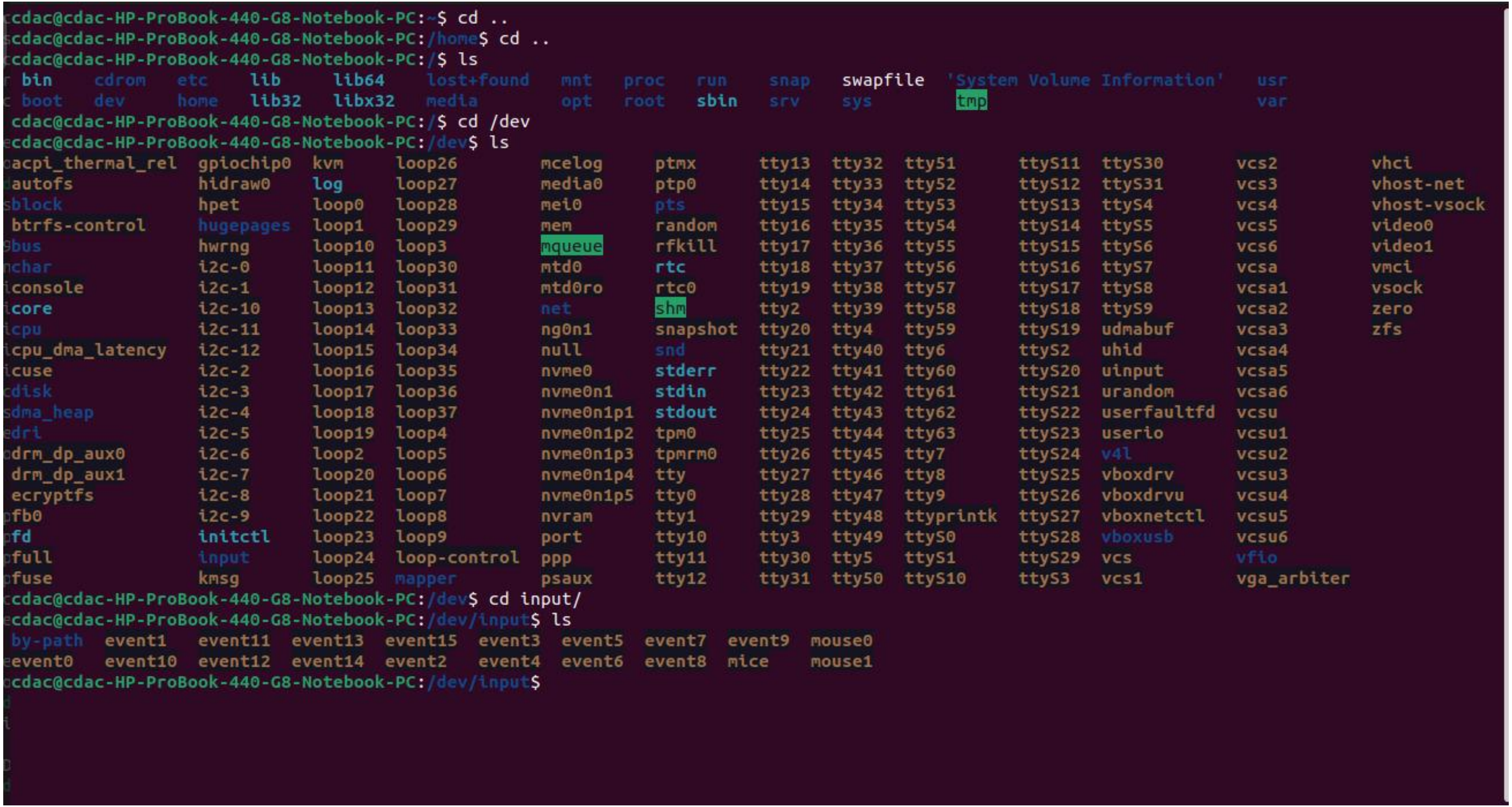
What does API do?

- The API provides a set of functions and commands that allow applications to create, modify, and delete files and directories, as well as to read and write data to and from storage devices.

- Linux uses different file systems such as ext4, XFS, Btrfs, JFS, and ZFS to manage and store data on storage devices.
  - ext2 - USB drives, legacy systems
  - ext3 - Older Linux systems
  - ext4 - Default on modern Linux
  - XFS - Servers, large files
  - btrfs - Backups, snapshots, containers

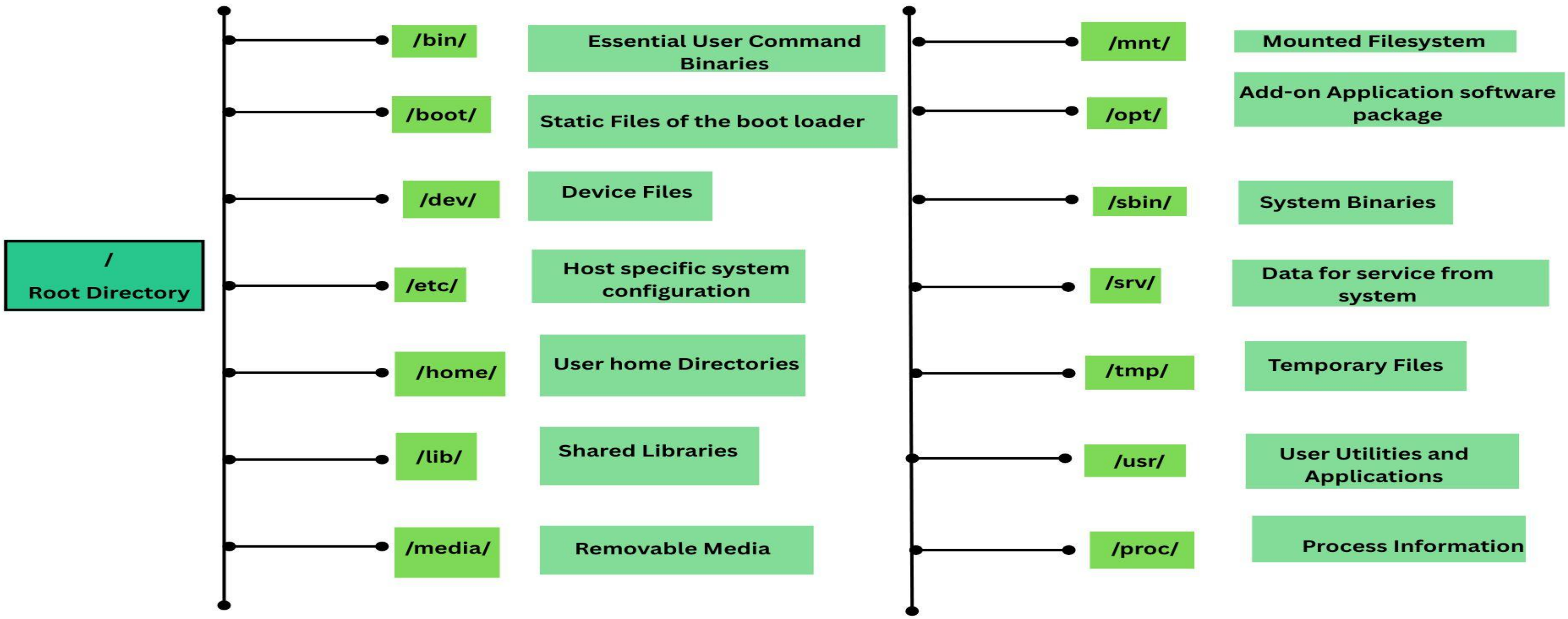


Types of File System in Linux



```
cdac@cdac-HP-ProBook-440-G8-Notebook-PC:~$ cd ..
cdac@cdac-HP-ProBook-440-G8-Notebook-PC:/home$ cd ..
cdac@cdac-HP-ProBook-440-G8-Notebook-PC:/$ ls
 bin    cdrom   etc    lib     lib64    lost+found   mnt   proc   run    snap   swapfile  'System Volume Information'   usr
 boot   dev     home   lib32   libx32   media        opt   root   sbin   srv    sys        tmp                          var
cdac@cdac-HP-ProBook-440-G8-Notebook-PC:/$ cd /dev
cdac@cdac-HP-ProBook-440-G8-Notebook-PC:/dev$ ls
acpi_thermal_rel  gpiochip0  kvm     loop26        mcelog     ptmx      tty13  tty32  tty51      ttyS11  ttyS30       vcs2         vhci
autofs            hidraw0    log     loop27        media0     ptp0      tty14  tty33  tty52      ttyS12  ttyS31       vcs3         vhost-net
block             hpet       loop0   loop28        mei0       pts       tty15  tty34  tty53      ttyS13  ttyS4        vcs4         vhost-vsock
btrfs-control     hugepages  loop1   loop29        mem        random    tty16  tty35  tty54      ttyS14  ttyS5        vcs5         video0
bus               hwrng      loop10  loop3         mqueue     rfkill    tty17  tty36  tty55      ttyS15  ttyS6        vcs6         video1
char              i2c-0      loop11  loop30        mtd0       rtc       tty18  tty37  tty56      ttyS16  ttyS7        vcsa         vmci
console           i2c-1      loop12  loop31        mtd0ro     rtc0      tty19  tty38  tty57      ttyS17  ttyS8        vcsa1        vsock
core              i2c-10     loop13  loop32        net        shm       tty2   tty39  tty58      ttyS18  ttyS9        vcsa2        zero
cpu               i2c-11     loop14  loop33        ng0n1      snapshot  tty20  tty4   tty59      ttyS19  udmabuf      vcsa3        zfs
cpu_dma_latency   i2c-12     loop15  loop34        null       snd       tty21  tty40  tty6       ttyS2   uhid         vcsa4
cuse              i2c-2      loop16  loop35        nvme0      stderr    tty22  tty41  tty60      ttyS20  uinput       vcsa5
disk              i2c-3      loop17  loop36        nvme0n1    stdin     tty23  tty42  tty61      ttyS21  urandom      vcsa6
dma_heap          i2c-4      loop18  loop37        nvme0n1p1  stdout    tty24  tty43  tty62      ttyS22  userfaultfd  vcsu
dri               i2c-5      loop19  loop4         nvme0n1p2  tpm0      tty25  tty44  tty63      ttyS23  userio       vcsu1
drm_dp_aux0       i2c-6      loop2   loop5         nvme0n1p3  tpmrm0    tty26  tty45  tty7       ttyS24  v4l          vcsu2
drm_dp_aux1       i2c-7      loop20  loop6         nvme0n1p4  tty       tty27  tty46  tty8       ttyS25  vboxdrv      vcsu3
ecryptfs          i2c-8      loop21  loop7         nvme0n1p5  tty0      tty28  tty47  tty9       ttyS26  vboxdrvu     vcsu4
fb0               i2c-9      loop22  loop8         nvram      tty1      tty29  tty48  ttyprintk  ttyS27  vboxnetctl   vcsu5
fd                initctl    loop23  loop9         port       tty10     tty3   tty49  ttyS0      ttyS28  vboxusb      vcsu6
full              input      loop24  loop-control  ppp        tty11     tty30  tty5   ttyS1      ttyS29  vcs          vfio
fuse              kmsg       loop25  mapper        psaux      tty12     tty31  tty50  ttyS10     ttyS3   vcs1         vga_arbiter
cdac@cdac-HP-ProBook-440-G8-Notebook-PC:/dev$ cd input/
cdac@cdac-HP-ProBook-440-G8-Notebook-PC:/dev/input$ ls
by-path  event1   event11  event13  event15  event3  event5  event7  event9  mouse0
event0   event10  event12  event14  event2   event4  event6  event8  mice    mouse1
cdac@cdac-HP-ProBook-440-G8-Notebook-PC:/dev/input$
```

Linux Directory



/
Root Directory

/bin/	Essential User Command Binaries
/boot/	Static Files of the boot loader
/dev/	Device Files
/etc/	Host specific system configuration
/home/	User home Directories
/lib/	Shared Libraries
/media/	Removable Media
/mnt/	Mounted Filesystem
/opt/	Add-on Application software package
/sbin/	System Binaries
/srv/	Data for service from system
/tmp/	Temporary Files
/usr/	User Utilities and Applications
/proc/	Process Information

Commands associated with files/directories & other basic commands, Operators like redirection, pipe

A directory can be thought of as a **virtual container that holds files and other directories within it**.

- **/ (root directory):**
  The root directory is the top-level directory in the Linux file system. All other directories and files are contained within the root directory.
- **/bin:**
  The /bin stands for binaries. This directory contains essential command-line tools and programs that are required for basic system administration tasks.
- **/etc:**
  The /etc directory contains system configuration files that are used by various applications and services on the system.
- **/home:**
  The /home directory contains the home directories of users on the system. Each user has their own subdirectory within /home where they can store their personal files and settings.
- **/opt:**
  The /opt directory is used to store additional software packages that are not part of the core system.
- **/tmp:**
  The /tmp directory contains temporary files that are created by applications and services running on the system.
- **/usr:**
  The /usr directory contains user-level programs, libraries, documentation, and shared data files.
- **/var:**
  The /var directory contains variable data files that change frequently, such as log files and system databases.

Commands associated with files/directories & other basic commands

Command	Description	Example
ls	List files and directories	ls -l, ls -a
cd	Change directory	cd Documents/
pwd	Print current directory path	pwd
mkdir	Create a new directory	mkdir my_folder
rmdir	Remove an empty directory	rmdir old_folder
rm	Delete a file or directory	rm file.txt, rm -r folder/
cp	Copy file or directory	cp a.txt b.txt, cp -r dir1 dir2

Command	Description	Example
echo	Print text to terminal or file	echo "Hello"
man	Show manual/help for commands	man ls

Command	Description	Example
mv	Move or rename file/directory	mv old.txt new.txt
touch	Create a new empty file	touch notes.txt
stat	Show file or directory details	stat file.txt
cat	Display file content	cat file.txt
head	Show first 10 lines of a file	head file.txt
tail	Show last 10 lines of a file	tail file.txt
chmod	Change file permissions	chmod 755 script.sh
chown	Change file ownership	sudo chown user file.txt
find	Find files/directories by name or type	find /home -name "*.txt"
locate	Fast search using database	locate myfile.pdf

<table>
<tr><th colspan="3">Redirection Operators</th></tr>
<tr><th>Command</th><th>Description</th><th>Example</th></tr>
<tr><td>> (Output Redirection)</td><td>Redirects standard output to a file, overwriting its contents</td><td>echo "Hello" > file.txt
-file.txt is replaced with the text “Hello”</td></tr>
<tr><td>>>(Append Output)</td><td>Redirects standard output to a file, appending to its contents</td><td>echo "Hello again" >> file.txt
-Adds “more text ” to the end of file.txt</td></tr>
<tr><td>< (Input Redirection)</td><td>Redirects standard input from a file</td><td>sort < file.txt</td></tr>
<tr><td>Pipe Operator (|)</td><td>Used to chain multiple commands together, passing the output of one as input to another.
ls -l | grep "txt"</td><td>ls -l | grep "txt"</td></tr>
<tr><td colspan="2">cat file.txt | sort | uniq > sorted.txt</td><td>Reads file.txt
Sorts the lines
Sorts the lines
Saves to sorted.txt</td></tr>
</table>

What are file permissions and how to set them?

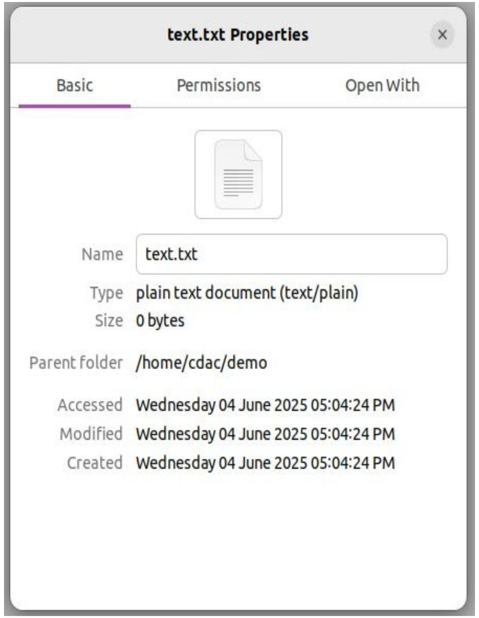
There are **two ways to check the permissions**:

1) Using the graphical user interface (GUI)
2) The command-line interface (CLI)

Permissions tab shows the permissions for each file **divided into three categories**:

- Owner (the user who created the file/directory).
- Group (which the owner belongs to).
- Others (all other users).

1) Check Permissions Using GUI



text.txt Properties

Name text.txt

Type plain text document (text/plain)

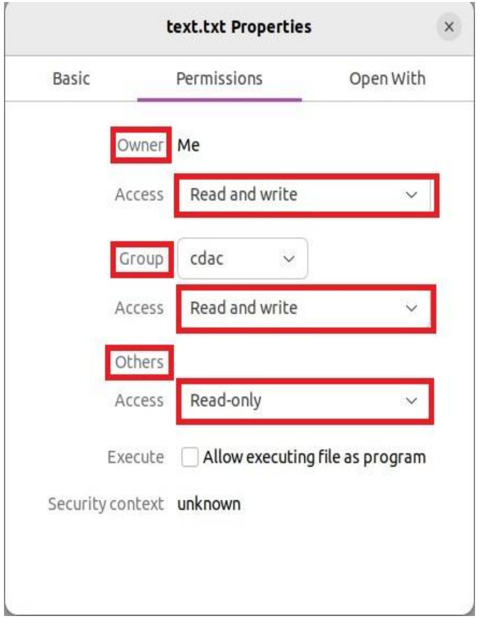
Size 0 bytes

Parent folder /home/cdac/demo

Accessed Wednesday 04 June 2025 05:04:24 PM

Modified Wednesday 04 June 2025 05:04:24 PM

Created Wednesday 04 June 2025 05:04:24 PM



text.txt Properties

Owner Me

Access Read and write

Group cdac

Access Read and write

Others

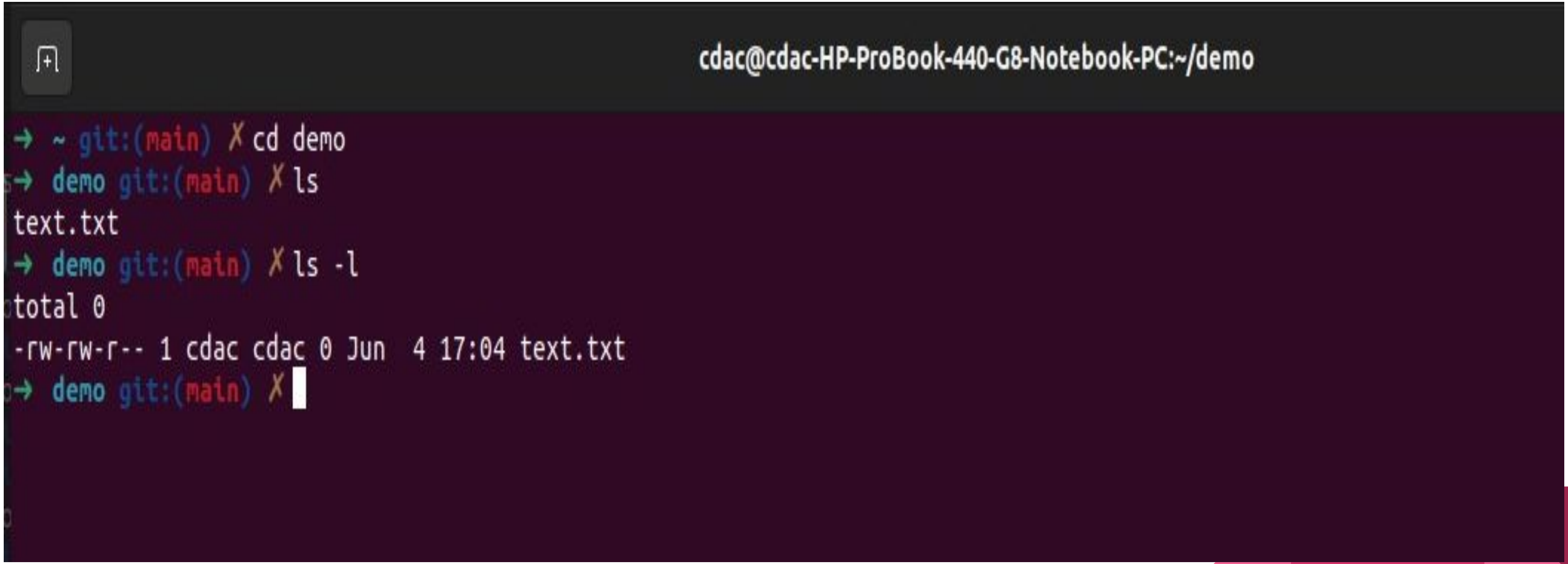
Access Read-only

Execute □ Allow executing file as program

Security context unknown

2) The command-line interface (CLI)

- Use the **ls command to list information about files/directories**
- Each category has three permission types: read (r), write (w), and execute (x)



cdac@cdac-HP-ProBook-440-G8-Notebook-PC:~/demo

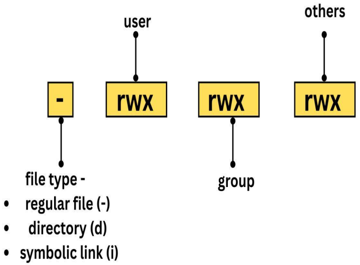
```
➜  ~ git:(main) ✗ cd demo
➜  demo git:(main) ✗ ls
text.txt
➜  demo git:(main) ✗ ls -l
total 0
-rw-rw-r-- 1 cdac cdac 0 Jun  4 17:04 text.txt
➜  demo git:(main) ✗ 
```

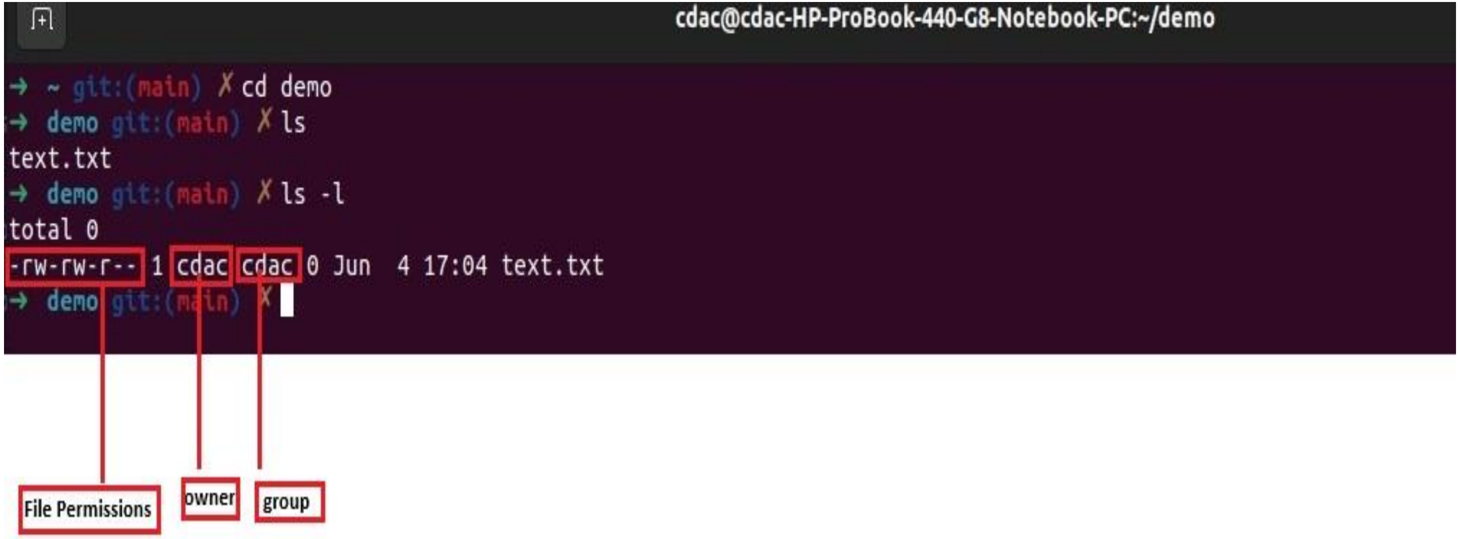
Permission Types

- **Read. (r) -** The read permission allows users to view the contents of a file or list the contents of a directory.
- **Write. (w) -** The write permission allows users to modify a file's contents or add, remove, or rename files within a directory.
- **Execute. (x) -** The execute permission allows users to execute a file or traverse (i.e., enter) a directory. For files, execute permission is required to run the file as a **program** or **script**. For directories, execute permission is required to access the contents of the directory.

Check Permissions in Command-Line with ls Command

- Use the **ls command** to list information about files/directories.
- You can also add the -l option to the command to see the information in a long list format.





```
➜  ~ git:(main) ✗ cd demo
➜  demo git:(main) ✗ ls
text.txt
➜  demo git:(main) ✗ ls -l
total 0
-rw-rw-r-- 1 cdac cdac 0 Jun  4 17:04 text.txt
➜  demo git:(main) ✗
```

Permissions

How to give permissions to file?

- **Syntax** - chmod permissions filename

  Where permissions can be read, write, execute or a combination of them. filename is the name of the file for which the permissions need to change.

How to change permissions?

- We can change permissions using two modes:
  1. **Symbolic mode**: This method uses symbols like u, g, o to represent users, groups, and others.
     Permissions are represented as r (read), w (write), x (execute).
     You can modify permissions using +, - and =.

Example: chmod u+x filename

+ → Adds a permission to a file or directory
– → Removes the permission
= → Sets the permission if not present before. Also overrides the permissions if set earlier.

2. Absolute mode:

- This method represents permissions as 3-digit octal numbers ranging from 0-7 to represent permissions and mathematical operators to modify.

  Ex - chmod ugo+rwx file_name → chmod 777 file_name

  In above, both of them provide full read ,write and execute permission to all the group

Octal	Binary	File Mode
0	000	- - -
1	001	- - x
2	010	- w -
3	011	- w x
4	100	r - -
5	101	r- x
6	110	rw-
7	111	rwx

Permissions (chmod, chown, etc)

- **chmod** - change file/directory permissions
  Symbolic Mode
  Ex - chmod u+x file.sh → Add execute to user
  chmod g-w report.txt→Remove write from group
  chmod 0=r file.txt → Set others to read-only

  chmod 775 file.sh →User: rwx (7), Group: r-x (5) , Others: r-x (5)

- **chown** - **Change Ownership**
  Ex - chown username file.txt
  chown user:group file.txt

- **chgrp** - **Change Group**
  Ex - chgrp developers file.txt

Access Control List (ACL)

- ACL allows **setting individual permissions for multiple users/groups**

Set ACL - setfacl -m u:john:rwx file.txt → Give 'john' full access

setfacl -m g:staff:rw file.txt → Group 'staff' read-write

View ACL - getfacl file.txt

Remove ACL - setfacl -x u:john file .txt

setfacl -b file.txt → Remove all ACLs

Network Commands

Command	Purpose	Example
telnet	Connect to remote system (insecure)	telnet example.com 23
ftp	Transfer files(unsecure)	ftp ftp.example.com
ssh	Secure remote login (encrypted)	ssh user@host
sftp	Secure FTP over SSH	sftp user@host
finger	View user info (login, shell,etc.,)	finger username

System variables like – PS1, PS2 etc.

- In Linux, **System variables** like PS1, PS2, and others are **environment variables** used to configure various aspects of the shell environment.

Variable	Description
PS1	Primary prompt string (default command prompt)
PS2	Secondary prompt string (used for multi-line commands)
PS3	Prompt for select command (used in shell scripts)
PS4	Used for debugging (shown when running with set -x)

How to set them System Variables?

Modify Temporarily (Only for Current Session)

You can set or change them directly in the shell:

```
PS1="[\u@\h \W]\$ "     # Example custom prompt
PS2="> "                # Change secondary prompt
```

Make the Change Permanent:

Edit the ~/.bashrc

```
nano ~/.bashrc
```

Add or modify lines like:

```
export PS1="[\u@\h \W]\$ "
```

After saving, apply changes with:

```
source ~/.bashrc
```

Ex - export PS1="\[\e[1;32m\]\u@\h:\w\$ \[\e[0m\]"

The above will make the prompt green with

user@host:path format.

Sequence	Meaning
\u	Username
\h	Hostname
\w	Current working directory
\W	Basename of the current directory
\t	Current time
\d	Date
\$	$ for normal user, # for root

Shell Programming

What is Shell?

- If we are using any major operating system, we are indirectly interacting with the **shell**.
- While running any Linux distribution, we are interacting with the shell by using the terminal
- A **shell** in Linux is a **command-line interpreter** that allows users to interact with the operating system.
- It takes input from the user, processes it and passes it to the kernel for execution.

It can be used to:

- Run programs
- Manage files
- Execute system commands
- Automate tasks with shell scripts

What are different shells in Linux?

- Linux supports multiple shells. Each has its own features and syntax, but they all serve the same core purpose.

<table>
<tr><td colspan="3"><b>Bourne shell</b> − If you are using a Bourne-type shell, the <b>$</b> character is the default prompt.</td></tr>
<tr><th>Shell</th><th>Description</th><th>Command to use</th></tr>
<tr><td><b>Bash</b> (Bourne Again Shell)</td><td>Most commonly used shell; default in most Linux distros</td><td>bash</td></tr>
<tr><td><b>Sh</b> (Bourne Shell)</td><td>Original Unix shell; simple and portable</td><td>sh</td></tr>
<tr><td><b>Zsh</b> (Z Shell)</td><td>Advanced shell with better scripting, plugins, and completion</td><td>zsh</td></tr>
<tr><td><b>Ksh</b> (Korn Shell)</td><td>Combines features of sh and csh, used in enterprise systems</td><td>ksh</td></tr>
</table>

C shell − If you are using a C-type shell, the % character is the default prompt		
Shell	**Description**	**Command to use**
Csh (C Shell)	C-like syntax; less commonly used today	csh
(TENEX C Shell) tcsh	Enhanced version of csh with command-line editing	tcsh

Shell Variables

- In shell scripts, variables act as containers for holding strings and they do not possess memory addresses.
- Variables in shell scripts are mostly used for referring and altering data within the script.

Examples

Variable Names:

- A variable name could contain any alphabet (a-z, A-Z), any digits (0-9), and an underscore (_).
- However, a variable name must start with an alphabet or underscore.
- It can never start with a number.
- Shell variables are named in UPPERCASE by convention.

Note: It must be noted that no other special character such as !,*,- except underscore can be used in a variable name because all other special characters have special meanings in Shell Scripting

Ex: **Valid Variable Names -** ABC, !ABD, $ABC **Invalid variable names -** 2_AN, _AV_3, AV232, &QAID

Defining Variables:

- We use the equals symbol (=) to declare a variable in Linux.

Syntax: variable_name=<variable data>

Ex - my_message="Hello World"

- Note that there must be no spaces around the "**=**" sign

Accessing Variable

- Variable data could be accessed by appending the variable name with '$' as follows:

```
VAR_1="Devil"

VAR_2="OWL”

echo "$VAR_1$VAR_2"
```

Output: DevilOWL

Unsetting Variables

- The unset command directs a shell to delete a variable and its stored data from list of variables.
- It can be used as follows:

```
var1="Devil"

var2=23

echo $var1 $var2

unset var1

echo $var1 $var2
```

Output:

```
DEVIL 23

23
```

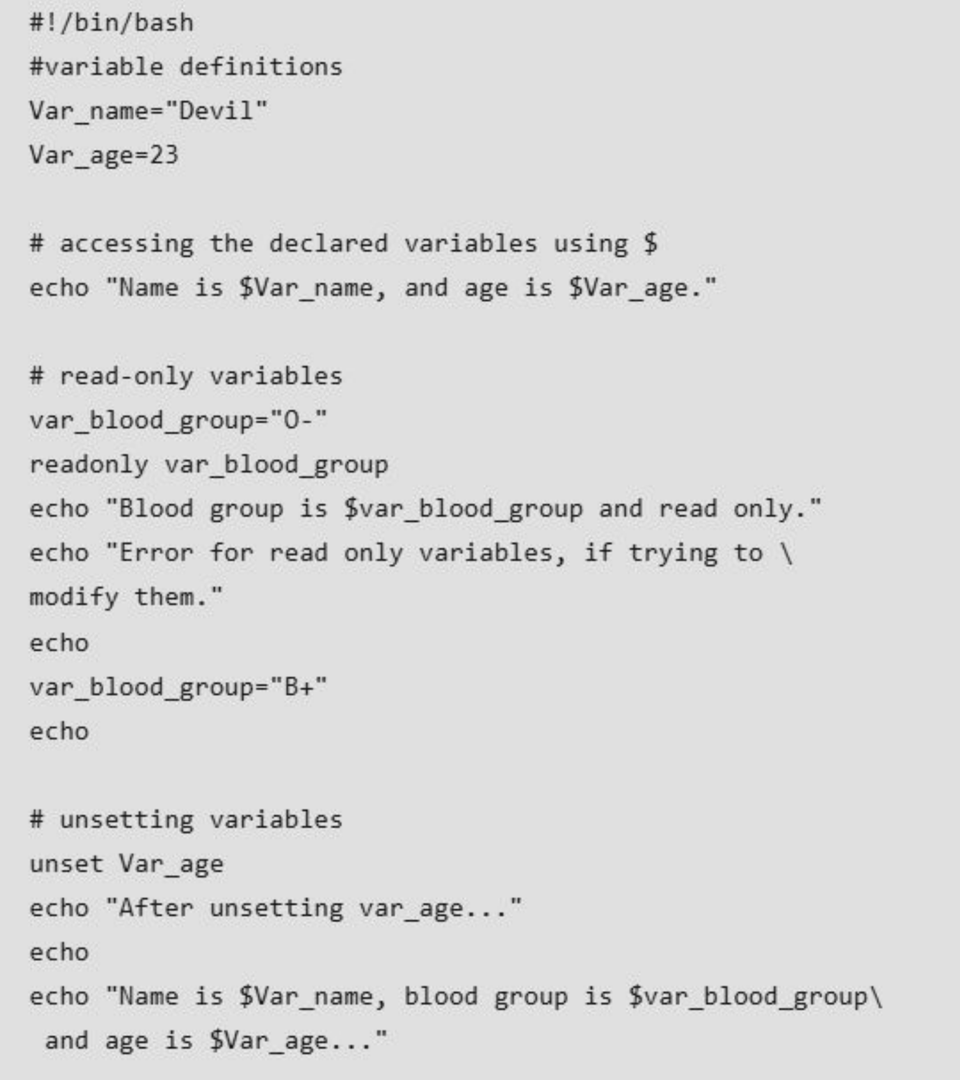
Read only Variables

- These variables are read only i.e., their values could not be modified later in the script

```
var1="Devil"
var2=23
readonly var1
echo $var1 $var2
var1=23
echo $var1 $var2
```

Output : Devil 23

```
./bash1: line 8: var1: readonly variable
                    Devil 23
```



```
#!/bin/bash
#variable definitions
Var_name="Devil"
Var_age=23

# accessing the declared variables using $
echo "Name is $Var_name, and age is $Var_age."

# read-only variables
var_blood_group="O-"
readonly var_blood_group
echo "Blood group is $var_blood_group and read only."
echo "Error for read only variables, if trying to \
modify them."
echo
var_blood_group="B+"
echo

# unsetting variables
unset Var_age
echo "After unsetting var_age..."
echo
echo "Name is $Var_name, blood group is $var_blood_group\
 and age is $Var_age..."
```

Variable Types

1) Local Variable:

- A local variable is a variable that is present within the current instance of the shell.
- Local variables is temporary storage of data within a shell script.
- It is not available to programs that are started by the shell. They are set at the command prompt.
  For Example: `name=Jayesh`

  In this case the local variable is (name) with the value of Jayesh.

2) Environment Variables:

- These variables are commonly used to configure the behavior script and programs that are run by shell.
- Environment variables are only created once, after which they can be used by any user.

  For Example: `export PATH=/usr/local/bin:$PATH` would add `/usr/local/bin` to the beginning of the shell's search path for executable programs.

Shell Variables −It is a special variable that is set by the shell and is required by the shell in order to function correctly.

Some of these variables are environment variables whereas others are local variables.

For Example: `$PWD` = Stores working directory

`$HOME` = Stores user's home directory

`$SHELL` = Stores the path to the shell program that is being used.

Wildcard Symbols in Linux Shell

- Wildcards are special characters used in the shell to represent **one or more characters** in file and directory names.
- They're especially useful in commands like ls, cp, mv, rm, etc.
- Wildcards are also called **globs**.

Wildcard	Meaning / Matches	Example
*	Matches **zero or more** characters	ls *.txt (all .txt files)
?	Matches **exactly one** character	ls file?.txt (e.g., file1.txt, fileA.txt)
[]	Matches **any one character** inside brackets	ls file[123].txt (matches file1.txt, file2.txt, file3.txt)
[^]	Matches **any one character not** in brackets	ls file[^1].txt (matches all except file1.txt)
{}	Matches **a comma-separated list** of strings	ls {file1,file2}.txt (matches both file1.txt and file2.txt)
~	Represents the **home directory**	cd ~ (goes to home folder)
\	**Escapes** the next character (treat as normal)	echo * prints *

Shell Meta Characters in Linux

- **Shell metacharacters** are special characters that the shell interprets in a specific way to control input, output, command chaining, wildcard expansion, etc

Read & Echo

- The read and echo commands are fundamental tools for **interacting with users** in a shell script.
- read – takes **input** from the user.

Syntax: read [variable_name]

Ex -

```
echo "Enter your name:"

read name

echo "Hello, $name!"
```

Output:

```
Enter your name:     Vimal

Hello, Vimal!
```

- **read with Multiple Variables:**

```
echo "Enter two values:"

read a b

echo "First: $a, Second: $b"
```

- **read with Prompt (Using -p flag)**

```
read -p "Enter your course name: " course
echo "You are learning $course"
```

- **Silent Input (Password style) with -s**

```
read -sp "Enter password: " password
echo
echo "Password received."
```

echo – displays output to the terminal.

Newline (-e) - echo -e "Line1\nLine2"

No newline (-n) - echo -n "Same line "

Concepts of Operating Systems

- Vineela

Session 3: Shell Programming

Lecture:

- Decision loops (if else, test, nested if else, case controls, while…until, for)
- Regular expressions; Arithmetic expressions
- More examples in Shell Programming

Decision loops (if else, test, nested if else, case controls, while…until, for)

1. **If Statement**

Syntax:

```
if [ condition ]
        then
        Commands
    fi
```

Example:

```
if [ $age -ge 18 ]

        then

        echo "You are eligible to vote."

        fi
```

2. **if-else Statement**

Syntax:

```
if [ $marks -ge 35 ]

then

        echo "Pass"

else

        echo "Fail"
fi
```

3. if-elif-else Statement

```
if [ $marks -ge 75 ];
        then
                echo "Distinction"
        elif [ $marks -ge 35 ];
                then
                        echo "Pass"
                else
                        echo "Fail"
fi
```

4. Nested if Statement

```
if [ $age -ge 18 ];
        then
                if [ $citizen = "yes" ];
                                then
                                        echo "Eligible to vote"
                                else
                                        echo "Not a citizen"
                fi
                else
                        echo "Underage"
fi
```

test Command & case Statement (Switch-like)

- **test Command (Alternative to [ ])**

```
if test $a -gt $b
        then
                echo "$a is greater"
fi
```

★ [condition] is just a shortcut for test condition

- **case Statement (Switch-like)**

Syntax:

```
case word in
        pattern1)
                Statement(s) to be executed if pattern1 matches
        ;;
        pattern2)
                Statement(s) to be executed if pattern2 matches
         ;;
        pattern3)
                Statement(s) to be executed if pattern3 matches
         ;;
        *)
        Default condition to be executed
         ;;
esac
```

Example of Switch-like:

```
read -p "Enter choice (start/stop): " action
case $action in
                start)
                        echo "Starting service...";;
                stop)
                        echo "Stopping service...";;
                *)
                echo "Invalid option";;
 esac
```

Loop - while & for

- **while Loop**

```
                                                                    cou
nt=1
                while [ $count -le 5 ]
                do
                        echo "Count = $count"
                        count=$((count + 1))
                done
```

until Loop - Executes **until the condition becomes true**.

```
        n=1
                until [ $n -gt 3 ]
        do
                echo "n = $n"
                n=$((n+1))
        done
```

- **for Loop**

List-based:

```
for name in Alice Bob Charlie

do
        echo "Hi $name"

done
```

- **C-style:**

```
for (( i=1; i<=5; i++ ))
        do
                echo "i = $i"
        done
```

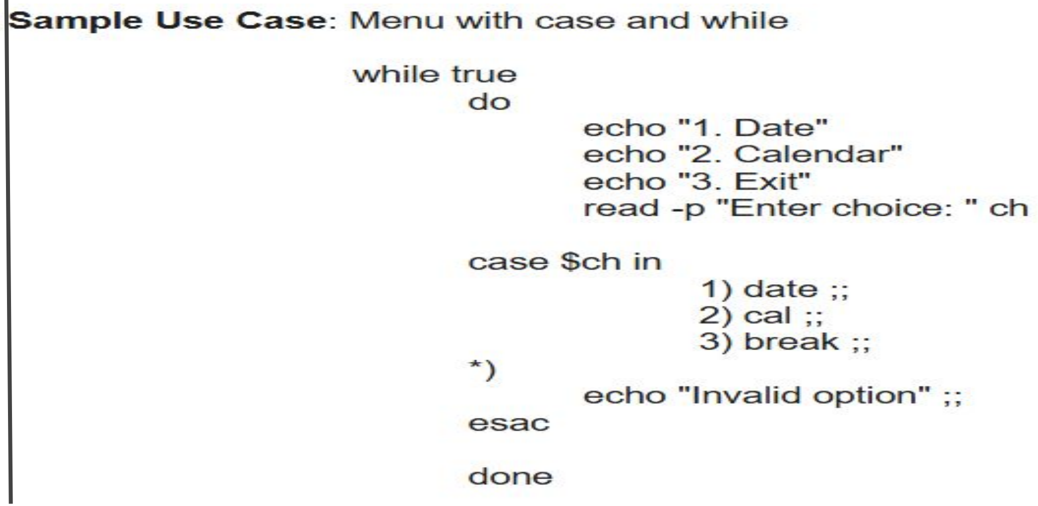
- **Loop Control Statements**

  break - Exits the loop

  continue - Skips current iteration, goes to

  next loop

  exit - Exits the script entirely



Sample Use Case: Menu with case and while

```
while true
        do
                echo "1. Date"
                echo "2. Calendar"
                echo "3. Exit"
                read -p "Enter choice: " ch

        case $ch in
                        1) date ;;
                        2) cal ;;
                        3) break ;;
        *)
                echo "Invalid option" ;;
        esac

        done
```

Regular Expressions

- **Regular expressions (regex)** are patterns **used to match character combinations in text**. In shell scripting, you commonly use them with tools like **grep, sed, and awk** for searching, replacing, and processing text.

Pattern	Meaning	Example	Matches
.	Any single character	c.t	cat, cut, c9t
*	Zero or more of previous char	lo*	l, lo, loo
^	Start of line	^Hi	Hi there
[^]	Not any of the chars inside	[^aeiou]	Any consonant
[]	Any one char inside	[aeiou]	a, e, i, etc.
$	End of line	bye$	goodbye
\	Escape special characters	\\.	A literal dot .

- Use grep -E or egrep for extended regex patterns (+, {}, \|, etc.)

Arithmetic Expressions

- **Using let Command**

```
let result=5+3

          echo $result
```

Output: 8

Note: No $ before variables inside let.

- **Using Double Parenthesis ((...)) Syntax**

```
a=10
b=5
((sum = a + b))
          echo "Sum = $sum"
```

Supported Operators → +, - , * , / , % , *= , += , -=

- **Using $((...)) for Inline Evaluation**

```
a=7
b=2
echo "Multiplication: $((a * b))"
```

- **Using expr (older but widely compatible)**

```
a=20
b=6
result=`expr $a / $b`
echo "Division: $result"
```

Note:

- ❖ You must add spaces between operands and operators.
- ❖ For **decimal numbers**, always use bc.
- ❖ Prefer (( )) and $(( )) over expr for modern scripts.
- ❖ Use let when assigning values directly.

Sessions 4 & 5: Processes

Lecture:

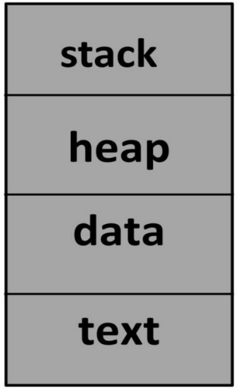
- What is process; preemptive and non-preemptive processes
- Difference between process and thread
- Process management; Process life cycle
- What are schedulers – Short term, Medium term and Long term
- Process scheduling algorithms – FCFS, Shortest Job First, Priority, RR, Queue. Belady's Anomaly
- Examples associated with scheduling algorithms to find turnaround time to find the better performing scheduler
- Process creation using fork; waitpid and exec system calls; Examples on process creation; Parent and child processes
- Orphan and zombie processes

What is Process?

- **Early computers** allowed **only one program** to be executed at a time but **contemporary computer systems** allow **multiple programs** to be loaded into memory and executed concurrently.
- A System consists of a collection of processes - **OS processes** executing system code and **user processes** executing user code
- All these processes execute concurrently with the CPU multiplexed among them.
- A process is a **program which is in execution**
- A process created by the main process is called a **child process.**
- Process management involves various tasks like creation, **scheduling**, termination of processes and a **dead lock**
- For a program to be executed, it must be mapped to absolute addresses and loaded into memory.
- As the program executes, it accesses program instructions and data from memory by generating these absolute addresses.
- Eventually, the program terminates, its memory space is declared available, and the next program can be loaded and executed

Process Architecture

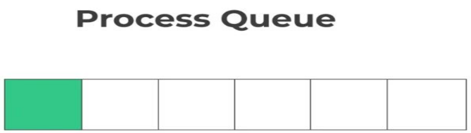
- **Stack:** The Stack stores temporary data like function parameters, returns addresses, and local variables.
- **Heap**: Allocates memory, which may be processed during run time.
- **Data:** It contains the global as well as static variable.
- **Text:** Text Section includes the current activity, which is represented by the value of the **Program Counter** (register within a CPU that holds the memory address of the next instruction to be executed)



Process Life Cycle

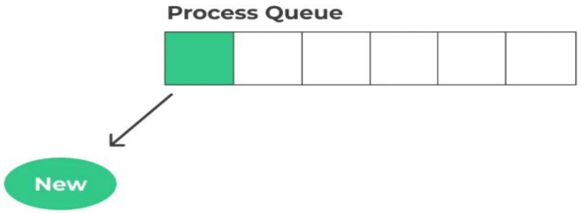
- **New** → When a process is started/created first, it is in this **New** state
- **Ready** → The process may enter this state after starting or while running, but the scheduler may interrupt it to assign the CPU to another process.
- **Running** → When the OS scheduler assigns a processor to a process, the process state gets set to running, and the processor executes the process instructions.
- **Waiting** → If a process needs to wait for any resource, such as for user input or for a file to become available, it enters the waiting state.
- **Terminated or Exit** → The process is relocated to the terminated state, where it waits for removal from the main memory once it has completed its execution or been terminated by the operating system.

- Imagine a unit process that executes a simple addition operation and prints it.



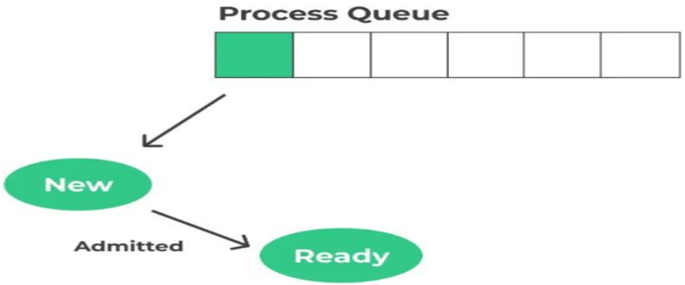
New State

- Process it submitted to the process queue, it in turns acknowledges submission.
- Once submission is acknowledged, the process is given new status.



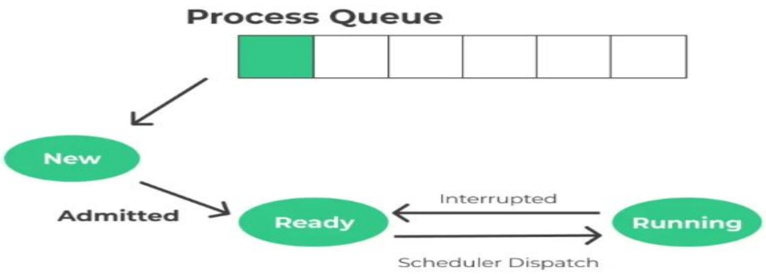
Ready State

- It then goes to Ready State, at this moment the process is waiting to be assigned a processor by the OS



Running State

- Once the Processor is assigned, the process is being executed and turns in Running State.



Wait and Termination State

- Now the process can follow the following transitions –
  - The process may have all resources it needs and may get directly executed and goes to Termination State.
  - Process may need to go to waiting state any of the following
    - Access to Input/Output device (Asking user the values that are required to be added) via console
    - Process maybe intentionally interrupted by OS, as a higher priority operation maybe required, to be completed first
    - A resource or memory access that maybe locked by another process, so current process goes to waiting state and waits for the resource to get free.
  - Once requirements are completed i.e. either it gets back the priority to executed or requested locked resources are available to use, the process will go to running state again where, it may directly go to termination state or may be required to wait again for a possible required input/resource/priority interrupt.
- Termination

Summary

1. New – New Process Created
2. Ready – Process Ready for Processor/computing power allocation
3. Running – Process getting executing
4. Wait – Process waiting for signal
5. Terminated – Process execution completed

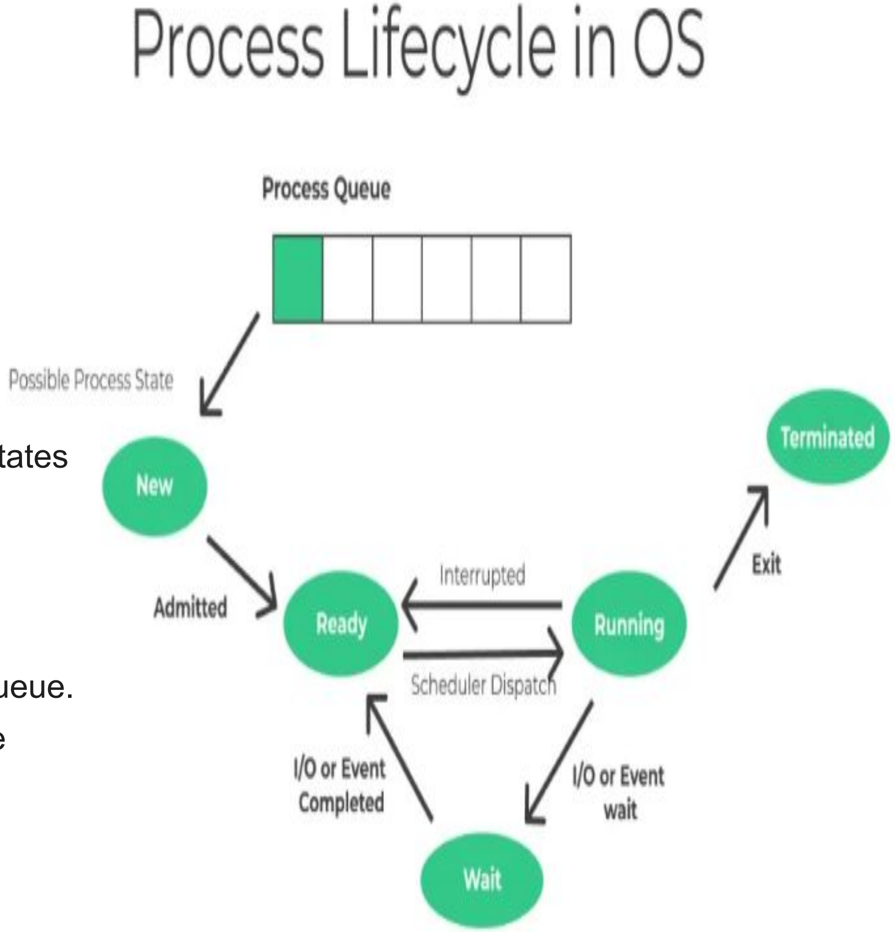
Apart from the above some new systems also propose 2 more states of process which are –

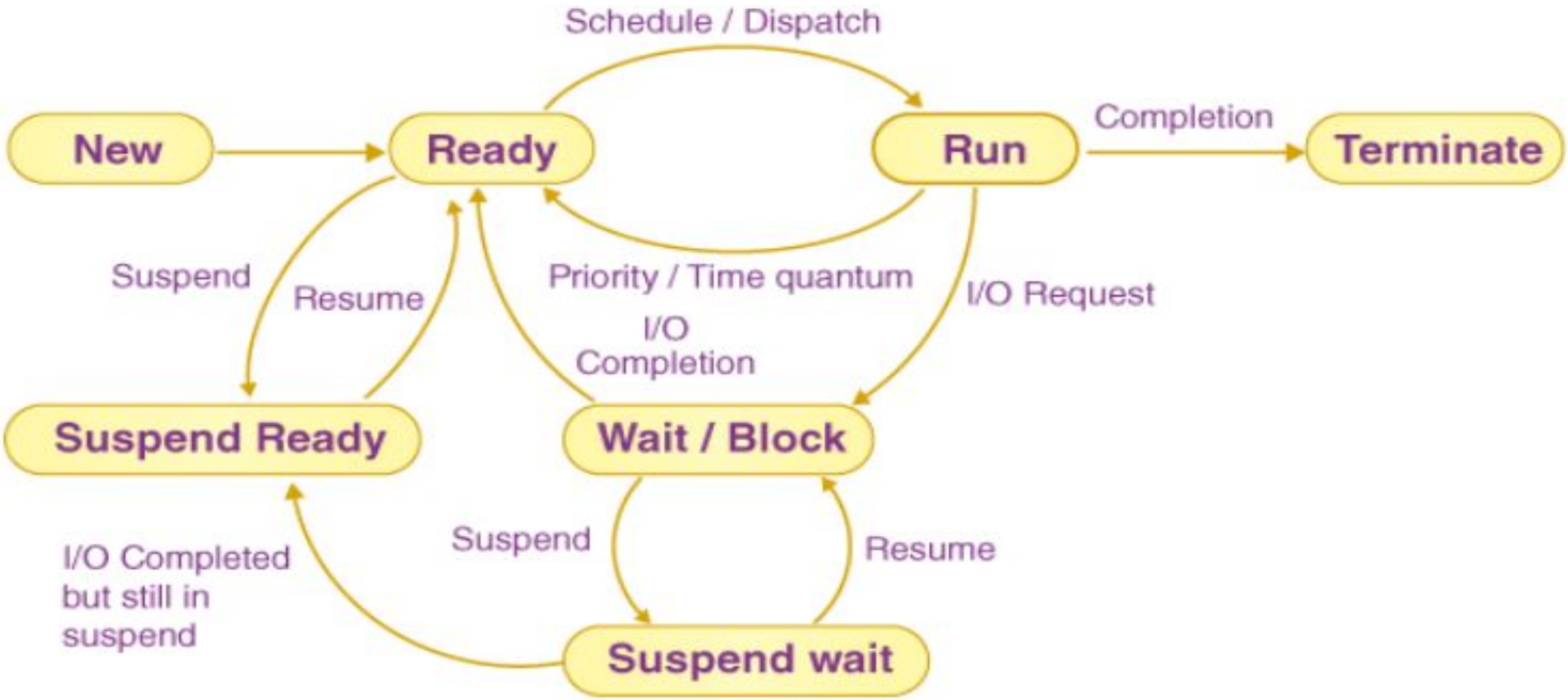
1. **Suspended Ready** –

   There maybe no possibility to add a new process in the queue. In such cases its can be said to be suspended ready state

2. **Suspended Block** –

   If the waiting queue is full

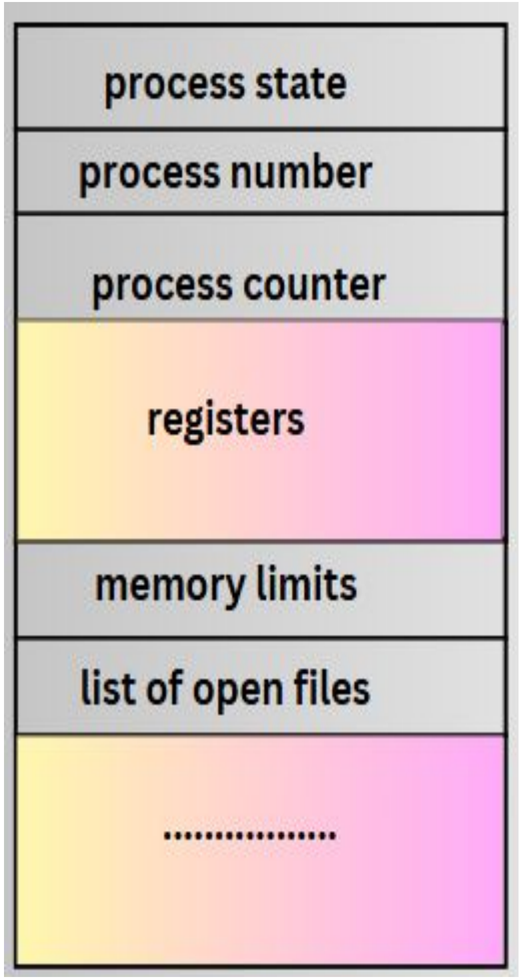

Process Lifecycle in OS



Process State Diagram

Process Control Block (PCB)

- Every process has a process control block, which is a data structure managed by the operating system.
- An integer process ID (or PID) is used to identify the PCB. Also called as **Task Control block**
- **Process state -** The process's present state may be it's new,ready, waiting,running,waiting, halted and so on.
- **Process privileges -** This is required in order to grant or deny access to system resources.
- **Process ID -** Each process in the OS has its own **unique** identifier.
- **Pointer -** It refers to a pointer that points to the **parent process**.
- **Program counter -** The program counter refers to a pointer that points to the address of the process's next instruction.
- **CPU registers -** Processes must be stored in various CPU registers for execution in the running state.



Process Control Block (PCB)

- **CPU scheduling information -** Process priority and additional scheduling information are required for the process to be scheduled
- **Memory management information -** This includes information from the page table, memory limitations, and segment table, all of which are dependent on the amount of memory used by the OS.
- **Accounting information -** This comprises CPU use for process execution, time constraints, and execution ID and other things.
- **IO status information -** This section includes a list of the process's I/O devices allocated to the process, a list of open files, and so on.
- PCB serves as the repository for any information that may very from process to process
- **The PCB architecture is fully dependent on the operating system, and different operating systems may include different information**

preemptive and non-preemptive processes

Preemptive process

- In operating systems, scheduling is the method by which processes are given access the CPU
- Process scheduling is performed by the CPU to decide the next process to be executed
- There are two primary types of CPU scheduling:
  a. preemptive
  b. non-preemptive

Different Types of CPU Scheduling Algorithms

There are mainly two types of scheduling methods:

Preemptive Scheduling Method:

- Preemptive scheduling is used **when a process switches from running state to ready state or from the waiting state to the ready state**
- If something is pre-emptive, it is done before other people can act, especially to prevent them from doing something else.

Non-preemptive scheduling

- Non-preemptive scheduling algorithms refer to the class of CPU scheduling technique where once a process is allocated the CPU, **it holds the CPU till the process gets terminated or is pushed to the waiting state.**
- No process is interrupted until it runs to completion.
- The scheduler allocates another process to the CPU only after the currently allocated process terminates and relinquishes control on the CPU.

Difference Between Preemptive and Non-Preemptive Scheduling

Preemptive Scheduling	Non-Preemptive Scheduling
Resources are allocated according to the cycles for a limited time.	Resources are used and then held by the process until it gets terminated.
The process can be interrupted, even before the completion.	The process is not interrupted until its life cycle is complete.
Starvation may be caused, due to the insertion of priority process in the queue.	Starvation can occur when a process with large burst time occupies the system.
Maintaining queue and remaining time needs storage overhead.	No such overheads are required.
Fair scheduling can be applied where all the processes can get equal chance for CPU access	A process may monopolize the CPU.
Deadlocks can be easily avoided.	Deadlocks may occur.

What is the Need for a CPU Scheduling Algorithm?

- It ensure that whenever the CPU remains idle, the OS has at least selected one of the processes available in the ready-to-use line.
- In Multiprogramming, if the long-term scheduler selects multiple I/O binding processes then most of the time, the CPU remains idle.

Terminologies Used in CPU Scheduling

- **Arrival Time:** The time at which the process arrives in the ready queue.
- **Completion Time:** The time at which the process completes its execution.
- **Burst Time:** Time required by a process for CPU execution.
- **Turn Around Time:** Time Difference between completion time and arrival time.
- **Waiting Time(W.T):** Time Difference between turn around time and burst time.
- **Turn Around Time = Completion Time – Arrival Time**
- **Waiting Time = Turn Around Time – Burst Time**

Thread

- A thread refers to an **execution unit in the process** that has its own programme counter, stack, as well as a set of registers.
- Now, thread execution is possible within any OS's process. Furthermore, **each and every thread belongs to one single process.**
- Multiple threads can easily communicate information and important data, such as code segments or files, as well as data segments.
- Apart from that, a process can have several threads

Ex - Multiple tabs in a browser, for example, can be considered threads.

MS Word employs many threads to prepare the text in one thread, receive input in another thread, and so on.

Components of Thread

A thread has the following three components:

1. Program Counter
2. Register Set
3. Stack space

Why do we need Thread?

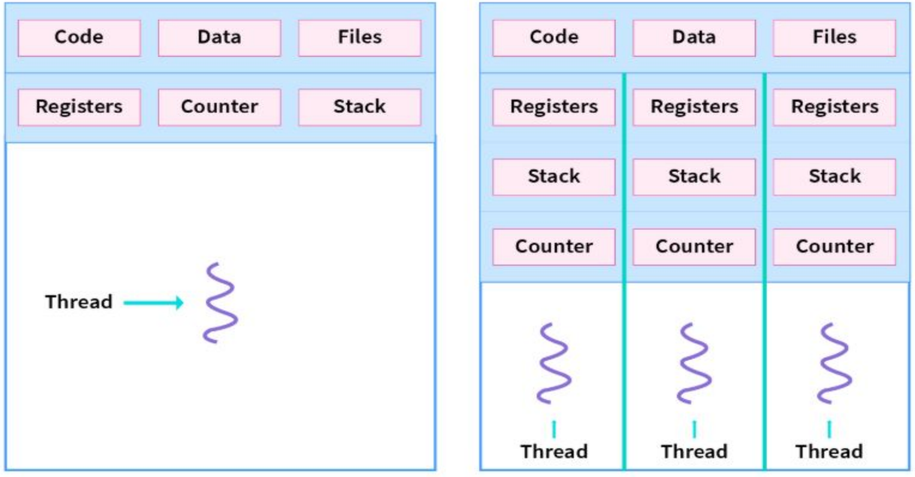
- Creating a new thread in a current process requires significantly less time than creating a new process.
- Threads can share common data without needing to communicate with each other.
- When working with threads, context switching is faster.
- Terminating a thread requires less time than terminating a process.

Single and Multi threaded Process

- Single threaded processes contain the execution of instructions in a single sequence. In other words, **one command is processes at a time.**
- The opposite of single threaded processes are multithreaded processes.
- These processes **allow the execution of multiple parts of a program at the same time.**
- These are lightweight processes available within the process.

Multithreaded Processes Implementation

- Multithreaded processes can be implemented as user-level threads or kernel-level threads.



Single-threaded process

Multithreaded process

User-level Threads and Kernel level Threads

User-level Threads

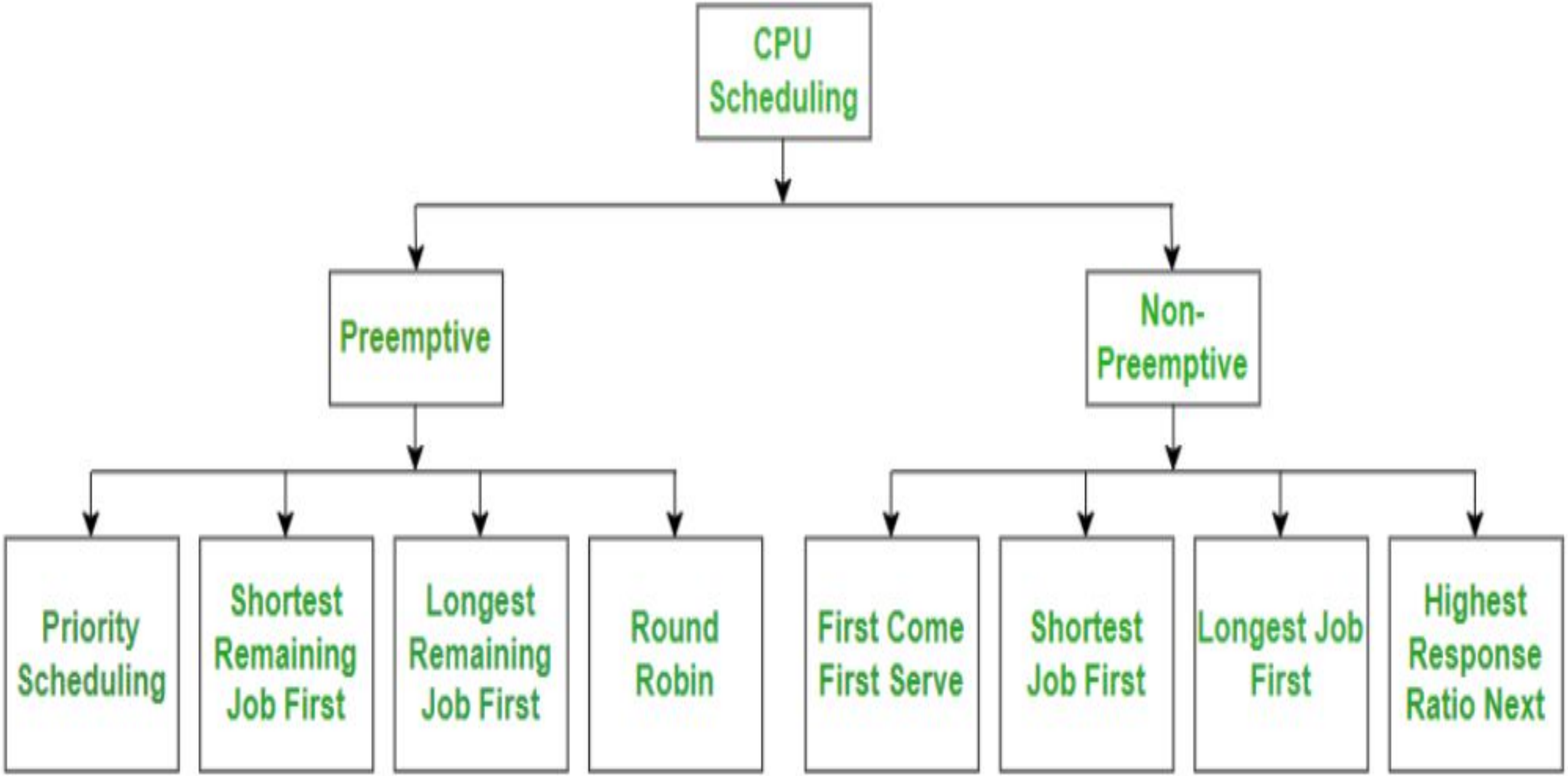
- The user-level threads are implemented by users and the kernel is not aware of the existence of these threads.
- It handles them as if they were single-threaded processes.
- User-level threads are small and much faster than kernel level threads.
- Also, there is no kernel involvement in synchronization for user-level threads.

Kernel-level Threads

- Kernel-level threads are handled by the operating system directly and the thread management is done by the kernel.
- The context information for the process as well as the process threads is all managed by the kernel.
- Because of this, kernel-level threads are slower than user-level threads.

Process vs Thread

Process	Thread
Processes use more resources and hence they are termed as heavyweight processes.	Threads share resources and hence they are termed as lightweight processes.
Creation and termination times of processes are slower.	Creation and termination times of threads are faster compared to processes.
Processes have their own code and data/file.	Threads share code and data/file within a process.
Communication between processes is slower.	Communication between threads is faster.
Context Switching in processes is slower.	Context switching in threads is faster.
Processes are independent of each other.	Threads, on the other hand, are interdependent. (i.e they can read, write or change another thread's data)
Eg: Opening two different browsers.	**Eg**: Opening two tabs in the same browser.



CPU Scheduling

Preemptive

Non-Preemptive

Priority Scheduling

Shortest Remaining Job First

Longest Remaining Job First

Round Robin

First Come First Serve

Shortest Job First

Longest Job First

Highest Response Ratio Next

What are schedulers - Short term, Medium term and Long term

Short-Term Scheduler (CPU scheduler)

- The short-term scheduler selects processes from the **ready queue** that are residing in the main memory and allocates **CPU** to one of them.
- As compared to long-term schedulers, a short-term scheduler has to be used very often i. e. **the frequency of execution of short-term schedulers is high.**
- The short-term scheduler is invoked whenever an event occurs. Such an event may lead to the **interruption of the current process** or it may provide an opportunity to preempt the currently running process in favor of another.
- The example of such events are:
  - Clock ticks (time-based interrupts)
  - I/O interrupts and 1/0 completions
  - Operating system calls
  - Sending and receiving of signals

Medium-Term Scheduler

- The medium-term scheduler is required at the times when a **suspended** or **swapped-out process is to be brought into a pool of ready processes.**
- This is done because there is a limit on the number of active processes that can reside in the main memory.
- The medium-term scheduler is **in-charge of handling the swapped-out process**.
- It has nothing to do with when a process remains suspended.
- However, once the suspending condition is removed, the medium terms scheduler attempts to allocate the required amount of main memory and swap the process in and make it ready.
- Thus, the medium-term scheduler plans the CPU scheduling for processes that have been waiting for the completion of another process or an I/O task.

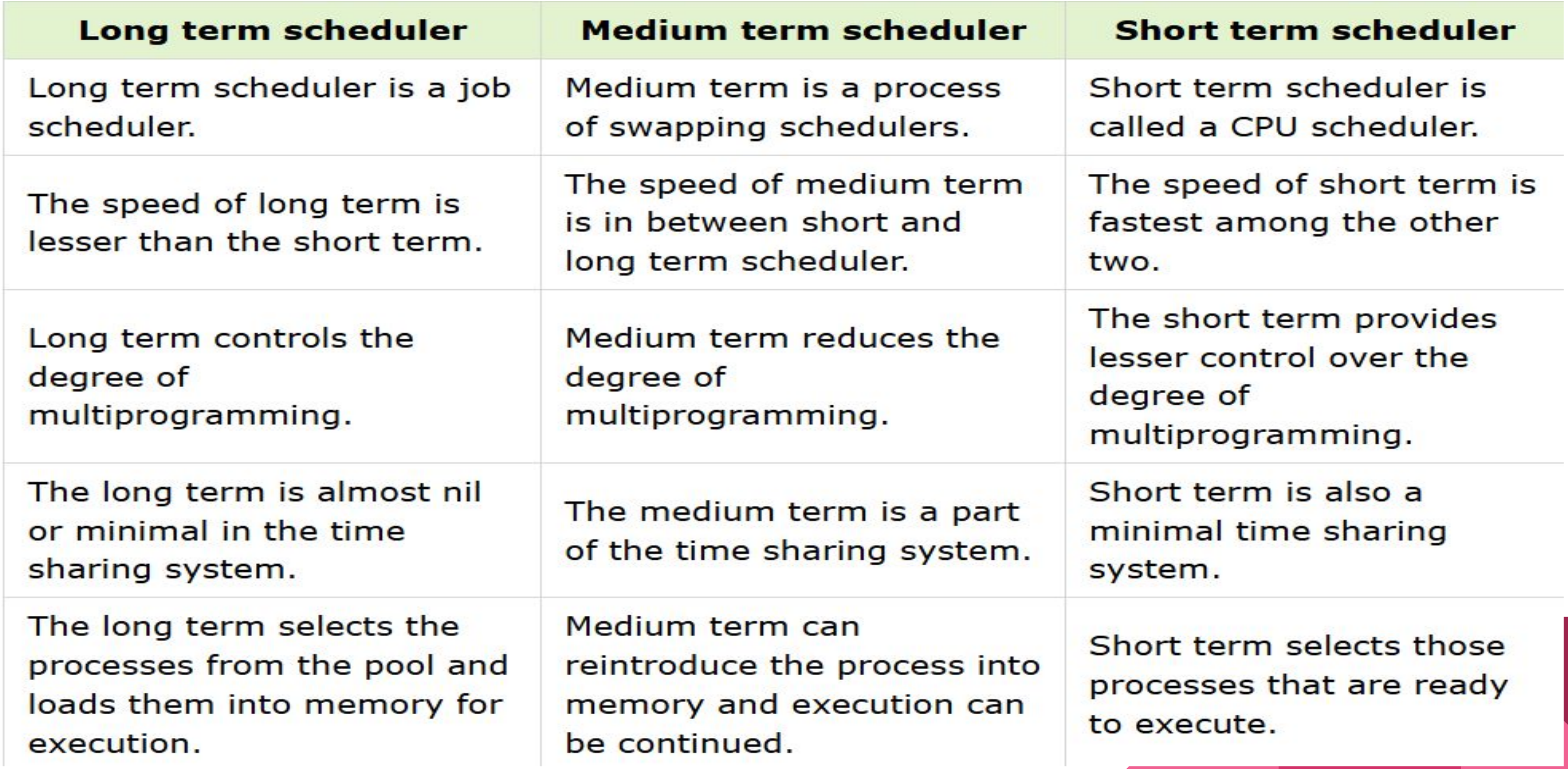
Long-Term Scheduler (job scheduler)

- The long-term scheduler works with the **batch queue** and selects the **next batch job** to be executed.
- Processes, which are resource intensive and have a low priority are called **batch jobs.**
- **For example, a user requests for printing a bunch of files.**
- We can also say that a **long-term scheduler selects the processes or jobs from secondary storage device**
- **eg, a disk and loads them into the memory for execution**.
- The long-term scheduler is called “**long-term**” **because the time for which the scheduling is valid is long**.
- This scheduler shows the **best performance** by selecting a good process mix of I/O-bound and CPU-bound processes.

Summary of Short term, Medium term and Long term Schedulers

- **Short Term Scheduler** decides which process to execute next, responsible for CPU scheduling, or It carry process from ready queue to running state.
- **Medium Term Scheduler** handles process swapping between main memory and secondary storage, manages the degree of multiprogramming.
- **Long Term Scheduler** decide which processes to admit to the system, or it selects the processes or jobs from secondary storage device eg, a disk and loads them into the main memory for execution.

Short term VS Medium term VS Long term Schedulers



Long term scheduler	Medium term scheduler	Short term scheduler
Long term scheduler is a job scheduler.	Medium term is a process of swapping schedulers.	Short term scheduler is called a CPU scheduler.
The speed of long term is lesser than the short term.	The speed of medium term is in between short and long term scheduler.	The speed of short term is fastest among the other two.
Long term controls the degree of multiprogramming.	Medium term reduces the degree of multiprogramming.	The short term provides lesser control over the degree of multiprogramming.
The long term is almost nil or minimal in the time sharing system.	The medium term is a part of the time sharing system.	Short term is also a minimal time sharing system.
The long term selects the processes from the pool and loads them into memory for execution.	Medium term can reintroduce the process into memory and execution can be continued.	Short term selects those processes that are ready to execute.

FCFS - First Come First Serve CPU Scheduling

- The processes are attended to in the order **in which they arrive in the ready queue**, much **like customers lining up at a grocery store.**

How Does FCFS Work?

1. **Arrival:** Processes enter the system and are placed in a queue in the order they arrive.
2. **Execution:** The CPU takes the first process from the front of the queue, executes it until it is complete, and then removes it from the queue.
3. **Repeat:** The CPU takes the next process in the queue and repeats the execution process.

This continues until there are no more processes left in the queue.

FCFS CPU Scheduling:

Example of FCFS CPU Scheduling:

To understand the First Come, First Served (FCFS) scheduling algorithm effectively, we’ll use two examples –

- one where all processes arrive at the same time,
- another where processes arrive at different times.

We’ll create **Gantt charts** for both scenarios and **calculate the turnaround time and waiting time for each process**.

Scenario 1: Processes with Same Arrival Time

Consider the following table of arrival time and burst time for three processes P1, P2 and P3

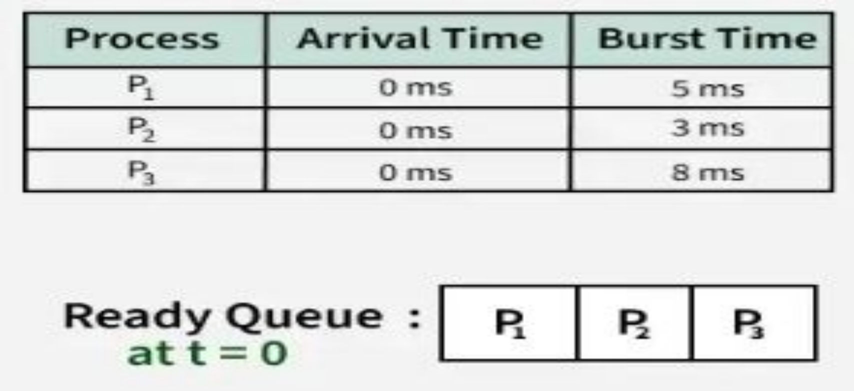
process	Arrival Time	Burst Time
P1	0	5
P2	0	3
P3	0	8

Step-by-Step Execution:

1. **P1** will start first and run for 5 units of time (from 0 to 5).
2. **P2** will start next and run for 3 units of time (from 5 to 8).
3. **P3** will run last, executing for 8 units (from 8 to 16).

FCFS Scheduling

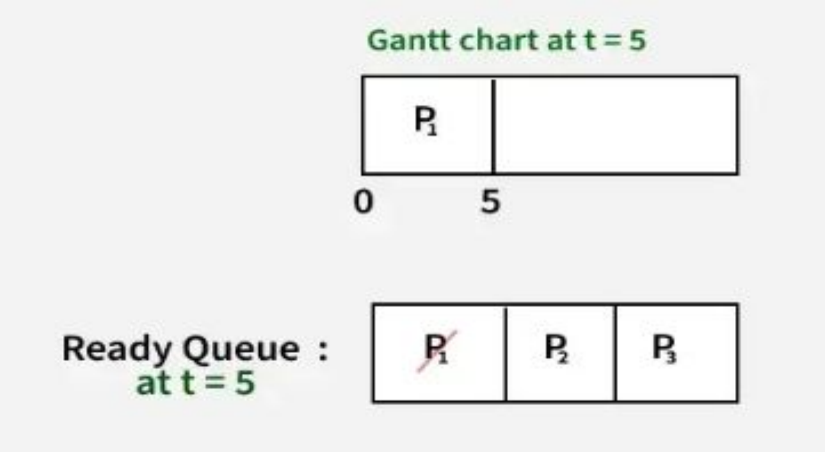
Step No 1



Process	Arrival Time	Burst Time
P_1	0 ms	5 ms
P_2	0 ms	3 ms
P_3	0 ms	8 ms

Ready Queue : P_1 | P_2 | P_3
at t = 0

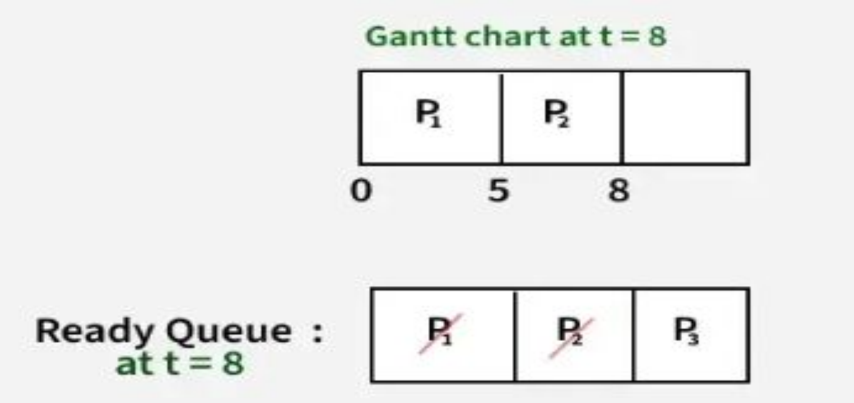
Step No 2



Gantt chart at t = 5

Ready Queue : ~~P_1~~ | P_2 | P_3
at t = 5

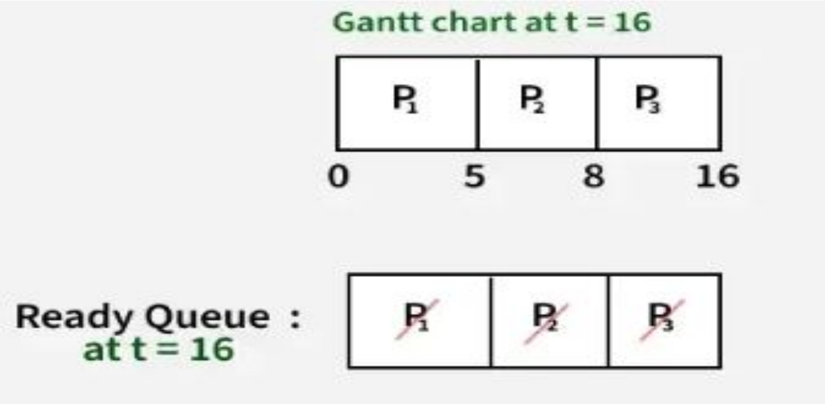
Step No 3



Gantt chart at t = 8

Ready Queue : ~~P_1~~ | ~~P_2~~ | P_3
at t = 8

Step No 4



Gantt chart at t = 16

Ready Queue : ~~P_1~~ | ~~P_2~~ | ~~P_3~~
at t = 16

Turnaround Time = Completion Time - Arrival Time

Waiting Time = Turnaround Time - Burst Time

AT : Arrival Time
BT : Burst Time or CPU Time
TAT : Turn Around Time
WT : Waiting Time

- **Average Turn around time** = 9.67
- **Average waiting time** = 4.33

Processes	AT	BT	CT	TAT	WT
P1	0	5	5	5-0 = 5	5-5 = 0
P2	0	3	8	8-0 = 8	8-3 = 5
P3	0	8	16	16-0 = 16	16-8 = 8

Scenario 2: Processes with Different Arrival Times

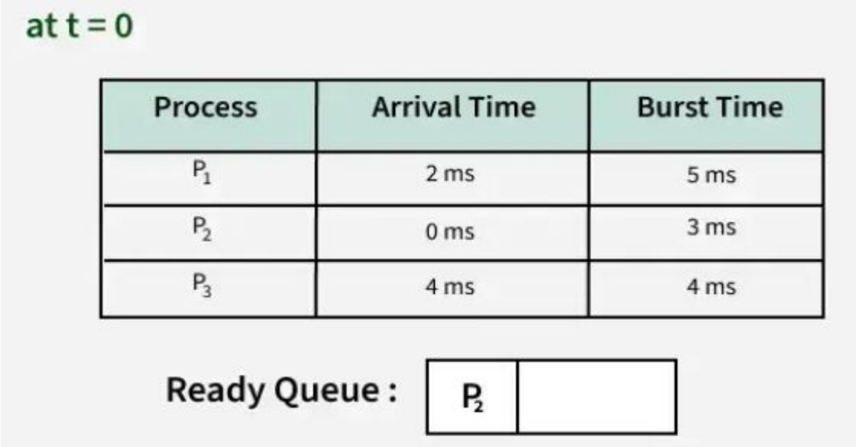
Consider the following table of arrival time and burst time for three processes P1, P2 and P3

Process	Burst Time (BT)	Arrival Time (AT)
P1	5 ms	2 ms
P2	3 ms	0 ms
P3	4 ms	4 ms

Step-by-Step Execution:

- **P2** arrives at time 0 and runs for 3 units, so its completion time is:
  Completion Time of P2 = 0 + 3 =3
- **P1** arrives at time 2 but has to wait for **P2** to finish. **P1** starts at time 3 and runs for 5 units. Its completion time is:
  Completion Time of P1 = 3 + 5 = 8
- **P3** arrives at time 4 but has to wait for **P1** to finish. **P3** starts at time 8 and runs for 4 units. Its completion time is:
  Completion Time of P3 = 8 + 4 = 12

Step No 1



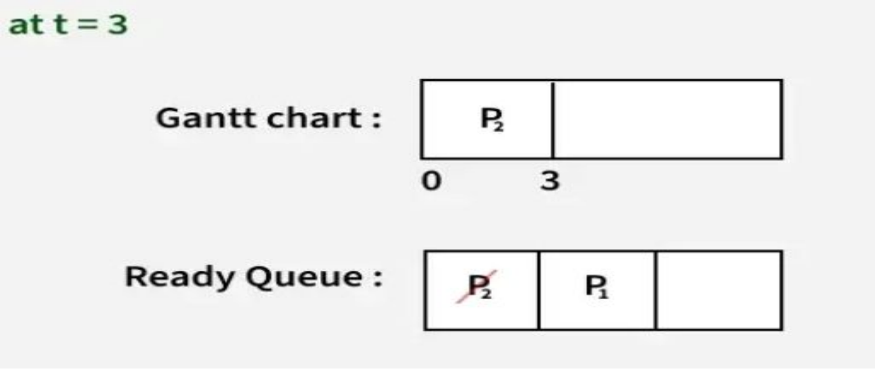
at t = 0

Process	Arrival Time	Burst Time
P_1	2 ms	5 ms
P_2	0 ms	3 ms
P_3	4 ms	4 ms

Ready Queue :

P_2	

Step No 2



at t = 3

Gantt chart :

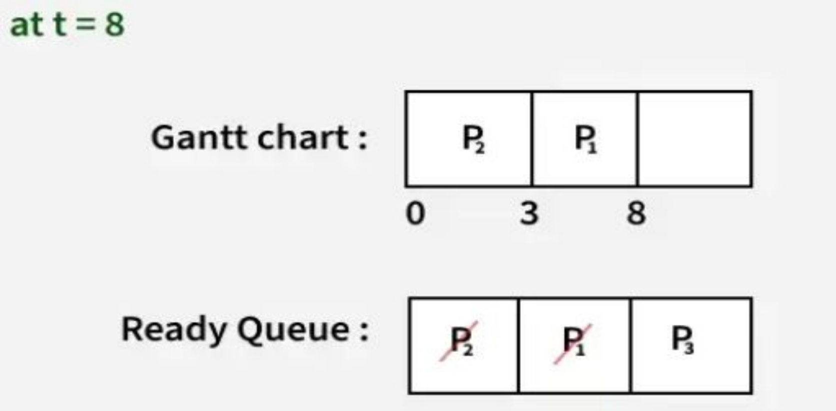
P_2	

0 3

Ready Queue :

~~P_2~~	P_1	

Step No 3



at t = 8

Gantt chart :

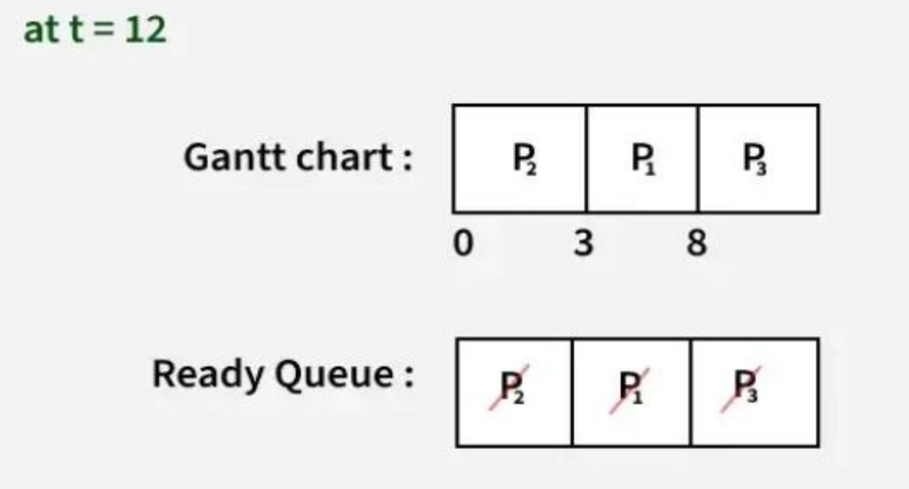
P_2	P_1	

0 3 8

Ready Queue :

~~P_2~~	~~P_1~~	P_3

Step No 4



at t = 12

Gantt chart :

P_2	P_1	P_3

0 3 8

Ready Queue :

~~P_2~~	~~P_1~~	~~P_3~~

Now, lets calculate average waiting time and turn around time:

- **Average Turnaround time** = 5.67
- **Average waiting time** = 1.67

Process	Completion Time (CT)	Turnaround Time (TAT = CT – AT)	Waiting Time (WT = TAT – BT)
P2	3 ms	3 ms	0 ms
P1	8 ms	6 ms	1 ms
P3	12 ms	8 ms	4 ms

Advantages and Disadvantages of FCFS

Advantages of FCFS

- The simplest and basic form of CPU Scheduling algorithm
- Every process gets a chance to execute in the order of its arrival.
- Easy to implement, it doesn’t require complex data structures.
- It is well suited for batch systems where the longer time periods for each process are often acceptable.

Disadvantages of FCFS

- FCFS can result in long waiting times, especially if a long process arrives before a shorter one. This is known as the **convoy effect**, where shorter processes are forced to wait behind longer processes, leading to inefficient execution.
- The average waiting time in the FCFS is much higher than in the others
- Processes that are at the end of the queue, have to wait longer to finish.
- It is not suitable for time-sharing operating systems where each process should get the same amount of CPU time.

SJF (SHORTEST JOB FIRST) Scheduling or Shortest Job Next (SJN)/Shortest Remaining Time First (SRTF)

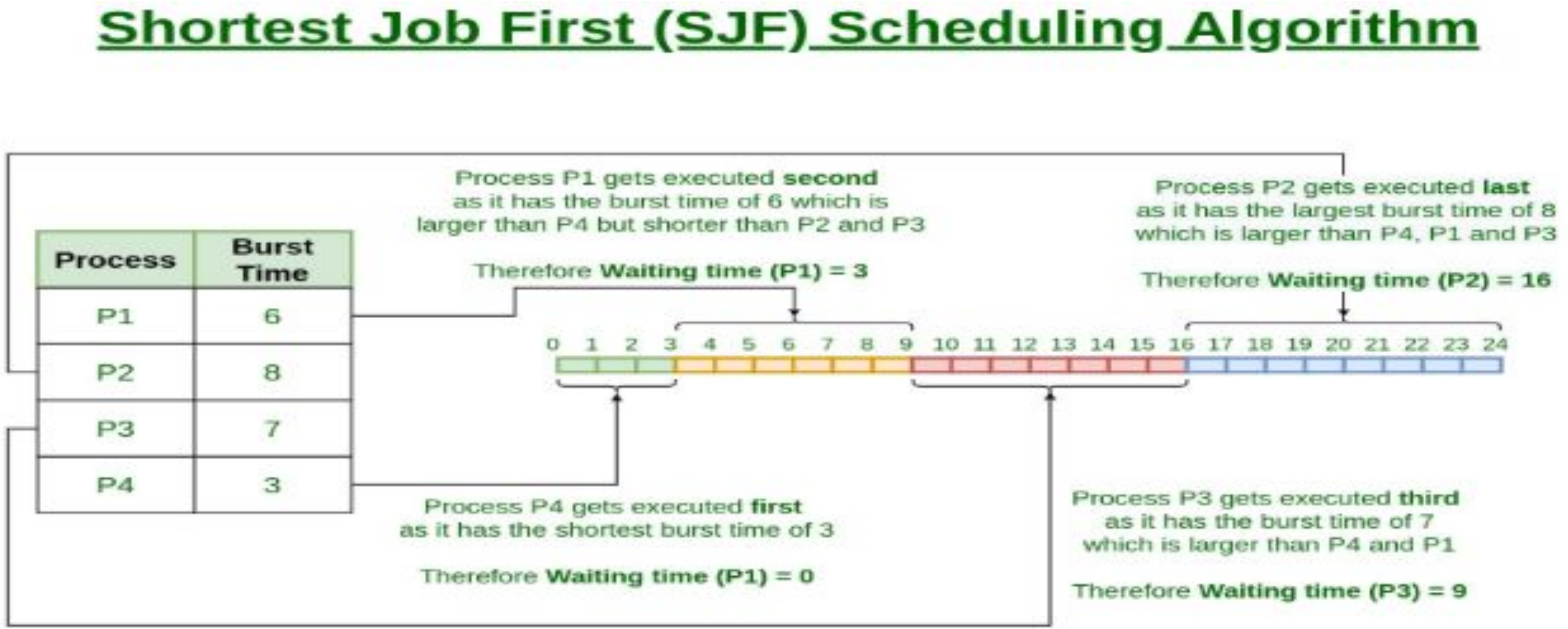
- In the Shortest Job First scheduling algorithm, the processes are **scheduled in ascending order of their CPU burst times**
- Here, if a short process enters the ready queue while a longer process is executing, process switch occurs by which the executing process is swapped out to the ready queue while the newly arrived shorter process starts to execute.
- Thus the short term scheduler is invoked either when a new process arrives in the system or an existing process completes its execution.

Features of SJF Algorithm

- SJF allocates CPU to the process with shortest execution time.
- In cases where two or more processes have the same burst time, arbitration is done among these processes on first come first serve basis.
- There are both preemptive and non-premptive
- It minimises the average waiting time of the processes.

Implementation of SJF Scheduling

- Sort all the processes according to the arrival time.
- Then select that process that has minimum arrival time and minimum Burst time.
- After completion of the process make a pool of processes (a ready queue) that arrives afterward till the completion of the previous process and select that process in that queue which is having minimum Burst time.



Shortest Job First (SJF) Scheduling Algorithm

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

Process P1 gets executed **second** as it has the burst time of 6 which is larger than P4 but shorter than P2 and P3

Therefore **Waiting time (P1) = 3**

Process P2 gets executed **last** as it has the largest burst time of 8 which is larger than P4, P1 and P3

Therefore **Waiting time (P2) = 16**

Process P4 gets executed **first** as it has the shortest burst time of 3

Therefore **Waiting time (P1) = 0**

Process P3 gets executed **third** as it has the burst time of 7 which is larger than P4 and P1

Therefore **Waiting time (P3) = 9**

Generalized Activity Normalization Time Table (GANTT) chart

- It is a production control tool and horizontal bar chart used for graphical representation of schedule that helps to plan in an efficient way, coordinate, and track some particular tasks in project.

Advantages of SJF Scheduling

- SJF reduces the average waiting time.
- SJF is generally used for long term scheduling.
- It is suitable for the jobs running in batches, where run times are already known.
- SJF is probably optimal in terms of average Turn Around Time (TAT).

Disadvantages of SJF Scheduling

- In SJF job completion time must be known earlier.
- Many times it becomes complicated to predict the length of the upcoming CPU request.

Priority Scheduling

- **Priority scheduling** is one of the **most common scheduling algorithms** used by the operating system to schedule processes based on their priority. Each process is assigned a priority.
- The process with the **highest priority is to be executed first** and so on.
- **Processes with the same priority are executed on a first-come first served basis**.

Ways to decide the Priority

- Priority can be decided based on memory requirements, time requirements or any other resource requirement.
- Also priority can be decided on the ratio of average I/O to average CPU burst time.

Priority Scheduling can be implemented in two ways: Non-Preemptive Priority Scheduling and Preemptive Priority Scheduling

Non-Preemptive Priority Scheduling

- In Non-Preemptive Priority Scheduling, the CPU is not taken away from the running process. Even if a higher-priority process arrives, the currently running process will complete first.

Ex: A high-priority process must wait until the currently running process finishes.

Example of Non-Preemptive Priority Scheduling:

Consider the following table of arrival time and burst time for three processes P1, P2 and P3:

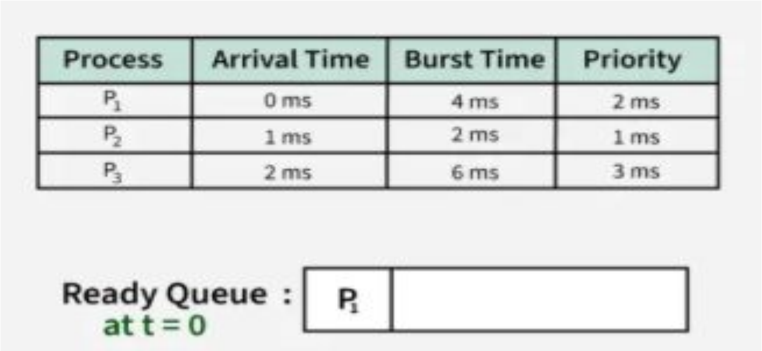
Process	Arrival Time	Burst Time	Priority
P1	0	4	2
P2	1	2	1
P3	2	6	3

Note: Lower number represents higher priority.

Step-by-Step Execution:

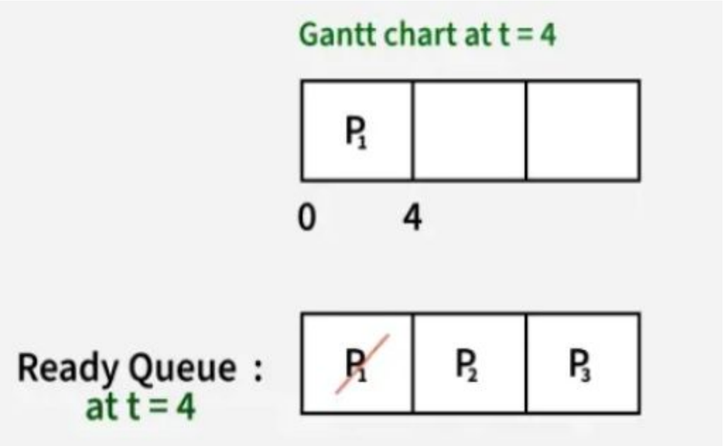
- **At Time 0:** Only P1 has arrived. P1 starts execution as it is the only available process, and it will continue executing till t = 4 because it is a non-preemptive approach.
- **At Time 4:** P1 finishes execution. Both P2 and P3 have arrived. Since P2 has the highest priority (Priority 1), it is selected next.
- **At Time 6:** P2 finishes execution. The only remaining process is P3, so it starts execution.
- **At Time 12:** P3 finishes execution

Step No 1

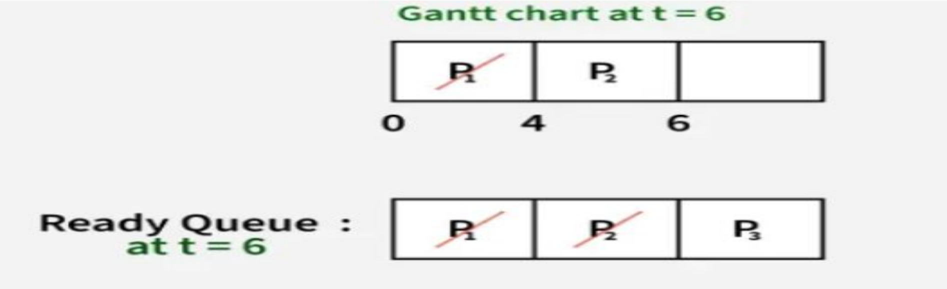


Process	Arrival Time	Burst Time	Priority
P_1	0 ms	4 ms	2 ms
P_2	1 ms	2 ms	1 ms
P_3	2 ms	6 ms	3 ms

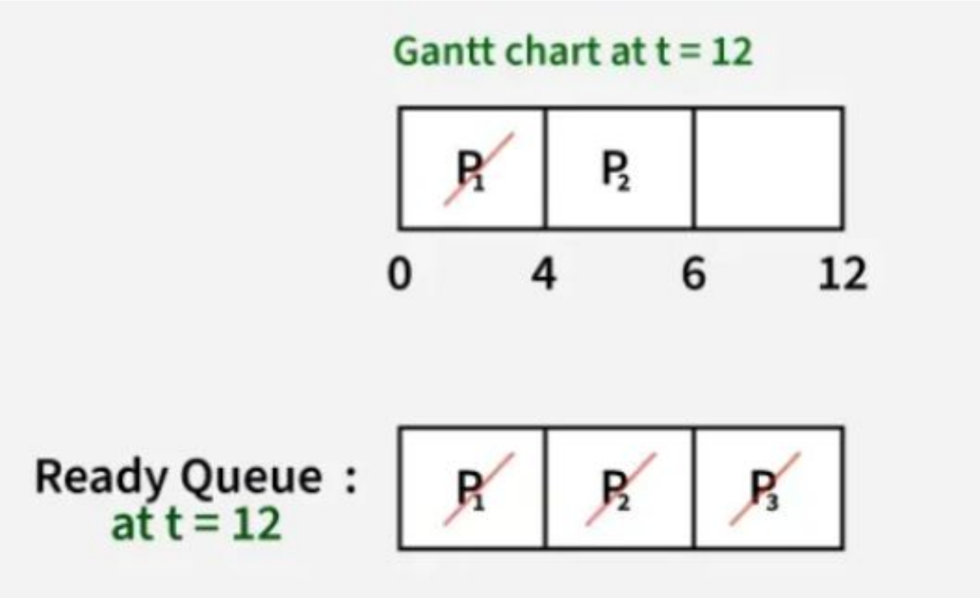
Step No 2



Step No 3



Step No 4



Advantages and Disadvantages of Priority Scheduling

Advantages of Priority Scheduling

- Implementation is simple, since scheduler doesnot require doing any prior calculations.
- Once CPU defines the relative relevance (priorities) of the processes, the order of execution is easily predictable.
- Higher priority processes are almost served immediately.
- Priority scheduling is particularly helpful in systems that have variety of processes each with their own needs.

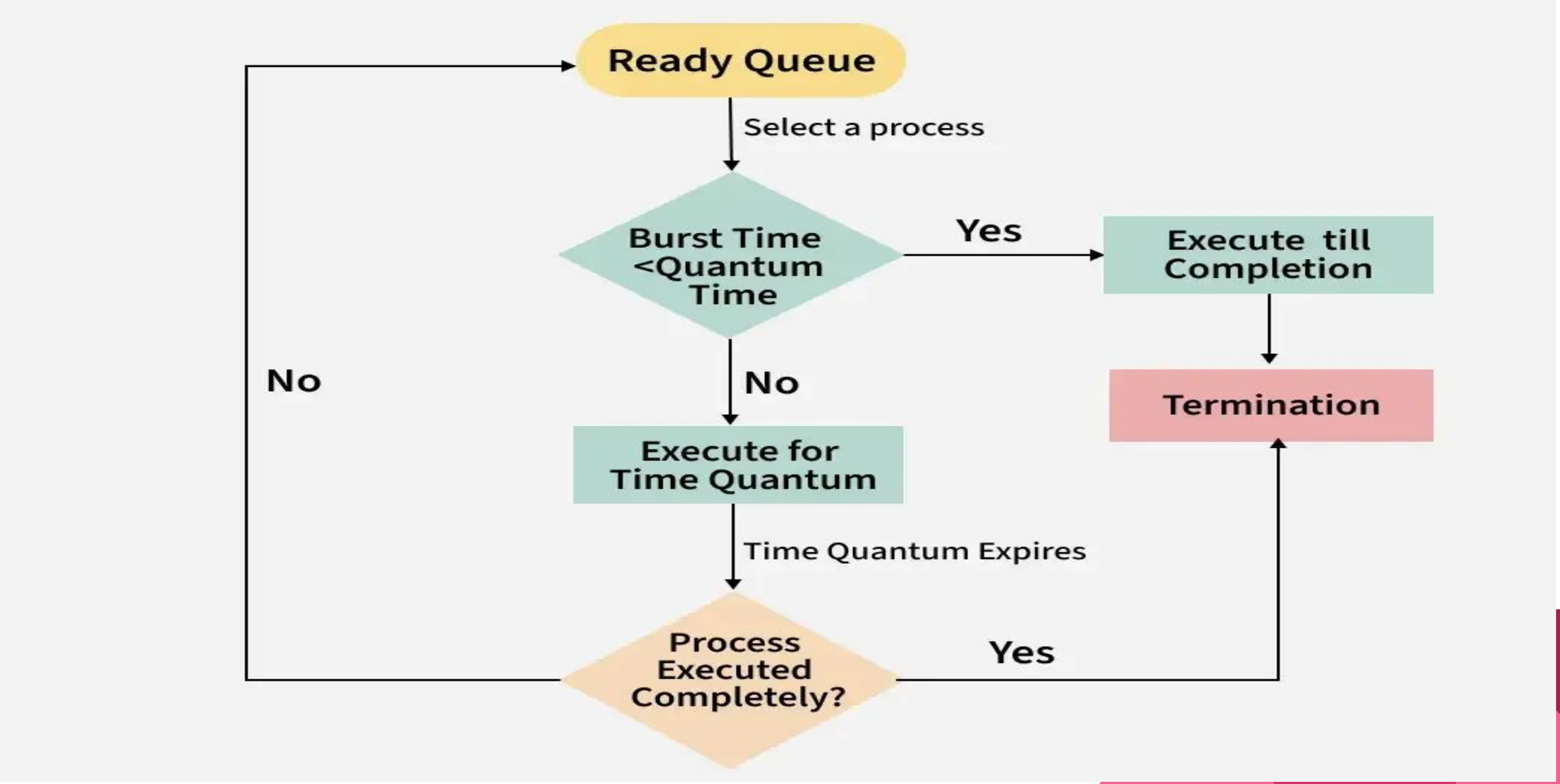
Disadvantages of Priority Scheduling

- In static priority systems, lower priority processes may need to wait indefinitely, this results in stagnation.
- In non-preemptive priority scheduling, often a large process keeps shorter processes waiting for long time.
- In preemptive priority scheduling, a low priority process may be repeatedly pre-empted by intermittent streams of high priority processes requiring frequent context switches.

Round Robin (RR) Scheduling

- **Round Robin Scheduling is one of the most efficient** and the most widely used **not only in process scheduling in operating systems but also in network scheduling.**
- This scheduling strategy derives its name from an age **old round-robin principle which advocated that all participants are entitled to equal share of assets or opportunities in a turn wise manner.**
- In RR scheduling, **each process gets equal time slices (or time quanta) for which it executes in the CPU in turn wise manner.**
- When a process gets its turn, it executes for the assigned time slice and then relinquishes the CPU for the next process in queue.
- If the process has burst time left, then it is sent to the end of the queue. Processes enter the queue on first come first serve basis.
- Round Robin scheduling is **preemptive**, which means that a running process can be interrupted by another process and sent to the ready queue even when it has not completed its entire execution in CPU.
- It is a **preemptive version of First Come First Serve (FCFS) scheduling algorithm**.

Working of Round Robin Scheduling

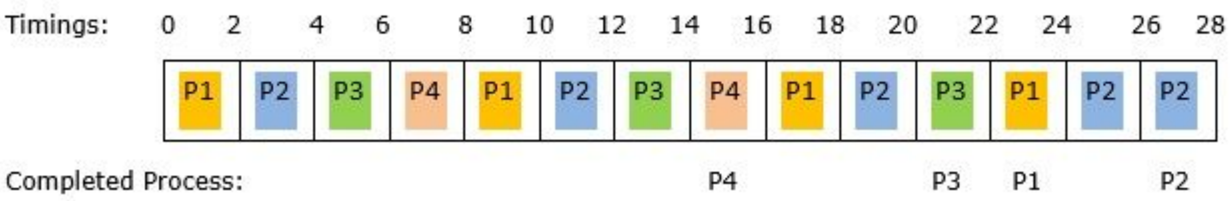


Example of Round Robin Scheduling

- Let us consider a system that has four processes which have arrived at the same time
- in the order P1, P2, P3 and P4.
- 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.

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

GANTT Chart with time quantum of 2ms



Average Turnaround Time

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

  (TATP1+TATP2+TATP3+TATP4) / 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

  = (WTP1+WTP2+WTP3+WTP4)/4

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

- **Average Waiting Time** = Sum of Waiting Time of Each Process / Number of processes

  = (WTP1 + WTP2 + WTP3 + WTP4) / 4

  = (0 + 2 +15 + 8) / 4 = **6.25 ms**

Advantages of Round Robin Scheduling

- Round Robin scheduling is the most a fair scheduling algorithm
- Starvation is totally eliminated in RR scheduling.
- It does not require any complicated method to calculate the CPU burst time
- It is pretty simple to implement and so finds application in a wide range of situations.
- Convoy effect does not occur in RR scheduling

Disadvantages of Round Robin Scheduling

- The performance is highly dependent upon the chosen time quantum. This requires prudent analysis before implementation, failing which required results are not received.
- RR scheduling does not give any scope to assign priorities to processes. So, system processes which need high priority gets the same preference as background processes. This may often hamper the overall performance of a system.

Types of Scheduling Queues

Job Queue (In Disk)

- This queue contains all the processes or jobs in the list that are waiting to be processed.
- When a job is created, it goes into the job queue and waits until it is ready for processing.
  - Contains all submitted jobs.
  - Processes are stored here in a wait state until they are ready to go to the execution stage.
  - This is the first and most basic state that acts as a default storage of new jobs added to a scheduling system.
  - Long Term Scheduler Picks a process from Job Queue and moves to ready queue.

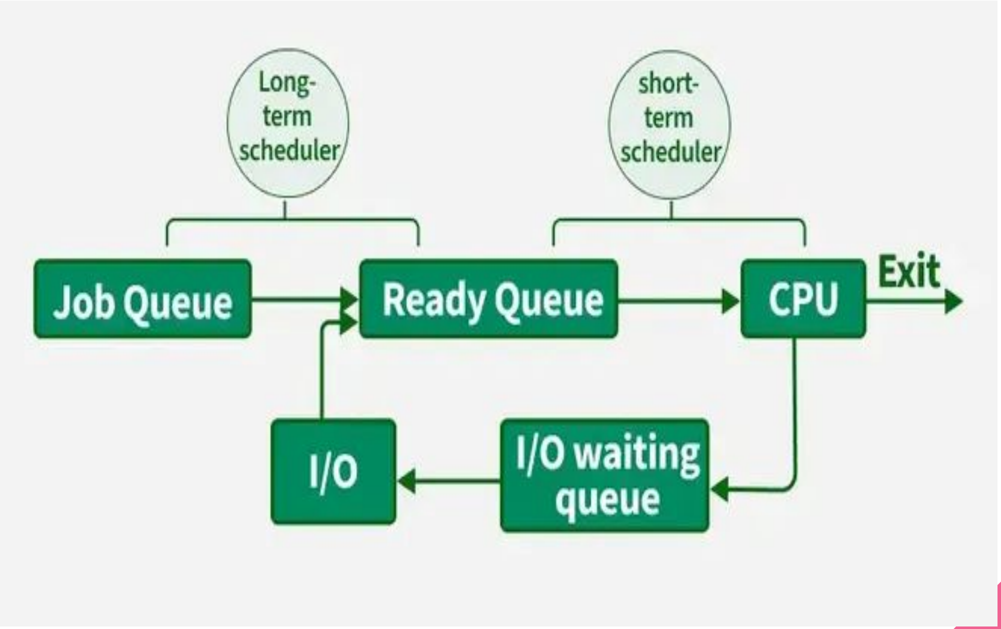
Ready Queue (In Main Memory)

- This queue contains all the processes ready to be fetched from the memory, for execution.
- When the process is initiated, it joins the ready queue to wait for the CPU to be free.
- The operating system assigns a process to the executing processor from this queue based on the scheduling algorithm it implements.
  - Contains processes (mainly their PCBs) waiting for the CPU to execute various processes it contains.
  - They are controlled using a scheduling algorithm like FCFS, SJF, or Priority Scheduling.
  - Short Term Scheduler picks a process from Ready Queue and moves the selected process to running state.

Block or Device Queues (In Main Memory)

- The processes which are blocked due to unavailability of an I/O device are added to this queue.
- Every device has its own block queue.

Flow of Movement of processes in the above different Queues



Belady’s Anomaly

- Belady's Anomaly can be **seen in the concept of Page replacement**.
- Any process is divided into pages and put in the frames of the main memory.
- But the number of frames is fixed, i.e.- they are not infinite. After some time, when new pages are needed to load, the old pages are removed or swapped from the memory.
- When a process is initialized, it requests the allocation of frames into the memory.
- There will be two situations:
    - First, if there is free space in frames for pages to be loaded
    - Second if there is no free space in frames, then the frame that is for the longest time is replaced by new frame.
- When the number of frames is manipulated, there comes a situation when number of faults increases as we increase the number of frames.
- Normally, as more page frames are available, the operating system has more flexibility to keep the necessary pages in memory, which should reduce the number of page faults. However, in the case of Belady’s Anomaly, this intuition fails, and we observe an unexpected increase in page faults with more available frames.
- **Page fault**- When there is no required page present in RAM (secondary memory) on calling from the CPU, this is known as a "Page Fault".

- Till now we know that whenever we execute a program, then a process is created and would be terminated after the completion of the execution.
- What if we need to create a process within the program and may be wanted to schedule a different task for it.

Can this be achieved?

- Till now we know that whenever we execute a program, then a process is created and would be terminated after the completion of the execution.
- What if we need to create a process within the program and may be wanted to schedule a different task for it.

Can this be achieved?

- Yes, obviously through process creation.
- Of course, after the job is done it would get terminated automatically or you can terminate it as needed.

Process creation using fork

- Process creation is achieved through the **fork() system call**.
- The newly created process is called the **child process** and the process that initiated it (or the process when execution is started) is called the **parent process**.
- **After the fork() system call, now we have two processes - parent and child processes.**

How to differentiate them? - it is through their return values.

- After creation of the child process, let us see the fork() system call details.

```
#include <sys/types.h>
#include <unistd.h>          →          Creates the child process
pid_t fork(void);
```

- After this call, there are two processes, **the existing one is called the parent process and the newly created one is called the child process**.
- The fork() system call returns either of the three values −
  - ➔ Negative value to indicate an error, i.e., unsuccessful in creating the child process.
  - ➔ Returns a zero for child process.
  - ➔ Returns a positive value for the parent process. This value is the process ID of the newly created child process.

Process creation using fork

Simple program

File name: basicfork.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
      fork();
      printf("Called fork() system call\n");
      return 0;
    }
```

Execution/Output

```
Called fork() system call
Called fork() system call
```

Note − Usually after fork() call, the child process and the parent process would perform different tasks. If the same task needs to be run, then for each fork() call it would run 2 power n times, where **n** is the number of times fork() is invoked.

In the above case, fork() is called once, hence the output is printed twice (2 power 1).

Process creation using fork

File name: pids_after_fork.c

```
#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main() {

        pid_t pid, mypid, myppid;

        pid = getpid();

        printf("Before fork: Process id is %d\n", pid);

        pid = fork();

       if (pid < 0) {

                perror("fork() failure\n");

                return 1;

            }
// Child process

  if (pid == 0) {

              printf("This is child process\n");

              mypid = getpid();

              myppid = getppid();

              printf("Process id is %d and PPID is %d\n", mypid, myppid);

  } else {
// Parent process

    sleep(2);

    printf("This is parent process\n");

    mypid = getpid();

    myppid = getppid();

    printf("Process id is %d and PPID is %d\n", mypid, myppid);

    printf("Newly created process id or child pid is %d\n", pid);

  }

  return 0;

}
```

Compilation and Execution Steps

```
Before fork: Process id is 166629
This is child process
Process id is 166630 and PPID is 166629
Before fork: Process id is 166629
This is parent process
Process id is 166629 and PPID is 166628
Newly created process id or child pid is 166630
```

How to the process be terminated?

- A process can terminate in either of the two ways −

  Abnormally, occurs on delivery of certain signals, say terminate signal.

  Normally, using _exit() system call (or _Exit() system call) or exit() library function.
- The difference between _exit() and exit() is mainly the cleanup activity.
- The **exit()** does some cleanup before returning the control back to the kernel, while the **_exit()** (or _Exit()) would return the control back to the kernel immediately.

- What happens if the parent process finishes its task early than the child process and then quits or exits?
- Now who would be the parent of the child process?

init process

- Let us consider an example program, where the parent process does not wait for the child process, which **results into init process becoming the new parent for the child process**.

File name: parentprocess_nowait.c

```
#include<stdio.h>
int main() {
        int pid;
        pid = fork();

        // Child process

        if (pid == 0) {
                system("ps -ef");
                sleep(10);
                system("ps -ef");
                }
        else {
            sleep(3);
           }
            return 0;
    }
```

- The parent of the child process is init process, which is the very first process initiating all the tasks.
- To monitor the child process execution state, to check whether the child process is running or stopped or to check the execution status, etc. the wait() system calls and its variants is used.

wait() , waitpid() & waitid()

- The variants of system calls to monitor the child process/es
  - wait()
  - waitpid()
  - waitid()

- **wait()** system call would wait for one of the children to terminate and return its termination status in the buffer as explained below.

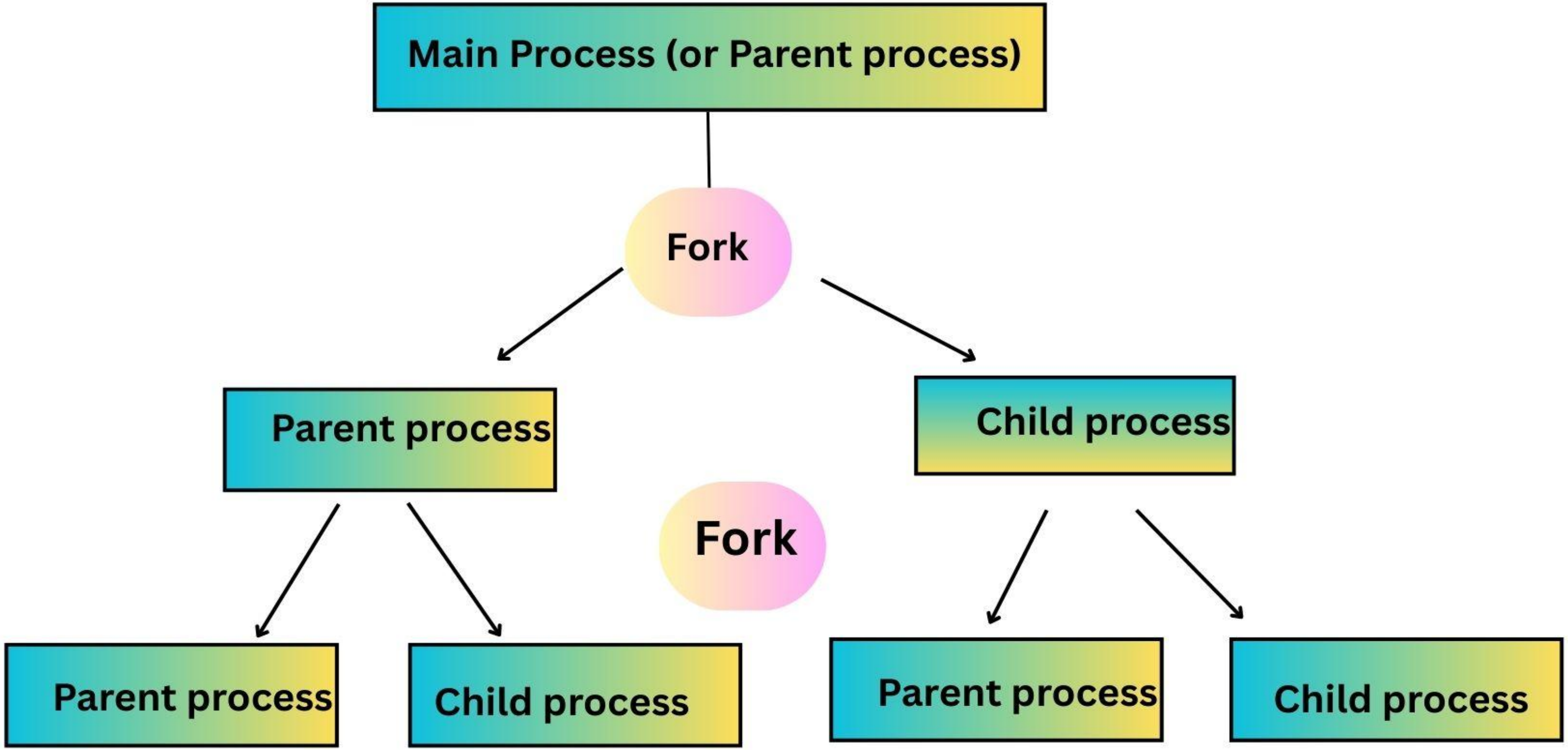
```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

- This call returns the process ID of the terminated child on success and -1 on failure.
- The wait() system call suspends the execution of the current process and waits indefinitely until one of its children terminates.
- The termination status from the child is available in status.

/* File name: parentprocess_waits.c */

```
#include<stdio.h>
int main() {
            int pid;
            int status;
            pid = fork();

   // Child process
            if (pid == 0) {
                        system("ps -ef");
                        sleep(10);
                        system("ps -ef");
                        return 3;
            //exit status is 3 from child process
            } else {
                  sleep(3);
                  wait(&status);
                  printf("In parent process: exit status
from child is decimal %d, hexa %0x\n", status, status);
                  }
                  return 0;
            }
```


Main Process (or Parent process)
Fork
Parent process
Child process
Fork
Parent process
Child process
Parent process
Child process

wait() , waitpid() & waitid()

- wait() system call has limitation such as it can only wait until the exit of the next child.

When to use waitpid()?

- If we need to wait for a specific child it is not possible using wait(), however, it is possible using waitpid() system call.
- The **waitpid() system call would wait for specified children to terminate and return its termination status in the buffer** as explained below.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```

- The above call returns the process ID of the terminated child on success and -1 on failure.
- The **waitpid() system call suspends the execution of the current process and waits indefinitely until the specified children (as per pid value) terminates.**
- The termination status from the child is available in the status.

wait() , waitpid() & waitid()

- The value of pid can be either of the following −

  <-1 − Wait for any child process whose process group ID is equal to the absolute value of pid.

  -1 − Wait for any child process, which equals to that of wait() system call.

  0 − Wait for any child process whose process group ID is equal to that of the calling process.

  >0 − Wait for any child process whose process ID is equal to the value of pid

- By default, **waitpid() system call waits only for the terminated children but this default behavior can be modified using the options argument**.

  let us check for waitid() system call. This system call waits for the child process to change state.

  ```
  #include <sys/wait.h>
  int waitpid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
  ```

- The above system call waits for the child process to change the state and this call suspends the current/calling process until any of its child process changes its state.
- The argument infop is to record the current state of the child.
- This call returns immediately, if the process has already changed its state.
- The value of idtype can be either of the following −

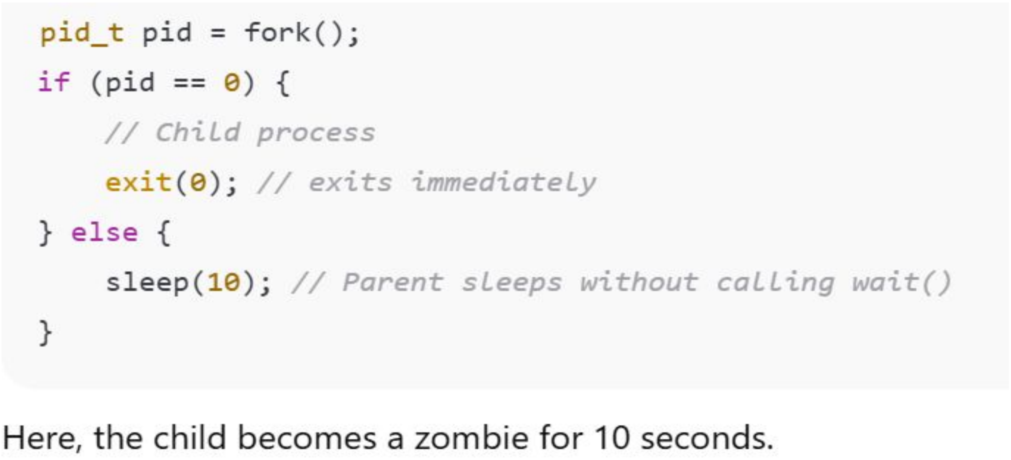
  P_PID − Wait for any child process whose process ID is equal to that of id.

Difference between fork() and exec()

SNO	fork()	exec()
1.	It is a system call in the C programming language	It is a system call of operating system
2.	It is used to create a new process	exec() runs an executable file
3.	Its return value is an integer type	It does not creates new process
4.	It does not takes any parameters.	Here the Process identifier does not changes
5.	It can return three types of integer values	In exec() the machine code, data, heap, and stack of the process are replaced by the new program.

Zombie processes

- A **Zombie Process** is a process that has **completed its execution**, but it's **entry still remains in the process table** to allow the parent process to read its exit status.
- It is created when a **child process finishes**, but the **parent has not yet called wait()**.
- It still occupies an entry in the **process table**.
- It's **not actually running** — just waiting for the parent to collect the exit info.
- If too many zombies accumulate, they can **exhaust system resources**.



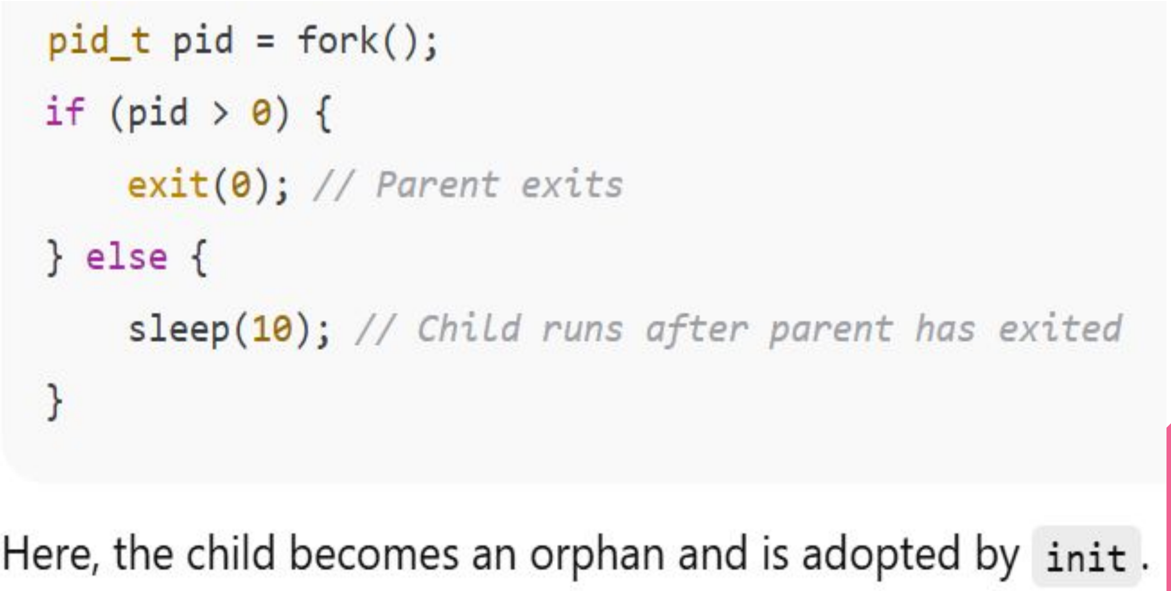
```
pid_t pid = fork();
if (pid == 0) {
    // Child process
    exit(0); // exits immediately
} else {
    sleep(10); // Parent sleeps without calling wait()
}
```

Here, the child becomes a zombie for 10 seconds.

Orphan Process

An **Orphan Process** is a **child process whose parent has terminated** before the child finishes execution.

- The **init process (PID 1)** adopts the orphan process.
- The orphan continues to run normally.
- The OS ensures the child is not left unmanaged.



```
pid_t pid = fork();
if (pid > 0) {
    exit(0); // Parent exits
} else {
    sleep(10); // Child runs after parent has exited
}
```

Here, the child becomes an orphan and is adopted by `init`.

Concepts of Operating Systems

- Vineela

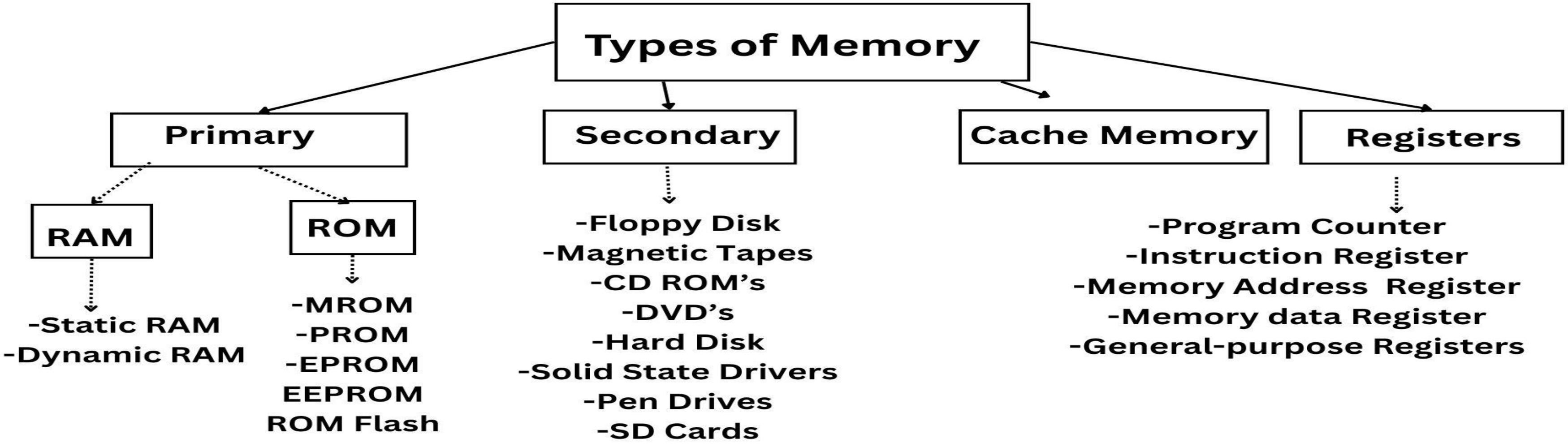
Sessions 6 & 7

Memory Management

- What are different types of memories; What is the need of Memory management
- Continuous and Dynamic allocation
- First Fit, Best Fit, worst Fit
- Compaction
- Internal and external fragmentation
- Segmentation – What is segmentation; Hardware requirement for segmentation; segmentation table and its interpretation
- Paging – What is paging; hardware required for paging; paging table; Translation look aside buffer
- Concept of dirty bit
- Shared pages and reentrant code
- Throttling
- IO management

Memory Management

- A physical device that stores data or information temporarily or permanently in it is called **Memory** where data is stored and processed.
- The **task of subdividing the memory among different processes** is called Memory Management.
- The main aim is **to achieve efficient utilization of memory**.



RAM (Random Access Memory)

- RAM typically contains 8-bits wherein each memory location, typically are stored.
- It can be possible to read and write to and from a RAM location respectively
- The drawback of RAM is that it is volatile.

From the memory, data can be **accessed in two different ways** – Sequential Access and Random Access

- **Sequential Access** − In this, it is **mandatory to access information strictly in order.**

  Ex - If there are 4000 memory locations, it have to be accessed in the order of 1, 2, 3,…,4000.

  Thus, it takes minimum time to access information from location 0 and at most time to access information from location 4000.

  Ex - Magnetic tape is an example that employs sequential access.

- **Random Access** − In a random access technique, it can be possible to access a memory location in any order.

  Ex - One can read from the 4000 locations in the order of 1500, 1210, 3060, 1640, 1352, and so on.

Read Only Memory (ROM)

- It is non-volatile in nature.
- Only reading operation is possible from a ROM location. Thus, in a computer, ROM is used for storing information which is not lost when power is switched off.

The different versions of ROM are:

1- Mask-Programmed ROM

- It derives this name because the information is written to this type of ROM at the time of manufacture by applying a suitable mask
- Once written, the information cannot be changed, even by the manufacturer
- It is **used in equipment produced in large quantities**

2 - Programmable Read Only Memory (PROM)

- The user writes information to this type of ROM using a PROM programmer.
- Once written, the information cannot be changed.
- Like Mask-Programmed ROM, the information is permanent.
- It is more expensive than mask ROM but **allows purchasing in smaller quantities**.

Read Only Memory (ROM)

3- Erasable Programmable Read Only Memory (EPROM)

- As its content is **erasable and rewritable**, the user can modify it multiple times.
- Data is **erased using strong ultraviolet (UV) light** on the quartz window of the EPROM chip, which removes all content.
- Users can purchase a single piece of EPROM and rewrite its content multiple times.

4 - Electrically Erasable Programmable Read Only Memory (EEPROM)

- Unlike EPROM, **EEPROM data is erased using electrical signals** rather than UV light.
- EEPROM allows selective data erasure and is more expensive than other ROM types.
- It is gaining popularity due to its flexibility.

Secondary Memory

- Computer secondary memory stores data and programs permanently, even when the computer is off.
- The secondary memory is **also known as external memory or auxiliary memory**.

Classification of Secondary Memory

- Magnetic Storage
-Hard Disk Drive (HDD)
-Floppy Disk (Obsolete)
-Magnetic Tape

- Optical Storage
-CD (Compact Disc)
-CD-ROM (Read-Only Memory)
-CD-R (Recordable)
-CD-RW (Rewritable)

-DVD (Digital Versatile Disc)
-DVD-ROM
-DVD-R/DVD+R
-DVD-RW/DVD+RW

-Blu-ray Disc (BD)

-Flash Storage (Solid-State Storage)
-Solid-State Drive (SSD)
-USB Flash Drive (Pen Drive)
-Memory Cards (SD Card, microSD, etc.)

-Cloud Storage
-Google Drive
-Dropbox
-OneDrive
-Amazon S3

-**Hybrid Storage**
-Hybrid Drive (HDD + SSD Combination)
-SSHD (Solid-State Hybrid Drive)

Cache Memory

- Cache memory is smaller and faster than RAM. It is placed closer to the CPU than the RAM.

Register Memory

- Register memory, which is also called processor registers or "registers," is the smallest and fastest type of computer memory that is directly integrated into the CPU.

Important functions of Registers:

- **Instruction Execution** − Registers hold the instructions that the CPU is currently running. This includes the operation code (opcode) and associated operands with it.

- **Data Storage** − Registers store CPU-processed data. This can provide memory addresses, intermediate values during arithmetic or logical operations, and other data needed by the instructions being executed.

- **Addressing** − Memory addresses are used to store or retrieve data from memory locations in RAM or other parts of the computer's memory hierarchy.

Types of Registers

- Program Counter (PC) − Stores the memory address of the next instruction to be fetched and executed.
- Instruction Register (IR) − Holds the current instruction being executed by the CPU.
- Memory Address Register (MAR) − Stores the memory address of data being read from or written to memory.
- Memory Data Register (MDR) − Contains the actual data being read from or written to memory.
- General-Purpose Registers (GPRs) − Used for general data storage and manipulation during program execution.

Why to manage Memory?

- To improve both utilization of the CPU and the speed of the computer’s response to its users

What is Memory Allocation?

- When a program or process is to be executed, it needs some space in the memory
- For this reason, some part of the memory has to be allotted to a process according to its requirements
- This process is called Memory Allocation

Continuous and Dynamic Allocation

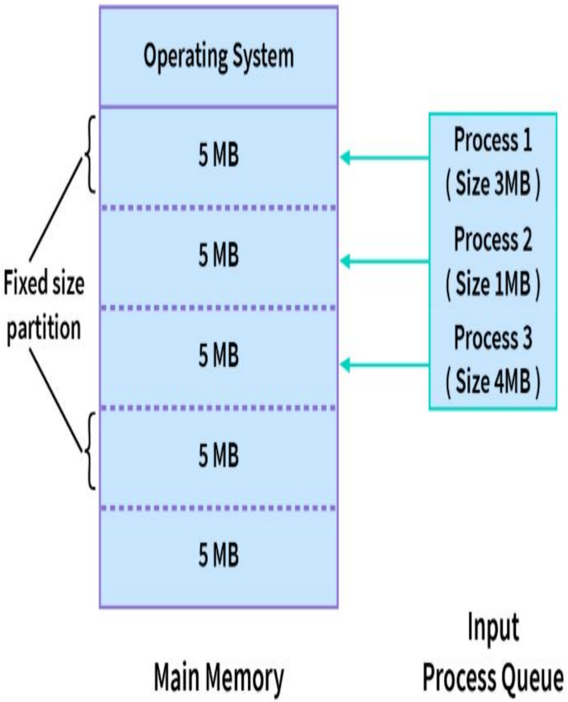
- Contiguous Memory Allocation is a type of memory allocation technique where processes are allotted a continuous block of space in memory.
- This block can be of fixed size or can be of variable size depending on the requirements of the process

What is Contiguous Memory Allocation in OS?

- As the name implies, we allocate contiguous blocks of memory to each process from the totally empty space based on its size.
- This allocation can be done in two ways:
  1. Fixed-size Partition Scheme
  2. Variable-size Partition Scheme

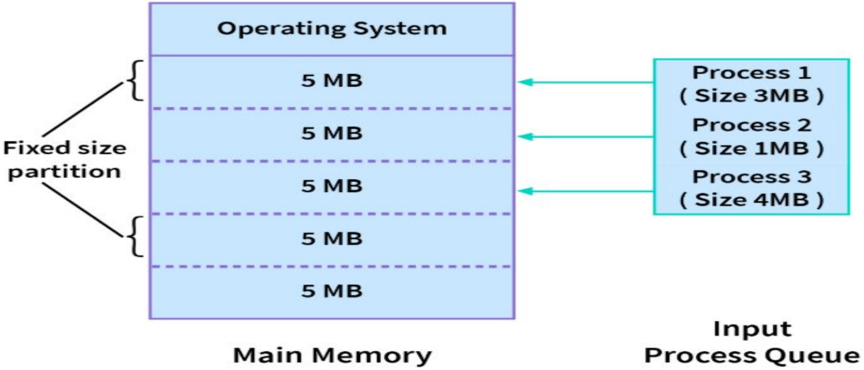
Fixed-size Partition Scheme (Static Partitioning)

- In the diagram above, we have 3 processes in the input queue that have to be allotted space in the memory.
- As we are following the fixed-size partition technique, the memory has fixed-sized blocks.
- First process, which is of size 3MB is also allotted a 5MB block
- Second process, which is of size 1MB, is also allotted a 5MB block, and the 4MB process is also allotted a 5MB block.
- So, the process size doesn't matter. Each is allotted the same fixed-size memory block.
- It is clear that in this scheme, the number of continuous blocks into which the memory will be divided will be decided by the amount of space each block covers, and this, in turn, will dictate how many processes can stay in the main memory at once.



Variable-size Partition Scheme (Dynamic Partitioning)

- In this no fixed blocks or partitions are made in the memory.
- Instead, each process is allotted a variable-sized block depending upon its requirements.
- That means, that whenever a new process wants some space in the memory, if available, this amount of space is allotted to it.
- Hence, the size of each block depends on the size and requirements of the process which occupies it.
- In the diagram below, there are no fixed-size partitions.
- Instead, the first process needs 3MB memory space and hence is allotted that much only. Similarly, the other 3 processes are allotted only that much space that is required by them.
- As the blocks are variable-sized, which is decided as processes arrive.



- So far, we've seen the two types of schemes for contiguous memory allocation.
- But what happens when a new process comes in and has to be allotted a space in the main memory?
- How is it decided which block or segment it will get?

Processes that have been assigned continuous blocks of memory will fill the main memory at any given time. However, when a process completes, it leaves behind **an empty block known as a hole**. This space could also be used for a new process. Hence, the main memory consists of processes and holes, and any one of these holes can be allotted to a new incoming process. We have **three strategies to allot a hole to an incoming process:**

- ➔ **First-Fit**
- ➔ **Best-Fit**
- ➔ **Worst-Fit**

Strategies Used for Contiguous Memory Allocation Input Queues

First-Fit : Allot the process to the first hole, which is big enough.

- This is a **very basic strategy** in which we start from the beginning and **allot the first hole,** which is **big enough as per the requirements of the proces**s.
- The first-fit strategy can also be implemented in a way ,where we can **start our search for the first-fit hole from the place we left off last time**.

Best-Fit : Allot the smallest hole that satisfies the requirements of the process.

- This is a **greedy strategy** that aims to **reduce any memory wasted because of internal fragmentation** in the case of static partitioning, and hence we allot that hole to the process, which is the **smallest hole that fits the requirements of the process.**
- Hence, we need to first **sort the holes according to their sizes and pick the best fit** for the process without wasting memory.

Worst-Fit : Allot the largest size hole among all to the incoming process.

- This strategy is the **opposite of the Best-Fit strategy.** We **sort the holes according to their sizes and choose the largest hole to be allotted to the incoming process**.
- The idea behind this allocation is that as the process is allotted a large hole, it will have a lot of space left behind as internal fragmentation.
- Hence, this **will create a hole that will be large enough to accommodate a few other processes.**

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.

Why to go for Compaction?

- While allocating memory to process, the operating system often faces a problem when there’s a sufficient amount of free space within the memory to satisfy the memory demand of a process.
- however the process’s memory request can’t be fulfilled because the free memory available is in a non-contiguous manner, this problem is referred to as **external fragmentation.**
- To solve such kinds of problems compaction technique is used.

How it does?

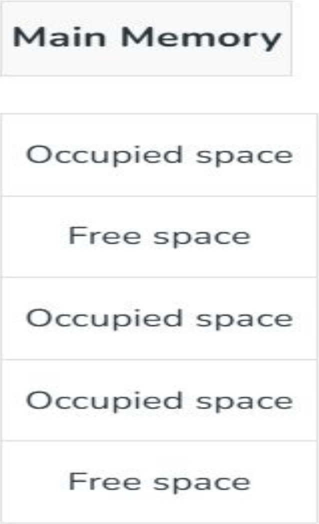
- By moving all the processes towards one end of the memory and all the available free space towards the other end of the memory so that it becomes contiguous.

When to use Compaction?

- 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

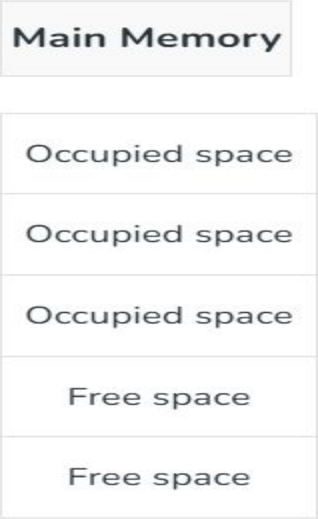
- Before compaction, the main memory has some free space between occupied space. This condition is known as **external fragmentation**.
- Due to less free space between occupied spaces, large processes cannot be loaded into them.



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

Advantages and Disadvantages of Compaction

Advantages

- Reduces external fragmentation.
- Make memory usage efficient.
- Since memory becomes contiguous more processes can be loaded to memory, thereby increasing scalability of OS.
- Fragmentation of file system can be temporarily removed by compaction.
- Improves memory utilization as their is less gap between memory blocks.

Disadvantages

- System efficiency reduces and latency is increased.
- A huge amount of time is wasted in performing compaction.
- CPU sits idle for a long time.
- Not always easy to perform compaction.
- It may cause deadlocks since it disturbs the memory allocation process.

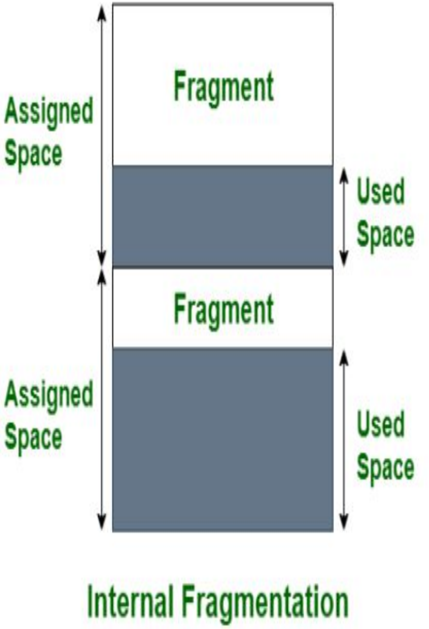
Internal and External Fragmentation

- Memory fragmentation is a prevalent problem in operating systems that can **result in the inefficient use of memory resources**.

- There are two types of fragmentation: internal and external

What is Internal Fragmentation?

- Whenever a method is requested for the memory, the mounted-sized block is allotted to the method.
- In the case, where the memory allocated to the method is somewhat larger than the memory requested, then **the difference between allotted and requested memory is called Internal Fragmentation**.



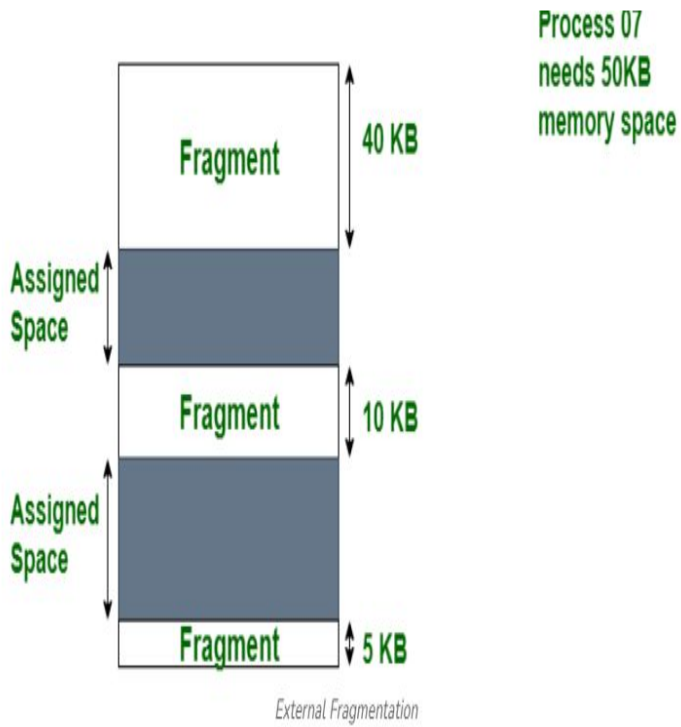
Internal Fragmentation

How to solve 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.

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.
- In the 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.



External Fragmentation

Internal fragmentation	External fragmentation
This happens when the method or process is smaller than the memory	This happens when the method or process is removed
The solution is the best-fit block	The solution is compaction and paging
This occurs when memory is divided into fixed-sized partitions.	This occurs when memory is divided into variable size partitions based on the size of processes.
The difference between memory allocated and required space or memory is called Internal fragmentation	The unused spaces formed between non-contiguous memory fragments are too small to serve a new process, which is called External fragmentation
It occurs with paging and fixed partitioning	It occurs with segmentation and dynamic partitioning
It occurs on the allocation of a process to a partition greater than the process's requirement. The leftover space causes degradation system performance.	It occurs on the allocation of a process to a partition greater which is exactly the same memory space as it is required.
It occurs in worst fit memory allocation method	It occurs in best fit and first fit memory allocation method

Segmentation

- The chunks that a program is divided into which are not necessarily all of the exact sizes are called segments
- Each segment has a name and a length
- It is a memory - management scheme that supports the programmer view of memory
- A logical address space is a collection of segments which varies in length

The address generated by the CPU is divided into:

- **Segment number (s):** Number of bits required to represent the segment.
- **Segment offset (d):** Number of bits required to represent the position of data within a segment.

Ex - Programmer (stack, math library, the main program)when writing a simple Sqrt() function is not concerned with

- whether the stack is stored before or after Sqrt()

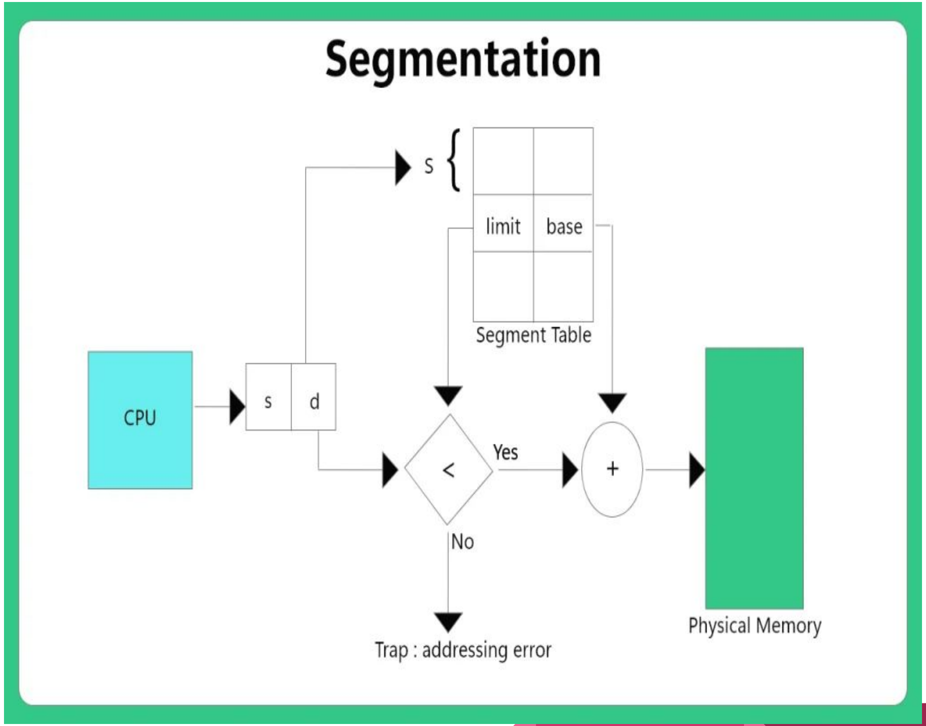
- not cares what addresses in memory these elements occupy.

Compiler create separate segments for the following:

1) The Code
2) Global Variables
3) The heap, from which memory is allocated
4) The stacks used by each thread
5) The standard C library

Segmentation Table and its interpretation

- Although the programmer can now refer to objects in the program by a two-dimensional address, the actual physical memory is still, a one-dimensional sequence of bytes.
- Thus, we define an implementation to map two-dimensional programmer-defined addresses into one-dimensional physical address.
- This mapping is effected by a segment table.
- Each entry in segment table has a segment base (contains starting physical address)and segment limit(specifies the length of the segment).
- A logical address consists of two parts: a segment number s and an offset in to that segment , d.
- segment number s - index to the segment table
- Offset d - logical address (between 0 and base limit) - if its not, we trap to OS.
- When an offset is legal, it is added to segment base to produce address in physical memory of the desired byte.



Paging

- Virtual memory is often implemented using **paging**, which breaks memory into fixed-size blocks called **pages** (in the virtual address space) and **page frames** (in the physical memory)
- The OS uses a **page table** to map virtual addresses to physical addresses, allowing data to be stored in non-contiguous areas of physical memory
- When a process accesses a page that is not currently in physical memory, a **page fault** occurs, and the OS loads the required page from secondary storage (usually a hard drive or SSD) into RAM.

Demand paging

- It is a process that **keeps pages of a process that are infrequently used in secondary memory, and pulls them only when required to satisfy the demand**.
- As a result, when a context switch happens, the OS begins executing the new program after loading the first page and only retrieves the application's referenced pages.
- If the software addresses a page that is not available in the main memory because it was swapped, the processor deems it as an invalid memory reference.
- **Ex** - Wishlisting clothes in e-commerce site and decide to buy them only when needed.

How Demand Paging Works:

Step 1 - Initial Setup:

- The program is initially stored on disk.
- Only essential parts (like the initial instructions) are loaded into RAM.

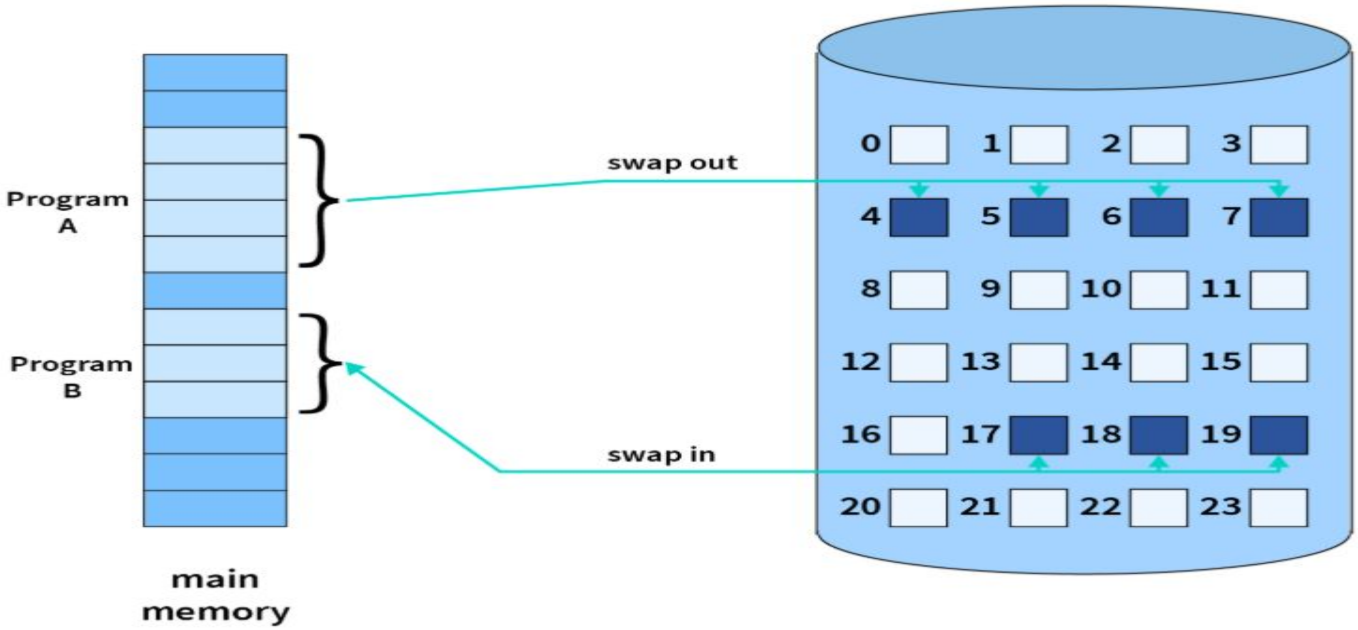
Page Table:

- Each process has a **page table** that keeps track of pages in memory.
- Pages not currently in RAM are marked as **invalid**.

Step 2 - Page Fault:

- If the CPU tries to access a page not in RAM, a **page fault** occurs.
- The OS then: → Pauses the program
  → Loads the required page from disk into memory.
  → Updates the page table Resumes execution.

Example of Demand Paging



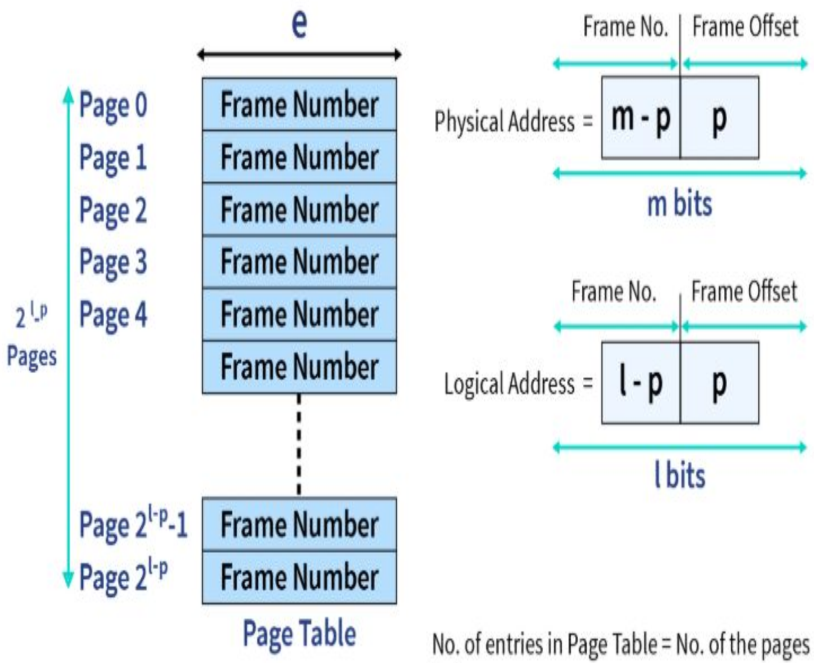
- In the above image, when Program A finishes executing, it swaps out the memory that was in use.
- Program B then swaps in the memory that was required by it to fulfill the timely demand.

Page Frames

- A page frame is used to structure physical memory.
- A page frame size is a power of two bytes and varies between platforms.
- The CPU accesses the processes through their logical addresses while the main memory recognizes the physical address only.
- The Memory Management Unit eases this process by converting the page number (logical address) to the frame number (physical address). The offset is the same in both.

Page Table

- The page table maps the page number to its frame number.
- A page table is a logical data structure that is used by a virtual memory to record the mapping between virtual and physical addresses.
- The programs performed by the accessing process use virtual addresses, whereas the hardware, notably the random-access memory (RAM) subsystem, uses physical addresses.
- The page table is an important part of virtual address translation, which is required to access data in memory.



No. of entries in Page Table = No. of the pages in the process

Page Table Size = 2^{l-p} X e bytes

e = m-p (Frame Size) bits

Paging

- The **process of retrieving processes in the form of pages from the secondary storage into the main memory** is known as paging.
- Paging is a memory management scheme that eliminates the need for a contiguous allocation of physical memory
- The basic purpose of paging is to separate each procedure into pages.
- Paging is a function of memory management where a computer will store and retrieve data from a device's secondary storage to the primary storage.
- The primary concept behind paging is to break each process into individual pages resulting the separation of the primary memory into frames.

Mechanism of Paging

- One page of the process must be saved in one of the given memory frames. These pages can be stored in various memory locations, but finding contiguous frames/holes is always the main goal. Process pages are usually only brought into the main memory when they are needed; else, they are stored in the secondary storage.
- The frame sizes may vary depending on the OS. Each frame must be of the same size. Since the pages present in paging are mapped on to the frames, the page size should be similar to the frame size.

Dirty bit

- A dirty bit, also known as a modified bit or write bit, is a flag that is used in computer systems **to indicate whether a particular memory address or disk block has been modified since it was last written to.**
- Each page or frame has a modify bit associated with it
- Modify bit is set whenever page is written into - This indicates page has been modified
- When page selected for replacement, modify bit is examined.
- **If the page is dirty** (meaning it has been modified in memory), the OS will write the page back to the disk (or swap space) to ensure that the changes are saved.
- **If the page is not dirty** (meaning it has not been modified), the OS can safely discard the page without writing it back to the disk because the contents on disk are still valid.
- In order to reduce the page fault service time, a special bit called the dirty bit can be associated with each page.

How does the dirty bit work?

- When a process modifies a memory address or writes data to a disk block, the dirty bit for that address or block indicates it has been changed.
- This allows the system to keep track of which portions of memory or disk need to be saved or written back to secondary storage when resources become scarce or when a shutdown occurs.

Why is the dirty bit important in caching?

- Caching is a technique used to improve performance by storing frequently accessed data closer to the processor or in a faster storage medium.
- When data is read from the cache, it is typically marked as clean because it matches the corresponding data in the main memory or disk.
- However, when the cached data is modified, the dirty bit is set to indicate that the data in the cache has been changed and needs to be written back to the main memory or disk at some point.
- This ensures that the changes made to the data are not lost and are propagated to the appropriate location.

Shared Pages

- **Shared pages** refer to **memory pages** that are **shared between multiple processes**.
- This is done to **reduce memory usage** and **improve efficiency**, especially when processes use the same code or data.
- Paging system also has a problem with sharing as one of its design issues.
- On a large computer system, that is capable of running multiple programs at once, it is common for multiple users to be occupied with the same program at the same time.
- Now, simply **share the pages in order to prevent having two distinct copies of the same page stored in your memory** at the same time.
- Pages that can only be read are generally shareable, such as the text of a program, however, data pages are not.

What results to Shared Pages?

- When two or more than two processes (referred to collectively as multiple processes) share some code, it can result in a problem with shared pages

Shared Pages

How It Works

1. The OS loads a **shared library** (e.g., libc.so) into memory.
2. Multiple processes **map** the same physical pages into their **virtual address space**.
3. **Page tables** of each process point to the **same physical memory page**.
4. If a process tries to **write** to a read-only shared page:
    - A **page fault** occurs
    - The OS makes a **private copy** of the page (COW)

Example

Suppose two processes, A and B, both use the printf() function:

- printf() comes from libc, loaded once into memory
- A and B's page tables point to the same physical page for printf()
- They **share** the code → memory saved

Shared Pages

Example of Shared Pages

- Let's say that process X and process Y are both running the editor and sharing its pages. What would happen?
- If the scheduler makes the decision to remove process X from memory, evicting all of its pages and filling the empty page frames with the other program will cause process Y to generate a large number of page faults in order to restore them.
- If the scheduler makes the decision to remove process Y from memory, evicting all of its pages and filling the empty page frames with the other program.
- In a similar fashion, whenever the process X comes to an end, it is essential to be able to discover that the pages are still in use. This ensures that the disc space associated with those pages is not accidentally freed.

Reentrant code (Pure Code)

- Reentrant code is code that can be safely interrupted in the middle of execution and called again ("re-entered") before the previous executions are finished **without affecting the outcome**.
- Reentrant functions or routines are designed in such a way that they maintain their integrity and can be called concurrently without causing problems like data corruption or unexpected behavior.

Key Characteristics of Reentrant Code:

1. **No static or global state**: Reentrant code does not rely on shared or static variables, as they can be overwritten when re-entered. Any state information should be local to the function.

2. **No side effects**: A reentrant function should not modify any external state or depend on it.

3. **Atomic operations**: The function must execute in a way that ensures it can be interrupted and resumed without interfering with other invocations.

4. **Thread safety**: Reentrant code is often thread-safe, though thread safety and reentrancy are not exactly the same. Thread safety ensures that multiple threads can use the code concurrently without conflict, while reentrancy ensures that the same function can be re-entered at any point.

Finally Reentrant code is Independent, Interrupt-safe , Shareable and Essential for system-level programming

Example of Reentrant code

Non-Reentrant Code (has global variable):

```
int counter = 0;
void increment() {
                    counter++;
                    }
```

This is **not reentrant** because:

- Uses global variable counter
- If interrupted, another execution may modify counter, leading to incorrect results

Reentrant Code (uses local variable):

```
int increment(int x) {
                    return x + 1;
                    }
```

This is **reentrant** because:

- No shared/global data
- Can be safely called by multiple threads or during interrupts

Throttling

- It is a technique **used to manage system resources,** such as CPU time, memory and network bandwidth, to ensure that processes and applications do not overwhelm the system.
- By controlling the rate at which certain tasks or processes are executed, the operating system can maintain performance, fairness, and responsiveness, especially in multi-tasking environments, where many processes are running concurrently.
- **Types of Throttling in an OS:**

  CPU Throttling (Process Scheduling):

  ➔ This involves limiting the CPU resources allocated to a process to prevent one process from consuming all the available CPU time.

  ➔ This is often done by **process scheduling algorithms** and is especially important in systems with multiple processes or threads.

  ➔ **Example**: In an OS with multiple processes, the kernel uses scheduling algorithms like Round-Robin or Priority Scheduling to allocate CPU time to each process. If a process consumes too much CPU time, the OS might throttle it by reducing its priority or placing it in a waiting queue.

Throttling

I/O Throttling:

- ➔ Throttling can be applied to disk and network I/O operations to prevent a single process from monopolizing disk access or network bandwidth.

Example: The OS might limit the read/write rate for a specific process or user to ensure fair disk access and prevent the system from slowing down due to excessive disk activity.

Network Throttling:

- ➔ Network throttling involves limiting the data transfer rate over a network interface to ensure that no single process or user consumes all the bandwidth.

Example: A file download process might be throttled to prevent it from using all available network bandwidth, which could slow down other applications, such as web browsing or online communication.

Memory Throttling (Swap Throttling):

- ➔ In systems with limited physical memory (RAM), when the system runs out of available memory, the operating system may "throttle" processes by swapping memory pages in and out of disk storage (swap space) to free up space in RAM.

Example: If the system is running out of memory, the OS might use techniques like **Out-of-Memory (OOM) Killer** in Linux to throttle or terminate processes that are consuming too much memory.

IO Management

- **I/O Management** refers to the processes and mechanisms that an operating system (OS) uses to manage input and output (I/O) operations, such as communication between the system and external devices like keyboards, mice, displays, disk drives, printers, and network interfaces.
- I/O management ensures that the system can handle these operations efficiently, providing fair access to I/O devices, managing data transfer and maintaining the stability and performance of the system.

Key Components of I/O Management

I/O Devices:

- I/O devices can be categorized into **input devices** (keyboard, mouse) and **output devices** (monitor, printer).
- **Storage devices** like hard drives, SSDs, and network devices are also integral parts of I/O management.

Device Drivers:

- A **device driver** is a software component that acts as a bridge between the OS and hardware devices.
- It translates the generic I/O commands from the OS into device-specific commands.
- Drivers abstract the hardware complexities and allow the OS to communicate with various devices uniformly.

IO Management

I/O Control:

- I/O control refers to the software and mechanisms that govern how input and output operations are initiated, managed, and completed. This includes interactions between the OS, device drivers, and hardware.
- I/O control ensures that the OS can access and control multiple I/O devices simultaneously without interference

I/O Scheduling:

- I/O scheduling is the process of determining the order in which I/O requests are serviced by the OS. Efficient I/O scheduling is crucial for optimizing system performance and reducing latency.
- Algorithms like **First-Come, First-Served (FCFS)**, **Shortest Seek Time First (SSTF)**, and **Look/Seek Algorithms** are used to manage I/O requests to storage devices

Buffering:

- Buffering is used to store data temporarily while it is being transferred between devices or between devices and memory.
- Buffers help reduce the time difference between fast processors and slower I/O devices.
- **Double buffering** and **circular buffering** are common techniques to ensure smooth data transfer.

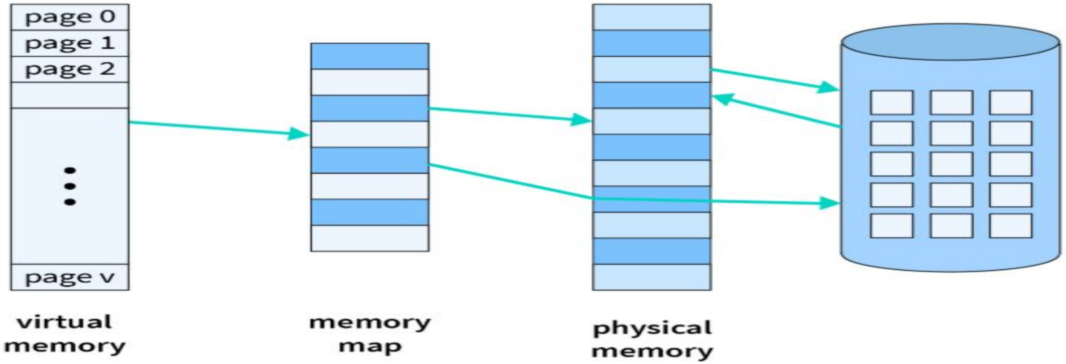
Session 8:

Virtual Memory

- What is virtual memory
- Demand paging
- Page faults
- Page replacement algorithms

What is virtual memory

- It is a part of the secondary storage that gives the user the **illusion that it is a part of the main memory**
- It helps in running multiple applications with low main memory and increases the degree of multiprogramming in systems without exhausting the RAM (Random Access Memory)
- It is commonly **implemented using demand paging**.



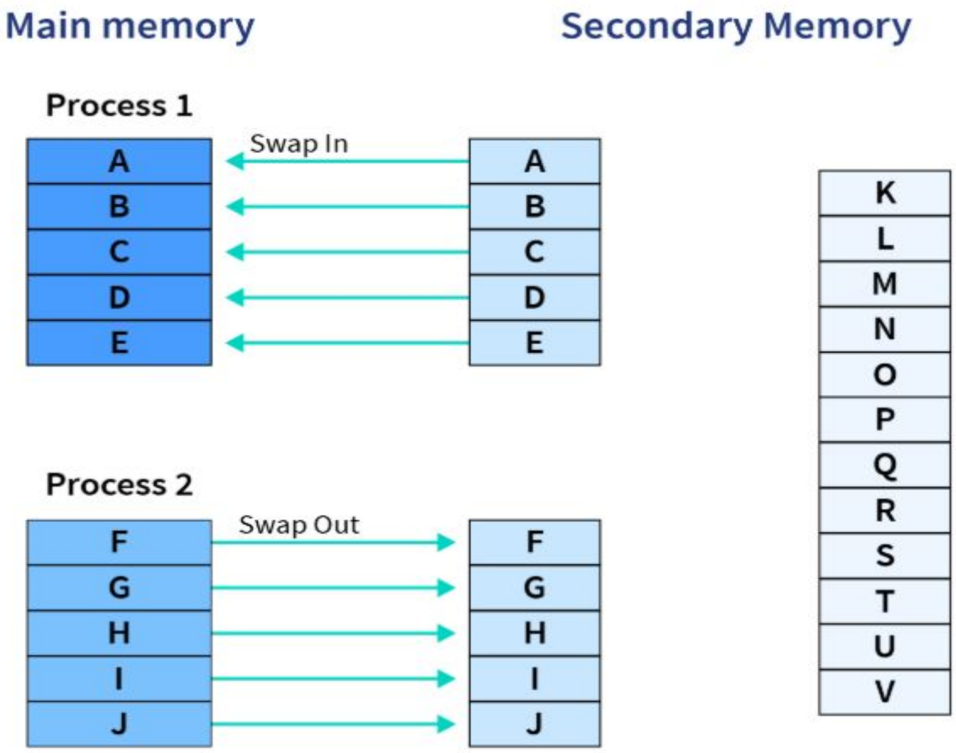
How Virtual Memory Works in a Modern OS?

Consider a system with 8 GB of RAM and a process that needs 16GB of memory:

- The OS allocates **virtual memory** addresses for the process, which it can use as if it has 16GB of contiguous memory.
- The OS creates a **page table** to map these virtual addresses to physical addresses in RAM.
- Since the system only has 8 GB of RAM, only part of the process’s memory will reside in RAM at any given time.
- When the process tries to access memory that isn’t in RAM, the **page table** indicates that the page is not present, and the OS triggers a **page fault**.
- The OS then swaps out another part of memory from RAM (perhaps an unused page from another process) and swaps in the required page from the swap space on disk.
- This process continues as needed, allowing the process to access the memory it needs while not exceeding the physical RAM.

Swap In and Swap Out

- When the primary memory (RAM) is insufficient to store data required by several applications, we use a method known as swap out to transfer certain programs from RAM to the hard drive.
- Similarly, when RAM becomes available, we swap in the applications from hard disk to RAM.
- We may manage many processes inside the same RAM by using swaps. Swapping aids in the creation of virtual memory and is cost-effective.



Key Terms:

- **Page Table**: A data structure used by the OS to map virtual addresses to physical addresses.
- **Page Fault**: A fault that occurs when a process accesses a virtual memory page that is not currently in physical memory.
- **Swap Space**: A portion of the disk used by the OS to store pages that are not currently in physical memory.
- **Thrashing**: A condition where the system spends too much time swapping pages in and out of memory, resulting in poor performance.

Page Fault

- This occurs when a program tries to access a part of memory (**a page**) that is **not currently in RAM**.
- The operating system must step in to retrieve the page from **secondary storage** (like a hard drive or SSD) and load it into **main memory** (RAM).

Types of Page Faults:

1. **Minor Page Fault (Soft Page Fault)**
   - The page is not in RAM but is **in memory cache (e.g., in a different part of memory)**.
   - Fast to resolve
2. **Major Page Fault (Hard Page Fault)**
   - The page is not in RAM or cache and must be loaded **from disk**.
   - Slower and more costly.
3. **Invalid Page Fault**
   - The memory address is invalid or the process doesn't have access.
   - Leads to **segmentation fault** or **process termination**.

Page Fault Handling Process:

- **CPU tries to access a page**.
- **Page table lookup** shows page is not in memory.
- **Page fault interrupt** is triggered.
- OS suspends the process and:

  → Finds a free memory frame (or uses page replacement).

  → Loads the required page from disk.

  → Updates the page table.
- **Process resumes** execution as if nothing happened.

Example of Page Fault:

A program accesses address 0x0040, which is in **page 2**.

- Page 2 is not in RAM → **Page fault** occurs.
- OS fetches page 2 from disk.
- Loads it into an available frame.
- Updates page table → maps page 2 to new frame.
- Process continues.

Page Replacement Algorithms

- Page replacement algorithms are crucial in operating systems for memory management, specifically when dealing with **virtual memory**.
- These algorithms decide which memory pages to swap out when a new page needs to be loaded, but memory is full.

1. FIFO (First-In, First-Out) - Removes the oldest page in memory (the one loaded first).

- **Pros**: Simple to implement.
- **Cons**: May remove frequently used pages → **Belady’s anomaly** (more frames may increase page faults).

2. LRU (Least Recently Used) - Replaces the page that hasn't been used for the longest time.

- **Pros**: Good performance, mimics real-world usage
- **Cons**: Requires tracking usage → more complex and expensive to implement.

3. Optimal (OPT) - Replaces the page that won’t be used for the longest time in the future.

- **Pros**: Best possible performance.
- **Cons**: **Not practical** — future knowledge is required; used mainly for benchmarking.

4. Clock (Second Chance) - Circular buffer with a reference bit for each page. If the bit is 0, replace it; if 1, set it to 0 and move on.

- **Pros**: Efficient approximation of LRU, low overhead.
- **Cons**: Slightly more complex than FIFO.

5. NRU (Not Recently Used) - Uses reference and modify bits; pages are classified into 4 categories, and one from the lowest category is replaced.

- **Pros**: Prioritizes unmodified and unreferenced pages.
- **Cons**: Needs periodic bit resetting.

6. LFU (Least Frequently Used) - Replaces the page with the fewest accesses.

- **Pros**: Good if past frequency predicts future use.
- **Cons**: May hold onto pages that were heavily used in the past but are no longer needed.

7. MFU (Most Frequently Used) - Opposite of LFU; removes the most frequently used pages.

- **Pros**: Based on the idea that pages used often may no longer be needed.
- **Cons**: Less commonly used; generally performs worse.

Concepts of Operating Systems

- Vineela

Session 9:

Deadlock

- Necessary conditions of deadlock
- Deadlock prevention and avoidance
- Semaphore
- Mutex
- Producer consumer problem
- Dead-lock vs Starvation

Deadlock

- Deadlock is a situation in which two or more processes require resources to complete their execution, but those resources are held by another process., due to which the execution of the process is not completed.
- **For example** - suppose there are two friends and both want to play computer games, due to which they fight. One has a remote control, and the other has a CD of games.
- Due to this, neither of the two friends can play, but neither of them is ready to cooperate. This situation is called deadlock.

Necessary conditions of deadlock

- **Mutual Exclusion -** At least one resource must be held in a **non-shareable mode**. Only one process can use the resource at any given time.

  Example: A printer cannot be used by two processes at once.
- **Hold and Wait -** A process is holding at least one resource and is **waiting to acquire additional resources** that are currently being held by other processes.

  Example: Process A holds a file and waits for a printer held by Process B.
- **No Preemption**: Resources **cannot be forcibly taken** from the processes holding them; they must be released voluntarily.

  Example: You can't just take a file from a process — it must release it on its own.
- **Circular Wait**: A set of processes {P_1, P_2, ..., P_n} exists such that **P_1 is waiting for a resource held by P_2, P_2 is waiting for a resource held by P_3, ..., and P_n is waiting for a resource held by P_1**, forming a circular chain.
  - **Example**: P1 → P2 → P3 → P1 (circular dependency).

Deadlock prevention and avoidance

NOTE : Deadlock can occur **only if all four** of these conditions hold **simultaneously**. If **even one** of them is prevented or broken, **deadlock cannot occur**.

- Deadlock prevention and avoidance are the strategies used in operating systems to ensure that processes do not get stuck waiting for resources indefinitely.

Deadlock Prevention

Deadlock prevention works by eliminating one of the four necessary conditions for deadlock:

1. **Mutual Exclusion** – Ensuring that resources are shared whenever possible.
2. **Hold and Wait** – Requiring processes to request all resources at once, preventing them from holding some while waiting for others.
3. **No Preemption** – Allowing the system to forcibly take resources from a process if needed.
4. **Circular Wait** – Imposing an ordering on resource requests to prevent circular dependencies.

Deadlock Avoidance

- Deadlock avoidance, on the other hand, does not eliminate conditions but ensures that the system never enters an unsafe state. It relies on algorithms like:
    - **Banker's Algorithm** – Ensures that resource allocation always leaves the system in a safe state.
    - **Resource Allocation Graph** – Tracks dependencies and prevents cycles from forming.

Semaphore

- A Semaphore is a synchronization mechanism used in operating systems to manage access **to shared resources and prevent race conditions** in concurrent processes.

- Semaphores help ensure **mutual exclusion**, **process synchronization**, and **deadlock prevention** in multi-threaded environments

Types of Semaphores

1. Counting Semaphore – Allows multiple processes to access a resource up to a certain limit.
2. Binary Semaphore – Works like a lock, allowing only one process at a time.

Operations

- Wait (P operation) – Decreases the semaphore value; if it's already zero, the process waits.
- Signal (V operation) – Increases the semaphore value, allowing waiting processes to proceed.

Mutex

- A mutex (short for **mutual exclusion**) is a synchronization mechanism used in operating systems **to control access to shared resources in a multi-threaded environment.**

How Mutex Works

- A **mutex lock** ensures that only one thread can access a critical section at a time.
- When a thread acquires the mutex, other threads must wait until it is released.
- Once the thread finishes its operation, it releases the mutex, allowing another thread to proceed.

Types of Mutex

- Recursive Mutex – Allows the same thread to lock the mutex multiple times without causing a deadlock.
- Error-Checking Mutex – Prevents a thread from acquiring a mutex it already holds, helping detect programming errors.

Mutex vs Semaphore

- Mutex is a binary lock (either locked or unlocked), while semaphores can allow multiple threads to access a resource up to a defined limit.
- Mutex is used for mutual exclusion, whereas semaphores can be used for synchronization.

Producer-Consumer Problem

- The Producer-Consumer Problem is a classic synchronization problem in operating systems that illustrates the need for proper coordination between processes or threads that share a common resource, such as a buffer.

Problem Overview : There are two types of processes:

- Producer: Generates data and puts it into a shared buffer.
- Consumer: Takes data from the shared buffer and processes it.

Problem:

- The buffer has limited size.
- The producer must wait if the buffer is full.
- The consumer must wait if the buffer is empty.
- Both must not access the buffer at the same time (to avoid race conditions).

Mutual Exclusion : Ensure only one process accesses the buffer at a time.
Synchronization : Coordinate producer and consumer to prevent overfilling or underflow.
Critical Section : The part of the code where shared resources are accessed.

Solution Approaches : Using Semaphores

- Print Spooler: Produces print jobs and adds them to a queue; printer (consumer) processes them.
- Data Streaming:Video/audio producer generates frames/samples; consumer renders them.

Dead-lock vs Starvation

Feature	Deadlock	Starvation
Definition	Circular waiting with no progress	Process waits indefinitely
System state	Entire group of processes are stuck	Other processes may continue
Cause	Circular wait on resources	Biased resource allocation (e.g. priority)
Detection	Detectable using algorithms	Harder to detect
Resolution	Needs deadlock recovery or prevention	Needs fair scheduling (e.g., aging)
Processes affected	All in the cycle	One or few low-priority processes

- Deadlock = No one can proceed.
- Starvation = One can't proceed, but others can.
- Deadlock is often more critical and harder to resolve than starvation.
- Starvation can happen without deadlock, but deadlock often causes starvation.