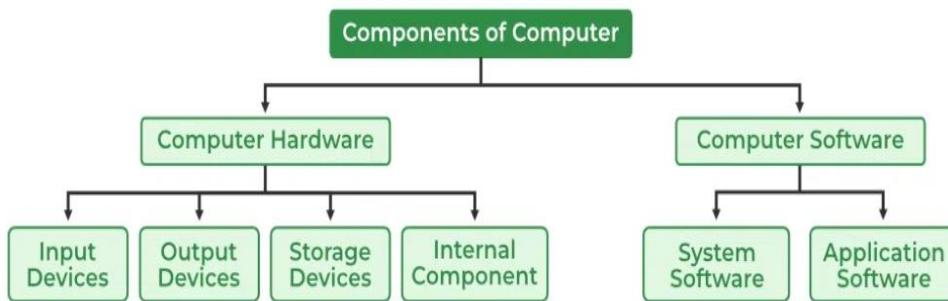


A computer system is divided into two categories: Hardware and Software.

**Hardware** refers to the physical and visible components of the system such as a monitor, CPU, keyboard and mouse.

**Software**, on the other hand, refers to a set of instructions which enable the hardware to perform a specific set of tasks.



### Types of Computer Software

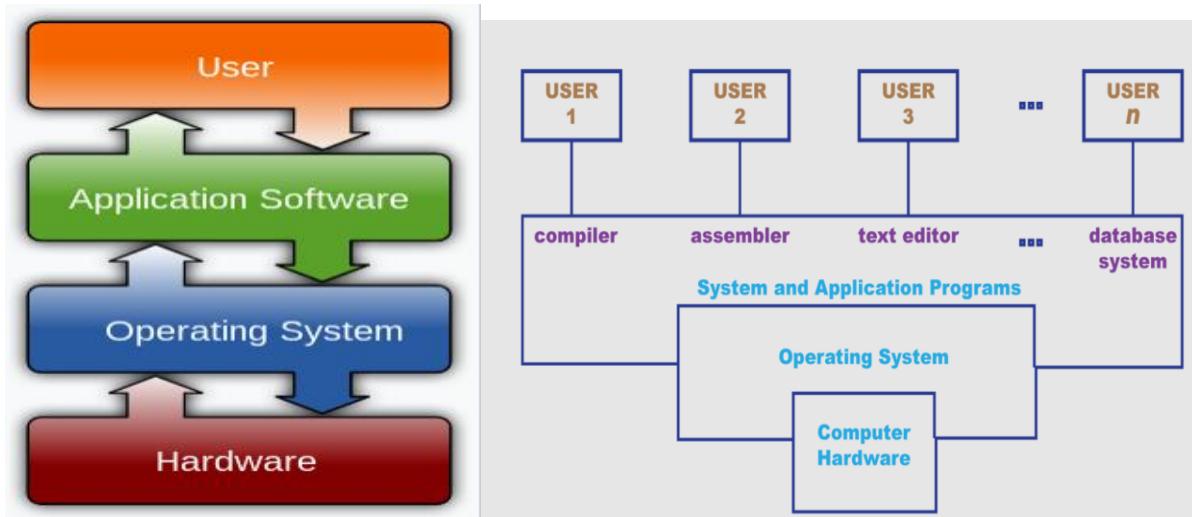
- System Software
- Application Software

**System Software:** System Software is a component of Computer Software that directly operates with Computer Hardware which has the work to control the Computer's Internal Functioning and also takes responsibility for controlling Hardware Devices such as Printers, Storage Devices, etc. Types of System Software include Operating systems, Language processors, and Device Drivers.

**Application Software:** Application Software are the software that works the basic operations of the computer. It performs a specific task for users. Application Software basically includes Word Processors, Spreadsheets, etc. Types of Application software include General Purpose Software, Customized Software, etc.

### Definition:

Operating System can be defined as a system software which acts as an interface between user and the hardware. It manages all other applications and programs in a computer, and it is loaded into the computer by a boot program.

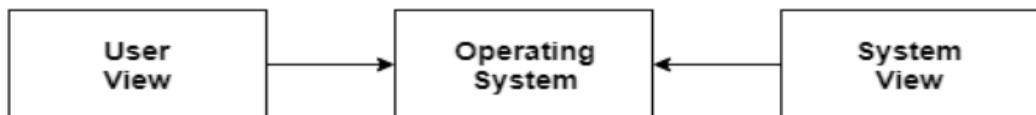


### Operating System Objectives

- To make the computer system convenient to use in an efficient manner.
- To hide the details of the hardware resources from the users.
- To provide users a convenient interface to use the computer system.
- To act as an intermediary between the hardware and its users, making it easier for the users to access and use other resources.
- To manage the resources of a computer system.
- To keep track of who is using which resource, granting resource requests, and mediating conflicting requests from different programs and users.
- To provide efficient and fair sharing of resources among users and programs.

### User View and System View

The operating system can be observed from the point of view of the user or the system. This is known as the user view and the system view respectively. More details about these are given as follows



**User View:**

The user view depends on the system interface that is used by the users. The different types of user view experiences can be explained as follows –

- If the user is using a **personal computer**, the operating system is largely designed to make the interaction easy. Some attention is also paid to the performance of the system, but there is no need for the operating system to worry about resource utilization. This is because the personal computer uses all the resources available and there is no sharing.
- If the user is using a system connected to a **mainframe or a minicomputer**, the operating system is largely concerned with resource utilization. This is because there may be multiple terminals connected to the mainframe and the operating system makes sure that all the resources such as CPU, memory, I/O devices etc. are divided uniformly between them.
- If the user is sitting on a **workstation** connected to other workstations through networks, then the operating system needs to focus on both individual usage of resources and sharing though the network. This happens because the workstation exclusively uses its own resources but it also needs to share files etc. with other workstations across the network.
- If the user is using a **handheld computer** such as a mobile, then the operating system handles the usability of the device including a few remote operations. The battery level of the device is also taken into account.
- There are some devices that contain very less or no user view because there is no interaction with the users. Examples are embedded computers in home devices, automobiles etc.

#### **System View:**

From the System point of view the operating system is the program which is most intermediate with the hardware.

- The system views the operating system as a **resource allocator**. There are many resources such as CPU time, memory space, file storage space, I/O devices etc. that are required by processes for execution. It is the duty of the operating system to allocate these resources judiciously to the processes so that the computer system can run as smoothly as possible.
- The operating system can also work as a **control program**. It manages all the processes and I/O devices so that the computer system works smoothly and there are no errors. It makes sure that the I/O devices work in a proper manner without creating problems.

## **Operating System Services**

Operating system services are fundamental functionalities provided by an operating system (OS) to support the execution of application software.

An Operating System provides services to both the users and to the programs. It provides programs, an environment to execute. It provides users, services to execute the programs in a convenient manner.

Following are the services provided by operating systems to both users and to the programs:

### **Users**

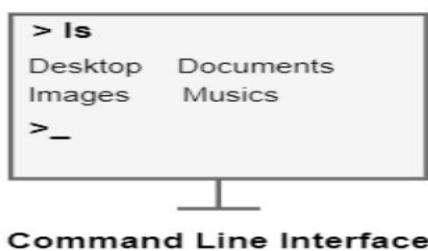
- User Interface
- Program Execution
- I/O Operations
- File-System Manipulation
- Communication
- Error Detection

### **System/Programs**

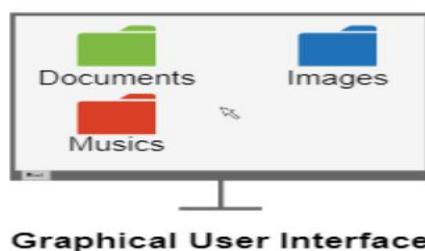
- Resource Allocation
- Accounting
- Protection and Security

#### ***1. User Interface***

An interface allow users to interact with the system more easily and intuitively. Many operating systems provide user interfaces, such as command-line interfaces (CLI) ,graphical user interfaces (GUI) and Batch Based Interface.



**Command Line Interface**



**Graphical User Interface**

A Command-Line Interface (CLI) is a text-based user interface used to interact with software and operating systems. Through a CLI, users can input text commands to perform specific tasks

A GUI usually consists of all the graphical icons displayed on a computer screen, visual indicators like widgets, texts, labels, and text navigation. Thus, a user can directly perform actions with a click of the mouse or keyboard.

A Batch-based interface in computing refers to a method of interaction between the user and the computer system or software where commands or a series of commands are prepared in advance and then executed all at once, rather than being entered and executed one at a time.

## ***2. Program Execution***

The OS loads a program into memory and then executes that program. It also makes sure that once started that program can end its execution, either normally or forcefully. The major steps during program management are:

- Loading a program into memory.
- Executing the program.
- Making sure the program completes its execution.

## ***3. I/O Operations***

I/O operations are required during the execution of a program. To maintain efficiency and protection of the program, users cannot directly govern the I/O devices instead the OS allows to read or write operations with any file using the I/O devices and also allows access to any required I/O device when required.

## ***4. File System Manipulation***

A program is read and then written in the form of directories and files. These files can be stored on the storage disk for the long term.

Following are the major activities of an operating system with respect to file management.

- Program needs to read a file or write a file.
- The operating system gives the permission to the program for operation on file.
- Permission varies from read-only, read-write, denied and so on.
- Operating System provides an interface to the user to create/delete files.
- Operating System provides an interface to the user to create/delete directories.
- Operating System provides an interface to create the backup of file system.

## **5. Communication systems**

Processes need to swap information among themselves. These processes can be from the same computer system or different computer systems as long as they are connected through communication lines in a network. This can be done with the help of OS support using shared memory or message passing. The OS also manages routing, connection strategies, and the problem of contention and security.

## **6. Error Detection**

Errors may occur in any of the resources like CPU, I/O devices, or memory hardware. The OS keeps a lookout for such errors, corrects errors when they occur, and makes sure that the system works uninterruptedly.

## **7. Resource Allocation/Resource Management**

Resource management in operating systems (OS) is a critical function that involves managing the hardware and software resources of a computer system. These resources include the CPU (central processing unit), memory (both RAM and storage), and networking components.

### **CPU /Process Management:**

The OS is responsible for deciding which processes get to use the CPU when there are multiple processes needing to run. This is handled through CPU scheduling algorithms (e.g., Round-Robin, Shortest Job First, Priority Scheduling) that aim to optimize CPU utilization and ensure a fair and efficient allocation of CPU time to processes.

### **Memory Management:**

Memory management involves allocating and managing the system's primary memory or RAM. The OS must track which parts of memory are in use and by whom, allocate memory to processes when they need it, and deallocate it when they are done. Techniques such as paging and segmentation are used to manage memory efficiently and protect the memory space of processes from each other.

### **Network Resource Management:**

For systems connected to a network, the OS manages the network configuration and communication protocols to enable data exchange between computers. This includes managing network interfaces, routing, bandwidth allocation, and implementing security measures for network connections.

## **8. Accounting**

This keeps a check of which resource is being used by a user and for how long it is being used. This is usually done for statistical purposes.

## **9. Protection and Security**

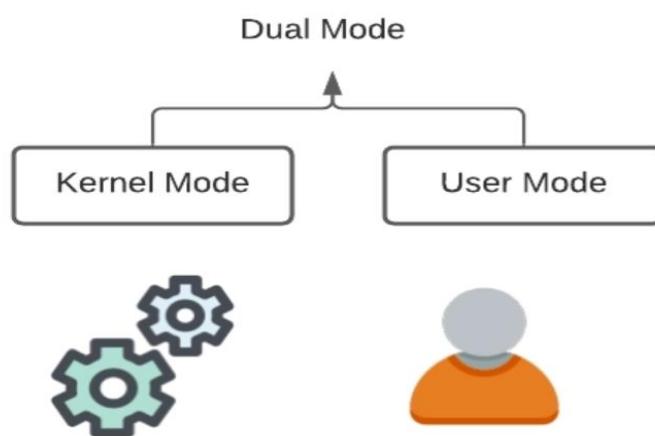
Considering a computer system having multiple users the concurrent execution of multiple processes, then the various processes must be protected from each another's activities. Protection refers to mechanism or a way to control the access of programs, processes, or users to the resources defined by a computer systems.

Following are the major activities of an operating system with respect to protection.

- OS ensures that all access to system resources is controlled.
- OS ensures that external I/O devices are protected from invalid access attempts.
- OS provides authentication feature for each user by means of a password

## **Operating System Operation**

In order to ensure the proper execution of the operating system, we must be able to distinguish between the execution of operating-system code and user defined code. The approach taken by most computer systems is to provide hardware support that allows us to differentiate among various modes of execution. At the very least, we need two separate modes of operation: **user mode** and **kernel mode** (also called **supervisor mode**, **system mode**, or **privileged mode**)



### **User mode**

When the computer system run user applications like creating a text document or using any application program, then the system is in the user mode. When the user application requests for a

service from the operating system or an interrupt occurs or system call, then there will be a transition from user to kernel mode to fulfill the requests.

The mode bit is set to 1 in the user mode. It is changed from 1 to 0 when switching from user mode to kernel mode.

### Kernel Mode

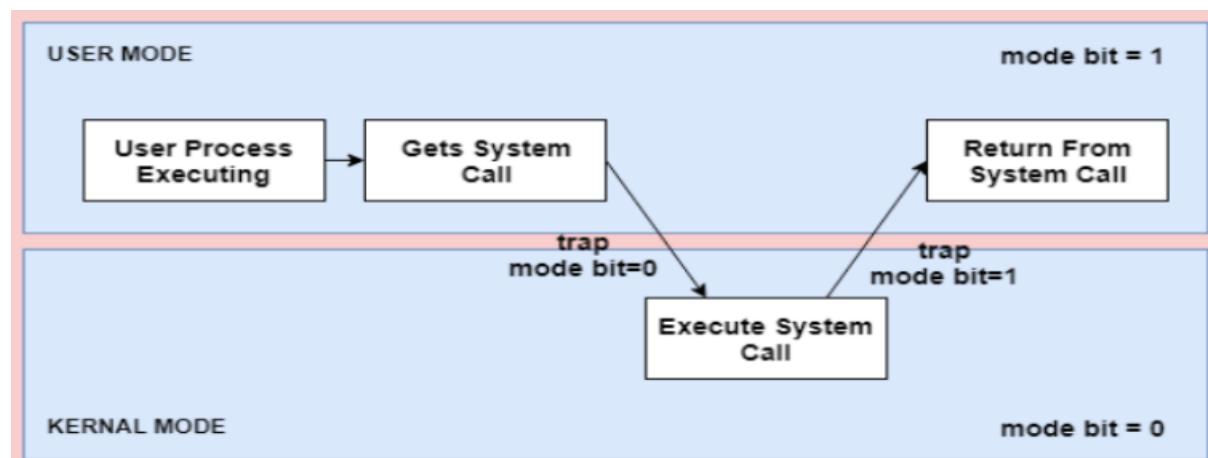
When the system boots, hardware starts in kernel mode and when operating system is loaded, it starts user application in user mode. To provide protection to the hardware, we have privileged instructions which execute only in kernel mode. If user attempt to run privileged instruction in user mode then it will treat instruction as illegal and traps to OS.

Some of the privileged instructions are:

1. Handling Interrupts
2. To switch from user mode to kernel mode.
3. Input-Output management.

The mode bit is set to 0 in the kernel mode. It is changed from 0 to 1 when switching from kernel mode to user mode.

A diagram that illustrates the transition from user mode to kernel mode and back again is as follows



- At system boot time, the hardware starts in **kernel mode**. Thus, whenever the operating system gains control of the computer, it is in **kernel mode**.
- The operating system is then loaded and starts user applications in **user mode**. The system always **switches to user mode** (by setting the mode bit to 1) before passing control to a user program.

- When the computer system is executing on behalf of a user application, **the system is in user mode**. However, when a user application requests a service from the operating system (via a system call), it must **transition from user to kernel mode** to fulfill the request.

**Or**

- Whenever a trap (Traps are occurred by the user program to invoke the functionality of the OS. Assume the user application requires something to be printed on the screen, and it would set off a trap, and the operating system would write the data to the screen.) or interrupt occurs, the hardware switches from **user mode to kernel mode** (that is, changes the state of the mode bit to 0).

### **Timer**

- At every point, we must ensure that the operating system maintains control over the CPU. We can't allow a user program to get stuck in an infinite loop or to fail to call system services and never return control to the operating system. Now to accomplish this goal, we can use a **timer**. A timer can be set to interrupt the computer after a specified period. The period may be *fixed* (1/60 sec) or *variable* (1ms to 1s).
- A **variable timer** is generally implemented by a fixed-rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs.

## Operating System Structures

Operating System (OS) structure refers to the way an operating system is designed and organized to manage hardware and software resources, provide services to users and applications, and ensure efficient and secure operation of a computer system. There are several different approaches to operating system structure, each with its own advantages and disadvantages.

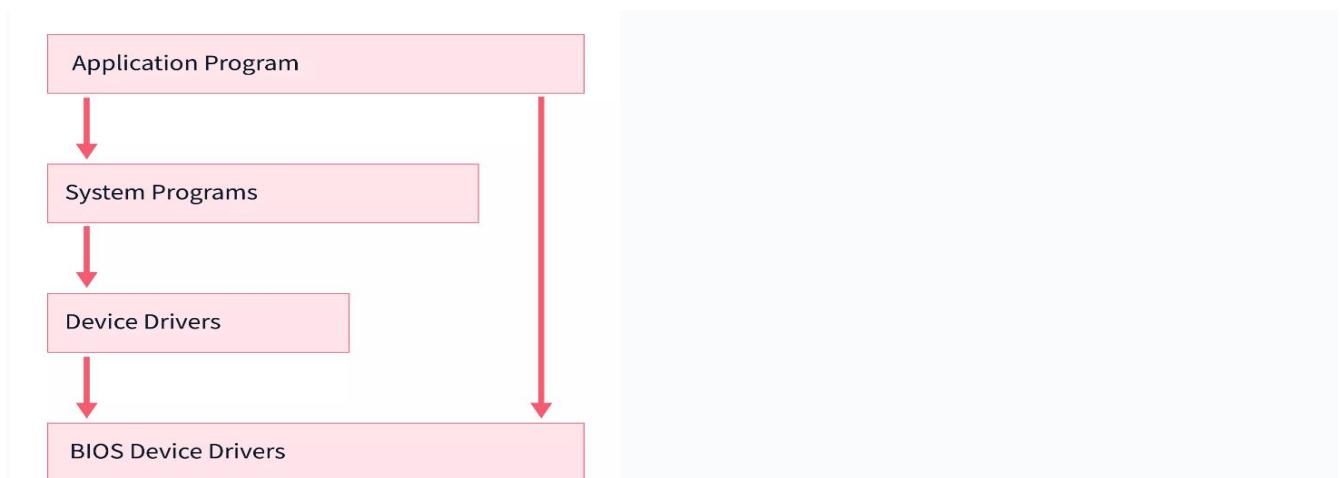
**Different types of structures implementing Operating Systems as mentioned below.**

1. Simple Structure
2. Layered Structure
3. Micro Kernel Structure
4. Modular Structure
5. Hybrid Structure

### Simple Structure

It is the simplest Operating System Structure and is not well defined. It can only be used for small and limited systems. In this structure, the interfaces and levels of functionality are well separated, hence programs can access I/O routines which can cause unauthorized access to I/O routines.

This structure is implemented in MS-DOS operating system. The **MS-DOS operating System is made up of various layers, each with its own set of functions.**



### Advantages of Simple Structure

- It is easy to develop because of the limited number of interfaces and layers.
- Offers good performance due to lesser layers between hardware and applications.

- Minimal overhead, suitable for resource-constrained environments.

### **Disadvantages of Simple Structure**

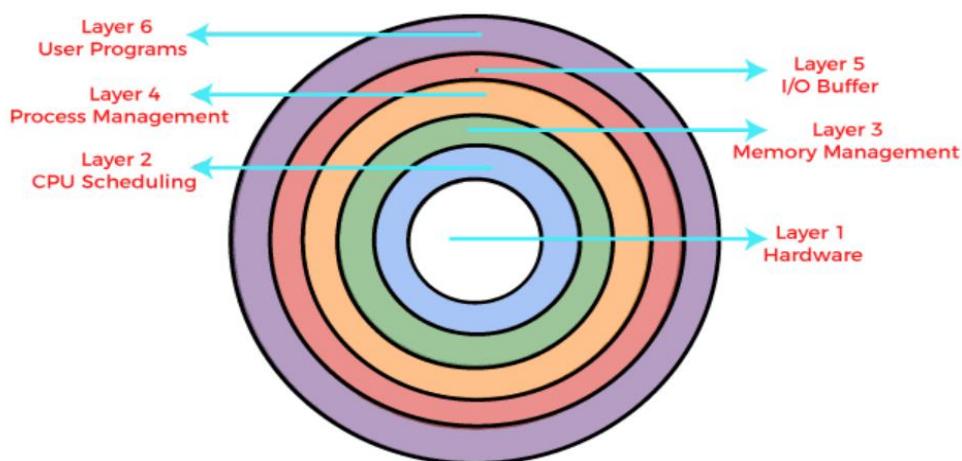
- If one user program fails, the entire operating system crashes.
- Limited functionality.
- Abstraction or data hiding is not present as layers are connected and communicate with each other.
- Layers can access the processes going in the Operating System, which can lead to data modification and can cause Operating System to crash.

### **Layered Structure**

In this type of structure, OS is divided into layers or levels. The hardware is on the bottom layer (layer 0), while the user interface is on the top layer (layer N). These layers are arranged in a hierarchical way in which the top-level layers use the functionalities of their lower-level levels. **Example:Linux**  
The following are some of the key characteristics of a layered operating system structure:

- Each layer is responsible for a specific set of tasks. This makes it easier to understand, develop, and maintain the operating system.
- Layers are typically arranged in a hierarchy. This means that each layer can only use the services provided by the layers below it.
- Layers are independent of each other. This means that a change to one layer should not affect the other layers.

Below is the Image illustrating the Layered structure in OS:



### **Advantages of Layered Structure**

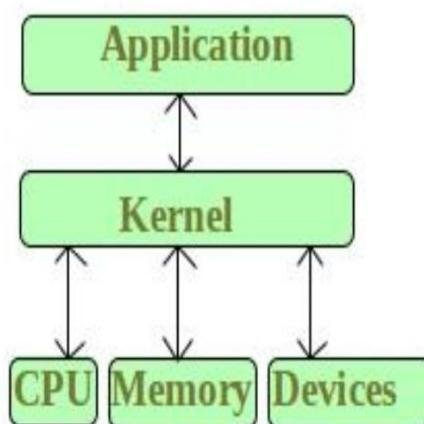
- A layered structure is highly modular, meaning that each layer is responsible for a specific set of tasks. This makes it easier to understand, develop, and maintain the operating system.
- Each layer has its functionalities, so work tasks are isolated, and abstraction is present up to some level.
- Debugging is easier as lower layers are debugged, and then upper layers are checked.

### **Disadvantages of Layered Structure**

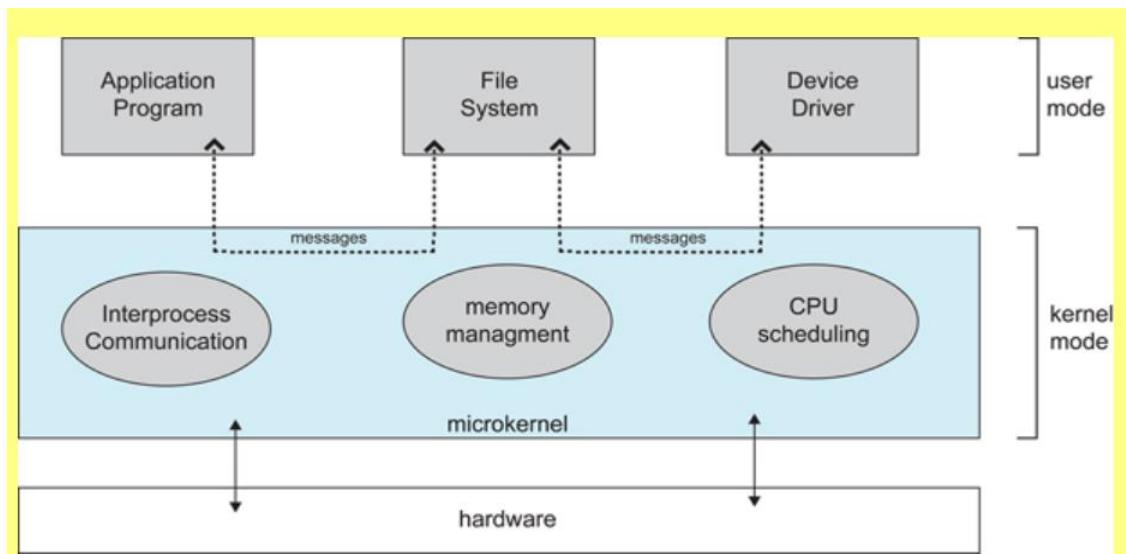
- In Layered Structure, layering causes degradation in performance.
- It takes careful planning to construct the layers since higher layers only utilize the functions of lower layers.
- There can be some performance overhead associated with the communication between layers. This is because each layer must pass data to the layer above it.

### **Micro-Kernel structure**

**Kernel** is the core part of an operating system that manages system resources. It also acts as a bridge between the application and hardware of the computer. It is one of the first programs loaded on start-up (after the Bootloader).



A microkernel is a type of operating system kernel that is designed to provide only the most basic services required for an operating system to function, such as memory management and process scheduling. Other services, such as device drivers and file systems, are implemented as user-level processes that communicate with the microkernel via message passing. **Ex:Mac OS**



### **Advantages of Micro-kernel structure:**

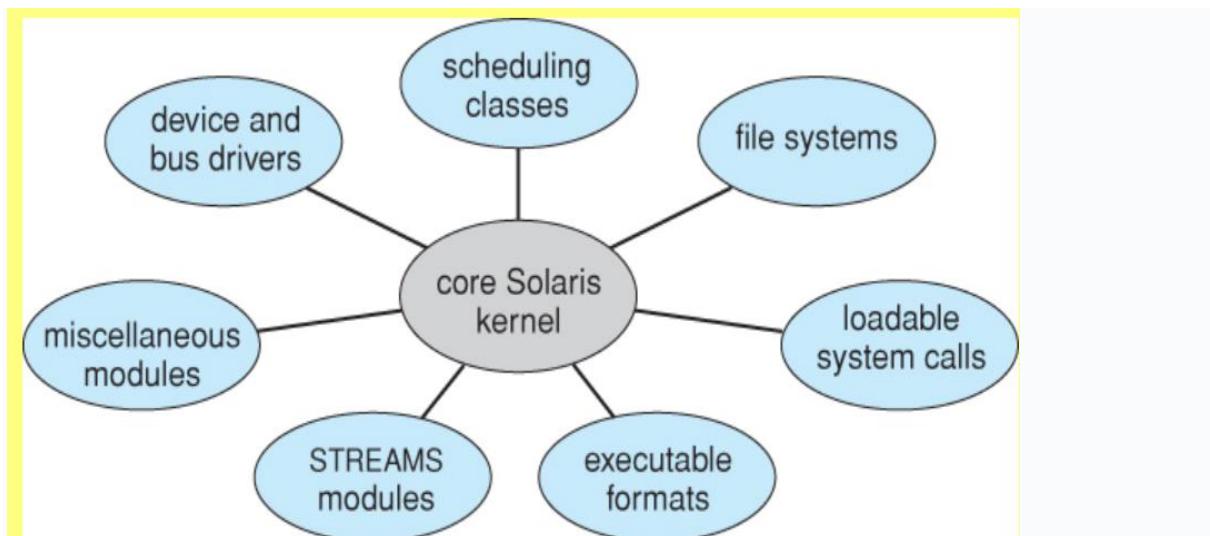
- It allows the operating system to be portable between platforms.
- Enhanced system stability and security.
- As each Micro-Kernel is isolated, it is safe and trustworthy.
- Because Micro-Kernels are smaller, they can be successfully tested.
- If any component or Micro-Kernel fails, the remaining operating System is unaffected and continues to function normally.

### **Disadvantages of Micro-kernel structure:**

- Increased inter-module communication reduces system performance.
- System is complex to be constructed.
- Complexity in managing user-space components.

### **Modular Structure**

In a modular operating system structure, the operating system is divided into a set of independent modules. Each module is responsible for a specific task, such as memory management, process scheduling, or device drivers. Modules can be loaded and unloaded dynamically, as needed.**EX:Solaris**



### Advantages of Modular Structure

- A modular structure is highly modular, meaning that each module is independent of the others. This makes it easier to understand, develop, and maintain the operating system.
- A modular structure is very flexible. New modules can be added easily, and existing modules can be modified or removed without affecting the rest of the operating system.

### Disadvantages of Modular Structure

- There can be some performance overhead associated with the communication between modules. This is because modules must communicate with each other through well-defined interfaces.
- A modular structure can be more complex than other types of operating system structures. This is because the modules must be carefully designed to ensure that they interact correctly

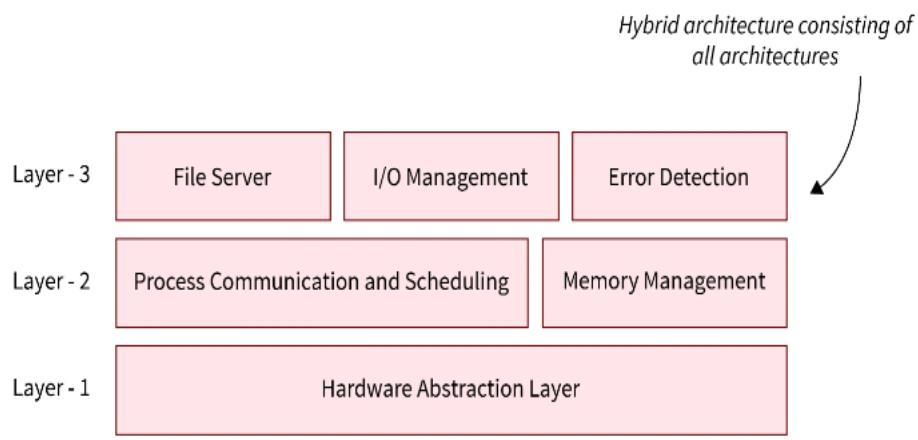
### Hybrid Structure

Hybrid Structure as the name suggests consists of a hybrid of all the Structure explained so far and hence it has properties of all of those architectures which makes it highly useful in present-day operating systems. The hybrid-Structure consists of three layers.

- 1) Hardware abstraction layer:** It is the interface between the kernel and hardware and is present at the lowest level.
- 2) Microkernel Layer:** This is the old microkernel that we know and it consists of CPU scheduling, memory management, and inter-process communication.

**3) Application Layer:** It acts as an interface between the user and the microkernel. It contains functionalities like a file server, error detection, I/O device management, etc.

**Example: Microsoft Windows NT kernel implements a hybrid architecture of the operating system.**



#### **Advantages:**

1. Since it is a hybrid of other structures it allows various structures to provide their services respectively.
2. It is easy to manage because it uses a layered approach.
3. The number of layers is relatively lesser.
4. Security and protection are relatively improved.

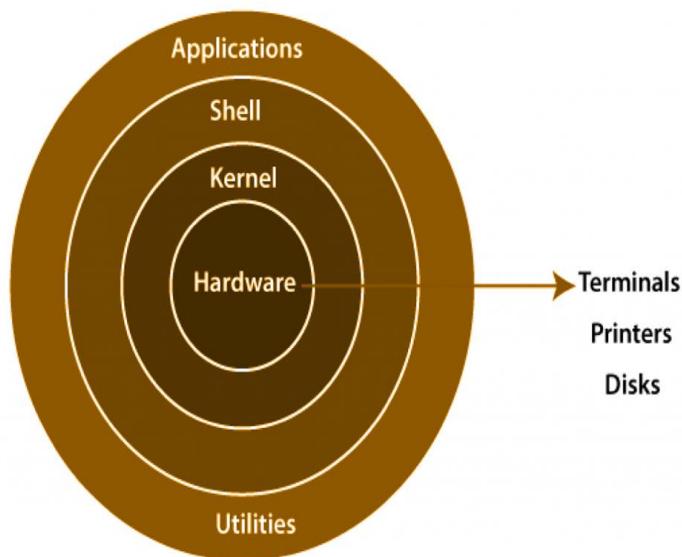
#### **Disadvantage:**

1. It increases overall complexity of system by implementing both structure (monolithic and micro) and making the system difficult to understand.

## Linux Architecture

Linux is similar to other operating systems you may have used before, such as Windows, macOS (formerly OS X), or iOS. Like other operating systems, Linux has a graphical interface, and the same types of software you are accustomed to, such as word processors, photo editors, video editors, and so on. Linux was created in 1991 by Linus Torvalds, a then-student at the University of Helsinki. Torvalds built Linux as a free and open source alternative to Minix, another Unix clone that was predominantly used in academic settings. He originally intended to name it “Freax,” but later renamed it as “Linux” after a combination of Torvalds’ first name and the word Unix.

The diagram illustrates the structure of the Linux system, according to the layers concept.



The Linux architecture is largely composed of elements such as the Applications, Shell , Kernel, Hardware layer, System Utilities and Libraries.

### Applications

Applications are the programs that the user runs on top of the architecture. The applications are the user space element that includes database applications, media players, web browsers, and presentations.

### Shell

Shell is the interface that interacts with humans and processes the commands that are given for the execution. One can call it an interpreter because it takes the command from the keyboard and makes it understandable to the kernel.

Shells are categorized into two sections:

*Command Line Shell*

*Graphical User Shell*

**The command line shell** is the user interface where the user types commands in a text form. When the user provides the command in the terminal, the shell interprets the commands for the kernel. The shell also has some built-in commands that help the user to navigate, manage, and change the file system

**The graphical user shell** is the user interface using the system's peripheral components like a mouse, and keyboard. It is beneficial for users who are not familiar with commands. These shells are used to make the desktop environment easier.

## **Kernel**

In Linux operating systems, the kernel is the core component that acts as the bridge between computer hardware and applications. It is responsible for managing the system's resources and allowing software and hardware to communicate with each other.

## **System Utilities and Libraries**

The system utilities and libraries provide a wide range of functions to manage the system. Low-level hardware complexity to high-level user support is served by the system utilities and libraries.

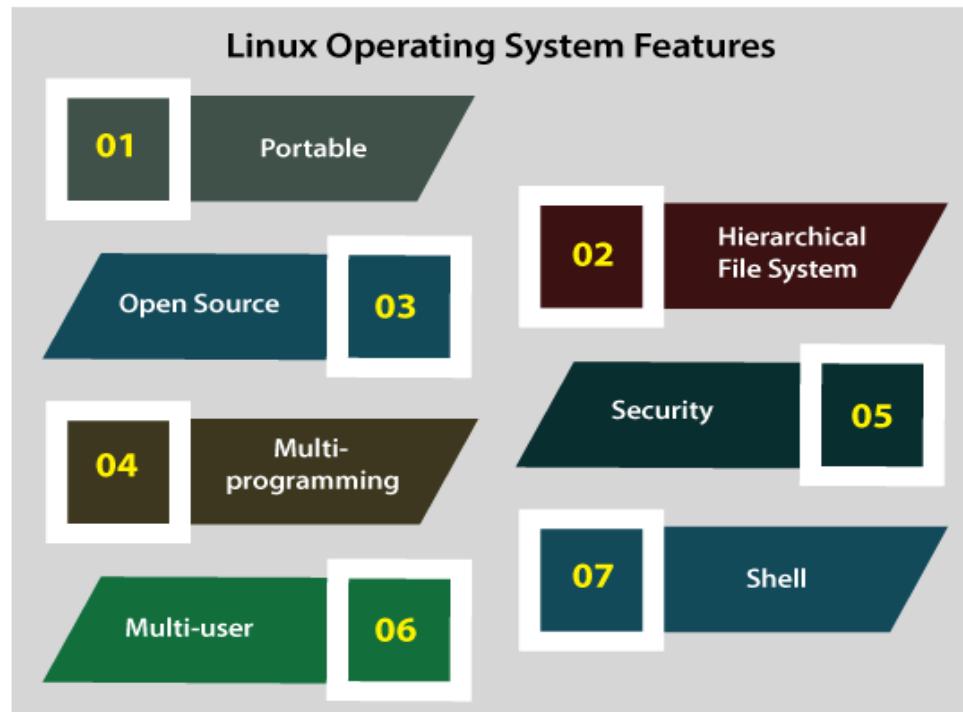
- **System Utilities:** It is the program that performs the task which is given by the users and manages the system.(Antivirus,Winrar,Winzip etc)
- **System Libraries:** Functions through which the system interacts with the kernel and handles all the functionalities without the kernel privileges

## **Hardware**

Hardware layer of Linux is the lowest level of operating system track. It plays a vital role in managing all the hardware components. It includes device drivers, kernel functions, memory management, CPU control, and I/O operations. This layer generalizes hard complexity, by providing an interface for software by assuring proper functionality of all the components.

**Some Linux distributions are:** MX Linux, Manjaro, Linux Mint, elementary, Ubuntu, Debian, Solus, Fedora, openSUSE, Deepin

## Linux Operating System Features



1. **Portable:** Linux OS can perform different types of hardware and the kernel of Linux supports the installation of any type of hardware environment.
2. **Hierarchical file system:** Linux OS affords a typical file structure where user files or system files are arranged.
3. **Open source:** Linux operating system source code is available freely and for enhancing the capability of the Linux OS, several teams are performing in collaboration.
4. **Multiprogramming:** Linux OS can be defined as a multiprogramming system. It means more than one application can be executed at the same time.
5. **Security:** Linux OS facilitates user security systems with the help of various features of authentication such as controlled access to specific files, password protection, or data encryption.
6. **Multi-user:** Linux OS can also be defined as a multi-user system. It means more than one user can use the resources of the system such as **application programs**, **memory**, or **RAM** at the same time.
7. **Shell:** Linux operating system facilitates a unique interpreter program. This type of program can be applied for executing commands of the operating system. It can be applied to perform various types of tasks such as call application programs and others.

## Linux Commands

### echo command

The **echo** command in Linux and Unix-like operating systems is a built-in command used primarily to display messages or output text to the terminal or into a file.

### Syntax

echo [option] [string]

#### 1. How to input a text to get the output using the echo command?

Input : echo "Linux Commands"

Output:- Linux Commands

#### 2. How to print a variable?

x=100

Input : echo "The value of x =\$x"

Output : The value of x = 100

The following tabular gives you the possible options in the echo command:

Options	Description
-e	Enable interpretation of backslash escape sequences
-E	Disable interpretation of backslash escape sequences
\a	Alert return
\n	To add a new line
\t	To add a Horizontal tab spaces
\v	To add a Vertical tab spaces
\c	To stop displaying the output from this stage
\r	Carriage Return
\	To perform as Backslash ('\\')
\b	To behave as the backspace
\f	Adding a Form Feed into your texts

#### 3. How to add a new line?

Input: echo -e "Linux \n commands"

Output:Linux  
Commands

Note: The Linux "echo" command provides the "-e" option, which instructs the command to interpret backslash escape sequences in the specified text. By utilizing the '\n' sequence, we can

add a new line and print the subsequent text on a new line below the previous one. This enables us to format the output and make it more readable by separating it into distinct lines.

## 5. How to add a horizontal tab to your content?

```
echo -e "Linux \t commands"
```

Output: Linux    commands

## PATH

PATH is an Environment variable in Linux .Environment variables are dynamic values that affect the processes or programs on a computer. It essentially tells the shell which directories to search through to find the executable files (programs or scripts) that match the command names entered by the user. This search process occurs in the order the directories are listed within the **PATH** variable.

**To view the path** (echo command)

```
$echo $PATH
```

### Output

```
usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/local/games
```

The **PATH** variable contains a list of directories separated by colons (:). When a command is entered, the shell searches for an executable file with the command's name starting from the first directory listed in the **PATH** variable and proceeds to the next if it's not found. This continues until either the executable is found or the list is exhausted. If the executable is not found in any of the directories listed in **PATH**, the shell typically returns a "command not found" error.

## Adding a Directory to the PATH Environment Variable(export Command)

A directory can be added to PATH in two ways: at the start or the end of a path.

Adding a directory (/the/file/path for example) to the **start of PATH** will mean it is checked first:

```
export PATH=/the/file/path:$PATH
```

```
echo $PATH
```

### Output :

```
/the/file/path:usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/local/games
```

Adding a directory to the **end of PATH** means it will be checked after all other directories:

```
export PATH=$PATH:/the/file/path
```

```
echo $PATH
```

**Output :**

```
usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/local/games:/ the/file/path
```

Multiple directories can be added to PATH at once by adding a colon: between the directories:

```
export PATH=$PATH:/the/file/path:/the/file/path2
```

```
echo $PATH
```

**Output :**

```
usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/local/games:/
```

```
the/file/path:/the/file/path2
```

## man

The "man" is a short term for manual page. In unix like operating systems such as linux, man is an interface to view the system's reference manual. A user can request to display a man page by simply typing man followed by a space and then argument. Here its argument can be a command, utility or function. A manual page associated with each of these arguments is displayed.

**The basic man command syntax is:**

```
man [option] [section number] [command name]
```

**option** – the search result output.

**section number** – the section in which to look for the man page.

**command name** – the name of the command which man page you want to see.

But generally [option(s) and section number ] are not used. Only command is written as an argument.

**some options include**

- **-k KEYWORD** (Search for the **KEYWORD** in the whole **manual** page and shows all the **matches**)
- **-f KEYWORD** (Look for a short description of any **KEYWORD** or Command)
- **-d, --default** (Resets the **man** command behavior to **default**)
- **-i, --ignore-case** (**Ignore case sensitivity** of the command)
- **-l, --match-case** (Looking inside the man page with **case sensitivity**)

- **-a, --all** (Shows all manual pages that match the specific keyword or command)

By default, **man** looks in all the available sections of the manual and shows the first match (even if the page exists in several sections). Providing a section number instructs the **man** command to look in a specific section.

There are nine sections of the manual:

1. **General commands:** Commands used in the terminal.
2. **System calls:** Functions the kernel provides.
3. **Library functions:** Functions in program libraries.
4. **Special files:** Usually devices found in **/dev** and related drivers.
5. **File formats and conventions:** File formats like **etc/passwd**.
6. **Games:** Descriptions of commands that display database quotes.
7. **Miscellaneous:** Various descriptions, including macro packages and conventions, boot parameters, and others.
8. **System administration commands:** Commands mostly reserved for root.
9. **Kernel Routines:** Information about internal kernel operations.

## Example

### Syntax of command without option and section

**man ls**

This command will display all the information about 'ls' command as shown in the screen shot.

```
LS(1)                               User Commands                               LS(1)

NAME
    ls - list directory contents

SYNOPSIS
    ls [OPTION]... [FILE]...

DESCRIPTION
    List information about the FILEs (the current directory by default).
    Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.

    Mandatory arguments to long options are mandatory for short options too.

    -a, --all
        do not ignore entries starting with .

    -A, --almost-all
        do not list implied . and ..

    --author
    Manual page ls(1) line 1 (press h for help or q to quit)
```

### Syntax for a particular section:

**man section\_number command**

**Example :** man 2 passwd

The **man 2 passwd** command in Linux is intended to show the manual page for the **passwd** system call, which is found in section 2 of the manual. Section 2 of the manual pages typically documents system calls, which are interfaces provided by the Linux kernel.

### **printf**

*printf* command is used to output a given string, number or any other format specifier. The command operates the same way as *printf* in C, C++, and Java programming languages.

**Example**

\$ printf "%s" "Hello, Welcome to KMIT"

**Output:** Hello, Welcome to KMIT

### **script**

**script** command in Linux is used to make typescript or record all the terminal activities. After executing the *script* command it starts recording everything printed on the screen including the inputs and outputs until exit. By default, all the terminal information is saved in the file *typescript* , if no argument is given.

### **script**

We can specify a filename as an argument to save the output to a different file:

**script my\_session.log**

To end the recording session, type **exit** or press **Ctrl-D**. This will return you to your normal terminal session and stop recording.

**Example:**

**script my\_session.log**

**echo "Hello, World!"**

**ls**

**exit**

### **passwd**

The **passwd** command in Linux and Unix-like operating systems is used to change the password of a user account. It can be used by both the system administrators to change other users' passwords and by individual users to update their own passwords.

## Basic Usage

For a user to change their own password, they would simply type:

**passwd**

Upon execution, the system prompts the user to enter their current password (for verification), followed by the new password, and then to retype the new password for confirmation.

## uname

The **uname** command in Linux and Unix-like operating systems is used to display system information. It provides details about the kernel name, version, and other system information. By default, without any options, **uname** will print the kernel name.

## Options

The **uname** command supports several options that can be used to display specific system information:

- **-a, --all**: Print all available system information (kernel name, nodename, kernel release, kernel version, machine, processor, hardware platform, and operating system).
- **-s, --kernel-name**: Print the kernel name.
- **-n, --nodename**: Print the network node hostname.
- **-r, --kernel-release**: Print the kernel release.
- **-v, --kernel-version**: Print the kernel version.
- **-m, --machine**: Print the machine hardware name (e.g., x86\_64).
- **-p, --processor**: Print the processor type or "unknown".
- **-i, --hardware-platform**: Print the hardware platform or "unknown".
- **-o, --operating-system**: Print the operating system.

## Example

### Displaying the Kernel Name

*Simply typing uname without any options will display the kernel name*

*Example output might be Linux for a Linux system.*

## who

The **who** command in Linux and Unix-like operating systems is used to display information about users who are currently logged into the system. It provides a list of users, the terminals they are logged in from, the login time, and sometimes the host from which they are accessing the system.

### Basic Usage

To run the command, simply type:

**who**

This will display a list that typically includes the username, terminal name , the date and time of login, and the remote host name or IP address from which the user is accessing the system.

### Options

The **who** command supports several options that can alter its output or provide additional information:

- **-a, --all:** Show all information, combining the effects of many other options.
- **-m:** Same as **who am i**, showing information about the current terminal. It's equivalent to running **who** with the **\$USER** variable.
- **-q, --count:** Display only the names and number of users currently logged on.
- **-H, --heading:** Include column headers in the output.
- **-r, --runlevel:** Show the current runlevel. This is useful in multi-user environments and for system administrators.
- **-u, --users:** Show the current login name, terminal line, login time, idle time and the exit status of a process. The idle time is particularly useful for finding out how long a terminal has been inactive.

## Date

The **date** command in Linux and Unix-like operating systems is used to display or set the system's date and time. By default, running the **date** command without any options will display the current date and time according to the system's settings.

### Basic Usage

To display the current date and time, simply type:

**date**

**Output : Tue Mar 9 12:34:56 PST 2021**

## Options

The **date** command supports several options to format the output, set the system's date and time, and more. Some commonly used options include:

- **+%FORMAT**: Allows you to specify the output format of the date/time. For example, **date +"%Y-%m-%d %H:%M:%S"** would output the date and time in the format **YYYY-MM-DD HH:MM:SS**.
- **-u, --utc, --universal**: Displays or sets the Coordinated Universal Time (UTC).

we can pass the strings like "yesterday", "monday", "last monday" "next monday", "next month", "next year," and many more.

**Consider the below commands:**

1. **date -d now**
2. **date -d yesterday**
3. **date -d tomorrow**
4. **date -d "next monday"**
5. **date -d "last monday"**

The above commands will display the dates accordingly.

## Set or Change Date in Linux

To change the system clock manually, use the **set** command. For example, to set the date and time to **5:30 PM, May 13, 2010**, type:

```
date --set="20100513 05:30"
```

## Linux Commands

### pwd

The **pwd** command in Linux and Unix-like operating systems stands for "Print Working Directory." It is used to display the full pathname of the current working directory. The current working directory is the directory in which the user is currently operating in the shell.

#### Basic Usage

To use the command, simply type:

### pwd

Upon execution, **pwd** outputs the absolute path of the current working directory to the terminal. This path is a full path from the root of the filesystem (/) to the directory you are currently in.

#### Example

If you are currently in the home directory of a user named **user**, running **pwd** might output something like: /home/user

### cd

In Linux, the **cd** command stands for "change directory." It's a shell command used to change the current working directory in the command line interface. The **cd** command is one of the most frequently used commands in Linux, as navigating between directories is a common task.

**For example, if you are working in /home/username/Documents and want to go in the Pictures subdirectory of the same directory, you can write cd Pictures.**

**If you want to go to a new directory, you can write cd followed by the absolute path of the directory – cd /home/username/Music.**

Command	Description
cd	to go to the home folder
cd..	to move one directory up
cd-	move to your previous directory

### ls

The **ls** command in Linux is used to list the contents of a directory. It's one of the most commonly used commands in Linux for navigating the filesystem and viewing the files and subdirectories

contained within a directory. By default, `ls` will list the contents of the current working directory.

### ls command syntax

#### ***ls [Options] [File]***

Some of the common option tags that you can use with the ls command:

Option	Description
<code>ls -R</code>	lists all the files in the sub-directories as well
<code>ls -S</code>	sorts and lists all the contents in the specified directory by size
<code>ls -al</code>	list the files and directories with detailed information
<code>ls -a</code>	shows the hidden files in the specified directory

### touch

The **touch** command in Linux is used primarily to create new, empty files and to change the timestamps of existing files. It's a versatile utility for file handling, particularly useful for developers and system administrators for various tasks such as file manipulation, script creation, and timestamp management.

### Creating New Files

To create a new file, simply use the **touch** command followed by the name of the file you want to create. If the file does not exist, **touch** will create a new, empty file with that name. If the file already exists, **touch** will update its access and modification times to the current time without altering the file content.

***touch newfile.txt***

### Changing File Timestamps

The primary function of **touch** is to update the timestamps on a file. Every file in Linux has three main timestamps:

- **Access time (atime)**: The last time the file was read.
- **Modification time (mtime)**: The last time the file's content was modified.
- **Change time (ctime)**: The last time the file's metadata (e.g., permissions) or content was changed.

Using **touch**, you can update the access and modification times to the current system time. This can be useful for various purposes, such as triggering processes that are dependent on file timestamps or avoiding archiving based on outdated times.

## Options

**touch** comes with several options that allow you to specify which timestamps to update and to use custom timestamps rather than the current time:

- **-a**: Change only the access time.
- **-m**: Change only the modification time.
- **-t [STAMP]**: Use a specific timestamp instead of the current time. The STAMP argument is in the format **[[CC]YY]MMDDhhmm[.ss]** where each letter represents a component of the date and time (year, month, day, hour, minute, second).

For example, to change the modification time of a file to a specific date and time, you could use:

```
touch -m -t 202307040930 newfile.txt
```

This command sets the modification time of **newfile.txt** to July 4, 2023, at 9:30 AM.

## Creating Multiple Files

**touch** can also be used to create multiple files at once by specifying more than one filename:

```
touch file1.txt file2.txt file3.txt
```

This command creates three new files named **file1.txt**, **file2.txt**, and **file3.txt** in the current directory, or updates their timestamps if they already exist.

## mv

The **mv** command in Linux is used for moving or renaming files and directories. It is a versatile command that is frequently used for organizing files, updating their locations, or changing their names. The basic syntax for the **mv** command is:

```
mv [options] source destination
```

Here are the primary ways to use the **mv** command:

## Moving Files

To move a file from one location to another, you specify the current path of the file as the source and the desired path as the destination.

```
mv /path/to/source/file.txt /path/to/destination/
```

This command moves **file.txt** from its current location to a different directory.

## Renaming Files

To rename a file, you use **mv** with the current filename as the source and the new filename as the destination, within the same directory.

***mv oldfilename.txt newfilename.txt***

This command renames **oldfilename.txt** to **newfilename.txt**.

## Moving Multiple Files

You can also move multiple files to a directory by listing each file as a source before the destination directory.

***mv file1.txt file2.txt /path/to/destination/***

This command moves **file1.txt** and **file2.txt** to the specified directory.

## rm

The **rm** (remove) command in Linux is used to delete files and directories from the filesystem. It's a powerful tool that must be used with caution, as once a file is deleted using **rm**, it typically cannot be recovered easily.

**Removing Files:** To remove a single file, use the **rm** command followed by the filename:

***rm filename.txt***

This command deletes **filename.txt** from the filesystem.

**Removing Multiple Files:** You can also delete multiple files at once by listing each file as an argument:

***rm file1.txt file2.txt file3.txt***

**Removing Directories:** To remove a directory and its contents, you need to use the **-r** (or **--recursive**) option, which tells **rm** to remove directories and their contents recursively.

***rm -r directoryname***

This command deletes **directoryname** and everything within it, including all files and subdirectories.

**To modify the command, add the following options:**

- **-i** – prompts a confirmation before deletion.
- **-f** – allows file removal without a confirmation.
- **-r** – deletes files and directories recursively.

## **mkdir**

The **mkdir** command in Linux is used to create new directories. It stands for "make directory," and it allows users to create one or multiple directories at once.

**To create a single directory, use the `mkdir` command followed by the name of the directory you want to create:**

***mkdir newdirectory***

This command creates a new directory named **newdirectory** in the current working directory.

### **Creating Multiple Directories**

You can create multiple directories at once by listing each one as an argument to the **mkdir** command: ***mkdir dir1 dir2 dir3***

Sometimes you might want to create a directory and its parent directories at the same time. You can do this using the **-p** (or **--parents**) option, which tells **mkdir** to create any necessary parent directories as well:

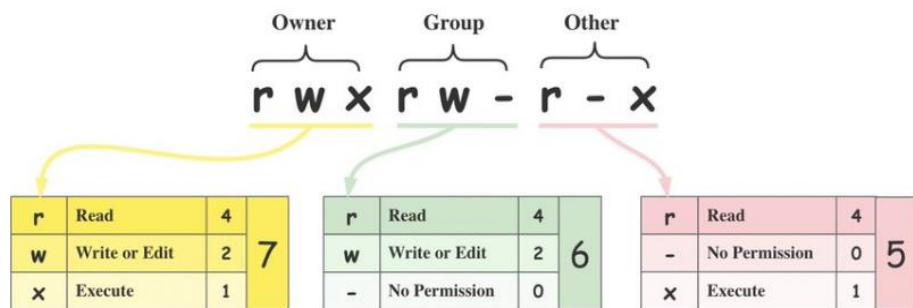
***mkdir -p parentdir/subdir/subsubdir***

This command creates the directory **subsubdir** and also creates **parentdir** and **subdir** if they don't already exist.

Here are several common **mkdir** command options:

- **-p** – creates a directory between two existing folders. For example, ***mkdir -p Music/2024/Songs*** creates a new **2024** directory.
- **-m** – sets the folder permissions. For instance, enter ***mkdir -m777 directory*** to create a directory with read, write, and execute permissions for all users.

Binary	Octal	String Representation	Permissions
000	0 (0+0+0)	---	No Permission
001	1 (0+0+1)	--x	Execute
010	2 (0+2+0)	-w-	Write
011	3 (0+2+1)	-wx	Write + Execute
100	4 (4+0+0)	r--	Read
101	5 (4+0+1)	r-x	Read + Execute
110	6 (4+2+0)	r-w-	Read + Write
111	7 (4+2+1)	rwx	Read + Write + Execute



- **-v** – prints a message for each created directory.

## rmdir

The **rmdir** command in Linux is used to remove empty directories. It stands for "remove directory," and its primary purpose is to delete directories that are no longer needed. However, it's important to note that **rmdir** will only remove a directory if it is empty. If the directory contains files or subdirectories, **rmdir** will not delete it and will instead show an error message.

**To remove a single empty directory, use the **rmdir** command followed by the name of the directory you want to delete:**

***rmdir dirname***

This command removes the directory named **dirname** if it is empty.

## Removing Multiple Directories

You can remove multiple empty directories at once by listing each one as an argument to the **rmdir** command:

***rmdir dir1 dir2 dir3***

## Removing Parent Directories

To remove a directory and its empty parent directories at the same time, you can use the **-p** (or **--parents**) option. This option tells **rmdir** to remove each directory in the specified path, working its way up the path as long as the directories are empty:

```
rmdir -p parentdir/subdir
```

This command attempts to remove **subdir** and then **parentdir** if they are both empty. If **parentdir** contains other files or directories, it will not be removed.

## tar

The **tar** command in Linux is a highly versatile tool used for creating and manipulating archive files. The name "tar" stands for "Tape Archive," reflecting its original purpose for writing data to tape drives. Over time, it has become a standard utility for file archiving and compression in Unix-like operating systems. **tar** archives multiple files and directories into a single file (often called a tarball), which can be optionally compressed using various compression algorithms. Here's how to use the **tar** command with some of its most common options:

### Creating an Archive

To create a **.tar** archive, use the **-c** option (for create), **-f** followed by the archive file name, and then list the files and directories you want to archive:

```
tar -cf archive_name.tar file1 directory1 file2
```

This command creates an archive named **archive\_name.tar** containing **file1**, **directory1**, and **file2**.

### Extracting an Archive

To extract the contents of a **.tar** archive, use the **-x** option (for extract) and **-f** followed by the archive file name:

```
tar -xf archive_name.tar
```

This command extracts the contents of **archive\_name.tar** into the current working directory.

### Viewing the Contents of an Archive

To view the contents of a **.tar** archive without extracting it, use the **-t** option (for list) and **-f** followed by the archive file name:

**`tar -tf archive_name.tar`**

This command lists the files and directories contained in **archive\_name.tar**.

## **gzip**

gzip command compresses files. Each single file is compressed into a single file. The compressed file consists of a GNU zip header and deflated data. If given a file as an argument, gzip compresses the file, adds a “.gz” suffix, and deletes the original file.

**`gzip [Options] [filenames]`**

This syntax allows users to compress a specified file. Now, let's delve into some practical examples to illustrate the usage of the gzip command.

Options	Description
<code>-f</code>	Forcefully compress a file even if a compressed version with the same name already exists.
<code>-k</code>	Compress a file and keep the original file, resulting in both the compressed and original files.
<code>-L</code>	Display the gzip license for the software.
<code>-v</code>	Display the name and percentage reduction for each file compressed or decompressed.
<code>-d</code>	Decompress a file that was compressed using the gzip command

## **Basic Compression using gzip Command in Linux**

To compress a file named “mydoc.txt,” the following command can be used:

**Example:**

**`gzip mydoc.txt`**

This command will create a compressed file of mydoc.txt named as mydoc.txt.gz and delete the original file.

## **How to decompress a gzip file in Linux?**

The basic syntax of the gzip command for decompressing a file is as follows:

**`gzip -d filename.gz`**

This command decompresses the specified gzip file, leaving the original uncompressed file intact.

### **Keeping the Original File Using gzip Command in Linux**

By default, gzip removes the original file after compression. To retain the original file, use the -k option:

***gzip -k example.txt***

This command compresses “example.txt” and keeps the original file intact.

### **Verbose Mode Using gzip Command in Linux**

To obtain more details during compression or decompression, the -v option is employed:

***gzip -v example.txt***

Verbose mode provides information such as file sizes and progress during the compression or decompression process.

### **Force Compression Using gzip Command in Linux**

In cases where the compressed file already exists, the -f option forcefully overwrites it:

***gzip -f example.txt***

This command compresses “example.txt” and overwrites any existing “example.txt.gz” file

### **Compressing Multiple Files Using gzip Command in Linux**

Gzip can compress multiple files simultaneously by providing their names as arguments:

***gzip file1.txt file2.txt file3.txt***

This command compresses “file1.txt,” “file2.txt,” and “file3.txt” individually.

## **cat**

The **cat** command in Linux is a short form for "concatenate." It is one of the most frequently used commands in Unix and Linux operating systems for various purposes. The primary function of **cat** is to read (display), combine, and copy text files. Here are some common uses and examples of how the **cat** command can be utilized:

### **Display the Contents of a File**

To display the contents of a file, simply use **cat** followed by the file name. For example:

***cat filename.txt***

This command will display the contents of **filename.txt** on the standard output (usually the terminal).

## Combine Multiple Files

`cat` can also be used to combine several files into one. For example, to concatenate the contents of `file1.txt` and `file2.txt` and display the combined content in the terminal, you can use:

```
cat file1.txt file2.txt
```

*To save the combined content into a new file, you can redirect the output to another file:*

```
cat file1.txt file2.txt > combined.txt
```

## Append Content to a File

You can use `cat` to append content to the end of an existing file. For example, to append `file2.txt` to `file1.txt`, you can use:

```
cat file2.txt >> file1.txt
```

This command appends the contents of `file2.txt` to `file1.txt` without overwriting the original content of `file1.txt`.

## Create a New File

`cat` can be used to create a new file by redirecting its output. For example:

```
cat > newfile.txt
```

After running this command, the terminal waits for you to input text. You can type the content of the new file, and once done, press **Ctrl+D** to save and exit.

## Display Line Numbers

To display the contents of a file with line numbers, you can use `cat` with the `-n` option:

```
cat -n filename.txt
```

## Process Concepts

### Process

A process is a program in execution. For example, when we write a program in C or C++ and compile it, the compiler creates binary code. The original code and binary code are both programs. When we actually run the binary code, it becomes a process.

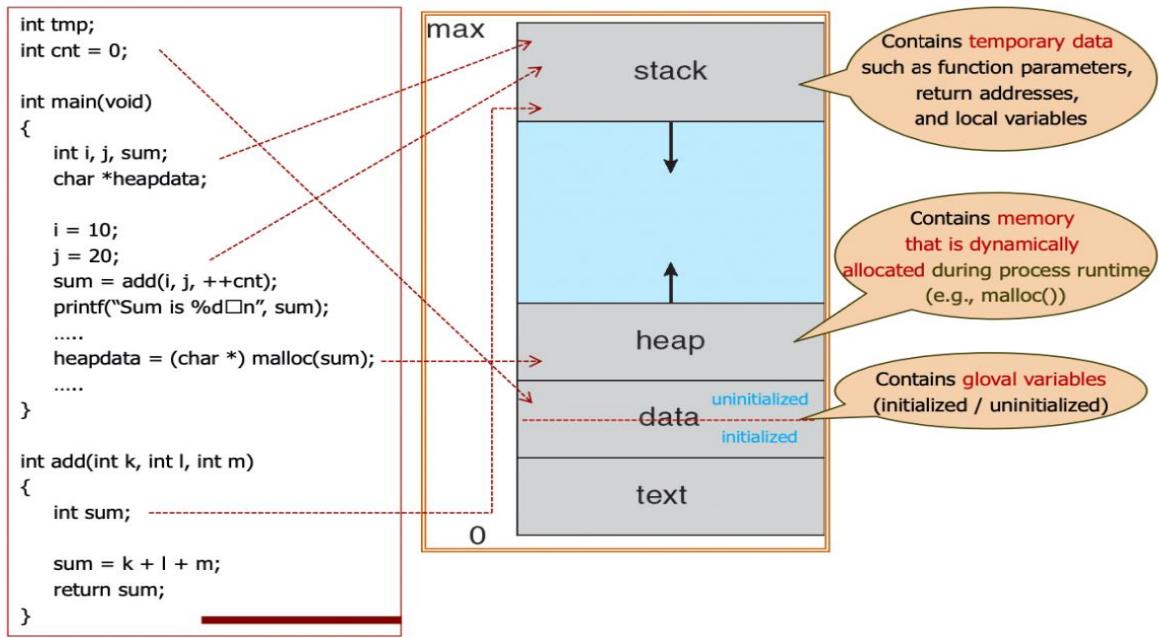
A process is an ‘active’ entity instead of a program, which is considered a ‘passive’ entity. A single program can create many processes when run multiple times; for example, when we open a .exe or binary file multiple times, multiple instances begin (multiple processes are created).

### Process vs Program

- A process consists of instruction execution in machine code, but a program consists of instruction in any programming language.
- A process is a dynamic object, but a program is a static object.
- Process resides in main memory, but program resides in secondary memory.
- The period for the process is limited, but the period for the program is unlimited.
- A process is an active entity, and a program is a passive entity.

**Process memory** is divided into four sections for efficient working:

- The **Text section** is made up of the compiled program code, read in from non-volatile storage when the program is launched.
- The **Data section** is made up of the global and static variables, allocated and initialized prior to executing the main.
- The **Heap** is used for the dynamic memory allocation and is managed via calls to new, delete, malloc, free, etc.
- The **Stack** is used for local variables. Space on the stack is reserved for local variables when they are declared.



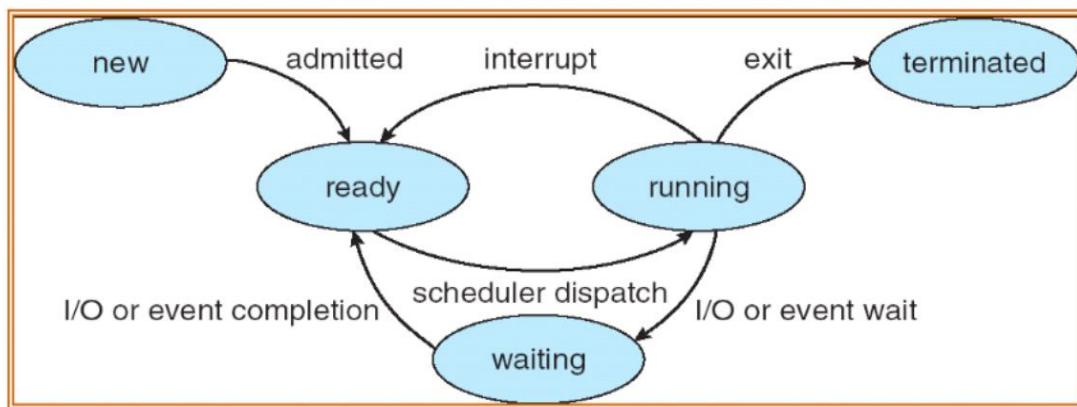
## Process States

Process states in operating system are a way to manage resources efficiently by keeping track of the current state of each process. There are several process states in operating system, they are:

- **New State:** When a process is first created or is initialized by the operating system, it is in the new state. In this state, the process is being prepared to enter the ready state.
- **Ready State:** When a process is ready to execute, it is in the ready state. In this state, the process is waiting for the CPU to be allocated to it so that it can start executing its instructions. A process can remain in the ready state for an indeterminate period, waiting for the CPU to become available.
- **Running State:** When the CPU is allocated to a process, it enters the running state. In this state, the process executes its instructions and uses system resources such as memory, CPU, and I/O devices. Only one process can be in the running state at a time, and the operating system determines which process gets access to the CPU using scheduling algorithms.
- **Blocked State:** Sometimes, a process needs to wait for a particular event, such as user input or data from a disk drive. In such cases, the process enters the blocked state. In this state, the process is not using the CPU, but it is not ready to run either. The process remains in the blocked state until the event it is waiting for occurs.

- **Terminated State:** The terminated state is reached when a process completes its execution or terminates by the operating system. In this state, the process no longer uses any system resources, and its memory space is deallocated.

Now let us see the state diagram of these process states –



**Step 1** – Whenever a new process is created, it is admitted into ready state.

**Step 2** – If no other process is present at running state, it is dispatched to running based on scheduler dispatcher.

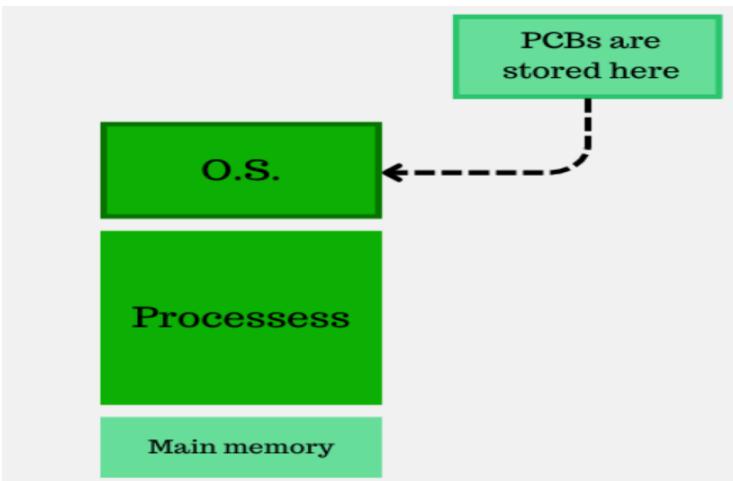
**Step 3** – If any higher priority process is ready, the uncompleted process will be sent to the waiting state from the running state.

**Step 4** – Whenever I/O or event is completed the process will send back to ready state based on the **interrupt signal** given by the running state.

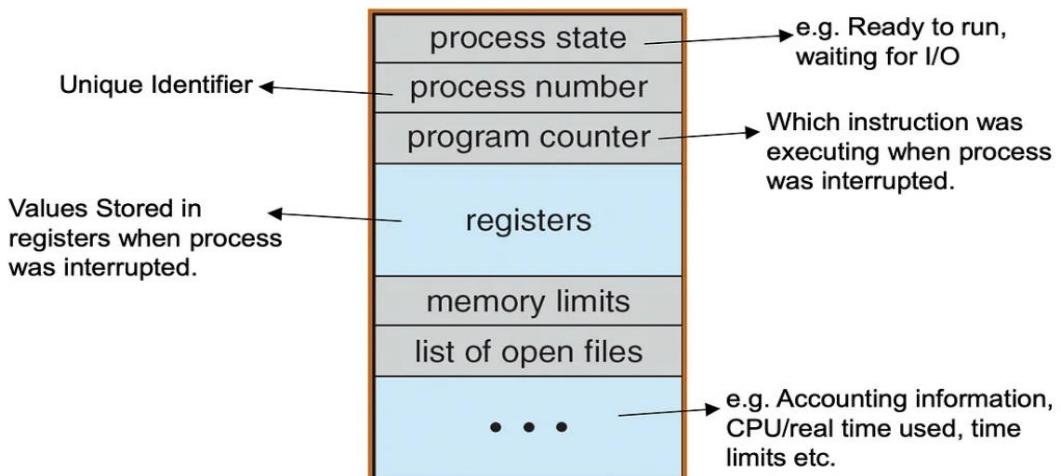
**Step 5** – Whenever the execution of a process is completed in running state, it will exit to terminate state, which is the completion of process.

### Process Control Block

Process Control Block is a data structure that contains information of the process related to it. The process control block is also known as a task control block, entry of the process table, etc. Each time a process is created, a unique PCB is created and is removed when the process is completed. The PCB is created with the aim of helping the OS to manage the enormous amounts of tasks that are being carried out in the system.



### **PCB Contains the following attributes**



#### **Process State**

Throughout its existence, each process goes through various phases. The present state of the process is defined by the process state.

#### **Process Number**

When a new process is created by the user, the operating system assigns a unique ID i.e. a process-ID to that process. This ID helps the process to be distinguished from other processes existing in the system. The operating system has a limit on the maximum number of processes it is capable of dealing with, let's say the OS can handle most N processes at a time.

So, process-ID will get the values from 0 to N-1.

#### **Program Counter**

The address of the next instruction to be executed is specified by the program counter. The address of the program's first instruction is used to initialize the program counter before it is executed.

The value of the program counter is incremented automatically to refer to the next instruction when each instruction is executed. This process continues till the program ends.

## Registers

The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.

## List of open files

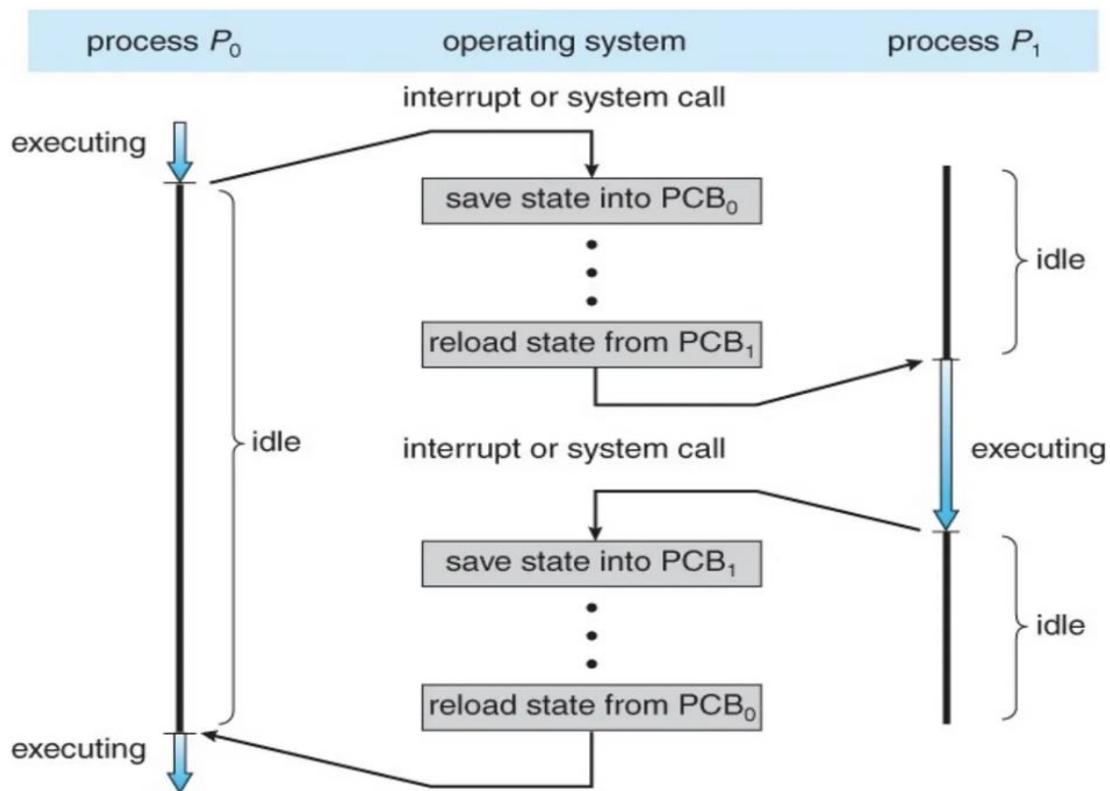
This contains information about those files that are used by that process. This helps the operating system close all the opened files at the termination state of the process.

## Context Switching

A context switching is a mechanism that involves switching of the CPU from one process or task to another. While performing switching the operating system saves the state of a running process so it can be resumed later, and then loads the state of another process. A context switching allows a single CPU to handle multiple process requests simultaneously without the need for any additional processors.

Following are the reasons that describe the need for context switching in the Operating system.

1. If a high priority process falls into the ready queue, the currently running process will be shut down or stopped by a high priority process to complete its tasks in the system.
2. If any running process requires I/O resources in the system, the current process will be switched by another process to use the CPUs. And when the I/O requirement is met, the old process goes into a ready state to wait for its execution in the CPU. Context switching stores the state of the process to resume its tasks in an operating system. Otherwise, the process needs to restart its execution from the initial level.
3. If any interrupts occur while running a process in the operating system, the process status is saved as registers using context switching. After resolving the interrupts, the process switches from a wait state to a ready state to resume its execution at the same point later, where the operating system interrupted occurs.



**The following steps describe the basic sequence of events when moving between processes:**

1. The CPU executes Process 0.
2. A triggering event occurs, such as an interrupt or system call.
3. The system pauses Process 0 and saves its state (context) to PCB 0, the process control block that was created for that process.
4. The system selects Process 1 from the queue and loads the process's state from PCB 1.
5. The CPU executes Process 1, picking up from where it left off (if the process had already been running).
6. When the next triggering event occurs, the system pauses Process 1 and saves its state to PCB 1.
7. The Process 0 state is reloaded, and the CPU executes the process, once again picking up from where it left off. Process 1 remains in an idle state until it is called again.

#### ***Advantage of Context Switching***

Context switching is used to achieve multitasking . Multitasking gives an illusion to the users that more than one process are being executed at the same time. But in reality, only one task is being executed at a particular instant of time by a processor. Here, the context

switching is so fast that the user feels that the CPU is executing more than one task at the same time.

### ***The disadvantage of Context Switching***

The disadvantage of context switching is that it requires some time for context switching i.e. the context switching time. Time is required to save the context of one process that is in the running state and then getting the context of another process that is about to come in the running state. During that time, there is no useful work done by the CPU from the user perspective. So, context switching is pure overhead in this condition.

## Process Operations

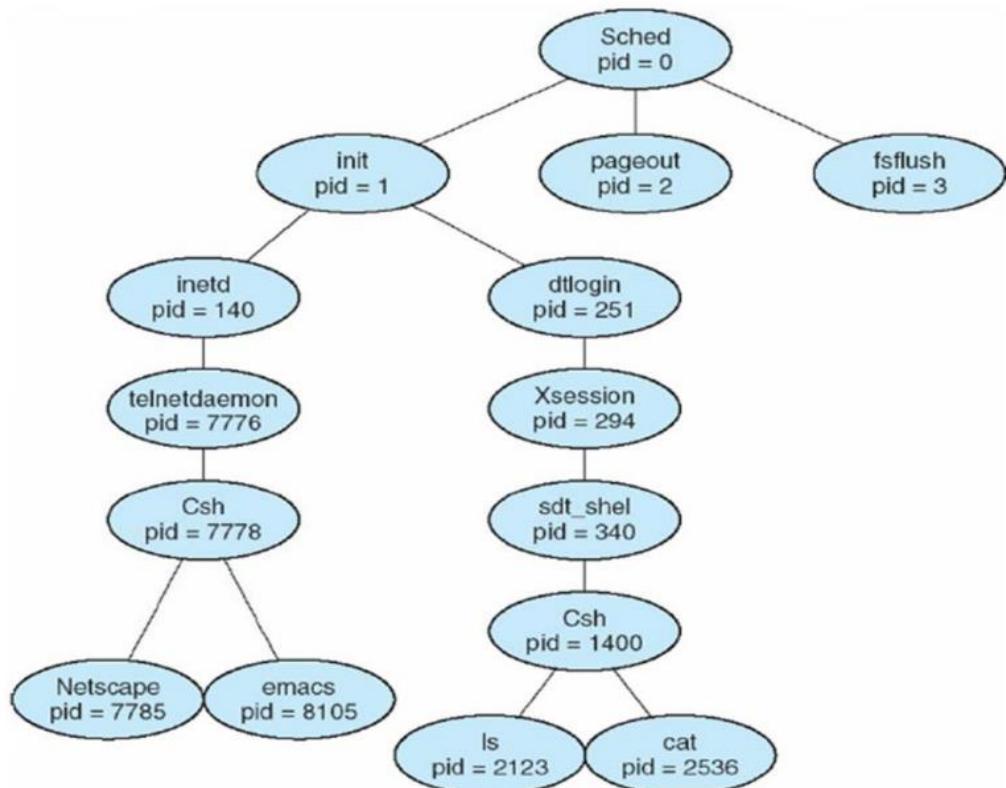
The operations of process carried out by an operating system are primarily of two types:

1. Process creation
2. Process termination

### Process creation

Process creation in operating systems (OS) is a fundamental operation that involves the initiation of a new process. The creating process is called the **parent** while the created process is called the **child**. Each child process may in turn create new child process. Every process in the system is identified with a process identifier(**PID**) which is unique for each process. A process uses a System call to create a child process.

Figure Below illustrates a typical process tree for the Solaris operating system, showing the name of each process (daemon) and its pid.



In Linux and other Unix-like operating systems, a daemon is a background process that runs independently of any interactive user session. Daemons typically start at system boot and run continuously until the system is shut down. They perform various system-level tasks, such as

managing hardware devices, handling network requests, scheduling jobs, or any other kind of service that needs to run in the background without direct user intervention

Daemon	Operation
<b>init</b>	The <b>init</b> process is the first process started by the Linux/Unix kernel and holds the process ID (PID) 1.
<b>pageout</b>	The <b>pageout</b> daemon is responsible for managing virtual memory,
<b>fsflush</b>	The <b>fsflush</b> process is a background daemon responsible for synchronizing cached data to disk.
<b>inetd</b>	<b>inetd</b> , short for "internet super-server", is a daemon that listens on designated ports for incoming connections and then launches the appropriate program. <b>inetd</b> is responsible for networking services such as telnet and ftp
<b>dtlogin</b>	<b>Desktop login- dtlogin</b> is the display manager component responsible for managing user logins to the graphical environment.

In Solaris, the process at the top of the tree is the sched process, with pid of 0. The sched process creates several children processes-including pageout and fsflush. The sched process also creates the init process, which serves as the root parent process for all user processes. we see two children of init- inetd and dtlogin. When a user logs in, dtlogin creates an X-windows session (Xsession), which in turns creates the sdt\_shel process. (**sdt\_shel**, part of the CDE (Common Desktop Environment), is a graphical user interface tool designed to provide users with an easy way to access various system functions and applications). Below sdt\_shel, a user's command-line shell-the C-shell or csh-is created. In this command line interface, the user can then invoke various child processes, such as the ls and cat commands. We also see a csh process with pid of 7778 representing a user who has logged onto the system using telnet. This user has started the Netscape browser (pid of 7785) and the emacs editor (pid of 8105). (Emacs is a text editor which provides a robust platform for executing complex editing tasks, writing code in various programming languages, managing files, reading emails, browsing the web, and even playing games.)

### Resource Sharing Between the process

- In general, a process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. When a process creates a subprocess, that subprocess may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process.

- The parent may have to partition its resources among its children, or it may be able to share some resources (such as Memory or files) among several of its children.
- Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many subprocesses.
- In addition to the various physical and logical resources that a process obtains when it is created, initialization data (input) may be passed along by the parent process to the child process.
- For example, consider a process whose function is to display the contents of a file-say, img.jpg-on the screen of a terminal. When it is created, it will get, as an input from its parent process, the name of the file img.jpg, and it will use that file name, open the file, and write the contents out.
- It may also get the name of the output device. Some operating systems pass resources to child processes. On such a system, the new process may get two open files, img.jpg and the terminal device, and may simply transfer the datum between the two.

## **Process Execution**

When a process creates a new process, two possibilities exist in terms of execution:

- The parent continues to execute concurrently with its children.
- The parent waits until some or all of its children have terminated.

## **Process AddressSpace**

There are also two possibilities in terms of the address space of the new process:

- The child process is a duplicate of the parent process (it has the same program and data as the parent). (Ex:fork() System Call)
- The child process has a new program loaded into it.(Ex: exec() System call)

## **Process termination**

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit () system call. At that point, the process may return a status value (typically an integer) to its parent process (via the wait() system call). All the resources of the process-including physical and virtual memory, open files, and I/O buffers-are deallocated by the operating system.
- Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call (for example, TerminateProcess () in Win32). Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, users could arbitrarily kill each other's jobs. Note that a parent needs to know the identities of its children.

Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.

➤ A parent may terminate the execution of one of its children for a variety of reasons, such as these:

- The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates. If a Parent process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as **cascading termination**.

## System calls - Process Creation & termination in Linux

### 1.fork() System call

In Linux OS a new process is created by the fork () system call.

- The “fork()” system call is used to create a new process, known as the child process, from an existing process, known as the parent process. When “fork()” is called, the operating system creates a new process by duplicating the entire address space (memory), file descriptors, and other attributes of the parent process.
- The child process is an exact copy of the parent process at the time of the “fork()” call. However, the child process has its own process ID (PID) and is assigned a unique PID by the operating system. Both the parent and child processes then continue execution from the point immediately after the “fork()” call.
- The “fork()” system call returns different values to the parent and child processes. In the parent process, the return value is the PID of the child process, while in the child process, the return value is 0. This allows the parent and child processes to differentiate themselves and execute different sections of code if needed.
- **Return value of fork()** On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.

*The fork() system call is often used to implement process spawning, where a parent process creates multiple child processes to perform tasks concurrently or in a parallel manner.*

### 2.exec() System Call

- The “exec()” system call is used to replace the current process with a new program or executable. It loads a new program into the current process’s memory, overwriting the existing program’s code, data, and stack.
- When “exec()” is called, the operating system performs the following steps:
  - a. The current process’s memory is cleared, removing the old program’s instructions and data.
  - b. The new program is loaded into memory.
  - c. The program’s entry point is set as the starting point of execution.
  - d. The new program’s code begins execution.
- The “exec()” system call is often used after a “fork()” call in order to replace the child process’s code with a different program. This allows the child process to execute a different program while preserving the parent process’s execution.

*Together, the “fork()” and “exec()” system calls enable processes to create child processes and replace their own execution with different programs, facilitating process creation, concurrency, and program execution in operating systems.*

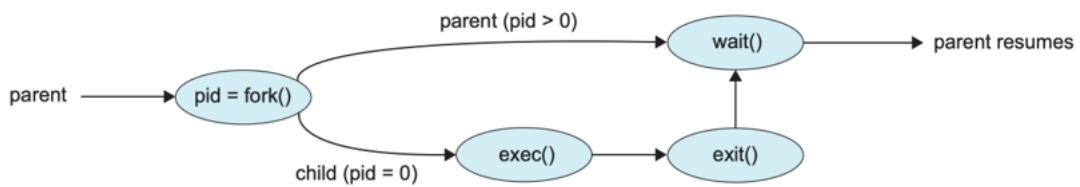
### 3.Wait() System call

- In some systems, a process may have to wait for another process to complete its execution before proceeding. When a parent process makes a child process, the parent process execution is suspended until the child process is finished. The **wait()** system call is used to suspend the parent process. Once the child process has completed its execution, control is returned to the parent process.
- **wait()** system call takes only one parameter which stores the status information of the process. Pass NULL as the value if you do not want to know the exit status of the child process and are simply concerned with making the parent wait for the child.
- On success, **wait** returns the PID of the terminated child process while on failure it returns -1. (  
`pid_t wait(int *wstatus)`)

### 4.exit() System call

The **exit()** is a system call that is used to terminate the process. After the use of **exit()** system call, all the resources used in the process are retrieved by the operating system and then terminate the process.

*Lets depict all the system calls in the form of a process transition diagram.*



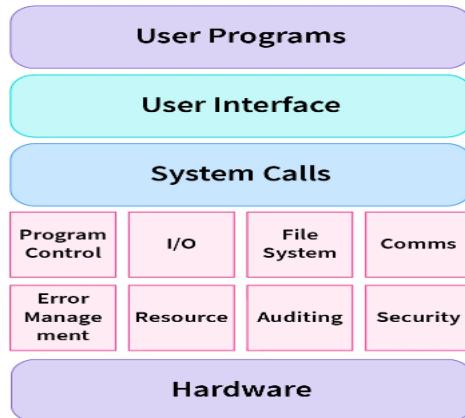
A process begins its life when it is created. A process goes through different states before it gets terminated.

- The first state that any process goes through is the creation of itself. Process creation happens through the use of fork() system call, (*A system call is a function that a user program uses to ask the operating system for a particular service .System calls serve as the interface between an operating system and a process*) which creates a new process(child process) by duplicating an existing one(parent process). The process that calls fork() is the parent, whereas the new process is the child.
- In most cases, we may want to execute a different program in child process than the parent. The exec() family of function calls creates a new address space and loads a new program into it.
- Finally, a process exits or terminates using the exit() system call. This function frees all the resources held by the process(except for pcb).
- A parent process can enquire about the status of a terminated child using wait() system call. When the parent process uses wait() system call, the parent process is blocked till the child on which it is waiting terminates.

UNIX SYSTEM CALLS	DESCRIPTION	WINDOWS API CALLS	DESCRIPTION
<b>Process Control</b>			
<code>fork()</code>	Create a new process.	<code>CreateProcess()</code>	Create a new process.
<code>exit()</code>	Terminate the current process.	<code>ExitProcess()</code>	Terminate the current process.
<code>wait()</code>	Make a process wait until its child processes terminate.	<code>WaitForSingleObject()</code>	Wait for a process or thread to terminate.
<code>exec()</code>	Execute a new program in a process.	<code>CreateProcess()</code> or <code>ShellExecute()</code>	Execute a new program in a new process.
<code>getpid()</code>	Get the unique process ID.	<code>GetCurrentProcessId()</code>	Get the unique process ID.

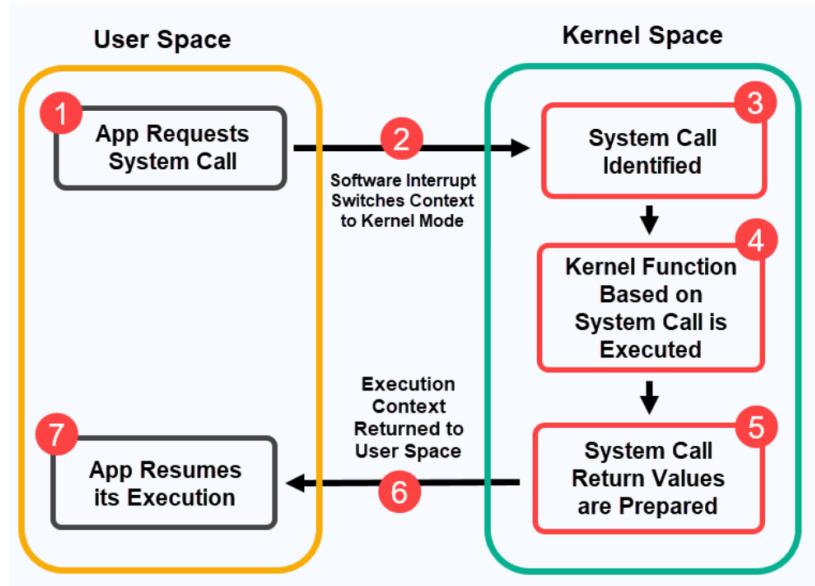
## System call interface for process management

A system call is a function that a user program uses to ask the operating system for a particular service which cannot be carried out by normal function. System calls provide the interface between the process and the operating system.



## Working of System calls

This high-level overview explains how system calls work:



- 1. System Call Request.** The application requests a system call by invoking its corresponding function. For instance, the program might use the `read()` function to read data from a file.
- 2. Context Switch to Kernel Space.** A software interrupt or special instruction is used to trigger a context switch and transition from the user mode to the kernel mode.
- 3. System Call Identified.** The system uses an index to identify the system call and address the corresponding kernel function.

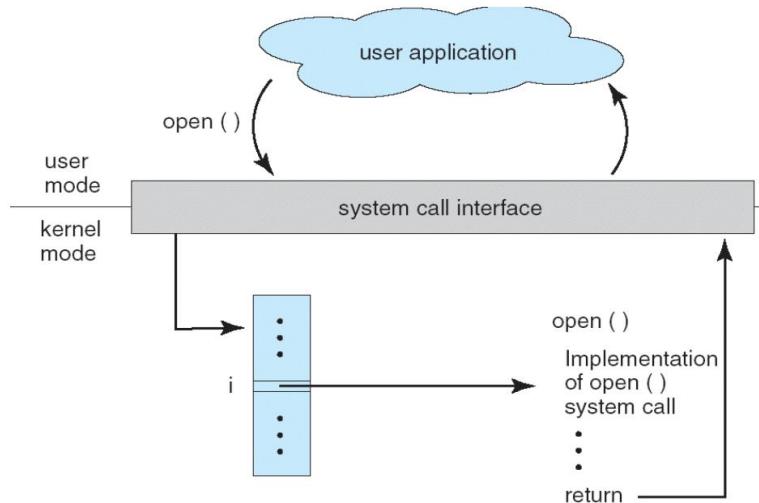
**4. Kernel Function Executed.** The kernel function corresponding to the system call is executed. For example, reading data from a file.

**5. System Prepares Return Values.** After the kernel function completes its operation, any return values or results are prepared for the user application.

**6. Context Switch to User Space.** The execution context is switched back from kernel mode to user mode.

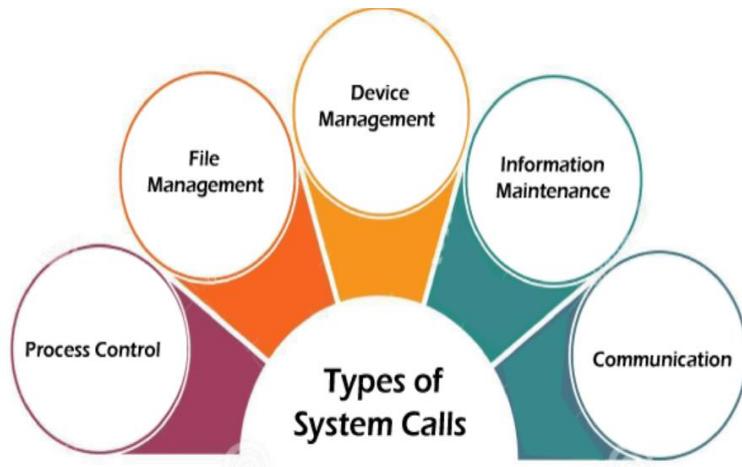
**7. Resume Application.** The application resumes its execution from where it left off, now with the results or effects of the system call.

*Example*



**System calls are divided into 5 categories mainly :**

- Process Control
- File Management
- Device Management
- Information Maintenance
- Communication



## Process Management System Calls

System calls in Linux	Description
fork	Create a new process.
vfork	Replace the current process with a new one.
exit	Terminate a process
wait	It makes a parent process stop its execution till the termination of the child process.
waitpid	It makes a parent process stop its execution till the termination of the specified child process.(Multiple child process)
exec	It loads a new program in child process

### fork()

It is the primary method of process creation on Unix-like operating systems. This function creates a new copy called the child out of the original process, that is called the parent. When the parent process closes or crashes for some reason, it also kills the child process.

#### Syntax:

pid=fork();

- The operating system is using a unique id for every process to keep track of all processes. And for that, fork() doesn't take any parameter and return an int value as following:
- **Zero:** if it is the child process (the process created). **Positive value:** if it is the parent process.
- The difference is that, in the parent process, fork() returns a value which represents the **process ID** of the child process. But in the child process, **fork()** returns the value 0.
- **Negative value:** if an error occurred.

### ***Sample Program***

```
/*When working with fork(), <sys/types.h> can be used for type pid_t for processes ID's as pid_t is defined in <sys/types.h>.
```

```
The header file <unistd.h> is where fork() is defined so you have to include it to your program to use fork().*/
```

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    /* fork a process */
    pid_t p;
    p = fork();
    /* the child and parent will execute every line of code after the fork (each separately) */
    if(p == 0)
        printf("Hello child pid=%d\n", getpid());
    else if(p > 0)
        printf("Hello parent pid=%d\n", getpid());
    else
        printf("Error: fork() failed\n");
    return 0;
}
```

### ***The output will be:***

```
Hello parent pid=1601
```

```
Hello child pid=1602
```

### ***vfork()***

The **vfork()** is another system call that is utilized to make a new process. The child process is the new process formed by the **vfork()** system call, while the parent process is the process that uses the **vfork()** system call. The vfork() system call doesn't create separate address spaces for both parent and child processes.

- Because the child and parent processes share the same address space, the child process halts the parent process's execution until the child process completes its execution.
- If one of the processes changes the code, it is visible to the other process transferring the same pages. Assume that the parent process modifies the code; it will be reflected in the child process code.

<b>fork()</b>	<b>vfork()</b>
Child process and parent process has separate address spaces.	Child process and parent process shares the same address space.
Parent and child process execute simultaneously.	Parent process remains suspended till child process completes its execution.
If the child process alters any page in the address space, it is invisible to the parent process as the address space are separate.	If child process alters any page in the address space, it is visible to the parent process as they share the same address space.

### **Syntax:**

```
pid=vfork();
```

### **Sample Program**

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    /* fork a process */
    pid_t p;
    p = vfork();
    /* the child and parent will execute every line of code after the fork (each separately) */
    if(p == 0)
    {
        // Child process
    }
}
```

```

        printf("This is the child process. PID: %d\n", getpid());
        printf("Child process is exiting with exit() \n");
        exit(0);
    }

else if(p > 0)
{
    // Parent process might wait for child process to terminate
    printf("This is the parent process. PID: %d\n", getpid());
}

else
    printf("Error: fork() failed\n");

return 0;
}

```

### ***output***

This is the child process. PID: 91  
 Child process is exiting with exit()  
 This is the parent process. PID: 90

## **Wait()**

This system call is used in processes that have a parent-child relationship. It makes a parent process stop its execution till the termination of the child process. The program creates a child process via the fork() system call and then calls the wait() system call to wait for the child process to finish its execution.

### ***Syntax***

Below, we can see the syntax for the wait() system call. To use this in our C programs, we will have to include the Standard C library sys/wait.h.

pid\_t wait(int \*status);

### ***Parameters and return value***

The system call takes one argument named status. This argument represents a pointer to an integer that will store the exit status of the terminated child program. We can even replace status with NULL parameter.

When the system call completes, it can return the following.

- **Process ID:** The process ID of the child process that is terminated, which has the object type pid\_t.

- **Error value:** If there is any error during system call execution, it will return -1, which can be used for error handling.

### *Sample program*

```
#include <stdio.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    pid_t p, childpid;
    p = fork();
    // code for the child process
    if(p== 0){
        printf("Child: I am running!!\n\n");
        printf("Child: I have PID: %d\n\n", getpid());
        exit();
    }
    // code for the parent process
    else{
        // print parent running message
        printf("Parent: I am running and waiting for child to finish!!\n\n");
        // call wait system call
        childpid = wait(NULL);
        // print the details of the child process
        printf("Parent: Child finished execution!, It had the PID: %d,\n", childpid);
    }
    return 0;
}
```

### *Output:*

Parent: I am running and waiting for child to finish!!

Child: I am running!!

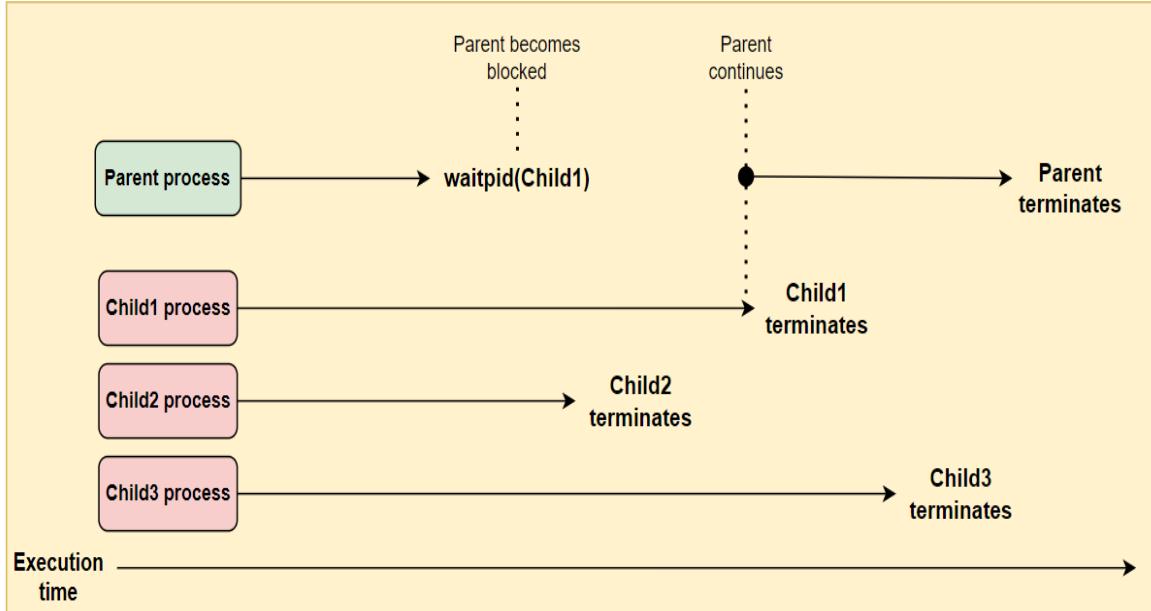
Child: I have PID: 102

Parent: Child finished execution!, It had the PID: 102

## **waitpid()**

This system call waits for a specific process to finish its execution. This system call can be accessed using our C programs' library sys/wait.h.

Let's understand how the `waitpid()` function works with the help of a diagram.



The diagram shows that we have a parent process with 3 child processes, namely, Child1, Child2, and Child3. If we want the parent to wait for a specific child, we could use the `waitpid()` function.

In the diagram above, we made the parent process wait for Child1 to finish its execution using the `waitpid()` function. When the parent process is called, the `waitpid()` function gets blocked by the operating system and can't continue its execution till the specified child process Child1 is terminated.

If we were to replace the process in the `waitpid()` function with Child3, then the parent would only continue when Child3 had terminated.

### **Syntax**

```
pid_t waitpid(pid_t pid, int *status_ptr, int options);
```

Let's discuss the three arguments that we provide to the system call.

- **pid:** Here, we provide the process ID of the process we want to wait for. If the provided pid is 0, it will wait for any arbitrary child to finish.
- **status\_ptr:** This is an integer pointer used to access the child's exit value. If we want to ignore the exit value, we can use NULL here.
- **options:** Here, we can add additional flags to modify the function's behavior. The various flags are discussed below:
  - **WCONTINUED:** It is used to report the status of any child process that has been terminated and those that have resumed their execution after being stopped.

- WNOHANG: It is used when we want to retrieve the status information immediately when the system call is executed. If the status information is not available, it returns an error.
- WUNTRACED: It is used to report any child process that has stopped or terminated.

#### Return value

The system call will return the process ID of the child process that was terminated. If there is any error while waiting for the child process via the waitpid() system call, it will return -1, which corresponds to an error.

#### *Sample Program*

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
    pid_t pids[2], wpid;
    // Fork first child process
    pids[0] = fork();
    if (pids[0] == 0) {
        // First child process
        printf("First child process: PID = %d\n", getpid());
        printf("First child process exiting\n");
        exit(0); // First child exits
    }
    // Fork second child process
    pids[1] = fork();
    if (pids[1] == 0)
    {
        // Second child process
        printf("Second child process: PID = %d\n", getpid());
        printf("Second child process exiting\n");
        exit(0); // Second child exits
    }
    if(pid[0]>0 && pid[1]>0)
```

```

{
// Parent process: wait specifically for the first child
wpid = waitpid(pids[1], NULL,0);
printf("Parent: Proceeding after the second child with pid=%d has finished.\n",pid[1]);
}
return 0;
}

```

### ***Output***

First child process: PID = 119

First child process exiting

Second child process: PID = 120

Second child process exiting

Parent: Proceeding after the second child with pid =120 has finished

### **exec()**

The “exec()” system call is used to replace the current process with a new program or executable. It loads a new program into the current process’s memory, overwriting the existing program’s code, data, and stack.

When “exec()” is called, the operating system performs the following steps:

- The current process’s memory is cleared, removing the old program’s instructions and data.
- The new program is loaded into memory.
- The program’s entry point is set as the starting point of execution.
- The new program’s code begins execution.

The “exec()” system call is often used after a “fork()” call in order to replace the child process’s code with a different program. This allows the child process to execute a different program while preserving the parent process’s execution.

Exec system call is a collection of functions and in C programming language, the standard names for these functions are as follows:

**(l- represents list, v represents vector)**

**execl():** Executes a program specified by a pathname, and arguments are passed as a variable-length list of strings terminated by a NULL pointer.

**Syntax:** int execl(const char \*path, const char \*arg, ..., NULL);

**Parameters:**

- **path:** Specifies the path to the executable file which the process will run.
- **arg:** Represents the list of arguments to be passed to the executable. The first argument typically is the name of the program being executed. The list of arguments must be terminated by **NULL** to indicate the end of the argument list.

**Return Value:**

- On success, **exec()** does not return; the new program image takes over the process.
- If an error occurs, it returns **-1**, and **errno** is set to indicate the error.

**execv()**: Executes a program specified by a pathname, and arguments are passed as an array of strings terminated by a **NULL** pointer.

**Syntax:** int execv(const char \*path, char \*const argv[]);

**Parameters:**

- **path:** Specifies the path to the executable file which the process will run.
- **argv:** An array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should be the name of the executed program. The array of pointers must be terminated by a **NULL** pointer.

**Return Value:**

- On success, **execv()** does not return because the current process image is replaced by the new program image.
- On failure, it returns **-1**, and **errno** is set to indicate the error.

**execl()**: Similar to **exec()**, but the program is searched for in the directories specified by the **PATH** environment variable.

**Syntax :** int execl(const char \*file, const char \*arg, ..., NULL);

**Parameters:**

- **file:** The name of the executable file to run. If this name contains a slash (/), then **execl()** will treat it as a path and will not search the **PATH** environment variable.
- **arg:** Represents the list of arguments to be passed to the executable. The first argument, by convention, should be the name of the program being executed. The list of arguments must be terminated by **NULL** to indicate the end of the argument list.

**Return Value:**

- On success, **execl()** does not return; the new program image takes over the process.
- On failure, it returns **-1**, and **errno** is set to indicate the error.

**execvp()**: Similar to **execv()**, but the program is searched for in the directories specified by the **PATH** environment variable.

**Syntax:** int execvp(const char \*file, char \*const argv[]);

**Parameters:**

- **file**: The name of the executable file to run. If this name contains a slash (/), then **execvp()** will treat it as a path and will not search the **PATH** environment variable.
- **argv**: An array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should be the name of the executed program. The array of pointers must be terminated by a **NULL** pointer to indicate the end of the argument list.

**Return Value:**

- On success, **execvp()** does not return because the current process image is replaced by the new program image.
- On failure, it returns **-1**, and **errno** is set to indicate the error.

**execle()**: Similar to **execl()**, but allows specifying the environment of the executed program as an array of strings.

**Syntax:** int execle(const char \*path, const char \*arg, ..., char \*const envp[]);

**Parameters:**

- **path**: Specifies the path to the executable file which the process will run.
- **arg**: Represents the list of arguments to be passed to the executable. The first argument, by convention, should be the name of the program being executed. The list of arguments must be terminated by **NULL** to indicate the end of the argument list.
- **...**: A variable number of arguments representing the arguments to be passed to the executable, terminated by a **NULL** pointer.
- **envp[]**: An array of strings, conventionally of the form **key=value**, which are passed as the environment of the new program. The array must be terminated by a **NULL** pointer.

**Return Value:**

- On success, **execle()** does not return because the current process image is replaced by the new program image.
- On failure, it returns **-1**, and **errno** is set to indicate the error.

**execve()**: The most general form, which executes a program specified by a pathname, takes an array of arguments and an array of environment variables.

**Syntax**: int execve(const char \*pathname, char \*const argv[], char \*const envp[]);

**Parameters:**

- **pathname**: Specifies the file path of the executable file which the process will run. This file must be a binary executable or a script with a shebang (#!) line.
- **argv[]**: An array of pointers to null-terminated strings that represent the argument list to be passed to the new program. The first argument, by convention, should be the name of the executed program. The array of pointers must be terminated by a NULL pointer.
- **envp[]**: An array of strings, conventionally of the form **key=value**, which are passed as the environment of the new program. The array must be terminated by a NULL pointer.

**Return Value:**

- On success, **execve()** does not return because the current process image is replaced by the new program image.
- On failure, it returns **-1**, and **errno** is set to indicate the error.

**Sample Program**

```
include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid = fork();
    if (pid == 0) {
        // Child process
        printf("Child process (PID %d): executing '/bin/ls'\n", getpid());
        // Replace the child process with '/bin/ls'
        // execl() takes the path to the program, the name of the program,
        // and any command-line arguments, ending with a NULL pointer.
        execl("/bin/ls", "ls", NULL);
    }
    else
    {
        // Parent process
        printf("Parent process (PID %d): waiting for child process\n", getpid());
    }
}
```

```
    }  
}  
}
```

***output***

Parent process (PID 142): waiting for child process

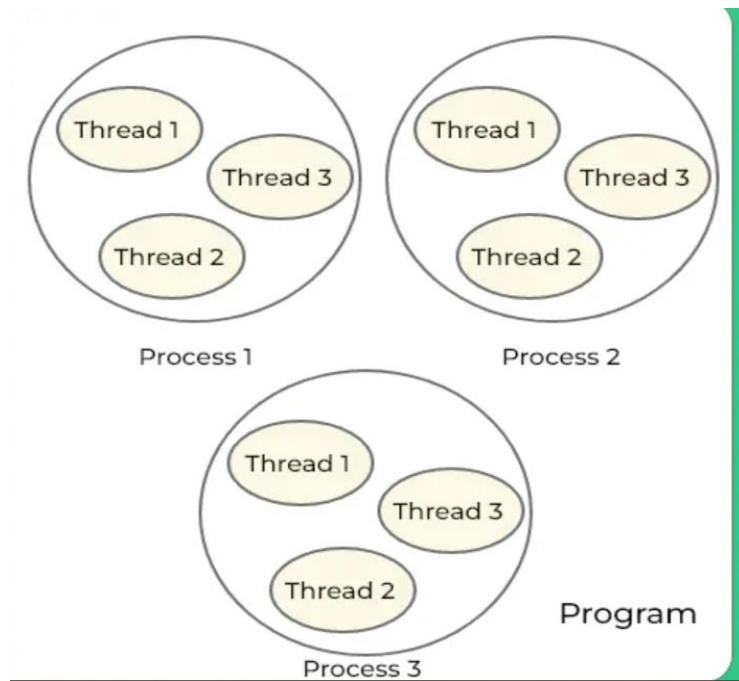
Child process (PID 143): executing '/bin/ls'

a.out exec.c fork.c vfork.c wait.c waitpid waitpid.c

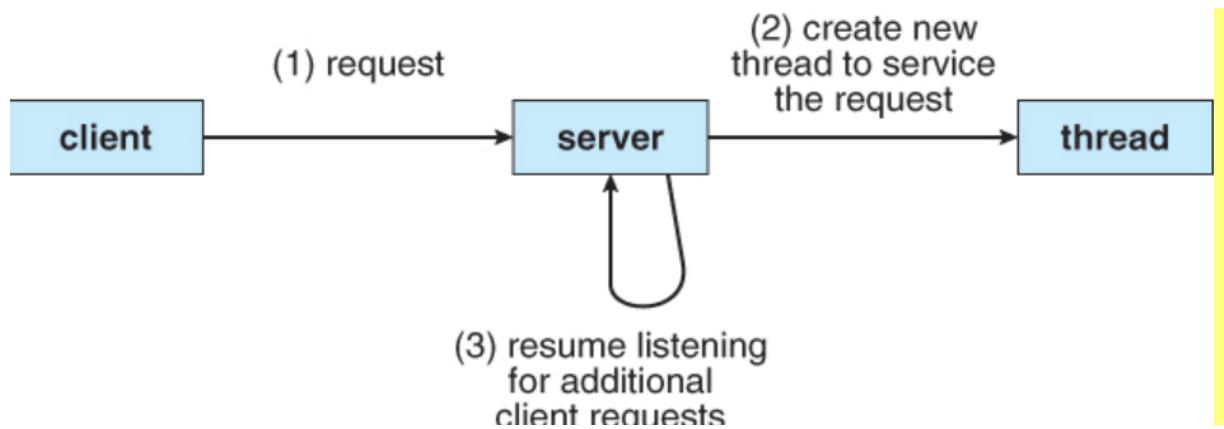
## Introduction to Threads

**Program, Process and Threads** are three basic concepts of the operating systems.

- Program is an executable file containing the set of instructions written to perform a specific job on your computer. Programs are not stored on the primary memory in your computer. They are stored on a disk or a secondary memory on your computer. They are read into the primary memory and executed by the kernel. A program is sometimes referred as **passive entity** as it resides on a secondary memory.
- Process is an executing instance of a program. A process is sometimes referred as **active entity** as it resides on the primary memory and leaves the memory if the system is rebooted. Several
- Thread is the smallest executable unit of a process. A thread is often referred to as a lightweight process due to its ability to run sequences of instructions independently while sharing the same memory space and resources of a process. A process can have multiple threads. Each thread will have their own task and own path of execution in a process. Threads are popularly used to improve the application through **parallelism**. Actually only one thread is executed at a time by the CPU, but the **CPU switches rapidly** between the threads to give an illusion that the threads are running parallelly.



- *For example, in a browser, multiple tabs can be different threads or While the movie plays on the device, various threads control the audio and video in the background.*
- *Another example is a web server - Multiple threads allow for multiple requests to be satisfied simultaneously, without having to service requests sequentially or to fork off separate processes for every incoming request.*



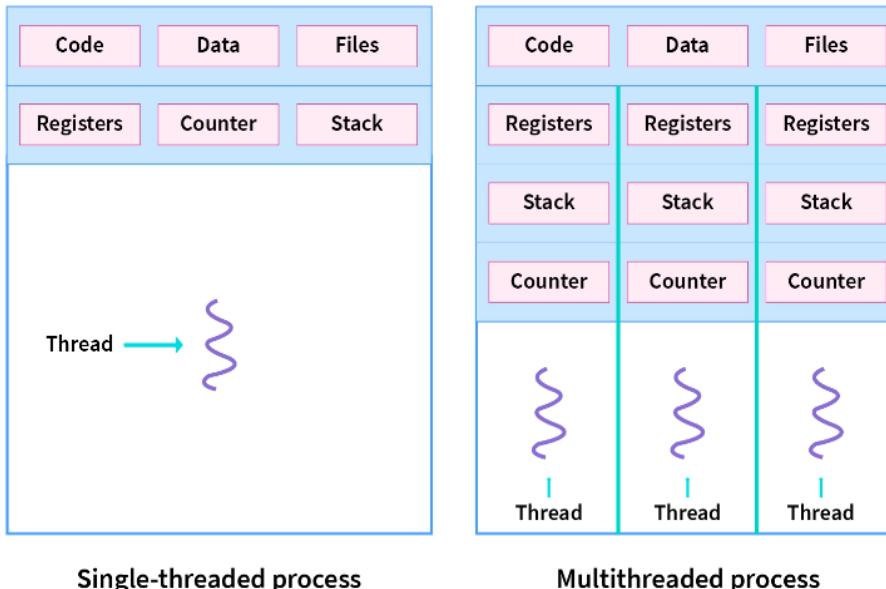
### Components of Threads in Operating System

The Threads in Operating System have the following three components.

- Stack Space
- Register Set
- Program Counter

**The given below figure shows the working of a single-threaded and a multithreaded process:**

A **single-threaded process** is a process with a single thread. A **multi-threaded process** is a process with multiple threads. As the diagram clearly shows that the multiple threads in it have its own registers, stack, and counter but they share the code and data segment.



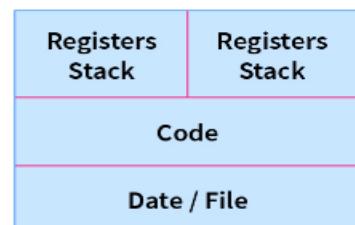
**Process simply means any program in execution while the thread is a segment of a process.** The main differences between process and thread are mentioned below:

<b>Process</b>	<b>Thread</b>
A Process simply means any program in execution.	Thread simply means a segment of a process.
The process consumes more resources	Thread consumes fewer resources.
The process requires more time for creation.	Thread requires comparatively less time for creation than process.
The process is a heavyweight process	Thread is known as a lightweight process
The process takes more time to terminate	The thread takes less time to terminate.
Processes have independent data and code segments	A thread mainly shares the data segment, code segment, files, etc. with its peer threads.
The process takes more time for context switching.	The thread takes less time for context switching.
Communication between processes needs more time as compared to thread.	Communication between threads needs less time as compared to processes.
For some reason, if a process gets blocked then the remaining processes can continue their execution	In case if a user-level thread gets blocked, all of its peer threads also get blocked.
g: Opening two different browsers.	Eg: Opening two tabs in the same browser.

**PROCESS**



**THREAD**

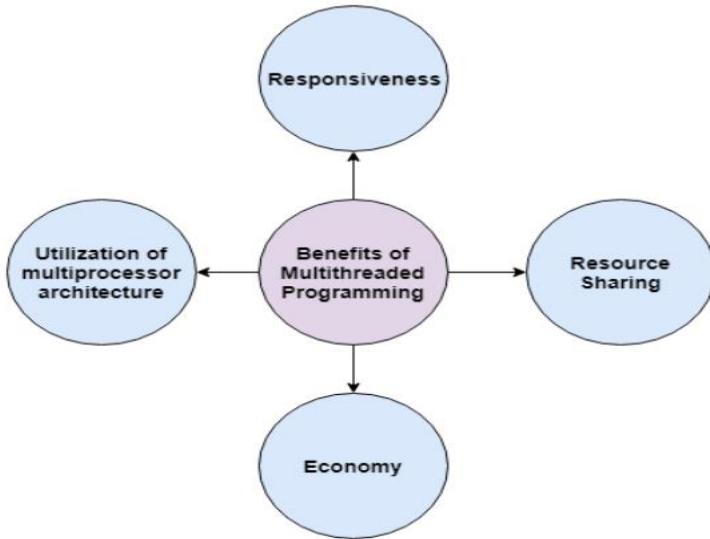


**Two Different Process**

**Two Threads of a single process**

## **Benefits**

The benefits of multithreaded programming can be broken down into four major categories:



- **Resource Sharing**

Processes may share resources only through techniques such as-Message Passing, Shared Memory

Such techniques must be explicitly organized by programmer. However, threads share the memory and the resources of the process to which they belong by default. A single application can have different threads within the same address space using resource sharing.

- **Responsiveness**

Program responsiveness allows a program to run even if part of it is blocked using multithreading. This can also be done if the process is performing a lengthy operation. For example - A web browser with multithreading can use one thread for user contact and another for image loading at the same time.

- **Utilization of Multiprocessor Architecture**

In a multiprocessor architecture, each thread can run on a different processor in parallel using multithreading. This increases concurrency of the system. This is in direct contrast to a single processor system, where only one process or thread can run on a processor at a time.

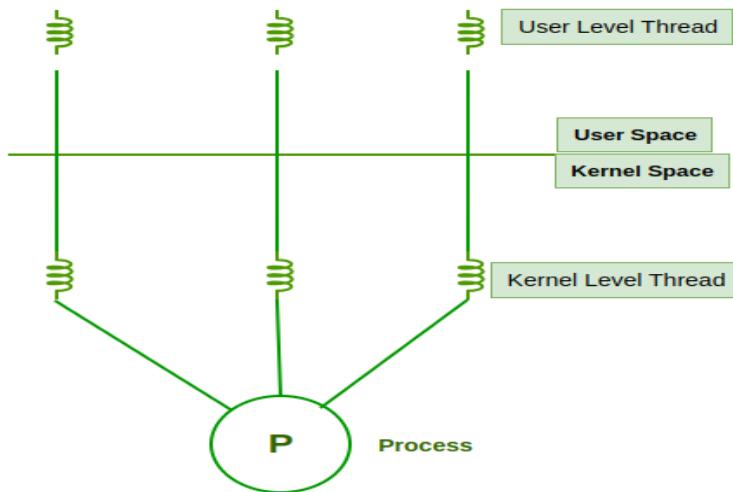
- **Economy**

It is more economical to use threads as they share the process resources. Comparatively, it is more expensive and time-consuming to create processes as they require more memory and resources. The overhead for process creation and management is much higher than thread creation and management.

## Thread Types

Threads are of two types. These are described below.

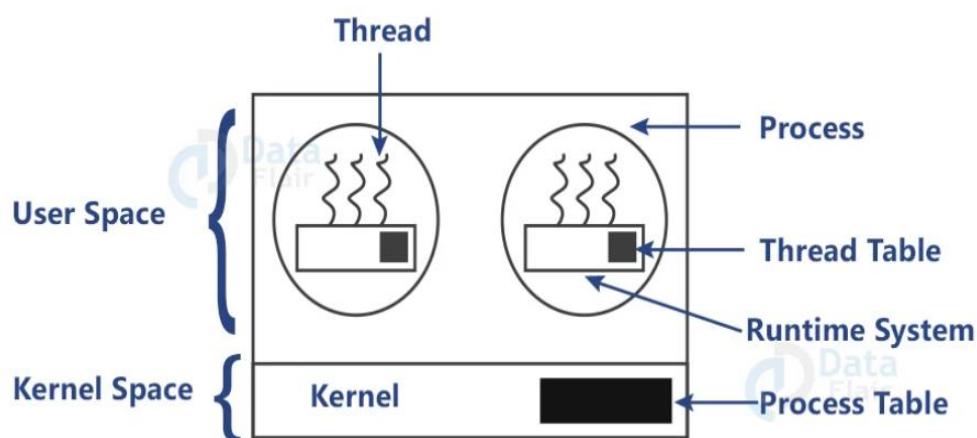
- User Level Thread
- Kernel Level Thread



## Threads

### User Level Threads

User Level Thread is a type of thread that is not created using system calls. The kernel has no work in the management of user-level threads. User-level threads can be easily implemented by the user. In case when user-level threads are single-handed processes, kernel-level thread manages them.  
examples: Java thread, POSIX threads, etc.



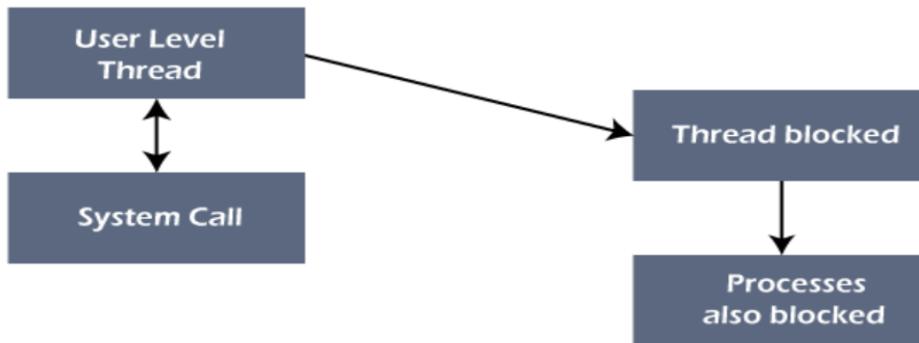
### Advantages of User-Level Threads

- Implementation of the User-Level Thread is easier than Kernel Level Thread.

- Context Switch Time is less in User Level Thread.
- User-Level Thread is more efficient than Kernel-Level Thread.
- Because of the presence of only Program Counter, Register Set, and Stack Space, it has a simple representation.

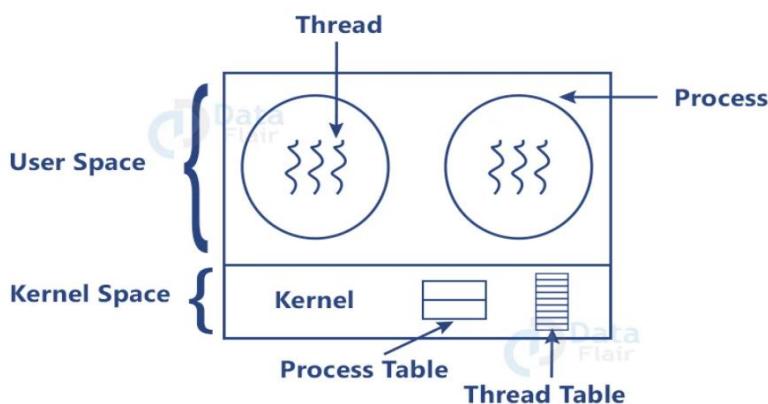
### Disadvantages of User-Level Threads

- There is a lack of coordination between Thread and Kernel.
- In case of a page fault, the whole process can be blocked.



### Kernel Level Threads

A kernel Level Thread is a type of thread that can recognize the Operating system easily. Kernel Level Threads has its own thread table where it keeps track of the system. The operating System Kernel helps in managing threads. Kernel Threads have somehow longer context switching time. Kernel helps in the management of threads. Example: Window Solaris.



### Advantages of Kernel-Level Threads

- It has up-to-date information on all threads.
- Applications that block frequently are to be handled by the Kernel-Level Threads.

- Whenever any process requires more time to process, Kernel-Level Thread provides more time to it.

### **Disadvantages of Kernel-Level threads**

- Kernel-Level Thread is slower than User-Level Thread.
- Implementation of this type of thread is a little more complex than a user-level thread.

### **User Level threads Vs Kernel Level Threads**

<b>User level Threads</b>	<b>Kernel Level Threads</b>
User thread are implemented by Users	Kernal threads are implemented by OS
OS doesn't recognise user level threads	Kernel threads are recognized by OS
Implementation is easy	Implementation is complicated
Context switch time is less	Context switch time is more
Context switch – no hardware support	hardware support is needed
If one user level thread perform blocking operation then entire process will be blocked	If one kernel level thread perform blocking operation then another thread can continue execution.

There are also hybrid models that combine elements of both user-level and kernel-level threads. For example, some operating systems use a hybrid model called the “two-level model”, where each process has one or more user-level threads, which are mapped to kernel-level threads by the operating system.

### **Advantages**

- Hybrid models combine the advantages of user-level and kernel-level threads, providing greater flexibility and control while also improving performance.
- Hybrid models can scale to larger numbers of threads and processors, which allows for better use of available resources.

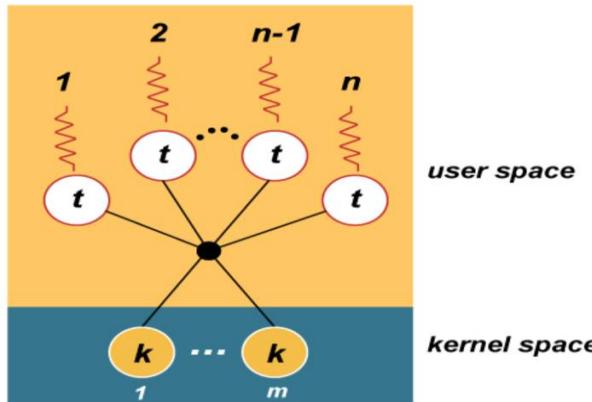
## **Disadvantages:**

- Hybrid models are more complex than either user-level or kernel-level threading, which can make them more difficult to implement and maintain.
- Hybrid models require more resources than either user-level or kernel-level threading, as they require both a thread library and kernel-level support.

***User threads are mapped to kernel threads by the threads library. The way this mapping is done is called the thread model. Multi threading model are of three types.***

### **Many to Many Model**

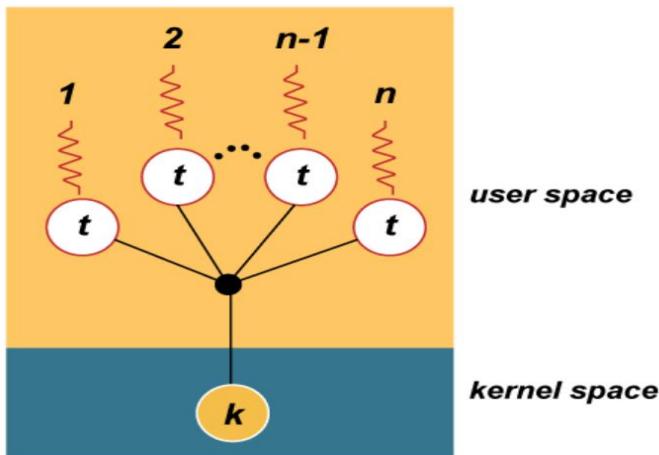
In this model, we have multiple user threads multiplex to same or lesser number of kernel level threads. Number of kernel level threads are specific to the machine, advantage of this model is if a user thread is blocked we can schedule others user thread to other kernel thread. Thus, System doesn't block if a particular thread is blocked. ***It is the best multi threading model.***



### **Many to One Model**

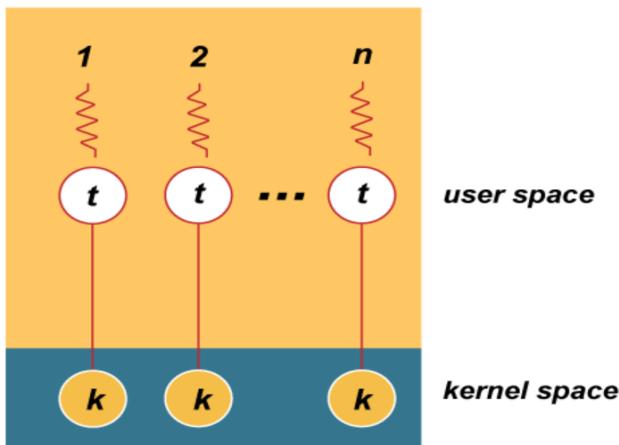
In this model, we have multiple user threads mapped to one kernel thread. In this model when a user thread makes a blocking system call entire process blocks. As we have only one kernel thread and only one user thread can access kernel at a time, so multiple threads are not able access multiprocessor at the same time. The thread management is done on the user level so it is more

efficient.



### One to One Model

In this model, one to one relationship between kernel and user thread. In this model multiple thread can run on multiple processor. Problem with this model is that creating a user thread requires the corresponding kernel thread. As each user thread is connected to different kernel , if any user thread makes a blocking system call, the other user threads won't be blocked.



### Threading Issues in OS

In a multithreading environment, there are many threading-related problems. Such as

- System Call
- Thread Cancellation
- Signal Handling
- Thread Specific Data
- Thread Pool

- Schedular Activation

### ***fork() and exec() System Calls***

- Discussing fork() system call, Let us assume that one of the threads belonging to a multi-threaded program has instigated a fork() call. Therefore, the new process is a duplication of fork(). Here, the question is as follows; will the new duplicate process made by fork() be multi-threaded like all threads of the old process or it will be a unique thread?
- Now, certain UNIX systems have two variants of fork(). fork can either duplicate all threads of the parent process to a child process or just those that were invoked by the parent process. The application will determine which version of fork() to use.
- When the next system call, namely exec() system call is issued, it replaces the whole programming with all its threads by the program specified in the exec() system call's parameters. Ordinarily, the exec() system call goes into queue after the fork() system call.
- However, this implies that the exec() system call should not be queued immediately after the fork() system call because duplicating all the threads of the parent process into the child process will be superfluous. Since the exec() system call will overwrite the whole process with the one given in the arguments passed to exec(). This means that in cases like this; a fork() which only replicates one invoking thread will do.

### ***Thread Cancellation***

- Thread Cancellation is the task of terminating a thread before it has completed.
- *For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be cancelled.*
- *Another situation might occur when a user presses a button on a Web browser that stops a Web page from loading any further. Often, a Web page is loaded using several threads-each image is loaded in a separate thread. When a user presses the stop button on the browser, all threads loading the page are cancelled.*
- A thread that is to be cancelled is often referred to as target thread. The Cancellation of a target thread may occur in two different scenarios:

***Asynchronous cancellation:*** One thread immediately terminates the target thread.

**Deferred Cancellation :** The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

- The difficulty with cancellation occurs in situations where resources have been allocated to a cancelled thread or where a thread is cancelled while in the midst of updating data it is sharing with other threads. This becomes especially troublesome with asynchronous cancellation. Often, the operating system will reclaim system resources from a cancelled thread but will not reclaim all resources. Therefore, cancelling a thread asynchronously may not free a necessary system-wide resource.
- With deferred cancellation, in contrast, one thread indicates that a target thread is to be cancelled, but cancellation occurs only after the target thread has checked a flag to determine whether or not it should be cancelled. The thread can perform this check at a point at which it can be cancelled safely known as cancellation points.

### ***Signal Handling***

- Signal Handling is used in UNIX systems to notify a process that a particular event has occurred. A signal may be received either synchronously or asynchronously, depending on the source of and the reason for the event being signalled.
- All signals, whether synchronous or asynchronous, follow the same pattern:

***A signal is generated by the occurrence of a particular event.***

***A generated signal is delivered to a process.***

***Once delivered, the signal must be handled.***

- Examples of synchronous signals include illegal memory access and division by 0. If a running program performs either of these actions, a signal is generated. Synchronous signals are delivered to the same process that performed the operation that caused the signal (that is the reason they are considered synchronous).
- When a signal is generated by an event external to a running process, that process receives the signal asynchronously. Examples of such signals include terminating a process with specific keystrokes (such as **Ctrl+C**) and having a timer expire. Typically, an asynchronous signal is sent to another process.
- A signal may be handled by one of two possible handlers:
  - A default signal handler
  - A user-defined signal handler

- Every signal has a default signal handler that is run by the kernel when handling that signal. This default action can be overridden by a user defined signal handler that is called to handle the signal.
- Signals are handled in different ways. Some signals (such as changing the size of a window) are simply ignored; others (such as an illegal memory access) are handled by terminating the program.
- Handling signals in single-threaded programs is straightforward: signals are always delivered to a process. However, delivering signals is more complicated in multithreaded programs, where a process may have several threads.
- In general the following options exist:

***Deliver the signal to the thread to which the signal applies.***

***Deliver the signal to every thread in the process.***

***Deliver the signal to certain threads in the process.***

***Assign a specific thread to receive all signals for the process.***

### ***Thread-Specific Data***

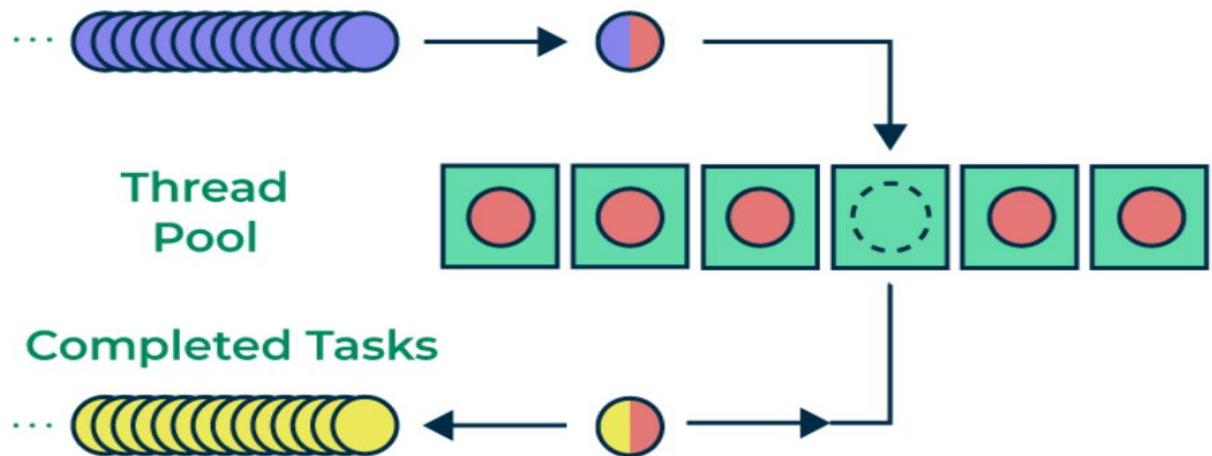
- Threads belonging to a process share the data of the process. Indeed, this sharing of data provides one of the benefits of multithreaded programming. However, in some circumstances, each thread need its own copy of certain data. We will call such data thread specific data.
- For example, in a transaction-processing system, we might service each transaction in a separate thread. Furthermore, each transaction might be assigned a unique identifier. To associate each thread with its unique identifier, we could use thread-specific data.

### ***Thread Pool***

- The server develops an independent thread every time an individual attempts to access a page on it. However, the server also has certain challenges. Bear in mind that no limit in the number of active threads in the system will lead to exhaustion of the available system resources because we will create a new thread for each request.
- The establishment of a fresh thread is another thing that worries us. The creation of a new thread should not take more than the amount of time used up by the thread in dealing with the request and quitting after because this will be wasted CPU resources.

- Hence, thread pool could be the remedy for this challenge. The notion is that as many fewer threads as possible are established during the beginning of the process. A group of threads that forms this collection of threads is referred as a thread pool. There are always threads that stay on the thread pool waiting for an assigned request to service.

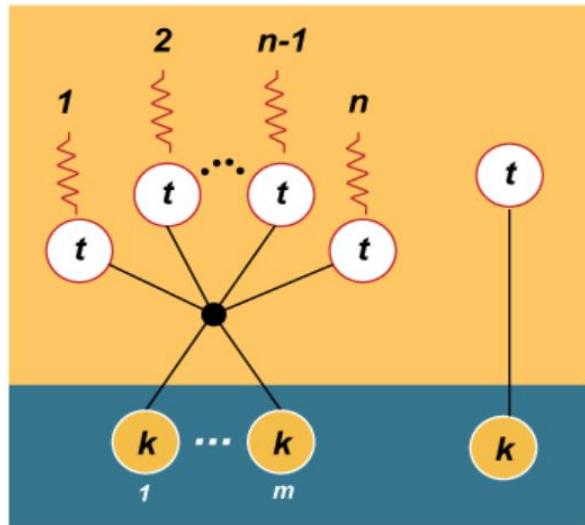
## Task Queue



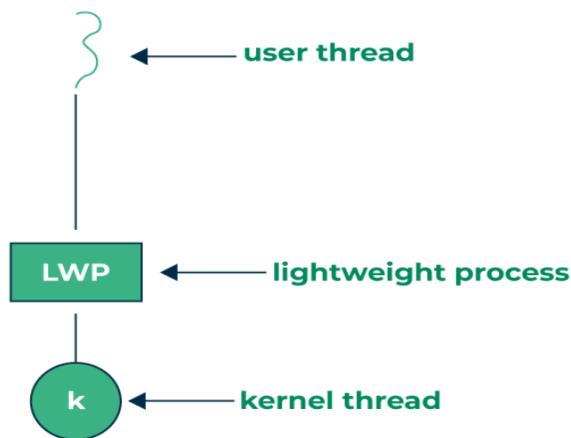
- A new thread is spawned from the pool every time an incoming request reaches the server, which then takes care of the said request. Having performed its duty, it goes back to the pool and awaits its second order.
- Whenever the server receives the request, and fails to identify a specific thread at the ready thread pool, it may only have to wait until some of the threads are available at the ready thread pool. It is better than starting a new thread whenever a request arises because this system works well with machines that cannot support multiple threads at once.

## Schedular Activation

- A final issue to be considered with multithreaded programs concerns communication between the kernel and the thread library, which may be required by the many-to-many and two-level models .



Many systems implementing either the many-to-many or the two-level model place an intermediate data structure between the user and kernel threads. This data structure—typically known as a lightweight process, or LWP—is shown in Figure.



- To the user-thread library, the LWP appears to be a virtual processor on which the application can schedule a user thread to run. Each LWP is attached to a kernel thread, and it is kernel threads that the operating system schedules to run on physical processors.
- If a kernel thread blocks (such as while waiting for an I/O operation to complete), the LWP blocks as well. Up the chain, the user-level thread attached to the LWP also blocks.
- An application may require any number of LWPs to run efficiently. Consider a CPU-bound application running on a single processor. In this scenario, only one thread can run at once, so one LWP is sufficient. An application that is I/O intensive may require multiple LWPs to execute. Typically, an LWP is required for each concurrent blocking system call. Suppose, for example, that five different file-read requests occur simultaneously. Five LWPs are needed, because all could be waiting for I/O completion

in the kernel. If a process has only four LWPs, then the fifth request must wait for one of the LWPs to return from the kernel.

- Furthermore, the kernel must inform an application about certain events. This procedure is known as an Upcall. Upcalls are handled by the thread library with an upcall handlers and upcall handlers must run on a virtual processor. One event that triggers an upcall, occurs when an application thread is about to block. In this scenario, the kernel makes an upcall to the application informing it that a thread is about to block and identifying the specific thread.
- The kernel then allocates a new virtual processor to the application. The application runs an upcall handler on this new virtual processor, which saves the state of the blocking thread and relinquishes the virtual processor on which the blocking thread is running. The upcall handler then schedules another thread that is eligible to run on the new virtual processor.
- When the event that the blocking thread was waiting for occurs, the kernel makes another upcall to the thread library informing it that the previously blocked thread is now eligible to run. The up call handler for this event also requires a virtual processor, and the kernel may allocate a new virtual processor or preempt one of the user threads and run the upcall handler on its virtual processor.

## Process Scheduling

To increase CPU utilization, multiple processes are loaded into the memory of the CPU and a process is selected from these processes. Loading multiple processes into the main memory is called multiprogramming and the act of determining which process is in the **ready** state, and should be moved to the **running** state is known as **Process Scheduling**.

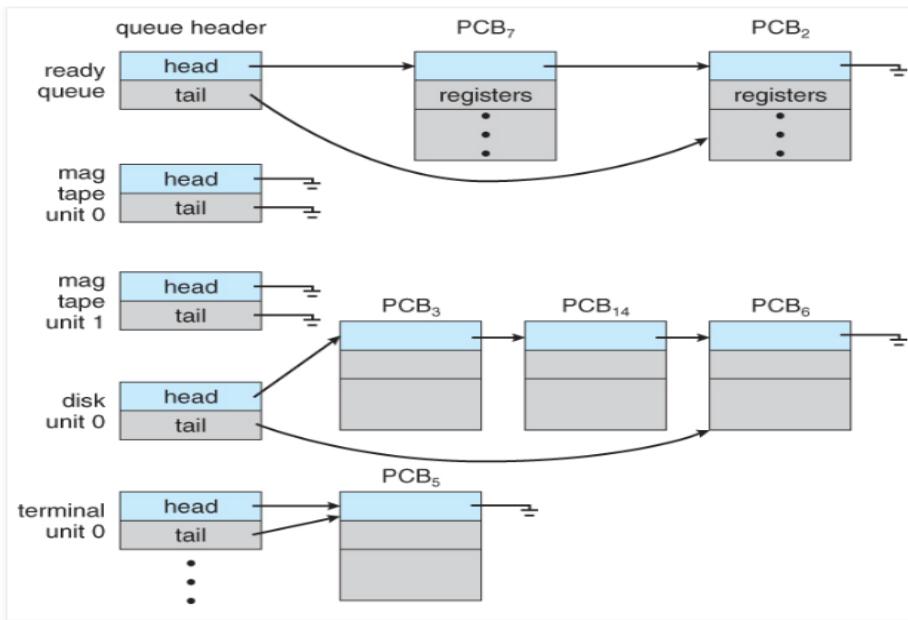
The goal of process scheduling is to achieve efficient utilization of the CPU, fast response time, and fair allocation of resources among all processes.

### Process Scheduling Queues

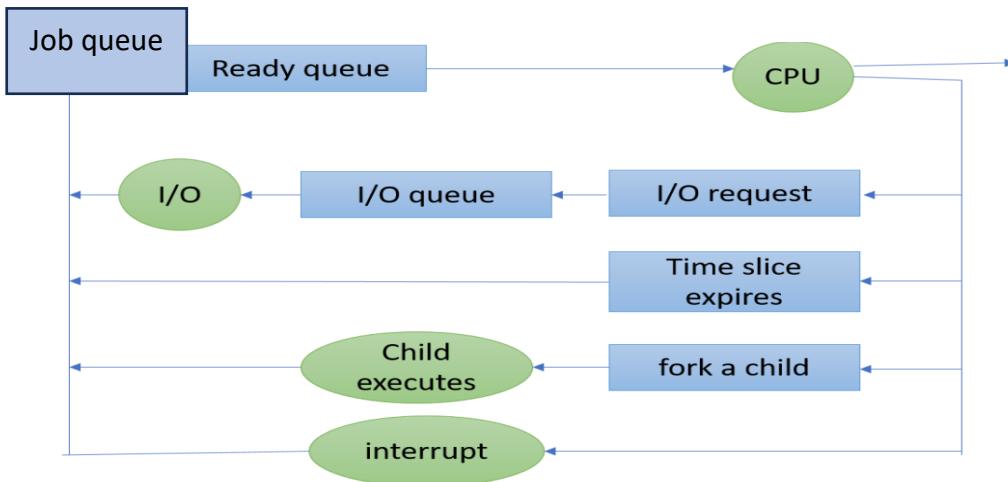
There are multiple states a process has to go through during execution. The OS maintains a separate queue for each state along with the process control blocks (PCB) of all processes. The PCB moves to a new state queue, after being unlinked from its current queue, when the state of a process changes.

These process scheduling queues are:

1. **Job Queue** – In starting, all the processes get stored in the job queue. It is maintained in the secondary memory.
2. **Ready Queue** – This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue. Ready queue is maintained in primary memory..
3. **Device Queue** – When the process needs some IO operation in order to complete its execution, OS changes the state of the process from running to waiting. The context (PCB) associated with the process gets stored on the waiting queue which will be used by the Processor when the process finishes the IO.Each IO Device will have a separate device Queue



A queuing diagram as shown in the figure below represents the scheduling process. There are Two queues – ready queue and device queue. A new process is first admitted into the ready queue. It waits until it is **dispatched** (allocated to the CPU for execution).:



1. The process terminates normally and it exits the system
2. The process issues a I/O request and moves into the device queue. Once it gets the I/O device, it is back into the ready queue
3. The time quantum of the process expires (remember in a time-sharing system, every process gets a fixed amount of CPU time). Then, the process is added back into the ready queue
4. The process creates a child process and waits for the child process to terminate. Hence, it itself is added to the ready queue)

5. A higher priority process interrupts the currently running process. Thus, forcing the current process to change its state to ready state.

### Process Schedulers

A scheduler is a special type of system software that handles process scheduling in numerous ways. It mainly selects the jobs that are to be submitted into the system and decides whether the currently running process should keep running or not. If not then which process should be the next one to run. A scheduler makes a decision:

- When the state of the current process changes from running to waiting due to an I/O request .
- If the current process terminates.
- When the scheduler needs to move a process from running to ready state as it has already run for its allotted interval of time.
- When the requested I/O operation is completed, a process moves from the waiting state to the ready state. So, the scheduler can decide to replace the currently-running process with a newly-ready one.
- 

**There are 3 kinds of schedulers-**



#### 1. Long-term Scheduler-

***The primary objective of long-term scheduler is to maintain a good degree of multiprogramming***

- Long-term scheduler is also known as **Job Scheduler**.
- It selects a balanced mix of I/O bound and CPU bound processes from the secondary memory (new state).

- **CPU Bound Jobs:** CPU-bound jobs are tasks or processes that necessitate a significant amount of CPU processing time and resources (Central Processing Unit). These jobs can put a significant strain on the CPU, affecting system performance and responsiveness.
- **I/O Bound Jobs:** I/O bound jobs are tasks or processes that necessitate a large number of input/output (I/O) operations, such as reading and writing to discs or networks. These jobs are less dependent on the CPU and can put a greater strain on the system's I/O subsystem.
- Then, it loads the selected processes into the main memory (ready state) for execution.

## 2. Short-term Scheduler-

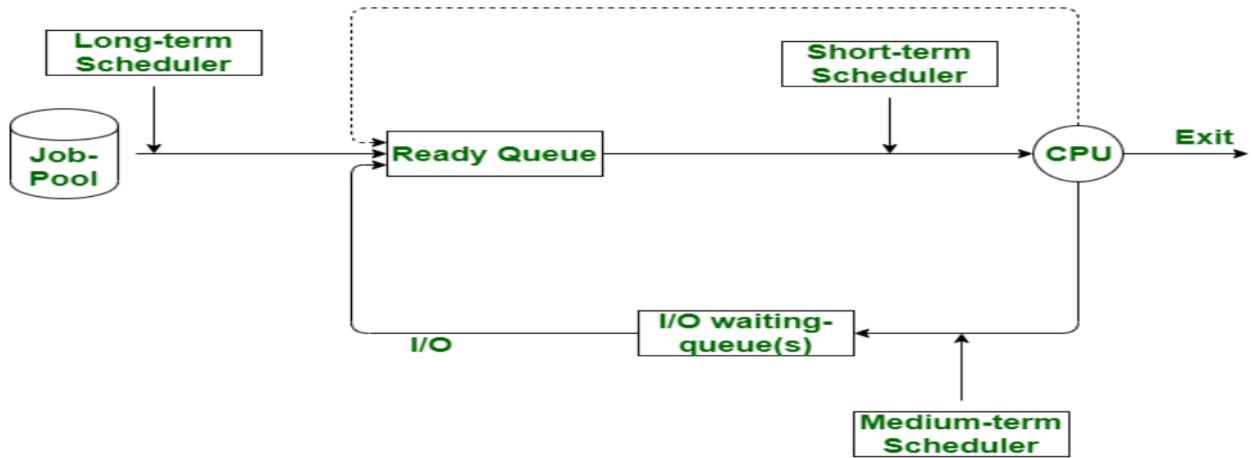
***The primary objective of short-term scheduler is to increase the system performance.***

- Short-term scheduler is also known as **CPU Scheduler**.
- It decides which process to execute next from the ready queue.
- After short-term scheduler decides the process, **Dispatcher** assigns the decided process to the CPU for execution.
- A dispatcher refers to a module that provides the control of the CPU to that process that gets selected by the short term-scheduler.
- A scheduler is something that helps in selecting a process out of various available processes.

## 3. Medium-term Scheduler-

***The primary objective of medium-term scheduler is to perform swapping.***

- Medium-term scheduler swaps-out the processes from main memory to secondary memory to free up the main memory when required.
- Thus, medium-term scheduler reduces the degree of multiprogramming.
- When a running process makes an I/O request it becomes suspended i.e., it cannot be completed. Thus, in order to remove the process from the memory and make space for others, the suspended process is sent to the secondary storage. This is known as swapping, and the process that goes through swapping is said to be swapped out or rolled out.
- After some time when main memory becomes available, medium-term scheduler swaps-in the swapped-out process to the main memory and its execution is resumed from where it left off.



*The major differences between long term, medium term and short term scheduler are as follows –*

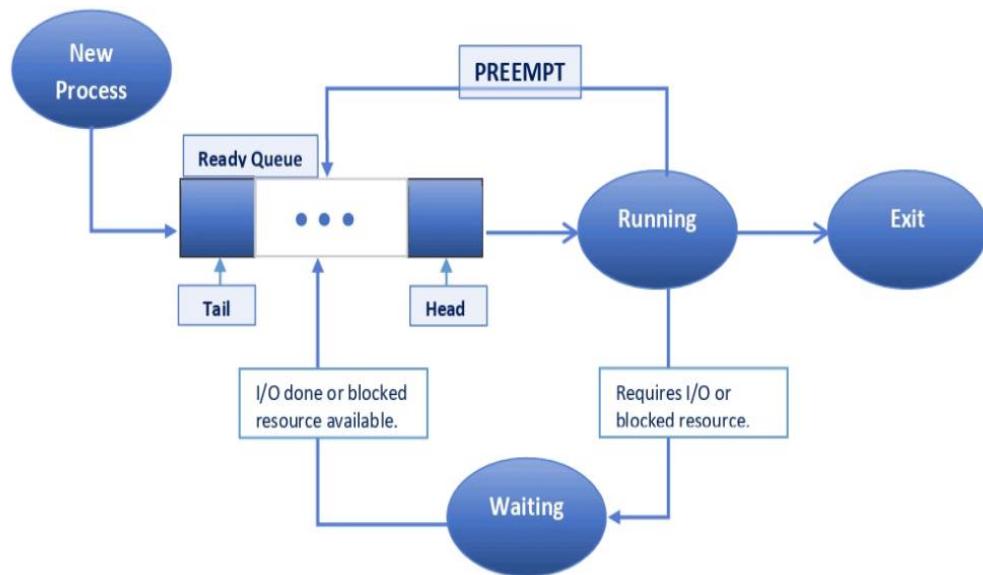
Long term scheduler	Medium term scheduler	Short term scheduler
Long term scheduler is a job scheduler.	Medium term is a process swapping scheduler.	Short term scheduler is called a scheduler.
The speed of long term is lesser than the short term.	The speed of medium term is between short and long scheduler.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

## CPU Scheduling

CPU Scheduling is a process that allows one process to use the CPU while another process is delayed due to unavailability of any resources such as I / O etc, thus making full use of the CPU. Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the short-term scheduler (or CPU scheduler). The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

**CPU Scheduling can be either Preemptive or Non preemptive**

- **Preemptive Scheduling** –The scheduling in which a running process can be interrupted if a high priority process enters the queue and is allocated to the CPU is called preemptive scheduling. In this case, the current process switches from the running queue to ready queue and the high priority process utilizes the CPU cycle.



- Once the allotted time slice is over or there is a higher-priority process ready for execution (depending on the algorithm), the running process is interrupted, and it switches to the ready state and joins the ready queue.
  - Also, whenever a process requires an I/O operation or some blocked resource, it switches to the waiting state. On completion of I/O or receipt of resources, the process switches to the ready state and joins the ready queue.

**Preemptive scheduling has a lot of advantages:**

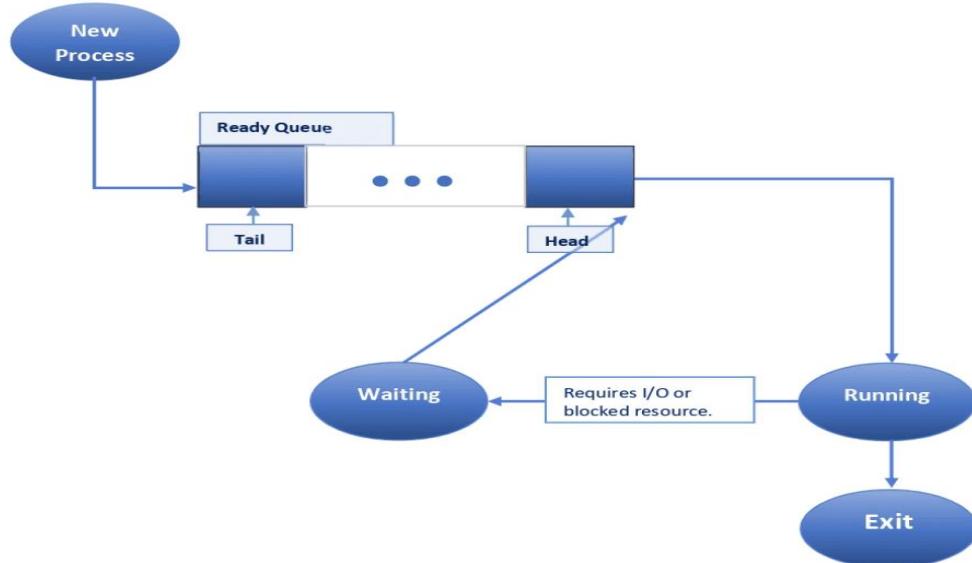
- It ensures no process can monopolize the CPU
  - Improves average response time

- Gives room for reconsideration of choice after every interruption, so if priorities change, a higher-priority process can take the CPU
- Avoids deadlocks

**It has some disadvantages too:**

- Extra time is required for interrupting a running process, switching between contexts, and scheduling newly arriving processes
- **Process starvation** can occur. That's a situation where a low-priority process waits indefinitely or for a long time due to the continuous arrival of higher-priority processes

**Non Preemptive Scheduling** – The scheduling in which a running process cannot be interrupted by any other process is called non-preemptive scheduling. Any other process which enters the queue has to wait until the current process finishes its CPU cycle.



A new process is inserted at the tail of the queue (typically in FCFS). Based on the specific algorithm, a process is selected for execution. The selected process continues until completion without any interruption.

However, if the process requires any I/O operation or some blocked resource while running, it will switch to the waiting state. On completion of the I/O operation or receipt of the needed resource, the process immediately returns to the top of the queue.

**Advantage :** Non-preemptive scheduling has minimal computational overhead and is very simple to understand and implement.

**Significant disadvantage** associated with this type of scheduling is the difficulty in handling priority scheduling. This is so because a process cannot be preempted. Let's suppose a less important process

with a long burst time (time needed to complete execution) is running. Then, a critical process (very high priority) arrives in the queue. This becomes a difficult scenario since a running process cannot be interrupted .This may lead to Deadlock.

### Scheduling Criteria in an Operating System

There are different *CPU* scheduling algorithms with different properties. The choice of algorithm is dependent on various different factors. There are many criteria suggested for comparing *CPU* schedule algorithms, some of which are:

- *CPU* utilization
- Throughput
- Turnaround time
- Waiting time
- Response time

**CPU utilization:**The main objective of any CPU scheduling algorithm is to keep the CPU as busy as possible. Theoretically, CPU utilization can range from 0 to 100 but in a real-time system, it varies from 40 to 90 percent depending on the load upon the system.

**Throughput:**A measure of the work done by the CPU is the number of processes being executed and completed per unit of time. This is called throughput. The throughput may vary depending on the length or duration of the processes.

**Arrival time:**The arrival time of a process is when a process is ready to be executed, which means it is finally in a ready state and is waiting in a queue for its turn to be executed.

**Burst Time:**The **burst time** of a process is the number of time units it requires to be executed by the CPU

**Completion Time:**The completion time is the time when the process stops executing, which means that the process has completed its burst time and is completely executed.

**Turnaround time:** For a particular process, an important criterion is how long it takes to execute that process. The time elapsed from the time of submission of a process to the time of completion is known as the turnaround time. Turn-around time is the sum of times spent waiting to get into memory, waiting in the ready queue, executing in CPU, and waiting for I/O.

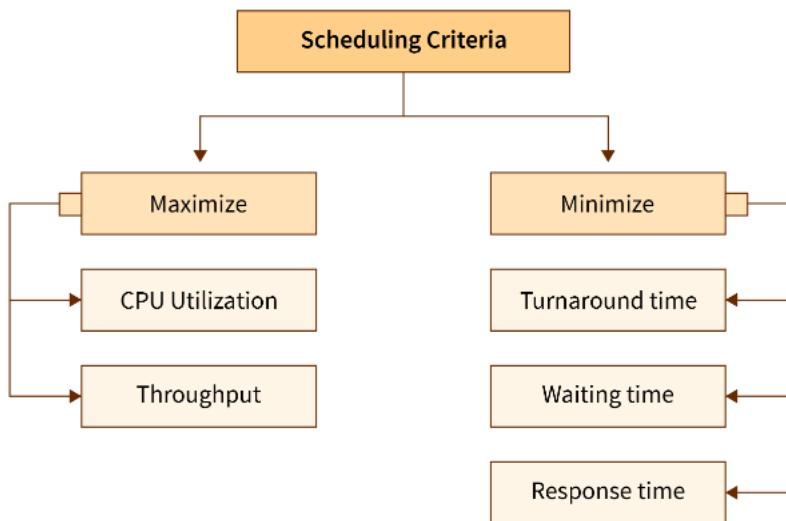
$$\text{Turn Around Time} = \text{Completion Time} - \text{Arrival Time}.$$

**Waiting time**-:A scheduling algorithm does not affect the time required to complete the process once it starts execution. It only affects the waiting time of a process i.e. time spent by a process waiting in the ready queue.

$$\text{Waiting Time} = \text{Turnaround Time} - \text{Burst Time}.$$

**Response time** :In an interactive system, turn-around time is not the best criterion. A process may produce some output fairly early and continue computing new results while previous results are being output to the user. Thus another criterion is the time taken from submission of the process of the request until the first response is produced. This measure is called response time.

$$\text{Response Time} = \text{CPU Allocation Time}(\text{when the CPU was allocated for the first}) - \text{Arrival Time}$$



The aim of the scheduling algorithm is to maximize and minimize the following:

#### **Maximize:**

- **CPU utilization** - It makes sure that the *CPU* is operating at its peak and is busy.
- **Throughput** - It is the number of processes that complete their execution per unit of time.

#### **Minimize:**

- **Waiting time**- It is the amount of waiting time in the queue.
- **Response time**- Time retired for generating the first request after submission.
- **Turnaround time**- It is the amount of time required to execute a specific process.

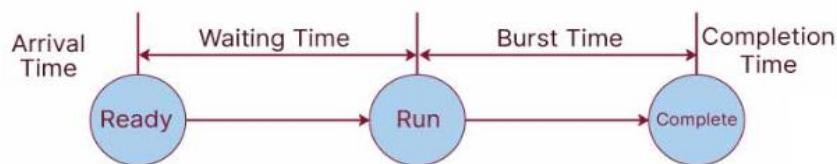
## CPU SCHEDULING

### **Need for CPU Scheduling**

In a single-processor system, only one job can be processed at a time; the rest must wait until the CPU gets free and can be rescheduled. Multiprogramming aims to have a process running at all times to maximize CPU utilization. The concept is straightforward: a process runs until it needs to wait, typically for an I/O request to complete. In a simple operating system, the CPU then stands idle. All this waiting time is wasted; no fruitful work can be performed. With multiprogramming, you can use this time to process other jobs productively with the help of Scheduling.

### **What is CPU Scheduling**

CPU Scheduling is the process which determines the process which will own the CPU for execution while other processes are in the queue. Whenever the CPU is idle, the Operating System selects one of the process which is ready in the queue. This task is carried out by CPU scheduler which selects one of the processes in memory that is ready for execution. Scheduling is used to increase the efficiency of CPU. CPU Scheduling is implemented to minimize Waiting Time, Response Time, and turnaround time and maximize CPU utilization and Throughput.



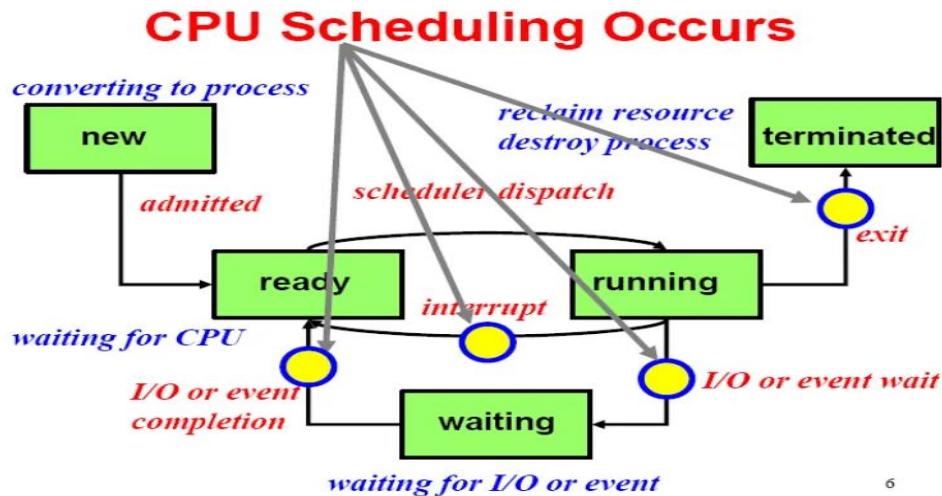
$$\text{Turnaround Time} = \text{Completion Time} - \text{Arrival Time}$$

$$TAT = \text{Completion Time} - \text{Arrival Time}$$

$$\text{Waiting Time} = \text{Turn Around Time} - \text{Burst Time}$$

$$\text{Response Time} = \text{First Response} - \text{Arrival Time}$$

*CPU scheduling decisions may take place under the following four circumstances:*



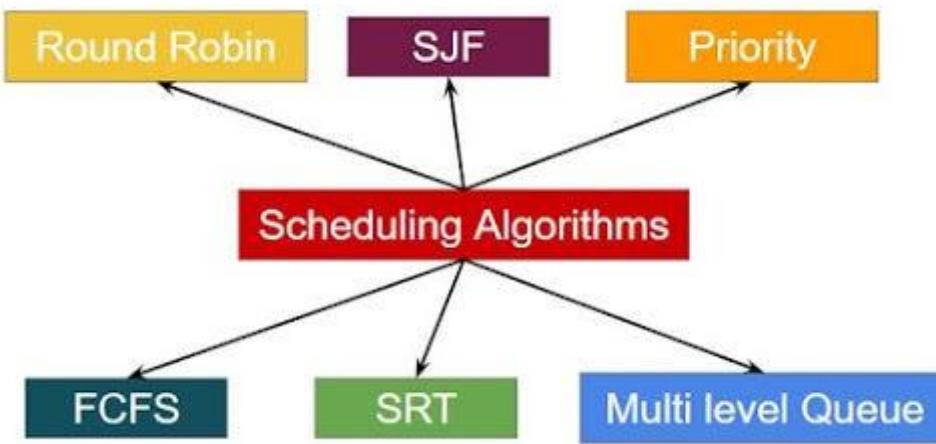
6

- When a process switches from the running state to the waiting state(for I/O request or invocation of wait for the termination of one of the child processes).
- When a process switches from the running state to the ready state (for example, when an interrupt occurs).
- When a process switches from the waiting state to the ready state(for example, completion of I/O).
- When a process terminates.

#### ***Types of CPU Scheduling Algorithm:***

There are 6 types of Scheduling Algorithms:

1. FCFS (First Come First Serve)
2. SJF (Shortest Job First)
3. SRT (Shortest Remaining Time)
4. RR (Round Robin Scheduling)
5. Priority Scheduling
6. Multiple level feedback Scheduling



### First Come First Serve Scheduling

It is the easy and simple CPU Scheduling Algorithm. In this algorithm, the process which requests the CPU first, gets the CPU first for execution. It can be implemented using FIFO (First-In, First-Out) Queue method.

*A simple real-life example of this algorithm is the cash counter. There is a queue on the counter. The person who arrives first at the counter receives the services first, followed by the second person, then third, and so on. The CPU process also works like this.*

#### Advantages of FCFS:

The following are some benefits of using the FCFS scheduling algorithm:

1. The job that comes first is served first.
2. It is the CPU scheduling algorithm's simplest form.
3. It is quite easy to program.

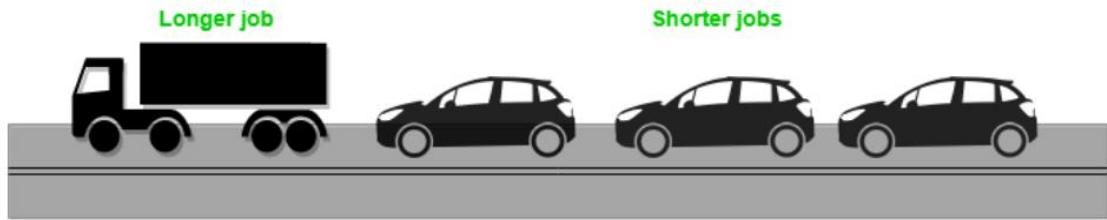
#### Disadvantages

1. It is **non-pre-emptive** algorithm, which means the **process priority** doesn't matter.

If a process with very least priority is being executed, more like **daily routine backup** process, which takes more time, and all of a sudden, some other high priority process arrives, like **interrupt to avoid system crash**, the high priority process will have to wait, and hence in this case, the system will crash, just because of improper process scheduling.

2. In FCFS, the Average Waiting Time is comparatively high.
3. Resources utilization in parallel is not possible, which leads to **Convoy Effect**, (Convoy Effect is a situation where many processes, who need to use a resource for short time are blocked by one process holding that resource for a long time and hence poor resource(CPU, I/O etc) utilization.

**Convoy Effect:** Convoy effect is phenomenon associated with the First Come First Serve (FCFS) algorithm, in which the whole Operating System slows down due to few slow processes.



### Example 1 -FCFS Scheduling:(Without Arrival Times)

TAT=Completion Time-Arrival Time

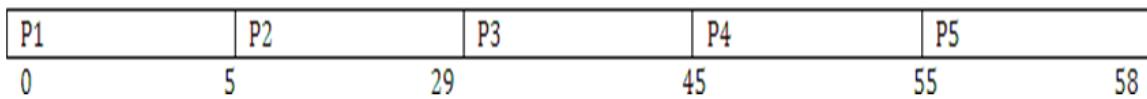
Waiting Time=Turn Around Time-Burst Time

Response Time =First Response - Arrival Time

Process	Burst time(milliseconds)
P1	5
P2	24
P3	16
P4	10
P5	3

### Gantt Chart for FCFS: (Generalized Activity Normalization Time Table (GANTT))

A Gantt chart is a horizontal bar chart used to represent operating systems CPU scheduling in graphical view that help to plan, coordinate and track specific CPU utilization factor like throughput, waiting time, turnaround time etc.



### Average turn around time:

TAT=Completion Time-Arrival Time

Turn around time for p1= 5-0=5.

Turn around time for p2=29-0=29

Turn around time for p3=45-0=45

Turn around time for p4=55-0=55

Turn around time for p5= 55-0=58

**Average turn around time=  $(5+29+45+55+58)/5 = 187/5 = 37.5$  millisecounds**

**Average waiting time:**

Waiting Time=Turn Around Time-Burst Time

Waiting time for p1=5-5=0

Waiting time for p2=29-24=5

Waiting time for p3=45-16=29

Waiting time for p4=55-10=45

Waiting time for p5=58-3=55

**Average waiting time=  $(0+5+29+45+55)/5 = 125/5 = 25$  ms.**

**Average Response Time :**

First Response - Arrival Time

Response Time for P1 =0-0=0

Response Time for P2 => 5-0 = 5

Response Time for P3 => 29-0 = 29

Response Time for P4 => 45-0 = 45

Response Time for P5 => 55-0 = 55

**Average Response Time =>  $(0+5+29+45+55)/5 => 25ms$**

Process	Burst time(milliseconds)	Completion Time	Turnaround Time	Waiting Time	Response time
P1	5	5	5	0	0
P2	24	29	29	5	5
P3	16	45	45	29	29
P4	10	55	55	45	45
P5	3	58	58	55	55

### **First Come First Serve:(With Arrival Times)**

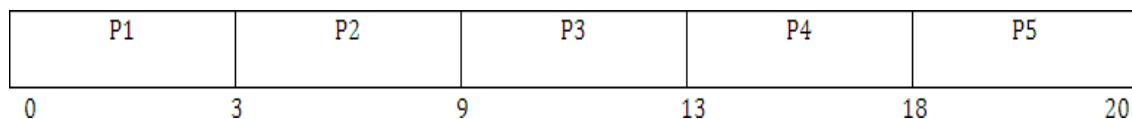
*TAT=Completion Time-Arrival Time*

*Waiting Time=Turn Around Time-Burst Time*

*First Response - Arrival Time*

PROCESS	BURST TIME	ARRIVAL TIME
P1	3	0
P2	6	2
P3	4	4
P4	5	6
P5	2	8

**Gantt Chart**



### **Average Turn Around Time**

*TAT=Completion Time-Arrival Time*

Turn Around Time for P1 => 3-0= 3

Turn Around Time for P2 => 9-2 = 7

Turn Around Time for P3 => 13-4=9

Turn Around Time for P4 => 18-6= 12

Turn Around Time for P5 => 20-8=12

*Average Turn Around Time => ( 3+7+9+12+12 )/5 =>43/5 = 8.50 ms.*

### **Average Response Time :**

*Response Time = First Response - Arrival Time*

Response Time of P1 = 0

Response Time of P2 => 3-2 = 1

Response Time of P3 => 9-4 = 5

Response Time of P4 => 13-6 = 7

Response Time of P5 => 18-8 =10

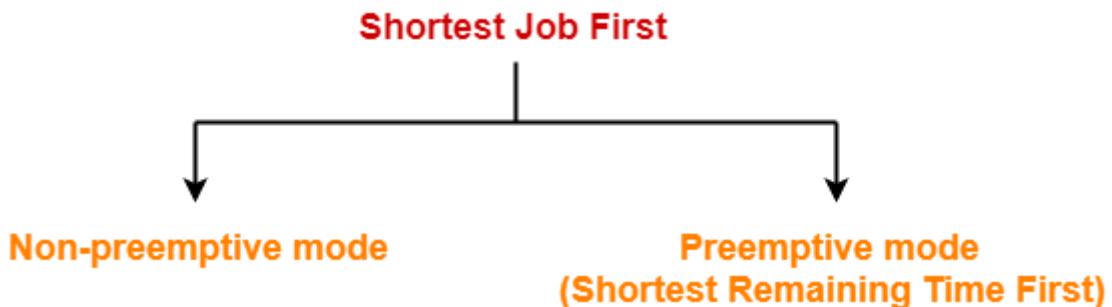
Average Response Time =>  $(0+1+5+7+10)/5 \Rightarrow 23/5 = 4.6 \text{ ms}$

Process	Burst Time	Arrival Time	Completion Time	Turnaround Time	Waiting Time	Response Time
P1	3	0	3	3	0	0
P2	6	2	9	7	1	1
P3	4	4	13	9	5	5
P4	5	6	18	12	7	7
P5	2	8	20	12	10	10

### Shortest Job First

Till now, we were scheduling the processes according to their arrival time (in FCFS scheduling). However, SJF scheduling algorithm, schedules the processes according to their burst time. In SJF scheduling, the process with the lowest burst time, among the list of available processes in the ready queue, is going to be scheduled next.

In case of a tie, it is broken by **FCFS Scheduling**.



### Non-Premptive Mode(Example)

Process Id	Arrival time	Burst time
P1	3	1
P2	1	4

P3	4	2
P4	0	6
P5	2	3



**Gantt Chart**

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

Process Id	Exit time/Finishing/Completion Time	Turn Around time	Waiting time
P1	7	$7 - 3 = 4$	$4 - 1 = 3$
P2	16	$16 - 1 = 15$	$15 - 4 = 11$
P3	9	$9 - 4 = 5$	$5 - 2 = 3$
P4	6	$6 - 0 = 6$	$6 - 6 = 0$
P5	12	$12 - 2 = 10$	$10 - 3 = 7$

**Now,**

- **Average Turn Around time =  $(4 + 15 + 5 + 6 + 10) / 5 = 40 / 5 = 8 \text{ unit}$**
- **Average waiting time =  $(3 + 11 + 3 + 0 + 7) / 5 = 24 / 5 = 4.8 \text{ unit}$**

### **Advantages**

- According to the definition, short processes are executed first and then followed by longer processes.
- The throughput is increased because more processes can be executed in less amount of time.

### **Disadvantages:**

- The time taken by a process must be known by the CPU beforehand, which is not possible.
- Longer processes will have more waiting time, eventually they'll suffer starvation.

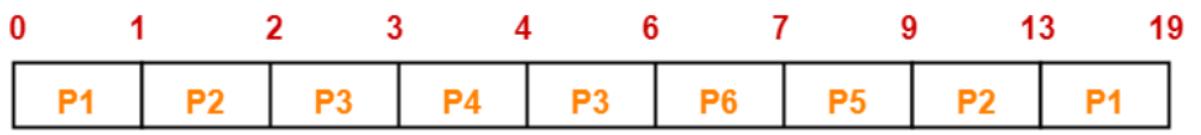
### **Shortest Remaining Time First(SRTF)**

The Preemptive version of Shortest Job First(SJF) scheduling is known as Shortest Remaining Time First (SRTF). With the help of the SRTF algorithm, the process having the smallest amount of time remaining until completion is selected first to execute.

Consider the set of 6 processes whose arrival time and burst time are given below

Process Id	Arrival time	Burst time
P1	0	7
P2	1	5
P3	2	3
P4	3	1
P5	4	2
P6	5	1

### **Gantt Chart-**



**Gantt Chart**

Now, we know-

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

Process Id	Exit time	Turn Around time	Waiting time
P1	19	$19 - 0 = 19$	$19 - 7 = 12$
P2	13	$13 - 1 = 12$	$12 - 5 = 7$
P3	6	$6 - 2 = 4$	$4 - 3 = 1$
P4	4	$4 - 3 = 1$	$1 - 1 = 0$
P5	9	$9 - 4 = 5$	$5 - 2 = 3$
P6	7	$7 - 5 = 2$	$2 - 1 = 1$

### Advantages

- Processes are executed faster than SJF, being the preemptive version of it.

### Disadvantages

- Context switching is done a lot more times and adds to the overhead time.
- Like SJF, it may still lead to starvation and requires the knowledge of process time beforehand.
- Impossible to implement in interactive systems where the required CPU time is unknown.

## Priority Scheduling Algorithm

Priority scheduling in OS is the scheduling algorithm that schedules processes according to the priority assigned to each of the processes. Higher priority processes are executed before lower priority process

### ***Priority of processes depends on some factors such as:***

- Time limit
- Memory requirements of the process
- Ratio of average I/O to average CPU burst time

There can be more factors on the basis of which the priority of a process/job is determined. This priority is assigned to the processes by the scheduler. These priorities of processes are represented as simple integers in a fixed range such as 0 to 7, or maybe 0 to 4095. These numbers depend on the type of system.

**Note :** Generally, the process with the Larger integer number will have low priority, and the process with the smaller integer value will have high priority.

Priorities can be static or dynamic, depending on the situation.

**Static priority:** It doesn't change priority throughout the execution of the process

**Dynamic priority:** In this dynamic priority, priority can be changed by the scheduler at a regular interval of time.

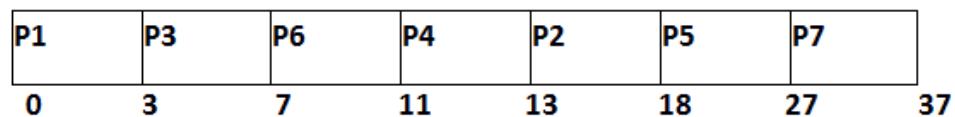
Priority scheduling can be of two types:

- Preemptive Priority Scheduling:** If the new process arrived at the ready queue has a higher priority than the currently running process, the CPU is preempted, which means the processing of the current process is stopped and the incoming new process with higher priority gets the CPU for its execution.
- Non-Preemptive Priority Scheduling:** In case of non-preemptive priority scheduling algorithm if a new process arrives with a higher priority than the current running process, the incoming process is put at the head of the ready queue, which means after the execution of the current process it will be processed.

### Non-Preemptive Priority Scheduling

Process ID	Priority	Arrival Time	Burst Time
1	2	0	3
2	6	2	5
3	3	1	4
4	5	4	2
5	7	6	9
6	4	5	4

7	10	7	10
---	----	---	----



process Id	Priority	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time	Response Time
1	2	0	3	3	3	0	0
2	6	2	5	18	16	11	13
3	3	1	4	7	6	2	3
4	5	4	2	13	9	7	11
5	7	6	9	27	21	12	18
6	4	5	4	11	6	2	7
7	10	7	10	37	30	18	27

$$\text{Avg Waiting Time} = (0+11+2+7+12+2+18)/7 = 52/7 \text{ units}$$

### Preemptive Priority CPU Scheduling

**Step-1:** Select the first process whose arrival time will be 0, we need to select that process because that process is only executing at time t=0.

**Step-2:** Check the priority of the next available process. Here we need to check for 3 conditions.

**if priority(current\_process) > priority(prior\_process)** :- then execute the current process.

**if priority(current\_process) < priority(prior\_process)** :- then execute the prior process.

**if priority(current\_process) = priority(prior\_process)** :- then execute the process which arrives first i.e., arrival time should be first.

**Step-3:** Repeat **Step-2** until it reaches the final process.

**Step-4:** When it reaches the final process, choose the process which is having the highest priority & execute it. Repeat the same step until all processes complete their execution.

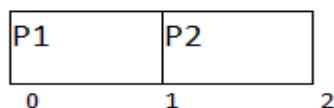
Process Id	Priority	Arrival Time	Burst Time
1	2(L)	0	1
2	6	1	7
3	3	2	3
4	5	3	6
5	4	4	5
6	10(H)	5	15
7	9	15	8

#### GANTT chart Preparation

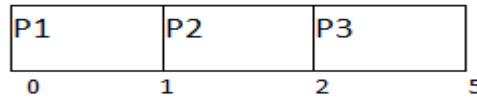
At time 0, P1 arrives with the burst time of 1 units and priority 2. Since no other process is available hence this will be scheduled till next job arrives or its completion (whichever is lesser).



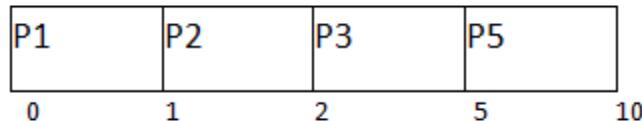
At time 1, P2 arrives. P1 has completed its execution and no other process is available at this time hence the Operating system has to schedule it regardless of the priority assigned to it.



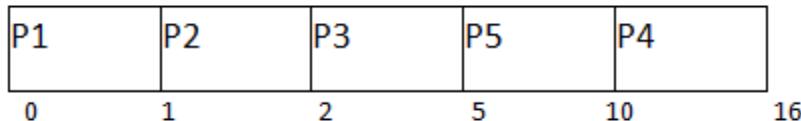
The Next process P3 arrives at time unit 2, the priority of P3 is higher to P2. Hence the execution of P2 will be stopped and P3 will be scheduled on the CPU.



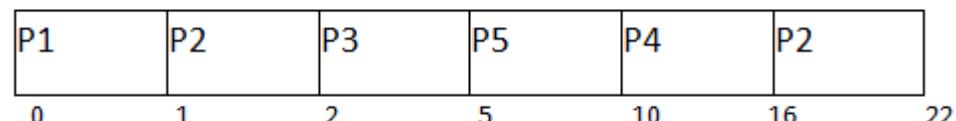
During the execution of P3, three more processes P4, P5 and P6 becomes available. Since, all these three have the priority lower to the process in execution so P3 can't preempt the process. P3 will complete its execution and then P5 will be scheduled with the priority highest among the available processes.



Meanwhile the execution of P5, all the processes got available in the ready queue. At this point, the algorithm will start behaving as Non Preemptive Priority Scheduling. Hence now, once all the processes get available in the ready queue, the OS just took the process with the highest priority and execute that process till completion. In this case, P4 will be scheduled and will be executed till the completion.



Since P4 is completed, the other process with the highest priority available in the ready queue is P2. Hence P2 will be scheduled next.



P2 is given the CPU till the completion. Since its remaining burst time is 6 units hence P7 will be scheduled after this.

P1	P2	P3	P5	P4	P2	P7	
0	1	2	5	10	16	22	30

The only remaining process is P6 with the least priority, the Operating System has no choice unless of executing it. This will be executed at the last.

P1	P2	P3	P5	P4	P2	P7	P6	
0	1	2	5	10	16	22	30	45

The Completion Time of each process is determined with the help of GANTT chart. The turnaround time and the waiting time can be calculated by the following formula.

1. Turnaround Time = Completion Time - Arrival Time
2. Waiting Time = Turn Around Time - Burst Time

Process Id	Priority	Arrival Time	Burst Time	Completion Time	Turn around Time	Waiting Time
1	2	0	1	1	1	0
2	6	1	7	22	21	14
3	3	2	3	5	3	0
4	5	3	6	16	13	7
5	4	4	5	10	6	1
6	10	5	15	45	40	25
7	9	6	8	30	24	16

$$\text{Avg Waiting Time} = (0+14+0+7+1+25+16)/7 = 63/7 = 9 \text{ units}$$

### **Advantages of priority scheduling in OS**

- Easy to use.
- Processes with higher priority execute first which saves time.
- The importance of each process is precisely defined.
- A good algorithm for applications with fluctuating time and resource requirements.

### **Disadvantages of priority scheduling in OS**

- We can lose all the low-priority processes if the system crashes.
- This process can cause starvation if high-priority processes take too much CPU time. The lower priority process can also be postponed for an indefinite time.

### **Ageing Technique**

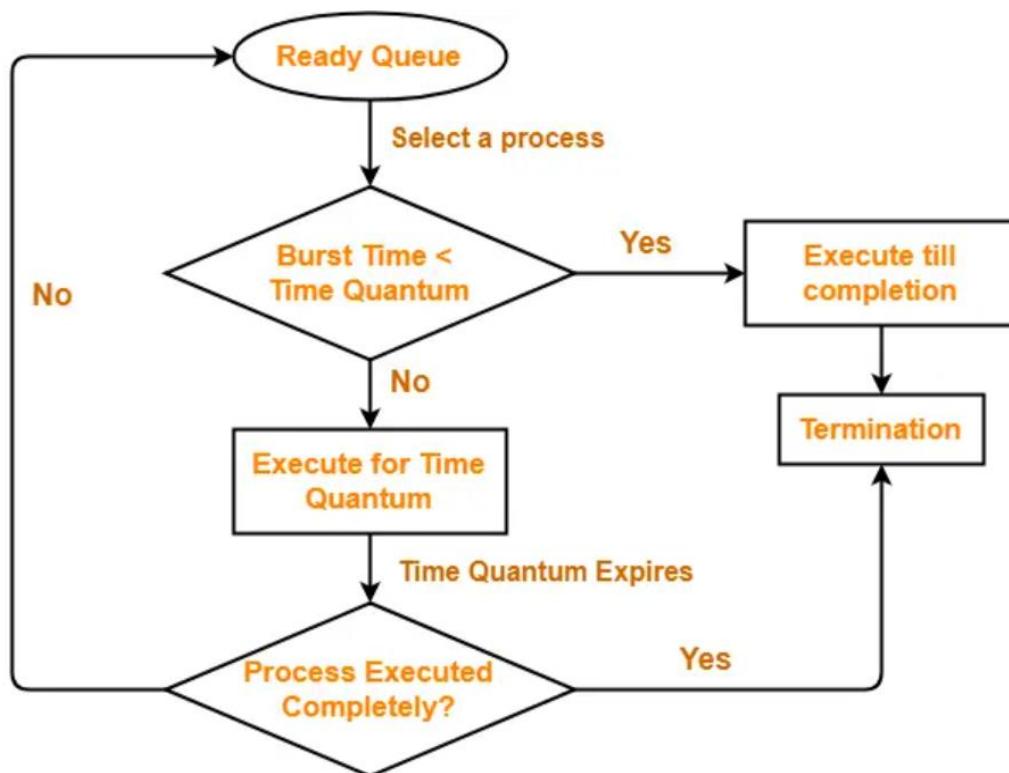
Ageing technique can help prevent starvation of a process. In this, we can increase the priority of the low-priority processes based on their waiting time. This ensures that no process waits for an indefinite time to get CPU time.

## Round Robin Scheduling,

In Round Robin Scheduling,

- CPU is assigned to the process on the basis of FCFS for a fixed amount of time.
- This fixed amount of time is called as **time quantum** or **time slice**.
- After the time quantum expires, the running process is preempted and sent to the ready queue.
- Then, the processor is assigned to the next arrived process.
- It is always preemptive in nature.

Round Robin Scheduling is FCFS Scheduling with preemptive mode.

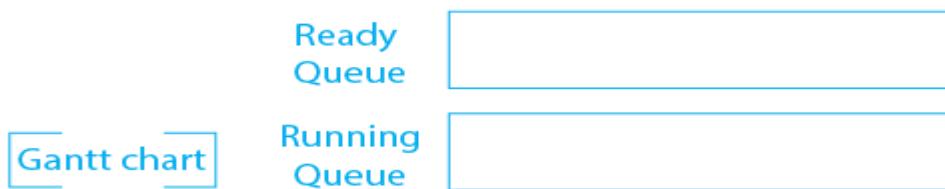


### Round-robin scheduling algorithm with an example.

Let's see how the round-robin scheduling algorithm works with an example. Here, we have taken an example to understand the working of the algorithm. We will also do a dry run to understand it better.

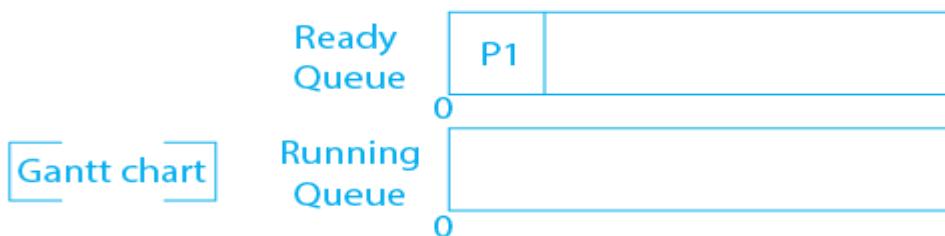
Process No	Arrived Time	Burst Time
P1	0	5
P2	1	4
P3	2	2
P4	4	1

In the above example, we have taken 4 processes P1, P2, P3, and P4 with an arrival time of 0,1,2, and 4 respectively. They also have burst times 5, 4, 2, and 1 respectively. Now, we need to create two queues the ready queue and the running queue which is also known as the **Gantt chart**.



**Step 1:** first, we will push all the processes in the ready queue with an arrival time of 0. In this example, we have only P1 with an arrival time of 0.

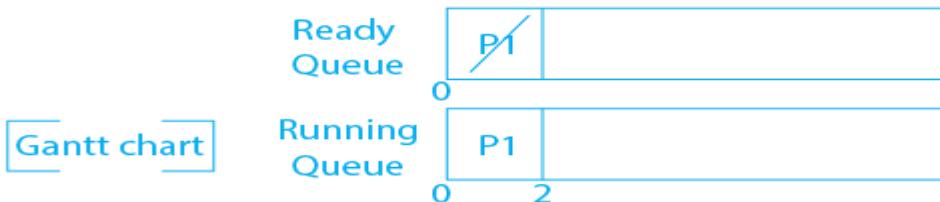
Process No	Arrived Time	Burst Time
P1	0	5
P2	1	4
P3	2	2
P4	4	1



This is how queues will look after the completion of the first step.

**Step 2:** Now, we will check in the ready queue and if any process is available in the queue then we will remove the first process from the queue and push it into the running queue. Let's see how the queue will be after this step.

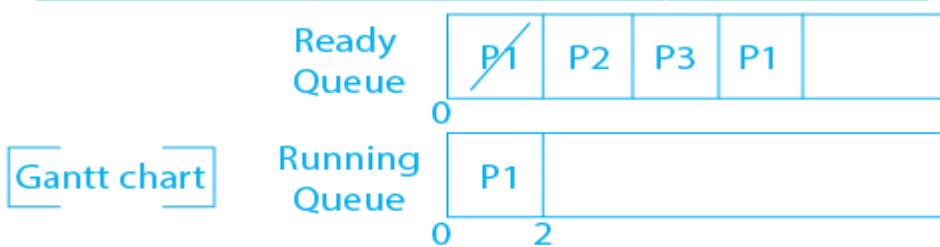
Process No	Arrived Time	Burst Time
P1	0	3
P2	1	4
P3	2	2
P4	4	1



In the above image, we can see that we have pushed process P1 from the ready queue to the running queue. We have also decreased the burst time of process P1 by 2 units as we already executed 2 units of P1.

**Step 3:** Now we will push all the processes arrival time within 2 whose burst time is not 0.

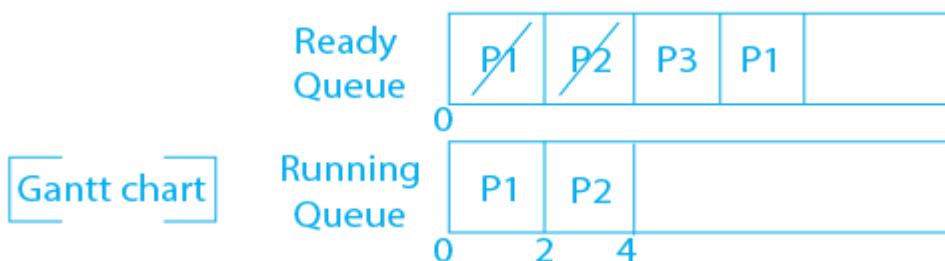
Process No	Arrived Time	Burst Time
P1	0	3
P2	1	4
P3	2	2
P4	4	1



In the above image, we can see that we have two processes with an arrival time within 2 P2 and P3 so, we will push both processes into the ready queue. Now, we can see that process P1 also has remaining burst time so we will also push process P1 into the ready queue again.

**Step 4:** Now we will see if there are any processes in the ready queue waiting for execution. If there is any process then we will add it to the running queue.

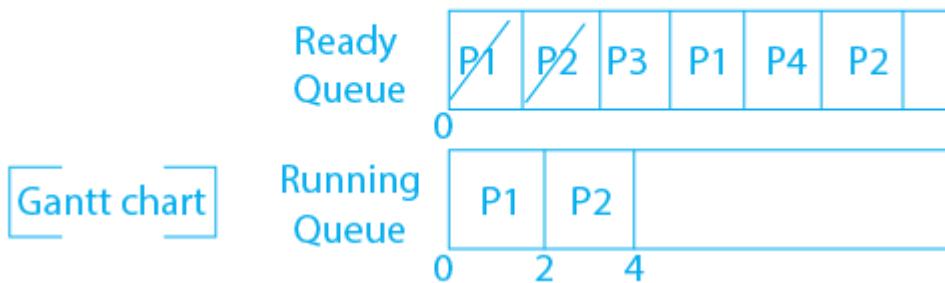
Process No	Arrived Time	Brust Time
P1	0	3
P2	1	4
P3	2	2
P4	4	1



In the above image, we can see that we have pushed process P2 from the ready queue to the running queue. We also decreased the burst time of the process P2 as it already executed 2 units.

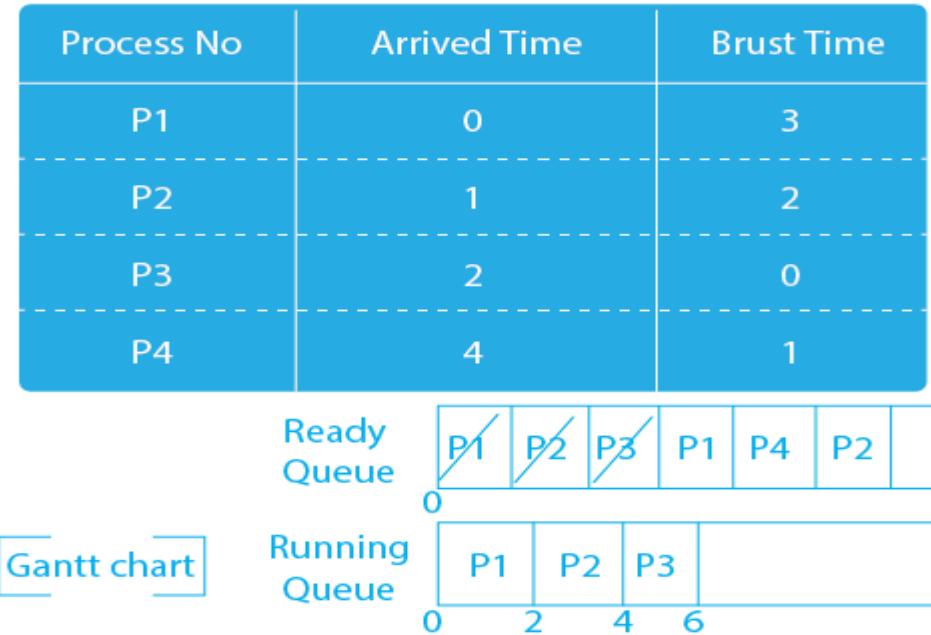
**Step 5:** Now we will push all the processes arrival time within 4 whose burst time is not 0.

Process No	Arrived Time	Brust Time
P1	0	3
P2	1	2
P3	2	2
P4	4	1



In the above image, we can see that we have one process with an arrival time within 4 P4 so, we will push that process into the ready queue. Now, we can see that process P2 also has remaining burst time so we will also push process P2 into the ready queue again.

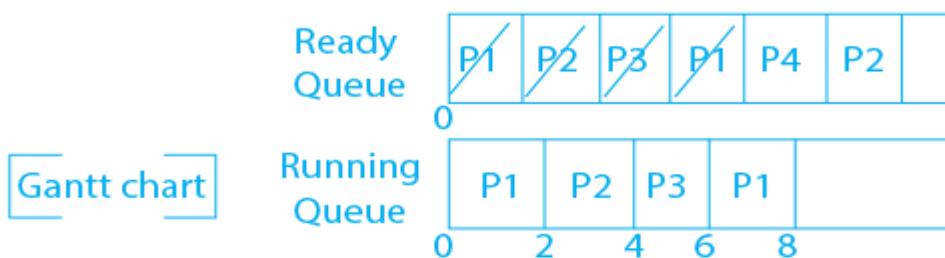
**Step 6:** Now we will see if there are any processes in the ready queue waiting for execution. If there is any process then we will add it to the running queue.



In the above image, we can see that we have pushed process P3 from the ready queue to the running queue. We also decreased the burst time of the process P3 as it already executed 2 units. Now, process P3's burst time becomes 0 so we will not consider it further.

**Step 7:** Now we will see if there are any processes in the ready queue waiting for execution. If there is any process then we will add it to the running queue.

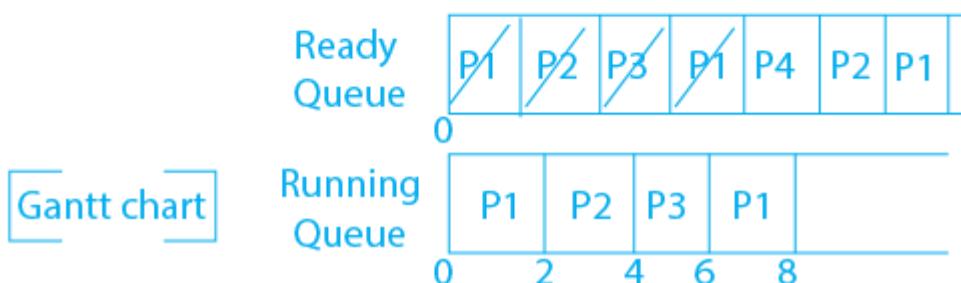
Process No	Arrived Time	Burst Time
P1	0	1
P2	1	2
P3	2	0
P4	4	1



In the above image, we can see that we have pushed process P1 from the ready queue to the running queue. We also decreased the burst time of the process P1 as it already executed 2 units.

**Step 8:** Now we will push all the processes arrival time within 8 whose burst time is not 0.

Process No	Arrived Time	Burst Time
P1	0	1
P2	1	2
P3	2	0
P4	4	1



In the above image, we can see that process P1 also has a remaining burst time so we will also push process P1 into the ready queue again.

**Step 9:** Now we will see if there are any processes in the ready queue waiting for execution. If there is any process then we will add it to the running queue.



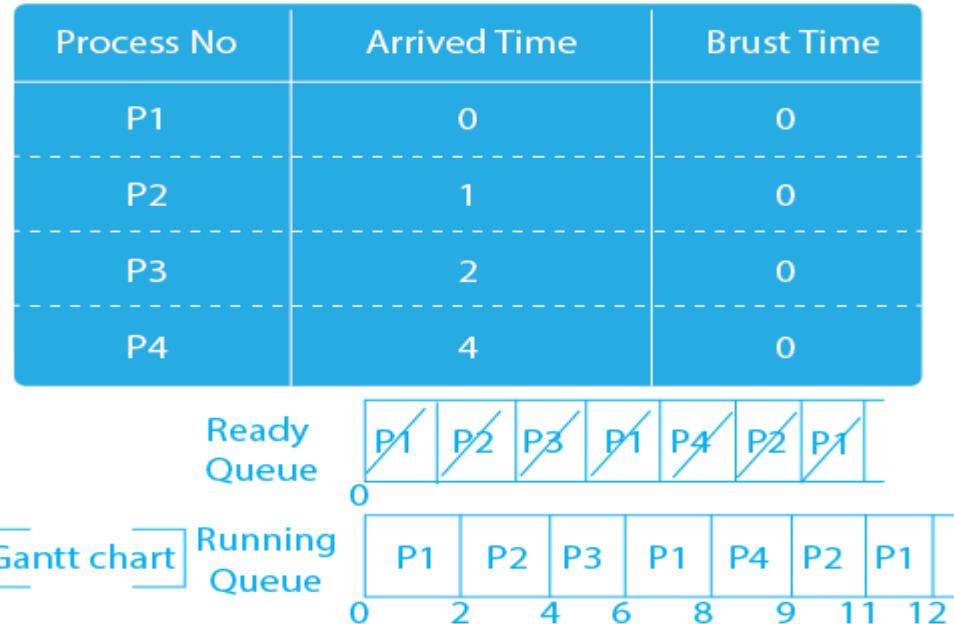
In the above image, we can see that we have pushed process P4 from the ready queue to the running queue. We also decreased the burst time of the process P4 as it already executed 1 unit. Now, process P4's burst time becomes 0 so we will not consider it further.

**Step 10:** Now we will see if there are any processes in the ready queue waiting for execution. If there is any process then we will add it to the running queue.



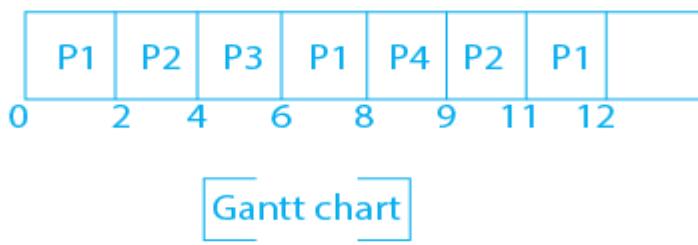
In the above image, we can see that we have pushed process P2 from the ready queue to the running queue. We also decreased the burst time of the process P2 as it already executed 2 units. Now, process P2's burst time becomes 0 so we will not consider it further.

**Step 11:** Now we will see if there are any processes in the ready queue waiting for execution. If there is any process then we will add it to the running queue.



In the above image, we can see that we have pushed process P1 from the ready queue to the running queue. We also decreased the burst time of the process P1 as it already executed 1 unit. Now, process P1's burst time becomes 0 so we will not consider it further. Now our ready queue is empty so we will not perform any task now.

After performing all the operations, our running queue also known as the **Gantt chart** will look like the below.



Let's calculate the other terms like Completion time, Turn Around Time (TAT), Waiting Time (WT), and Response Time (RT). Below are the equations to calculate the above terms.

$$\begin{array}{l}
 \text{Turn Around Time} = \text{Completion Time} - \text{Arrival Time} \\
 \text{Waiting Time} = \text{Turn Around Time} - \text{Burst Time}
 \end{array}$$

**Response Time = CPU first time – Arrival Time**

Let's calculate all the details for the above example.

Process No	Arrived Time	Burst Time	completion Time	TAT	WT	RT
P1	0	5	12	12	7	0
P2	1	4	11	10	6	1
P3	2	2	6	4	2	2
P4	4	1	9	5	4	4

#### Advantages-

- It gives the best performance in terms of average response time.
- It is best suited for time sharing system, client server architecture and interactive system.
- It doesn't face the issues of starvation or convoy effect.
- All the jobs get a fair allocation of CPU.
- It deals with all process without any priority

#### Disadvantages-

- It leads to starvation for processes with larger burst time as they have to repeat the cycle many times.
- Its performance heavily depends on time quantum.
- Lower time quantum results in higher the context switching overhead in the system and Higher time quantum makes it as FCFS
- Finding a correct time quantum is a quite difficult task in this system.
- Priorities can not be set for the processes.

## Multilevel queue scheduling

Multilevel queue scheduling is used when processes in the ready queue can be divided into different classes where each class has its own scheduling needs.

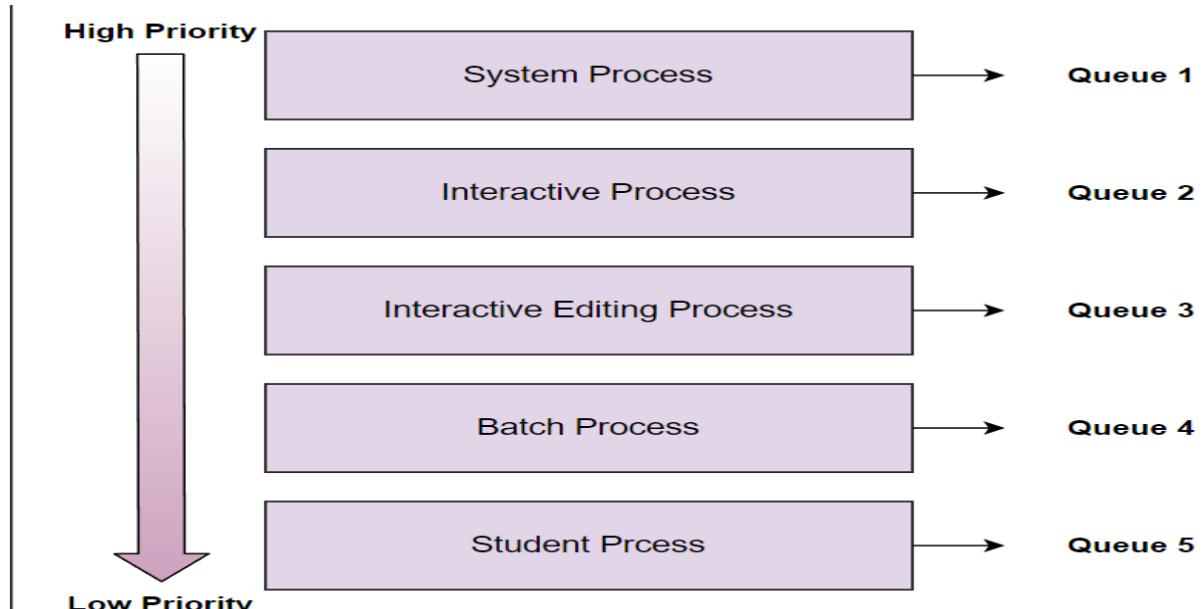
### Example

Suppose we have five following processes of different nature. Let's schedule them using a multilevel queue scheduling:

Process ID	Process Type	Description
P1	System Process	The core process of OS.
P2	Interactive Process	A process with some type of interaction.
P3	Interactive Editing Process	A type of interactive process.
P4	Batch Process	An OS feature that collects data and instructions into a batch before processing.
P5	Student Process	Process of lowest priority.

### Solution

Let's handle this process scheduling using multilevel queue scheduling. To understand this better, look at the figure below:



The above illustration shows the division of queues based on priority. Every single queue has an absolute priority over the low-priority queue, and no other process is allowed to execute until the high-

priority queues are empty. For instance, if the interactive editing process enters the ready queue while the batch process is underway, then the batch process will be preempted

## Advantages

1. You can use multilevel queue scheduling to apply different scheduling methods to distinct processes.
2. It will have low overhead in terms of scheduling.

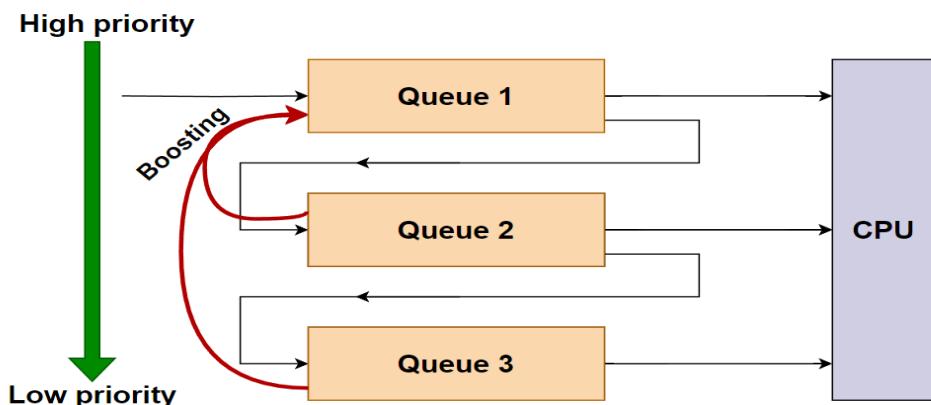
## Disadvantages

1. There is a risk of starvation for lower priority processes.
2. It is rigid in nature.

## Multilevel Feedback Queue Scheduling

**Multilevel feedback queue scheduling** it allows a process to move between the queue.

The diagram below illustrates the structure of an MLFQ. A new process is initially added to Queue 1, and if it fails to complete its execution, it moves to Queue 2 and vice versa. Once a process completes its execution, it is removed from the queue. Additionally, the MLFQ can utilize boosting to elevate low-level processes to higher queues.

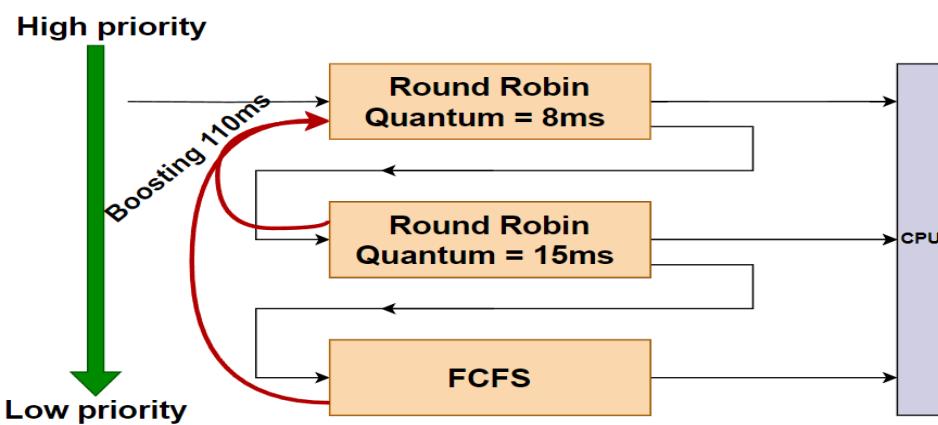


The steps involved in MLFQ scheduling when a process enters the system.

1. When a process enters the system, it is initially assigned to the highest priority queue.
2. The process can execute for a specific time quantum in its current queue.
3. If the process completes within the time quantum, it is removed from the system.
4. If the process does not complete within the time quantum, it is demoted to a lower priority queue and given a shorter time quantum.

5. This promotion and demotion process continues based on the behavior of the processes.
6. The high-priority queues take precedence over low-priority queues, allowing the latter processes to run only when high-priority queues are empty.
7. The feedback mechanism allows processes to move between queues based on their execution behavior.
8. The process continues until all processes are executed or terminated.

### **Example of MLFQ**



There are various scheduling algorithms that can be applied to each queue, including FCFS, shortest remaining time, and round robin. Now, we will explore a multilevel feedback queue configuration consisting of three queues.

- Queue 1 (Q1) follows a round robin schedule with a time quantum of 8 milliseconds.
- Queue 2 (Q2) also uses a round robin schedule with a time quantum of 15 milliseconds.
- Finally, Queue 3 (Q3) utilizes a first come, first serve approach.
- Additionally, after 110 milliseconds, all processes will be boosted to Queue 1 for high-priority execution.

#### ***The multilevel feedback queue scheduler has the following parameters:***

- The number of queues in the system.
- The scheduling algorithm for each queue in the system.
- The method used to determine when the process is upgraded to a higher-priority queue.
- The method used to determine when to demote a queue to a lower - priority queue.
- The method used to determine which process will enter in queue and when that process needs service.

#### **Advantages of Multilevel Feedback Queue Scheduling:**

- It is more flexible.
- It allows different processes to move between different queues.
- It prevents starvation by moving a process that waits too long for the lower priority queue to the higher priority queue.

**Disadvantages of Multilevel Feedback Queue Scheduling:**

- The selection of the best scheduler, it requires some other means to select the values.
- It produces more CPU overheads.
- It is the most complex algorithm.

## Race Condition

When more than one process is executing the same code or accessing the same memory or any shared variable in that condition there is a possibility that the output or the value of the shared variable is wrong so for that all the processes doing the race to say that my output is correct this condition known as a race condition.

### **Example :**

Suppose we have two process Producer and Consumer and counter variable is shared between these two. Initial Value of Counter is 5.

#### **Producer Routine**

```
register1 = counter  
register1 = register1 + 1  
counter= register1
```

#### **Consumer Routine**

```
register2 = counter  
register2 = register2 - 1  
counter= register2
```

Although both the producer and consumer routines above are correct separately, they may not function correctly when executed concurrently.

The concurrent execution of "counter++" and "counter--" is equivalent a sequential execution in which the lower-level statements presented previously are interleaved in some arbitrary order (but the order within each high-level statement is preserved). One such interleaving is To:

Consider this execution interleaving with "count = 5" initially:

S1: producer execute <code>register1 = counter</code>	{register1 = 5}
S2: producer execute <code>register1 = register1 + 1</code>	{register1 = 6}
S3: consumer execute <code>register2 = counter</code>	{register2 = 5}
S4: consumer execute <code>register2 = register2 - 1</code>	{register2 = 4}
S5: producer execute <code>counter = register1</code>	{counter = 6 }
S6: consumer execute <code>counter = register2</code>	{counter = 4}

Notice that we have arrived at the incorrect state "counter == 4", indicating that four buffers are full, when, in fact, five buffers are full. If we reversed the order of the statements at T4 and T5, we would arrive at the incorrect state "counter== 6". We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently. The only correct result, though, is counter == 5, which is generated correctly if the producer and consumer execute separately.

### Critical Section Problem

The regions of a program that try to access shared resources and may cause race conditions are called critical section. To avoid race condition among the processes, we need to assure that only one process at a time can execute within the critical section.

Process Synchronization is the coordination of execution of multiple processes in a multi-process system to ensure that they access shared resources in a controlled and predictable manner. On the basis of synchronization, processes are categorized as one of the following two types:

- **Independent Process:** The execution of one process does not affect the execution of other processes.
- **Cooperative Process:** A process that can affect or be affected by other processes executing in the system.

Process synchronization problem arises in the case of Cooperative processes also because resources are shared in Cooperative processes.

The critical-section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the entry Section The critical section may be followed by an exit Section. The remaining code is the remainder Section.

The general structure of a typical process Pi is shown in

```

do {
    entry section
    critical section
    exit section
    remainder section
} while (TRUE);

```

**Any solution to the critical section problem must satisfy three requirements:**

- **Mutual Exclusion:** If a process is executing in its critical section, then no other process is allowed to execute in the critical section.
- **Progress:** If no process is in the critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely. Essentially, the system must guarantee that every process can eventually enter its critical section.
- **Bounded Waiting:** There should be a **bound or a limit** on the number of times a particular **process can enter the critical section**. It should **not happen that the same process is taking up critical section every time** resulting in starvation of other processes. In this case, other processes would have to wait for an **infinite** amount of time, this should not happen.

### Synchronization Hardware

Some times the problems of the Critical Section are also resolved by hardware. The hardware solution is as follows:

1. Disabling Interrupts
2. Test and Set
3. Swap

## 4. Unlock and Lock

### ***Disabling Interrupts***

The critical-section problem could be solved simply in a uniprocessor environment if we could prevent interrupts from occurring while a shared variable was being modified. In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable. This is often the approach taken by nonpreemptive kernels. Unfortunately, this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, as the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases.

### ***Test and Set***

***Test and set algorithm*** uses a boolean variable '**lock**' which is initially initialized to false. This lock variable determines the entry of the process inside the critical section of the code.

- However, if processes with intent of the critical section, then it changes the boolean lock = false;

```
boolean TestAndSet(boolean *target)
```

```
{
```

```
    boolean returnValue = *target;
```

```
    *target = true;
```

```
    return returnValue;
```

```
}
```

```
while(1)
```

```
{
```

```
    while(TestAndSet(&lock));
```

```
        CRITICAL SECTION CODE;
```

```
        lock = false;
```

REMAINDER SECTION CODE;

}

- In the above algorithm the TestAndSet () function takes a boolean value and returns the same value. TestAndSet () function sets the lock variable to true.
- When lock variable is initially false the TestAndSet(lock) condition checks for TestAndSet(false). As TestAndSet function returns the same value as its argument, TestAndSet(false) returns false. Now, while loop while (TestAndSet(lock)) breaks and the process enters the critical section.
- As one process is inside the critical section and lock value is now 'true', if any other process tries to enter the critical section, then the new process checks for while (TestAndSet(true)) which will return **true** inside while loop and as a result the other process keeps executing the while loop.
- As no queue is maintained for the processes stuck in the while loop, bounded waiting is not ensured.

### ***Swap***

**Swap function** uses two boolean variables lock and key. Both lock and key variables are initially initialized to false

```
boolean lock = false;  
individual key = false;  
void swap(boolean *a, boolean *b)  
{  
    boolean temp = *a;  
    *a = *b;  
    *b = temp;  
}  
while(1)  
{
```

```

key=true;

while(key){

    swap(&lock,&key);

}

CRITICAL SECTION CODE

lock = false;

REMAINDER SECTION CODE

}

```

- In the code above when a process P1 enters the critical section of the program it first executes the while loop. As key value is set to true just before the for loop so swap(lock, key) swaps the value of lock and key. Lock becomes true and the key becomes false. In the next iteration of the while loop breaks and the process, P1 enters the critical section.
- The value of lock and key when P1 enters the critical section is lock = true and key = false.
- Let's say another process, P2, tries to enter the critical section while P1 is in the critical section. Let's take a look at what happens if P2 tries to enter the critical section.
- **key is set to true** again after the first while loop is executed i.e while(1). Now, the second while loop in the program i.e while(key) is checked. As key is true the process enters the second while loop. swap(lock, key) is executed again. as both key and lock are true so after swapping also both will be true. So, the while keeps executing and the process P2 keeps running the while loop until Process P1 comes out of the critical section and makes **lock false**.
- When Process P1 comes out of the critical section the value of lock is again set to false so that other processes can now enter the critical section.
- When a process is inside the critical section than all other incoming process trying to enter the critical section is not maintained in any order or queue. So any process out of all the waiting process can get the chance to enter the critical section as the lock becomes false. So, there may be a process that may wait indefinitely. So, **bounded waiting is not ensured in Swap algorithm also**.

### ***Unlock and lock***

- Unlock and Lock Algorithm uses TestAndSet to regulate the value of lock but it adds another value, waiting[i], for each process which checks whether or not a process has been waiting.
- A ready queue is maintained with respect to the process in the critical section. All the processes coming in next are added to the ready queue with respect to their process number, not necessarily sequentially.
- Once the ith process gets out of the critical section, it does not turn lock to false so that any process can avail the critical section now, which was the problem with the previous algorithms. Instead, it checks if there is any process waiting in the queue.
- The queue is taken to be a circular queue. j is considered to be the next process in line and the while loop checks from jth process to the last process and again from 0 to (i-1)th process if there is any process waiting to access the critical section.
- If there is no process waiting then the lock value is changed to false and any process which comes next can enter the critical section.
- If there is, then that process' waiting value is turned to false, so that the first while loop becomes false and it can enter the critical section. This ensures bounded waiting. So the problem of process synchronization can be solved through this algorithm.

```
// Shared variable lock initialized to false  
// and individual key initialized to false  
boolean lock = false;  
boolean key = false;  
boolean waiting[];  
boolean TestAndSet(boolean *target)  
{  
    boolean returnValue = *target;  
    * target = true;
```

```
    return returnValue;  
}  
  
while(1){  
    waiting[i] = true;  
    key = true;  
    while(waiting[i] && key){  
        key = TestAndSet(&lock);  
    }  
    CRITICAL SECTION CODE  
    j = (i+1) % n;  
    while(j != i && !waiting[j])  
        j = (j+1) % n;  
    if(j == i)  
        lock = false;  
    else  
        waiting[j] = false;  
    REMAINDER SECTION CODE  
}
```

## Peterson's Solution

Peterson's solution is a software-based solution, developed by Gary L Peterson in 1981, to the Critical Section problem. Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.

**There are two variables that the two processes share:**

- `turn(int)`: The variable `turn` indicates whose turn it is to enter the critical section. If `turn == i`, then  $P_i$  is allowed to enter their critical section.
- `flag (boolean)`: The `flag` array indicates if a process is ready to enter the critical section. If `flag[i] = true`, then it means that process  $P_i$  is ready.

`boolean flag [2] = {false, false}`

`int turn;`

**Algorithm for Process  $P_i$**

```
do
{
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        // Critical Section
    flag[i] = false ;
    // Remainder Section
} while (true);
```

**Algorithm for Process  $P_j$**

```
do
{
    flag[j] = true ;
    turn = i ;
    while (flag[i] && turn == i)
        // Critical Section
    flag[j] = false ;
    // Remainder Section
} while (true);
```

.

### **Mutual exclusion is preserved.**

To prove this, we note that each  $P_i$  enters its critical section only if either flag  $[j] == \text{false}$  or  $\text{turn} == i$ . Also note that, if both processes can be executing in their critical sections at the same time, then  $\text{flag}[0] == \text{flag}[1] == \text{true}$ . These two observations imply that  $P_0$  and  $P_1$  could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both. Hence, one of the processes -say,  $P_j$  -must have successfully executed the while statement, whereas  $P_i$ ; had to execute at least one additional statement ("turn== j"). However, at that time, flag  $[j] == \text{true}$  and  $\text{turn} == j$ , and this condition will persist as long as  $P_j$  is in its critical section; as a result, mutual exclusion is preserved.

### **The progress requirement is satisfied & bounded-waiting requirement is met.**

- To prove these two we note that a process  $P_i$ ; can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag  $[j] == \text{true}$  and  $\text{turn} == j$ ; this loop is the only one possible.
- If  $P_j$  is not ready to enter the critical section, then flag  $[j] == \text{false}$ , and  $P_i$  can enter its critical section.
- If  $P_j$  has set flag  $[j]$  to true and is also executing in its while statement, then either  $\text{turn} == i$  or  $\text{turn} == j$ . If  $\text{turn} == i$ , then  $P_i$ ; will enter the critical section. If  $\text{turn} == j$ , then  $P_j$  will enter the critical section.
- However, once  $P_j$  exits its critical section, it will reset flag  $[j]$  to false, allowing  $P_i$  to enter its critical section. If  $P_j$  resets flag  $[j]$  to true, it must also set turn to  $i$ . Thus, since  $P_i$  does not change the value of the variable turn while executing the while statement,  $P_i$  will enter the critical section (progress) after at most one entry by  $P_j$  (bounded waiting).

### **Advantages:**

Simple to understand and implement.

Guarantees mutual exclusion, progress, and bounded waiting under certain conditions.

### **Disadvantages:**

Limited to two processes.

Requires busy waiting, which can lead to wastage of CPU resources.

Not scalable to more than two processes without additional mechanisms.

## Semaphores

In 1965, Dijkstra proposed a new and very significant technique for managing concurrent processes by using the value of a simple non negative integer variable to synchronize the progress of interacting processes. This non negative integer variable is called a **semaphore**. So, it is basically a synchronizing tool and is accessed only through two low standard atomic operations, **wait** and **signal** designated by P(S) and V(S) respectively.

Entry to the critical section is controlled by the wait operation and exit from a critical region is taken care by signal operation.

### Wait Operation (P):

```
Wait(S)
{
    while (S<=0); // no operation
    S--;
}
```

- When a process/thread wants to access a shared resource, it first performs a "wait" operation on the semaphore associated with that resource.
- If the semaphore value is positive (greater than 0), it decrements the value by 1 and continues accessing the resource.
- If the semaphore value is zero, indicating that the resource is currently being used by another process/thread, the process/thread is blocked or put to sleep until the semaphore becomes available.

### Signal Operation (V):

```
Signal(S)
{
    S++;
}
```

- When a process/thread finishes using a shared resource, it performs a "signal" operation on the semaphore, which increments the semaphore value by 1.

- This operation indicates that the resource is now available for other processes/threads waiting on it. If there are any blocked processes/threads waiting for the semaphore, one of them is awakened and granted access to the resource.

### **Types of Semaphores-**

There are mainly two types of semaphores-

1. Counting Semaphores
2. Binary Semaphores or Mutexes

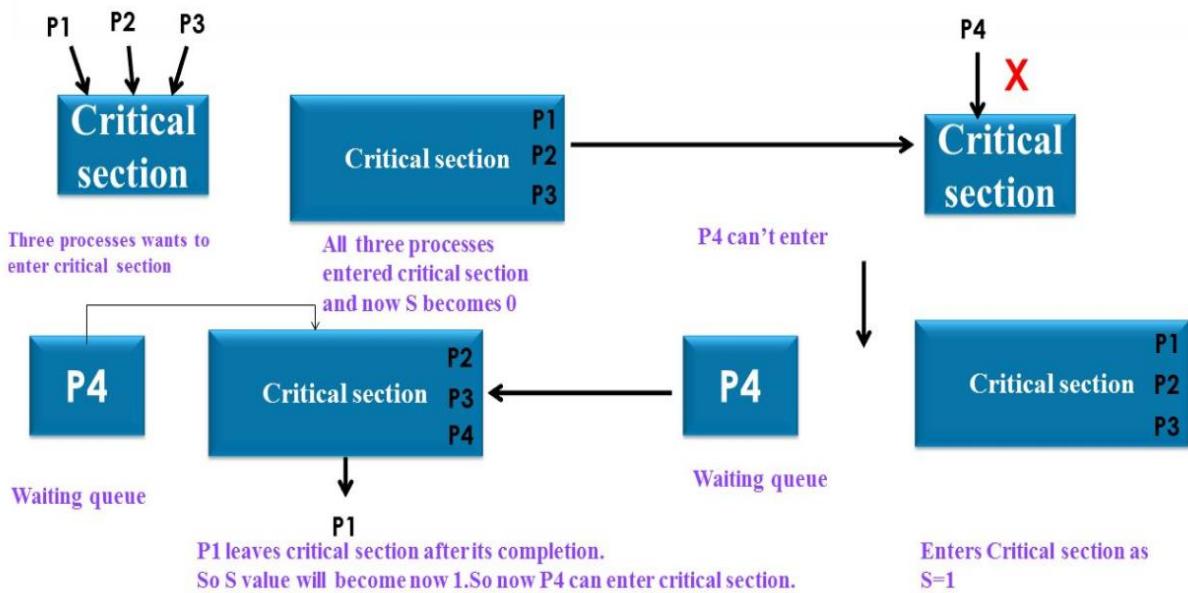
```
do {
    wait (S) ;
    // critical section
    signal(S);
    //remainder section
} while (TRUE)
```

### **Counting Semaphores**

1. Initialize the counting semaphore with a value that represents the maximum number of resources that can be accessed simultaneously.
2. When a process attempts to access the shared resource, it first attempts to acquire the semaphore using the `wait()` or `P()` function.
3. The semaphore value is checked. If it is greater than zero, the process is allowed to proceed and the value of the semaphore is decremented by one. If it is zero, the process is blocked and added to a queue of waiting processes.
4. When a process finishes accessing the shared resource, it releases the semaphore using the `signal()` or `V()` function.
5. The value of the semaphore is incremented by one, and any waiting processes are unblocked and allowed to proceed.
6. Multiple processes can access the shared resource simultaneously as long as the value of the semaphore is greater than zero.

7. The counting semaphore provides a way to manage access to shared resources and ensure that conflicts are avoided, while also allowing multiple processes to access the resource at the same time.

- Assuming Initial Value of Semaphore=3

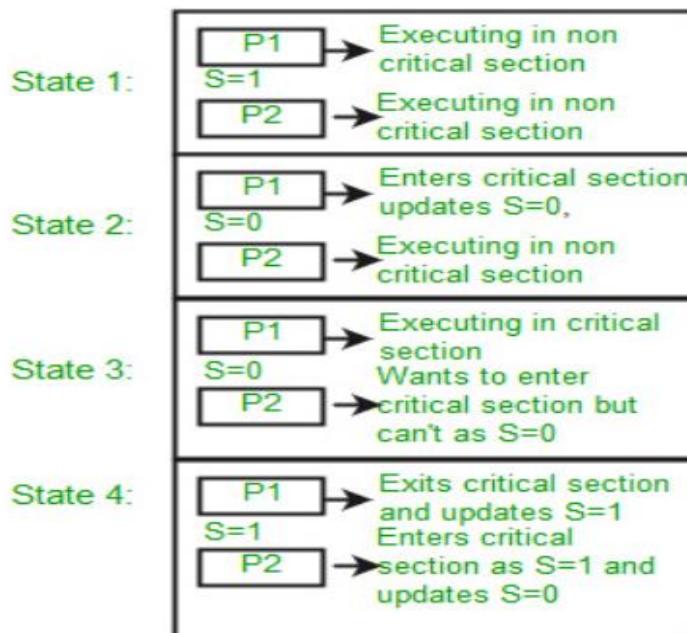


## Binary Semaphore

This is also known as a mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes.

### Implementation of Binary Semaphore

- Now, let us see how it implements mutual exclusion. Let there be two processes P1 and P2 and a semaphore s is initialized as 1.
- Now if suppose P1 enters in its critical section then the value of semaphore s becomes 0.
- Now if P2 wants to enter its critical section then it will wait until  $s > 0$ , this can only happen when P1 finishes its critical section and calls V operation on semaphore s.
- This way mutual exclusion is achieved. Look at the below image for details which is a Binary semaphore.



Criteria	Binary Semaphore	Counting Semaphore
<b>Definition</b>	A Binary Semaphore is a semaphore whose integer value range over 0 and 1.	A counting semaphore is a semaphore that has multiple values of the counter. The value can range over an unrestricted domain.
<b>Structure Implementation</b>	<pre>typedef struct {     int semaphore_variable; } binary_semaphore;</pre>	<pre>typedef struct {     int semaphore_variable;     Queue list; //A queue to store the list of task }counting_semaphore;</pre>
<b>Representation</b>	0 means that a process or a thread is accessing the critical section, other process should wait for it to exit the critical section. 1 represents the critical section is free.	The value can range from 0 to N, where N is the number of process or thread that has to enter the critical section.
<b>Mutual Exclusion</b>	Yes, it guarantees mutual exclusion, since just one process or thread can enter the critical section at a time.	No, it doesn't guarantee mutual exclusion, since more than one process or thread can enter the critical section at a time.

<b>Bounded wait</b>	No, it doesn't guarantees bounded wait, as only one process can enter the critical section, and there is no limit on how long the process can exist in the critical section, making another process to starve.	Yes, it guarantees bounded wait, since it maintains a list of all the process or threads, using a queue, and each process or thread get a chance to enter the critical section once. So no question of starvation.
<b>Starvation</b>	No waiting queue is present then FCFS (first come first serve) is not followed so,starvation is possible and busy wait present	Waiting queue is present then FCFS (first come first serve) is followed so,no starvation hence no busy wait.
<b>Number of instance</b>	Used only for a single instance of resource type R.it can be usedonly for 2 processes.	Used for any number of instance of resource of type R.it can be used for any number of processes.

- The main disadvantage of the semaphore is that it requires busy waiting. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a spinlock because the process spins while waiting for the lock.
- To overcome the need for busy waiting, we can modify the definition of wait () and Signal () semaphore operations. When the process executes the wait () operation and finds that the semaphore value is not positive it must wait. Rather than engaging in busy waiting the process can block itself.
- The block operation places a process into a waiting queue associated with the semaphore and the state of the process is switched to the waiting state. The process that is blocked waiting on a semaphore S should be restarted when some other process executes a signal () operation, the process is restarted by a wakeup () operation.

### **Definition of semaphore as a C**

struct typedef struct

```

{
int value;
struct process *list;
} semaphore;



- Each semaphore has an integer value and a list of processes list.
- When a process must wait on a semaphore, it is added to the list of processes.
- A signal() operation remove one process from the list of waiting processes and awakens that process.

```

***Wait () operation can be defined as***

```

Wait (semaphore *s)
{
S->value--;
If (S->Value<0)
{
Add this Process to s->value list;
block ();
}

```

***Signal operation can be defined as***

```

Signal (semaphore *S)
{
S->value++;
If (S-> value <=0)
{
Remove a process P from S-> list;
Wakeup (P);
}

```

The block () operation suspends the process that invokes it. The wakeup () operation resumes the execution of a blocked process P.

### **Deadlock and Starvation:**

The implementation of semaphore with a waiting queue may result in a situation where two more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. When such a state is reached that process are said to be deadlocked.

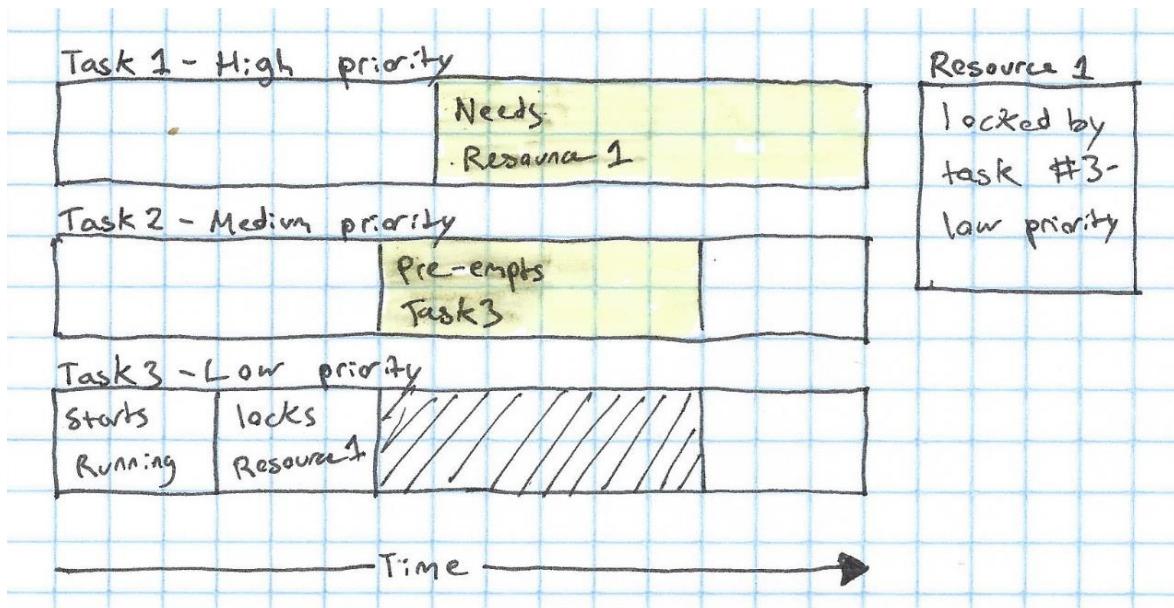
P0	P1
Wait(S);	wait(Q);
Wait(Q);	Wait(S);
.....	
Signal( S);	Signal (Q);
Signal (Q);	Signal (S);

P0 executes wait (S) and then P1 executes wait (Q). When p0 executes wait (Q), it must wait until p1 executes Signal (Q) in the same way P1 must wait until P0 executes signal (S). So p0 and p1 are deadlocked. The other problem related to deadlocks is indefinite blocking or starvation, a situation where processes wait indefinitely within the semaphore

P0 executes wait (S) and then P1 executes wait (Q). When p0 executes wait (Q), it must wait until p1 executes Signal (Q) in the same way P1 must wait until P0 executes signal (S). So p0 and p1 are deadlocked. The other problem related to deadlocks is indefinite blocking or starvation, a situation where processes wait indefinitely within the semaphore.

### **Priority inversion**

**Priority inversion** is a phenomenon that can occur in an operating system where a higher-priority task is blocked because it is waiting for a lower-priority task to release a resource it needs. This can happen in a system where tasks or threads have different priorities, and a lower-priority task is currently holding a resource that a higher-priority task needs to execute.



For example, consider a system with three tasks: 1, 2, and 3. Task 1 has the highest priority, task 2 has medium priority, and task 3 has the lowest priority. If task 3 is currently in Critical Section and task 1 needs to enter into critical section in that case task 1 may be blocked until task 2 completes, even though task 1 has a higher priority than task 2. This is known as Bounded Priority Inversion.

In meantime Task2 just wants a normal execution and does not want to enter into critical section. Then according to priority scheduling task 2 can pre-empt task 1 (but still task 1 holds the lock of Critical section) and continue the normal execution. Now task 1 has to wait for completion of task 2 and task 1 though the priorities of the both are lower than the Task1. This is known as Unbounded Priority Inversion.

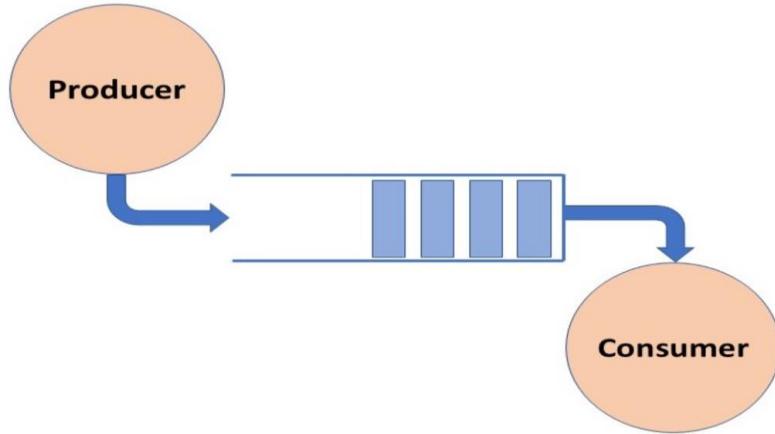
Priority inversion can cause significant problems in real-time systems, where the timely execution of high-priority tasks is critical. If a high-priority task is blocked for an extended period due to priority inversion, it can cause delays and potentially impact the system's overall performance.

One of the solutions to this problem is *Priority Inheritance*. In *Priority Inheritance*, when L is in the critical section, L inherits the priority of H at the time when H starts pending for the critical section. By doing so, M doesn't interrupt L and H doesn't wait for M to finish. Please note that inheriting of priority is done temporarily i.e. L goes back to its old priority when L comes out of the critical section.

## Classical Problems of Synchronization

Bounded Buffer Problem(Producer - consumer problem) is example classic problems of synchronization. Producer process produce data item that consumer process consumes later.

- Buffer is used between producer and consumer. Buffer size may be fixed or variable. The producer portion of the application generates data and stores it in a buffer, and the consumer reads data from the buffer.



- The producer should produce data only when the buffer is not full. In case it is found that the buffer is full, the producer is not allowed to store any data into the memory buffer.
- Data can only be consumed by the consumer if and only if the memory buffer is not empty. In case it is found that the buffer is empty, the consumer is not allowed to use any data from the memory buffer.
- Accessing memory buffer should not be allowed to producer and consumer at the same time.

The solution to the Producer-Consumer problem involves three *semaphore* variables.

- **semaphore Full:** Tracks the space filled by the Producer process. It is initialized with a value of 0 as the buffer will have 0 filled spaces at the beginning
- **semaphore Empty:** Tracks the empty space in the buffer. It is initially set to **buffer\_size** as the whole buffer is empty at the beginning.
- **semaphore mutex:** Used for mutual exclusion so that only one process can access the shared buffer at a time..

At any particular time, the current value of empty denotes the number of vacant slots in the buffer, while full denotes the number of occupied slots.

```
do
{
    // process will wait until the empty > 0 and further decrement of 'empty'
    wait(empty);
    // To acquire the lock
    wait(mutex);

    /* Here we will perform the insert operation in a particular slot */

    // To release the lock
    signal(mutex);
    // increment of 'full'
    signal(full);
}
while(TRUE)
```

- When we look at the above code for a producer, we can see that it first waits until at least one slot is vacant.
- `wait(empty)` decreases the value of the semaphore variable "empty" by one, indicating that when the producer produces anything, the value of the empty space in the buffer decreases. If the buffer is full, or the value of the semaphore variable "empty" is 0, the program will stop and no production will take place.
- `wait(mutex)` sets the semaphore variable "mutex" to zero, preventing any other process from entering the critical section.
- The buffer is then locked, preventing the consumer from accessing it until the producer completes its function.
- `signal(mutex)` is being used to mark the semaphore variable "mutex" to "1" so that other processes can arrive into the critical section though because the production is finished and the insert operation is also done.
- So, After the producer has filled a slot in the buffer, the lock is released.

- signal(full) is utilized to increase the semaphore variable "full" by one because after inserting the data into the buffer, one slot is filled in the buffer and the variable "full" must be updated.

○

```

do
{
    // need to wait until full > 0 and then decrement the 'full'
    wait(full);
    // To acquire the lock
    wait(mutex);

    /* Here we will perform the remove operation in a particular slot */

    // To release the lock
    signal(mutex);
    // increment of 'empty'
    signal(empty);
}
while(TRUE);

```

- The consumer waits until the buffer has at least one full slot.
- wait(full) is used to reduce the semaphore variable "full" by one since the variable "full" must be reduced by one of the consumers consuming some data.
- wait(mutex) sets the semaphore variable "mutex" to "0", preventing any other processes from entering the critical section.
- And soon after that, the consumer then acquires a lock on the buffer.
- The consumer then completes the data removal operation by removing data from one of the filled slots.
- So because the consumption and remove operations are complete, signal(mutex) is being used to set the semaphore variable "mutex" to "1" so that other processes can enter the critical section now.
- The lock is then released by the consumer.

- Because one slot space in the buffer is released after extracting the data from the buffer, signal(empty) is used to raise the variable "empty" by one.

### Reader's writer problem

The readers-writers problem relates to an object such as a file that is shared between multiple processes. . The problem statement is, if a database or file is to be shared among several concurrent process, there can be broadly 2 types of users

- Readers – Reader are those processes/users which only read the data
- Writers – Writers are those processes which also write, that is, they change the data .

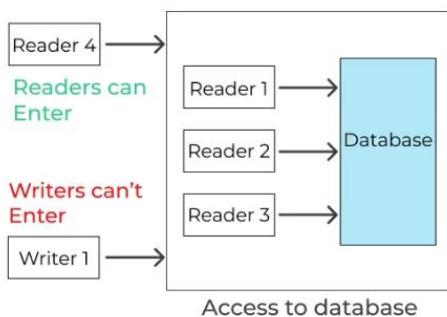
It is allowed for 2 or more readers to access shared data, simultaneously as they are not making any change and even after the reading the file format remains the same.

But if one writer(Say w1) is editing or writing the file then it should locked and no other writer(Say w2) can make any changes until w1 has finished writing the file.

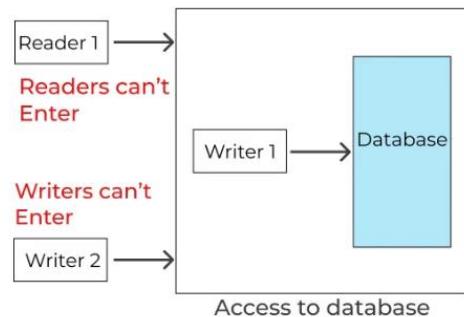
Writers are given to be exclusive access to shared database while writing to database. This is called Reader's writer problem.

### Readers-Writers Problem in Operating System

When Readers are accessing the Database



When Writers are accessing the Database



### **Variables used –**

- Mutex – mutex (used for mutual exclusion, when readcount is changed) initialised as 1
- Semaphore – wrt (used by both readers and writers) initialised as 1
- readers\_count – Counter of number of people reading the file initialised as 0

### **Functions –**

There are two functions –

1. wait() – performs as –, which basically decrements value of semaphore
2. signal() – performs as ++. which basically increments value of semaphore

### **Reader Process**

```
while (TRUE)
{
    // Acquire lock
    wait(m);
    readCount++;
    if (readCount == 1)
    {
        wait(w);
    }

    // Release lock
    signal(m);

    wait(m);
    readCount--;
    if (readCount == 0)
    {
        signal(w);
    }

    // Release lock
    signal(m);
}
```

### **Writer Process**

```
do {
```

```

wait(wrt);
// writing is performed
signal(wrt);
} while (TRUE);

```

### Dining philosopher's problem

The dining philosopher's problem is the classical problem of synchronization which says that Five philosophers are sitting around a circular table and their job is to think and eat alternatively. A bowl of noodles is placed at the center of the table along with five chopsticks for each of the philosophers. To eat a philosopher needs both their right and a left chopstick. A philosopher can only eat if both immediate left and right chopsticks of the philosopher is available.



### Solution of Dining Philosophers Problem

A solution of the Dining Philosophers Problem is to use a semaphore to represent a chopstick. A chopstick can be picked up by executing a wait operation on the semaphore and released by executing a signal semaphore.

The structure of the chopstick is shown below –

```
semaphore chopstick [5];
```

Initially the elements of the chopstick are initialized to 1 as the chopsticks are on the table and not picked up by a philosopher.

The structure of a random philosopher i is given as follows –

```
do {  
    wait( chopstick[i] );  
    wait( chopstick[ (i+1) % 5 ] );  
    . . .  
    . EATING THE RICE  
    . . .  
    signal( chopstick[i] );  
    signal( chopstick[ (i+1) % 5 ] );  
    . . .  
    . THINKING  
    . . .  
} while(1);
```

In the above structure, first wait operation is performed on chopstick[i] and chopstick[ (i+1) % 5]. This means that the philosopher i has picked up the chopsticks on his sides. Then the eating function is performed.

After that, signal operation is performed on chopstick[i] and chopstick[ (i+1) % 5]. This means that the philosopher i has eaten and put down the chopsticks on his sides. Then the philosopher goes back to thinking.

### **Difficulty with the solution**

The above solution makes sure that no two neighboring philosophers can eat at the same time. But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever. Then none of them can eat and deadlock occurs.

Some of the ways to avoid deadlock are as follows –

- There should be at most four philosophers on the table.

- An even philosopher should pick the right chopstick and then the left chopstick while an odd philosopher should pick the left chopstick and then the right chopstick.
- A philosopher should only be allowed to pick their chopstick if both are available at the same time.

## Monitors

Semaphores can be very useful for solving concurrency problems, **but only if programmers use them properly**. If even one process fails to abide by the proper use of semaphores, either accidentally or deliberately, then the whole system breaks down.

- Suppose that a process interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed, resulting in the following execution:

```
signal(mutex);
```

critical section

```
wait(mutex);
```

In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections. Note that this situation may not always be reproducible.

- Suppose that a process replaces signal (mutex) with wait (mutex). That is, it executes

```
wait(mutex);
```

critical section

```
wait(mutex);
```

In this case, a deadlock will occur.

- Suppose that a process omits the wait (mutex), or the signal (mutex), or both.

In this case, either mutual exclusion is violated or a deadlock will occur.

*For this reason a higher-level language construct has been developed, called monitors.*

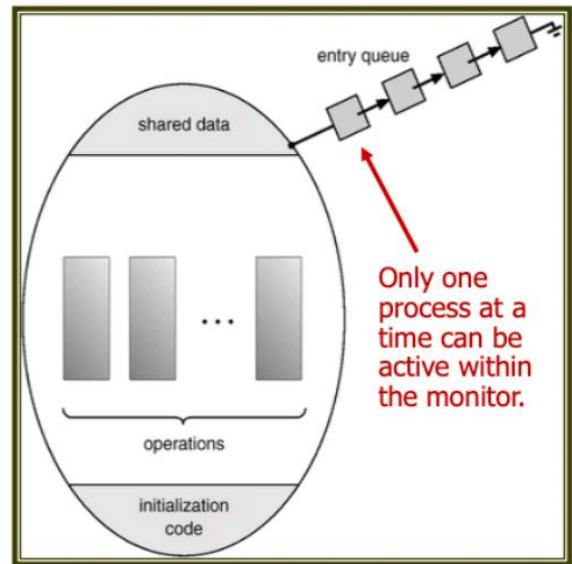
### Monitor Usage

A monitor is essentially a class, in which all data is private, and with the special restriction that only one method within any given monitor object may be active at the same time. An additional restriction is that monitor methods may only access the shared data within the monitor and any data passed to them as parameters. I.e. they cannot access any data external to the monitor.

```

monitor monitor-name
{
    shared variable declarations
    procedure body P1 (...) {
        ...
    }
    procedure body Pn (...) {
        ...
    }
    {
        initialization code
    }
}

```



In order to fully realize the potential of monitors, we need to introduce one additional new data type, known as a **condition**. A variable of type condition has only two legal operations, wait and signal. I.e. if X was defined as type condition, then legal operations would be X.wait( ) and X.signal( ).

The wait operation blocks a process until some other process calls signal, and adds the blocked process onto a list associated with that condition.

The signal process does nothing if there are no processes waiting on that condition. Otherwise it wakes up exactly one process from the condition's list of waiting processes.

**Signal and wait** - When process P issues the signal to wake up process Q, P then waits, either for Q to leave the monitor or on some other condition.

**Signal and continue** - When P issues the signal, Q waits, either for P to exit the monitor or for some other condition.

### Dining Philosophers Solution using Monitors

Monitors are used because they give a deadlock free solution to the Dining Philosophers problem. It is used to gain access over all the state variables and condition variables. After implying monitors, it imposes a restriction that a philosopher may pickup his chopsticks only if both of them are available at the same time.

To code the solution, we need to distinguish among three states in which may find a philosopher.

- THINKING
- HUNGRY
- EATING

### Example

Here is implementation of the Dining Philosophers problem using Monitors –

```
monitor DiningPhilosophers {
```

```
    enum {THINKING, HUNGRY, EATING} state[5];  
    condition self[5];  
  
    void pickup(int i) {  
        state[i] = HUNGRY;  
        test(i);  
        if (state[i] != EATING) {  
            self[i].wait();  
        }  
    }  
  
    void putdown(int i) {  
        state[i] = THINKING;  
        test((i + 4) % 5);  
        test((i + 1) % 5);  
    }  
  
    void test(int i) {  
        if (state[(i + 4) % 5] != EATING &&  
            state[i] == HUNGRY &&  
            state[(i + 1) % 5] != EATING) {  
            state[i] = EATING;  
            self[i].signal();  
        }  
    }  
}
```

```

}

initialization code() {
    for(int i=0;i<5;i++)
        state[i] = THINKING;
}

}

```

Before start eating, each philosopher must invoke the pickup() operation. It indicates that the philosopher is hungry, means that the process wants to use the resource. It also set the state to EATING in test() only if the philosopher's left and right neighbors are not eating. If the philosopher is unable to eat, then wait() operation is invoked. After the successful completion of the operation, the philosopher may now eat.

Keeping that in mind, the philosopher now invokes the putdown() operation. After leaving forks, it checks on his neighbors. If they are HUNGRY and both of its neighbors are not EATING, then invoke signal() and offer them to eat.

Thus a philosopher must invokes the pickup() and putdown() operations simultaneously which ensures that no two neighbors are eating at the same time, thus achieving mutual exclusion. Thus, it prevents the deadlock. But there is a possibility that one of the philosopher may starve to death.

## **Implementing Monitor using Semaphore**

**Let's implement a monitor mechanism using semaphores.**

Following is a step-by-step implementation:

**Step 1:** Initialize a semaphore mutex to 1.

**Step 2:** Provide a semaphore mutex for each monitor.

**Step 3:** A process must execute wait (mutex) before entering the monitor and must execute signal (mutex) after leaving the monitor.

**Step 4:** Since a signaling process must wait until the resumed process either leaves or waits, introduce an additional semaphore, S, and initialize it to 0.

**Step 5:** The signaling processes can use S to suspend themselves. An integer variable S\_count is also provided to count the number of processes suspended next. Thus, each external function Fun is replaced by

*wait (mutex);*

*body of Fun*

*if (S\_count > 0)*

*signal (S);*

*else*

*signal (mutex);*

Mutual exclusion within a monitor is ensured.

**Let's see how condition variables are implemented.**

**Step 1:** x is condition.

**Step 2:** Introduce a semaphore x\_num and an integer variable x\_count.

**Step 3:** Initialize both semaphores to 0.

x.wait() is now implemented as: x.wait()

*x\_count++;*

*if (S\_count > 0)*

*signal (S);*

*else*

*signal (mutex);*

*wait (x\_num);*

*x\_count--;*

The operation x.signal () can be implemented as

*if (x\_count > 0) {*

*S\_count++;*

*signal (x\_num);*

*wait (S);*

*S\_count--;*}

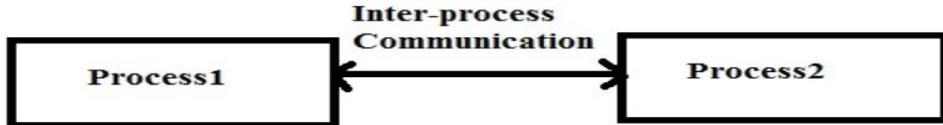
## Main Differences between the Semaphore and Monitor

Some of the main differences are as follows:

1. A semaphore is an integer variable that allows many processes in a parallel system to manage access to a common resource like a multitasking OS. On the other hand, a monitor is a synchronization technique that enables threads to mutual exclusion and the **wait()** for a given condition to become true.
2. When a process uses shared resources in semaphore, it calls the **wait()** method and blocks the resources. When it wants to release the resources, it executes the **signal()**. In contrast, when a process uses shared resources in the monitor, it has to access them via procedures.
3. Semaphore is an integer variable, whereas monitor is an abstract data type.
4. In semaphore, an integer variable shows the number of resources available in the system. In contrast, a monitor is an abstract data type that permits only a process to execute in the crucial section at a time.
5. Semaphores have no concept of condition variables, while monitor has condition variables.
6. A semaphore's value can only be changed using the **wait()** and **signal()**. In contrast, the monitor has the shared variables and the tool that enables the processes to access them.

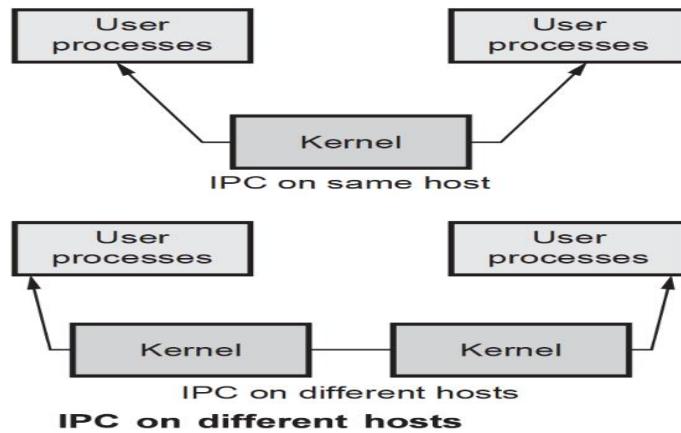
## Inter Process Communication (IPC) using Pipes

Inter Process Communication (IPC) is a mechanism that involves communication of one process with another process.



Communication can be of two types

- Between related processes initiating from only one process, such as parent and child processes.
- Between unrelated processes, or two or more different processes.



## Need for IPC

- Sharing of information: Inter-process communication helps in sharing information when several processes try to access a particular file concurrently. It is, therefore, necessary for the processes to cooperate with each other so that the information provided is correct.
- Speed: Dividing a big task into smaller tasks and then running them concurrently can speed up the entire functioning of the system. We need inter-process communication in such a case. The multiprocessor environment is a very good example of this.
- Modularity: When a system is divided into modules, such modules may require information sharing.
- Convenience: Allowing multiple processes to run simultaneously provides a better experience to any user

In inter-process communication (IPC), pipes are a fundamental mechanism that allows processes to communicate data with each other. One of the ways to achieve Inter Process Communication is through pipes. It is nothing but a mechanism using which the output of a single process is directed into the input of another process.

**Advantages:**

1. Simplicity: Pipes are a simple and straightforward way for processes to communicate with each other.
2. Efficiency: Pipes are an efficient way for processes to communicate, as they can transfer data quickly and with minimal overhead.
3. Reliability: Pipes are a reliable way for processes to communicate, as they can detect errors in data transmission and ensure that data is delivered correctly.
4. Flexibility: Pipes can be used to implement various communication protocols, including one-way and two-way communication.

**Disadvantages:**

1. Limited capacity: Pipes have a limited capacity, which can limit the amount of data that can be transferred between processes at once.
2. Unidirectional: In a unidirectional pipe, only one process can send data at a time, which can be a disadvantage in some situations.
3. Synchronization: In a bidirectional pipe, processes must be synchronized to ensure that data is transmitted in the correct order.
4. Limited scalability: Pipes are limited to communication between a small number of processes on the same computer, which can be a disadvantage in large-scale distributed systems

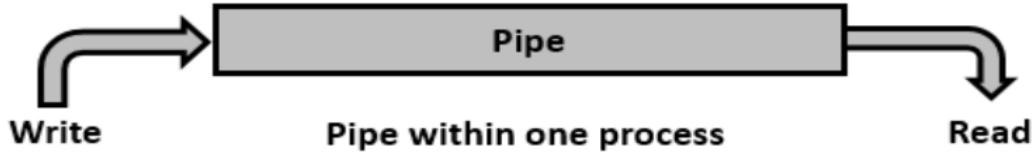
**Pipes can be categorized into two types: unnamed (or anonymous) pipes and named pipes.**

***Unnamed Pipes or Traditional Pipes***

- Unnamed pipes provide a way for processes to communicate in a unidirectional fashion, meaning data flows in only one direction. Unnamed pipes are used when communication needs to occur between a parent and its child process. The lifetime of an unnamed pipe is usually tied to the duration of the processes using it; it does not persist after the processes .

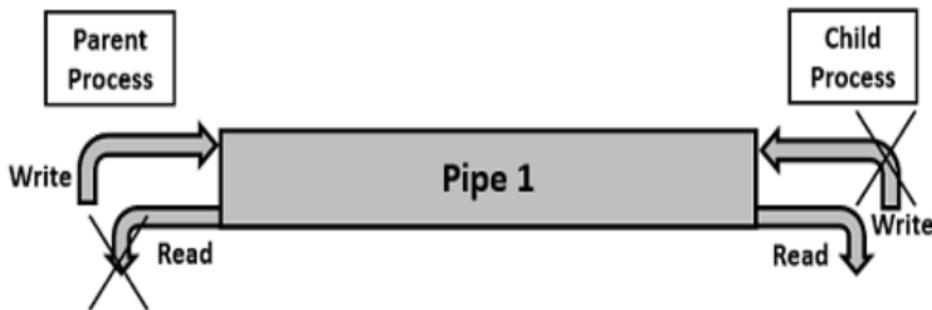
A usual pipe file has an input end and an output end. We can read from the output end and write into a pipe from the input end. There is an array of two pointers present in a pipe descriptor. One

pointer is for storing the input end of the pipe and the other pointer is for storing the output end of the pipe.



Let us assume that two processes, Process One and Process Two, want to communicate with each other using pipes.

- The first Process should keep its write end of the pipe open and the read end closed.
- Process Two should keep its read end open and close its write end. On successful creation, a pipe is given a fixed size in bytes.



- If the pipe is not full and some process wants to write into the pipe, the write request is executed spontaneously. However, if the pipe is full when the process wants to write into the pipe, it is blocked until the state of the pipe changes.
- The same things happen if a reading process tries to read more bytes than what is currently in the pipe and gets blocked. On the other hand, the reading process is executed without any issues if it tries to read-only sufficient bytes that are there in the pipe. Only a single process can access a pipe at any particular point in time.

### ***System Calls***

- **int pipe (int fd[2]);** The input parameter is an array of two file descriptors  $fd[0]$  and  $fd[1]$ . A file descriptor is in fact an integer value. The system call returns a  $-1$  in case of a failure. If the call is successful, it will return two integer values which are file descriptors  $fd[0]$ & $fd[1]$ . In pipes,  $fd[0]$  and  $fd[1]$  are used for reading and writing, respectively.
- **Write(fd,buf,count)**

- **fd**: The file descriptor to which data is to be written. When using unnamed pipes, this is typically the write-end of the pipe.
- **buf**: A pointer to the buffer containing the data to be written.
- **count**: The number of bytes to write from the buffer.
- **Return Value**
- On success, **write()** returns the number of bytes written. This number may be less than **count** if there is insufficient space available on the pipe.
- On error, -1 is returned, and **errno** is set to indicate the error.

#### **Read(fd,buf,count):**

- **fd**: The file descriptor from which data is read. For unnamed pipes, this is typically the read end.
- **buf**: A pointer to the buffer where the read data should be stored.
- **count**: The maximum number of bytes to read.
- **Return Value**
- On successful completion, **read()** returns the number of bytes actually read and placed in the buffer. This value might be less than the number requested for several reasons, such as if fewer bytes are actually available at the moment in the pipe buffer.
- A return value of 0 indicates end-of-file (EOF); for a pipe, this means that no more data can be read because all file descriptors referring to the write end of the pipe have been closed.
- On error, -1 is returned, and **errno** is set to indicate the error.

#### **Sample Program**

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main()
{
int fd[2],n;
char buffer[100];
pid_t p;
pipe(fd); //creates a unidirectional pipe with two end fd[0] and fd[1]
p=fork();
if(p>0) //parent
{
Close(fd[0]);
printf("Parent Passing value to child\n");
write(fd[1],"hello\n",6); //fd[1] is the write end of the pipe
wait();
}
```

```

    }
else // child
{
close(fd[1]);
printf("Child printing received value\n");
n=read(fd[0],buffer,100); //fd[0] is the read end of the pipe
write(1,buffer,n);
}
}

```

### FIFO

- In computing, a named pipe (also known as a **FIFO**) is one of the methods for inter-process communication. It is an extension to the traditional pipe concept on Unix.
- A named pipe, however, can last as long as the system is up, beyond the life of the process. It can be deleted if no longer used.
- Usually a named pipe appears as a file and generally processes attach to it for inter-process communication. A FIFO file is a special kind of file on the local storage which allows two or more processes to communicate with each other by reading/writing to/from this file.
- A FIFO special file is entered into the filesystem by calling *mkfifo()* in C. Once we have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it.
- The entire IPC process will consist of three programs:

Program1: to create a named pipe

Program2: process that will write into the pipe (sender process)

Program3: process that will receive data from pipe (receiver process)

#### //Program1: to create a named pipe

```

#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
int main()
{
    int res;
    res = mkfifo("fifo1",0777); //creates a named pipe with the name fifo1
    printf("named pipe created\n");
}
//Now compile and run this program.

```

#### //Program2: Writing to a fifo/named pipe ( sender.c )

```

#include<unistd.h>
#include<stdio.h>
#include<fcntl.h>
int main()
{
    int res,n;
    res=open("fifo1",O_WRONLY);
    write(res,"Message",7);
    printf("Sender Process %d sent the data\n",getpid());
}

```

- //Program 3: Reading from the named pipe ( receiver.c )

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

```

#include<fcntl.h>
int main()
{
    int res,n;
    char buffer[100];
    res=open("fifo1",O_RDONLY);
    n=read(res,buffer,100);
    printf("Reader process %d started\n",getpid());
    printf("Data received by receiver %d is: %s\n",getpid(), buffer);
}

```

The major differences between named and unnamed pipes are:-

1. As suggested by their names, a named type has a specific name which can be given to it by the user. Named pipe if referred through this name only by the reader and writer. All instances of a named pipe share the same pipe name. On the other hand, unnamed pipes is not given a name. It is accessible through two file descriptors that are created through the function pipe(fd[2]), where fd[1] signifies the write file descriptor, and fd[0] describes the read file descriptor.
2. An unnamed pipe is only used for communication between a child and it's parent process, while a named pipe can be used for communication between two unnamed process as well. Processes of different ancestry can share data through a named pipe.
3. A named pipe exists in the file system. After input-output has been performed by the sharing processes, the pipe still exists in the file system independently of the process, and can be used for communication between some other processes. On the other hand, an unnamed pipe vanishes as soon as it is closed, or one of the process (parent or child) completes execution.

4. Named pipes can be used to provide communication between processes on the same computer or between processes on different computers across a network, as in case of a distributed system. Unnamed pipes are always local; they cannot be used for communication over a network.
5. A Named pipe can have multiple process communicating through it, like multiple clients connected to one server. On the other hand, an unnamed pipe is a one-way pipe that typically transfers data between a parent process and a child process.

### Semaphores: **semget**, **semop**, **semctl**,

Linux provides a **Semaphore API**, which has a set of functions that allow developers to work with semaphores in linux efficiently with their programs. Let us explore the key functions of the API, **semget**

The **semget()** function is used to create a **new semaphore** or get an existing semaphore's identifier.

#### Syntax:

```
int semget(key_t key, int num_sems, int sem_flags);
```

It takes three arguments:

- **key:**

Used to identify the semaphore set. It can be either `IPC_PRIVATE` (to create a new semaphore set) or an identifier used to get an existing semaphore set.

- **num\_sems:**

Number of semaphores to create in the semaphore set. Typically, this is set to 1 for single semaphores.

- **sem\_flags:**

Permission flags for the semaphore linux set, specified as an octal number. It determines the read, write, and execute permissions for the semaphore set.

It returns the linux semaphore identifier (`semid`) if successful or -1 on failure.

### **semctl()**

The **semctl()** function is used to control semaphore behavior.

#### Syntax:

```
int semctl(int semid, int sem_num, int command, ...);
```

It takes three arguments:

- **semid:**

Semaphore linux identifier representing the semaphore set on which the operation will be performed.

- **sem\_num:**

Semaphore number within the semaphore set on which the operation will be performed. For single semaphores, this is usually set to 0.

- **command:**

Specifies the operation to be performed on the Semaphore linux set. The most used values are:

- **IPC\_STAT:**

Get the semaphore set status and store it in a struct semid\_ds.

- **IPC\_SET:**

Set the semaphore set status using the information from a struct semid\_ds.

- **IPC\_RMID:**

Remove the semaphore set from the system.

- **GETVAL:**

Get the value of the semaphore.

- **GETPID:**

Get the process ID of the last operation that modified the semaphore.

- **Optional Argument:**

Depending on the command used, there might be an optional fourth argument, which is a pointer to a union semun. The union semun allows specifying additional parameters for some commands, such as the new value for SETVAL or an array of semaphore values for SETALL.

This function returns the result of the specified command, depending on the operation performed.

On failure, it returns -1 to indicate an error.

## **semop()**

The **semop()** function is used to perform operations, such as locking and unlocking semaphores in linux.

### **Syntax:**

```
int semop(int semid, struct sembuf *sops, size_t num_sops);
```

It takes three arguments:

- **semid:** The linux semaphore identifier.

- **sops:** Pointer to an array of struct sembuf, specifying the Semaphore linux operations to be performed.
- **num\_sops:** Number of elements in the sops array, indicating the number of semaphore operations to be performed.

The struct sembuf data structure contains three fields:

- **sem\_num:** Semaphore number within the semaphore set. For single semaphores, this is usually set to 0.
- **sem\_op:** The operation to be performed on the semaphore, which can be a positive or negative value.
- **sem\_flg:** Flags that control the behavior of the operation, such as SEM\_UNDO (automatic semaphore release on process termination) or IPC\_NOWAIT (return immediately if the semaphore is not available).

This function returns 0 if the semaphore operations are successful or -1, on error.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/sem.h>
int main() {
    int semaphore = semget(IPC_PRIVATE, 1, 0666 | IPC_CREAT);
    struct sembuf sem_op;

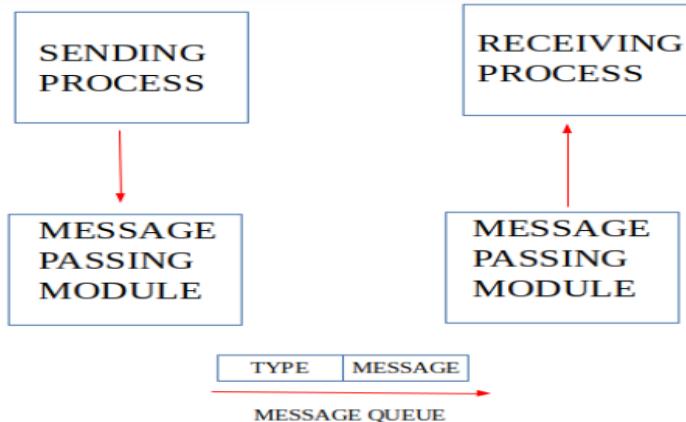
    if (semaphore < 0) {
        perror("Semaphore creation failed");
        exit(1);
    }
    printf("Semaphore created successfully.\n");
    sem_op.sem_num = 0;
    sem_op.sem_op = -1; // Lock the semaphore
    sem_op.sem_flg = 0;
    if (semop(semaphore, &sem_op, 1) == -1) {
        perror("Failed to lock semaphore");
        exit(1);
    }
    printf("Semaphore locked.\n");
    // Critical Section - Access shared resources here
    printf("In critical section.\n");
    sem_op.sem_op = 1; // Unlock the semaphore
    if (semop(semaphore, &sem_op, 1) == -1) {
        perror("Failed to unlock semaphore");
        exit(1);
    }
}
```

```
}

printf("Semaphore unlocked.\n");
if (semctl(semaphore, 0, IPC_RMID) == -1) {
    perror("Failed to remove semaphore");
    exit(1);
}
printf("Semaphore removed.\n");
return 0;
}
```

## Message Queue

A **message queue** is an inter-process communication (IPC) mechanism that allows processes to exchange data in the form of messages between two processes. It allows processes to communicate asynchronously by sending messages to each other where the messages are stored in a queue, waiting to be processed, and are deleted after being processed.



To perform communication using message queues, following are the steps –

- Step 1** – Create a message queue or connect to an already existing message queue (`msgget()`)
- Step 2** – Write into message queue (`msgsnd()`)
- Step 3** – Read from the message queue (`msgrcv()`)
- Step 4** – Perform control operations on the message queue (`msgctl()`)

### **msgget**

**Purpose:** The **msgget** system call is used to obtain access to a message queue. If a new queue is to be created, **msgget** can also initialize it.

**Syntax:** The function is defined in C as follows:

```
int msgget(key_t key, int msgflg);
```

**key\_t key:** This is an identifier for the message queue. It can be specified explicitly by the programmer, or **IPC\_PRIVATE** can be used to generate a unique key.

**int msgflg:** These flags determine the action to take if the message queue already exists or doesn't exist. It can also include permissions for the queue. Common flags include **IPC\_CREAT** (create the message queue if it does not exist), **IPC\_EXCL** (ensure the message queue is being created for the first time), along with the usual permission bits (like **0666**).

**Return Value:** It returns the message queue identifier (a non-negative integer) on success. If it fails, it returns -1 and sets the **errno** variable to indicate the error.

## **msgsnd**

**Purpose:** `msgsnd` sends a message to the message queue. It allows processes to communicate asynchronously by placing messages onto a queue, which can then be read by other processes.

**Syntax:** `int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);`

**int msqid:** This is the message queue identifier returned by `msgget`.

**const void \*msgp:** This is a pointer to the message that needs to be sent. The message must start with a long integer that represents the message type, followed by the actual data.

**size\_t msgsz:** This is the size of the message data in bytes, not including the message type.

**int msgflg:** These flags control what the function should do if the message queue is full. Common flags include **IPC\_NOWAIT** which causes the function to return immediately if the message cannot be sent because the queue is full.

**Return Value:** On success, `msgsnd` returns 0. On failure, it returns -1

## **msgrcv**

**Purpose :** `msgrcv` is used to receive messages from a queue. It can be configured to retrieve messages of specific types or simply fetch messages in a first-in, first-out (FIFO) order.

**Syntax :** `size_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);`

- **int msqid:** This is the message queue identifier returned by `msgget`.
- **\*void msgp:** This is a pointer to the message buffer into which the received message will be placed. This buffer should be structured to match the message being received, typically starting with a long integer that indicates the message type.
- **size\_t msgsz:** This specifies the size of the message buffer, excluding the size of the message type field.
- **long msgtyp:** This parameter determines which message to receive from the queue:

If **msgtyp** is 0, the next message in the queue is received.

If **msgtyp** is greater than 0, the next message of that type is received.

If **msgtyp** is less than 0, the first message of the lowest type that is less than or equal to the absolute value of **msgtyp** is received.

- **int msgflg:** These flags determine how the function behaves if the desired message is not immediately available. Common flags include:

**IPC\_NOWAIT:** Return immediately if no message is available that matches the criteria.

**MSG\_EXCEPT**: Used with a positive **msgtyp** to receive the next message in the queue that is not of type **msgtyp**.

**MSG\_NOERROR**: If the message is larger than **msgsz**, truncate it instead of returning an error.

**Return Value**: On success, **msgrcv** returns the size of the received message. If it fails, it returns -1

## **msgctl**

**Purpose** : **msgctl** is used to control or modify the properties of a message queue, or to remove a message queue entirely from the system.

**Syntax** : int msgctl(int msqid, int cmd, struct msqid\_ds \*buf);

- **int msqid**: This is the message queue identifier returned by **msgget**.
- **int cmd**: This command argument specifies the action to be performed on the message queue. Common commands include:

**IPC\_STAT**: Copy information from the kernel data structure associated with **msqid** into the **msqid\_ds** structure pointed to by **buf**.

**IPC\_SET**: Set the message queue attributes using the information in the **msqid\_ds** structure pointed to by **buf**.

**IPC\_RMID**: Remove the message queue. This command immediately deletes the queue and all its messages.

- **\*struct msqid\_ds buf**: This is a pointer to an instance of **msqid\_ds** structure which is used to store or modify message queue attributes. This structure is defined as follows:

```
struct msqid_ds {  
    struct ipc_perm msg_perm; // Ownership and permissions  
    time_t      msg_stime; // Time of last msgsnd  
    time_t      msg_rtime; // Time of last msgrcv  
    time_t      msg_ctime; // Time of last change  
    unsigned long msg_cbytes; // Current number of bytes in queue (non-standard)  
    msgqnum_t   msg_qnum; // Current number of messages in queue  
    msglen_t    msg_qbytes; // Maximum number of bytes allowed in queue  
    pid_t       msg_lspid; // PID of last msgsnd  
    pid_t       msg_lrpid; // PID of last msgrcv  
};
```

**Return Value**: On success, **msgctl** returns 0. If it fails, it returns -1

## Sample Program

### Sender Process

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct msghdr {
    long mtype; // message type must be > 0
    char mtext[200]; // message data
};

int main() {
    key_t key = 1234; // predefined key
    int msgid;
    struct msghdr msg;
    // Create a message queue
    msgid = msgget(key, 0666 | IPC_CREAT);
    if (msgid < 0) {
        perror("msgget");
        exit(EXIT_FAILURE);
    }
    // Prepare a message to send
    msg.mtype = 1;
    strcpy(msg.mtext, "Hello, this is message 1");
    // Send the message
    if (msgsnd(msgid, &msg, strlen(msg.mtext), 0) < 0) {
        perror("msgsnd");
        exit(EXIT_FAILURE);
    }
    printf("Sent: %s\n", msg.mtext);
    // Prepare another message
    msg.mtype = 2;
```

```

strcpy(msg.mtext, "Hello, this is message 2");

// Send the message

if (msgsnd(msgid, &msg, strlen(msg.mtext), 0) < 0) {
    perror("msgsnd");
    exit(EXIT_FAILURE);
}

printf("Sent: %s\n", msg.mtext);

return 0;
}

```

### **Receiver Process**

```

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/ipc.h>

#include <sys/msg.h>

struct msbuf {

    long mtype; // message type must be > 0

    char mtext[200]; // message data

};

int main() {

    key_t key = 1234; // predefined key

    int msgid;

    struct msbuf msg;

    // Access the message queue

    msgid = msgget(key, 0666 | IPC_CREAT);

    if (msgid < 0) {

        perror("msgget");

        exit(EXIT_FAILURE);

    }

    // Receive a message of any type

    if (msgrcv(msgid, &msg, sizeof(msg.mtext), 0, 0) < 0) {

        perror("msgrcv");
    }
}

```

```

    exit(EXIT_FAILURE);
}

printf("Received: %s\n", msg.mtext);

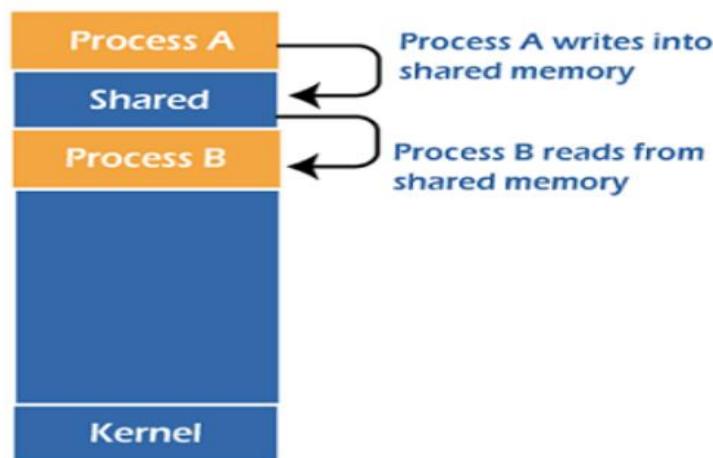
// Optionally remove the message queue
if (msgctl(msqid, IPC_RMID, NULL) < 0) {
    perror("msgctl");
    exit(EXIT_FAILURE);
}

return 0;
}

```

### Shared memory

Inter Process Communication through shared memory is a concept where two or more process can access the common memory and communication is done via this shared memory where changes made by one process can be viewed by another process.



The problem with pipes, fifo and message queue – is that for two process to exchange information.

The information has to go through the kernel.

- Server reads from the input file.
- The server writes this data in a message using either a pipe, fifo or message queue.
- The client reads the data from the IPC channel, again requiring the data to be copied from kernel's IPC buffer to the client's buffer.
- Finally, the data is copied from the client's buffer.

A total of four copies of data are required (2 read and 2 write). So, shared memory provides a way by letting two or more processes share a memory segment. With Shared Memory the data is only copied twice – from input file into shared memory and from shared memory to the output file.

**shmget:** This function is used to allocate a shared memory segment.

**Syntax:** int shmget(key\_t key, size\_t size, int shmflg);

- **key\_t key:** Unique identifier for the shared memory segment.
- **size\_t size:** The size of the shared memory segment in bytes.
- **int shmflg:** Permission flags and creation flags (**IPC\_CREAT, IPC\_EXCL**).

**Returns:** On success, it returns the identifier of the shared memory segment; on error, it returns -1

**Shmat :** This function attaches the shared memory segment identified by the shared memory ID to the address space of the calling process.

**Syntax :** void \*shmat(int shmid, const void \*shmaddr, int shmflg);

- **int shmid:** Shared memory identifier returned by **shmget**.
- **\*const void \*shmaddr:** Suggested starting address for the attachment; usually set to **NULL** to let the system choose.
- **int shmflg:** Flags for the operation, typically **0** or **SHM\_RDONLY** for read-only access.

**Returns:** On success, it returns the address of the attached shared memory segment; on failure, it returns **(void \*) -1**.

**Shmdt:** This function detaches the shared memory segment from the address space of the calling process.

**Syntax:** int shmdt(const void \*shmaddr);

- **\*const void \*shmaddr:** Address of the shared memory segment to detach.

**Returns:** On success, returns 0; on error, returns -1.

**Shmctl:** This function performs various control operations on the shared memory segment.

**Syntax:** int shmctl(int shmid, int cmd, struct shmid\_ds \*buf);

- **int shmid:** Shared memory identifier.
- **int cmd:** Control command (**IPC\_STAT, IPC\_SET, IPC\_RMID**).
- **\*struct shmid\_ds \*buf:** Pointer to a **shmid\_ds** structure to store or modify shared memory metadata.

```

struct shmid_ds {
    struct ipc_perm shm_perm; // Operation permission structure
    size_t      shm_segsz; // Size of segment in bytes
    time_t      shm_atime; // Last attach time
    time_t      shm_dtime; // Last detach time
    time_t      shm_ctime; // Last change time
    pid_t       shm_cpid; // PID of creator
    pid_t       shm_lpid; // PID of last operation
    shmatt_t   shm_nattch; // Number of current attaches
};


```

**Return Value:** On success, returns 0; on error, returns -1.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main() {
    key_t key = 1234; // Identifier for the shared memory segment
    int shmid;
    char *data;

    // Create the shared memory segment
    shmid = shmget(key, 1024, 0666 | IPC_CREAT);
    if (shmid < 0) {
        perror("shmget");
        exit(1);
    }

    // Attach the shared memory segment
    data = shmat(shmid, NULL, 0);
    if (data == (char *)(-1)) {
        perror("shmat");
        exit(1);
    }
}

```

```
}

// Write to the shared memory
printf("Writing to shared memory: \"Hello, World!\"\n");
strncpy(data, "Hello, World!", 1024);

// Detach the shared memory
if (shmdt(data) < 0) {
    perror("shmdt");
    exit(1);
}

// Reattach the shared memory to read from it
data = shmat(shmid, NULL, 0);
if (data == (char *)(-1)) {
    perror("shmat");
    exit(1);
}

// Read from the shared memory
printf("Reading from shared memory: \"%s\"\n", data);

// Detach the shared memory
if (shmdt(data) < 0) {
    perror("shmdt");
    exit(1);
}

// Remove the shared memory segment
if (shmctl(shmid, IPC_RMID, NULL) < 0) {
    perror("shmctl");
    exit(1);
}

return 0;
```

## Ipc status commands

Interprocess Communication (IPC) mechanisms in Unix-like operating systems, such as semaphores, message queues, and shared memory, offer status commands to help manage and monitor IPC resources. These status commands are generally invoked using control system calls like **semctl**, **msgctl**, and **shmctl**, particularly with the **IPC\_STAT** command.

### 1. Shared Memory (shmctl with IPC\_STAT)

**shmctl** with the **IPC\_STAT** command is used to fetch the status of a shared memory segment.

**Syntax:** int shmctl(int shmid, int cmd, struct shmid\_ds \*buf);

- **shmid:** Identifier for the shared memory segment.
- **cmd:** Command, where **IPC\_STAT** is used to get the status.
- **buf:** Pointer to **shmid\_ds** structure where the status data will be stored.

**Data Fetched Includes:**

- Segment size, owner, permissions
- Time of last attach, detach, and changes
- PIDs of creator and last operator
- Number of current attachments

### 2. Message Queues (msgctl with IPC\_STAT)

**msgctl** with the **IPC\_STAT** command retrieves the status of a message queue.

**Syntax:** int msgctl(int msqid, int cmd, struct msqid\_ds \*buf);

- **msqid:** Identifier for the message queue.
- **cmd:** Command, where **IPC\_STAT** retrieves the current status.
- **buf:** Pointer to **msqid\_ds** structure to store the status data.

**Data Fetched Includes:**

- Queue size, owner, permissions
- Time of last send and receive operations
- Number of messages currently in the queue
- Total bytes allowed in the queue
- PIDs of processes that last sent or received a message

### 3. Semaphores (semctl with IPC\_STAT)

**semctl** with the **IPC\_STAT** command is used to obtain the status of a semaphore set.

**Syntax:** int semctl(int semid, int semnum, int cmd, union semun arg);

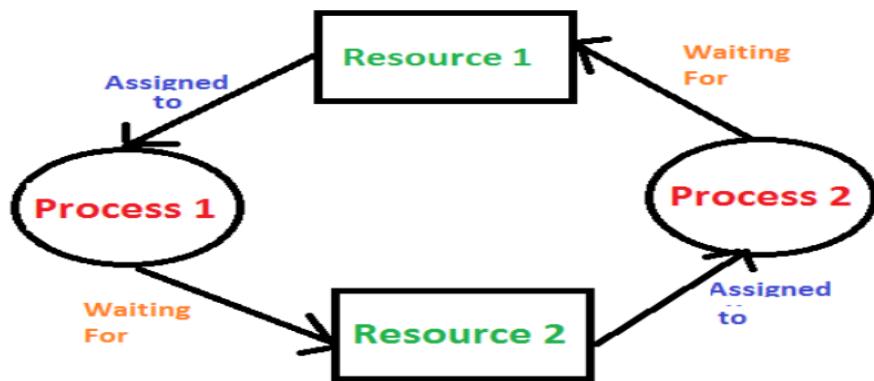
- **semid**: Identifier for the semaphore set.
- **seminum**: Semaphore number within the set (usually ignored for **IPC\_STAT**).
- **cmd**: Command, with **IPC\_STAT** used for fetching status.
- **arg**: A union where one member is a pointer to **semid\_ds**.

**Data Fetched Includes:**

- Semaphore array sizes, owner, permissions
- Time of last operations
- Number of semaphores in the set

## Deadlocks

In a multiprogramming system, multiple processes compete for limited resources. When a process requests resources that are unavailable at the time, it goes into a waiting state. Sometimes, a waiting process cannot change its state again because other processes hold the resources it needs, resulting in a deadlock. A **deadlock** is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.



## System Model

A system model or structure comprises a fixed number of resources that must be shared among different processes. These resources are divided into different types, each having a specific number of identical instances. Some examples of resource types are memory space, CPU cycles, directories and files, and I/O devices such as keyboards, printers, and CD/DVD drives. For instance, when a system has two CPUs, the resource type CPU has two instances.

Under the standard mode of operation, any process may use a resource in only the below-mentioned sequence:

1. Request: When the request can't be approved immediately (where the case may be when another process is utilizing the resource), the requesting job must wait until it can obtain the resource.
2. Use: The process can run on the resource (like when the resource is a printer, its job/process is to print on the printer).
3. Release: The process releases the resource (like terminating or exiting any specific process).

## Deadlock Characterization

### **Necessary conditions**

Deadlocks can arise if the following four conditions hold simultaneously in a system:

- Mutual exclusion: If there is a non-shareable resource, only one process can use a resource at a time.
- Hold and wait: A process holding at least one resource is waiting to acquire additional resources held by other waiting processes.
- No preemption: A resource can be released only voluntarily by the process that is holding it after the process has completed its task. It is not possible to preempt a resource forcibly from a process.
- Circular wait: If there exists a set  $\{P_0, P_1, \dots, P_n\}$  of waiting processes such that  $P_0$  is waiting for a resource that is held by  $P_1$ ,  $P_1$  is waiting for a resource that is held by  $P_2$ , ...,  $P_{n-1}$  is waiting for a resource that is held by  $P_n$ , and  $P_n$  is waiting for a resource that is held by  $P_0$ .

In order to describe deadlocks in a more precise way directed graphs are used that are called **system Resource Allocation Graph**.

- This Graph acts as the pictorial representation of the state of the system.
- The Resource Allocation graph mainly consists of a **set of vertices V** and a **set of Edges E**.
- This graph mainly contains all the information related to the processes that are holding some resources and also contains the information of the processes that are waiting for some more resources in the system.
- Also, this graph contains all the information that is related to all the instances of the resources which means the information about available resources and the resources which are being used by the process

### **Components of Resource Allocation Graph**

Given below are the components of RAG:

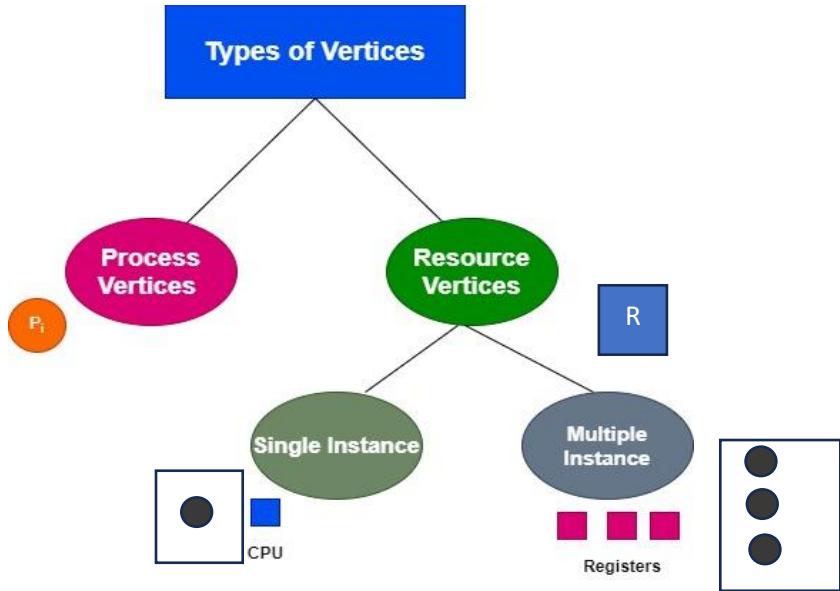
1. Vertices
2. Edges

#### **Vertices**

There are two kinds of vertices used in the resource allocation graph and these are:

Process Vertices : These vertices are used in order to represent process vertices. The circle is used in order to draw the process vertices and the name of the process is mentioned inside the circle.

**Resource Vertices:** These vertices are used in order to represent resource vertices. The rectangle is used in order to draw the resource vertices and we use dots inside the circle to mention the number of instances of that resource.



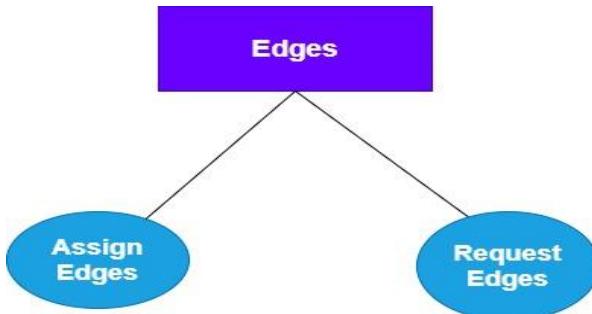
In the system, there may exist a number of instances and according to them, there are two types of resource vertices and these are single instances and multiple instances.

**Single Instance:** In a single instance resource type, there is a single dot inside the box. The single dot mainly indicates that there is one instance of the resource.

**Multiple Instance:** In multiple instance resource types, there are multiple dots inside the box, and these Multiple dots indicate that there are multiple instances of the resources.

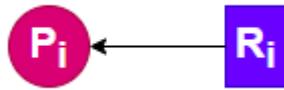
### **Edges**

In the Resource Allocation Graph, Edges are further categorized into two:



#### **1. Assign Edges**

Assign Edges are mainly used to represent the allocation of resources to the process. We can draw assign edges with the help of an arrow in which mainly the arrowhead points to the process, and the tail points to the instance of the resource.



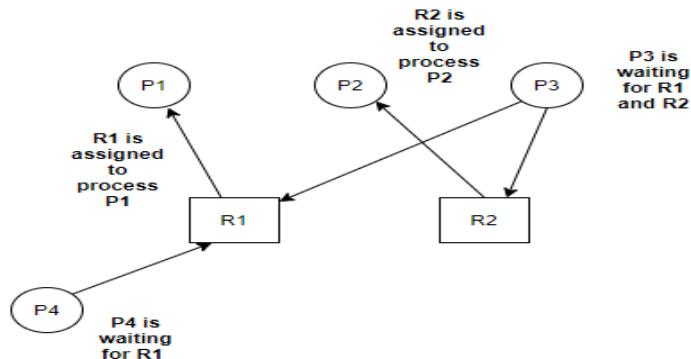
## 2. Request Edges

Request Edge is mainly used to signify the waiting state of the process. Likewise in assigned edge, an arrow is used to draw an arrow edge. But Here the arrowhead points to the instance of a resource, and the tail points to the process.

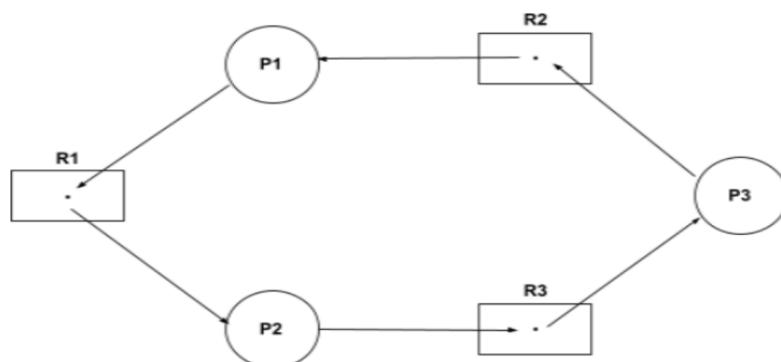


**Single Instance RAG :** If there is a cycle in the Resource Allocation Graph and each resource in the cycle provides only one instance, then the processes will be in deadlock.

Suppose there are Four Processes P1, P2, P3, P4, and two resources R1 and R2, where P1 is holding R1 and P2 is holding R2, P3 is waiting for R1 and R2 while P4 is waiting for resource R1.



In the above example, there is no circular dependency so there are no chances for the occurrence of deadlock.



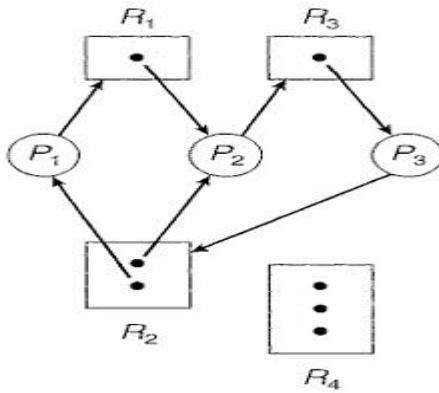
In the above single instances RAG example, it can be seen that P2 is holding the R1 and waiting for R3. P1 is waiting for R1 and holding R2 and, P3 is holding the R3 and waiting for R2. So, it can be concluded that none of the processes will get executed. It leads to Deadlock

### Multiple Instance RAG Example

If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

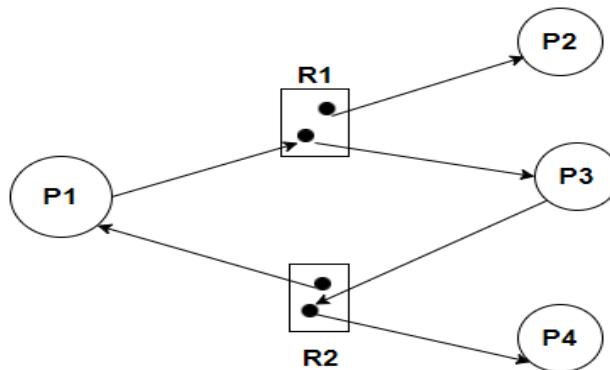
suppose that process P3 requests an instance of resource type R2. Since no resource instance is currently available, a request edge P3 → R2 is added to the graph (At this point, two minimal cycles exist in the system: P1 → R1 → P2 → R3 → P3 → R2 → P1

P2 → R3 → P3 → R2 → P2



Processes P1, P2, and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3 is waiting for either process P1 or process P2 to release resource R2. In addition, process P1 is waiting for process P2 to release resource R1.

Suppose there are four processes P1, P2, P3, P4 and there are two instances of resource R1 and two instances of resource R2:



**Multiple Instance Resource Allocation graph with a cycle but no deadlock**

One instance of R2 is assigned to process P1 and another instance of R2 is assigned to process P4, Process P1 is waiting for resource R1.

One instance of R1 is assigned to Process P2 while another instance of R2 is assigned to process P3, Process P3 is waiting for resource R2. However, there is no deadlock. Observe that process P4 may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle.

***In summary, if a resource-allocation graph does not have a cycle, then the system is not in a deadlocked state.***

***If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.***

***If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.***

#### Deadlock Vs Starvation

<b>Deadlock</b>	<b>Starvation</b>
A deadlock is a condition in operating systems in which no process proceeds for execution and wait for resources that have been acquired by some other processes. It is also known as circular wait.	Starvation is a situation when a process keeps waiting (and starving) for a resource that is being held by other highpriority processes.
In a deadlock, all the involved processes keep waiting for each other to get completed.	In this situation, the high priority processes are executed, and low priority processes are blocked.
In a deadlock, none of them can execute because they are waiting for the other process to complete.	In starvation, the low priority process starves due to lack of resources.
The resources are blocked by the process.	Resources are utilized by high priority process continuously.
The necessary conditions for deadlock are mutual exclusion, hold and wait, no pre-emption, and circular wait.	In starvation, priorities are assigned to process.
Deadlock can be prevented by avoiding conditions such as mutual exclusion, hold and wait, no preemption and circular wait.	Starvation can be prevented using the "Aging" technique.

## **Methods for Handling Deadlocks**

To handle deadlock, there are several techniques that can be used in the operating system. Some of the most common deadlock handling techniques are:

1. **Prevention:** This technique involves designing the system in such a way that deadlock can never occur. This is achieved by ensuring that at least one of the four necessary conditions for deadlock (mutual exclusion, hold and wait, no pre-emption, and circular wait) does not occur. Prevention can be expensive and can reduce system performance, so it is not always the best option.
2. **Avoidance:** This technique involves ensuring that the system does not enter a deadlock state by using algorithms that can predict if a resource allocation request will result in a deadlock. The system will only grant resource requests that do not lead to a deadlock state. This technique requires a lot of computation and can be resource-intensive.
3. **Detection and Recovery:** This technique involves detecting when a deadlock has occurred and taking steps to recover from it. One common approach is to use an algorithm that periodically checks for deadlock. If a deadlock is detected, the system will then recover from it by releasing resources or killing processes.
4. **Ignoring Deadlock:** In some cases, it may be less expensive to simply ignore deadlock and let the processes remain blocked until the system is restarted or the resources are manually released.

## **Deadlock Prevention**

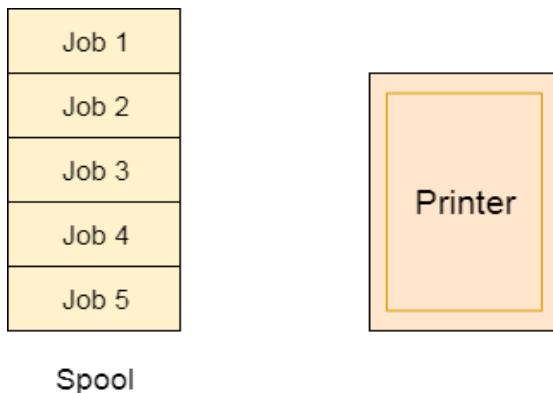
Let us take an example of a chair, as we know that chair always stands on its four legs. Likewise, for the deadlock problem, all the above given four conditions are needed. If anyone leg of the chair gets broken, then definitely it will fall. The same is the situation with the deadlock if we become able to violate any condition among the four and do not let them occur together then there can be prevented from the deadlock problem.

### **Mutual Exclusion**

This condition must hold for non-sharable resources. In contrast, Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock. A good example of a sharable resource is Read-only files because if several processes attempt to open a read-only file at the same time, then they can be granted simultaneous access to the file.

A process need not to wait for the sharable resource. Spooling can be an effective approach to violate mutual exclusion .

For a device like printer, spooling can work. There is a memory associated with the printer which stores jobs from each of the process into it. Later, Printer collects all the jobs and print each one of them according to FCFS. By using this mechanism, the process doesn't have to wait for the printer and it can continue whatever it was doing. Later, it collects the output when it is produced.



**Spool**

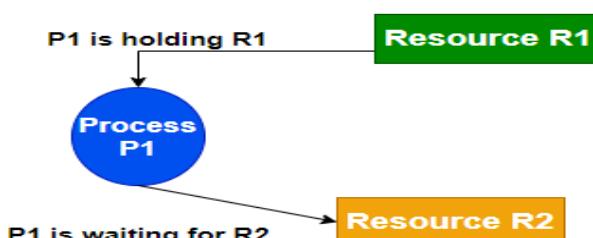
Although, Spooling can be an effective approach to violate mutual exclusion but it suffers from two kinds of problems.

1. This cannot be applied to every resource.
2. After some point of time, there may arise a race condition between the processes to get space in that spool.

We cannot force a resource to be used by more than one process at the same time since it will not be fair enough and some serious problems may arise in the performance. Therefore, we cannot violate mutual exclusion for a process practically.

### **Hold and Wait**

Hold and wait condition occurs when a process holds a resource and is also waiting for some other resource in order to complete its execution. Thus, if we did not want the occurrence of this condition then we must guarantee that when a process requests a resource, it does not hold any other resource.



**Hold and wait condition**

There are some protocols that can be used in order to ensure that the Hold and Wait condition never occurs:

- According to the first protocol; Each process must request and gets all its resources before the beginning of its execution.
- The second protocol allows a process to request resources only when it does not occupy any resource.

Let us illustrate the difference between these two protocols:

We will consider a process that mainly copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer. If all the resources must be requested at the beginning of the process according to the first protocol, then the process requests the DVD drive, disk file, and printer initially. It will hold the printer during its entire execution, even though the printer is needed only at the end.

While the second method allows the process to request initially only the DVD drive and disk file. It copies the data from the DVD drive to the disk and then releases both the DVD drive and the disk file. The process must then again request the disk file and printer. After copying the disk file to the printer, the process releases these two resources as well and then terminates.

### **Disadvantages of Both Protocols**

- Utilization of resources may be low, since resources may be allocated but unused for a long period. In the above-given example, for instance, we can release the DVD drive and disk file and again request the disk file and printer only if we can be sure that our data will remain on the disk file. Otherwise, we must request all the resources at the beginning of both protocols.
- There is a possibility of starvation. A process that needs several popular resources may have to wait indefinitely because at least one of the resources that it needs is always allocated to some other process.

### **No Preemption**

The third necessary condition for deadlocks is that there should be no preemption of resources that have already been allocated. In order to ensure that this condition does not hold the following protocols can be used :

- According to the First Protocol: "If a process that is already holding some resources requests another resource and if the requested resources cannot be allocated to it, then it must release all the resources currently allocated to it."

- According to the Second Protocol: "When a process requests some resources, if they are available, then allocate them. If in case the requested resource is not available then we will check whether it is being used or is allocated to some other process waiting for other resources. If that resource is not being used, then the operating system preempts it from the waiting process and allocate it to the requesting process. And if that resource is being used, then the requesting process must wait".

The second protocol can be applied to those resources whose state can be easily saved and restored later for example CPU registers and memory space, and cannot be applied to resources like printers and tape drivers.

### **Circular Wait**

The Fourth necessary condition to cause deadlock is circular wait, In order to ensure violate this condition we can do the following:

Assign a priority number to each resource. There will be a condition that any process cannot request for a lesser priority resource. This method ensures that not a single process can request a resource that is being utilized by any other process and due to which no cycle will be formed.

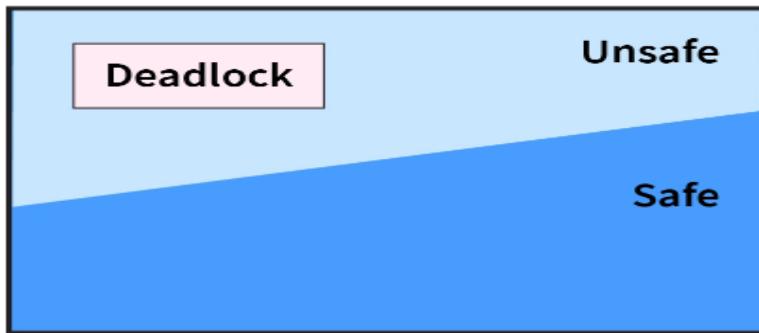
Example: Assume that R5 resource is allocated to P1, if next time P1 asks for R4, R3 that are lesser than R5; then such request will not be granted. Only the request for resources that are more than R5 will be granted.

S.No	Necessary Conditions	Approach	Practical Implementation
1	Mutual Exclusion	The approach used to violate this condition is spooling.	Not possible
2	Hold and wait	In order to violate this condition, the approach is to request all the resources for a process initially	Not possible
3	No Preemption	In order to violate this condition, the approach is snatch all the resources from the process.	Not possible
4	Circular Wait	In this approach is to assign priority to each resource and order them numerically	<b>possible</b>

## Deadlock Avoidance

Deadlock Avoidance work by letting the Operating System know the Process requirements of resources to complete their execution, and accordingly operating system checks if the requirements can be satisfied or not.

### Safe State and Unsafe State



A safe state refers to a system state where the allocation of resources to each process ensures the avoidance of deadlock. The successful execution of all processes is achievable, and the likelihood of a deadlock is low. The system attains a safe state when a suitable sequence of resource allocation enables the successful completion of all processes.

Conversely, an unsafe state implies a system state where a deadlock may occur. The successful completion of all processes is not assured, and the risk of deadlock is high. The system is insecure when no sequence of resource allocation ensures the successful execution of all processes.

### Example

- To illustrate, we consider a system with twelve magnetic tape drives and three processes: P0, P1, and P2. Process P0 requires ten tape drives, process P1 may need as many as four tape drives, and process P2 may need up to nine tape drives. Suppose that, at time t0, process P0 is holding five tape drives, process P1 is holding two tape drives, and process P2 is holding two tape drives. (Thus, there are three free tape drives.)

	<u>Maximum Needs</u>	<u>Current Needs</u>	<u>maximum need to finished</u>
P <sub>0</sub>	10	5	5
P <sub>1</sub>	4	2	2
P <sub>2</sub>	9	2	7

- At time t0, the system is in a safe state. The sequence satisfies the safety condition. Process P1 can immediately be allocated all its tape drives and then return them (the system will then have five available tape drives); then process P0 can get all its tape drives and return them (the system will then have ten available tape drives); and finally process P2 can get all its tape drives and return them (the system will then have all twelve tape drives available).

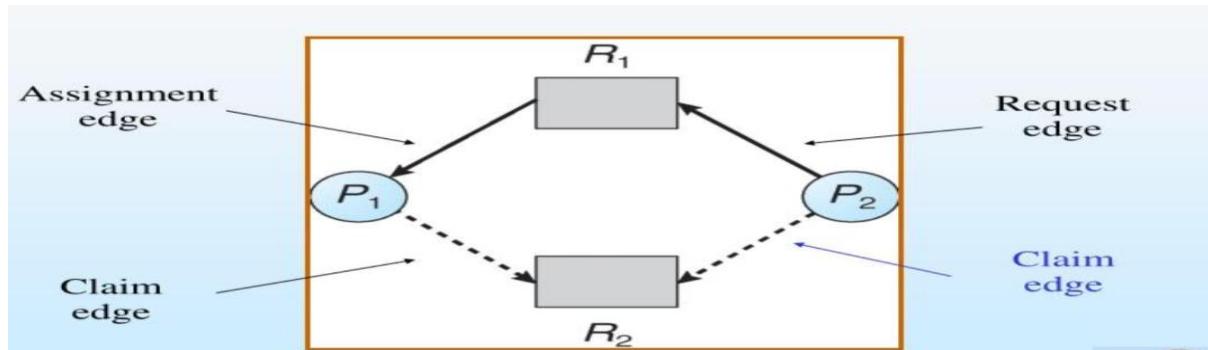
- A system can go from a safe state to an unsafe state. Suppose that, at time t1, process P2 requests and is allocated one more tape drive. The system is no longer in a safe state. At this point, only process P1 can be allocated all its tape drives. When it returns them, the system will have only four available tape drives. Since process P0 is allocated five tapes drives but has a maximum of ten, it may request five more tape drives. If it does so, it will have to wait, because they are unavailable. Similarly, process P2 may request six additional tape drives and have to wait, resulting in a deadlock. Our mistake was in granting the request from process P2 for one more tape drive. If we had made P2 wait until either of the other processes had finished and released its resources, then we could have avoided the deadlock.

### Deadlock Avoidance Algorithms

- When resource categories have only single instances of their resources, Resource- Allocation Graph Algorithm is used. In this algorithm, a cycle is a necessary and sufficient condition for deadlock.
- When resource categories have multiple instances of their resources, Banker's Algorithm is used. In this algorithm, a cycle is a necessary but not a sufficient condition for deadlock.

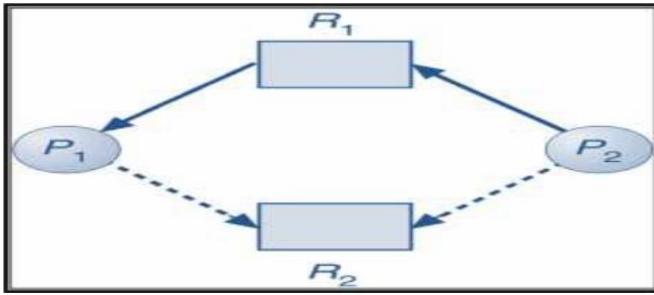
### Resource Allocation graph

Resource Allocation graph technique is used for deadlock avoidance when there is a single instance of every resource. In resource allocation graph for deadlock avoidance we introduce a third kind of edge called the claim edge which is a dotted line from a process towards a resource meaning that the resource can be requested by the process in future.

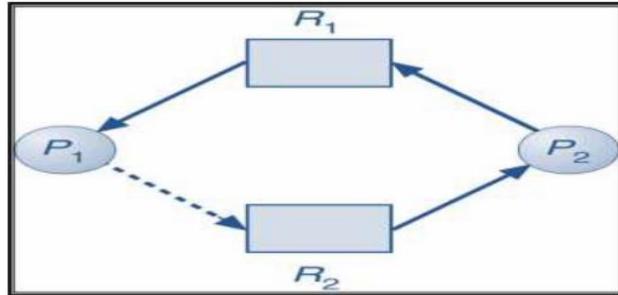


- Whenever a process requests for a resource the claim edge is changed to request edge and if the resource can be granted the request edge is changed to assignment edge.
- After this change look for a cycle in the graph.
- If no cycle exists, then the system is in safe state and the deadlock will not occur
- else the system is in unsafe state and deadlock may or may not occur.

Consider the resource-allocation graph in Figure below. P2 requests for resource R2 as shown using the claim edge from P2 to R2 .



This is converted to an assignment edge in the resource-allocation graph as shown in Figure .



The graph is now checked for cycles. Since there is a cycle, the system understands that allocating  $R_2$  to  $P_2$  will lead to a deadlock. The only instance of  $R_1$  is assigned to  $P_1$  and the only instance of  $R_2$  is assigned to  $P_2$ .  $P_2$  has requested for  $R_1$ . It is known from the claim edge that process  $P_1$  may request for  $R_2$ . If  $P_1$  requests for  $R_2$  (before releasing  $R_1$ , then both the processes  $P_1$  and  $P_2$  will wait for a resource held by the other process. The system will go to a deadlocked state. Therefore,  $R_2$  is not allocated to  $P_2$ . Thus, deadlocks are avoided.

This method can be used to avoid deadlocks only if there is a single instance of each resource type. If there are more instances of each resource type, Banker's algorithm is used for deadlock avoidance.

### **Bankers Algorithm**

Banker's algorithm is a **deadlock avoidance algorithm**. It is named so because this algorithm is used in banking systems to determine whether a loan can be granted or not.

When working with a banker's algorithm, it requests to know about three things:

1. How much each process can request for each resource in the system. It is denoted by the **[MAX]** request.
2. How much each process is currently holding each resource in a system. It is denoted by the **[ALLOCATED]** resource.
3. It represents the number of each resource currently available in the system. It is denoted by the **[AVAILABLE]** resource.

Following are the important data structures terms applied in the banker's algorithm as follows. Suppose  $n$  is the number of processes, and  $m$  is the number of each type of resource used in a computer system.

1. **Available:** It is an array of length ' $m$ ' that defines each type of resource available in the system. When  $\text{Available}[j] = K$ , means that ' $K$ ' instances of Resources type  $R[j]$  are available in the system.

2. **Max:** It is a  $[n \times m]$  matrix that indicates each process  $P[i]$  can store the maximum number of resources  $R[j]$  (each type) in a system.
3. **Allocation:** It is a matrix of  $m \times n$  orders that indicates the type of resources currently allocated to each process in the system. When Allocation  $[i, j] = K$ , it means that process  $P[i]$  is currently allocated  $K$  instances of Resources type  $R[j]$  in the system.
4. **Need:** It is an  $M \times N$  matrix sequence representing the number of remaining resources for each process. When the Need*[i][j]* = k, then process  $P[i]$  may require  $K$  more instances of resources type  $R_j$  to complete the assigned work.  

$$Ned[i][j] = Max[i][j] - Allocation[i][j].$$
5. **Finish:** It is the vector of the order  $\mathbf{m}$ . It includes a Boolean value (true/false) indicating whether the process has been allocated to the requested resources, and all resources have been released after finishing its task.

### Safety Algorithm

1. Let Work and Finish be vectors of length  $m$  and  $n$ , respectively. Initialize Work = Available and Finish*[i]* = false for  $i = 0, 1 \dots n - 1$ .
2. Find an index  $i$  such that both a. Finish*[i]* == false b. Need*[i]* ≤ Work If no such  $i$  exists, go to step 4.(end of all process)
3. Work = Work + Allocation*[i]* Finish*[i]* = true Go to step 2.
4. If Finish*[i]* == true for all  $i$ , then the system is in a safe state.

This algorithm may require an order of  $m \times n^2$  operations to determine whether a state is safe.

### Example:

Let us consider the following snapshot for understanding the banker's algorithm:

Processes	Allocation A B C	Max A B C	Available A B C
P0	1 1 2	4 3 3	2 1 0
P1	2 1 2	3 2 2	

P2	4 0 1	9 0 2	
P3	0 2 0	7 5 3	
P4	1 1 2	1 1 2	

1. calculate the content of the need matrix?
2. Check if the system is in a safe state?
3. Determine the total sum of each type of resource?

**Solution:**

1. The Content of the need matrix can be calculated by using the formula given below:

$$\text{Need} = \text{Max} - \text{Allocation}$$

Process	Need		
	A	B	C
P <sub>0</sub>	3	2	1
P <sub>1</sub>	1	1	0
P <sub>2</sub>	5	0	1
P <sub>3</sub>	7	3	3
P <sub>4</sub>	0	0	0

2. Let us now check for the safe state.

**Safe sequence:**

1. For process P<sub>0</sub>, Need = (3, 2, 1) and work = (2, 1, 0)  
Need <= work = False  
So, the system will move to the next process.

2. For Process P<sub>1</sub>, Need = (1, 1, 0)

$$\begin{aligned}\text{Available} &= (2, 1, 0) \\ \text{Need} &\leq \text{Available} = \text{True} \\ \text{Request of P}_1 &\text{ is granted.} \\ \text{work} &= \text{work} + \text{Allocation} \\ &= (2, 1, 0) + (2, 1, 2) \\ &= (4, 2, 2) \text{ (New work)}\end{aligned}$$

3. For Process P<sub>2</sub>, Need = (5, 0, 1)

$$\begin{aligned}\text{work} &= (4, 2, 2) \\ \text{Need} &\leq \text{work} = \text{False} \\ \text{So, the system will move to the next process.}\end{aligned}$$

**4. For Process P3, Need = (7, 3, 3)**

work = (4, 2, 2)

Need <= work = False

So, the system will move to the next process.

**5. For Process P4, Need = (0, 0, 0)**

work = (4, 2, 2)

Need <= work e = True

Request of P4 is granted.

work = work + Allocation

$$= (4, 2, 2) + (1, 1, 2)$$

= (5, 3, 4) now, (New work)

**6. Now again check for Process P2, Need = (5, 0, 1)**

work = (5, 3, 4)

Need <= work = True

Request of P2 is granted.

work = work + Allocation

$$= (5, 3, 4) + (4, 0, 1)$$

= (9, 3, 5) now, (New work)

**7. Now again check for Process P3, Need = (7, 3, 3)**

work = (9, 3, 5)

Need <= work = True

The request for P3 is granted.

work = work + Allocation

$$= (9, 3, 5) + (0, 2, 0) = (9, 5, 5)$$

**8. Now again check for Process P0, = Need (3, 2, 1)**

work (9, 5, 5)

Need <= work = True

So, the request will be granted to P0.

Safe sequence: < P1, P4, P2, P3, P0 >

**The system allocates all the needed resources to each process. So, we can say that the system is in a safe state.**

The total amount of resources will be calculated by the following formula:

The total amount of resources= sum of columns of allocation + Available

$$= [8 \ 5 \ 7] + [2 \ 1 \ 0] = [10 \ 6 \ 7]$$

### Resource-Request Algorithm

Next, we describe the algorithm for determining whether requests can be safely granted.

Let Request<sub>i</sub> be the request vector for process P<sub>i</sub>. If Request<sub>i</sub> [ j ] == k, then process P<sub>i</sub> wants k instances of resource type R<sub>j</sub>. When a request for resources is made by process P<sub>i</sub>, the following actions are taken:

1. If Request<sub>i</sub> ≤ Need<sub>i</sub>, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If Request<sub>i</sub> ≤ Available, go to step 3. Otherwise, P<sub>i</sub> must wait, since the resources are not available.

3. Have the system pretend (offering) to have allocated the requested resources to process  $P_i$  by modifying the state as follows:

$\text{Available} = \text{Available} - \text{Request}_i$  ;

$\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i$  ;

$\text{Need}_i = \text{Need}_i - \text{Request}_i$  ;

4. Apply safety algorithm. If the resulting resource-allocation state is safe, the transaction is completed, and process  $P_i$  is allocated its resources. However, if the new state is unsafe, then  $P_i$  must wait for  $\text{Request}_i$ , and the old resource allocation state is restored.

### **Example**

If the system is in a safe state, can the following requests be granted, why or why not? Please also run the safety algorithm on each request as necessary.

- a.  $P_1$  requests  $(2,1,1,0)$

We cannot grant this request, because we do not have enough available instances of resource A.

- b.  $P_1$  requests  $(0,2,1,0)$

There are enough available instances of the requested resources, so first let's pretend to accommodate the request and see the system looks like:

	Allocation				Max				Available					Need Matrix			
	A	B	C	D	A	B	C	D	A	B	C	D		A	B	C	D
$P_0$	0	1	1	0	0	2	1	0	1	3	1	0		0	1	0	0
$P_1$	1	4	4	1	1	6	5	2						0	2	1	1
$P_2$	1	3	6	5	2	3	6	6						1	0	0	1
$P_3$	0	6	3	2	0	6	5	2						0	0	2	0
$P_4$	0	0	1	4	0	6	5	6						0	6	4	2

Now we need to run the safety algorithm:

Initially																																		
	Work vector		Finish matrix																															
	1		$P_0$		False																													
	3		$P_1$		False																													
	1		$P_2$		False																													
	0		$P_3$		False																													
				$P_4$		False																												

Need1 (0,2,1,1) is not less than work, so we need to move on to P2.

Need2 (1,0,0,1) is not less than work, so we need to move on to P3.

Need3 (0,0,2,0) is less than or equal to work. Let's update work and finish:

Work vector	Finish matrix	
1	P <sub>0</sub>	True
10	P <sub>1</sub>	False
5	P <sub>2</sub>	False
2	P <sub>3</sub>	True
	P <sub>4</sub>	False

Let's take a look at Need4 (0,6,4,2). This is less than work, so we can update work and finish:

Work vector	Finish matrix	
1	P <sub>0</sub>	True
10	P <sub>1</sub>	False
6	P <sub>2</sub>	False
6	P <sub>3</sub>	True
	P <sub>4</sub>	True

We can now go back to P1. Need1 (0,2,1,1) is less than work, so work and finish can be updated:

Work vector	Finish matrix	
1	P0	True
14	P1	True
10	P2	False
7	P3	True
	P4	True

Finally, Need2 (1,0,0,1) is less than work, so we can also accommodate this. Thus, the system is in a safe state when the processes are run in the following order:P0,P3,P4,P1,P2. We therefore can grant the resource request.

## Deadlock Detection & Recovery

If a system neither uses deadlock prevention nor a deadlock avoidance strategy then it might enter into a deadlock. Thus, it becomes important to use deadlock detection and recovery policy. A Deadlock detection algorithm, determines whether the system is in deadlock state or not? If yes, then the system uses a deadlock recovery method to break the deadlock.

### Deadlock Detection

There are two deadlock detection methods depending upon the number of instances of each resource:

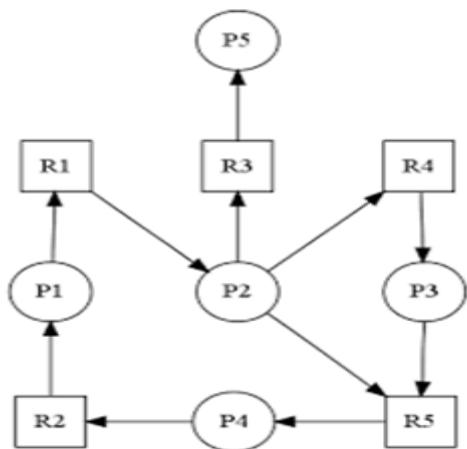
1. Single instance of each resource
2. Multiple instance of each resource

#### Single Instance of Each Resource: *wait-for-graph*

When there is a single instance of each resource the system can use wait-for-graph for deadlock detection. The key points about the wait-for-graph are:

1. Obtained from resource allocation graph
2. Remove the resource nodes – from the resource allocation graph remove the resource nodes
3. Collapse the corresponding edges

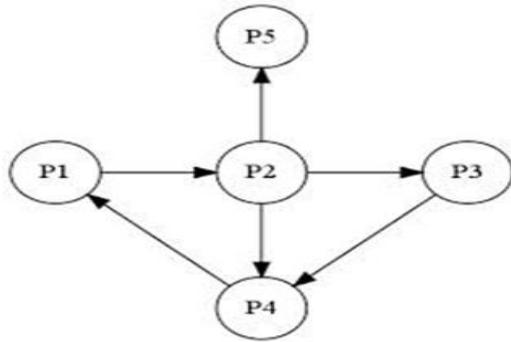
### Resource Allocation Graph



An Edge from Pi to Pj means that process Pi is waiting for a resource held by process Pj. Now, if there are two edges Pi → Rq and Rq → Pj in resource allocation graph, for some resource Rq the collapse these into one single edge from Pi → Pj to make the wait-for-graph. Finally, if there is a cycle in wait-for-graph, then the system is in deadlock else not.

For example, in the above figure, P2 is requesting R3 which is held by P5. Hence, we remove the resource R3 and collapse the edge between P2 and P5. wait-for-graph reflects this change. Similarly, the collapse of edges between P4 and P1 and all other processes is done.

### Wait for Graph



### Multiple Instance of Each Resource

This algorithm is similar to Banker's Algorithm.( **Deadlock Detection Algorithm**)

It uses three data structures:

1. *Available*: A vector of length  $m$  indicates the number of available resources of each type.
2. *Allocation*: An  $n \times m$  matrix defines the number of resources of each type currently allocated to each process
3. *Request*: An  $n \times m$  matrix indicates the current request of each process. For example, if  $\text{Request}[ij] = k$ , then process  $P_i$  is requesting  $k$  more instances of the resource type.  $R_j$ .

### Deadlock Detection Algorithm

1. Let **Work** and **Finish** be vectors of length  $m$  and  $n$ , respectively

**Initialize:**

(a) **Work** = **Available**

(b) For  $i = 1, 2, \dots, n$ , if  $\text{Allocation}_{i,i} \neq 0$ , then

**Finish**[ $i$ ] = **false**; otherwise, **Finish**[ $i$ ] = **true**.

2. Find an index  $i$  such that both:

(a) **Finish**[ $i$ ] == **false**

(b)  $\text{Request}_{i,i} \leq \text{Work}$

If no such  $i$  exists, go to step 4.

3. **Work** = **Work** + **Allocation** <sub>$i$</sub>

**Finish**[ $i$ ] = **true**

**go to step 2.**

**4. If  $\text{Finish}[i] == \text{false}$ , for some  $i$ ,  $1 \leq i \leq n$ , then the system is in deadlock state.**

**Moreover, if  $\text{Finish}[i] == \text{false}$ , then  $P_i$  is deadlocked.**

**Algorithm requires an order of  $O(m \times n^2)$  operations to detect whether the system is in deadlocked state.**

### **Example 1 of deadlock detection**

Consider 5 processes  $P_0$  through  $P_4$ ; 3 resource types  $A$ ,  $B$  and  $C$ . There are 7 instances of  $A$ , 2 instances of  $B$  and 6 instances of  $C$ .

The snapshot at time  $T_0$  is given below:

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	$A \ B \ C$	$A \ B \ C$	$A \ B \ C$
$P_0$	0 1 0	0 0 0	0 0 0
$P_1$	2 0 0	2 0 2	
$P_2$	3 0 3	0 0 0	
$P_3$	2 1 1	1 0 0	
$P_4$	0 0 2	0 0 2	

We now simulate the algorithm for the above example. Initially,  $\text{work} = (0,0,0)$ ;  $\text{Finish}[i] = \text{false}$  for  $i = 0, 1, 2, 3, 4$   $i = 0$

We check if  $\text{Request}_0 \leq \text{work}$ ? Yes

Therefore,  $\text{Work} = \text{Work} + \text{Allocation}_0 = (0,0,0) + (0,1,0) = (0,1,0)$   $\text{Finish}[0] = \text{true}$ ,  $P_0$  added to safe sequence  $\langle P_0 \rangle$

$\text{Work} = (0,1,0);$

Is  $\text{Request}_1 \leq \text{work}$ ? No

Since  $\text{Request}_1$  is not less than Available, check the next process.

$\text{Work} = (0,1,0);$

Is  $\text{Request}_2 \leq \text{work}$ ? Yes

$\text{Work} = \text{Work} + \text{Allocation}_2 = (0,1,0) + (3,0,3) = (3,1,3)$

$\text{Finish}[2] = \text{true}$ ,  $P_2$  added to safe sequence  $\langle P_0, P_2 \rangle$

Work = (3,1,3);

Is Request3  $\leq$  work? Yes

Work = Work + Allocation3 = (3,1,3) + (2,1,1) = (5,2,4) Finish[3] = true,

$P_3$  added to safe sequence and the safe sequence is now  $\langle P_0, P_2, P_3 \rangle$

Work = (5,2,4);

Is Request4  $\leq$  work? Yes

Work = Work + Allocation4 = (5,2,4) + (0,0,2) = (5,2,6) Finish[4] = true,

$P_4$  added to safe sequence and the safe sequence is  $\langle P_0, P_2, P_3, P_4 \rangle$

Now, we check again from the beginning all the other processes that were not added to the safe sequence.

Work = (5,2,6);

Is Request1  $\leq$  work? Yes

Work = Work + Allocation1 = (5,2,6) + (2,0,0) = (7,2,6)

Finish[1] = true,

$P_1$  added to safe sequence and the safe sequence now is  $\langle P_0, P_2, P_3, P_4, P_1 \rangle$

Sequence  $\langle P_0, P_2, P_3, P_4, P_1 \rangle$  now results in  $Finish[i] = \text{true}$  for all  $i$ .

There can be more than one safe sequence, that is there can be correct safe sequences other than  $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ . We have found one safe sequence. Since there is at least one safe sequence, the system is in a safe state. There is no deadlock in the system.

### Example 2 of deadlock detection

Let process  $P_2$  now make an additional request for an instance of resource type C. The Request matrix is changed as shown below, after including the request of an instance of resource type C by process  $P_2$ .

		<u>Request</u>		
		A	B	C
$P_0$		0	0	0
$P_1$	2	0	2	

*P*<sub>2</sub> 0 0 1

*P*<sub>3</sub> 1 0 0

*P*<sub>4</sub> 0 0 2

Now, let us check if the system will be in a safe state. The deadlock detection algorithm is run again.

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	A B C	A B C	A B C
<i>P</i> <sub>0</sub>	0 1 0	0 0 0	0 0 0
<i>P</i> <sub>1</sub>	2 0 0	2 0 2	
<i>P</i> <sub>2</sub>	3 0 3	0 0 1	
<i>P</i> <sub>3</sub>	2 1 1	1 0 0	
<i>P</i> <sub>4</sub>	0 0 2	0 0 2	

Initially, Work = Available = (0,0,0); Finish[i] = false for i = 0,1,2,3,4 When i = 0,

Check if Request0  $\leq$  work? Yes

Work = Work + Allocation0 = (0,0,0) + (0,1,0) = (0,1,0)

Finish[0] = true, *P*<sub>0</sub> is added to safe sequence < *P*<sub>0</sub> >

Work is now (0,1,0);

Is Request1  $\leq$  work? No Is

Request2  $\leq$  work? No Is

Request3  $\leq$  work ? No Is

Request4  $\leq$  work? No

Since the request of all the processes cannot be allocated, the system is not in a safe state. Though it is possible to reclaim the resources held by process *P*<sub>0</sub>, there are insufficient resources to fulfill other processes' requests. Thus, a deadlock exists, consisting of processes *P*<sub>1</sub>, *P*<sub>2</sub>, *P*<sub>3</sub>, and *P*<sub>4</sub>.

## 2. When to invoke detection algorithm?

1. Every time a process requests for a resource

- Advantages

- It is easy to identify the processes involved in the deadlock
  - Also, the process which caused the deadlock is known. This is possible because the system invokes the algorithm after each request.
  - Disadvantage:Overhead in computation time
2. After fixed-interval of time
- Disadvantage:Can not tell which process caused the deadlock

### **3. Deadlock Recover**

If a deadlock is detected in the system, the next step is to recover from it. There are two methods to do so:

1. Process termination
2. Preempt resources from some of the deadlocked processes

#### **3.1 Process Termination**

1. Terminate all the deadlocked processes.
2. Terminate processes one by one until the deadlock is broken. However, the issue is to decide which process to terminate. Certain factors that can be used are:
  - How long the process has computed?
  - How much the process has finished its working?
  - What is the priority of the process?
  - How many resources are being used by the process?
  - How many total processes will be required to be terminated?

#### **3.2 Resource preemption**

In resource preemption some resources are preempted from certain processes and these resources can then be allocated to other process. But, certain issues that should be addressed are:

1. Select a victim – which process to select for preempting the resources.
2. Rollback – the process whose resources are preempted, can not continue execution. So, a process
  - must be rolled back to some safe state.
  - Or abort the process and restart
3. Starvation – it must be ensured that the same process does not get selected for resource preemption, as this will lead to starvation.

## List of Practice Programs

1. Write a C program to create child process and allow parent process to display “parent” and the child to display “child” on the screen
2. Write a C program to Calculate average waiting time & Turn around time using FCFS Scheduling.
3. Write a C program to Calculate average waiting time & Turnaround time using SJF Scheduling.

### **fork()**

It is the primary method of process creation on Unix-like operating systems. This function creates a new copy called the child out of the original process, that is called the parent. When the parent process closes or crashes for some reason, it also kills the child process.

#### **Syntax:**

```
pid=fork();
```

- The operating system is using a unique id for every process to keep track of all processes. And for that, fork() doesn't take any parameter and return an int value as following:
- **Zero:** if it is the child process (the process created). **Positive value:** if it is the parent process.
- The difference is that, in the parent process, fork() returns a value which represents the **process ID** of the child process. But in the child process, `fork()` returns the value 0.
- **Negative value:** if an error occurred.

## **First Come First Serve Scheduling**

It is the easy and simple CPU Scheduling Algorithm. In this algorithm, the process which requests the CPU first, gets the CPU first for execution. It can be implemented using FIFO (First-In, First-Out) Queue method.

*A simple real-life example of this algorithm is the cash counter. There is a queue on the counter. The person who arrives first at the counter receives the services first, followed by the second person, then third, and so on. The CPU process also works like this.*

#### **Advantages of FCFS:**

The following are some benefits of using the FCFS scheduling algorithm:

1. The job that comes first is served first.
2. It is the CPU scheduling algorithm's simplest form.
3. It is quite easy to program.

#### **Disadvantages**

1. It is **non-pre-emptive** algorithm, which means the **process priority** doesn't matter.

If a process with very least priority is being executed, more like **daily routine backup** process, which takes more time, and all of a sudden, some other high priority process arrives, like **interrupt to avoid system crash**, the high priority process will have to wait, and hence in this case, the system will crash, just because of improper process scheduling.

2. In FCFS, the Average Waiting Time is comparatively high.

Resources utilization in parallel is not possible, which leads to **Convoy Effect**, (Convoy Effect is a situation where many processes, who need to use a resource for short time are blocked by one process holding that resource for a long time and hence poor resource(CPU, I/O etc) utilization

### **Shortest Job First**

Till now, we were scheduling the processes according to their arrival time (in FCFS scheduling). However, SJF scheduling algorithm, schedules the processes according to their burst time. In SJF scheduling, the process with the lowest burst time, among the list of available processes in the ready queue, is going to be scheduled next.

In case of a tie, it is broken by **FCFS Scheduling**.

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

### **Lab Practice Programs (Ex: 2 & Ex :3)**

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. There are six popular process scheduling algorithms .

- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-First Scheduling
- Shortest Remaining Time
- Priority Scheduling non preemptive
- Priority Scheduling preemptive
- Round Robin(RR) Scheduling

**The FCFS algorithm** is the simplest of scheduling algorithms in OS. This is because the deciding principle behind it is just as its name suggests- on a first come basis. The job that requests execution first gets the CPU allocated to it, then the second, and so on.

**The Shortest Job First (SJF)** is a CPU scheduling algorithm that selects the shortest jobs on priority and executes them. The idea is that jobs with short burst time get done quickly, making CPU available for other, longer jobs/ processes.

**Shortest remaining time (SRT)** is the preemptive version of the SJF algorithm.The processor is allocated to the job closest to completion but it can be preempted by a newer ready job with shorter time to completion.

**Priority scheduling is a non-preemptive algorithm** and one of the most common scheduling algorithms in batch systems.Each process is assigned a priority. Process with highest priority is to be executed first and so on.Processes with same priority are executed on first come first served basis.

**Preemptive Priority CPU Scheduling Algorithm** is a pre-emptive method of CPU scheduling algorithm that works **based on the priority** of a process. In this algorithm, the scheduler schedules the tasks to work as per the priority, which means that a higher priority process should be executed first. In case of any conflict, i.e., when there is more than one process with equal priorities, then the pre-emptive priority CPU scheduling algorithm works on the basis of FCFS (First Come First Serve) algorithm.

**Round Robin** is the pre-emptive process scheduling algorithm. Each process is provided a fix time to execute, it is called a **quantum**. Once a process is executed for a given time period, it is preempted and other process executes for a given time period.

***Identify the type of CPU scheduling algorithm for each of the scenario and implement the same in C Language. Calculate Average Turn Around time and Average Waiting time for each of the scheduling algorithm by taking arrival times accordingly.***

***Calculate the times by using gantt chart in observation and crosscheck the same with the output of the program.***

1.Let's consider a school computer lab where students use computers for various educational activities, including research, programming assignments, and multimedia projects based on their arrival times. Here are some example tasks (student activities) that the lab server might receive throughout the day:

- **Student A needs to complete an online quiz:** Burst Time = 15 minutes, Arrival Time = 08:30 AM
- **Student B is working on a programming assignment:** Burst Time = 40 minutes, Arrival Time = 08:40 AM
- **Student C wants to edit a multimedia project:** Burst Time = 25 minutes, Arrival Time = 08:50 AM
- **Student D needs to research for a presentation:** Burst Time = 30 minutes, Arrival Time = 09:00 AM

**Objective:**

Your task as a developer is to help the students to carry their tasks by using a scheduling algorithm. Calculate the waiting time and turnaround time for each task, and determine the average waiting and turnaround time for the day's tasks.

2.Imagine an animation studio, Pixie Animations, which specializes in creating short animated films and digital commercials. Due to the studio's growing popularity and increased client demands, the animation rendering tasks have piled up. Throughout a typical day, various rendering tasks are submitted to the server. Each task has a specific duration determined by the complexity of the animation scenes involved. The goal is to minimize the overall waiting time and maximize resource utilization by strategically scheduling the tasks based on their duration. Write the Corresponding CPU Scheduling algorithm.

Here are some example tasks that the server might receive on a typical day:

- **Task A - Rendering a simple character animation:** Burst Time = 15 minutes, Arrival Time = 09:00 AM
- **Task B - Rendering a complex explosion scene:** Burst Time = 30 minutes, Arrival Time = 09:10 AM
- **Task C - Rendering a detailed background scene:** Burst Time = 20 minutes, Arrival Time = 09:15 AM

- **Task D - Rendering a short dialogue scene:** Burst Time = 10 minutes, Arrival Time = 09:30 AM

3. Imagine an Emergency Medical Dispatch Center (EMDC) tasked with coordinating emergency medical responses across a large urban area. The center receives calls about various emergencies—accidents, medical crises, and urgent patient transfers. Identify the algorithm helps prioritize shorter-duration tasks, allowing the center to quickly dispatch assistance for multiple emergencies, potentially saving more lives by reducing the response time.

#### **Task Simulation:**

Here are some examples of calls received by the EMDC one morning:

1. **Call A - Rapid response for a cardiac arrest at a home:** Initial Burst Time = 7 minutes, Arrival Time = 08:00 AM
  2. **Call B - Ambulance dispatch for a minor car accident:** Initial Burst Time = 5 minutes, Arrival Time = 08:01 AM
  3. **Call C - Coordinate a helicopter for a critical transfer from a rural area:** Initial Burst Time = 3 minutes, Arrival Time = 08:02 AM
  4. **Call D - Send paramedics to a multiple injury accident on a highway:** Initial Burst Time = 20 minutes, Arrival Time = 08:15 AM
  5. **Call E - Dispatch an ambulance for a sudden stroke case:** Initial Burst Time = 10 minutes, Arrival Time = 08:20 AM
4. Airlines categorize passengers into different groups based on various criteria such as frequent flyer status, class of service, special needs, and more. Each group is assigned a priority level, and within that priority, boarding is handled in the order that passengers arrive at the gate
- **Priority Assignment:**
    1. **First Priority:** First-class passengers, high-tier frequent flyers, and passengers requiring special assistance (e.g., wheelchair users). Arrival time: 3 and assistance time: 5min
    2. **Second Priority:** Business class and mid-tier frequent flyers. Arrival time: 0 and assistance time: 10min
    3. **Third Priority:** Premium economy passengers. Arrival time: 4 and assistance time: 7min
    4. **Fourth Priority:** Economy class passengers. Arrival time: 1 and assistance time: 8min
    5. **Final Priority:** Basic economy or last-minute ticket purchasers, Arrival time: 5 and assistance time: 6min

5. An IT help desk in a large corporation handles a range of technical issues and requests from employees. Given the varied nature and urgency of these issues, which scheduling system is employed to ensure that critical IT support requests are handled promptly, while less urgent issues are addressed in an orderly fashion. IT staff can switch between tasks based on real-time reevaluation of priorities as new tickets arrive.

*Considering this as the inputs: Calculate Average Turn Around time and Average Waiting time*

<b>Process Id</b>	<b>Priority</b>	<b>Arrival Time</b>	<b>Burst Time</b>
1	2(L)	0	1
2	6	1	7
3	3	2	3
4	5	3	6
5	4	4	5
6	10(H)	5	15
7	9	15	8

6. A local library system incorporates several branches, each equipped with computers used for catalog searches, digital archives access, and various educational programs. During the peak hours of library operation, patrons often need to use computers for tasks like research, document processing, or accessing public records. The library staff sets up a system where each user is allotted a 2-minute slot. If a user's task is not completed within this period, they go back to the end of the queue for another turn. identify the scheduling algorithm.

<b>Process Id</b>	<b>Arrival time</b>	<b>Burst time</b>
P1	0	5

P2	1	3
P3	2	1
P4	3	2
P5	4	3

7. Assume you own a small community bookstore that regularly needs to update its inventory records at the end of the business day. The inventory update is divided into two main tasks: tallying sales and restocking items. Due to the store's unique cataloging system, these tasks need to be handled sequentially. Create a simple application that uses UNIX/Linux processes, to separate the tallying of sales from the restocking of items.

**Expected Output:**

**Parent Output:** "Daily sales tallied successfully."

**Child Output:** "Items restocked based on sales data."

**Final Output by Parent:** "Inventory update complete."

#### **Exercise : 4**

- 4a. Write a C program that illustrate communication between two unrelated process using named pipes  
4b. Write a C program that receives a message from message queue and display them

***These Two programs are to be run on Linux terminal by creating two programs sender and receiver for passing the data .Compile and run the programs separately on two different terminals.***

***The Remaining three Programs are to run in WSL Terminal.***

- 4c. Write a C program to allow cooperating process to lock a resource for exclusive use (using semaphore)..implement this for bank account transaction system

**Requirements:**

1. The initial account balance is \$1000.
2. Use POSIX semaphores to synchronize access to the shared account balance.
3. Implement two functions: **deposit(int amount)** and **withdraw(int amount)**:
  - **deposit(int amount)**: Increases the account balance by the specified **amount** and prints the new balance.
  - **withdraw(int amount)**: Decreases the account balance by the specified **amount** if there are sufficient funds. If not, it prints an error message indicating insufficient funds.
  - The child process should perform a deposit & the parent process should attempt to withdraw.

**Ouput:**

Withdrawing 200 from account  
New balance after withdrawal: 800  
Depositing 500 to account  
New balance after deposit: 1500

- 4d. Write a C program that illustrate the suspending and resuming process using signal

Program demonstrates creating a child process using **fork()**. The parent process should be able to:

1. Properly handle errors for **fork()** and other system calls.
  - The parent process should manage the child process by using **SIGSTOP**, **SIGCONT**, **Wait()**

## **Output**

Parent process started  
Child process is running...  
Parent has suspended the child process  
Parent has resumed the child process

4e. Write a C program that implements producer-Consumer system with two process using semaphore

The program uses POSIX semaphores to synchronize a producer process and a consumer process. The producer should generate a simple integer to simulate producing an item and place it in a buffer. The consumer should then read this integer from the buffer to simulate consuming the item. To simplify, the buffer can hold only one item at a time.

### **Requirements:**

1. Use two POSIX named semaphores:
  - **semProducer**: This semaphore should indicate when the buffer is empty and it's safe for the producer to add a new item.
  - **semConsumer**: This semaphore should indicate when the buffer contains an item that needs to be consumed.
2. Manage process creation using **fork()** and ensure the parent process waits for the child process to complete before it exits.

## **Output**

Producing item  
Consuming item  
Producing item  
Consuming item  
Producing item  
Consuming item  
Producing item  
Consuming item  
Producing item  
Consuming item

## **Named Pipe**

- Usually a named pipe appears as a file and generally processes attach to it for inter-process communication. A FIFO file is a special kind of file on the local storage which allows two or more processes to communicate with each other by reading/writing to/from this file.
- A FIFO special file is entered into the filesystem by calling *mkfifo()* in C. Once we have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it.

### **Message queue**

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. A new queue is created or an existing queue opened by **msgget()**.

System calls used for message queues:

**ftok()**: is use to generate a unique key.

**msgget()**: either returns the message queue identifier for a newly created message queue or returns the identifiers for a queue which exists with the same key value.

**msgsnd()**: Data is placed on to a message queue by calling **msgsnd()**.

**msgrcv()**: messages are retrieved from a queue.

**msgctl()**: It performs various operations on a queue. Generally it is use to destroy message queue.

### **Semaphores**

The POSIX system in Linux presents its own built-in semaphore library. To use it, we have to :

1. Include **semaphore.h**
2. Compile the code by linking with -lpthread -lrt

To lock a semaphore or wait we can use the **sem\_wait** function:

```
int sem_wait(sem_t *sem);
```

To release or signal a semaphore, we use the **sem\_post** function:

```
int sem_post(sem_t *sem);
```

A semaphore is initialised by using **sem\_init**(for processes or threads) or **sem\_open** (for IPC).

```
sem_init(sem_t *sem, int pshared, unsigned int value);
```

Where,

- **sem** : Specifies the semaphore to be initialized.

- **pshared** : This argument specifies whether or not the newly initialized semaphore is shared between processes or between threads. A non-zero value means the semaphore is shared between processes and a value of zero means it is shared between threads.
- **value** : Specifies the value to assign to the newly initialized semaphore.

To destroy a semaphore, we can use **sem\_destroy**.

```
sem_destroy(sem_t *mutex);
```

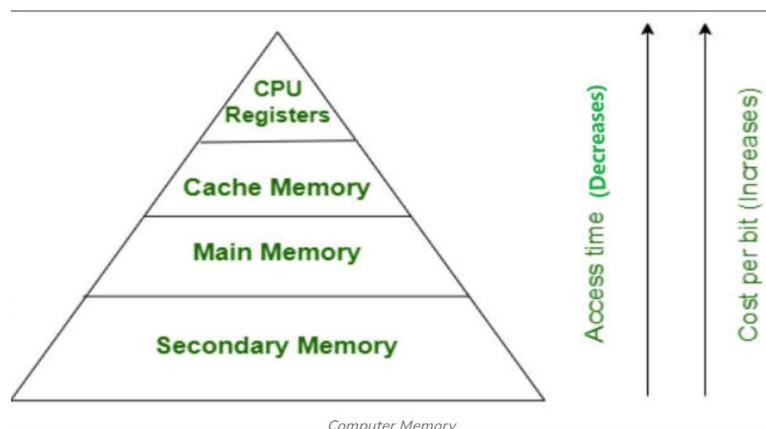
To declare a semaphore, the data type is **sem\_t**.

**Unit-4**  
**Memory Management and Virtual Memory**  
**Topic 1**

The term memory can be defined as a collection of data in a specific format. It is used to store instructions and process data. The memory comprises a large array or group of words or bytes, each with its own location. The primary purpose of a computer system is to execute programs. These programs, along with the information they access, should be in the main memory during execution. The CPU fetches instructions from memory according to the value of the program counter.

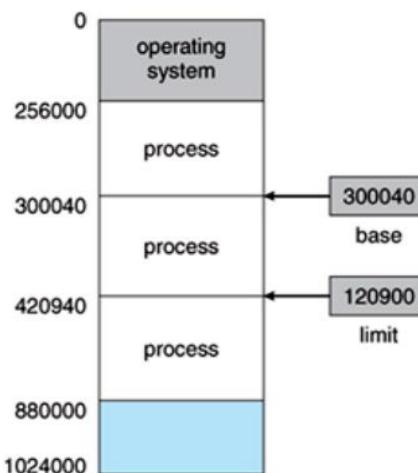
#### **Basic Hardware**

- The CPU can only access its registers and main memory. It cannot, for example, make direct access to the hard drive, so any data stored there must first be transferred into the main memory chips before the CPU can work with it.
- Memory accesses to registers are very fast, generally one clock tick, and a CPU may be able to execute more than one machine instruction per clock tick.
- Memory accesses to main memory are comparatively slow, and may take a number of clock ticks to complete. In such cases, the processor normally needs to stall, since it does not have the data required to complete the instruction that it is executing. This situation is intolerable because of the frequency of memory accesses. The remedy is to add fast memory between the CPU and main memory. A memory buffer used to accommodate a speed differential, called a cache memory.



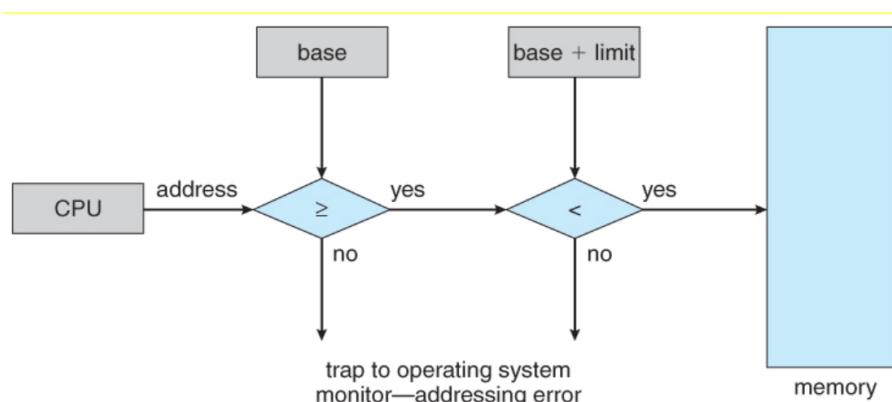
Not only are we concerned with the relative speed of accessing physical memory, but we also must ensure correct operation to protect the operating system from access by user processes and, in

addition, to protect user processes from one another. This protection must be provided by the hardware. We first need to make sure that each process has a separate memory space. To do this, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses. We can provide this protection by using two registers, usually a base and a limit.



The base register holds the smallest legal physical memory address; the limit register specifies the size of the range. For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).

Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers. Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error .



This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users. The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction. Since privileged instructions can be executed only in kernel mode, and since only the operating system executes in kernel mode, only the operating system can load the base and limit registers.

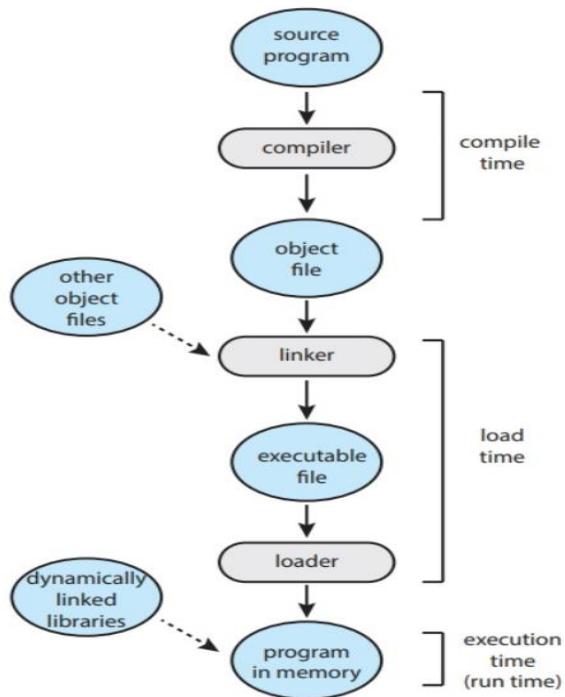
## **Address Binding**

**Address Binding** is the association of program instructions and data to the actual physical memory location. There are various types of address binding in the operating system.

There are 3 types of Address Binding:

1. **Compile Time Address Binding:** If you know at compile time where the process will reside in memory, then absolute code can be generated. For example, if you know that a user process will reside starting at location R, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code. The MS-DOS .COM-format programs are bound at compile time.
2. **Load Time Address Binding:** If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code. In this case, final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value
3. **Execution Time Address Binding:** If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work.

**Fig : Multistep processing of a user program.**



## Logical versus Physical Address Space

### Logical Address

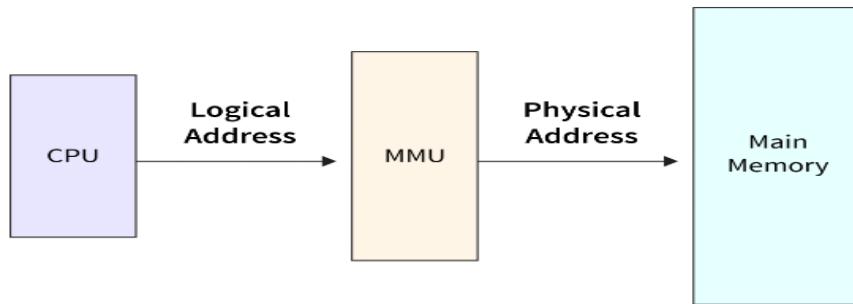
- Logical address is generated by CPU while a program is running.
- The logical address is virtual address as it does not exist physically it is also known as Virtual Address.
- This address is used as a reference to access the physical memory location by CPU.
- Logical Address Space is set of all logical addresses generated by a program.

### Physical Address:

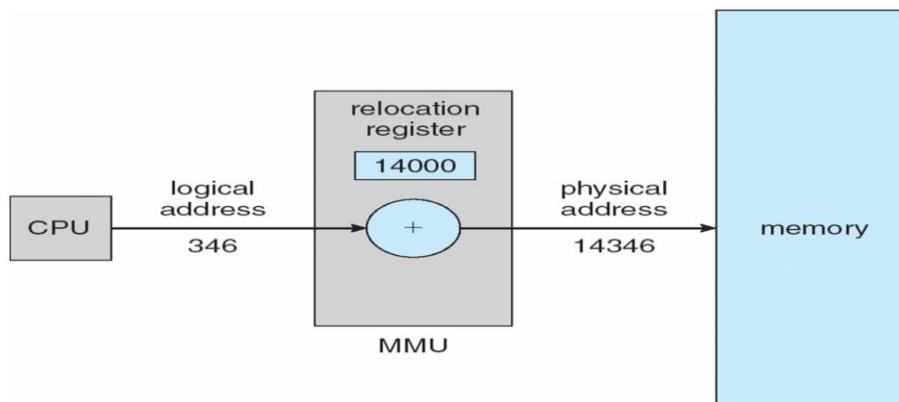
- Physical address identifies a physical location of required data in a memory.
- The user never directly deals with the physical address but can access by its corresponding logical address.
- The set of all physical addresses corresponding to these logical addresses is physical address space

The user program generates the logical address and thinks that the program is running in this logical address but the program needs physical memory for its execution, therefore the logical address must

be mapped to the physical address by MMU before they are used. Memory Management Unit or address translation unit is a hardware device used for mapping logical address to its corresponding physical address.



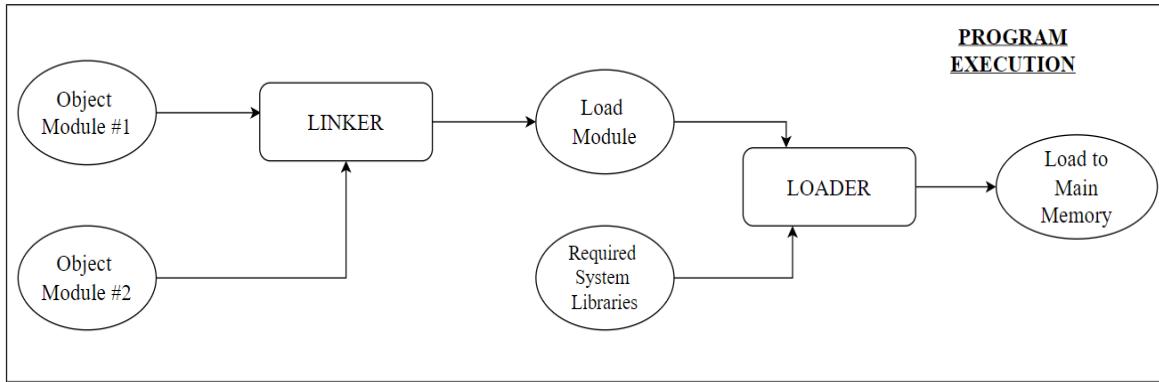
Let us understand the concept of mapping with the help of a simple MMU scheme and that is a **base-register scheme**.



In the above diagram, the base register is termed the **Relocation register**. The value in the relocation register is added to every address that is generated by the user process at the time when the address is sent to the memory.

Suppose the **base is at 14000**, then an attempt by the user to address location **0** is relocated dynamically to **14000**; thus access to location **346** is mapped to **14346**.

## Linking and Loading



Working of loading and linking

Linking is a process of collecting and maintaining pieces of code and data into a single file. It is performed by a linker. A linker is special program that combines the object files, generated by compiler/assembler and other pieces of code to originate an executable file has .exe extension. In the context of operating systems, linking can be done in two main ways: static linking and dynamic linking.

**Static Linking:** : *Combines all necessary library code with the program code at compile time.*

**Process:** The linker takes all the object files and libraries and merges them into a single executable file. All external references are resolved during compilation.

**Advantages:**

- The executable is self-contained, with no dependencies on external libraries at runtime.
- Simplifies distribution as all necessary code is bundled.

**Disadvantages:**

- Increases the size of the executable file.
- Wastes disk space and memory as each program has its own copy of the library code.
- Updates to libraries require recompilation of the program

**Dynamic Linking:** *Links the necessary library code to the program at runtime rather than at compile time.*

**Process:**

- The program includes stubs (With dynamic linking, **we use references to the library in the system where function definitions are available**. The information about the library in which a function definition exists is stored in stubs) instead of the actual library routines
- When the program runs, the stubs locate and load the required routines from shared libraries.
- If a required routine is already in memory, the stub links to it; otherwise, it loads the routine into memory.
- The stub replaces itself with the address of the routine, allowing direct execution in future calls.

**Advantages:**

- Reduces the size of the executable file.
- Saves disk space and memory as multiple programs can share the same library code in memory.
- Facilitates easy updates to libraries, as programs automatically use the new version without recompilation.
- Multiple versions of a library can coexist, with programs using the appropriate version based on version information.

**Disadvantages:**

- Requires runtime support from the operating system.
- Potentially introduces runtime overhead during the initial linking process.

## **Loading**

To bring the program from secondary memory to main memory is called Loading. It is performed by a loader. It is a special program that takes the input of executable files from the linker, loads it to the main memory, and prepares this code for execution by a computer. There are two types of loading in the operating system:

**Static Loading:** Loading the entire program into the main memory before the start of the program execution is called static loading. If static loading is used then accordingly static linking is applied.

- **Dynamic Loading:** Loading the program into the main memory on demand is called dynamic loading. If dynamic loading is used then accordingly dynamic linking is applied. To obtain better memory-space utilization, we can use dynamic loading. With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format. The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If it has not, the relocatable linking loader

is called to load the desired routine into memory and to update the program's address tables to reflect this change. Then control is passed to the newly loaded routine.

- The advantage of dynamic loading is that an unused routine is never loaded. This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines. In this case, although the total program size may be large, the portion that is used (and hence loaded) may be much smaller.
- Dynamic loading does not require special support from the operating system. It is the responsibility of the users to design their programs to take advantage of such a method. Operating systems may help the programmer, however, by providing library routines to implement dynamic loading.

## Unit-IV

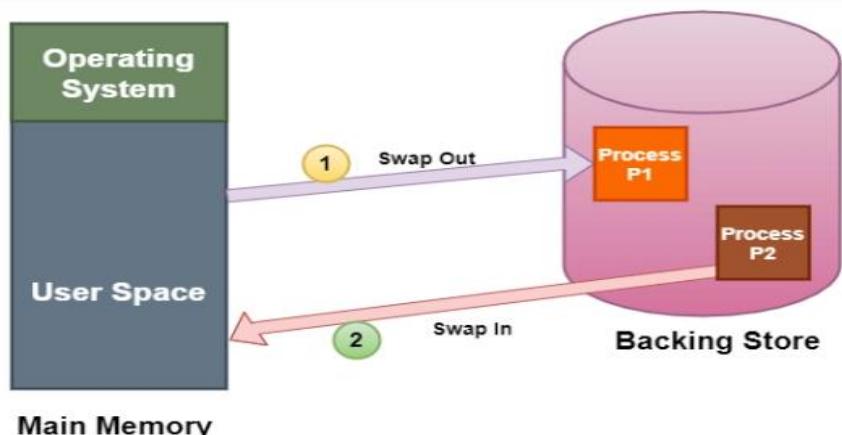
### Topic 2: Swapping and Contiguous Allocation

#### Swapping

**Swapping in OS** is a memory management technique that temporarily swaps processes from main memory to secondary memory or vice versa which helps to increase the degree of multi-programming and increase main memory utilisation.

There are two steps in the process of Swapping in the operating system:

1. **Swap In:** Swap-in is the process of bringing a process from secondary storage/hard disc to main memory (RAM).
2. **Swap Out:** Swap-out takes the process out of the Main memory and puts it in the secondary storage (backing store)



*In the above diagram, process P1 is swap out so the process with more memory requirement or higher priority will be executed to increase the overall efficiency of the operating system. While the process P2 is swap in for its execution from the secondary memory to the main memory(RAM).*

- In a multi-programming system with a *round-robin scheduling algorithm*, when a time slice expires, memory manager starts to swap out the process that have finished and swap in another process to the memory.
- Another swapping policy is algorithm. When any higher priority process arrives for execution, the memory management swaps out the lower priority process so that it can load and execute the higher priority process. After completing the execution of higher priority process, the lower priority

process is swap back and continues its execution. This type of swapping is also called **roll out** (swap out) and **roll in** (swap in).

- Normally a previously swapped out process will be swapped back into the same memory space but it depends on the method. If address binding method is execution time binding then a process can be swapped to a different memory space as physical address are computed during execution time.
- Swapping requires a fast backing store like a disk. It must be large enough to accommodate copies of all memory images for all users.
- When CPU scheduler decides to execute a process, it calls dispatcher that check whether the next process in the queue is in main memory. If it is not and there is not enough free memory available, the dispatcher swaps out any current process and swaps in the desired process.
- If we want to swap a process it must be completely idle. A process waiting for an I/O operation should never be swapped to free-up its memory. If we want to swap out process p1 and swap in process p2, the I/O operation of process p1 might attempt to use memory that belongs to process p2. Main solution of this problem is never to swap a process with pending I/O .

**Example:** Suppose the user process's size is 2048KB and is a standard hard disk where swapping has a data transfer rate of 1Mbps. Now we will calculate how long it will take to transfer from main memory to secondary memory.

User process size is 2048Kb

Data transfer rate is 1Mbps = 1024 kbps

Time = process size / transfer rate

$$= 2048 / 1024 = 2 \text{ seconds} = 2000 \text{ milliseconds}$$

Now taking swap-in and swap-out time, the process will take 4000 milliseconds.

### **Advantages of Swapping**

1. It helps the CPU to manage multiple processes within a single main memory.
2. It helps to create and use virtual memory.
3. Swapping allows the CPU to perform multiple tasks simultaneously. Therefore, processes do not have to wait very long before they are executed.
4. It improves the main memory utilization.

### **Disadvantages of Swapping**

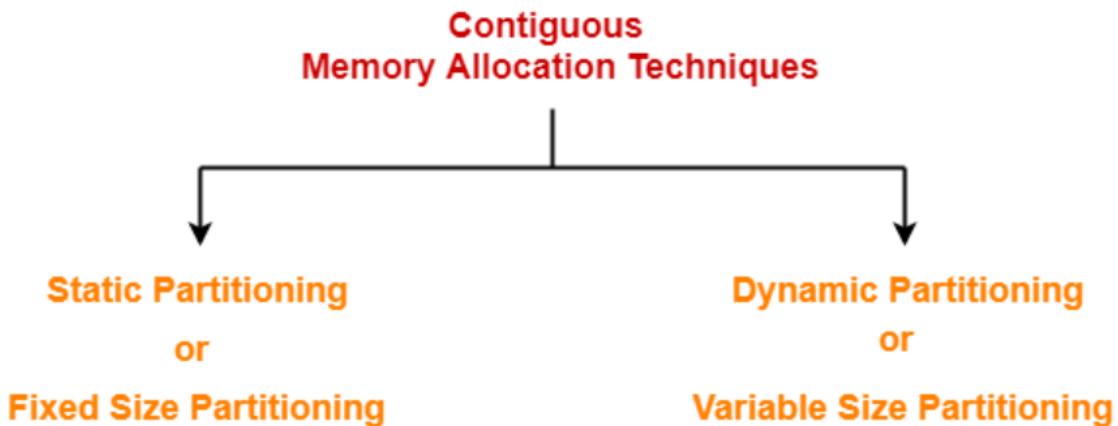
1. If the computer system loses power, the user may lose all information related to the program in case of substantial swapping activity.
2. If the swapping algorithm is not good, the composite method can increase the number of Page Fault and decrease the overall processing performance.

### **Contiguous 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. One such memory allocation technique is contiguous memory allocation. As the name implies, we allocate contiguous blocks of memory to each process using this technique.

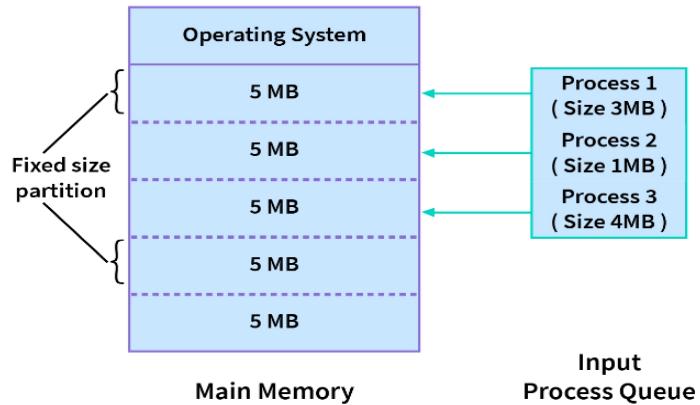
### **Contiguous Memory Allocation Techniques**

Whenever a process has to be allocated space in the memory, following the contiguous memory allocation technique, we have to allot the process a continuous empty block of space to reside. This allocation can be done in two ways:

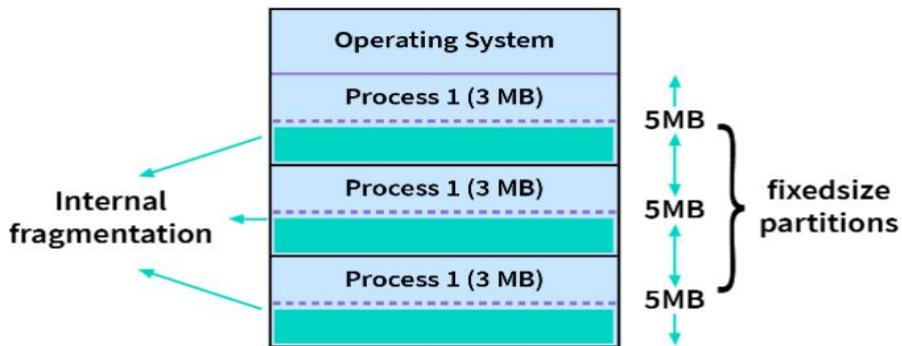


#### **Fixed-size/Static Partitioning**

In this type of contiguous memory allocation technique, **each process is allotted a fixed-size continuous block in the main memory**. That means there will be continuous blocks of fixed size into which the complete memory will be divided, and each time a process comes in, it will be allotted one of the free blocks. Because irrespective of the size of the process, each is allotted a block of the same size memory space. This technique is also called 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.



The first process, which is of size 3MB is also allotted a 5MB block, and the 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.

**Note:** *The number of processes that can stay in the memory at once is called the degree of multiprogramming. Hence, the degree of multiprogramming of the system is decided by the number of blocks created in the memory.*

### Advantages

1. Because all of the blocks are the same size, this scheme is simple to implement. All we have to do now is divide the memory into fixed blocks and assign processes to them.
2. It is easy to keep track of how many blocks of memory are left, which in turn decides how many more processes can be given space in the memory.

- As at a time multiple processes can be kept in the memory, this scheme can be implemented in a system that needs multiprogramming.

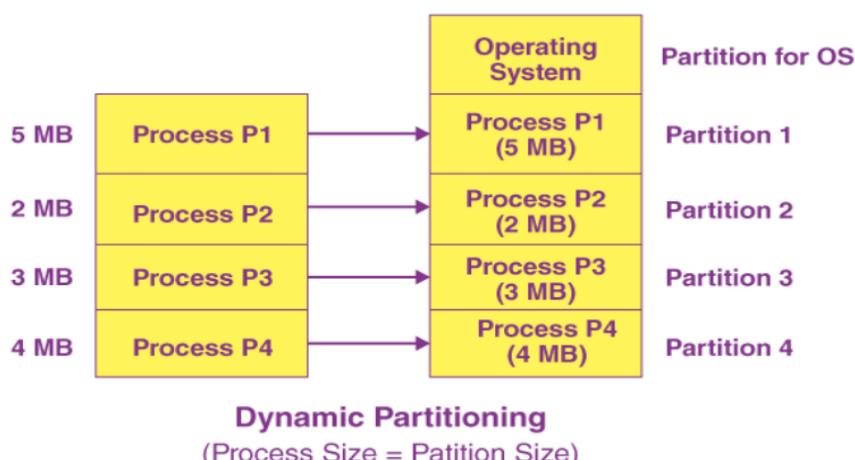
### Disadvantages

- As the size of the blocks is fixed, we will not be able to allot space to a process that has a greater size than the block.
- The size of the blocks decides the degree of multiprogramming, and only that many processes can remain in the memory at once as the number of blocks.
- If the size of the block is greater than the size of the process, we have no other choice but to assign the process to this block, but this will lead to much empty space left behind in the block. This empty space could've been used to accommodate a different process. This is called internal fragmentation. Hence, this technique may lead to space wastage.

***The problem of internal fragmentation may arise due to the fixed sizes of the memory blocks. It may be solved by assigning space to the process via dynamic partitioning. Dynamic partitioning allocates only the amount of space requested by the process. As a result, there is no internal fragmentation.***

### Variable-size /Dynamic Partitioning

In this type of contiguous memory allocation technique, **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.



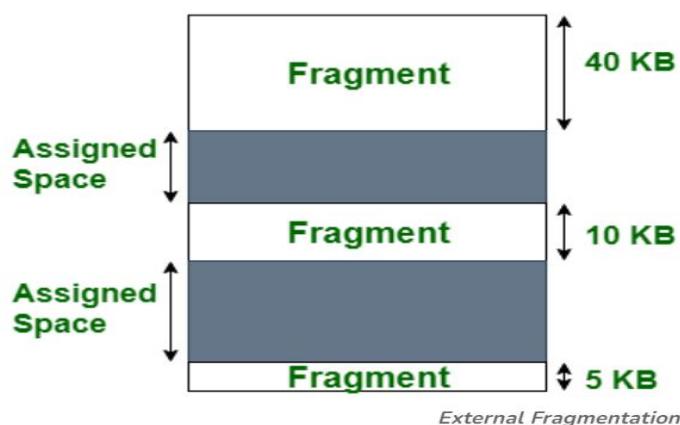
The very first partition has to be reserved for the OS. The left space gets divided into various parts. The actual size of every partition would be equal to the process size. The size of the partition varies according to the requirement of the process. This way, internal fragmentation can be easily avoided.

### Advantages

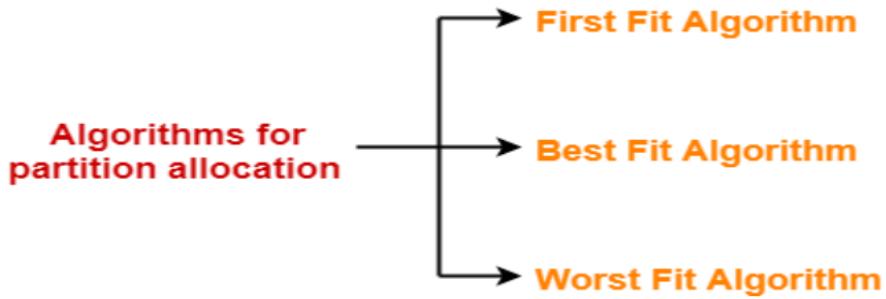
1. As the processes have blocks of space allotted to them as per their requirements, there is no internal fragmentation. Hence, there is no memory wastage in this scheme.
2. The number of processes that can be in the memory at once will depend upon how many processes are in the memory and how much space they occupy. Hence, it will be different for different cases and will be dynamic.
3. As there are no blocks that are of fixed size, even a process of big size can be allotted space.

### Disadvantages

1. Because this approach is dynamic, a variable-size partition scheme is difficult to implement.
2. It is difficult to keep track of processes and the remaining space in the memory.
3. Since there is no internal fragmentation, it doesn't mean there would be no external fragmentation either. Now, let us consider three processes, i.e. P1 (1 MB), P2 (3 MB) and P3 (1 MB), that are being loaded in their respective partitions of the main memory. P1, P2 and P3 get completed after some time, and the assigned space gets freed. Here, we have three unused partitions available (40K, 10K, 5K) in the main memory. However, we can't use them to load 55K process P4 in the memory because they aren't located contiguously.



## Partition Allocation Strategies/Algorithms

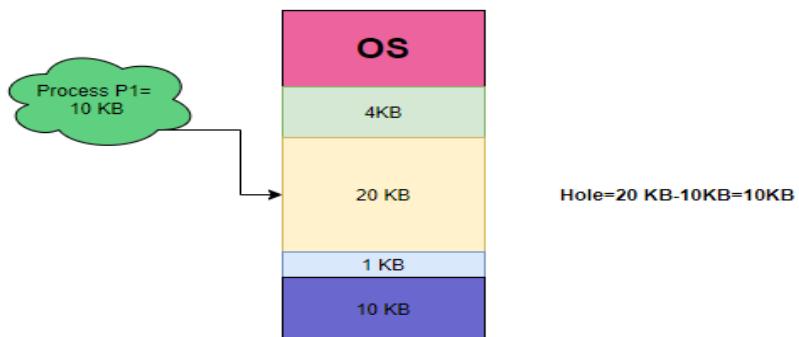


### 1. First Fit Allocation

According to this strategy, allocate the **first hole or first free partition** to the process that is big enough. This searching can start either from the beginning of the set of holes or from the location where the previous first-fit search ended. Searching can be stopped as soon as we find a free hole that is large enough.

*Let us take a look at the example given below:*

Process P1 of size 10KB has arrived and then the first hole that is enough to meet the requirements of size 20KB is chosen and allocated to the process.

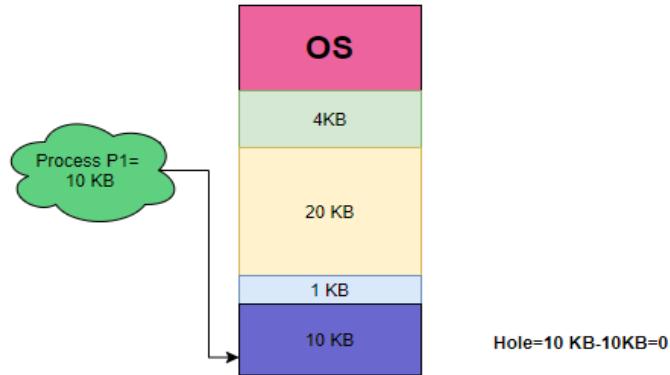


### 2. Best Fit Allocation

With this strategy, the smallest free partition/ hole that is big enough and meets the requirements of the process is allocated to the process. This strategy searches the entire list of free partitions/holes in order to find a hole whose size is either greater than or equal to the size of the process.

*Let us take a look at the example given below:*

Process P1 of size 10KB is arrived and then the smallest hole to meet the requirements of size 10 KB is chosen and allocated to the process.

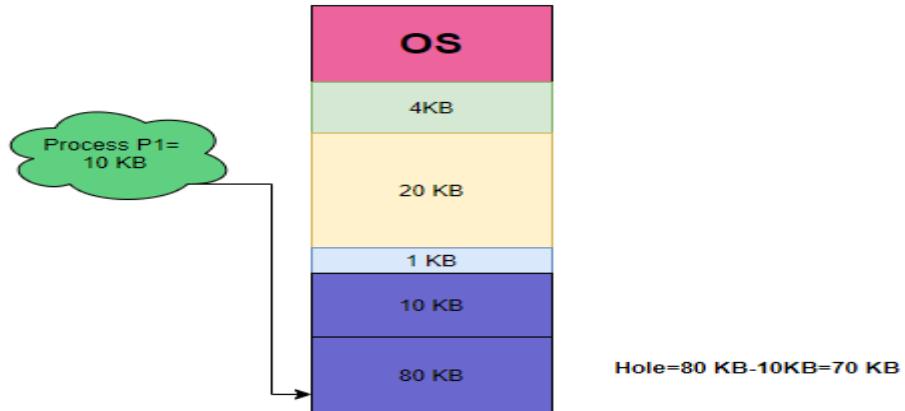


### 3. Worst Fit Allocation

With this strategy, the Largest free partition/ hole that meets the requirements of the process is allocated to the process. It is done so that the portion that is left is big enough to be useful. This strategy is just the opposite of First Fit. This strategy searches the entire list of holes in order to find the largest hole and then allocate the largest hole to process.

*Let us take a look at the example given below:*

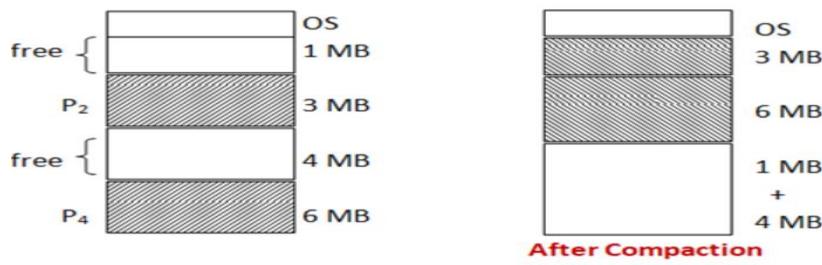
Process P1 of size 10KB has arrived and then the largest hole of size 80 KB is chosen and allocated to the process.



### Compaction

In memory management, swapping creates multiple fragments in the memory because of the processes moving in and out. Compaction is a method for removing external fragmentation.

Compaction is a technique to collect all the free memory present in the form of fragments into one large chunk of free memory, which can be used to run other processes. It does that by moving all the processes towards one end of the memory and all the available free space towards the other end of the memory so that it becomes contiguous.



The compaction process usually consists of two steps:

1. Copying all pages that are not in use to one large contiguous area.
2. Then, write the pages that are in use into the newly freed space.

### Advantages of Compaction

- Reduces external fragmentation.
- Make memory usage efficient.
- Memory becomes contiguous.
- Since memory becomes contiguous more processes can be loaded to memory, thereby increasing scalability of OS.
- Improves memory utilization as there is less gap between memory blocks.

### Disadvantages of Compaction

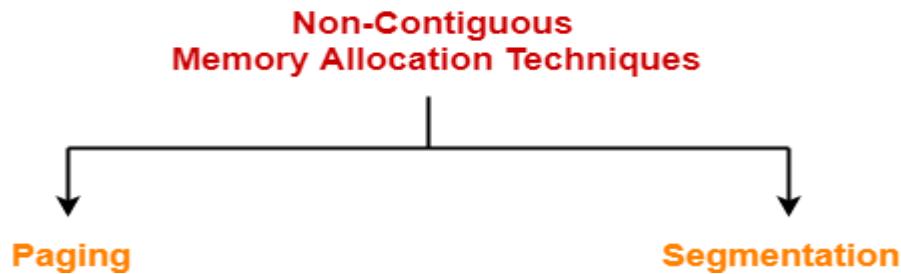
- System efficiency reduces and latency is increased.
- A huge amount of time is wasted in performing compaction.
- CPU sits idle for a long time.
- Not always easy to perform compaction.
- It may cause deadlocks since it disturbs the memory allocation process.

## Unit 4

### Topic 3: Paging-I

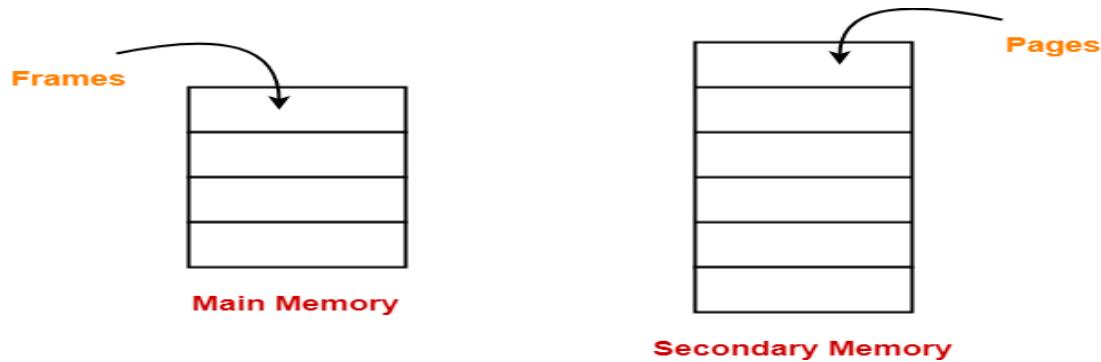
To overcome the problem of External fragmentation in contiguous memory allocation is we go for **non-contiguous memory allocation**. It allows to store parts of a single process in a non-contiguous fashion. Thus, different parts of the same process can be stored at different places in the main memory.

There are two popular techniques used for non-contiguous memory allocation-



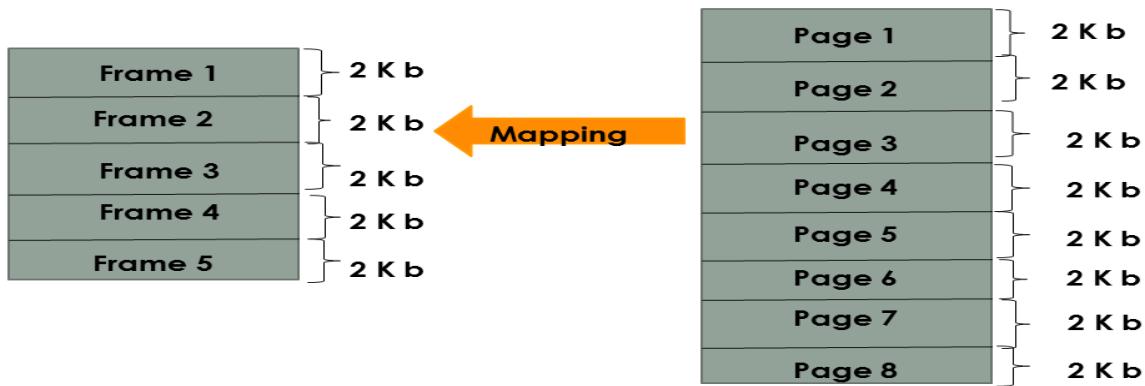
#### Paging

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



The process pages are stored at a different location in the main memory. The thing to note here is that the size of the page and frame will be the same. A page is mapped into a frame

## Basic



CPU always generates a logical address consisting of two parts-

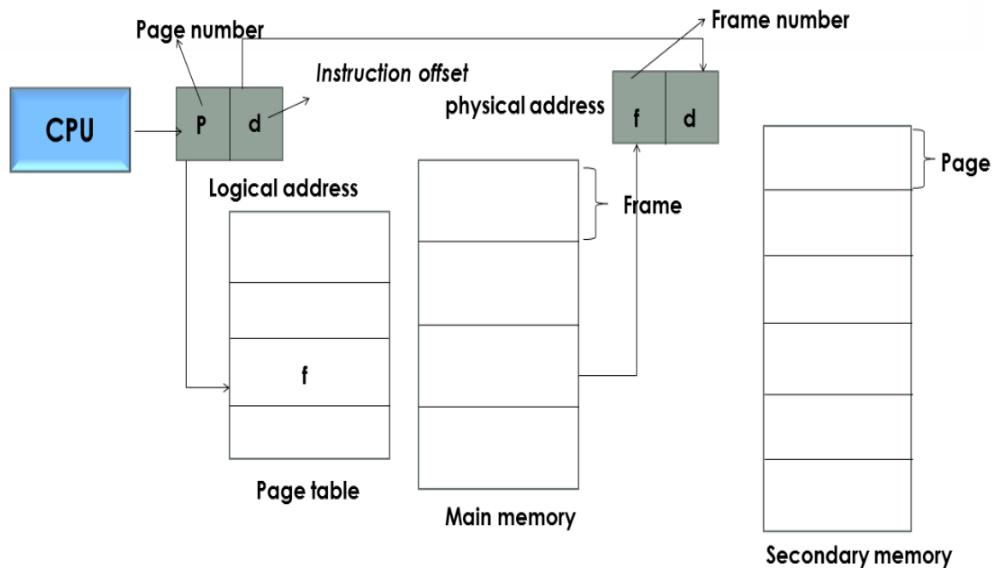
Page Number: specifies the specific page of the process from which CPU wants to read the data.

Page Offset : specifies the specific word/instruction on the page that CPU wants to read.

**A physical address is needed to access the main memory and consists of and offset .**

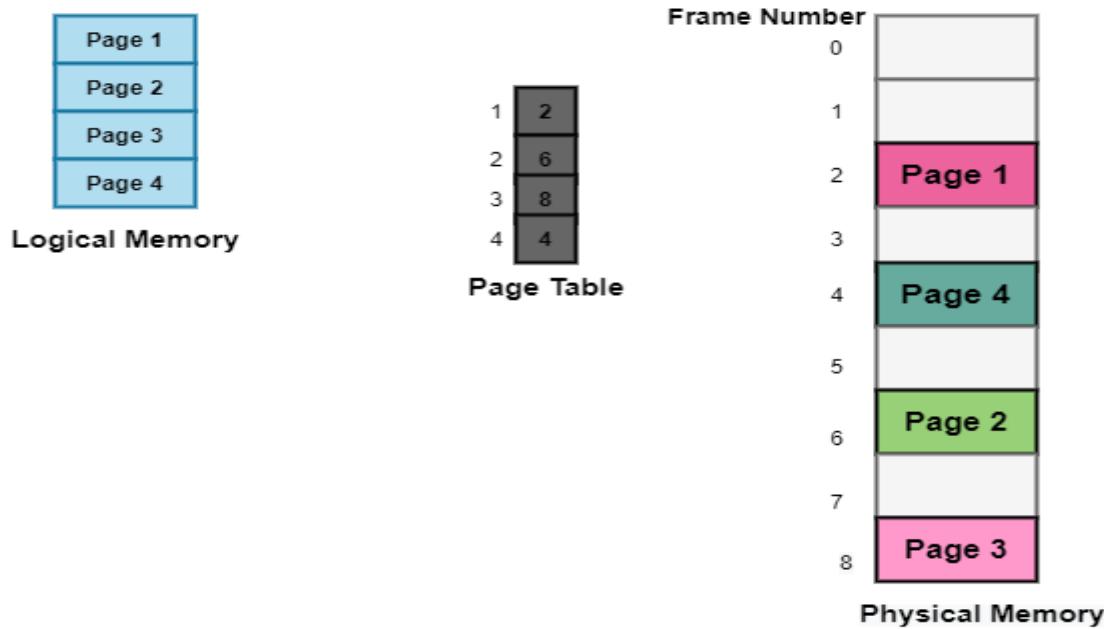
frame number : specifies the specific frame where the required page is stored.

Page Offset : Specifies the specific word that has to be read from that page



The logical address that is generated by the CPU is translated into the physical address with the help of the page table. Thus page table mainly provides the corresponding frame number where that page is stored in the main memory.

## Method



***The above diagram shows the paging model of Physical and logical memory.***

The page size (like the frame size) is defined by the hardware. The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy.

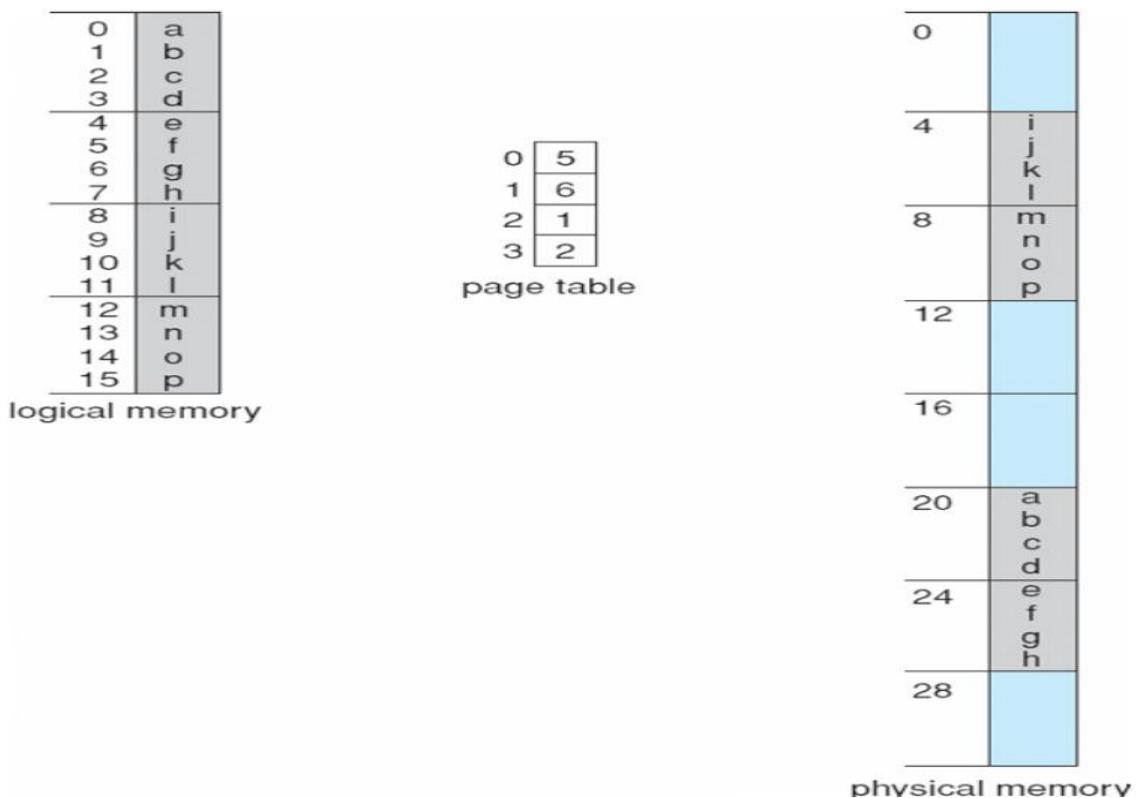
If the size of the logical address space is  $2^m$ , and a page size is  $2^n$  addressing units (bytes or words) then the high-order  $m-n$  bits of a logical address designate the page number, and the  $n$  low-order bits designate the page offset. Thus, the logical address is as follows:



Example, consider the memory in Figure below. Here, in the logical address,  $n=2$  and  $m=4$ . Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the user's view of memory can be mapped into physical memory.

- ♣ Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [=  $(5 \times 4) + 0$ ].
- ♣ Logical address 3 (page 0, offset 3) maps to physical address 23 [=  $(5 \times 4) + 3$ ].
- ♣ Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [=  $(6 \times 4) + 0$ ].

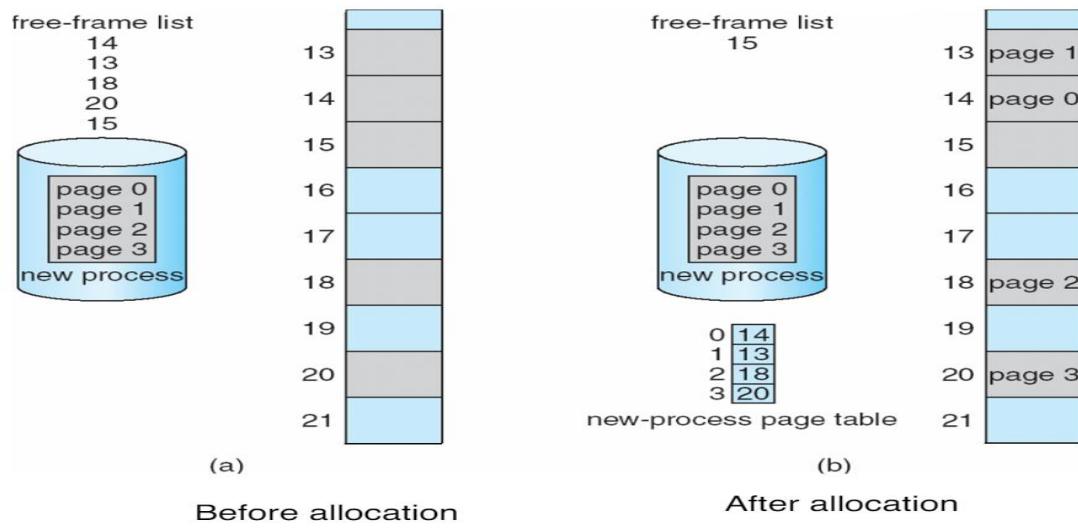
- ♣ Logical address 13 maps to physical address 9.



When we use a paging scheme, we have no external fragmentation: any free frame can be allocated to a process that needs it. However, we may have some internal fragmentation. For example, if page size is 2,048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes. It will be allocated 36 frames, resulting in internal fragmentation of  $2,048 - 1,086 = 962$  bytes. In the worst case, a process would need 11 pages plus 1 byte. It would be allocated 11 + 1 frames, resulting in internal fragmentation of almost an entire frame.

When a process arrives in the system to be executed, its size, expressed in pages, is examined. Each page of the process needs one frame. Thus, if the process requires  $n$  pages, at least  $n$  frames must be available in memory. If  $n$  frames are available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, its frame number is put into the page table, and so on.

## Free Frames



### Hardware Support

In Operating System, for each process page table will be created, which will contain a Page Table Entry (PTE). This PTE will contain information like frame number which will tell where in the main memory the actual page is residing.

Now the question is where to place the page table, such that overall access time (or reference time) will be less. The problem initially was to fast access the main memory content based on the address generated by the CPU (i.e. logical/virtual address). Initially, some people thought of using registers to store page tables, as they are high-speed memory so access time will be less.

The idea used here is, to place the page table entries in registers, for each request generated from the CPU (virtual address), it will be matched to the appropriate page number of the page table, which will now tell where in the main memory that corresponding page resides. Everything seems right here, but the problem is registered size is small (in practice, it can accommodate a maximum of 0.5k to 1k page table entries) and the process size may be big hence the required page table will also be big (let's say this page table contains 1M entries), so registers may not hold all the entries of the Page table. So, this is not a practical approach.

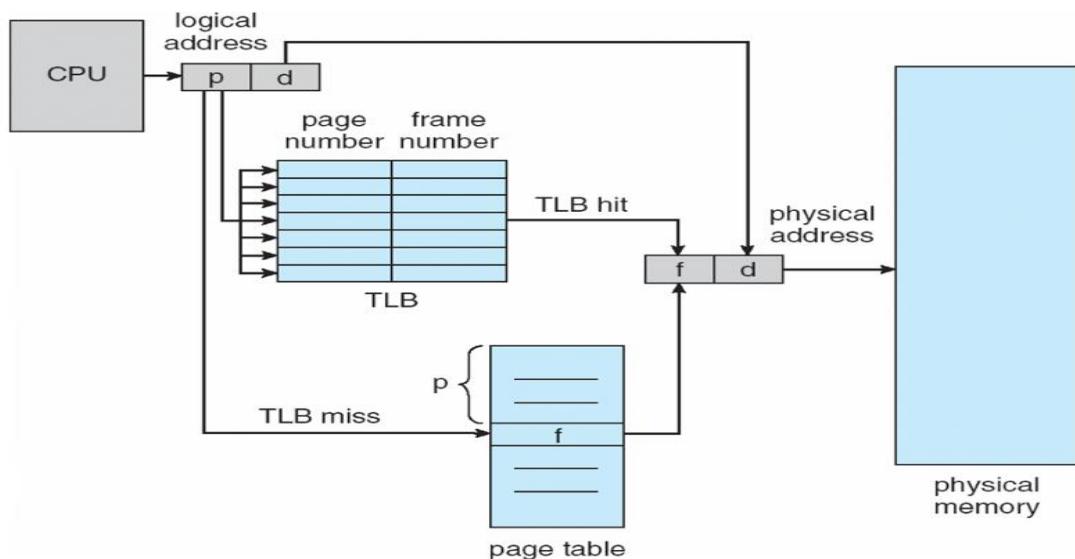
The entire page table was kept in the main memory to overcome this size issue. but the problem here is two main memory references are required:

1. To find the frame number
2. To go to the address specified by frame number

The two-memory access problem can be solved by the use of a special fast-lookup hardware cache called *associative memory* or *translation look-aside buffers (TLBs)*. Each entry in the TLB consists of two parts: **a key** (or tag) and **a value**. When the **associative memory** is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding **value field** is returned. Typically, the number of entries in a **TLB** is small, often numbering between 64 and 1,024!

**The TLB is used with pages tables in the following way :**

### Paging Hardware With TLB



- The TLB contains only a few of the **page-table** entries. When a logical address is generated by the CPU, its page number is presented to the TLB.
- if the page number is found (known as **TLB hit**), its frame number is immediately available and is used to access memory.
- If the page number is not found in the TLB (known as **TLB miss**) a memory reference to the page table must be made.
- when the frame number is obtained, we can use it to access memory.
- In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the **next reference**.
- If the TLB is already full of entries, the operating system must **select one** for replacement.
- Replacement policies range from **least recently used (LRU)** to random.

The percentage of times that a particular page number is found in the TLB is called **TLB Hit Ratio**. Let us now see an example of how effective memory access time is calculated: Assume it takes 20 nsecs to search TLB and 100 nsecs to access memory, what is the effective memory access time, if the hit ratio is 80%?

If the page number is present in the TLB, memory access time = 120 nsecs (20+100)

If page number is not present in the TLB, memory access time = 220 nsecs (20+100+100)

If 80 percent hit ratio, effective memory-access time =  $0.80 \times 120 + 0.20 \times 220 = 140$  nsecs.

## Protection

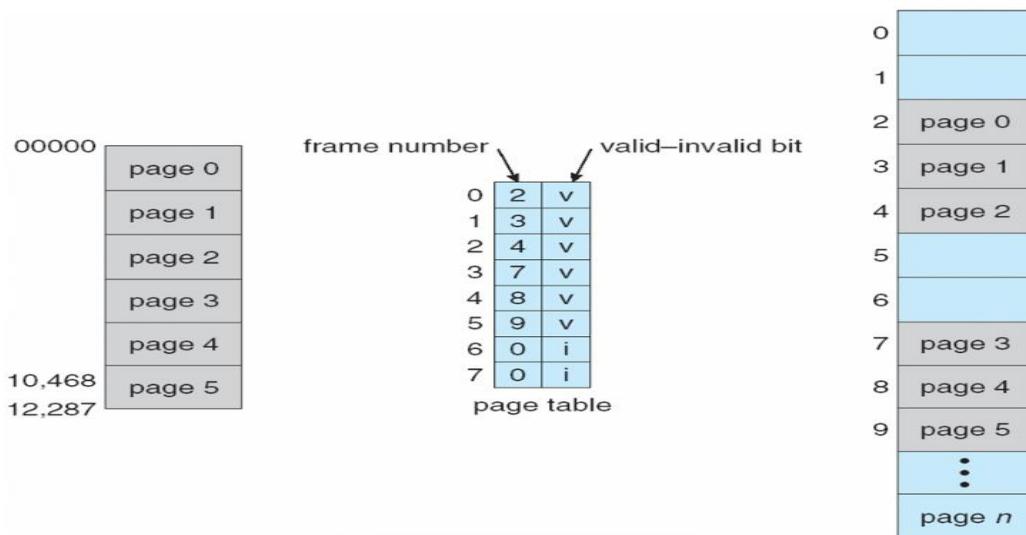
- Memory protection in a paged environment is accomplished by **protection bits** associated with each frame. Normally, these bits are kept in the page table. One bit can define a page to be read-write or read-only.
- Every reference to memory goes through the page table to find the correct frame number. At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page.
- An attempt to write to a read-only page causes a hardware trap to the operating system (or memory-protection violation).
- One additional bit is generally attached to each entry in the page table: a **valid-invalid** bit.

When this bit is set to ``valid'', the associated page is in the process's logical address space and is thus a legal (or valid) page.

When the bit is set to ``invalid'', the page is not in the process's logical address space.

- Illegal addresses are trapped by use of the valid-invalid bit. The OS sets this bit for each page to allow or disallow access to the page.
- Suppose, for example, that in a system with a 14-bit address space (0 to 16383), we have a program that should use only addresses 0 to 10468.
  - item Addresses in pages 0, 1, 2, 3, 4, and 5 are mapped normally through the page table.
  - Any attempt to generate an address in pages 6 or 7, however, will find that the valid-invalid bit is set to invalid, and the computer will trap to the OS (invalid page reference).

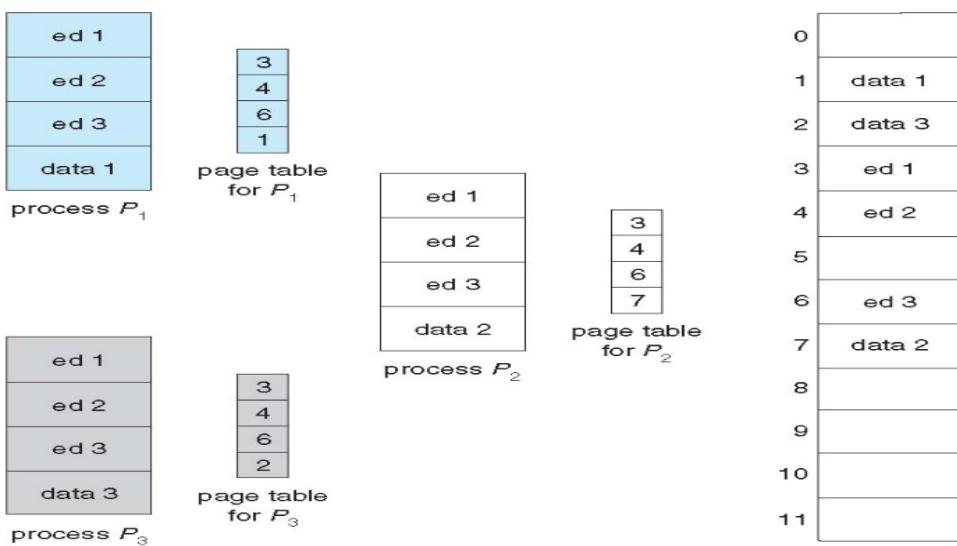
## Valid (v) or Invalid (i) Bit In A Page Table



### Shared Pages

An advantage of paging is the possibility of sharing common code. This consideration is particularly important in a time-sharing environment. Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users. If the code is reentrant code (or pure code however, it can be shared, as shown in Figure )

### Shared Pages Example



Here we see a three-page editor-each page 50 KB in size (the large page size is used to simplify the figure)-being shared among three processes. Each process has its own data page. Re-entrant code is non-self-modifying code: it never changes during execution. Thus, two or more processes can execute

the same code at the same time. Each process has its own copy of registers and data storage to hold the data for the process's execution. The data for two different processes will be of course, be different. Only one copy of the editor need be kept in physical memory.

Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames. Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user. The total space required is now 2)50 KB instead of 8,000 KB-a significant savings. Other heavily used programs can also be shared -compilers, window systems, run-time libraries, database systems, and so on. To be sharable, the code must be reentrant. The read-only nature of shared code should not be left to the correctness of the code; the operating system should enforce this property.

## Unit-IV

### Topic 4: Paging-II (Structure of Page Table)

Page tables can consume a significant amount of memory.

- For example: A 32-bit virtual address space using 4 kB pages.

$$\text{Page size} = 4 * 2^{10} (\text{4 KB}) = 2^2 * 2^{10} = 2^{12} \text{ bytes}$$

$$\text{Space for page numbers (Logical Address space -page size)} = 2^{32} - 2^{12} = 2^{20} \text{ bytes}$$

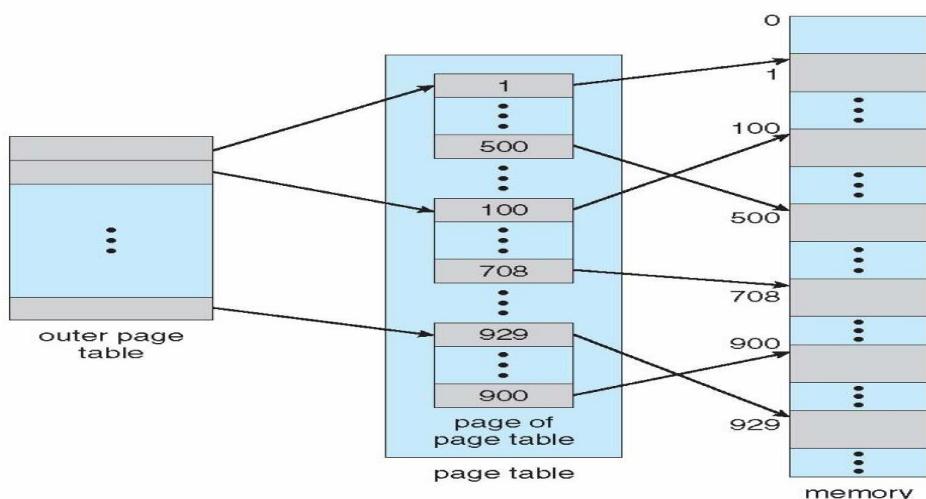
- So page table may consists of up to 1 million entries, which cannot be stored continuously in main memory. There are many techniques used for storing these page tables

***Some of the common techniques that are used for structuring the Page table are as follows:***

1. Hierarchical Paging
2. Hashed Page Tables
3. Inverted Page Tables

#### Hierarchical Paging

- Another name for Hierarchical Paging is multilevel paging.
- There might be a case where the page table is too big to fit in a contiguous space, so we may have a hierarchy with several levels.
- In this type of Paging the logical address space is broke up into Multiple page tables.
- Hierarchical Paging is one of the simplest techniques and for this purpose, a two-level page table and three-level page table can be used.



## Two Level Page Table

Consider a system having 32-bit logical address space and a page size of 1 KB and it is further divided into:

Page size =  $1 * 2^{10}$ (1 KB)= $2^0 * 2^{10} = 2^{10}$  bytes=10 bits for (displacement / Offset)

Total bits for Logical address=32

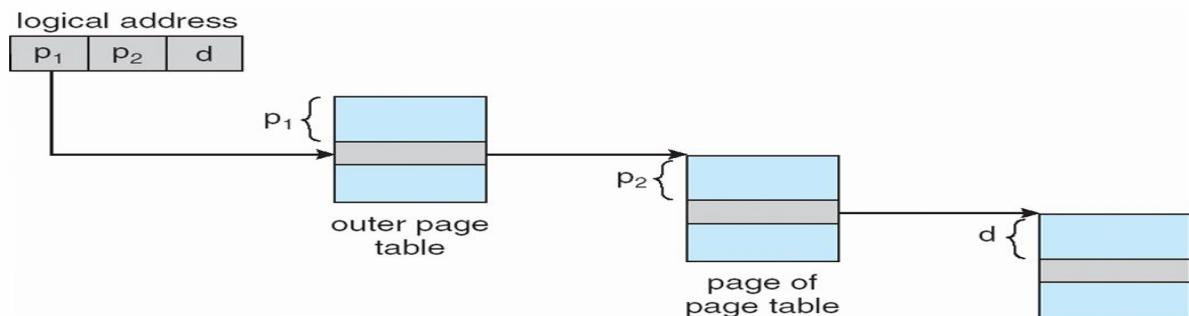
No.of bits for page number=bits for logical address-offset bits=(32-10)=22

22 is split up into two parts=12+10

page number		page offset
$p_1$	$p_2$	$d$
12	10	10

In the above diagram, P1 is an index into the **Outer Page** table. P2 indicates the displacement within the page of the **Inner page** Table. As address translation works from outer page table inward so is known as **forward-mapped Page Table**.

***Below given figure below shows the Address Translation scheme for a two-level page table***



## Three Level Page Table

For a system with 64-bit logical address space, a two-level paging scheme is not appropriate. Let us suppose that the page size, in this case, is 4KB. If in this case, we will use the two-page level scheme then the addresses will look like this:

Page size =  $4 * 2^{10}$ (4 KB)= $2^2 * 2^{10} = 2^{12}$  bytes=12 bits for (displacement / Offset)

Total bits for Logical address=64

No.of bits for page number=bits for logical address-offset bits=(64-12)=52

52 is split up into two parts=42+10

The outer page table consists of 242 entries, or 244 bytes. The obvious way to avoid such a large table is to divide the outer page table into smaller pieces. (This approach is also used on some 32-bit processors for added flexibility and efficiency.)

outer page	inner page	page offset
$p_1$	$p_2$	$d$
42	10	12

We can divide the outer page table in various ways. We can page the outer page table, giving us a three-level paging scheme. Suppose that the outer page table is made up of standard-size pages (210 entries, or 212 bytes). In this case, a 64-bit address space is still daunting:

2nd outer page	outer page	inner page	offset
$p_1$	$p_2$	$p_3$	$d$
32	10	10	12

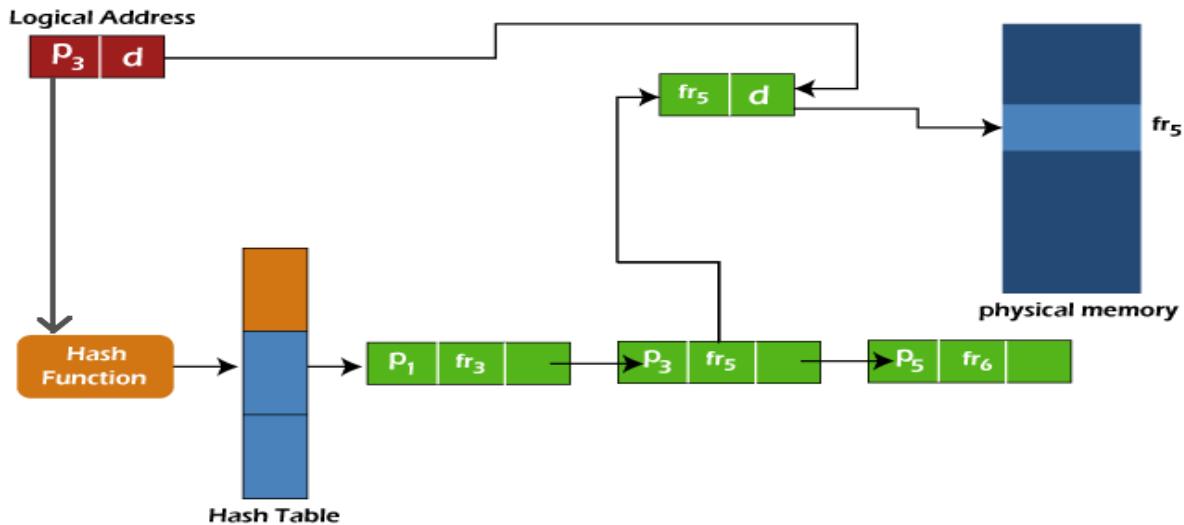
### ***Hashed Page Table***

***Hashed page tables*** are a technique for structuring page tables in memory. Hashed page tables are common in address spaces greater than 32 bits. In a hashed page table, the virtual addresses are hashed into the hash table. Each element in the table comprises a linked list of elements to avoid collisions. The hash value used is the virtual page number, i.e., all the bits that are not part of the page offset. For each element in the hash table, there are three fields available,

1. The virtual Page Number (which is the hash value).
2. The value of the mapped page frame.
3. A pointer to the next element in the linked list.

### ***Working of Hashed Page Table***

We would understand the working of the hashed page table with the help of an example. The CPU generates a logical address for the page it needs. Now, this logical address needs to be mapped to the physical address. This logical address has two entries, i.e., a page number ( $P_3$ ) and an offset, as shown below.



- The page number from the logical address is directed to the hash function.
- The hash function produces a hash value corresponding to the page number.
- This hash value directs to an entry in the hash table.
- As we have studied earlier, each entry in the hash table has a link list. Here the page number is compared with the first element's first entry. If a match is found, then the second entry is checked.

In this example, the logical address includes page number  $P_3$  which does not match the first element of the link list as it includes page number  $P_1$ . So we will move ahead and check the next element; now, this element has a page number entry, i.e.,  $P_3$ , so further, we will check the frame entry of the element, which is  $fr_5$ . We will append the offset provided in the logical address to this frame number to reach the page's physical address. So, this is how the hashed page table works to map the logical address to the physical address.

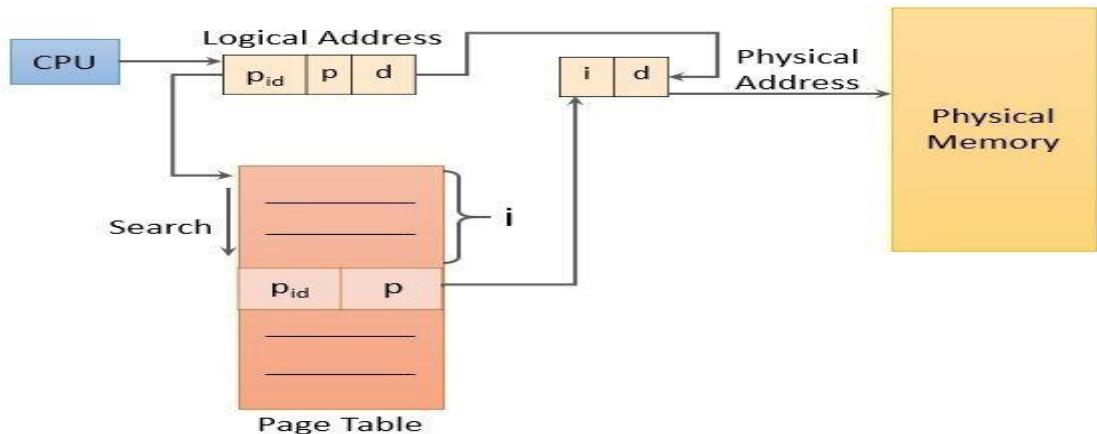
A variation of this scheme that is favorable for 64-bit address spaces has been proposed. This variation uses clustered page table which are similar to hashed page tables except that each entry in the hash table refers to several pages (such as 16) rather than a single page. Therefore, a single page-table entry can store the mappings for multiple physical-page frames. Clustered page tables are particularly useful for sparse address spaces, where memory references are noncontiguous and scattered throughout the address space.

### Inverted Page Table

The concept of normal paging says that every process maintains its own page table which includes the entries of all the pages belonging to the process. The large process may have a page table with millions of entries. Such a page table consumes a large amount of memory.

Consider we have six processes in execution. So, six processes will have some or the other of their page in the main memory which would compel their page tables also to be in the main memory consuming a lot of space. This is the drawback of the paging concept.

The inverted page table is the solution to this wastage of memory. The concept of an inverted page table involves the existence of single-page table which has entries of all the pages (may they belong to different processes) in main memory along with the information of the process to which they are associated. To get a better understanding consider the figure below of inverted page table.



### Structure of Inverted Page Table

The CPU generates the logical address for the page it needs to access. This time the logical address consists of three entries process id, page number and the offset. The process id identifies the process, of which the page has been demanded, page number indicates which page of the process has been asked for and the offset value indicates the displacement required.

The match of process id along with associated page number is searched in the page table and say if the search is found at the  $i^{th}$  entry of page table then  $i$  and offset together generates the physical address for the requested page. This is how the logical address is mapped to a physical address using the inverted page table.

Though the inverted page table reduces the wastage of memory but it increases the search time. This is because the entries in an inverted page table are sorted on the basis of physical address whereas the lookup is performed using logical address. It happens sometimes that the entire table is searched to find the match.

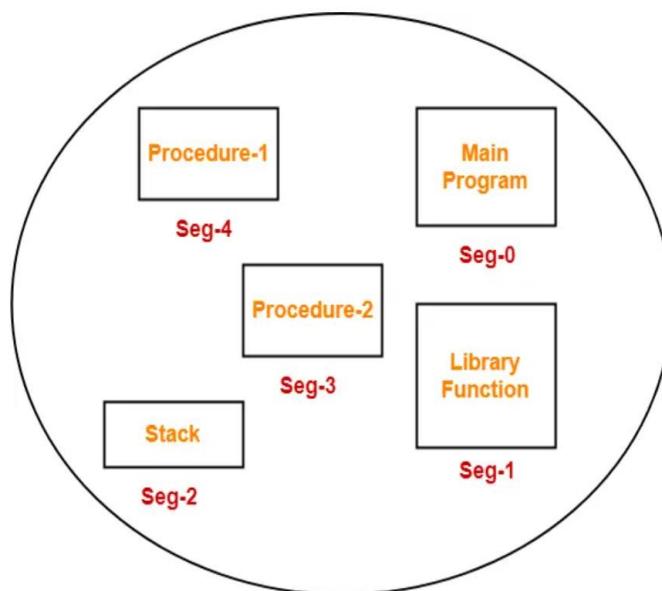
## Unit-IV

### Topic 5: Segmentation

Like Paging, Segmentation is another non-contiguous memory allocation technique. In Paging, the process was divided into equal-sized pages irrespective of the fact that what is inside the pages. It also divides some relative parts of a process into different pages which should be loaded on the same page. It decreases the efficiency of the system and doesn't give the user's view of a process. In Segmentation, similar modules are loaded in the same segments. It gives the user's view of a process and also increases the efficiency of the system as compared to Paging.

#### Basic Method

A Program is basically a collection of segments. And a segment is a logical unit such as: main program, procedure, function, method, object, local variable and global variables, symbol table, common block, stack and arrays. A computer system that is using segmentation has a logical address space that can be viewed as multiple segments.



And the size of the segment is of the variable that it may grow or shrink. As we had already told you that during the execution each segment has a name and length. And the address mainly specifies both thing name of the segment and the displacement within the segment. Therefore the user specifies each address with the help of two quantities: segment name and offset. For simplified Implementation segments are numbered; thus referred to as segment number rather than segment name. Thus the logical address consists of two tuples: <segment-number, offset> where,

### **Segment Number(s):**

Segment Number is used to represent the number of bits that are required to represent the segment.

### **Offset(d)**

Segment offset is used to represent the number of bits that are required to represent the size of the segment.

## **Hardware**

The details about each segment are stored in a table called a segment table. Segment table is stored in one (or many) of the segments. In the segment table each entry has :

### **Segment Base/base address:**

The segment base mainly contains the starting physical address where the segments reside in the memory.

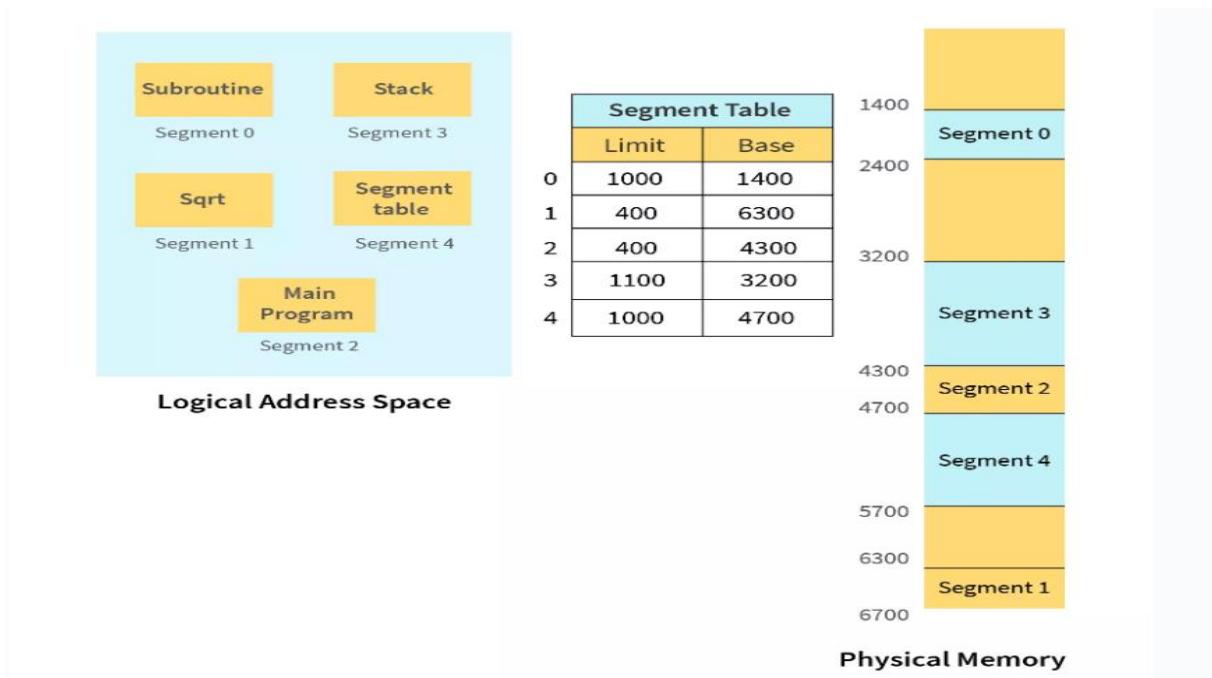
### **Segment Limit:**

The segment limit is mainly used to specify the length of the segment.

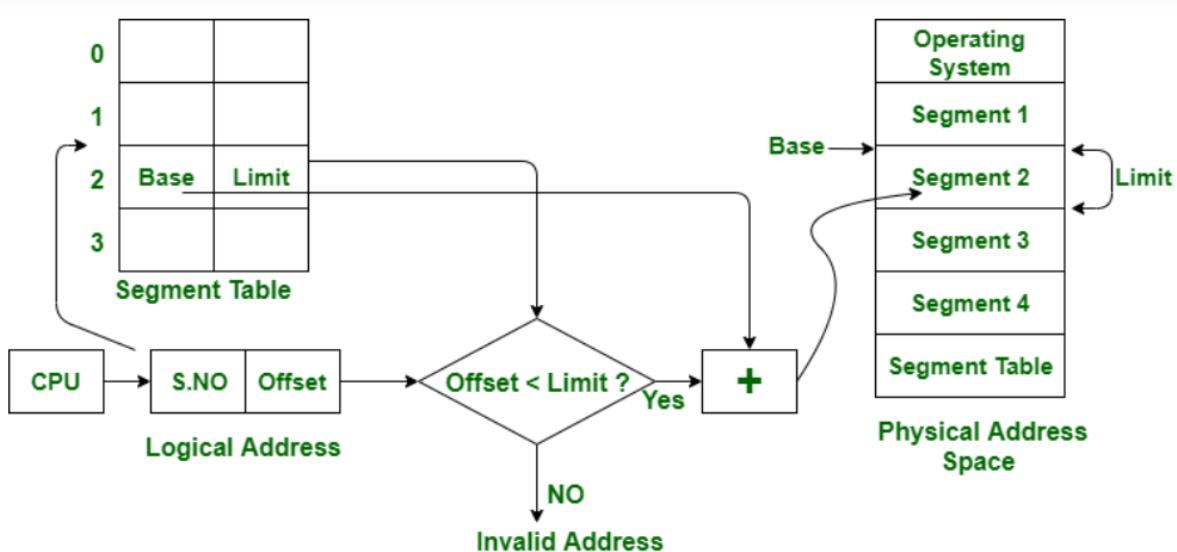
### ***Let's take the example of segmentation to understand how it works.***

Let us assume we have five segments namely: Segment-0, Segment-1, Segment-2, Segment-3, and Segment-4. Initially, before the execution of the process, all the segments of the process are stored in the physical memory space. We have a segment table as well. The segment table contains the beginning entry address of each segment (denoted by **base**). The segment table also contains the length of each of the segments (denoted by **limit**).

As shown in the image below, the base address of Segment-0 is 1400 and its length is 1000, the base address of Segment-1 is 6300 and its length is 400, the base address of Segment-2 is 4300 and its length is 400, and so on. The pictorial representation of the above segmentation with its segment table is shown below.



With the help of segment table and hardware assistance, the operating system can easily translate a logical address into physical address on execution of a program. The **Segment number** is mapped to the segment table. The limit of the respective segment is compared with the offset. If the offset is less than the limit then the address is valid otherwise it throws an error as the address is invalid. In the case of valid addresses, the base address of the segment is added to the offset to get the physical address of the actual word in the main memory.



### **Advantages of Segmentation**

1. No internal fragmentation
2. Average Segment Size is larger than the actual page size.
3. Less overhead
4. It is easier to relocate segments than entire address space.
5. The segment table is of lesser size as compared to the page table in paging.

### **Disadvantages**

1. It can have external fragmentation.
2. it is difficult to allocate contiguous memory to variable sized partition.
3. Costly memory management algorithms.

### **Paging Vs Segmentation**

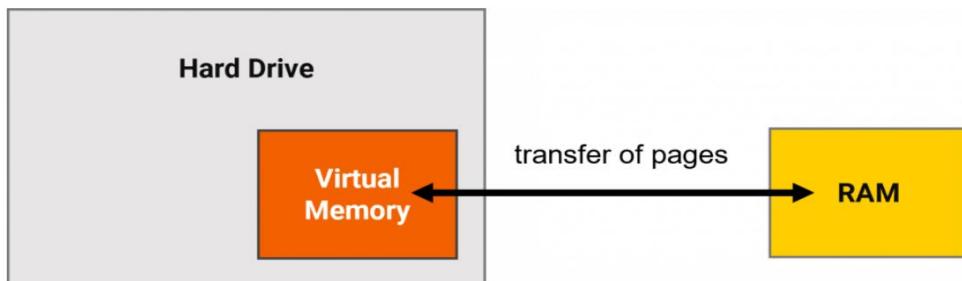
<b>Paging</b>	<b>Segmentation</b>
A page is a physical unit of information.	A segment is a logical unit of information.
Frames on main memory are required	No frames are required
The page is of the fixed block size	The page is of the variable block size
It leads to internal fragmentation	It leads to external fragmentation
The page size is decided by hardware in paging	Segment size is decided by the user in segmentation
It does not allow logical partitioning and protection of application components	It allows logical partitioning and protection of application components
Paging involves a page table that contains the base address of each page	Segmentation involves the segment table that contains the segment number and offset

## Unit-IV

### Topic 6 : Virtual Memory

Suppose you have a furniture shop that is not big enough to accommodate all the furniture, then you will keep it somewhere else, like in a warehouse. And you will keep that furniture in your shop which will be in demand. And if you don't require some particular furniture, you will keep that in the warehouse. And according to trend or demand, you will keep exchanging the furniture items from the warehouse. The same concept is for virtual memory.(warehouse is main memory and furniture's are process).

Virtual Memory is a storage scheme that provides user an illusion of having a very big main memory. This is done by treating a part of secondary memory as the main memory. In this scheme, User can load the bigger size processes than the available main memory by having the illusion that the memory is available to load the process.



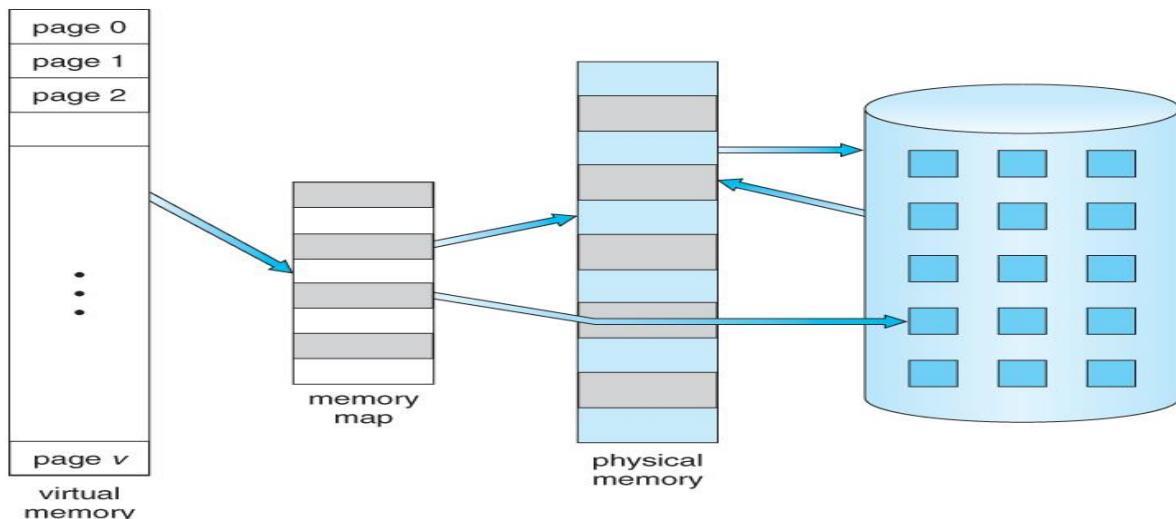
Instead of loading one big process in the main memory, the Operating System loads the different parts of more than one process in the main memory. By doing this, the degree of multiprogramming will be increased and therefore, the CPU utilization will also be increased.

***In practice, most real processes do not need all their pages, or at least not all at once, for several reasons:***

- Error handling code is not needed unless that specific error occurs, some of which are quite rare.
- Arrays are often over-sized for worst-case scenarios, and only a small fraction of the arrays are actually used in practice.

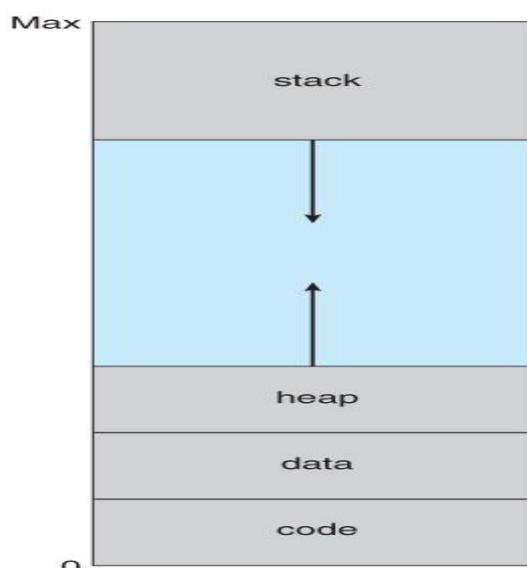
***The ability to load only the portions of processes that were actually needed ( and only when they were needed ) has several benefits:***

- Programs could be written for a much larger address space ( virtual memory space ) than physically exists on the computer.
- Because each process is only using a fraction of their total address space, there is more memory left for other programs, improving CPU utilization and system throughput.
- Less I/O is needed for swapping processes in and out of RAM, speeding things up.



**Diagram showing virtual memory that is larger than physical memory**

- Figure below shows **virtual address space**, which is the programmers logical view of process memory storage. The actual physical layout is controlled by the process's page table.



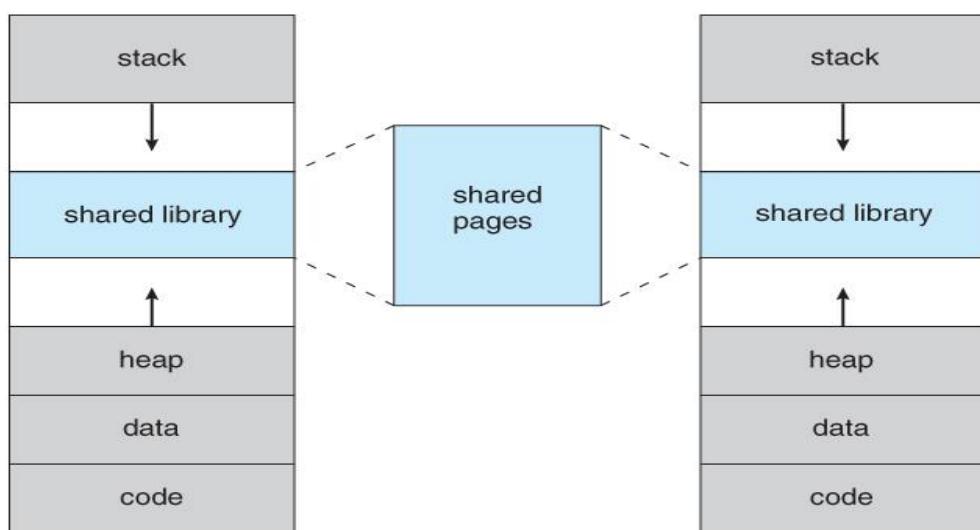
Note that the address space shown in Figure is **sparse** - A great hole in the middle of the address space is never used, unless the stack and/or the heap grow to fill the hole.

**Virtual memory also allows the sharing of files and memory by multiple processes, with several benefits:**

System libraries can be shared by mapping them into the virtual address space of more than one process.

Processes can also share virtual memory by mapping the same block of memory to more than one process.

Process pages can be shared during a fork( ) system call, eliminating the need to copy all of the pages of the original ( parent ) process.



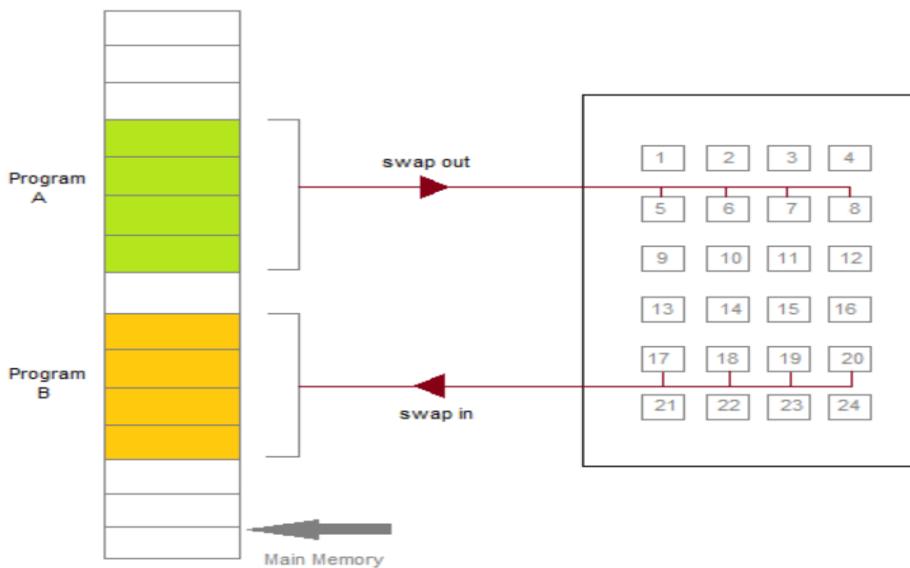
**Figure : Shared library using virtual memory**

### Demand Paging

- According to the concept of Virtual Memory, in order to execute some process, only a part of the process needs to be present in the main memory which means that only a few pages will only be present in the main memory at any time.
- However, deciding, which pages need to be kept in the main memory and which need to be kept in the secondary memory, is going to be difficult because we cannot say in advance that a process will require a particular page at particular time.
- Therefore, to overcome this problem, there is a concept called Demand Paging is introduced. It suggests keeping all pages of the frames in the secondary memory until they are required. In other words, it says that do not load any page in the main memory until it is required. Whenever any page

is referred for the first time in the main memory, then that page will be found in the secondary memory.

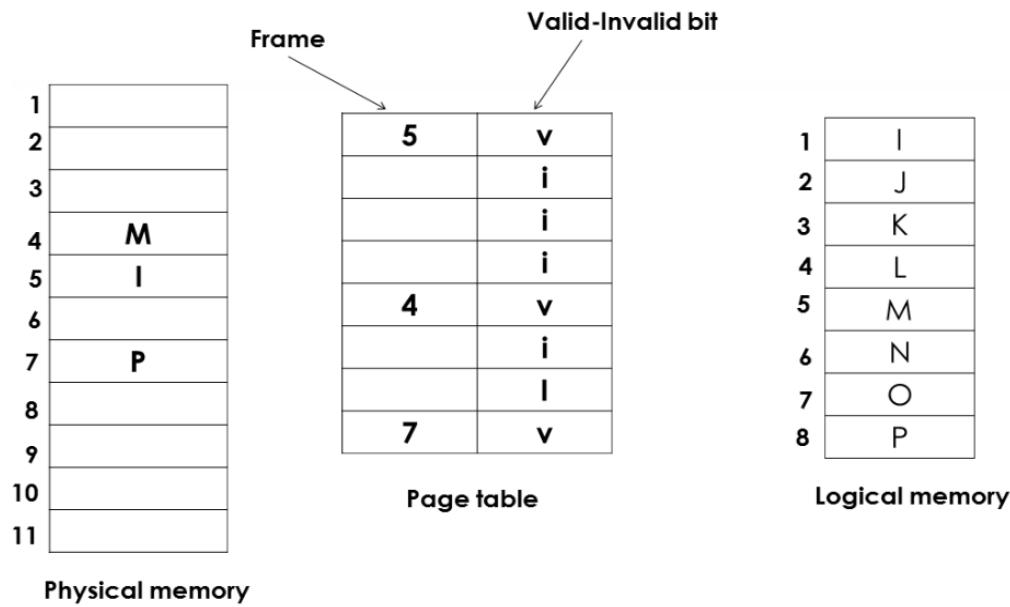
- The demand paging system is somehow similar to the paging system with swapping where processes mainly reside in the main memory(usually in the hard disk). Thus demand paging is the process that solves the above problem only by swapping the pages on Demand.
- This is also known as **lazy swapper**( It never swaps the page into the memory unless it is needed).Swapper that deals with the individual pages of a process are referred to as **Pager**.



Some form of hardware support is used to distinguish between the pages that are in the memory and the pages that are on the disk. Thus for this purpose Valid-Invalid scheme is used:

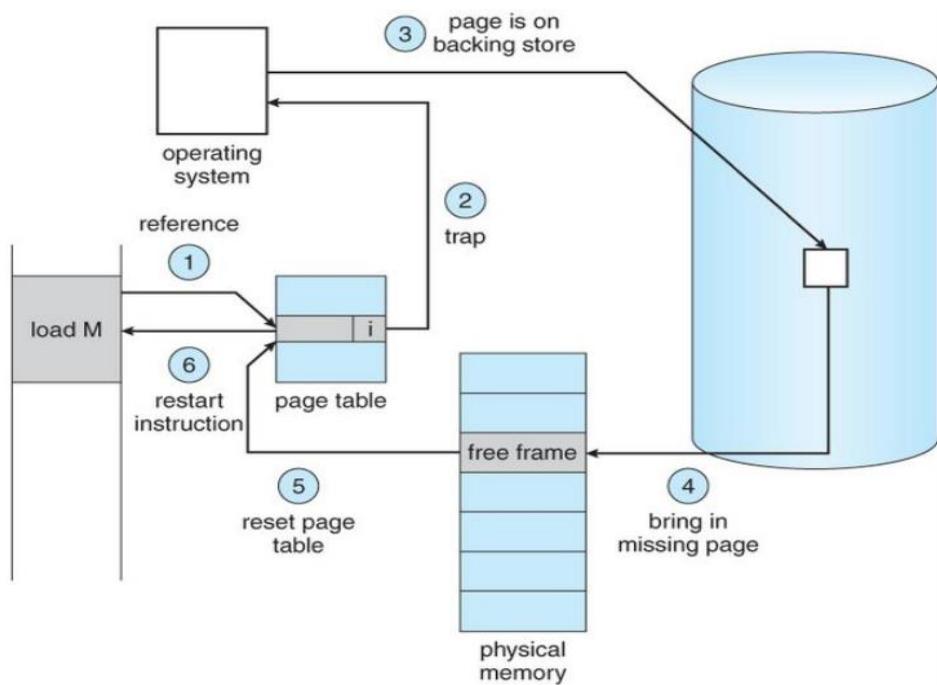
- With each page table entry, a valid-invalid bit is associated( where **1** indicates **in the memory** and **0** indicates **not in the memory**)
  - Initially, the valid-invalid bit is set to 0 for all table entries.
- If the bit is set to "**valid**", then the associated page is **both legal and is in memory**.
  - If the bit is set to "**invalid**" then it indicates that the **page is either not valid or the page is valid but is currently not on the disk**.
- For the **pages** that are **brought into the memory**, the **page table** is set as usual.
  - But for the **pages** that are **not currently in the memory**, the **page table** is either **simply marked as invalid** or it contains the **address of the page on the disk**.

During the translation of address, if the valid-invalid bit in the page table entry is 0 then it leads to **page fault**.



**The above figure is to indicates the page table when some pages are not in the main memory.**

If the referred page is not present in the main memory then there will be a miss and the concept is called Page miss or page fault. Let us understand the procedure to handle the page fault as shown with the help of the below diagram:



1. First of all, internal table(that is usually the process control block) for this process in order to determine whether the reference was valid or invalid memory access.
2. If the reference is invalid, then we will terminate the process. If the reference is valid, but we have not bought in that page so now we just page it in.
3. Then we **locate the free frame list** in order to find the free frame.
4. Now a disk operation is scheduled in order to read the **desired page into the newly allocated frame**.
5. When the disk is completely read, then the **internal table is modified** that is kept with the process, and the page table that mainly indicates the page is now in memory.
6. Now we will restart the instruction that was interrupted due to the trap. Now the process can access the page as though it had always been in memory.

*In some cases when initially no pages are loaded into the memory, pages in such cases are only loaded when are demanded by the process by generating page faults. It is then referred to as Pure Demand Paging.*

### Performance of Demand Paging

To understand the impact of demand paging on system performance, we compute the effective access time (EAT) for memory. Let:

- $ma$  be the memory access time (10 to 200 ns).
- $p$  be the probability of a page fault ( $0 \leq p \leq 1$ ).

When no page faults occur, EAT is simply  $ma$ . If a page fault occurs, additional time is required to handle it.

Let  $p$  be the probability of a page fault ( $0 \leq p \leq 1$ ). We would expect  $p$  to be close to zero—that is, we would expect to have only a few page faults. Then  $EAT = (1-p) \times ma + p \times \text{page fault time}$

To compute the effective access time, we must know how much time is needed to service a page fault.

A page fault causes the following sequence to occur:

1. Trap to the operating system.
2. Save the user registers and process state.
3. Determine that the interrupt was a page fault.
4. Check that the page reference was legal and determine the location of the page on the disk Issue a read from the disk to a free frame:
  - a. Wait in a queue for this device until the read request is serviced.

- b. Wait for the device seek and/ or latency time.
- c. Begin the transfer of the page to a free frame.
- 5. While waiting, allocate the CPU to some other user (CPU scheduling, optional).
- 6. Receive an interrupt from the disk I/O subsystem (I/O completed).
- 7. Save the registers and process state for the other user (if step 6 is executed).
- 8. Determine that the interrupt was from the disk
- 9. Correct the page table and other tables to show that the desired page is now in memory.
- 10. Wait for the CPU to be allocated to this process again.
- 11. Restore the user registers, process state, and new page table, and then resume the interrupted instruction.

Not all of these steps are necessary in every case. In any case, we are faced with three major components of the page-fault service time

Service the page-fault interrupt.

Read in the page.

Restart the process.

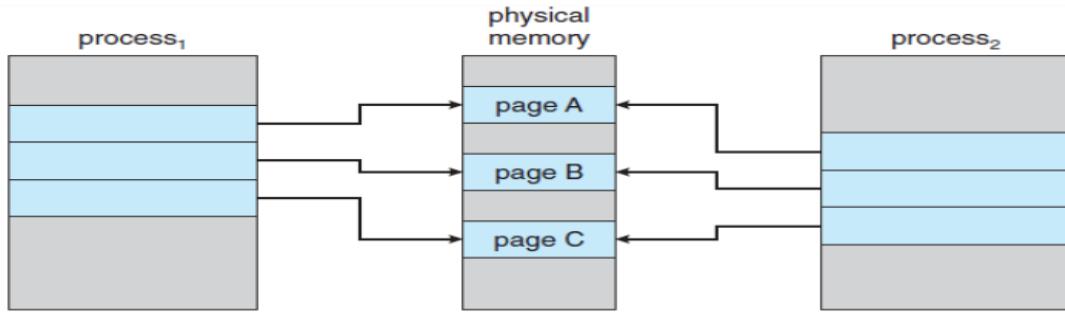
An additional aspect of demand paging is the handling and overall use of swap space. Swap Space Optimization can be done by

- Fast Swap Space: Allocated in large blocks, faster than file system.
- File System Paging: Initially demand pages from the file system; subsequently use swap space for better throughput.
- Hybrid Approach: Pages binary files from the file system, overwrites when replaced, and uses swap space for stack/heap. Used in Solaris and BSD UNIX.

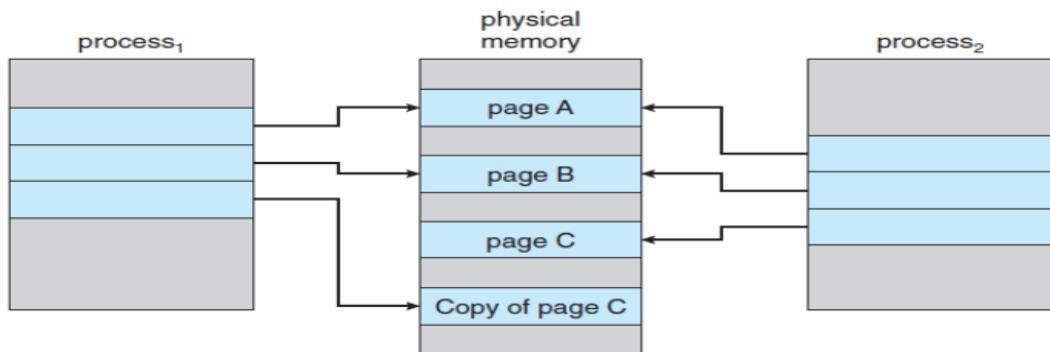
### **Copy-on-Write**

Copy-on-Write Recall that the fork( ) system call creates a child process that is a duplicate of its parent. Traditionally, fork( ) worked by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent. However, considering that many child processes invoke the exec() system call immediately after creation, the copying of the parent's address space may be unnecessary. Instead, we can use a technique known as copy-on-write, which works by allowing the parent and child processes initially to share the same pages. These shared pages are marked as copy-on-write pages, meaning that if either process writes to a shared page, a copy of the shared page is created. Copy-on-

write is illustrated in Figures below, which show the contents of the physical memory before and after process 1 modifies page C



**Before process 1 modifies page C.**



**After process 1 modifies page C.**

When it is determined that a page is going to be duplicated using copy on-write, it is important to note the location from which the free page will be allocated. Many operating systems provide a pool of free pages for such requests. These free pages are typically allocated when the stack or heap for a process must expand or when there are copy-on-write pages to be managed. Operating systems typically allocate these pages using a technique known as Zero-fill-on-demand. Zero-fill-on-demand pages have been zeroed-out before being allocated, thus erasing the previous contents.

## Unit-IV

### Topic 7: Page Replacement Algorithms

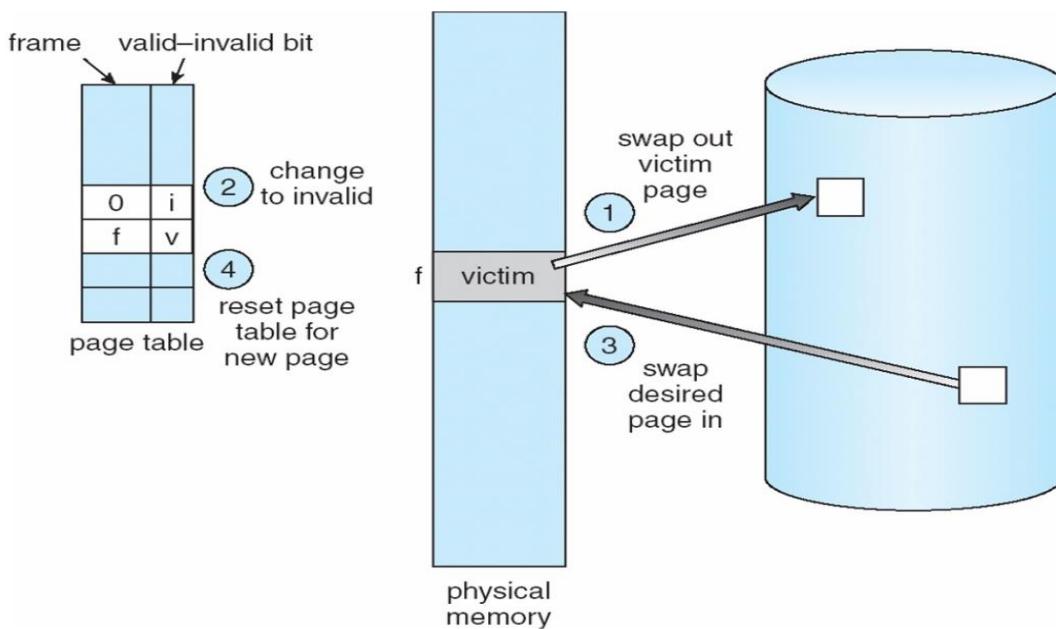
#### Need for Page Replacement

As studied in Demand Paging, only certain pages of a process are loaded initially into the memory. This allows us to get more processes into memory at the same time. but what happens when a process requests for more pages and no free memory is available to bring them in then the Operating system needs to decide which page to replace. The operating system must use any page replacement algorithm in order to select the victim frame.

#### Basic Page Replacement

If there is no free frame follow the steps as depicted in the diagram

1. Use a page replacement algorithm to select a victim frame. Write the victim page to the disk.
2. change the page & frame tables accordingly.
3. Read the desired page into the free frame.
4. Update the page and frame tables and restart the process



Notice that, if no frames are free, two page transfers (one out and one in) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly. We can reduce this overhead by using a modify or a dirty bit. Each page or frame has a modify bit associated with it in the hardware. It indicates that any word or byte in the page is modified.

***When we select a page for replacement, we examine its modify bit.***

- If the bit is set, we know that the page has been modified & in this case we must write that page to the disk.
- If the bit is not set, then if the copy of the page on the disk has not been overwritten, then we can avoid writing the memory page on the disk as it is already there.

To implement demand paging, two problems should be solved:

- Frame-allocation algorithm
- Page replacement algorithm

When multiple processes are in main memory, the operating system should decide how many frames should be allocated to each process. Suppose there are 10 processes and there are 50 frames, a decision has to be made as to how many frames should be given to each process. This is called frame allocation.

Whenever page replacement is needed, the operating system should decide which page should be selected for replacement. This is done by a page replacement algorithm.

## **Page Replacement Algorithms**

A page replacement algorithm determines how the victim page (the page to be replaced) is selected when a page fault occurs. Page replacement algorithms in an operating system require specific inputs to function correctly. These inputs allow the algorithm to make decisions about which page to replace when a page fault occurs. Here are the primary inputs needed:

**Reference String:** A sequence of memory addresses or page numbers that represent the order in which pages are accessed by a process.

**Number of Frames:** The number of available frames in the physical memory.

In the context of page replacement algorithms and memory management in operating systems, "page hit" and "page miss" (or page fault) are key concepts that describe the success or failure of accessing a page in memory.

**Page Hit:** A page hit occurs when the page requested by the process is already present in physical memory. No replacement is needed

**Page Miss (Page Fault):** A page miss, also known as a page fault, occurs when the page requested by the process is not present in physical memory. The operating system must load the requested page from secondary storage (like a hard disk or SSD) into physical memory.

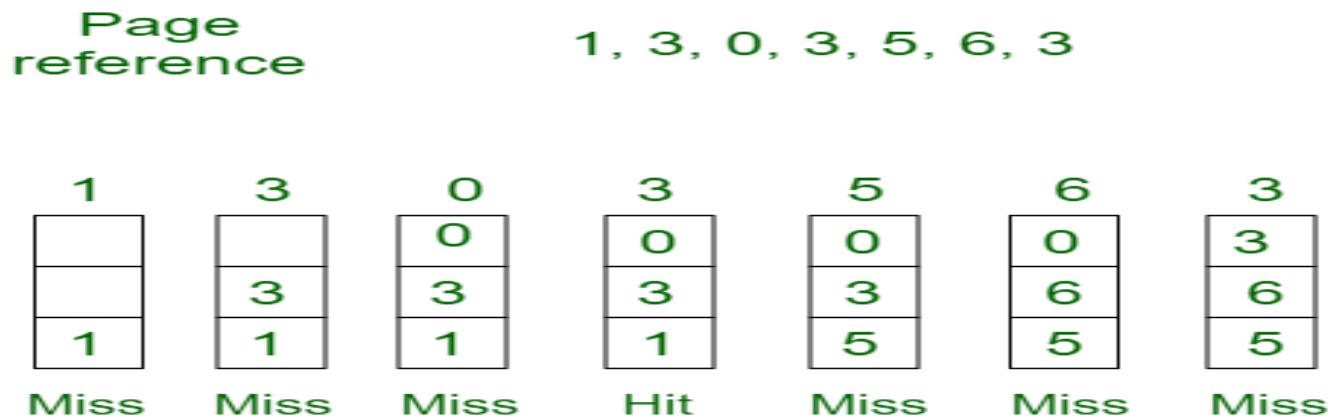
***The aim of any replacement algorithm is to minimize the page fault rate.***

## Page Replacement Algorithms

1. FIFO Page Replacement
2. Optimal Page Replacement
3. LRU Page Replacement

**1. First In First Out (FIFO):** This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

**Example 1:** Consider page reference string 1, 3, 0, 3, 5, 6, 3 with 3-page frames. Find the number of page faults.



**Total Page Fault = 6**

Initially, all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots → **3 Page Faults**.

when 3 comes, it is already in memory so → **0 Page Faults**. Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e 1. → **1 Page Fault**. 6 comes, it is also not available in memory so it replaces the oldest page slot i.e 3 → **1 Page Fault**. Finally, when 3 come it is not available so it replaces **0 1 page fault**.

**Belady's anomaly** proves that it is possible to have more page faults when increasing the number of page frames while using the First in First Out (FIFO) page replacement algorithm. For example, if we consider reference strings 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4, and 3 slots, we get 9 total page faults, but if we increase slots to 4, we get 10-page faults.

2. **Optimal Page replacement:** In this algorithm, pages are replaced which would not be used for the longest duration of time in the future.

- **Example-2:** Consider the page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 with 4 page frame. Find number of page fault.

Page reference    7,0,1,2,0,3,0,4,2,3,0,3,2,3                  No. of Page frame - 4

7	0	1	2	0	3	0	4	2	3	0	3	2	3
Miss	Miss	Miss	Miss	Hit	Miss	Hit	Miss	Hit	Hit	Hit	Hit	Hit	Hit
7	7	7	7	7	3	3	3	3	3	3	3	3	3

Total Page Fault = 6

Initially, all slots are empty, so when 7 0 1 2 are allocated to the empty slots → **4 Page faults**  
 0 is already there so → **0 Page fault**. when 3 came it will take the place of 7 because it is not used for the longest duration of time in the future.—>**1 Page fault**. 0 is already there so → **0 Page fault**. 4 will takes place of 1 → **1 Page Fault**.

Now for the further page reference string → **0 Page fault** because they are already available in the memory.

Optimal page replacement is perfect, but not possible in practice as the operating system cannot know future requests. The use of Optimal Page replacement is to set up a benchmark so that other replacement algorithms can be analyzed against it.

3. **LRU:** The LRU stands for the **Least Recently Used**. It keeps track of page usage in the memory over a short period of time. It works on the concept that pages that have been highly used in the past are likely to be significantly used again in the future. It removes the page that has not been utilized in the memory for the longest time. LRU is the most widely used algorithm because it provides fewer page faults than the other methods.

- **Example-3:** Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 with 4 page frames. Find number of page faults.

Page reference	7,0,1,2,0,3,0,4,2,3,0,3,2,3	No. of Page frame - 4												
7	0	1	2	0	3	0	4	2	3	0	3	2	3	
7	7	7	7	7	3	3	3	3	3	3	3	3	3	
Miss	Miss	Miss	Miss	Hit	Miss	Hit	Miss	Hit	Hit	Hit	Hit	Hit	Hit	
Total Page Fault = 6														

Here LRU has same number of page fault as optimal but it may differ according to question.

- Initially, all slots are empty, so when 7 0 1 2 are allocated to the empty slots → **4 Page faults**

0 is already there so → **0 Page fault**. when 3 came it will take the place of 7 because it is least recently used → **1 Page fault**

0 is already in memory so → **0 Page fault**.

4 will take place of 1 → **1 Page Fault**

Now for the further page reference string → **0 Page fault** because they are already available in the memory.

Implementing the Least Recently Used (LRU) page replacement algorithm can be done using either a counter-based approach or a stack-based approach.

**LRU Counter-Based Approach:** In this method, each page table entry is associated with a counter that keeps track of the last time the page was accessed.

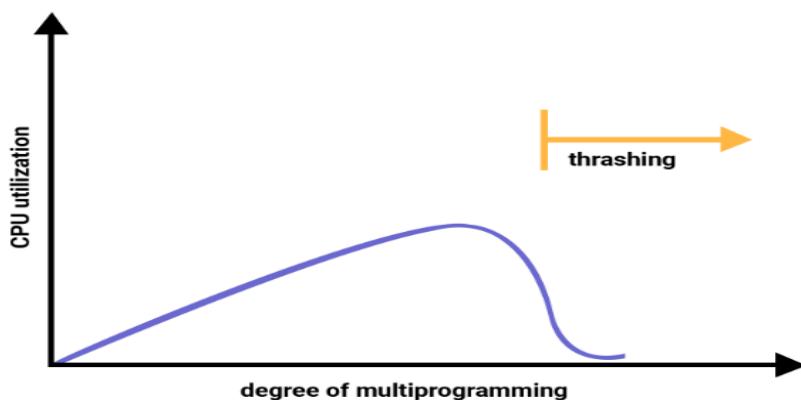
- Let a global counter to zero.
- Each page has a corresponding counter initialized to zero. Each time a page is accessed, increment the global counter and update the page's counter with the current value of the global counter.
- When a page fault occurs and a page needs to be replaced, choose the page with the smallest counter value (indicating it was used the longest time ago).

**LRU Stack-Based Approach:** In this method, a stack (or a doubly linked list) is used to maintain the order of page accesses. The most recently accessed page is moved to the top of the stack, and the least recently accessed page is at the bottom.

- Start with an empty stack.
- Each time a page is accessed, if it is already in the stack, move it to the top. If it is not in the stack and the stack is full, remove the bottom page and add the new page to the top.
- The page at the bottom of the stack is the least recently used and should be replaced when necessary

### Thrashing

Thrashing in operating systems refers to a situation where the system spends a significant amount of time swapping pages in and out of memory, rather than executing application code. This typically occurs when the system is under heavy load and the working set of all processes cannot fit into the physical memory available.



### Causes of Thrashing

1. **Insufficient Memory:** When the system's RAM is too small to handle the current load, it relies heavily on virtual memory, causing frequent page swaps.
2. **High Multiprogramming Level:** Running too many processes simultaneously can lead to an excessive demand for memory.
3. **Poor Locality of Reference:** If the processes do not exhibit good locality of reference, meaning they frequently access pages scattered throughout memory, it can increase the rate of page faults.
4. **Over-committing Memory:** Allocating more memory to processes than the physical memory available, relying on virtual memory to compensate.

## **Effect of Thrashing**

At the time, when thrashing starts then the operating system tries to apply either the **Global page replacement** Algorithm or the **Local page replacement** algorithm.

### **Global Page Replacement**

The Global Page replacement has access to bring any page, whenever thrashing found it tries to bring more pages. Actually, due to this, no process can get enough frames and as a result, the thrashing will increase more and more. Thus the global page replacement algorithm is not suitable whenever thrashing happens.

### **Local Page Replacement**

Unlike the Global Page replacement, the local page replacement will select pages which only belongs to that process. Due to this, there is a chance of a reduction in the thrashing. As it is also proved that there are many disadvantages of Local Page replacement. Thus local page replacement is simply an alternative to Global Page replacement.

### **Techniques used to handle the thrashing**

As we have already told you the Local Page replacement is better than the Global Page replacement but local page replacement has many disadvantages too, so it is not suggestible. Thus, given below are some other techniques that are used:

#### **1.Working-Set Model**

✓ This model is based on the above-stated concept of the Locality Model. The basic principle states that if we allocate enough frames to a process to accommodate its current locality, it will only fault whenever it moves to some new locality. But if the allocated frames are lesser than the size of the current locality, the process is bound to thrash. According to this model, based on parameter A, the working set is defined as the set of pages in the most recent 'A' page references. Hence, all the actively used pages would always end up being a part of the working set.

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

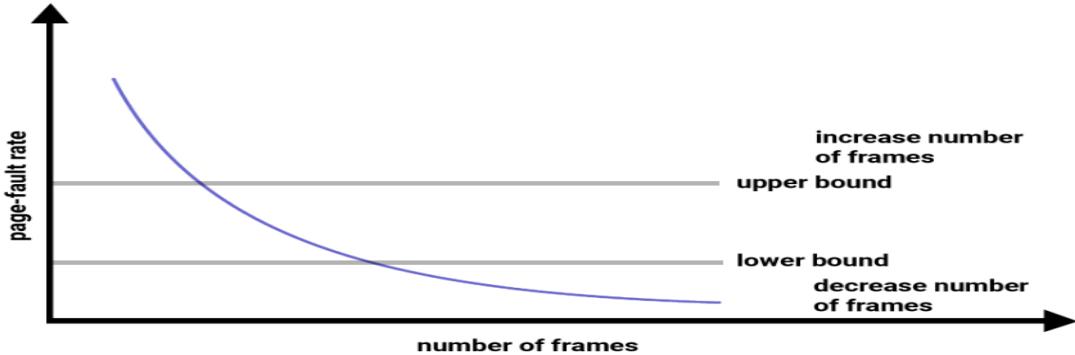
If D is the total demand for frames and  $WSS_i$  is the working set size for process i,

$$D = \sum WSS_i$$

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

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

## 2. Page Fault Frequency



*Figure: Page fault frequency*

A more direct approach to handle thrashing is the one that uses the Page-Fault Frequency concept.

When the Page fault is too high, then we know that the process needs more frames. Conversely, if the page fault-rate is too low then the process may have too many frames.

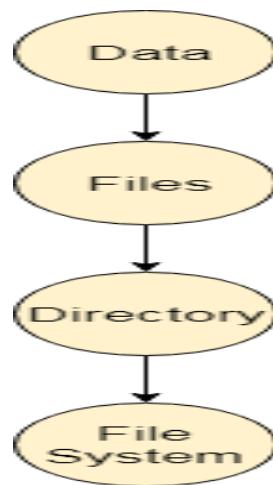
We can establish upper and lower bounds on the desired page faults. If the actual page-fault rate exceeds the upper limit then we will allocate the process to another frame. And if the page fault rate falls below the lower limit then we can remove the frame from the process.

Thus with this, we can directly measure and control the page fault rate in order to prevent thrashing.

## Unit-IV

### **Topic 8: File System Interface and Operations**

A file can be defined as a data structure which stores the sequence of records. Files are stored in a file system, which may exist on a disk or in the main memory. Files can be simple (plain text) or complex (specially-formatted). The collection of files is known as Directory. The collection of directories at the different levels, is known as File System.



The information in a file is defined by its creator. Many different types of information may be stored in a file-source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on. A file has a certain defined which depends on its type. A text file is a sequence of characters organized into lines (and possibly pages). A source file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements. An object file is a sequence of bytes organized into blocks understandable by the system's linker. An executable file is a series of code sections that the loader can bring into memory and execute.

#### **Attributes of the File**

##### **1.Name**

Every file carries a name by which the file is recognized in the file system. One directory cannot have two files with the same name.

##### **2.Identifier**

Along with the name, Each File has its own extension which identifies the type of the file. For example, a text file has the extension **.txt**, A video file can have the extension **.mp4**.

##### **3.Type**

In a File System, the Files are classified in different types such as video files, audio files, text files, executable files, etc.

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

#### 4.Location

In the File System, there are several locations on which, the files can be stored. Each file carries its location as its attribute.

#### 5.Size

The Size of the File is one of its most important attribute. By size of the file, we mean the number of bytes acquired by the file in the memory.

#### 6.Protection

The admin of the computer may want the different protections for the different files. Therefore, each file carries its own set of permissions to the different group of Users.

#### 7.Time and Date

Every file carries a time stamp which contains the time and date on which the file is last modified

### Operations on the File

A file is a collection of logically related data that is recorded on the secondary storage in the form of sequence of operations. The content of the files are defined by its creator who is creating the file. The various operations which can be implemented on a file such as read, write, open and close etc. are called

file operations. These operations are performed by the user by using the commands provided by the operating system. Some common operations are as follows:

**1. Create operation:**

This operation is used to create a file in the file system. It is the most widely used operation performed on the file system. To create a new file of a particular type the associated application program calls the file system. This file system allocates space to the file. As the file system knows the format of directory structure, so entry of this new file is made into the appropriate directory.

**2. Open operation:**

This operation is the common operation performed on the file. Once the file is created, it must be opened before performing the file processing operations. When the user wants to open a file, it provides a file name to open the particular file in the file system. It tells the operating system to invoke the open system call and passes the file name to the file system.

**3. Write operation:**

This operation is used to write the information into a file. A system call write is issued that specifies the name of the file and the length of the data has to be written to the file. Whenever the file length is increased by specified value and the file pointer is repositioned after the last byte written.

**4. Read operation:**

This operation reads the contents from a file. A Read pointer is maintained by the OS, pointing to the position up to which the data has been read.

**5. Re-position or Seek operation:**

The seek system call re-positions the file pointers from the current position to a specific place in the file i.e. forward or backward depending upon the user's requirement. This operation is generally performed with those file management systems that support direct access files.

**6. Delete operation:**

Deleting the file will not only delete all the data stored inside the file it is also used so that disk space occupied by it is freed. In order to delete the specified file the directory is searched. When the directory entry is located, all the associated file space and the directory entry is released.

**7. Truncate operation:**

Truncating is simply deleting the file except deleting attributes. The file is not completely deleted although the information stored inside the file gets replaced.

**8. Close operation:**

When the processing of the file is complete, it should be closed so that all the changes made permanent and all the resources occupied should be released. On closing it deallocates all the internal descriptors that were created when the file was opened.

### **9. Append operation:**

This operation adds data to the end of the file.

### **10. Rename operation:**

This operation is used to rename the existing file.

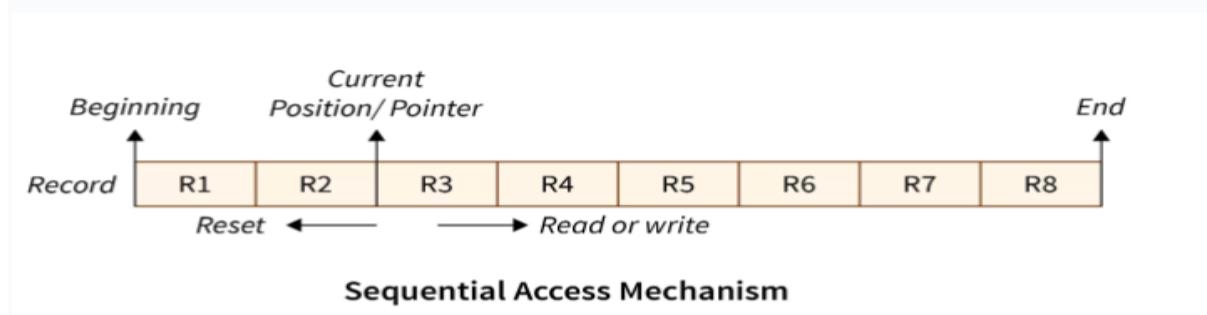
## **File Access Method**

File access methods in OS refer to the techniques used to access read and write data from and to a file. There are various file access methods in os like:

- Sequential Access
- Direct Access
- Indexed Sequential Access

### **Sequential Access**

The operating system reads the file word by word in a sequential access method of file accessing. A pointer is made, which first links to the file's base address. If the user wishes to read the first word of the file, the pointer gives it to them and raises its value to the next word. This procedure continues till the file is finished. It is the most basic way of file access. The data in the file is evaluated in the order that it appears in the file and that is why it is easy and simple to access a file's data using a sequential access mechanism. For example, editors and compilers frequently use this method to check the validity of the code.



**Examples of devices that use sequential access** – Sequential access is commonly used in devices such as tape drives, which require reading or writing data in a linear or sequential order.

### **Advantages of Sequential Access:**

- The sequential access mechanism is very easy to implement.

- It uses lexicographic order to enable quick access to the next entry.

#### **Disadvantages of Sequential Access:**

- Sequential access will become slow if the next file record to be retrieved is not present next to the currently pointed record.
- Adding a new record may need relocating a significant number of records of the file.

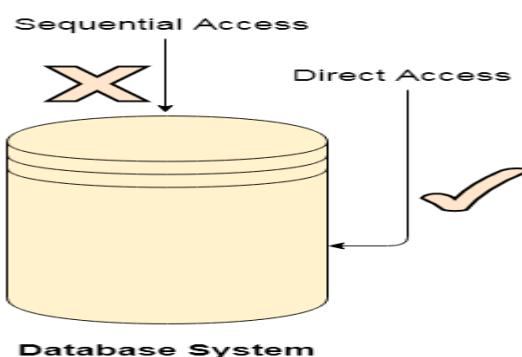
#### **Direct Access**

The Direct Access is mostly required in the case of database systems. In most of the cases, we need filtered information from the database. The sequential access can be very slow and inefficient in such cases.

Suppose every block of the storage stores 4 records and we know that the record we needed is stored in 10th block. In that case, the sequential access will not be implemented because it will traverse all the blocks in order to access the needed record.

In direct access, data is read or written directly to the physical location in the file. The data can be accessed by using the record number, byte position, or block number. This allows for fast and efficient access to specific data within the file.

For the direct-access method, the file operations must be modified to include the block number as a parameter. Thus, we have read n, where n is the block number, rather than read next, and write n rather than write next. An alternative approach is to retain read next and write next, as with sequential



**Examples of devices that use direct access** – Direct access is commonly used in devices such as magnetic disk drives, optical disk drives, and flash memory. These devices require fast and efficient access to specific data, which makes direct access an ideal file access method. Direct access is also commonly used in database systems, where fast access to specific records is required.

#### **Advantages of Direct/Relative Access:**

- The files can be retrieved right away with a direct access mechanism, reducing the average access time of a file.
- There is no need to traverse all of the blocks that come before the required block to access the record.

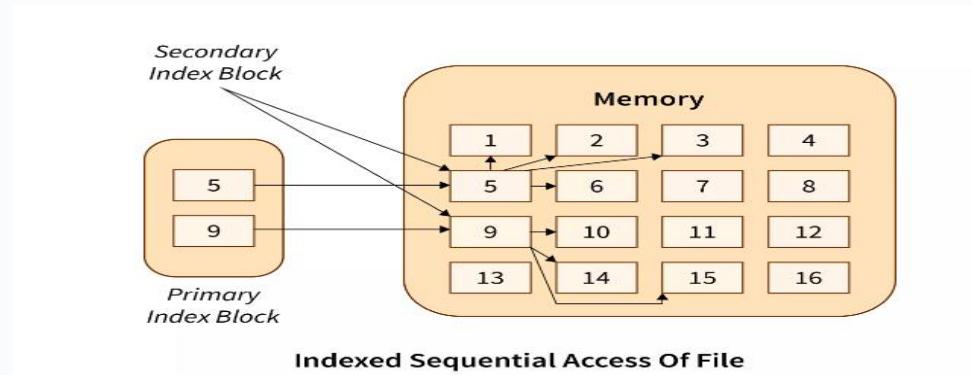
#### **Disadvantages of Direct/Relative Access:**

- The direct access mechanism is typically difficult to implement due to its complexity.
- Organizations can face security issues as a result of direct access as the users may access/modify the sensitive information. As a result, additional security processes must be put in place.

#### **Indexed Sequential Access**

It's the other approach to accessing a file that's constructed on top of the sequential access mechanism. This method is practically similar to the pointer-to-pointer concept in which we store the address of a pointer variable containing the address of some other variable/record in another pointer variable. The indexes, similar to a book's index (pointers), contain a link to various blocks present in the memory. To locate a record in the file, we first search the indexes and then use the pointer-to-pointer concept to navigate to the required file.

Primary index blocks contain the links of the secondary inner blocks which contain links to the data in the memory.



#### **Advantages of Indexed Sequential Access:**

- If the index table is appropriately arranged, it accesses the records very quickly.
- Records can be added at any position in the file quickly.

#### **Disadvantages of Indexed Sequential Access:**

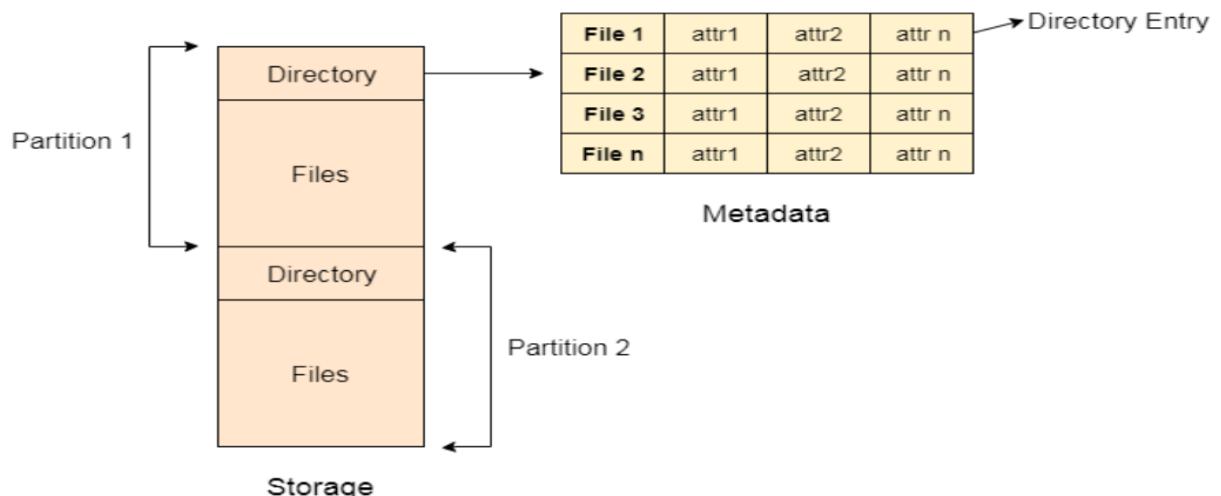
- When compared to other file access methods, it is costly and less efficient.
- It needs additional storage space.

## Unit-IV

### Topic 9: DIRECTORY AND DISK STRUCTURE

Systems stores millions of files on disk. Therefore, it is necessary to organize the disk to manage this large number of files. Directory can be defined as the listing of the related files on the disk. The directory may store some or the entire file attributes. To get the benefit of different file systems on the different operating systems, A hard disk can be divided into the number of partitions of different sizes. The partitions are also called volumes or mini disks.

Each partition must have at least one directory in which, all the files of the partition can be listed. A directory entry is maintained for each file in the directory which stores all the information related to that file.



*Fig: A typical file-system organization*

#### **Storage Structure**

- A general-purpose computer system has multiple storage devices, and those devices can be sliced up into volumes that hold file systems.
- Computer systems may have zero or more file systems, and the file systems may be of varying types.
- For example, a typical Solaris system may have dozens of file systems of a dozen different types

Consider the types of file systems in the Solaris

- **tmpfs**—a “temporary” file system that is created in volatile main memory and has its contents erased if the system reboots or crashes
- **lofs**—a “loop back” file system that allows one file system to be accessed in place of another one
- **procfs**—a virtual file system that presents information on all processes as a filesystem
- **ufs, zfs**—general-purpose file systems

### **Operations Performed on Directory**

Similar to how operations can be performed on files, there are operations that can be performed on directories. Some of the operations that can be performed on directories are given below:

- Search for a file – A directory has information about all the files present in the directory. To search for a file, it is necessary to search the directory.
- Create a file – To create a file, an entry is created in the directory. For this, it is necessary to write into the directory.
- Delete a file – To delete a file, it is necessary to remove the name of the file and all other details about the file from the directory. This again needs write permissions in the directory.
- List a directory – For listing the contents of a directory, it is necessary to read from the directory.
- Rename a file – To change the name of the file, it is necessary to read, write and search in the directory. Note that renaming a file may change the position of the file name in the directory.
- Traverse the file system – This also needs reading and searching operations on the directory.

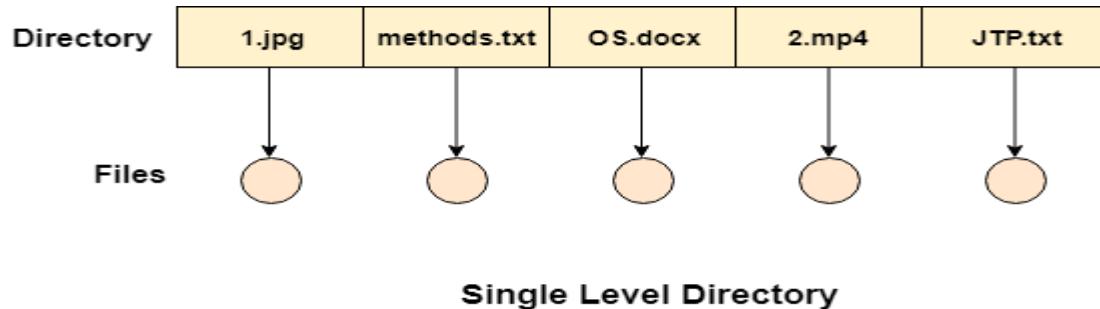
### **The Logical Directory Structures in OS**

We have mainly five different types of directory structures in OS

- Single level directory
- Two-level directory
- Tree structure or hierarchical directory
- Acyclic graph directory
- General graph directory structure

### **Single Level Directory**

The simplest method is to have one big list of all the files on the disk. The entire system will contain only one directory which is supposed to mention all the files present in the file system. The directory contains one entry per each file present on the file system.



This type of directories can be used for a simple system.

### Advantages

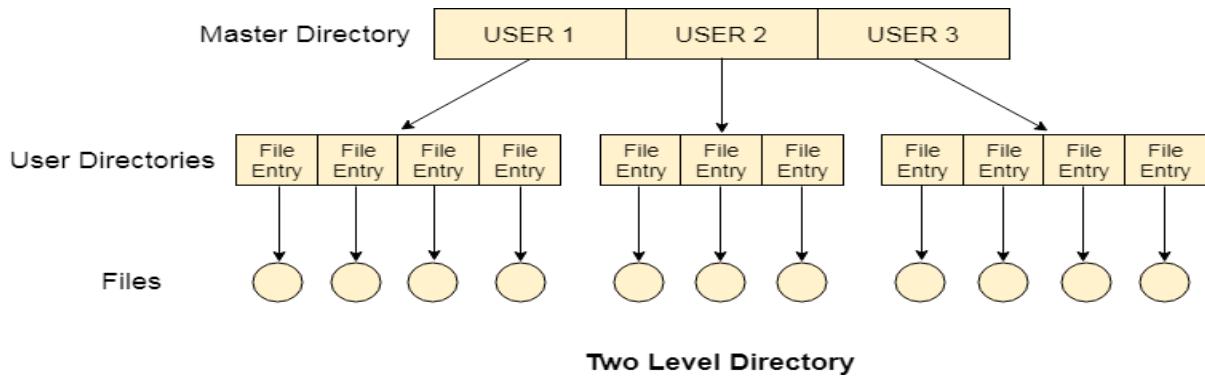
1. Implementation is very simple.
2. If the sizes of the files are very small then the searching becomes faster.
3. File creation, searching, deletion is very simple since we have only one directory.

### Disadvantages

1. We cannot have two files with the same name.
2. The directory may be very big therefore searching for a file may take so much time.
3. Protection cannot be implemented for multiple users.
4. There are no ways to group same kind of files.
5. Choosing the unique name for every file is a bit complex and limits the number of files in the system because most of the Operating System limits the number of characters used to construct the file name.

## Two Level Directory

In two level directory systems, we can create a separate directory for each user. There is one master directory which contains separate directories dedicated to each user. For each user, there is a different directory present at the second level, containing group of user's file. The system doesn't let a user to enter in the other user's directory without permission.



### Path Name

- In two-level directory, this tree structure has (Master File Directory) MFD as root of path through (User file Directory) UFD to user file name at leaf.
- Path name :: username + filename
- Standard syntax -- /user/file.ext

### Advantages

- Searching is very easy.
- There can be two files with the same name in two different user directories. Since they are not in the same directory, the same name can be used.
- Grouping is easier.
- A user cannot enter another user's directory without permission.
- Implementation is easy.

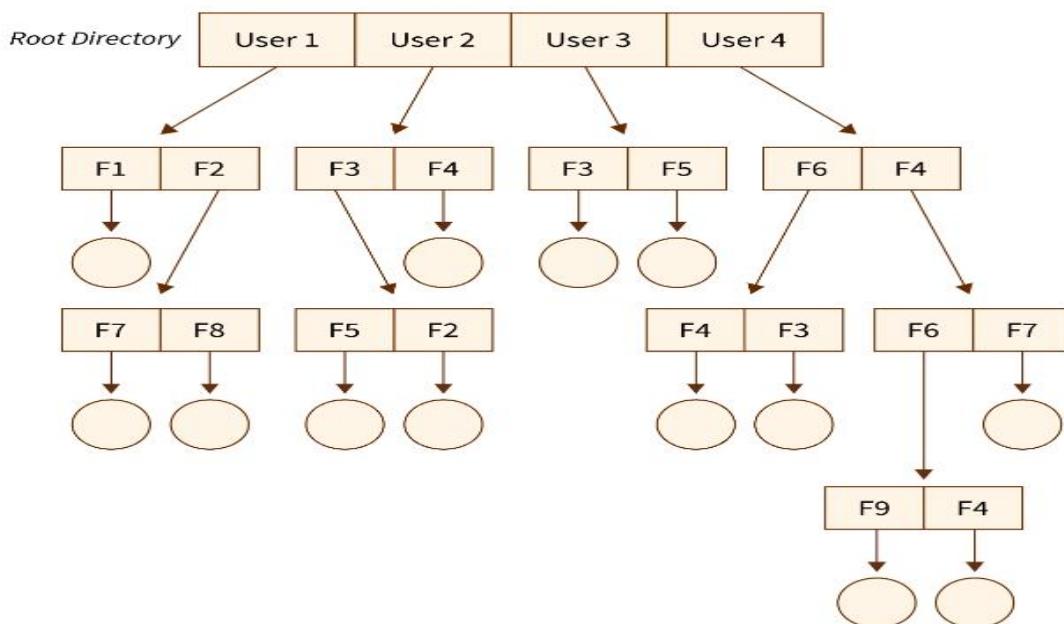
### Disadvantages

- One user cannot share a file with another user.
- Even though it allows multiple users, still a user cannot keep two same type files in a user directory.
- It does not allow users to create subdirectories.

### Tree-structured directory

The two-level directory was extended to have multiple levels and the tree-structured directory was formed. The tree is the most common directory structure. **Figure below shows an example of a tree-structured directory.**

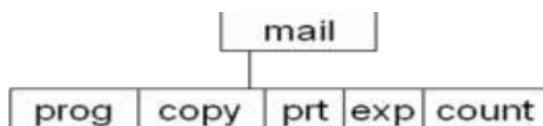
### Tree Directory Structure



In this structure, directory itself is a file. A directory and sub directory contains a set of files. Internal format is same for all directories. Commonly used directory structure is tree structure. Tree has a root directory. All files in disk have a unique path name.

- A path name describes the path that the operating system must take to get from some starting point in the file system to the destination object. Each process has its own working directory. Path names are of two types : relative and absolute.
- Relative path name : Its path starts from current working directory (may start with ..).
- Absolute path name: Its path starts from root directory (starts from "/").
- Tree structured directory differentiate file and subdirectory by using one bit representation. Zero (0) represents file and one (1) represents subdirectory.
- Whenever a file is newly created, it is created in the current directory. A new subdirectory is also created in the current directory. The command used for creating a subdirectory is **mkdir <dir-name>** where dir-name is the name of the subdirectory that is created. For example, if in current directory /mail, a subdirectory called *count* is to be created, then the command **mkdir count** is used.

**Figure below shows that a subdirectory *count* is created beneath the directory *mail*.**



- When process makes reference to a file for opening, file system search this file in the current directory. If needed file is not found in the current directory, then user either change the directory or give the full path of the file. System calls are used for performing any operation on the files or directory.
- Empty directory and its entry are easily deleted by operating system. If directory contains files and subdirectory, i.e. non-empty directory, some operating systems not delete the directory.
- To delete a directory, user must first delete all the files and subdirectory in that directory.

### **Advantages:**

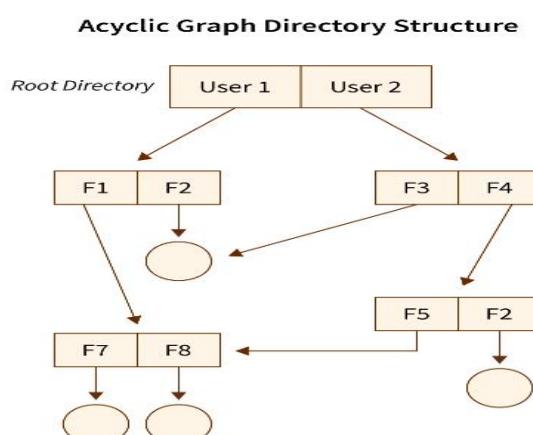
- This directory structure allows subdirectories inside a directory.
- The searching is easier.
- File sorting of important and unimportant becomes easier.
- This directory is more scalable than the other two directory structures explained.

### **Disadvantages:**

- As the user isn't allowed to access other user's directory, this prevents the file sharing among users.
- As the user has the capability to make subdirectories, if the number of subdirectories increase the searching may become complicated.
- Users cannot modify the root directory data.

### **The Acyclic Graph Directory Structure**

Overcoming the drawbacks posed by the tree-structured directory structure,i.e, the restriction that it cannot have multiple parent directories and also cannot share files between users - The Acyclic Graph directory structure in OS came into the picture.Mostly, this is used in situations such as, when two users or two programmers are collaborating on a project and they need to access the files.



- It is very interesting to note that a shared directory or file is not the same as two copies of the file. When there are two copies of files, each user can view the copy rather than the original, but if one user changes the file content, the changes will not appear in the other's copy.
- Only one original file exists for shared copy. Any changes made by one user are immediately visible to the other user. When user create file in shared directory, it automatically appear in all the shared subdirectories.

### **Implementation of shared files and directory is done through links**

- a. The Hard Link:** This is also called as the physical link. If we want to delete the files in the acyclic graph directory structures then we need to remove the actual files only if all the references to the files are deleted that is, no link that even references the main file should be established. Here we don't leave a suspended link.
- b. The Symbolic/Soft-Link:** This is also called as the logical link. If we want to delete the files in the acyclic graph directory structures then we need to simply delete the files and need to keep in mind that only a dangling/ hanging point is left. Here we leave a suspended link.

#### **Advantages:**

- In the **Acyclic Graph** directory structure in OS we can share files between users.
- Here we can search the files easily as compared to the tree-structured directory structure as here we have different-different paths to one file.

#### **Disadvantages:**

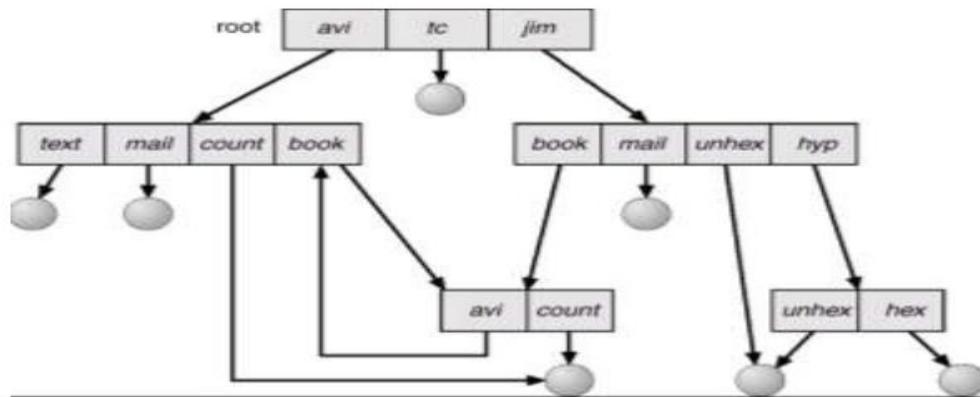
- In the Acyclic Graph directory structure in OS as we can share the files via linking, so there are chances that in the case when we want to delete a file in a directory it may create a problem.

### **The General Graph Directory Structure**

As observed in the acyclic-graph directory structure in OS the drawback of links that are established and need to be either terminated or suspended to reach the files/directory is resolved with The General Graph directory structure in OS taking a vital place in the different types of a directory structure in OS.

In this type of structure, the users are free to create different sub-directories for different file types. Adding to the above, With the help of the file paths if the users feel the need to access the location of the files then that is made possible by the General Graph Directory Structure.

***Below is the pictorial representation of The General Graph directory structure in OS :***



But if cycles are allowed to exist in directories, there is a possibility of searching a subdirectory twice, that is, it is possible to go from a parent directory to a subdirectory and from the subdirectory to the parent directory and so on. This may result in an infinite loop. To avoid this situation, one way is to limit the number of directories accessed during a search .The second way is to allow only links to file and not to subdirectories.

Let us now see how deletion of a file is handled in a general graph directory. Since cycles are allowed, if there is a self-referencing directory, there is a possibility to have a non-zero reference count even after deletion. To overcome this, garbage collection is to be done. During garbage collection, the file system is traversed and everything that can be accessed is marked in the first pass. In second pass, everything that is not marked is collected onto a list of free space. But this method is extremely time-consuming for a disk-based file system.

#### **Advantages:**

- The General Graph directory structure allows the cycle or creation of a directory within a directory.
- This directory structure is known to be a flexible version compared to other directory structures.

#### **Disadvantages:**

- As this directory structure allows the creation of multiple sub-directories a lot of garbage collection can be required.
- If compared to the other directory structure in OS the General Graph directory structure is a costly structure to be chosen.

## Unit-IV

### Topic 10 : Protection and File System Structure

It is necessary to keep files safe from physical damage (reliability) and from improper access (protection). For keeping the files reliable, it is necessary to have duplicate copies of files. We can also periodically copy disk files to tape at regular intervals. The owner/creator of the file should be able to control what can be done on a file and by whom.

#### **Types of Access :**

The files which have direct access of the any user have the need of protection. The files which are not accessible to other users doesn't require any kind of protection. The mechanism of the protection provide the facility of the controlled access by just limiting the types of access to the file. Access can be given or not given to any user depends on several factors, one of which is the type of access required. Several different types of operations can be controlled:

- **Read** – Reading from a file.
- **Write** – Writing or rewriting the file.
- **Execute** – Loading the file and after loading the execution process starts.
- **Append** – Writing the new information to the already existing file, editing must be end at the end of the existing file.
- **Delete** – Deleting the file which is of no use and using its space for the another data.
- **List** – List the name and attributes of the file.

One way in which access can be controlled is to have access control lists and groups.

#### **Access Control Lists and Groups**

With each file and directory an access-control list (ACL) is attached. The ACL has the names of users and the types of access allowed for each user. When a user requests access to a particular file, the access list is checked. If the user is listed for that particular access, access is allowed. Else, user is denied access.



This technique has two undesirable consequences:

- Constructing such a list may be a tedious and unrewarding task, especially if we do not know in advance the list of users in the system.
- The directory entry, previously of fixed size, now must be of variable size, resulting in more complicated space management.

These problems can be resolved by use of a condensed version of the access list. To condense the length of the access-control list, many systems recognize three classifications of users in connection with each file:

**Owner:** The user who created the file is the owner.

**Group:** A set of users who are sharing the file and need similar access is a group, or work group.

**Universe/others:** All other users in the system constitute the universe.

In UNIX, directory and file protection are handled similarly. Each file has three fields – owner, group and others. For each field there are three bits, r, w and x, corresponding to read, write and execute permissions. Suppose, the owner is provided read, write and execute permissions, r, w and x are set to 1 respectively. The octal number corresponding to the permissions allowed is 7 (111 in binary). Suppose the group is provided permissions for read and write, but no permissions for execution, the octal number for permissions is 6 (110 in binary). If others (universe, public) are provided only execute permissions, the octal number for permissions is 1 (001 in binary).

r w x

---

a) owner access	7	⇒	1 1 1
b) group access	6	⇒	1 1 0
c) public access	1	⇒	0 0 1

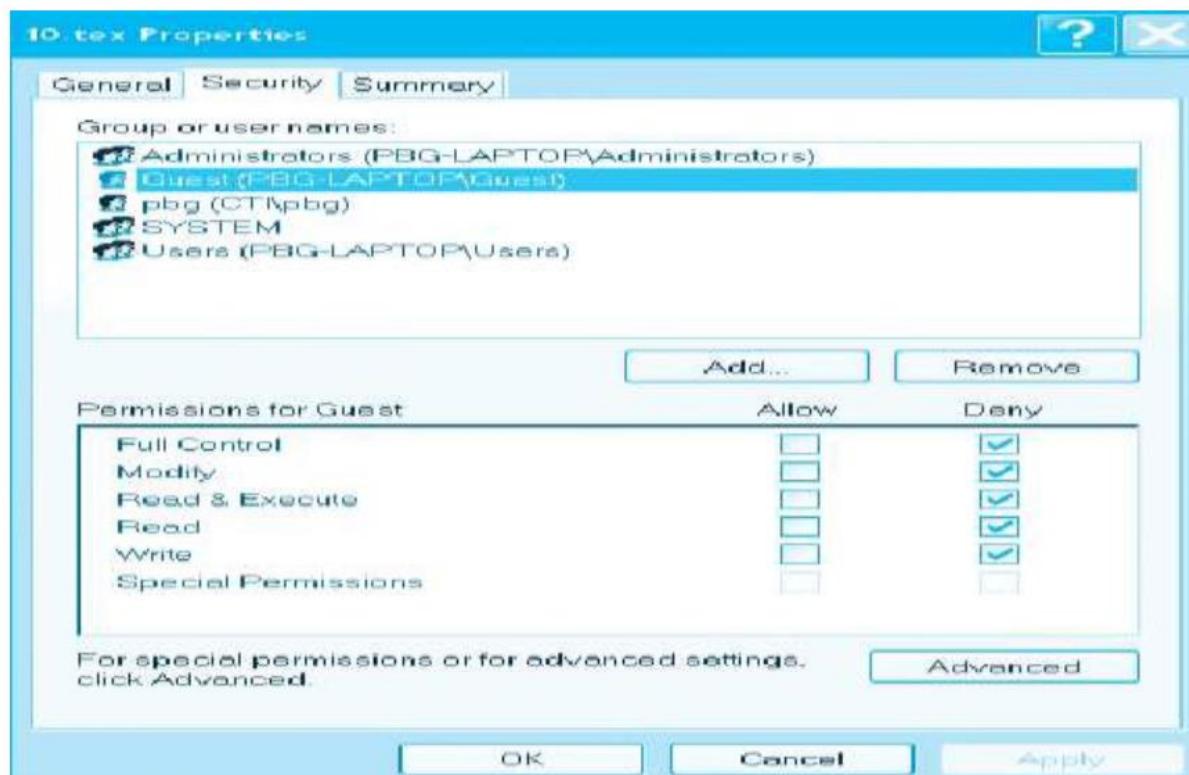
---

Thus, the access word for the file is 761 (rwx).

-rw-rw-r--	1	pbg	staff	31200	Sep 3 08:30	intro.ps
drwx-----	5	pbg	staff	512	Jul 8 09:33	private/
drwxrwxr-x	2	pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2	pbg	student	512	Aug 3 14:13	student-proj
-rw-r--r--	1	pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1	pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4	pbg	faculty	512	Jul 31 10:31	lib/
drwx-----	3	pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3	pbg	staff	512	Jul 8 09:35	test/

**Figure above shows a sample UNIX directory listing.** The first file in the list is intro.ps. This file has read and write permissions for owner, read and write permissions for group and read permissions for others. The second row corresponds to a subdirectory called private in the listed directory. The letter 'd' before the access permissions indicates that 'private' is a directory. This directory 'private' has read, write and execute permissions for owner, but no permissions for group and others.

**Figure below shows a example of access control provided for guest user in Windows XP. The guest user is denied all permissions.**



### Other Approaches

Another approach to the protection problem is to associate a password with each file. Just as access to the computer system is often controlled by a password, access to each file can be controlled in the

same way. If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file.

The use of passwords has a few disadvantages, however.

First, the number of passwords that a user needs to remember may become large, making the scheme impractical.

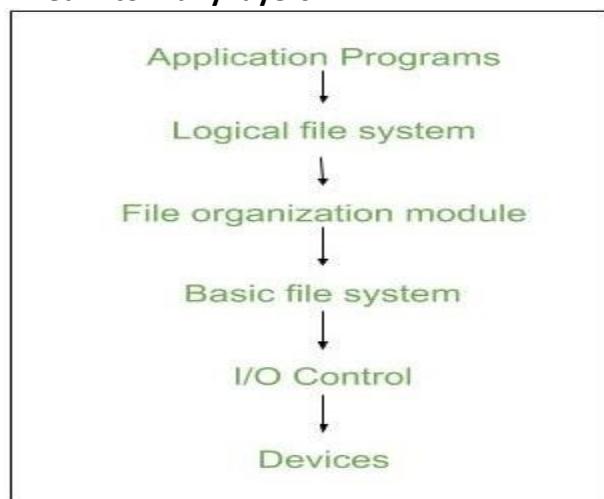
Second, if only one password is used for all the files, then once it is discovered, all files are accessible; protection is on an all-or-none basis.

Some systems allow users to associate password with a subdirectory. In this case, it is not necessary to assign a password for each and every file within this directory. It is enough to assign a single password for the entire directory.

### File System Structure

The **file system structure** defines how information in key directories and files (existing and future) are organized and how they are stored in an operating system.

**The file system is organized into many layers:**



#### I/O control level

- This layer is placed just above the I/O devices. This level comprises of the device drivers and the interrupt handlers. The device drivers act as an interface between the devices and the operating system. They help to transfer information between the main memory and the disk.
- An example of input given to a device driver is – ‘retrieve block 123’. In response to this, the device driver has to send low-level hardware specific instructions to the disk controller. The disk controller assists in reading block 123 from the disk.

#### Basic file system

- This layer issues generic commands to the device driver to read and write physical blocks on the disk. The input received from the I/O control layer is sent from this layer. Memory buffers and caches are maintained by the operating system in the main memory.
- The basic file system layer is responsible for managing the memory buffers and caches. Each memory buffer can hold contents equal to the size of a block. Each buffer holds the contents of a disk block. The contents that are read from the disk are copied to the buffers and can even be used later.
- The cache holds frequently used file-system metadata. This can be the contents of the file or attributes of the file such as the owner of the file, size of the file and so on. If the file is used frequently, the metadata can be used from the cache itself. It is not necessary to read from the disk each time.

### **File-organization module**

- This layer uses the functionalities of the basic file system layer. This layer knows about files and their logical and physical blocks. The logical blocks are with respect to a particular file. The logical blocks are numbered from 0 to N for a particular file.
- Physical blocks do not match the logical numbers. Logical block  $i$  need not be kept in block number  $i$  in the physical memory. It can be kept in any physical disk block. Therefore, it is necessary to know the location of the location of the file in the disk (That is the locations of the disk blocks where the contents of the file are kept in the disk).
- Therefore, appropriate data structures are maintained by the file-organization module to know the mapping between the logical block number and the physical block number. The file-organization module also has a free-space manager, which tracks unallocated disk blocks.

### **Logical file system**

- This layer lies above the file-organization module. This layer manages the metadata information of a file. The metadata information includes all details about a file except the actual contents of the file, for example, the name of the file, the size of the file and so on.
- This layer also manages the directory structure. It maintains the file-structure via file-control blocks. A File-control block (FCB) (inode in UNIX) has information about a file – owner, size, permissions, time of access, location of file contents and so on.

### **Advantages of layered file system**

- Duplication of code is minimized. The code for I/O control and basic file-system layers can be used by multiple file systems.
- The layers above these two layers can be modified for different file systems. That is, each file system can then have its own logical file system and file-organization modules, while the I/O control and basic file-system layers being the same.

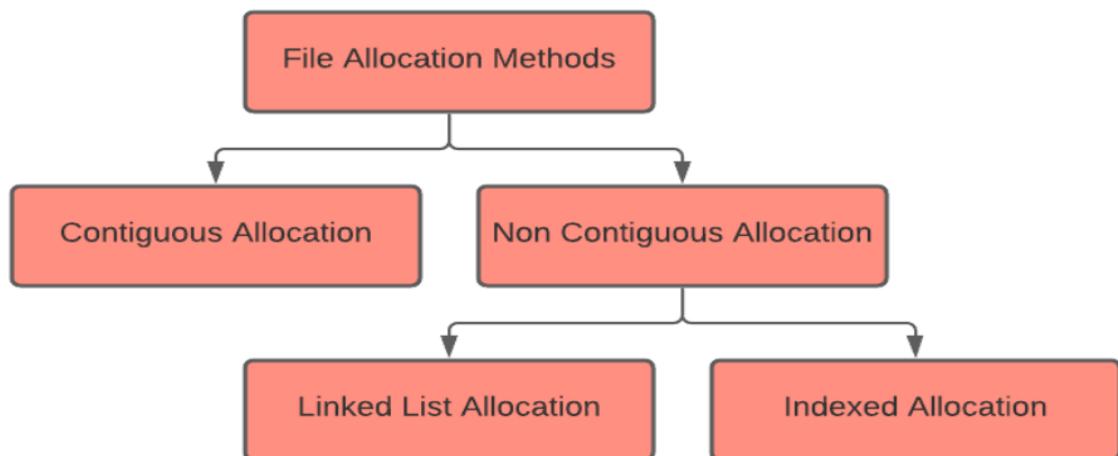
### **Disadvantages**

- Having a layered file system can introduce more operating-system overhead, resulting in decreased performance. The decision about how many layers to use, what each layer should do is a challenge.
- Many file-systems are in use today – UNIX file system, FAT, FAT32, NTFS, ext3, ext4, Google. The FAT, FAT32 and NTFS are used in Windows operating systems. Ext3 and ext4 are used in Linux operating systems. Google has its own distributed file system called the Google File System (GFS).

## Unit-IV

### Topic 11(File Allocation Methods)

- Generally, files are stored in secondary storage devices such as hard disks. So to manage the space of hard disks efficiently and store the files fastly the Operating\_System uses file allocation methods.
- The file system divides the file logically into multiple blocks. These blocks now need to be stored in a secondary storage device such as a hard disk. The hard disk contains multiple sectors or physical blocks in which the data actually reside. Now the file system will decide where to store these blocks of a file in the sectors of the hard disk using file allocation methods.

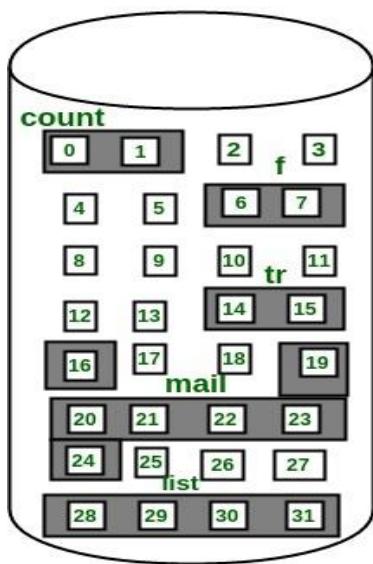


#### Contiguous Allocation

- In this scheme, each file occupies a contiguous set of blocks on the disk. For example, if a file requires  $n$  blocks and is given a block  $b$  as the starting location, then the blocks assigned to the file will be:  $b, b+1, b+2, \dots, b+n-1$ .
- This means that given the starting block address and the length of the file (in terms of blocks required), we can determine the blocks occupied by the file. The directory entry for a file with contiguous allocation contains

- Address of starting block
- Length of the allocated portion.

The file 'mail' in the following figure starts from the block 19 with length = 6 blocks. Therefore, it occupies 19, 20, 21, 22, 23, 24 blocks.



Directory		
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

- Contiguous allocation also suffers from external fragmentation. Small free disk spaces are created after allocation free block and deleting files.
- External fragmentation means there will require free space available but that is not contiguous. To solve the problem of external fragmentation, compaction method is used.
- One more problem is that how to calculate the space needed for a file. It is difficult to estimate the required space.

### Advantages

- It is very easy to implement.
- There is a minimum amount of seek time.
- The disk head movement is minimum.
- Memory access is faster.
- It supports sequential as well as direct access.

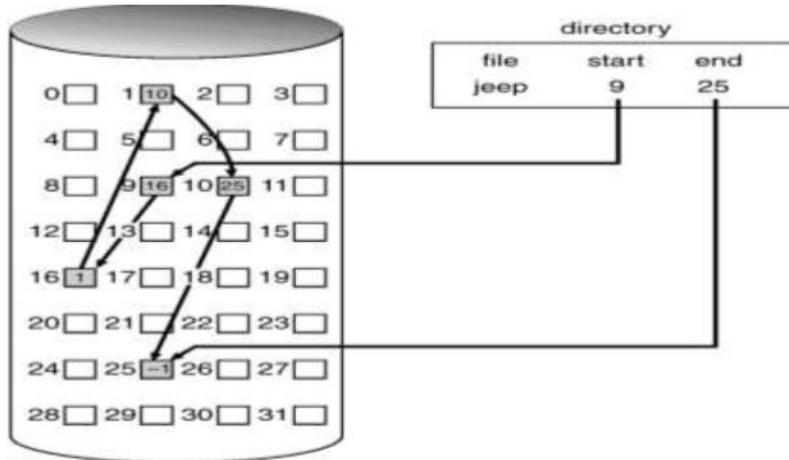
### Disadvantages

- At the time of creation, the file size must be initialized.
- As it is pre-initialized, the size cannot increase.
- Due to its constrained allocation, it is possible that the disk would fragment internally or externally.

### Linked File Allocation

The linked allocation solves all problems of contiguous allocation. In linked allocation, each file is a linked list of disk blocks. The allocated disk blocks may be scattered anywhere on the disk. The directory entry has a pointer to the first and the last blocks of the file.

**Figure below shows an example of linked allocation.**



There is a file named *jeep*. The starting disk block number 9 is given in the directory entry. Disk block 9 has the address of the next disk block, 16. Disk block 16 points to disk block 1 and so on. The last disk block of the file, disk block 25 has a pointer value of -1 indicating that it is the last disk block of the file.

#### Advantages

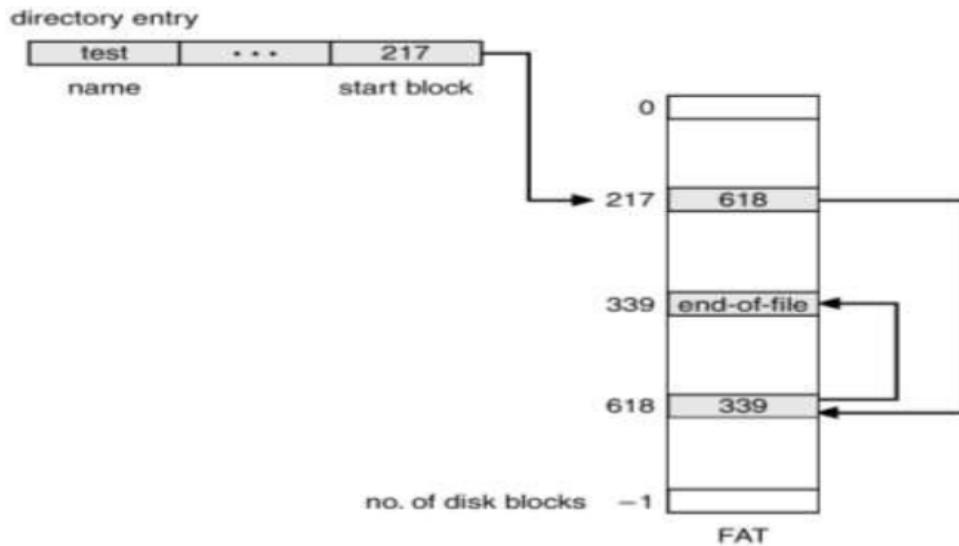
- There is no external fragmentation.
- The directory entry just needs the address of starting block.
- The memory is not needed in contiguous form, it is more flexible than contiguous file allocation.

#### Disadvantages

- It does not support random access or direct access.
- If pointers are affected so the disk blocks are also affected.(no reliability)
- Extra space is required for pointers in the block.

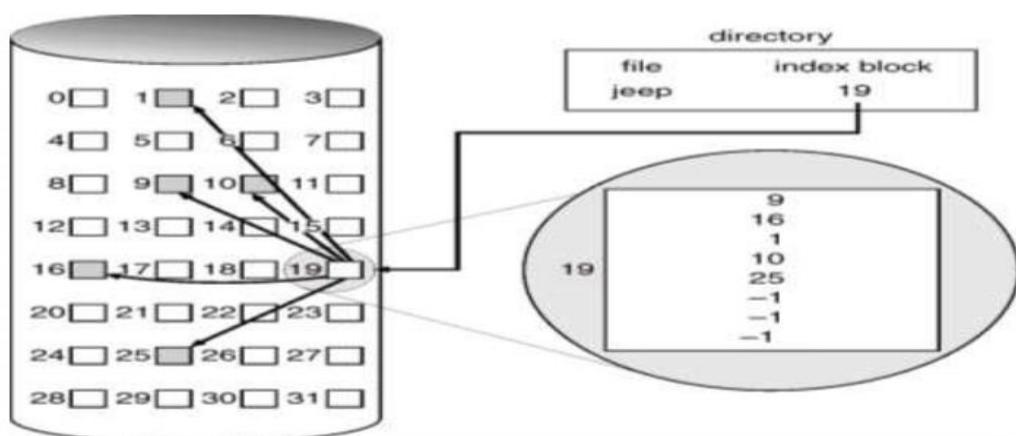
A variation of linked allocation scheme is used in the File-allocation table (FAT). This is the disk-space allocation scheme used by MS-DOS. A section of the disk at the beginning of each volume contains this table (FAT).

Figure below shows an example file allocation table. The directory entry has the address of the first disk block number (217). This 217 is used as an index into the FAT to get the entry where 618 is stored. This means that the next disk block number is 618. While using 618 as an index into the FAT, 339 is got, which means that, 339 is the next disk block. When 339 is used as index, the entry shows that end of file is reached.



### Indexed Allocation

- The indexed allocation solves the external fragmentation and size-declaration problem of contiguous allocation. It also solves the direct access problem in linked allocation.
- In indexed allocation all pointers are brought together into one block called the *index block*. Each file has an index block, which is an array of disk-block addresses. The *i*th entry in the index block points to the *i*th block of the file.



**Figure above shows an example of indexed allocation. The directory entry has the address of the index block (disk block 19). The index block has the addresses of the disk blocks (9, 16, 1, 10 and 25) of the file.**

**Advantages:**

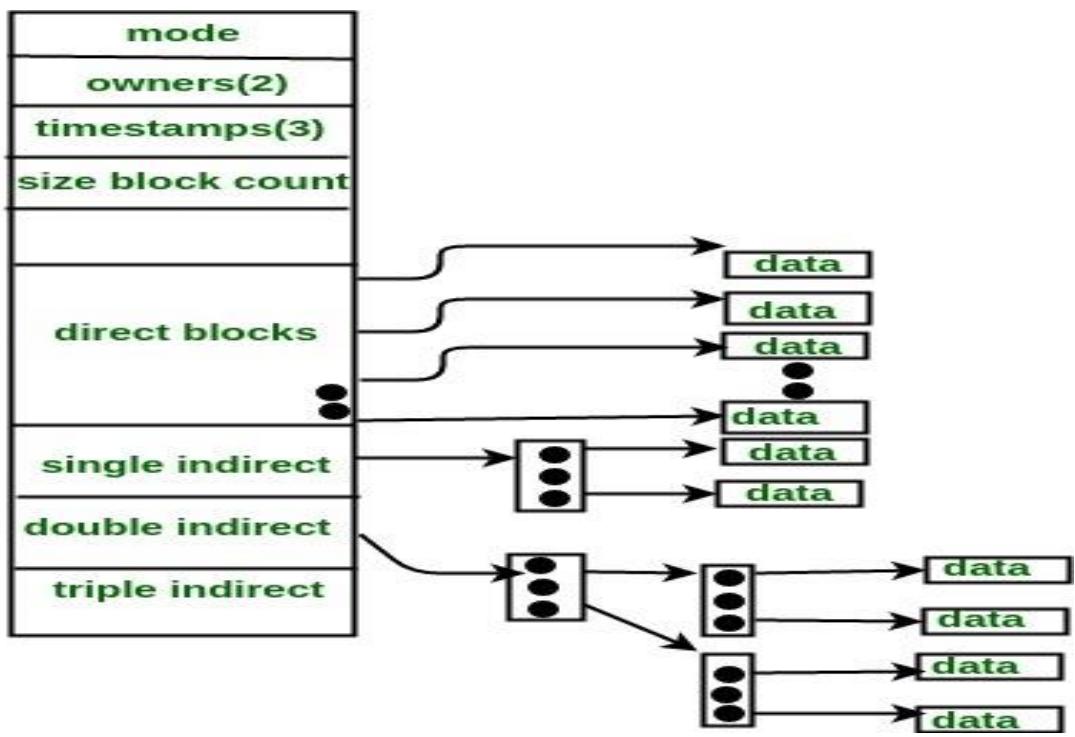
- This supports direct access to the blocks occupied by the file and therefore provides fast access to the file blocks.
- It overcomes the problem of external fragmentation.

**Disadvantages:**

- The pointer overhead for indexed allocation is greater than linked allocation.
  - For very small files, say files that expand only 2-3 blocks, the indexed allocation would keep one entire block (index block) for the pointers which is inefficient in terms of memory utilization.
- However, in linked allocation we lose the space of only 1 pointer per block.

**For files that are very large, single index block may not be able to hold all the pointers. Following mechanisms can be used to resolve this:**

1. **Linked scheme:** This scheme links two or more index blocks together for holding the pointers. Every index block would then contain a pointer or the address to the next index block.
2. **Multilevel index:** In this policy, a first level index block is used to point to the second level index blocks which in turn points to the disk blocks occupied by the file. This can be extended to 3 or more levels depending on the maximum file size.
3. **Combined Scheme:** In this scheme, a special block called the **Inode (information Node)** contains all the information about the file such as the name, size, authority, etc and the remaining space of Inode is used to store the Disk Block addresses which contain the actual file as shown in the image below. The first few of these pointers in Inode point to the **direct blocks** i.e the pointers contain the addresses of the disk blocks that contain data of the file.
  - The next few pointers point to indirect blocks. Indirect blocks may be single indirect, double indirect or triple indirect.
  - **Single Indirect block** is the disk block that does not contain the file data but the disk address of the blocks that contain the file data.
  - Similarly, **double indirect blocks** do not contain the file data but the disk address of the blocks that contain the address of the blocks containing the file data.



## Unit-IV

### Topic 12: Free Space Management

- Free space management is a critical aspect of operating systems as it involves managing the available storage space on the hard disk or other secondary storage devices
- The system maintains a free space list by keep track of the free disk space. The free space list contains all the records of the free space disk block.
- When we create a file, we first search for the free space in the memory and then check in the free space list for the required amount of space that we require for our file.
- if the free space is available then allocate this space to the new file. After that, the allocating space is deleted from the free space list. Whenever we delete a file then its free memory space is added to the free space list.

There are some methods or techniques to implement a free space list. These are as follows:

1. Bitmap
2. Linked list
3. Grouping
4. Counting
5. Space Maps

#### **Bitmap**

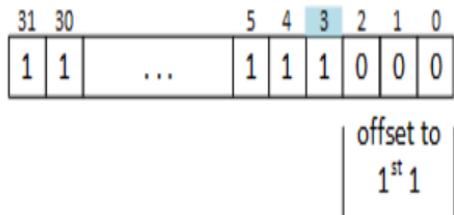
- This technique is used to implement the free space management. When the free space is implemented as the bitmap or bit vector then each block of the disk is represented by a bit. When the block is free its bit is set to 1 and when the block is allocated the bit is set to 0.
- The main advantage of the bitmap is it is relatively simple and efficient in finding the first free block and also the consecutive free block in the disk. Many computers provide the bit manipulation instruction which is used by the users.

**2,3,4,5,9,10,13**

Blocks →	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Bits →	0	0	1	1	1	1	0	0	0	1	1	0	0	1

The calculation of the block number is done by the formula: **(number of bits per words) X (number of 0-value word) + Offset of first 1 bit**

(u)



Number of bits per word = 32  
number of 0-value words = 3  
offset to the first 1 bit = 3  
 $32 * 3 + 3 = 96 + 3 = 99$

### Advantages

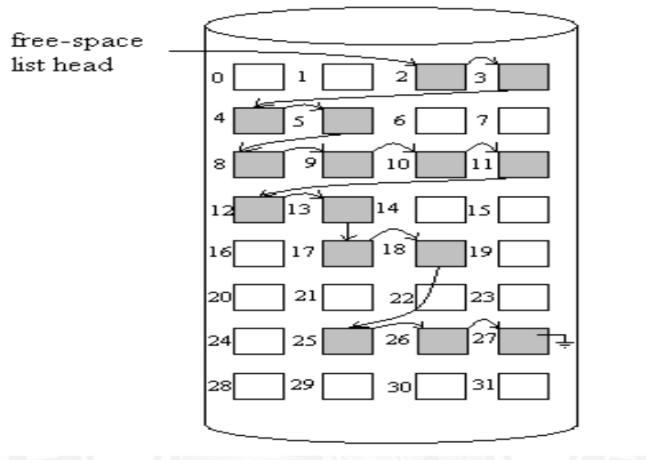
- This technique is relatively simple.
- This technique is very efficient to find the free space on the disk.

### Disadvantages

- Again, we see hardware features driving software functionality. Unfortunately, bit vectors are inefficient unless the entire vector is kept in main memory (and is written to disk occasionally for recovery needs). Keeping it in main memory is possible for smaller disks but not necessarily for larger ones.

### Linked list

- Another approach to free-space management is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory. This first block contains a pointer to the next free disk block, and so on.
- If for example blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 were free and the rest of the blocks were allocated. In this situation, we would keep a pointer to block 2 as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, **as shown in figure below**



### Advantages

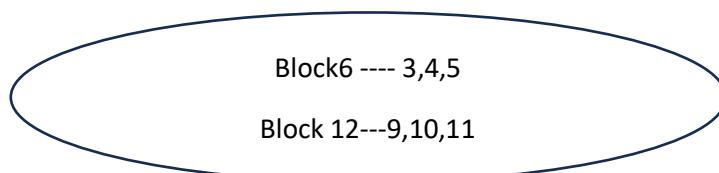
- Whenever a file is to be allocated a free block, the operating system can simply allocate the first block in free space list and move the head pointer to the next free block in the list.

### Disadvantages

- Searching the free space list will be very time consuming; each block will have to be read from the disk, which is read very slowly as compared to the main memory.
- Not Efficient for faster access.

### Grouping

- This is also the technique of free space management. In this, there is a modification of the free-list approach which stores the address of the n free blocks. In this the first n-1 blocks are free but the last block contains the address of the n blocks. When we use the standard linked list approach the addresses of a large number of blocks can be found very quickly.
- For example, consider a disk having 16 blocks where block numbers 3, 4, 5, 6, 9, 10, 11, 12, 13, and 14 are free, and the rest of the blocks, i.e., block numbers 1, 2, 7, 8, 15 and 16 are allocated to some files.
- If we apply the Grouping method considering  $n$  to be 4, Block 6 will store the addresses of Block 3, Block 4, and Block 5. Similarly, Block 12 will store the addresses of Block 9, Block 10, and Block 11. Block 11 will store the addresses of Block 12, Block 13, and Block 14. This is also represented in the following figure-



### **Advantage**

1. By using this method, we can easily find addresses of a large number of free blocks easily and quickly.

### **Disadvantage**

1. We need to change the entire list if one block gets occupied.

### **Counting**

- This is the fourth method of free space management in operating systems. This method is also a modification of the linked list method. This method takes advantage of the fact that several contiguous blocks may be allocated or freed simultaneously.
- In this method, a linked list is maintained but in addition to the pointer to the next free block, a count of free contiguous blocks that follow the first block is also maintained.
- Thus each free block in the disk will contain two things-
  1. A pointer to the next free block.
  2. The number of free contiguous blocks following it.

For example, consider a disk having 16 blocks where block numbers 3, 4, 5, 6, 9, 10, 11, 12, 13, and 14 are free, and the rest of the blocks, i.e., block numbers 1, 2, 7, 8, 15 and 16 are allocated to some files. If we apply the counting method, Block 3 will point to Block 4 and store the count 4 (since Block 3, 4, 5, and 6 are contiguous). Similarly, Block 9 will point to Block 10 and keep the count of 6 (since Block 9, 10, 11, 12, 13, and 14 are contiguous). This is also represented in the following figure-

Block 3 -> 4  
Block 9 -> 6

### **Advantages**

- Fast allocation of a large number of consecutive free blocks.
- Random access to the free block is possible.
- The overall list is smaller in size.

### **Disadvantages**

- Each free block requires more space for keeping the count in the disk.
- For efficient insertion, deletion, and traversal operations. We need to store the entries in B-tree.

- The entire area is reduced.

### **Space map**

ZFS (Zettabyte File System) is an advanced file system and logical volume manager designed by Sun Microsystems. One of its most notable features is its sophisticated free space management, which provides high performance, reliability, and efficiency.

**Selects a Metaslab:** *ZFS divides each virtual storage pool into metaslabs, which are chunks of storage (usually hundreds of MBs to a few GBs).*

ZFS uses a metaslab selection algorithm to choose the most appropriate metaslab for the allocation request. This can be based on factors like the amount of free space, recent usage patterns, and fragmentation levels.

**Updates the Spacemap:** *Spacemaps are stored in a log-structured format, recording free space transactions (allocations and deallocations) in a sequential manner*

- ZFS records the allocation in the metaslab's spacemap, adding an entry that specifies the start block and the length of the allocated space.

**Updates Metadata:** *Space maps track space allocation changes, recording the start block, length, and type of operation (allocation or deallocation).*

- The allocation is also reflected in the relevant metadata structures, ensuring that the file system remains consistent.

### **Advantages**

- Space maps offer a powerful and efficient way to manage free space in large and dynamic storage systems. Their advantages include efficient space tracking, improved performance, consistency, and flexibility.

### **Disadvantages**

**Implementation Complexity:** Managing space maps is more complex than simpler structures like bit vectors. This complexity can make the implementation more difficult and increase the potential for bugs.

- **Metadata Overhead:** While space maps reduce metadata overhead compared to tracking each block individually, they still consume memory to store the ranges and associated data. This overhead can become significant in extremely large storage systems.
- **Search Overhead:** Searching for free space in a highly fragmented space map can be slower compared to simpler structures, especially if the data structure used for the space map is not optimized for quick searches.

## Unit-IV

### Topic 13: System calls for File Management

These system call are responsible for all services related to file manipulation, such as writing into a file, reading from a file, creating a file, and so on.

*Some system calls for file management like*

**create()**: The create() function is used to create a new empty file in C. We can specify the permission and the name of the file which we want to create using the create() function.

#### **syntax**

```
int create(char *filename, mode_t mode);
```

#### **Parameter**

**filename**: name of the file which you want to create

**mode**: indicates permissions of the new file.

The mode can be specified using a combination of the following constants, defined in <sys/stat.h>:

- S\_IRUSR (or S\_IREAD): Read permission for the owner.
- S\_IWUSR (or S\_IWRITE): Write permission for the owner.
- S\_IXUSR (or S\_IEXEC): Execute (or search, if it's a directory) permission for the owner.
- S\_IRGRP: Read permission for the group.
- S\_IWGRP: Write permission for the group.
- S\_IXGRP: Execute (or search) permission for the group.
- S\_IROTH: Read permission for others.
- S\_IWOTH: Write permission for others.
- S\_IXOTH: Execute (or search) permission for others.

These constants can be combined using the bitwise OR operator (|) to set multiple permissions.

#### **Return Value**

returns file descriptor)

return -1 when an error

**open ()**: open() system call is used to know the file descriptor of user-created files. Since read and write use file descriptor as their 1st parameter so to know the file descriptor open() system call is used.

**Syntax:**

```
fd = open (file_name, mode, permission);
```

Example:

```
fd = open ("file", O_CREAT | O_RDWR, 0777);
```

Here,

file\_name is the name to the file to open.

mode is used to define the file opening modes such as

O_RDONLY	Opens the file in read-only mode.
O_WRONLY	Opens the file in write-only mode.
O_RDWR	Opens the file in read and write mode.
O_CREAT	Create a file if it doesn't exist.
O_EXCL	Prevent creation if it already exists.
O_APPEND	Opens the file and places the cursor at the end of the contents.
O_ASYNC	Enable input and output control by signal.
O_CLOEXEC	Enable close-on-exec mode on the open file.
O_NONBLOCK	Disables blocking of the file opened.
O_TMPFILE	Create an unnamed temporary file at the specified path.

permission is used to define the file permissions.

**Return value:** Function returns the file descriptor.

**read (): read() system call** is used to read the content from the file. It can also be used to read the input from the keyboard by specifying the **0** as file descriptor .

**Syntax:**

```
length = read(file_descriptor , buffer, max_len);
```

Example:

```
n = read(0, buff, 50);
```

**Here,**

- **file\_descriptor** is the file descriptor of the file.
- **buffer** is the name of the buffer where data is to be stored.
- **max\_len** is the number specifying the maximum amount of that data can be read

**Return value:** If successful read returns the number of bytes actually read.

**write (): write() system call** is used to write the content to the file.

**Syntax:**

```
length = write(file_descriptor , buffer, len);
```

Example:

```
n = write(fd, "Hello world!", 12);
```

**Here,**

- **file\_descriptor** is the file descriptor of the file.
- **buffer** is the name of the buffer to be stored.
- **len** is the length of the data to be written.

**Return value:** If successful write() returns the number of bytes actually written.

**close (): close() system call** is used to close the opened file, it tells the operating system that you are done with the file and close the file.

**Syntax:**

```
int close(int fd);
```

**Here,**

- **fd** is the file descriptor of the file to be closed.

**Return value:** If file closed successfully it returns 0, else it returns -1.

```
#include<unistd.h>
#include<fcntl.h>
#include<sys/stat.h>
#include<sys/types.h>
#include<stdio.h>

int main()
{
    int n,fd;
    char buff[50];
    printf("Enter text to write in the file:\n");
    n= read(0, buff, 50);
    fd=open("file",O_CREAT | O_RDWR, 0777);
    write(fd, buff, n);
    write(1, buff, n);
    close(int fd);
    return 0;
}
```

**Iseek():** Iseek() system call repositions the read/write file offset i.e., it changes the positions of the read/write pointer within the file. In every file any read or write operations happen at the position pointed to by the pointer. Iseek() system call helps us to manage the position of this pointer within a file.

*e.g., let's suppose the content of a file F1 is "1234567890" but you want the content to be "12345hello". You simply can't open the file and write "hello" because if you do so then "hello" will be written in the very beginning of the file. This means you need to reposition the pointer after '5' and then start writing "hello". Iseek() will help to reposition the pointer and write() will be used to write "hello"*

The first parameter is the file descriptor of the file, which you can get using open() system call. the second parameter specifies how much you want the pointer to move and the third parameter is the

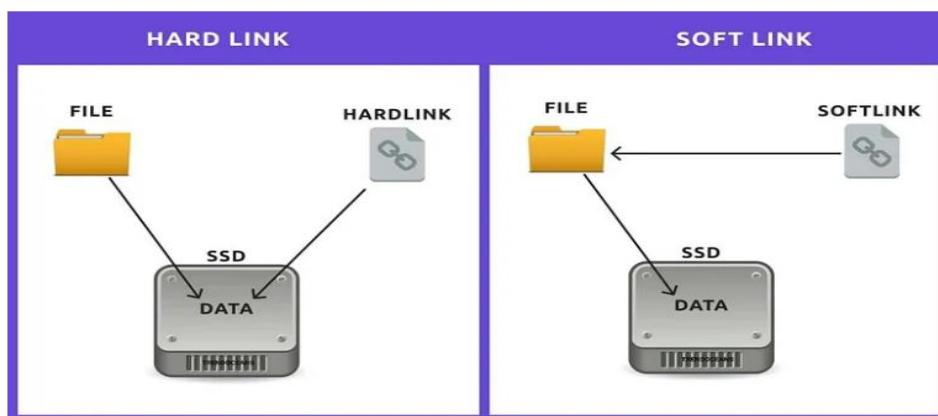
reference point of the movement i.e., beginning of file(SEEK\_SET), current position(SEEK\_CUR) or pointer or end of file(SEEK\_END).

#### Examples:

- lseek(fd,5,SEEK\_SET) – this moves the pointer 5 positions ahead starting from the beginning of the file
- lseek(fd,5,SEEK\_CUR) – this moves the pointer 5 positions ahead from the current position in the file
- lseek(fd,-5,SEEK\_CUR) – this moves the pointer 5 positions back from the current position in the file
- lseek(fd,-5,SEEK\_END) -> this moves the pointer 5 positions back from the end of the file

On success, lseek() returns the position of the pointer within the file as measured in bytes from the beginning of the file. But, on failure, it returns -1.

A link in UNIX is a pointer to a file. Like pointers in any programming languages, links in UNIX are pointers pointing to a file or a directory. Creating links is a kind of shortcuts to access a file. Links allow more than one file name to refer to the same file. There are two types of links :



#### 1. Soft Link:

A soft link (also known as Symbolic link) acts as a pointer or a reference to the file name. It does not access the data available in the original file. If the earlier file is deleted, the soft link will be pointing to a file that does not exist anymore.

#### 2. Hard Link :

A hard link acts as a copy (mirrored) of the selected file. It accesses the data available in the original file. If the earlier selected file is deleted, the hard link to the file will still contain the data of that file.

#### link() & symlink()

In Unix-like operating systems, `link()` and `symlink()` are system calls used to create hard links and symbolic links (symlinks) respectively. Here's an explanation of each:

**link():** The `link()` function creates a new link (hard link) to an existing file.

```
#include <unistd.h>
int link(const char *oldpath, const char *newpath);
```

- **Parameters:**

`const char *oldpath`: The path name of an existing file.

`const char *newpath`: The path name of the new link to be created.

- **Returns:**

0 on success.

-1 on failure, and `errno` is set to indicate the error

**symlink():** The `symlink()` function creates a symbolic link (symlink) to a target file.

```
#include <unistd.h>
int symlink(const char *target, const char *linkpath);
```

- **Parameters:**

`const char *target`: The path name of the target file that the symlink will point to.

`const char *linkpath`: The path name where the symbolic link will be created.

- **Returns:**

0 on success.

-1 on failure, and `errno` is set to indicate the error.

**Unlink():** The `unlink()` function in Unix-like operating systems is used to delete a name (unlink a file) from the file system. It removes the link to a file, and if that was the last link to the file and no processes have the file open, the file itself is deleted and the space it was using is made available for reuse.

```
#include <unistd.h>
int unlink(const char *pathname);
```

**Parameters**

`const char *pathname`: The path name of the file to be unlinked (deleted).

**Return Value**

Returns 0 on success.

Returns -1 on failure, and sets `errno` to indicate the error

```

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

int main() {
    const char *sourcefile = "source.txt";
    const char *hardlink = "hardlink.txt";
    const char *symlinkfile = "shortcut";
    // Creating a hard link
    if (link(sourcefile, hardlink) == -1) {
        perror("link");
        return 1;
    }
    printf("Hard link created: %s -> %s\n", hardlink, sourcefile);
    // Creating a symbolic link
    if (symlink(sourcefile, symlinkfile) == -1) {
        perror("symlink");
        return 1;
    }
    printf("Symbolic link created: %s -> %s\n", symlinkfile, sourcefile);
    // Deleting the source file using unlink()
    if (unlink(sourcefile) == -1) {
        perror("unlink");
        return 1;
    }
    printf("File '%s' deleted successfully.\n", sourcefile);
    return 0;
}

```

**stat(), fstat() and lstat():** The stat, fstat, and lstat functions in Unix-like operating systems are used to retrieve detailed information about files and directories. They are part of the system's API and are declared in the <sys/stat.h> header file. Here's an overview of each function, their similarities, and differences:

**stat(): The stat function retrieves information about a file specified by its pathname.**

```

#include <sys/stat.h>
int stat(const char *pathname, struct stat *buf);

```

**Parameters:**

`const char *pathname`: The path to the file.

`struct stat *buf`: A pointer to a stat structure where the information will be stored.

**Returns:**

0 on success.

-1 on failure, and `errno` is set to indicate the error.

**fstat()**: The fstat function retrieves information about an open file referred to by a file descriptor.

```
#include <sys/stat.h>
int fstat(int fd, struct stat *buf);
```

**Parameters:**

`int fd`: The file descriptor of the file.

`struct stat *buf`: A pointer to a stat structure where the information will be stored.

**Returns:**

0 on success.

-1 on failure, and `errno` is set to indicate the error.

**lstat()**: The lstat function is similar to stat, but it retrieves information about a symbolic link itself rather than the file it points to.

```
#include <sys/stat.h>
int lstat(const char *pathname, struct stat *buf);
```

**Parameters:**

`const char *pathname`: The path to the file or symbolic link.

`struct stat *buf`: A pointer to a stat structure where the information will be stored.

**Returns:**

0 on success.

-1 on failure, and `errno` is set to indicate the error.

***stat Structure***

All three functions use the stat structure to store information about the file. Here is a brief description of some of the fields in the stat structure:

```
struct stat {
    dev_t    st_dev;    // Device ID
    ino_t    st_ino;    // Inode number
    mode_t   st_mode;   // File type and mode
    nlink_t  st_nlink;  // Number of hard links
```

```
    uid_t    st_uid;    // User ID of owner
    gid_t    st_gid;    // Group ID of owner
    dev_t    st_rdev;   // Device ID (if special file)
    off_t    st_size;   // Total size, in bytes
    blksize_t st_blksize; // Block size for filesystem I/O
    blkcnt_t st_blocks; // Number of 512B blocks allocated
    time_t    st_atime;  // Time of last access
    time_t    st_mtime;  // Time of last modification
    time_t    st_ctime;  // Time of last status change
};

#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>

void print_stat_info(const char *label, const struct stat *fileStat) {
    printf("%s:\n", label);
    printf("File size: %ld bytes\n", fileStat->st_size);
    printf("Number of links: %lu\n", fileStat->st_nlink);
    printf("File inode: %lu\n", fileStat->st_ino);
    printf("File mode: %o\n", fileStat->st_mode);
    printf("File UID: %d\n", fileStat->st_uid);
    printf("File GID: %d\n", fileStat->st_gid);
    printf("Last access time: %ld\n", fileStat->st_atime);
    printf("Last modification time: %ld\n", fileStat->st_mtime);
    printf("Last status change time: %ld\n", fileStat->st_ctime);
    printf("\n");
}

int main() {
    const char *filename = "example.txt";
    const char *symlinkname = "example_symlink";
    // Using stat()
```

```

struct stat statBuf;
if (stat(filename, &statBuf) == -1) {
    perror("stat");
    return 1;  }

print_stat_info("stat()", &statBuf);

// Using fstat()

int fd = open(filename, O_RDONLY);
if (fd == -1) {
    perror("open");
    return 1;  }

struct stat fstatBuf;
if (fstat(fd, &fstatBuf) == -1) {
    perror("fstat");
    close(fd);
    return 1;  }

print_stat_info("fstat()", &fstatBuf);
close(fd);

// Using lstat()

struct stat lstatBuf;
if (lstat(symlinkname, &lstatBuf) == -1) {
    perror("lstat");
    return 1;  }

print_stat_info("lstat()", &lstatBuf);

if (S_ISLNK(lstatBuf.st_mode)) {
    printf("%s is a symbolic link\n", symlinkname);
} else {
    printf("%s is not a symbolic link\n", symlinkname);
}  return 0;

```

**chmod():**The chmod command is used to change the permissions of a file or directory in Unix-like operating systems.

The command `chmod("newfiles", 00444);` sets the permissions of the directory or file named `newfiles` to 444.

Here's what 00444 means in terms of permissions:

- The leading 0 indicates that this is an octal number.
- The 444 indicates the permission settings for the owner, group, and others.

In octal notation: 4 corresponds to read-only permission.

**chown():** The `chown` function in Unix-like operating systems is used to change the ownership of a file or directory. It allows you to specify a new owner and optionally a new group for a given file or directory

```
#include <unistd.h>
int chown(const char *pathname, uid_t owner, gid_t group);
```

#### Parameters

- `const char *pathname`: The path to the file or directory whose ownership you want to change.
- `uid_t owner`: The user ID (UID) of the new owner. If this is -1, the owner is not changed.
- `gid_t group`: The group ID (GID) of the new group. If this is -1, the group is not changed.

#### Return Value

- Returns 0 on success.
- Returns -1 on failure, and `errno` is set to indicate the error.

#### Example

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

int main() {
    const char *path = "example.txt";
    uid_t new_owner = 1001; // New owner UID
    gid_t new_group = 1001; // New group GID
    if (chown(path, new_owner, new_group) == -1) {
        perror("chown");
        return 1;
    }
    printf("Ownership of '%s' changed to UID %d and GID %d\n", path, new_owner, new_group);
    return 0;
}
```

## Unit-IV

### Topic 14: Directory System calls

**Opendir()** : The opendir function opens a directory stream corresponding to the directory name, and returns a pointer to the directory stream.

```
#include <dirent.h>
```

```
DIR *opendir(const char *name);
```

**Parameters:**

- const char \*name: A pointer to a string containing the name of the directory to be opened.

**Returns:**

- A pointer to a DIR structure on success.
- NULL on failure, and errno is set to indicate the error.

**Example:**

```
#include <dirent.h>
```

```
#include <errno.h>
```

```
DIR *dir = opendir("some_directory");
```

```
if (dir == NULL) {
```

```
    // Handle error
```

```
} else {
```

```
    // Directory opened successfully
```

```
}
```

**readdir()**: The readdir function reads the next directory entry from the directory stream.

```
#include <dirent.h>
```

```
struct dirent *readdir(DIR *dirp);
```

**Parameters:**

- DIR \*dirp: A pointer to the directory stream (opened by opendir).

**Returns:**

- A pointer to a struct dirent representing the next directory entry.
- NULL at the end of the directory stream or on error (check errno to distinguish).

**Example:**

```
#include <dirent.h>
```

```

#include <stdio.h>
#include <errno.h>
DIR *dir = opendir("some_directory");
if (dir != NULL) {
    struct dirent *entry;
    while ((entry = readdir(dir)) != NULL) {
        printf("%s\n", entry->d_name);
    }
    closedir(dir);
} else {
    // Handle error
}

```

**closedir()** : The closedir function closes the directory stream.

```

#include <dirent.h>
int closedir(DIR *dirp);

```

**Parameters:**

- DIR \*dirp: A pointer to the directory stream to be closed.

**Returns:**

- 0 on success.
- -1 on failure, and errno is set to indicate the error.

**Example**

```

#include <dirent.h>
#include <errno.h>
DIR *dir = opendir("some_directory");
if (dir != NULL) {
    // Read entries
    closedir(dir);
} else {
    // Handle error
}

```

**mkdir():** The mkdir function creates a new directory with the specified name and permissions.

**int mkdir(char\* name, int mode);**

**Parameters:**

- char\* name: A pointer to a string that specifies the name of the directory to be created.
- int mode: An integer that specifies the permissions for the new directory, usually given in octal format (e.g., 0755).

**Returns:**

- 0 on success.
- -1 on failure, and errno is set to indicate the error.

**Example:**

```
#include <sys/stat.h>
#include <errno.h>

int result = mkdir("new_directory", 0755);
if (result == 0) {
    // Directory created successfully
} else {
    // Handle error
}
```

**rmdir :** The rmdir function removes an empty directory with the specified name.

**int rmdir(char\* name);**

**Parameters:**

- char\* name: A pointer to a string that specifies the name of the directory to be removed.

**Returns:**

- 0 on success.
- -1 on failure, and errno is set to indicate the error.

**Example:**

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

int result = rmdir("new_directory");
if (result == 0) {
    // Directory removed successfully
} else {
```

```
// Handle error  
}
```

***The directory must be empty before it can be removed using rmdir. If the directory is not empty or does not exist, rmdir will fail and set errno appropriately.***

### **umask**

The umask (short for "user file creation mask") is a function and command in Unix-like operating systems that sets the default permissions for newly created files and directories. The umask value is subtracted from the default permissions to determine the actual permissions assigned when files and directories are created.

#### **Understanding umask**

The default permissions before applying the umask are:

- **Directories:** 777 (rwxrwxrwx)
- **Files:** 666 (rw-rw-rw-)

The umask value is used to mask out certain permission bits so that they are not set when new files and directories are created.

#### **Setting and Viewing umask**

You can view the current umask value by running the umask command without arguments:

```
$ umask
```

```
output : 0022.
```

You can set the umask value by running the umask command with an argument:

```
$ umask 0022
```

#### **How umask Affects Permissions**

The umask value is subtracted from the default permissions to determine the actual permissions:

If the umask is 0022:

##### **Directory**

Default permissions: 777

umask: 0022

Resulting permissions:  $777 - 022 = 755$

##### **Files**

Default permissions: 666

umask: 0022

Resulting permissions: 666 - 022 = 644

```
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
    // Set umask to 0022
    umask(0022);

    // Create a new file with default permissions
    int fd = open("example.txt", O_CREAT | O_WRONLY, 0666);
    if (fd == -1) {
        perror("open");
        return 1;
    }

    // Write to the file
    write(fd, "Hello, World!\n", 14);

    // Close the file
    close(fd);

    // Check the permissions of the created file
    printf("File 'example.txt' created with umask 0022\n");
    return 0;
}
```

## Unit V

### Topic-1 : Mass Storage Structure

Systems designed to store enormous volumes of data are referred to as mass storage devices. The basic idea of Mass Storage is to create a Data Backup or Data Recovery System.

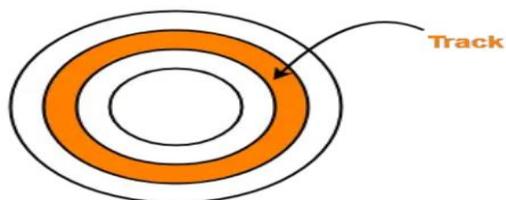
The Mass Storage Structure Devices are:

1. Magnetic Disks
2. Solid State Disks
3. Magnetic Tapes

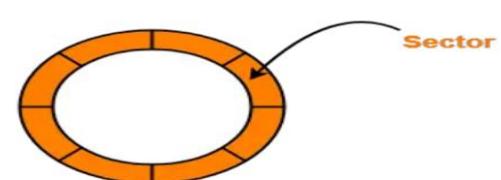
#### Magnetic Disk

Magnetic disk is a storage device that is used to write, rewrite and access data. It uses a magnetization process.

- The entire disk is divided into **platters**.
- Each platter consists of concentric circles called as **tracks**.
- These tracks are further divided into **sectors** which are the smallest divisions in the disk.

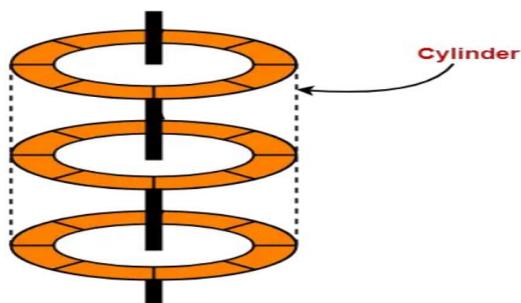


Disk divided into tracks

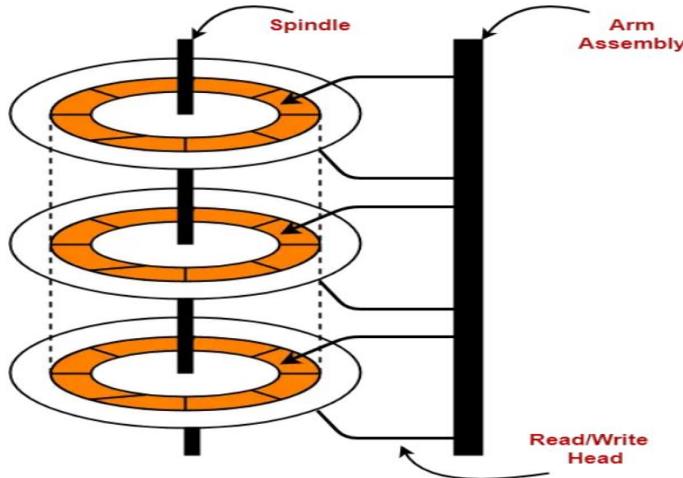


Track divided into sectors

- A **cylinder** is formed by combining the tracks at a given radius of a disk pack.



- There exists a mechanical arm called as **Read / Write head**.
- It is used to read from and write to the disk.
- Head has to reach at a particular track and then wait for the rotation of the platter.
- The rotation causes the required sector of the track to come under the head.
- Each platter has 2 surfaces- top and bottom and both the surfaces are used to store the data.
- Each surface has its own read / write head.



The access time of a record on a disk includes three components such as seek time, latency time, and data transfer time.

**Seek time** – The time required to arrange the read/write head at the desired track is called seek time. For example, suppose that the read/write head is on track 2 and the record to be read is on track 5, then the read/write head must move from track 2 to track 5. The average seeks time on a modern disk is 8 to 12 ms.

**Rotational delay or latency time** – The time required to position the read/write head on a specific sector when the head has already been placed on the desired track is called rotational delay. The rotational delay is based on the speed of rotation of the disk.

**Transfer Time:** Transfer time is the time to transfer the data. It depends on the rotating speed of the disk and number of bytes to be transferred.

$$\text{Disk Access Time} = \text{Seek Time} + \text{Rotational Latency} + \text{Transfer Time}$$

### Solid-State Disks – New

- Solid-state drive (SSD) is a solid-state storage device that uses integrated circuit assemblies as memory to store data. SSD is also known as a solid-state disk although SSDs do not have physical disks. There are no moving mechanical components in SSD. This makes them different from conventional electromechanical drives such as Hard Disk Drives (HDDs) or floppy disks, which contain movable read/write heads and spinning disks. SSDs are typically more resistant to physical shock, run silently, and have quicker access time, and lower latency compared to electromechanical devices. It is a type of **non-volatile** memory that retains data even when power is lost. SSDs may be constructed from random-access memory (RAM) for applications requiring fast access but not necessarily data persistence after power loss. Batteries can be employed as integrated power sources in such devices to retain data for a certain amount of time after external power is lost.

## Magnetic Tapes

Prior to the advent of hard disk drives, magnetic tapes were frequently utilized for secondary storage; today, they are mostly used for backups. It might take a while to get to a specific location on a magnetic tape, but once reading or writing starts, access rates are on par with disk drives.

## Disk Structure

Tape drive capacities may be anywhere from 20 and 200 GB, and compression can increase that capacity by double. Modern disk drives use logical block addressing (LBA) to manage data storage. This system treats the disk as a large one-dimensional array of logical blocks, which are the smallest units of data transfer. Typically, a logical block is 512 bytes, but other sizes like 1,024 bytes can also be used. Logical blocks are sequentially mapped onto the disk's physical sectors. Here's the typical mapping process:

1. **Start at Sector 0:** The first sector of the first track on the outermost cylinder.
2. **Proceed Sequentially:** Mapping continues through the sectors of that track, then through other tracks in the cylinder, and finally through other cylinders from the outermost to the innermost.

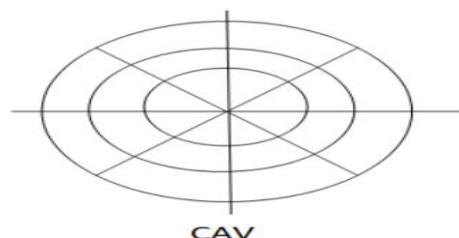
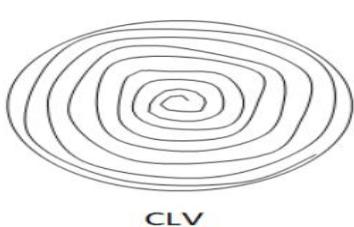
Using this mapping, a logical block number can theoretically be translated into a physical disk address consisting of:

- **Cylinder Number**
- **Track Number within that Cylinder**
- **Sector Number within that Track**

The disk structure (architecture) can be of two types – Constant Linear Velocity (CLV) , Constant Angular Velocity (CAV)

CLV – The density of bits per track is uniform. The farther a track is from the center of the disk, the greater its length, so the more sectors it can hold. As we move from outer zones to inner zones, the number of sectors per track decreases. This architecture is used in CD-ROM and DVD-ROM.

CAV – There is same number of sectors in each track. The sectors are densely packed in the inner tracks. The density of bits decreases from inner tracks to outer tracks to keep the data rate constant.



## **Disk Attachment**

Disk attachment refers to the process of physically connecting a storage device, such as a hard disk drive or solid-state drive, to a computer system. The purpose of disk attachment is to enable the computer system to read and write data to the storage device.

### ***Host-Attached Storage***

- Host-attached storage (HAS) is a form of internal computer storage that can be attached to a host computer, such as a PC or server. Host-attached devices are often used for backup purposes and can include tape drives, optical drives, **hard disk drives (HDDs)**, **solid state drives (SSDs)**, **USB flash drives**, and other similar media.
- A common example of host-attached storage is the use of an external USB flash drive for data transfer between computers.
- Host-attached storage is usually very fast, which makes it a popular choice for high-performance applications. It's also relatively cheap compared to network-attached storage (NAS), which is another form of internal computer storage that can be accessed by multiple systems at once.
- Host-attached storage systems are often used in data centers because of their high performance and reliability. These devices are usually more expensive than NAS devices, but they offer many benefits over other types of internal computer storage.

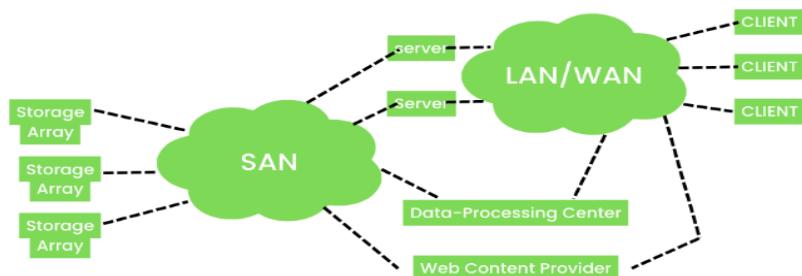
### ***Network-Attached Storage (NAS)***

- A Network-Attached Storage (NAS) is a computer that is connected to a network and shared by multiple users. NAS can also be called a file server, or storage appliance. The term "storage" refers to any device that stores data; this includes hard drives and flashes memory modules.
- NAS has its own processor and memory so it can perform many tasks simultaneously without slowing down other devices on the network; this makes it an ideal solution for businesses who need high availability but don't have enough CPU power available at their desktops or laptops.
- NAS systems are often used as backup solutions for PCs because they allow users to access files from anywhere in the world through remote access services provided by vendors such as **Symantec Remote Access Server (RAS)**.



### **Storage Area Network (SAN)**

- Storage area networks (SAN) are a network of storage devices that can be used to store and retrieve data from a shared central repository. A SAN is usually used for large data storage and retrieval, which requires high availability, scalability, reliability, and performance. The most common type of SAN uses fiber channel adapters to connect the host with its disk arrays.
- A block-based storage protocol like **Fibre Channel Protocol (FCP)** allows multiple devices within the same fabric to communicate with each other in order to share resources such as disks or tape drives. In contrast to other protocols such as **Network File System (NFS)**, FCP does not require any special software drivers on either end because all communication takes place using standard protocols like TCP/IP.
- Another common type of SAN uses a network called **InfiniBand (IB)**, which is a high-speed serial interconnect that provides better performance than traditional Ethernet networks. When used in conjunction with FCP, IB allows data to be transferred faster and more efficiently between two computers.
- A SAN can be implemented in two ways: as an external or internal device. An external SAN is used to connect storage devices to a network, while an internal SAN is integrated into the operating system of a computer. A disk is basically a device that communicates with the host to fetch data or store data in it with the help of I/O devices.



## Unit-V

### Topic 2: Disk Scheduling

Disc scheduling is an important process in operating systems that determines the order in which disk access requests are serviced. The objective of disc scheduling is to minimize the time it takes to access data on the disk and to minimize the time it takes to complete a disk access request.

#### **Various Disk Scheduling Algorithms**



#### **FCFS Disk Scheduling Algorithm**

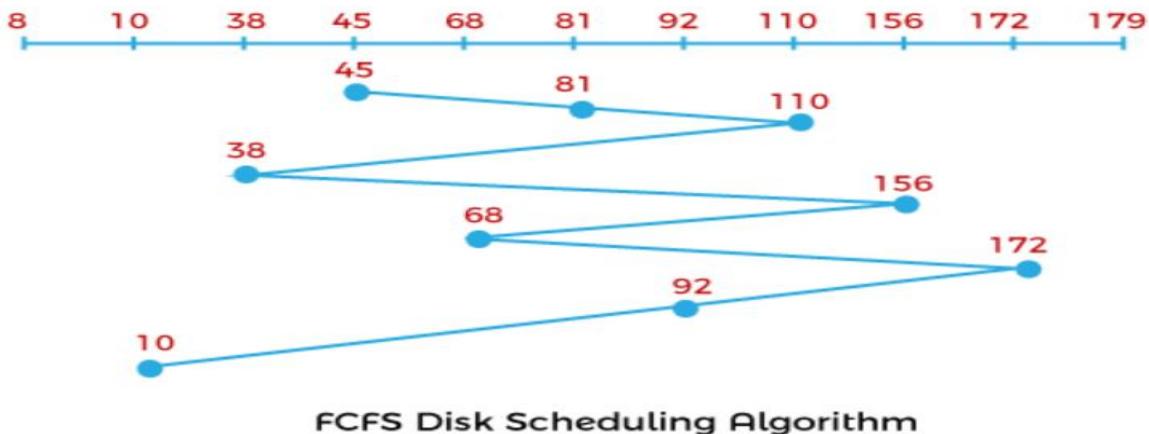
FCFS stands for **First-Come-First-Serve**. It is a very easy algorithm among the all-disk scheduling algorithms. In this scheduling algorithm, the process which requests the processor first receives the processor allocation first. It is managed with a FIFO queue.

#### **Example:**

Let's take a disk with **180** tracks (**0-179**) and the disk queue having input/output requests in the following order: **81, 110, 38, 156, 68, 172, 92, 10**. The initial head position of the Read/Write head is **45**. Find the total number of track movements of the Read/Write head using the FCFS algorithm.

The initial head point is **45**,

### Solution:



Total head movements=  $(81-45) + (110-81) + (110-38) + (156-38) + (156-68) + (172-68) + (172-92) + (92-10)$

### Advantages:

Implementation is easy.

No chance of starvation.

### Disadvantages:

'Seek time' increases.

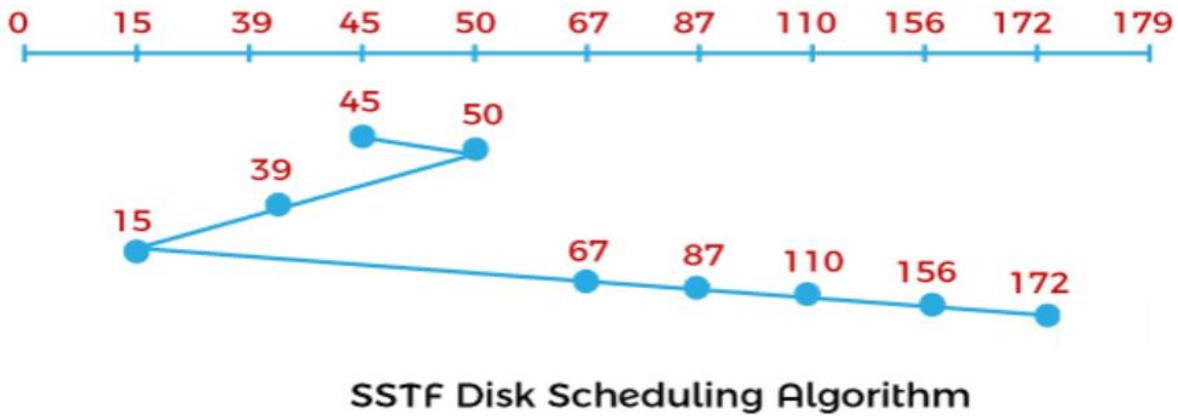
Not so efficient.

### **SSTF Disk Scheduling Algorithm**

SSTF stands for **Shortest Seek Time First**, and it serves the request that is closest to the current position of the head. The direction of the head pointer is quite important in this algorithm. When a tie happens between requests, the head will serve the request in its current direction. In comparison to the FCFS, the SSTF algorithm is very efficient in terms of the total seek time.

### Example:

Let's take an example to understand the SSTF Disk Scheduling Algorithm. Let's take a disk with **180 tracks (0-179)** and the disk queue having input/output requests in the following order: **87, 110, 50, 172, 67, 156, 39, 15**. The initial head position of the Read/Write head is **45** and will move in the left-hand side direction. Find the total number of track movements of the Read/Write head using the SSTF algorithm.



Initial head point is **45**,

Total head movements=  $(50-45) + (50-39) + (39-15) + (67-15) + (87-67) + (110-87) + (156-110) + (172-156)$

#### **Advantages:**

1. In this algorithm, disk response time is less.
2. More efficient than FCFS.

#### **Disadvantages:**

1. Less speed of algorithm execution.
2. Starvation can be seen.

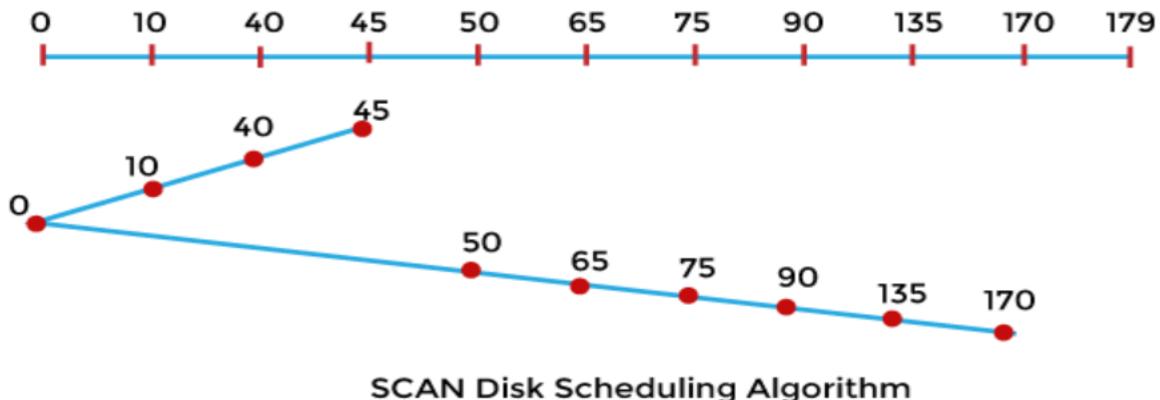
### **SCAN Disk Scheduling Algorithm**

It is also known as the **Elevator algorithm**. In this algorithm, the head may move in both directions, i.e., the disk arm begins to move from one end of the disk to the other end and servicing all requests until it reaches the other end of the disk. After reaching the other end, the head position direction is changed and further continues servicing the requests till the end of the disk.

#### **Example:**

Let's take a disk with **180** tracks (**0-179**) and the disk queue having input/output requests in the following order: **75, 90, 40, 135, 50, 170, 65, 10**. The initial head position of the Read/Write head is **45** and will move on the left-hand side. Find the total number of track movements of the Read/Write head using the SCAN algorithm.

**Solution:**



Total head movements,

Initial head point is 45,

$$\begin{aligned} &= (45-40) + (40-10) + (10-0) + (50-0) + (65-50) + (75-65) + (90-75) + (135-90) + (170-135) \\ &= 5 + 30 + 10 + 50 + 15 + 10 + 15 + 45 + 35 = 215 \end{aligned}$$

### Advantages

1. In the SCAN disk scheduling algorithm, low variance happens in the waiting time and response time.
2. The starvation is avoided in this disk scheduling algorithm.

### Disadvantages

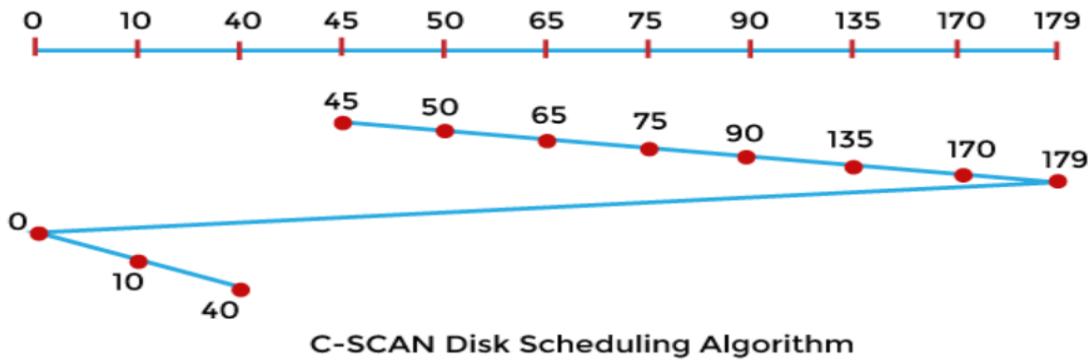
1. If no requests remain to be serviced, the head moves till the end of the disk.

### C-SCAN Disk Scheduling

It is also known as the **Circular Elevator algorithm** or **Circular SCAN**. It is an improved version of the SCAN disk scheduling algorithm. In this algorithm, the head works for requests in a single direction, i.e., it scans all the way to the end of a direction and then jumps to another end and services the requests in the same direction.

### Example:

Let's take a disk with **180** tracks (**0-179**) and the disk queue having input/output requests in the following order: **75, 90, 40, 135, 50, 170, 65, 10**. The initial head position of the Read/Write head is **45** and will move on the right-hand side. Find the total number of track movements of the Read/Write head using the C-SCAN algorithm.



Total head movements,

The initial head point is 45,

$$= (50-45) + (65-50) + (75-65) + (90-75) + (135-90) + (179-135) + (179-170) + (179-0) + (40-10)$$

#### **Advantages**

1. It gives a uniform waiting time.
2. It gives a better response time.
3. The head travels from one end to the other disks end and serves all requests along the way.
4. The C-SCAN algorithm is the improved version of the SCAN scheduling algorithm.

#### **Disadvantages**

1. If no requests remain to be serviced, the head will travel to the end of the disk.
2. It generates more search movements than the SCAN algorithm.

#### **LOOK Disk Scheduling**

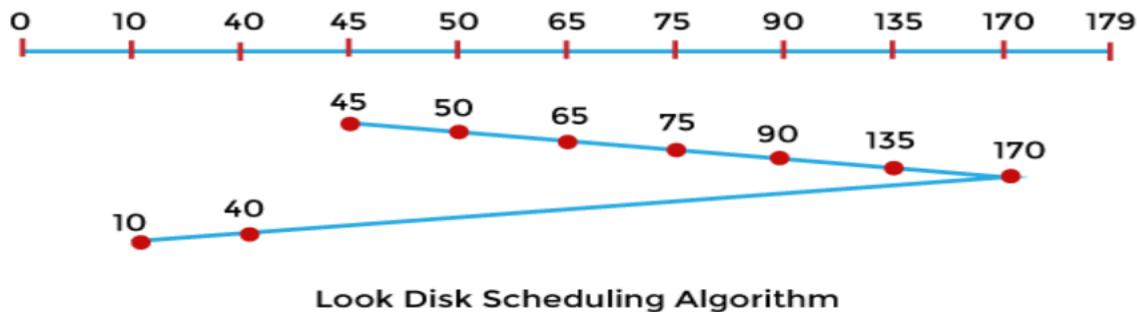
It is the more advanced version of the SCAN disk scheduling algorithm. In this algorithm, the head begins at one end of the disk and works its way to the other end, and serving all requests along the way. When the head reaches the end of one end's last request, it changes direction and returns to the first request, servicing all requests in between. Unlike SCAN, instead of going to the last track, this head goes to the last request and then changes direction.

#### **Example:**

Let's take an example to understand the LOOK Disk Scheduling Algorithm. Let's take a disk with **180 tracks (0-179)** and the disk queue having input/output requests in the following order: **75, 90, 40, 135, 50, 170, 65, 10**. The initial head position of the Read/Write head is 45 and

would move on the right-hand side. Find the total number of track movements of the Read/Write head using the LOOK disk scheduling algorithm.

**Solution:**



Total head movements,

The initial head point is 45,

$$\begin{aligned}
 &= (50-45) + (65-50) + (75-65) + (90-75) + (135-90) + (170-135) + (170-40) + (40-10) \\
 &= 5 + 15 + 10 + 15 + 45 + 35 + 130 + 30 = 285
 \end{aligned}$$

**Advantages**

1. It provides better performance in comparison to the SCAN algorithm.
2. The LOOK scheduling algorithm avoids starvation.
3. The head will not move to the end of the disk if no more requests are fulfilled.
4. Waiting time and response time have a low variance.

There **Disadvantages**

1. is an overhead of finding the final requests.

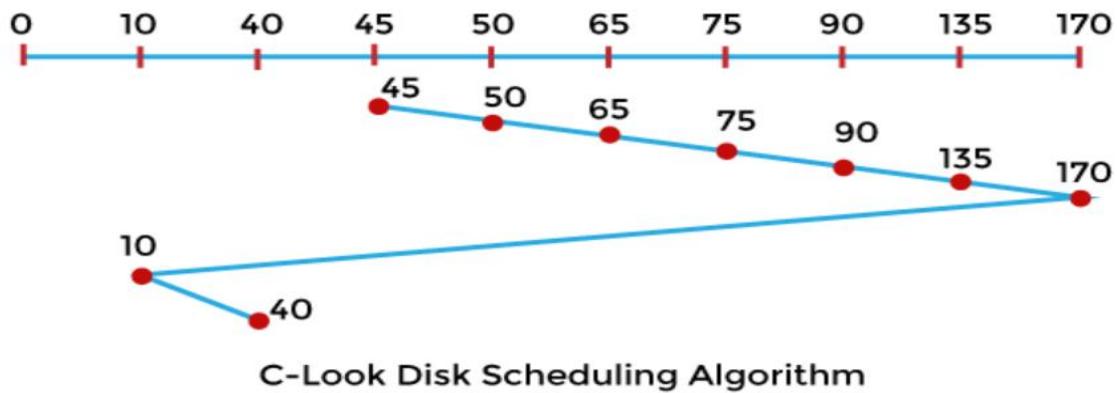
**C-LOOK Disk Scheduling Algorithm**

It is a combination of the LOOK and SCAN disk scheduling algorithms. In this disk scheduling algorithm, the head begins from the initial request to the last request in the other direction and serves all requests in between. The head jumps in the other direction after finishing the last request at one end and proceeds towards the remaining requests, completing them in the same direction as previously. Unlike LOOK, it only responds to requests in one direction.

**Example:**

Let's take an example to understand the **C-LOOK** Disk Scheduling Algorithm. Let's take a disk with **180 tracks (0-179)** and the disk queue having input/output requests in the following order: **75, 90, 40, 135, 50, 170, 65, 10**. The initial head position of the Read/Write head is 45 and

would move on the right-hand side. Find the total number of track movements of the Read/Write head using the C-LOOK disk scheduling algorithm.



Total head movements,

The initial head point is 45,

$$= (50-45) + (65-50) + (75-65) + (90-75) + (135-90) + (170-135) + (170-10) + (40-10)$$

#### Advantages

1. It provides better performance compared to the LOOK disk scheduling algorithm.
2. The starvation is avoided in the C-LOOK disk scheduling algorithm.
3. If no requests are to be served, the head doesn't have to go all the way to the end of the disk in the C-LOOK disk scheduling algorithm.
4. In C-LOOK, there is minimal waiting time for cylinders that are only visited by the head.
5. Waiting time and response time have a low variance.

#### Disadvantages

1. The overhead of finding the end requests is present in C-LOOK.

#### Disk Management

The operating system is responsible for several other aspects of disk management.

Disk formatting 2. Booting from disk 3. Bad-block recovery.

#### Disk Formatting

Disk formatting is of two types.

Physical formatting or low level formatting. b) Logical formatting.

#### Physical formatting :

- Disk must be formatted before storing a data.

- Disk must be divided into sectors that the disk controller can read/write.
- Low level formatting fills the disk with a special data structure for each sector.
- Data structure consists of three fields: header, data area and trailer.
- Header and trailer contain information used by the disk controller.
- Sector number and Error Correcting Codes (ECC) contained in the header and trailer.
- For writing data to the sector - ECC is updated.
- For reading data from the sector - ECC is recalculated.
- If there is mismatch in stored and calculated value then data area is corrupted.

### **Logical formatting:**

- After disk is partitioned, logical formatting used.
- Operating system stores the initial file system data structures onto the disk.

### **Boot Blocks**

- The process of starting or restarting a computer system or any other computing device is called booting. Block which contains all the data and instructions required for starting the booting process of the system is referred to as boot block. It is also known as boot sector, as it is a sector in the memory device which contains all the instructions required to boot the system.
- Boot block generally resides in the first sector of the data storage device like hard disk and is designed in a standard format so that the BIOS (Basic Input Output System) can understand and execute it.

### *Components of a Boot Block*

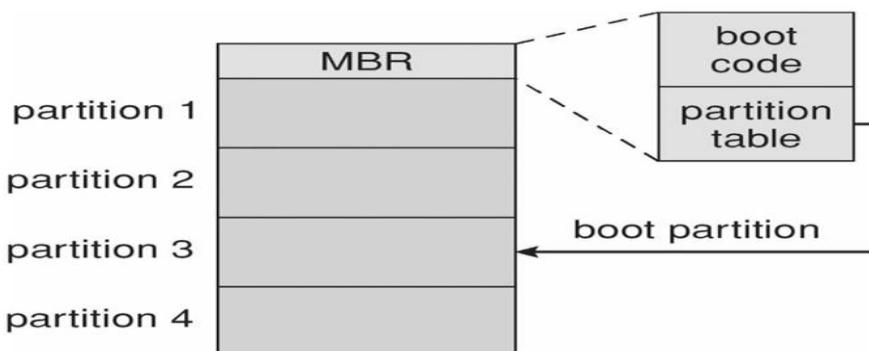


Fig: Booting from disk in Windows 2000.

The following are the important components of the boot block –

- **Master Boot Record (MBR)** – The master boot record (MBR) is a first part of a storage device that contains the boot block, boot code, partition table, and other required data

and instructions. In a computer system, the master boot record is an essential part that helps to understand how the storage device is organized and which partition is required to boot.

- **Bootloader** – Bootloader is a computer program which is responsible for starting the system and loading the operating system into the main memory. It executes all the essential steps required to initiate the booting process.
- **Boot Code** – Boot code, also called bootstrap code, is another important component of boot block. Bootstrap code consists of all the essential instructions written in low-level language like machine language or assembly language. Bootstrap code performs some important functions, including configuration of system components, initialization of hardware parts, loading of operating system into the main memory, etc.
- **Partition Table** – Partition table is another important component of the boot block. It is basically a data table that contains information about the different partition of the data storage device. It helps in identifying the active partition on the disk from which the operating system is to be loaded into the main memory.

### ***Bad Blocks***

- A storage region or sector of a data storage device like hard disk drive, flash drive, optical disk, etc. that cannot be used for storage and retrieval of data due to permanent damage is referred to as a bad block. Sometimes, a **bad block** is also known as a **bad sector**.

Depending on the disk and controller in use, these blocks are handled in a variety of ways;

**Method 1: “Handled manually:** If blocks go bad during normal operation, a special program must be run manually to search for the bad blocks and to lock them . Data that resided on the bad blocks usually are lost.

**Method 2: “sector sparing or forwarding”** The controller maintains a list of bad blocks on the disk. Then the controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as sector sparing or forwarding.

**Method 3: “sector slipping”** For an example, suppose that logical block 17 becomes defective, and the first available spare follows sector 202. Then, sector slipping would remap all the sectors from 17 to 202, moving them all down one spot. That is, sector 202 would be copied into the spare, then sector 201 into 202, and then 200 into 201, and so on, until sector 18 is copied into sector 19. Slipping the sectors in this way frees up the space of sector 18, so sector 17 can be mapped to it.

## Unit V

### Topic 3: Protection

Protection is especially important in a multiuser environment when multiple users use computer resources such as CPU, memory, etc. It is the operating system's responsibility to offer a mechanism that protects each process from other processes.

#### ***Goals of Protection***

The primary goals of protection in operating systems are to ensure the confidentiality, integrity, and availability of system resources and data. Here are the key goals in detail:

- **Confidentiality:** Ensure that only authorized users and processes can access sensitive information.
- **Integrity:** Ensure that data is not altered or tampered with by unauthorized users or processes.
- **Availability:** Ensure that system resources (CPU, memory, disk space, etc.) are available for authorized users and processes when needed.
- **Controlled Access:** Allow the specification of detailed access controls to system resources.
- **Isolation:** Ensure that processes run in isolated environments to prevent them from interfering with each other.

#### ***Principles of Protection***

The principles of protection in operating systems are foundational guidelines that help design and implement security mechanisms to safeguard system resources and data. Here are the key principles:

**Least Privilege:** Each user or process should have the minimum level of access rights necessary to perform their tasks. Assign minimal permissions and escalate privileges only when required, reducing the risk of unauthorized access or damage.

**Separation of Duties:** Divide responsibilities among multiple users or processes to prevent any single entity from having excessive control. Ensure that critical tasks require collaboration or approval from multiple parties, reducing the risk of insider threats and errors.

**Economy of Mechanism:** Security mechanisms should be as simple and straightforward as possible. Simplify design and implementation to reduce complexity, making it easier to understand, verify, and manage.

**Complete Mediation:** Every access to a resource must be checked for proper authorization. Ensure that access control checks are performed every time a resource is accessed, preventing bypassing of security checks.

**Separation of Privilege:** system should not grant permission based on a single condition.: Require multiple conditions to be met before granting access, such as two-factor authentication or multi-party approval processes.

**Least Common Mechanism:** Minimize the sharing of mechanisms among users to prevent indirect attacks. Reduce shared resources and interfaces, isolating users and processes as much as possible.

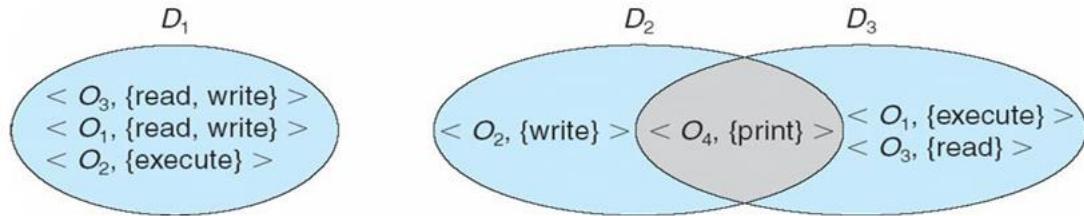
### **Domian of Protection**

- A computer system is a collection of processes and objects. Objects are both hardware objects (such as the CPU, memory segments, printers, disks, and tape drives) and software objects (such as files, programs, and semaphores). Each object (resource) has a unique name that differentiates it from all other objects in the system.
- The operations that are possible may depend on the object. For example, a CPU can only be executed on. Memory segments can be read and written, whereas a CD-ROM or DVD-ROM can only be read. Tape drives can be read, written, and rewound. Data files can be created, opened, read, written, closed, and deleted; program files can be read, written, executed, and deleted.
- A process should be allowed to access only those resources for which it has authorization and currently requires to complete process . This requirement is known as need to know principle.

### ***Domain Structure***

- A domain is a set of objects and types of access to these objects. Each domain is an ordered pair of . <objects set,Rights set>

- Example, if domain D has the access right  $\langle \text{File } f, \{\text{read}, \text{write}\} \rangle$ , then all process executing in domain D can both read and write file F, and cannot perform any other operation on that object.
- Domains do not need to be disjoint; they may share access rights. For example, in below figure, we have three domains: D1 D2, and D3. The access right  $\langle O_4, \{\text{print}\} \rangle$  is shared by D2 and D3, it implies that a process executing in either of these two domains can print object O4.



- A domain can be realized in different ways, it can be a user, process or a procedure. ie. each user as a domain, each process as a domain or each procedure as a domain.

## Unit-V

### Topic 4: Access Matrix

The Access Matrix is a security model for a computer system's protection state. It is described as a matrix. An access matrix is used to specify the permissions of each process running in the domain for each object.

#### **Structure of Matrix:**

**Rows:** Rows define the domain processes that perform some operation on each object. The domain is usually a user or process.

**Column:** columns are the objects on which domain processes perform operations. Objects can be files, devices, or any system resources.

**Cells:** cells represent the access right granted to domain processes to operate on objects. Examples are Read, write, delete, execute, etc.

*Let us understand the structure of the access matrix with an example of a system command where different user roles have additional access rights to perform some operations on the system command.*

**Domain:** Domain represents the users with different roles such as manager, employee, and staff.

**Object:** The object represents the system command.

**Access rights:** access rights are the rights granted to different users for performing some operation on commands that are read, executed, and have no access.

Every matrix cell reflects a set of access rights granted to domain processes, i.e., each entry (**i, j**) describes the set of operations that a domain **D<sub>i</sub>** process may invoke on object **O<sub>j</sub>**.

object domain \	F <sub>1</sub>	F <sub>2</sub>	F <sub>3</sub>	printer
D <sub>1</sub>	read		read	
D <sub>2</sub>				print
D <sub>3</sub>		read	execute	
D <sub>4</sub>	read write		read write	

- In the above diagram, there are four domains and four objects—three files (F<sub>1</sub>, F<sub>2</sub>, F<sub>3</sub>) and one printer. A process executing in domain D<sub>1</sub> can read files F<sub>1</sub> and F<sub>3</sub>. A process executing in domain D<sub>4</sub> has the same privileges as one executing in domain D<sub>1</sub>; but in addition, it can also write onto files F<sub>1</sub> and F<sub>3</sub>.

- When a user creates a new object O<sub>j</sub>, the column O<sub>j</sub> is added to the access matrix with the appropriate initialization entries, as dictated by the creator.
- The process executing in one domain can be switched to another domain. When we switch a process from one domain to another, we are executing an operation (switch) on an object (the domain).

object domain \ object domain	$F_1$	$F_2$	$F_3$	laser printer	$D_1$	$D_2$	$D_3$	$D_4$
$D_1$	read		read			switch		
$D_2$				print			switch	switch
$D_3$		read	execute					
$D_4$	read write		read write		switch			

- Domain switching from domain D<sub>i</sub> to domain D<sub>j</sub> is allowed if and only if the access right switch access(i,j). Thus, in the given figure, a process executing in domain D<sub>2</sub> can switch to domain D<sub>3</sub> or to domain D<sub>4</sub>. A process in domain D<sub>4</sub> can switch to D<sub>1</sub>, and one in domain D<sub>1</sub> can switch to domain D<sub>2</sub>.

*Allowing controlled change in the contents of the access-matrix entries requires three additional operations: copy, owner, and control.*

object domain	$F_1$	$F_2$	$F_3$
$D_1$	execute		write*
$D_2$	execute	read*	execute
$D_3$	execute		

(a)

object domain	$F_1$	$F_2$	$F_3$
$D_1$	execute		write*
$D_2$	execute	read*	execute
$D_3$	execute	read	

(b)

The ability to copy an access right from one domain (or row) of the access matrix to another is denoted by an asterisk (\*) appended to the access right. The copy right allows the copying of the access right only within the column for which the right is defined. In the below figure, a process executing in domain  $D_2$  can copy the read operation into any entry associated with file  $F_2$ . Hence, the access matrix of figure (a) can be modified to the access matrix shown in figure (b).

This scheme has two variants:

1. A right is copied from  $\text{access}(i,j)$  to  $\text{access}(k,j)$ ; it is then removed from  $\text{access}(i,j)$ . This action is a transfer of a right, rather than a copy.
2. Propagation of the copy right- limited copy. Here, when the right  $R^*$  is copied from  $\text{access}(i,j)$  to  $\text{access}(k,j)$ , only the right  $R$  (not  $R^*$ ) is created. A process executing in domain  $D_k$  cannot further copy the right  $R$ .

We also need a mechanism to allow addition of new rights and removal of some rights. The owner right controls these operations. If  $\text{access}(i,j)$  includes the owner right, then a process executing in domain  $D_i$ , can add and remove any right in any entry in column  $j$ .

For example, in below figure (a), domain  $D_1$  is the owner of  $F_1$ , and thus can add and delete any valid right in column  $F_1$ . Similarly, domain  $D_2$  is the owner of  $F_2$  and  $F_3$  and thus can add and remove any valid right within these two columns. Thus, the access matrix of figure(a) can be modified to the access matrix shown in figure(b) as follows.

object domain \	$F_1$	$F_2$	$F_3$
$D_1$	owner execute		write
$D_2$		read* owner	read* owner write
$D_3$	execute		

(a)

object domain \	$F_1$	$F_2$	$F_3$
$D_1$	owner execute		write
$D_2$		owner read* write*	read* owner write
$D_3$		write	write

(b)

A mechanism is also needed to change the entries in a row. If access(i,j) includes the control right, then a process executing in domain  $D_i$ , can remove any access right from row j. For example, in figure, we include the control right in access(  $D_4$ ). Then, a process executing in domain  $D_4$  can modify domain  $D_3$ .

object domain \	$F_1$	$F_2$	$F_3$	laser printer	$D_1$	$D_2$	$D_3$	$D_4$
$D_1$	read		read			switch		
$D_2$				print			switch	switch control
$D_3$		read	execute					
$D_4$	write		write		switch			

## Implementation Of Access Matrix

Different methods of implementing the access matrix (which is sparse)

- Global Table
- Access Lists for Objects
- Capability Lists for Domains
- Lock-Key Mechanism

## **1. Global Table**

- This is the simplest implementation of access matrix.
- A set of ordered triples  $\langle \text{domain}, \text{object}, \text{rights-set} \rangle$  is maintained in a file. Whenever an operation M is executed on an object  $O_j$ , within domain  $D_i$ , the table is searched for a triple  $\langle D_i, O_j, R_k \rangle$ . If this triple is found, the operation is allowed to continue; otherwise, an exception (or error) condition is raised.

Drawbacks -

The table is usually large and thus cannot be kept in main memory. Additional I/O is needed

## **2. Access Lists for Objects**

- Each column in the access matrix can be implemented as an access list for one object. The empty entries are discarded. The resulting list for each object consists of ordered pairs  $\langle \text{domain}, \text{rights-set} \rangle$ .
- It defines all domains access right for that object. When an operation M is executed on object  $O_j$  in  $D_i$ , search the access list for object  $O_j$ , look for an entry  $\langle D_i, R_k \rangle$  with  $M \in R_k$ . If the entry is found, we allow the operation; if it is not, we check the default set. If M is in the default set, we allow the access. Otherwise, access is denied, and an exception condition occurs. For efficiency, we may check the default set first and then search the access list.

## **3. Capability Lists for Domains**

- A capability list for a domain is a list of objects together with the operations allowed on those objects. An object is often represented by its name or address, called a capability.
- To execute operation M on object  $O_j$ , the process executes the operation M, specifying the capability for object  $O_j$  as a parameter. Simple possession of the capability means that access is allowed.

### **Capability Format**

<b>Object Descriptor</b>	<b>Rights of The Subject</b>
	<b>Read , Write , Execute</b>

Capabilities are distinguished from other data in one of two ways:

1. Each object has a tag to denote its type either as a capability or as accessible data.
2. The address space associated with a program can be split into two parts. One part is

accessible to the program and contains the program's normal data and instructions. The other part, containing the capability list, is accessible only by the operating system.

#### **4. A Lock-Key Mechanism**

- The lock-key scheme is a compromise between access lists and capability lists.
- Each object has a list of unique bit patterns, called locks. Each domain has a list of unique bit patterns, called keys.
- A process executing in a domain can access an object only if that domain has a key that matches one of the locks of the object.