# Concepts of Operating System

## Session 1:

# *Introduction to OS*

- Operating System Is a software that acts as an interface between the user, application software and the computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs conveniently and efficiently.
- An operating system is concerned with the allocation of resources and services, such as memory, processors, devices, and information. The operating system correspondingly includes programs to manage these resources, such as a traffic controller, a scheduler, a memory management module, I/O programs, and a file system.

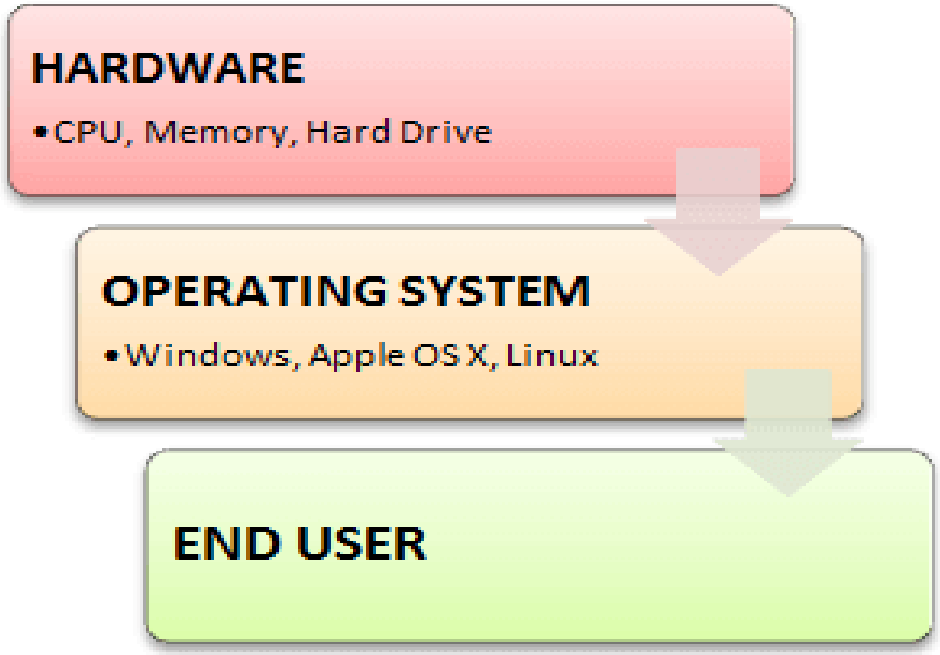
**EXAPMLE OF OPERATING SYSTEM**

1.Ms DOS

2.Windows ( 98, 2000,ME, XP, Vista,7,10,11)

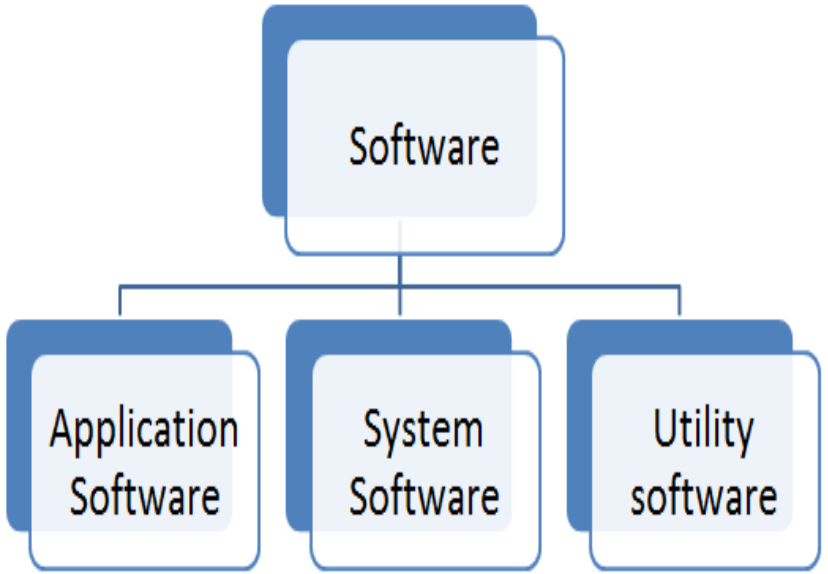
3.Linux

4.Unix



# How Operating System is different from other application software ?

- Computer software is divided into system software and application software.
- System software is the type of software that is used to run hardware devices. Operating systems, language processors and device drivers are few examples for system software.
- Application software is the type of software that is designed to achieve a specific user requirement. Word processors, spreadsheets, web browsers are some application software.
- Apart from these types, there is another software called utility software. This software assists the tasks of the system. Antivirus software, file and disk management tools are some examples of utility software.



# History of Operating System

- **The First Generation (1940's to early 1950's)**
  - No Operating System
  - All programming was done in absolute machine language, often by wiring up plug-boards to control the machine’s basic functions.
- **The Second Generation (1955-1965)**
  - First operating system was introduced in the early 1950's.It was called GMOS
  - Created by General Motors for IBM's machine the 701.
  - Single-stream batch processing systems
- **The Third Generation (1965-1971)**
  - Introduction of multiprogramming
  - Development of Minicomputer
- **The Fourth Generation (1971-1982)**
  - Development of PCs
  - Birth of Windows/MaC OS
- **The Fifth Generation (1982-Present Day)**
  - ULSI technology
  - Development of true artificial intelligence
  - Development of Natural language processing
  - Advancement in Parallel Processing

## Components / functions of OS

- An operating system is a large and complex system that can only be created by partitioning into small parts. These pieces should be a well-defined part of the system, carefully defining inputs, outputs, and functions.
- Although Windows, Mac, UNIX, Linux, and other OS do not have the same structure, most operating systems share similar OS system components, such as file, memory, process, I/O device management.
- The components of an operating system play a key role to make a variety of computer system parts work together. There are the following components of an operating system, such as:
    1. Kernel
    2. Process Management
    3. Interrupt
    4. Memory Management
    5. File Management
    6. I/O Device Management
    7. Network Management
    8. Security Management

## Components / functions of OS Continue…….

- **Kernel**
  - The kernel in the OS provides the basic level of control on all the computer peripherals. In the operating system, the kernel is an essential component that loads firstly and remains within the main memory.
- **Process Management**
  - The process management component is the way to manage the multiple processes running simultaneously on the OS. Process management manages all the running processes and makes sure that all of them run efficiently.
- **Interrupt**
  - In the operating system, interrupts are essential because they give a reliable technique for the OS to communicate & react to their surroundings.
- **Memory Management**
  - The functionality of an OS is nothing but memory management which manages main memory & moves processes backward and forward between disk & main memory during implementation. This tracks each & every memory position; until it is assigned to some process otherwise it is open. It verifies how much memory can be allocated to processes and also makes a decision to know which process will obtain memory at what time.
- **File Management**
  - Following are the features of file management:
  - Creation and deletion of files and directories.
  - Manipulation of files and directories.
  - Maps files onto secondary storage.
  - File backup on a stable storage media.

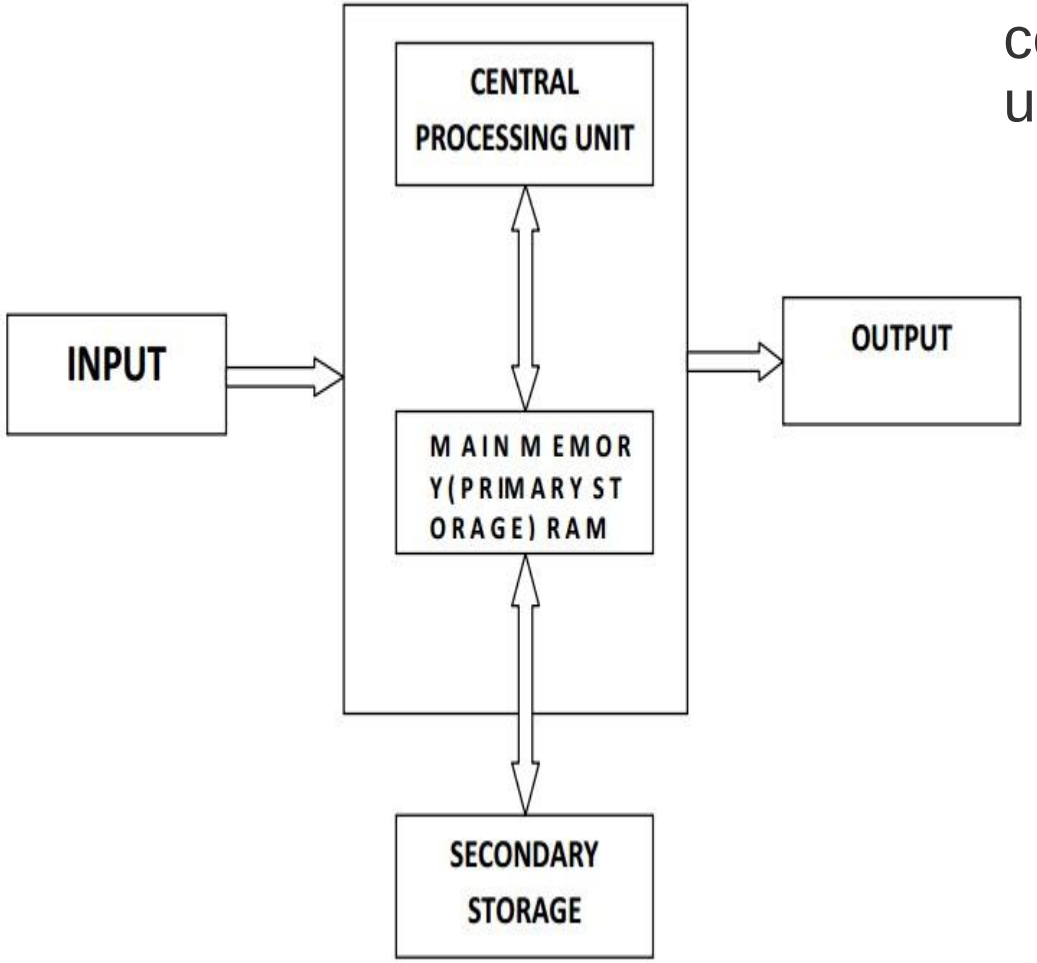
## Components / functions of OS Continue…….

- **I/O Device Management**
  - The operating system is responsible to hide the details of the hardware devices from the user. The I/O device management provides the drivers for a specific hardware device.

- **Network Management**
  - Network management administers and manages computer networks. Network management administers and manages computer networks. Its services include performance management, fault analysis, network provisioning, and service quality management.

- **Security Management**
  - It is necessary to protect the processes from each other’s activities. Security management ensures that the operating files, memory, CPU, and other hardware resources have proper authorization from the OS. No process can do its own I/O

## Basic computer organization




- The main component of basic organization of a computer system is micro-processor(C.P.U), memory unit(MU), and input-output devices.

1. Central processing unit: CPU is a brain of computer. It controls the computer system. It converts data to information.

2. Arithmetic and logic unit: This is a part of CPU. It consists of two units. One is arithmetic unit and another one is logic unit.

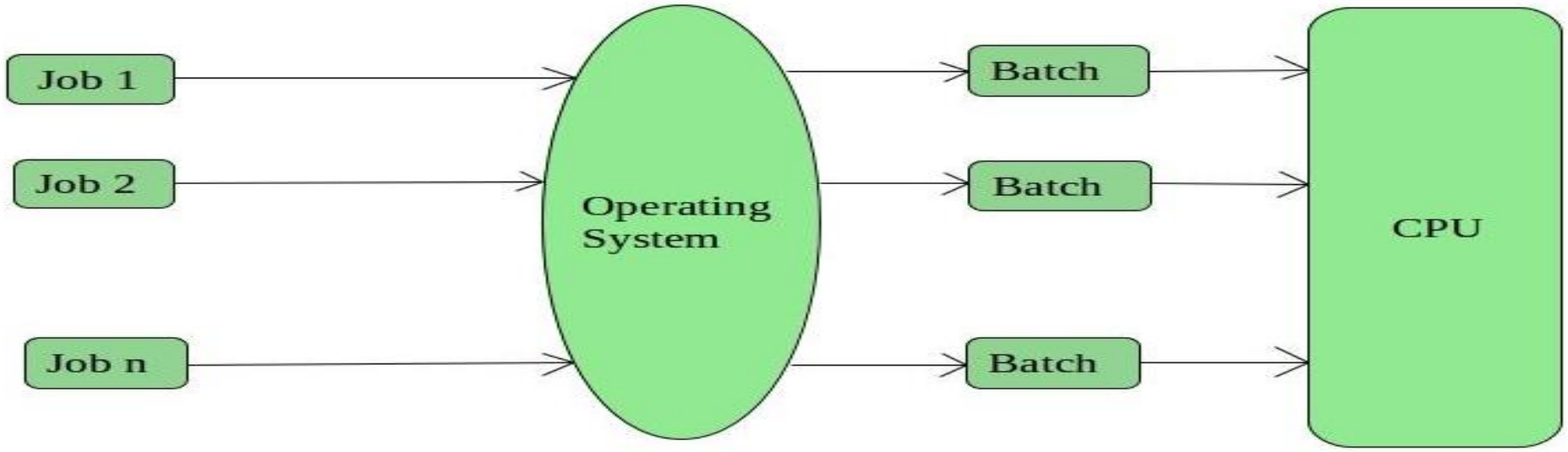
3. Input unit and output unit: This unit controls input and output devices. Input devices are keyboard, mouse etc. and output devices are printer, monitor, plotter, etc.

# Types of operating system

1. Batch Operating System
2. Multiprogramming Operating System
3. Time-Sharing OS
4. Multiprocessing OS
5. Distributed OS
6. Network OS
7. Real Time OS
8. Embedded OS

# 1. Batch Operating System

- The users of this type of operating system does not interact with the computer directly.
- Each user prepares his job on an off-line device like punch cards and submits it to the computer operator
- There is an operator which takes similar jobs having the same requirement and group them int



# 1. Batch Operating System cont..

**Advantages of Batch Operating System:**

- Processors of the batch systems know how long the job would be when it is in queue
- Multiple users can share the batch systems
- The idle time for the batch system is very less
- It is easy to manage large work repeatedly in batch systems

**Disadvantages of Batch Operating System:**

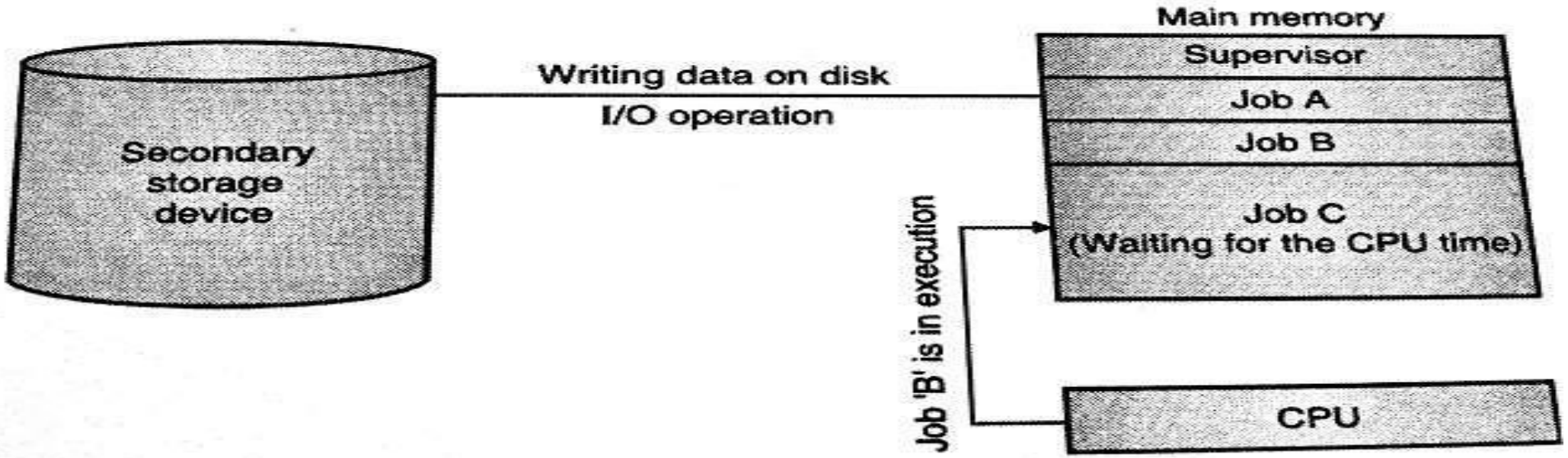
- The computer operators should be well known with batch systems
- Batch systems are hard to debug
- It is sometimes costly
- The other jobs will have to wait for an unknown time if any job fails

**Examples of Batch based Operating System:**

IBM's MVS

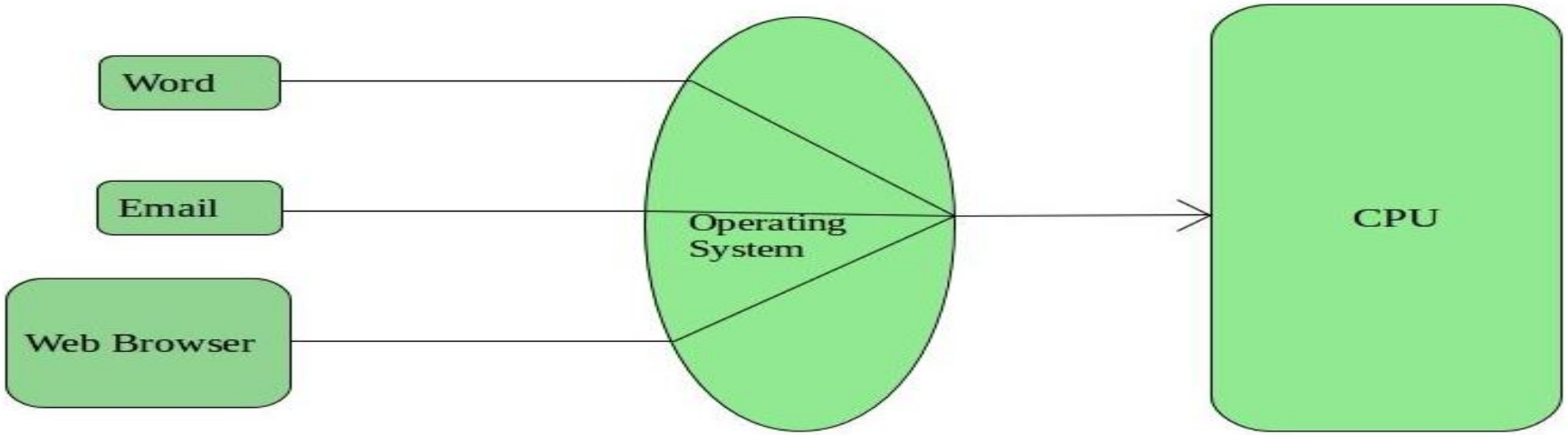
# 2. Multiprogramming Operating System:

- This type of OS is used to execute more than one jobs simultaneously by a single processor.
- It increases CPU utilization by organizing jobs so that the CPU always has one job to execute.
- Multiprogramming operating systems use the mechanism of job scheduling and CPU scheduling.



# 3. Time-Sharing Operating Systems

- Each task is given some time to execute so that all the tasks work smoothly.
- These systems are also known as **Multi-tasking Systems.**
- The task can be from a single user or different users also.
- The time that each task gets to execute is called quantum.
- After this time interval is over OS switches over to the next task.



# 3. Time-Sharing Operating Systems cont..

- **Advantages of Time-Sharing OS:**
  - Each task gets an equal opportunity
  - Fewer chances of duplication of software
  - CPU idle time can be reduced
- **Disadvantages of Time-Sharing OS:**
  - Reliability problem
  - One must have to take care of the security and integrity of user programs and data
- **Examples of Time-Sharing Oss**

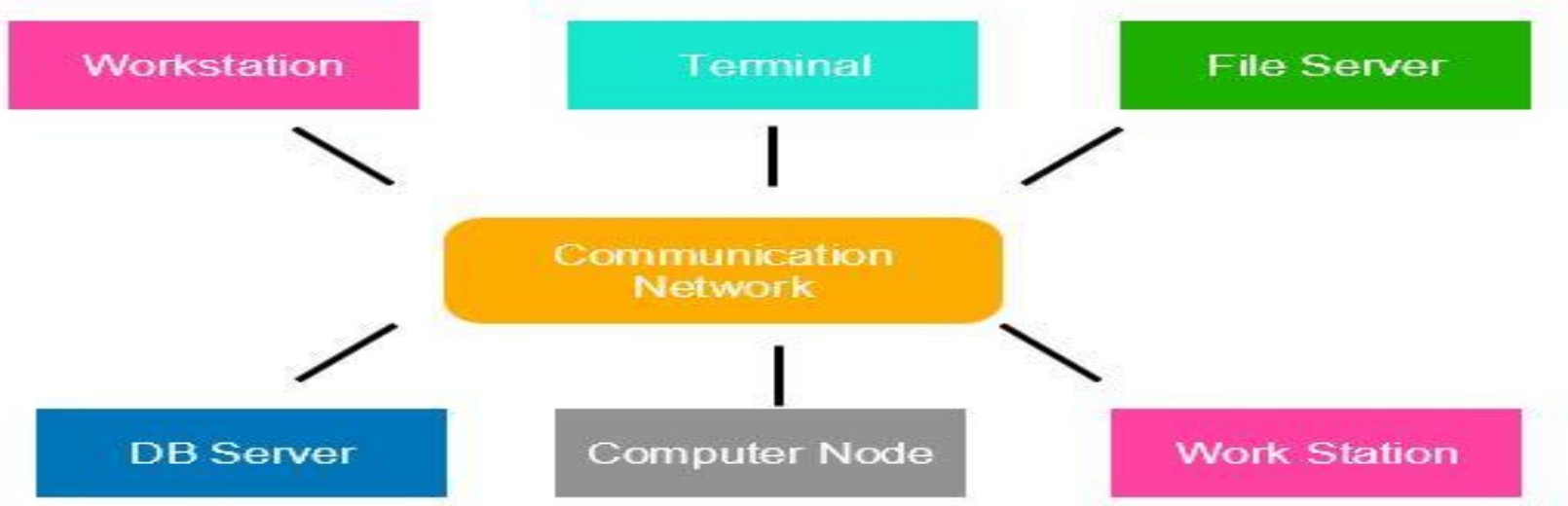
  Multics, Unix, etc.

# 4. Multiprocessor operating systems

- Multiprocessor operating systems are also known as parallel OS or tightly coupled OS.
- Such operating systems have more than one processor in close communication that sharing the computer bus, the clock and sometimes memory and peripheral devices.
- It executes multiple jobs at the same time and makes the processing faster.
- If one processor fails then other processor should retrieve the interrupted process state so execution of process can continue.
- Inter-processes communication mechanism is provided and implemented in hardware.

# 5. Distributed Operating System

- Various autonomous interconnected computers communicate with each other using a shared communication network.
- Independent systems possess their own memory unit and CPU.
- These are referred to as **loosely coupled systems**.
- Examples:- Locus



# 6. Network Operating System

- These systems run on a server and provide the capability to manage data, users, groups, security, applications, and other networking functions.
- These types of operating systems allow shared access of files, printers, security, applications, and other networking functions over a small private network.
- The “ other" computers arc called client computers, and each computer that connects to a network server must be running client software designed to request a specific service.

# 6. Network Operating System

**Advantages of Network Operating System:**

- Highly stable centralized servers
- Security concerns are handled through servers
- New technologies and hardware up-gradation are easily integrated into the system
- Server access is possible remotely from different locations and types of systems

**Disadvantages of Network Operating System:**

- Servers are costly
- User has to depend on a central location for most operations
- Maintenance and updates are required regularly

**Examples of Network Operating System are:**

Microsoft Windows Server 2003/2008/2012, UNIX, Linux, Mac OS X, Novell NetWare, etc.

# 7. Real-Time Operating System

- These types of OSs serve real-time systems.
- The time interval required to process and respond to inputs is very small.
- This time interval is called **response time**.
- **Real-time systems** are used when there are time requirements that are very strict like
  - missile systems,
  - air traffic control systems,
  - robots, etc.

# 8. Embedded Operating System

- An embedded operating system is one that is built into the circuitry of an electronic device.
- Embedded operating systems are now found in automobiles, bar-code scanners, cell phones, medical equipment, and personal digital assistants.
- The most popular embedded operating systems for consumer products, such as PDAs, include the following:
  - Windows XP Embedded
  - Windows CE .NET:- it supports wireless communications, multimedia and Web browsing. It also allows for the use of smaller versions of Microsoft Word, Excel, and Outlook.
  - Palm OS:- It is the standard operating system for Palm-brand PDAs as well as other proprietary handheld devices.
  - Symbian:- OS found in “ smart” cell phones from Nokia and Sony Ericsson

# Popular types of OS

- Desktop Class
  - ❖Windows
  - ❖OS X
  - ❖Unix/Linux
  - ❖Chrome OS
- Server Class
  - ❖Windows Server
  - ❖Mac OS X Server
  - ❖Unix/Linux
- Mobile Class
  - ❖Android
  - ❖iOS
  - ❖Windows Phone

# Kernel

- Kernel is a software code that reside in central core of OS. It has complete control over system.
- When operation system boots, kernel is first part of OS to load in main memory.
- Kernel remains in main memory for entire duration of computer session.
- Kernel performs it’s task like executing processes and handling interrupts in kernel space.
- User performs it’s task in user area of memory.
- This memory separation is made in order to prevent user data and kernel data from interfering with each other.
- Kernel does not interact directly with user, but it interacts using SHELL and other programs and hardware.

# Kernel cont…

- Kernel includes:-

  1. Scheduler: It allocates the Kernel's processing time to various processes.

  2. Supervisor: It grants permission to use computer system resources to each process.

  3. Interrupt handler : It handles all requests from the various hardware devices which compete for kernel services.

  4. Memory manager : allocates space in memory for all users of kernel service.

- kernel provides services for process management, file management, I/O management, memory management.
- System calls are used to provide these type of services.

# System Call and Interrupt

- **System call** is the programmatic way in which a computer program/user application requests a service from the kernel of the operating system on which it is executed.
- Application program is just a user-process. Due to security reasons , user applications are not given access to privileged resources(the ones controlled by OS).
- When they need to **do any I/O** or have **some more memory** or wait for **signal/interrupt,** it requests operating system to facilitate all these. This **request is made through System Call.**
- An interrupt is a signal emitted by hardware or software when a process or an event needs immediate attention. It alerts the processor to a high-priority process requiring interruption of the current working process. In I/O devices, one of the bus control lines is dedicated for this purpose and is called the ***Interrupt Service Routine*** (ISR).
- System calls are also called **software-interrupts.**

# Concepts of Operating System

## Session 2:

## *Introduction to Linux*

- Linux is an operating system, it is just like Windows and Mac OS X. Operating system is software that leverages hardware of the devices such as the laptop, desktop or tabs to the most. In a simple way, we can say the operating system is a bridge between the software and the hardware. Without OS, it is not possible to run or execute software or program.
- Operating systems based on Linux are known as *Linux distributions* or *distros*. Examples include Debian, Ubuntu, Fedora, CentOS, Gentoo, Arch Linux, and many others.
- The Linux kernel has been under active development since 1991, and has proven to be extremely versatile and adaptable. You can find computers that run Linux in a wide variety of contexts all over the world, from web servers to cell phones.
- However, newcomers to Linux may find it somewhat difficult to approach, as Linux filesystems have a different structure than those found on Windows or MacOS. Additionally, Linux-based operating systems depend heavily on working with the command line interface, while most personal computers rely on graphical interfaces.

# Who created Linux?

- Linux is an operating system or a kernel which germinated as an idea in the mind of young and bright **Linus Torvalds** when he was a computer science student. He used to work on the **UNIX OS (proprietary software)** and thought that it needed improvements.
- However, when his suggestions were rejected by the designers of UNIX, he thought of launching an OS which will be **receptive to changes, modifications suggested by its users**.
- So **Linus devised a Kernel** named Linux in 1991.
- As time passed by, he collaborated with other **programmers in places like MIT** and applications for Linux started to appear. So around 1991, a working Linux operating system with some applications was officially launched, and this was the start of one of the **most loved and open-source OS options available today**.
- The earlier versions of Linux OS were not so user-friendly as they were in use by computer programmers and **Linus Torvalds never had it in mind to commercialize** his product.

# Benefits of Linux

- The main advantage of Linux was that programmers were able to use the Linux Kernel to design their own custom operating systems. With time, a new range of user-friendly OS's stormed the computer world. Now, **Linux is one of the most popular and widely used Kernel,** and it is the backbone of popular operating systems like **Debian, Knoppix, Ubuntu, and Fedora**. Nevertheless, the list does not end here as there are thousands of Best versions of Linux OS based on the Linux Kernel available which offer a variety of functions to the users.

- Linux Kernel is normally used in combination of GNU project by Dr. Richard Stallman. All modern distributions of Linux are actually distributions of Linux/GNU

# Benefits of Linux Continue…..

Linux OS now enjoys popularity at its prime, and it’s famous among programmers as well as regular computer users around the world. Its main benefits are –

- It offers a **free operating system**. You do not have to shell hundreds of dollars to get the OS like Windows!
- Being open-source, anyone with programming knowledge can modify it.
- It is easy to learn Linux for beginners
- The Linux operating systems now offer **millions of programs/applications and Linux software's to choose from**, most of them are free!
- Once you have Linux installed you no longer need an antivirus! Linux is a highly secure system. More so, there is a global development community constantly looking at ways to enhance its security. With each upgrade, the OS becomes more secure and robust
- Linux freeware is the OS of choice for Server environments due to its stability and reliability (Mega-companies like Amazon, Facebook, and Google use Linux for their Servers). A Linux based server could run non-stop without a reboot for years on end.

## Is it for me?

- Users, who are new to Linux, usually shun it by falsely considering it as a difficult and technical OS to operate but, to state the truth, in the last few years Linux operating systems have become a lot more user-friendly than their counterparts like Windows**,** so trying them is the best way to know whether Linux suits you or not.
- There are **thousands of Best Linux OSs** and Linux softwares available based on the Linux Kernel; most of them offer **state-of-the-art security and applications**, **all of it for free!**

## Differentiate between Unix and Linux

- UNIX is called the mother of operating systems which laid out the foundation to Linux. Unix is designed mainly for mainframes and is in enterprises and universities. While Linux is fast becoming a household name for computer users, developers, and server environment. You may have to pay for a Unix kernel while in Linux it is free.
- But, the **commands used on both the operating systems are usually the same.** There is not much difference between UNIX and Linux. Though they might seem different, at the core, they are essentially the same. Since **Linux is a clone of UNIX**. So learning one is same as learning another.

## KEY DIFFERENCE

- Comparing Unix vs Linux, Linux source code is available to the general public whereas, in Unix, the source code is proprietary.
- UNIX OS was created in the late 1960s at AT&T Bell Labs whereas Linux is an operating system built by Linus Torvalds at the University of Helsinki in 1991.
- The main difference between Linux and Unix is Linux is a clone of Unix
- When we compare Unix to Linux, Linux default shell is BASH while the Unix shell is Bourne Shell.
- One key Unix and Linux difference is that Linux threat detection and solution are very fast while Unix users require longer wait times to get the proper bug fixing patch.
- Important versions of Linux are Redhat, Ubuntu, OpenSuse, Solaris, whereas important versions of Unix are HP-UX, AIS, BSD, etc.

# Features of Linux Operating System

Following are the important features of Linux Operating System

- Support multitasking
- Programs consist of one or more processes, and each process have one or more threads
- It can easily co-exists along with other Operating systems.
- Comparing features of Linux and Unix, Linux can run multiple user programs
- Individual accounts are protected because of appropriate authorization
- When comparing Unix vs Linux, Linux is a replica of UNIX but does not use its code.

# Difference between Linux and Windows

- Linux and Windows both are operating systems. Linux is open source and is free to use whereas Windows is a proprietary.
- Following are the important differences between Linux and Windows.

| Sr. No. | Key | Linux | Windows |
|---|---|---|---|
| 1 | Open Source | Linux is Open Source and is free to use. | Windows is not open source and is not free to use. |
| 2 | Case sensitivity | Linux file system is case sensitive. | Windows file system is case insensitive. |
| 3 | kernel type | Linux uses monolithic kernel. | Windows uses micro kernel. |
| 4 | Efficiency | Linux is more efficient in operations as compared to Windows. | Windows is less efficient in operations. |
| 5 | Path Seperator | Linux uses forward slash as path seperator between directorioes. | Windows uses backward slash as a path seperator. |
| 6 | Security | Linux is highly secure as compared to Windows. | Windows provides less security as compared to Linux. |

# Live Stats

**https://www.internetlivestats.com/**

**https://gs.statcounter.com/**

# Linux File System

- A Linux file system is a structured collection of files on a disk drive or a partition. A partition is a segment of memory and contains some specific data. In our machine, there can be various partitions of the memory. Generally, every partition contains a file system.
- The general-purpose computer system needs to store data systematically so that we can easily access the files in less time. It stores the data on hard disks (HDD) or some equivalent storage type. There may be below reasons for maintaining the file system:
- Primarily the computer saves data to the RAM storage; it may lose the data if it gets turned off. However, there is non-volatile RAM (Flash RAM and SSD) that is available to maintain the data after the power interruption.
- Data storage is preferred on hard drives as compared to standard RAM as RAM costs more than disk space. The hard disks costs are dropping gradually comparatively the RAM.

**The Linux**

file system contains the following sections:

- The root directory (/)
- A specific data storage format (EXT3, EXT4, BTRFS, XFS and so on)
- A partition or logical volume having a particular file system.

# Linux File System Features

In Linux, the file system creates a tree structure. All the files are arranged as a tree and its branches. The topmost directory called the **root (/) directory**. All other directories in Linux can be accessed from the root directory.

- **Specifying paths:** Linux does not use the backslash (\) to separate the components; it uses forward slash (/) as an alternative. For example, as in Windows, the data may be stored in C:\ My Documents\ Work, whereas, in Linux, it would be stored in /home/ My Document/ Work.
- **Partition, Directories, and Drives:** Linux does not use drive letters to organize the drive as Windows does. In Linux, we cannot tell whether we are addressing a partition, a network device, or an "ordinary" directory and a Drive.
- **Case Sensitivity:** Linux file system is case sensitive. It distinguishes between lowercase and uppercase file names. Such as, there is a difference between test.txt and Test.txt in Linux. This rule is also applied for directories and Linux commands.
- **File Extensions:** In Linux, a file may have the extension '.txt,' but it is not necessary that a file should have a file extension. While working with Shell, it creates some problems for the beginners to differentiate between files and directories. If we use the graphical file manager, it symbolizes the files and folders.
- **Hidden files:** Linux distinguishes between standard files and hidden files, mostly the configuration files are hidden in Linux OS. Usually, we don't need to access or read the hidden files. The hidden files in Linux are represented by a dot (.) before the file name (e.g., .ignore). To access the files, we need to change the view in the file manager or need to use a specific command in the shell.

# Types of Linux File System

When we install the Linux operating system, Linux offers many file systems such as **Ext, Ext2, Ext3, Ext4, JFS, ReiserFS, XFS, btrfs,** and **swap**.

## 1. Ext, Ext2, Ext3 and Ext4 file system

The file system Ext stands for **Extended File System**. It was primarily developed for **MINIX OS**. The Ext file system is an older version, and is no longer used due to some limitations.

**Ext2** is the first Linux file system that allows managing two terabytes of data. Ext3 is developed through Ext2; it is an upgraded version of Ext2 and contains backward compatibility. The major drawback of Ext3 is that it does not support servers because this file system does not support file recovery and disk snapshot.

**Ext4** file system is the faster file system among all the Ext file systems. It is a very compatible option for the SSD (solid-state drive) disks, and it is the default file system in Linux distribution.

## 2. JFS File System

JFS stands for **Journaled File System**, and it is developed by **IBM for AIX Unix**. It is an alternative to the Ext file system. It can also be used in place of Ext4, where stability is needed with few resources. It is a handy file system when CPU

power is limited.

# Types of Linux File System continue……

### 3. ReiserFS File System

ReiserFS is an alternative to the Ext3 file system. It has improved performance and advanced features. In the earlier time, the ReiserFS was used as the default file system in SUSE Linux, but later it has changed some policies, so SUSE returned to Ext3. This file system dynamically supports the file extension, but it has some drawbacks in performance.

### 4. XFS File System

XFS file system was considered as high-speed JFS, which is developed for parallel I/O processing. NASA still using this file system with its high storage server (300+ Terabyte server).

### 5. Btrfs File System

Btrfs stands for the **B tree file system**. It is used for fault tolerance, repair system, fun administration, extensive storage configuration, and more. It is not a good suit for the production system.

### 6. Swap File System

The swap file system is used for memory paging in Linux operating system during the system hibernation. A system that never goes in hibernate state is required to have swap space equal to its RAM size.

# Session 2 Continue…….

# LINUX Installation

- Here we are using virtual machine VMWARE to install Linux

# What is Virtual machine?

- A **Virtual Machine** (VM) is a compute resource that uses software instead of a physical computer to run programs and deploy apps. One or more virtual “guest” machines run on a physical “host” machine. Each virtual machine runs its own operating system and functions separately from the other VMs, even when they are all running on the same host. This means that, for example, a virtual MacOS virtual machine can run on a physical PC.
- Virtual machine technology is used for many use cases across on-premises and cloud environments. More recently, public cloud services are using virtual machines to provide virtual application resources to multiple users at once, for even more cost efficient and flexible compute.

# What are virtual machines used for?

- Virtual machines (VMs) allow a business to run an operating system that behaves like a completely separate computer in an app window on a desktop. VMs may be deployed to accommodate different levels of processing power needs, to run software that requires a different operating system, or to test applications in a safe, sandboxed environment.
-
- Virtual machines have historically been used for server virtualization, which enables IT teams to consolidate their computing resources and improve efficiency. Additionally, virtual machines can perform specific tasks considered too risky to carry out in a host environment, such as accessing virus-infected data or testing operating systems. Since the virtual machine is separated from the rest of the system, the software inside the virtual machine cannot tamper with the host computer.

# How do virtual machines work?

- The virtual machine runs as a process in an application window, similar to any other application, on the operating system of the physical machine. Key files that make up a virtual machine include a log file, NVRAM setting file, virtual disk file and configuration file.

# Advantages / Disadvantages of virtual machines

**Advantages**

- Virtual machines are easy to manage and maintain, and they offer several advantages over physical machines:
- VMs can run multiple operating system environments on a single physical computer, saving physical space, time and management costs.
- Virtual machines support legacy applications, reducing the cost of migrating to a new operating system. For example, a Linux virtual machine running a distribution of Linux as the guest operating system can exist on a host server that is running a non-Linux operating system, such as Windows.
- VMs can also provide integrated disaster recovery and application provisioning options.

**Disadvantages**

- While virtual machines have several advantages over physical machines, there are also some potential disadvantages:
- Running multiple virtual machines on one physical machine can result in unstable performance if infrastructure requirements are not met.
- Virtual machines are less efficient and run slower than a full physical computer. Most enterprises use a combination of physical and virtual infrastructure to balance the corresponding advantages and disadvantages.

# The two types of virtual machines

- Users can choose from two different types of virtual machines—process VMs and system VMs:
- **A process virtual machine** allows a single process to run as an application on a host machine, providing a platform-independent programming environment by masking the information of the underlying hardware or operating system. An example of a process VM is the Java Virtual Machine, which enables any operating system to run Java applications as if they were native to that system.
- **A system virtual machine** is fully virtualized to substitute for a physical machine. A system platform supports the sharing of a host computer's physical resources between multiple virtual machines, each running its own copy of the operating system. This virtualization process relies on a hypervisor, which can run on bare hardware, such as VMware ESXi, or on top of an operating system.

# VMWARE Virtual Machine

- VMware Workstation Player is **free for personal non-commercial use** (business and non-profit use is considered commercial use). If you would like to learn about virtual machines or use them at home you are welcome to use VMware Workstation Player for free.

# Benefits of Virtual Machines

There are many benefits to using virtual machines in your organization, including the following:

- **Cost savings**
- Before virtualization, organizations spent a significant percentage of their IT spending on acquiring physical servers for hosting applications. As virtualization gained headway, organizations shifted to buying more robust hardware that could accommodate more virtual systems. This is significantly less costly than buying more hardware. Moreover, you can extend the life of old software with VMs.
- **Easy provisioning**
- It can take hours or days to replace physical servers hit by a disastrous event. In contrast, virtualized environments take only minutes to bring up again, since you can always replicate or clone them.
- **Increased productivity**
- IT teams are more productive since they have to maintain only the host machines. For traditional datacenters that have not yet moved to a virtualized environment, hardware maintenance is more tedious.
- **Efficient DevOps**
- Maintenance operations have minimal impact on your production environment. In many cases, there is no need for downtime when performing maintenance. In addition, VMs allow for simplified testing and development processes for applications and websites.
- **Environment-friendly IT operations**
- The fewer the number of servers, the more you save in terms of power consumption. This leads to lower operating expenses for you and less carbon footprint from your datacenter.

# Linux Commands

Session 2 Continue…….

# Commands are used to work with files and directories.

## Pwd

This command displays the present working directory where you are currently in.
In the following example I am inside aditya directory which is inside
the home directory.

```
$ pwd /home/aditya
```

## ls

This command will list the content of a directory.
In the following example we are listing the content of a directory.

```
$ ls happy helloworld.txt super
```

## ls -la

This command will list all the content of a directory including the hidden files
and directories.
In the following example we are listing all the content of a directory

```
$ ls -la
```

# Commands are used to work with files and directories. Continue..

## mkdir

This command will create a new directory, provided it doesn't exists.
In the following example we are creating a new directory example.
$ mkdir example

## mkdir -p

This command will create nested directories.
In the following example we are creating world directory which is inside the hello directory which is inside the example directory.
$ mkdir -p example/hello/world

## rmdir

This command will remove/delete an existing directory, provided it is empty.
In the following example will are removing/deleting an existing directory example.
$ rmdir example

## cd

This command is used to change directory.
In the following command we are moving to root directory.
$ cd /
In the following command we are moving to /var/www/html directory.
$ cd /var/www/html

## cd ..

This command will take us one level up the directory tree.
$ cd ..
Example: If we are inside world directory which is inside the hello directory i.e. /hello/world then, cd .. will take us one level up to the hello directory.

Commands are used to work with files and directories.

## touch filename

This command will creates a new file.
In the following example we are creating a new file hello.txt.

```
$ touch hello.txt
```

## rm filename

This command will delete a file.
In the following example we are deleting a file by the name hello.txt.

```
$ rm hello.txt
```

## rm -f filename

This command forcefully deletes a file.
In the following example we are forcefully deleting a file by the name hello.txt.

```
$ rm -f hello.txt
```

## rm -r directory

This command deletes a directory recursively along with its content.
In the following example we are deleting a directory example along with its content.

```
$ rm -r example
```

## rm -rf directory

This command forcefully and recursively deletes a directory along with its content.
In the following example we are forcefully deleting a directory example along with its content.

```
$ rm -rf example
```

Be careful while performing delete operation.

Commands are used to work with files and directories. Continue..

## cp file1 file2

This command copies the content of file file1 into file file2.If file file2 doesn't exists then it is created. If it exists then its content is overwritten.In the following example we are copying the content of file hello.txt to hi.txt.

```
$ cp hello.txt hi.txt
```

## cp -r dir1 dir2

This command copies the content of directory dir1 into directory dir2. If directory dir2 doesn't exists then it is created. If it exists then its content is overwritten. In the following example we are copying the content of directory awesome to directory superawesome.

```
$ cp –r awesome superawesome
```

## mv – rename files and directories

We can use `mv` command to rename files and directories.

In the following example we are renaming file `hello.txt` to `hi.txt`.

```
$ mv hello.txt hi.txt
```

In the following example we are renaming directory `awesome` to `superawesome`.

Commands are used to work with files and directories. Continue..

## mv – move files and directories

We can also use `mv` command to move files and directories.

In the following example we are moving file `hello.txt` from directory example to directory awesome.

```
$ mv /example/hello.txt /awesome/
```

In the following example we are moving directory `example` inside `/var/www/html` directory.

```
$ mv example/ /var/www/html/
```

## cat filename

This will print the content of a file.

In the following example we will get the content of the file hello.txt in the terminal.

```
$ cat hello.txt
```

The quick brown fox jumps over the lazy dog.

Commands are used to work with files and directories.

Continue

## head filename

This command will print the first 10 lines of a file.
In the following example we will get the first 10 lines of the file fruits.txt.

```
$ head fruits.txt
```

## tail filename

This command will print the last 10 lines of a file.
In the following example we will get the last 10 lines of the file fruits.txt.

```
$ tail fruits.
```

## tail -f filename

This will print the last 10 lines of a file and will keep printing new lines as they get appended to the file.
This is useful when checking live activity logs.
In the following example we will get the last 10 lines of the file log.txt and the new content as they get appended to the file.

```
$ tail -f log.txt
```

## Understanding Linux File Permissions

Linux File Permissions

The basic Linux permissions model works by associating each system file with an owner and a group and assigning permission access rights for three different classes of users:

- The file owner.
- The group members.
- Others (everybody else).

File ownership can be changed using the chown and chgrp commands.

Three file permissions types apply to each class of users:

- The read permission.
- The write permission.
- The execute permission.

This concept allows you to control which users can read the file, write to the file, or execute the file.

To view the file permissions, use the ls command:

ls -l file_name

```
Output

-rw-r--r-- 12 linuxize users 12.0K Apr  28 10:10 file_name
|[-][-][-]-   [------] [---]
| |  |  | |      |       |
| |  |  | |      |       +-----------> 7. Group
| |  |  | |      +-------------------> 6. Owner
| |  |  | +--------------------------> 5. Alternate Access Method
| |  |  +----------------------------> 4. Others Permissions
| |  +------------------------------> 3. Group Permissions
| +--------------------------------> 2. Owner Permissions
+----------------------------------> 1. File Type
```

The first character indicates the file type. It can be a regular file (-), directory (d), a symbolic link , or other special types of files. The following nine characters represent the file permissions, three triplets of three characters each. The first triplet shows the owner permissions, the second one group permissions, and the last triplet shows everybody else permissions.

In the example above (rw-r--r--) means that the file owner has read and write permissions (rw-), the group and others have only read permissions (r--).

File permissions have a different meaning depending on the file type.

Each of the three permission triplets can be constructed of the following characters and have different effects, depending on whether they are set to a file or to a directory:

**Effect of Permissions on Files**

| Permission | Character | Meaning on File |
|---|---|---|
| **Read** | - | The file is not readable. You cannot view the file contents. |
| | r | The file is readable. |
| **Write** | - | The file cannot be changed or modified. |
| | w | The file can be changed or modified. |
| **Execute** | - | The file cannot be executed. |
| | x | The file can be executed. |

**Effect of Permissions on Directories (Folders)**

Directories are special types of files that can contain other files and directories.

| Permission | Character | Meaning on Directory |
|---|---|---|
| **Read** | - | The directory's contents cannot be shown. |
| | r | The directory's contents can be shown.<br>(e.g., You can list files inside the directory with ls.) |
| **Write** | - | The directory's contents cannot be altered. |
| | w | The directory's contents can be altered.<br>(e.g., You can create new files, delete files ..etc.) |

| Permission | Character | Meaning on Directory |
|---|---|---|
| **Execute** | - | The directory cannot be changed to. |
| | x | The directory can be navigated using cd. |

### Changing File permissions

The File permissions can be changed using the chmod command. Only root, the file owner, or user with sudo privileges can change the permissions of a file. Be extra careful when using chmod, especially when recursively changing the permissions. The command can accept one or more files and/or directories separated by space as arguments.

Permissions can be specified using a symbolic mode, numeric mode, or a reference file.

### Symbolic (Text) Method

The syntax of the chmod command when using the symbolic mode has the following format:

chmod [OPTIONS] [ugoa…][-+=]perms…[,…] FILE...

The first set of flags ([ugoa…]), users flags, defines the users' classes for which the permissions to the file are changed.

- u - The file owner.
- g - The users who are members of the group.
- o - All other users.
- a - All users, identical to ugo.

When the users' flag is omitted, it defaults to a.

The second set of flags ([-+=]), the operation flags, defines whether the permissions are to be removed, added, or set:

- - - Removes the specified permissions.
- + - Adds specified permissions.
- = - Changes the current permissions to the specified permissions. If no permissions are given after the = symbol, all permissions from the specified user class are removed.

The permissions (perms...) are explicitly set using either zero or one or more of the following letters: r, w, x, X, s, and t. Use a single letter from the set u, g, and o when copying permissions from one to another users' class.

When setting permissions for more than one user classes ([,…]), use commas (without spaces) to separate the symbolic modes.
Here are some examples of how to use the chmod command in symbolic mode:

- Give the members of the group permission to execute the file, but not to read and write to it:

  ```
  chmod g=x filename
  ```

- Remove the write permission for all users:

  ```
  chmod a-w filename
  ```

- Recursively remove the execute permission for other users:

  ```
  chmod -R o-x dirname
  ```

- Remove the read, write, and execute permission for all users except the file’s owner:

  ```
  chmod og-rwx filename
  ```

  The same thing can also be accomplished by using the following form:

  ```
  chmod og= filename
  ```

- Give read, write and execute permission to the file’s owner, read permissions to the file’s group, and no permissions to all other users:

  ```
  chmod u=rwx,g=r,o= filename
  ```

**Numeric Method**

The syntax of the chmod command when using the symbolic mode has the following format:

chmod [OPTIONS] NUMBER FILE...

When using the numeric mode, you can set the permissions for all three user classes (owner, group, and all others) at the same time.

The permission number can be a 3 or 4-digits number. When 3 digits number is used, the first digit represents the permissions of the file's owner, the second one the file's group, and the last one all other users.

Each write, read, and execute permissions have the following number value:

- r (read) = 4
- w (write) = 2
- x (execute) = 1
- no permissions = 0

The permissions number of a specific user class is represented by the sum of the values of the permissions for that group.

To find out the file's permissions in numeric mode, simply calculate the totals for all users' classes. For example, to give read, write and execute permission to the file's owner, read and execute permissions to the file's group and only read permissions to all other users, you would do the following:

- Owner: rwx=4+2+1=7
- Group: r-x=4+0+1=5
- Others: r-x=4+0+0=4

Using the method above, we come up to the number 754, which represents the desired permissions.

Here are some examples of how to use the `chmod` command in numeric mode:

- Give the file's owner read and write permissions and only read permissions to group members and all other users:

  ```
  chmod 644 dirname
  ```

- Give the file's owner read, write and execute permissions, read and execute permissions to group members and no permissions to all other users:

| | |
|---|---|
| chmod 750 dirname | |

# Shell Scripting

# Bash Shell Scripting Definition

**Bash**

Bash is a command language interpreter. It is widely available on various operating systems and is a default command interpreter on most GNU/Linux systems. The name is an acronym for the '**B**ourne-**A**gain **SH**ell'.

**Shell**

Shell is a macro processor which allows for an interactive or non-interactive command execution.

**Scripting**

Scripting allows for an automatic commands execution that would otherwise be executed interactively one-by-one.

# What is Shell

Well, the terminal window in front of you contains shell, and shell allows you by use of commands to interact with your computer, hence retrieve or store data, process information and various other simple or even extremely complex tasks.

Try it now! Use your keyboard and type some commands such as **`date`**, **`cal`**, **`pwd`** or **`ls`** followed by the **`ENTER`** key.

What you have just done, was that by use of commands and shell you interacted with your computer to retrieve a current date and time (date), looked up a calendar (cal), checked the location of your current working directory (pwd) and retrieved a list of all files and directories located within (ls).

# What is Scripting

Now, imagine that the execution of all the above commands is your daily task. Every day you are required to execute all of the above commands without fail as well as store the observed information. Soon enough this will become an extremely tedious task destined for failure. Thus the obvious notion is to think of some way to execute all given commands together. This is where scripting becomes your salvation.

To see what is meant by scripting, use shell in combination with your favorite text editor eg. vi to create a new file called **`task.sh`** containing all the above commands, each on a separate line. Once ready, make your new file executable using **`chmod`** command with an option **`+x`**. Lastly, execute your new script by prefixing its name with `./`.

many GNU/Linux systems, thus we have been using it even without realising. This is why our previous shell script works even without us defining bash as an interpreter. To see what is your default interpreter execute command **`echo $SHELL`**:

```
$ echo $SHELL /bin/bash
```

There are various other shell interpreters available, such as Korn shell, C shell and more. From this reason, it is a good practice to define the shell interpreter to be used explicitly to interpret the script's content.

To define your script's interpreter as Bash, first locate a full path to its executable binary using **`which`** command, prefix it with a `#!` and insert it as the first line of your script. There are various other techniques how to define shell interpreter, but this is a solid start.

# File Names and Permissions

Next, let's briefly discuss file permissions and filenames. You may have already noticed that in order to execute shell script the file needs to be made executable by use of **`chmod +x FILENAME`** command. By default, any newly created files are not executable regardless of its file extension suffix. In fact, the file extension on GNU/Linux systems mostly does not have any meaning apart from the fact, that upon the execution of **`ls`** command to list all files and directories it is immediately clear that file with extension **`.sh`** is plausibly a shell script and file with **`.jpg`** is likely to be a lossy compressed image.

On GNU/Linux systems a **`file`** command can be used to identify a type of the file.

## Script Execution

Next, let's talk about an alternative way on how to run bash scripts. In a highly simplistic view, a bash script is nothing else just a text file containing instructions to be executed in order from top to bottom. How the instructions are interpreted depends on defined shebang or the way the script is executed. Another way to execute bash scripts is to call bash interpreter explicitly eg. `$ bash date.sh`, hence executing the script without the need to make the shell script executable and without declaring shebang directly within a shell script. By calling bash executable binary explicitly, the content of our file `date.sh` is loaded and interpreted as Bash Shell Script.

## Hello World Bash Shell Script

Now, it is time to write our first, most basic bash shell script. The whole purpose of this script is nothing else but print “Hello World” using echo command to the terminal output. Using any text editor create a new file named hello-world.sh containing the below code:

```
#!/bin/bash

echo "Hello World“
```

Once ready, make your script executable with thechmod command and execute it using relative path ./hello-world.sh:

```
$ chmod +x hello-world.sh
$ linuxconfig.org:~$ ./hello-world.sh
Hello World
$
```

# Variables

Variables are the essence of programming. Variables allow a programmer to store data, alter and reuse them throughout the script. Create a new script **`welcome.sh`** with the following content:

```
#!/bin/bash

greeting="Welcome"
user=$(whoami)
day=$(date +%A)

echo "$greeting back $user! Today is $day, which is the best day of the entire week!"
echo "Your Bash shell version is: $BASH_VERSION.  Enjoy!“
```

After running the above welcome.sh script, you will see an output similar to the one below:

```
$ ./welcome.sh
Welcome back linuxconfig! Today is Wednesday, which is the best day of the entire week!
Your Bash shell version is: 4.4.12(1)-release. Enjoy!
```

# Functions

Functions allow a programmer to organize and reuse code, hence increasing the efficiency, execution speed as well as readability of the entire script.
It is possible to avoid using functions and write any script without including a single function in it. However, you are likely to end up with a chunky, inefficient and hard to troubleshoot code. This can be extremely useful if the output or calculation you require consists of multiple commands, and it will be expected multiple times throughout the script execution. Functions are defined by using the function keyword and followed by function body enclosed by curly brackets.

Let create a simple shell function to be used to print user details and will make two function calls, thus printing user details twice upon a script execution.

The function name is user_details, and function body enclosed inside curly brackets consists of the group of two echo commands. Every time a function call is made by using the function name, both echo commands within our function definition are executed. It is important to point out that the function definition must precede function call, otherwise the script will return function not found error:

| Description | Numeric Comparison | String Comparison |
|---|---|---|
| Shell comparison example: | [ 100 -eq 50 ]; echo $? | [ "GNU" = "UNIX" ]; echo $? |
| less than | -lt | < |
| greater than | -gt | > |
| equal | -eq | = |
| not equal | -ne | != |
| less or equal | -le | N/A |
| greater or equal | -ge | N/A |

After reviewing the above table, let's say, we would like to compare numeric values like two integers `1` and `2`. The following video example will first define two variables `$a` and `$b` to hold our integer values.
Next, we use square brackets and numeric comparison operators to perform the actual evaluation. Using `echo $?` command, we check for a return value of the previously executed evaluation. There or two possible outcomes for every evaluation, true or false. If the return value is equal to `0`, then the comparison evaluation is true. However, if the return value is equal to `1`, the evaluation resulted as false.

we compare two integers using the numeric comparison operator to determine if they are equal in value. Remember, `0` signals true, while `1` indicates false:

```
#!/bin/bash
string_a="UNIX"
string_b="GNU"
echo "Are $string_a and $string_b strings equal?"
[ $string_a = $string_b ]
echo $?
num_a=100
num_b=100
echo "Is $num_a equal to $num_b ?"
[ $num_a -eq $num_b ]
echo $?
```

Save the above script as eg_comparison.sh file, make it executable and execute:

```
$ chmod +x compare.sh
$ ./compare.sh
Are UNIX and GNU strings equal?
1
Is 100 equal to 100 ?
0
```

## Conditional Statements

Conditionals allow the programmer to implement decision making within a shell script based on certain conditions or events.

The conditionals we are referring to are of course, if, then and else. For example, we can improve our backup script by implementing a sanity check to compare the number of files and directories within a source directory we intend to backup and the resulting backup file. The pseudocode for this kind of implementation will read as follows:

IF the number of files between the source and destination target is equal THEN print the OK message, ELSE, print ERROR.
Let's start by creating a simple bash script depicting a basic if/then/else construct.

```
#!/bin/bash

num_a=100
num_b=200

if [ $num_a -lt $num_b ]
then
    echo "$num_a is less than $num_b!"
fi
```

## Bash Loops

Loops are looping constructs used to iterate through any given number of tasks until all items in a specified list were completed or predefined conditions were met. There are three basic loop types available to our disposal.

### For Loop

For loop is used to iterate through any given code for any number of supplied items in the list. Let's start with a simple for loop example:

```
#!/bin/bash

for i in 1 2 3; do
   echo $i
done
```

The above for loop has used the echo command to print all items 1, 2 and 3 in the list. Using a semicolon allows us to execute for loop on a single command line.

The for loop consists of four Shell Reserved Words: for, in, do, done. The above code can therefore also be read as: FOReach item INlist 1, 2 and 3 assign each item temporarily into a variable i after which DOecho $i in order to print the item as STDOUT and keep printing until all items INthe list are DONE.

## Bash Loops Continue.....

**While Loop**

The next loop construct on our list is while loop. This particular loop acts on a given condition. Meaning, it will keep executing code enclosed withing DOand DONEwhile the specified condition is true. Once the specified condition becomes false, the execution will stop. Consider the following example:

```
#!/bin/bash

counter=0
while [ $counter -lt 3 ]; do
   let counter+=1
   echo $counter
done
```

This particular while loop will keep executing the enclosed code only while the counter variable is less than 3. This condition is set on Line 4. During each loop iteration, on Lines 5the variable counter is incremented by one. Once the variable counter is equal 3, the condition defined on Lines 4 becomes false and while loop execution is terminated.

## Bash Loops Continue.....

### Until Loop

The until loop does the exact opposite of the while loop. Until loop also acts on a preset condition. However, the code enclosed between DO and DONE is repeatedly executed only until this condition changes from false to true. The execution of until loop is illustrated using the below example:

```
#!/bin/bash

counter=6
until [ $counter -lt 3 ]; do
    let counter-=1
    echo $counter
done
```

```
$ cat until.sh
#!/bin/bash
counter=6
until [ $counter -lt 3 ]; do
        let counter-=1
        echo $counter
done
$ ./until.sh
 5
4
3
2
```

If you understood the above while loop script, the until loop will be somewhat self-explanatory. The script starts with the variable counter set to 6. The condition defined on Line 4of this particular until loop is to keep executing the enclosed code until the condition becomes true.

# Shell Scripting 2

## if..elif..else Statement

The Bash if..elif..else statement takes the following form

```
#!/bin/bash

echo -n "Enter a number: "
read VAR

if [ $VAR -gt 10 ]
then
  echo "The variable is greater than 10."
elif [ $VAR -eq 10 ]
then
  echo "The variable is equal to 10."
else
  echo "The variable is less than 10."
fi
```

## Nested if Statements #

Bash allows you to nest if statements within if statements. You can place multiple if statements inside another if statement.

The following script will prompt you to enter three numbers and print the largest number among the three numbers.

```
#!/bin/bash
echo -n "Enter the first number: "
read VAR1
echo -n "Enter the second number: "
read VAR2
echo -n "Enter the third number: "
read VAR3

if [[ $VAR1 -ge $VAR2 ]]
then
  if [[ $VAR1 -ge $VAR3 ]]
  then
    echo "$VAR1 is the largest number."
  else
    echo "$VAR3 is the largest number."
  fi
else
  if [[ $VAR2 -ge $VAR3 ]]
  then
    echo "$VAR2 is the largest number."
  else
    echo "$VAR3 is the largest number."
  fi
fi
```

**Multiple Conditions**

The logical OR and AND operators allow you to use multiple conditions in the if statements.

Here is another version of the script to print the largest number among the three numbers. In this version, instead of the nested if statements, we’re using the logical AND (&&) operator.

```
#!/bin/bash

echo -n "Enter the first number: "
read VAR1
echo -n "Enter the second number: "
read VAR2
echo -n "Enter the third number: "
read VAR3

if [[ $VAR1 -ge $VAR2 ]] && [[ $VAR1 -ge $VAR3 ]]
then
  echo "$VAR1 is the largest number."
elif [[ $VAR2 -ge $VAR1 ]] && [[ $VAR2 -ge $VAR3 ]]
then
  echo "$VAR2 is the largest number."
else
  echo "$VAR3 is the largest number."
fi
```

# case Statement Syntax

The Bash case statement takes the following form:

```
case  in
   Pattern 1) Statement 1;;
   Pattern n) Statement n;;
esac
```

- Each case statement starts with the case keyword, followed by the case expression and the in keyword. The statement ends with the esac keyword.
- You can use multiple patterns separated by the | operator. The ) operator terminates a pattern list.
- A pattern can have special characters .
- A pattern and its associated commands are known as a clause.
- Each clause must be terminated with ;;.
- The commands corresponding to the first pattern that matches the expression are executed.
- It is a common practice to use the wildcard asterisk symbol (*) as a final pattern to define the default case. This pattern will always match.
- If no pattern is matched, the return status is zero. Otherwise, the return status is the exit status of the executed commands.

# Case Statement Example

Here is an example using the `case` statement in a bash script that will print the official language of a given country:

```
#!/bin/bash
echo -n "Enter the name of a country: "
read COUNTRY
echo -n "The official language of $COUNTRY is "
case $COUNTRY in
  Lithuania)
    echo -n "Lithuanian"
    ;;
  Romania | Moldova)
    echo -n "Romanian"
    ;;
  Italy | "San Marino" | Switzerland | "Vatican City")
    echo -n "Italian"
    ;;
  *)
    echo -n "unknown"
    ;;
esac
```

# Basic Operators in Shell Scripting

There are 5 basic operators in bash/shell scripting**:**

Arithmetic Operators
Relational Operators
Boolean Operators
Bitwise Operators
File Test Operators

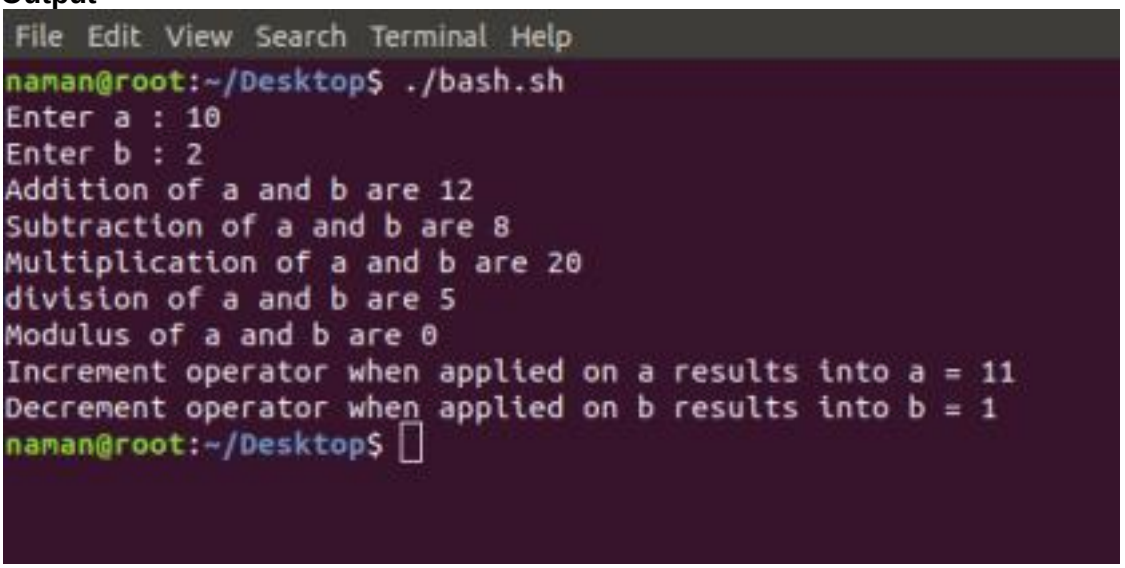
**1. Arithmetic Operators**

These operators are used to perform normal arithmetic's/mathematical operations. There are 7 arithmetic operators:

**Addition (+):** Binary operation used to add two operands.
**Subtraction (-):** Binary operation used to subtract two operands.
**Multiplication (*):** Binary operation used to multiply two operands.
**Division (/):** Binary operation used to divide two operands.
**Modulus (%)**: Binary operation used to find remainder of two operands.
I**ncrement Operator (++):** Unary operator used to increase the value of operand by one.
**Decrement Operator (- -):** Unary operator used to decrease the value of a operand by one

```
 #!/bin/bash
 #reading data from the user
read a b
add= $((a + b))
echo “Addition of a and b are $add”
sub= $((a - b))
echo “Subtraction of a and b are $sub”
mul= $((a * b))
echo “Multiplication of a and b are $mul”
div= $((a / b))
echo “division of a and b are $div”
mod= $((a % b))
echo “Modulus of a and b are $mod”
((++a))
echo “Increment operator when applied on "a" results into a = $a”
((--b))
echo “Decrement operator when applied on "b" results into b = $b”
```

**Output**




**2. Relational Operators:**

Relational operators are those operators which define the relation between two operands. They give either true or false depending upon the relation. They are of 6 types:
**'=='** Operator: Double equal to operator compares the two operands. Its returns true is they are equal otherwise returns false.
**'!='** Operator: Not Equal to operator return true if the two operands are not equal otherwise it returns false.
**'<'** Operator: Less than operator returns true if first operand is less than second operand otherwise returns false.
**'<='** Operator: Less than or equal to operator returns true if first operand is less than or equal to second operand otherwise returns false
**'>'** Operator: Greater than operator return true if the first operand is greater than the second operand otherwise return false.
**'>='** Operator: Greater than or equal to operator returns true if first operand is greater than or equal to second operand otherwise returns false

```
#!/bin/bash
#reading data from the user
read -p 'Enter a : ' a
read -p 'Enter b : ' b
if(( $a==$b ))
then
      echo a is equal to b.
else
      echo a is not equal to b.
fi
if(( $a!=$b ))
then
      echo a is not equal to b.
else
      echo a is equal to b.
fi
if(( $a<$b ))
then
      echo a is less than b.
else
      echo a is not less than b.
fi
if(( $a<=$b ))
then
      echo a is less than or equal to b.
else
      echo a is not less than or equal to b. fi
if(( $a>$b ))
then
      echo a is greater than b.
else
      echo a is not greater than b.
fi
if(( $a>=$b ))
then
      echo a is greater than or equal to b. else
      echo a is not greater than or equal to b. fi
```

**Output**

```
File  Edit  View  Search  Terminal  Help
naman@root:~/Desktop$ ./bash.sh
Enter a : 10
Enter b : 5
a is not equal to b.
a is not equal to b.
a is not less than b.
a is not less than or equal to b.
a is greater than b.
a is greater than or equal to b.
naman@root:~/Desktop$
```

**3. Logical Operators** : They are also known as boolean operators. These are used to perform logical operations. They are of 3 types:

**Logical AND (&&):** This is a binary operator, which returns true if both the operands are true otherwise returns false.

**Logical OR (||):** This is a binary operator, which returns true is either of the operand is true or both the operands are true and return false if none of then is false.

**Not Equal to (!):** This is a unary operator which returns true if the operand is false and returns false if the operand is true.

```
#!/bin/bash
#reading data from the user
read -p 'Enter a : ' a
read -p 'Enter b : ' b
if(($a == "true" & $b == "true" ))
then
      echo “Both are true”.
else
      echo “Both are not true”.
fi
if(($a == "true" || $b == "true" ))
then
      echo “At Least one of them is true”.
else
      echo “None of them is true”.
fi
if(( ! $a == "true" ))
then
      echo “a" was initially false”.
else
      echo “a" was initially true”.
fi
```

**Output**

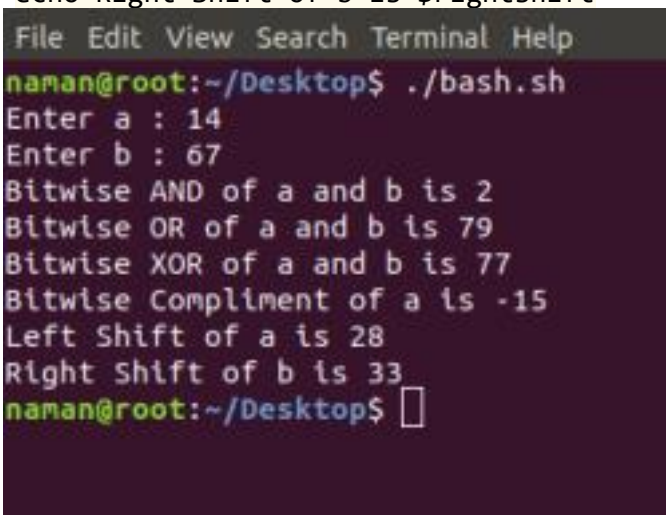
```
File  Edit  View  Search  Terminal  Help
naman@root:~/Desktop$ ./bash.sh
Enter a : true
Enter b : false
Both are true.
Atleast one of them is true.
a was intially true.
naman@root:~/Desktop$
```

**4. Bitwise Operators:** A bitwise operator is an operator used to perform bitwise operations on bit patterns. They are of 6 types:

**Bitwise And (&):** Bitwise & operator performs binary AND operation bit by bit on the operands.

**Bitwise OR (|):** Bitwise | operator performs binary OR operation bit by bit on the operands.
**Bitwise XOR (^):** Bitwise ^ operator performs binary XOR operation bit by bit on the operands.
**Bitwise complement (~):** Bitwise ~ operator performs binary NOT operation bit by bit on the operand.
**Left Shift (<<):** This operator shifts the bits of the left operand to left by number of times specified by right operand.
**Right Shift (>>):** This operator shifts the bits of the left operand to right by number of times specified by right operand.

```
#!/bin/bash
#reading data from the user
read -p 'Enter a : ' a
read -p 'Enter b : ' b
bitwiseAND=$(( a&b ))
echo Bitwise AND of a and b is $bitwiseAND
bitwiseOR=$(( a|b ))
echo Bitwise OR of a and b is $bitwiseOR
bitwiseXOR=$(( a^b ))
echo Bitwise XOR of a and b is $bitwiseXOR
bitiwiseComplement=$(( ~a ))
echo Bitwise Compliment of a is $bitiwiseComplement
leftshift=$(( a<<1 ))
echo Left Shift of a is $leftshift
rightshift=$(( b>>1 ))
echo Right Shift of b is $rightshift
```

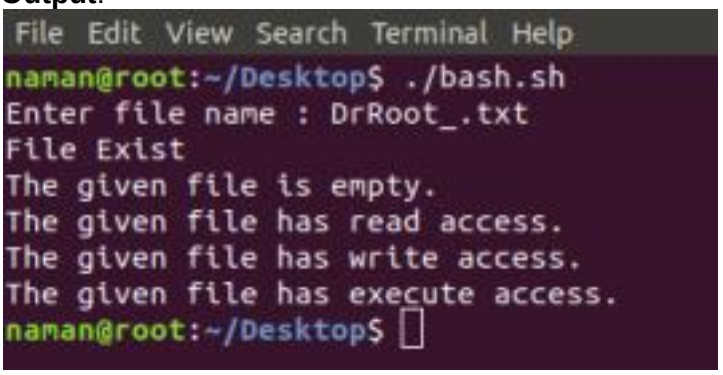



**5. File Test Operator**: These operators are used to test a particular property of a file.
**-b operator**: This operator check whether a file is a block special file or not. It returns true if the file is a block special file otherwise false.
**-c operator:** This operator checks whether a file is a character special file or not. It returns true if it is a character special file otherwise false.
**-d operator:** This operator checks if the given directory exists or not. If it exists then operators returns true otherwise false.
**-e operator**: This operator checks whether the given file exists or not. If it exits this operator returns true otherwise false.
**-r operator:** This operator checks whether the given file has read access or not. If it has read access then it returns true otherwise false.
**-w operator**: This operator check whether the given file has write access or not. If it has write then it returns true otherwise false.

**-x operator:** This operator check whether the given file has execute access or not. If it has execute access then it returns true otherwise false.
**-s operator**: This operator checks the size of the given file. If the size of given file is greater than 0 then it returns true otherwise it is false.

```
#!/bin/bash
#reading data from the user
read -p 'Enter file name : ' FileName
if [ -e $FileName ]
then
      echo File Exist
else
      echo File doesnot exist
fi
if [ -s $FileName ]
then
      echo The given file is not empty.
else
      echo The given file is empty.
fi
if [ -r $FileName ]
then
      echo The given file has read access.
else
      echo The given file does not has read access. fi
if [ -w $FileName ]
then
      echo The given file has write access.
else
      echo The given file does not has write access. fi
if [ -x $FileName ]
then
      echo The given file has execute access.
else
      echo The given file does not has execute access. fi
```

**Output**:



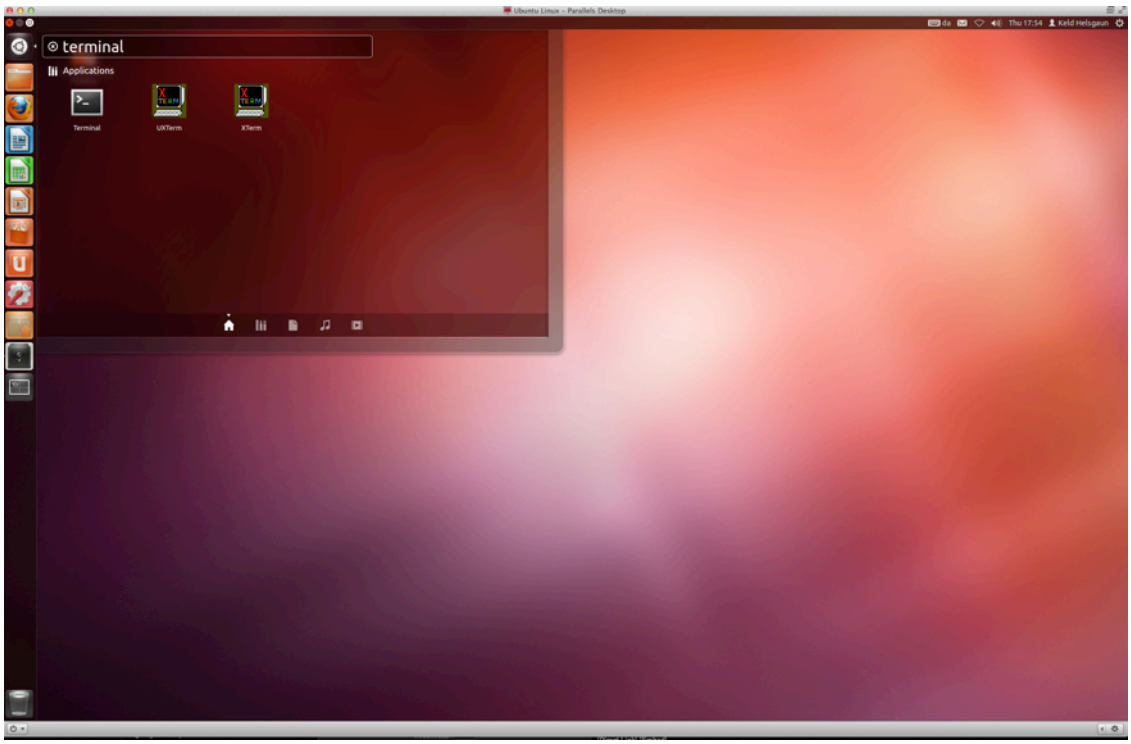
# How to Compile and Run a C Program on Ubuntu Linux

Keld Helsgaun
Roskilde University, February 2013

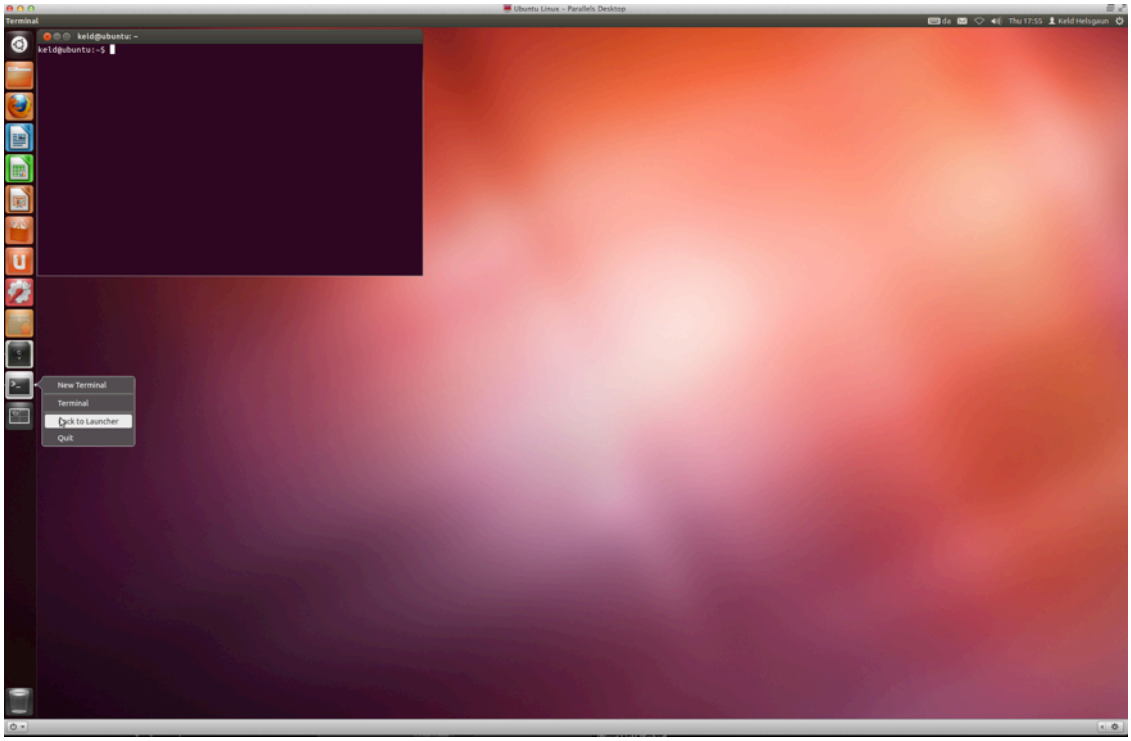
This document shows how to compile and run a C program on Ubuntu Linux using the gcc compiler.

**Step 1. Open up a terminal**

Search for the terminal application in the Dash tool (located as the topmost item in the Launcher). Open up a terminal by clicking on the icon.



For ease of future access to the terminal application, right click its icon in the Launcher and select “Lock to Launcher”.



**Step 2. Use a text editor to create the C source code.**

Type the command

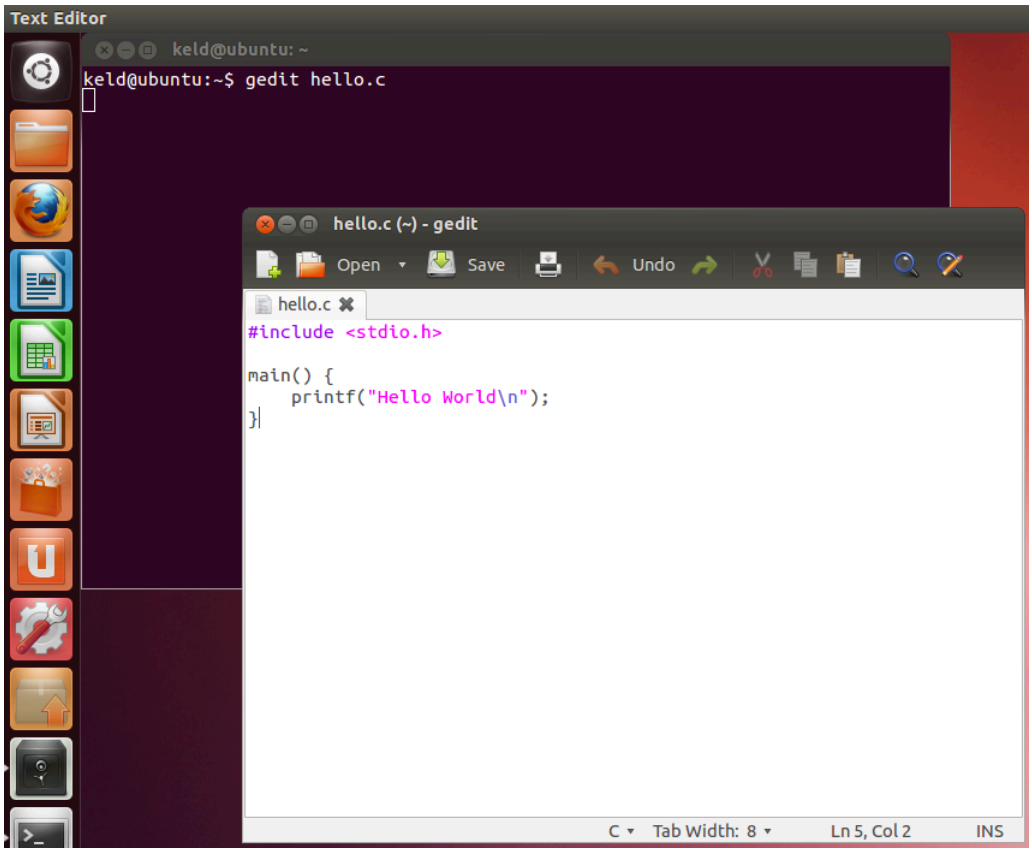
```
gedit hello.c
```

and enter the C source code below:

```
#include <stdio.h>

main() {
    printf("Hello World\n");
}
```

Close the editor window.

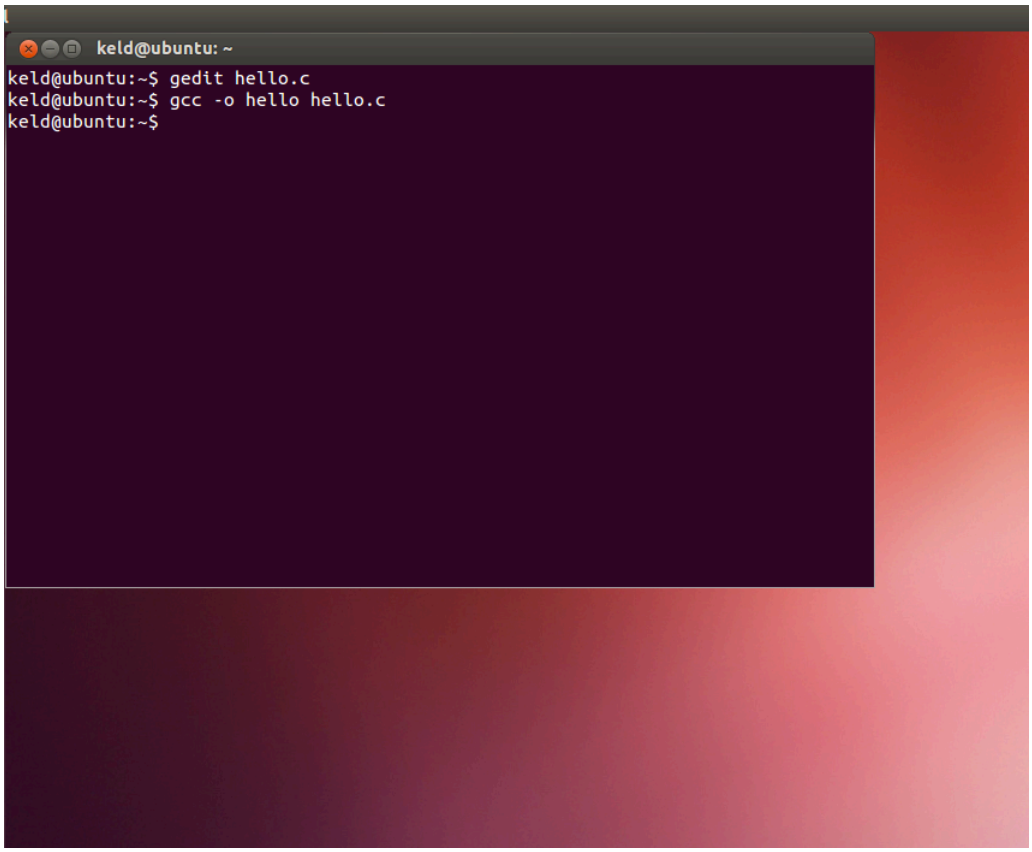


**Step 3. Compile the program.**

Type the command

```
gcc -o hello hello.c
```

This command will invoke the GNU C compiler to compile the file `hello.c` and output (-o) the result to an executable called `hello`.



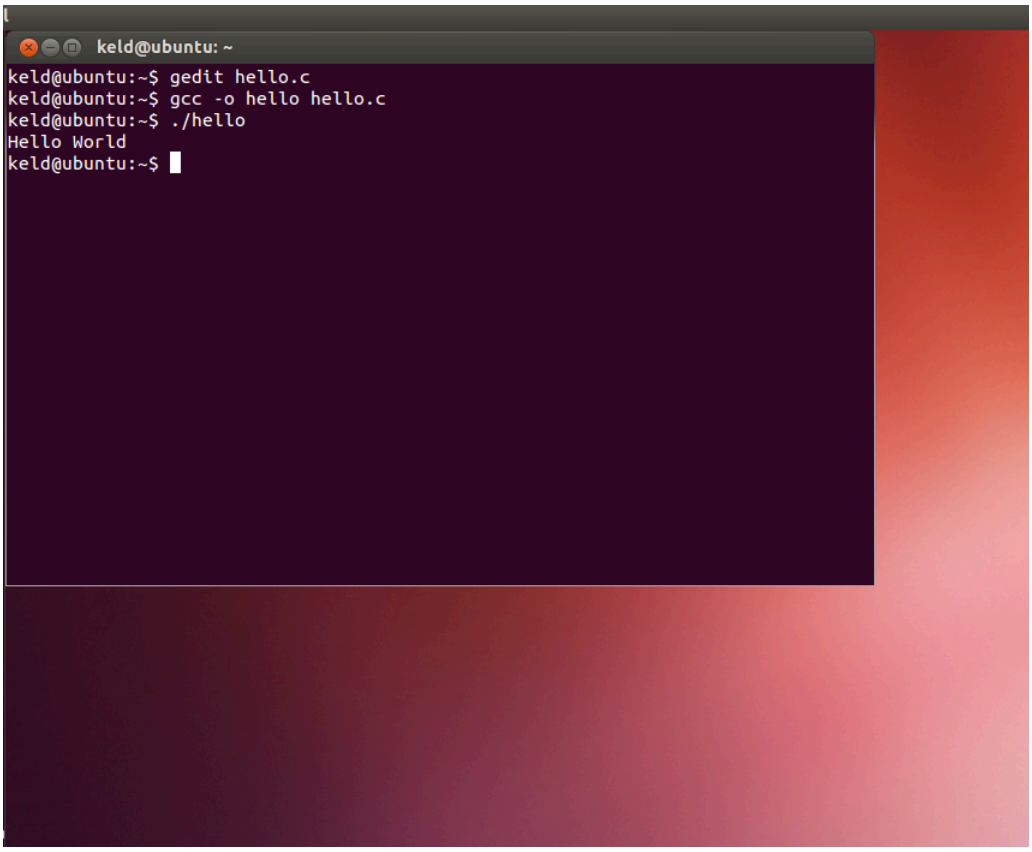
**Step 4. Execute the program.**

Type the command

```
./hello
```

This should result in the output

```
Hello World
```

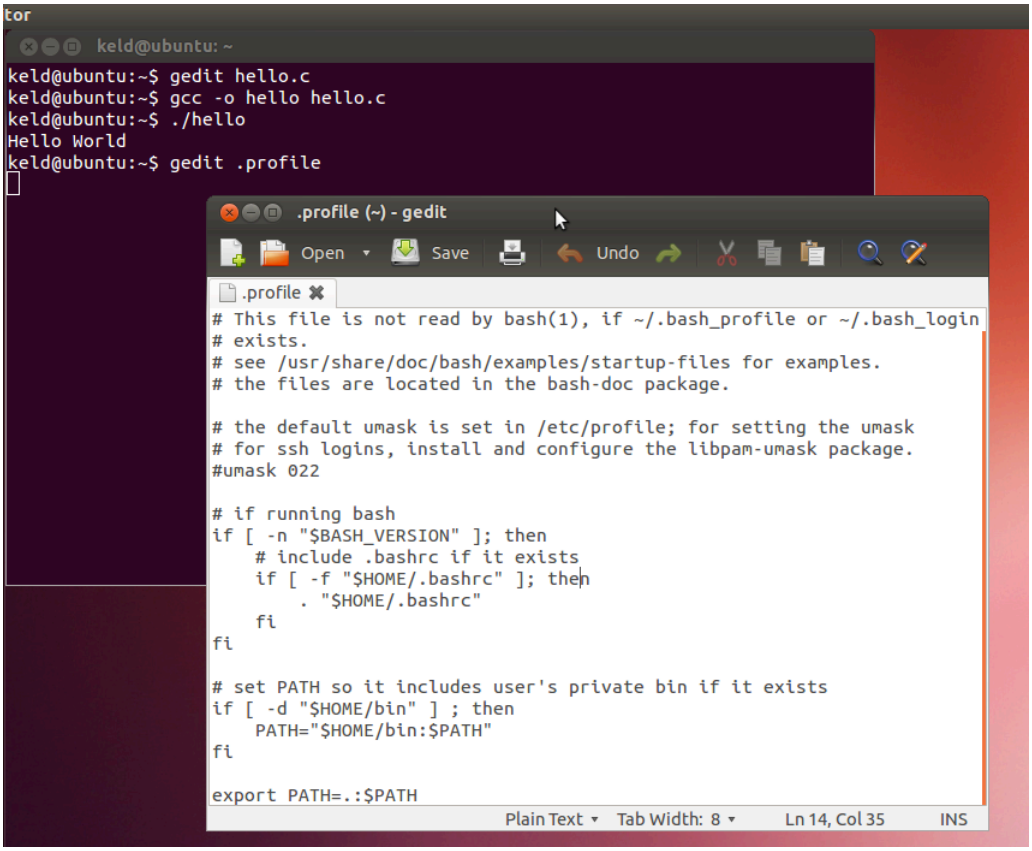


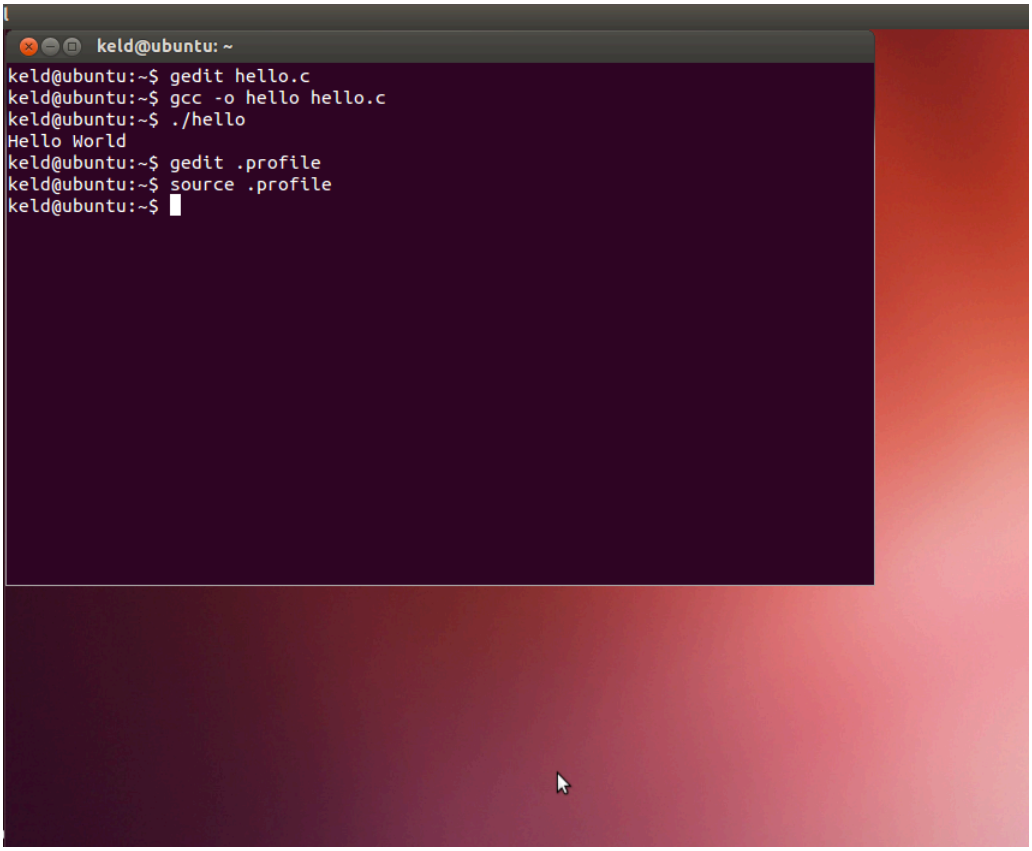
## Optional step

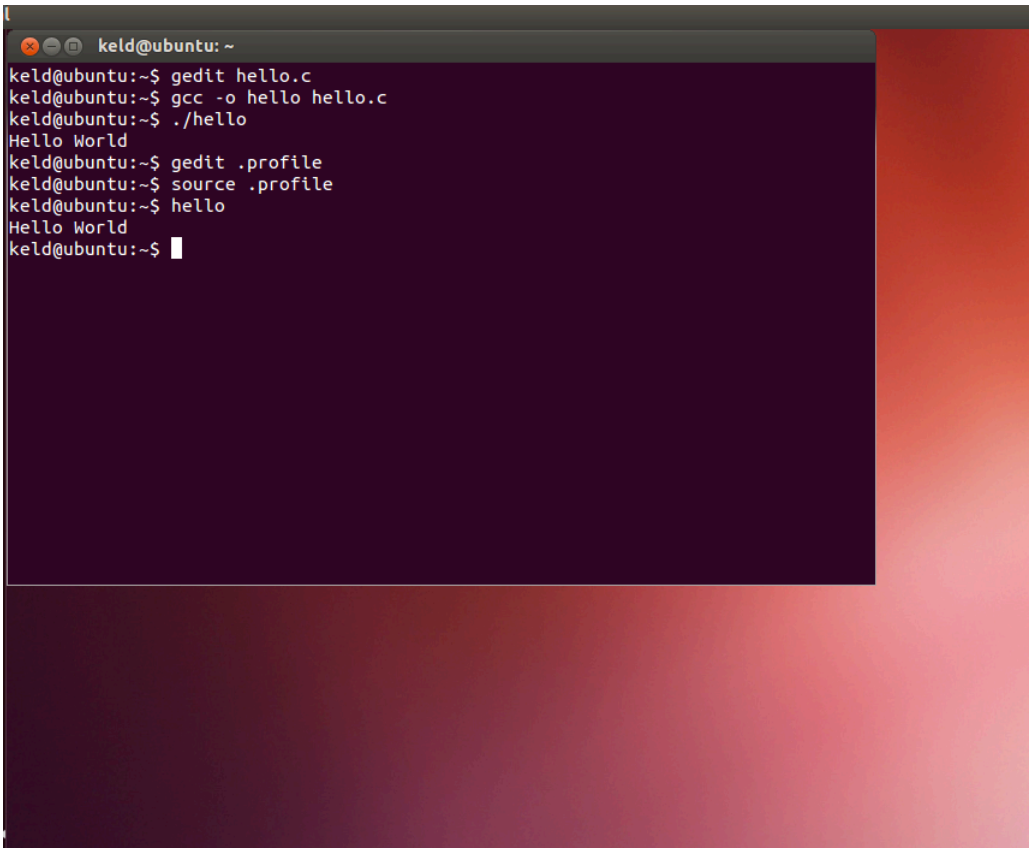
In order to avoid the./ prefix each time a program is to be executed, insert the following as the last line in the file `.profile` (located in your home folder):

```
export PATH=.:$PATH
```

This step needs only to be done once.




keld@ubuntu: ~
keld@ubuntu:~$ gedit hello.c
keld@ubuntu:~$ gcc -o hello hello.c
keld@ubuntu:~$ ./hello
Hello World
keld@ubuntu:~$ gedit .profile
keld@ubuntu:~$ source .profile
keld@ubuntu:~$


keld@ubuntu: ~
keld@ubuntu:~$ gedit hello.c
keld@ubuntu:~$ gcc -o hello hello.c
keld@ubuntu:~$ ./hello
Hello World
keld@ubuntu:~$ gedit .profile
keld@ubuntu:~$ source .profile
keld@ubuntu:~$ hello
Hello World
keld@ubuntu:~$

# Session 4,5,6

Process Management and Scheduling

# Process

A process or running process refers to a set of instructions currently being processed by the computer processor.

For example, in Windows you can see each of the processes running by opening the Processes tab in Task Manager. Windows Processes are Windows Services and background programs you normally don't see running on the computer. A process may be a printer program that runs in the background and monitors the ink levels and other printer settings while the computer is running.

A typical computer has multiple processes running all the time to help manage the operating system, its hardware, and the software running on the computer.

# Process Continue……

A process consists four major elements:

- Steps and decisions — the flowchart. A series of steps and decisions describing the way work is completed.
- Variability of processing time and flow — the pattern of processing times.
- Timing and interdependence — when the arrivals happen, when people work, etc.
- Assignment of resources — how many and where are they assigned.

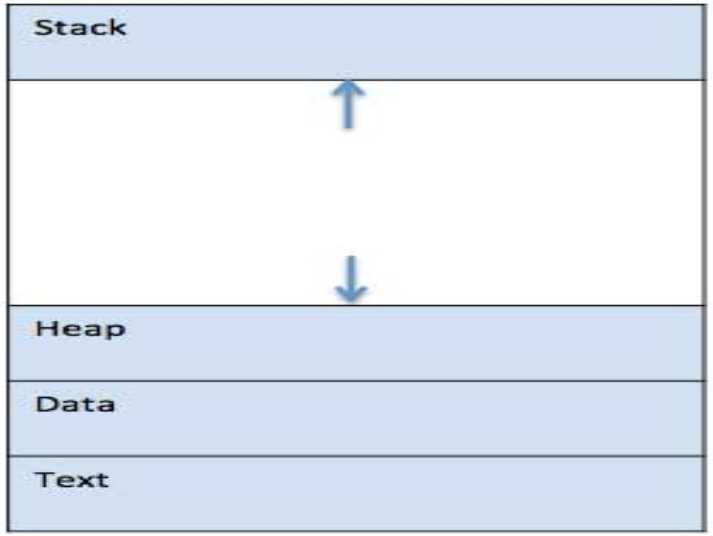
In UNIX and some other operating systems, a process is started when a program is initiated (either by a user entering a shell command or by another program). Like a task, a process is a running program with which a particular set of data is associated so that the process can be kept track of.

A process can initiate a subprocess, which is a called a child process (and the initiating process is sometimes referred to as its parent ). A child process is a replica of the parent process and shares some of its resources, but cannot exist if the parent is terminated.

Processes can exchange information or synchronize their operation through several methods of interprocess communication ( IPC ).

# Process Continue……

When a program is loaded into the memory and it becomes a process, it can be divided into four sections ─ stack, heap, text and data. The following image shows a simplified layout of a process inside main memory −




| | Component & Description |
|---|---|
| 1 | **Stack** The process Stack contains the temporary data such as method/function parameters, return address and local variables. |
| 2 | **Heap** This is dynamically allocated memory to a process during its run time. |
| 3 | **Text** This includes the current activity represented by the value of Program Counter and the contents of the processor's registers. |
| 4 | **Data** This section contains the global and static variables. |

# Process Life Cycle

When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardized.

**1. Start** This is the initial state when a process is first started/created.

**2. Ready** The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after Start state or while running it by but interrupted by the scheduler to assign CPU to some other process.

**3. Running** Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.

**4. Waiting** Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.

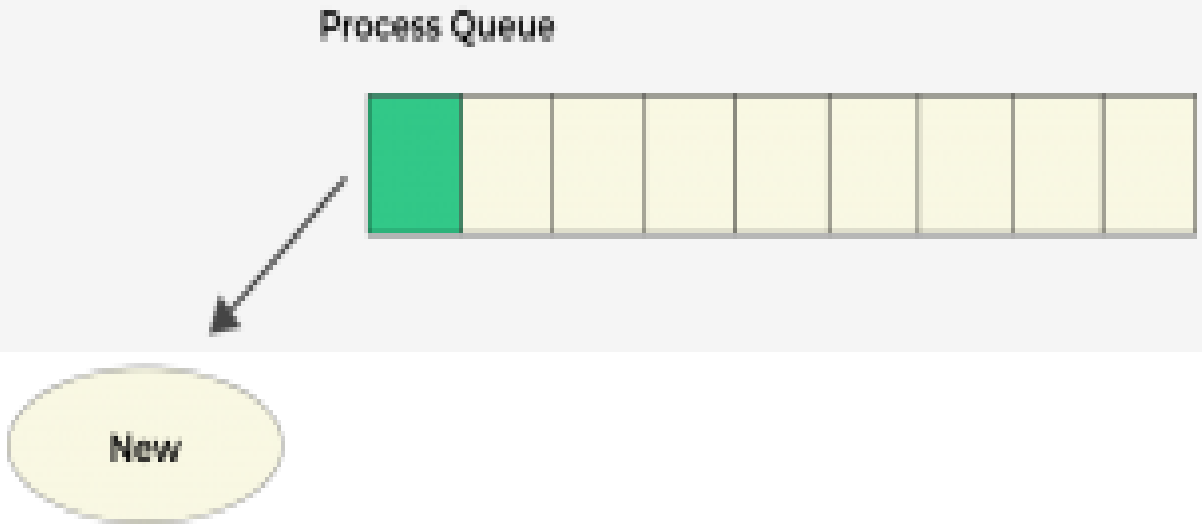
**5. Terminated or Exit** Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory.

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

New State

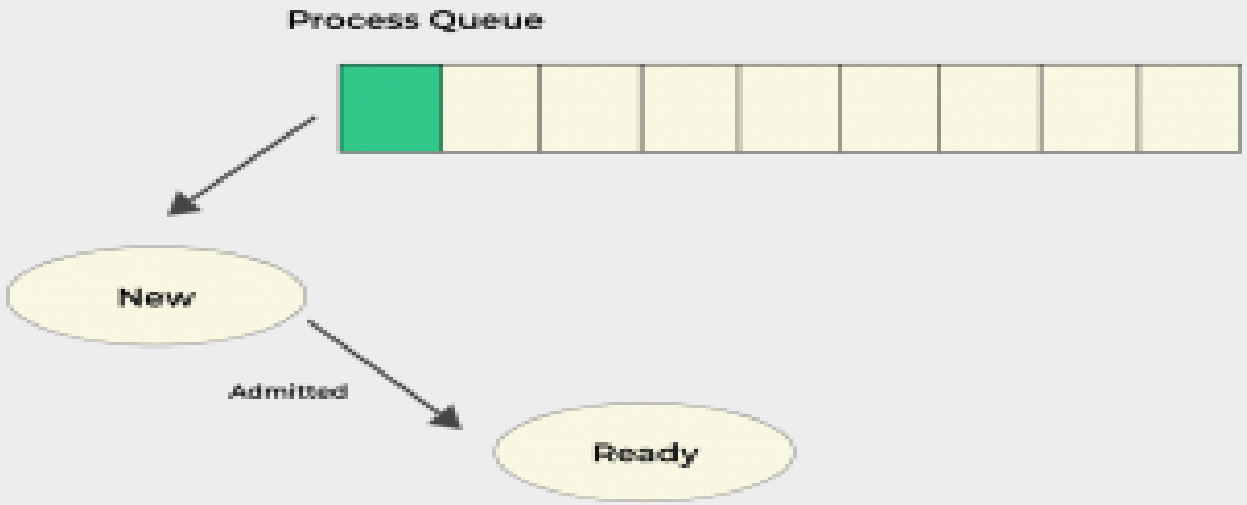
•Process it submitted to the process queue, it in turns acknowledges submission.

•Once submission is acknowledged, the process is given **new status.**



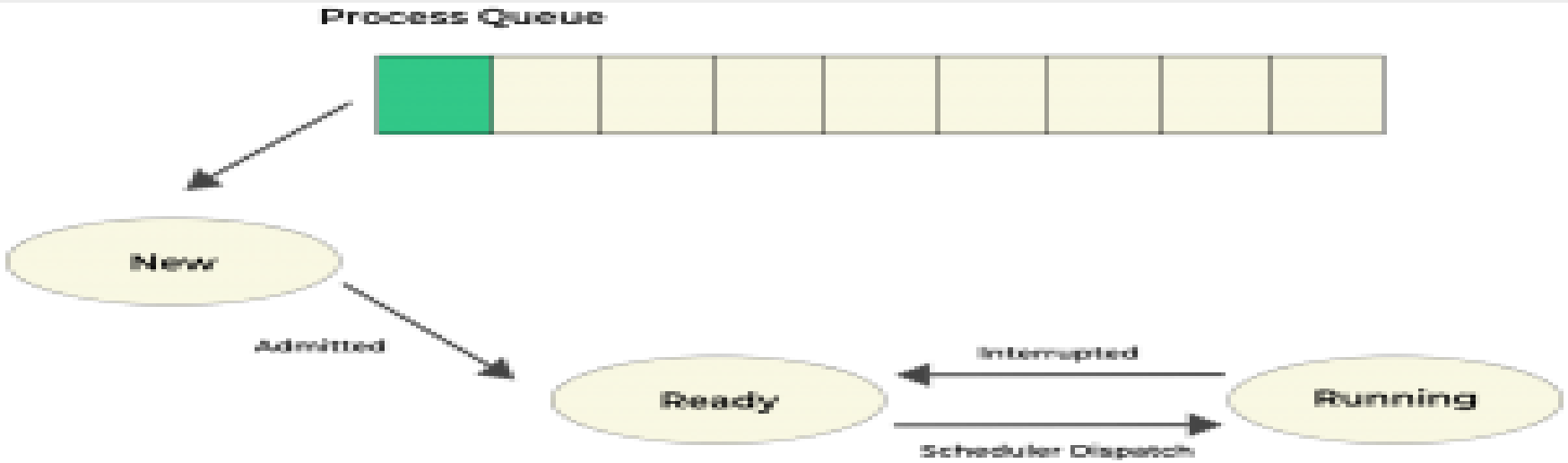
**Ready State**

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




**Running State**

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



**Wait and Termination State**

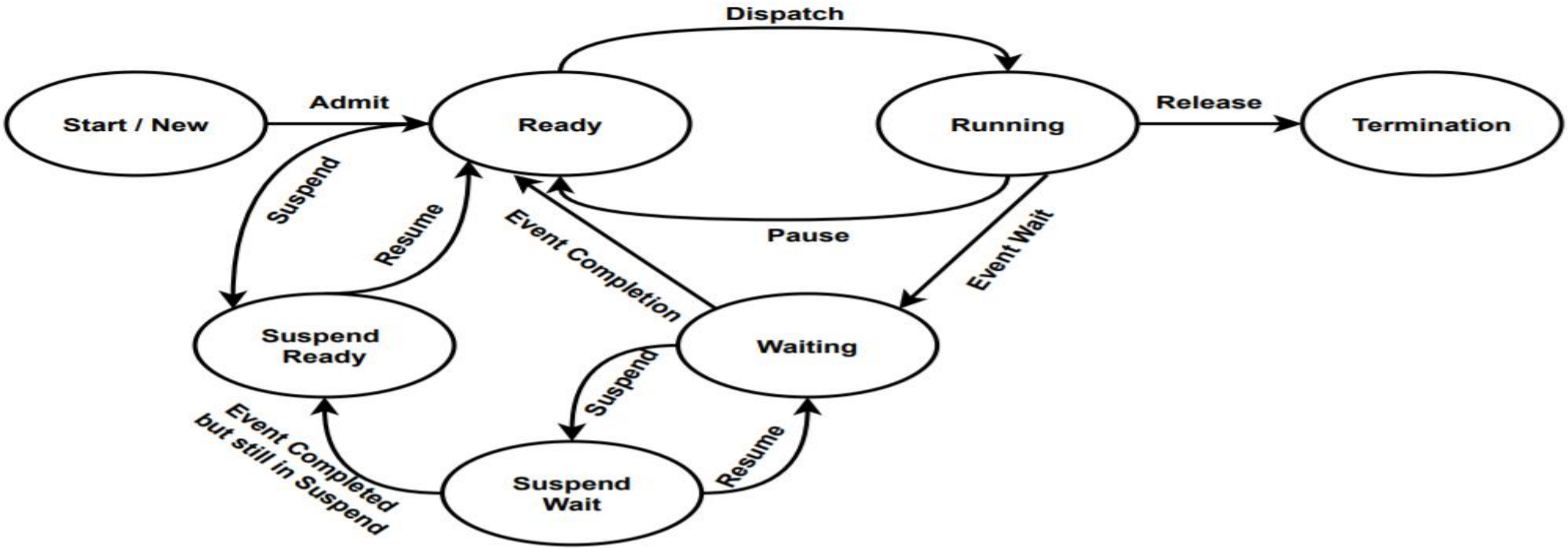
- Now the process can follow the following transitions –
  - The process may have all resources it needs and may get directly executed and goes to **Termination State**.
  - Process may need to go to **waiting state** any of the following
    - Access to Input/Output device (Asking user the values that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

## Summary

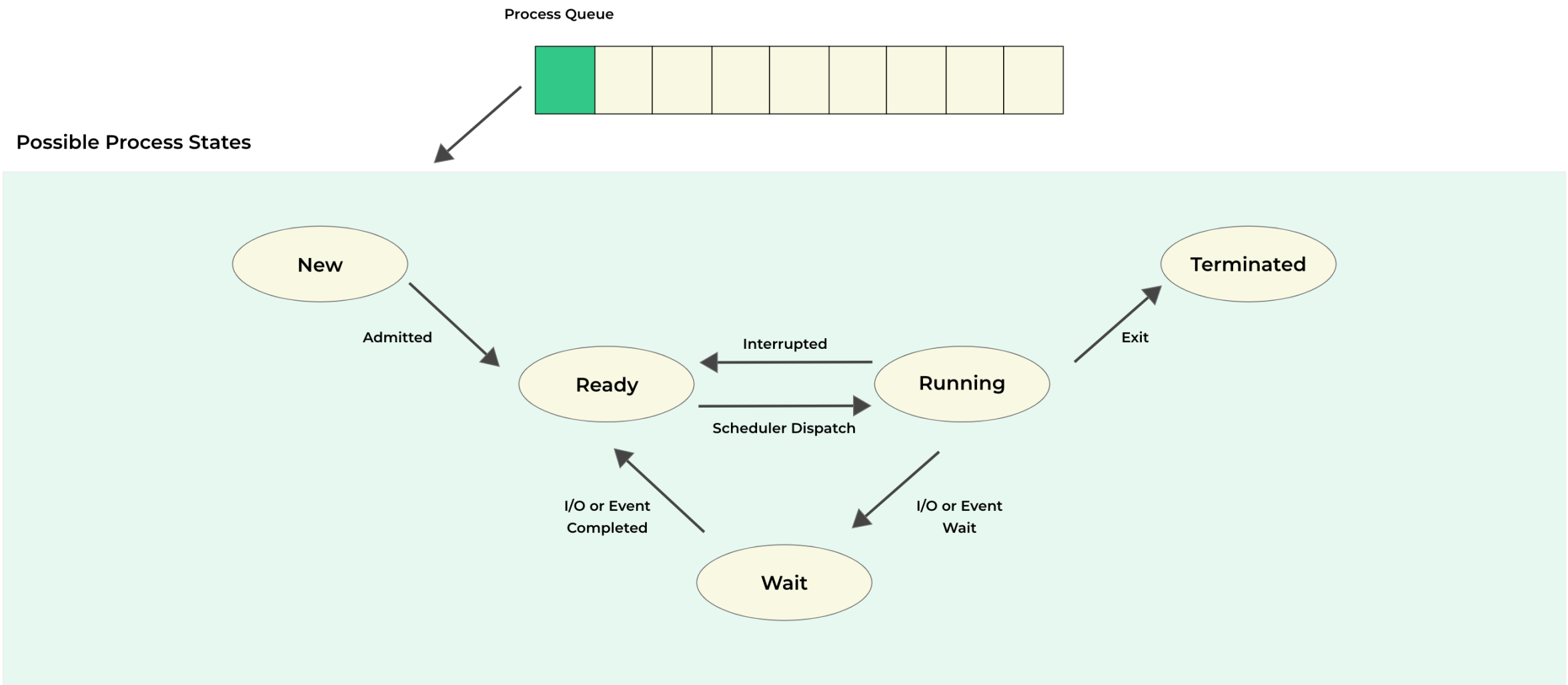
**1.New –** New Process Created
**2.Ready –** Process Ready for Processor/computing power allocation
**3.Running –** Process getting executing
**4.Wait –** Process waiting for signal
**5.Terminated –** Process execution completed

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

**1.Suspended Ready –** There maybe no possibility to add a new process in the queue. In such cases its can be said to be suspended ready state.
**2.Suspended Block –** If the waiting queue is full



# Process Lifecyle in OS



## Process Schedulers

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

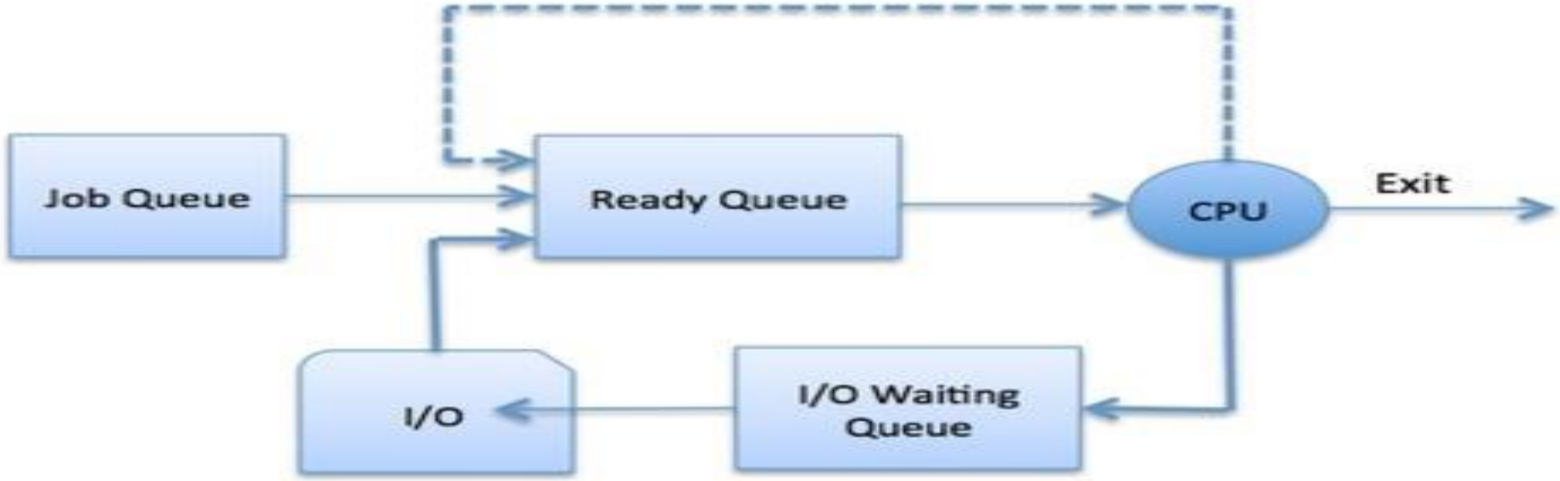
Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

The Operating System maintains the following important process scheduling queues –

**Job queue –** This queue keeps all the processes in the system.

**Ready queue –** This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.

**Device queues –** The processes which are blocked due to unavailability of an I/O device constitute this queue.



# Scheduling

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

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

There are three types of process scheduler.

1. **Long Term or job scheduler :**
   It brings the new process to the 'Ready State'. It controls ***Degree of Multi-programming***, i.e., number of process present in ready state at any point of time. It is important that the long-term scheduler make a careful selection of both IO and CPU bound process. IO bound tasks are which use much of their time in input and output operations while CPU bound processes are which spend their time on CPU. The job scheduler increases efficiency by maintaining a balance between the two.

2. **Short term or CPU scheduler :**
   It is responsible for selecting one process from ready state for scheduling it on the running state. Note: Short-term scheduler only selects the process to schedule it doesn't load the process on running. Here is when all the scheduling algorithms are used. The CPU scheduler is responsible for ensuring there is no starvation owing to high burst time processes.
   ***Dispatcher*** is responsible for loading the process selected by Short-term scheduler on the CPU (Ready to Running State) Context switching is done by dispatcher only. A dispatcher does the following:
   0. Switching context.
   1. Switching to user mode.
   2. Jumping to the proper location in the newly loaded program.
3. **Medium-term scheduler :**
   It is responsible for suspending and resuming the process. It mainly does swapping (moving processes from main memory to disk and vice versa). Swapping may be necessary to improve the process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up. It is helpful in maintaining a perfect balance between the I/O bound and the CPU bound. It reduces the degree of multiprogramming.

Scheduling of processes/work is done to finish the work on time.

Below is different time with respect to a process.

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

**Why do we need scheduling?**

A typical process involves both I/O time and CPU time. In a uni programming system like MS-DOS, time spent waiting for I/O is wasted and CPU is free during this time. In multi programming systems, one process can use CPU while another is waiting for I/O. This is possible only with process scheduling.

# Different Scheduling Algorithms

## First Come First Serve Method

**First Come First Serve (FCFS)** is an operating system scheduling algorithm that automatically executes queued requests and processes in order of their arrival. It is the easiest and simplest CPU scheduling algorithm. In this type of algorithm, processes which requests the CPU first get the CPU allocation first. This is managed with a FIFO queue. The full form of FCFS is First Come First Serve.
As the process enters the ready queue, its PCB (Process Control Block) is linked with the tail of the queue and, when the CPU becomes free, it should be assigned to the process at the beginning of the queue.

## Characteristics of FCFS method

- It supports non-preemptive and pre-emptive scheduling algorithm.
- Jobs are always executed on a first-come, first-serve basis.
- It is easy to implement and use.
- This method is poor in performance, and the general wait time is quite high.

## Example of FCFS scheduling

A real-life example of the FCFS method is buying a movie ticket on the ticket counter. In this scheduling algorithm, a person is served according to the queue manner. The person who arrives first in the queue first buys the ticket and then the next one. This will continue until the last person in the queue purchases the ticket. Using this algorithm, the CPU process works in a similar manner.

## How FCFS Works? Calculating Average Waiting Time

Here is an example of five processes arriving at different times. Each process has a different burst time.

| Process | Burst time | Arrival time |
|---|---|---|
| P1 | 6 | 2 |
| P2 | 2 | 5 |
| P3 | 8 | 1 |
| P4 | 3 | 0 |
| P5 | 4 | 4 |

Using the FCFS scheduling algorithm, these processes are handled as follows.

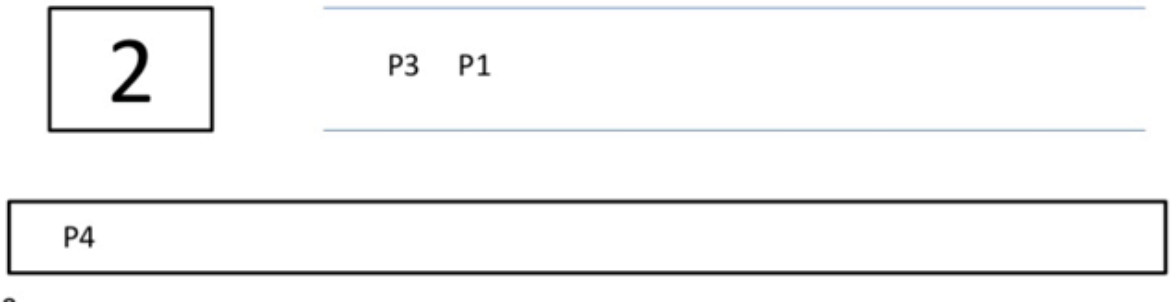
**Step 0)** The process begins with P4 which has arrival time 0

**Step 1)** At time=1, P3 arrives. P4 is still executing. Hence, P3 is kept in a queue.

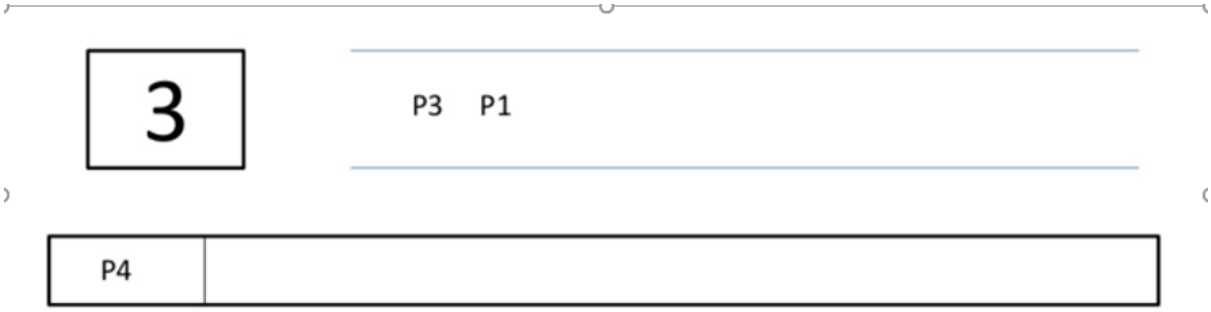
| Process | Burst time | Arrival time |
|---|---|---|
| P1 | 6 | 2 |
| P2 | 2 | 5 |
| P3 | 8 | 1 |
| P4 | 3 | 0 |
| P5 | 4 | 4 |

**Step 2)** At time= 2, P1 arrives which is kept in the queue.

| Process | Burst time | Arrival time |
|---|---|---|
| P1 | 6 | 2 |
| P2 | 2 | 5 |
| P3 | 8 | 1 |
| P4 | 3 | 0 |
| P5 | 4 | 4 |

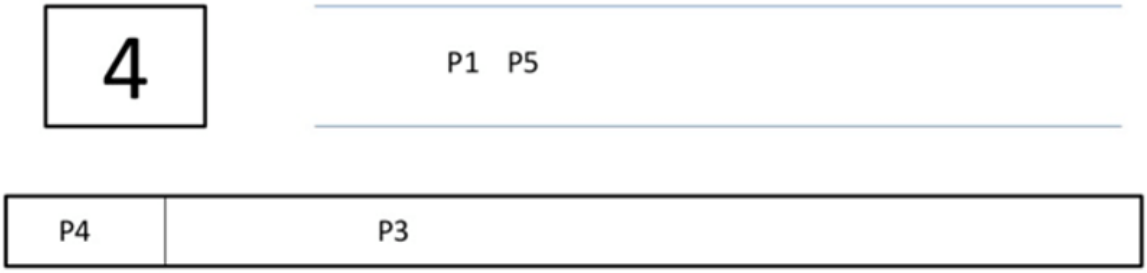



**Step 3)** At time=3, P4 process completes its execution.



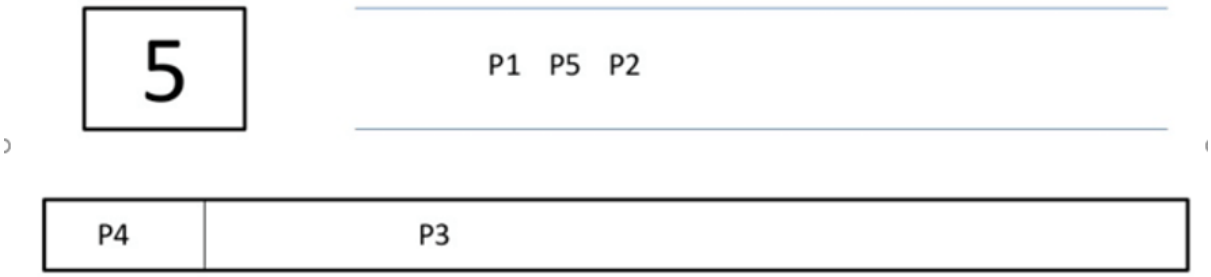

**Step 4)** At time=4, P3, which is first in the queue, starts execution.

| Process | Burst time | Arrival time |
|---|---|---|
| P1 | 6 | 2 |
| P2 | 2 | 5 |
| P3 | 8 | 1 |
| P4 | 3 | 0 |
| P5 | 4 | 4 |

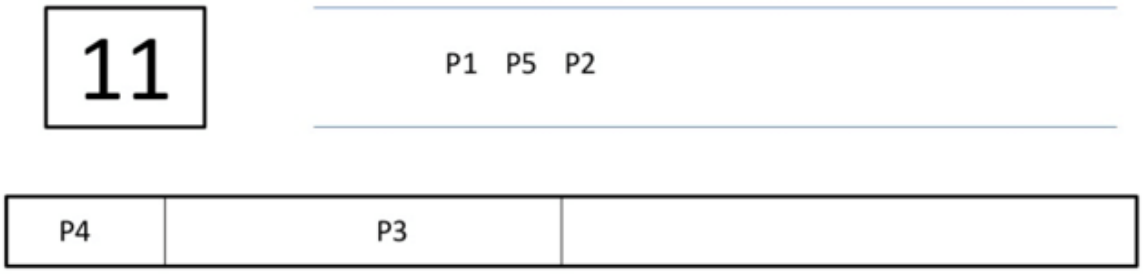



**Step 5)** At time =5, P2 arrives, and it is kept in a queue.

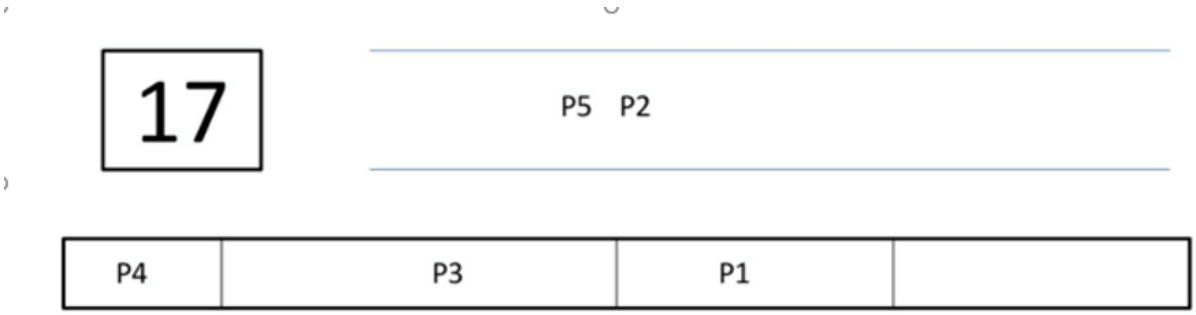
| Process | Burst time | Arrival time |
|---|---|---|
| P1 | 6 | 2 |
| P2 | 2 | 5 |
| P3 | 8 | 1 |
| P4 | 3 | 0 |
| P5 | 4 | 4 |



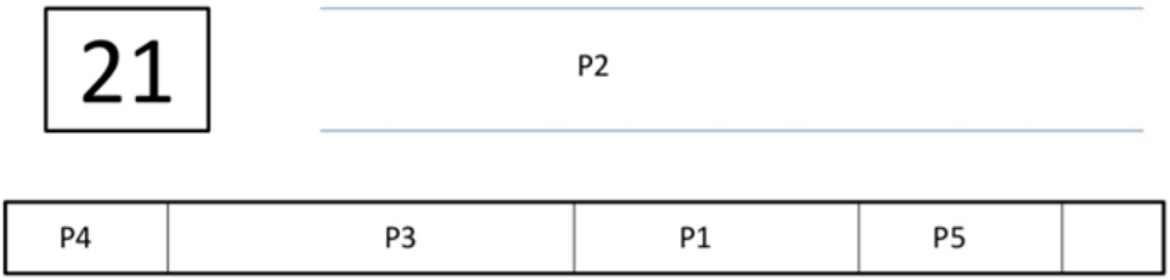

**Step 6)** At time 11, P3 completes its execution.



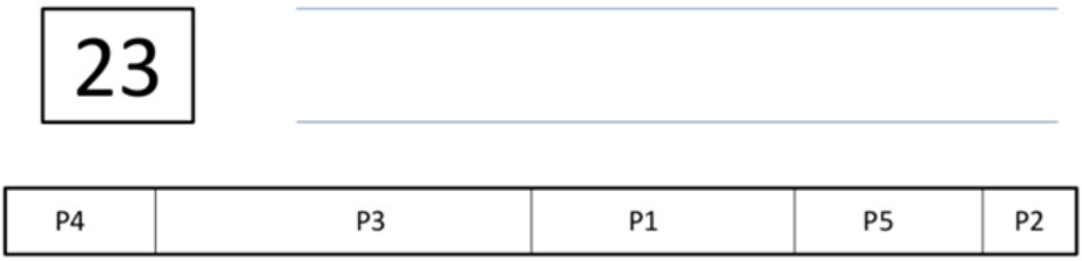

**Step 7)** At time=11, P1 starts execution. It has a burst time of 6. It completes execution at time interval 17



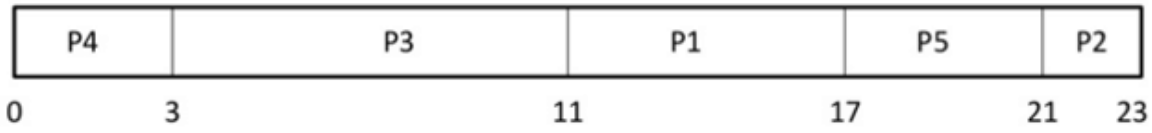

**Step 8)** At time=17, P5 starts execution. It has a burst time of 4. It completes execution at time=21




**Step 9)** At time=21, P2 starts execution. It has a burst time of 2. It completes execution at time interval 23




**Step 10)** Let's calculate the average waiting time for above example.




***Waiting time = Start time - Arrival time***

P4 = 0-0 = 0

P3 = 3-1 = 2

PI = 11-2 = 9

P5= 17-4 = 13

P2= 21-5= 16

Average Waiting Time

$$\frac{0+2+9+13+16}{5}$$

= 40/5= 8

## Advantages of FCFS

Here, are pros/benefits of using FCFS scheduling algorithm:

- The simplest form of a CPU scheduling algorithm
- Easy to program
- First come first served

## Disadvantages of FCFS

Here, are cons/ drawbacks of using FCFS scheduling algorithm:

- It is a Non-Preemptive CPU scheduling algorithm, so after the process has been allocated to the CPU, it will never release the CPU until it finishes executing.
- The Average Waiting Time is high.
- Short processes that are at the back of the queue have to wait for the long process at the front to finish.
- Not an ideal technique for time-sharing systems.
- Because of its simplicity, FCFS is not very efficient.

## Summary:

- Definition: FCFS is an operating system scheduling algorithm that automatically executes queued requests and processes by order of their arrival
- It supports non-preemptive and pre-emptive scheduling
- algorithm.
- FCFS stands for First Come First Serve
- A real-life example of the FCFS method is buying a movie ticket on the ticket counter.
- It is the simplest form of a CPU scheduling algorithm
- It is a Non-Preemptive CPU scheduling algorithm, so after the process has been allocated to the CPU, it will never release the CPU until it finishes executing.

# Priority Scheduling

**Priority Scheduling** is a method of scheduling processes that is based on priority. In this algorithm, the scheduler selects the tasks to work as per the priority.

The processes with higher priority should be carried out first, whereas jobs with equal priorities are carried out on a round-robin or FCFS basis. Priority depends upon memory requirements, time requirements, etc.

# Types of Priority Scheduling

Priority scheduling divided into two main types:

## Preemptive Scheduling

In Preemptive Scheduling, the tasks are mostly assigned with their priorities. Sometimes it is important to run a task with a higher priority before another lower priority task, even if the lower priority task is still running. The lower priority task holds for some time and resumes when the higher priority task finishes its execution.

## Non-Preemptive Scheduling

In this type of scheduling method, the CPU has been allocated to a specific process. The process that keeps the CPU busy, will release the CPU either by switching context or terminating. It is the only method that can be used for various hardware platforms. That's because it doesn't need special hardware (for example, a timer) like preemptive scheduling.

# Characteristics of Priority Scheduling

- A CPU algorithm that schedules processes based on priority.
- It used in Operating systems for performing batch processes.

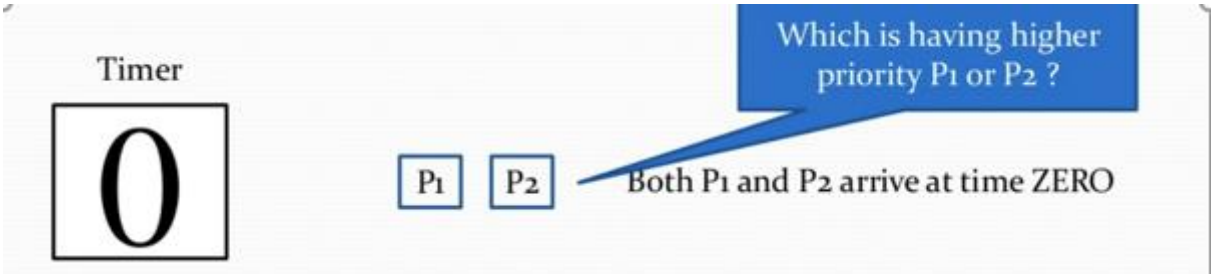
- If two jobs having the same priority are READY, it works on a FIRST COME, FIRST SERVED basis.
- In priority scheduling, a number is assigned to each process that indicates its priority level.
- Lower the number, higher is the priority.
- In this type of scheduling algorithm, if a newer process arrives, that is having a higher priority than the currently running process, then the currently running process is preempted.

## Example of Priority Scheduling

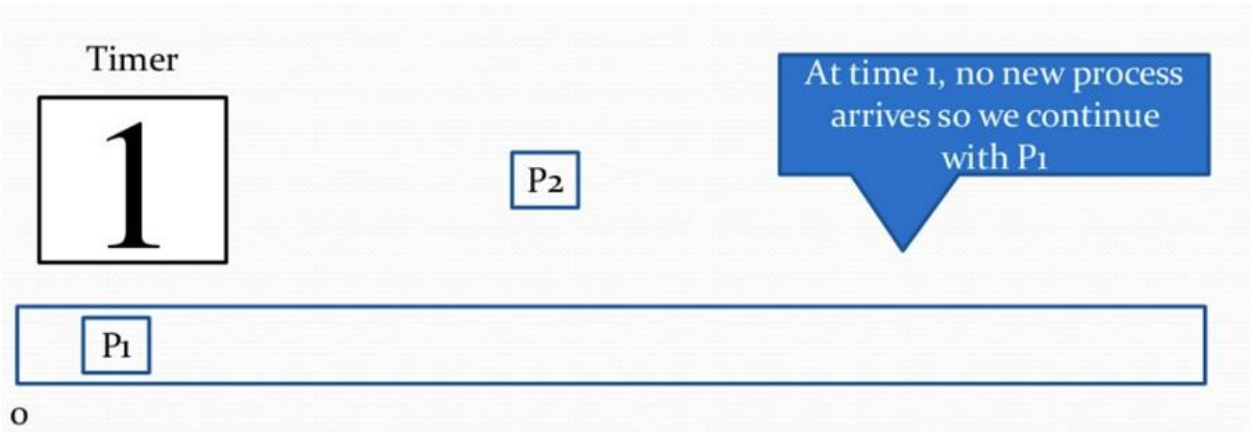
Consider following five processes P1 to P5. Each process has its unique priority, burst time, and arrival time.

| Process | Priority | Burst time | Arrival time |
|---|---|---|---|
| P1 | 1 | 4 | 0 |
| P2 | 2 | 3 | 0 |
| P3 | 1 | 7 | 6 |
| P4 | 3 | 4 | 11 |
| P5 | 2 | 2 | 12 |

**Step 0)** At time=0, Process P1 and P2 arrive. P1 has higher priority than P2. The execution begins with process P1, which has burst time 4.

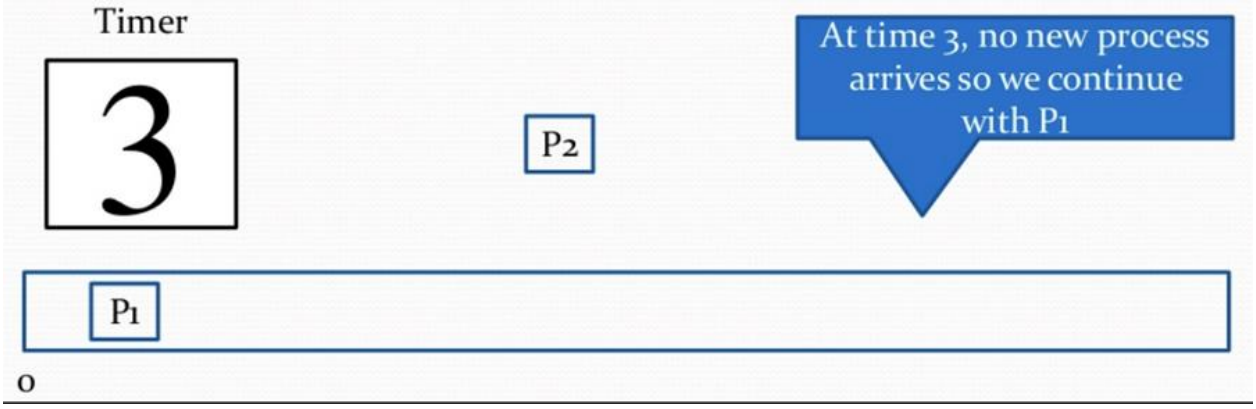



**Step 1)** At time=1, no new process arrive. Execution continues with P1.




**Step 2)** At time 2, no new process arrives, so you can continue with P1. P2 is in the waiting queue.

**Step 3)** At time 3, no new process arrives so you can continue with P1. P2 process still in the waiting queue.

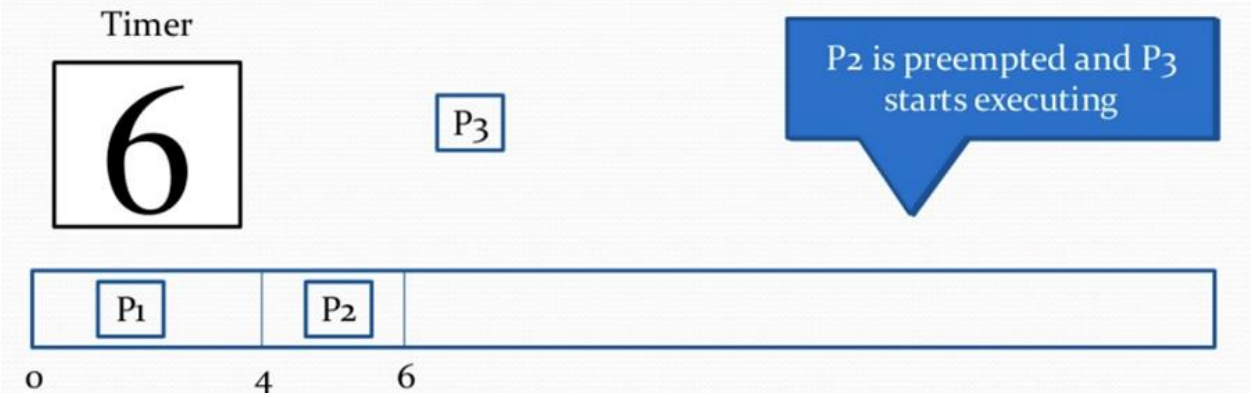



**Step 4)** At time 4, P1 has finished its execution. P2 starts execution.

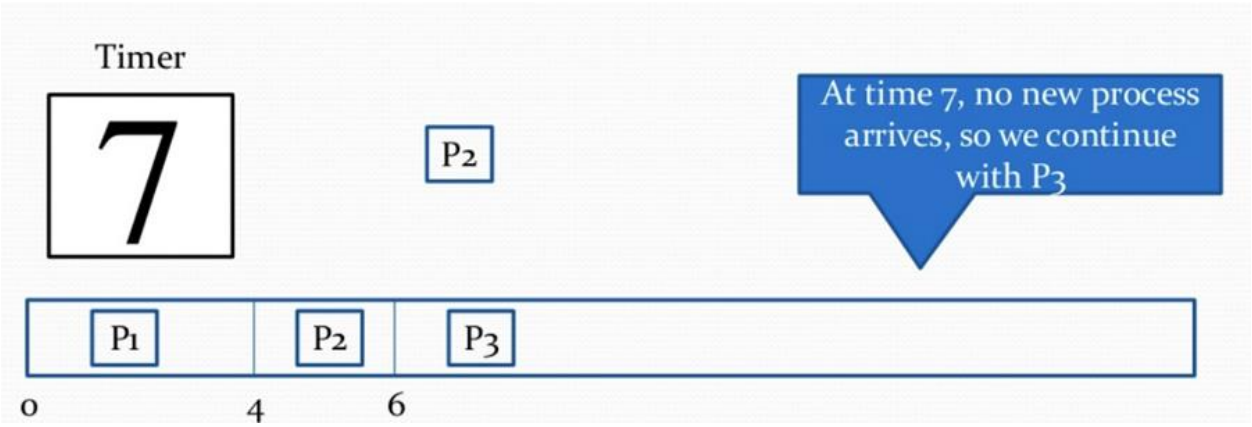
**Step 5)** At time= 5, no new process arrives, so we continue with P2.

**Step 6)** At time=6, P3 arrives. P3 is at higher priority (1) compared to P2 having priority (2). P2 is preempted, and P3 begins its execution.

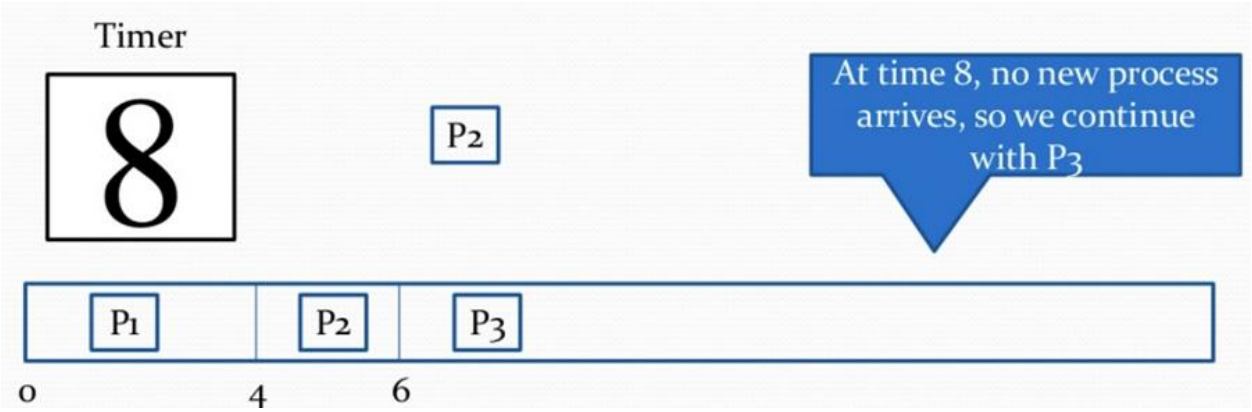
| Process | Priority | Burst time | Arrival time |
|---|---|---|---|
| P1 | 1 | 4 | 0 |
| P2 | 2 | 1 out of 3 pending | 0 |
| P3 | 1 | 7 | 6 |
| P4 | 3 | 4 | 11 |
| P5 | 2 | 2 | 12 |



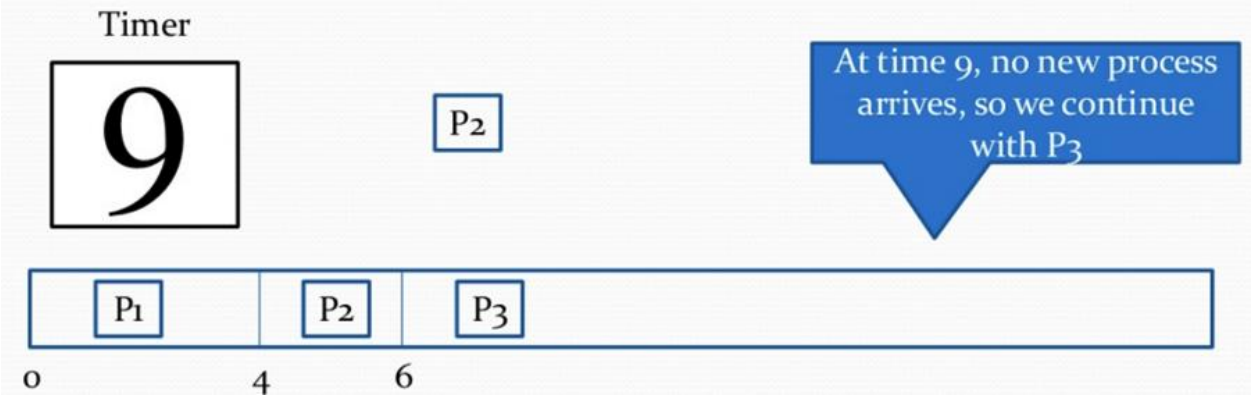

**Step 7) At** time 7, no-new process arrives, so we continue with P3. P2 is in the waiting queue.



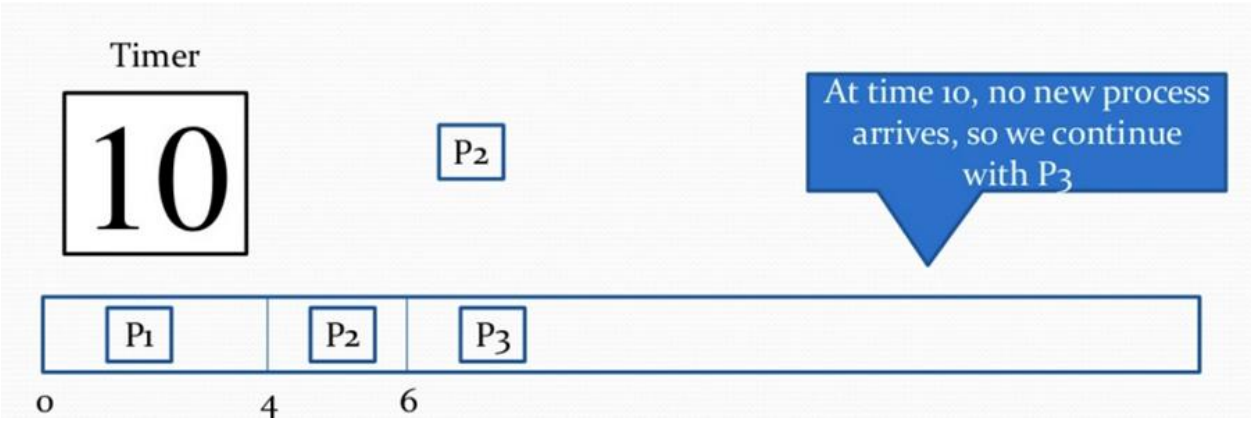

**Step 8)** At time= 8, no new process arrives, so we can continue with P3.




**Step 9)** At time= 9, no new process comes so we can continue with P3.

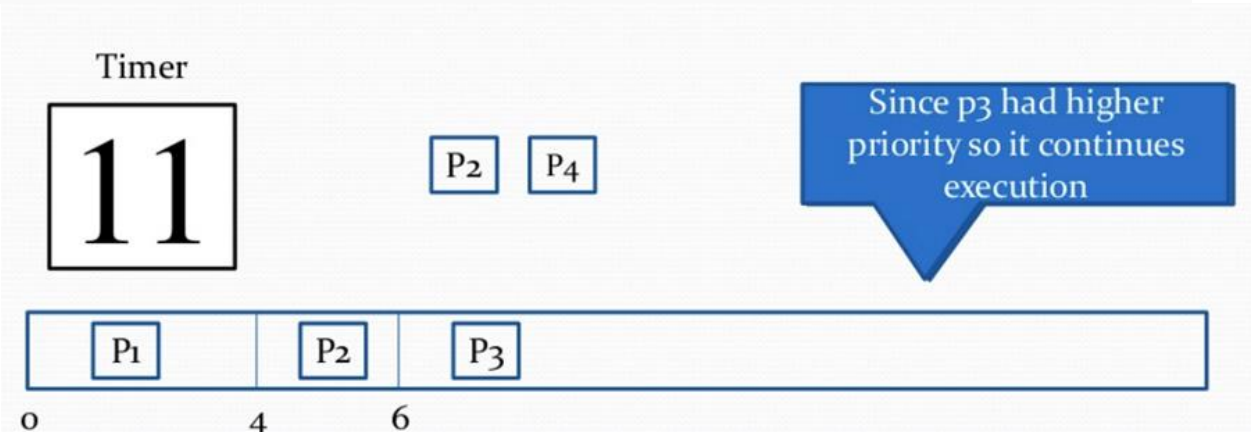



**Step 10)** At time interval 10, no new process comes, so we continue with P3

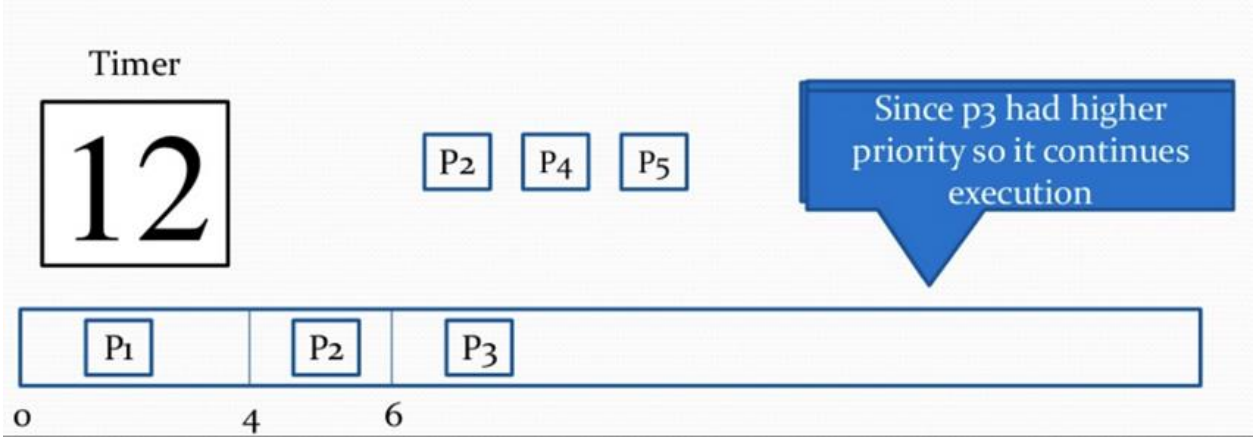



**Step 11)** At time=11, P4 arrives with priority 4. P3 has higher priority, so it continues its execution.

| Process | Priority | Burst time | Arrival time |
|---|---|---|---|
| P1 | 1 | 4 | 0 |
| P2 | 2 | 1 out of 3 pending | 0 |
| P3 | 1 | 2 out of 7 pending | 6 |
| P4 | 3 | 4 | 11 |
| P5 | 2 | 2 | 12 |

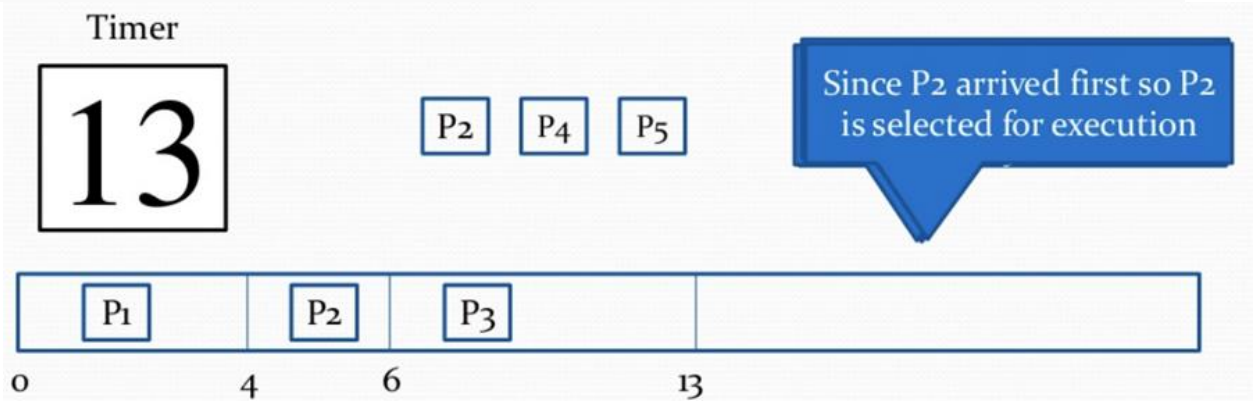



**Step 12)** At time=12, P5 arrives. P3 has higher priority, so it continues execution.

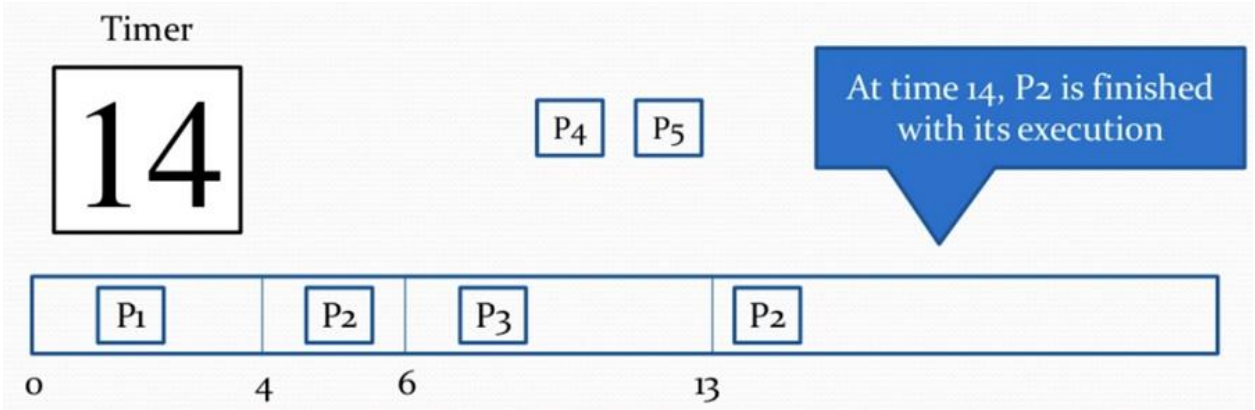



**Step 13)** At time=13, P3 completes execution. We have P2,P4,P5 in ready queue. P2 and P5 have equal priority. Arrival time of P2 is before P5. So P2 starts execution.

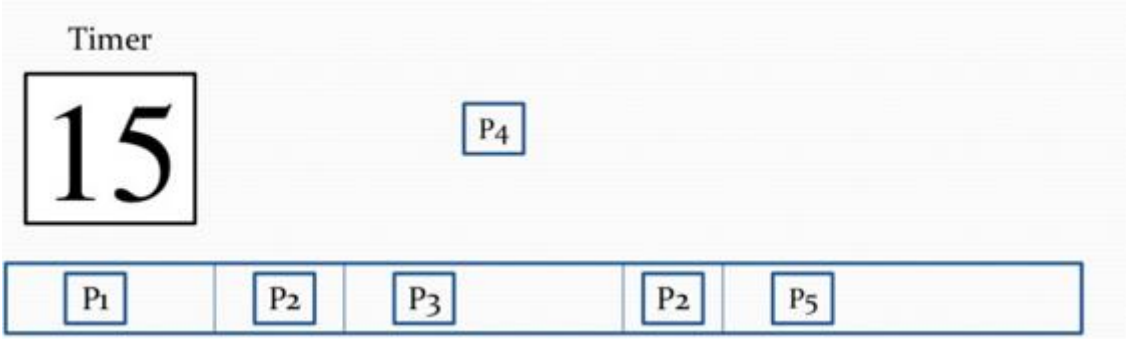
| Process | Priority | Burst time | Arrival time |
|---|---|---|---|
| P1 | 1 | 4 | 0 |
| P2 | 2 | 1 out of 3 pending | 0 |
| P3 | 1 | 7 | 6 |
| P4 | 3 | 4 | 11 |
| P5 | 2 | 2 | 12 |



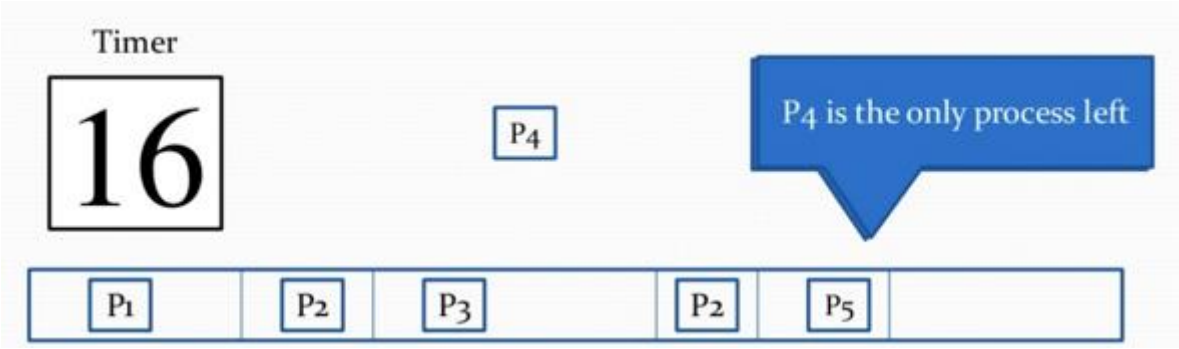

**Step 14)** At time =14, the P2 process has finished its execution. P4 and P5 are in the waiting state. P5 has the highest priority and starts execution.



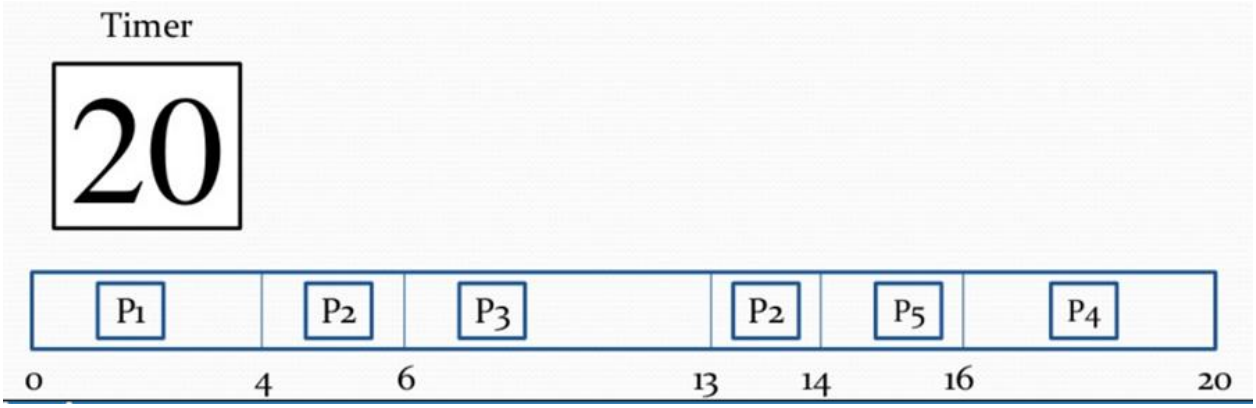

**Step 15)** At time =15, P5 continues execution.




**Step 16)** At time= 16, P5 is finished with its execution. P4 is the only process left. It starts execution.




**Step 17)** At time =20, P5 has completed execution and no process is left.




**Step 18)** Let's calculate the average waiting time for the above example.

Waiting Time = start time – arrival time + wait time for next burst

```
P1 = o - o = o
P2 =4 - o + 7 =11
P3= 6-6=0
P4= 16-11=5
Average Waiting time = (0+11+0+5+2)/5 = 18/5= 3.6
```

## Advantages of priority scheduling

Here, are benefits/pros of using priority scheduling method:

- Easy to use scheduling method
- Processes are executed on the basis of priority so high priority does not need to wait for long which saves time

- This method provides a good mechanism where the relative important of each process may be precisely defined.
- Suitable for applications with fluctuating time and resource requirements.

## Disadvantages of priority scheduling

Here, are cons/drawbacks of priority scheduling

- If the system eventually crashes, all low priority processes get lost.
- If high priority processes take lots of CPU time, then the lower priority processes may starve and will be postponed for an indefinite time.
- This scheduling algorithm may leave some low priority processes waiting indefinitely.
- A process will be blocked when it is ready to run but has to wait for the CPU because some other process is running currently.
- If a new higher priority process keeps on coming in the ready queue, then the process which is in the waiting state may need to wait for a long duration of time.

## Summary:

- Priority scheduling is a method of scheduling processes that is based on priority. In this algorithm, the scheduler selects the tasks to work as per the priority.
- In Priority Preemptive Scheduling, the tasks are mostly assigned with their priorities.
- In Priority Non-preemptive scheduling method, the CPU has been allocated to a specific process.
- Processes are executed on the basis of priority so high priority does not need to wait for long which saves time
- If high priority processes take lots of CPU time, then the lower priority processes may starve and will be postponed for an indefinite time.

## Shortest Job First Scheduling

**Shortest Job First (SJF)** is an algorithm in which the process having the smallest execution time is chosen for the next execution. This scheduling method can be preemptive or non-preemptive. It significantly reduces the average waiting time for other processes awaiting execution. The full form of SJF is Shortest Job First.
**There are basically two types of SJF methods:**

- Non-Preemptive SJF
- Preemptive SJF

## Characteristics of SJF Scheduling

- It is associated with each job as a unit of time to complete.

- This algorithm method is helpful for batch-type processing, where waiting for jobs to complete is not critical.
- It can improve process throughput by making sure that shorter jobs are executed first, hence possibly have a short turnaround time.
- It improves job output by offering shorter jobs, which should be executed first, which mostly have a shorter turnaround time.

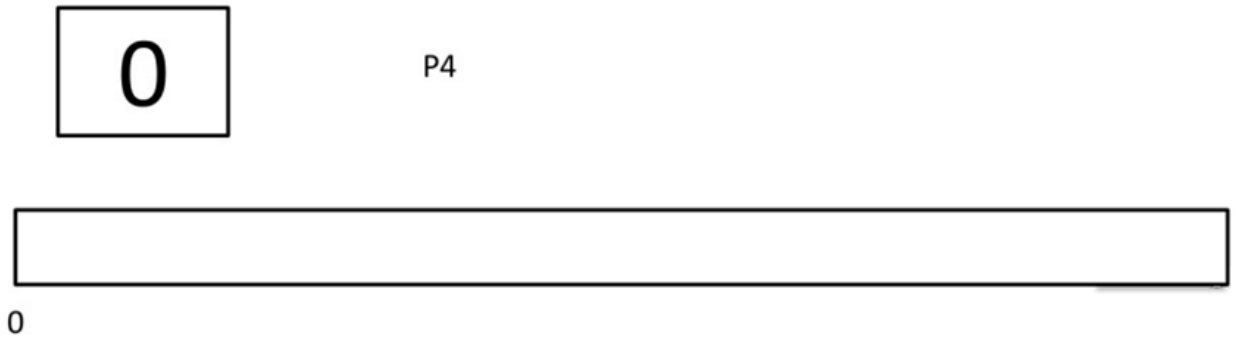
## Non-Preemptive SJF

In non-preemptive scheduling, once the CPU cycle is allocated to process, the process holds it till it reaches a waiting state or terminated.

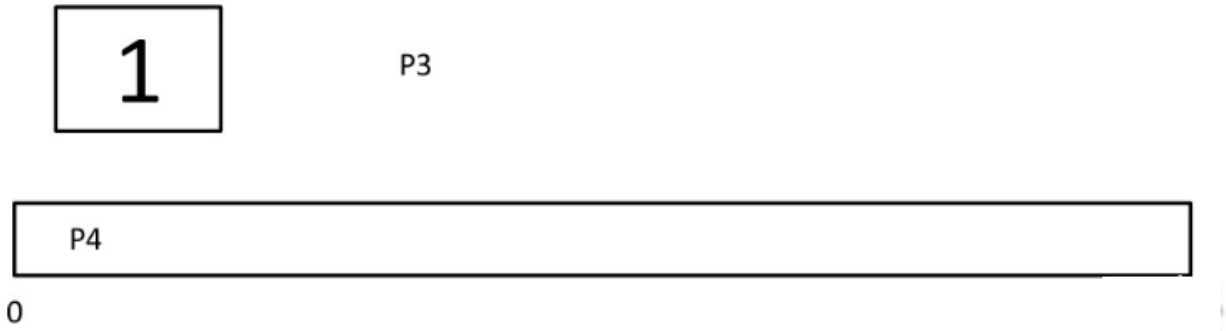
Consider the following five processes each having its own unique burst time and arrival time.

| Process Queue | Burst time | Arrival time |
|---|---|---|
| P1 | 6 | 2 |
| P2 | 2 | 5 |
| P3 | 8 | 1 |
| P4 | 3 | 0 |
| P5 | 4 | 4 |

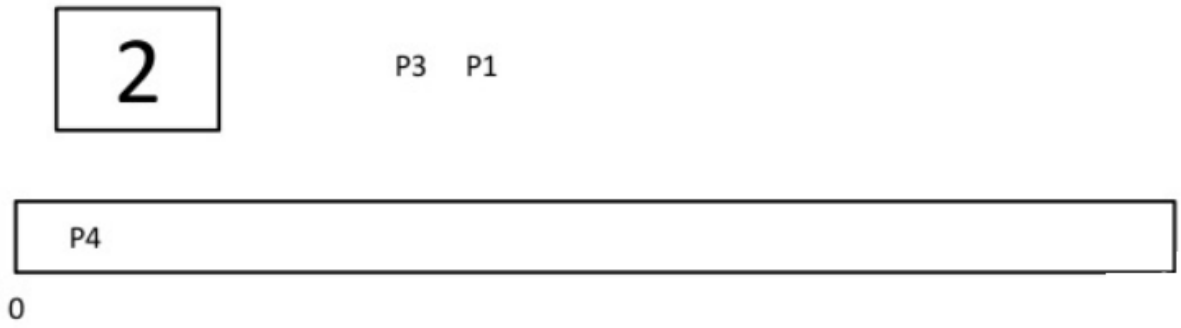
**Step 0)** At time=0, P4 arrives and starts execution.



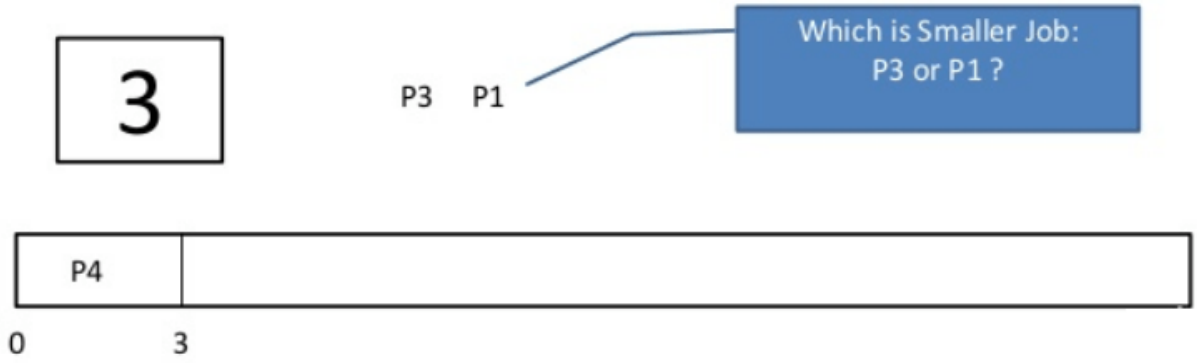

**Step 1)** At time= 1, Process P3 arrives. But, P4 still needs 2 execution units to complete. It will continue execution.



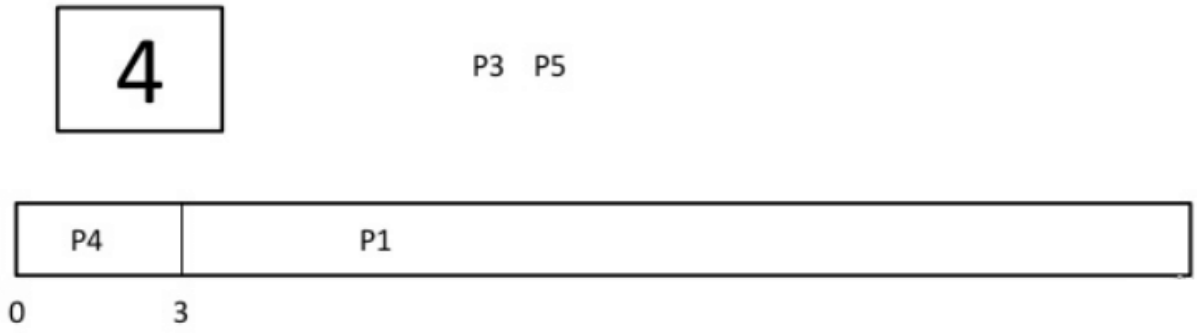

**Step 2)** At time =2, process P1 arrives and is added to the waiting queue. P4 will continue execution.



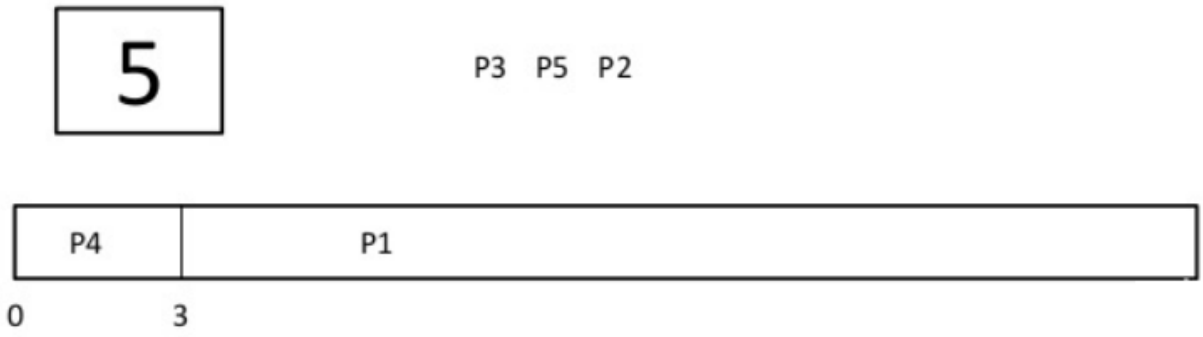

**Step 3)** At time = 3, process P4 will finish its execution. The burst time of P3 and P1 is compared. Process P1 is executed because its burst time is less compared to P3.



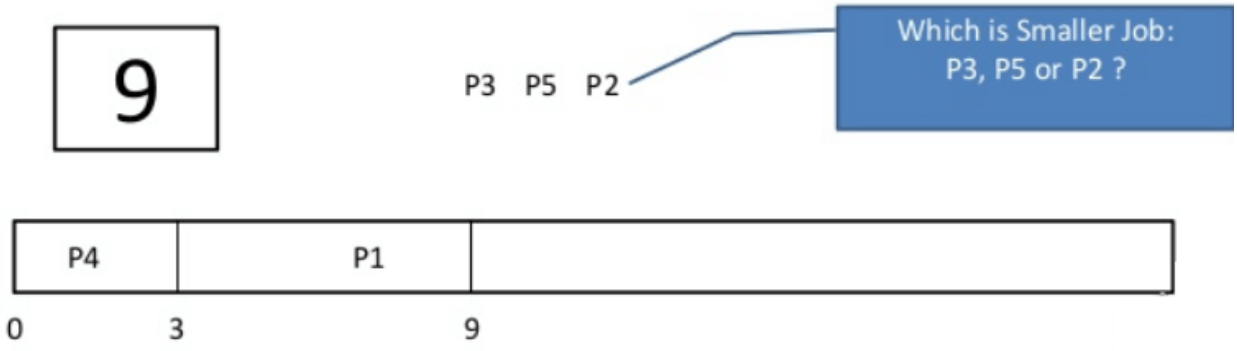

**Step 4)** At time = 4, process P5 arrives and is added to the waiting queue. P1 will continue execution.



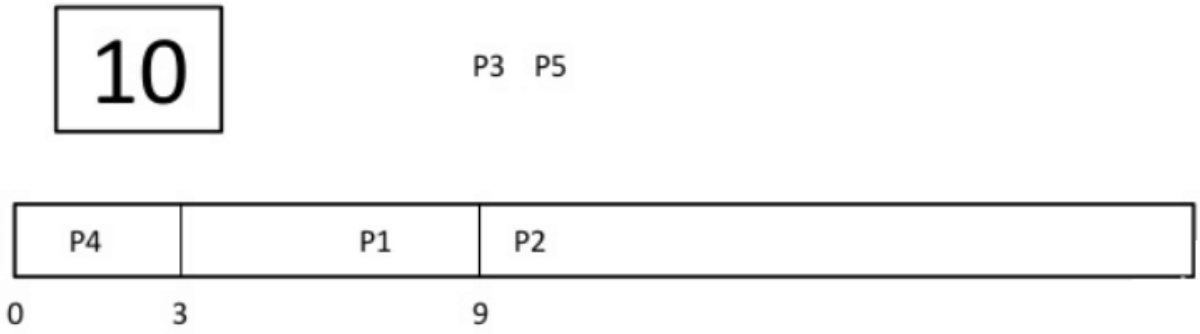

**Step 5)** At time = 5, process P2 arrives and is added to the waiting queue. P1 will continue execution.



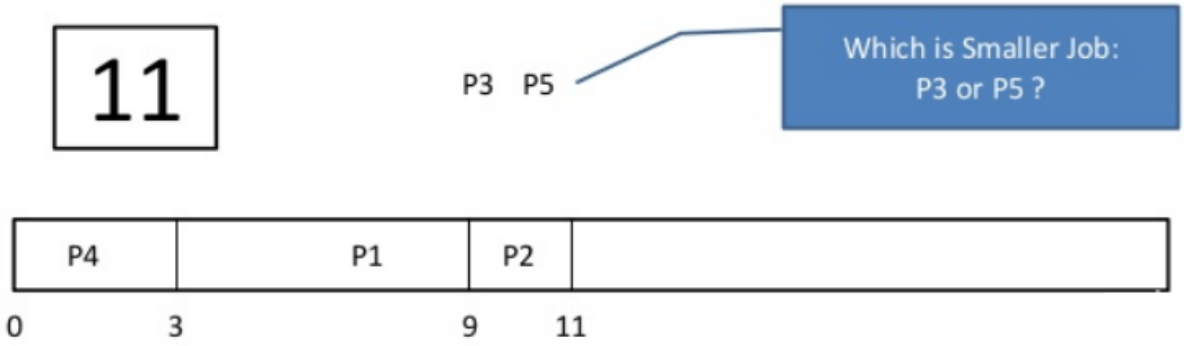
**Step 6)** At time = 9, process P1 will finish its execution. The burst time of P3, P5, and P2 is compared. Process P2 is executed because its burst time is the lowest.



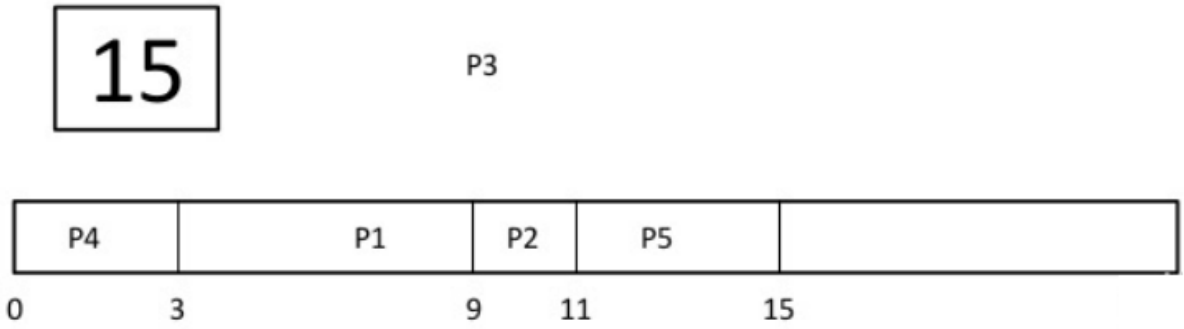

**Step 7)** At time=10, P2 is executing and P3 and P5 are in the waiting queue.



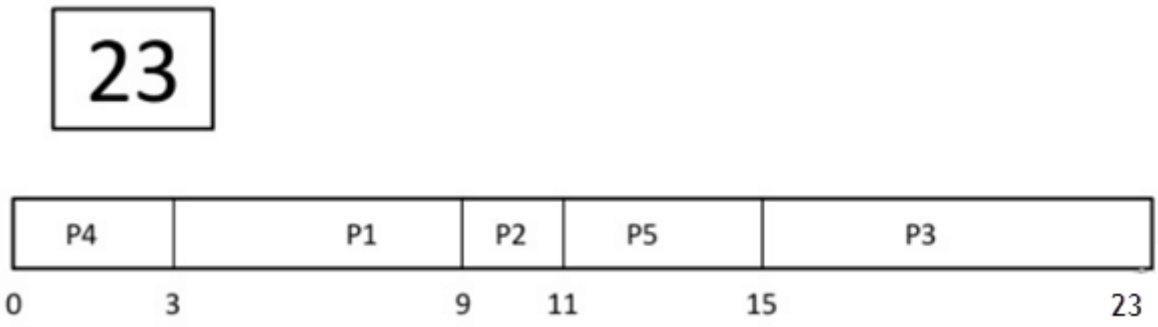

**Step 8)** At time = 11, process P2 will finish its execution. The burst time of P3 and P5 is compared. Process P5 is executed because its burst time is lower.




**Step 9)** At time = 15, process P5 will finish its execution.



**Step 10)** At time = 23, process P3 will finish its execution.




**Step 11)** Let's calculate the average waiting time for above example.

```
Wait time
P4= 0-0=0
P1=  3-2=1
P2= 9-5=4
P5= 11-4=7
P3= 15-1=14
Average Waiting Time= 0+1+4+7+14/5 = 26/5 = 5.2
```

# Preemptive SJF

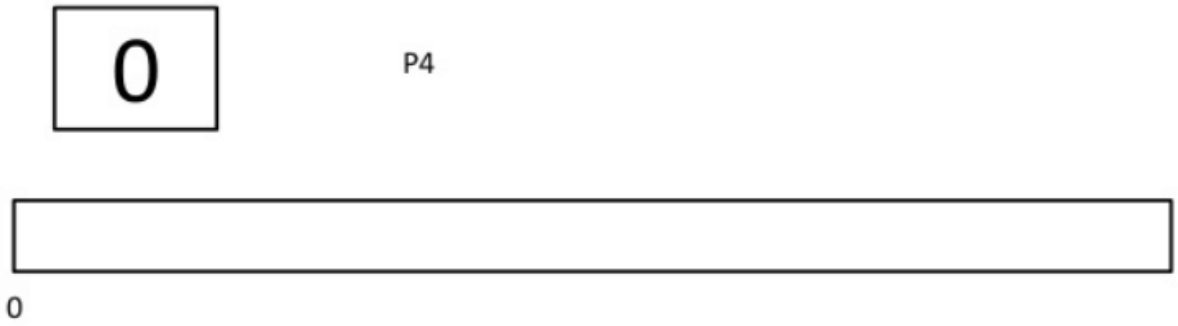
In Preemptive SJF Scheduling, jobs are put into the ready queue as they come. A process with shortest burst time begins execution. If a process with even a shorter burst time arrives, the current process is removed or preempted from execution, and the shorter job is allocated CPU cycle.

Consider the following five process:

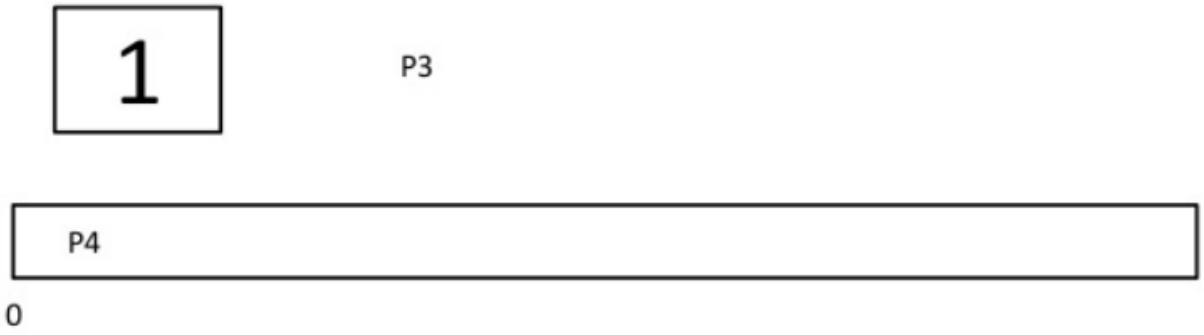
| Process Queue | Burst time | Arrival time |
|---|---|---|
| P1 | 6 | 2 |
| P2 | 2 | 5 |
| P3 | 8 | 1 |
| P4 | 3 | 0 |
| P5 | 4 | 4 |

**Step 0)** At time=0, P4 arrives and starts execution.

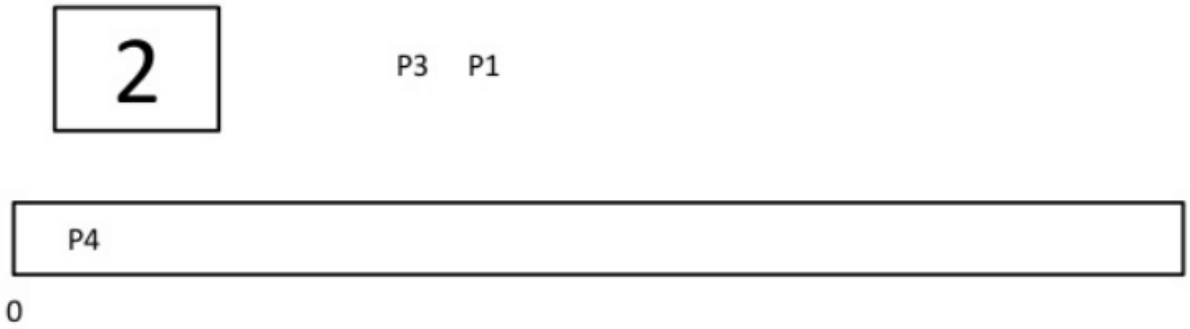
| Process Queue | Burst time | Arrival time |
|---|---|---|
| P1 | 6 | 2 |
| P2 | 2 | 5 |
| P3 | 8 | 1 |
| P4 | 3 | 0 |
| P5 | 4 | 4 |



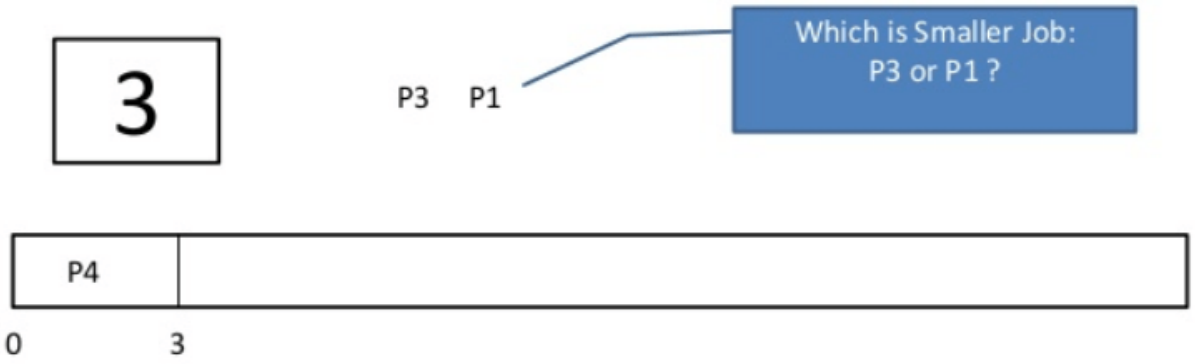

**Step 1)** At time= 1, Process P3 arrives. But, P4 has a shorter burst time. It will continue execution.




**Step 2)** At time = 2, process P1 arrives with burst time = 6. The burst time is more than that of P4. Hence, P4 will continue execution.

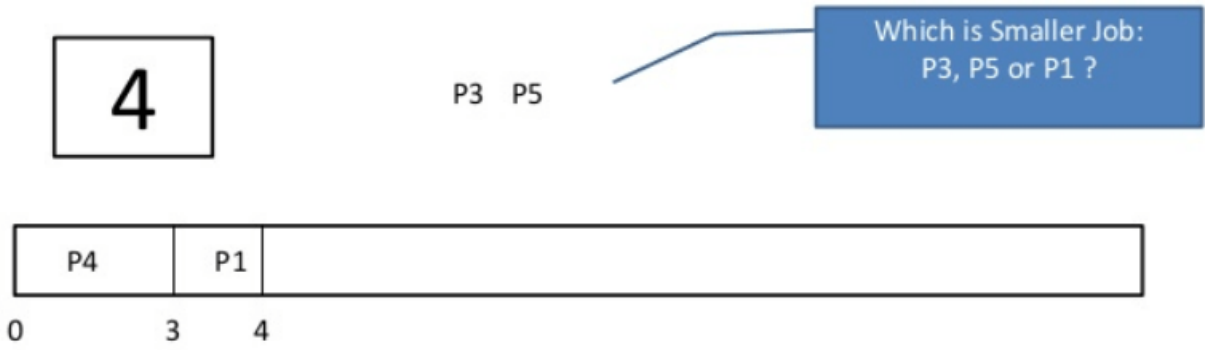



**Step 3)** At time = 3, process P4 will finish its execution. The burst time of P3 and P1 is compared. Process P1 is executed because its burst time is lower.



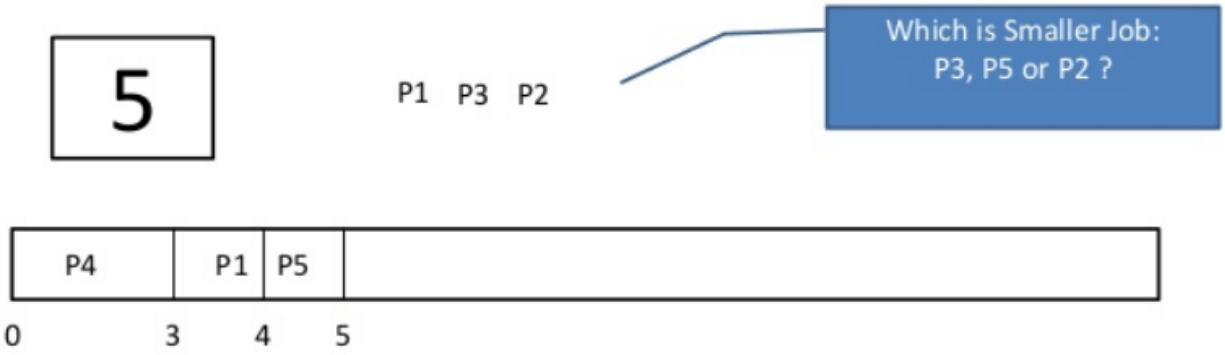

**Step 4)** At time = 4, process P5 will arrive. The burst time of P3, P5, and P1 is compared. Process P5 is executed because its burst time is lowest. Process P1 is preempted.

| Process Queue | Burst time | Arrival time |
|---|---|---|
| P1 | 5 out of 6 is remaining | 2 |
| P2 | 2 | 5 |
| P3 | 8 | 1 |
| P4 | 3 | 0 |
| P5 | 4 | 4 |

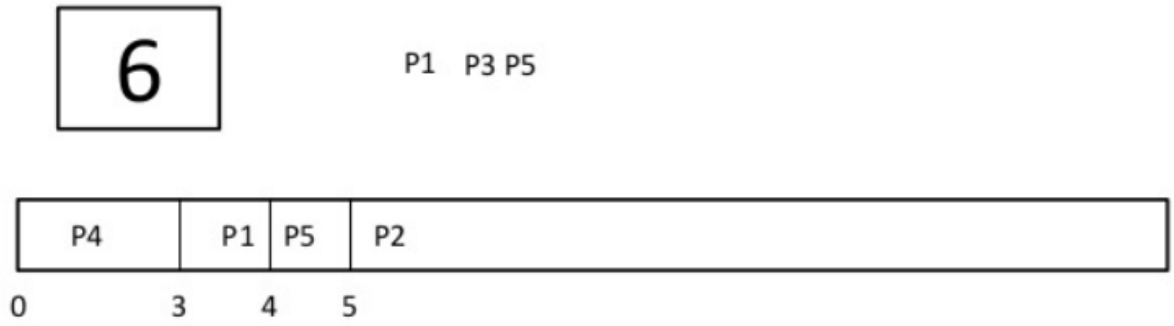



**Step 5)** At time = 5, process P2 will arrive. The burst time of P1, P2, P3, and P5 is compared. Process P2 is executed because its burst time is least. Process P5 is preempted.

| Process Queue | Burst time | Arrival time |
|---|---|---|
| P1 | 5 out of 6 is remaining | 2 |
| P2 | 2 | 5 |
| P3 | 8 | 1 |
| P4 | 3 | 0 |
| P5 | 3 out of 4 is remaining | 4 |

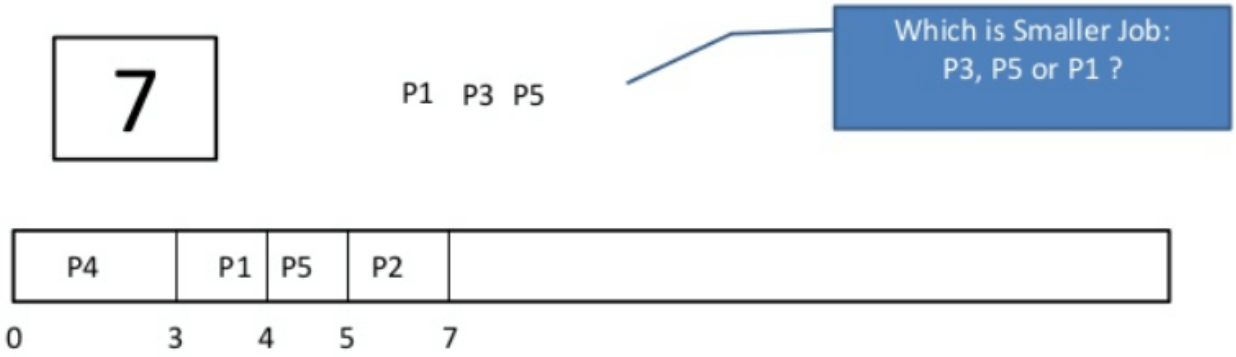



**Step 6)** At time =6, P2 is executing.

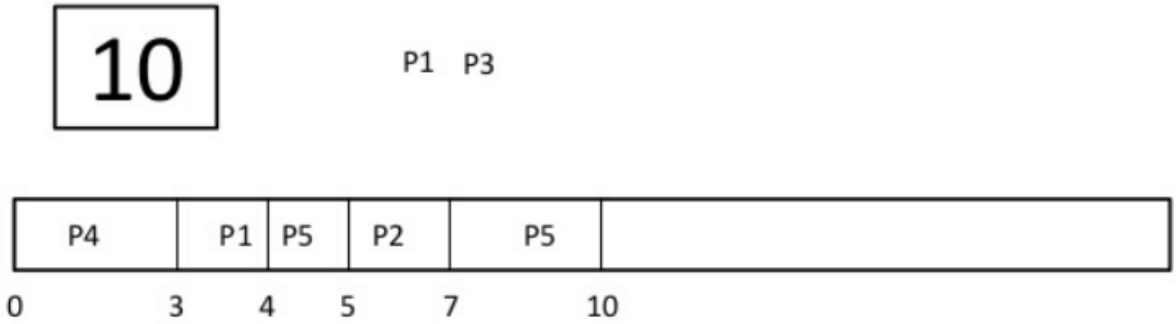



**Step 7)** At time =7, P2 finishes its execution. The burst time of P1, P3, and P5 is compared. Process P5 is executed because its burst time is lesser.

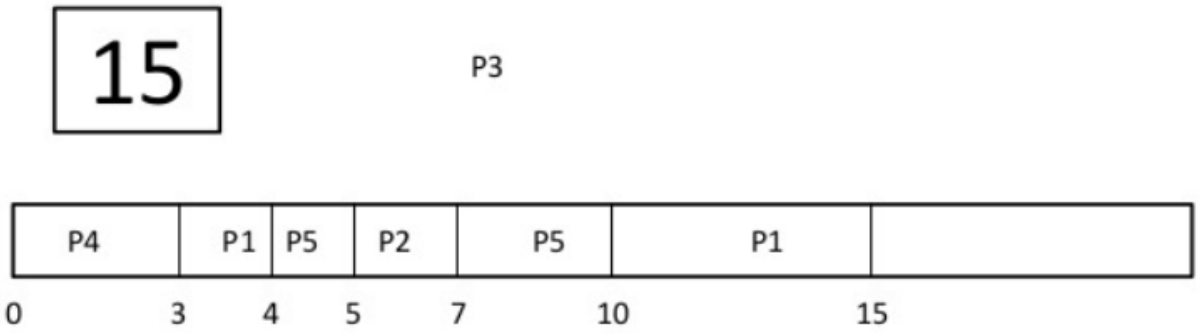
| Process Queue | Burst time | Arrival time |
|---|---|---|
| P1 | 5 out of 6 is remaining | 2 |
| P2 | 2 | 5 |
| P3 | 8 | 1 |
| P4 | 3 | 0 |
| P5 | 3 out of 4 is remaining | 4 |



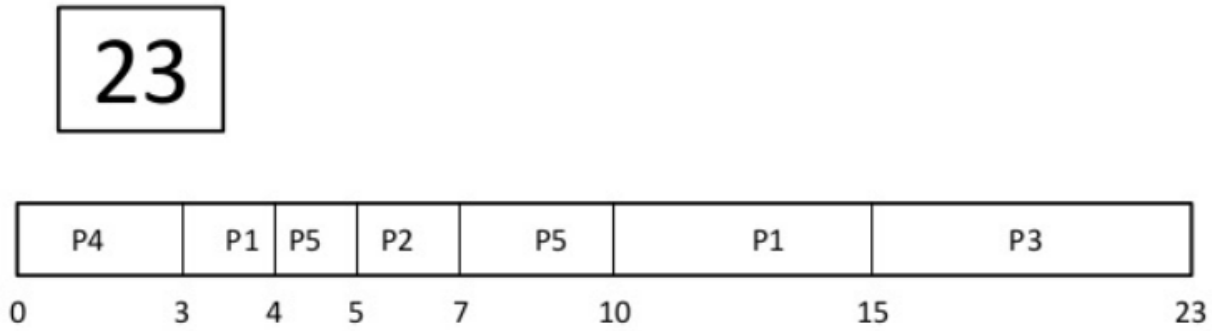

**Step 8)** At time =10, P5 will finish its execution. The burst time of P1 and P3 is compared. Process P1 is executed because its burst time is less.




**Step 9)** At time =15, P1 finishes its execution. P3 is the only process left. It will start execution.




**Step 10)** At time =23, P3 finishes its execution.




**Step 11)** Let's calculate the average waiting time for above example.

```
Wait time
P4= 0-0=0
P1=  (3-2) + 6 =7
P2= 5-5 = 0
P5= 4-4+2 =2
P3= 15-1 = 14
Average Waiting Time = 0+7+0+2+14/5 = 23/5 =4.6
```

## Advantages of SJF

Here are the benefits/pros of using SJF method:

- SJF is frequently used for long term scheduling.
- It reduces the average waiting time over FIFO (First in First Out) algorithm.
- SJF method gives the lowest average waiting time for a specific set of processes.
- It is appropriate for the jobs running in batch, where run times are known in advance.
- For the batch system of long-term scheduling, a burst time estimate can be obtained from the job description.
- For Short-Term Scheduling, we need to predict the value of the next burst time.
- Probably optimal with regard to average turnaround time.

## Disadvantages/Cons of SJF

Here are some drawbacks/cons of SJF algorithm:

- Job completion time must be known earlier, but it is hard to predict.
- It is often used in a batch system for long term scheduling.

- SJF can’t be implemented for CPU scheduling for the short term. It is because there is no specific method to predict the length of the upcoming CPU burst.
- This algorithm may cause very long turnaround times or starvation.
- Requires knowledge of how long a process or job will run.
- It leads to the starvation that does not reduce average turnaround time.
- It is hard to know the length of the upcoming CPU request.
- Elapsed time should be recorded, that results in more overhead on the processor.

## Summary

- SJF is an algorithm in which the process having the smallest execution time is chosen for the next execution.
- SJF Scheduling is associated with each job as a unit of time to complete.
- This algorithm method is helpful for batch-type processing, where waiting for jobs to complete is not critical.
- There are basically two types of SJF methods 1) Non-Preemptive SJF and 2) Preemptive SJF.
- In non-preemptive scheduling, once the CPU cycle is allocated to process, the process holds it till it reaches a waiting state or terminated.
- In Preemptive SJF Scheduling, jobs are put into the ready queue as they come.
- Although a process with short burst time begins, the current process is removed or preempted from execution, and the job which is shorter is executed 1st.
- SJF is frequently used for long term scheduling.
- It reduces the average waiting time over FIFO (First in First Out) algorithm.
- In SJF scheduling, Job completion time must be known earlier, but it is hard to predict.
- SJF can’t be implemented for CPU scheduling for the short term. It is because there is no specific method to predict the length of the upcoming CPU burst.

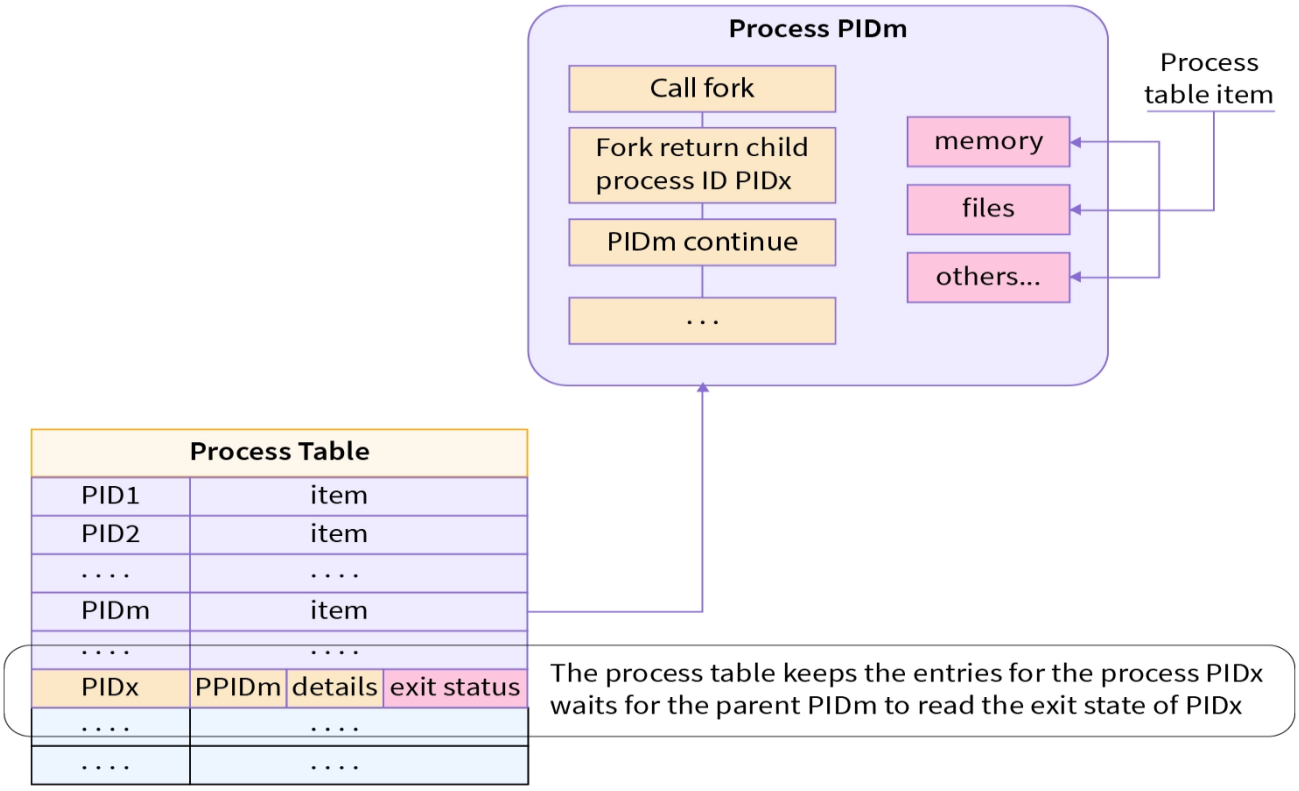
## Zombie and Orphan Process in OS

Processes in OS bear parent child relationship and they have their entry in the system. An orphan process is formed when it's parent dies while the process continues to execute, while zombie process is a process which has terminated but it's entry is there in the system.

### What is the Zombie process?

**Zombie process** is also known as ***"dead"*** process. Ideally when a process completes its execution, it's entry from the process table should be removed but this does not happen in case of zombie a process.

**Analogy:** Zombie, mythological, is a dead person revived physically. Similarly, a zombie process in os is a dead process (completed its execution) but is still revived (it's entry is present in memory).

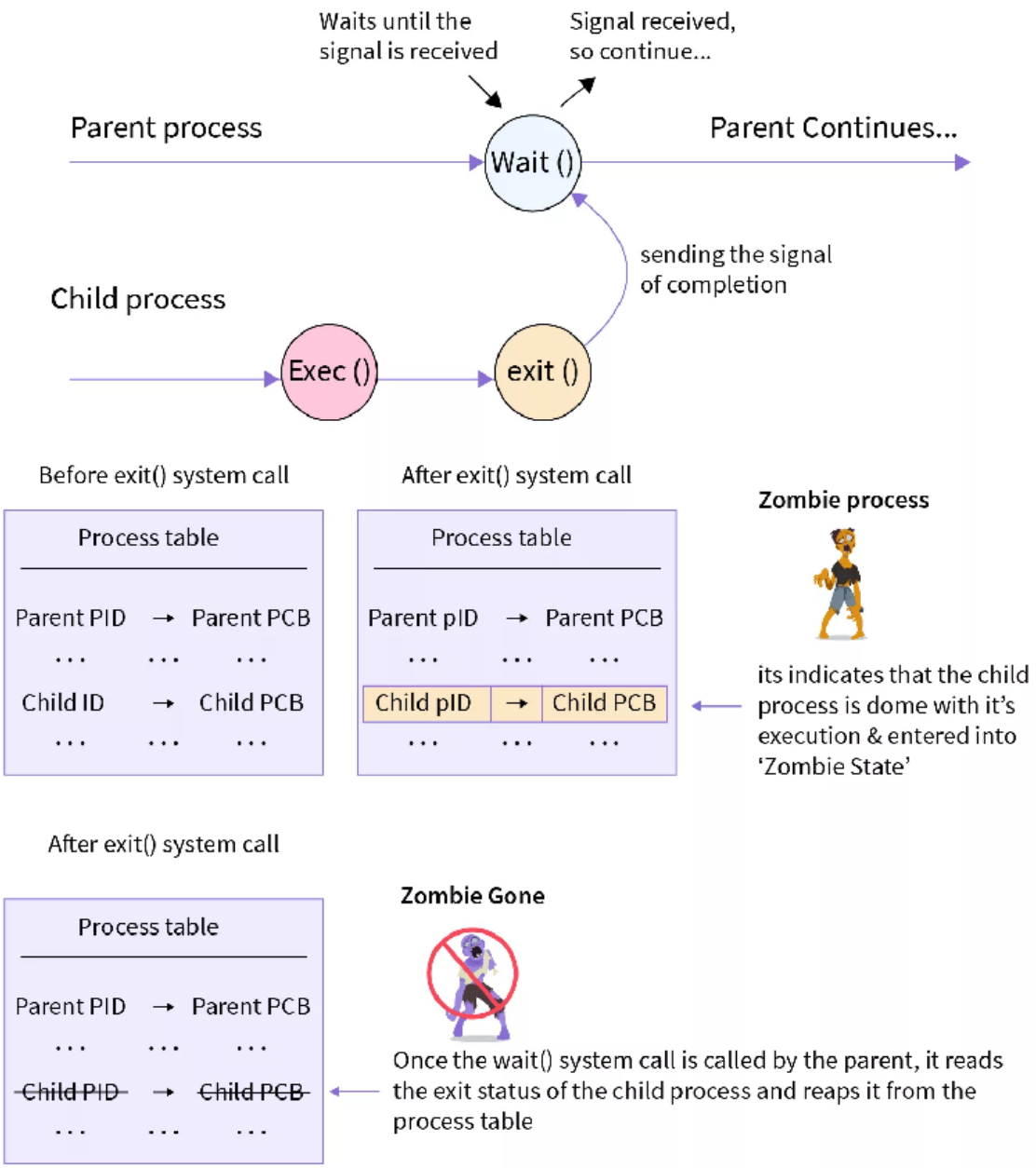



**Note:** Process table is a data structure in RAM to store information about a process.

### What happens with the zombie processes?

- wait() system call is used for removal of zombie processes.
- wait() call ensures that the parent doesn't execute or sits idle till the child process is completed.

- When the child process completes executing ,the parent process removes entries of the child process from the process table. This is called "reaping of child".




## Why are lots of zombie processes harmful to the system ?

A lot of zombie processes in os are harmful as

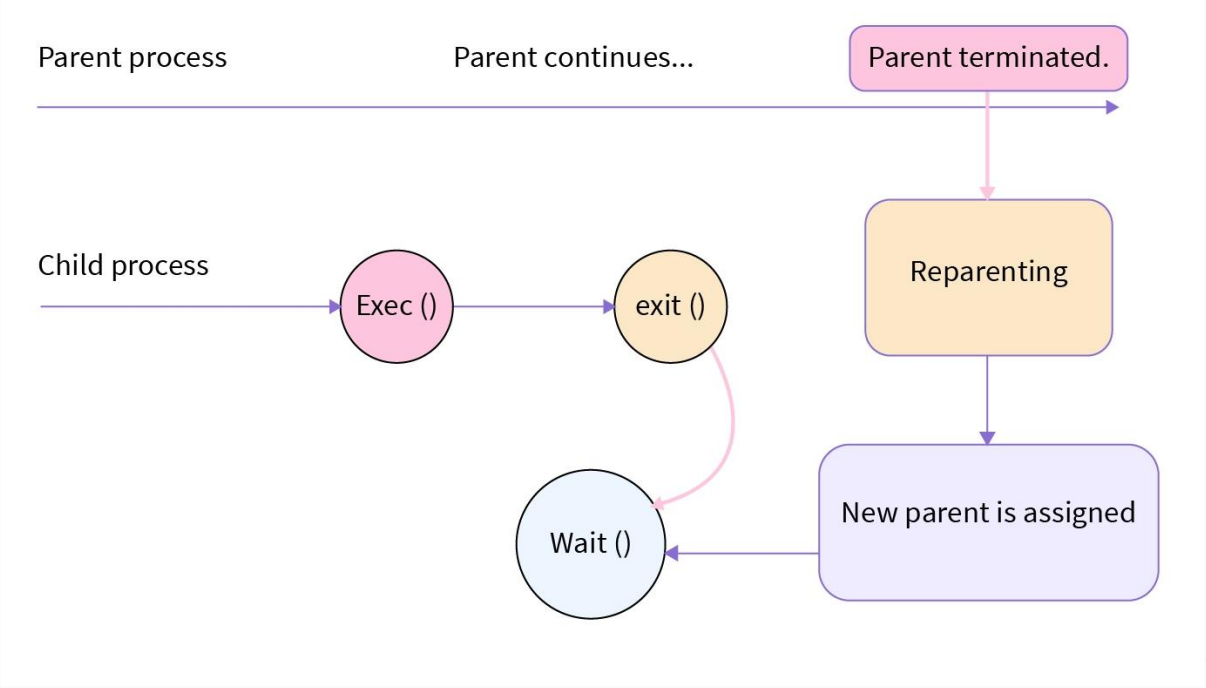
- The OS has one process table of finite size , so lots of zombie processes will results in a full process table.
- A full process table means that OS cannot create a new process when required and Zombie processes in os are of no use as the process has died but it's entry is occupying the space in memory

## What is the Orphan Process?

- In the real world orphans are those children whose parents are dead.
- Similarly, a process which is executing (is alive) but it's parent process has terminated (dead) is called an orphan process.

## What will happen with the orphan processes?

- In the real world orphans are adopted by guardians who look after them.
- Similarly,the orphan process in linux is adopted by a new process , which is mostly init process (pid=1) . This is called ***re-parenting.***
- Reparenting is done by the kernel,when the kernel detects an orphan process in os, and assigns a new parent process.
- New parent process asks the kernel for cleaning of the PCB of the orphan process and the new parent waits till the child completes its execution.




## Why are too many Orphan processes harmful to the system?

- Orphan processes in OS hold resources when present the system.
- Orphan processes in a large number can overload the init process and hang-up a system.

## Zombie Process Code Example

In this code given below, we'll see how a zombie process is created.

- The child process completes its execution by using exit() system call.
- So when the child finishes it's execution **‘SIGCHLD’** signal is delivered to the parent process by the kernel. Parents should,ideally, read the child's status from the process table and then delete the child's entry.
- But here the parent does not wait for the child to terminate , rather it does it's own subsequent job, i.e. here sleeping for 60 seconds.
- So the child's exit status is never read by the parent and the child's entry continues to remain there in the process table even when the child has died.

**Note:** Kernel sends a SIGCHLD signal to the parent process to indicate that the child process has ended ·

```
// A C program to demonstrate Zombie Process.

#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
        // Fork returns process id in parent process

    pid_t child_pid = fork();

        // Parent process
        if (child_pid > 0)
                sleep(60);

        // Child process
        else
                exit(0);

        return 0;
}
```

**Orphan Process Code Example**

In the example,the parent process sleeps for 20 seconds while the child process sleeps for 30 seconds.

1. So after sleeping for 20 seconds the parent completes its execution while the child process is still there at least till 30 seconds.
2. When the child process becomes an orphan process , the kernel reassigns the parent process to the child process.
3. As a result the parent process id of the child process before and after sleep() will be different.

**Note:** Kernel is central component of an operating system that manages operations of computer and hardware.

```
// C program to demonstrate Orphan process
#include<stdio.h>
#include <sys/types.h>
#include <unistd.h>

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

	if(pid == 0)
	{
		printf("I am the child, my process ID is %d\n",getpid());
		printf("My parent's process ID is %d\n",getppid());
		sleep(30);
		printf("\nAfter sleep\nI am the child, my process ID is
%d\n",getpid());
		printf("My parent's process ID is %d\n",getppid());
		exit(0);
	}
	else
	{
		sleep(20);
		printf("I am the parent, my process ID is %d\n",getpid());
		printf("The parent's parent, process ID is %d\n",getppid());
		printf("Parent terminates\n");
	}
   return 0;
}
```

# Session 7:

# Signals and Signal Handling

Signals are software interrupts which sent to a program to indicate that an important event has occurred. This event can vary from user requests to illegal memory access errors. Some signals, such as the interrupt signal, indicate that a user has asked the program to do something that is not in the usual flow of control. Depending on the source of and the reason for the event being signalled, A Signal can be either synchronously or asynchronously. All signals, whether synchronous or asynchronous, follow the same pattern −

- The occurrence of a particular event generates a signal.
- Next, the signal is delivered to a process.
- The signal must be handled, once delivered.

Illegal memory access and division by 0 are included in Synchronous signal include. If either of these actions are performed by a running program, a signal is generated. This signals are delivered to the same process that performed the operation that caused the signal. This is the reason they are considered as synchronous.

When an event generates a signal external to a running process, that process receives the signal asynchronously. E.g. Such signals include terminating a process with specific keystrokes (such as <control><C>) and having a timer expire. Generally, an asynchronous signal is sent to another process. A Signal may be handled by following one of two possible handlers −

- A default signal handler
- A user-defined signal handler

A default signal handler is associated with every signal that the kernel runs when handling that signal. User-defined signal handler can override this default action 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.

There are fix set of signals that can be sent to a process. signal are identified by integers. Signal number have symbolic names. For example **SIGCHLD** is number of the signal sent to the parent process when child terminates.

**Examples:**

```
#define SIGHUP  1   /* Hangup the process */
#define SIGINT  2   /* Interrupt the process */
#define SIGQUIT 3   /* Quit the process */
#define SIGILL  4   /* Illegal instruction. */
#define SIGTRAP 5   /* Trace trap. */
#define SIGABRT 6   /* Abort. */
```

**OS Structures for Signals**

- For each process, the operating system maintains 2 integers with the bits corresponding to a signal numbers.

- The two integers keep track of: **pending signals and blocked signals**
- With 32 bit integers, up to 32 different signals can be represented.

**Example :**
In the example below, the SIGINT ( = 2) signal is blocked and no signals are pending.

A signal is sent to a process setting the corresponding bit in the pending signals integer for the process. Each time the OS selects a process to be run on a processor, the pending and blocked integers are checked. If no signals are pending, the process is restarted normally and continues executing at its next instruction.

If 1 or more signals are pending, but each one is blocked, the process is also restarted normally but with the signals still marked as pending. If 1 or more signals are pending and NOT blocked, the OS executes the routines in the process's code to handle the signals.

When an event generates a signal external to a running process, that process receives the signal asynchronously. E.g. Such signals include terminating a process with specific keystrokes (such as <control><C>) and having a timer expire. Generally, an asynchronous signal is sent to another process. A Signal may be handled by following one of two possible handlers −

- A default signal handler
- A user-defined signal handler

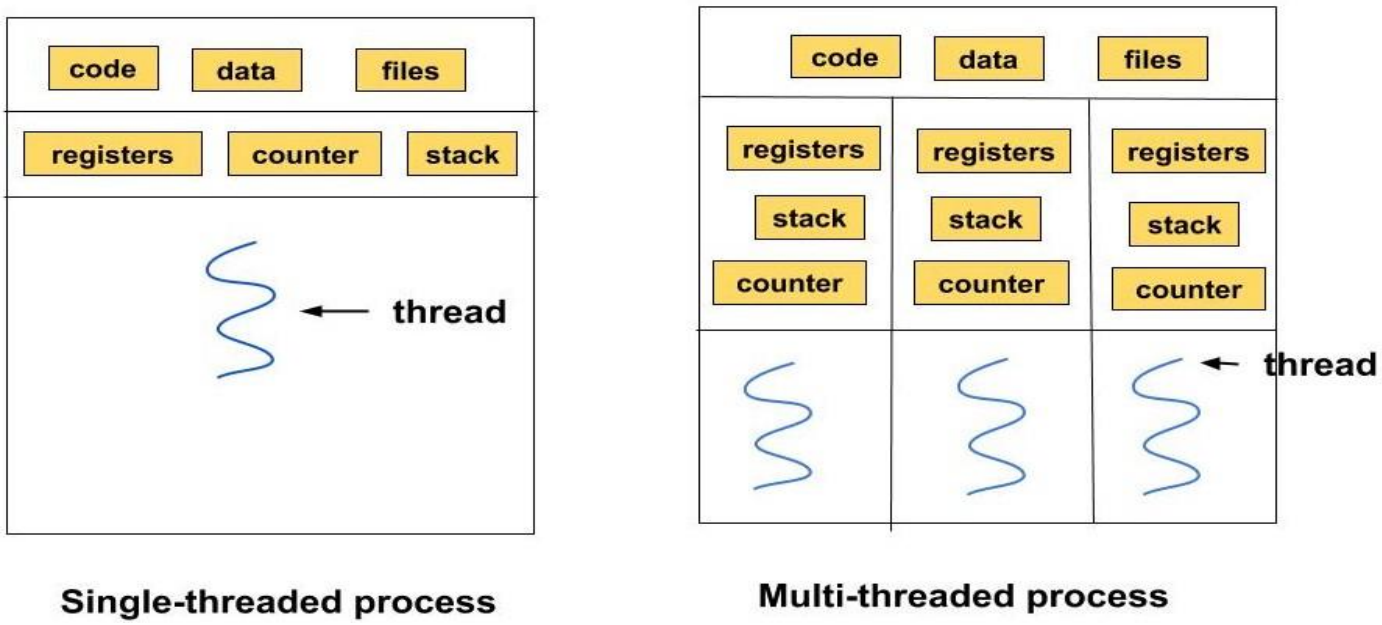
A default signal handler is associated with every signal that the kernel runs when handling that signal. User-defined signal handler can override this default action that is called to handle the signal. Signals are handled in different ways. Some signals (such as changing the size of a window) are simply ignored; others (such as an illegal memory access) are handled by terminating the program.

# Thread

A thread is an execution unit that has its own program counter, a stack and a set of registers that reside in a process. Threads can't exist outside any process. Also, each thread belongs to exactly one process. The information like code segment, files, and data segment can be shared by the different threads.

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.

**Threads are also known as light-weight processes.**

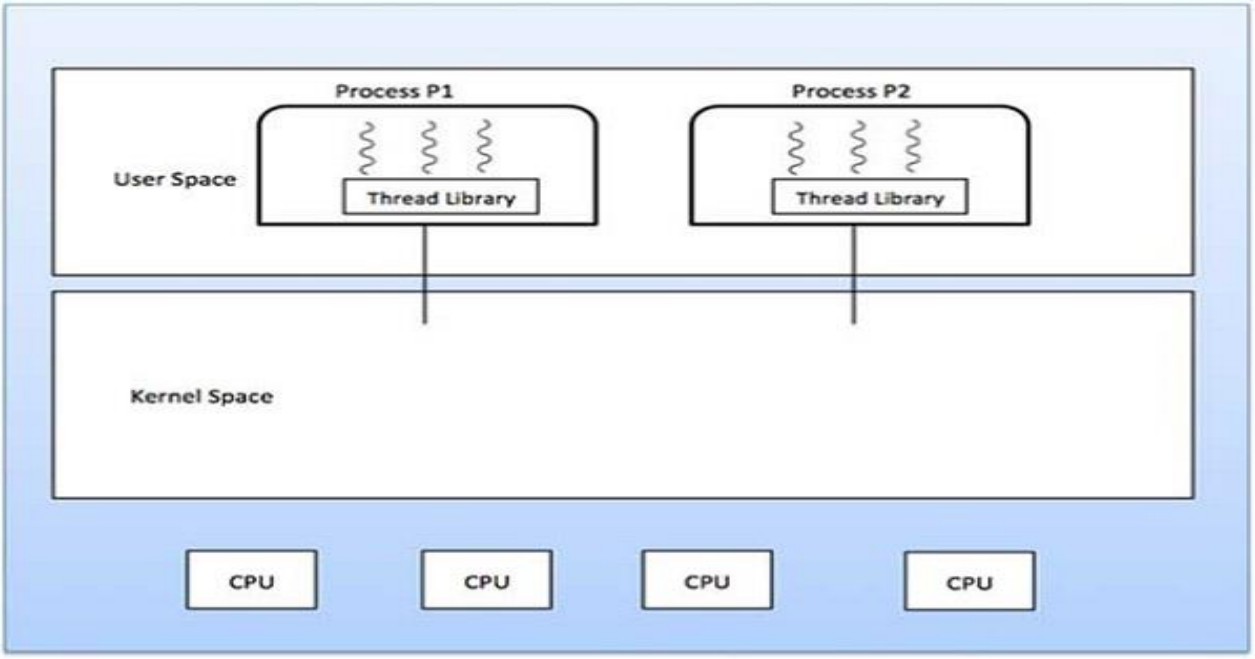



The diagram above shows the single-threaded process and the multi-threaded process. A **single-threaded process** is a process with a single thread. A **multi-threaded process** is a process with multiple threads. As the diagram clearly shows that the multiple threads in it have its own registers, stack,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

# Types of Thread

## User Level Threads

In this case, the thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application starts with a single thread.




## Advantages

- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.
- Scheduling can be application specific in the user level thread.
- User level threads are fast to create and manage.

## Disadvantages

- In a typical operating system, most system calls are blocking.
- Multithreaded application cannot take advantage of multiprocessing.

# Kernel Level Threads

In this case, thread management is done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.

The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

## Advantages

- Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
- If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
- Kernel routines themselves can be multithreaded.

## Disadvantages

- Kernel threads are generally slower to create and manage than the user threads.
- Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

## Advantages of threads

1. **Performance:** Threads improve the overall performance(throughput, computational speed, responsiveness) of a program.
2. **Resource sharing:** As the threads can share the memory and resources of any process it allows any application to perform multiple activities inside the same address space.
3. **Utilization of Multiple Processor Architecture:** The different threads can run parallel on the multiple processors hence, this enables the utilization of the processor to a large extent and efficiency.
4. **Reduced Context Switching Time:** The threads minimize the context switching time as in Thread Switching, the virtual memory space remains the same.
5. **Concurrency:** Thread provides concurrency within a process.
6. **Parallelism:** Parallel programming techniques are easier to implement.

## Difference between process and thread

1. **Definition:** Process means a program that is currently under execution, whereas thread is an entity that resides within a process that can be scheduled for execution.
2. **Termination Time:** The processes take more time to terminate, whereas threads take less time to terminate.
3. **Creation Time:** The process creation time takes more time as compared to thread creation time.
4. **Context Switching Time:** Process context switching takes more time as compared to the thread context switching.
5. **Communication:** The communication between threads requires less time as compared to the communication between processes.
6. **Resources:** Processes are also called heavyweight processes as they use more resources. The threads are called light-weight processes as they share resources.
7. **Memory:** A Process is run in separate memory space, whereas threads run in shared memory space.

8. **Sharing Data:** Different processes have different copies of data, files, and codes whereas threads share the same copy of data, file and code segments.
9. **Example:** Opening a new browser (say Chrome, etc) is an example of creating a process. At this point, a new process will start to execute. On the contrary, opening multiple tabs in the browser is an example of creating the thread.

## Multithreading Models

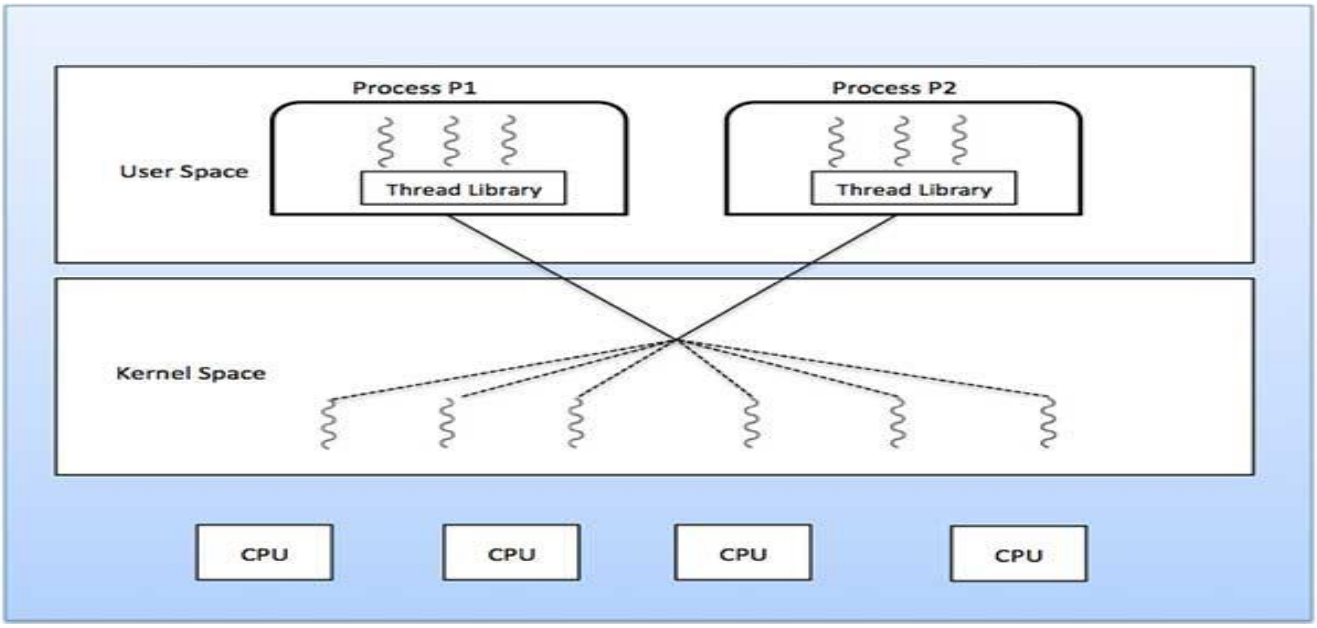
Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process. Multithreading models are three types

- Many to many relationship.
- Many to one relationship.
- One to one relationship.

## Many to Many Model

The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads.

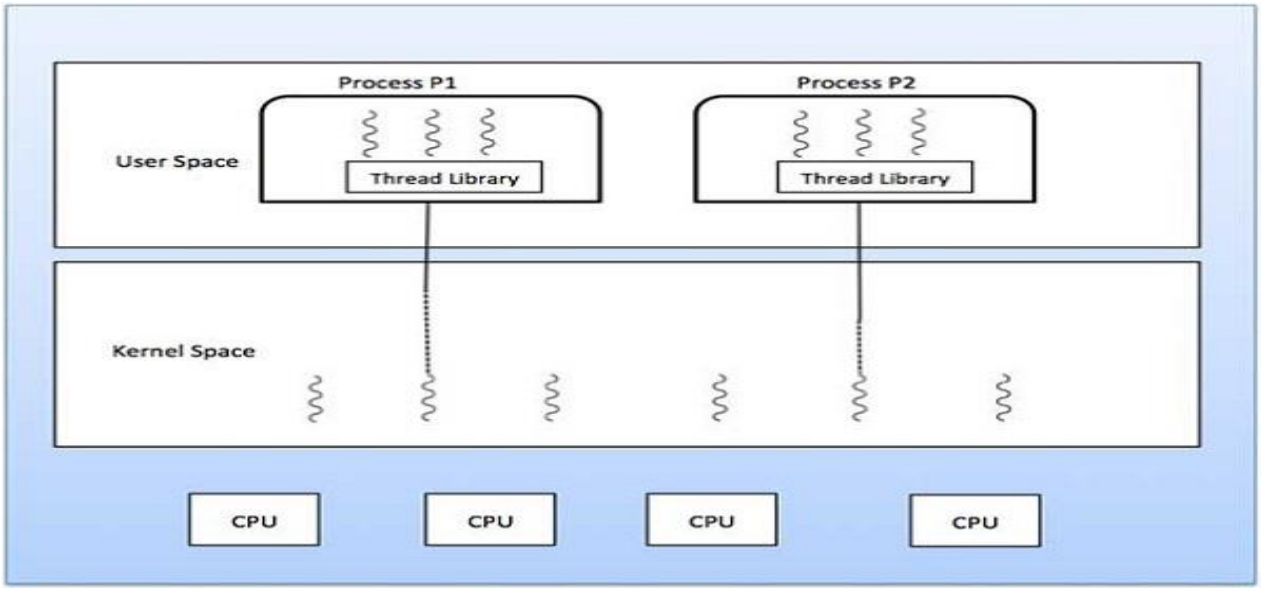
The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.




## Many to One Model

Many-to-one model maps many user level threads to one Kernel-level thread. Thread management is done in user space by the thread library. When thread makes a blocking system call, the entire process will be blocked. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

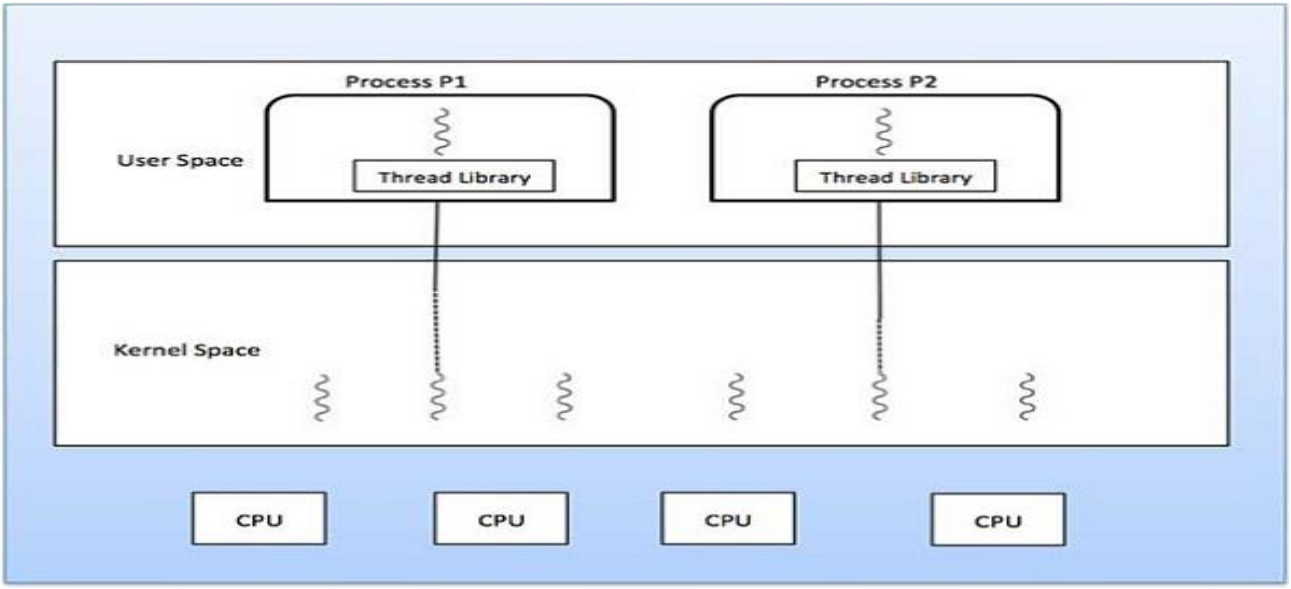
If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.




## One to One Model

There is one-to-one relationship of user-level thread to the kernel-level thread. This model provides more concurrency than the many-to-one model. It also allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.

Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model.



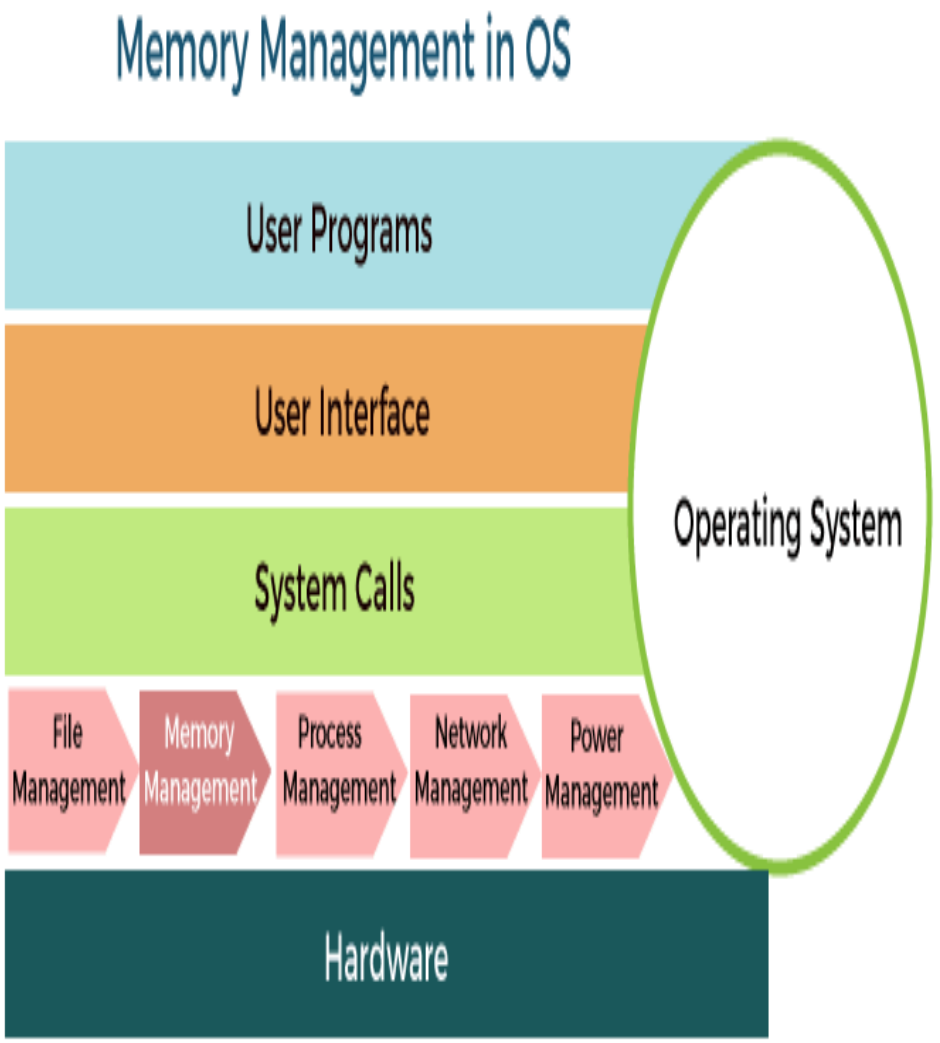
# Session 8 & 9

**What do you mean by memory management?**

Memory is the important part of the computer that is used to store the data. Its management is critical to the computer system because the amount of main memory available in a computer system is very limited. At any time, many processes are competing for it. Moreover, to increase performance, several processes are executed simultaneously. For this, we must keep several processes in the main memory, so it is even more important to manage them effectively.

Memory management plays several roles in a computer system.
Following are the important roles in a computer system:

- Memory manager is used to keep track of the status of memory locations, whether it is free or allocated. It addresses primary memory by providing abstractions so that software perceives a large memory is allocated to it.
- Memory manager permits computers with a small amount of main memory to execute programs larger than the size or amount of available memory. It does this by moving information back and forth between primary memory and secondary memory by using the concept of swapping.
- The memory manager is responsible for protecting the memory allocated to each process from being corrupted by another process. If this is not ensured, then the system may exhibit unpredictable behavior.
- Memory managers should enable sharing of memory space between processes. Thus, two programs can reside at the same memory location although at different times.

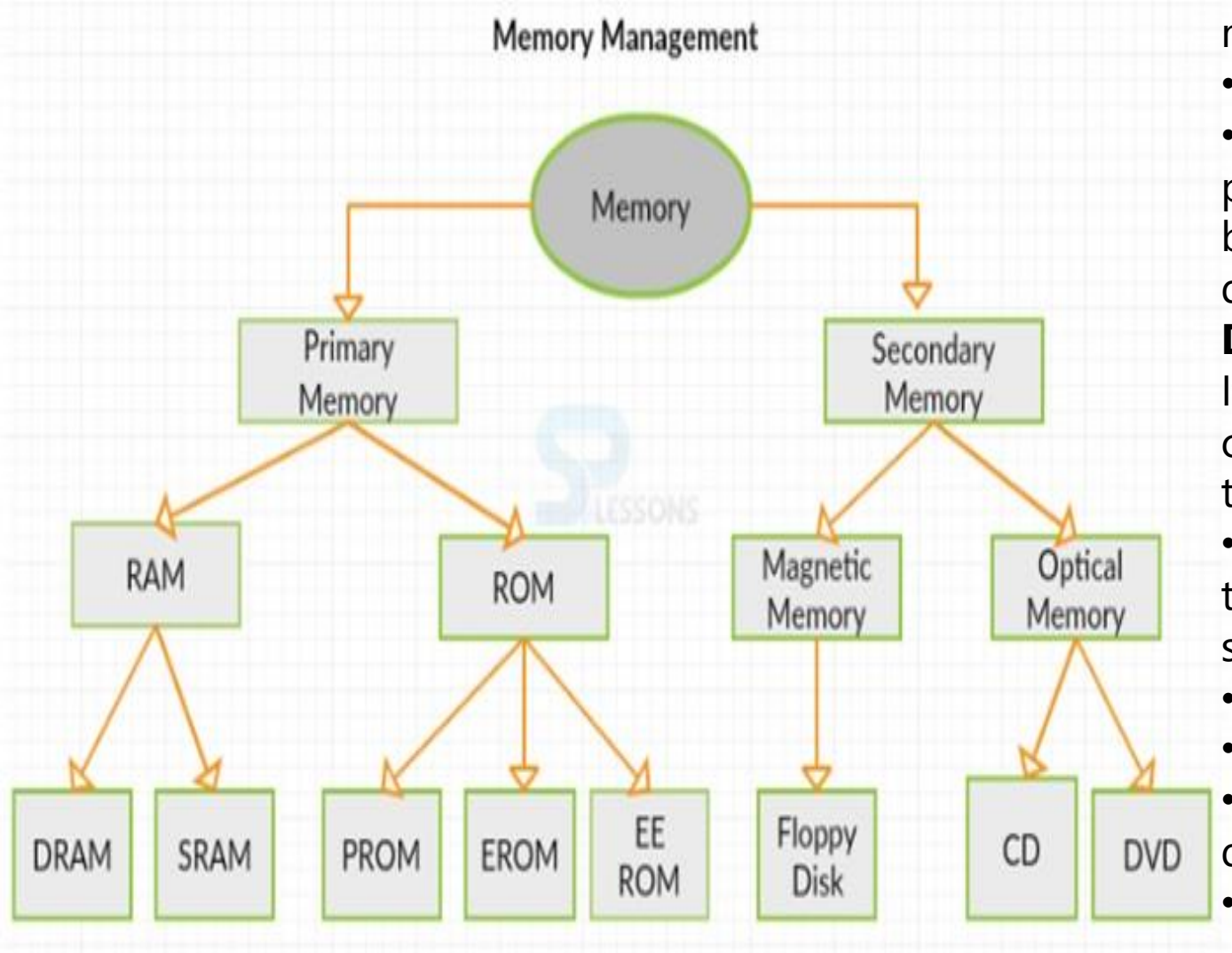


# Types of computer memory

There are two types of memories:

Primary memory

Secondary memory




**SRAM (Static random access memory)**

It holds data in a static form, that is, as long as the memory has the power as the dynamic RAM, it is not needed to refresh it again and again.

•Static RAM provides faster access to data and is more expensive than DRAM as each cell must contain multiple transistors.

•SRAM does not use capacitors.

•SRAM is also highly recommended for use in PCs, peripheral equipment, printers, LCD screens, hard disk buffers, router buffers and buffers in CDROM / CDRW drives.

**Dynamic RAM (Dynamic random access memory)**

It is a type of random-access memory used in computing devices. It is made up of capacitors and transistors.

•This type of memory uses separate capacitors or transistors to stores each bit of data and it has two states of value in one bit called 0 and 1.

•As compared with other RAM's it is less expensive.

•Data were written by DRAM at the byte-level.

•In DRAM, data is written at the byte-level and it reads data at the multiple-byte page level.

•DRAM requires less power than other RAMs.

## Types of Memory Allocation

### 1. Static Memory Allocation

Static memory allocation is performed when the compiler compiles the program and generate object files, linker merges all these object files and creates a single executable file, and loader loads this single executable file in main memory, for execution. In static memory allocation, the size of the data required by the process must be known **before** the execution of the process initiates.

If the data sizes are not known before the execution of the process, then they have to be guessed. If the data size guessed is larger than the required, then it leads to **wastage** of memory. If the guessed size is smaller, then it leads to inappropriate execution of the process.

### 2. Dynamic Memory Allocation

Dynamic memory allocation is performed while the program is in execution. Here, the memory is allocated to the entities of the program when they are to be used for the first time while the program is running.

The actual size, of the data required, is known at the run time so, it allocates the exact memory space to the program thereby, reducing the memory wastage.

Dynamic memory allocation provides flexibility to the execution of the program. As it can decide what amount of memory space will be required by the program. If the program is large enough then a dynamic memory allocation is performed on the different parts of the program, which is to be used currently. This reduces memory wastage and improves the performance of the system.

## Difference Between Contiguous and Non-contiguous Memory Allocation

### Contiguous Memory Allocation

- In this allocation type, the consecutive blocks of memory is allocated to a file/process.
- It executes quickly in comparison to non-contiguous memory.
- It is easy to be controlled by the operating system.
- Minimum overhead since there are not many address translation when a process is being executed.
- There is internal fragmentation in contiguous memory allocation.
- There are different types of partitions: Single partition allocation and multi-partition allocation.
- Memory gets wasted.

### Non-contiguous Memory Allocation

- In this type of allocation, separate blocks of memory are allocated to a file/process.
- It executes slowly in comparison to contiguous memory.
- It is difficult to be controlled by the operating system.
- More overhead since it needs to translate multiple addresses.
- It includes paging.
- It includes segmentation.
- No memory is wasted.
- External fragmentation occurs in this type of allocation.
- The swapped-in processes can be arranged in any location in the memory.

# How do we allocate memory for a Process?

Processes enter the main memory and stay till they get completely executed.
Once a process exits, previously occupied memory is freed creating free spaces in the memory. These spaces of unallocated memory are termed **Holes**.
In a Variable Partition Allocation we use three strategies to allocate memory:

**First fit**
An incoming process gets allocated into a Hole of size greater than or equal to it. Searching for Holes can start from the beginning of the memory or from where the previous first fit search ended.

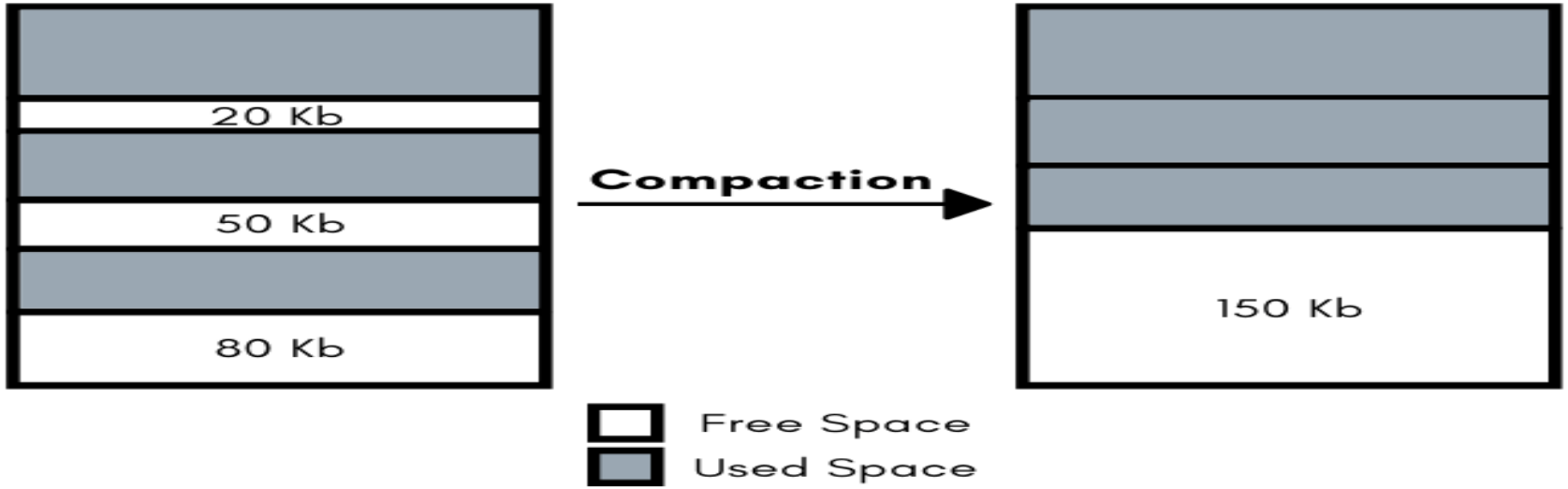
Speed and Simplicity are the perks of this strategy. However, it's not the most optimal in terms of the size of memory allocated.

**Best fit**
We find the least sized Hole capable of holding the incoming process. This generates the smallest left over Holes. This is the most optimal strategy based on Memory Utilization. This strategy is slower than the First fit method.

**Worst Fit**
We find the largest Hole and allocate the incoming process to it.
As the name might suggest, its cons are plenty. First, the time taken to implement Worst fit is higher than that of First fit and Best fit strategies. Second, it's the least efficient strategy based on Memory Utilization.

**What is compaction is OS?**

• Compaction is a process in which the free space is collected in a large memory chunk to make some space available for processes.
•In memory management, swapping creates multiple fragments in the memory because of the processes moving in and out.
•Compaction refers to combining all the empty spaces together and processes.
• Compaction helps to solve the problem of fragmentation, but it requires too much of CPU time.
•It moves all the occupied areas of store to one end and leaves one large free space for incoming jobs, instead of numerous small ones.
• In compaction, the system also maintains relocation information and it must be performed on each new allocation of job to the memory or completion of job from memory.



## Difference between Internal Fragmentation and External Fragmentation

| Sr. No. | Key | Internal Fragmentation | External Fragmentation |
|---|---|---|---|
| 1 | Definition | When there is a difference between required memory space vs allotted memory space, problem is termed as Internal Fragmentation. | When there are small and non-contiguous memory blocks which cannot be assigned to any process, the problem is termed as External Fragmentation. |
| 2 | Memory Block Size | Internal Fragmentation occurs when allotted memory blocks are of fixed size. | External Fragmentation occurs when allotted memory blocks are of varying size. |
| 3 | Occurrence | Internal Fragmentation occurs when a process needs more space than the size of allotted memory block or use less space. | External Fragmentation occurs when a process is removed from the main memory. |
| 4 | Solution | Best Fit Block Search is the solution for internal fragmentation. | Compaction is the solution for external fragmentation. |
| 5 | Process | Internal Fragmentation occurs when Paging is employed. | External Fragmentation occurs when Segmentation is employed. |

# Segmentation

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

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

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

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

Given below are the types of Segmentation:

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

## Characteristics of Segmentation

Some characteristics of the segmentation technique are as follows:

- The Segmentation partitioning scheme is **variable-size.**
- Partitions of the secondary memory are commonly known as **segments**.
- Partition size mainly depends upon the length of modules.
- Thus with the help of this technique, secondary memory and main memory are divided into unequal-sized partitions.

## Need of Segmentation

One of the important drawbacks of memory management in the Operating system is the separation of the user's view of memory and the actual physical memory. and Paging is a technique that provides the separation of these two memories.

User'view is basically mapped onto physical storage. And this mapping allows differentiation between Physical and logical memory. It is possible that the operating system divides the same function into different pages and those pages may or may not be loaded into the memory at the same time also Operating system does not care about the User's view of the process. Due to this technique system's efficiency decreases.

Segmentation is a better technique because it divides the process into segments.

# Segmentation Architecture

## Segment Table

A Table that is used to store the information of all segments of the process is commonly known as Segment Table. Generally, there is no simple relationship between logical addresses and physical addresses in this scheme.

•The mapping of a two-dimensional Logical address into a one-dimensional Physical address is done using the segment table.

•This table is mainly stored as a separate segment in the main memory.

•The table that stores the base address of the segment table is commonly known as the Segment table base register (STBR)

In the segment table each entry has :

**1.Segment Base/base address:** The segment base mainly contains the starting physical address where the segments reside in the memory.

**2.Segment Limit:** The segment limit is mainly used to specify the length of the segment.

**Segment Table Base Register(STBR)** The STBR register is used to point the segment table's location in the memory.

**Segment Table Length Register(STLR)** This register indicates the number of segments used by a program.

| | Limit | Base |
|---|---|---|
| Segment 0 → | 1400 | 1400 |
| Segment 1 → | 400 | 6200 |
| Segment 2 → | 1100 | 4400 |
| Segment 3 → | 1300 | 4800 |

Segment Table

# Segmentation Architecture Continue….

## Segmentation Hardware

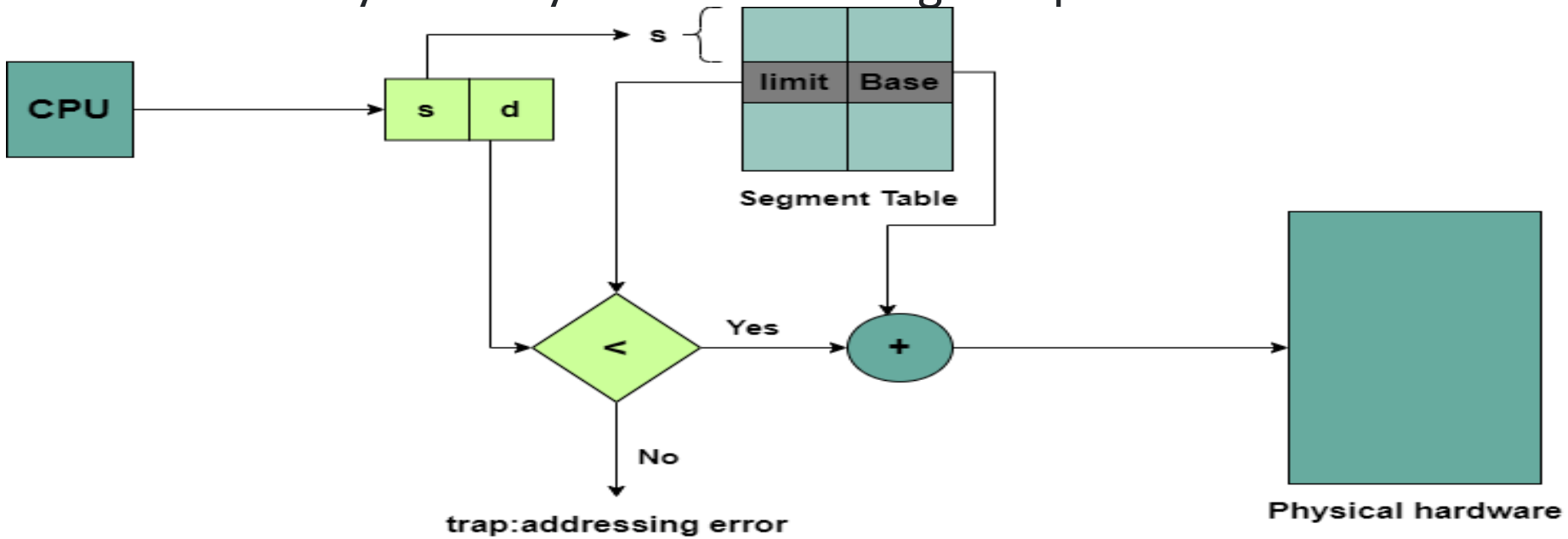
Given below figure shows the segmentation hardware :

The logical address generated by CPU consist of two parts:

- Segment Number(s): It is used as an index into the segment table.
- Offset(d): It must lie in between '0' and 'segment limit'.In this case, if the Offset exceeds the segment limit then the trap is generated.

Thus; **correct offset+segment base= address in Physical memory**

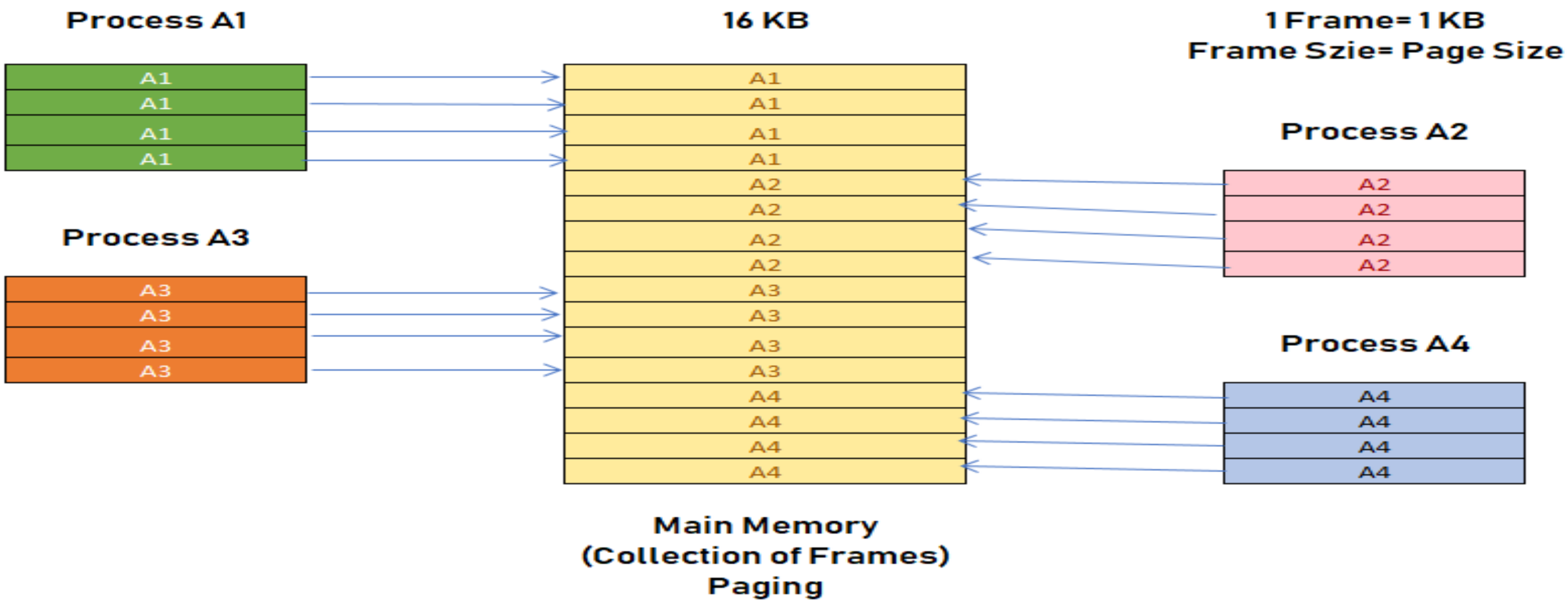
and segment table is basically an array of base-limit register pair.



# Paging

**Paging** is a storage mechanism that allows OS to retrieve processes from the secondary storage into the main memory in the form of pages. In the Paging method, the main memory is divided into small fixed-size blocks of physical memory, which is called frames. The size of a frame should be kept the same as that of a page to have maximum utilization of the main memory and to avoid external fragmentation. Paging is used for faster access to data, and it is a logical concept.

For example, if the main memory size is 16 KB and Frame size is 1 KB. Here, the main memory will be divided into the collection of 16 frames of 1 KB each.
There are 4 separate processes in the system that is A1, A2, A3, and A4 of 4 KB each. Here, all the processes are divided into pages of 1 KB each so that operating system can store one page in one frame.
At the beginning of the process, all the frames remain empty so that all the pages of the processes will get stored in a contiguous way.


Process A1
A1
A1
A1
A1
Process A3
A3
A3
A3
A3
16 KB
A1
A1
A1
A1
A2
A2
A2
A2
A3
A3
A3
A3
A4
A4
A4
A4
1 Frame= 1 KB
Frame Szie= Page Size
Process A2
A2
A2
A2
A2
Process A4
A4
A4
A4
A4
Main Memory
(Collection of Frames)
Paging

## How paging works

Paging works by writing data to, and reading it from, secondary storage for use in primary storage. Paging is a basic function in memory management for a computer's operating system (OS) as well -- this includes Windows, Unix, Linux and macOSs.

In a memory management system that takes advantage of paging, the OS reads data from secondary storage in blocks called pages, all of which have identical size. The physical region of memory containing a single page is called a frame. When paging is used, a frame does not have to comprise a single physically contiguous region in secondary storage. This approach offers an advantage over earlier memory management methods, because it facilitates more efficient and faster use of storage.

## Advantages of Paging

Here, are advantages of using Paging method:

- Easy to use memory management algorithm
- No need for external Fragmentation
- Swapping is easy between equal-sized pages and page frames.

## Disadvantages of Paging

Here, are drawback/ cons of Paging:

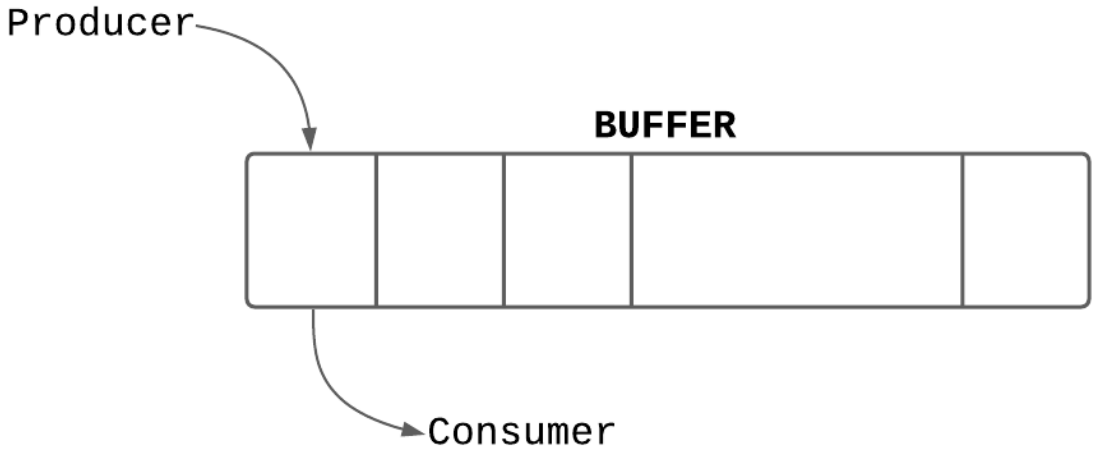
- May cause Internal fragmentation
- Page tables consume additonal memory.
- Multi-level paging may lead to memory reference overhead.

## Producer-Consumer Problem using Semaphores

One of the classic synchronization problems is the bounded buffer problem, commonly known as the **producer-consumer problem**.

In the **producer-consumer problem**, there is one Producer who produces things, and there is one Consumer who consumes the products which are produced by the producer. The producers and consumers share the same fixed-size memory buffer.
The Producer's role is to produce data, store it in the buffer, and then generate data again. The Consumer's task is to consume the data from the buffer.
The Producer-Consumer problem is a classic multi-process synchronization problem, which implies we're aiming to synchronize many processes.
When the consumer is consuming an item from the buffer, the producer should not add items into the buffer, and vice versa. As a result, only one producer or consumer should access the buffer at a time.

**What's the problem here?**
Let's consider that there are n slots in the buffer, and each slot may store one unit of data. On the buffer, there are two processes running: producer and consumer.




A producer tries to fill an empty space in the buffer with data. A consumer attempts to retrieve data from that buffer slot. If those two processes run concurrently, as you may have anticipated, they will not give the desired outcome.
There must be a method to make both the producer and the customer operate independently.

The following are some of the issues that might arise in the Producer-Consumer:

- The producer should generate data only if the buffer is not full. When the buffer is filled, the producer should not be able to add any more data to it.
- When the buffer is not empty, the consumer can consume the data. The consumer should not be able to take any data from the buffer if it is empty.
- The buffer should not be used by both the producer and the consumer at the same time.

**What's the solution then?**

Semaphores can be used to solve the three difficulties listed above. A semaphore S is an integer variable with just two standard operations: wait() and signal().

The signal() method increases the value of semaphore by one, whereas the wait() operation decreases it by one.

```
wait(S){
while(S<=0);   // busy waiting in the while loop
S--;     // decreasing S by one
}

signal(S){
S++;     // increasing S by one
}
```

There are two types of semaphores:

**Binary Semaphore -** This is related to, but not the same as, a mutex lock. It can only have two possible values: 0 and 1. Its value is set to 1 at the start. It is used to implement a solution to a critical section problem including multiple processes.

**Counting Semaphore -** Its value can traverse an unbounded domain. It's used to limit who has access to a resource with numerous instances.

To solve this problem, We employ three semaphore variables:-

- **mutex** - The lock is acquired and released using a mutex, a binary semaphore.
- **empty** - empty is a counting semaphore that is initialized on the basis of the number of slots present in the buffer, at first all the slots are empty.
- **full** - a counting semaphore with a value of zero as its starting value.

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.

**The Producer Solution**

The pseudocode for the producer function will look like this:

```
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.

This is how we address the producer section of the producer-consumer problem.

**The Producer Solution**

The pseudocode for the producer function will look like this:

```
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.

This is how we address the producer section of the producer-consumer problem.

**The Consumer Solution**

The pseudocode for the producer function will look like this:

```
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.

This is how we overcome the producer-consumer problem.

# Session 9 and 10

# Translation Lookaside Buffer (TLB) in Paging

In Operating System (Memory Management Technique : Paging), for each process page table will be created, which will contain Page Table Entry (PTE). This PTE will contain information like frame number (The address of main memory where we want to refer), and some other useful bits (e.g., valid/invalid bit, dirty bit, protection bit etc). This page table entry (PTE) 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 address generated by CPU (i.e logical/virtual address). Initially, some people thought of using registers to store page table, as they are high-speed memory so access time will be less.

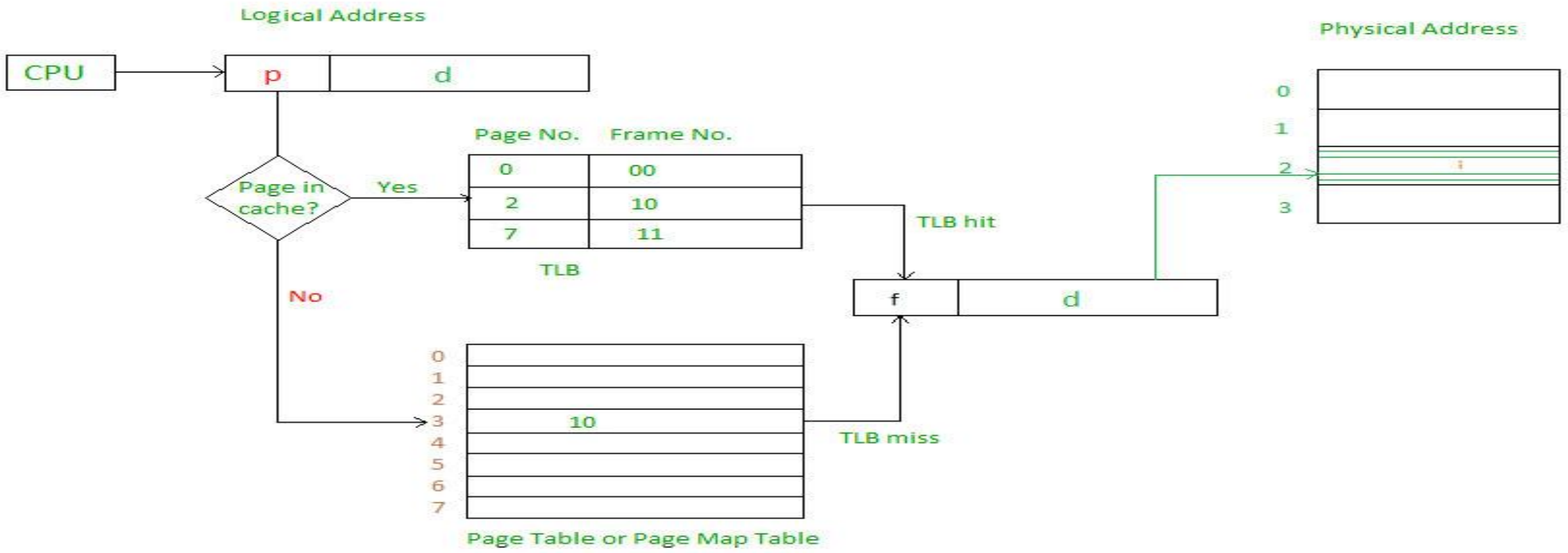
The idea used here is, place the page table entries in registers, for each request generated from CPU (virtual address), it will be matched to the appropriate page number of the page table, which will now tell where in the main memory that corresponding page resides. Everything seems right here, but the problem is register size is small (in practical, it can accommodate maximum of 0.5k to 1k page table entries) and process size may be big hence the required page table will also be big (lets say this page table contains 1M entries), so registers may not hold all the PTE's of Page table. So this is not a practical approach.

To overcome this size issue, the entire page table was kept in main memory. but the problem here is two main memory references are required:

- To find the frame number
- To go to the address specified by frame number

## Translation Lookaside Buffer (TLB) in Paging Continue…….

To overcome this problem a high-speed cache is set up for page table entries called a Translation Lookaside Buffer (TLB). Translation Lookaside Buffer (TLB) is nothing but a special cache used to keep track of recently used transactions. TLB contains page table entries that have been most recently used. Given a virtual address, the processor examines the TLB if a page table entry is present (TLB hit), the frame number is retrieved and the real address is formed. If a page table entry is not found in the TLB (TLB miss), the page number is used as index while processing page table. TLB first checks if the page is already in main memory, if not in main memory a page fault is issued then the TLB is updated to include the new page entry.



# Translation Lookaside Buffer (TLB) in Paging

**Steps in TLB hit:**

1. CPU generates virtual (logical) address.
2. It is checked in TLB (present).
3. Corresponding frame number is retrieved, which now tells where in the main memory page lies.

**Steps in TLB miss:**

1.CPU generates virtual (logical) address.

2.It is checked in TLB (not present).

3.Now the page number is matched to page table residing in main memory (assuming page table contains all PTE).

4.Corresponding frame number is retrieved, which now tells where in the main memory page lies.

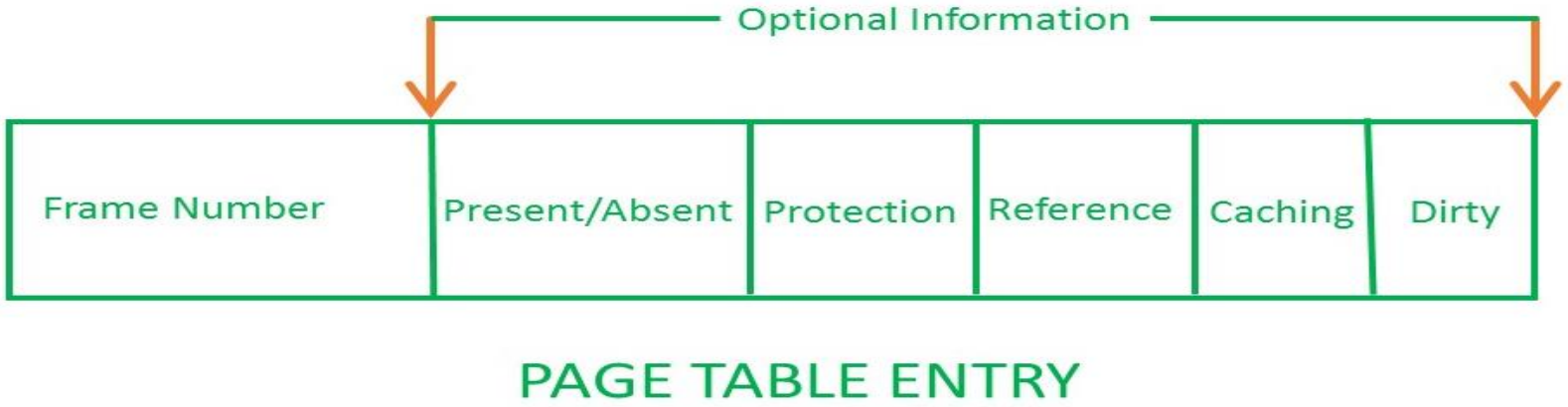
5.The TLB is updated with new PTE (if space is not there, one of the replacement technique comes into picture i.e either FIFO, LRU or MFU etc).

# What is Dirty Bit?

In order to reduce the page fault service time, a special bit called the dirty bit can be associated with each page.

The dirty bit is set to “1” by the hardware whenever the page is modified. (written into). When we select a victim by using a page replacement algorithm, we examine its dirty bit. If it is set, that means the page has been modified since it was swapped in. In this case we have to write that page into the backing store.

However if the dirty bit is reset, that means the page has not been modified since it was swapped in, so we don’t have to write it into the backing store. The copy in the backing store is valid.



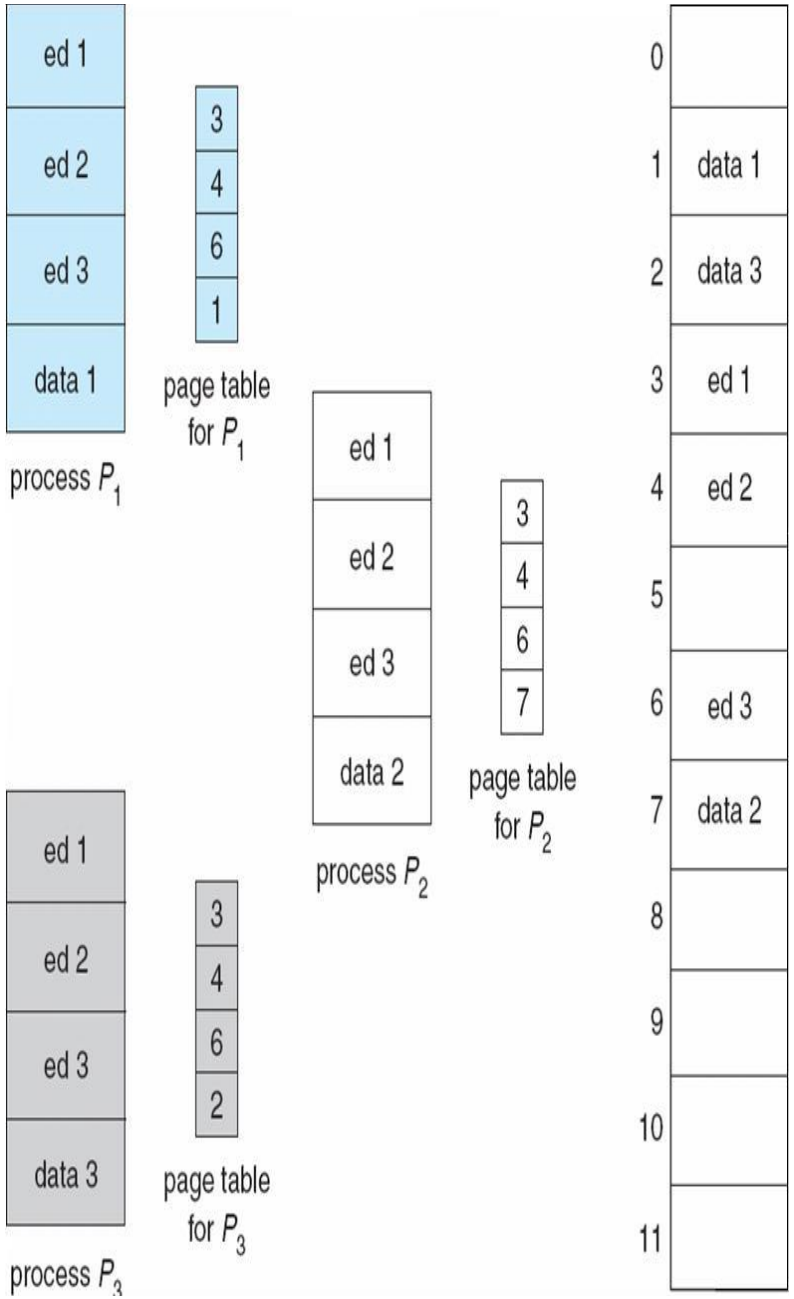
# Shared Pages

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,

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. Reentrant 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, 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 2,150 KB instead of 8,000 KB—a significant savings.



# WHAT IS THROTTLING?

**Basically, CPU throttling is the means through which a CPU controls its clock speed and voltage either automatically depending on the workload or due to the heat generated. It can also be affected by user assigned limits on current, voltage or power.**

**Automatic CPU throttling is also know as dynamic voltage and frequency scaling (DVFS) and is found on almost all modern CPUs. This technique of controlling clock speed is particularly useful among laptops and mobile devices where controlling heat and conserving power is essential.**
**Hence CPU throttling is a bigger concern on laptops as supposed to on desktops.**

**During low-memory conditions, a system will slow down the processing of I/O memory requests, typically processing one sequence at a time in the order the request was received. I/O throttling slows down a system but typically will prevent the system from crashing.**

**What is Virtual Memory?**

**Virtual Memory** is a storage mechanism which offers user an illusion of having a very big main memory. It is done by treating a part of secondary memory as the main memory. In Virtual memory, the user can store processes with a bigger size than the available main memory.

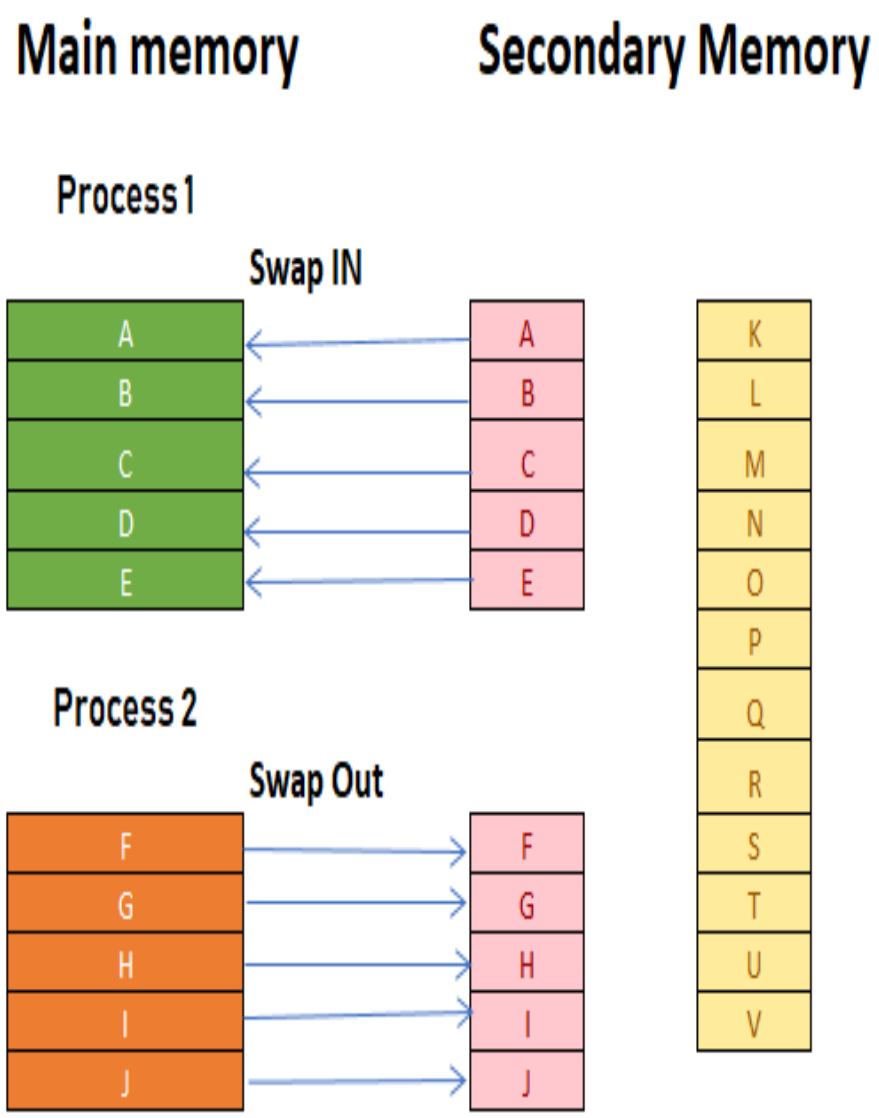
Therefore, instead of loading one long process in the main memory, the OS loads the various parts of more than one process in the main memory. Virtual memory is mostly implemented with demand paging and demand segmentation.

**What is Demand Paging?**

A demand paging mechanism is very much similar to a paging system with swapping where processes stored in the secondary memory and pages are loaded only on demand, not in advance.

So, when a context switch occurs, the OS never copy any of the old program's pages from the disk or any of the new program's pages into the main memory. Instead, it will start executing the new program after loading the first page and fetches the program's pages, which are referenced.

During the program execution, if the program references a page that may not be available in the main memory because it was swapped, then the processor considers it as an invalid memory reference. That's because the page fault and transfers send control back from the program to the OS, which demands to store page back into the memory.



# Revision

## Session 10 Continue….

## Session 11:- Deadlock_Samaphores_Mutex

**Page Fault –** A page fault happens when a running program accesses a memory page that is mapped into the virtual address space but not loaded in physical memory.

Since actual physical memory is much smaller than virtual memory, page faults happen. In case of a page fault, Operating System might have to replace one of the existing pages with the newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce the number of page faults.

**Page Replacement Algorithms :**

**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-1Consider page reference string 1, 3, 0, 3, 5, 6, 3 with 3 page frames. Find the number of page faults.**

Page reference 1, 3, 0, 3, 5, 6, 3

| 1 | 3 | 0 | 3 | 5 | 6 | 3 |
|---|---|---|---|---|---|---|
| | | 0 | 0 | 0 | 0 | 3 |
| | 3 | 3 | 3 | 3 | 6 | 6 |
| 1 | 1 | 1 | 1 | 5 | 5 | 5 |
| Miss | Miss | Miss | Hit | Miss | Miss | Miss |

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**

## 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, 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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| | | 1 | 1 | 1 | 1 | 1 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 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

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. Least Recently Used –**

In this algorithm, page will be replaced which is least recently used.

**Example-3**Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| | | 1 | 1 | 1 | 1 | 1 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 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 their 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 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.

# Deadlock in Operating System

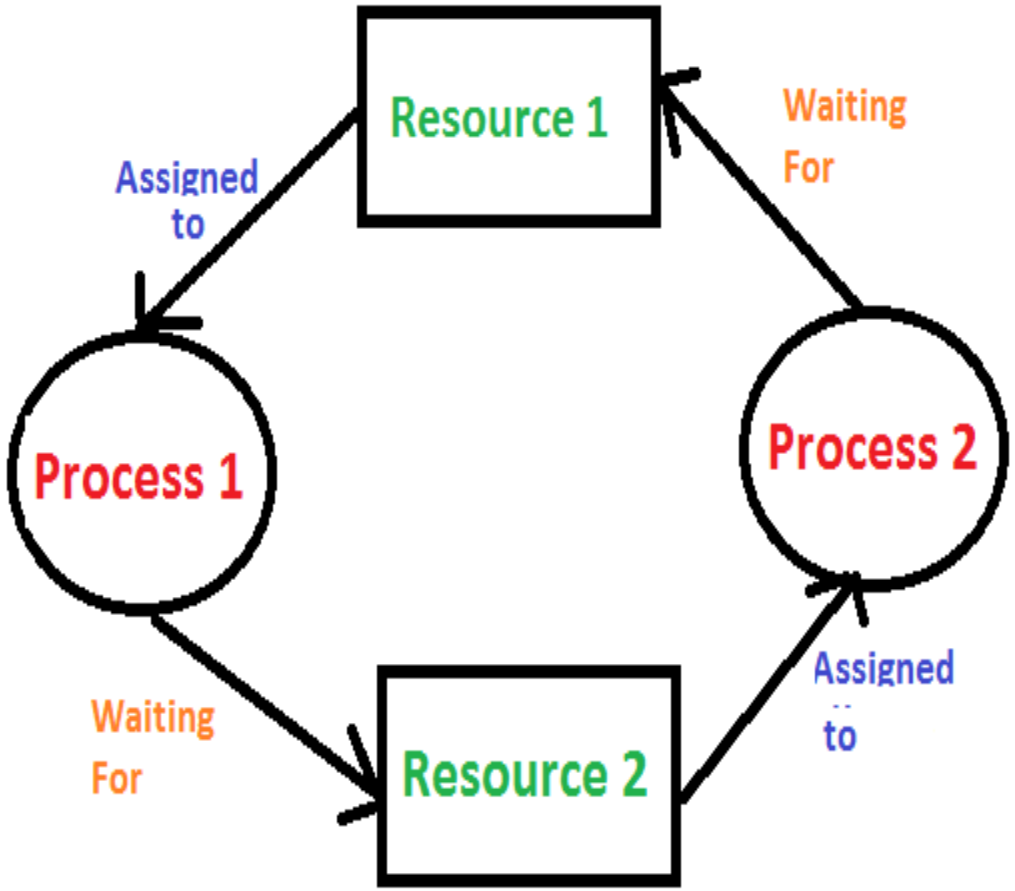
A process in operating system uses resources in the following way.
1) Requests a resource
2) Use the resource
3) Releases the resource

**A deadlock is a situation in which two computer programs sharing the same resource are effectively preventing each other from accessing the resource, resulting in both programs ceasing to function.**

**The earliest computer operating systems ran only one program at a time. All of the resources of the system were available to this one program. Later, operating systems ran multiple programs at once, interleaving them. Programs were required to specify in advance what resources they needed so that they could avoid conflicts with other programs running at the same time. Eventually some operating systems offered dynamic allocation of resources. Programs could request further allocations of resources after they had begun running. This led to the problem of the deadlock. Here is the simplest example:**

**Program 1 requests resource A and receives it.**
**Program 2 requests resource B and receives it.**
**Program 1 requests resource B and is queued up, pending the release of B.**
**Program 2 requests resource A and is queued up, pending the release of A.**

**Now neither program can proceed until the other program releases a resource. The operating system cannot know what action to take. At this point the only alternative is to abort (stop) one of the programs.**




**Deadlock can arise if** the **following four conditions hold simultaneously (Necessary Conditions)**

***Mutual Exclusion:*** Two or more resources are non-shareable (Only one process can use at a time)

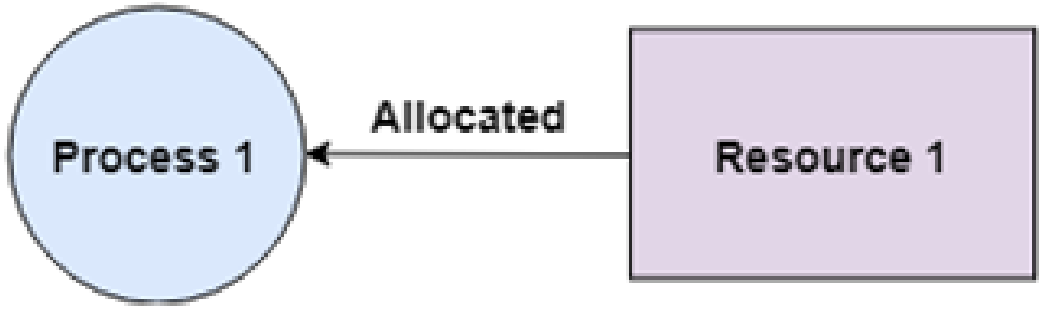
***Hold and Wait:*** A process is holding at least one resource and waiting for resources.

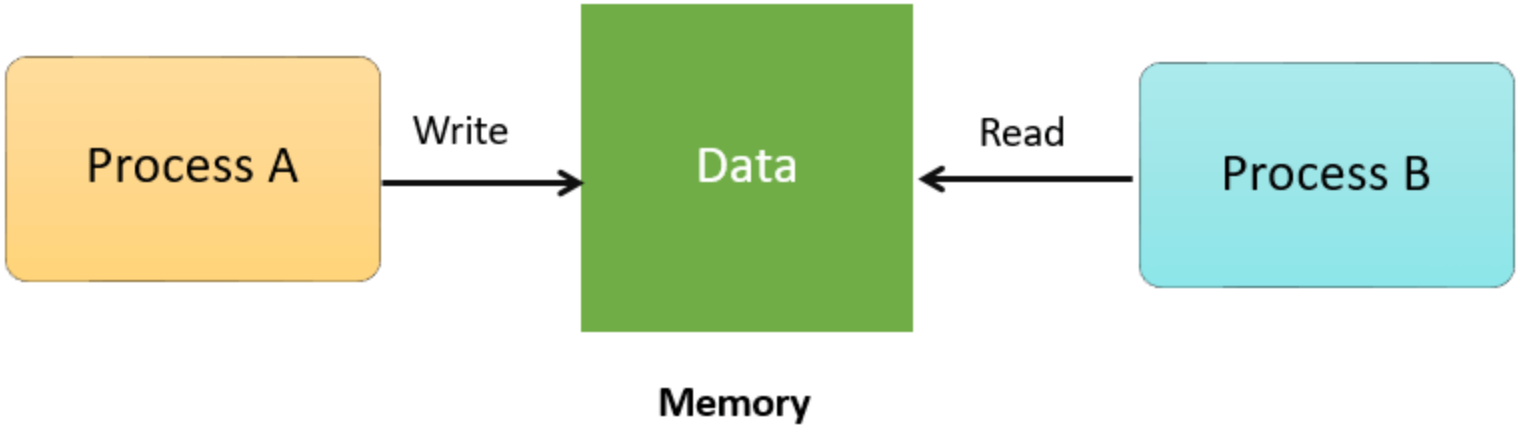
***No Preemption:*** A resource cannot be taken from a process unless the process releases the resource.

***Circular Wait:*** A set of processes are waiting for each other in circular form.

•**Mutual Exclusion**

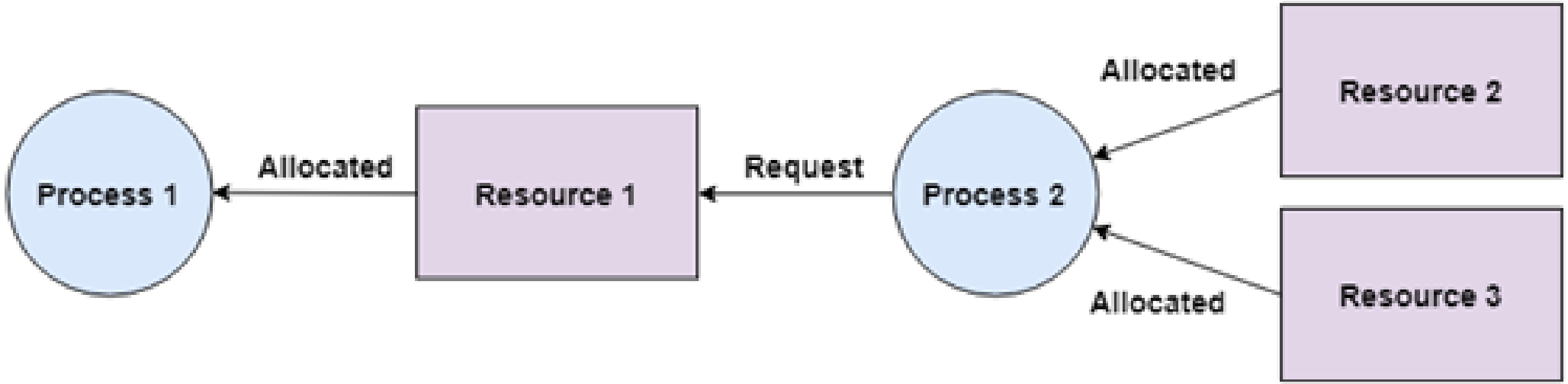
There should be a resource that can only be held by one process at a time. In the diagram below, there is a single instance of Resource 1 and it is held by Process 1 only.



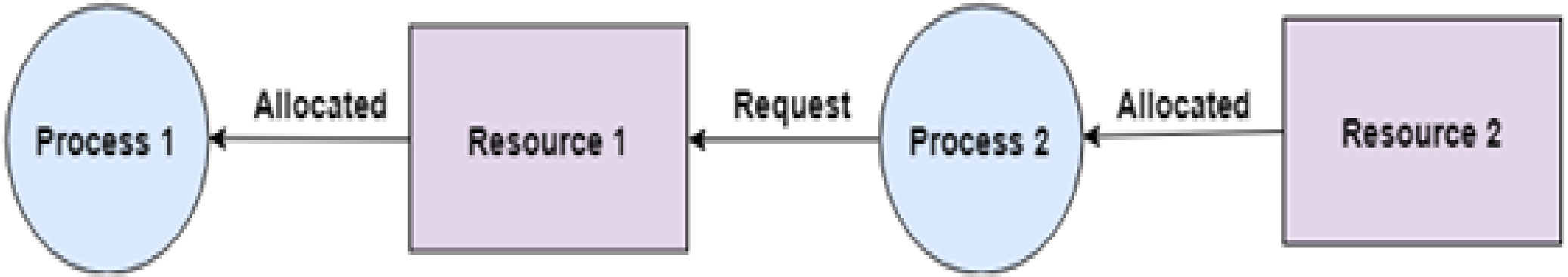
**Hold and Wait**

A process can hold multiple resources and still request more resources from other processes which are holding them. In the diagram given below, Process 2 holds Resource 2 and Resource 3 and is requesting the Resource 1 which is held by Process 1.



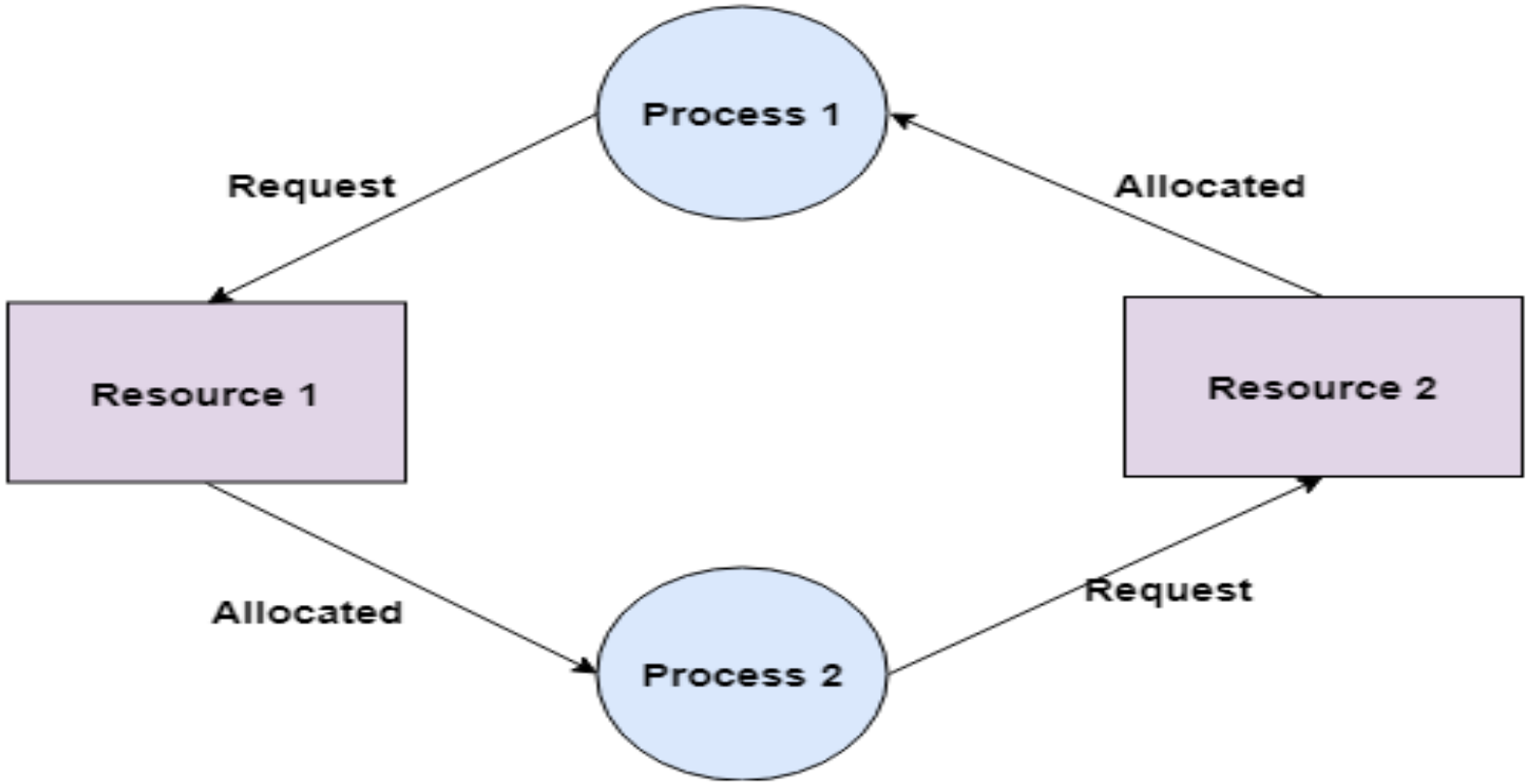
# No Preemption

A resource cannot be preempted from a process by force. A process can only release a resource voluntarily. In the diagram below, Process 2 cannot preempt Resource 1 from Process 1. It will only be released when Process 1 relinquishes it voluntarily after its execution is complete.



# Circular Wait

A process is waiting for the resource held by the second process, which is waiting for the resource held by the third process and so on, till the last process is waiting for a resource held by the first process. This forms a circular chain. For example: Process 1 is allocated Resource2 and it is requesting Resource 1. Similarly, Process 2 is allocated Resource 1 and it is requesting Resource 2. This forms a circular wait loop.



## Deadlock Detection

A deadlock can be detected by a resource scheduler as it keeps track of all the resources that are allocated to different processes. After a deadlock is detected, it can be resolved using the following methods −

- All the processes that are involved in the deadlock are terminated. This is not a good approach as all the progress made by the processes is destroyed.
- Resources can be preempted from some processes and given to others till the deadlock is resolved.

## Deadlock Prevention

It is very important to prevent a deadlock before it can occur. So, the system checks each transaction before it is executed to make sure it does not lead to deadlock. If there is even a slight chance that a transaction may lead to deadlock in the future, it is never allowed to execute.

## Deadlock Avoidance

It is better to avoid a deadlock rather than take measures after the deadlock has occurred. The wait for graph can be used for deadlock avoidance. This is however only useful for smaller databases as it can get quite complex in larger databases.

## Methods For Handling Deadlocks

Methods that are used in order to handle the problem of deadlocks are as follows:

### 1. Ignoring the Deadlock

According to this method, it is assumed that deadlock would never occur.This approach is used by many operating systems where they assume that deadlock will never occur which means operating systems simply ignores the deadlock. This approach can be beneficial for those systems that are only used for browsing and for normal tasks. Thus ignoring the deadlock method can be useful in many cases but it is not perfect in order to remove the deadlock from the operating system.

### 2.Deadlock Prevention

As we have discussed in the above section, that all four conditions: Mutual Exclusion, Hold and Wait, No preemption, and circular wait if held by a system then causes deadlock to occur. The main aim of the deadlock prevention method is to violate any one condition among the four; because if any of one condition is violated then the problem of deadlock will never occur. As the idea behind this method is simple but the difficulty can occur during the physical implementation of this method in the system.

**3.Avoiding the Deadlock**

This method is used by the operating system in order to check whether the system is in a safe state or in an unsafe state. This method checks every step performed by the operating system. Any process continues its execution until the system is in a safe state. Once the system enters into an unsafe state, the operating system has to take a step back.

Basically, with the help of this method, the operating system keeps an eye on each allocation, and make sure that allocation does not cause any deadlock in the system.

**4.Deadlock detection and recovery**

With this method, the deadlock is detected first by using some algorithms of the resource-allocation graph. This graph is mainly used to represent the allocations of various resources to different processes. After the detection of deadlock, a number of methods can be used in order to recover from that deadlock.

One way is **preemption** by the help of which a resource held by one process is provided to another process.

The second way is to **roll back**, as the operating system keeps a record of the process state and it can easily make a process roll back to its previous state due to which deadlock situation can be easily eliminate.

The third way to overcome the deadlock situation is by killing one or more processes.

# Critical Section Problem

The critical section is a code segment where the shared variables can be accessed. An atomic action is required in a critical section i.e. only one process can execute in its critical section at a time. All the other processes have to wait to execute in their critical sections.

**Solution to the Critical Section Problem**

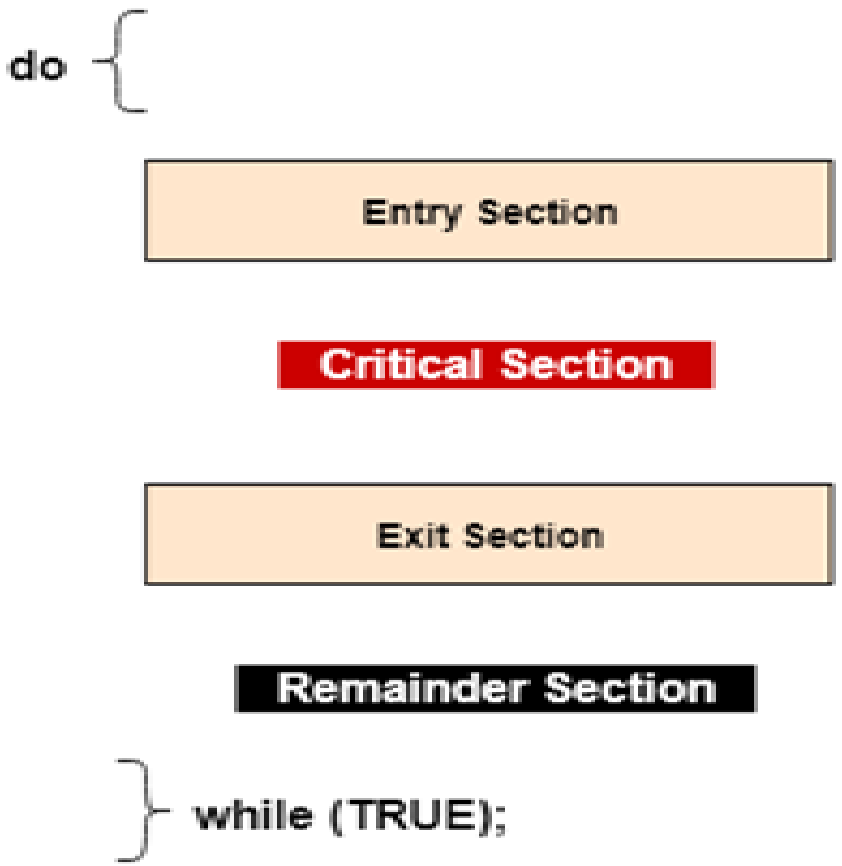
The critical section problem needs a solution to synchronize the different processes. The solution to the critical section problem must satisfy the following conditions –

•**Mutual Exclusion** Mutual exclusion implies that only one process can be inside the critical section at any time. If any other processes require the critical section, they must wait until it is free.

•**Progress** Progress means that if a process is not using the critical section, then it should not stop any other process from accessing it. In other words, any process can enter a critical section if it is free.

•**Bounded Waiting** Bounded waiting means that each process must have a limited waiting time. It should not wait endlessly to access the critical section.

**A diagram that demonstrates the critical section is as follows –**




In the above diagram, the entry section handles the entry into the critical section. It acquires the resources needed for execution by the process. The exit section handles the exit from the critical section. It releases the resources and also informs the other processes that the critical section is free.

## Semaphore

A semaphore is a variable that indicates the number of resources that are available in a system at a particular time and this semaphore variable is generally used to achieve the process synchronization. It is generally denoted by "S". You can use any other variable name of your choice.

A semaphore uses two functions i.e. wait() and signal(). Both these functions are used to change the value of the semaphore but the value can be changed by only one process at a particular time and no other process should change the value simultaneously.

The wait() function is used to decrement the value of the semaphore variable "S" by one if the value of the semaphore variable is positive. If the value of the semaphore variable is 0, then no operation will be performed.

```
wait(S)
{
while (S == 0); //there is ";" sign here
S--;
}
```

The signal() function is used to increment the value of the semaphore variable by one.

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

## Types of Semaphore

There are two types of semaphores:

**Binary Semaphores:** In Binary semaphores, the value of the semaphore variable will be 0 or 1. Initially, the value of semaphore variable is set to 1 and if some process wants to use some resource then the *wait()* function is called and the value of the semaphore is changed to 0 from 1. The process then uses the resource and when it releases the resource then the *signal()* function is called and the value of the semaphore variable is increased to 1. If at a particular instant of time, the value of the semaphore variable is 0 and some other process wants to use the same resource then it has to wait for the release of the resource by the previous process. In this way, process synchronization can be achieved.

**Counting Semaphores:** In Counting semaphores, firstly, the semaphore variable is initialized with the number of resources available. After that, whenever a process needs some resource, then the *wait()* function is called and the value of the semaphore variable is decreased by one. The process then uses the resource and after using the resource, the *signal()* function is called and the value of the semaphore variable is increased by one. So, when the value of the semaphore variable goes to 0 i.e all the resources are taken by the process and there is no resource left to be used, then if some other process wants to use resources then that process has to wait for its turn. In this way, we achieve the process synchronization.

## Advantages of Semaphores

Some of the advantages of semaphores are as follows −

•Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.

•There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.

•Semaphores are implemented in the machine independent code of the microkernel. So they are machine independent.

## Disadvantages of Semaphores

Some of the disadvantages of semaphores are as follows −

•Semaphores are complicated so the wait and signal operations must be implemented in the correct order to prevent deadlocks.

•Semaphores are impractical for last scale use as their use leads to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.

•Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.

**Mutex in Operating System**

Mutex lock in OS is essentially a variable that is binary nature that provides code wise functionality for mutual exclusion. At times, there maybe multiple threads that may be trying to access same resource like memory or I/O etc. To make sure that there is no overriding. Mutex provides a locking mechanism.

Only one thread at a time can take the ownership of a mutex and apply the lock. Once it done utilizing the resource and it may release the mutex lock.

**OR**

**Mutex** is a program object that allows multiple program threads to share the same resource, such as file access, but not simultaneously. When a program is started a mutex is created with a unique name. After this stage, any thread that needs the resource must lock the mutex from other threads while it is using the resource. The mutex is set to unlock when the data is no longer needed or the routine is finished.

# Difference between a semaphore and a mutex

## Semaphore

- It is used in the signaling mechanism.
- It is an integer variable.
- The ‘wait’ and ‘signal’ operations can be used to modify a semaphore.
- If no resources are available, the process requires that a resource that needs to be executed has to ‘wait’.
- The process will have to ‘wait’ until the count of the semaphore is higher than 0.
- Multiple program threads can be used with a semaphore.
- The value of a semaphore can be changed by any process that releases or obtains the resource.
- There are different kinds of semaphores: Counting semaphore and Binary semaphore.
- The value of a semaphore can be changed using ‘wait’ operation and ‘signal’ operation.
- The process is occupied if all resources are used.
- The process requesting the resource uses the ‘wait’ operation and blocks itself until the count of the semaphore becomes greater than 1.

## Mutex

- It is used in locking mechanism.
- It is an object.
- It is modified by that process (only) that requests or releases a resource.
- If the mutex is locked, then the process will have to wait.
- The process is kept in a queue when mutex s locked.
- This can be accessed only after the mutex has been unlocked.
- Multiple program threads can be used in mutex (but not simultaneously).
- The lock on the object can be released by the process only, which would have obtained the lock on it.
- It doesn’t have any subtypes.
- The object is either locked or unlocked.
- If the object has already been locked, the process that requests the resources will wait.
- It will be queued by the system before the lock has been released.

# Starvation

Starvation happens when a low priority program requests a system resource but cannot run because a higher priority program has been employing that resource for a long time. When a process is ready to start executing, it waits for the CPU to allocate the necessary resources. However, because other processes continue to block the required resources, the process must wait indefinitely.

In most priority scheduling algorithms, the problem of starvation arises. The resource is frequently assigned to the higher priority process in a priority scheduling method, which helps to prevent the lower priority process from obtaining the requested resource.

Starvation is an issue that can be solved through aging. Aging raises the priority of a procedure that has been waiting for resources for a long period. It also helps to prevent a low-priority procedure from waiting indefinitely for resources.

There are some common **causes** of starvation as follows:

1. Starvation may occur if there aren't enough resources to provide to every process as needed.
2. Starvation can occur if a process is never given the resources it needs for execution due to faulty resource allocation decisions.
3. If higher priority operations constantly monopolize the processor, a lower priority process may have to wait indefinitely.

Some **solutions** that may be implemented in a system that helps to handle starvation are as follows:

1. The resource allocation priority scheme should contain concepts such as aging, in which the priority of a process increases the longer it waits. It prevents starvation.
2. An independent manager may be used for the allocation of resources. This resource manager distributes resources properly and tries to prevent starvation.
3. Random process selection for resource allocation or processor allocation should be avoided since it promotes starvation.

# Main differences between the deadlock and starvation

Here, you will learn the main differences between deadlock and starvation. Various differences between deadlock and starvation are as follows:

1. Deadlock happens when every process holds a resource and waits for another process to hold another resource. In contrast, starvation happens when a low priority program requests a system resource but cannot run because a higher priority program has been employing that resource for a long time.
2. In a deadlock, none of the processes can proceed to execution; instead, each process is blocked while waiting for resources to be acquired by another process. On the other hand, starvation is a situation in which higher-priority processes have an infinite ability to acquire resources. Moreover, lower-priority processes are prevented from getting resources, resulting in their indefinite blocking.
3. Deadlock happens when four conditions exist simultaneously: mutual exclusion, hold and wait, no preemption, and circular wait. In contrast, starvation happens when process priorities are enforced while distributing resources or when resource management is unmanaged.
4. In a deadlock situation, the process blocks resources. In contrast, high-priority processes continue to use the requested resources in starvation.
5. Deadlock is also known as circular wait, whereas starvation is known as a Lived lock.

## What is dirty bit?

When a bit is modified by the CPU and not written back to the storage, it is called as a dirty bit. This bit is present in the memory cache or the virtual storage space.

# What Does Dirty Bit Mean?

Dirty bit is a colloquial name for a specific bit in computer memory that traditional computing systems use to indicate a modification or write-to. Although there is not much information on the origin of this term, one could understand the concept of the dirty bit being a temporary place marker that will eventually be erased.

A dirty bit is also known as a modified bit.

## Throttling

Adjusting the clock speed of the CPU. Also called "dynamic frequency scaling," CPU throttling is commonly used to automatically slow down the computer when possible to use less energy and conserve battery, especially in laptops. CPU throttling can also be adjusted manually to make the system quieter, because the fan can then run slower.

# What is Inter Process Communication?

**Inter process communication (IPC)** is used for exchanging data between multiple threads in one or more processes or programs. The Processes may be running on single or multiple computers connected by a network. The full form of IPC is Inter-process communication.

It is a set of programming interface which allow a programmer to coordinate activities among various program processes which can run concurrently in an operating system. This allows a specific program to handle many user requests at the same time.
Since every single user request may result in multiple processes running in the operating system, the process may require to communicate with each other.

## Approaches to Interprocess Communication

We will now discuss some different approaches to inter-process communication which are as follows:

These are a few different approaches for Inter- Process Communication:

1. **Pipes**
2. **Shared Memory**
3. **Message Queue**
4. **FIFO**

**Pipe:-**

The pipe is a type of data channel that is unidirectional in nature. It means that the data in this type of data channel can be moved in only a single direction at a time. Still, one can use two-channel of this type, so that he can able to send and receive data in two processes. Typically, it uses the standard methods for input and output. These pipes are used in all types of POSIX systems and in different versions of window operating systems as well.

**Shared Memory:-**

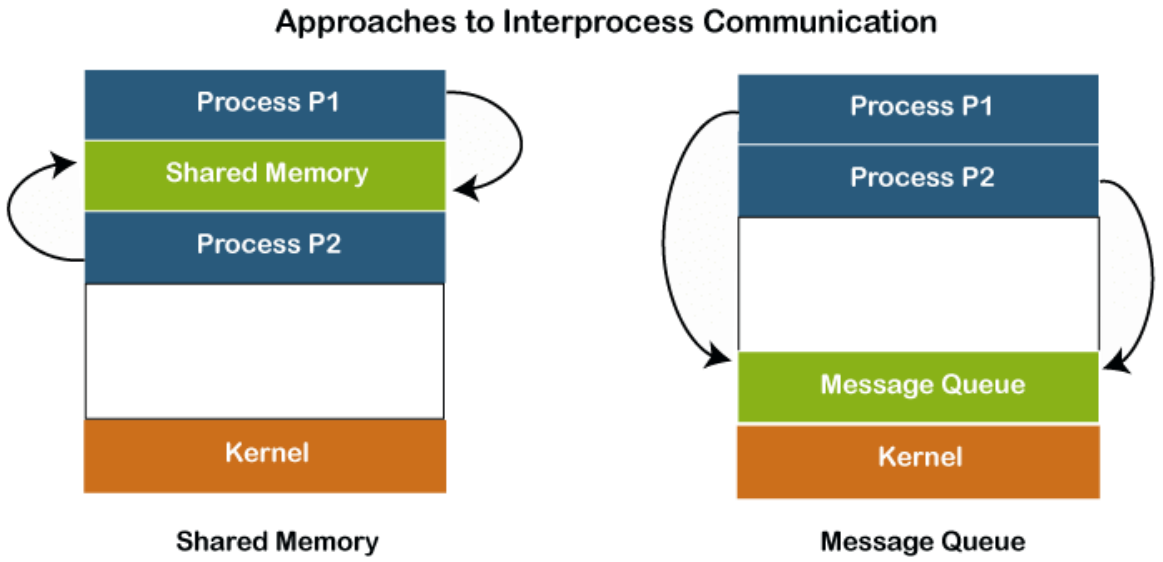
It can be referred to as a type of memory that can be used or accessed by multiple processes simultaneously. It is primarily used so that the processes can communicate with

each other. Therefore the shared memory is used by almost all POSIX and Windows operating systems as well.

**Message Queue:-**

In general, several different messages are allowed to read and write the data to the message queue. In the message queue, the messages are stored or stay in the queue unless their recipients retrieve them. In short, we can also say that the message queue is very helpful in inter-process communication and used by all operating systems.

To understand the concept of Message queue and Shared memory in more detail, let's take a look at its diagram given below:




# Named Pipe or 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 traditional pipe is “unnamed” and lasts only as long as the process.
- 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.

# IPC using Message Queues

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.

The message queue is a buffer that is used in non-shared memory environments, where tasks communicate by passing messages to each other rather than by accessing shared variables. Tasks share a common buffer pool. The message queue is an unbounded FIFO queue that is protected from concurrent access by different threads.

Events are asynchronous. When a class sends an event to another class, rather than sending it directly to the target reactive class, it passes the event to the operating system message queue. The target class retrieves the event from the head of the message queue when it is ready to process it. Synchronous events can be passed using triggered operations instead.

## Functions of Message Queue

There are four important functions that we will use in the programs to achieve IPC using message queues.

**1. int msgget (key_t key, int msgflg);**

We use the msgget function to create and access a message queue. It takes two parameters.

- The first parameter is a key that names a message queue in the system.
- The second parameter is used to assign permission to the message queue and is ORed with IPC_CREAT to create the queue if it doesn't already exist. If the queue already exists, then IPC_CREAT is ignored. On success, the msgget function returns a positive number which is the queue identifier, while on failure, it returns -1.

**2. int msgsnd (int msqid, const void *msg_ptr, size_t msg_sz, int msgflg);**

This function allows us to add a message to the message queue.

- The first parameter (msgid) is the message queue identifier returned by the msgget function.
- The second parameter is the pointer to the message to be sent, which must start with a long int type.
- The third parameter is the size of the message. It must not include the long int message type.
- The fourth and final parameter controls what happens if the message queue is full or the system limit on queued messages is reached. The function on success returns 0 and place the copy of message data on the message queue. On failure, it returns -1.

There are two constraints related to the structure of the message. First, it must be smaller than the system limit and, second, it must start with a long int. This long int is used as a message type in the receive function. The best structure of the message is shown below.

Many tasks can write messages into the queue, but only one can read messages from the queue at a time. The reader waits on the message queue until there is a message to process. Messages can be of any size.

```
struct my_message
{
long int message_type;
/* The data you wish to transfer */
```

}

Since the message_type is used in message reception, you can't simply ignore it. You must declare your data structure to include it, and it's also wise to initialize it to contain a known value.

**3. int msgrcv (int msqid, void *msg_ptr, size_t msg_sz, long int msgtype, int msgflg);**

This function retrieves messages from a message queue.

- The first parameter (msgid) is the message queue identifier returned by the msgget function.
- As explained above, the second parameter is the pointer to the message to be received, which must start with a long int type.
- The third parameter is the size of the message.
- The fourth parameter allows implementing priority. If the value is 0, the first available message in the queue is retrieved. But if the value is greater than 0, then the first message with the same message type is retrieved. If the value is less than 0, then the first message having the type value same as the absolute value of msgtype is retrieved. In simple words, 0 value means to receive the messages in the order in which they were sent, and non zero means receive the message with a specific message type.
- The final parameter controls what happens if the message queue is full or the system limit on queued messages is reached. The function on success returns 0 and place the copy of message data on the message queue. On failure, it returns -1.

**4. int msgctl (int msqid, int command, struct msqid_ds *buf);**

The final function is msgctl, which is the control function.

- The first parameter is the identifier returned by the msgget function.
- The second parameter can have one out of the below three values.

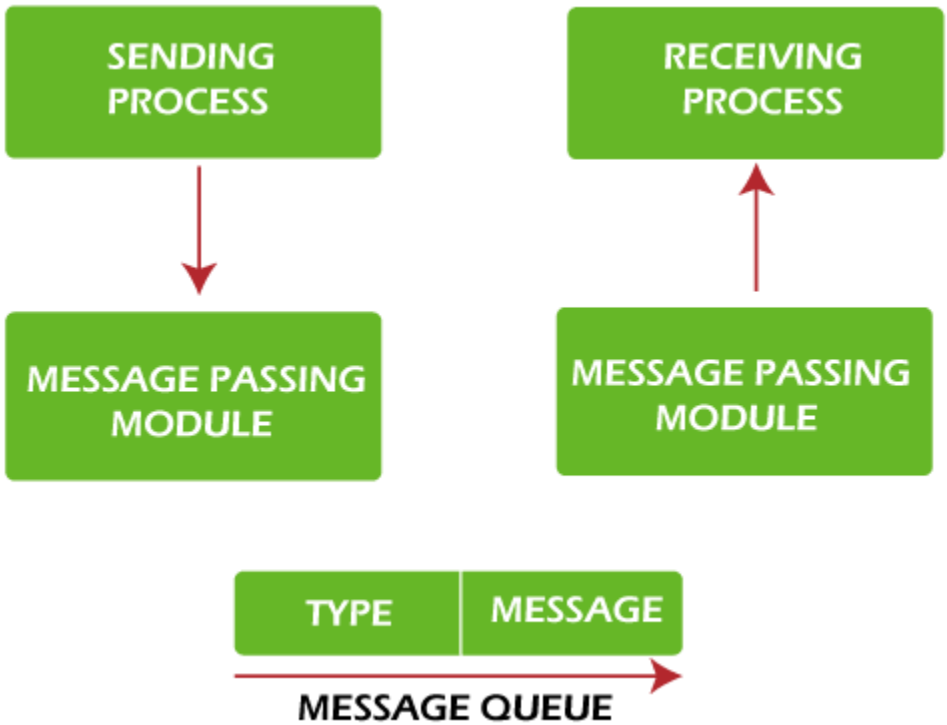
| Command | Description |
| --- | --- |

| | |
|---|---|
| IPC_STAT | It sets the data in the msqid_ds structure to reflect the values associated with the message queue. |
| IPC_SET | If the process has permission to do so, this sets the values associated with the message queue to those provided in the msqid_ds data structure. |
| IPC_RMID | It deletes the message queue. |

The msgctl function returns 0 on success and -1 on error. The send or receive function will fail if a message queue is deleted while a process is waiting in an msgsnd or msgrcv function.

## Steps to Perform IPC using Message Queues

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. Below are the following steps to perform communication using message queues.




1. A new queue is created or an existing queue opened by **msgget()**.
2. New messages are added to the end of a queue by **msgsnd()**. Every message has a positive long integer type field, a non-negative length, and the actual data bytes

(corresponding to the length), all specified to msgsnd() when the message is added to a queue.

3. Messages are fetched from a queue by **msgrcv()**. We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field. All processes can exchange information through access to a common system message queue. The sending process places a message onto a queue that another process can read. Each message is given an identification or type so that processes can select the appropriate message. The process must share a common key to gain access to the queue in the first place.
4. Perform control operations on the message queue **msgctl()**.

## Examples

**Program 1:** Let's write a program for IPC using Message Queues to send data to a message queue.

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#define MAX_TEXT 512   //maximum length of the message that can be sent allowed
struct my_msg{
    long int msg_type;
    char some_text[MAX_TEXT];
};
int main()
{
    int running=1;
    int msgid;
```

```
    struct my_msg some_data;
    char buffer[50]; //array to store user input
    msgid=msgget((key_t)14534,0666|IPC_CREAT);
    if (msgid == -1) // -1 means the message queue is not created
    {
        printf("Error in creating queue\n");
        exit(0);
    }

    while(running)
    {
        printf("Enter some text:\n");
        fgets(buffer,50,stdin);
        some_data.msg_type=1;
        strcpy(some_data.some_text,buffer);
        if(msgsnd(msgid,(void *)&some_data, MAX_TEXT,0)==-1) // msgsnd returns -1 if the message
is not sent
        {
            printf("Msg not sent\n");
        }
        if(strncmp(buffer,"end",3)==0)
        {
            running=0;
        }
    }
}
```

## Output

The above program gives this output.

```
[root@localhost ~]# ./send
Enter some text
red
Enter some text
hat
Enter some text
end
```

**How does it Work?**

The structure my_msg declares the long int variable and the char array to store the data sent to the message queue. Then the message queue is created using the msgget() function. Next, read data from the user into the buffer using fgets() and then copy it into the variable some_text of the structure some_data. Finally, send the data to the queue using the msgsnd() function. The strcmp function is used to stop sending the data by comparing the first three characters of the data. If the data starts with "end", it means no more data is to be sent.

**Program 2:** Now, let's write a program for IPC using Message Queues to receive or read message from the above created message queue.

```
#include<stdlib.h>
 #include<stdio.h>
 #include<string.h>
 #include<unistd.h>
 #include<sys/types.h>
 #include<sys/ipc.h>
 #include<sys/msg.h>
 struct my_msg{
        long int msg_type;
        char some_text[BUFSIZ];
 };
 int main()
 {
        int running=1;
```

```
    int msgid;
    struct my_msg some_data;
    long int msg_to_rec=0;
    msgid=msgget((key_t)12345,0666|IPC_CREAT);
    while(running)
    {
        msgrcv(msgid,(void *)&some_data,BUFSIZ,msg_to_rec,0);
        printf("Data received: %s\n",some_data.some_text);
        if(strncmp(some_data.some_text,"end",3)==0)
        {
            running=0;
        }
    }
    msgctl(msgid,IPC_RMID,0);
}
```

Output

The above code gives the following output.

```
[root@localhost ~]# ./rec
Data received: red

Data received: hat

Data received: end

[root@localhost ~]# _
```

### How does it work?

The msg_to_rec variable is set to 0 so that the data is received in the same order as sent. The while is used to continuously receive the data using the mgrcv() function until the text received is "end", which we check using the strcmp function. The data is read using the structure my_msg.