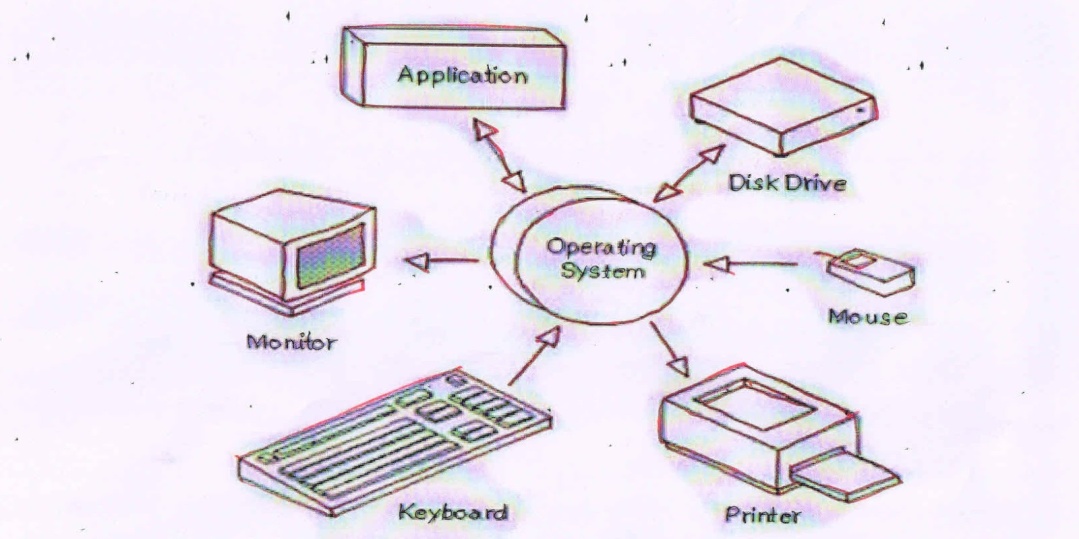
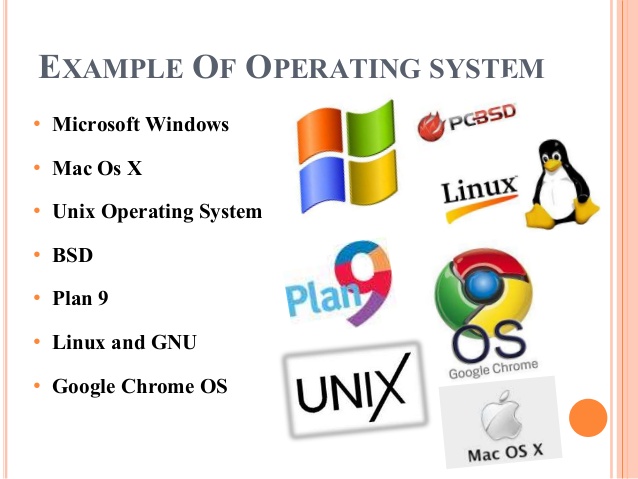
**Operating system**

**Introduction to OS**

* **What is OS; How is it different from other application software; Why is it hardware dependent**
* **Different components of OS**
* **Basic computer organization required for OS**
* **Examples of well known OS including mobile OS, embedded system OS, Real Time OS, desktop OS server machine OS etc. ; How are these different from each other and why**
* **Functions of OS**
* **User and Kernel space and mode; Interrupts and system calls**
* **What is OS-** Operating System lies in the category of system software It basically manages all the resources of the computer. An operating system acts as an interface between the software and different parts of the computer or the computer

****

****

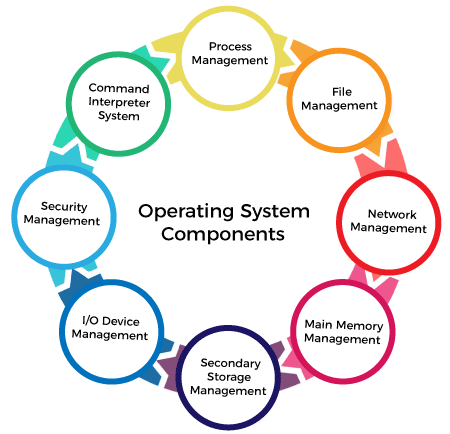
* **How is it different from other application software** : The difference between operating system and application software is that an**operating system is a system software that works as the interface between the user and the hardware while the application software is a program that performs a specific task**. It is not possible to install application software to the computer without an operating system.

**Component of operating system :-**

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. ***Process Management***
2. ***File Management***
3. ***Network Management***
4. ***Main Memory Management***
5. ***Secondary Storage Management***
6. ***I/O Device Management***
7. ***Security Management***
8. ***Command Interpreter System***



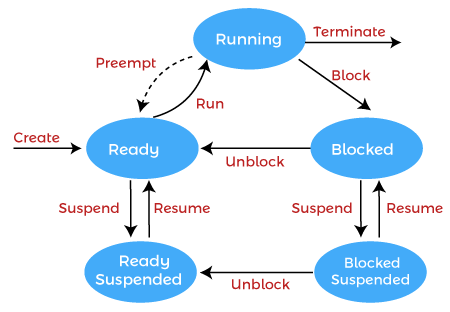
### Process Management

The process management component is a procedure for managing many processes running simultaneously on the operating system. Every running software application program has one or more processes associated with them.

For example, when you use a search engine like Chrome, there is a process running for that browser program.

Process management keeps processes running efficiently. It also uses memory allocated to them and shutting them down when needed.

The execution of a process must be sequential so, at least one instruction should be executed on behalf of the process.

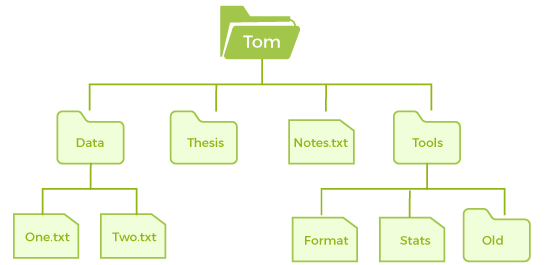


* File Management

A file is a set of related information defined by its creator. It commonly represents programs (both source and object forms) and data. Data files can be alphabetic, numeric, or alphanumeric.

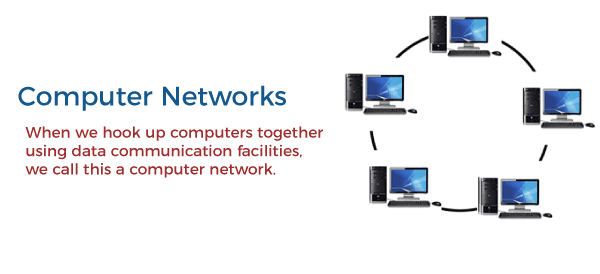
The operating system has the following important activities in connection with file management:

* File and directory creation and deletion.
* For manipulating files and directories.
* Mapping files onto secondary storage.
* Backup files on stable storage media.



### Network Management

Network management is the process of administering and managing computer networks. It includes performance management, provisioning of networks, fault analysis, and maintaining the quality of service.



A distributed system is a collection of computers or processors that never share their memory and clock. In this type of system, all the processors have their local memory, and the processors communicate with each other using different communication cables, such as fiber optics or telephone lines.

The computers in the network are connected through a communication network, which can configure in many different ways. The network can fully or partially connect in network management, which helps users design routing and connection strategies that overcome connection and security issues.

**Functions of Network management**

Network management provides the following functions, such as:

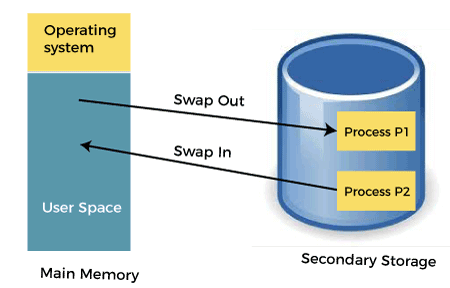
* Distributed systems help you to various computing resources in size and function. They may involve minicomputers, microprocessors, and many general-purpose computer systems.
* A distributed system also offers the user access to the various resources the network shares.
* It helps to access shared resources that help computation to speed up or offers data availability and reliability.

### Main Memory management

Main memory is a large array of storage or bytes, which has an address. The memory management process is conducted by using a sequence of reads or writes of specific memory addresses.

It should be mapped to absolute addresses and loaded inside the memory to execute a program. The selection of a memory management method depends on several factors.

However, it is mainly based on the hardware design of the system. Each algorithm requires corresponding hardware support. Main memory offers fast storage that can be accessed directly by the CPU. It is costly and hence has a lower storage capacity. However, for a program to be executed, it must be in the main memory.



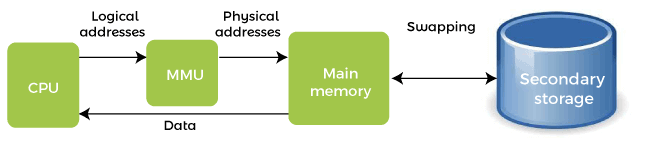
**Functions of Memory management**

An Operating System performs the following functions for Memory Management in the operating system:

* It helps you to keep track of primary memory.
* Determine what part of it are in use by whom, what part is not in use.
* In a multiprogramming system, the OS decides which process will get memory and how much.
* Allocates the memory when a process requests.
* It also de-allocates the memory when a process no longer requires or has been terminated.

### Secondary-Storage Management

The most important task of a computer system is to execute programs. These programs help you to access the data from the main memory during execution. This memory of the computer is very small to store all data and programs permanently. The computer system offers secondary storage to back up the main memory.



Today modern computers use hard drives/SSD as the primary storage of both programs and data. However, the secondary storage management also works with storage devices, such as USB flash drives and CD/DVD drives. Programs like assemblers and compilers are stored on the disk until it is loaded into memory, and then use the disk is used as a source and destination for processing.

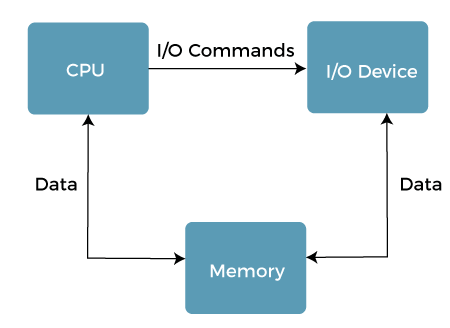
**Functions of Secondary storage management**

Here are some major functions of secondary storage management in the operating system:

* Storage allocation :- AUTOMATIC specifies that storage is allocated upon each entry to the block that contains the storage declaration.
* Free space management : Free space management in Operating System. The system keeps tracks of the free disk blocks for allocating space to files when they are created. Also, to reuse the space released from deleting the files, free space management becomes crucial.
* Disk scheduling :- Disk scheduling is done by operating systems to **schedule I/O requests arriving for the disk**. Disk scheduling is also known as I/O scheduling

### I/O Device Management

One of the important use of an operating system that helps to hide the variations of specific hardware devices from the user.



**Functions of I/O management**

The I/O management system offers the following functions, such as:

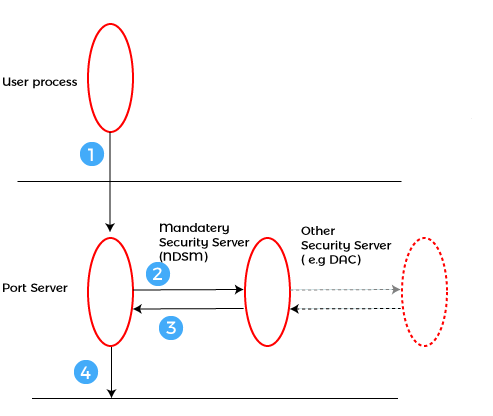
* It offers a buffer caching system
* It provides general device driver code
* It provides drivers for particular hardware devices.
* I/O helps you to know the individualities of a specific device.

#### NOTE: The user's program can't execute I/O operations directly. The operating system should provide some medium to perform this.

### Security Management

The various processes in an operating system need to be secured from other activities. Therefore, various mechanisms can ensure those processes that want to operate files, memory CPU, and other hardware resources should have proper authorization from the operating system.

Security refers to a mechanism for controlling the access of programs, processes, or users to the resources defined by computer controls to be imposed, together with some means of enforcement.

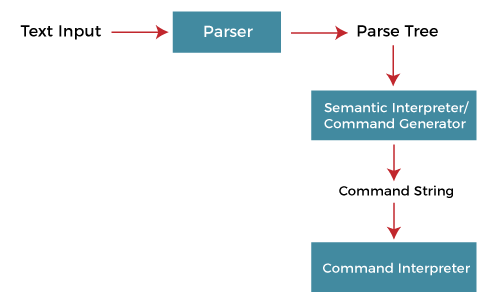


For example, memory addressing hardware helps to confirm that a process can be executed within its own address space. The time ensures that no process has control of the CPU without renouncing it. Lastly, no process is allowed to do its own I/O to protect, which helps you to keep the integrity of the various peripheral devices.

Security can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can prevent the foulness of a healthy subsystem by a malfunctioning subsystem. An unprotected resource cannot misuse by an unauthorized or incompetent user.

### Command Interpreter System

One of the most important components of an operating system is its command interpreter. The command interpreter is the primary interface between the user and the rest of the system.



Many commands are given to the operating system by control statements. A program that reads and interprets control statements is automatically executed when a new job is started in a batch system or a user logs in to a time-shared system. This program is variously called.

* The control card interpreter,
* The command-line interpreter,
* The shell (in UNIX), and so on.

Its function is quite simple, get the next command statement, and execute it. The command statements deal with process management, I/O handling, secondary storage management, main memory management, file system access, protection, and networking.

* **Examples of well known OS including mobile OS, embedded system OS, Real Time OS, desktop OS server machine OS etc.**

The most well-known mobile OSs are **Android, iOS, Windows phone OS, and Symbian**. The market share ratios of those OSs are Android 47.51%, iOS 41.97%, Symbian 3.31%, and Windows phone OS 2.57%. There are some other mobile OSs that are less used (BlackBerry, Samsung, etc.)

**1. Android OS:** The [Android operating system](https://www.javatpoint.com/android-tutorial) is the most popular [operating system](https://www.javatpoint.com/operating-system-interview-questions) today. It is a mobile OS based on the **Linux Kernel** and **open-source software**. The android operating system was developed by **Google**. The first Android device was launched in **2008**.

**2. Bada (Samsung Electronics):** Bada is a Samsung mobile operating system that was launched in 2010. The Samsung wave was the first mobile to use the bada operating system. The bada operating system offers many mobile features, such as 3-D graphics, application installation, and multipoint-touch.

**3. BlackBerry OS:** The BlackBerry [operating system](https://www.javatpoint.com/operating-system) is a mobile operating system developed by **Research In Motion** (RIM). This operating system was designed specifically for BlackBerry handheld devices. This operating system is beneficial for the corporate users because it provides synchronization with Microsoft Exchange, Novell GroupWise email, Lotus Domino, and other business software when used with the BlackBerry Enterprise Server.

**4. iPhone OS / iOS:** The iOS was developed by the Apple inc for the use on its device. The iOS operating system is the most popular operating system today. It is a very secure operating system. The iOS operating system is not available for any other mobiles.

**5. Symbian OS:** Symbian operating system is a mobile operating system that provides a high-level of integration with communication. The Symbian operating system is based on the java language. It combines middleware of wireless communications and personal information management (PIM) functionality. The Symbian operating system was developed by **Symbian Ltd** in **1998** for the use of mobile phones. **Nokia** was the first company to release Symbian OS on its mobile phone at that time.

**6. Windows Mobile OS:** The window mobile OS is a mobile operating system that was developed by **Microsoft**. It was designed for the pocket PCs and smart mobiles.

**7. Harmony OS:** The harmony operating system is the latest mobile operating system that was developed by Huawei for the use of its devices. It is designed primarily for IoT devices.

**8. Palm OS:** The palm operating system is a mobile operating system that was developed by **Palm Ltd** for use on personal digital assistants (PADs). It was introduced in **1996**. Palm OS is also known as the **Garnet OS**.

**9. WebOS (Palm/HP):** The WebOS is a mobile operating system that was developed by **Palm**. It based on the **Linux Kernel**. The HP uses this operating system in its mobile and touchpads.

* Functions of OS:-

**Important functions of an operating System:**

1. **Security –**   
   The operating system uses password protection to protect user data and similar other techniques. it also prevents unauthorized access to programs and user data.
2. **Control over system performance –**   
   Monitors overall system health to help improve performance. records the response time between service requests and system response to having a complete view of the system health. This can help improve performance by providing important information needed to troubleshoot problems.
3. **Job accounting –**   
   Operating system Keeps track of time and resources used by various tasks and users, this information can be used to track resource usage for a particular user or group of users.
4. **Error detecting aids –**   
   The operating system constantly monitors the system to detect errors and avoid the malfunctioning of a computer system.
5. **Coordination between other software and users –**   
   Operating systems also coordinate and assign interpreters, compilers, assemblers, and other software to the various users of the computer systems.
6. **Memory Management –**   
   The operating system manages the Primary Memory or Main Memory. Main memory is made up of a large array of bytes or words where each byte or word is assigned a certain address. Main memory is fast storage and it can be accessed directly by the CPU. For a program to be executed, it should be first loaded in the main memory. An Operating System performs the following activities for memory management:

It keeps track of primary memory, i.e., which bytes of memory are used by which user program. The memory addresses that have already been allocated and the memory addresses of the memory that has not yet been used. In multiprogramming, the OS decides the order in which processes are granted access to memory, and for how long. It Allocates the memory to a process when the process requests it and deallocates the memory when the process has terminated or is performing an I/O operation. 

1. **Processor Management –**   
   In a multi-programming environment, the OS decides the order in which processes have access to the processor, and how much processing time each process has. This function of OS is called process scheduling. An Operating System performs the following activities for processor management.

Keeps track of the status of processes. The program which performs this task is known as a traffic controller. Allocates the CPU that is a processor to a process. De-allocates processor when a process is no more required. 

1. **Device Management –**   
   An OS manages device communication via their respective drivers. It performs the following activities for device management. Keeps track of all devices connected to the system. designates a program responsible for every device known as the Input/Output controller. Decides which process gets access to a certain device and for how long. Allocates devices in an effective and efficient way. Deallocates devices when they are no longer required.
2. **File Management –**   
   A file system is organized into directories for efficient or easy navigation and usage. These directories may contain other directories and other files. An Operating System carries out the following file management activities. It keeps track of where information is stored, user access settings and status of every file, and more… These facilities are collectively known as the file system.

Moreover, Operating System also provides certain services to the computer system in one form or the other.   
The Operating System provides certain services to the users which can be listed in the following manner:

1. **Program Execution**: The Operating System is responsible for the execution of all types of programs whether it be user programs or system programs. The Operating System utilizes various resources available for the efficient running of all types of functionalities.
2. **Handling Input/Output Operations**: The Operating System is responsible for handling all sorts of inputs, i.e, from the keyboard, mouse, desktop, etc. The Operating System does all interfacing in the most appropriate manner regarding all kinds of Inputs and Outputs.   
   For example, there is a difference in the nature of all types of peripheral devices such as mice or keyboards, the Operating System is responsible for handling data between them.
3. **Manipulation of File System**: The Operating System is responsible for making decisions regarding the storage of all types of data or files, i.e, floppy disk/hard disk/pen drive, etc. The Operating System decides how the data should be manipulated and stored.
4. **Error Detection and Handling**: The Operating System is responsible for the detection of any type of error or bugs that can occur while any task. The well-secured OS sometimes also acts as a countermeasure for preventing any sort of breach to the Computer System from any external source and probably handling them.
5. **Resource Allocation:** The Operating System ensures the proper use of all the resources available by deciding which resource to be used by whom for how much time. All the decisions are taken by the Operating System.
6. **Accounting:** The Operating System tracks an account of all the functionalities taking place in the computer system at a time. All the details such as the types of errors that occurred are recorded by the Operating System.
7. **Information and Resource Protection:** The Operating System is responsible for using all the information and resources available on the machine in the most protected way. The Operating System must foil an attempt from any external resource to hamper any sort of data or information.

User mode and Kernel Mode :

**User Mode:** When a Program is booted up on an Operating system let’s say windows, then it launches the program in user mode. And when a user-mode program requests to run, a process and virtual address space (address space for that process) is created for it by windows. [User-mode](https://www.geeksforgeeks.org/dual-mode-operations-os/) programs are less privileged than user-mode applications and are not allowed to access the system resources directly. For instance, if an application under user-mode wants to access system resources, it will have to first go through the Operating system kernel by using syscalls.

**Kernel Mode:** The kernel is the core program on which all the other operating system components rely, it is used to access the hardware components and schedule which processes should run on a computer system and when, and it also manages the application software and hardware interaction. Hence it is the most privileged program, unlike other programs it can directly interact with the hardware. When programs running under user mode need hardware access for example webcam, then first it has to go through the kernel by using a syscall, and to carry out these requests the CPU switches from user mode to kernel mode at the time of execution. After finally completing the execution of the process the [CPU again switches back to the user mode](https://www.geeksforgeeks.org/user-mode-and-kernel-mode-switching/).

Session 2 :

• Working basics of file system

• Commands associated with files/directories & other basic commands. Operators like  
redirection, pipe  
• What are file permissions and how to set them  
• Permissions (chmod, chown, etc); access control list; network commands (telenet, ftp, ssh,  
sftp, finger)  
• System variables like – PS1, PS2 etc. How to set them  
Shell Programming  
• What is shell; What are different shells in Linux?  
• Shell variables; Wildcard symbols  
• Shell meta characters; Command line arguments; Read, Echo  
Lab:  
• Working with various OS commands  
• Shell programs related to Session 2.

* Working basics of file system:

A file is a collection of related information that is recorded on secondary storage. Or file is a collection of logically related entities. From user’s perspective a file is the smallest allotment of logical secondary storage.

**The name  of the file is divided into two parts as shown below:**

* name
* extension, separated by a period.

**FILE DIRECTORIES:**   
Collection of files is a file directory. The directory contains information about the files, including attributes, location and ownership. Much of this information, especially that is concerned with storage, is managed by the operating system. The directory is itself a file, accessible by various file management routines.

**Information contained in a device directory are:**

* Name
* Type
* Address
* Current length
* Maximum length
* Date last accessed
* Date last updated
* Owner id
* Protection information

**Operation performed on directory are:**

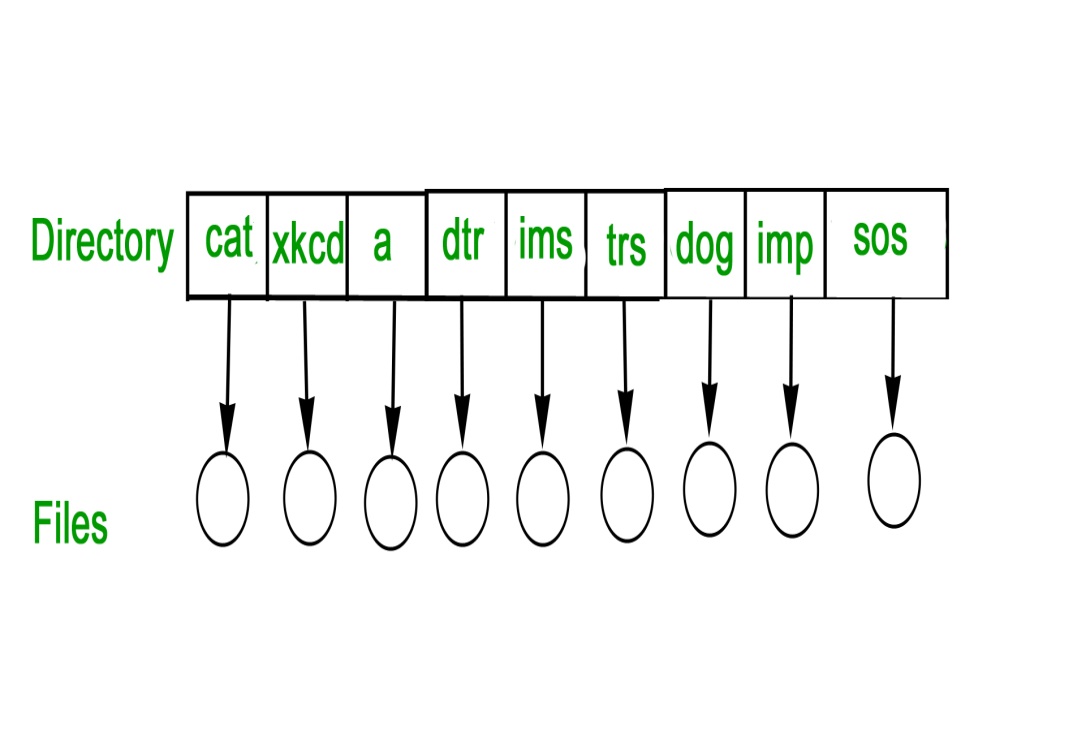
* Search for a file
* Create a file
* Delete a file
* List a directory
* Rename a file
* Traverse the file system

**Advantages of maintaining directories are:**

* **Efficiency:** A file can be located more quickly.
* **Naming:** It becomes convenient for users as two users can have same name for different files or may have different name for same file.
* **Grouping:** Logical grouping of files can be done by properties e.g. all java programs, all games etc.

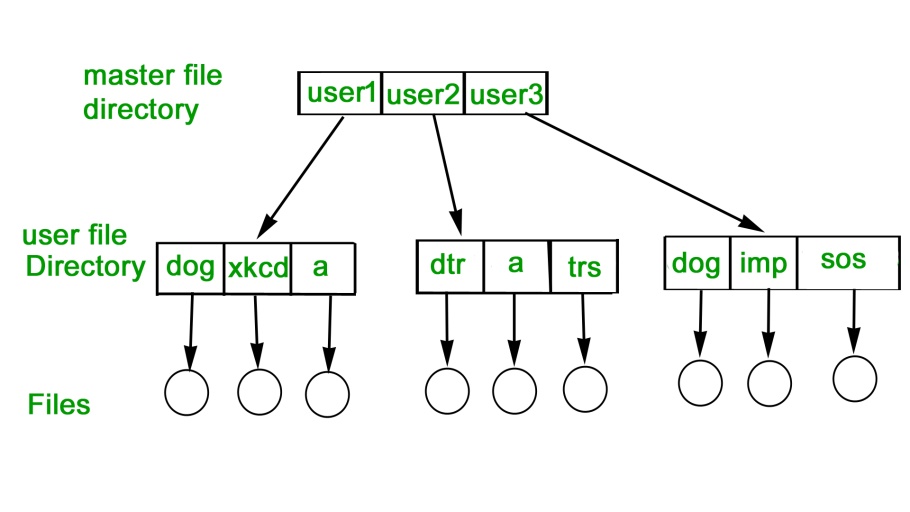
**SINGLE-LEVEL DIRECTORY**   
In this a single directory is maintained for all the users.

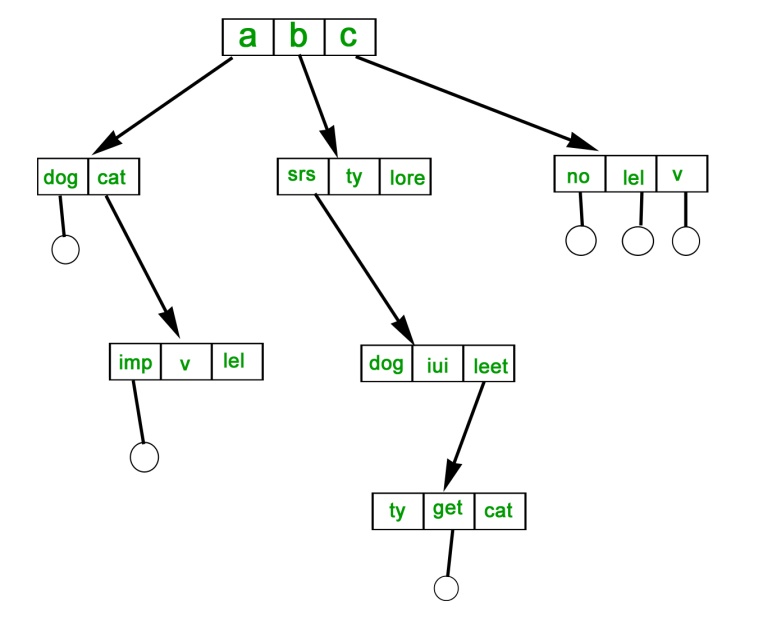
* **Naming problem:** Users cannot have same name for two files.
* **Grouping problem:** Users cannot group files according to their need.



**TWO-LEVEL DIRECTORY**   
In this separate directories for each user is maintained. 

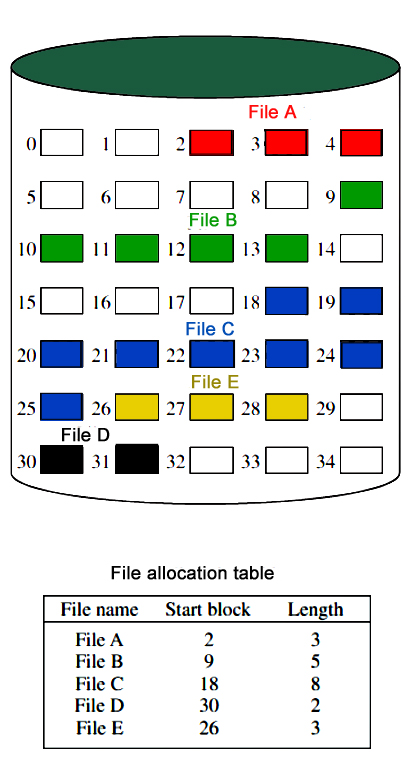
* Path name:Due to two levels there is a path name for every file to locate that file.
* Now,we can have same file name for different user.
* Searching is efficient in this method.



**TREE-STRUCTURED DIRECTORY :**   
Directory is maintained in the form of a tree. Searching is efficient and also there is grouping capability. We have absolute or relative path name for a file.   
 

**FILE ALLOCATION METHODS** **:**

**1. Continuous Allocation –**  
A single continuous set of blocks is allocated to a file at the time of file creation. Thus, this is a pre-allocation strategy, using variable size portions. The file allocation table needs just a single entry for each file, showing the starting block and the length of the file. This method is best from the point of view of the individual sequential file. Multiple blocks can be read in at a time to improve I/O performance for sequential processing. It is also easy to retrieve a single block. For example, if a file starts at block b, and the ith block of the file is wanted, its location on secondary storage is simply b+i-1. 



**Disadvantage –**

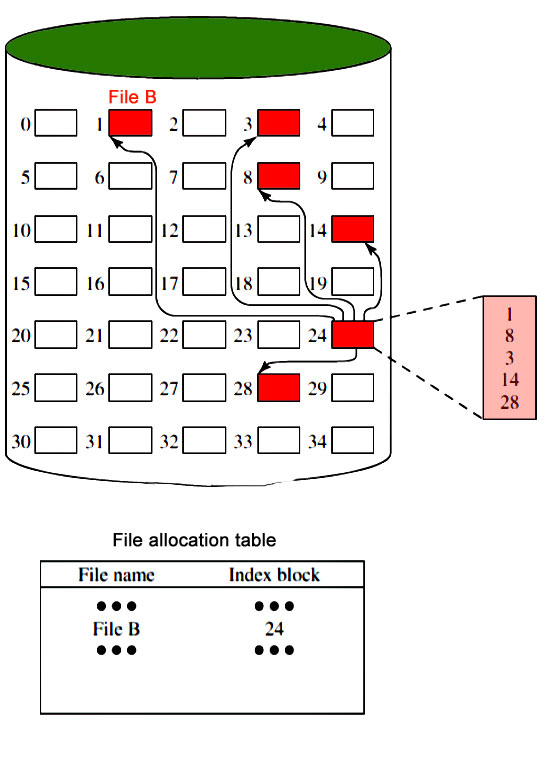
* External fragmentation will occur, making it difficult to find contiguous blocks of space of sufficient length. Compaction algorithm will be necessary to free up additional space on disk.
* Also, with pre-allocation, it is necessary to declare the size of the file at the time of creation.

**2. Linked Allocation(Non-contiguous allocation) –**  
Allocation is on an individual block basis. Each block contains a pointer to the next block in the chain. Again the file table needs just a single entry for each file, showing the starting block and the length of the file. Although pre-allocation is possible, it is more common simply to allocate blocks as needed. Any free block can be added to the chain. The blocks need not be continuous. Increase in file size is always possible if free disk block is available. There is no external fragmentation because only one block at a time is needed but there can be internal fragmentation but it exists only in the last disk block of file.

**Disadvantage –**

* Internal fragmentation exists in last disk block of file.
* There is an overhead of maintaining the pointer in every disk block.
* If the pointer of any disk block is lost, the file will be truncated.
* It supports only the sequential access of files.

**3. Indexed Allocation –**  
It addresses many of the problems of contiguous and chained allocation. In this case, the file allocation table contains a separate one-level index for each file: The index has one entry for each block allocated to the file. Allocation may be on the basis of fixed-size blocks or variable-sized blocks. Allocation by blocks eliminates external fragmentation, whereas allocation by variable-size blocks improves locality. This allocation technique supports both sequential and direct access to the file and thus is the most popular form of file allocation. 



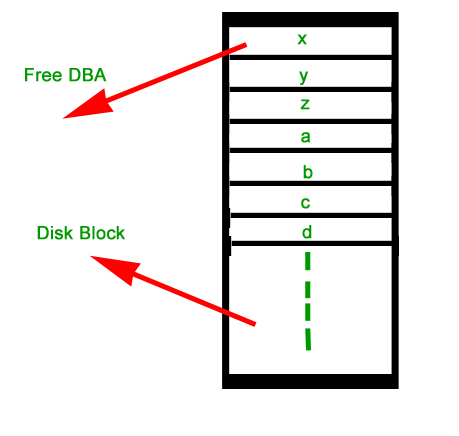
**Disk Free Space Management :**

Just as the space that is allocated to files must be managed ,so the space that is not currently allocated to any file must be managed. To perform any of the file allocation techniques,it is necessary to know what blocks on the disk are available. Thus we need a disk allocation table in addition to a file allocation table.The following are the approaches used for free space management. 

1. **Bit Tables** : This method uses a vector containing one bit for each block on the disk. Each entry for a 0 corresponds to a free block and each 1 corresponds to a block in use.   
   For example: 00011010111100110001

In this vector every bit correspond to a particular block and 0 implies that, that particular block is free and 1 implies that the block is already occupied. A bit table has the advantage that it is relatively easy to find one or a contiguous group of free blocks. Thus, a bit table works well with any of the file allocation methods. Another advantage is that it is as small as possible.

1. **Free Block List** : In this method, each block is assigned a number sequentially and the list of the numbers of all free blocks is maintained in a reserved block of the disk.



A pipe is a form of redirection (transfer of standard output to some other destination) that is used in Linux and other Unix-like operating systems to send the output of one command/program/process to another command/program/process for further processing. The Unix/Linux systems allow stdout of a command to be connected to stdin of another command. You can make it do so by using the pipe character **‘|’**.

Pipe is used to combine two or more commands, and in this, the output of one command acts as input to another command, and this command’s output may act as input to the next command and so on. It can also be visualized as a temporary connection between two or more commands/ programs/ processes. The command line programs that do the further processing are referred to as filters.

This direct connection between commands/ programs/ processes allows them to operate simultaneously and permits data to be transferred between them continuously rather than having to pass it through temporary text files or through the display screen.   
Pipes are unidirectional **i.e data flows from left to right through the pipeline.**

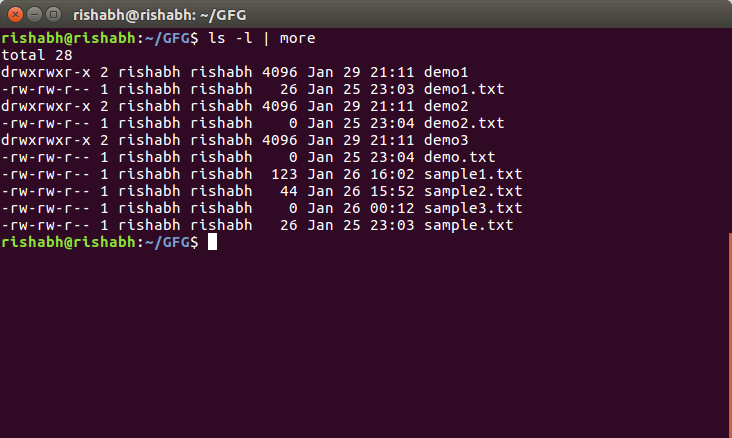
**Syntax :**

command\_1 | command\_2 | command\_3 | .... | command\_N

**Example :**   
**1. Listing all files and directories and give it as input to more command.**

$ ls -l | more

**Output :**



s

# File Permissions

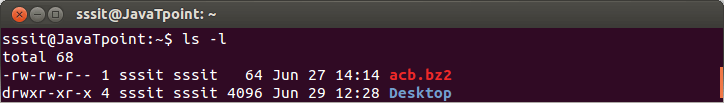
All the three owners (user owner, group, others) in the Linux system have three types of permissions defined. Nine characters denotes the three types of permissions.

1. **Read (r) :** The read permission allows you to open and read the content of a file. But you can't do any editing or modification in the file.
2. **Write (w) :** The write permission allows you to edit, remove or rename a file. For instance, if a file is present in a directory, and write permission is set on the file but not on the directory, then you can edit the content of the file but can't remove, or rename it.
3. **Execute (x):** In Unix type system, you can't run or execute a program unless execute permission is set.But in Windows, there is no such permission available.

**Permissions are listed below:**

|  |  |  |
| --- | --- | --- |
| **permission** | **on a file** | **on a directory** |
| r (read) | read file content (cat) | read directory content (ls) |
| w (write) | change file content (vi) | create file in directory (touch) |
| x (execute) | execute the file | enter the directory (cd) |

## Permission Set



Look at the above snapshot, there are ten characters (-rw-rw-r--) before the user owner. We'll describe these ten characters here.

**File permissions for (-rw-rw-r--)**

46.3M

891

How to find Nth Highest Salary in SQL

|  |  |  |
| --- | --- | --- |
| **position** | **characters** | **ownership** |
| 1 | - | denotes file type |
| 2-4 | rw- | permission for user |
| 5-7 | rw- | permission for group |
| 8-10 | r-- | permission for other |

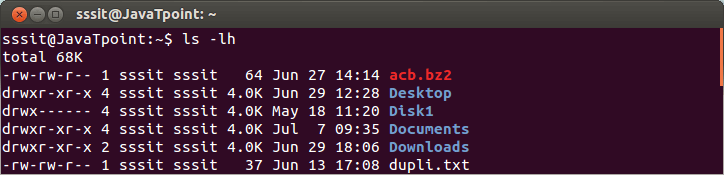
When you are the **User owner**, then the user owner permission applies to you. Other permissions are not relevant to you.

When you are the **Group** then the group permission applies to you. Other permissions are not relevant to you.

When you are the**Other,** then the other permission applies to you. User and group permissions are not relevant to you.

**Permission Example**

Now we'll show some examples how permissions can be seen for a file or directory.



Look at the above snapshot, different directories and files have different permissions.

First letter (-) or **d** represents the files and directories respectively.

Now, from remaining nine letters,**first** triplet represents the permission for **user owner.** Second triplet represents the permission for **group owner. Third** triplet represents the permission for**other** .

## Setting Permissions With chmod

You can change the permissions with chmod command accordingly to your need. Below are some examples to change the permissions for different groups.

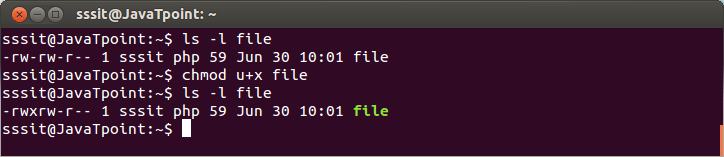
To add permissions to a group.

**Syntax:**

1. chmod **<groupName>**+**<permissionName>** **<fileName>**

**Example:**

1. chmod u+x file



Look at the above snapshot, permission to execute is added to the user owner group.

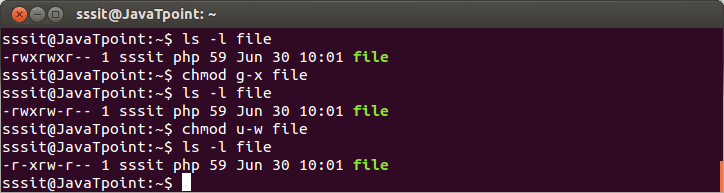
To remove permissions from a group

**Syntax:**

1. chmod **<groupName>**-**<permissionName>** **<fileName>**

**Example:**

1. chmod g-x file
2. chmod u-w file



Look at the above snapshot, permission to execute is removed from the group and permission to write is removed from the user owner.

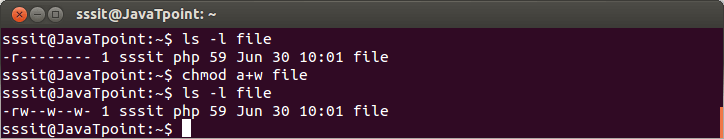
To add permission to all the groups together

**Syntax:**

1. chmod a+**<permissionName>** **<fileName>**

**Example:**

1. chmod a+w file



Look at the above snapshot, we have given permission to write for all the groups.

#### Note: Similarly, you can also remove the permission for all the groups.

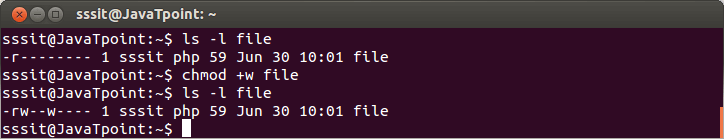
To add permission to all the groups without typing a

**Syntax:**

1. chmod +**<permissionName>** **<fileName>**

**Example:**

1. chmod +w file



Look at the above snapshot, this example is same as the earlier one only difference is that we haven't typed **a** in this.

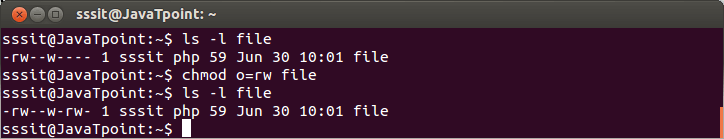
To set explicit permission

**Syntax:**

1. chmod **<groupName>**=**<permissions>** **<fileName>**

**Example:**

1. chmod o=rw file



Look at the above snapshot, we have set explicit permission to read and write for others.

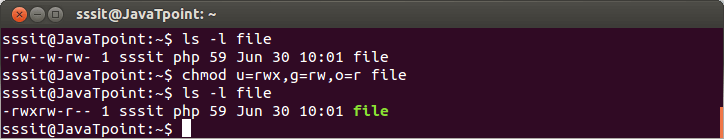
To set explicit permissions for different groups

**Syntax:**

1. chmod **<groupName>**=**<permissions>** **<fileName>**

**Example:**

1. chmod u=rwx,g=rw,o=r file



Look at the above snapshot, we have set permissions for all the three groups.

## Setting Octal Permissions

Octal permissions can also be set for the groups.

For example, to set**r** octal will be**4,** to set**w** octal will be**2,** to set**x** octal will be**1.**

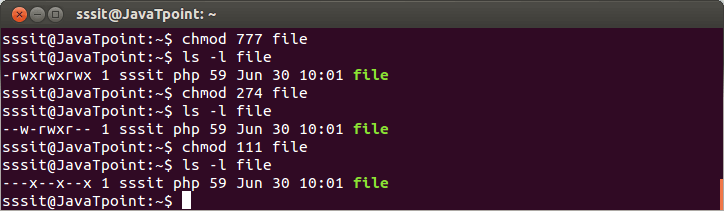
**Octal Table:**

|  |  |  |
| --- | --- | --- |
| **binary** | **octal** | **permissions** |
| 000 | 0 | --- |
| 001 | 1 | --x |
| 010 | 2 | -w- |
| 011 | 3 | -wx |
| 100 | 4 | r-- |
| 101 | 5 | r-x |
| 110 | 6 | rw- |
| 111 | 7 | rwx |

From this we can conclude that,

1. 777 = rwxrwxrwx
2. 765 = rwxrw-r-x
3. 654 = rw-r-xr--

and so on.



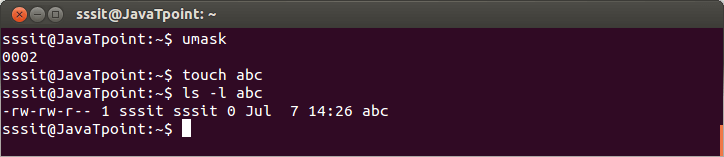
Look at the above snapshot, we have shown some random octal examples with the numbers **777, 274** and **111.**

## umask

While creating a file or directory, by default a set of permissions are applied. These default permissions are viewed by **umask** command.

For safety reasons all Unix systems doesn't provide execution permission to newly created files.

Adding execution permission is upto you.



**mkdir -m**

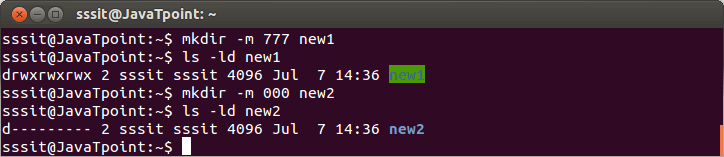
The 'mkdir -m' command can be used to set the mode.

**Syntax:**

1. mkdir -m **<mode>** **<fileName>**

**Example:**

1. mkdir -m 777 new1
2. mkdir -m 000 new2



Look at the above snapshot, we have created two files**new1**and **new2**with mode**777** and**000**respectively.

**cp -p**

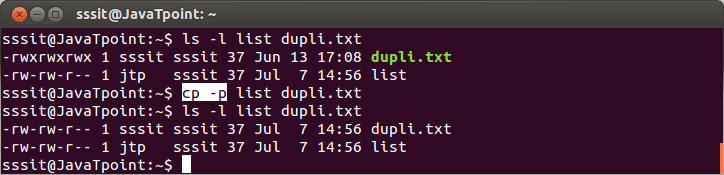
The 'cp -p' command preserves the permissions and time stamps from source files.

**Syntax:**

1. cp -p **<sourceFile>** **<destinationFile>**

**Example:**

1. cp -p list dupli.txt



Look at the above snapshot, earlier permissions for files list and dupli.txt were different. But after passing the command **"cp -p list dupli.txt"**, both the files have same permissions.

* Permissions (chmod, chown, etc); access control list; network commands (telenet, ftp, ssh,  
  sftp, finger.

# Linux/Unix SSH, Ping, FTP, Telnet Communication Commands

By[Mary Brent](https://www.guru99.com/author/mary)UpdatedDecember 31, 2022

While working on a Linux operating system, you may need to **communicate with other devices**. For this, there are some basic utilities that you can make use of.

These utilities can help you communicate with:

* networks,
* other Linux systems
* and remote users

So, let us learn them one by one.

* [SSH](https://www.guru99.com/communication-in-linux.html#1)
* [Ping](https://www.guru99.com/communication-in-linux.html#2)
* [FTP](https://www.guru99.com/communication-in-linux.html#3)
* [Telnet](https://www.guru99.com/communication-in-linux.html#4)

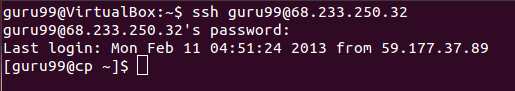
Click [here](https://www.guru99.com/faq#faq1) if the video is not accessible

## SSH

SSH which stands for Secure Shell, It is used to connect to a remote computer securely. Compare to Telnet, SSH is secure wherein the client /server connection is authenticated using a digital certificate and passwords are encrypted. Hence it’s widely used by system administrators to control remote Linux servers.

The syntax to log into a remote Linux machine using SSH is

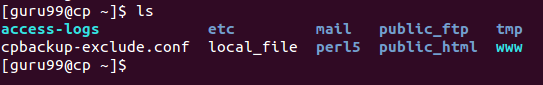
SSH username@ip-address or hostname



Once you are logged in, you can execute any commands that you do in your terminal

**Example:**

ls



**Example:**

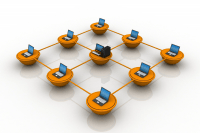
pwd

Linux/Unix SSH, Ping, FTP, Telnet Communication Commands

## Ping

This utility is commonly used to check whether your **connection to the server** is healthy or not.This command is also used in –

* Analyzing network and host connections
* Tracking network performance and managing it
* Testing hardware and software issues

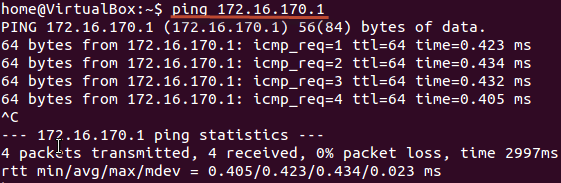


Command Syntax:-

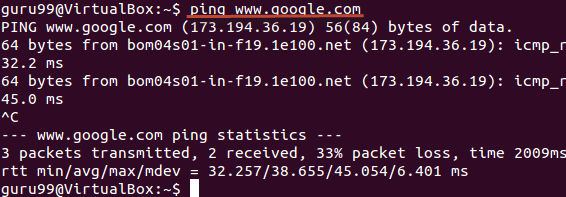
ping hostname="" or=""

Example :

ping 172.16.170.1

****

ping google.com



Here, A system has sent 64 bytes data packets to the IP Address (172.16.170.1) or the Hostname(www.google.com). If even one of data packets does not return or is lost, it would suggest an error in the connection. Usually, internet connectivity is checked using this method.

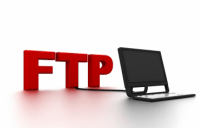
You may Press **Ctrl + c** to **exit** from the ping loop.

## FTP

FTP **is file transfer protocol**. It’s the **most preferred protocol for** **data transfer** amongst computers.

You can use FTP to –

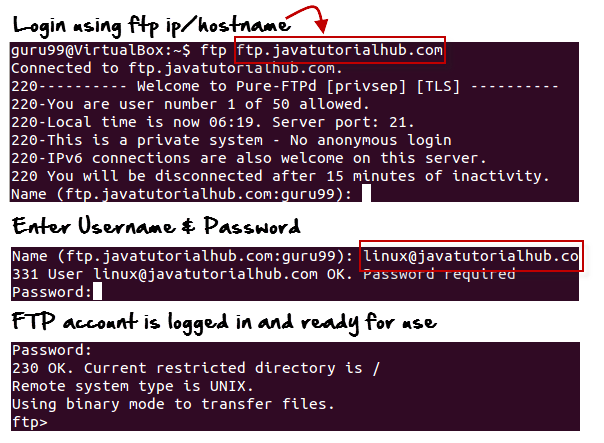
* Logging in and establishing a connection with a remote host
* Upload and download files
* Navigating through directories
* Browsing contents of the directories



The syntax to establish an FTP connection to a remote host is –

ftp hostname="" or=""

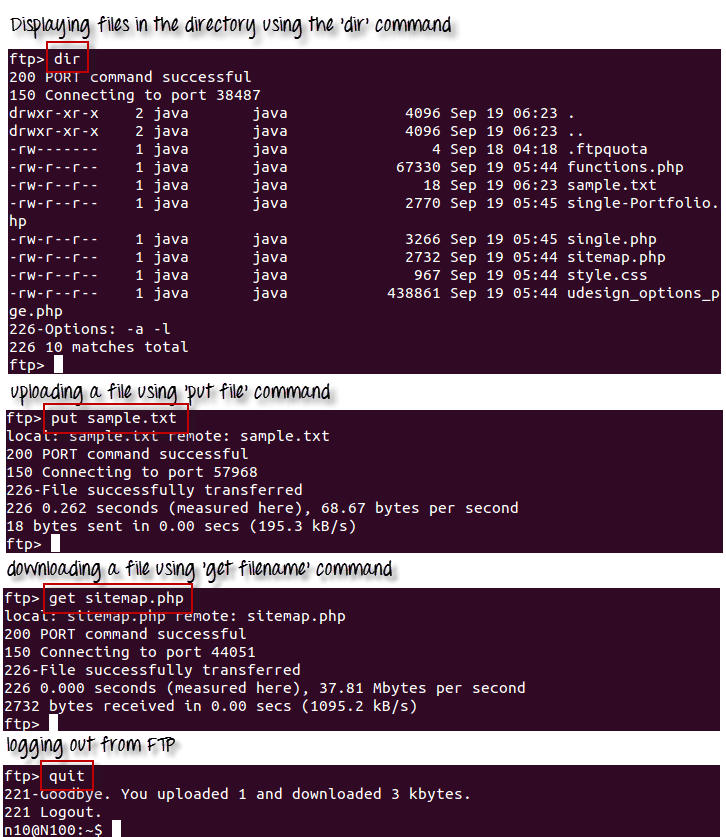
Once you enter this command, it will ask you for **authentication** via username and password.



Once a connection is established, and you are logged in, you may use the following commands to perform different actions.

|  |  |
| --- | --- |
| **Command** | **Function** |
| dir | Display files in the current directory of a remote computer |
| cd “dirname” | change directory to “dirname” on a remote computer |
| put file | upload ‘file’ from local to remote computer |
| get file | Download ‘file’ from remote to local computer |
| quit | Logout |

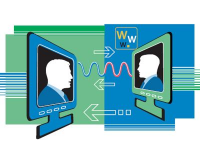
Let us run some of the important commands.



## Telnet

Telnet helps to –

* connect to a remote Linux computer
* run programs remotely and conduct administration



This utility is similar to the Remote Desktop feature found in Windows Machine.

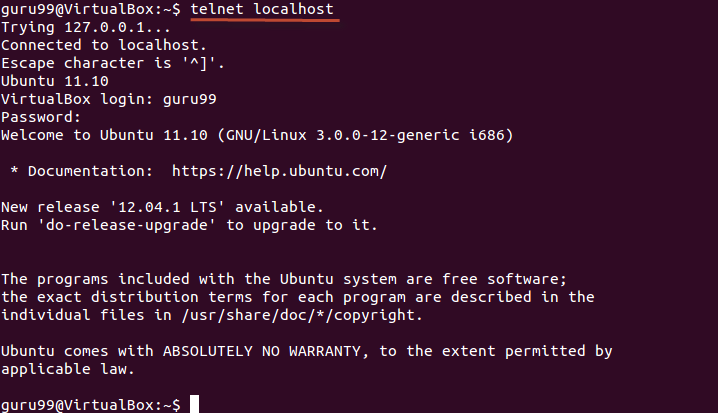
The syntax for this utility is:

telnet hostname="" or=""

Example:

telnet localhost

For demonstration purpose, we will connect to your computer (localhost). The utility will ask your username and password.

****

Once authenticated, you can execute commands just like you have done so far, using the Terminal. The only difference is, if you are connected to a remote host, the commands will be executed on the remote machine, and not your local machine.

You may exit the telnet connection by entering the command ‘logout’

## System variables like – PS1, PS2 etc. How to set them Shell Programming • What is shell; What are different shells in Linux? • Shell variables; Wildcard symbols • Shell meta characters; Command line arguments; Read, Echo

## Summary:

* Communication between Linux/UNIX and other different computers, networks and remote users is possible.
* The ping command checks whether the connection with a hostname or IP-address is working or not. Run ‘ping IP address or Hostname’ on the terminal
* FTP is preferred protocol for sending and receiving large files. You can establish an FTP connection to a remote host and then use commands for uploading, downloading files, checking file and browsing them
* Telnet utility helps you to connect to a remote Linux computer and work on it

In this chapter, we will discuss in detail about the Unix environment. An important Unix concept is the **environment**, which is defined by environment variables. Some are set by the system, others by you, yet others by the shell, or any program that loads another program.

A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.

For example, first we set a variable TEST and then we access its value using the **echo** command −

$TEST="Unix Programming"

$echo $TEST

It produces the following result.

Unix Programming

Note that the environment variables are set without using the **$** sign but while accessing them we use the $ sign as prefix. These variables retain their values until we come out of the shell.

When you log in to the system, the shell undergoes a phase called **initialization** to set up the environment. This is usually a two-step process that involves the shell reading the following files −

* /etc/profile
* profile

The process is as follows −

* The shell checks to see whether the file **/etc/profile** exists.
* If it exists, the shell reads it. Otherwise, this file is skipped. No error message is displayed.
* The shell checks to see whether the file **.profile** exists in your home directory. Your home directory is the directory that you start out in after you log in.
* If it exists, the shell reads it; otherwise, the shell skips it. No error message is displayed.

As soon as both of these files have been read, the shell displays a prompt −

$

This is the prompt where you can enter commands in order to have them executed.

**Note** − The shell initialization process detailed here applies to all **Bourne** type shells, but some additional files are used by **bash** and **ksh**.

## The .profile File

The file **/etc/profile** is maintained by the system administrator of your Unix machine and contains shell initialization information required by all users on a system.

The file **.profile** is under your control. You can add as much shell customization information as you want to this file. The minimum set of information that you need to configure includes −

* The type of terminal you are using.
* A list of directories in which to locate the commands.
* A list of variables affecting the look and feel of your terminal.

You can check your **.profile** available in your home directory. Open it using the vi editor and check all the variables set for your environment.

## Setting the Terminal Type

Usually, the type of terminal you are using is automatically configured by either the **login** or **getty** programs. Sometimes, the auto configuration process guesses your terminal incorrectly.

If your terminal is set incorrectly, the output of the commands might look strange, or you might not be able to interact with the shell properly.

To make sure that this is not the case, most users set their terminal to the lowest common denominator in the following way −

$TERM=vt100

$

## Setting the PATH

When you type any command on the command prompt, the shell has to locate the command before it can be executed.

The PATH variable specifies the locations in which the shell should look for commands. Usually the Path variable is set as follows −

$PATH=/bin:/usr/bin

$

Here, each of the individual entries separated by the colon character **(:)** are directories. If you request the shell to execute a command and it cannot find it in any of the directories given in the PATH variable, a message similar to the following appears −

$hello

hello: not found

$

There are variables like PS1 and PS2 which are discussed in the next section.

## PS1 and PS2 Variables

The characters that the shell displays as your command prompt are stored in the variable PS1. You can change this variable to be anything you want. As soon as you change it, it'll be used by the shell from that point on.

For example, if you issued the command −

$PS1='=>'

=>

=>

=>

Your prompt will become =>. To set the value of **PS1** so that it shows the working directory, issue the command −

=>PS1="[\u@\h \w]\$"

[root@ip-72-167-112-17 /var/www/tutorialspoint/unix]$

[root@ip-72-167-112-17 /var/www/tutorialspoint/unix]$

The result of this command is that the prompt displays the user's username, the machine's name (hostname), and the working directory.

There are quite a few **escape sequences** that can be used as value arguments for PS1; try to limit yourself to the most critical so that the prompt does not overwhelm you with information.

|  |  |
| --- | --- |
| **Sr.No.** | **Escape Sequence & Description** |
| 1 | **\t**  Current time, expressed as HH:MM:SS |
| 2 | **\d**  Current date, expressed as Weekday Month Date |
| 3 | **\n**  Newline |
| 4 | **\s**  Current shell environment |
| 5 | **\W**  Working directory |
| 6 | **\w**  Full path of the working directory |
| 7 | **\u**  Current user’s username |
| 8 | **\h**  Hostname of the current machine |
| 9 | **\#**  Command number of the current command. Increases when a new command is entered |
| 10 | **\$**  If the effective UID is 0 (that is, if you are logged in as root), end the prompt with the # character; otherwise, use the $ sign |

You can make the change yourself every time you log in, or you can have the change made automatically in PS1 by adding it to your **.profile** file.

When you issue a command that is incomplete, the shell will display a secondary prompt and wait for you to complete the command and hit **Enter** again.

The default secondary prompt is **>** (the greater than sign), but can be changed by re-defining the **PS2** shell variable −

Following is the example which uses the default secondary prompt −

$ echo "this is a

> test"

this is a

test

$

The example given below re-defines PS2 with a customized prompt −

$ PS2="secondary prompt->"

$ echo "this is a

secondary prompt->test"

this is a

test

$

## Environment Variables

Following is the partial list of important environment variables. These variables are set and accessed as mentioned below −

|  |  |
| --- | --- |
| **Sr.No.** | **Variable & Description** |
| 1 | **DISPLAY**  Contains the identifier for the display that **X11** programs should use by default. |
| 2 | **HOME**  Indicates the home directory of the current user: the default argument for the cd **built-in** command. |
| 3 | **IFS**  Indicates the **Internal Field Separator** that is used by the parser for word splitting after expansion. |
| 4 | **LANG**  LANG expands to the default system locale; LC\_ALL can be used to override this. For example, if its value is **pt\_BR**, then the language is set to (Brazilian) Portuguese and the locale to Brazil. |
| 5 | **LD\_LIBRARY\_PATH**  A Unix system with a dynamic linker, contains a colonseparated list of directories that the dynamic linker should search for shared objects when building a process image after exec, before searching in any other directories. |
| 6 | **PATH**  Indicates the search path for commands. It is a colon-separated list of directories in which the shell looks for commands. |
| 7 | **PWD**  Indicates the current working directory as set by the cd command. |
| 8 | **RANDOM**  Generates a random integer between 0 and 32,767 each time it is referenced. |
| 9 | **SHLVL**  Increments by one each time an instance of bash is started. This variable is useful for determining whether the built-in exit command ends the current session. |
| 10 | **TERM**  Refers to the display type. |
| 11 | **TZ**  Refers to Time zone. It can take values like GMT, AST, etc. |
| 12 | **UID**  Expands to the numeric user ID of the current user, initialized at the shell startup. |

Following is the sample example showing few environment variables −

$ echo $HOME

/root

]$ echo $DISPLAY

$ echo $TERM

xterm

$ echo $PATH

/usr/local/bin:/bin:/usr/bin:/home/amrood/bin:/usr/local/bin

$

Mcq link bin bash:

<https://www.tutorialspoint.com/unix/unix_questions_answers.htm>

Session 3:

Shell Programming  
• Decision loops (if else, test, nested if else, case controls, while...until, for)

# Conditional Statements | Shell Script

* **Difficulty Level :** [Easy](https://www.geeksforgeeks.org/easy/)
* **Last Updated :** 27 Feb, 2020

 Read

 Discuss

**Conditional Statements:** There are total 5 conditional statements which can be used in bash programming

1. if statement
2. if-else statement
3. if..elif..else..fi statement (Else If ladder)
4. if..then..else..if..then..fi..fi..(Nested if)
5. switch statement

Their description with syntax is as follows:

**if statement**  
This block will process if specified condition is true.  
***Syntax:***

if [ expression ]

then

statement

fi

**if-else statement**  
If specified condition is not true in if part then else part will be execute.  
***Syntax***

if [ expression ]

then

statement1

else

statement2

fi

**if..elif..else..fi statement (Else If ladder)**  
To use multiple conditions in one if-else block, then elif keyword is used in shell. If expression1 is true then it executes statement 1 and 2, and this process continues. If none of the condition is true then it processes else part.  
***Syntax***

if [ expression1 ]

then

statement1

statement2

.

.

elif [ expression2 ]

then

statement3

statement4

.

.

else

statement5

fi

**if..then..else..if..then..fi..fi..(Nested if)**  
Nested if-else block can be used when, one condition is satisfies then it again checks another condition. In the syntax, if expression1 is false then it processes else part, and again expression2 will be check.  
***Syntax:***

if [ expression1 ]

then

statement1

statement2

.

else

if [ expression2 ]

then

statement3

.

fi

fi

**switch statement**  
case statement works as a switch statement if specified value match with the pattern then it will execute a block of that particular pattern  
When a match is found all of the associated statements until the double semicolon (;;) is executed.  
A case will be terminated when the last command is executed.  
If there is no match, the exit status of the case is zero.

***Syntax:***

case in

Pattern 1) Statement 1;;

Pattern n) Statement n;;

esac

**Example Programs**

**Example 1:**  
Implementing if statement

|  |
| --- |
| #Initializing two variables  a=10  b=20    #Check whether they are equal  if [ $a == $b ]  then      echo "a is equal to b"  fi    #Check whether they are not equal  if [ $a != $b ]  then      echo "a is not equal to b"  fi |

**Output**

$bash -f main.sh

a is not equal to b

***Example 2:***  
Implementing if.else statement

|  |
| --- |
| #Initializing two variables  a=20  b=20    if [ $a == $b ]  then      #If they are equal then print this      echo "a is equal to b"  else      #else print this      echo "a is not equal to b"  fi |

**Output**

$bash -f main.sh

a is equal to b

**Example 3:**  
Implementing switch statement

|  |
| --- |
| CARS="bmw"    #Pass the variable in string  case "$CARS" in      #case 1      "mercedes") echo "Headquarters - Affalterbach, Germany" ;;        #case 2      "audi") echo "Headquarters - Ingolstadt, Germany" ;;        #case 3      "bmw") echo "Headquarters - Chennai, Tamil Nadu, India" ;;  esac |

**Output**

$bash -f main.sh

Headquarters - Chennai, Tamil Nadu, India.

**Note:** Shell scripting is a case-sensitive language, which means proper syntax has to be followed while writing the scripts.

**Looping Statements in Shell Scripting:** There are total 3 looping statements which can be used in bash programming 

1. while statement
2. for statement
3. until statement

To alter the flow of loop statements, two commands are used they are, 

1. break
2. continue

Their descriptions and syntax are as follows: 

* **while statement**   
  Here command is evaluated and based on the result loop will executed, if command raise to false then loop will be terminated   
  ***Syntax***

* **for statement**   
  The for loop operate on lists of items. It repeats a set of commands for every item in a list.   
  Here var is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.   
  ***Syntax***

* **until statement**   
  The until loop is executed as many as times the condition/command evaluates to false. The loop terminates when the condition/command becomes true.

Regular expression in shell Programming

The following arithmetic operators are supported by Bourne Shell.

Assume variable **a** holds 10 and variable **b** holds 20 then −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + (Addition) | Adds values on either side of the operator | `expr $a + $b` will give 30 |
| - (Subtraction) | Subtracts right hand operand from left hand operand | `expr $a - $b` will give -10 |
| \* (Multiplication) | Multiplies values on either side of the operator | `expr $a \\* $b` will give 200 |
| / (Division) | Divides left hand operand by right hand operand | `expr $b / $a` will give 2 |
| % (Modulus) | Divides left hand operand by right hand operand and returns remainder | `expr $b % $a` will give 0 |
| = (Assignment) | Assigns right operand in left operand | a = $b would assign value of b into a |
| == (Equality) | Compares two numbers, if both are same then returns true. | [ $a == $b ] would return false. |
| != (Not Equality) | Compares two numbers, if both are different then returns true. | [ $a != $b ] would return true. |

It is very important to understand that all the conditional expressions should be inside square braces with spaces around them, for example **[ $a == $b ]** is correct whereas, **[$a==$b]** is incorrect.

All the arithmetical calculations are done using long integers.

## Example

Here is an example which uses all the arithmetic operators −

[Live Demo](http://tpcg.io/IgpuvS)

#!/bin/sh

a=10

b=20

val=`expr $a + $b`

echo "a + b : $val"

val=`expr $a - $b`

echo "a - b : $val"

val=`expr $a \\* $b`

echo "a \* b : $val"

val=`expr $b / $a`

echo "b / a : $val"

val=`expr $b % $a`

echo "b % a : $val"

if [ $a == $b ]

then

echo "a is equal to b"

fi

if [ $a != $b ]

then

echo "a is not equal to b"

fi

The above script will produce the following result −

a + b : 30

a - b : -10

a \* b : 200

b / a : 2

b % a : 0

a is not equal to b

The following points need to be considered when using the Arithmetic Operators −

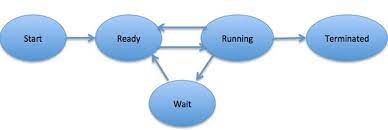
* There must be spaces between the operators and the expressions. For example, 2+2 is not correct; it should be written as 2 + 2.
* Complete expression should be enclosed between **‘ ‘**, called the inverted commas.
* You should use **\** on the **\*** symbol for multiplication.
* **if...then...fi** statement is a decision-making statement which has been explained in the next chapter

.

Session 4 ,5,6:

What is process; preemptive and non-preemptive processes;

A process is **a running program that serves as the foundation for all computation**. The procedure is not the same as computer code, although it is very similar. In contrast to the program, which is often regarded as some 'passive' entity, a process is an 'active' entity.



# Preemptive and Non-Preemptive Scheduling

* **Difficulty Level :** [Easy](https://www.geeksforgeeks.org/easy/)
* **Last Updated :** 27 May, 2022

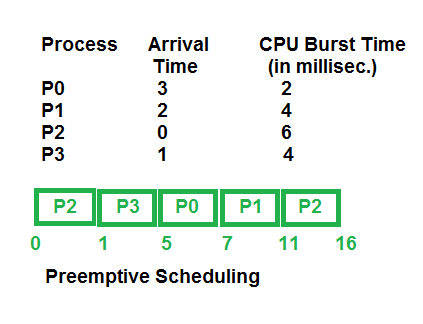
 Read

 Discuss

Prerequisite – [CPU Scheduling](https://www.geeksforgeeks.org/gate-notes-operating-system-process-scheduling/)

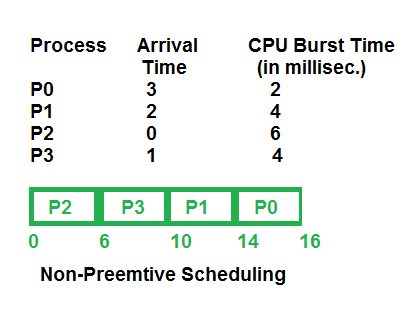
**1. Preemptive Scheduling:**   
Preemptive scheduling is used when a process switches from running state to ready state or from the waiting state to ready state. The resources (mainly CPU cycles) are allocated to the process for a limited amount of time and then taken away, and the process is again placed back in the ready queue if that process still has CPU burst time remaining. That process stays in the ready queue till it gets its next chance to execute.

Algorithms based on preemptive scheduling are: [Round Robin (RR)](https://www.geeksforgeeks.org/program-round-robin-scheduling-set-1/),[Shortest Remaining Time First (SRTF)](https://www.geeksforgeeks.org/program-shortest-job-first-scheduling-set-2srtf-make-changesdoneplease-review/), [Priority (preemptive version)](https://www.geeksforgeeks.org/program-for-preemptive-priority-cpu-scheduling/), etc.



**2. Non-Preemptive Scheduling:**   
Non-preemptive Scheduling is used when a process terminates, or a process switches from running to the waiting state. In this scheduling, once the resources (CPU cycles) are allocated to a process, the process holds the CPU till it gets terminated or reaches a waiting state. In the case of non-preemptive scheduling does not interrupt a process running CPU in the middle of the execution. Instead, it waits till the process completes its CPU burst time, and then it can allocate the CPU to another process.

Algorithms based on non-preemptive scheduling are:[Shortest Job First (SJF basically non preemptive)](https://www.geeksforgeeks.org/program-shortest-job-first-sjf-scheduling-set-1-non-preemptive/) and [Priority (non preemptive version)](https://www.geeksforgeeks.org/operating-system-priority-scheduling-different-arrival-time-set-2/), etc.



**Key Differences Between Preemptive and Non-Preemptive Scheduling:**

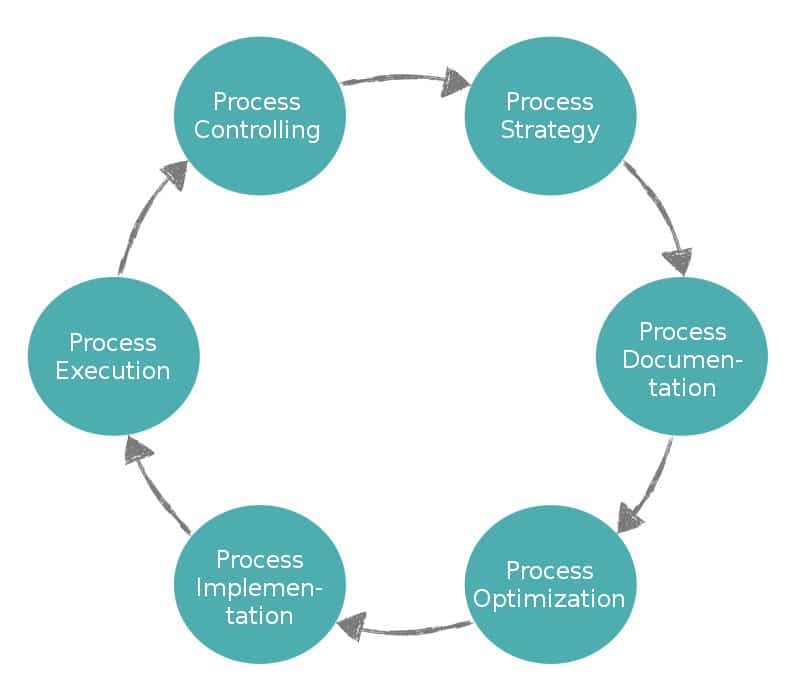
1. In preemptive scheduling, the CPU is allocated to the processes for a limited time whereas, in Non-preemptive scheduling, the CPU is allocated to the process till it terminates or switches to the waiting state.
2. The executing process in preemptive scheduling is interrupted in the middle of execution when higher priority one comes whereas, the executing process in non-preemptive scheduling is not interrupted in the middle of execution and waits till its execution.
3. In Preemptive Scheduling, there is the overhead of switching the process from the ready state to running state, vise-verse and maintaining the ready queue. Whereas in the case of non-preemptive scheduling has no overhead of switching the process from running state to ready state.
4. In preemptive scheduling, if a high-priority process frequently arrives in the ready queue then the process with low priority has to wait for a long, and it may have to starve. , in the non-preemptive scheduling, if CPU is allocated to the process having a larger burst time then the processes with small burst time may have to starve.
5. Preemptive scheduling attains flexibility by allowing the critical processes to access the CPU as they arrive into the ready queue, no matter what process is executing currently. Non-preemptive scheduling is called rigid as even if a critical process enters the ready queue the process running CPU is not disturbed.
6. Preemptive Scheduling has to maintain the integrity of shared data that’s why it is cost associative which is not the case with Non-preemptive Scheduling.

**Comparison Chart:**

| Parameter | PREEMPTIVE SCHEDULING | NON-PREEMPTIVE SCHEDULING |
| --- | --- | --- |
| Basic | In this resources(CPU Cycle) are allocated to a process for a limited time. | Once resources(CPU Cycle) are allocated to a process, the process holds it till it completes its burst time or switches to waiting state. |
| Interrupt | Process can be interrupted in between. | Process can not be interrupted until it terminates itself or its time is up. |
| Starvation | If a process having high priority frequently arrives in the ready queue, a low priority process may starve. | If a process with a long burst time is running CPU, then later coming process with less CPU burst time may starve. |
| Overhead | It has overheads of scheduling the processes. | It does not have overheads. |
| Flexibility | flexible | rigid |
| Cost | cost associated | no cost associated |
| CPU Utilization | In preemptive scheduling, CPU utilization is high. | It is low in non preemptive scheduling. |
| Waiting Time | Preemptive scheduling waiting time is less. | Non-preemptive scheduling waiting time is high. |
| Response Time | Preemptive scheduling response time is less. | Non-preemptive scheduling response time is high. |
| Examples | Examples of preemptive scheduling are Round Robin and Shortest Remaining Time First. | Examples of non-preemptive scheduling are First Come First Serve and Shortest Job First. |

# Process Management Lifecycle

The **process management lifecycle** is a model for the continuous implementation and improvement of processes and business process management in a company. The lifecycle consists of the following six phases:



1. **Process strategy** – Develop a plan for process improvement.
2. **Process documentation** – Document the processes currently in use.
3. **Process optimization** – Optimize processes based on the defined process strategy.
4. **Process implementation** – Introduce the optimized processes into the existing process landscape.
5. **Process execution** – Carry out the new processes and record their performance.
6. **Process controlling** – Use the recorded performance data and key figures to check whether guideline values are being achieved or whether there are bottlenecks and  potential for further improvement.

## Why is the process management life cycle important?

The process management lifecycle represents a structured procedure for the management and improvement of processes, making process management easier overall. It ensures the right people know what has to be done in each phase and what’s coming next. In addition, the cycle is helpful for a company just starting to use strategic, systematic [process management](https://appian.com/bpm-basics.html), as it clearly lays out all the necessary steps.

For companies already practicing process management, the process management lifecycle makes it easy to check which phase they are in, what has to be considered, and whether all relevant information and persons have been included. This step-by-step approach guides companies, helping them ask the right questions to improve their processes for maximum business results.

### Types of Schedulers

There are three types of schedulers available which are as follows −

* Long Term Scheduler
* Short Term Scheduler
* Medium Term Scheduler

## Differences

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

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

Process scheduling algorithms – FCFS, Shortest Job First, Priority, RR, Queue. Belady’s Anomaly.

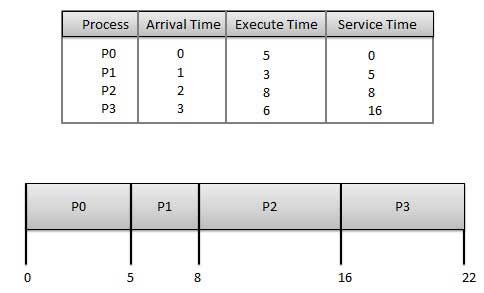
A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. There are six popular process scheduling algorithms which we are going to discuss in this chapter −

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

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

## First Come First Serve (FCFS)

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



**Wait time** of each process is as follows −

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

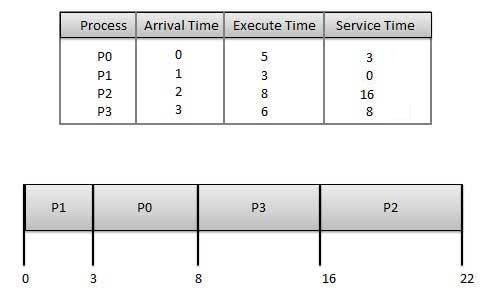
Average Wait Time: (0+4+6+13) / 4 = 5.75

## Shortest Job Next (SJN)

* This is also known as **shortest job first**, or SJF
* This is a non-preemptive, pre-emptive scheduling algorithm.
* Best approach to minimize waiting time.
* Easy to implement in Batch systems where required CPU time is known in advance.
* Impossible to implement in interactive systems where required CPU time is not known.
* The processer should know in advance how much time process will take.

Given: Table of processes, and their Arrival time, Execution time

|  |  |  |  |
| --- | --- | --- | --- |
| **Process** | **Arrival Time** | **Execution Time** | **Service Time** |
| P0 | 0 | 5 | 0 |
| P1 | 1 | 3 | 5 |
| P2 | 2 | 8 | 14 |
| P3 | 3 | 6 | 8 |



**Waiting time** of each process is as follows −

|  |  |
| --- | --- |
| **Process** | **Waiting Time** |
| P0 | 0 - 0 = 0 |
| P1 | 5 - 1 = 4 |
| P2 | 14 - 2 = 12 |
| P3 | 8 - 3 = 5 |

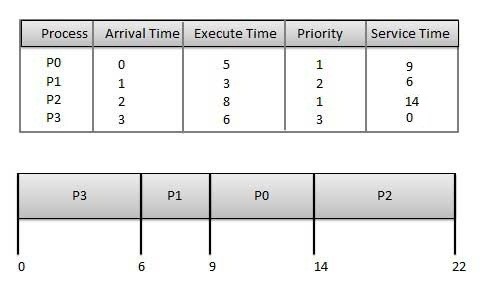
Average Wait Time: (0 + 4 + 12 + 5)/4 = 21 / 4 = 5.25

## Priority Based Scheduling

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

Given: Table of processes, and their Arrival time, Execution time, and priority. Here we are considering 1 is the lowest priority.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Process** | **Arrival Time** | **Execution Time** | **Priority** | **Service Time** |
| P0 | 0 | 5 | 1 | 0 |
| P1 | 1 | 3 | 2 | 11 |
| P2 | 2 | 8 | 1 | 14 |
| P3 | 3 | 6 | 3 | 5 |



**Waiting time** of each process is as follows −

|  |  |
| --- | --- |
| **Process** | **Waiting Time** |
| P0 | 0 - 0 = 0 |
| P1 | 11 - 1 = 10 |
| P2 | 14 - 2 = 12 |
| P3 | 5 - 3 = 2 |

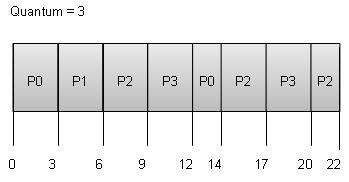
Average Wait Time: (0 + 10 + 12 + 2)/4 = 24 / 4 = 6

## Shortest Remaining Time

* Shortest remaining time (SRT) is the preemptive version of the SJN algorithm.
* The processor is allocated to the job closest to completion but it can be preempted by a newer ready job with shorter time to completion.
* Impossible to implement in interactive systems where required CPU time is not known.
* It is often used in batch environments where short jobs need to give preference.

## Round Robin Scheduling

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



**Wait time** of each process is as follows −

|  |  |
| --- | --- |
| **Process** | **Wait Time : Service Time - Arrival Time** |
| P0 | (0 - 0) + (12 - 3) = 9 |
| P1 | (3 - 1) = 2 |
| P2 | (6 - 2) + (14 - 9) + (20 - 17) = 12 |
| P3 | (9 - 3) + (17 - 12) = 11 |

Average Wait Time: (9+2+12+11) / 4 = 8.5

## Multiple-Level Queues Scheduling

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

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

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

# Difference between fork() and exec()

* **Difficulty Level :** [Medium](https://www.geeksforgeeks.org/medium/)
* **Last Updated :** 08 Nov, 2022

 Read

 Discuss

 Courses

 Practice

 Video

Every application(program) comes into execution through means of process, **process** is a running instance of a program. Processes are created through different system calls, most popular are **fork()** and **exec()**

[**fork()**](https://www.geeksforgeeks.org/fork-system-call/)

pid\_t pid = fork();

fork() creates a new process by duplicating the calling process, The new process, referred to as child, is an exact duplicate of the calling process, referred to as parent, except for the following :

1. The child has its own unique process ID, and this PID does not match the ID of any existing process group.
2. The child’s parent process ID is the same as the parent’s process ID.
3. The child does not inherit its parent’s memory locks and [semaphore](https://www.geeksforgeeks.org/semaphores-and-its-types/) adjustments.
4. The child does not inherit outstanding asynchronous I/O operations from its parent nor does it inherit any asynchronous I/O contexts from its parent.

**Return value of fork()** On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.

**exec()**

The exec() family of functions **replaces** the current process image with a new process image. It loads the program into the current process space and runs it from the entry point. The [exec() family](https://www.geeksforgeeks.org/exec-family-of-functions-in-c/) consists of following functions, I have implemented **execv()** in following C program, you can try rest as an exercise

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

int execlp(const char \*file, const char \*arg, ...);

int execle(const char \*path, const char \*arg, ...,

char \* const envp[]);

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

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

int execvpe(const char \*file, char \*const argv[],

char \*const envp[]);

**fork vs exec**

* fork starts a new process which is a copy of the one that calls it, while exec replaces the current process image with another (different) one.
* Both parent and child processes are executed simultaneously in case of fork() while Control never returns to the original program unless there is an exec() error.
* C

|  |
| --- |
| // C program to illustrate use of fork() &  // exec() system call for process creation    #include <stdio.h>  #include <sys/types.h>  #include <unistd.h>  #include <stdlib.h>  #include <errno.h>  #include <sys/wait.h>    int main(){  pid\_t pid;  int ret = 1;  int status;  pid = fork();    if (pid == -1){        // pid == -1 means error occurred      printf("can't fork, error occurred\n");      exit(EXIT\_FAILURE);  }  else if (pid == 0){        // pid == 0 means child process created      // getpid() returns process id of calling process      // Here It will return process id of child process      printf("child process, pid = %u\n",getpid());      // Here It will return Parent of child Process means Parent process it self      printf("parent of child process, pid = %u\n",getppid());        // the argv list first argument should point to      // filename associated with file being executed      // the array pointer must be terminated by NULL      // pointer      char \* argv\_list[] = {"ls","-lart","/home",NULL};        // the execv() only return if error occurred.      // The return value is -1      execv("ls",argv\_list);      exit(0);  }  else{      // a positive number is returned for the pid of      // parent process      // getppid() returns process id of parent of      // calling process  // Here It will return parent of parent process's ID      printf("Parent Of parent process, pid = %u\n",getppid());      printf("parent process, pid = %u\n",getpid());            // the parent process calls waitpid() on the child          // waitpid() system call suspends execution of          // calling process until a child specified by pid          // argument has changed state          // see wait() man page for all the flags or options          // used here          if (waitpid(pid, &status, 0) > 0) {                if (WIFEXITED(status) && !WEXITSTATUS(status))              printf("program execution successful\n");                else if (WIFEXITED(status) && WEXITSTATUS(status)) {                  if (WEXITSTATUS(status) == 127) {                        // execv failed                      printf("execv failed\n");                  }                  else                      printf("program terminated normally,"                      " but returned a non-zero status\n");              }              else              printf("program didn't terminate normally\n");          }          else {          // waitpid() failed          printf("waitpid() failed\n");          }      exit(0);  }  return 0;  } |

**Output:**

parent process, pid = 11523

child process, pid = 14188

Program execution successful

**Let us see the differences in a tabular form -:**

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

This article is contributed by **Mandeep Singh**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [write.geeksforgeeks.org](http://write.geeksforgeeks.org/) or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

### 1.0 fork and exec system calls

Suppose we wish to write a "shell program" which would execute another program. Now, in a computing system, a [process](https://www.softprayog.in/programming/program-process-threads) executes a program. So this shell program (or, process, at run time) needs to create a process which would execute a program. Here, two system calls are of interest, fork and exec.

### 1.1 fork system call

#include <sys/types.h>

#include <unistd.h>

pid\_t fork (void);

When a process makes the fork system call, a new process is created which is a clone of the calling process. The code, data and the stack of the new process is copied from the calling process. The newly created process is called the child process, whereas the calling process is termed the parent process. However, there is a difference between the parent and child processes. The return value of fork in the child process is 0, whereas, in the parent process, the process id of the child process is returned. Indeed, the two processes use this difference to figure out whether they are the parent or child.

What is the use of having a process which is a copy of its parent? Not much. But, then, the fork system call is mostly used in conjunction with a variation of exec. In Linux, there is an execve system call and there are six functions with names starting with exec and are front-ends to the execve system call. When we say exec in context of Linux, we mean either the execve system call or one of the six functions described later in this tutorial.

### 1.2 execve system call

The execve system call (execve(2)) is the starting point of our discussion on exec.

#include <unistd.h>

int execve (const char \*filename, char \*const argv [],

char \*const envp []);

What happens when a process makes the execve system call? Its code and data segments are initialized by the program contained in the file identified by filename. The most important thing to note is that it is the same process (its pid is the same as before), but is executing a new program. Incidentally, execve is one example where the difference between a program and a process cannot be emphasized enough. A process is an entity created by the kernel to execute a program written to do some tasks. argv is an array of arguments to the program, where the zeroth element in the array is the file name of the program itself. envp is an array of environment variables in the format, name = value. The last element in both argv and envp must be NULL. Both argv and envp can be accessed in the main function of the program, which is called as,

int main (int argc, char \*argv [], char \*envp [])

However, the third parameter, envp is not specified in POSIX.1, which stipulates that the environment variables should be accessed via the external variable environ, (environ (7)).

execve does not return on success. It can't, for the code segment has been initialized from the new program being executed and the return address (in the previous) program is lost for ever. However, if execve is unsuccessful, -1 is returned, and errno is set accordingly.

### 1.3 exec family of functions

There is an exec family of six functions (exec(3)), which provide front-ends to the execve system call. These are,

#include <unistd.h>

extern char \*\*environ;

int execl (const char \*path, const char \*arg0, const char \*arg1, ..., (char \*) NULL);

int execlp (const char \*file, const char \*arg0, const char \*arg1, ..., (char \*) NULL);

int execle (const char \*path, const char \*arg0, const char \*arg1, ..., (char \*) NULL, char \*const envp []);

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

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

int execvpe (const char \*file, char \*const argv[], char \*const envp[]);

The names of the first five of above functions are of the form execXY. X is either l or v depending upon whether arguments are given in the list format (arg0, arg1, ..., NULL) or arguments are passed in an array (vector). Y is either absent or is either a p or an e. In case Y is p, the PATH environment variable is used to search for the program. If Y is e, then the environment passed in envp array is used. In case of execvpe, X is v and Y is e. The execvpe function is a GNU extension. It is named so as to differentiate it from the execve system call (execve (2)).

### 1.4 Difference between fork and exec

The major difference is that in case of fork, a new child process is created, which is a clone of the parent process. When a process executes exec, no new process is created. The calling process is overwritten by the program whose filename is passed as the first argument. In most cases, the fork system call is followed by an exec call in the newly created child process. The use case is like this. A process executes the fork system call, which creates a new child process. The child process, then, *exec*'s the program to be executed. So, fork and exec are mostly used together. Without fork, exec is of limited use. And, without exec, fork is hardly of any use.

### 2.0 An example

As an example, let's write two programs, parent and child. We will execute parent from the shell's command line. The parent would fork a child process and the latter would exec the child program in it. The parent would wait for the child to do its work and terminate.

### 2.1 Parent

// parent.c: the parent program

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <unistd.h>

#include <string.h>

#include <sys/wait.h>

int main (int argc, char \*\*argv)

{

int i = 0;

long sum;

int pid;

int status, ret;

char \*myargs [] = { NULL };

char \*myenv [] = { NULL };

printf ("Parent: Hello, World!\n");

pid = fork ();

if (pid == 0) {

// I am the child

execve ("child", myargs, myenv);

}

// I am the parent

printf ("Parent: Waiting for Child to complete.\n");

if ((ret = waitpid (pid, &status, 0)) == -1)

printf ("parent:error\n");

if (ret == pid)

printf ("Parent: Child process waited for.\n");

}

### 2.2 Child

// child.c: the child program

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define A 500

#define B 600

#define C 700

int main (int argc, char \*\*argv)

{

int i, j;

long sum;

// Some arbitrary work done by the child

printf ("Child: Hello, World!\n");

for (j = 0; j < 30; j++ ) {

for (i =0; i < 900000; i++) {

sum = A \* i + B \* i \* i + C;

sum %= 543;

}

}

printf ("Child: Work completed!\n");

printf ("Child: Bye now.\n");

exit (0);

}

### 2.3 Executing the parent (and child)

$ # compile parent

$ gcc parent.c -o parent

$ #compile child

$ gcc child.c -o child

$ # run parent (and child)

$ ./parent

Parent: Hello, World!

Parent: Waiting for Child to complete.

Child: Hello, World!

Child: Work completed!

Child: Bye now.

Parent: Child process waited for.

$

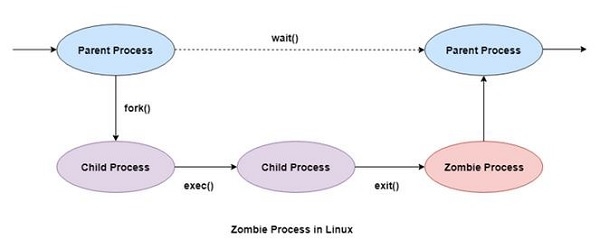
We took a rather simple example just to illustrate the concepts of parent and child programs and the respective processes. A more realistic example is the shell. The shell is the parent process. It reads each line of input from the command line, forks a child shell process, which in turn exec's the command. The shell parent process waits for the child to complete and then prompts for the next command;

Details about the zombie, orphan and daemon processes are given as follows

## Zombie Processes

A zombie process is a process whose execution is completed but it still has an entry in the process table. Zombie processes usually occur for child processes, as the parent process still needs to read its child’s exit status. Once this is done using the wait system call, the zombie process is eliminated from the process table. This is known as reaping the zombie process.

A diagram that demonstrates the creation and termination of a zombie process is given as follows



Zombie processes don't use any system resources but they do retain their process ID. If there are a lot of zombie processes, then all the available process ID’s are monopolized by them. This prevents other processes from running as there are no process ID’s available.

## Orphan Processes

Orphan processes are those processes that are still running even though their parent process has terminated or finished. A process can be orphaned intentionally or unintentionally.

An intentionally orphaned process runs in the background without any manual support. This is usually done to start an indefinitely running service or to complete a long-running job without user attention.

An unintentionally orphaned process is created when its parent process crashes or terminates. Unintentional orphan processes can be avoided using the process group mechanism.

Orphan and zombie examples:

# Zombie and Orphan Process in OS

## Introduction

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.

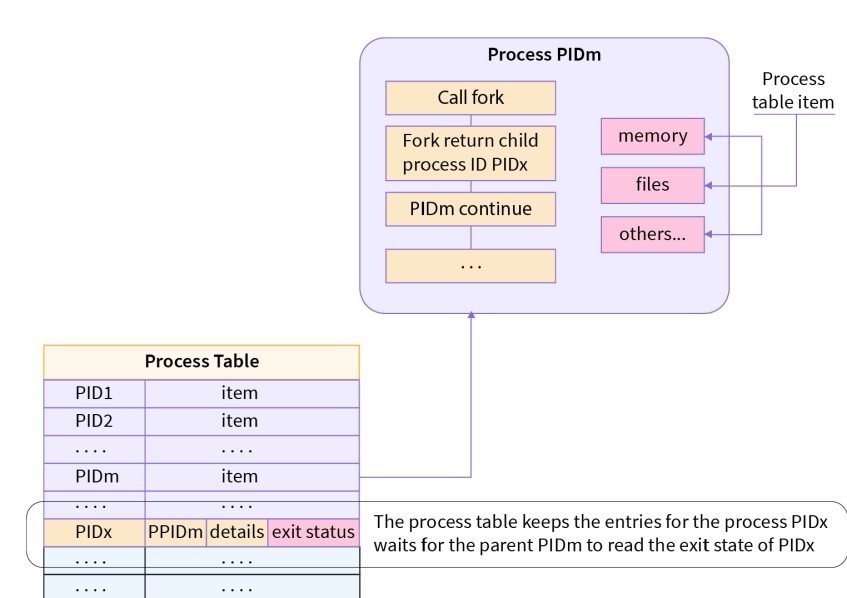
## Scope of article

* This article defines zombie and orphan processes with code in C language.
* This article also covers what happens to a zombie or an orphan process and how they can be harmful to the system.

## What is the Zombie process?

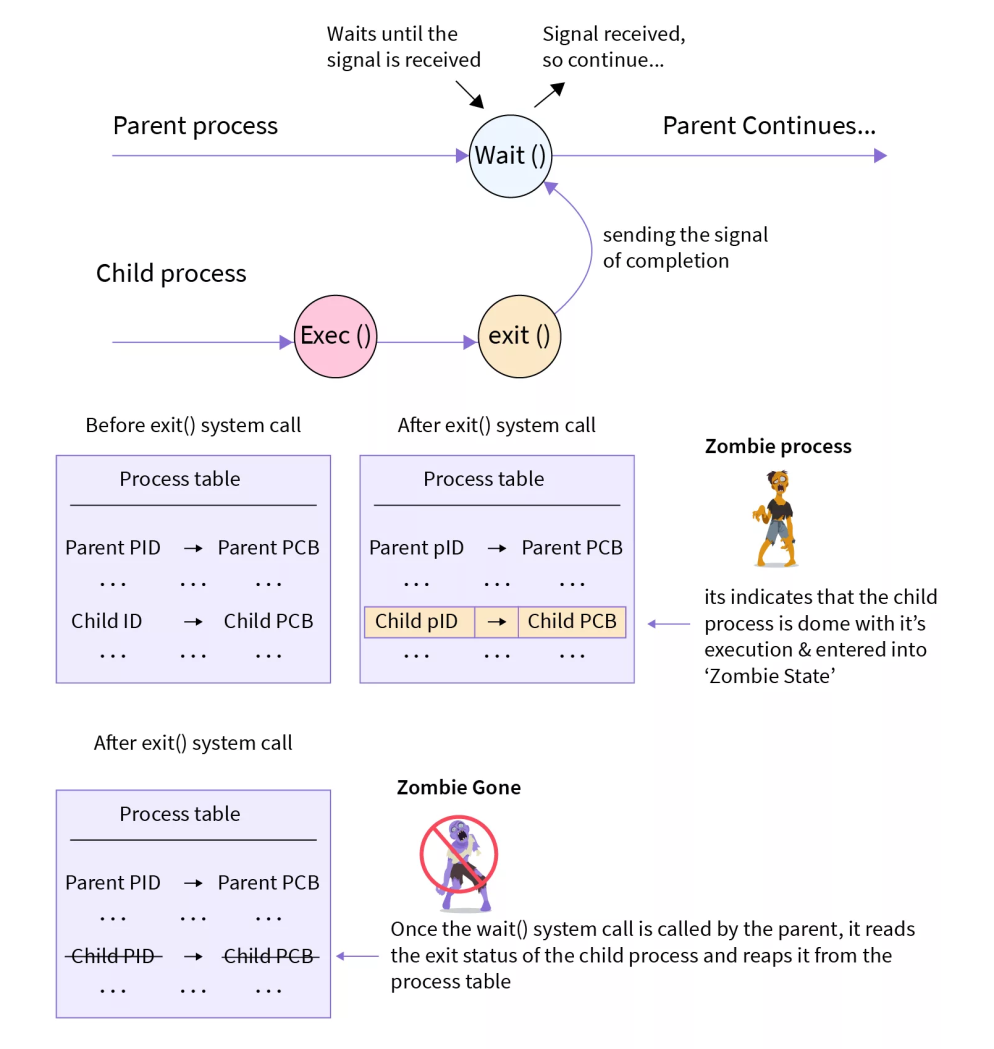
**Zombie process** is also known as ***"dead"*** process. Ideally when a process completes its execution, it's entry from the process table should be removed but this does not happen in case of zombie a process.

**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".
* 

## 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;

}

## 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

**An extreme case - Fork bomb** The program below creates an infinitely many zombie processes because the parent does not wait for it's child process.

#include <unistd.h>

int main()

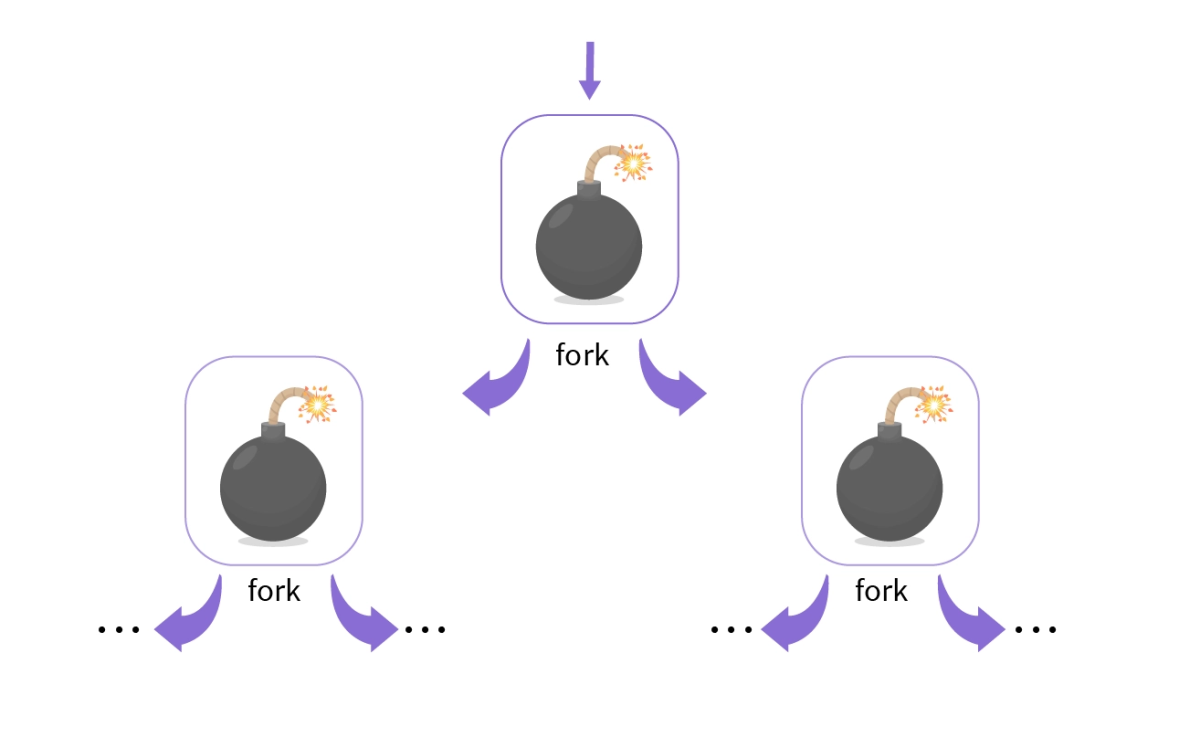
{

while(1)

fork();

return 0;

}



* PIDs in os are finite , when all the PIDs have been consumed by Zombie Process , no new process can be created.
* Solution : Reboot the system

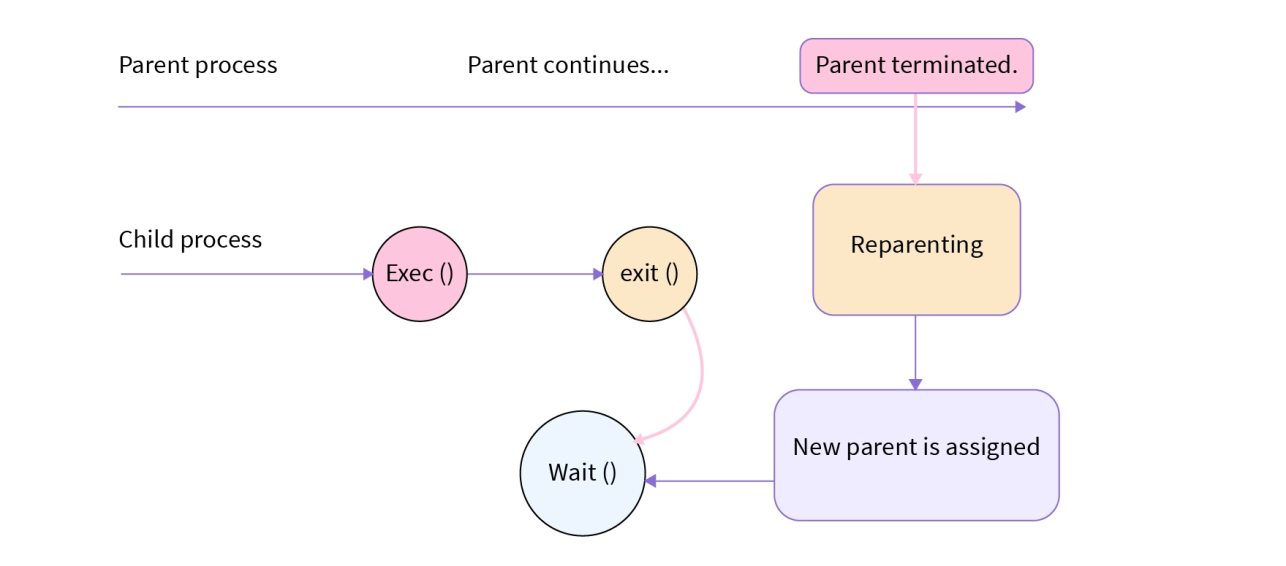
## What is the Orphan Process?

We'll again use real life analogy to understand 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.



## 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;

}

**Output**

My parent's process ID is 32005

I am the parent, my process ID is 32005

The parent's parent, process ID is 31998

Parent terminates

After sleep

I am the child, my process ID is 32006

My parent's process ID is 1

## 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.

## Conclusion

Let's go through what we have learnt till now.

* In linux OS, processes have parent child relationships.
* A zombie process in OS is one that has completed it’s execution but its entry in the process table.
* wait() system call is used to deal with zombie processes.
* An orphan process in OS is one which is executing but it’s parent process has terminated is called an orphan process.
* Kernel allocates a new process as parent process to orphan process. Mostly the new parent is the init process (pid=1).
* Too many zombie processes and orphan processes are harmful.

SESSION 7:

# Signals

## Mandatory assignment

Signals are a limited form of inter-process communication (IPC), typically used in [Unix](https://en.wikipedia.org/wiki/Unix), [Unix-like](https://en.wikipedia.org/wiki/Unix-like), and other [POSIX](https://en.wikipedia.org/wiki/POSIX)-compliant operating systems. [1](http://www.it.uu.se/education/course/homepage/os/vt18/module-2/signals/#fn:wp-signal-ipc) A signal is used to notify a process of an synchronous or asynchronous event.

When a signal is sent, the operating system interrupts the target process’ normal flow of execution to deliver the signal. If the process has previously registered a signal handler, that routine is executed. Otherwise, the default signal handler is executed. [1](http://www.it.uu.se/education/course/homepage/os/vt18/module-2/signals/#fn:wp-signal-ipc)

Each signal is represented by an integer value. Instead of using the numeric values directly, the named constants defined in [signals.h](http://pubs.opengroup.org/onlinepubs/7908799/xsh/signal.h.html" \t "_blank) should be used.

## Clone repository

If you haven’t done so already, you must [clone](http://www.it.uu.se/education/course/homepage/os/vt18/module-2/clone-repository) the module-2 repository.

## Open file

Open the file module-2/mandatory/src/signals.c in the [source code editor](https://en.wikipedia.org/wiki/Source_code_editor) of your choice.

## Study the source code

Study the C source code.

### Header files

First a number of header files are included to get access to a few functions and constants from the [C Standard library](https://www.tutorialspoint.com/c_standard_library).

### Global variable done

A global variable done is initialized to false.

bool done = false;

Later this variable is going to be used to be changed by a signal handler.

### divide\_by\_zero

The divide\_by\_zero function attempts to [divide by zero](https://en.wikipedia.org/wiki/Division_by_zero).

### segfault

The function segfault attempts to [dereference](https://en.wikipedia.org/wiki/Dereference_operator) a [NULL pointer](https://en.wikipedia.org/wiki/Null_pointer) causing a [segmentation fault](https://en.wikipedia.org/wiki/Segmentation_fault).

### signal\_handler

The signal\_handler function will handle signals sent to the process. A [switch statement](https://www.tutorialspoint.com/cprogramming/switch_statement_in_c.htm) is used to determine which signal has been received. An alternative is to use one signal handling function for each signal but here a single signal handling function is used.

### main

All C programs starts to execute in the main function.

* The process ID (PID) is obtained using the [getpid](http://pubs.opengroup.org/onlinepubs/009695399/functions/getpid.html" \t "_blank) function and printed to the terminal with [printf](https://www.tutorialspoint.com/c_standard_library/c_function_printf.htm" \t "_blank).
* A number of lines are commented out, we’ll get back to these later.
* The function [puts](https://www.tutorialspoint.com/c_standard_library/c_function_puts.htm) is used to print the string I'm done! on a separate line to the terminal.
* Finally, [exit](https://www.tutorialspoint.com/c_standard_library/c_function_exit.htm) is used to terminate the process with exit status EXIT\_SUCCESS defined in [stdlib.h](https://www.tutorialspoint.com/c_standard_library/stdlib_h.htm" \t "_blank).

## Program, executable and process

Let’s repeat the differences between a program, an executable and a process.

**Program**

A set of instructions which is in human readable format. A passive entity stored on secondary storage.

**Executable**

A compiled form of a program including machine instructions and static data that a computer can load and execute. A passive entity stored on secondary storage.

**Process**

A program loaded into memory and executing or waiting. A process typically executes for only a short time before it either finishes or needs to perform I/O (waiting). A process is an active entity and needs resources such as CPU time, memory etc to execute.

## The make build tool

The [make](https://en.wikipedia.org/wiki/Make_(software)) build tool is used together with the [Makefile](https://en.wikipedia.org/wiki/Makefile" \t "_blank) to compile all programs in the moudule-2/mandatory/src directory.

## Compile all programs

From a terminal, navigate to the module-2/mandatory directory. To compile all programs, type make and press enter.

$ make

When compiling, make places all executables in the bin directory.

## First test run

Run the signals program.

$ ./bin/signals

You should now see output similar to this in the terminal.

My PID = 81430

I'm done!

Note that the PID value you see will be different.

## New process

Run the program a few times. Note that each time you run the same program the process used to execute the program gets a new process ID (PID).

## C comments

In C, // is used to start a comment reaching to the end of the line.

## Division by zero

To make the program divide by zero, uncomment the following line.

// divide\_by\_zero();

Compile with make.

$ make

Run the program.

$ ./bin/signals

In the terminal you should see something similar to this.

My PID = 81836

[2] 81836 floating point exception ./bin/signals

Division by zero causes an exception. When the OS handles the exception it sends the SIGFPE (fatal arithmetic error) signal to the process executing the division by zero. The default handler for the SIGFPE signal terminates the process and this is exactly what happened here.

Run the program a few times. Each time you run the program the same error (division by zero) happens, causing an exception, causing the OS to send the process the SIGFPE signal, causing the process to terminate.

Synchronous signals

Synchronous signals are delivered to the same process that performed the operation that caused the signal. Division by zero makes the OS send the process the synchronous signal SIGFPE.

## Installing a signal handler

A program can install a signal handler using the signal function.

signal(sig, handler);

**sig**

The signal you want to specify a signal handler for.

**handler**

The function you want to use for handling the signal.

## Handling SIGFPE

Uncomment the following line to install the signal\_handler function as the signal handler for the SIGFPE signal.

// signal(SIGFPE, signal\_handler);

Compile with make.

$ make

Run the program.

$ ./bin/signals

In the terminal you should see something similar to this.

My PID = 81979

Caught SIGFPE: arithmetic exception, such as divide by zero.

This time the signal doesn’t terminate the process immediately. When the process receives the SIGFPE signal the function signal\_handler is executed with the signal number as argument. After printing a message to the terminal the signal handler terminates the process with status EXIT\_FAILURE.

## No more division by zero

Comment out the following line.

divide\_by\_zero();

Compile and run the program Make sure you see output similar to this in the terminal.

My PID = 82040

I'm done!

## Segfault

A segmentation fault (aka segfault) are caused by a program trying to read or write an illegal memory location. To make the program cause a segfault, uncomment the following line.

// segfault();

Compile with make.

$ make

Run the program.

$ ./bin/signals

In the terminal you should see something similar to this.

My PID = 82084

[2] 82084 segmentation fault ./bin/signals

The illegal memory access causes an exception. When the OS handles the exception it sends the SIGSEGV signal to the process executing the illegal memory access. The default handler for the SIGSEGV signal terminates the process and this is exactly what happened here.

Run the program a few times. Each time you run the program the same error (illegal memory access) happen, causing an exception, causing the OS to send the process the SIGSEGV signal, causing the process to terminate.

Synchronous signals

Synchronous signals are delivered to the same process that performed the operation that caused the signal. An illegal memory access makes the OS send the process the synchronous signal SIGSEGV.

## Handling SIGSEGV

Add code to install the function signal\_handler as the signal handler for the SIGSEGV signal. When you run the program you should output similar to this in the terminal.

My PID = 82161

Caught SIGSEGV: segfault.

## No more segfault

Comment out the following line.

segfault();

Compile and run the program Make sure you see output similar to this in the terminal.

My PID = 82040

I'm done!

## Wait for a signal

The pause function is used to block a process until it receives a signal (any signal will do). Uncomment the following line.

// pause();

Compile and run the program. You should see output similar to this in the terminal.

My PID = 82249

The process is now blocked, waiting for any signal to be sent to the process.

## Ctrl+C

To terminate the process, press Ctrl+C in the terminal. Note that once the process terminates you get the terminal prompt back.

My PID = 82249

^C

$

Asynchronous signals

Asynchronous signals are generated by an event external to a running process. Pressing Ctrl+C is an external event causing the OS to send the asynchronous SIGINT (terminal interrupt) signal to the process.

The default signal SIGINT handler terminates the process.

## Handling SIGINT

Add code to install the function signal\_handler as the signal handler for the SIGINT signal. When you run the program the process blocks waiting for any signal. When you press Ctrl+C you should now see output similar to this in the terminal.

My PID = 82477

^CCaught SIGINT: interactive attention signal, probably a ctrl+c.

I'm done!

$

## Open a second terminal

Open a second terminal.

## Sending signals from the terminal

Compile and run the program in one of the terminals. The program should block waiting for any signal. Note the PID of the blocked process.

My PID = 82629

The command kill can be used to send signals to processes from the terminal. To send the SIGINT signal to the blocked process, execute the following command in the terminal where you replace <PID> with the PID of the blocked process.

$ kill -s INT <PID>

In the other terminal you should now see the blocked process execute the signal handler, then continue in main after pause(), print I'm done! and terminate.

My PID = 82629

Caught SIGINT: interactive attention signal, probably a ctrl+c.

I'm done!

$

## Handle SIGUSR1

Add code to make the program print “Hello!” when receiving the SIGUSR1 signal.

* Compile and run the program from one terminal.
* Send the SIGUSR1 signal to the process from the other terminal using the kill command where you replace <PID> with the PID of the blocked process.

$ kill -s SIGUSR1 <PID>

## Don’t terminate on SIGUSR1

How can you make the program print Hello! every time the signal SIGUSR1 is received without terminating?

### Set the global variable done to true

In the signal\_hanlder function, set the global variable done to true when handling the SIGINT signal.

### Block until done

In main, replace the line:

pause();

, with the following while loop:

while (!done);

In C ! is the logical not operator. This while loop repeatedly checks the global variable done until it becomes true.

### Compile, run and test

Compile and run the program from one terminal and send signals to the process from the other terminal.

* Are you able to send multiple SIGUSR1 signals to the process?
* Are you able to break out of the while loop and terminate the process by sending the signal SIGINT to the process, or by pressing Ctrl+C from the terminal?

### Bug?

Depending on your compiler the program may not break out of the while(!done) loop

An optimizing compiler

An optimizing compiler may detect that the variable done is not changed in the while(!done); loop and replace the loop with if (!false);.

### Volatile

Do you remember the volatile keyword?

Volatile

The volatile keyword is used to make sure that the contents of a variable is always read from memory.

Make the global variable done volatile.

### Compile, run and test

Compile and run the program from one terminal and send signals to the process from the other terminal.

* Make sure you are able to send multiple SIGUSR1 signals to the process.
* Make sure you can terminate the process by sending the signal SIGINT to the process, or by pressing Ctrl+C from the terminal.

### sig\_atomic\_t

The data type sig\_atomic\_t guarantees that reading and writing a variable happen in a single instruction, so there’s no way for a signal handler to run “in the middle” of an access. In general, you should always make any global variables changed by a signal handler be of the data type sig\_atomic\_t.

Change the datatype of the global variable done from bool to sig\_atomic\_t.

### Use pause instead

Using a while loop to repeatedly check the global variable done is not a very efficient use of the CPU. A better way is to change the loop to:

while (pause()) {

if (done) break;

};

THREADS:

# Thread in Operating System

* **Difficulty Level :** [Easy](https://www.geeksforgeeks.org/easy/)
* **Last Updated :** 28 Jun, 2021

 Read

 Discuss

**What is a Thread?**  
A thread is a path of execution within a process. A process can contain multiple threads.  
**Why Multithreading?**  
A thread is also known as lightweight process. The idea is to achieve parallelism by dividing a process into multiple threads. For example, in a browser, multiple tabs can be different threads. MS Word uses multiple threads: one thread to format the text, another thread to process inputs, etc. More advantages of multithreading are discussed below  
**Process vs Thread?**  
The primary difference is that threads within the same process run in a shared memory space, while processes run in separate memory spaces.  
Threads are not independent of one another like processes are, and as a result threads share with other threads their code section, data section, and OS resources (like open files and signals). But, like process, a thread has its own program counter (PC), register set, and stack space.  
***Advantages of Thread over Process***  
1. Responsiveness: If the process is divided into multiple threads, if one thread completes its execution, then its output can be immediately returned.

2. Faster context switch: Context switch time between threads is lower compared to process context switch. Process context switching requires more overhead from the CPU.

3. Effective utilization of multiprocessor system: If we have multiple threads in a single process, then we can schedule multiple threads on multiple processor. This will make process execution faster.

4. Resource sharing: Resources like code, data, and files can be shared among all threads within a process.  
Note: stack and registers can’t be shared among the threads. Each thread has its own stack and registers.

5. Communication: Communication between multiple threads is easier, as the threads shares common address space. while in process we have to follow some specific communication technique for communication between two process.

6. Enhanced throughput of the system: If a process is divided into multiple threads, and each thread function is considered as one job, then the number of jobs completed per unit of time is increased, thus increasing the throughput of the system.  
**Types of Threads**  
There are two types of threads.  
User Level Thread  
Kernel Level Thread.

# Threads and its types in Operating System

* **Difficulty Level :** [Easy](https://www.geeksforgeeks.org/easy/)
* **Last Updated :** 07 Apr, 2022

 Read

 Discuss

Thread is a single sequence stream within a process. Threads have same properties as of the process so they are called as light weight processes. Threads are executed one after another but gives the illusion as if they are executing in parallel. Each thread has different states. Each thread has

1. A program counter
2. A register set
3. A stack space

Threads are not independent of each other as they share the code, data, OS resources etc.

**Similarity between Threads and Processes –**

* Only one thread or process is active at a time
* Within process both execute sequential
* Both can create children

**Differences between Threads and Processes –**

* Threads are not independent, processes are.
* Threads are designed to assist each other, processes may or may not do it

**Types of Threads:**

1. **User Level thread (ULT) –** Is implemented in the user level library, they are not created using the system calls. Thread switching does not need to call OS and to cause interrupt to Kernel. Kernel doesn’t know about the user level thread and manages them as if they were single-threaded processes.
   * **Advantages of ULT –**
     + Can be implemented on an OS that doesn’t support multithreading.
     + Simple representation since thread has only program counter, register set, stack space.
     + Simple to create since no intervention of kernel.
     + Thread switching is fast since no OS calls need to be made.
   * **Limitations of ULT –**
     + No or less co-ordination among the threads and Kernel.
     + If one thread causes a page fault, the entire process blocks.
2. **Kernel Level Thread (KLT) –** Kernel knows and manages the threads. Instead of thread table in each process, the kernel itself has thread table (a master one) that keeps track of all the threads in the system. In addition kernel also maintains the traditional process table to keep track of the processes. OS kernel provides system call to create and manage threads.
   * **Advantages of KLT –**
     + Since kernel has full knowledge about the threads in the system, scheduler may decide to give more time to processes having large number of threads.
     + Good for applications that frequently block.
   * **Limitations of KLT –**
     + Slow and inefficient.
     + It requires thread control block so it is an overhead.

**Summary:**

1. Each ULT has a process that keeps track of the thread using the Thread table.
2. Each KLT has Thread Table (TCB) as well as the Process Table (PCB).

# Program to create Threads in Linux

In this post we discuss the use of ***pthread\_create*** function create threads in linux. Various program to create threads in Linux are discussed below that shows how to create threads, how to pass input to thread and how to return value from thread.

## Syntax

#include<pthread.h>

int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr,

void \*(\*start\_routine) (void \*), void arg);

The first parameter is the buffer which will contain the ID of the new thread, if pthread\_create is successful. The second parameter specifies the attributes of the thread. This parameter is generally NULL until you want to change the default settings. The third parameter is the name the function which the thread will execute. Hence, everything that you want the thread to do should be defined in this function. Lastly, the fourth parameter is the input to the function in the third parameter. If the function in the third parameter does not take any input then the fourth parameter is NULL.

##### Program 1: Program to create threads in linux. Thread prints 0-4 while the main process prints 20-24

#include<stdio.h>  
#include<stdlib.h>  
#include<unistd.h>  
#include<pthread.h>  
void \*thread\_function(void \*arg);  
 int i,j;  
 int main() {  
 pthread\_t a\_thread; //thread declaration

pthread\_create(&a\_thread, NULL, thread\_function, NULL);   
//thread is created  
 pthread\_join(a\_thread, NULL); //process waits for thread to finish . //Comment this line to see the difference  
 printf("Inside Main Program\n");  
 for(j=20;j<25;j++)  
 {  
 printf("%d\n",j);  
 sleep(1);  
 }  
 }

void \*thread\_function(void \*arg) {   
// the work to be done by the thread is defined in this function  
 printf("Inside Thread\n");  
 for(i=0;i<5;i++)  
 {  
 printf("%d\n",i);  
 sleep(1);  
 }  
 }

**Note**: To compile any program which involves creation of thread(s) use pthread library (lpthread)  
Suppose the above program is named “Thread.c”, then to compile write  
$gcc Thread.c -lpthread  
To run the command remains same  
$./a.out

### Output

0

1

2

3

4

Inside Main Program

20

21

22

23

24

### How it works?

pthread\_create() creates a new thread which starts to execute thread\_function. This function creates a loop which prints 0-4. The sleep function makes the thread go to sleep after each digit is printed. pthread\_join() makes the main function wait until the newly created thread finishes its execution. So the control returns to the main function only when the thread finishes. Then the main function prints “Inside Main program” and executes the loop from 20-24.

## How a thread returns a value to the main process?

pthread\_exit() is used to return a value from the thread back to the main process. The program below shows this. The program also shows how to pass value to a thread from the main process.

##### **Program 2:** Program to create a thread. The thread prints numbers from zero to n, where value of n is passed from the main process to the thread. The main process also waits for the thread to finish first and then prints from 20-24.

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<pthread.h>

#include<string.h>

void \*thread\_function(void \*arg);

int i,n,j;

int main() {

char \*m="5";

pthread\_t a\_thread; //thread declaration

void \*result;

pthread\_create(&a\_thread, NULL, thread\_function, m); //thread is created

pthread\_join(a\_thread, &result);

printf("Thread joined\n");

for(j=20;j<25;j++)

{

printf("%d\n",j);

sleep(1);

}

printf("thread returned %s\n",(char \*)result);

}

void \*thread\_function(void \*arg) {

int sum=0;

n=atoi(arg);

for(i=0;i<n;i++)  
 {  
 printf("%d\n",i);  
 sleep(1);  
 }  
 pthread\_exit("Done"); // Thread returns "Done"  
 }

### How it works?

When we call pthread\_create() function a value if passed to the thread by passing that value as the fourth parameter of the pthread\_create() function.

## How to pass multiple values to a thread using structure?

##### Program 3: Program to create a thread. The thread is passed more than one input from the main process. For passing multiple inputs we need to create structure and include all the variables that are to be passed in this structure.

#include<stdio.h

#include<pthread.h>

struct arg\_struct { //structure which contains multiple variables that are to passed as input to the thread

int arg1;

int arg2;

};

void \*arguments(void \*arguments)

{

struct arg\_struct \*args=arguments;

printf("%d\n", args -> arg1);

printf("%d\n", args -> arg2);

pthread\_exit(NULL);

}

int main()  
 {  
 pthread\_t t;  
 struct arg\_struct args;  
 args.arg1 = 5;  
 args.arg2 = 7;  
 pthread\_create(&t, NULL, arguments, &args);   
//structure passed as 4th argument  
 pthread\_join(t, NULL); /\* Wait until thread is finished \*/  
}

Note: Just as above we can also pass an array to a thread.

# Memory Management in Operating System

* **Difficulty Level :** [Medium](https://www.geeksforgeeks.org/medium/)
* **Last Updated :** 03 Jan, 2023

 Read

 Discuss

The term Memory can be defined as a collection of data in a specific format. It is used to store instructions and process data. The memory comprises a large array or group of words or bytes, each with its own location. The primary motive of a computer system is to execute programs. These programs, along with the information they access, should be in the main memory during execution. The CPU fetches instructions from memory according to the value of the program counter.

To achieve a degree of multiprogramming and proper utilization of memory, memory management is important. Many memory management methods exist, reflecting various approaches, and the effectiveness of each algorithm depends on the situation.

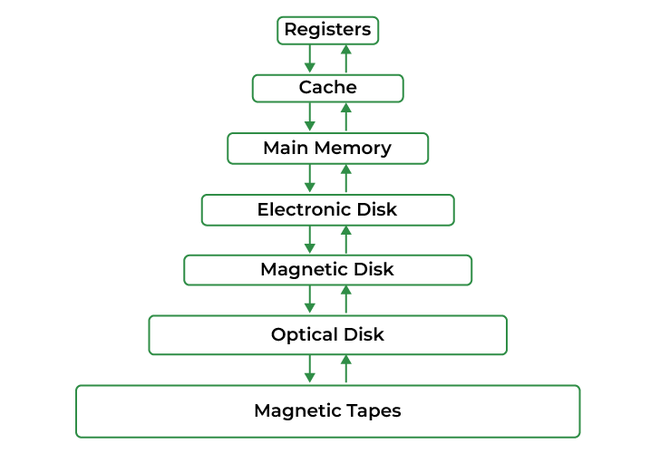
*Here, we will cover the following memory management topics:*

* *What is Main Memory*
* *What is Memory Management*
* *Why memory Management is required*
* *Logical address space and Physical address space*
* *Static and dynamic loading*
* *Static and dynamic linking*
* *Swapping*
* *Contiguous Memory allocation*
  + *Memory Allocation*
    - *First Fit*
    - *Best Fit*
    - *Worst Fit*
  + *Fragmentation*
    - *Internal Fragmentation*
    - *External Fragmentation*
  + *Paging*

**Now before, We start memory management let us know what is main memory.**

### What is Main Memory:

The main memory is central to the operation of a modern computer. Main Memory is a large array of words or bytes, ranging in size from hundreds of thousands to billions. Main memory is a repository of rapidly available information shared by the CPU and I/O devices. Main memory is the place where programs and information are kept when the processor is effectively utilizing them.  Main memory is associated with the processor, so moving instructions and information into and out of the processor is extremely fast.  Main memory is also known as RAM(Random Access Memory). This memory is a volatile memory.RAM lost its data when a power interruption occurs.



### What is Memory Management :

In a multiprogramming computer, the operating system resides in a part of memory and the rest is used by multiple processes. The task of subdividing the memory among different processes is called memory management. Memory management is a method in the operating system to manage operations between main memory and disk during process execution. The main aim of memory management is to achieve efficient utilization of memory.

### Why Memory Management is required:

* Allocate and de-allocate memory before and after process execution.
* To keep track of used memory space by processes.
* To minimize fragmentation issues.
* To proper utilization of main memory.
* To maintain data integrity while executing of process.

**Now we are discussing the concept of logical address space and Physical address space:**

### Logical and Physical Address Space:

**Logical Address space:**An address generated by the CPU is known as a “Logical Address”. It is also known as a Virtual address. Logical address space can be defined as the size of the process. A logical address can be changed.

**Physical Address space:**An address seen by the memory unit (i.e the one loaded into the memory address register of the memory) is commonly known as a “Physical Address”. A Physical address is also known as a Real address. The set of all physical addresses corresponding to these logical addresses is known as Physical address space. A physical address is computed by MMU. The run-time mapping from virtual to physical addresses is done by a hardware device Memory Management Unit(MMU). The physical address always remains constant.

### Static and Dynamic Loading:

Loading a process into the main memory is done by a loader. There are two different types of loading :

* **Static loading**:- loading the entire program into a fixed address. It requires more memory space.
* **Dynamic loading**:- The entire program and all data of a process must be in physical memory for the process to execute. So, the size of a process is limited to the size of physical memory. To gain proper memory utilization, dynamic loading is used. In dynamic loading, a routine is not loaded until it is called. All routines are residing on disk in a relocatable load format. One of the advantages of dynamic loading is that unused routine is never loaded. This loading is useful when a large amount of code is needed to handle it efficiently.

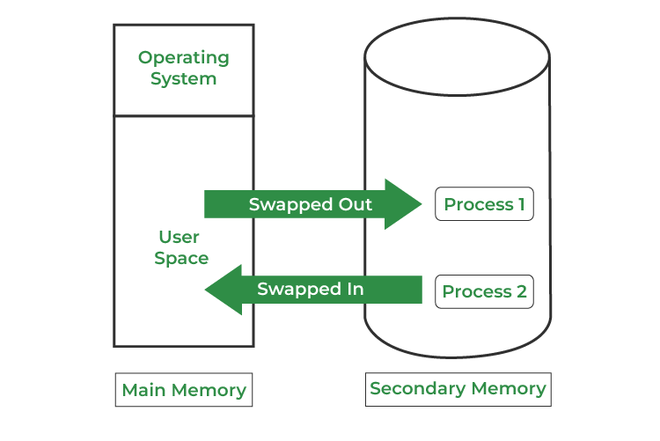
### Static and Dynamic linking:

To perform a linking task a linker is used. A linker is a program that takes one or more object files generated by a compiler and combines them into a single executable file.

* **Static linking:**In static linking, the linker combines all necessary program modules into a single executable program. So there is no runtime dependency. Some operating systems support only static linking, in which system language libraries are treated like any other object module.
* **Dynamic linking:** The basic concept of dynamic linking is similar to dynamic loading. In dynamic linking, “Stub” is included for each appropriate library routine reference. A stub is a small piece of code. When the stub is executed, it checks whether the needed routine is already in memory or not. If not available then the program loads the routine into memory.

### Swapping :

When a process is executed it must have resided in memory. Swapping is a process of swapping a process temporarily into a secondary memory from the main memory, which is fast as compared to secondary memory. A swapping allows more processes to be run and can be fit into memory at one time. The main part of swapping is transferred time and the total time is directly proportional to the amount of memory swapped. Swapping is also known as roll-out, roll in, because if a higher priority process arrives and wants service, the memory manager can swap out the lower priority process and then load and execute the higher priority process. After finishing higher priority work, the lower priority process swapped back in memory and continued to the execution process.



*swapping in memory managment*

**memory management with monoprogramming(without swapping):**

this is the simplest memory management approach the memory is divided into two sections:

* one part for operating system
* second part for user program

|  |  |
| --- | --- |
| fence register | |
| operating system | user program |

* in this approach operating system keep tracks of first and last location available for allocation of user program
* operating system is loaded either at bottom or at top
* interrupt vector are often loaded in low memory therefore it makes sense to load operating system in low memory
* sharing of data and code does not make much sense in single process environment
* operating system can be protected from user program with the help of fence register.

**Advantage**

it is simple management approach

**Disadvantage**

it does not support multiprogramming

memory is wasted

**multiprogramming with fixed partitions(without swapping):**

* memory partitions scheme with fixed number of partitions was introduced to support multiprogramming. this scheme is based on contiguous allocation
* each partition is block of contiguous memory
* memory is partition into fixed number of partition
* each partition is of fixed size

**an example of partition memory is shown below:**

as shown in fig. memory is partition into 5 regions the region is reserved for updating system the remaining four partitions are for user program

|  |
| --- |
| operating system |
| p1 |
| p2 |
| p3 |
| p4 |

                                                                                                fig. fixed size partitioning

**partition table:**

once partitions are defined operating system keeps track of status of  memory partitions it is done through data structure called partition table

|  |  |  |  |
| --- | --- | --- | --- |
| sr.no | starting address  of partition | size of  partition | status |
| 1 | 0k | 200k | allocated |
| 2 | 200k | 100k | free |
| 3 | 300k | 150k | free |
| 4 | 450k | 250k | allocated |

**sample partition table**

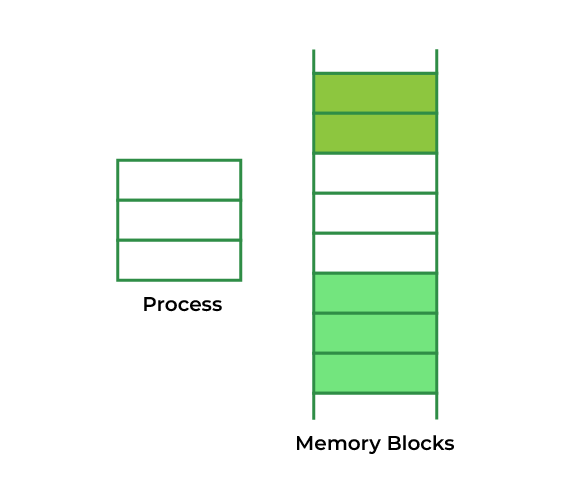
**logical versus physical address**

an address generated by CPU is commonly referred to a logical address. the address seen by memory unit is known as physical address

logical address can be mapped to physical address by hardware with the help of base register this is known as dynamic relocation of memory reference.

**Contiguous  Memory Allocation :**

The main memory should oblige both the operating system and the different client processes.  Therefore, the allocation of memory becomes an important task in the operating system.  The memory is usually divided into two partitions: one for the resident operating system and one for the user processes. We normally need several user processes to reside in memory simultaneously. Therefore, we need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory. In adjacent memory allotment, each process is contained in a single contiguous segment of memory.



*contiguous memory allocation*

### Memory allocation:

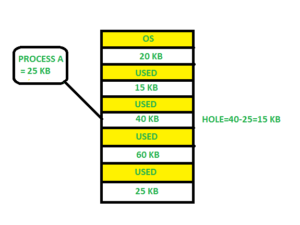
To gain proper memory utilization, memory allocation must be allocated efficient manner. One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions and each partition contains exactly one process. Thus, the degree of multiprogramming is obtained by the number of partitions.

**Multiple partition allocation**: In this method, a process is selected from the input queue and loaded into the free partition. When the process terminates, the partition becomes available for other processes.

**Fixed partition allocation:** In this method, the operating system maintains a table that indicates which parts of memory are available and which are occupied by processes. Initially, all memory is available for user processes and is considered one large block of available memory. This available memory is known as a “Hole”. When the process arrives and needs memory, we search for a hole that is large enough to store this process. If the requirement is fulfilled then we allocate memory to process, otherwise keeping the rest available to satisfy future requests. While allocating a memory sometimes dynamic storage allocation problems occur, which concerns how to satisfy a request of size n from a list of free holes. There are some solutions to this problem:

**First fit:-**

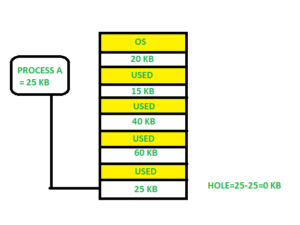
In the first fit, the first available free hole fulfills the requirement of the process allocated.



Here, in this diagram 40 KB memory block is the first available free hole that can store process A (size of 25 KB), because the first two blocks did not have sufficient memory space.

**Best fit:-**

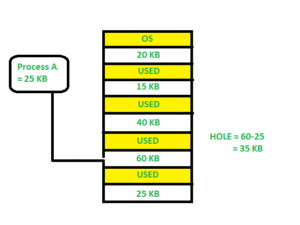
In the best fit, allocate the smallest hole that is big enough to process requirements. For this, we search the entire list, unless the list is ordered by size.



Here in this example, first, we traverse the complete list and find the last hole 25KB is the best suitable hole for Process A(size 25KB).

In this method memory utilization is maximum as compared to other memory allocation techniques.

**Worst fit:-**In the worst fit, allocate the largest available hole to process. This method produces the largest leftover hole.



Here in this example, Process A (Size 25 KB) is allocated to the largest available memory block which is 60KB. Inefficient memory utilization is a major issue in the worst fit.

### Fragmentation:

Fragmentation is defined as when the process is loaded and removed after execution from memory, it creates a small free hole. These holes can not be assigned to new processes because holes are not combined or do not fulfill the memory requirement of the process.  To achieve a degree of multiprogramming, we must reduce the waste of memory or fragmentation problems. In the operating systems two types of fragmentation:

**Internal fragmentation:**

Internal fragmentation occurs when memory blocks are allocated to the process more than their requested size. Due to this some unused space is leftover and creates an internal fragmentation problem.

 Example: Suppose there is a fixed partitioning is used for memory allocation and the different size of block 3MB, 6MB, and 7MB space in memory. Now a new process p4 of size 2MB comes and demand for the block of memory. It gets a memory block of 3MB but 1MB block memory is a waste, and it can not be allocated to other processes too. This is called internal fragmentation.

**External fragmentation:**

In external fragmentation, we have a free memory block, but we can not assign it to process because blocks are not contiguous.

Example: Suppose (consider above example) three process p1, p2, p3 comes with size 2MB, 4MB, and 7MB respectively. Now they get memory blocks of size 3MB, 6MB, and 7MB allocated respectively. After allocating process p1 process and p2 process left 1MB and 2MB. Suppose a new process p4 comes and demands a 3MB block of memory, which is available, but we can not assign it because free memory space is not contiguous.  This is called external fragmentation.

Both the first fit and best-fit systems for memory allocation affected by external fragmentation. To overcome the external fragmentation problem Compaction is used. In the compaction technique, all free memory space combines and makes one large block. So, this space can be used by other processes effectively.

Another possible solution to the external fragmentation is to allow the logical address space of the processes to be noncontiguous, thus permit a process to be allocated physical memory wherever the latter is available.

### Paging:

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory. This scheme permits the physical address space of a process to be non-contiguous.

* Logical Address or Virtual Address (represented in bits): An address generated by the CPU
* Logical Address Space or Virtual Address Space (represented in words or bytes): The set of all logical addresses generated by a program
* Physical Address (represented in bits): An address actually available on a memory unit
* Physical Address Space (represented in words or bytes): The set of all physical addresses corresponding to the logical addresses

**Example:**

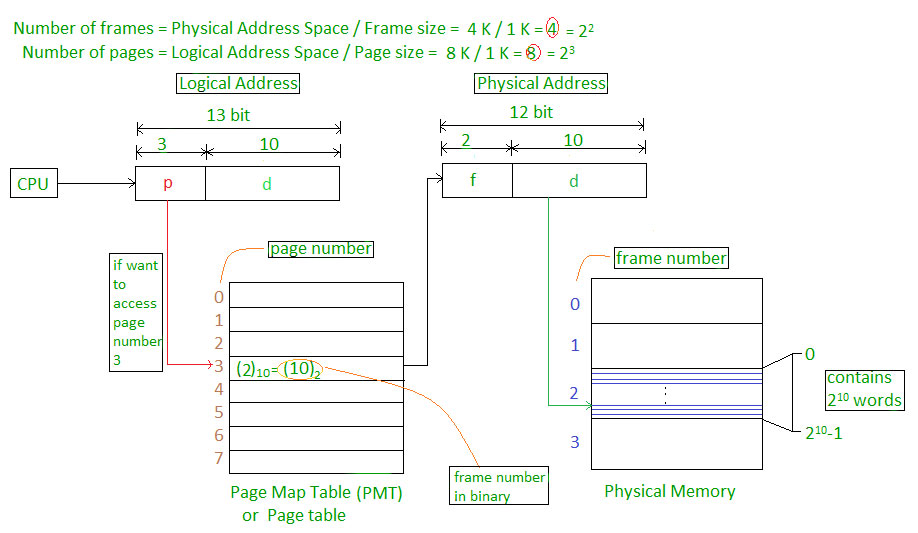
* If Logical Address = 31 bits, then Logical Address Space = 231 words = 2 G words (1 G = 230)
* If Logical Address Space = 128 M words = 27 \* 220 words, then Logical Address = log2 227 = 27 bits
* If Physical Address = 22 bits, then Physical Address Space = 222 words = 4 M words (1 M = 220)
* If Physical Address Space = 16 M words = 24 \* 220 words, then Physical Address = log2 224 = 24 bits

The mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device and this mapping is known as the paging technique.

* The Physical Address Space is conceptually divided into several fixed-size blocks, called **frames**.
* The Logical Address Space is also split into fixed-size blocks, called **pages**.
* Page Size = Frame Size

Let us consider an example:

* Physical Address = 12 bits, then Physical Address Space = 4 K words
* Logical Address = 13 bits, then Logical Address Space = 8 K words
* Page size = frame size = 1 K words (assumption)



The address generated by the CPU is divided into

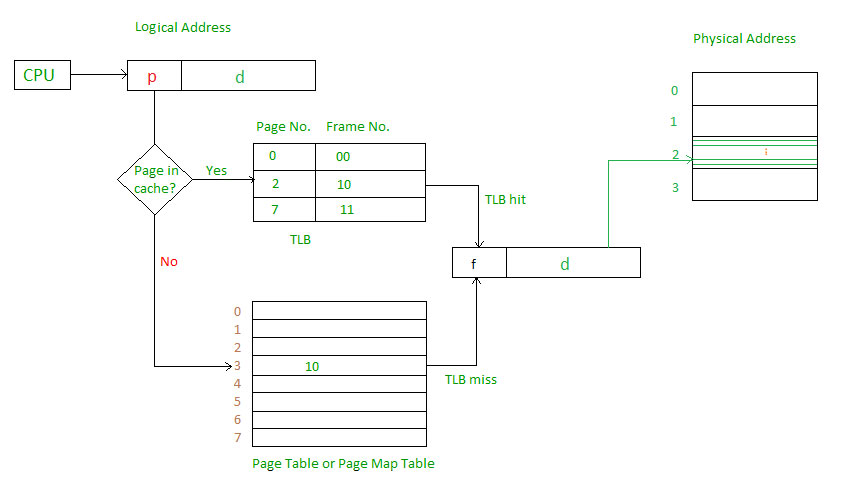
* **Page number(p):** Number of bits required to represent the pages in Logical Address Space or Page number
* **Page offset(d):** Number of bits required to represent a particular word in a page or page size of Logical Address Space or word number of a page or page offset.

Physical Address is divided into

* **Frame number(f):** Number of bits required to represent the frame of Physical Address Space or Frame number frame
* **Frame offset(d):** Number of bits required to represent a particular word in a frame or frame size of Physical Address Space or word number of a frame or frame offset.

The hardware implementation of the page table can be done by using dedicated registers. But the usage of register for the page table is satisfactory only if the page table is small. If the page table contains a large number of entries then we can use TLB(translation Look-aside buffer), a special, small, fast look-up hardware cache.

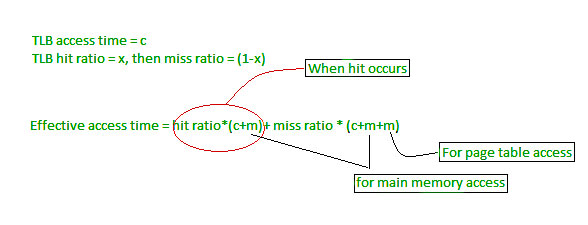
* The TLB is an associative, high-speed memory.
* Each entry in TLB consists of two parts: a tag and a value.
* When this memory is used, then an item is compared with all tags simultaneously. If the item is found, then the corresponding value is returned.



Main memory access time = m

If page table are kept in main memory,

Effective access time = m(for page table) + m(for particular page in page table)



# What is dirty bit? - operating system

|  |
| --- |
| 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 is dirty bit? A dirty bit is a flag that indicates whether an attribute needs to be updated. Such situations usually occur when a bit in a memory cache or virtual memory page that has been changed by a processor but has not been updated in the storage. |

* [Next Page »](https://www.careerride.com/os-page-fault.aspx)

[**What is page fault and when does it occur? - operating system**](https://www.careerride.com/os-page-fault.aspx)

A page is a fixed length memory block used as a transferring unit between physical memory and an external storage......

[**36 Solaris Interview Questions and Answers - Freshers, Experienced**](https://www.careerride.com/solaris-interview-questions.aspx)

Solaris interview questions and answers, Solaris interview FAQ - Explain the types of installations in Solaris, How to add the our own run control script in server, What are the RAID levels in Solaris?, What are NFS daemons?............

[**OS Synchronization Interview Questions and Answers**](https://www.careerride.com/Interview-Questions-OS-Synchronization.aspx)

OS Synchronization - In this series, we have covered all about OS Synchronization and answered the questions that might be asked during an interview.

# Reentrant Function

* **Difficulty Level :** [Easy](https://www.geeksforgeeks.org/easy/)
* **Last Updated :** 26 Apr, 2018

 Read

 Discuss

A function is said to be reentrant if there is a provision to interrupt the function in the course of execution, service the interrupt service routine and then resume the earlier going on function, without hampering its earlier course of action. Reentrant functions are used in applications like hardware interrupt handling, recursion, etc.  
  
   
The function has to satisfy certain conditions to be called as reentrant:  
1. It may not use global and static data. Though there are no restrictions, but it is generally not advised. because the interrupt may change certain global values and resuming the course of action of the reentrant function with the new data may give undesired results.  
  
   
2. It should not modify it’s own code. This is important because the course of action of the function should remain the same throughout the code. But, this may be allowed in case the interrupt routine uses a local copy of the reentrant function every time it uses different values or before and after the interrupt.  
   
3. Should not call another non-reentrant function.

**Thread safety and Reentrant functions**  
Reentrancy is distinct from, but closely related to, thread-safety. A function can be thread-safe and still not reentrant. For example, a function could be wrapped all around with a mutex (which avoids problems in multithreading environments), but if that function is used in an interrupt service routine, it could starve waiting for the first execution to release the mutex. The key for avoiding confusion is that reentrant refers to only one thread executing. It is a concept from the time when no multitasking operating systems existed. (Source :<https://en.wikipedia.org/wiki/Reentrancy_(computing)>)

Example of Non-Reentrant Functions:

|  |
| --- |
| // A non-reentrant example  // [The function depends on global variable i]  int i;  // Both fun1() and fun2() are not reentrant  // fun1() is NOT reentrant because it uses global variable i  int fun1()  {      return i \* 5;  }  // fun2() is NOT reentrant because it calls a non-reentrant  // function  int fun2()  {     return fun1() \* 5;  } |

**Example of Reentrant Functions:**  
In the below code, fun2 is a reentrant function. If an interrupt that pauses its execution and shifts the control to fun1. After fun1 completes, the control is again transferred to fun2 and it reenters the execution phase.

|  |
| --- |
| // Both fun1() and fun2() are reentrant  int fun1(int i)  {      return i \* 5;  }  int fun2(int i)  {     return fun1(i) \* 5;  } |

Article compiled by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**What is throttling?**

Throttling is a mechanism in Intel® Processors to reduce the clock speed when the temperature in the system reaches above TJ Max (or Tcase). This is to protect the processor and to indicate to the user that there is an overheating issue in their system that they need to monitor.

**Advantages of throttling function:**

* It prevents frequent calling of the function.
* It makes the website faster and controls the rate at which a particular function is called.

Session 10:

# Virtual Memory in Operating System

* **Difficulty Level :** [Medium](https://www.geeksforgeeks.org/medium/)
* **Last Updated :** 14 Jul, 2022

 Read

 Discuss

Virtual Memory is a storage allocation scheme in which secondary memory can be addressed as though it were part of the main memory. The addresses a program may use to reference memory are distinguished from the addresses the memory system uses to identify physical storage sites, and program-generated addresses are translated automatically to the corresponding machine addresses.

The size of virtual storage is limited by the addressing scheme of the computer system and the amount of secondary memory is available not by the actual number of the main storage locations.

It is a technique that is implemented using both hardware and software. It maps memory addresses used by a program, called virtual addresses, into physical addresses in computer memory.

1. All memory references within a process are logical addresses that are dynamically translated into physical addresses at run time. This means that a process can be swapped in and out of the main memory such that it occupies different places in the main memory at different times during the course of execution.
2. A process may be broken into a number of pieces and these pieces need not be continuously located in the main memory during execution. The combination of dynamic run-time address translation and use of page or segment table permits this.

If these characteristics are present then, it is not necessary that all the pages or segments are present in the main memory during execution. This means that the required pages need to be loaded into memory whenever required. Virtual memory is implemented using Demand Paging or Demand Segmentation.

**Demand Paging :**   
The process of loading the page into memory on demand (whenever page fault occurs) is known as demand paging.   
The process includes the following steps :

virtual_mem

1. If the CPU tries to refer to a page that is currently not available in the main memory, it generates an interrupt indicating a memory access fault.
2. The OS puts the interrupted process in a blocking state. For the execution to proceed the OS must bring the required page into the memory.
3. The OS will search for the required page in the logical address space.
4. The required page will be brought from logical address space to physical address space. The page replacement algorithms are used for the decision-making of replacing the page in physical address space.
5. The page table will be updated accordingly.
6. The signal will be sent to the CPU to continue the program execution and it will place the process back into the ready state.

Hence whenever a page fault occurs these steps are followed by the operating system and the required page is brought into memory.

**Advantages :**

* More processes may be maintained in the main memory: Because we are going to load only some of the pages of any particular process, there is room for more processes. This leads to more efficient utilization of the processor because it is more likely that at least one of the more numerous processes will be in the ready state at any particular time.
* A process may be larger than all of the main memory: One of the most fundamental restrictions in programming is lifted. A process larger than the main memory can be executed because of demand paging. The OS itself loads pages of a process in the main memory as required.
* It allows greater multiprogramming levels by using less of the available (primary) memory for each process.

**Page Fault Service Time :**   
The time taken to service the page fault is called page fault service time. The page fault service time includes the time taken to perform all the above six steps.

Let Main memory access time is: m

Page fault service time is: s

Page fault rate is : p

Then, Effective memory access time = (p\*s) + (1-p)\*m

**Swapping:**

Swapping a process out means removing all of its pages from memory, or marking them so that they will be removed by the normal page replacement process. Suspending a process ensures that it is not runnable while it is swapped out. At some later time, the system swaps back the process from the secondary storage to the main memory. When a process is busy swapping pages in and out then this situation is called thrashing.

**Thrashing :**

virtual_mem_2

At any given time, only a few pages of any process are in the main memory and therefore more processes can be maintained in memory. Furthermore, time is saved because unused pages are not swapped in and out of memory. However, the OS must be clever about how it manages this scheme. In the steady-state practically, all of the main memory will be occupied with process pages, so that the processor and OS have direct access to as many processes as possible. Thus when the OS brings one page in, it must throw another out. If it throws out a page just before it is used, then it will just have to get that page again almost immediately. Too much of this leads to a condition called Thrashing. The system spends most of its time swapping pages rather than executing instructions. So a good page replacement algorithm is required.

In the given diagram, the initial degree of multiprogramming up to some extent of point(lambda), the CPU utilization is very high and the system resources are utilized 100%. But if we further increase the degree of multiprogramming the CPU utilization will drastically fall down and the system will spend more time only on the page replacement and the time is taken to complete the execution of the process will increase. This situation in the system is called thrashing.

**Causes of Thrashing :**

1. **High degree of multiprogramming**: If the number of processes keeps on increasing in the memory then the number of frames allocated to each process will be decreased. So, fewer frames will be available for each process. Due to this, a page fault will occur more frequently and more CPU time will be wasted in just swapping in and out of pages and the utilization will keep on decreasing.

For example:   
Let free frames = 400   
**Case 1**: Number of process = 100   
Then, each process will get 4 frames.

**Case 2**: Number of processes = 400   
Each process will get 1 frame.   
Case 2 is a condition of thrashing, as the number of processes is increased, frames per process are decreased. Hence CPU time will be consumed in just swapping pages. 

1. **Lacks of Frames**: If a process has fewer frames then fewer pages of that process will be able to reside in memory and hence more frequent swapping in and out will be required. This may lead to thrashing. Hence sufficient amount of frames must be allocated to each process in order to prevent thrashing.

**Recovery of Thrashing :**

* Do not allow the system to go into thrashing by instructing the long-term scheduler not to bring the processes into memory after the threshold.
* If the system is already thrashing then instruct the mid-term scheduler to suspend some of the processes so that we can recover the system from thrashing.

Session 11:

# What is Deadlock in Operating System (OS)?

Every process needs some resources to complete its execution. However, the resource is granted in a sequential order.

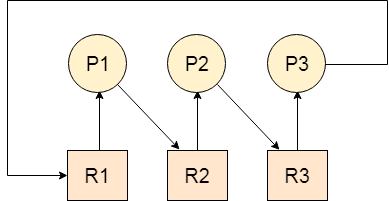
1. The process requests for some resource.
2. OS grant the resource if it is available otherwise let the process waits.
3. The process uses it and release on the completion.

A Deadlock is a situation where each of the computer process waits for a resource which is being assigned to some another process. In this situation, none of the process gets executed since the resource it needs, is held by some other process which is also waiting for some other resource to be released.

Let us assume that there are three processes P1, P2 and P3. There are three different resources R1, R2 and R3. R1 is assigned to P1, R2 is assigned to P2 and R3 is assigned to P3.

After some time, P1 demands for R1 which is being used by P2. P1 halts its execution since it can't complete without R2. P2 also demands for R3 which is being used by P3. P2 also stops its execution because it can't continue without R3. P3 also demands for R1 which is being used by P1 therefore P3 also stops its execution.

this scenario, a cycle is being formed among the three processes. None of the process is progressing and they are all waiting. The computer becomes unresponsive since all the processes got blocked.



### Difference between Starvation and Deadlock

|  |  |  |
| --- | --- | --- |
| **Sr.** | **Deadlock** | **Starvation** |
| 1 | Deadlock is a situation where no process got blocked and no process proceeds | Starvation is a situation where the low priority process got blocked and the high priority processes proceed. |
| 2 | Deadlock is an infinite waiting. | Starvation is a long waiting but not infinite. |
| 3 | Every Deadlock is always a starvation. | Every starvation need not be deadlock. |
| 4 | The requested resource is blocked by the other process. | The requested resource is continuously be used by the higher priority processes. |
| 5 | Deadlock happens when Mutual exclusion, hold and wait, No preemption and circular wait occurs simultaneously. | It occurs due to the uncontrolled priority and resource management. |

## Necessary conditions for Deadlocks

1. **Mutual Exclusion**

A resource can only be shared in mutually exclusive manner. It implies, if two process cannot use the same resource at the same time.

1. **Hold and Wait**

A process waits for some resources while holding another resource at the same time.

1. **No preemption**

The process which once scheduled will be executed till the completion. No other process can be scheduled by the scheduler meanwhile.

1. **Circular Wait**

All the processes must be waiting for the resources in a cyclic manner so that the last process is waiting for the resource which is being held by the first process.

# Strategies for handling Deadlock

## 1. Deadlock Ignorance

Deadlock Ignorance is the most widely used approach among all the mechanism. This is being used by many operating systems mainly for end user uses. In this approach, the Operating system assumes that deadlock never occurs. It simply ignores deadlock. This approach is best suitable for a single end user system where User uses the system only for browsing and all other normal stuff.

There is always a tradeoff between Correctness and performance. The operating systems like Windows and Linux mainly focus upon performance. However, the performance of the system decreases if it uses deadlock handling mechanism all the time if deadlock happens 1 out of 100 times then it is completely unnecessary to use the deadlock handling mechanism all the time.

In these types of systems, the user has to simply restart the computer in the case of deadlock. Windows and Linux are mainly using this approach.

## 2. Deadlock prevention

Deadlock happens only when Mutual Exclusion, hold and wait, No preemption and circular wait holds simultaneously. If it is possible to violate one of the four conditions at any time then the deadlock can never occur in the system.

The idea behind the approach is very simple that we have to fail one of the four conditions but there can be a big argument on its physical implementation in the system.

We will discuss it later in detail.

## 3. Deadlock avoidance

In deadlock avoidance, the operating system checks whether the system is in safe state or in unsafe state at every step which the operating system performs. The process continues until the system is in safe state. Once the system moves to unsafe state, the OS has to backtrack one step.

In simple words, The OS reviews each allocation so that the allocation doesn't cause the deadlock in the system.

We will discuss Deadlock avoidance later in detail.

## 4. Deadlock detection and recovery

This approach let the processes fall in deadlock and then periodically check whether deadlock occur in the system or not. If it occurs then it applies some of the recovery methods to the system to get rid of deadlock.

We will discuss deadlock detection and recovery later in more detail since it is a matter of discussion.

# Deadlock Prevention

If we simulate deadlock with a table which is standing on its four legs then we can also simulate four legs with the four conditions which when occurs simultaneously, cause the deadlock.

However, if we break one of the legs of the table then the table will fall definitely. The same happens with deadlock, if we can be able to violate one of the four necessary conditions and don't let them occur together then we can prevent the deadlock.

Let's see how we can prevent each of the conditions.

## 1. Mutual Exclusion

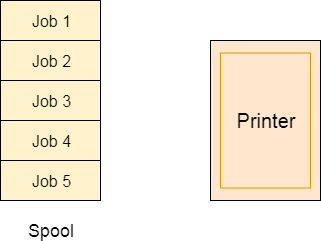
Mutual section from the resource point of view is the fact that a resource can never be used by more than one process simultaneously which is fair enough but that is the main reason behind the deadlock. If a resource could have been used by more than one process at the same time then the process would have never been waiting for any resource.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

However, if we can be able to violate resources behaving in the mutually exclusive manner then the deadlock can be prevented.

### Spooling

For a device like printer, spooling can work. There is a memory associated with the printer which stores jobs from each of the process into it. Later, Printer collects all the jobs and print each one of them according to FCFS. By using this mechanism, the process doesn't have to wait for the printer and it can continue whatever it was doing. Later, it collects the output when it is produced.



Although, Spooling can be an effective approach to violate mutual exclusion but it suffers from two kinds of problems.

1. This cannot be applied to every resource.
2. After some point of time, there may arise a race condition between the processes to get space in that spool.

We cannot force a resource to be used by more than one process at the same time since it will not be fair enough and some serious problems may arise in the performance. Therefore, we cannot violate mutual exclusion for a process practically.

## 2. Hold and Wait

Hold and wait condition lies when a process holds a resource and waiting for some other resource to complete its task. Deadlock occurs because there can be more than one process which are holding one resource and waiting for other in the cyclic order.

However, we have to find out some mechanism by which a process either doesn't hold any resource or doesn't wait. That means, a process must be assigned all the necessary resources before the execution starts. A process must not wait for any resource once the execution has been started.

**!(Hold and wait) = !hold or !wait (negation of hold and wait is, either you don't hold or you don't wait)**

This can be implemented practically if a process declares all the resources initially. However, this sounds very practical but can't be done in the computer system because a process can't determine necessary resources initially.

Process is the set of instructions which are executed by the CPU. Each of the instruction may demand multiple resources at the multiple times. The need cannot be fixed by the OS.

The problem with the approach is:

1. Practically not possible.
2. Possibility of getting starved will be increases due to the fact that some process may hold a resource for a very long time.

## 3. No Preemption

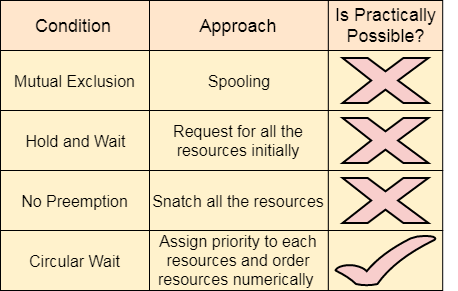
Deadlock arises due to the fact that a process can't be stopped once it starts. However, if we take the resource away from the process which is causing deadlock then we can prevent deadlock.

This is not a good approach at all since if we take a resource away which is being used by the process then all the work which it has done till now can become inconsistent.

Consider a printer is being used by any process. If we take the printer away from that process and assign it to some other process then all the data which has been printed can become inconsistent and ineffective and also the fact that the process can't start printing again from where it has left which causes performance inefficiency.

## 4. Circular Wait

To violate circular wait, we can assign a priority number to each of the resource. A process can't request for a lesser priority resource. This ensures that not a single process can request a resource which is being utilized by some other process and no cycle will be formed.



Among all the methods, violating Circular wait is the only approach that can be implemented practically.

session 12&13

# What is Inter Process Communication?

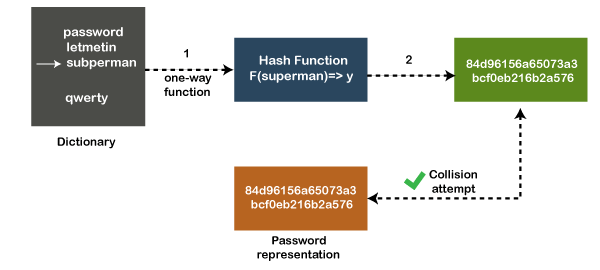
In general, Inter Process Communication is a type of mechanism usually provided by the operating system (or OS). The main aim or goal of this mechanism is to provide communications in between several processes. In short, the intercommunication allows a process letting another process know that some event has occurred.

Let us now look at the general definition of inter-process communication, which will explain the same thing that we have discussed above.

### Definition

"Inter-process communication is used for exchanging useful information between numerous threads in one or more processes (or programs)."

To understand inter process communication, you can consider the following given diagram that illustrates the importance of inter-process communication:



### Role of Synchronization in Inter Process Communication

It is one of the essential parts of inter process communication. Typically, this is provided by interprocess communication control mechanisms, but sometimes it can also be controlled by communication processes.

These are the following methods that used to provide the synchronization:

1. **Mutual Exclusion**
2. **Semaphore**
3. **Barrier**
4. **Spinlock**

**Mutual Exclusion:-**

It is generally required that only one process thread can enter the critical section at a time. This also helps in synchronization and creates a stable state to avoid the race condition.

**Semaphore:-**

Semaphore is a type of variable that usually controls the access to the shared resources by several processes. Semaphore is further divided into two types which are as follows:

1. Binary Semaphore
2. Counting Semaphore

**Barrier:-**

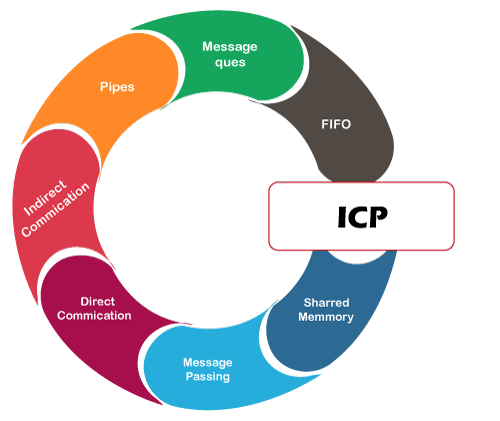
A barrier typically not allows an individual process to proceed unless all the processes does not reach it. It is used by many parallel languages, and collective routines impose barriers.

**Spinlock:-**

Spinlock is a type of lock as its name implies. The processes are trying to acquire the spinlock waits or stays in a loop while checking that the lock is available or not. It is known as busy waiting because even though the process active, the process does not perform any functional operation (or task).

### 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. **Direct Communication**
5. **Indirect communication**
6. **Message Passing**
7. **FIFO**

To understand them in more detail, we will discuss each of them individually.

**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.

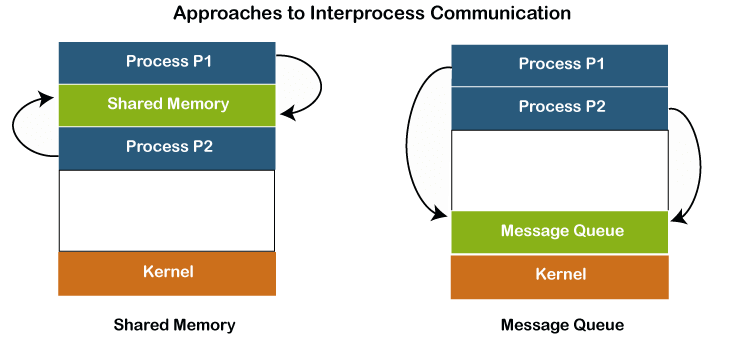
**Shared Memory:-**

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:



**Message Passing:-**

It is a type of mechanism that allows processes to synchronize and communicate with each other. However, by using the message passing, the processes can communicate with each other without restoring the hared variables.

Usually, the inter-process communication mechanism provides two operations that are as follows:

* send (message)
* received (message)

#### Note: The size of the message can be fixed or variable.

**Direct Communication:-**

In this type of communication process, usually, a link is created or established between two communicating processes. However, in every pair of communicating processes, only one link can exist.

**Indirect Communication**

Indirect communication can only exist or be established when processes share a common mailbox, and each pair of these processes shares multiple communication links. These shared links can be unidirectional or bi-directional.

**FIFO:-**

It is a type of general communication between two unrelated processes. It can also be considered as full-duplex, which means that one process can communicate with another process and vice versa.

### Some other different approaches

* **Socket:-**

It acts as a type of endpoint for receiving or sending the data in a network. It is correct for data sent between processes on the same computer or data sent between different computers on the same network. Hence, it used by several types of operating systems.

* **File:-**

A file is a type of data record or a document stored on the disk and can be acquired on demand by the file server. Another most important thing is that several processes can access that file as required or needed.

* **Signal:-**

As its name implies, they are a type of signal used in inter process communication in a minimal way. Typically, they are the massages of systems that are sent by one process to another. Therefore, they are not used for sending data but for remote commands between multiple processes.

Usually, they are not used to send the data but to remote commands in between several processes.

### Why we need interprocess communication?

There are numerous reasons to use inter-process communication for sharing the data. Here are some of the most important reasons that are given below:

* It helps to speedup modularity
* Computational
* Privilege separation
* Convenience
* Helps operating system to communicate with each other and synchronize their actions as well.