# OPERATING SYSTEM: Fundamental OS Concepts

## 2017 - 2018

## MCS

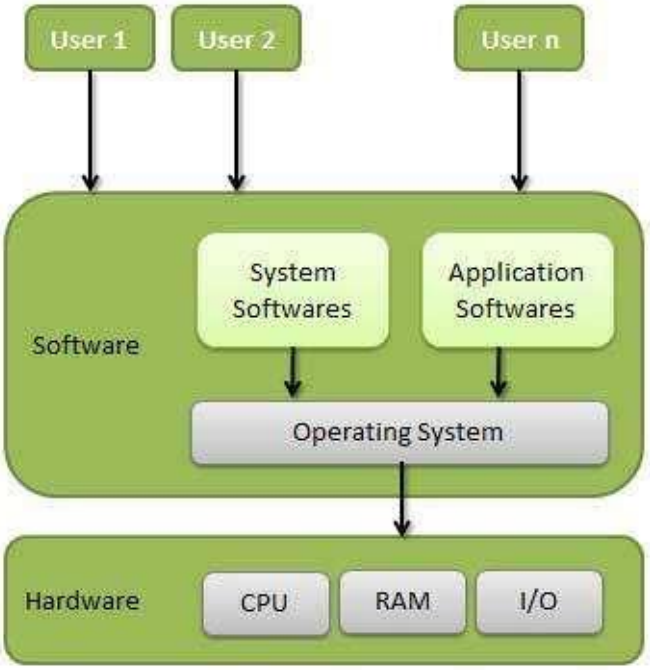
# 1. OPERATING SYSTEM - Overview

An Operating System (OS) is an interface between a computer user and computer hardware. An operating system is a software which performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers.

Some popular Operating Systems include Linux Operating System, Windows Operating System, VMS, OS/400, AIX, z/OS, etc.

## Definition

An operating system is a program that acts as an interface between the user and the computer hardware and controls the execution of all kinds of programs.




Following are some of important functions of an operating System.

- Memory Management
- Processor Management
- Device Management
- File Management
- Security
- Control over system performance

- Job accounting
- Error detecting aids
- Coordination between other software and users

## Memory Management

Memory management refers to management of Primary Memory or Main Memory. Main memory is a large array of words or bytes where each word or byte has its own address.

Main memory provides a fast storage that can be accessed directly by the CPU. For a program to be executed, it must be in the main memory. An Operating System does the following activities for memory management:

- Keeps tracks of primary memory, i.e., what part of it are in use by whom, what part are not in use.
- In multiprogramming, the OS decides which process will get memory when and how much.
- Allocates the memory when a process requests it to do so.
- De-allocates the memory when a process no longer needs it or has been terminated.

## Processor Management

In multiprogramming environment, the OS decides which process gets the processor when and for how much time. This function is called **process scheduling**. An Operating System does the following activities for processor management:

- Keeps tracks of processor and status of process. The program responsible for this task is known as **traffic controller**.
- Allocates the processor (CPU) to a process.
- De-allocates processor when a process is no longer required.

## Device Management

An Operating System manages device communication via their respective drivers. It does the following activities for device management:

- Keeps tracks of all devices. The program responsible for this task is known as the **I/O controller**.

- Decides which process gets the device when and for how much time.
- Allocates the device in the most efficient way.
- De-allocates devices.

## File Management

In computing, a **file system** or **filesystem** is used to control how data is stored and retrieved. Without a file system, information placed in a storage medium would be one large body of data with no way to tell where one piece of information stops and the next begins. By separating the data into pieces and giving each piece a name, the information is easily isolated and identified. Taking its name from the way paper-based information systems are named, each group of data is called a "file". The structure and logic rules used to manage the groups of information and their names is called a "file system".

A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions.

An Operating System does the following activities for file management:

- Keeps track of information, location, uses, status etc. The collective facilities are often known as **file system**.
- Decides who gets the resources.
- Allocates the resources.
- De-allocates the resources.

## Other Important Activities

Following are some of the important activities that an Operating System performs:

- **Security** -- By means of password and similar other techniques, it prevents unauthorized access to programs and data.
- **Control over system performance** -- Recording delays between request for a service and response from the system.
- **Job accounting** -- Keeping track of time and resources used by various jobs and users.
- **Error detecting aids** -- Production of dumps, traces, error messages, and other debugging and error detecting aids.
- **Coordination between other software and users** -- Coordination and assignment of compilers, interpreters, assemblers and other software to the various

users of the computer systems.

# 2. OPERATING SYSTEM – Types

Operating systems are there from the very first computer generation and they keep evolving with time. In this chapter, we will discuss some of the important types of operating systems which are most commonly used.

## Batch Operating System

The users of a batch operating system do not interact with the computer directly. Each user prepares his job on an off-line device like punch cards and submits it to the computer operator. To speed up processing, jobs with similar needs are batched together and run as a group. The programmers leave their programs with the operator and the operator then sorts the programs with similar requirements into batches.

The problems with Batch Systems are as follows:

- Lack of interaction between the user and the job.
- CPU is often idle, because the speed of the mechanical I/O devices is slower than the CPU.
- Difficult to provide the desired priority.

## Time-sharing Operating Systems

Time-sharing is a technique which enables many people, located at various terminals, to use a particular computer system at the same time. Time-sharing or multitasking is a logical extension of multiprogramming. Processor's time which is shared among multiple users simultaneously is termed as time-sharing.

The main difference between Multiprogrammed Batch Systems and Time-Sharing Systems is that in case of Multiprogrammed batch systems, the objective is to maximize processor use, whereas in Time-Sharing Systems, the objective is to minimize response time.

Multiple jobs are executed by the CPU by switching between them, but the switches occur so frequently. Thus, the user can receive an immediate response. For example, in a transaction processing, the processor executes each user program in a short burst or quantum of computation. That is, if **n** users are present, then each user can get a time quantum. When the user submits the command, the response time is in few seconds at most.

The operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time. Computer systems that were designed primarily as batch systems have been modified to time-sharing systems.

Advantages of Timesharing operating systems are as follows:

- Provides the advantage of quick response
- Avoids duplication of software
- Reduces CPU idle time

Disadvantages of Time-sharing operating systems are as follows:

- Problem of reliability
- Question of security and integrity of user programs and data
- Problem of data communication

## Distributed Operating System

Distributed systems use multiple central processors to serve multiple real-time applications and multiple users. Data processing jobs are distributed among the processors accordingly.

- A *distributed system is a collection of processors that* do not share memory or a clock. Each processor has its own local memory and clock.
- The processors in the system are connected through a communication network.
- A distributed system provides user access to various system resources.

The processors communicate with one another through various communication lines (such as high-speed buses or telephone lines). These are referred as **loosely coupled systems** or distributed systems. Processors in a distributed system may vary in size and function. These processors are referred as sites, nodes, computers, and so on.

The advantages of distributed systems are as follows:

- With resource sharing facility, a user at one site may be able to use the resources available at another.
- Speedup the exchange of data with one another via electronic mail.
- If one site fails in a distributed system, the remaining sites can potentially continue operating.
- Better service to the customers.
- Reduction of the load on the host computer.
- Reduction of delays in data processing.

## Network Operating System

A Network Operating System runs on a server and provides the server the capability to manage data, users, groups, security, applications, and other networking functions. The primary purpose of the network operating system is to allow shared file and printer access among multiple computers in a network, typically a local area network (LAN), a private network or to other networks.

Examples of network operating systems include Microsoft Windows Server 2003, Microsoft Windows Server 2008, UNIX, Linux, Mac OS X, Novell NetWare, and BSD.

The advantages of network operating systems are as follows:

- Centralized servers are highly stable.
- Security is server managed.
- Upgrades to new technologies and hardware can be easily integrated into the system.
- Remote access to servers is possible from different locations and types of systems.

The disadvantages of network operating systems are as follows:

- High cost of buying and running a server.
- Dependency on a central location for most operations.
- Regular maintenance and updates are required.

## Real-Time Operating System

A real-time system is defined as a data processing system in which the time interval required to process and respond to inputs is so small that it controls the environment. The time taken by the system to respond to an input and display of required updated information is termed as the **response time**. So in this method, the response time is very less as compared to online processing.

Real-time systems are used when there are rigid time requirements on the operation of a processor or the flow of data and real-time systems can be used as a control device in a dedicated application. A real-time operating system must have well-defined, fixed time constraints, otherwise the system will fail. For example, Scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots, air traffic control systems, etc.

There are two types of real-time operating systems.

**Hard real-time systems**

Hard real-time systems guarantee that critical tasks complete on time. In hard real-time systems, secondary storage is limited or missing and the data is stored in ROM. In these systems, virtual memory is almost never found.

**Soft real-time systems**

Soft real-time systems are less restrictive. A critical real-time task gets priority over other tasks and retains the priority until it completes. Soft real-time systems have limited utility than hard real-time systems. For example, multimedia, virtual reality, Advanced Scientific Projects like undersea exploration and planetary rovers, etc.

# 3. OPERATING SYSTEM – Services

An Operating System provides services to both the users and to the programs.

- It provides programs an environment to execute.
- It provides users the services to execute the programs in a convenient manner.

Following are a few common services provided by an operating system:

- Program execution
- I/O operations
- File System manipulation
- Communication
- Error Detection
- Resource Allocation
- Protection

## Program Execution

Operating systems handle many kinds of activities from user programs to system programs like printer spooler, name servers, file server, etc. Each of these activities is encapsulated as a process.

A process includes the complete execution context (code to execute, data to manipulate, registers, OS resources in use). Following are the major activities of an operating system with respect to program management:

- Loads a program into memory
- Executes the program
- Handles program's execution

- Provides a mechanism for process synchronization
- Provides a mechanism for process communication
- Provides a mechanism for deadlock handling

Deadlock: A process requests resources; if the resources are not available at that time, the process enters a wait state. It may happen that waiting processes will never again change state, because the resources they have requested are held by other waiting processes. This situation is called deadlock.

## I/O Operation

An I/O subsystem comprises of I/O devices and their corresponding driver software. Drivers hide the peculiarities(peculiarities(the quality of being strange or unfamiliar)) of specific hardware devices from the users.

An Operating System manages the communication between user and device drivers.

- I/O operation means read or write operation with any file or any specific I/O device.
- Operating system provides the access to the required I/O device when required.

## File System Manipulation

A file represents a collection of related information. Computers can store files on the disk (secondary storage), for long-term storage purpose. Examples of storage media include magnetic tape, magnetic disk and optical disk drives like CD, DVD. Each of these media has its own properties like speed, capacity, data transfer rate and data access methods.

A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions. Following are the major activities of an operating system with respect to file management:

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

## Communication

In case of distributed systems which are a collection of processors that do not share memory,

peripheral devices, or a clock, the operating system manages communications between all the processes. Multiple processes communicate with one another through communication lines in the network.

The OS handles routing and connection strategies, and the problems of contention and security. Following are the major activities of an operating system with respect to communication:

- Two processes often require data to be transferred between them.
- Both the processes can be on one computer or on different computers, but are connected through a computer network.
- Communication may be implemented by two methods, either by Shared Memory or by Message Passing.

## Error Handling

Errors can occur anytime and anywhere. An error may occur in CPU, in I/O devices or in the memory hardware. Following are the major activities of an operating system with respect to error handling:

- The OS constantly checks for possible errors.
- The OS takes an appropriate action to ensure correct and consistent computing.

## Resource Management

In case of multi-user or multi-tasking environment, resources such as main memory, CPU cycles and files storage are to be allocated to each user or job. Following are the major activities of an operating system with respect to resource management:

- The OS manages all kinds of resources using schedulers.
- CPU scheduling algorithms are used for better utilization of CPU.

## Protection

Considering a computer system having multiple users and concurrent execution of multiple processes, the various processes must be protected from each other's activities.

Protection refers to a mechanism or a way to control the access of programs, processes, or users to the resources defined by a computer system. Following are the major activities of an operating system with respect to protection:

- The OS ensures that all access to system resources is controlled.

- The OS ensures that external I/O devices are protected from invalid access attempts.
- The OS provides authentication features for each user by means of passwords.

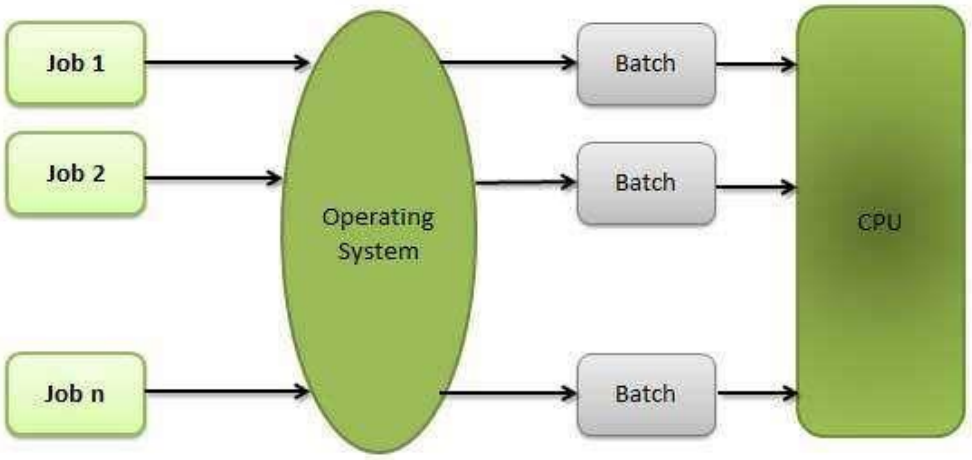
# 4. OPERATING SYSTEM – Properties

Following are few of very important tasks that Operating System handles

## Batch Processing

Batch processing is a technique in which an Operating System collects the programs and data together in a batch (a group of things or people dealt with at the same time or considered similar in type ) before processing starts. An operating system does the following activities related to batch processing:

- The OS defines a job which has predefined sequence of commands, programs and data as a single unit.
- The OS keeps a number of jobs in memory and executes them without any manual information.
- Jobs are processed in the order of submission, i.e., first come first served fashion.
- When a job completes its execution, its memory is released and the output for the job gets copied into an output spool (a tube-shaped object with top and bottom edges that stick out and around which a length of thread, wire, film, etc. is wrapped in order to store it *a spool **of** cotton/film* ) for later printing or processing.




**Advantages**

- Batch processing takes much of the work of the operator to the computer.
- Increased performance as a new job get started as soon as the previous job is finished, without any manual intervention.

**Disadvantages**

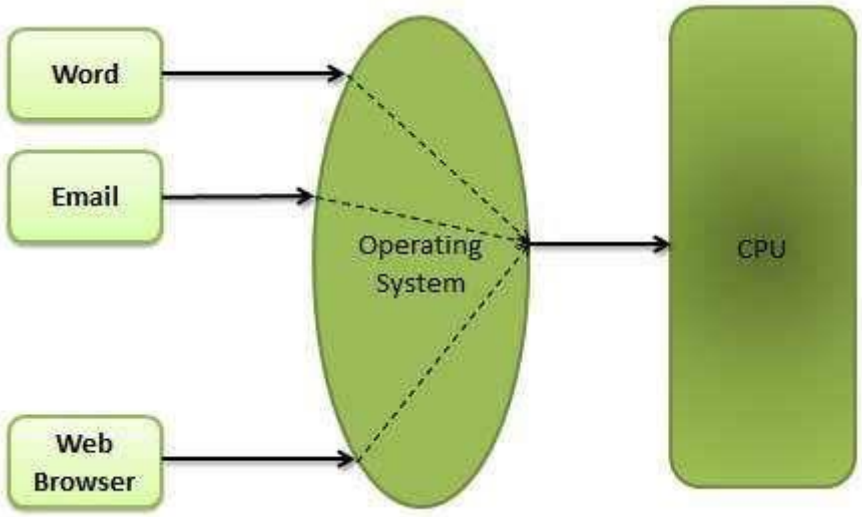
- Difficult to debug programs.
- A job could enter an infinite loop.
- Due to lack of protection scheme, one batch job can affect other pending jobs.

## Multitasking

Multitasking is when multiple jobs are executed by the CPU simultaneously by switching between them. Switches occur so frequently that the users may interact with each program while it is running. An OS does the following activities related to multitasking:

- The user gives instructions to the operating system or to a program directly, and receives an immediate response.
- The OS handles multitasking in the way that it can handle multiple operations / executes multiple programs at a time.
- Multitasking Operating Systems are also known as Time-sharing systems.
- These Operating Systems were developed to provide interactive use of a computer system at a reasonable cost.
- A time-shared operating system uses the concept of CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared CPU.

- Each user has at least one separate program in memory.

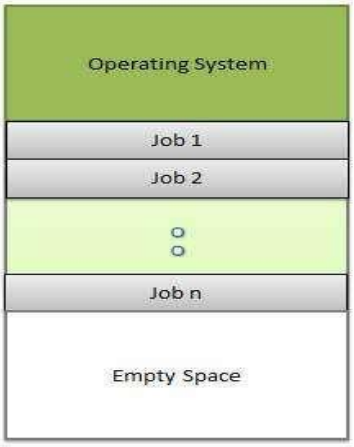



- A program that is loaded into memory and is executing is commonly referred to as a **process**.
- When a process executes, it typically executes for only a very short time before it either finishes or needs to perform I/O.
- Since interactive I/O typically runs at slower speeds, it may take a long time to complete. During this time, a CPU can be utilized by another process.
- The operating system allows the users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user.
- As the system switches CPU rapidly from one user/program to the next, each user is given the impression that he/she has his/her own CPU, whereas actually one CPU is being shared among many users.

## Multiprogramming

Sharing the processor, when two or more programs reside in memory at the same time, is referred as **multiprogramming**. Multiprogramming assumes a single shared processor. Multiprogramming increases CPU utilization by organizing jobs so that the CPU always has one to execute.

The following figure shows the memory layout for a multiprogramming system.




An OS does the following activities related to multiprogramming.

- The operating system keeps several jobs in memory at a time.
- This set of jobs is a subset of the jobs kept in the job pool.
- The operating system picks and begins to execute one of the jobs in the memory.
- Multiprogramming operating systems monitor the state of all active programs and system resources using memory management programs to ensures that the CPU is never idle, unless there are no jobs to process.

**Advantage**

- High and efficient CPU utilization.
- User feels that many programs are allotted CPU almost simultaneously.

**Disadvantages**

- CPU scheduling is required.
- To accommodate many jobs in memory, memory management is required.

## Interactivity

Interactivity refers to the ability of users to interact with a computer system. An Operating system does the following activities related to interactivity:

- Provides the user an interface to interact with the system.
- Manages input devices to take inputs from the user. For example, keyboard.
- Manages output devices to show outputs to the user. For example, Monitor.

The response time of the OS needs to be short, since the user submits and waits for the result.

## Real-Time Systems

Real-time systems are usually dedicated, embedded systems. An operating system does the following activities related to real-time system activity.

- In such systems, Operating Systems typically read from and react to sensor data.
- The Operating system must guarantee response to events within fixed periods of time to ensure correct performance.

## Distributed Environment

A distributed environment refers to multiple independent CPUs or processors in a computer system. An operating system does the following activities related to distributed environment:

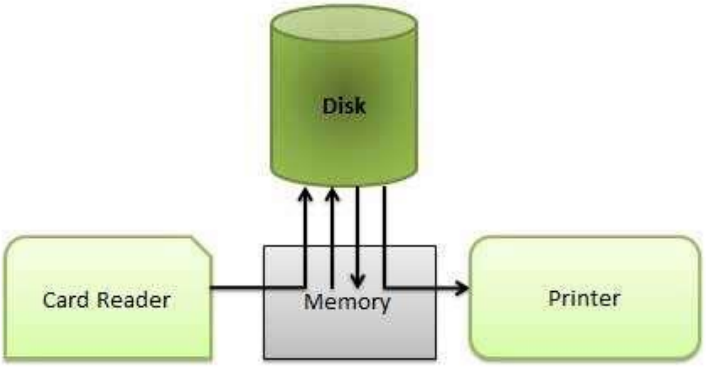
- The OS distributes computation logics among several physical processors.
- The processors do not share memory or a clock. Instead, each processor has its own local memory.
- The OS manages the communications between the processors. They communicate with each other through various communication lines.

## Spooling

Spooling is an acronym for simultaneous peripheral operations on line. Spooling refers to putting data of various I/O jobs in a buffer. This buffer is a special area in memory or hard disk which is accessible to I/O devices.

An operating system does the following activities related to distributed environment:

- Handles I/O device data spooling as devices have different data access rates.
- Maintains the spooling buffer which provides a waiting station where data can rest while the slower device catches up.
- Maintains parallel computation because of spooling process as a computer can perform I/O in parallel fashion. It becomes possible to have the computer read data from a tape, write data to disk and to write out to a tape printer while it is doing its computing task.




**Advantages**

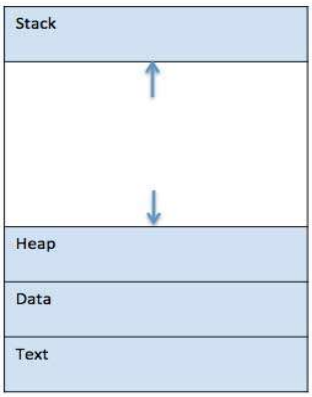
- The spooling operation uses a disk as a very large buffer.
- Spooling is capable of overlapping I/O operation for one job with processor operations for another job.

# 5. OPERATING SYSTEM – Processes

## Process

A process is basically a program in execution. The execution of a process must progress in a sequential fashion.
A process is defined as an entity which represents the basic unit of work to be implemented in the system.

To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program.
When a program is loaded into the memory and it becomes a process, it can be divided into four sections ─ stack, heap, text and data. The following image shows a simplified layout of a process inside main memory:



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

## Program

A program is a piece of code which may be a single line or millions of lines. A computer program is usually written by a computer programmer in a programming language. For example, here is a simple program written in C programming language:

```
#include <stdio.h>

int main() {

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

    return 0;

}
```

A computer program is a collection of instructions that performs a specific task when executed by a computer. When we compare a program with a process, we can conclude that a process is a dynamic instance of a computer program.

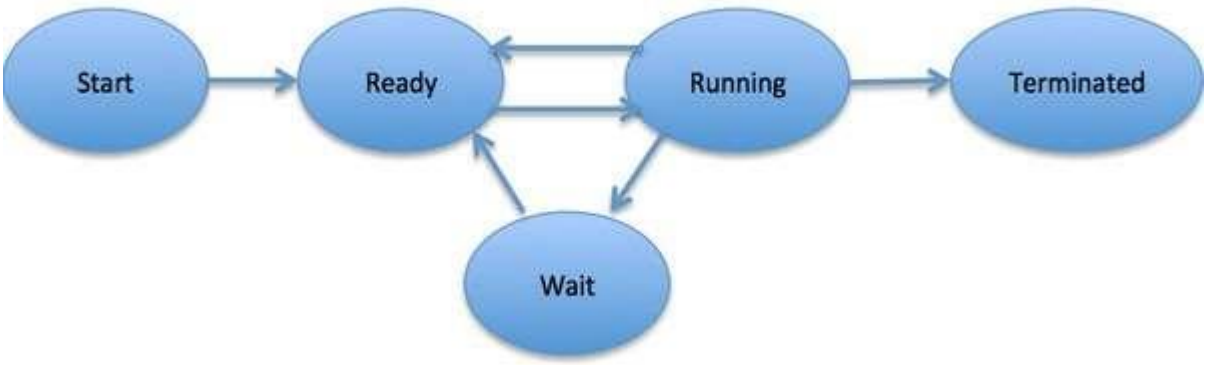
A part of a computer program that performs a well-defined task is known as an **algorithm**. A collection of computer programs, libraries and related data are referred to as a **software**.

## Process Life Cycle

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

In general, a process can have one of the following five states at a time.

| S.N | State & Description |
|---|---|
| 1 | **Start**<br>This is the initial state when a process is first started/created. |
| 2 | **Ready**<br>The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after **Start** state or while running it by but interrupted by the scheduler to assign CPU to some other process. |
| 3 | **Running**<br>Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions. |
| 4 | **Waiting**<br>Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available. |
| 5 | **Terminated or Exit**<br>Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory. |



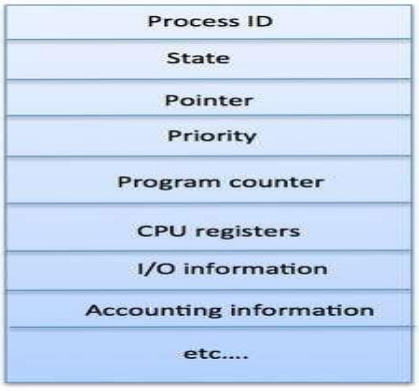

## Process Control Block (PCB)

A Process Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID). A PCB keeps all the information needed to keep track of a process as listed below in the table:

| S.N. | Information & Description |
|---|---|

| | |
|---|---|
| 1 | **Process State**<br>The current state of the process i.e., whether it is ready, running, waiting, or whatever. |
| 2 | **Process privileges**<br>This is required to allow/disallow access to system resources. |
| 3 | **Process ID**<br>Unique identification for each of the process in the operating system. |
| 4 | **Pointer**<br>A pointer to parent process. |
| 5 | **Program Counter**<br>Program Counter is a pointer to the address of the next instruction to be executed for this process. |
| 6 | **CPU registers**<br>Various CPU registers where process need to be stored for execution for running state. |
| 7 | **CPU Scheduling Information**<br>Process priority and other scheduling information which is required to schedule the process. |
| 8 | **Memory management information**<br>This includes the information of page table, memory limits, Segment table depending on memory used by the operating system. |
| 9 | **Accounting information**<br>This includes the amount of CPU used for process execution, time limits, execution ID etc. |
| 10 | **IO status information**<br>This includes a list of I/O devices allocated to the process. |

The architecture of a PCB is completely dependent on Operating System and may contain different information in different operating systems. Here is a simplified diagram of a PCB:



The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates.

# 6. OPERATING SYSTEM – Process Scheduling

## Definition

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

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

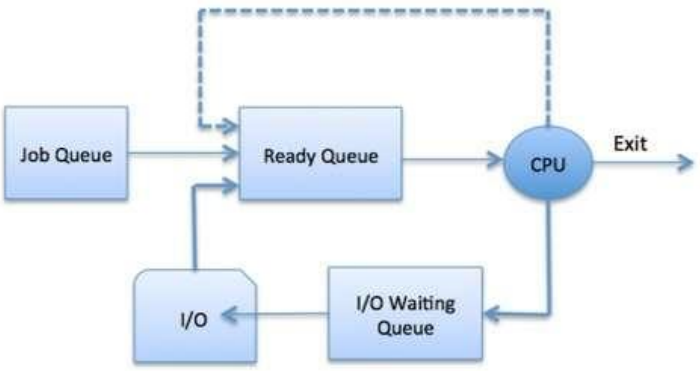
## Process Scheduling Queues

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

The Operating System maintains the following important process scheduling queues:

- **Job queue** - Set of all processes in the system

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



The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.). The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system; in the above diagram, it has been merged with the CPU.

## Two-State Process Model

Two-state process model refers to running and non-running states which are described below.

| S.N. | State & Description |
|---|---|
| 1 | **Running**<br>When a new process is created, it enters into the system as in the running state. |
| 2 | **Not Running**<br>Processes that are not running are kept in queue, waiting for their turn to execute. Each entry in the queue is a pointer to a particular process. Queue is implemented by using linked list. Use of dispatcher is as follows. When a process is interrupted, that process is transferred in the waiting queue. If the process has completed or aborted, the process is discarded. In either case, the dispatcher then selects a process from the queue to execute. |

## Schedulers

Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types:

- Long-Term Scheduler
- Short-Term Scheduler
- Medium-Term Scheduler

## Long-Term Scheduler

It is also called a **job scheduler**. A long-term scheduler determines which programs are admitted to the system for processing. It selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling.

The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

On some systems, the long-term scheduler may not be available or minimal. Time-sharing operating systems have no long term scheduler. When a process changes the state from new to ready, then there is use of long-term scheduler.

## Short-Term Scheduler

It is also called as **CPU scheduler**. Its main objective is to increase system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them.

Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers.

## Medium-Term Scheduler

Medium-term scheduling is a part of **swapping**. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium-term scheduler is in- charge of handling the swapped out-processes.

A running process may become suspended if it makes an I/O request. A suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other processes, the suspended process is moved to the secondary storage. This process is called **swapping**, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

## Comparison among Schedulers

| S.N. | Long-Term Scheduler | Short-Term Scheduler | Medium-term Schedule |
|---|---|---|---|
| 1 | It is a job scheduler | It is a CPU scheduler | It is a process swapping scheduler. |
| 2 | Speed is lesser than short term scheduler | Speed is fastest among other two | Speed is in between both short and long term scheduler. |
| 3 | It controls the degree of multiprogramming | It provides lesser control over degree of multiprogramming | It reduces the degree of multiprogramming. |

| | | | |
|---|---|---|---|
| 4 | It is almost absent or minimal in time sharing system | It is also minimal in time sharing system | It is a part of Time sharing systems. |
| 5 | It selects processes from pool and loads them into memory for execution | It selects those processes which are ready to execute | It can re-introduce the process into memory and execution can be continued. |

## Context Switch

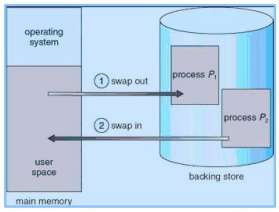
A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.

When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.

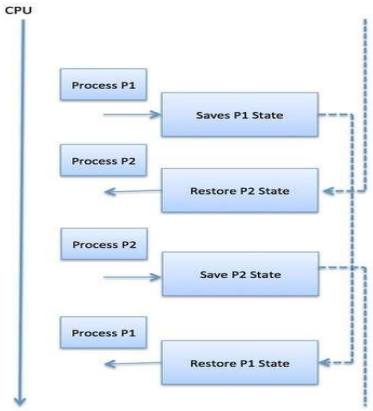
## Swapping

A process, can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution. Assume a multiprogramming environment with a round

robin CPU-scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished, and to swap in another process to the memory space that has been freed (Fig 6.2). When each process finishes its quantum, it will be swapped with another process.




.2 Swapping of two processes using a disk as a blocking store




Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers. When the process is switched, the following information is stored for later use.

- Program Counter
- Scheduling information
- Base and limit register value
- Currently used register
- Changed State
- I/O State information
- Accounting information

## 7. Operating System – Scheduling Algorithms

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

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

These algorithms are either **non-preemptive** or **preemptive**. Non-preemptive algorithms are designed so that once a process enters the running state; it cannot be preempted until it completes its allotted time, whereas the preemptive scheduling is based on priority where a

scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

## First Come, First Served (FCFS)

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

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


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

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

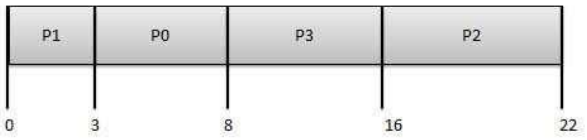
Average Wait Time: (0+4+6+13) / 4 = 5.75

## Shortest Job Next (SJN)

- This is also known as **shortest job first**, or SJF.
- This is a non-preemptive 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 the required CPU time is not known.

- The processer should know in advance how much time a process will take.

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




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

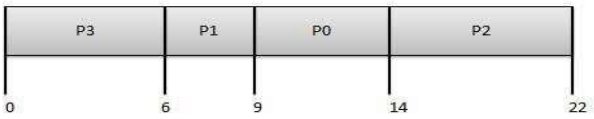
| Process | Wait Time : Service Time - Arrival Time |
|---|---|
| P0 | 3 - 0 = 3 |
| P1 | 0 - 0 = 0 |
| P2 | 16 - 2 = 14 |
| P3 | 8 - 3 = 5 |

Average Wait Time: (3+0+14+5) / 4 = 5.50

## 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.

| Process | Arrival Time | Execute Time | Priority | Service Time |
|---|---|---|---|---|
| P0 | 0 | 5 | 1 | 9 |
| P1 | 1 | 3 | 2 | 6 |
| P2 | 2 | 8 | 1 | 14 |
| P3 | 3 | 6 | 3 | 0 |



Wait time of each process is as follows:

| Process | Wait Time : Service Time - Arrival Time |
|---|---|
| P0 | 9 - 0 = 9 |
| P1 | 6 - 1 = 5 |
| P2 | 14 - 2 = 12 |
| P3 | 0 - 0 = 0 |

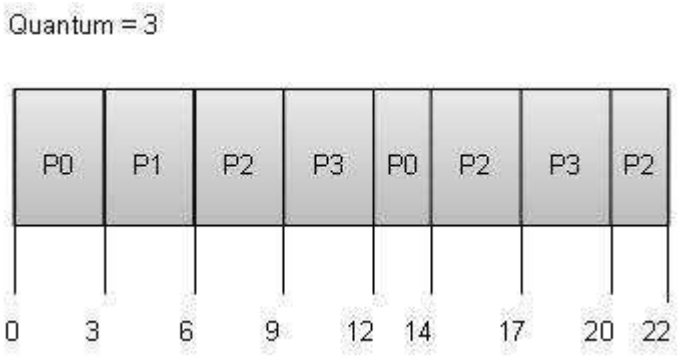
Average Wait Time: (9+5+12+0) / 4 = 6.5

## 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 be given preference.

## Round Robin Scheduling

- Round Robin is a 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.

## 8. Operating System – Multithreading

### Introduction

Multithreading is the ability of a central processing unit (CPU) or a single core in a multi-core processor to execute multiple processes or threads concurrently, appropriately supported by the operating system.

### What is a Thread?

A thread is the smallest unit of processing that can be performed in an OS. In most modern operating systems, a thread exists within a process - that is, a single process may contain multiple threads.

A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.

A thread shares with its peer threads few information like code segment, data segment and

open files. When one thread alters a code segment memory item, all other threads see that.

In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its executable code and the values of its variables at any given time.

A thread is also called a **lightweight process**. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. Threads have been successfully used in implementing network servers and web server. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors.

For example, in a browser, multiple tabs can be different threads. MS word uses multiple threads, one thread to format the text, other thread to process inputs etc.

## Process vs Thread?

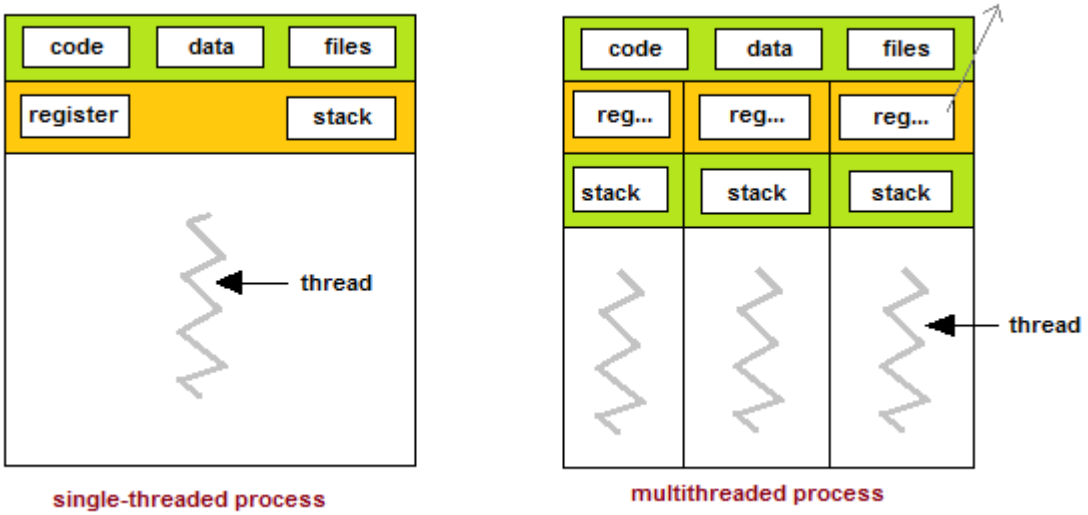
The typical difference is that threads within the same process run in a shared memory space, while processes run in separate memory spaces.

Threads are not independent of one other like processes as a result threads shares 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), a register set, and a stack space.

### What are Threads?

Thread is an execution unit which consists of its own program counter, a stack, and a set of registers. Threads are also known as Lightweight processes. Threads are popular way to improve application through parallelism. The CPU switches rapidly back and forth among the threads giving illusion that the threads are running in parallel.

As each thread has its own independent resource for process execution, multpile processes can be executed parallely by increasing number of threads.




## Difference between Process and Thread

| S.N. | Process | Thread |
|---|---|---|
| 1 | Process is heavy weight or resource intensive. | Thread is lightweight, taking lesser resources than a process. |
| 2 | Process switching needs interaction with operating system. | Thread switching does not need to interact with operating system. |
| 3 | In multiple processing environments, each process executes the same code but has its own memory and file resources. | All threads can share same set of open files, child processes. |
| 4 | If one process is blocked, then no other process can execute until the first process is unblocked. | While one thread is blocked and waiting, a second thread in the same task can run. |
| 5 | Multiple processes without using threads use more resources. | Multiple threaded processes use fewer resources. |
| 6 | In multiple processes each process operates independently of the others. | One thread can read, write or change another thread's data. |

## Advantages of Thread

- Threads minimize the context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.

- It is more economical to create and context switch threads.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

***Advantages of Thread over Process***

*1. Responsiveness:* If the process is divided into multiple threads, if one thread completed its execution, then its output can be immediately responded.

*2. Faster context switch:* Context switch time between threads is less compared to process context switch. Process context switch is more overhead for 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 file can be shared among all threads within a process.
Note : stack and registers can't be shared among the threads. Each thread have its own stack and registers.

*5. Communication:* Communication between multiple thread is easier as thread 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 process is divided into multiple threads and each thread function is considered as one job, then the number of jobs completed per unit time is increased. Thus, increasing the throughput of the system.

## Types of Thread

Threads are implemented in following two ways:

- **User Level Threads** -- User managed threads
- **Kernel Level Threads** -- Operating System managed threads acting on kernel, an operating system core.

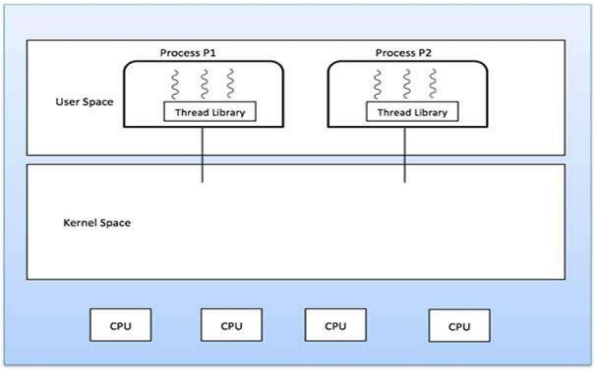
**User threads**, are above the kernel and without kernel support. These are the threads that application programmers use in their programs.

**Kernel threads** are supported within the kernel of the OS itself. All modern OSs support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.

## User Level Threads

As we indicated earlier, user level threads are managed by a user level library however, they still require a kernel system call to operate. It does not mean that the kernel knows anything about thread management. Not at all, It only takes care of the execution part. The lack of cooperation between user level threads and the kernel is a known disadvantage. In this case, the kernel may not favor a process that has many threads. User level threads are typically fast. Creating threads, switching between threads and synchronizing threads only needs a procedure call. They are a good choice for non blocking tasks otherwise the entire process will block if any of the threads blocks.

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




**Advantages**

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

**Disadvantages**

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

## Kernel Level Threads

Kernel level threads are managed by the OS, therefore, thread operations (ex. Scheduling) are implemented in the kernel code. This means kernel level threads may favor thread heavy processes. Moreover, they can also utilize multiprocessor systems by splitting threads on different processors or cores. They are a good choice for processes that block frequently. If one thread blocks it does not cause the entire process to block. Kernel level threads have disadvantages as well. They are slower than user level threads due to the management overhead. Kernel level context switch involves more steps than just saving some registers. Finally, they are not portable because the implementation is operating system dependent.

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

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

**Advantages**

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

**Disadvantages**

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

## Multithreading Models

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

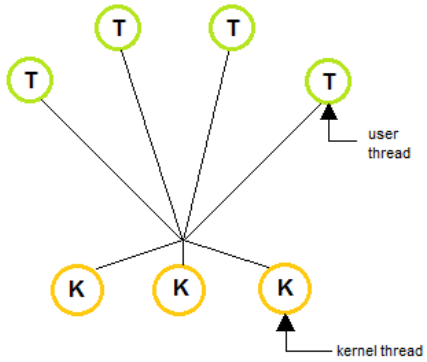
- Many-to-many relationship
- Many-to-one relationship
- One-to-one relationship

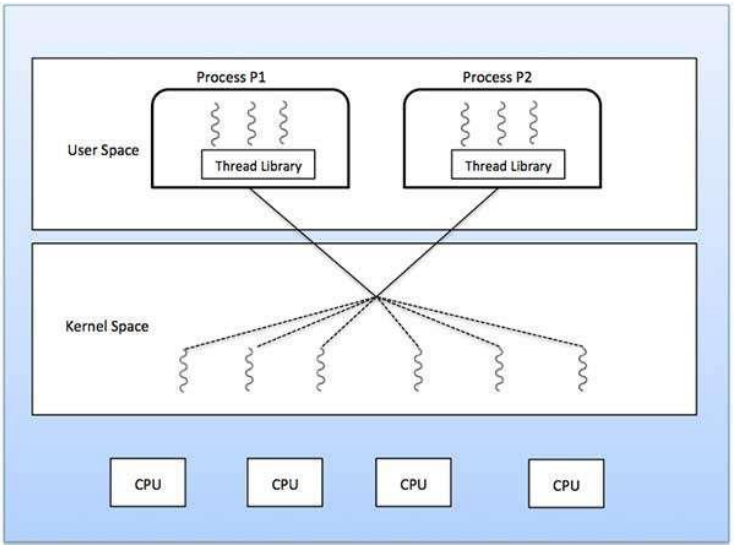
## Many-to-Many Model

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

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

- The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.
- Users can create any number of the threads.
- Blocking the kernel system calls does not block the entire process.
- Processes can be split across multiple processors.

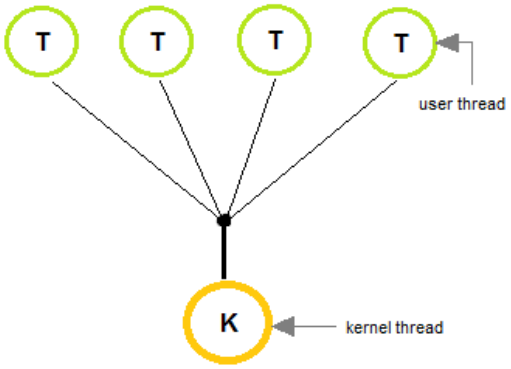


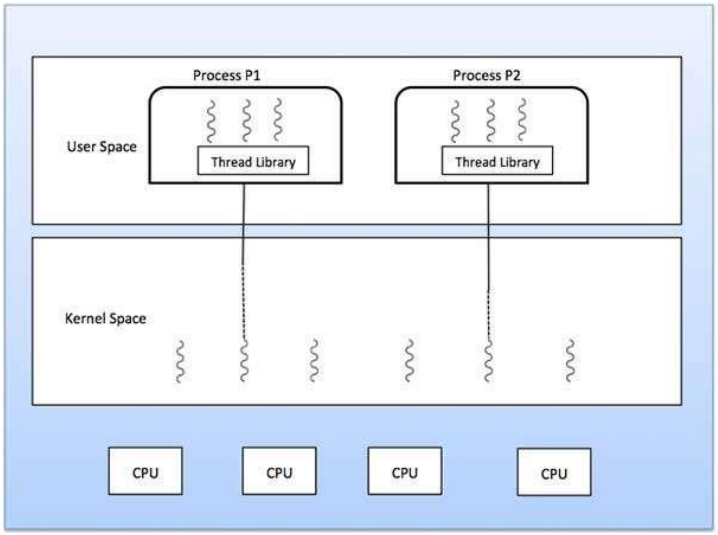



## Many-to-One Model

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

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



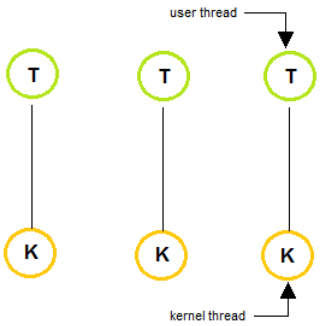


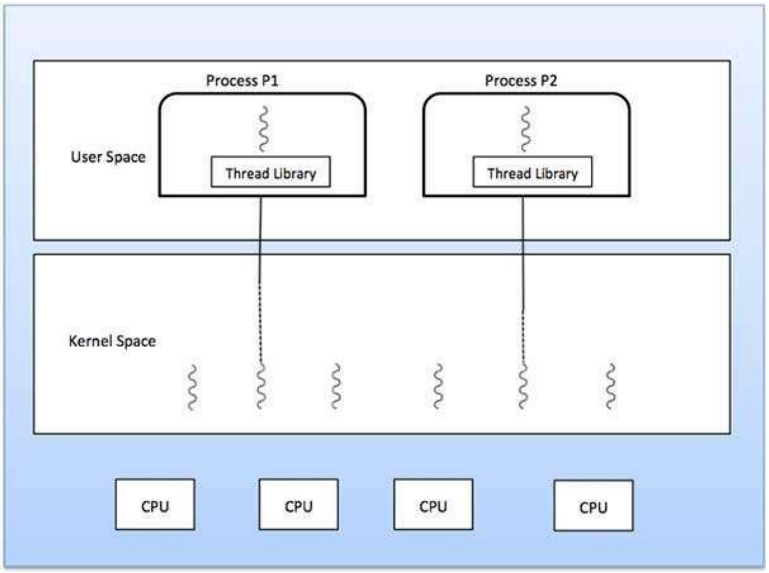

## One-to-One Model

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

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

Linux and Windows from 95 to XP implement the one-to-one model for threads.






## Difference between User-Level & Kernel-Level Thread

| S.N. | User-Level Threads | Kernel-Level Thread |
|---|---|---|
| 1 | User-level threads are faster to create and manage. | Kernel-level threads are slower to create and manage. |
| 2 | Implementation is by a thread library at the user level. | Operating system supports creation of Kernel threads. |
| 3 | User-level thread is generic and can run on any operating system. | Kernel-level thread is specific to the operating system. |
| 4 | Multi-threaded applications cannot take advantage of multiprocessing. | Kernel routines themselves can be multithreaded. |

## Thread Libraries

Thread libraries provides programmers with API for creating and managing of threads.

Thread libraries may be implemented either in user space or in kernel space. The user space involves API functions implemented solely within user space, with no kernel support. The kernel space involves system calls, and requires a kernel with thread library support.

**There are three types of thread :**

- POSIX Pitheads, may be provided as either a user or kernel library, as an extension to the POSIX standard.
- Win32 threads, are provided as a kernel-level library on Windows systems.

- Java threads - Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either Pitheads or Win32 threads depending on the system

**Benefits of Multithreading**

1. Responsiveness
2. Resource sharing, hence allowing better utilization of resources.
3. Economy. Creating and managing threads becomes easier.
4. Scalability. One thread runs on one CPU. In Multithreaded processes, threads can be distributed over a series of processors to scale.

Context Switching is smooth. Context switching refers to the procedure followed by CPU to change from one task to another.

## Multithreading Issues

1. **Thread Cancellation**.

   Thread cancellation means terminating a thread before it has finished working. There can be two approaches for this, one is **Asynchronous cancellation**, which terminates the target thread immediately. The other is **Deferred cancellation** allows the target thread to periodically check if it should be cancelled.

2. **Signal Handling**.

   Signals are used in UNIX systems to notify a process that a particular event has occurred. Now in when a Multithreaded process receives a signal, to which thread it must be delivered? It can be delivered to all, or a single thread.

3. **fork() System Call**.

   fork() is a system call executed in the kernel through which a process creates a copy of itself. Now the problem in Multithreaded process is, if one thread forks, will the entire process be copied or not?

4. **Security Issues** because of extensive sharing of resources between multiple threads.

There are many other issues that you might face in a multithreaded process, but there are appropriate solutions available for them. Pointing out some issues here was just to study both sides of the coin.

# 9. Operating System – Memory Management

Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution. Memory management keeps track of each and every memory location, regardless of either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.

## Memory Management

Main Memory refers to a physical memory that is the internal memory to the computer. The word main is used to distinguish it from external mass storage devices such as disk drives. Main memory is also known as RAM. The computer is able to change only data that is in main memory. Therefore, every program we execute and every file we access must be copied from a storage device into main memory.

All the programs are loaded in the main memory for execution. Sometimes complete program is loaded into the memory, but some times a certain part or routine of the program is loaded into the main memory only when it is called by the program, this mechanism is called **Dynamic Loading**, this enhance the performance.

Also, at times one program is dependent on some other program. In such a case, rather than loading all the dependent programs, CPU links the dependent programs to the main executing program when its required. This mechanism is known as **Dynamic Linking**.

## Process Address Space

The process address space is the set of logical addresses that a process references in its code.

For example, when 32-bit addressing is in use, addresses can range from 0 to 0x7fffffff; that is, 2^31 possible numbers, for a total theoretical size of 2 gigabytes.

The operating system takes care of mapping the logical addresses to physical addresses at the

time of memory allocation to the program. There are three types of addresses used in a program before and after memory is allocated:

| S.N. | Memory Addresses & Description |
|---|---|
| 1 | **Symbolic addresses**<br>The addresses used in a source code. The variable names, constants, and instruction labels are the basic elements of the symbolic address space. |
| 2 | **Relative addresses**<br>At the time of compilation, a compiler converts symbolic addresses into relative addresses. |
| 2 | **Physical addresses**<br>The loader generates these addresses at the time when a program is loaded into main memory. |

Virtual and physical addresses are the same in compile-time and load-time address-binding schemes. Virtual and physical addresses differ in execution-time address-binding scheme.

The set of all logical addresses generated by a program is referred to as a **logical address space**. The set of all physical addresses corresponding to these logical addresses is referred to as a **physical address space**.

The runtime mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device. MMU uses the following mechanism to convert virtual address to physical address.

- The value in the base register is added to every address generated by a user process, which is treated as offset at the time it is sent to memory. For example, if the base register value is 10000, then an attempt by the user to use address location 100 will be dynamically reallocated to location 10100.
    - The user program deals with virtual addresses; it never sees the real physical addresses.

## Static vs Dynamic Loading

The choice between Static or Dynamic Loading is to be made at the time of computer program being developed. If you have to load your program statically, then at the time of compilation, the complete programs will be compiled and linked without leaving any external program or module dependency. The linker combines the object program with other

necessary object modules into an absolute program, which also includes logical addresses.

If you are writing a Dynamically loaded program, then your compiler will compile the program and for all the modules which you want to include dynamically, only references will be provided and rest of the work will be done at the time of execution.

At the time of loading, with **static loading**, the absolute program (and data) is loaded into memory in order for execution to start.

If you are using **dynamic loading**, dynamic routines of the library are stored on a disk in relocatable form and are loaded into memory only when they are needed by the program.

## Static vs Dynamic Linking

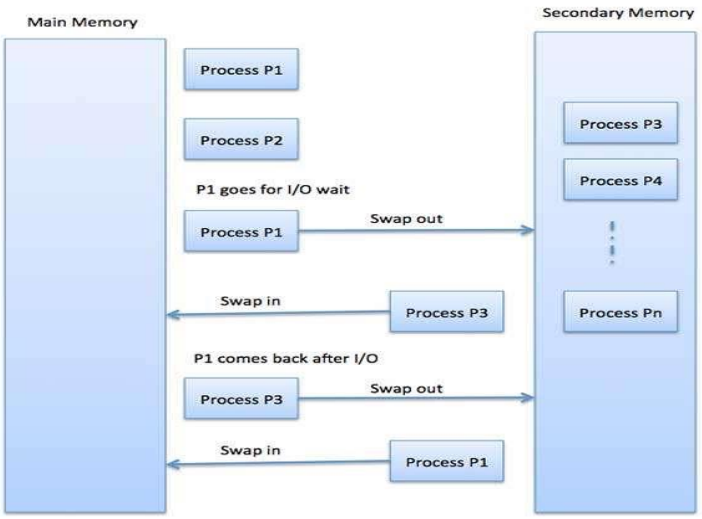
As explained above, when static linking is used, the linker combines all other modules needed by a program into a single executable program to avoid any runtime dependency.

When dynamic linking is used, it is not required to link the actual module or library with the program, rather a reference to the dynamic module is provided at the time of compilation and linking. Dynamic Link Libraries (DLL) in Windows and Shared Objects in Unix are good examples of dynamic libraries.

## Swapping

Swapping is a mechanism in which a process can be swapped temporarily out of main memory (or move) to secondary storage (disk) and make that memory available to other processes. At some later time, the system swaps back the process from the secondary storage to main memory.

Though performance is usually affected by swapping process but it helps in running multiple and big processes in parallel and that's the reason **Swapping is also known as a technique for memory compaction**.




The total time taken by swapping process includes the time it takes to move the entire process to a secondary disk and then to copy the process back to memory, as well as the time the process takes to regain main memory.

Let us assume that the user process is of size 2048KB and on a standard hard disk where swapping will take place has a data transfer rate around 1 MB per second. The actual transfer of the 1000K process to or from memory will take

| |
|---|
| 2048KB / 1024KB per second<br>= 2 seconds<br>= 200 milliseconds |

Now considering in and out time, it will take complete 400 milliseconds plus other overhead where the process competes to regain main memory.

## Memory Allocation

Main memory usually has two partitions:

- **Low Memory** -- Operating system resides in this memory.
- **High Memory** -- User processes are held in high memory.

Operating system uses the following memory allocation mechanism.

| S.N. | Memory Allocation & Description |
|---|---|

| | |
|---|---|
| 1 | **Single-partition allocation**<br><br>In this type of allocation, relocation-register scheme is used to protect user processes from each other, and from changing operating-system code and data. Relocation register contains value of smallest physical address whereas limit register contains range of logical addresses. Each logical address must be less than the limit register. |
| 2 | **Multiple-partition allocation**<br><br>In this type of allocation, main memory is divided into a number of fixed-sized partitions where each partition should contain only one process. When a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. |

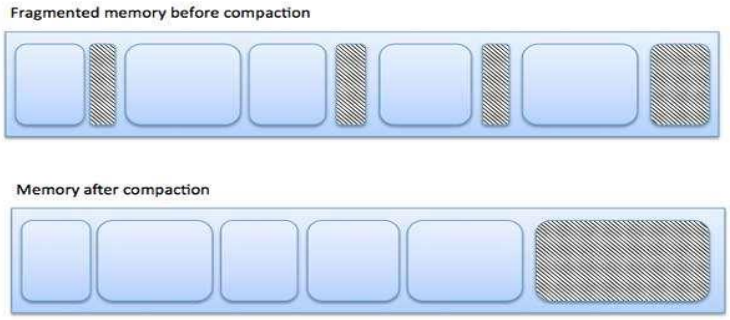
## Fragmentation

As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation.

Fragmentation is of two types:

| S.N. | Fragmentation & Description |
|---|---|
| 1 | **External fragmentation**<br><br>Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous, so it cannot be used. |
| 2 | **Internal fragmentation**<br><br>Memory block assigned to process is bigger. Some portion of memory is left unused, as it cannot be used by another process. |

The following diagram shows how fragmentation can cause waste of memory and a compaction technique can be used to create more free memory out of fragmented memory:




External fragmentation can be reduced by compaction or shuffle memory contents to place all free memory together in one large block. To make compaction feasible, relocation should be dynamic.

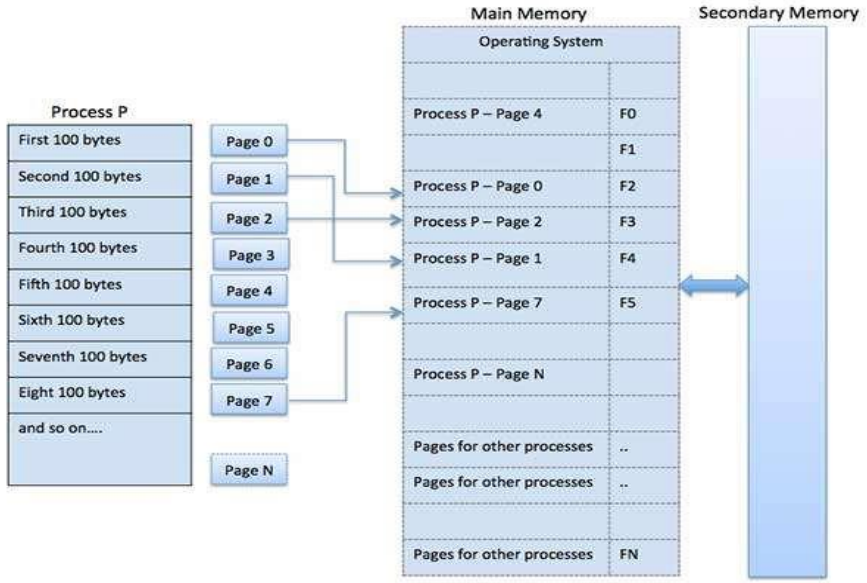
The internal fragmentation can be reduced by effectively assigning the smallest partition but large enough for the process.

## Paging

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard that's set up to emulate the computer's RAM. Paging technique plays an important role in implementing virtual memory.

Paging is a memory management technique in which process address space is broken into blocks of the same size called **pages** (size is power of 2, between 512 bytes and 8192 bytes). The size of the process is measured in the number of pages.

Similarly, main memory is divided into small fixed-sized blocks of (physical) memory called **frames** and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.




**Address Translation**

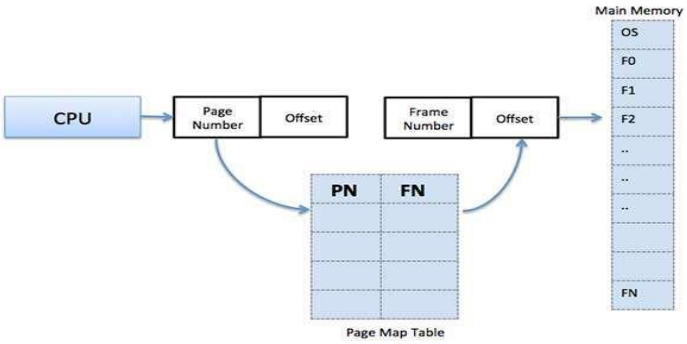
Page address is called **logical address** and represented by **page number** and the **offset**.

| Logical Address = Page number + page offset |
|---|

Frame address is called **physical address** and represented by a **frame number** and the **offset**.

| Physical Address = Frame number + page offset |
|---|

A data structure called **page map table** is used to keep track of the relation between a page of a process to a frame in physical memory.




When the system allocates a frame to any page, it translates this logical address into a physical address and create entry into the page table to be used throughout execution of the program.

When a process is to be executed, its corresponding pages are loaded into any available memory frames. Suppose you have a program of 8Kb but your memory can accommodate only 5Kb at a given point in time, then the paging concept will come into picture. When a computer

runs out of RAM, the operating system (OS) will move idle or unwanted pages of memory to secondary memory to free up RAM for other processes and brings them back when needed by the program.

This process continues during the whole execution of the program where the OS keeps removing idle pages from the main memory and write them onto the secondary memory and bring them back when required by the program.

**Advantages and Disadvantages of Paging**

Here is a list of advantages and disadvantages of paging:

- Paging reduces external fragmentation, but still suffer from internal fragmentation.
- Paging is simple to implement and assumed as an efficient memory management technique.
- Due to equal size of the pages and frames, swapping becomes very easy.
- Page table requires extra memory space, so may not be good for a system having small RAM.

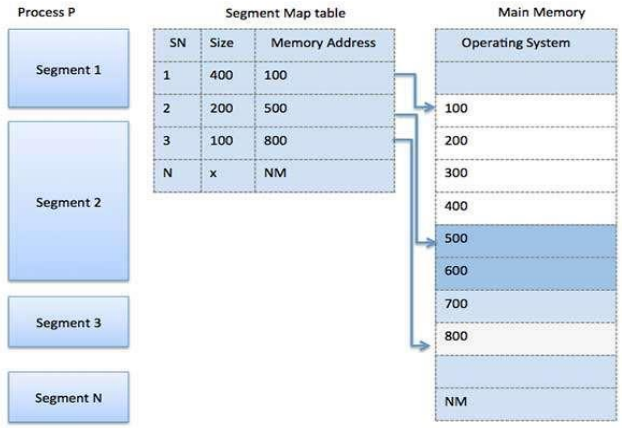
## Segmentation

Segmentation is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions. Each segment is actually a different logical address space of the program.

When a process is to be executed, its corresponding segmentation are loaded into non-contiguous memory though every segment is loaded into a contiguous block of available memory.

Segmentation memory management works very similar to paging but here segments are of variable-length where as in paging pages are of fixed size.

A program segment contains the program's main function, utility functions, data structures, and so on. The operating system maintains a **segment map table** for every process and a list of free memory blocks along with segment numbers, their size and corresponding memory locations in main memory. For each segment, the table stores the starting address of the segment and the length of the segment. A reference to a memory location includes a value that identifies a segment and an offset.




# 10. Operating System – virtual memory

## Definition

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called **virtual memory** and it is a section of a hard disk that's set up to emulate the computer's RAM.

**Virtual memory** is a **memory** management capability of an **OS** that uses hardware and software to allow a computer to compensate for physical **memory** shortages by temporarily transferring data from random access **memory** (RAM) to disk storage.

In the absence of virtual memory, a message "Sorry, you cannot load any more applications. Please close an application to load a new one." would be displayed. With the use of virtual memory, a computer can look for empty areas of RAM which is not being used currently and copies them on to the hard disk device. This process frees up RAM to load new applications. As it is done automatically, the user do not even know that it is happening, and the user feels like RAM has unlimited space even though the RAM capacity is 32 MB.

The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory serves two purposes. First, it allows us to extend the use of physical memory by using disk. Second, it allows us to have memory protection, because each virtual address is translated to a physical address.

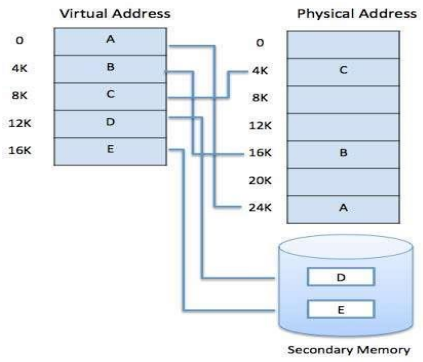
Following are the situations, when entire program is not required to be loaded fully in main memory.

- User written error handling routines are used only when an error occurred in the data or computation.
- Certain options and features of a program may be used rarely.
- Many tables are assigned a fixed amount of address space even though only a small

amount of the table is actually used.

- The ability to execute a program that is only partially in memory would counter many benefits.
- Less number of I/O would be needed to load or swap each user program into memory.
- A program would no longer be constrained by the amount of physical memory that is available.
- Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.

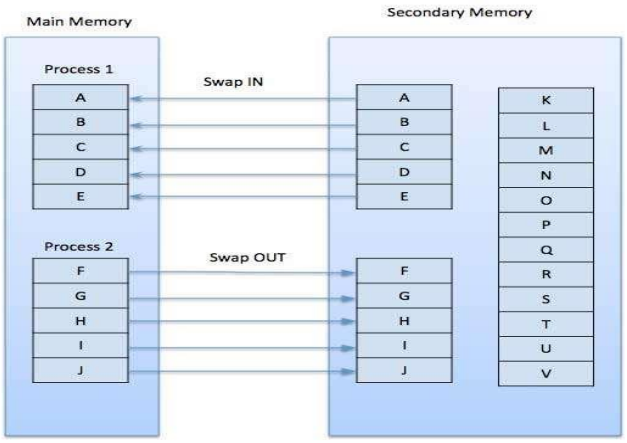
Modern microprocessors intended for general-purpose use, a memory management unit, or MMU, is built into the hardware. The MMU's job is to translate virtual addresses into physical addresses. A basic example is given below:




Virtual memory is commonly implemented by demand paging. It can also be implemented in a segmentation system. Demand segmentation can also be used to provide virtual memory.

## Demand Paging

A demand paging system is quite similar to a paging system with swapping where processes reside in secondary memory and pages are loaded only on demand, not in advance. When a context switch occurs, the operating system does not copy any of the old program's pages out to the disk or any of the new program's pages into the main memory Instead, it just begins executing the new program after loading the first page and fetches that program's pages as they are referenced.




While executing a program, if the program references a page which is not available in the main memory because it was swapped out a little ago, the processor treats this invalid memory reference as a **page fault** and transfers control from the program to the operating system to demand the page back into the memory.

### Advantages

Following are the advantages of Demand Paging:

- Large virtual memory: Job no longer constrained by the size of physical memory (concept of virtual memory)
- More efficient use of memory.
- There is no limit on degree of multiprogramming.
- 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 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 main memory as required.
- It allows greater multiprogramming levels by using less of the available (primary) memory for each process.

**Disadvantage**

- Number of tables and the amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.
- Individual programs face extra latency when they access a page for the first time. So prepaging, a method of remembering which pages a process used when it last executed and preloading a few of them, is used to improve performance.
- Programs running on low-cost, low-power embedded systems may not have a memory management unit that supports page replacement.
- Memory management with page replacement (selects a page to replace) algorithms becomes slightly more complex.

## Page Replacement Algorithm

Page replacement algorithms are the techniques using which an Operating System decides which memory pages to swap out, write to disk when a page of memory needs to be allocated. Paging happens whenever a page fault occurs and a free page cannot be used for

allocation purpose accounting to reason that pages are not available or the number of free pages is lower than required pages.

When the page that was selected for replacement and was paged out, is referenced again, it has to read in from disk, and this requires for I/O completion. This process determines the quality of the page replacement algorithm: the lesser the time waiting for page-ins, the better is the algorithm.

A page replacement algorithm looks at the limited information about accessing the pages provided by hardware, and tries to select which pages should be replaced to minimize the total number of page misses, while balancing it with the costs of primary storage and processor time of the algorithm itself. There are many different page replacement algorithms. We evaluate an algorithm by running it on a particular string of memory reference and computing the number of page faults.

When a page fault occurs, the operating system has to choose a page to remove from memory to make room for the page that has to be brought in. The page replacement is done by swapping the required pages from backup storage to main memory and vice-versa. If the page to be removed has been modified while in memory, it must be rewritten to the disk to bring the disk copy up to date. If, however, the page has not been changed (e.g., it contains program text), the disk copy is already up to date, so no rewrite is needed. The page to be read in just overwrites the page being evicted. A page replacement algorithm is evaluated by running the particular algorithm on a string of memory references and compute the page faults.Referenced string is a sequence of pages being referenced. Page fault is not an error. Contrary to what their name might suggest, page faults are not errors and are common and necessary to increase the amount of memory available to programs in any operating system that utilizes virtual memory, including Microsoft Windows, Mac OS X, Linux and Unix.

Each operating system uses different page replacement algorithms. To select the particular algorithm, the algorithm with lowest page fault rate is considered.

- ✓ Optimal Page Algorithm
- ✓ First In First Out (FIFO) Algorithm
- ✓ Least Recently Used (LRU) Algorithm
- ✓ Page Buffering Algorithm
- ✓ Least Frequently Used (LFU) Algorithm
- ✓ Most Frequently Used (MFU) Algorithm

## Reference String

The string of memory references is called reference string. Reference strings are generated artificially or by tracing a given system and recording the address of each memory reference. The latter choice produces a large number of data, where we note two things.

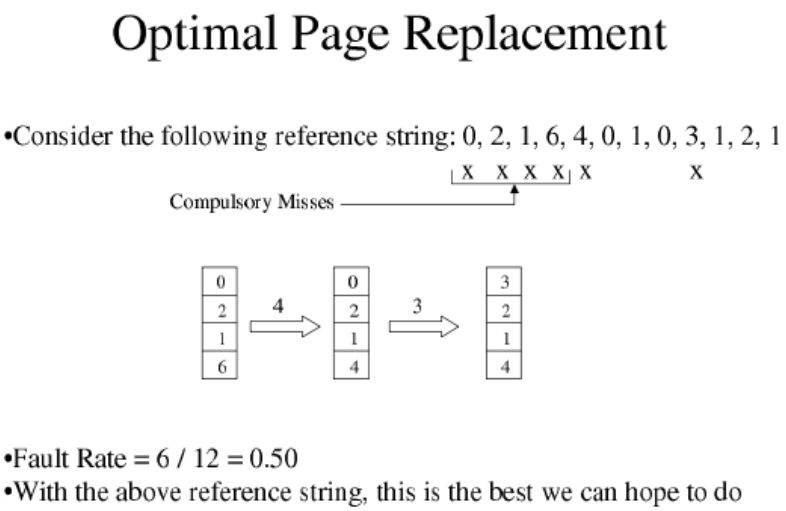
- For a given page size, we need to consider only the page number, not the entire address.
- If we have a reference to a page **p**, then any immediately following references to page **p** will never cause a page fault. Page p will be in memory after the first reference; the immediately following references will not fault.
- For example, consider the following sequence of addresses -

  123,215,600,1234,76,96

  If page size is 100, then the reference string is 1,2,6,12,0,0

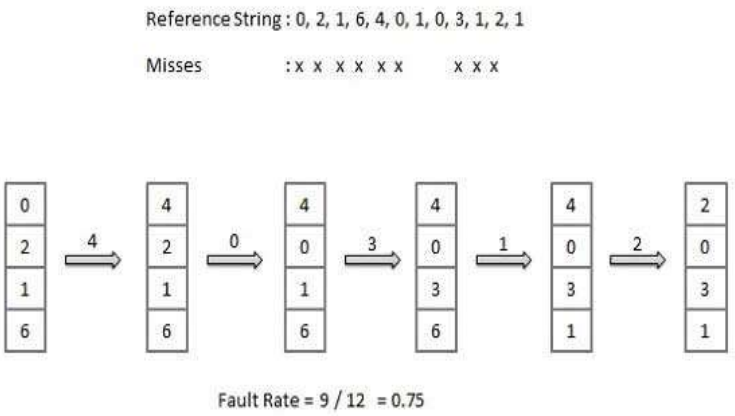
## Optimal Page Algorithm

- An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. An optimal page-replacement algorithm exists, and has been called OPT or MIN.
- Replace the page that will not be used for the longest period of time. Use the time when a page is to be used.



## First In First Out (FIFO) Algorithm

- Oldest page in main memory is the one which will be selected for replacement.
- Easy to implement, keep a list, replace pages from the tail and add new pages at the head.




**Issues:**

•Poor replacement policy

•FIFO doesn't consider the page usage.

## Least Recently Used (LRU) Algorithm

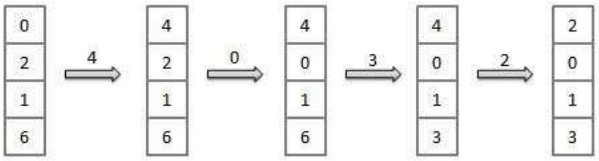
In this algorithm, the page that has not been used for longest period of time is selected for replacement.

Although LRU is theoretically realizable, it is not cheap. To fully implement LRU, it is necessary to maintain a linked list of all pages in memory, with the most recently used page at the front and the least recently used page at the rear. The difficulty is that the list must be updated on every memory reference. Finding a page in the list, deleting it, and then moving it to the front is a very time- consuming operation, even in hardware (assuming that such hardware could be built).

- Page which has not been used for the longest time in main memory is the one which will be selected for replacement.
- Easy to implement, keep a list, replace pages by looking back into time.

Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses : x x x x x x x x



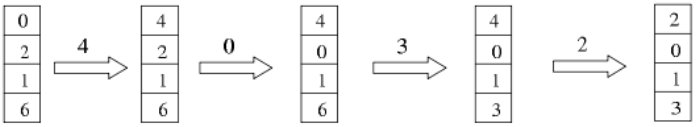

Fault Rate = 8 / 12 = 0.67

# LRU

•Consider the following reference string: 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

x x x x x x x x

Compulsory Misses




•Fault Rate = 8 / 12 = 0.67

## Page Buffering Algorithm

- To get a process start quickly, keep a pool of free frames.
- On page fault, select a page to be replaced.
- Write the new page in the frame of free pool, mark the page table and restart the process.
- Now write the dirty page out of disk and place the frame holding replaced page in free pool.

## Least Frequently Used (LFU) Algorithm

- The page with the smallest count is the one which will be selected for replacement.
- This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again.

## Most Frequently Used (MFU) Algorithm

- This algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

# 11. OPERATING SYSTEM - DEADLOCK

## Introduction

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a wait state. It may happen that waiting processes will never again change state, because the resources they have requested are held by other waiting processes. This situation is called deadlock.

If a process requests an instance of a resource type, the allocation of any instance of the type will satisfy the request. If it will not, then the instances are not identical, and the resource type classes have not been defined properly.

A process must request a resource before using it, and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. **Request**: If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
2. **Use:** The process can operate on the resource.
3. **Release**: The process releases the resource

## Deadlock Characterization

In deadlock, processes never finish executing and system resources are tied up, preventing other jobs from ever starting.

**Necessary Conditions**

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. **Mutual exclusion:** At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource,

the requesting process must be delayed until the resource has been released.

2. **Hold and wait :** There must exist a process that is holding at least one resource and is waiting to acquire additional resources that are currently being held by other processes.

3. **No preemption :** Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process, has completed its task.

4. **Circular wait:** There must exist a set {P0, P1, ..., Pn } of waiting processes such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, …., Pn-1 is waiting for a resource that is held by Pn, and Pn is waiting for a resource that is held by P0.

## Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a system resource- allocation graph. The set of vertices V is partitioned into two different types of nodes P = {P1,

P2, … Pn} the set consisting of all the active processes in the system; and R = {R1, R2, …, R1},

the set consisting of all resource types in the system.

A directed edge from process Pi to resource type Rj is denoted by Pi → Rj, it signifies that process Pi requested an instance of resource type Rj and is currently waiting for that resource.

A directed edge from resource type Rj toprocess Pi is denoted by Rj_ Pi it signifies that an instance of resource type Rj has been allocated to process Pi. A directed edge Pi_ Rj is called a request edge; a directed edge Rj _ Pi is called an assignment edge.

When process Pi requests an instance of resource type Rj, a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is instantaneously transformed to an assignment edge. When the process no longer needs access to the, resource it releases the resource, and as a result the assignment edge is deleted.

Definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If, on the other hand, the graph contains the cycle, then a deadlock must exist.

If each resource type has several instances, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resources types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.

A set of vertices *V* and a set of edges *E*.

- V is partitioned into two types:

o *P* = {*P*1, *P*2, …, *Pn*}, the set consisting of all the processes in the system.

o *R* = {*R*1, *R*2, …, *Rm*}, the set consisting of all resource types in the system.

- request edge – directed edge *P*1 *Rj*
- assignment edge – directed edge *Rj Pi*

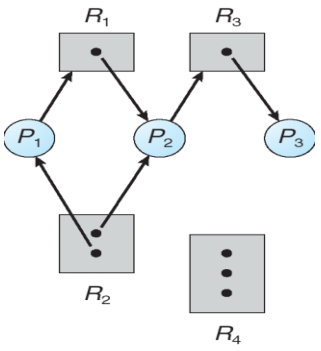



Fig. Resource Allocation Graph

If each resource type has several instance, then a cycle does not necessarily imply that a deadlock incurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

Suppose that process P3requests an instance of resource type R2 Since no resource instance is currently available, a request edge P3 → R2 is added to the graph. At this point, two minimal cycles exist in the system:

P1 → R1 → P2 → R3 → P3 → R2 → P1

P2 → R3 → P3 → R2 → P2

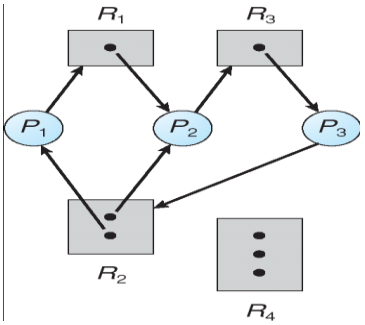



Fig. Resource Allocation Graph with Deadlock

Processes P1, P2, and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3, on the other hand, is waiting for either process P1 or process P2 to release resource R2. In addition, process PI is waiting for process P2 to release resource R1.

## Method For Handling Deadlock

There are are three different methods for dealing with the deadlock problem:

- We can use a protocol to ensure that the system will never enter a deadlock state.
- We can allow the system to enter a deadlock state and then recover.
- We can ignore the problem all together, and pretend that deadlocks never occur in the system. This solution is the one used by most operating systems, including UNIX.

Deadlock avoidance, on the other hand, requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, we can decide for each request whether or not the process should wait. Each request requires that the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process, to decide whether the current request can be satisfied or must be delayed.

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur If a system does not ensure that a deadlock will never occur, and also does not provide a mechanism for deadlock detection and recovery, then we

may arrive at a situation where the system is in a deadlock state yet has no way of recognizing what has happened.

## Deadlock Prevention

For a deadlock to occur, each of the four necessary-conditions must hold. By ensuring that at least on one these conditions cannot hold, we can prevent the occurrence of a deadlock.

### 1 Mutual Exclusion

The mutual-exclusion condition must hold for non-sharable resources. For example, a printer cannot be simultaneously shared by several processes. Sharable resources, on the other hand, do not require mutually exclusive access, and thus cannot be involved in a deadlock.

### 2 Hold and Wait

1. When whenever a process requests a resource, it does not hold any other resources. One protocol that be used requires each process to request and be allocated all its resources before it begins execution.
2. An alternative protocol allows a process to request resources only when the process has none. A process may request some resources and use them. Before it can request any additional resources, however it must release all the resources that it is currently allocated here are two main disadvantages to these protocols. First, resource utilization may be low, since many of the resources may be allocated but unused for a long period. In the example given, for instance, we
can release the tape drive and disk file, and then again request the disk file and printer, only if we can be sure that our data will remain on the disk file. If we cannot be assured that they will, then we must request all resources at the beginning for both protocols.
Second, starvation is possible.

### 3 No Preemption

If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are preempted. That is this resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

### 4 Circular Wait

Circular-wait condition never holds is to impose a total ordering of all resource types, and to require that each process requests resources in an increasing order of enumeration. Let R = {R1, R2, ..., Rn} be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one

precedes another in our ordering. Formally, we define a one-to-one function F: R _ N, where N is the set of natural numbers.

## Deadlock Avoidance

Prevent deadlocks requests can be made. The restraints ensure that at least one of the necessary conditions for deadlock cannot occur, and, hence, that deadlocks cannot hold. Possible side effects of preventing deadlocks by this, melted, however, are Tow device utilization and reduced system throughput.

An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested. For example, in a system with one tape drive and one printer, we might be told that process P will request first the tape drive, and later the printer, before releasing both resources. Process Q on the other hand, will request first the printer, and then the tape drive. With this knowledge of the complete sequence of requests and releases for each process we can decide for each request whether or not the process should wait.

A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular wait condition. The resource allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

### 1 Safe State

A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence. A sequence of processes <P1, P2, .. Pn> is a safe sequence for the current allocation state if, for each Pi the resources that Pj can still request can be satisfied by the currently available resources plus the resources held by all the Pj, with j < i. In this situation, if the resources that process Pi needs are not immediately available, then Pi can wait until all Pj have finished. When they have finished, Pi can obtain all of its needed resources, complete its designated task return its allocated resources, and terminate. When Pi terminates, Pi + 1 can obtain its needed resources, and so on.

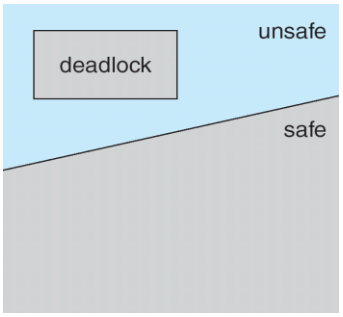



Fig. Safe, Unsafe & Deadlock State

If no such sequence exists, then the system state is said to be unsafe.

### 2 Resource-Allocation Graph Algorithm

Suppose that process Pi requests resource Rj. The request can be granted only if converting the request edge Pi → Rj to an assignment edge Rj → Pi does not result in the formation of a cycle in the resource-allocation graph.

### 3 Banker's Algorithm

The resource-allocation graph algorithm is not applicable to a resource-allocation system with multiple instances of each resource type. The deadlock-avoidance algorithm that we describe next is applicable to such a system, but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the banker's algorithm.

## Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur.

• An algorithm that examines the state of the system to determine whether a deadlock has occurred.

• An algorithm to recover from the deadlock.

### 1 Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph. We obtain this graph from the resource-allocation graph by removing the nodes of type resource and collapsing the appropriate edges.

### 2 Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with

multiple instances of each resource type.

The algorithm used are :

- **Available**: A vector of length m indicates the number of available resources of each type.

- **Allocation**: An n x m matrix defines the number of resources of each type currently allocated to each process.

- **Request**: An n x m matrix indicates the current request of each process. If Request [i, j] = k, then process P, is requesting k more instances of resource type Rj.

### 3 Detection-Algorithm Usage

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken.

## Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives exist. One possibility is to inform the operator that a deadlock has spurred, and to let the operator deal with the deadlock manually. The other possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to preempt some resources from one or more of the deadlocked processes.

### 1 Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

• **Abort all deadlocked processes**: This method clearly will break the dead – lock cycle, but at a great expense, since these processes may have computed for a long time, and the results of these partial computations must be discarded, and probably must be recomputed.

• **Abort one process at a time until the deadlock cycle is eliminated:** This method incurs considerable overhead, since after each process is aborted a deadlock-detection algorithm must be invoked to determine whether a processes are still deadlocked.

### 2 Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until he deadlock cycle is broken. The three issues are considered to recover from deadlock

1. **Selecting a victim**
2. **Rollback**
3. **Starvation**

# 12. Operating System – I/O Hardware

One of the important jobs of an Operating System is to manage various I/O devices including mouse, keyboards, touch pad, disk drives, display adapters, USB devices, Bit- mapped screen, LED, Analog-to-digital converter, On/off switch, network connections, audio I/O, printers etc.

An I/O system is required to take an application I/O request and send it to the physical device, then take whatever response comes back from the device and send it to the application. I/O devices can be divided into two categories:

- **Block devices**: A block device is one with which the driver communicates by sending entire blocks of data. For example, Hard disks, USB cameras, Disk-On-Key etc.
- **Character devices:** A character device is one with which the driver communicates by sending and receiving single characters (bytes, octets). For example, serial ports, parallel ports, sounds cards etc.

## Device Controllers

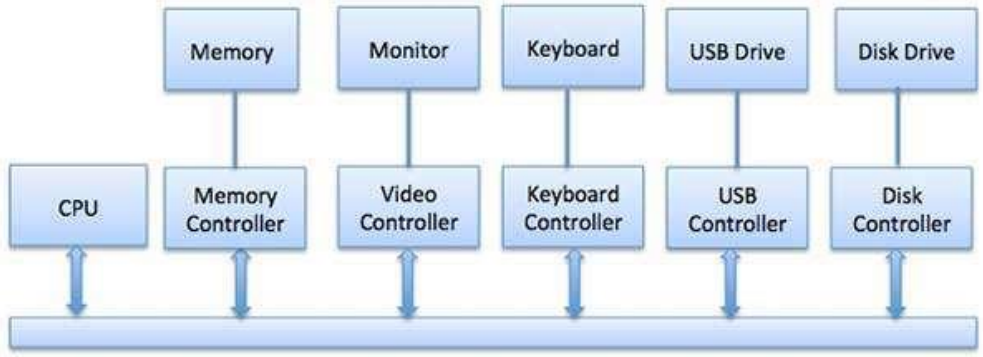
Device drivers are software modules that can be plugged into an OS to handle a particular device. Operating System takes help from device drivers to handle all I/O devices.

The Device Controller works like an interface between a device and a device driver. I/O units (Keyboard, mouse, printer, etc.) typically consist of a mechanical component and an electronic component where electronic component is called the device controller.

There is always a device controller and a device driver for each device to communicate with the Operating Systems. A device controller may be able to handle multiple devices. As an interface its main task is to convert serial bit stream to block of bytes, perform error correction as

necessary.

Any device connected to the computer is connected by a plug and socket, and the socket is connected to a device controller. Following is a model for connecting the CPU, memory, controllers, and I/O devices where CPU and device controllers all use a common bus for communication.




## Synchronous vs Asynchronous I/O

- **Synchronous I/O** — In this scheme CPU execution waits while I/O proceeds
- **Asynchronous I/O** — I/O proceeds concurrently with CPU execution

## Communication to I/O Devices

The CPU must have a way to pass information to and from an I/O device. There are three approaches available to communicate with the CPU and Device.

- Special Instruction I/O
- Memory-mapped I/O
- Direct memory access (DMA)

### Special Instruction I/O

This uses CPU instructions that are specifically made for controlling I/O devices. These instructions typically allow data to be sent to an I/O device or read from an I/O device.

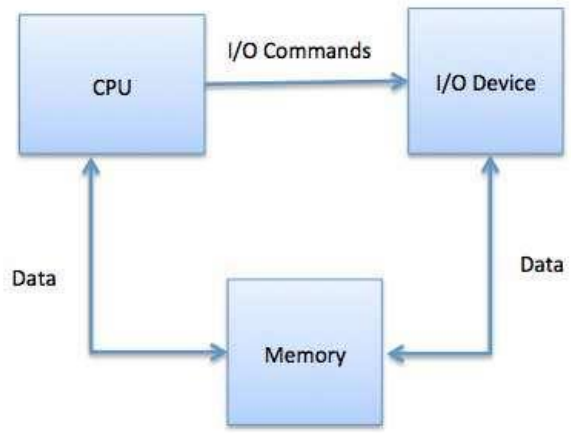
## Memory-mapped I/O

Each controller has a few registers that are used for communicating with the CPU. By writing into these registers, the operating system can command the device to deliver data, accept data, switch itself on or off, or otherwise perform some action. By reading from these registers, the operating system can learn what the device's state is, whether it is prepared to accept a new

command, and so on.

In addition to the control registers, many devices have a data buffer that the operating system can read and write. For example, a common way for computers to display pixels on the screen is to have a video RAM, which is basically just a data buffer, available for programs or the operating system to write into.

When using memory-mapped I/O, the same address space is shared by memory and I/O devices. The device is connected directly to certain main memory locations so that I/O device can transfer block of data to/from memory without going through CPU




While using memory mapped IO, OS allocates buffer in memory and informs I/O device to use that buffer to send data to the CPU. I/O device operates asynchronously with CPU, interrupts CPU when finished.

The advantage to this method is that every instruction which can access memory can be used to manipulate an I/O device. Memory mapped IO is used for most high-speed I/O devices like disks, communication interfaces.

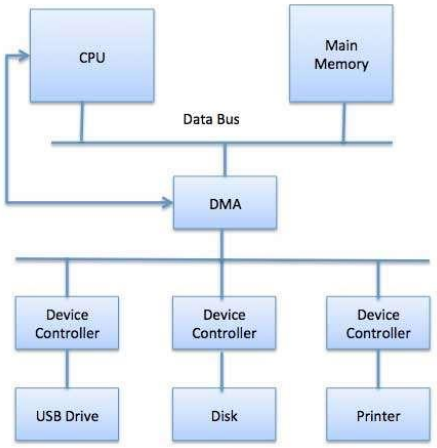
## Direct Memory Access (DMA)

Slow devices like keyboards will generate an interrupt to the main CPU after each byte is transferred. If a fast device such as a disk generated an interrupt for each byte, the operating system would spend most of its time handling these interrupts. So a typical computer uses direct memory access (DMA) hardware to reduce this overhead

Direct Memory Access (DMA) means CPU grants I/O module authority to read from or write to memory without involvement. DMA module itself controls exchange of data between main memory and the I/O device. CPU is only involved at the beginning and end of the transfer and

interrupted only after entire block has been transferred.

Direct Memory Access needs a special hardware called DMA controller (DMAC) that manages the data transfers and arbitrates access to the system bus. The controllers are programmed with source and destination pointers (where to read/write the data), counters to track the number of transferred bytes, and settings, which includes I/O and memory types, interrupts and states for the CPU cycles.




The operating system uses the DMA hardware as follows:

| Step | Description |
|---|---|
| 1 | Device driver is instructed to transfer disk data to a buffer address X. |
| 2 | Device driver then instruct disk controller to transfer data to buffer. |
| 3 | Disk controller starts DMA transfer. |
| 4 | Disk controller sends each byte to DMA controller. |
| 5 | DMA controller transfers bytes to buffer, increases the memory address, decreases the counter C until C becomes zero. |
| 6 | When C becomes zero, DMA interrupts CPU to signal transfer completion. |

## Polling vs Interrupts I/O

A computer must have a way of detecting the arrival of any type of input. There are two ways that this can happen, known as **polling** and **interrupts**. Both of these techniques allow the processor to deal with events that can happen at any time and that are not related to the

process it is currently running.

## Polling I/O

Polling is the simplest way for an I/O device to communicate with the processor the processor. The process of periodically checking status of the device to see if it is time for the next I/O operation, is called polling. The I/O device simply puts the information in a Status register, and the processor must come and get the information.

Most of the time, devices will not require attention and when one does it will have to wait until it is next interrogated by the polling program. This is an inefficient method and much of the processors time is wasted on unnecessary polls.

Compare this method to a teacher continually asking every student in a class, one after another, if they need help. Obviously the more efficient method would be for a student to inform the teacher whenever they require assistance.

## Interrupts I/O

An alternative scheme for dealing with I/O is the interrupt-driven method. An interrupt is a signal to the microprocessor from a device that requires attention.

A device controller puts an interrupt signal on the bus when it needs CPU's attention when CPU receives an interrupt, It saves its current state and invokes the appropriate interrupt handler using the interrupt vector (addresses of OS routines to handle various events). When the interrupting device has been dealt with, the CPU continues with its original task as if it had never been interrupted.

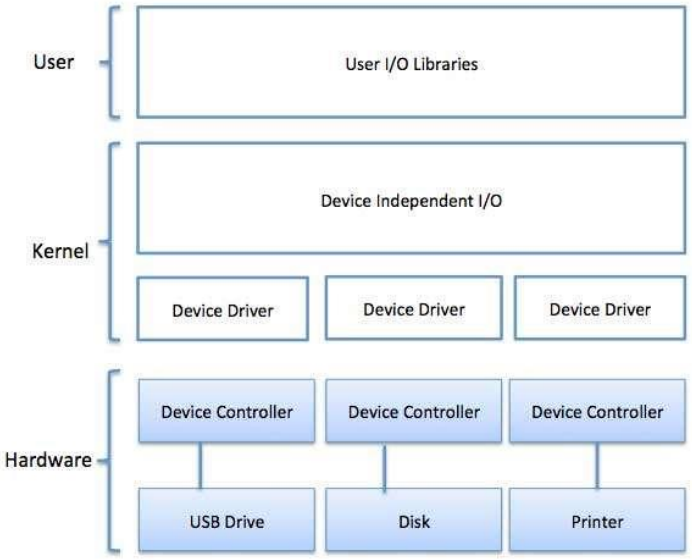
# 13. Operating System – I/O Software

In computing, **input/output** or I/O is the communication between an information processing system, such as a computer, and the outside world, possibly a human or another information processing system.

I/O software is often organized in the following layers:

- **User Level Libraries:** This provides simple interface to the user program to perform input and output. For example, **stdio** is a library provided by C and C++ programming languages.
- **Kernel Level Modules:** This provides device driver to interact with the device controller and device independent I/O modules used by the device drivers.
- **Hardware**: This layer includes actual hardware and hardware controller which interact with the device drivers and makes hardware alive.

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




## Device Drivers

Device drivers are software modules that can be plugged into an OS to handle a particular

device. Operating System takes help from device drivers to handle all I/O devices. Device drivers encapsulate device-dependent code and implement a standard interface in such a way that code contains device-specific register reads/writes. Device driver, is generally written by the device's manufacturer and delivered along with the device on a CD-ROM.

A device driver performs the following jobs:

- To accept request from the device independent software above to it.
- Interact with the device controller to take and give I/O and perform required error handling
- Making sure that the request is executed successfully

How a device driver handles a request is as follows: Suppose a request comes to read a block N. If the driver is idle at the time a request arrives, it starts carrying out the request immediately. Otherwise, if the driver is already busy with some other request, it places the new request in the queue of pending requests.

## Interrupt handlers

An interrupt handler, also known as an interrupt service routine or ISR, is a piece of software or more specifically a callback function in an operating system or more specifically in a device driver, whose execution is triggered by the reception of an interrupt.

When the interrupt happens, the interrupt procedure does whatever it has to in order to handle the interrupt, updates data structures and wakes up process that was waiting for an interrupt to happen.

The interrupt mechanism accepts an address ─ a number that selects a specific interrupt handling routine/function from a small set. In most architectures, this address is an offset stored in a table called the interrupt vector table. This vector contains the memory addresses of specialized interrupt handlers.

## Device-Independent I/O Software

The basic function of the device-independent software is to perform the I/O functions that are common to all devices and to provide a uniform interface to the user-level software. Though it is difficult to write completely device independent software but we can write some modules which are common among all the devices. Following is a list of functions of device-independent I/O Software:

- Uniform interfacing for device drivers
- Device naming ─ Mnemonic names mapped to Major and Minor device numbers
- Device protection

- Providing a device-independent block size
- Buffering because data coming off a device cannot be stored in final destination.
- Storage allocation on block devices
- Allocation and releasing dedicated devices
- Error Reporting

## User-Space I/O Software

These are the libraries which provide richer and simplified interface to access the functionality of the kernel or ultimately interactive with the device drivers. Most of the user-level I/O software consists of library procedures with some exception like spooling system which is a way of dealing with dedicated I/O devices in a multiprogramming system.

I/O Libraries (e.g., stdio) are in user-space to provide an interface to the OS resident device-independent I/O SW. For example putchar(), getchar(), printf() and scanf() are example of user level I/O library stdio available in C programming.

## Kernel I/O Subsystem

Kernel I/O Subsystem is responsible to provide many services related to I/O. Following are some of the services provided:

- **Scheduling** - Kernel schedules a set of I/O requests to determine a good order in which to execute them. When an application issues a blocking I/O system call, the request is placed on the queue for that device. The Kernel I/O scheduler rearranges the order of the queue to improve the overall system efficiency and the average response time experienced by the applications.
- **Buffering** - Kernel I/O Subsystem maintains a memory area known as **buffer** that stores data while they are transferred between two devices or between a device with an application operation. Buffering is done to cope with a speed mismatch between the producer and consumer of a data stream or to adapt between devices that have different data transfer sizes.
- **Caching** - Kernel maintains cache memory which is region of fast memory that holds copies of data. Access to the cached copy is more efficient than access to the original.
- **Spooling and Device Reservation** - A spool is a buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams. The spooling system copies the queued spool files to the printer one at a time. In some operating systems,

spooling is managed by a system daemon process. In other operating systems, it is handled by an in kernel thread.

- **Error Handling** - An operating system that uses protected memory can guard against many kinds of hardware and application errors.

# 14. Operating System – File System

## File

A file is a named collection of related information that is recorded on secondary storage such as magnetic disks, magnetic tapes and optical disks. In general, a file is a sequence of bits, bytes, lines or records whose meaning is defined by the files creator and user.

## File Structure

A File Structure should be according to a required format that the operating system can understand.

- A file has a certain defined structure according to its type.
- A text file is a sequence of characters organized into lines.
- A source file is a sequence of procedures and functions.
- An object file is a sequence of bytes organized into blocks that are understandable by the machine.
- When an operating system defines different file structures, it also contains the code to support these file structure. Unix, MS-DOS support minimum number of file structure.

## File Type

File type refers to the ability of the operating system to distinguish different types of file such as text files source files and binary files etc. Many operating systems support many types of files. Operating system like MS-DOS and UNIX have the following types of files:

**Ordinary files**

- These are the files that contain user information.
- These may have text, databases or executable program.
- The user can apply various operations on such files like add, modify, delete or even remove the entire file.

**Directory files**

- These files contain list of file names and other information related to these files.

**Special files**

- These files are also known as device files.
- These files represent physical device like disks, terminals, printers, networks, tape drive etc.

These files are of two types:

- **Character special files -** data is handled character by character as in case of terminals or printers.
- **Block special files -** data is handled in blocks as in the case of disks and tapes.

## File Access Mechanisms

File access mechanism refers to the manner in which the records of a file may be accessed. There are several ways to access files:

- Sequential access
- Direct/Random access
- Indexed sequential access

**Sequential access**

A sequential access is that in which the records are accessed in some sequence, i.e., the information in the file is processed in order, one record after the other. This access method is the most primitive one. Example: Compilers usually access files in this fashion.

**Direct/Random access**

- Random access file organization provides, accessing the records directly.
- Each record has its own address on the file with by the help of which it can be directly accessed for reading or writing.
- The records need not be in any sequence within the file and they need not be in adjacent locations on the storage medium.

**Indexed sequential access**

- This mechanism is built up on base of sequential access.
- An index is created for each file which contains pointers to various blocks.
- Index is searched sequentially and its pointer is used to access the file directly.

## Space Allocation

Files are allocated disk spaces by operating system. Operating systems deploy following three main ways to allocate disk space to files.

- Contiguous Allocation
- Linked Allocation
- Indexed Allocation

**Contiguous Allocation**

- Each file occupies a contiguous address space on disk.
- Assigned disk address is in linear order.
- Easy to implement.
- External fragmentation is a major issue with this type of allocation technique.

**Linked Allocation**

- Each file carries a list of links to disk blocks.
- Directory contains link / pointer to first block of a file.
- No external fragmentation
- Effectively used in sequential access file.
- Inefficient in case of direct access file.

**Indexed Allocation**

- Provides solutions to problems of contiguous and linked allocation.
- An index block is created having all pointers to files.
- Each file has its own index block which stores the addresses of disk space occupied by the file.
- Directory contains the addresses of index blocks of files.

# 15. Operating System – Security

Security refers to providing a protection system to computer system resources such as CPU, memory, disk, software programs and most importantly data/information stored in the computer system. If a computer program is run by an unauthorized user, then he/she may cause severe damage to computer or data stored in it. So a computer system must be protected against unauthorized access, malicious access to system memory, viruses, worms etc. We're going to discuss following topics in this chapter:

- Authentication
- One Time passwords
- Program Threats
- System Threats
- Computer Security Classifications

## Authentication

Authentication refers to identifying each user of the system and associating the executing programs with those users. It is the responsibility of the Operating System to create a protection system which ensures that a user who is running a particular program is authentic. Operating Systems generally identifies/authenticates users using following three ways:

- **Username / Password** - User need to enter a registered username and password with Operating system to login into the system.
- **User card/key** - User need to punch card in card slot, or enter key generated by key generator in option provided by operating system to login into the system.
- **User attribute - fingerprint/ eye retina pattern/ signature** - User need to pass his/her attribute via designated input device used by operating system to login into the system.

## One Time passwords

One-time passwords provide additional security along with normal authentication. In One-Time Password system, a unique password is required every time user tries to login into the system. Once a one-time password is used, then it cannot be used again. One-time password are implemented in various ways.

- **Random numbers** - Users are provided cards having numbers printed along with corresponding alphabets. System asks for numbers corresponding to few alphabets randomly chosen.
- **Secret key** - User are provided a hardware device which can create a secret id mapped with user id. System asks for such secret id which is to be generated every time prior to login.
- **Network password** - Some commercial applications send one-time passwords to user on registered mobile/ email which is required to be entered prior to login.

## Program Threats

Operating system's processes and kernel do the designated task as instructed. If a user program made these process do malicious tasks, then it is known as **Program Threats**. One of the common example of program threat is a program installed in a computer which can store and send user credentials via network to some hacker. Following is the list of some well-known program threats.

- **Trojan Horse** - Such program traps user login credentials and stores them to send to malicious user who can later on login to computer and can access system resources.
- **Trap Door** - If a program which is designed to work as required, have a security hole in its code and perform illegal action without knowledge of user then it is called to have a trap door.
- **Logic Bomb** - Logic bomb is a situation when a program misbehaves only when certain conditions met otherwise it works as a genuine program. It is harder to detect.
- **Virus** - Virus as name suggest can replicate themselves on computer system. They are highly dangerous and can modify/delete user files, crash systems. A virus is generally a small code embedded in a program. As user accesses the program, the virus starts getting embedded in other files/ programs and can make system unusable for user.

## System Threats

System threats refers to misuse of system services and network connections to put user in trouble (a suggestion that something unpleasant or violent will happen, especially if a particular action or order is not followed). System threats can be used to launch program threats on a

complete network called as program attack. A system threats creates such an environment that operating system resources/ user files are misused.

Following is the list of some well-known system threats.

- **Worm** -Worm is a process which can choked down a system performance by using system resources to extreme levels. A Worm process generates its multiple copies where each copy uses system resources, prevents all other processes to get required resources. Worms processes can even shut down an entire network.
- **Port Scanning** - Port scanning is a mechanism or means by which a hacker can detects system vulnerabilities to make an attack on the system.
- **Denial of Service** - Denial of service attacks normally prevents user to make legitimate use of the system. For example, a user may not be able to use internet if denial of service attacks browser's content settings.

## Computer Security Classifications

As per the U.S. Department of Defense Trusted Computer System's Evaluation Criteria there are four security classifications in computer systems: A, B, C, and D. This is widely used specifications to determine and model the security of systems and of security solutions. Following is the brief description of each classification.

| S.N. | Classification Type | Description |
|---|---|---|
| 1 | Type A | Highest Level. Uses formal design specifications and verification techniques. Grants a high degree of assurance of process security. |
| 2 | Type B | Provides mandatory protection system. Have all the properties of a class C2 system. Attaches a sensitivity label to each object. It is of three types.<br>• **B1** - Maintains the security label of each object in the system. Label is used for making decisions to access control.<br>• **B2** - Extends the sensitivity labels to each system resource, such as storage objects, supports covert channels and auditing of events. |

| | | |
|---|---|---|
| | | Provides protection and user accountability using audit capabilities. It is of two types.<br><br>**C1** - Incorporates controls so that users can protect their private information and keep other users from accidentally reading / deleting their data. UNIX versions are mostly Cl class. |
| | | Lowest level. Minimum protection. MS-DOS, Window 3.1 fall in |

# 2. Operating System – Linux

Linux is one of popular version of UNIX operating System. It is open source as its source code is freely available. It is free to use. Linux was designed considering UNIX compatibility. Its functionality list is quite similar to that of UNIX.

## What is Unix?

The Unix operating system is a set of programs that act as a link between the computer and the user.

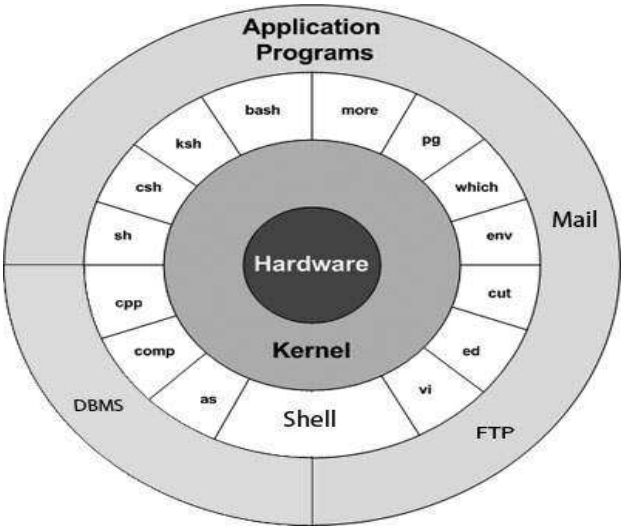
The computer programs that allocate the system resources and coordinate all the details of the computer's internals is called the **operating system** or the **kernel**.

Users communicate with the kernel through a program known as the **shell**. The shell is a command line interpreter; it translates commands entered by the user and converts them into a language that is understood by the kernel.

- Unix was originally developed in 1969 by a group of AT&T employees Ken Thompson, Dennis Ritchie, Douglas McIlroy, and Joe Ossanna at Bell Labs.
- There are various Unix variants available in the market. Solaris Unix, AIX, HP Unix and BSD are a few examples. Linux is also a flavor of Unix which is freely available.
- Several people can use a Unix computer at the same time; hence Unix is called a multiuser system.
- A user can also run multiple programs at the same time; hence Unix is a multitasking environment.

## Unix Architecture

Here is a basic block diagram of a Unix system –




The main concept that unites all the versions of Unix is the following four basics :

- **Kernel:** The kernel is the heart of the operating system. It interacts with the hardware and most of the tasks like memory management, task scheduling and file management.
- **Shell:** The shell is the utility that processes your requests. When you type in a command at your terminal, the shell interprets the command and calls the program that you want. The shell uses standard syntax for all commands. C Shell, Bourne Shell and Korn Shell are the most famous shells which are available with most of the Unix variants.
- **Commands and Utilities:** There are various commands and utilities which you can make use of in your day to day activities. **cp, mv, cat** and **grep,** etc. are few examples of commands and utilities. There are over 250 standard commands plus numerous others provided through 3rd party software. All the commands come along with various options.
- **Files and Directories:** All the data of Unix is organized into files. All files are then organized into directories. These directories are further organized into a tree-like structure called the **filesystem**.

**System Bootup**

If you have a computer which has the Unix operating system installed in it, then you simply need to turn on the system to make it live.

As soon as you turn on the system, it starts booting up and finally it prompts you to log into the system, which is an activity to log into the system and use it for your day-to-day activities.

## Login Unix

When you first connect to a Unix system, you usually see a prompt such as the following:

login:

**To log in**

- Have your userid (user identification) and password ready. Contact your system administrator if you don't have these yet.
- Type your userid at the login prompt, then press **ENTER**. Your userid is **case-sensitive**, so be sure you type it exactly as your system administrator has instructed.
- Type your password at the password prompt, then press **ENTER**. Your password is also case-sensitive.
- If you provide the correct userid and password, then you will be allowed to enter into the system. Read the information and messages that comes up on the screen, which is as follows.

```
login : amrood
amrood's password:
Last login: Sun Jun 14 09:32:32 2009 from 62.61.164.73
$
```

[
[

You will be provided with a command prompt (sometime called the **$** prompt ) where you type all your commands. For example, to check calendar, you need to type the **cal** command as follows –

```
$ cal
```

June 2009

```
Su M  Tu W  Th Fr  Sa
      1   2   3   4   5   6
 7    8   9  10  11  12  13
14  15  16  17  18  19  20
21  22  23  24  25  26  27
28  29  30
```

**Change Password**

All Unix systems require passwords to help ensure that your files and data remain your own and that the system itself is secure from hackers and crackers. Following are the steps to change your password –

**Step 1:** To start, type password at the command prompt as shown below.

**Step 2:** Enter your old password, the one you're currently using.

**Step 3:** Type in your new password. Always keep your password complex enough so that nobody can guess it. But make sure, you remember it.

**Step 4:** You must verify the password by typing it again.

```
$ passwd
Changing password for amrood
(current) Unix password:******
New Unix password:*******
Retype new Unix password:*******
passwd: all authentication tokens updated successfully
$
```

**Note** − We have added asterisk (*) here just to show the location where you need to enter the current and new passwords otherwise at your system. It does not show you any character when you type.

**Listing Directories and Files**

All data in Unix is organized into files. All files are organized into directories. These directories are organized into a tree-like structure called the filesystem.

You can use the **ls** command to list out all the files or directories available in a directory. Following is the example of using **ls** command with **-l** option.

```
$ ls -l
total 19621
drwxrwxr-x  2 amroo  amrood     4096 De  25 09:59  uml
-rw-rw-r--  1 amroo  amrood     5341 De  25 08:38  uml.jpg
drwxr-xr-x  2 amroo  amrood     4096 Feb 15  2006 univ
drwxr-xr-x  2 root   root       4096 De   9  2007 urlspedia
-rw-r--r--  1 root   root      27648 De   9  2007 urlspedia.tar
drwxr-xr-x  8 root   root       4096 No  25  2007 usr
-rwxr-xr-x  1 root   root       3192 No  25  2007 webthumb.php
-rw-rw-r--  1 amroo  amrood    20480 No  25  2007 webthumb.tar
```

```
-rw-rw-r--   1 amroo  amrood      5654 Au   9  2007 yourfile.mid
-rw-rw-r--   1 amroo  amrood     16625 Au   9  2007 yourfile.swf
                d                 5     g
$
```

Here entries starting with **d.....** represent directories. For example, uml, univ and urlspedia are directories and rest of the entries are files.

**Who Are You?**

While you're logged into the system, you might be willing to know : **Who am I**? The easiest way to find out "who you are" is to enter the **whoami** command −

```
$
 whoami
 amrood
$
```

Try it on your system. This command lists the account name associated with the current login. You can try **who am i** command as well to get information about yourself.

**Who is Logged in?**

Sometime you might be interested to know who is logged in to the computer at the same time. There are three commands available to get you this information, based on how much you wish to know about the other users: **users, who,** and **w**.

```
$ users
 amrood bablu qadir

$ who
amrood ttyp0 Oct 8 14:10 (limbo)
bablu   ttyp2 Oct 4 09:08 (calliope)
qadir   ttyp4 Oct 8 12:09 (dent)

$
```

Try the **w** command on your system to check the output. This lists down information associated with the users logged in the system.

## Logging Out

When you finish your session, you need to log out of the system. This is to ensure that

nobody else accesses your files.

**To log out**

- Just type the **logout** command at the command prompt, and the system will clean up everything and break the connection.

**System Shutdown**

The most consistent way to shut down a Unix system properly via the command line is to use one of the following commands

| Command | Description |
|---|---|
| **halt** | Brings the system down immediately |
| **init 0** | Powers off the system using predefined scripts to synchronize and clean up the system prior to shutting down |
| **init 6** | Reboots the system by shutting it down completely and then restarting it |
| **poweroff** | Shuts down the system by powering off |
| **reboot** | Reboots the system |
| **shutdown** | Shuts down the system |

You typically need to be the super user or root (the most privileged account on a Unix system) to shut down the system. However, on some standalone or personally-owned Unix boxes, an administrative user and sometimes regular users can do so.

# Unix – File Management

In this chapter, we will discuss in detail about file management in Unix. All data in Unix is organized into files. All files are organized into directories. These directories are organized into a tree-like structure called the filesystem.

When you work with Unix, one way or another, you spend most of your time working with files. This chapter will help you understand how to create and remove files, copy and rename them, create links to them, etc.

In Unix, there are three basic types of files –

- **Ordinary Files** − An ordinary file is a file on the system that contains data, text, or program instructions. In this lecture, you look at working with ordinary files.
- **Directories** − Directories store both special and ordinary files. For users familiar with Windows or Mac OS, Unix directories are equivalent to folders.
- **Special Files** − Some special files provide access to hardware such as hard drives, CD-ROM drives, modems, and Ethernet adapters. Other special files are similar to aliases or shortcuts and enable you to access a single file using different names.

## Listing Files

To list the files and directories stored in the current directory, use the following command:

```
$ls
```

Here is the sample output of the above command –

```
$ls

bin         hosts   lib       res.03
ch07        hw1     pub       test_results
ch07.bak    hw      res.01    users
docs        2       res.02    work
```

The command **ls** supports the **-l** option which would help you to get more information about the listed files –

```
$ls -l
total 1962188

drwxrwxr-x  2 amrood amrood      4096 Dec 25 09:59 uml
-rw-rw-r--  1 amrood amrood      5341 Dec 25 08:38 uml.jpg
drwxr-xr-x  2 amroo  amrood      4096 Feb 15  2006 univ
drwxr-xr-x  2 root   root       4096 De   9  2007 urlspedia
-rw-r--r--  1 root   root      27648 De   9  2007 urlspedia.tar
drwxr-xr-x  8 root   root       4096 No  25  2007 usr
drwxr-xr-x  2  200    300       4096 No  25  2007 webthumb-1.01
-rwxr-xr-x  1 root   root       3192 No  25  2007 webthumb.php
-rw-rw-r--  1 amroo  amrood     20480 No  25  2007 webthumb.tar
-rw-rw-r--  1 amroo  amrood      5654 Au   9  2007 yourfile.mid
-rw-rw-r--  1 amroo  amrood     16625 Au   9  2007 yourfile.swf
```

```
drwxr-xr-x  11 amroo  amrood        4096 Ma  29   2007 zlib-1.2.3
$
```

Here is the information about all the listed columns –

- **First Column**: Represents the file type and the permission given on the file. Below is the description of all type of files.
- **Second Column**: Represents the number of memory blocks taken by the file or directory.
- **Third Column**: Represents the owner of the file. This is the Unix user who created this file.
- **Fourth Column**: Represents the group of the owner. Every Unix user will have an associated group.
- **Fifth Column**: Represents the file size in bytes.
- **Sixth Column**: Represents the date and the time when this file was created or modified for the last time.
- **Seventh Column**: Represents the file or the directory name.

In the **ls -l** listing example, every file line begins with a **d**, **-**, or **l**. These characters indicate the type of the file that's listed.

| Prefix | Description |
|---|---|
| **-** | Regular file, such as an ASCII text file, binary executable, or hard link |
| **b** | Block special file. Block input/output device file such as a physical hard drive |
| **c** | Character special file. Raw input/output device file such as a physical hard drive |
| **d** | Directory file that contains a listing of other files and directories |
| **l** | Symbolic link file. Links on any regular file |
| **p** | Named pipe. A mechanism for interprocess communications |
| **s** | Socket used for interprocess communication |

## Metacharacters

Metacharacters have a special meaning in Unix. For example, ***** and **?** are metacharacters. We

use ***** to match 0 or more characters, a question mark (**?**) matches with a single character. For Example −

```
$ls ch*.doc
```

Displays all the files, the names of which start with **ch** and end with **.doc** –

```
ch01-1.doc   ch010.doc  ch02.doc   ch03-2.doc
ch04-1.doc   ch040.doc  ch05.doc   ch06-2.doc
ch01-2.doc ch02-1.doc c
```

Here, ***** works as meta character which matches with any character. If you want to display all the files ending with just **.doc**, then you can use the following command –

```
$ls *.doc
```

## Hidden Files

An invisible file is one, the first character of which is the dot or the period character (.). Unix programs (including the shell) use most of these files to store configuration information.

Some common examples of the hidden files include the files −

- **.profile** − The Bourne shell ( sh) initialization script
- **.kshrc** − The Korn shell ( ksh) initialization script
- **.cshrc** − The C shell ( csh) initialization script
- **.rhosts** − The remote shell configuration file

To list the invisible files, specify the **-a** option to **ls** –

```
$ ls -a

.           .profile    docs    lib     test_results
..          .rhosts     hosts   pub     users
.emacs      bin         hw1     res.01  work
.exrc       ch07        hw2     res.02
.kshrc      ch07.bak    hw2     res.02
$
```

- Single dot (**.**) − This represents the current directory.
- Double dot (**..**) − This represents the parent directory.

## Creating Files

You can use the **vi** editor to create ordinary files on any Unix system. You simply need to give the following command −

```
$ vi filename
```

The above command will open a file with the given filename. Now, press the key **i** to come into the edit mode. Once you are in the edit mode, you can start writing your content in the file as in the following program –

```
This is unix file....I created it for the first time..... I'm going to save
this content in this file.
```

Once you are done with the program, follow these steps −

- Press the key **esc** to come out of the edit mode.
- Press two keys **Shift + Z** together to come out of the file completely. You will now have a file created with **filename** in the current directory.

```
$ vi filename
$
```

## Editing Files

You can edit an existing file using the **vi** editor. We will discuss in short how to open an existing file −

```
$ vi filename
```

Once the file is opened, you can come in the edit mode by pressing the key **i** and then you can proceed by editing the file. If you want to move here and there inside a file, then first you need to come out of the edit mode by pressing the key **Esc.** After this, you can use the following keys to move inside a file –

- **l** key to move to the right side.
- **h** key to move to the left side.
- **k** key to move upside in the file.
- **j** key to move downside in the file.

So using the above keys, you can position your cursor wherever you want to edit. Once you are positioned, then you can use the **i** key to come in the edit mode. Once you are done with the editing in your file, press **Esc** and finally two keys **Shift + ZZ** together to come out of the

file completely.

## Display Content of a File

You can use the **cat** command to see the content of a file. Following is a simple example to see the content of the above created file −

```
$ cat filename
This is unix file....I created it for the first time..... I'm going to save
this content in this file.
$
```

You can display the line numbers by using the **-b** option along with the **cat** command as follows –

```
$ cat -b filename

1  This is unix file....I created it for the first time.....

2  I'm going to save this content in this file.

$
```

## Counting Words in a File

You can use the **wc** command to get a count of the total number of lines, words, and characters contained in a file. Following is a simple example to see the information about the file created above −

```
$ wc filename
2  19 103 filename
$
```

Here is the detail of all the four columns −

- **First Column**: Represents the total number of lines in the file.
- **Second Column**: Represents the total number of words in the file.
- **Third Column**: Represents the total number of bytes in the file. This is the actual size of the file.
- **Fourth Column**: Represents the file name.

You can give multiple files and get information about those files at a time. Following is simple syntax −

```
$ wc filename1 filename2 filename3
```

## Copying Files

To make a copy of a file use the **cp** command. The basic syntax of the command is −

```
$ cp source_file destination_file
```

Following is the example to create a copy of the existing file **filename**.

```
$ cp filename copyfile
$
```

You will now find one more file **copyfile** in your current directory. This file will exactly be the same as the original file **filename**.

## Renaming Files

To change the name of a file, use the **mv** command. Following is the basic syntax −

```
$ mv old_file new_file
```

The following program will rename the existing file **filename** to **newfile.**

```
$ mv filename newfile
$
```

The **mv** command will move the existing file completely into the new file. In this case, you will find only **newfile** in your current directory.

## Deleting Files

To delete an existing file, use the **rm** command. Following is the basic syntax −

```
$ rm filename
```

**Caution:** A file may contain useful information. It is always recommended to be careful while using this **Delete** command. It is better to use the **-i** option along with **rm** command.

Following is the example which shows how to completely remove the existing file

**filename**.

```
$ rm filename
$
```

You can remove multiple files at a time with the command given below –

```
$ rm filename1 filename2 filename3
$
```

## Standard Unix Streams

Under normal circumstances, every Unix program has three streams (files) opened for it when it starts up −

- **stdin** − This is referred to as the *standard input* and the associated file descriptor is 0. This is also represented as STDIN. The Unix program will read the default input from STDIN.
- **stdout** − This is referred to as the *standard output* and the associated file descriptor is 1. This is also represented as STDOUT. The Unix program will write the default output at STDOUT
- **stderr** − This is referred to as the *standard error* and the associated file descriptor is 2. This is also represented as STDERR. The Unix program will write all the error messages at STDERR.

## Unix – Directory Management

A directory is a file the solo job of which is to store the file names and the related information. All the files, whether ordinary, special, or directory, are contained in directories.
Unix uses a hierarchical structure for organizing files and directories. This structure is often referred to as a directory tree. The tree has a single root node, the slash character (/), and all other directories are contained below it.

### Home Directory

The directory in which you find yourself when you first login is called your home directory. You will be doing much of your work in your home directory and subdirectories that you'll be creating to organize your files.

You can go in your home directory anytime using the following command −

```
$cd ~
$
```

Here **~** indicates the home directory. Suppose you have to go in any other user's home directory, use the following command –

```
$cd ~username
$
```

To go in your last directory, you can use the following command –

```
$cd -
$
```

### Absolute / Relative Pathnames

Directories are arranged in a hierarchy with root **(/)** at the top. The position of any file within the hierarchy is described by its pathname.

Elements of a pathname are separated by a /. A pathname is absolute, if it is described in relation to root, thus absolute pathnames always begin with a /.
Following are some examples of absolute filenames.

```
/etc/passwd
/users/sjones/chem/notes
/dev/rdsk/Os3
```

A pathname can also be relative to your current working directory. Relative pathnames never begin with /. Relative to user amrood's home directory, some pathnames might look like this –

```
chem/notes
personal/res
```

To determine where you are within the filesystem hierarchy at any time, enter the command **pwd** to print the current working directory –

```
$pwd
/user0/home/amrood

$
```

## Listing Directories

To list the files in a directory, you can use the following syntax −

```
$ls dirname
```

Following is the example to list all the files contained in **/usr/local** directory –

```
$ls /usr/local

X11     bin     gimp    jikes   sbin
ace     doc     include lib     share
atalk   etc     info    man     ami
```

## Creating Directories

Directories are created by the following command −

```
$mkdir dirname
```

Here, directory is the absolute or relative pathname of the directory you want to create. For example, the command –

```
$mkdir mydir
$
```

Creates the directory **mydir** in the current directory. Here is another example –

```
$mkdir /tmp/test-dir
$
```

This command creates the directory **test-dir** in the **/tmp** directory. The **mkdir** command produces no output if it successfully creates the requested directory.

If you give more than one directory on the command line, **mkdir** creates each of the directories. For example, −

```
$mkdir docs pub
$
```

Creates the directories docs and pub under the current directory.

## Creating Parent Directories

Sometimes when you want to create a directory, its parent directory or directories might not exist. In this case, **mkdir** issues an error message as follows −

```
$mkdir /tmp/amrood/test
mkdir: Failed to make directory "/tmp/amrood/test"; No
such file or directory
$
```

In such cases, you can specify the **-p** option to the **mkdir** command. It creates all the necessary directories for you. For example –

```
$mkdir -p /tmp/amrood/test
$
```

The above command creates all the required parent directories.

## Removing Directories

Directories can be deleted using the **rmdir** command as follows −

```
$rmdir dirname
$
```

**Note** − To remove a directory, make sure it is empty which means there should not be any file or sub-directory inside this directory.

You can remove multiple directories at a time as follows −

```
$rmdir dirname1 dirname2 dirname3
$
```

The above command removes the directories dirname1, dirname2, and dirname3, if they are empty. The **rmdir** command produces no output if it is successful.

## Changing Directories

You can use the **cd** command to do more than just change to a home directory. You can use it to change to any directory by specifying a valid absolute or relative path. The syntax is as given below −

```
$cd dirname
$
```

Here, **dirname** is the name of the directory that you want to change to. For example, the command –

```
$cd /usr/local/bin
$
```

Changes to the directory **/usr/local/bin**. From this directory, you can **cd** to the directory **/usr/home/amrood** using the following relative path –

```
$cd ../../home/amrood
$
```

## Renaming Directories

The **mv (move)** command can also be used to rename a directory. The syntax is as follows:

```
$mv olddir newdir
$
```

You can rename a directory **mydir** to **yourdir** as follows –

```
$mv mydir yourdir
$
```

## The directories .(dot) and ..(dot dot)

The **filename .** (dot) represents the current working directory; and the **filename ..** (dot dot) represents the directory one level above the current working directory, often referred to as the parent directory.

If we enter the command to show a listing of the current working directories/files and use the **-a option** to list all the files and the **-l option** to provide the long listing, we will receive the following result.

```
$ls -la
drwxrwxr-x   4   teacher   class   2048   Jul   16 17.56 .
drwxr-xr-x   60  root              1536   Jul   13 14:18 ..
----------   1   teacher   class   4210   May   1 08:27 .profile
-rwxr-xr-x   1   teacher   class   1948   May   12 13:42 memo
$
```

# Unix – File Permission/ Access Mode

In this chapter, we will discuss in detail about file permission and access modes in Unix. File ownership is an important component of Unix that provides a secure method for storing files. Every file in Unix has the following attributes –

- **Owner permissions** − The owner's permissions determine what actions the owner of the file can perform on the file.
- **Group permissions** − The group's permissions determine what actions a user, who is a member of the group that a file belongs to, can perform on the file.
- **Other (world) permissions** − The permissions for others indicate what action all other users can perform on the file.

## The Permission Indicators

While using **ls -l** command, it displays various information related to file permission as follows −

```
$ls -l /home/amrood
-rwxr-xr--  1 amrood   users 1024   Nov 2 00:10   myfile
drwxr-xr--- 1 amrood   users 1024   Nov 2 00:10   mydir
```

Here, the first column represents different access modes, i.e., the permission associated with a file or a directory.

The permissions are broken into groups of threes, and each position in the group denotes a specific permission, in this order: read (r), write (w), execute (x) −

- The first three characters (2-4) represent the permissions for the file's owner. For example, **-rwxr-xr--** represents that the owner has read (r), write (w) and execute (x) permission.
- The second group of three characters (5-7) consists of the permissions for the group to which the file belongs. For example, **-rwxr-xr--** represents that the group has read (r) and execute (x) permission, but no write permission.
- The last group of three characters (8-10) represents the permissions for everyone else.

For example, **-rwxr-xr--** represents that there is **read (r)** only permission.

## File Access Modes

The permissions of a file are the first line of defense in the security of a Unix system. The basic building blocks of Unix permissions are the **read**, **write**, and **execute** permissions, which have been described below −

### Read

Grants the capability to read, i.e., view the contents of the file.

### Write

Grants the capability to modify, or remove the content of the file.

### Execute

User with execute permissions can run a file as a program.

## Directory Access Modes

Directory access modes are listed and organized in the same manner as any other file. There are a few differences that need to be mentioned:

### Read

Access to a directory means that the user can read the contents. The user can look at the **filenames** inside the directory.

### Write

Access means that the user can add or delete files from the directory.

### Execute

Executing a directory doesn't really make sense, so think of this as a traverse permission.

A user must have **execute** access to the **bin** directory in order to execute the **ls** or the **cd** command.

## Changing Permissions

To change the file or the directory permissions, you use the **chmod** (change mode) command. There are two ways to use chmod — the symbolic mode and the absolute mode.

### Using chmod in Symbolic Mode

The easiest way for a beginner to modify file or directory permissions is to use the symbolic mode. With symbolic permissions you can add, delete, or specify the permission set you want by using the operators in the following table.

| chmod Operator | Description |
|---|---|
| + | Adds the designated permission(s) to a file or directory. |
| - | Removes the designated permission(s) from a file or directory. |
| = | Sets the designated permission(s). |

Here's an example using **testfile**. Running **ls -1** on the testfile shows that the file's permissions are as follows –

```
$ls -l testfile
-rwxrwxr--  1 amrood   users 1024  Nov 2 00:10  testfile
```

Then each example **chmod** command from the preceding table is run on the testfile, followed by **ls –l**, so you can see the permission changes –

```
$chmod o+wx testfile
$ls -l testfile
-rwxrwxrwx  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod u-x testfile
$ls -l testfile
-rw-rwxrwx  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod g=rx testfile
$ls -l testfile
-rw-r-xrwx  1 amrood   users 1024  Nov 2 00:10  testfile
```

Here's how you can combine these commands on a single line:

```
$chmod o+wx,u-x,g=rx testfile
$ls -l testfile
-rw-r-xrwx  1 amrood   users 1024  Nov 2 00:10  testfile
```

## Using chmod with Absolute Permissions

The second way to modify permissions with the chmod command is to use a number to specify each set of permissions for the file.

Each permission is assigned a value, as the following table shows, and the total of each set of permissions provides a number for that set.

| | | |
|---|---|---|

| Number | Octal Permission Representation | Ref |
|---|---|---|
| **0** | No permission | --- |
| **1** | Execute permission | --x |
| **2** | Write permission | -w- |
| **3** | Execute and write permission: 1 (execute) + 2 (write) = 3 | -wx |
| **4** | Read permission | r-- |
| **5** | Read and execute permission: 4 (read) + 1 (execute) = 5 | r-x |
| **6** | Read and write permission: 4 (read) + 2 (write) = 6 | rw- |
| **7** | All permissions: 4 (read) + 2 (write) + 1 (execute) = 7 | rwx |

[[

Here's an example using the testfile. Running **ls -1** on the testfile shows that the file's permissions are as follows –

```
$ls -l testfile
-rwxrwxr--  1 amrood   users 1024  Nov 2 00:10  testfile
```

Then each example **chmod** command from the preceding table is run on the testfile, followed by **ls –l**, so you can see the permission changes –

```
$ chmod 755 testfile
$ls -l testfile
-rwxr-xr-x  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod 743 testfile
$ls -l testfile
-rwxr---wx  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod 043 testfile
$ls -l testfile
----r---wx  1 amrood   users 1024  Nov 2 00:10  testfile
```

## Changing Owners and Groups

While creating an account on Unix, it assigns a **owner ID** and a **group ID** to each user. All the permissions mentioned above are also assigned based on the Owner and the Groups.

Two commands are available to change the owner and the group of files −

- **chown** − The **chown** command stands for **"change owner"** and is used to change the owner of a file.
- **chgrp** − The **chgrp** command stands for **"change group"** and is used to change the group of a file.

## Changing Ownership

The **chown** command changes the ownership of a file. The basic syntax is as follows −

```
$ chown user filelist
```

The value of the user can be either the **name of a user** on the system or the **user id (uid)** of a user on the system.

The following example will help you understand the concept −

```
$ chown amrood testfile
$
```

Changes the owner of the given file to the user **amrood**.

**NOTE:** The super user, root, has the unrestricted capability to change the ownership of any file but normal users can change the ownership of only those files that they own.

## Changing Group Ownership

The **chgrp** command changes the group ownership of a file. The basic syntax is as follows:

```
$ chgrp group filelist
```

The value of group can be the **name of a group** on the system or **the group ID (GID)** of a group on the system.

Following example helps you understand the concept:

```
$ chgrp special testfile
$
```

Changes the group of the given file to **special** group.

## SUID and SGID File Permission

Often when a command is executed, it will have to be executed with special privileges in order to accomplish its task.

As an example, when you change your password with the **passwd** command, your new password is stored in the file **/etc/shadow**.

As a regular user, you do not have **read** or **write** access to this file for security reasons, but when you change your password, you need to have the write permission to this file. This means that the **passwd** program has to give you additional permissions so that you can write to the file **/etc/shadow**.

Additional permissions are given to programs via a mechanism known as the **Set User ID (SUID)** and **Set Group ID (SGID)** bits.

When you execute a program that has the SUID bit enabled, you inherit the permissions of that program's owner. Programs that do not have the SUID bit set are run with the permissions of the user who started the program.

This is the case with SGID as well. Normally, programs execute with your group permissions, but instead your group will be changed just for this program to the group owner of the program.

The SUID and SGID bits will appear as the letter **"s"** if the permission is available. The SUID **"s"** bit will be located in the permission bits where the owners' **execute** permission normally resides.

For example, the command -

```
$ ls -l /usr/bin/passwd
-r-sr-xr-x   1    root    bin   19031 Feb 7 13:47   /usr/bin/passwd*
$
```

Shows that the SUID bit is set and that the command is owned by the root. A capital letter **S** in the execute position instead of a lowercase **s** indicates that the execute bit is not set.

If the sticky bit is enabled on the directory, files can only be removed if you are one of the following users −

- The owner of the sticky directory
- The owner of the file being removed
- The super user, root

To set the SUID and SGID bits for any directory try the following command −

```
$ chmod ug+s dirname
$ ls -l
drwsr-sr-x 2 root root   4096 Jun 19 06:45 dirname
$
```

# Unix – Process Management

In this chapter, we will discuss in detail about process management in Unix. When you execute a program on your Unix system, the system creates a special environment for that program. This environment contains everything needed for the system to run the program as if no other program were running on the system.

Whenever you issue a command in Unix, it creates, or starts, a new process. When you tried out the **ls** command to list the directory contents, you started a process. A process, in simple terms, is an instance of a running program.

The operating system tracks processes through a five-digit ID number known as the **pid** or the **process ID**. Each process in the system has a unique **pid**.

Pids eventually repeat because all the possible numbers are used up and the next pid rolls or starts over. At any point of time, no two processes with the same pid exist in the system because it is the pid that Unix uses to track each process.

## Starting a Process

When you start a process (run a command), there are two ways you can run it −

- Foreground Processes
- Background Processes

## Foreground Processes

By default, every process that you start runs in the foreground. It gets its input from the keyboard and sends its output to the screen.

You can see this happen with the **ls** command. If you wish to list all the files in your current directory, you can use the following command −

```
$ls ch*.doc
```

This would display all the files, the names of which start with **ch** and end with **.doc** –

```
ch01-1.doc    ch010.doc  ch02.doc    ch03-2.doc
ch04-1.doc    ch040.doc  ch05.doc    ch06-2.doc
ch01-2.doc    ch02-1.doc
```

The process runs in the foreground, the output is directed to my screen, and if the **ls** command wants any input (which it does not), it waits for it from the keyboard.

While a program is running in the foreground and is time-consuming, no other commands can

be run (start any other processes) because the prompt would not be available until the program finishes processing and comes out.

## Background Processes

A background process runs without being connected to your keyboard. If the background process requires any keyboard input, it waits.

The advantage of running a process in the background is that you can run other commands; you do not have to wait until it completes to start another!

The simplest way to start a background process is to add an ampersand (**&**) at the end of the command.

```
$ls ch*.doc &
```

This displays all those files the names of which start with **ch** and end with **.doc** –

```
ch01-1.doc    ch010.doc  ch02.doc    ch03-2.doc
ch04-1.doc    ch040.doc  ch05.doc    ch06-2.doc
ch01-2.doc    ch02-1.doc
```

Here, if the **ls** command wants any input (which it does not), it goes into a stop state until we move it into the foreground and give it the data from the keyboard.

That first line contains information about the background process - the job number and the process ID. You need to know the job number to manipulate it between the background and the foreground.

Press the Enter key and you will see the following −

```
[1]  +  Done                 ls ch*.doc &
$
```

The first line tells you that the **ls** command background process finishes successfully. The second is a prompt for another command.

## Listing Running Processes

It is easy to see your own processes by running the **ps** (process status) command as follows:

```
$ps
PID        TTY        TIME        CMD
18358      ttyp3      00:00:00    sh
18361      ttyp3      00:01:31    abiword
18789      ttyp3      00:00:00    ps
```

One of the most commonly used flags for ps is the **-f** ( f for full) option, which provides more information as shown in the following example –

```
$ps -f
UID        PID  PPID C STIME    TTY    TIME CMD
amrood    6738 3662 0 10:23:03 pts/6 0:00 first_one
amrood    6739 3662 0 10:22:54 pts/6 0:00 second_one
amrood    3662 3657 0 08:10:53 pts/6 0:00 -ksh
amrood    6892 3662 4 10:51:50 pts/6 0:00 ps -f
```

Here is the description of all the fields displayed by **ps -f** command –

| Column | Description |
|---|---|
| **UID** | User ID that this process belongs to (the person running it) |
| **PID** | Process ID |
| **PPID** | Parent process ID (the ID of the process that started it) |
| **C** | CPU utilization of process |
| **STIME** | Process start time |
| **TTY** | Terminal type associated with the process |
| **TIME** | CPU time taken by the process |
| **CMD** | The command that started this process |

There are other options which can be used along with **ps** command −

| Option | Description |
|---|---|
| **-a** | Shows information about all users |
| **-x** | Shows information about processes without terminals |
| **-u** | Shows additional information like -f option |
| **-e** | Displays extended information |

## Stopping Processes

Ending a process can be done in several different ways. Often, from a console-based command, sending a CTRL + C keystroke (the default interrupt character) will exit the command. This works when the process is running in the foreground mode.

If a process is running in the background, you should get its Job ID using the **ps** command. After that, you can use the **kill** command to kill the process as follows −

```
$ps -f
UID      PID   PPID C STIME     TTY    TIME CMD
amrood    6738 3662 0 10:23:03 pts/6 0:00 first_one
amrood    6739 3662 0 10:22:54 pts/6 0:00 second_one
amrood    3662 3657 0 08:10:53 pts/6 0:00 -ksh
amrood    6892 3662 4 10:51:50 pts/6 0:00 ps -f
$kill 6738
Terminated
```

Here, the **kill** command terminates the **first_one** process. If a process ignores a regular kill command, you can use **kill -9** followed by the process ID as follows –

```
$kill -9 6738
Terminated
```

## Parent and Child Processes

Each unix process has two ID numbers assigned to it: The Process ID (pid) and the Parent process ID (ppid). Each user process in the system has a parent process.

Most of the commands that you run have the shell as their parent. Check the **ps -f** example where this command listed both the process ID and the parent process ID.

## Zombie and Orphan Processes

Normally, when a child process is killed, the parent process is updated via a **SIGCHLD** signal. Then the parent can do some other task or restart a new child as needed. However, sometimes the parent process is killed before its child is killed. In this case, the "parent of all processes," the **init** process, becomes the new PPID (parent process ID). In some cases, these processes are called orphan processes.

When a process is killed, a **ps** listing may still show the process with a **Z** state. This is a zombie or defunct process. The process is dead and not being used. These processes are different from the orphan processes. They have completed execution but still find an entry in the process table.

## Daemon Processes

Daemons are system-related background processes that often run with the permissions of root and services requests from other processes.

A daemon has no controlling terminal. It cannot open **/dev/tty**. If you do a **"ps -ef"** and look at the **tty** field, all daemons will have a **?** for the **tty**.

To be precise, a daemon is a process that runs in the background, usually waiting for something to happen that it is capable of working with. For example, a printer daemon waiting for print commands.

If you have a program that calls for lengthy processing, then it's worth to make it a daemon and run it in the background.

## The top Command

The **top** command is a very useful tool for quickly showing processes sorted by various criteria.

It is an interactive diagnostic tool that updates frequently and shows information about physical and virtual memory, CPU usage, load averages, and your busy processes.

Here is the simple syntax to run top command and to see the statistics of CPU utilization by different processes −

```
$top
```

## Job ID Versus Process ID

Background and suspended processes are usually manipulated via **job number (job ID)**. This number is different from the process ID and is used because it is shorter.

In addition, a job can consist of multiple processes running in a series or at the same time, in parallel. Using the job ID is easier than tracking individual processes.

# Unix – Vi Editor

In this chapter, we will understand how the vi Editor works in Unix. There are many ways to edit files in Unix. Editing files using the screen-oriented text editor **vi** is one of the best ways. This editor enables you to edit lines in context with other lines in the file.

An improved version of the vi editor which is called the **VIM** has also been made available now. Here, VIM stands for **V**i **IM**proved.

vi is generally considered the de facto standard in Unix editors because −

- It's usually available on all the flavors of Unix system.
- Its implementations are very similar across the board.
- It requires very few resources.
- It is more user-friendly than other editors such as the **ed** or the **ex**.

You can use the **vi** editor to edit an existing file or to create a new file from scratch. You can also use this editor to just read a text file.

## Starting the vi Editor

The following table lists out the basic commands to use the vi editor −

| Command | Description |
|---|---|
| **vi filename** | Creates a new file if it already does not exist, otherwise opens an existing file. |
| **vi -R filename** | Opens an existing file in the read-only mode. |
| **view filename** | Opens an existing file in the read-only mode. |

Following is an example to create a new file **testfile** if it already does not exist in the current working directory −

```
$vi testfile
```

The above command will generate the following output –

```
|
~
~
~
```


You will notice a **tilde** (~) on each line following the cursor. A tilde represents an unused line. If a line does not begin with a tilde and appears to be blank, there is a space, tab, newline, or some other non-viewable character present.

You now have one open file to start working on. Before proceeding further, let us understand a few important concepts.

## Operation Modes

While working with the vi editor, we usually come across the following two modes −

- **Command mode** − This mode enables you to perform administrative tasks such as saving the files, executing the commands, moving the cursor, cutting (yanking) and pasting the lines or words, as well as finding and replacing. In this mode, whatever you type is interpreted as a command.
- **Insert mode** − This mode enables you to insert text into the file. Everything that's typed in this mode is interpreted as input and placed in the file.

vi always starts in the **command mode**. To enter text, you must be in the insert mode for which simply type **i**. To come out of the insert mode, press the **Esc** key, which will take you back to the command mode.

**Hint** − If you are not sure which mode you are in, press the Esc key twice; this will take you to the command mode. You open a file using the vi editor. Start by typing some characters and then come to the command mode to understand the difference.

## Getting Out of vi

The command to quit out of vi is **:q**. Once in the command mode, type colon, and 'q', followed by return. If your file has been modified in any way, the editor will warn you of this, and not let you quit. To ignore this message, the command to quit out of vi without saving is **:q!**. This lets you exit vi without saving any of the changes.

The command to save the contents of the editor is **:w**. You can combine the above command with the quit command, or use **:wq** and return.

The easiest way to **save your changes and exit vi** is with the **ZZ** command. When you are in the command mode, type **ZZ**. The **ZZ** command works the same way as the :**wq** command.

If you want to specify/state any particular name for the file, you can do so by specifying it after the **:w**. For example, if you wanted to save the file you were working on as another filename called **filename2**, you would type **:w filename2** and return.

## Moving within a File

To move around within a file without affecting your text, you must be in the command mode (press Esc twice). The following table lists out a few commands you can use to move around one character at a time −

| Command | Description |
|---|---|
| **k** | Moves the cursor up one line |
| **j** | Moves the cursor down one line |
| **h** | Moves the cursor to the left one character position |
| **l** | Moves the cursor to the right one character position |

The following points need to be considered to move within a file −

- vi is case-sensitive. You need to pay attention to capitalization when using the commands.
- Most commands in vi can be prefaced by the number of times you want the action to occur. For example, **2j** moves the cursor two lines down the cursor location.

There are many other ways to move within a file in vi. Remember that you must be in the command mode (**press Esc twice**). The following table lists out a few commands to move around the file –

| Command | Description |
|---|---|
| **0 or \|** | Positions the cursor at the beginning of a line |
| **$** | Positions the cursor at the end of a line |
| **w** | Positions the cursor to the next word |

| | |
|---|---|
| **b** | Positions the cursor to the previous word |
| **(** | Positions the cursor to the beginning of the current sentence |
| **)** | Positions the cursor to the beginning of the next sentence |
| **E** | Moves to the end of the blank delimited word |
| **{** | Moves a paragraph back |
| **}** | Moves a paragraph forward |
| **[[** | Moves a section back |
| **]]** | Moves a section forward |
| **n\|** | Moves to the column **n** in the current line |
| **1G** | Moves to the first line of the file |
| **G** | Moves to the last line of the file |
| **nG** | Moves to the **n$^{th}$** line of the file |
| **:n** | Moves to the **n$^{th}$** line of the file |
| **fc** | Moves forward to **c** |
| **Fc** | Moves back to **c** |
| **H** | Moves to the top of the screen |
| **nH** | Moves to the **n$^{th}$** line from the top of the screen |
| **M** | Moves to the middle of the screen |
| **L** | Move to the bottom of the screen |

| | |
|---|---|
| **nL** | Moves to the **n$^{th}$** line from the bottom of the screen |
| **:x** | Colon followed by a number would position the cursor on the line number represented by **x** |

## Control Commands

The following commands can be used with the Control Key to performs functions as given in the table below −

| Command | Description |
|---|---|
| **CTRL+d** | Moves forward 1/2 screen |
| **CTRL+f** | Moves forward one full screen |
| **CTRL+u** | Moves backward 1/2 screen |
| **CTRL+b** | Moves backward one full screen |
| **CTRL+e** | Moves the screen up one line |
| **CTRL+y** | Moves the screen down one line |
| **CTRL+u** | Moves the screen up 1/2 page |
| **CTRL+d** | Moves the screen down 1/2 page |
| **CTRL+b** | Moves the screen up one page |
| **CTRL+f** | Moves the screen down one page |
| **CTRL+I** | Redraws the screen |

## Editing Files

To edit the file, you need to be in the insert mode. There are many ways to enter the insert mode from the command mode −

| Command | Description |
|---|---|
| **i** | Inserts text before the current cursor location |
| **I** | Inserts text at the beginning of the current line |
| **a** | Inserts text after the current cursor location |
| **A** | Inserts text at the end of the current line |
| **o** | Creates a new line for text entry below the cursor location |
| **O** | Creates a new line for text entry above the cursor location |

[[

## Deleting Characters

Here is a list of important commands, which can be used to delete characters and lines in an open file −

| Command | Description |
|---|---|
| **x** | Deletes the character under the cursor location |
| **X** | Deletes the character before the cursor location |
| **dw** | Deletes from the current cursor location to the next word |
| **d^** | Deletes from the current cursor position to the beginning of the line |
| **d$** | Deletes from the current cursor position to the end of the line |
| **D** | Deletes from the cursor position to the end of the current line |
| **dd** | Deletes the line the cursor is on |

As mentioned above, most commands in vi can be prefaced by the number of times you want the action to occur. For example, **2x** deletes two characters under the cursor location and **2dd** deletes two lines the cursor is on.

It is recommended that the commands are practiced before we proceed further.

## Change Commands

You also have the capability to change characters, words, or lines in vi without deleting them. Here are the relevant commands –

| Command | Description |
|---|---|
| **cc** | Removes the contents of the line, leaving you in insert mode. |
| **cw** | Changes the word the cursor is on from the cursor to the lowercase **w** end of the word. |
| **r** | Replaces the character under the cursor. vi returns to the command mode after the replacement is entered. |
| **R** | Overwrites multiple characters beginning with the character currently under the cursor. You must use **Esc** to stop the overwriting. |
| **s** | Replaces the current character with the character you type. Afterward, you are left in the insert mode. |
| **S** | Deletes the line the cursor is on and replaces it with the new text. After the new text is entered, vi remains in the insert mode. |

## Copy and Paste Commands

You can copy lines or words from one place and then you can paste them at another place using the following commands −

| Command | Description |
|---|---|
| **yy** | Copies the current line. |
| **yw** | Copies the current word from the character the lowercase **w** cursor is on, until the end of the word. |

| | |
|---|---|
| **p** | Puts the copied text after the cursor. |
| **P** | Puts the yanked text before the cursor. |

## Advanced Commands

There are some advanced commands that simplify day-to-day editing and allow for more efficient use of vi −

| Command | Description |
|---|---|
| **J** | Joins the current line with the next one. A count of **j** commands join many lines. |
| << | Shifts the current line to the left by one shift width. |
| >> | Shifts the current line to the right by one shift width. |
| ~ | Switches the case of the character under the cursor. |
| **^G** | Press Ctrl and G keys at the same time to show the current filename and the status. |
| **U** | Restores the current line to the state it was in before the cursor entered the line. |
| **u** | This helps undo the last change that was done in the file. Typing 'u' again will re-do the change. |
| **J** | Joins the current line with the next one. A count joins that many lines. |
| **:f** | Displays the current position in the file in % and the file name, the total number of file. |
| **:f filename** | Renames the current file to filename. |
| **:w filename** | Writes to file filename. |
| **:e filename** | Opens another file with filename. |

| | |
|---|---|
| **:cd dirname** | Changes the current working directory to dirname. |
| **:e #** | Toggles between two open files. |
| **:n** | In case you open multiple files using vi, use **:n** to go to the next file in the series. |
| **:p** | In case you open multiple files using vi, use **:p** to go to the previous file in the series. |
| **:N** | In case you open multiple files using vi, use **:N** to go to the previous file in the series. |
| **:r file** | Reads file and inserts it after the current line. |
| **:nr file** | Reads file and inserts it after the line **n**. |

[

## Word and Character Searching

The vi editor has two kinds of searches: **string** and **character**. For a string search, the / and **?** commands are used. When you start these commands, the command just typed will be shown on the last line of the screen, where you type the particular string to look for.

These two commands differ only in the direction where the search takes place −

- The / command searches forwards (downwards) in the file.
- The **?** command searches backwards (upwards) in the file.

The **n** and **N** commands repeat the previous search command in the same or the opposite direction, respectively. Some characters have special meanings. These characters must be preceded by a backslash **(\)** to be included as part of the search expression.

| Character | Description |
|---|---|
| ^ | Searches at the beginning of the line (Use at the beginning of a search expression). |
| **.** | Matches a single character. |
| ***** | Matches zero or more of the previous character. |

| | |
|---|---|
| **$** | End of the line (Use at the end of the search expression). |
| **[** | Starts a set of matching or non-matching expressions. |
| < | This is put in an expression escaped with the backslash to find the ending or the beginning of a word. |
| > | This helps see the '<' character description above. |

The character search searches within one line to find a character entered after the command. The **f** and **F** commands search for a character on the current line only. **f** searches forwards and **F** searches backwards and the cursor moves to the position of the found character.

The **t** and **T** commands search for a character on the current line only, but for **t**, the cursor moves to the position before the character, and **T** searches the line backwards to the position after the character.

## SetCommands

You can change the look and feel of your vi screen using the following **:set** commands. Once you are in the command mode, type **:set** followed by any of the following commands.

| Command | Description |
|---|---|
| **:set ic** | Ignores the case when searching |
| **:set ai** | Sets autoindent |
| **:set noai** | Unsets autoindent |
| **:set nu** | Displays lines with line numbers on the left side |
| **:set sw** | Sets the width of a software tabstop. For example, you would set a shift width of 4 with this command — **:set sw=4** |
| **:set ws** | If *wrapscan* is set, and the word is not found at the bottom of the file, it will try searching for it at the beginning |
| **:set wm** | If this option has a value greater than zero, the editor will automatically "word wrap". For example, to set the wrap margin to two characters, you would type this: **:set wm=2** |
| **:set ro** | Changes file type to "read only" |

| | |
|---|---|
| **:set term** | Prints terminal type |
| **:set bf** | Discards control characters from input |

## Running Commands

The vi has the capability to run commands from within the editor. To run a command, you only need to go to the command mode and type **:!** command.

For example, if you want to check whether a file exists before you try to save your file with that filename, you can type **:! ls** and you will see the output of **ls** on the screen.

You can press any key (or the command's escape sequence) to return to your vi session.

## Replacing Text

The substitution command (**:s/**) enables you to quickly replace words or groups of words within your files. Following is the syntax to replace text −

```
:s/search/replace/g
```

The **g** stands for globally. The result of this command is that all occurrences on the cursor's line are changed.

## Important Points to Note

The following points will add to your success with vi −

- You must be in command mode to use the commands. (Press Esc twice at any time to ensure that you are in command mode.)
- You must be careful with the commands. These are case-sensitive.
- You must be in insert mode to enter text.

# UNIX – What is shell

A **Shell** provides you with an interface to the Unix system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.

Shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of a shell, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.

## Shell Prompt

The prompt, **$**, which is called the **command prompt**, is issued by the shell. While the prompt is displayed, you can type a command.

Shell reads your input after you press **Enter**. It determines the command you want executed by looking at the first word of your input. A word is an unbroken set of characters. Spaces and tabs separate words.

Following is a simple example of the **date** command, which displays the current date and time:

```
$date
Thu Jun 25 08:30:19 MST 2009
```

You can customize your command prompt using the environment variable PS1 explained in the Environment tutorial.

## Shell Types

In Unix, there are two major types of shells:

- **Bourne shell** — If you are using a Bourne-type shell, the **$** character is the default prompt.
- **C shell** — If you are using a C-type shell, the **%** character is the default prompt.

The Bourne Shell has the following subcategories –

- Bourne shell (sh)
- Korn shell (ksh)
- Bourne Again shell (bash)
- POSIX shell (sh)

The different C-type shells follow –

- C shell (csh)

- TENEX/TOPS C shell (tcsh)

The original Unix shell was written in the mid-1970s by Stephen R. Bourne while he was at the AT&T Bell Labs in New Jersey.

Bourne shell was the first shell to appear on Unix systems, thus it is referred to as "the shell".

Bourne shell is usually installed as **/bin/sh** on most versions of Unix. For this reason, it is the shell of choice for writing scripts that can be used on different versions of Unix.

In this chapter, we are going to cover most of the Shell concepts that are based on the Borne Shell.

## Shell Scripts

The basic concept of a shell script is a list of commands, which are listed in the order of execution. A good shell script will have comments, preceded by # sign, describing the steps.

There are conditional tests, such as value A is greater than value B, loops allowing us to go through massive amounts of data, files to read and store data, and variables to read and store data, and the script may include functions.

Shell scripts and functions are both interpreted. This means they are not compiled.

### Example Script

Assume we create a **test.sh** script. Note all the scripts would have the **.sh** extension. Before you add anything else to your script, you need to alert the system that a shell script is being started. This is done using the **shebang** construct. For example −

```
#!/bin/sh
```

This tells the system that the commands that follow are to be executed by the Bourne shell. *It's called a shebang because the # symbol is called a hash, and the ! symbol is called a bang.*

To create a script containing these commands, you put the shebang line first and then add the commands −

```
#!/bin/bash
pwd
ls
```

## Shell Comments

You can put your comments in your script as follows −

```
#!/bin/bash

# Author : Zara Ali
# Copyright (c) Tutorialspoint.com
# Script follows here:
pw
d ls
```

Save the above content and make the script executable –

```
$chmod +x test.sh
```

[

The shell script is now ready to be executed –

```
$./test.sh
```

Upon execution, you will receive the following result –

```
/home/amrood
index.htm   unix-basic_utilities.htm   unix-directories.htm test.sh
unix-communication.htm     unix-environment.htm
```

**Note:** To execute a program available in the current directory, use **./program_name**

## Extended Shell Scripts

Shell scripts have several required constructs that tell the shell environment what to do and when to do it. Of course, most scripts are more complex than the above one.

The shell is, after all, a real programming language, complete with variables, control structures, and so forth. No matter how complicated a script gets, it is still just a list of commands executed sequentially.

The following script uses the **read** command which takes the input from the keyboard and assigns it as the value of the variable PERSON and finally prints it on STDOUT.

```
#!/bin/sh

# Author : Zara Ali
# Copyright (c) Tutorialspoint.com
# Script follows here:

echo "What is your name?"
read PERSON
echo "Hello, $PERSON"
```

Here is a sample run of the script –

```
$./test.sh
What is your name?
Zara Ali
Hello, Zara Ali
$
```

# UNIX – File System Basics

A file system is a logical collection of files on a partition or disk. A partition is a container for information and can span an entire hard drive if desired.

Your hard drive can have various partitions which usually contain only one file system, such as one file system housing the **/file system** or another containing the **/home file system**.

One file system per partition allows for the logical maintenance and management of differing file systems.

Everything in Unix is considered to be a file, including physical devices such as DVD-ROMs, USB devices, and floppy drives.

## Directory Structure

Unix uses a hierarchical file system structure, much like an upside-down tree, with root (**/**) at the base of the file system and all other directories spreading from there.

A Unix filesystem is a collection of files and directories that has the following properties –

- It has a root directory (**/**) that contains other files and directories.
- Each file or directory is uniquely identified by its name, the directory in which it resides, and a unique identifier, typically called an **inode**.
- By convention, the root directory has an **inode** number of **2** and the **lost+found** directory has an **inode** number of **3**. Inode numbers **0** and **1** are not used. File inode numbers can be seen by specifying the **-i option** to **ls command**.
- It is self-contained. There are no dependencies between one filesystem and another.

The directories have specific purposes and generally hold the same types of information for easily locating files. Following are the directories that exist on the major versions of Unix –

| Directory | Description |
|---|---|
| **/** | This is the root directory which should contain only the directories needed at the top level of the file structure |
| **/bin** | This is where the executable files are located. These files are available to all users |
| **/dev** | These are device drivers |

| | |
|---|---|
| **/etc** | Supervisor directory commands, configuration files, disk configuration files, valid user lists, groups, ethernet, hosts, where to send critical messages |
| **/lib** | Contains shared library files and sometimes other kernel-related files |
| **/boot** | Contains files for booting the system |
| **/home** | Contains the home directory for users and other accounts |
| **/mnt** | Used to mount other temporary file systems, such as **cdrom** and **floppy** for the **CD-ROM** drive and **floppy diskette drive**, respectively |
| **/proc** | Contains all processes marked as a file by **process number** or other information that is dynamic to the system |
| **/tmp** | Holds temporary files used between system boots |
| **/usr** | Used for miscellaneous purposes, and can be used by many users. Includes administrative commands, shared files, library files, and others |
| **/var** | Typically contains variable-length files such as log and print files and any other type of file that may contain a variable amount of data |
| **/sbin** | Contains binary (executable) files, usually for system administration. For example, ***fdisk*** and ***ifconfig*** utlities |
| **/kernel** | Contains kernel files |

## Navigating the File System

Now that you understand the basics of the file system, you can begin navigating to the files you need. The following commands are used to navigate the system –

| Command | Description |
|---|---|
| **cat filename** | Displays a filename |
| **cd dirname** | Moves you to the identified directory |
| **cp file1 file2** | Copies one file/directory to the specified location |

| | |
|---|---|
| **file filename** | Identifies the file type (binary, text, etc) |
| **find filename dir** | Finds a file/directory |
| **head filename** | Shows the beginning of a file |
| **less filename** | Browses through a file from the end or the beginning |
| **ls dirname** | Shows the contents of the directory specified |
| **mkdir dirname** | Creates the specified directory |
| **more filename** | Browses through a file from the beginning to the end |
| **mv file1 file2** | Moves the location of, or renames a file/directory |
| **pwd** | Shows the current directory the user is in |
| **rm filename** | Removes a file |
| **rmdir dirname** | Removes a directory |
| **tail filename** | Shows the end of a file |
| **touch filename** | Creates a blank file or modifies an existing file or its attributes |
| **whereis filename** | Shows the location of a file |
| **which filename** | Shows the location of a file if it is in your PATH |

## ThedfCommand

The first way to manage your partition space is with the **df (disk free)** command. The command **df -k (disk free)** displays the **disk space usage in kilobytes**, as shown below

```
$df -k

Filesystem      1K-blocks      Used   Available Use% Mounted on
/dev/vzfs        10485760   7836644     2649116  75% /
```

```
/devices                0         0           0   0% /devices
$
```

Some of the directories, such as **/devices**, shows 0 in the kbytes, used, and avail columns as well as 0% for capacity. These are special (or virtual) file systems, and although they reside on the disk under **/**, by themselves they do not consume disk space.

The **df -k** output is generally the same on all Unix systems. Here's what it usually includes

| Column | Description |
|---|---|
| **Filesystem** | The physical file system name |
| **kbytes** | Total kilobytes of space available on the storage medium |
| **used** | Total kilobytes of space used (by files) |
| **avail** | Total kilobytes available for use |
| **capacity** | Percentage of total space used by files |
| **Mounted on** | What the file system is mounted on |

You can use the **-h (human readable) option** to display the output in a format that shows the size in easier-to-understand notation.

## The du Command

The **du (disk usage) command** enables you to specify directories to show disk space usage on a particular directory.

This command is helpful if you want to determine how much space a particular directory is taking. The following command displays number of blocks consumed by each directory. A single block may take either 512 Bytes or 1 Kilo Byte depending on your system.

```
$du /etc
10      /etc/cron.d
126     /etc/default
6       /etc/dfs
...
$
```

The **-h** option makes the output easier to comprehend –

```
$du -h /etc
5k     /etc/cron.d
63k    /etc/default
3k     /etc/dfs
...
$
```

## Mounting the File System

A file system must be mounted in order to be usable by the system. To see what is currently mounted (available for use) on your system, use the following command –

```
$ mount
/dev/vzfs on / type reiserfs (rw,usrquota,grpquota)
proc on /proc type proc (rw,nodiratime)
devpts on /dev/pts type devpts (rw)
$
```

The **/mnt** directory, by the Unix convention, is where temporary mounts (such as CD-ROM drives, remote network drives, and floppy drives) are located. If you need to mount a file system, you can use the mount command with the following syntax –

```
mount -t file_system_type device_to_mount directory_to_mount_to
```

For example, if you want to mount a **CD-ROM** to the directory **/mnt/cdrom**, you can type –

```
$ mount -t iso9660 /dev/cdrom /mnt/cdrom
```

This assumes that your CD-ROM device is called **/dev/cdrom** and that you want to mount it to **/mnt/cdrom**. Refer to the mount man page for more specific information or type mount **-h** at the command line for help information.

After mounting, you can use the cd command to navigate the newly available file system through the mount point you just made.

## Unmounting the File System

To unmount (remove) the file system from your system, use the **umount** command by identifying the mount point or device.

For example, **to unmount cdrom**, use the following command –

```
$ umount /dev/cdrom
```

The **mount command** enables you to access your file systems, but on most modern Unix systems, the **automount function** makes this process invisible to the user and requires no intervention.

## User and Group Quotas

The user and group quotas provide the mechanisms by which the amount of space used by a single user or all users within a specific group can be limited to a value defined by the administrator.

Quotas operate around two limits that allow the user to take some action if the amount of space or number of disk blocks start to exceed the administrator defined limits –

- **Soft Limit** – If the user exceeds the limit defined, there is a grace period that allows the user to free up some space.
- **Hard Limit** – When the hard limit is reached, regardless of the grace period, no further files or blocks can be allocated.

There are a number of commands to administer quotas –

| Command | Description |
|---|---|
| **quota** | Displays disk usage and limits for a user of group |
| **edquota** | This is a quota editor. Users or Groups quota can be edited using this command |
| **quotacheck** | Scans a filesystem for disk usage, creates, checks and repairs quota files |
| **setquota** | This is a command line quota editor |
| **quotaon** | This announces to the system that disk quotas should be enabled on one or more filesystems |

| | |
|---|---|
| **quotaoff** | This announces to the system that disk quotas should be disabled for one or more filesystems |
| **repquota** | This prints a summary of the disc usage and quotas for the specified file systems |