UNIT I

INTRODUCTION TO OPERATING SYSTEM

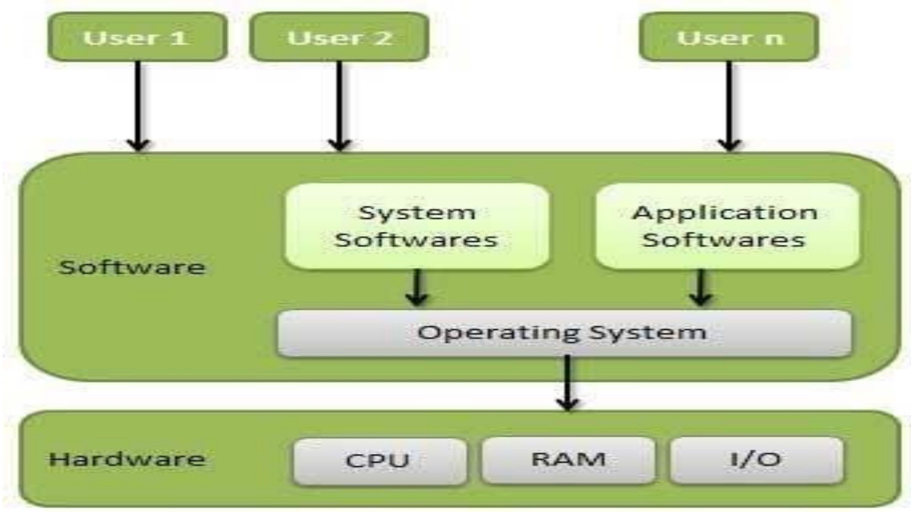
1.1 OVER VIEW OF OPERATING SYSTEMS:

- An Operating System (OS) is an interface between computer user and computer hardware.
- The operating system (OS) is the most important program that runs on a computer. Every general-purpose computer must have an operating system to run other programs and applications. Computer operating systems perform basic tasks, such as recognizing input from the keyboard, sending output to the display screen, keeping track of files and directories on the disk.
- The purpose of operating system is to provide an environment in which a user can execute programs in convenient and efficient manner.
- An operating system is 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.
- **Operating system** (**OS**) manages all of the **software** and **hardware** on the computer. Most of the time, there are several different computer programs running at the same time, and they all need to access your computer's **central processing unit (CPU)**, **memory**, and **storage**. The operating system coordinates all of this to make sure each program gets what it needs.
- Some popular Operating Systems include Linux, UNIX, Windows (XP, Vista, 7), Apple MacOS, IOS 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.

OS is the interaction point or communication point between users and hardware.

Physical parts of the computer are called hardware. A set of programs given to the computer is called software.



OPERATING SYSTEM MODEL

SYSTEM SOFTWARE: System software is a collection of programs designed to operate, control and extend the processing capabilities of the computer itself. Compiler and Interpreter etc are the examples of system software.

APPLICATION SOFTWARE: Application software is a set of programs designed for a particular application. Examples of Application software are: Microsoft word, Microsoft Excel.

FUNCTIONS OF OPERATING SYSTEMS:

Following are some of important functions of an operating System.

1) 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 in the main memory.
- A process is basically a program under execution

  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 is not in use?
- 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.

2) Processor Management:

The OS decides which process gets the processor (CPU) 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.

3) Device Management:

An Operating System does the following activities for device management −

- Keeps tracks of all devices. 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 efficient way.
- De-allocates devices.

4) File Management:

A file system is normally organized into directories for easy navigation and usage. An Operating System does the following activities for file management −

- Keeps track of information and location of files etc. The collective facilities are often known as **file system**.
- Creating & deleting files.
- Creating & deleting directories.

5) Other Important Activities:

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

- **Security** – OS prevents unauthorized access to programs and data by means of password and similar other techniques,
- **Control over system performance** – OS maintains control over system performance by recording delays between request for a service and response from the system.
- **Job accounting** −OS Keeps track of time and resources used by various jobs and users.
  CPU time, memory, secondary storage like hard disks, network throughput, battery power, external devices are all resources of a computer which an operating system manages.
- **Error detecting aids** − Operating System is responsible for the productions of dumps (unwanted data), error messages, error debugging and error detection aids.
- **Coordination between other softwares and users** − Coordination and assignment of compilers, interpreters, assemblers and other software to the various users of the computer systems is done by OS.

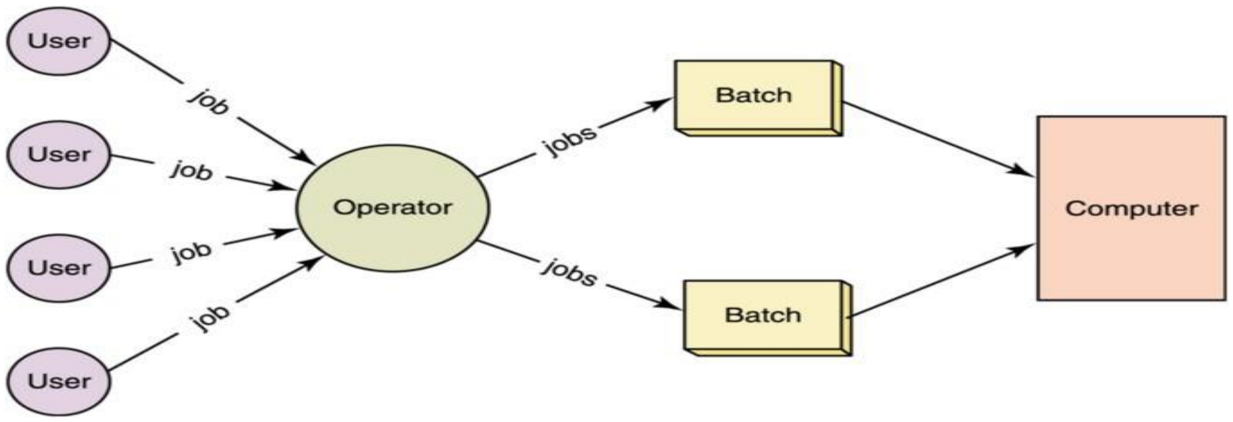
1.2 TYPES OF OPERATING SYSTEMS:

1) BATCH OPERATING SYSTEM:

Early computers were physically large machines. In these systems the user did not interact directly with the computer system. Instead the user prepared a job which consists of programming data and some control information and then submitted it to the computer operator and after some time the output would appear.

Batch processing is a technique in which an Operating System collects the programs and data together in a batch 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 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 for later printing or processing.
- To speed up processing, jobs with similar needs are batched together and run as a group.



BATCH OPERATING SYSTEM MODEL

Advantage

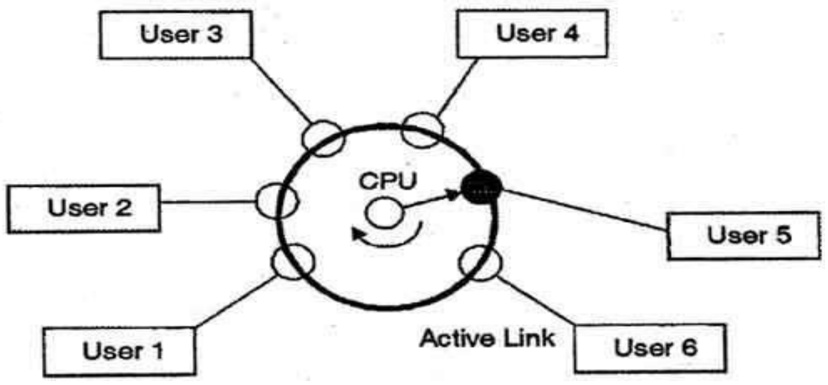
- Increased performance as a new job gets started as soon as the previous job is finished, without any manual intervention.

Disadvantages

- Due to lack of protection scheme, one batch job can affect pending jobs.
- Lack of interaction between the user and the computer system.

2) TIME-SHARING OPERATING SYSTEM:

- **Time-sharing is a technique which enables many people, located at various terminals, to use a particular computer system at the same time.**
- Processor's time which is shared among multiple users simultaneously is termed as time-sharing.
- The time sharing system is also known as Multi User System or Multitasking Operating System
- 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 **Operating System** 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 operating system allows the users to share the computer simultaneously.
- The OS handles multitasking in the way that it can handle multiple operations/executes multiple programs at a time.
- 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 to provide each user with a small portion of a time-shared CPU.
- CPU scheduling is a process which allows one process to use the CPU while the execution of another process is on hold (in waiting state) due to unavailability of any resource like I/O etc, thereby making full use of CPU. The aim of CPU scheduling is to make the system efficient, fast and fair.
- The time sharing system provides the direct access to a large number of users where CPU time is divided among all the users on scheduled basis.
- The OS allocates a set of time to each user.
- When this time is expired, it passes control to the next user on the system.
- This short period of time during that a user gets attention of the CPU is known as a time slice or *a* quantum.
- The concept of time sharing system is shown in figure.



- In above figure the user 5 is active but user 1, user 2, user 3, and user 4 are in waiting state whereas user 6 is in ready status.
- As soon as the time slice of user 5 is completed, the control moves on to the next readyuser i.e. user 6. In this state user 2, user 3, user 4, and user 5 are in waiting state and user 1 is in ready state.
- The process continues in the same way and so on.
- 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.

Advantages of Timesharing operating systems are as follows −

- Provides the advantage of quick response.
- Reduces CPU idle time.

Disadvantages of Time-sharing operating systems are as follows –

- The system must have memory management & protection, since several jobs are kept in memory at the same time.
- It provides mechanism for concurrent execution which requires complex CPU scheduling schemes.

3) DISTRIBUTED OPERATING SYSTEM:

- The motivation behind developing distributed operating systems is the availability of powerful and inexpensive microprocessors and advances in communication technology.
- These advancements in technology have made it possible to design and develop distributed systems comprising of many computers that are inter connected by communication networks. The main benefit of distributed systems is its low price.
- Distributed systems use multiple central processors to serve multiple real-time applications and multiple users.
- Data processing jobs are distributed among the processors accordingly.
- 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.
- Distributed OS is one where all of computers that are connected can share in tasks. For instance, if you and your friend are connected with each other using distributed OS then you can use a program which is actually on someone else's computer. LOCUS and MICROS are the best examples of distributed operating systems.

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.
- 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.

The disadvantages of distributed systems are as follows –

- Security problem due to sharing

4) 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.
- A typical configuration for a network operating system is a collection of personal computers along with a common printer, server and file server for archival storage, all tied together by a local network.
- Network OS is used to manage networked computers, means there would be a server and one or more computers will be managed by that server like in your college, you might have got one dedicated server and it will manage your individual computers or laptop.
- Some examples of network operating systems include Novell NetWare, Microsoft Windows NT, Microsoft Windows 2000, Microsoft Windows XP, Sun Solaris, Linux, etc

The advantages of network operating systems are as follows –

- 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.
- Regular maintenance and updates are required.

5) REAL TIME OPERATING SYSTEM:

- Real time Operating Systems are very fast and quick respondent systems. These systems are used in an environment where a large number of events must be accepted and processed in a short time.
- Real time processing requires quick transaction and characterized by supplying immediate response.
- The primary function of the real time operating system is to manage certain systemresources, such as the CPU, memory, and time. Each resource must be shared among the competing processes to accomplish the overall function of the system.
- A real-time operating system must have well-defined, fixed time constraints, otherwise the system will fail.
- Real-time operating system is designed for real-time applications, such as embedded systems, industrial robots, scientific research equipments and others.

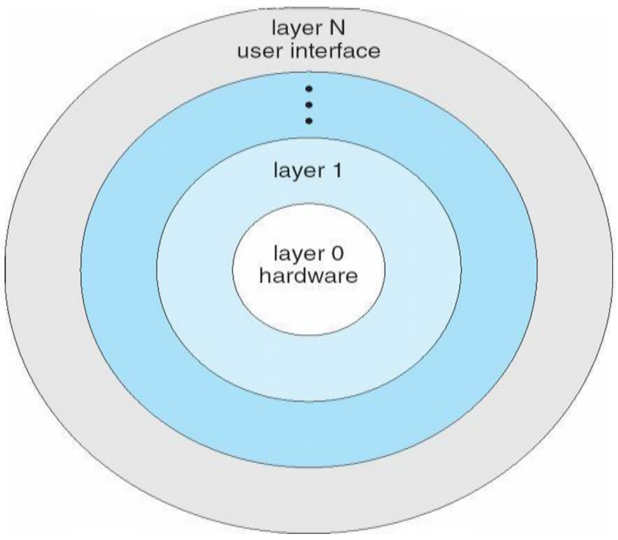
1.3 OPERATING SYSTEM STRUCTURES:

1) Simple Structure:

- There are several commercial systems that don‘t have a well- defined structure suchoperating systems begin as small, simple & limited systems and then grow beyond their original scope.
- MS-DOS (Microsoft Disk Operating System) is an example of such system. It was not divided into modules carefully. In MS-DOS, the interfaces and levels of functionality are not well separated.
- So MS-DOS was vulnerable to malicious programs, causing the entire system to crash when user programs fail.
- Another example of limited structuring is the UNIX operating system.

2) Layered Approach:

- In the layered approach, the OS is broken into a number of layers (levels) each built on top of lower layers. The bottom layer (layer 0) is the hardware & top most layer (layer N) is the user interface.
- The main advantage of the layered approach is modularity.
- The layers are selected such that each uses functions (or operations) & services of only lower layer.
- This approach simplifies debugging & system verification, i.e. the first layer can be debugged without concerning the rest of the system. Once the first layer is debugged, its correct functioning is assumed while the 2nd layer is debugged & so on.
- If an error is found during the debugging of a particular layer, the error must be on that layer because the layers below it are already debugged. Thus the design & implementation of the system are simplified when the system is broken down into layers.
- Each layer is implemented using only operations provided by lower layers. A layer doesn‘t need to know how these operations are implemented; it only needs to know what these operations do.
- The layer approach was first used in the operating system. It was defined in six layers.
- The major difficulty with the layered approach involves appropriately defining the various layers. Because a layer can use only lower-level layers, careful planning is necessary.
- It is less efficient than a non layered system.



Layers	Functions
5	User Program
4	I/O Management
3	Operator Process Communication
2	Memory Management
1	CPU Scheduling
0	Hardware

3) **Micro-kernel Approach:**

- A **kernel** is the central module of an operating system. It is the part of the operating system that loads first, and it remains in main memory. It manages the tasks of the **computer** and the hardware - most notably memory and CPU time.
- When the kernel became large and difficult to manage, in the mid-1980s, researchers at Carnegie Mellon University developed an operating system called **Mach** that modularized the kernel using the microkernel approach.
- This method structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs. The result is a smaller kernel. There is little confusion regarding which services should remain in the kernel and which should be implemented in user space.
- The main function of the microkernel is to provide a communication facility between the client program and the various services that are also running in user space.
- Communication is provided by *message passing*. For example if the client program wishes to access a file, it must interact with the file server. The client program and service never interact directly. Rather, they communicate indirectly by exchanging messages with the microkernel.
- One benefit of the microkernel approach is ease of extending the operating system.
- All new services are added to user space and consequently do not require modification of the kernel.
- When the kernel has to be modified, then the changes tend to be fewer, because the microkernel is a smaller kernel. The resulting operating system is easier to port from one hardware design to another.
- The microkernel also provided more security and reliability, since most services are running as user – rather than kernel – processes. If a service fails the rest of the operating system remains untouched.

1.4 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

1) **Program execution:** Operating systems handle many kinds of activities from user programs to system programs. Each of these activities is encapsulated as a process. A process includes the complete execution context (code to execute, data to manipulate). A process is program under execution. Following are the major activities of an **Operating System** with respect to program management –

- Loads a program into memory.
- Executes the program.
- Provides a mechanism for process communication.

2) **I/O Operation:** An I/O subsystem comprises of I/O devices and their corresponding driver software. 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.

3) **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 andusage. Following are the major activities of an **Operating System** with respect to file management –
   - 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.

4) **Communication:** Multiple processes communicate with one another through communication lines in the network. 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.

5) **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.

6) **Resource Management:** In 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.

7) **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.

1.5 SYSTEM CALLS:

- As we know that for performing any Operation a user must have to specify the Operation which he wants to operate on the Computer. We can say that For Performing any Operation a user must have to Request for a Service from the System.
- For Making any Request a user will prepare a Special call which is also known as the **System Call**.
- The System Call is the Request for running any Program and for performing any Operation on the System.
- When a user first time, starts the system then the system is in the user mode and when he request for a service then the user mode will be converted into the kernel mode which just listen the request of the user and process the request and display the results those are produced after the processing.
- When a user request for opening any folder or when he moves his mouse on the screen, then this is called as the system call which he is using for performing any operation.
- **System calls** provide an interface to the services made available by an operating system. These calls are generally available as routines written in C and C++.
- System Call is Programming interface to the services provided by the OS.
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call use.
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)
- System calls provide the interface between a process & the OS. These are usually available in the form of assembly language instruction.
- Some systems allow system calls to be made directly from a high level language program like C, BCPL and PERL etc.
- Systems calls occur in different ways depending on the computer in use. System calls can be roughly grouped into 5 major categories:
  1) Process Control
  2) File Manipulation
  3) Device Management
  4) Information maintenance
  5) Communication

1. Process Control:

- **End, abort:** A running program needs to be able to has its execution either normally (end) or abnormally (abort).
- **Load, execute:** A process or job executing one program may want to load and executes another program.
- **Create Process, terminate process:** There is a system call specifying for the purpose of creating a new process or job (create process or submit job). We may want to terminate a job or process that we created (terminates process, if we find that it is incorrect or no longer needed).
- **Get process attributes, set process attributes:** If we create a new job or process we should able to control its execution. This control requires the ability to determine & reset the attributes of a job or processes (get process attributes, set process attributes).
- **Wait time:** After creating new jobs or processes, we may need to wait for them to finish their execution (wait time).
- **Wait event, signal event:** We may wait for a specific event to occur (wait event). The jobs or processes then signal when that event has occurred (signal event).

2. File Manipulation:

- **Create file, delete file:** We first need to be able to create & delete files. Both the system calls require the name of the file & some of its attributes.
- **Open file, close file:** Once the file is created, we need to open it & use it. We close the file when we are no longer using it.
- **Read, write, reposition file:** After opening, we may also read, write or reposition the file (rewind or skip to the end of the file).
- **Get file attributes, set file attributes:** For either files or directories, we need to be able to determine the values of various attributes & reset them if necessary. Two system calls get file attribute & set file attributes are required for their purpose.

3. Device Management:

- **Request device, release device:** If there are multiple users of the system, we first request the device. After we finished with the device, we must release it.
- **Read, write, reposition:** Once the device has been requested & allocated to us, we can read, write & reposition the device.

4. Information maintenance:

- **Get time or date, set time or date:** Most systems have a system call to return the current date & time or set the current date & time.
- **Get system data, set system data:** Other system calls may return information about the system like number of current users, version number of OS, amount of free memory etc.
- **Get process attributes, set process attributes:** The OS keeps information about all its processes & there are system calls to access this information.

5. Communication:

- **Create, Delete Communication Connection:** Communication connection is created or deleted**.**
- **Send, Receive Messages:** Messages are sent or received between users.
- **Transfer Status Information:** Status information is transferred between users.

SYSTEM PROGRAMS:

System programs provide a convenient environment for program development & execution. They are divided into the following categories.

- **File manipulation:** These programs create, delete, copy, rename, print & manipulate files and directories.
- **Status information:** Some programs ask the system for date, time & amount of available memory or disk space, no. of users or similar status information.
- **File modification:** Several text editors are available to create and modify the contents of file stored on disk.
- **Programming language support:** compliers, assemblers & interpreters are provided to the user with the OS.
- **Programming loading and execution:** Once a program is assembled or compiled, it must be loaded into memory to be executed.
- **Application programs:** Most OS are supplied with programs that are useful to solvecommon problems or perform common operations. Ex: web browsers, word processors & text formatters etc.

1.6 VIRTUAL MACHINES:

- A virtual machine is a program that acts as a virtual computer.
- The fundamental idea behind a virtual machine is to abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate execution environment is running its own private computer.
- A virtual machine (VM) is a software program or operating system that not only exhibits the behaviour of a separate computer, but is also capable of performing tasks such as running applications and programs like a separate computer.
- A virtual machine, usually known as a guest is created within another computing environment referred as a "host." Multiple virtual machines can exist within a single host at one time.
- A virtual machine is also known as a guest.
- It runs on your current operating system – the “host” operating system – and provides virtual hardware to “guest” operating systems.
- The guest operating systems run in windows on your host operating system, just like any other program on your computer.
- The guest operating system runs normally, as if it were running on a physical computer. From the guest operating system’s perspective, the virtual machine appears to be a real, physical computer.
- Virtual machines provide their own virtual hardware, including a virtual CPU, memory, hard drive, network interface, and other devices.
- The virtual hardware devices provided by the virtual machines are mapped to real hardware on your physical machine.
- For example, a virtual machine’s virtual hard disk is stored in a file located on your hard drive.
- There are several reasons for creating a virtual machine, all of which are fundamentally related to being able to share the same hardware yet run several different execution environments (that is, different operating systems) concurrently.

Implementation: Although the virtual machine concept is useful, it is difficult to implement since much effort is required to provide an exact duplicate of the underlying machine. The CPU is being multiprogrammed among several virtual machines, which slows down thevirtual machines in various ways.

Advantages of virtual machines:

- Multiple OS environments can exist simultaneously on the same machine, isolated from each other;
- Virtual machines are widely available and are easy to manage and maintain.

Disadvantages of virtual machines:

- When multiple virtual machines are simultaneously running on a host computer, each virtual machine may introduce an unstable performance, which depends on the workload on the system by other running virtual machines;
- Virtual machine is not that efficient as a real one when accessing the hardware.

1.7OPERATING SYSTEM DESIGN AND IMPLEMENTATION:

VIEWS OF OPERATING SYSTEM:

Operating System is designed both by taking user view and system view into consideration.

Below is what the users and system thinks about Operating System.

1) User View:

- The goal of the Operating System is to maximize the work and minimize the effort of the user.
- Most of the systems are designed to be operated by single user; however in some systems multiple users can share resources, memory. In these cases Operating System is designed to handle available resources among multiple users and CPU efficiently.
- Operating System must be designed by taking both usability and efficient resource utilization into view.
- Operating System gives an effect to the user as if the processor is dealing only with the current task, but in background processor is dealing with several processes.
- The user view of the computer varies by the interface being used. The examples are - windows XP, vista, windows 7 etc.
- Most computer user sit in the in front of personal computer (pc), in this case the operating system is designed mostly for easy use with some attention paid to resource utilization.
- Some user sit at a terminal connected to a mainframe/minicomputer. In this case other users are accessing the same computer through the other terminals.
- There users share resources and may exchange the information. The operating system in this case is designed to maximize resources utilization to assume that all available CPU time, memory and I/O are used efficiently and no individual user takes more than his/her fair and share.
- The other users sit at workstations connected to network of other workstations and servers. These users have dedicated resources but they share resources such as networking and servers like file, compute and print server. Here the operating system is designed to compromise between individual usability and resource utilization.

2) System View:

- From the system point of view, Operating System is a program involved with the hardware.
- Operating System is allocator, which allocate memory, resources among various processes. It controls the sharing of resources among programs.
- It prevents improper usage of resources and handles errors.
- It is a program that runs all the time in the system in the form of Kernel.
- It controls application programs that are not part of Kernel. The kernel is the central module of an operating system (OS). It is the part of the operating system that loadsfirst, and it remains in main memory.
- From the computer point of view the operating system is the program which is most intermediate with the hardware.
- An operating system has resources as hardware and software which may be required to solve a problem like CPU time, memory space, file storage space and I/O devices and so on. That‘s why the operating system acts as manager of these resources.
- Another view of the operating system is it is a control program. A control programmanages the execution of user programs to present the errors in proper use of the computer. It is especially concerned of the user the operation and controls the I/O devices

DESIGN GOALS: The first problem in designing a system is to define goals and specifications. At the highest level, the design of the system will be affected by the choice of Hardware and the type of system: batch; time shared, network, distributed, real time, or general purpose. Beyond this highest design level, the requirements may be much harder to specify. The purpose of an operating system is to be provided an environment in which a user can execute programs. Its primary goal is to make the computer system convenient for the user. Its secondary goal is to use the computer hardware in efficient manner.

IMPLEMENTATION:

Once an operating system is designed, it must be implemented. Traditionally, operating systems have been written in assembly language. Now, however, they are most commonly written in higher-level languages such as C or C++.

CHAPTER-2

PROCESS MANAGEMENT

2.1 PROCESS CONCEPTS:

PROCESS:

- A process is basically a program under 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.
- A program is a passive entity, such as the contents of a file stored on disk whereas process is an active entity with a program counter specifying the next instruction to execute.
- A process is an instance of a computer program that is being executed. It contains the program code and its current activity.

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.

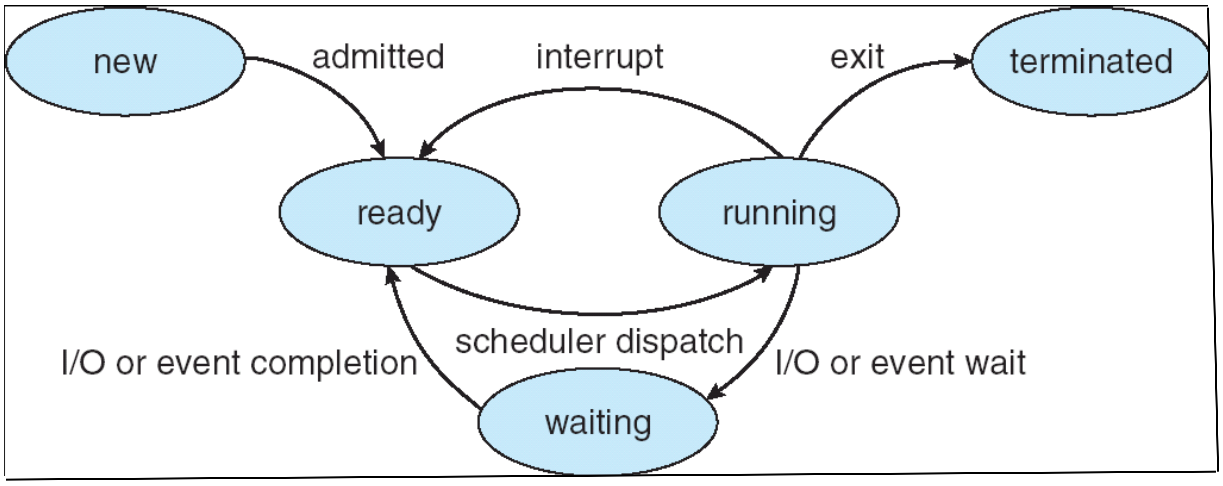
PROCESS LIFE CYCLE:

When a process executes, it passes through different states. The state of a process is defined in part by the current activity of that process.

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

1) **New:** The process is being created.
2) **Running:** Instructions are being executed.
3) **Waiting:** The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
4) **Ready:** The process is waiting to be assigned to a processor.
5) **Terminated:** The process has finished execution.

- I/O operation means read or write operation with any file or any specific I/O device.
- Scheduler dispatch- Allocates CPU to a process.



PROCESS LIFE CYCLE

1. A process switches from the ready state to the running state at scheduler dispatch
2. A process switches from the running state to the ready state when an interrupt occurs
3. A process switches from the running state to the waiting state as the result of an I/O request or event wait
4. A process switches from the waiting state to the ready state at completion of I/O or event
5. Finally a process terminates when it finishes its execution.

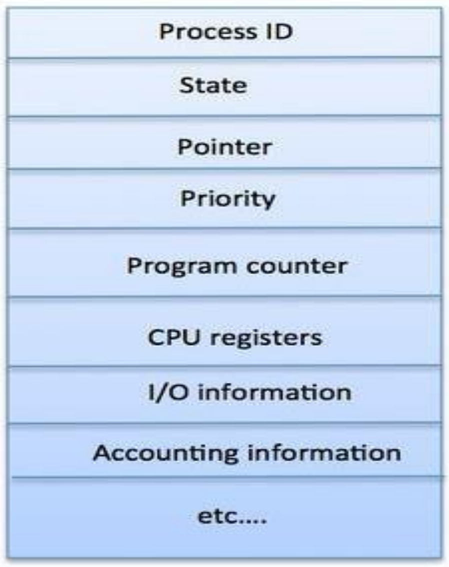
PROCESS CONTROL BLOCK (PCB):

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−

- **Process State**: The current state of the process i.e., whether it is ready, running, waiting, or whatever.
- **Process privileges**: This is required to allow/disallow access to system resources.
- **Process ID:** Unique identification for each of the process in the operating system.
- **Pointer**: A pointer to parent process.
- **Program Counter**: Program Counter is a pointer to the address of the next instruction to be executed for this process.
- **CPU registers**: Various CPU registers where process need to be stored for execution for running state.
- **CPU Scheduling Information**: Process priority and other scheduling information which is required to schedule the process.
- **Memory management information**: This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.
- **Accounting information**: This includes the amount of CPU used for process execution, time limits, execution ID etc.
- **I/O status information**: 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.
➢ The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates.

Here is a simplified diagram of a PCB:



2.2 OPERATIONS ON PROCESSES:

The processes in the system can execute concurrently, and they must be created and deleted dynamically. Thus, the operating system must provide a mechanism (or facility) for process creation and termination.

1) PROCESS CREATION:

- A process may create several new processes, via a create-process system call, during the course of execution.
- The creating process is called a parent process, whereas the new processes are called the children of that process.
- Each of these new processes may in turn create other processes.
- A new process is created by the forkO system call.
- In general, a process will need certain resources (such as CPU time, memory, files, I/O devices) to accomplish its task.
- When a process creates a sub process, that sub process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process.
- The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children.
- When a process is created it obtains in addition to the various physical and logical resources, initialization data (or input) that may be passed along from the parent process to the child process.

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

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

There are also two possibilities in terms of the address space of the new process:
1) The child process is a duplicate of the parent process (it has the same program and data as the parent).
2) The child process has a new program loaded into it.

2) PROCESS TERMINATION:

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit() system call.
- At that point, the process may return data (output) to its parent process (via the wait system call).
- All the resources of the process-including physical and virtual memory, open files, and I/O buffers-are de-allocated by the operating system.
- A process can cause the termination of another process via an appropriate system call (for example, abort).
- Usually, only the parent of the process that is to be terminated can invoke such a system call.
- Otherwise, users could arbitrarily kill each other's jobs. A parent therefore needs to know the identities of its children.
- Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.
- A parent may terminate the execution of one of its children for a variety of reasons, such as these:

1) The child has exceeded its usage of some of the resources that it has been allocated. This requires the parent to have a mechanism to inspect the state of its children.
2) The task assigned to the child is no longer required.
3) The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

- On such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated.

2.3 COOPERATING PROCESSES:

- The concurrent processes executing in the operating system may be either independent processes or cooperating processes.
- A process is independent if it cannot affect or be affected by the other processes executing in the system.
- Clearly, any process that does not share any data (temporary or persistent) with any other process is independent.
- On the other hand, a process is cooperating if it can affect or be affected by the other processes executing in the system.
- Clearly, any process that shares data with other processes is a cooperating process.

We may want to provide an environment that allows process cooperation for several reasons:

1) **Information sharing:** Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to these types of resources.
2) **Computation speedup:** If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Such a speedup can be achieved only if the computer has multiple processing elements (such as CPU’S or I/O channels).
3) **Modularity:** We may want to construct the system in a modular fashion, dividing the system functions into separate processes.
4) **Convenience:** Even an individual user may have many tasks to work at one time. For instance, a user may be editing, printing, and compiling in parallel.

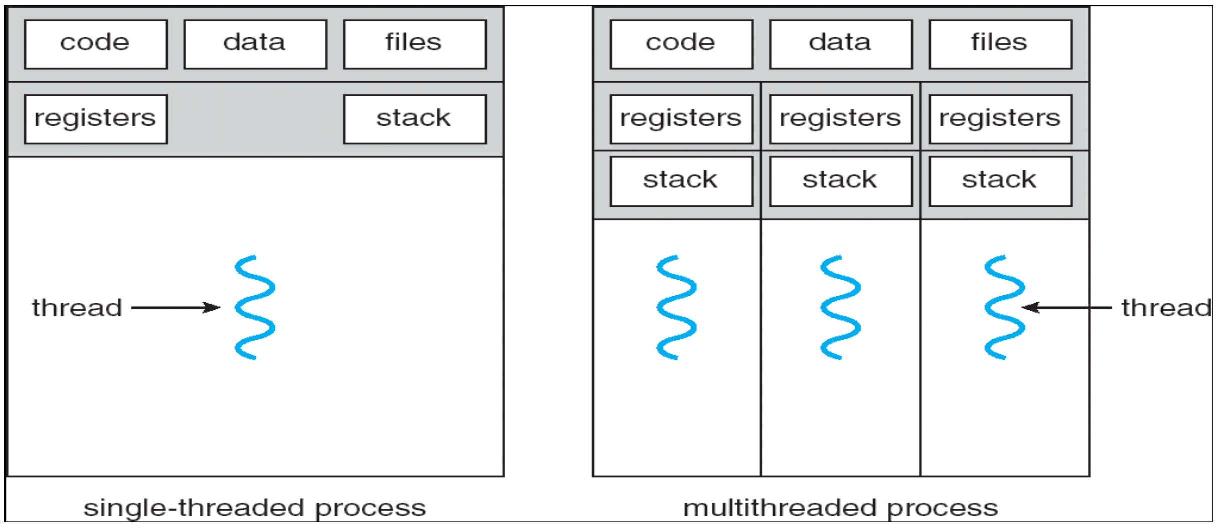
2.4 THREADS:

- A thread is a basic unit of CPU utilization. Thread comprises a thread ID, a program counter, a register set, and a stack.
- It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals. A traditional (or **heavyweight)** process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.
- 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 little information like code segment, data segment and open files.
- *A thread is a single sequence stream within in a process*. Because threads have some of the properties of processes, they are sometimes called *lightweight processes*. In a process, threads allow multiple executions of streams.
- 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 servers.
- Threads are used for small tasks, whereas processes are used for more 'heavyweight' tasks. Another difference between a thread and a process is that threads within the same process share the same address space, whereas different processes do not.

Difference between Process and Thread:

Process	Thread
Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
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.
If one process is blocked, then no other process can execute until the first processis unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Advantages of Thread:
The benefits of multithreaded programming can be broken down into four major categories:
1. Responsiveness: Multithreading in an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. For instance, a multithreaded web browser could still allow user interaction in one thread while an image was being loaded in another thread.
2. Resource sharing: By default, threads share the memory and the resources of the process to which they belong. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
3. Economy: Allocating memory and resources for process creation is costly. Because threads share resources of the process to which they belong, it is more economical to create and context-switch threads.
4. Utilization of multiprocessor architectures: The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors. A single threaded process can only run on one CPU, no matter how many are available. Multithreading on a multi-CPU machine increases concurrency.



SINGLE AND MULTITHREADED PROCESS

Example of Threads:
Let's suppose we have an online banking system, where people can log in and access their account information. Whenever someone logs in to their account online, they receive a separate and unique thread so that different bank account holders can access the central system simultaneously.

2.5 INTER PROCESS COMMUNICATION:

- Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information. There are two fundamental models of interprocess communication: shared memory system and message passing system.
- In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.
- In the message passing model, communication takes place by means of messages exchanged between the cooperating processes.

1) Shared-Memory System:

- Interprocess communication using shared memory requires communicating processes to establish a region of shared memory.
- Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment.
- Other processes that wish to communicate using this shared-memory segment must attach it to their address space.
- Recall that, normally, the operating system tries to prevent one process from accessing another process's memory.
- Shared memory requires that two or more processes agree to remove this restriction.
- They can then exchange information by reading and writing data in the shared areas.
- The form of the data and the location are determined by these processes and are not under the operating system's control.
- The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

2) Message-Passing System:

- The function of a message passing system is to allow processes to communicate with one another without the need to resort to shared data.
- Communication among the user processes is accomplished through the passing of messages.
- An IPC facility provides the two operations: send(message) and receive(message).
- Messages sent by a process can be of either fixed or variable size. If only fixed sized messages can be sent, the system-level implementation is straightforward.
- On the other hand, variable-sized messages require a more complex system-level implementation.
- If processes P and Q want to communicate, they must send messages to and receivemessages from each other; a communication link must exist between them.
- This link can be implemented in a variety of ways. We are concerned here not with the link's physical implementation (such as shared memory, hardware bus, or network), but rather with its logical implementation.

Naming: Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

1. Direct Communication: With direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send and receive primitives are defined as:

- Send (P, message)-Send a message to process P.
- Receive (Q , message) -Receive a message from process Q

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes.
- Exactly one link exists between each pair of processes.

2. Indirect Communication:

With indirect communication, the messages are sent to and received from mailboxes, or ports. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. In this scheme, a process can communicate with some other process via a number of different mailboxes. Two processes can communicate only if they share amailbox. The send and receive primitives are defined as follows:

- send (A, message) –Send a message to mailbox A.
- receive (A, message) -Receive a message from mailbox A.

In this scheme, a communication link has the following properties:

- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- A number of different links may exist between each pair of communicating processes, with each link corresponding to one mailbox.

2.6PROCESS 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.
- The act of determining which process in the ready state should be moved to the running state is known as Process Scheduling.

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** – Job queue consists of set of all processes in the system.
- **Ready queue** − Ready queue consists of set of all processes residing in main memory, ready and waiting to execute.
- **Device queues** − Device queue consists of set of processes waiting for an I/O device.

The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.).

Two-State Process Model:

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

1) **Running:** When a new process is created, it enters into the system as in the running state.

2) **Not Running:** 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.

SCHEDULERS: Schedulers are special system software which handles 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

1) **Long Term Scheduler:**
- It is also called a **job scheduler**. A long-term scheduler determines which programs are admitted to the system for processing.
- A long term scheduler or job scheduler selects processes from job pool (mass storage device, where processes are kept for later execution) and loads them into memory for execution. The long term scheduler controls the degree of multiprogramming (the number of processes in memory).
- Process loads into the memory for CPU scheduling.
- On some systems, the long-term scheduler may not be available or minimal.
- When a process changes the state from new to ready, then there is use of long-term scheduler.

2) **Short Term Scheduler:**
- It is also called as **CPU scheduler**.
- A short term scheduler or CPU scheduler selects from the main memory among the processes that are ready to execute and allocates the CPU to one of them.
- It is the change of ready state to running state of the process.
- 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.
- It provides lesser control over degree of multiprogramming.

3) **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.
- 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.
- The medium term scheduler available in all systems which is responsible for the swapping in and out operations which means loading the process into, main memory from secondary

memory (swap in) and take out the process from main memory and store it into the secondary memory (swap out).

- During extra load, this scheduler picks out big processes from the ready queue for some time, to allow smaller processes to execute, thereby reducing the number of processes in the ready queue.

Comparison among Schedulers:

S.No.	Long-Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
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 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.

2.7 SCHEDULING ALGORITHMS:

CPU Scheduling: All the processes which are ready to execute are placed in main memory and then selection of one of those processes is known as scheduling, and after selection that process gets the control of CPU.

Scheduling Criteria: The criteria for comparing CPU scheduling algorithms include the following:

- **CPU Utilization**: It means keeping the CPU as busy as possible.
- **Throughput**: It is nothing but the measure of work i.e., the number of processes that are completed per time unit.
- **Turnaround Time**: The interval from the time of submission of a process to the time of completion.
- **Waiting Time**: The sum of the periods spent waiting in the ready queue.
- **Response Time**: The time from the submission of a request until the first response is produced.

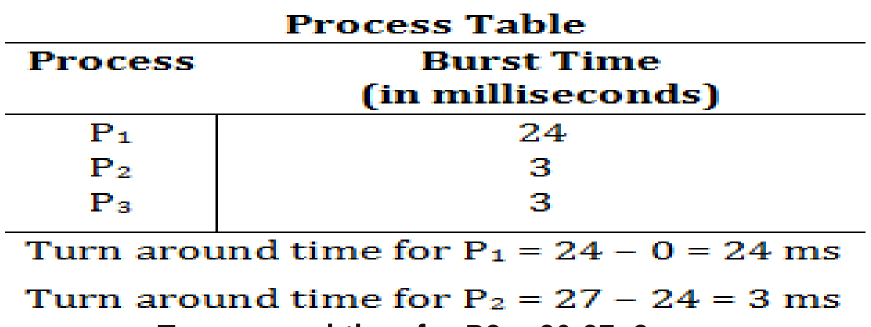
- **CPU burst time:** The duration for which a process gets control of the **CPU** and the concept of gaining control of the **CPU** is the **CPU burst**.

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. There are six popular process scheduling algorithms. 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.

1) First Come First Serve (FCFS) Scheduling:

- With this scheme, the process that requests the CPU first is allocated the CPU first.
- The implementation of the FCFS policy is easily managed with FIFO queue.
- When a process enters the ready queue, its PCB (Process Control Block) is linked onto the tail of the queue.
- When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.
- FCFS scheduling is non-preemptive, very simple, can be implemented with a FIFO queue, not a good choice when there are variable burst times.
- Drawback: causes short processes to wait for longer ones.



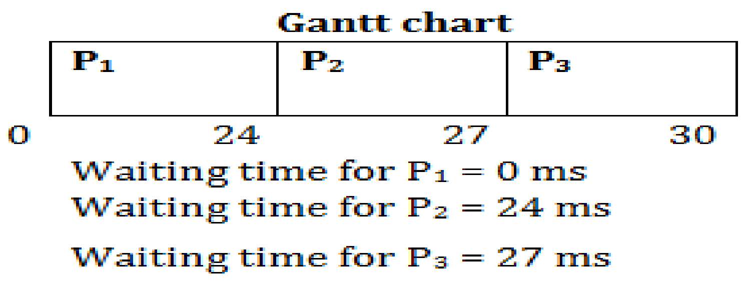
Process Table

Process	Burst Time (in milliseconds)
P_1	24
P_2	3
P_3	3

Turn around time for P_1 = 24 – 0 = 24 ms

Turn around time for P_2 = 27 – 24 = 3 ms

Turn around time for P3 = 30-27=3 ms



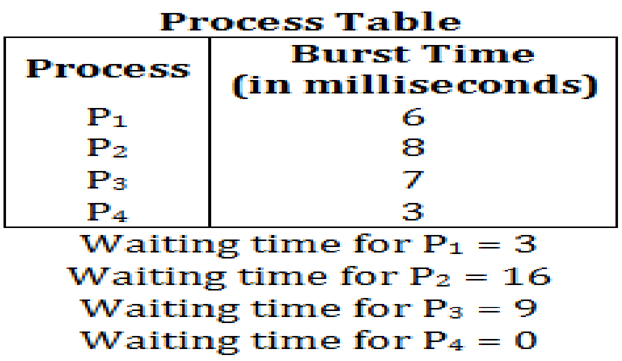
Waiting time for P_1 = 0 ms
Waiting time for P_2 = 24 ms

Waiting time for P_3 = 27 ms

Thus, the average waiting time is (0+ 24 + 27)/3 = 17 milliseconds.

2) Shortest Job First (SJF) Scheduling:

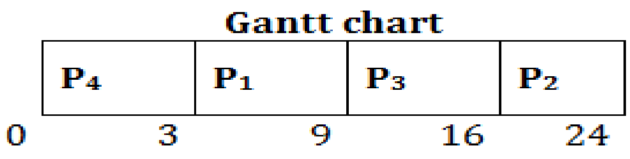
- When the CPU is available, it is assigned to the process that has the smallest next CPU burst.
- If the two processes have the same length or amount of next CPU burst, FCFS scheduling is used to break the tie.



Process Table

Process	Burst Time (in milliseconds)
P_1	6
P_2	8
P_3	7
P_4	3

Waiting time for P_1 = 3
Waiting time for P_2 = 16
Waiting time for P_3 = 9
Waiting time for P_4 = 0



Average waiting time

$$= \frac{3+16+9+0}{4} = 7 \text{ ms}$$

- A more appropriate term for this scheduling method would be the shortest next CPU burst algorithm because scheduling depends on the length of the next CPU burst of a process.
- The SJF algorithm can either be preemptive or non-preemptive.
- The choice arises when a new process arrives at the ready queue while a previous process is still executing.
- The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process.
- A preemptive SJF algorithm will preempt the currently executing process, whereas a, non-preemptive algorithm will allow the currently running process to finish its CPU burst.
- Preemptive SJF scheduling, is sometimes called shortest-remaining-time-first-scheduling.

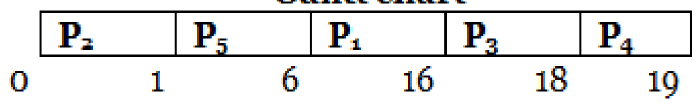
3) Priority Scheduling

- A priority is associated with each process and the CPU is allocated to the process with the highest priority and Equal priority processes are scheduled in FCFS order.
- We can be provided that low numbers represent high priority or low numbers represent low priority, According to the question, we need to assume anyone of the above.
- Priority scheduling can be either preemptive or non-preemptive.
- A preemptive priority scheduling algorithm will preempt the CPU, if the priority of the newly arrived process is higher than the priority of the currently running process.
- A non-preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

Process Table

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Gantt chart



Waiting time for P_1 = 6
Waiting time for P_2 = 0
Waiting time for P_3 = 16
Waiting time for P_4 = 18
Waiting time for P_5 = 1
Average waiting time

$$= \frac{6+0+16+18+1}{5} = 8.2 \text{ ms}$$

4) Round Robin Scheduling:

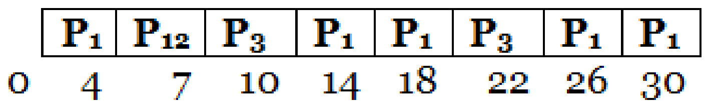
- The RR scheduling algorithm is designed especially for time sharing systems.
- It is similar to FCFS scheduling but preemption is added to switch between processes.
- A small unit of time called a time quantum or time slice is defined.
- If time quantum is too large, this just becomes FCFS.

Process Table

Process	Burst Time (in milliseconds)
P_1	24
P_2	3
P_3	3

Let's take time quantum = 4 ms. Then the resulting RR schedule is as Follows:

Gantt chart



P_1 waits for the 6 ms (10 – 4), P_2 waits for 4 ms and P_3 waits for 7 ms.

Thus, Average waiting time$= \frac{6+4+7}{3} = 5.66 \text{ ms}$

UNIT II

PROCESS SYNCHRONIZATION

- Process Synchronization means sharing system resources by processes in such a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data.
- On the basis of synchronization, processes are categorized as one of the following two types:
  1) **Independent Process:** Execution of one process does not affect the execution of other processes.
  2) **Cooperative Process**: Execution of one process affects the execution of other processes.
- Process synchronization problem arises in the case of Cooperative process because resources are shared in Cooperative processes.
- Co-operating process may either directly share a logical address space or be allotted to the shared data only through files or messages. This concurrent access is known as Process synchronization.

3.1 THE CRITICAL SECTION PROBLEM:

- Critical section is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one process. A critical section will usually terminate in fixed time, and a process will have to wait a until that fixed time.
- Critical section is a code segment that can be accessed by only one process at a time. Critical section contains shared variables which need to be synchronized to maintain consistency of data
- Consider a system consisting of n processes (P0, P1, ………Pn -1) each process has a segment of code which is known as critical section in which the process may be changing common variable, updating a table, writing a file and so on.
- The important feature of the system is that when the process is executing in its critical section no other process is to be allowed to execute in its critical section.
- The execution of critical sections by the processes is mutually exclusive.
- The critical section problem is to design a protocol that the process can use to co-operate.
- Each process must request permission to enter its critical section.
- The section of code implementing this request is the entry section.
- The critical section is followed on exit section.
- The remaining code is the remainder section.
- The general structure of a typical process Pi, is shown below.
- The entry section and exit section are enclosed in boxes to highlight these important segments of code.

General structure of a typical process Pi

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

A solution to the critical section problem must satisfy the following three conditions:

1. **Mutual Exclusion:** If process Pi is executing in its critical section then no any other process can be executing in their critical section.
2. **Progress:** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next.
3. **Bounded waiting:** There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

3.2 PETERSON’S SOLUTION:

- Peterson’s Solution is a classical software based solution to the critical section problem.
- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.
- The processes are named Pi and Pj. The Petersons solutions ask both the processes to share two data items.
- The two processes share two variables:
  - int turn;
  - Boolean flag[2];
- The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process Pi; is allowed to execute in its critical section. The flag array is used to indicate if a process is ready to enter its critical section. For example, if flag[i] is true, this value indicates that Pi; is ready to enter its critical section.

The structure of process Pi, in Peterson's solution:

```
do {
flag[i] = TRUE;
turn = j;
while ( flag[j] && turn == j);
CRITICAL SECTION
flag[i] = FALSE;
REMAINDER SECTION
} while (TRUE);
```

Peterson’s Solution preserves all three conditions:

- Mutual Exclusion is assured as only one process can access the critical section at any time.
- Progress is also assured, as a process outside the critical section does not block other processes from entering the critical section.
- Bounded Waiting is preserved as every process gets a fair chance.

Disadvantages of Peterson’s Solution:

- It involves Busy waiting (technique in which a process repeatedly checks to see if a condition is true).
- It is limited to 2 processes.

Solution to the critical-section problem using locks:

```
do {
//acquire lock
critical section
//release lock
remainder section
} while (TRUE);
```

3.3 SYNCHRONIZATION HARDWARE:

- Many systems provide hardware support for critical section code.
- The critical section problem could be solved easily in a single-processor environment if we could disallow interrupts to occur while a shared variable or resource is being modified.
- In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without pre-emption (The act of temporarily interrupting a task being carried out by a computer system).
- Unfortunately, this solution is not feasible in a multiprocessor environment.
- Disabling interrupt on a multiprocessor environment can be time consuming as the message is passed to all the processors.
- This message transmission lag, delays entry of threads into critical section and the system efficiency decreases.
- **TestAndSet:**
  TestAndSet is a hardware solution to the synchronization problem. In TestAndSet, we have a shared lock variable which can take either of the two values, 0 or 1.
  - 0 Unlock
  - 1 Lock
- Before entering into the critical section, a process inquires about the lock. If it is locked, it keeps on waiting till it become free and if it is not locked, it takes the lock and executes the critical section.
- **Mutual-exclusion implementation with TestAndSet ( )**

```
do {
while (TestAndSetLock(&lock) )
; // do nothing
// critical section
lock = FALSE;
// remainder section
} while (TRUE);
```

- In TestAndSet, Mutual exclusion and progress are preserved but bounded waiting cannot be preserved.

3.4 SEMAPHORES:

- In 1965, Dijkstra proposed a new and very significant technique for managing concurrent processes by using the value of a simple integer variable to synchronize the progress of interacting processes.
- Semaphore is a data structure that is used to make sure that multiple processes do not access a common resource or a critical section at the same time, in parallel programming environments. Semaphores are used to avoid dead locks.

- A semaphore is a variable whose value indicates the status of a common resource. Its purpose is to lock the resource being used. A process which needs the resource will check the semaphore for determining the status of the resource followed by the decision for proceeding. In multitasking operating systems, the activities are synchronized by using the semaphore techniques.
- This integer variable is called **semaphore**.
- So it is basically a synchronizing tool and is accessed only through two low standard atomic operations, wait and signal designated by P() and V() respectively.
- Mutual exclusion on the semaphore is enforced within P(S) and V(S). If a number of processes attempt P(S) simultaneously, only one process will be allowed to proceed & the other processes will be waiting. These operations are defined as under −

The classical definition of wait and signal are:

- **Wait:** Decrements the value of its argument S.

  The classical definition of **WAIT** is:

```
wait (s)
{
while (s <= 0);
s--;
}
```

- **Signal:** Increments the value of its argument S.

  The classical definition of the **SIGNAL** is:

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

Semaphores are of two types:

1) **Binary Semaphore:** Binary Semaphores have 2 methods associated with it (lock, unlock), When a resource is available, the process in charge set the semaphore to 1 else 0.
2) **Counting Semaphore:** A counting semaphore is an integer variable which may have value to be greater than one, typically used to allocate resources from a pool of identical resources.

Properties of Semaphores:

1. Simple to implement
2. Works with many processes
3. Can have many different critical sections with different semaphores
4. Each critical section has unique access to semaphores
5. Can permit multiple processes into the critical section at once, if desirable.

3.5 CLASSICAL PROBLEMS OF SYNCHRONIZATION:

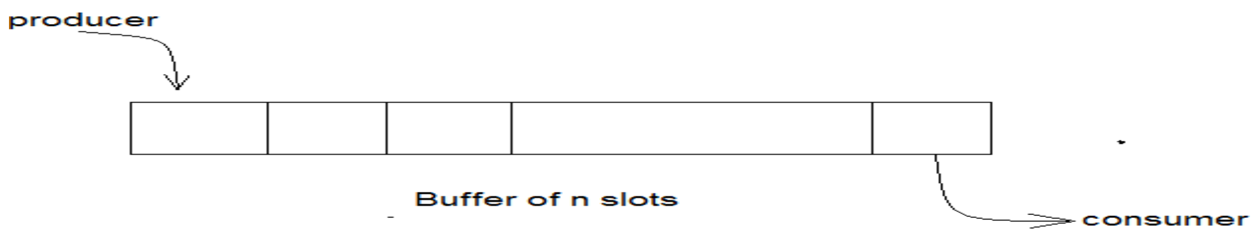
There are various types of problem which are proposed for synchronization scheme such as:

1) Bounded Buffer Problem:

Bounded buffer problem, which is also called **producer consumer problem**, is one of the classic problems of synchronization.

Problem Statement:

There is a buffer of **n** slots and each slot is capable of storing one unit of data. There are two processes running, namely, **producer** and **consumer**, which are operating on the buffer.



Bounded Buffer Problem:

A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently. There needs to be a way to make the producer and consumer work in an independent manner.

Solution:

One solution of this problem is to use semaphores. The semaphores which will be used here are:

- **mutex**, a binary semaphore which is used to acquire and release the lock.
- **empty**, a counting semaphore whose initial value is the number of slots in the buffer, since, initially all slots are empty.
- **full**, a counting semaphore whose initial value is 0.

At any instant, the current value of empty represents the number of empty slots in the buffer and full represents the number of occupied slots in the buffer.

Producer Operation:

The pseudo code of the **PRODUCER** function:

```
do {
wait(empty); // wait until empty>0 and then decrement 'empty'
wait(mutex); // acquire lock
 /* perform the insert operation in a slot */
signal(mutex); // release lock
signal(full);  // increment 'full'
} while(TRUE);
```

- Looking at the above code for a producer, we can see that a producer first waits until there is at least one empty slot.
- Then it decrements the **empty** semaphore because, there will now be one less empty slot, since the producer is going to insert data in one of those slots.

- Then, it acquires lock on the buffer, so that the consumer cannot access the buffer until producer completes its operation.
- After performing the insert operation, the lock is released and the value of **full** is incremented because the producer has just filled a slot in the buffer.

Consumer Operation:

The pseudo code of the **CONSUMER** function:

```
do {
wait(full); // wait until full>0 and then decrement ‘full’
wait(mutex);  // acquire the lock
 /* perform the remove operation in a slot */
 signal(mutex); // release the lock
 signal(empty); // increment ‘empty’
} while(TRUE);
```

- The consumer waits until there is at least one full slot in the buffer.
- Then it decrements the **full** semaphore because the number of occupied slots will be decreased by one, after the consumer completes its operation.
- After that, the consumer acquires lock on the buffer.
- Following that, the consumer completes the removal operation so that the data from one of the full slots is removed.
- Then, the consumer releases the lock.
- Finally, the **empty** semaphore is incremented by 1, because the consumer has just removed data from an occupied slot, thus making it empty.

2) Reader Writer Problem:

Reader writer problem is another example of a classic synchronization problem. There are many variants of this problem, one of which is examined below.

Problem Statement:

There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are **reader** and **writer**. Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource. When a **writer** is writing data to the resource, no other process can access the resource.A **writer** cannot write to the resource if there are non zero number of readers accessing the resource.

Solution:

From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currentlyaccessing that resource.

Here, we use one mutex **m** (for lock and unlock) and a semaphore **w**. An integer variable **read_count** is used to maintain the number of readers currently accessing the resource.The variable **read_count** is initialized to 0. A value of 1 is given initially to **m** and **w**.

Instead of having the process to acquire lock on the shared resource, we use the mutex **m** to make the process to acquire and release lock whenever it is updating the **read_count** variable.

The code for the **WRITER** process looks like this:

```
while(TRUE) {
wait(w);
/*perform the write operation */
signal(w);
}
```

The code for the **READER** process looks like this:

```
while(TRUE) {
wait(m);   //acquire lock
read_count++;
if(read_count == 1)
wait(w);
signal(m);  //release lock
 /* perform the reading operation */
 wait(m);   // acquire lock
 read_count--;
 if(read_count == 0)
 signal(w);
 signal(m);  // release lock
 }
```

Code Explained:

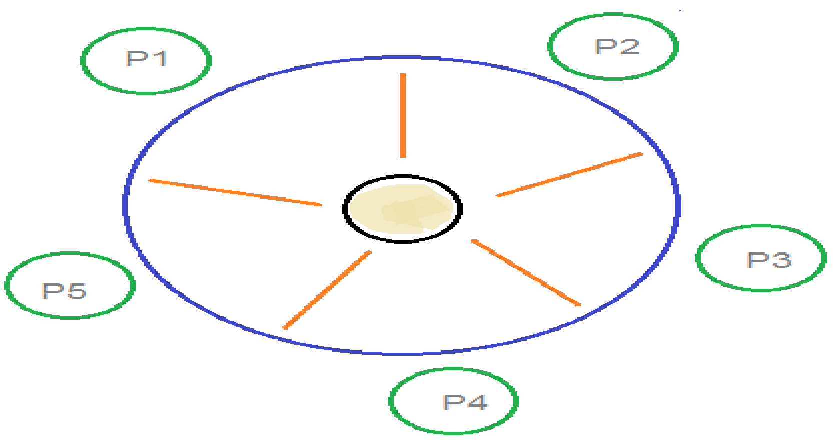
- As seen above in the code for the writer, the writer just waits on the **w** semaphore until it gets a chance to write to the resource.
- After performing the write operation, it increments **w** so that the next writer can access the resource.
- On the other hand, in the code for the reader, the lock is acquired whenever the **read_count** is updated by a process.
- When a reader wants to access the resource, first it increments the **read_count** value, then accesses the resource and then decrements the **read_count** value.
- The semaphore **w** is used by the first reader which enters the critical section and the last reader which exits the critical section.
- The reason for this is, when the first readers enters the critical section, the writer is blocked from the resource. Only new readers can access the resource now.
- Similarly, when the last reader exits the critical section, it signals the writer usingthe **w** semaphore because there are zero readers now and a writer can have the chance to access the resource.

3) Dining Philosopher Problem:

The dining philosopher's problem is another classic synchronization problem which is used to evaluate situations where there is a need of allocating multiple resources to multiple processes.

Problem Statement:

Consider there are five philosophers sitting around a circular dining table. The dining table has five chopsticks and a bowl of rice in the middle as shown in the below figure.



Dining Philosophers Problem:

At any instant, a philosopher is either eating or thinking. When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

Solution:

From the problem statement, it is clear that a philosopher can think for an indefinite amount of time. But when a philosopher starts eating, he has to stop at some point of time. The philosopher is in an endless cycle of thinking and eating.

An array of five semaphores, **stick[5]**, for each of the five chopsticks.

The code for each **PHILOSOPHER**:

```
while(TRUE) {
wait(stick[i]);
wait(stick[(i+1) % 5]);  // mod is used because if i=5, next
                 // chopstick is 1 (dining table is circular)
/* eat */
signal(stick[i]);
signal(stick[(i+1) % 5]);
/* think */
}
```

When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down.

But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever. The possible solutions for this are:

- A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.
- Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

3.6 CRITICAL REGIONS:

- According to the critical section problem using semaphore all processes must share a semaphore variable mutex which is initialized to one.
- Each process must execute wait (mutex) before entering the critical section and execute the signal (mutex) after completing the execution but there are various difficulties may arise with this approach like:

Case 1: Suppose that a process interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed, resulting in the following execution:

```
Signal (mutex);
. . . . . . . . . .
Critical Section
. . . . . . . . . .
Wait (mutex);
```

In this situation several processes may be executing in their critical sections simultaneously, which is violating mutual exclusion requirement.

Case 2: Suppose that a process replaces the signal (mutex) with wait (mutex). The execution is as follows:

```
Wait (mutex);
. . . . . . . . . . .
Critical Section
. . . . . . . . . . .
Wait (mutex);
```

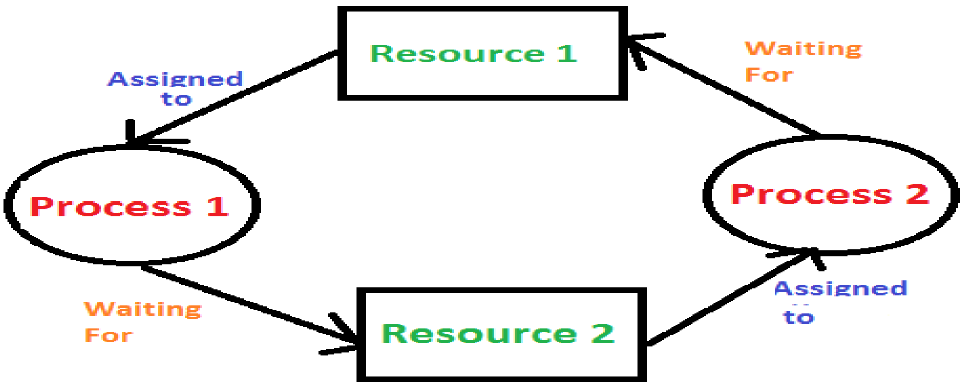
In this situation a deadlock will occur

Case 3: Suppose that a process omits the wait (mutex) and the signal (mutex). In this case the mutual exclusion is violated or a deadlock will occur.

UNIT II

DEADLOCKS

- In a multiprogramming environment several processes may compete for a finite number of resources.
- A process request resources, if the resource is available at that time a process enters the wait state.
- Waiting process may never change its state because the resources requested are held by other waiting process. This situation is known as deadlock.
- A deadlock is a situation in which two computer programs sharing the same resource are effectively preventing each other from accessing the resource, resulting in both programs ceasing to function.



DEADLOCK

Examples:

1) Process-1 requests the printer, gets it
Process-2 requests the scanner, gets it
Process-1 requests the scanner, waits
Process-2 requests the printer, waits, deadlocked!

2) A System has 2 disk drives. P1 and P2 each hold one disk drive and each needs another one.

4.1SYSTEM MODEL:

A system consists of a finite number of resources to be distributed among a number ofcompeting processes. The resources are partitioned into several types each of which consists of a number of identical instances. A process may utilize resources in the following sequence:

- **Request:** In this state one can request a resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
- **Use:** The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
- **Release:** In this state the process releases the resources.

4.2 DEADLOCK CHARACTERIZATION:

In a deadlock process never finish executing and system resources are tied up. A deadlock situation can arise if the following four conditions hold simultaneously in a system.

1) **Mutual Exclusion:** At a time only one process can use the resource. If another process requests that resource, requesting process must wait until the resource has been released.

2) **Hold and wait:** A process must be holding at least one resource and waiting to additional resource that is currently held by other processes.

3) **No Preemption:** Resources allocated to a process can‘t be forcibly taken out from it unless it releases that resource after completing the 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.

4.3 METHODS FOR HANDLING DEADLOCKS:

The problem of deadlock can deal with the following 3 ways:

1) We can use a protocol to prevent or avoid deadlock ensuring that the system will never enter to a deadlock state.
2) We can allow the system to enter a deadlock state, detect it and recover.
3) We can ignore the problem all together and pretend that deadlocks would never occur.

To ensure that deadlock never occur the system we can use either **deadlock prevention or deadlock avoidance scheme.**

Deadlock prevention provides a set of methods for ensuring that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.

- 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.

4.4 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 (Ex. read only files), on the other hand, do not require mutually exclusive access, and thus cannot be involved in a deadlock.

2) Hold and wait:

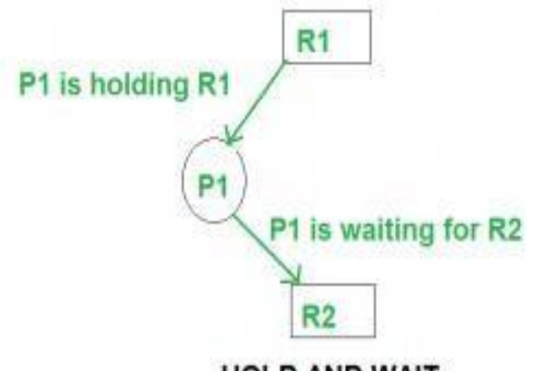
In order to ensure that hold and wait condition never holds in the system, we must guarantee that whenever a process requests a resource it does not hold any other resources. Two protocols can be used for this:

First protocol: Each process requests and be allocated all of its resources before it begins execution. This provision can be implemented by requiring that system calls requesting resources for a particular process precede any other system calls.

Second protocol: A process requests resources only when it 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.

Disadvantages for the above two protocols include:

Low resource utilization Starvation is possible.



HOLD AND WAIT

3) No Preemption:

In order to ensure that this condition does not hold, the following protocol may be used. If the process that is holding some resources requests another resource that cannot be immediately allocated to it (i.e. the process must wait), then all resources currently being held are preempted i.e. these resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting. The process will only be restarted when it can regain its old resources, as well as the new ones that it is requesting.

4) Circular Wait:

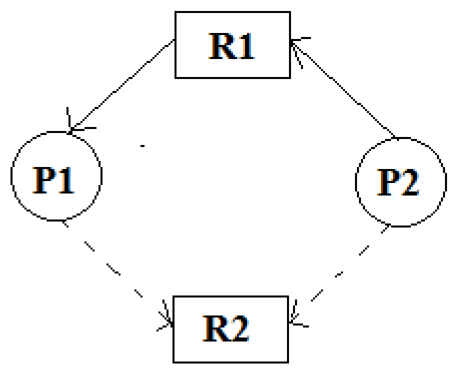
In order to ensure that the circular wait condition never holds, we may impose a total ordering of all resource types i.e. we assign to each resource type a unique integer number which allows us to compare two resources and determine whether one precedes another in our ordering.

4.5 DEADLOCK AVOIDANCE:

- Most prevention algorithms have poor resource utilization, and hence result in reduced throughputs.
- Instead, we can try to avoid deadlocks by making use prior knowledge about the usage of resources by processes including resources available, resources allocated, future requests and future releases by processes.
- Most deadlock avoidance algorithms need every process to tell in advance the maximum number of resources of each type that it may need.
- Based on all these information we may decide if a process should wait for a resource or not and thus avoid chances for circular wait.
- If a system is already in a safe state, we can try to stay away from an unsafe state and avoid deadlock.
- Deadlocks cannot be avoided in an unsafe state. A system can be considered to be in safe state if it is not in a state of deadlock and can allocate resources up to the maximum available.
- A safe sequence of processes and allocation of resources ensures a safe state.
- Deadlock avoidance algorithms try not to allocate resources to a process if it will make the system in an unsafe state.
- Since resource allocation is not done right away in some cases, deadlock avoidance algorithms also suffer from low resource utilization problem.

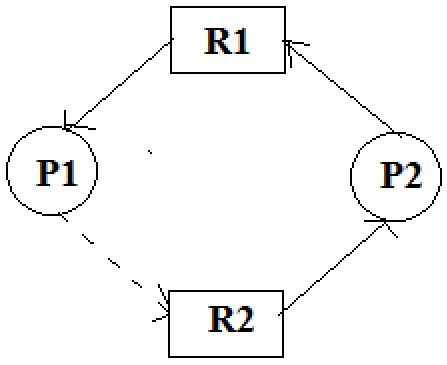
✓ A **resource allocation graph** is generally used to avoid deadlocks. If there are no cycles in the resource allocation graph, then there are no deadlocks. If there are cycles, there may be a deadlock. If there is only one instance of every resource, then a cycle implies a deadlock. Vertices of the resource allocation graph are resources and processes. The resource allocation graph has request edges and assignment edges. An edge from a process to resource is a request edge and an edge from a resource to process is an allocation edge. A calm edge denotes that a request may be made in future and is represented as a dashed line. Based on calm edges we can see if there is a chance for a cycle and then grant requests if the system will again be in a safe state.

Consider the image with calm edges as below:



Resource-allocation graph for deadlock avoidance

If R2 is allocated to p2 and if P1 request for R2, there will be a deadlock.



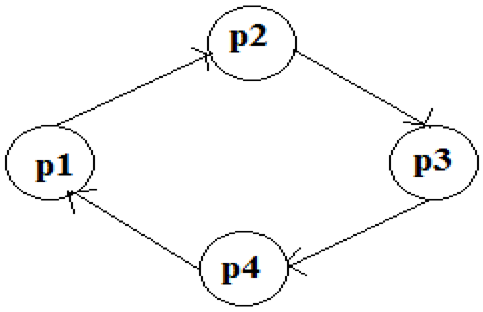
An unsafe state in a resource-allocation graph.

- ✓ The resource allocation graph is not much useful if there are multiple instances for a resource. In such a case, we can use **Banker's algorithm**. In this algorithm, every process must tell upfront the maximum resource of each type it need, subject to the maximum available instances for each type. Allocation of resources is made only, if the allocation ensures a safe state; else the processes need to wait. The Banker's algorithm can be divided into two parts: Safety algorithm if a system is in a safe state or not. The resource request algorithm make an assumption of allocation and see if the system will be in a safe state. If the new state is unsafe, the resources are not allocated and the data structures are restored to their previous state; in this case the processes must wait for the resource.

4.6 DEADLOCK DETECTION:

If deadlock prevention and avoidance are not done properly, as deadlock may occur and only things left to do is to detect the recover from the deadlock.

If all resource types has only single instance, then we can use a graph called wait-for-graph, which is a variant of resource allocation graph. Here, vertices represent processes and a directed edge from P1 to P2 indicates that P1 is waiting for a resource held by P2. Like in the case of resource allocation graph, a cycle in a wait-for-graph indicates a deadlock. So the system can maintain a **wait-for-graph** and check for cycles periodically to detect any deadlocks.



The wait-for-graph is not much useful if there are multiple instances for a resource, as a cycle may not imply a deadlock. In such a case, we can use an algorithm similar to Banker's algorithm to detect deadlock.

4.7 RECOVERY FROM DEADLOCKS:

When a detection algorithm determines that a deadlock exists, several alternatives exist. One possibility is to inform the operator that a deadlock has occurred, 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 deadlock cycle, but at a great expense; these processes may have computed for a long time, and the results of these partial computations must be discarded and probably recomputed later.

- **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 any 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 the deadlock cycle is broken. If preemption is required to deal with deadlocks, then three issues need to be addressed.

- **Selecting a victim:** Which resources and which processes are to be preempted?

- **Rollback:** If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must rollback the process to some safe state, and restart it from that state.

- **Starvation:** In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation that needs to be dealt with in any practical system. Clearly, we must ensure that a process can be picked as a victim only a small finite number of times. The most common solution is to include the number of rollbacks in the cost factor.