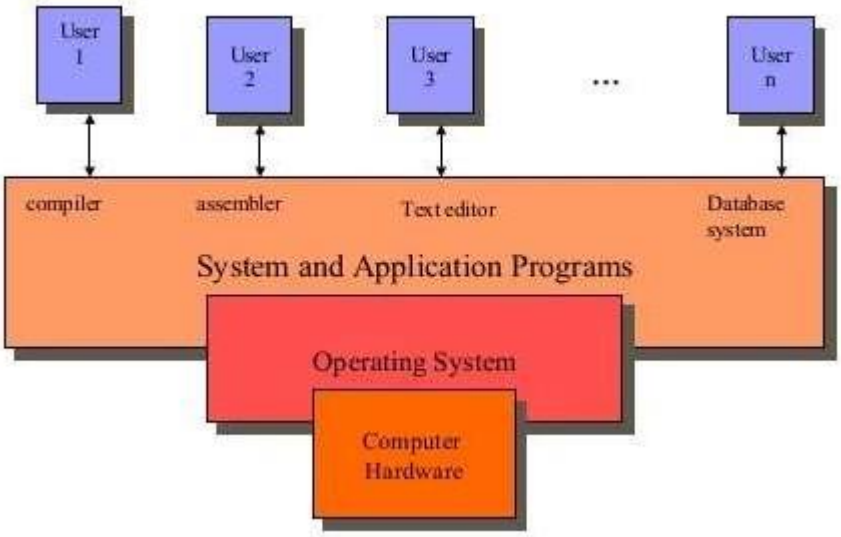
UNIT–1 INTRODUCTION TO OPERATING SYSTEM

Introduction of computer system

- Computer consists of the hardware, Operating System, system programs, application programs.
- The hardware consists of memory, CPU, ALU, I/O device, storage device and peripheral device.
- System program consists of compilers, loaders, editors, OS etc.
- Application program consists of database programs, business programs.
- Every computer must have an OS to run other programs.
- The OS controls & coordinates the use of the hardware among the various system programs and application programs for various tasks.
- It simply provides an environment within which other programs can do useful work.



OPERATING SYSTEM

Definition

- In the 1960's one might have defined OS as **"The software that controls the hardware"**.
- Operating System performs all the basic tasks like managing files, processes, and memory. Thus operating system acts as the manager of all the resources, i.e. **resource manager**.
- Operating system becomes an interface between the user and the machine. It is one of the most required software that is present in the device.
- Operating System is a type of software that works as an interface between the system program and the hardware.

Concept of OS

- The OS is a set of special programs that run on a computer system that allow it to work properly.
- It performs basic task as recognizing input from the keyboard, keeping track of files and directories on the disk, sending output to the display screen and controlling a peripheral device.
- The OS must support the following tasks. They are,
  - Provides the facilities to create, modification of program and data file using an editor.
  - Access to the compiler for translating the user program from high level language to machine language.
  - Provide a loader program to move the compiled program code to the computer memory for execution.

OS as a resource allocator

- OS keeps track of the status of each resource and decides who gets a resource, for how long and when.
- OS makes sure that different programs and users running at the same time do not interfere with each other.
- It is also responsible for security, ensuring that unauthorized users do not access the system.
- The primary objective of OS is to increase productivity of a processing resource such as computer hardware or user.
- The OS is the first program run on a computer when the computer boots up.

The OS acts as a manager of these resources and allocates them to specific programs and user as necessary for tasks.

OS can be explored from two view points:

1. The user view

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

- If the user is using a personal computer, the operating system is largely designed to make the interaction easy. Some attention is also paid to the performance of the system, but there is no need for the operating system to worry about resource utilization. This is because the personal computer uses all the resources available and there is no sharing.

- If the user is using a system connected to a mainframe or a minicomputer, the operating system is largely concerned with resource utilization. This is because there may be multiple terminals connected to the mainframe and the operating system makes sure that all the resources such as CPU, memory, I/O devices etc. are divided uniformly between them.
- If the user is sitting on a workstation connected to other workstations through networks, then the operating system needs to focus on both individual usage of resources and sharing though the network. This happens because the workstation exclusively uses its own resources but it also needs to share files etc. with other workstations across the network.
- If the user is using a handheld computer such as a mobile, then the operating system handles the usability of the device including a few remote operations. The battery level of the device is also taken into account.

There are some devices that contain very less or no user view because there is no interaction with the users. Examples are embedded computers in home devices, automobiles etc.

2. The system view

According to the computer system, the operating system is the bridge between applications and hardware. It is most intimate with the hardware and is used to control it as required.

The different types of system view for operating system can be explained as follows:

- The system views the operating system as a resource allocator. There are many resources such as CPU time, memory space, file storage space, I/O devices etc. that are required by processes for execution. It is the duty of the operating system to allocate these resources judiciously to the processes so that the computer system can run as smoothly as possible.
- The operating system can also work as a control program. It manages all the processes and I/O devices so that the computer system works smoothly and there are no errors. It makes sure that the I/O devices work in a proper manner without creating problems.
- Operating systems can also be viewed as a way to make using hardware easier.
- Computers were required to easily solve user problems. However it is not easy to work directly with the computer hardware. So, operating systems were developed to easily communicate with the hardware.

- An operating system can also be considered as a program running at all times in the background of a computer system (known as the kernel) and handling all the application programs. This is the definition of the operating system that is generally followed.

Names of OS

DOS, windows 3,windows 95/98, windows NT/2000, Unix, Linux etc.

Classification of OS

OS classified into two types:

1. **Character user interface (CUI) / Single user OS**
   - The user can interact with computer through commands.
   - We cannot use mouse here,
   - It is not user friendly,
   - If user knows the command only the user can interact with computer.
   - For example: DOS, Unix, Linux.

2. **Graphical user interface (GUI) / Multi user OS**
   - It is user friendly.
   - For example: Windows.

GOALS OF OS

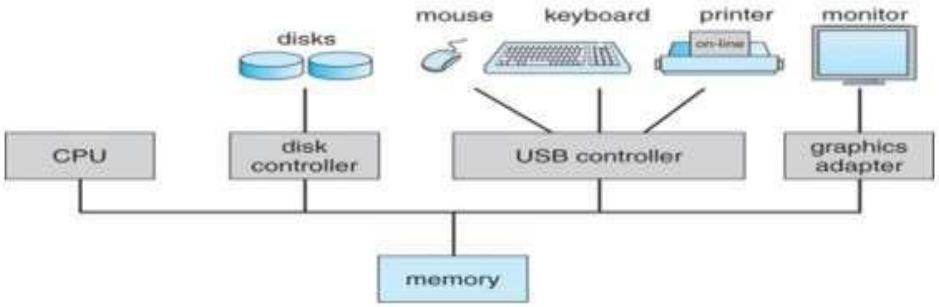
- Simplify the execution of user programs and make solving user problems easier.
- Use computer hardware efficiently.
- Make application software portable and versatile.
- Provide isolation, security and protection among user programs.
- Improve overall system reliability.

WHY SHOULD WE STUDY OPERATING SYSTEMS

- Need to understand interaction between the hardware and application.
- Need to understand basic principles in the design of computer systems.
- Increasing need for specialized operating systems. For example:
  - Real-time operating systems – aircraft control, multimedia services.
  - Embedded operating systems for devices – cell phones, sensors and controllers.

Computer System Organisation

The computer system is a combination of many parts such as peripheral devices, secondary memory, CPU etc. This can be explained more clearly using a diagram.



The salient points about the above figure displaying Computer System Organisation is −

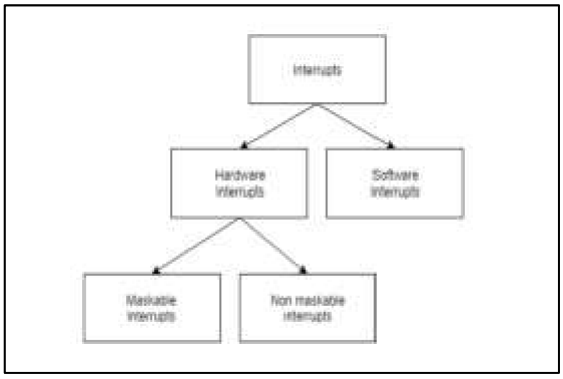
- Computer consists of processor, memory, and I/O components, with one or more modules of each type. These modules are connected through interconnection network.
- I/O devices and the CPU can execute concurrently.
- Each device controller is in charge of a particular device type.
- Each device controller has a local buffer.
- CPU moves data from/to main memory to/from local buffers.
- I/O is from the device to local buffer of controller.
- Device controller informs CPU that it has finished its operation by causing an interrupt.

Interrupt Handling

An interrupt is a necessary part of Computer System Organisation as it is triggered by hardware and software parts when they need immediate attention.

An interrupt can be generated by a device or a program to inform the operating system to halt its current activities and focus on something else.

Types of interrupts



Hardware and software interrupts are two types of interrupts. Hardware interrupts are triggered by hardware peripherals while software interrupts are triggered by software function calls.

Hardware interrupts are of further two types. Maskable interrupts can be ignored or disabled by the CPU while this is not possible for non-maskable interrupts.

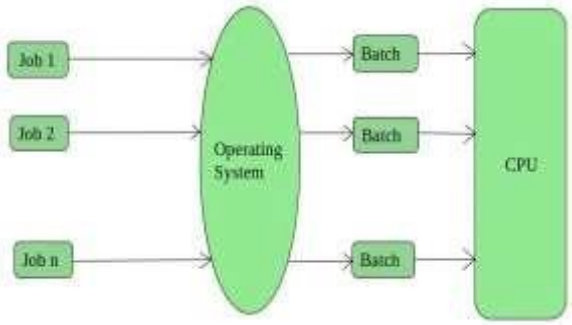
Types of Operating Systems

There are several types of Operating Systems which are mentioned below.

- Batch Operating System
- Multi-Programming System
- Multi-Processing System
- Multi-Tasking Operating System
- Time-Sharing Operating System
- Personal Computers
- Parallel Operating System
- Distributed Operating System
- Network Operating System
- Real-Time Operating System

1. Batch Operating System

This type of operating system does not interact with the computer directly. There is an operator which takes similar jobs having the same requirement and groups them into batches. It is the responsibility of the operator to sort jobs with similar needs.



Advantages

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

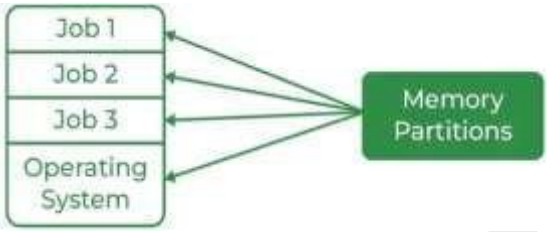
Disadvantages

- The computer operators should be well known with batch systems.
- Batch systems are hard to debug.
- It is sometimes costly.
- The other jobs will have to wait for an unknown time if any job fails.
- It is very difficult to guess or know the time required for any job to complete.

Examples

Payroll Systems, Bank Statements, etc.

2. Multi-Programming Operating System



Multiprogramming Operating Systems can be simply illustrated as more than one program is present in the main memory and any one of them can be kept in execution. This is basically used for better execution of resources.

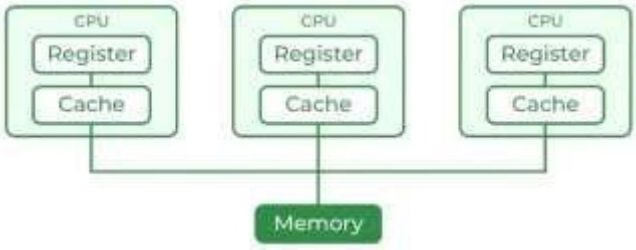
Advantages of Multi-Programming Operating System

- Multi Programming increases the Throughput of the System.
- It helps in reducing the response time.

Disadvantages of Multi-Programming Operating System

- There is not any facility for user interaction of system resources with the system.

3. Multi-Processing Operating System



It is a type of Operating System in which more than one CPU is used for the execution of resources. It betters the throughput of the System.

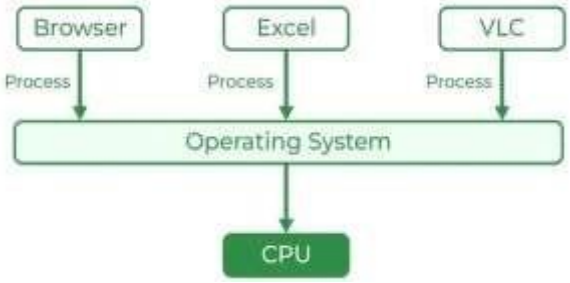
Advantages

- It increases the throughput of the system.
- As it has several processors, so, if one processor fails, we can proceed with another processor.

Disadvantages of Multi-Processing Operating System

- Due to the multiple CPU, it can be more complex and somehow difficult to understand.

4. Multi-Tasking Operating System



Multitasking Operating System is simply a multiprogramming Operating System with having facility of a Round-Robin Scheduling Algorithm. It can run multiple programs simultaneously. There are two types of Multi-Tasking Systems which are listed below.

Preemptive Multi-Tasking

The operating system can initiate a context switching from the running process to another process. In other words, the operating system allows stopping the execution of the currently running process and allocating the CPU to some other process.

Cooperative Multi-Tasking

The operating system never initiates context switching from the running process to another process. A context switch occurs only when the processes voluntarily yield control periodically or when idle or logically blocked to allow multiple applications to execute simultaneously. Also, in this multitasking, all the processes cooperate for the scheduling scheme to work.

Advantages

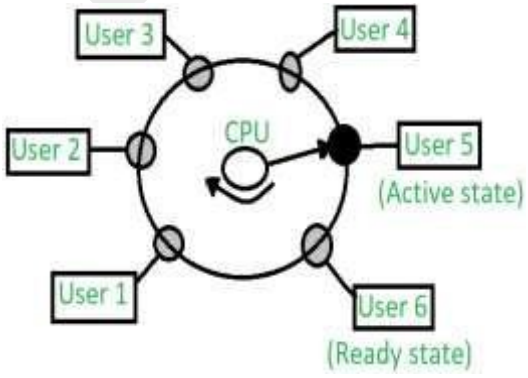
- Multiple Programs can be executed simultaneously in Multi-Tasking Operating System.
- It comes with proper memory management.

Disadvantages of Multi-Tasking Operating System

- The system gets heated in case of heavy programs multiple times.

5. Time-Sharing Operating Systems

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



Advantages

- Each task gets an equal opportunity.
- Fewer chances of duplication of software.
- CPU idle time can be reduced.
- **Resource Sharing:** Time-sharing systems allow multiple users to share hardware resources such as the CPU, memory, and peripherals, reducing the cost of hardware and increasing efficiency.
- **Improved Productivity:** Time-sharing allows users to work concurrently, thereby reducing the waiting time for their turn to use the computer. This increased productivity translates to more work getting done in less time.

- **Improved User Experience:** Time-sharing provides an interactive environment that allows users to communicate with the computer in real time, providing a better user experience than batch processing.

Disadvantages

- Reliability problem.
- One must have to take care of the security and integrity of user programs and data.
- Data communication problem.
- **High Overhead:** Time-sharing systems have a higher overhead than other operating systems due to the need for scheduling, context switching, and other overheads that come with supporting multiple users.
- **Complexity:** Time-sharing systems are complex and require advanced software to manage multiple users simultaneously. This complexity increases the chance of bugs and errors.
- **Security Risks:** With multiple users sharing resources, the risk of security breaches increases. Time-sharing systems require careful management of user access, authentication, and authorization to ensure the security of data and software.

Examples

- **IBM VM/CMS**: IBM VM/CMS is a time-sharing operating system that was first introduced in 1972. It is still in use today, providing a virtual machine environment that allows multiple users to run their own instances of operating systems and applications.
- **TSO (Time Sharing Option)**: TSO is a time-sharing operating system that was first introduced in the 1960s by IBM for the IBM System/360 mainframe computer. It allowed multiple users to access the same computer simultaneously, running their own applications.
- **Windows Terminal Services**: Windows Terminal Services is a time-sharing operating system that allows multiple users to access a Windows server remotely. Users can run their own applications and access shared resources, such as printers and network storage, in real-time.

6. Personal Computer

A personal computer (PC) is a microcomputer designed for use by one person at a time.

Prior to the PC, computers were designed for -- and only affordable for -- companies that attached terminals for multiple users to a single large mainframe computer whose resources were shared among all users. By the

1980s, technological advances made it feasible to build a small computer that an individual could own and use as a word processor and for other computing functions.

Whether they are home computers or business ones, PCs can be used to store, retrieve and process data of all kinds. A PC runs firmware that supports an operating system (OS), which supports a spectrum of other software. This software lets consumers and business users perform a range of general-purpose tasks, such as the following:

- word processing
- spreadsheets
- email
- instant messaging
- accounting
- database management
- internet access
- listening to music
- network-attached storage
- graphic design
- music composition
- video gaming
- software development
- network reconnaissance
- multimedia servers
- wireless network access hotspots
- video conferencing

Types

Personal computers fall into various categories, such as the following:

- **Desktop computers** usually have a tower, monitor, keyboard and mouse.
- **Tablets** are mobile devices with a touchscreen display.
- **Smartphones** are phones with computing capabilities.
- **Wearables** are devices users wear, such as smartwatches and various types of smart clothing.
- **Laptop computers** are portable personal computers that usually come with an attached keyboard and trackpad.
- **Notebook computers** are lightweight laptops.
- **Handheld computers** include advanced calculators and various gaming devices.

7. Parallel Operating System

Parallel Systems are designed to speed up the execution of programs by dividing the programs into multiple fragments and processing these fragments at the same time.

Advantages

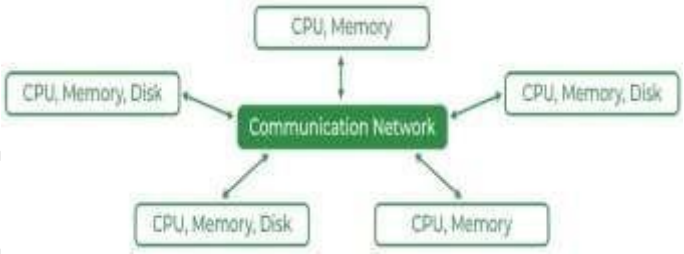
- **High Performance:** Parallel systems can execute computationally intensive tasks more quickly compared to single processor systems.
- **Cost Effective:** Parallel systems can be more cost-effective compared to distributed systems, as they do not require additional hardware for communication.

Disadvantages

- **Limited Scalability:** Parallel systems have limited scalability as the number of processors or cores in a single computer is finite.
- **Complexity:** Parallel systems are more complex to program and debug compared to single processor systems.
- **Synchronization Overhead:** Synchronization between processors in a parallel system can add overhead and impact performance.

8. Distributed Operating System

These types of operating systems are a recent advancement in the world of computer technology and are being widely accepted all over the world and, that too, at a great pace.



Various autonomous interconnected computers communicate with each other using a shared communication network. Independent systems possess their own memory unit and CPU. These are referred to as loosely coupled systems or distributed systems. These systems' processors differ in size and function.

The major benefit of working with these types of the operating system is that it is always possible that one user can access the files or software which are not actually present on his system but some other system connected within this network i.e., remote access is enabled within the devices connected in that network.

Types of Distributed Systems

The nodes in the distributed systems can be arranged in the form of client/server systems or peer to peer systems. Details about these are as follows −

Client/Server Systems

In client server systems, the client requests a resource and the server provides that resource. A server may serve multiple clients at the same time while a client is in contact with only one server. Both the client and server usually communicate via a computer network and so they are a part of distributed systems.

Peer to Peer Systems

The peer to peer systems contains nodes that are equal participants in data sharing. All the tasks are equally divided between all the nodes. The nodes interact with each other as required as share resources. This is done with the help of a network.

Advantages

- Failure of one will not affect the other network communication, as all systems are independent of each other.
- Electronic mail increases the data exchange speed.
- Since resources are being shared, computation is highly fast and durable.
- Load on host computer reduces.
- These systems are easily scalable as many systems can be easily added to the network.
- Delay in data processing reduces.

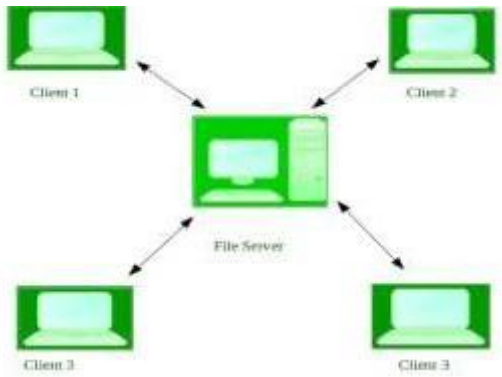
Disadvantages

- Failure of the main network will stop the entire communication.
- To establish distributed systems the language is used not well-defined yet.
- These types of systems are not readily available as they are very expensive. Not only that the underlying software is highly complex and not understood well yet.

Example: LOCUS

9. Network Operating System

These systems run on a server and provide the capability to manage data, users, groups, security, applications, and other networking functions. These types of operating systems allow shared access to files, printers, security, applications, and other networking functions over a small private network.



One more important aspect of Network Operating Systems is that all the users are well aware of the underlying configuration, of all other users within the network, their individual connections, etc. and that's why these computers are popularly known as tightly coupled systems.

Advantages

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

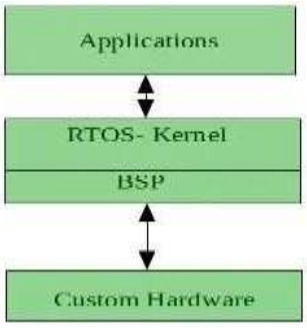
Disadvantages

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

Examples

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

10. Real-Time Operating System



These types of OSs serve real-time systems. The time interval required to process and respond to inputs is very small. This time interval is called **response time**.

Real-time systems are used when there are time requirements that are very strict like missile systems, air traffic control systems, robots, etc.

Types:

1. **Hard Real-Time Systems**

   Hard Real-Time OSs are meant for applications where time constraints are very strict and even the shortest possible delay is not acceptable. These systems are built for saving life like automatic parachutes or airbags which are required to be readily available in case of an accident. Virtual memory is rarely found in these systems.

2. **Soft Real-Time Systems**

   These OSs are for applications where time-constraint is less strict.

Advantages

- **Maximum Consumption:** Maximum utilization of devices and systems, thus more output from all the resources.

- **Task Shifting:** The time assigned for shifting tasks in these systems is very less. For example, in older systems, it takes about 10 microseconds in shifting from one task to another, and in the latest systems, it takes 3 microseconds.
- **Focus on Application:** Focus on running applications and less importance on applications that are in the queue.
- Real-time **operating system in** the **embedded system:** Since the size of programs is small, RTOS can also be used in embedded systems like in transport and others.
- **Error Free:** These types of systems are error-free.
- **Memory Allocation:** Memory allocation is best managed in these types of systems.

Disadvantages

- **Limited Tasks:** Very few tasks run at the same time and their concentration is very less on a few applications to avoid errors.
- **Use heavy system resources:** Sometimes the system resources are not so good and they are expensive as well.
- **Complex Algorithms:** The algorithms are very complex and difficult for the designer to write on.
- **Device driver and interrupt signals:** It needs specific device drivers and interrupts signal to respond earliest to interrupts.
- **Thread Priority:** It is not good to set thread priority as these systems are very less prone to switching tasks.

Examples

Scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots, air traffic control systems, etc.

Mainframe Systems

- First commercial systems: Enormous, expensive and slow.
- I/O: Punch cards and line printers.
- Single operator/programmer/user runs and debugs interactively:
- Standard library with no resource coordination
- Monitor that is always resident
- Inefficient use of hardware: poor *throughput* and poor *utilization*
- They initially executed one program at a time and were known as batch systems.

Throughput: Amount of useful work done per hour
Utilization: keeping all devices busy

Operating System Services

- **User Interface -** User interface is essential and all operating systems provide it. Users either interface with the operating system through command-line interface (CUI) or graphical user interface (GUI). Command interpreter executes next user-specified command. A GUI offers the user a mouse-based window and menu system as an interface.
- **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
- **I/O operations** - A running program may require I/O, which may involve a file or an I/O device.
- **File-system manipulation** - The file system is of particular interest. Obviously, programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.
- **Communications** – Processes may exchange information, on the same computer or between computers over a network. Communications may be via shared memory or through message passing (packets moved by the OS)
- **Error detection** – OS needs to be constantly aware of possible errors may occur in the CPU and memory hardware, in I/O devices, in user program. For each type of error, OS should take the appropriate action to ensure correct and consistent computing. Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system.

Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing

- **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them. Many types of resources such as CPU cycles, main memory, and file storage may have special allocation code, others such as I/O devices may have general request and release code.
- **Accounting -** To keep track of which users use how much and what kinds of computer resources
- **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other. **Protection** involves ensuring that all access to system resources is controlled. **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts. If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

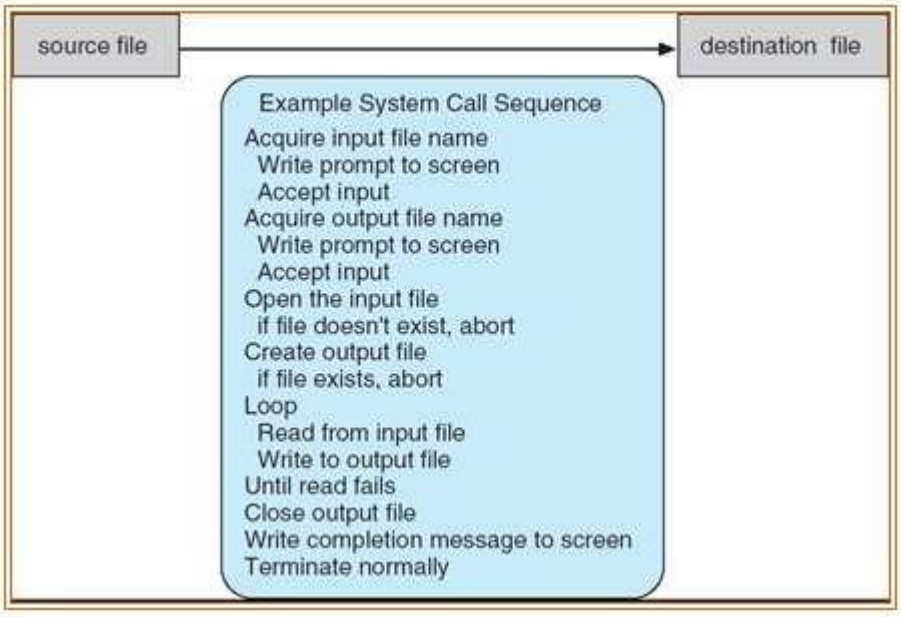
System Calls

- A system call is a way for a user program to interface with the operating system. The program requests several services, and the OS responds by invoking a series of system calls to satisfy the request.
- A system call can be written in assembly language or a high-level language like **C, C++** or **Pascal**.
- System calls are predefined functions that the operating system may directly invoke if a high-level language is used.
- A system call is a method for a computer program to request a service from the kernel of the operating system on which it is running.
- A system call is a method of interacting with the operating system via programs.
- A system call is a request from computer software to an operating system's kernel.
- A simple system call may take few nanoseconds to provide the result, like retrieving the system date and time. A more complicated system call, such as connecting to a network device, may take a few seconds. Most operating systems launch a distinct kernel thread for each system call to avoid bottlenecks. Modern operating systems are multi-threaded, which means they can handle various system calls at the same time.
- The **Application Program Interface (API)** connects the operating system's functions to user programs. It acts as a link between the operating system and a process, allowing user-level programs to request operating system services. The kernel system can only be accessed using system calls. System calls are required for any programs that use resources.

- When computer software needs to access the operating system's kernel, it makes a system call. The system call uses an API to expose the operating system's services to user programs. It is the only method to access the kernel system. All programs or processes that require resources for execution must use system calls, as they serve as an interface between the operating system and user programs.

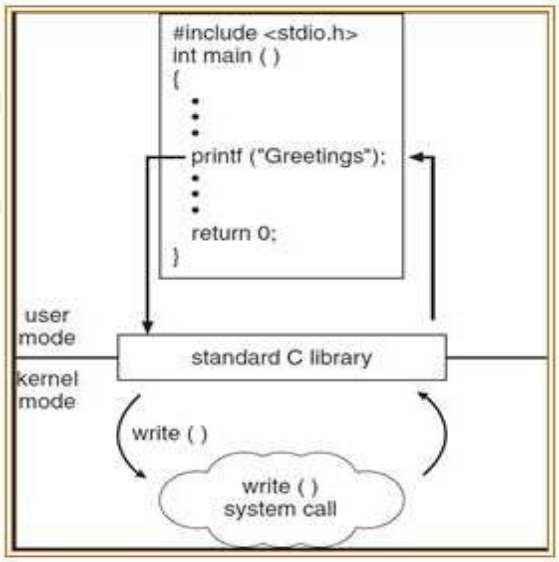
Example of System Calls

System call sequence to copy the contents of one file to another file



Standard C Library Example

C program invoking printf() library call, which call write() system call



There are various situations where we must require system calls in the operating system. Following of the situations are as follows:

1. It is must require when a file system wants to create or delete a file.
2. Network connections require the system calls to sending and receiving data packets.
3. If you want to read or write a file, you need to system calls.
4. If you want to access hardware devices, including a printer, scanner, you need a system call.
5. System calls are used to create and manage new processes.

Types of System Calls

There are commonly five types of system calls. These are as follows:

1. **Process Control**
2. **File Management**
3. **Device Management**
4. **Information Maintenance**
5. **Communication**

Process Control

Process control is the system call that is used to direct the processes. Some process control examples include creating, load, abort, end, execute, process, terminate the process, etc.

File Management

File management is a system call that is used to handle the files. Some file management examples include creating files, delete files, open, close, read, write, etc.

Device Management

Device management is a system call that is used to deal with devices. Some examples of device management include read, device, write, get device attributes, release device, etc.

Information Maintenance

Information maintenance is a system call that is used to maintain information. There are some examples of information maintenance, including getting system data, set time or date, get time or date, set system data, etc.

Communication

Communication is a system call that is used for communication. There are some examples of communication, including create, delete communication connections, send, receive messages, etc.

Examples of Windows and Unix system calls

Process	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	Fork() Exit() Wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	Open() Read() Write() Close()
Device Management	SetConsoleMode() ReadConsole() WriteConsole()	Ioctl() Read() Write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	Getpid() Alarm() Sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	Pipe() Shmget() Mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorgroup()	Chmod() Umask() Chown()

open()

The **open()** system call allows you to access a file on a file system. It allocates resources to the file and provides a handle that the process may refer to. Many processes can open a file at once or by a single process only. It's all based on the file system and structure.

read()

It is used to obtain data from a file on the file system. It accepts three arguments in general:

- A file descriptor.
- A buffer to store read data.
- The number of bytes to read from the file.

The file descriptor of the file to be read could be used to identify it and open it using **open()** before reading.

wait()

In some systems, a process may have to wait for another process to complete its execution before proceeding. When a parent process makes a child process, the parent process execution is suspended until the child process is finished. The **wait()** system call is used to suspend the parent process. Once the child process has completed its execution, control is returned to the parent process.

write()

It is used to write data from a user buffer to a device like a file. This system call is one way for a program to generate data. It takes three arguments in general:

- A file descriptor.
- A pointer to the buffer in which data is saved.
- The number of bytes to be written from the buffer.

fork()

Processes generate clones of themselves using the **fork()** system call. It is one of the most common ways to create processes in operating systems. When a parent process spawns a child process, execution of the parent process is interrupted until the child process completes. Once the child process has completed its execution, control is returned to the parent process.

close()

It is used to end file system access. When this system call is invoked, it signifies that the program no longer requires the file, and the buffers are flushed, the file information is altered, and the file resources are de-allocated as a result.

exec()

When an executable file replaces an earlier executable file in an already executing process, this system function is invoked. As a new process is not built, the old process identification stays, but the new process replaces data, stack, data, head, etc.

exit()

The **exit()** is a system call that is used to end program execution. This call indicates that the thread execution is complete, which is especially useful in multi-threaded environments. The operating system reclaims resources spent by the process following the use of the **exit()** system function.

Processes

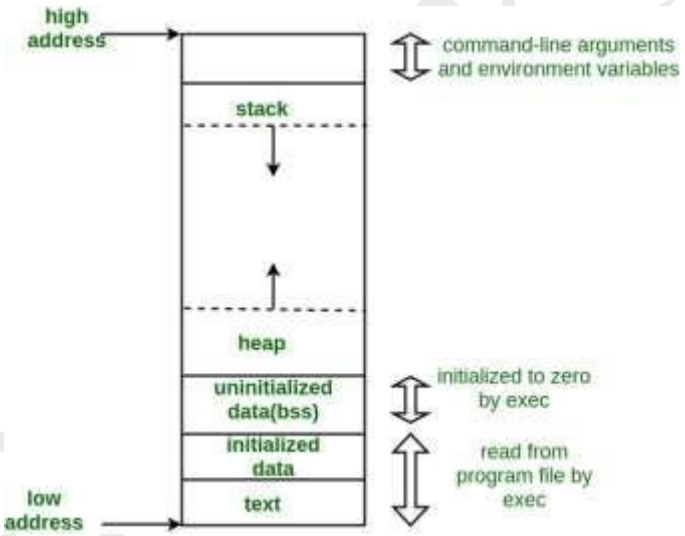
The term **"process"** was first used by the designers of the multics system in the 1960's. A process is a program in execution and process execution must progress in sequential fashion.

Process exits in a limited span of time.

Two or more process could be executing the same program, each using their own data and resource.

The process memory is divided into four sections for efficient operation:

- The **text category** is composed of integrated program code, which is read from fixed storage when the program is launched.
- The **data class** is made up of global and static variables, distributed and executed before the main action.
- Heap is used for flexible, or dynamic memory allocation and is managed by calls to new, delete, malloc, free, etc.
- The stack is used for local variables. The space in the stack is reserved for local variables when it is announced.



Process State

When process executes, it changes state. Process state is defined as the current activity of the process. Process state contains five states. Each process is one of the following states.

new: The process is being created.

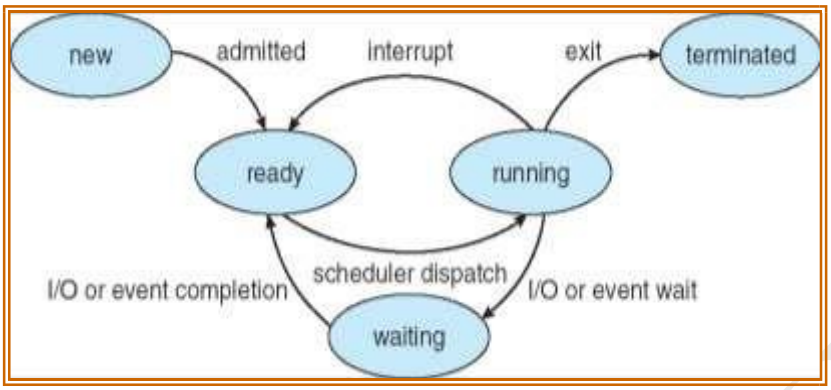
running: Instructions are being executed.

waiting: The process is waiting for some event to occur.

ready: The process is waiting to be assigned to a process.

terminated: The process has finished execution.

Diagram of Process State



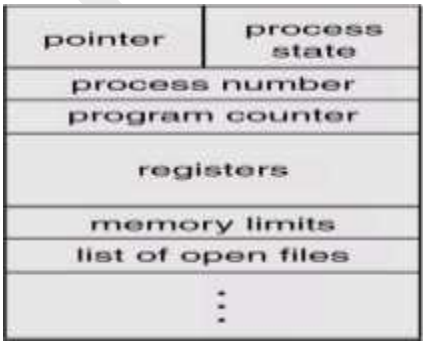
Dispatching

The assignment of the CPU to the first process on the ready list is called dispatching and is performed by a system entity called the Dispatcher.

Process Control Block (PCB)

The manifestation of a process in OS is a PCB or Process descriptor. Each process contains the PCB. PCB is a Data Structure containing certain important information about the process including,

- Unique identification of the process
- A pointer to the process's parent
- Process state
- Program counter
- CPU register
- Memory management information
- Account information



Pointer

Pointer points to the another PCB. Pointer is used for maintaining the scheduling list.

Process state

Process state may be new, ready, running, waiting and so on.

Program counter

It indicates the address of the next instruction to be executed.

CPU register

It includes general purpose register, stack pointer, and accumulators etc.

Memory management Informtion

locations including value of base and limit registers, page tables and other virtual memory information.

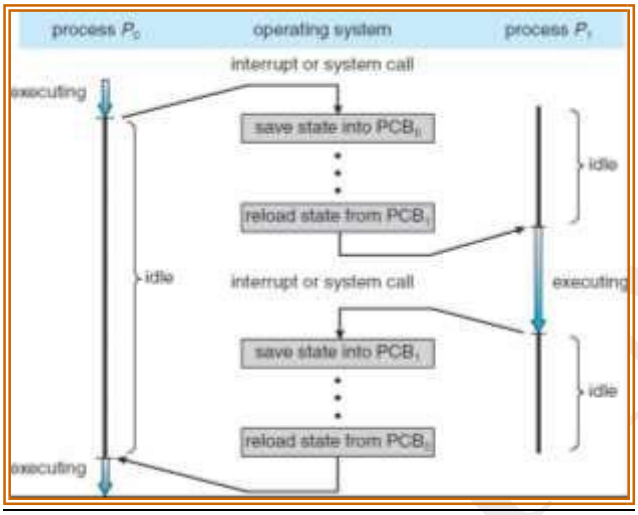
Accounting information

the amount of CPU and real time used, time limits, account numbers, job or process numbers etc.

I/O status information

List of I/O devices allocated to this process, a list of open files, and so on.

CPU Switch From Process to Process



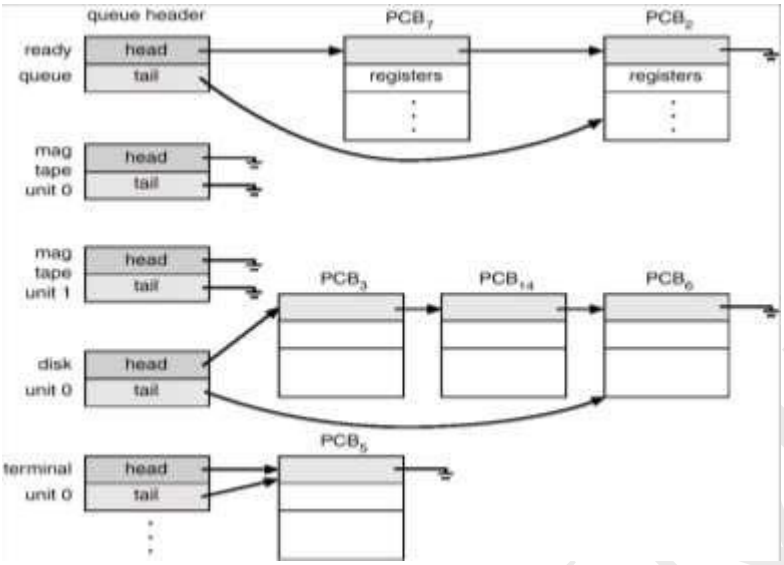
Process Scheduling Queues

Scheduling is to decide which process to execute and when. The objective of multi-program is, to have some process running at all times, so as to maximize CPU utilization. In Timesharing, switch the CPU frequently that users can interact with the program while it is running.

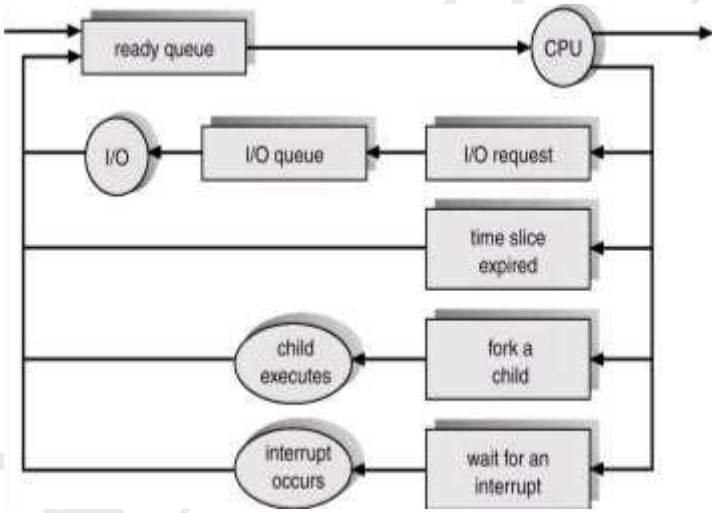
Scheduling Queues

1. **job queue** -The processes enter the system, they are put into a job queue. This Queue consists of all processes in the system.
2. **Ready queue** – set of all processes residing in main memory, and are ready and waiting to execute are kept on a list is called Ready Queue. This queue is generally stored as linked list. A ready queue header contains two pointers.(Head, Tail).
3. **Device queues** – set of processes waiting for an I/O device. Each device has its own queue.

Ready Queue and Various I/O Device Queues



Representation of Process Scheduling



A new process is initially put in the ready queue. It waits in the ready queue until it is selected for execution. Once a process is allocated CPU, the following events may occur.

- A process could issue an I/O request, and then be placed in an I/O queue.
- A process could create a new process
- The process could be removed forcibly from CPU, as a result of an interrupt and put back in the ready queue.
- When process terminates, it is removed from all queues.
- PCB and its other resources are de-allocated.

Schedulers

A process migrates between the various scheduling queues throughout its lifetime. The OS must select, for scheduling process. The selection process is carried out by a scheduler.

Long-term scheduler (or job scheduler)

- Selects which processes from this pool and loads them into memory for execution.
- It may take long time.
- The long-term scheduler executes less frequently.
- The long-term scheduler controls degree of multiprogramming.

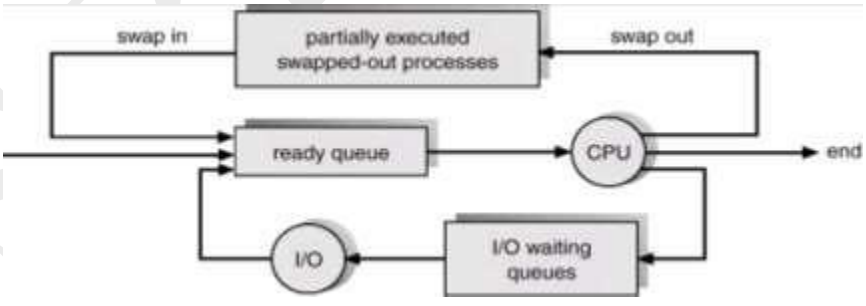
Short-term scheduler (or CPU scheduler)

- Selects which process should be ready to execute and allocates the CPU.
- The STS must select a new process for the CPU frequently.
- STS is executed at least once every 100 milliseconds.
- The STS must be fast.
- If it takes 10 milliseconds to decide to execute a process for 100 ms, then 9 % of CPU is used (or wasted) simply for scheduling work.

Medium-term scheduler

Some OS introduced a medium-term scheduler using swapping. It can be advantageous, to remove the processes from the memory and reduce the multiprogramming. At some later time, the process can be reintroduced into main memory and its execution can be continued when it left off. This scheme is called "**Swapping**".

Swapping improves the process mix (I/O and CPU), when main memory is unavailable.



- The long-term scheduler should make a careful selection. Because of the longer interval b/w executions, the LTS can afford to take more time to select a process for execution.
- The processes are either I/O bound or CPU bound.

- ❖ An I/O bound process spends more time doing I/O than it spends doing computation.
- ❖ A CPU bound process spends most of the time doing computation.
- ❖ The LT scheduler should select a good process mix of I/O-bound and CPU-bound processes.
- ❖ If all the processes are I/O bound, the ready queue will be empty.
- ❖ If all the processes are CPU bound, the I/O queue will be empty, the devices will go unused and the system will be unbalanced.
- ❖ Best performance by best combination of CPU-bound and I/O-bound process.

Context Switch

- ➢ Context switch is a task of switching the CPU to another process by saving the state of old process and loading the saved state for the new process.
- ➢ When a context switch occurs, the kernal saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.
- ➢ Context-switch time is overhead; the system does no useful work while switching.
- ➢ Context-switch time is highly dependent on hardware support.
- ➢ Typical range from 1 to 1000 microseconds.

Operations on Processes

The processes in the system can execute concurrently. The OS must provide a mechanism for creation and termination.

(i) Process creation

- A process may create several new processes.
- Processes are created and deleted dynamically.
- Process which creates another process is called a ***parent*** process; the created process is called a ***child*** process.
- Child process may create another sub process.
- Syntax for creating new process is: CREATE (process ID, attributes).
- A process needs certain resources (CPU time, memory, files, I/O devices) to accomplish its task.
- When a process creates a sub processes, that sub process may be to obtain its resource directly from the OS, or it may be constrained to a subset of resources of the parent process.
- When a process creates a new process, two possibilities in terms of **execution and resource sharing.**

  Resource sharing possibilities

  - Parent and children share all resources.
  - Children share subset of parent's resources.
  - Parent and child share no resources.

  Execution possibilities

  - Parent and children execute concurrently.
  - Parent waits until children terminate.
- There are also two possibilities in terms of the **address space** of the new process:
  - The child process is a duplicate of the parent process.
  - Child process has a program loaded into it.

Example

In UNIX:

- Each process is identified by its process identifier.
- **fork** system call creates new process.
- **exec** system call used after a **fork** to replace the process' memory space with a new program.
- The new process is a copy of the original process.
- The exec system call is used after a fork by one of the two processes to replace the process memory space with a new program.

DEC VMS:

- Creates a new process, loads a specified program into that process, and starts it running.

WINDOWS NT supports both models:

- Parent address space can be duplicated or
- parent can specify the name of a program for the OS to load into the address space of the new process.

(ii) Process Termination

- ❖ Process executes last statement and asks the operating system to decide it (**exit**).
- ❖ Output data from child to parent (via **wait**).
- ❖ Process' resources are de-allocated by operating system.
- ❖ Parent may terminate the execution of children processes (**abort**).
- ❖ Child has exceeded allocated resources.
- ❖ Task assigned to child is no longer required.
- ❖ Parent is exiting, Operating system does not allow child to continue if its parent terminates.
- ❖ Cascading termination. (All children terminated).

COOPERATING PROCESSES

- The concurrent process executing in the OS may be either independent process or cooperating process.
- Independent process **cannot** affect or be affected by the execution of another process.
- Cooperating process **can** affect or be affected by the execution of another process.

Advantages of process cooperation

1. **Information sharing:** several users may be interest in the same piece of information.
2. **Computation speed-up:** If we want a particular task to run faster, we must break it into subtasks and run in parallel.
3. **Modularity:** Constructing the system in modular fashion, dividing the system functions into separate process.
4. **Convenience:** User will have many tasks to work in parallel (Editing, compiling, printing).

Processes can communicate with each other through both:

- Shared Memory
- Message passing

The following figure shows a basic structure of communication between processes via the shared memory method and via the message passing method.

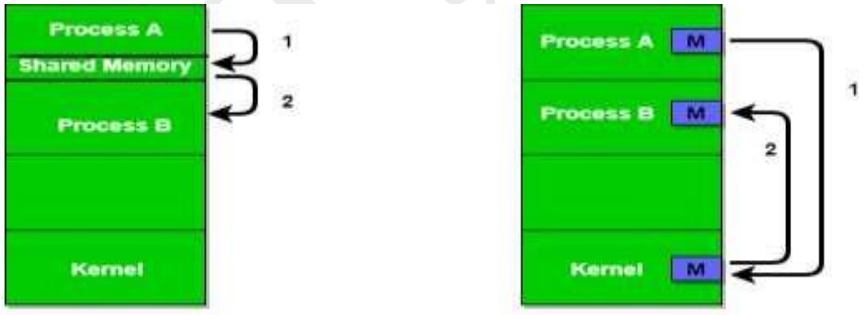


Figure 1 - Shared Memory and Message Passing

(i) Shared Memory

Communication between processes using shared memory requires processes to share some variable, and it completely depends on how the programmer will implement it.

One way of communication using shared memory can be imagined like this: Suppose process1 and process2 are executing simultaneously, and they share

some resources or use some information from another process. Process1 generates information about certain computations or resources being used and keeps it as a record in shared memory. When process2 needs to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly.

Processes can use shared memory for extracting information as a record from another process as well as for delivering any specific information to other processes.

Ex: Producer-Consumer problem

A producer process produces information that is consumed by a consumer process. For example, a print program produces characters that are consumed by the printer driver.

A producer can produce one item while the consumer is consuming another item. The Producer and Consumer must be synchronized. The consumer does not try to consume an item, the consumer must wait until an item is produced.

Unbounded-Buffer

- no practical limit on the size of the buffer.
- Producer can produce any number of items.
- Consumer may have to wait

Bounded-Buffer

- assumes that there is a fixed buffer size.

Bounded-Buffer – Shared-Memory Solution:

Shared data

```
#define BUFFER_SIZE 10
Typedef struct
{
. . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Bounded-Buffer – Producer Process:

```
item next Produced;
while (1)
{
 while (((in + 1) % BUFFER_SIZE) == out);        /* do nothing */
 buffer[in] = nextProduced;
 in = (in + 1) % BUFFER_SIZE;
}
```

Bounded-Buffer – Consumer Process:

```
item next Consumed;
while (1)
{
 while (in == out);                              /* do nothing */
 next Consumed = buffer[out];
 out = (out + 1) % BUFFER_SIZE;
}
```

(ii) Messaging Passing Method

In this method, processes communicate with each other without using any kind of shared memory. If two processes want to communicate with each other, they proceed as follows

:

- Establish a communication link (if a link already exists, no need to establish it again.)
- Start exchanging messages using basic primitives.
- The message size can be of fixed size or of variable size. If it is of fixed size, it is easy for an OS designer but complicated for a programmer and if it is of variable size then it is easy for a programmer but complicated for the OS designer.

- Cooperating process to communicate with each other via an inter process communication (IPC).
- IPC provides a Mechanism to allow processes to communicate and to synchronize their actions.
- If P and Q want to communicate, a communication link exists between them and exchange messages via send/receive. OS provides this facility.
- IPC facility provides two operations:

  Send (*message*) – message size fixed or variable.

  Receive (*message*)
- Implementation of communication link by following.
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

Methods for logical implementation of a link

i. Direct communication.
ii. Indirect communication.

Direct Communication

- Each processes must name each other explicitly:
  - Send (*P, message*) – send a message to process P.
  - Receive (*Q, message*) – receive a message from process Q.
- Links are established automatically.
- A link is associated with exactly one pair of communicating processes.
- Between each pair there exists exactly one link.
- The link may be unidirectional, but is usually bi-directional.
- This exhibits both symmetry and asymmetry in addressing

  Symmetry:

  Both the sender and the receiver processes must name the other to communicate.

  Asymmetry:

  Only sender names the recipient, the recipient is not required to name the sender. The send and receive primitives are as follows.
  - Send (P, message)– send a message to process P.
  - Receive (id, message)– receive a message from any process.

Disadvantage of direct communication

Changing a name of the process creates problems.

Indirect Communication

- The messages are sent and received from mailboxes (also referred to as ports).
- A mailbox is an object
- Process can place messages.
- Process can remove messages.
- Two processes can communicate only if they have a shared mailbox.

- Primitives are defined as:

  send (*A, message*) – send a message to mailbox A

  receive (*A, message*) – receive a message from mailbox A.
- A mailbox may be owned either by a process or by the OS.
- If the mailbox is owned by a process, then we distinguish b/w the owner (who can only receive msg through this mailbox) and the user (who can only send msg to the mailbox).
- A mailbox may be owned by the OS is independent and provide a mechanism,
  - create a mailbox
  - receive messages through mailbox
  - destroy a mail box.

Mailbox sharing problem

The processes P1, P2, and *P3* all share mailbox A. *Processes P1,* sends; *P2* and *P3* receive the message from A. Who gets a message?

Solutions

- Allow a link to be associated with at most two processes.
- Allow only one process at a time to execute a receive operation.
- Allow the system to select arbitrarily the receiver. The system may identify the receiver to the sender.

Synchronization

- Message passing may be either blocking or non-blocking.
- **Blocking** is considered **synchronous. Non-blocking** is considered **asynchronous.**
- **send** and **receive** primitives may be either blocking or non-blocking.

Blocking send

The sending process is blocked until the message is received by the receiving process or by the mailbox.

Non-blocking send

The sending process sends the message and resumes operation.

Blocking receive

The receiver blocks until a message is available.

Non-blocking receive

The receiver receives either a valid message or a null.

Buffering

- A link has some capacity that determines the number of messages that can reside in it temporarily.
- Queue of messages is attached to the link; implemented in one of three ways.

Zero capacity

- The link cannot have any messages in it.
- Sender must wait for receiver.

Bounded capacity

- finite length of *n* messages
- Sender must wait if link full.

Unbounded capacity

- infinite length
- Sender never waits.

CPU scheduling

CPU scheduling is the process of deciding which process will own the CPU to use while another process is suspended. The main function of the CPU scheduling is to ensure that whenever the CPU remains idle, the OS has at least selected one of the processes available in the ready-to-use line.

In Multiprogramming, if the long-term scheduler selects multiple I / O binding processes then most of the time, the CPU remains an idle. The function of an effective program is to improve resource utilization.

If most operating systems change their status from performance to waiting then there may always be a chance of failure in the system. So in order to minimize this excess, the OS needs to schedule tasks in order to make full use of the CPU and avoid the possibility of deadlock.

Objectives of Process Scheduling Algorithm

- Utilization of CPU at maximum level. Keep CPU as busy as possible.
- Allocation of CPU should be fair.
- Throughput should be Maximum. i.e. Number of processes that complete their execution per time unit should be maximized.
- Minimum turnaround time, i.e. time taken by a process to finish execution should be the least.
- There should be a minimum waiting time and the process should not starve in the ready queue.
- Minimum response time. It means that the time when a process produces the first response should be as less as possible.

Terminologies

- **Arrival Time:** Time at which the process arrives in the ready queue.
- **Completion Time:** Time at which process completes its execution.
- **Burst Time:** Time required by a process for CPU execution.
- **Turn Around Time:** Time Difference between completion time and arrival time.

  Turn Around Time = Completion Time – Arrival Time
- **Waiting Time(W.T):** Time Difference between turn around time and burst time.

  Waiting Time = Turn Around Time – Burst Time

THE SCHEDULING CRITERIA

CPU utilization:

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

Throughput:

The average CPU performance is the number of processes performed and completed during each unit. This is called throughput. The output may vary depending on the length or duration of the processes.

Turn round Time:

For a particular process, the important conditions are how long it takes to perform that process. The time elapsed from the time of process delivery to the time of completion is known as the conversion time. Conversion time is the amount of time spent waiting for memory access, waiting in line, using CPU, and waiting for I / O.

Waiting Time:

The Scheduling algorithm does not affect the time required to complete the process once it has started performing. It only affects the waiting time of the process i.e. the time spent in the waiting process in the ready queue.

Response Time:

In a collaborative system, turn around time is not the best option. The process may produce something early and continue to computing the new results while the previous results are released to the user. Therefore another method is the time taken in the submission of the application process until the first response is issued. This measure is called response time.

Types of CPU Scheduling Algorithms

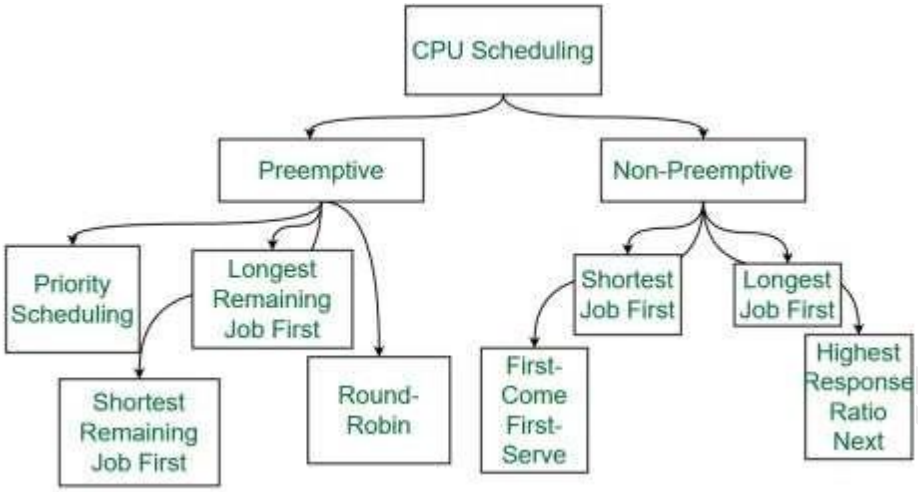
There are mainly two types of scheduling methods:

Preemptive Scheduling:

Preemptive scheduling is used when a process switches from running state to ready state or from the waiting state to the ready state.

Non-Preemptive Scheduling:

Non-Preemptive scheduling is used when a process terminates , or when a process switches from running state to waiting state.



1. First Come First Serve Scheduling:

FCFS considered to be the simplest of all operating system scheduling algorithms. First come first serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU first and is implemented by using FIFO queue.

Characteristics:

- FCFS supports non-preemptive and preemptive CPU scheduling algorithms.
- Tasks are always executed on a First-come, First-serve concept.
- FCFS is easy to implement and use.
- This algorithm is not much efficient in performance, and the wait time is quite high.

Advantages:

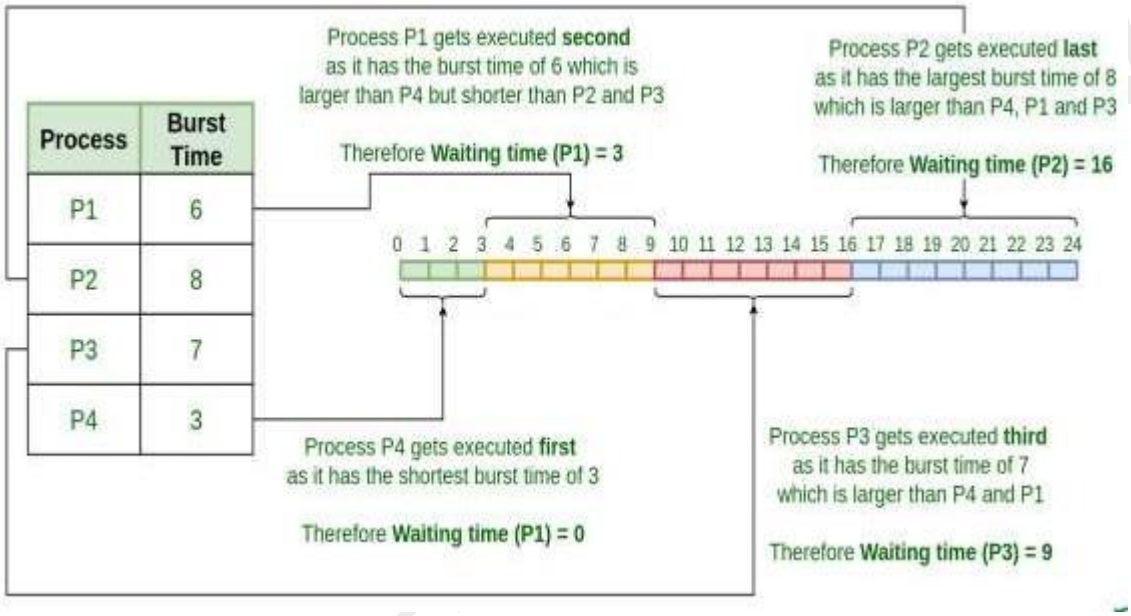
- Easy to implement
- First come, first serve method

Disadvantages:

- FCFS suffers from **Convoy effect**.
- The average waiting time is much higher than the other algorithms.
- FCFS is very simple and easy to implement and hence not much efficient.

2. Shortest Job First(SJF) Scheduling:

Shortest job first (SJF) is a scheduling process that selects the waiting process with the smallest execution time to execute next. This scheduling method may or may not be preemptive. Significantly reduces the average waiting time for other processes waiting to be executed. The full form of SJF is Shortest Job First.



Characteristics:

- Shortest Job first has the advantage of having a minimum average waiting time among all operating system scheduling algorithms.
- It is associated with each task as a unit of time to complete.
- It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of ageing.

Advantages:

- As SJF reduces the average waiting time thus, it is better than the first come first serve scheduling algorithm.
- SJF is generally used for long term scheduling

Disadvantages:

- One of the demerit SJF has is starvation.
- Many times it becomes complicated to predict the length of the upcoming CPU request

3. Longest Job First(LJF) Scheduling:

This is just opposite of shortest job first (SJF), as the name suggests this algorithm is based upon the fact that the process with the largest burst time is processed first. Longest Job First is non-preemptive in nature.

Characteristics:

- Among all the processes waiting in a waiting queue, CPU is always assigned to the process having largest burst time.
- If two processes have the same burst time then the tie is broken using FCFS i.e. the process that arrived first is processed first.
- LJF CPU Scheduling can be of both preemptive and non-preemptive types.

Advantages:

- No other task can schedule until the longest job or process executes completely.
- All the jobs or processes finish at the same time approximately.

Disadvantages:

- Generally, the LJF algorithm gives a very high average waiting time and average turn-around time for a given set of processes.
- This may lead to convoy effect.

4. Priority Scheduling:

Preemptive Priority CPU Scheduling Algorithm is a pre-emptive method of CPU scheduling algorithm that works **based on the priority** of a process. In this algorithm, the editor sets the functions to be as important, meaning that the most important process must be done first. In the case of any conflict, that is, where there are more than one processor with equal value, then the most important CPU planning algorithm works on the basis of the FCFS

Characteristics:

- Schedules tasks based on priority.
- When the higher priority work arrives while a task with less priority is executed, the higher priority work takes the place of the less priority one and
- The latter is suspended until the execution is complete.
- Lower is the number assigned, higher is the priority level of a process.

Advantages:

- The average waiting time is less than FCFS
- Less complex

Disadvantages:

- One of the most common demerits of the Preemptive priority CPU scheduling algorithm is the Starvation Problem. This is the problem in which a process has to wait for a longer amount of time to get scheduled into the CPU. This condition is called the starvation problem.

5. Round Robin Scheduling:

Round Robin is a CPU scheduling algorithm where each process is cyclically assigned a fixed time slot. It is the preemptive version of First come First Serve CPU Scheduling algorithm. Round Robin CPU Algorithm generally focuses on Time Sharing technique.

Characteristics:

- It's simple, easy to use, and starvation-free as all processes get the balanced CPU allocation.
- One of the most widely used methods in CPU scheduling as a core.
- It is considered preemptive as the processes are given to the CPU for a very limited time.

Advantages:

- Round robin seems to be fair as every process gets an equal share of CPU.
- The newly created process is added to the end of the ready queue.

6. Shortest Remaining Time First Scheduling (SRTF):

SRTF is the preemptive version of the Shortest job first which we have discussed earlier where the processor is allocated to the job closest to completion. In SRTF the process with the smallest amount of time remaining until completion is selected to execute.

Characteristics:

- SRTF algorithm makes the processing of the jobs faster than SJF algorithm, given it's overhead charges are not counted.
- The context switch is done a lot more times in SRTF than in SJF and consumes the CPU's valuable time for processing. This adds up to its processing time and diminishes its advantage of fast processing.

Advantages:

- In SRTF the short processes are handled very fast.
- The system also requires very little overhead since it only makes a decision when a process completes or a new process is added.

Disadvantages:

- Like the shortest job first, it also has the potential for process starvation.
- Long processes may be held off indefinitely if short processes are continually added.

7. Longest Remaining Time First:

The longest remaining time first is a preemptive version of the longest job first scheduling algorithm. This scheduling algorithm is used by the operating system to program incoming processes for use in a systematic way. This algorithm schedules those processes first which have the longest processing time remaining for completion.

Characteristics:

- Among all the processes waiting in a waiting queue, the CPU is always assigned to the process having the largest burst time.
- If two processes have the same burst time then the tie is broken using FCFS i.e. the process that arrived first is processed first.
- LJF CPU Scheduling can be of both preemptive and non-preemptive types.

Advantages:

- No other process can execute until the longest task executes completely.
- All the jobs or processes finish at the same time approximately.

Disadvantages:

- This algorithm gives a very high average waiting time and average turn-around time for a given set of processes.
- This may lead to a convoy effect.

8. Highest Response Ratio Next:

Highest Response Ratio Next is a non-preemptive CPU Scheduling algorithm and it is considered as one of the most optimal scheduling algorithms. The name itself states that we need to find the response ratio of all available processes and select the one with the highest Response Ratio. A process once selected will run till completion.

Characteristics:

- The **criteria** for HRRN is **Response Ratio** and the **mode** is **Non Preemptive.**
- HRRN is considered as the modification of Shortest Job First to reduce the problem of starvation.

- In comparison with SJF, during the HRRN scheduling algorithm, the CPU is allotted to the next process which has the **highest response ratio** and not to the process having less burst time.

Response Ratio = (W + S)/S

Here, **W -** Waiting time of the process

S - Burst time of the process.

Advantages:

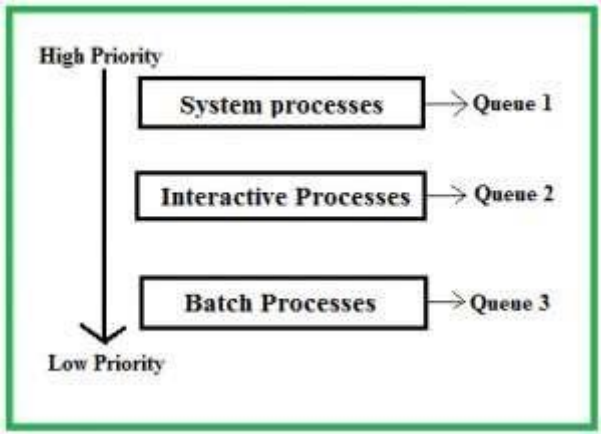
- HRRN Scheduling algorithm generally gives better performance than the shortest job first Scheduling.
- There is a reduction in waiting time for longer jobs and also it encourages shorter jobs.

Disadvantages:

- The implementation of HRRN scheduling is not possible as it is not possible to know the burst time of every job in advance.
- In this scheduling, there may occur an overload on the CPU.

9. Multiple Queue Scheduling:

Processes in the ready queue can be divided into different classes where each class has its own scheduling needs. For example, a common division is a **foreground (interactive)** process and a **background (batch)** process. These two classes have different scheduling needs. For this kind of situation **Multilevel Queue Scheduling** is used.



The description of the processes in the above diagram is as follows:

- **System Processes:** The CPU itself has its process to run, generally termed as System Process.
- **Interactive Processes:** An Interactive Process is a type of process in which there should be the same type of interaction.

- **Batch Processes:** Batch processing is generally a technique in the Operating system that collects the programs and data together in the form of a **batch** before the **processing** starts.

Advantages:

- The main merit of the multilevel queue is that it has a low scheduling overhead.

Disadvantages:

- Starvation problem
- It is inflexible in nature

10. Multilevel Feedback Queue Scheduling:

Multilevel Feedback Queue Scheduling (MLFQ) CPU Scheduling is like **Multilevel Queue Scheduling** but in this process can move between the queues. And thus, much more efficient than multilevel queue scheduling.

Characteristics:

- In a multilevel queue-scheduling algorithm, processes are permanently assigned to a queue on entry to the system, and processes are not allowed to move between queues.
- As the processes are permanently assigned to the queue, this setup has the advantage of low scheduling overhead,
- But on the other hand disadvantage of being inflexible.

Advantages:

- It is more flexible
- It allows different processes to move between different queues

Disadvantages:

- It also produces CPU overheads
- It is the most complex algorithm.

Comparison between various CPU Scheduling algorithms

Here is a brief comparison between different CPU scheduling algorithms:

Algorithm	Allocation is	Complexity	Average waiting time (AWT)	Pre emption	Star vation	Performance
FCFS	According to the arrival time of the processes, the CPU is allocated.	Simple and easy to implement	Large.	No	No	Slow
SJF	Based on the lowest CPU burst time (BT).	More complex than FCFS	Smaller than FCFS	No	Yes	Good
SRTF	Same as SJF the allocation of the CPU is based on the lowest CPU burst time (BT). But it is preemptive.	More complex than FCFS	Depending on arrival time, process size	Yes	Yes	Good
RR	According to the order of the process arrives with fixed time quantum (TQ)	The complexity depends on TQ	Large than SJF and Priority scheduling.	Yes	No	Fair
Priority Pre-emptive	According to the priority. The bigger priority task executes first	Less complex	Smaller than FCFS	Yes	Yes	Well
Priority non-preemptive	According to the priority with monitoring the new incoming higher priority jobs	Less complex than Priority preemptive	Smaller than FCFS	No	Yes	Most beneficial with batch systems

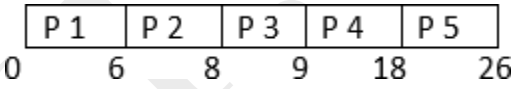
Algorithm	Allocation is	Complexity	Average waiting time (AWT)	Pre emption	Starvation	Performance
MLQ	According to the process that resides in the bigger queue priority	More complex than the priority	Smaller than FCFS	No	Yes	Good
MLFQ	According to the process of a bigger priority queue.	It is the most Complex	Smaller than all scheduling	No	No	Good

Example 1 (FCFS)

1. Process ID Process Name Burst Time (ms)

P 1	A	6
P 2	B	2
P 3	C	1
P 4	D	9
P 5	E	8

Gantt Chart



Process ID	Arrival Time (ms)	Burst Time (ms)	Completion Time (ms)	Turn Around Time (ms)	Waiting Time (ns)
P 1	0	6	6	6	0
P 2	2	2	8	8	6
P 3	3	1	9	9	8
P4	4	9	18	18	9
P 5	5	8	26	26	18

Average Turn Around Time = (6 + 8 + 9 +18 +26) / 5 = 67 / 5 = 13.4 ms
Average Waiting Time = (0 + 6 + 8 + 9 + 18) / 5 = 41 / 5 = 8.2 ms

Example 2 (FCFS)

ProcessID	Process Name	Burst Time
P 1	A	79
P 2	B	2
P 3	C	3
P 4	D	1
P 5	E	25
P 6	F	3

Process Id	Burst Time (BT)	Completion Time (CT)	Turn Around Time (TAT)	Waiting Time (WT)
P 1	79	79	79	0
P 2	2	81	81	79
P 3	3	84	84	81
P 4	1	85	85	84
P 5	25	110	110	85
P 6	3	113	113	110

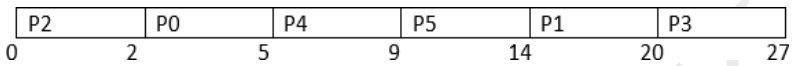
Avg Waiting Time = (0 + 79 + 81 + 84 + 85 + 110) /6 = 73.17 ms
Avg Turn Around Time = (79 + 81 + 84 + 85 + 110 +113) / 6 = 92 ms

Example 3 (SJF)

Process ID	Arrival Time	Burst Time
P0	1	3
P1	2	6
P2	1	2
P3	3	7
P4	2	4
P5	5	5

Non Pre-Emptive Shortest Job First CPU Scheduling

Gantt Chart:

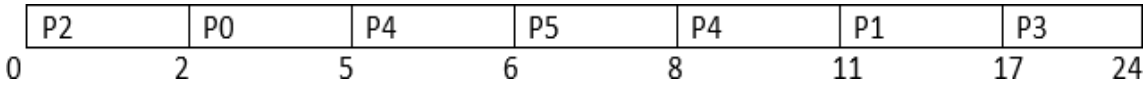


Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time TAT=CT–AT	Waiting Time WT=CT–BT
P0	1	3	5	4	1
P1	2	6	20	18	12
P2	0	2	2	2	0
P3	3	7	27	24	17
P4	2	4	9	7	4
P5	5	5	14	10	5

Average Waiting Time = (1 + 12 + 17 + 0 + 5 + 4) / 6 = 39 / 6 = 6.5 ms
Average Turn Around Time = (4 +18 + 2 +24 + 7 + 10) / 6 = 65 / 6 = 10.83 ms

Pre Emptive Shortest Job First CPU Scheduling

Gantt chart:



Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time TAT=CT-AT	Waiting Time WT=CT–BT
P0	1	3	5	4	1
P1	2	6	17	15	9
P2	0	2	2	2	0
P3	3	7	24	21	14
P4	2	4	11	9	5
P5	6	2	8	2	0

Average Turn Around Time = (4 +15 + 2 + 21 + 9 + 2) / 6 = 53 / 6 = 8.83 ms
Average Waiting Time = (1 + 9 + 0 + 14 + 5 + 0) /6 = 29 / 6 = 4.83 ms

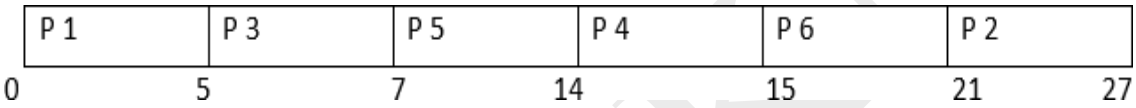
Example 4 (PRIORITY)

S. No	Process ID	Arrival Time	Burst Time	Priority
1	P 1	0	5	5
2	P 2	1	6	4
3	P 3	2	2	0
4	P 4	3	1	2
5	P 5	4	7	1
6	P 6	4	6	3

(5 has the least priority and 0 has the highest priority)

Solution:

Gantt Chart:



Process Id	Arrival Time	Burst Time	Priority	Completion Time	Turn Around Time TAT=CT-AT	Waiting Time WT=TAT-BT
P 1	0	5	5	5	5	0
P 2	1	6	4	27	26	20
P 3	2	2	0	7	5	3
P 4	3	1	2	15	12	11
P 5	4	7	1	14	10	3
P 6	4	6	3	21	17	11

Avg Waiting Time = (0 + 20 + 3 + 11 + 3 + 11) / 6 = 48 / 6 = 8 ms
Avg Turn Around Time = (5 + 26 + 5 + 11 + 10 + 17) / 6 = 74 / 6 = 12.33 ms

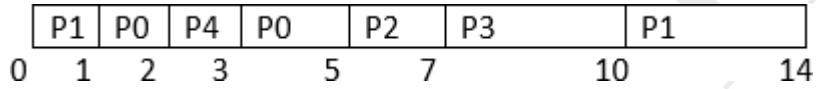
Example 5 (Round Robin)

Time Quantum = 1 ms

Process ID	Arrival Time	Burst Time
P0	1	3
P1	0	5
P2	3	2
P3	4	3
P4	2	1

Solution:

Gantt Chart:



Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time
P0	1	3	5	4	1
P1	0	5	14	14	9
P2	3	2	7	4	2
P3	4	3	10	6	3
P4	2	1	3	1	0

Avg Turn Around Time = (4+14+4+6+1) / 5 = 5.8 ms
Avg Waiting Time = (1+9+2+3+0) / 5 = 3 ms

UNIT - III

DEADLOCK

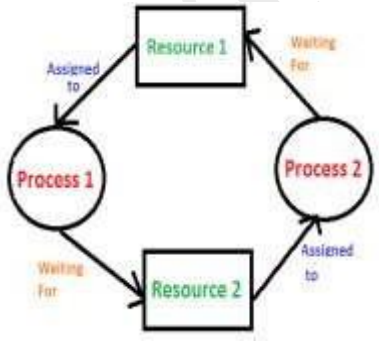
A process in operating system uses resources in the following way.

(i) Requests a resource
(ii) Use the resource
(iii) Releases the resource

*A **deadlock*** is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

Consider an example when two trains are coming toward each other on the same track and there is only one track, none of the trains can move once they are in front of each other.

A similar situation occurs in operating systems when there are two or more processes that hold some resources and wait for resources held by other(s). For example, in the below diagram, Process1 is holding Resource1 and waiting for Rsource2 which is acquired by Process2, and Process2 is waiting for Resource1.



Examples of Deadlock

1. The system has 2 tape drives. P1 and P2 each hold one tape drive and each needs another one.
2. Semaphores A and B, initialized to 1, P0, and P1 are in deadlock as follows:

   P0 executes wait(A) and preempts.
   P1 executes wait(B).
   Now P0 and P1 enter in deadlock.

P0	P1
wait(A);	wait(B)
wait(B);	wait(A)

3. Assume the space is available for allocation of 200K bytes, and the following sequence of events occurs.

P0	P1
Request 80KB;	Request 70KB;
Request 60KB;	Request 80KB;

NECESSARY CONDITIONS FOR DEADLOCK

- ***Mutual Exclusion***

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

- ***Hold and Wait***

  A process is holding at least one resource and waiting for resources.

- ***No Pre-emption***

  A resource cannot be taken from a process unless the process releases the resource.

- ***Circular Wait***

  A set of processes waiting for each other in circular form.

Resource Allocation Graph

The resource allocation graph is the pictorial representation of the state of a system. As its name suggests, the resource allocation graph is the complete information about all the processes which are holding some resources or waiting for some resources.

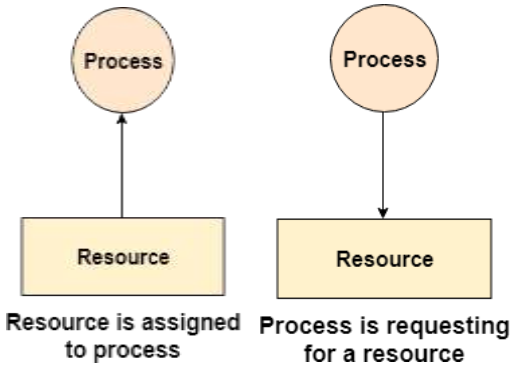
It also contains the information about all the instances of all the resources whether they are available or being used by the processes.

In Resource allocation graph, the process is represented by a Circle while the Resource is represented by a rectangle.

Vertices are mainly of two types, Resource and Process. Each of them will be represented by a different shape. Circle represents process while rectangle represents resource. A resource can have more than one instance. Each instance will be represented by a dot inside the rectangle.

Edges in RAG are also of two types, one represents **Assignment Edge** and other represents the wait of a process for a resource ie.**Request Edge**.

A resource is shown as assigned to a process if the tail of the arrow is attached to an instance to the resource and the head is attached to a process.



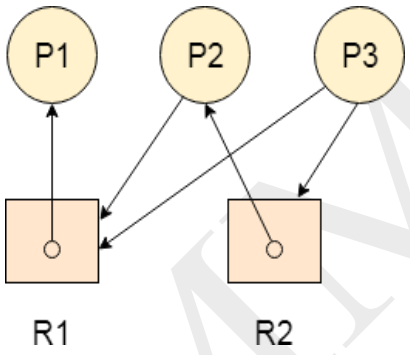
A process is shown as waiting for a resource if the tail of an arrow is attached to the process while the head is pointing towards the resource.

Example

Consider 3 processes P1, P2 and P3 and two types of resources R1 and R2. The resources are having 1 instance each.

According to the graph, R1 is being used by P1, P2 is holding R2 and waiting for R1, P3 is waiting for R1 as well as R2.

The graph is deadlock free since no cycle is being formed in the graph.



Using Resource Allocation Graph, it can be easily detected whether system is in a Deadlock state or not. The rules are

Rule-01: In a Resource Allocation Graph where all the resources are single instance,

- If a cycle is being formed, then system is in a deadlock state.
- If no cycle is being formed, then system is not in a deadlock state.

Rule-02: In a Resource Allocation Graph where all the resources are **NOT** single instance,

- If a cycle is being formed, then system may be in a deadlock state.
- **Banker's Algorithm** is applied to confirm whether system is in a deadlock state or not.
- If no cycle is being formed, then system is not in a deadlock state.
- Presence of a cycle is a necessary but not a sufficient condition for the occurrence of deadlock.

METHODS FOR HANDLING DEADLOCK

There are three ways to handle deadlock

1) Deadlock prevention or avoidance

PREVENTION

The idea is to not let the system into a deadlock state. This system will make sure that above mentioned four conditions will not arise. These techniques are very costly so we use this in cases where our priority is making a system deadlock-free.

One can zoom into each category individually, Prevention is done by negating one of the four necessary conditions for deadlock.

Eliminate mutual exclusion

It is not possible to dis-satisfy the mutual exclusion because some resources, such as the tape drive and printer, are inherently non-shareable.

Solve hold and Wait

Allocate all required resources to the process before the start of its execution, this way hold and wait condition is eliminated but it will lead to low device utilization. for example, if a process requires a printer at a later time and we have allocated a printer before the start of its execution printer will remain blocked till it has completed its execution. The process will make a new request for resources after releasing the current set of resources. This solution may lead to starvation.

Allow pre-emption

Preempt resources from the process when resources are required by other high-priority processes.

Circular wait Solution

Each resource will be assigned a numerical number. A process can request the resources to increase/decrease. order of numbering. For Example, if the P1 process is allocated R5 resources, now next time if P1 asks for R4, R3 lesser than R5 such a request will not be granted, only a request for resources more than R5 will be granted.

AVOIDANCE

Avoidance is kind of futuristic. By using the strategy of “Avoidance”, we have to make an assumption. We need to ensure that all information about resources that the process will need is known to us before the execution of the process.

Resource Allocation Graph

The resource allocation graph (RAG) is used to visualize the system's current state as a graph. The Graph includes all processes, the resources that are assigned to them, as well as the resources that each Process requests. Sometimes, if there are fewer processes, we can quickly spot a deadlock in the system by looking at the graph rather than the tables we use in Bankers's algorithm.

Banker's Algorithm

Bankers's Algorithm is a resource allocation and deadlock avoidance algorithm which test all the request made by processes for resources, it checks for the safe state, and after granting a request system remains in the safe state it allows the request, and if there is no safe state it doesn't allow the request made by the process.

In prevention and avoidance, we get the correctness of data but performance decreases.

2) Deadlock detection and recovery

If Deadlock prevention or avoidance is not applied to the software then we can handle this by deadlock detection and recovery, which consist of two phases.

In the first phase, we examine the state of the process and check whether there is a deadlock or not in the system.

If found deadlock in the first phase then we apply the algorithm for recovery of the deadlock.

3) Deadlock ignorance:

If a deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take. We use the ostrich algorithm for deadlock ignorance.

In Deadlock, ignorance performance is better than the above two methods but not the correctness of data.

SAFE STATE

A safe state can be defined as a state in which there is no deadlock. It is achievable if:

- If a process needs an unavailable resource, it may wait until the same has been released by a process to which it has already been allocated. if such a sequence does not exist, it is an unsafe state.
- All the requested resources are allocated to the process.

BANKER'S ALGORITHM

It is a banker algorithm used to **avoid deadlock** and **allocate resources** safely to each process in the computer system. The '**S-State'** examines all possible tests or activities before deciding whether the allocation should be allowed to each process. It also helps the operating system to successfully share the resources between all the processes.

The banker's algorithm is named because it checks whether a person should be sanctioned a loan amount or not to help the bank system safely simulate allocation resources.

Suppose the number of account holders in a particular bank is 'n', and the total money in a bank is 'T'. If an account holder applies for a loan; first, the bank subtracts the loan amount from full cash and then estimates the cash difference is greater than T to approve the loan amount. These steps are taken because if another person applies for a loan or withdraws some amount from the bank, it helps the bank manage and operate all things without any restriction in the functionality of the banking system.

Similarly, it works in an **operating system**. When a new process is created in a computer system, the process must provide all types of information to the operating system like upcoming processes, requests for their resources, counting them, and delays.

Based on these criteria, the operating system decides which process sequence should be executed or waited so that no deadlock occurs in a system. Therefore, it is also known as **deadlock avoidance algorithm** or **deadlock detection** in the operating system.

When working with a banker's algorithm, it requests to know about three things:

1. How much each process can request for each resource in the system. It is denoted by the [**MAX**] request.
2. How much each process is currently holding each resource in a system. It is denoted by the [**ALLOCATED**] resource.
3. It represents the number of each resource currently available in the system. It is denoted by the [**AVAILABLE**] resource.

Following are the important data structures terms applied in the banker's algorithm as follows:

Suppose n is the number of processes, and m is the number of each type of resource used in a computer system.

1. **Available**: It is an array of length 'm' that defines each type of resource available in the system. When Available[j] = K, means that 'K' instances of Resources type R[j] are available in the system.
2. **Max:** It is a [n x m] matrix that indicates each process P[i] can store the maximum number of resources R[j] (each type) in a system.
3. **Allocation:** It is a matrix of m x n orders that indicates the type of resources currently allocated to each process in the system. When Allocation [i, j] = K, it means that process P[i] is currently allocated K instances of Resources type R[j] in the system.
4. **Need:** It is an M x N matrix sequence representing the number of remaining resources for each process. When the Need[i] [j] = k, then process P[i] may require K more instances of resources type Rj to complete the assigned work.

   Need[i][j] = Max[i][j] - Allocation[i][j].
5. **Finish**: It is the vector of the order **m**. It includes a Boolean value (true/false) indicating whether the process has been allocated to the requested resources, and all resources have been released after finishing its task.

The Banker's Algorithm is the combination of the safety algorithm and the resource request algorithm to control the processes and avoid deadlock.

Safety Algorithm

It is a safety algorithm used to check whether or not a system is in a safe state or follows the safe sequence in a banker's algorithm:

Step1:

There are two vectors **Wok** and **Finish** of length m and n in a safety algorithm.

Initialize: Work = Available

Finish[i] = false; for I = 0, 1, 2, 3, 4… n - 1.

Step2:

Check the availability status for each type of resources [i], such as:

Need[i] <= Work

Finish[i] == false

If the i does not exist, go to step 4.

Step3:

Work = Work +Allocation(i) // to get new resource allocation

Finish[i] = true

Go to step2 to check the status of resource availability for the next process.

Step4:

If Finish[i] == true; it means that the system is safe for all processes.

Resource Request Algorithm

Let create a resource request array R[i] for each process P[i].

Step1:

When the number of **requested resources** of each type is less than the **Need** resources, go to step2 and if the condition fails, which means that the process P[i] exceeds its maximum claim for the resource. As the expression suggests:

If Request(i) <= Need, then go to step2, Else raise an error message.

Step2:

And when the number of requested resources of each type is less than the available resource for each process, go to step (3). As the expression suggests:

If Request(i) <= Available, then go to step3.

Else Process P[i] must wait for the resource.

Step3:

When the requested resource is allocated to the process by changing state:

Available = Available – Request

Allocation(i) = Allocation(i) + Request (i)

$Need_i = Need_i - Request_i$

When the resource allocation state is safe, its resources are allocated to the process P(i). And if the new state is unsafe, the Process P (i) has to wait for each type of Request R(i) and restore the old resource-allocation state.

Example:

Consider a system that contains five processes P1, P2, P3, P4, P5 and the three resource types A, B and C. Following are the resources types: A has 10, B has 5 and the resource type C has 7 instances.

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	3	3	2
P2	2	0	0	3	2	2			
P3	3	0	2	9	0	2			
P4	2	1	1	2	2	2			
P5	0	0	2	4	3	3			

Answer the following questions using the banker's algorithm:

1. What is the reference of the need matrix?
2. Determine if the system is safe or not.
3. What will happen if the resource request (1, 0, 2) for process P1 can the system accept this request immediately?
4. What will happen if the resource request (3, 3, 0) for process P5?
5. What will happen if the resource request (0, 2, 0) for process P1?

Ans.1:

Context of the need matrix is as Need [i] = Max [i] - Allocation [i]

Need for P1: (7, 5, 3) - (0, 1, 0) = 7, 4, 3
Need for P2: (3, 2, 2) - (2, 0, 0) = 1, 2, 2
Need for P3: (9, 0, 2) - (3, 0, 2) = 6, 0, 0
Need for P4: (2, 2, 2) - (2, 1, 1) = 0, 1, 1
Need for P5: (4, 3, 3) - (0, 0, 2) = 4, 3, 1

Process	Need		
	A	B	C
P1	7	4	3
P2	1	2	2
P3	6	0	0
P4	0	1	1
P5	4	3	1

Ans.2: Apply the Banker's Algorithm:

Available Resources of A, B and C are 3, 3, and 2.

Now we check if each type of resource request is available for each process.

Step 1:

For Process P1:

Need <= Available

7, 4, 3 <= 3, 3, 2 condition is **false**.

So, we examine another process, P2.

Step 2:

For Process P2:

Need <= Available

1, 2, 2 <= 3, 3, 2 condition **true**

New available = available + Allocation

(3, 3, 2) + (2, 0, 0) => 5, 3, 2

Similarly, we examine another process P3.

Step 3:

For Process P3:

P3 Need <= Available

6, 0, 0 < = 5, 3, 2 condition is **false**.

Similarly, we examine another process, P4.

Step 4:

For Process P4:

P4 Need <= Available

0, 1, 1 <= 5, 3, 2 condition is **true**

New Available resource = Available + Allocation

5, 3, 2 + 2, 1, 1 => 7, 4, 3

Similarly, we examine another process P5.

Step 5:

For Process P5:

P5 Need <= Available

4, 3, 1 <= 7, 4, 3 condition is **true**

New available resource = Available + Allocation

7, 4, 3 + 0, 0, 2 => 7, 4, 5

Now, we again examine each type of resource request for processes P1 and P3.

Step 6:

For Process P1:

P1 Need <= Available

7, 4, 3 <= 7, 4, 5 condition is **true**

New Available Resource = Available + Allocation

7, 4, 5 + 0, 1, 0 => 7, 5, 5

So, we examine another process P2.

Step 7:

For Process P3:

P3 Need <= Available

6, 0, 0 <= 7, 5, 5 condition is true

New Available Resource = Available + Allocation

7, 5, 5 + 3, 0, 2 => 10, 5, 7

Hence, we execute the banker's algorithm to find the safe state and the safe sequence like P2, P4, P5, P1 and P3.

Ans. 3:

For granting the Request (1, 0, 2), first we have to check that **Request <= Available**, that is (1, 0, 2) <= (3, 3, 2),

Since the condition is true, the process P2 may get the request immediately.

Allocation for P2 is (3,0,2) and new Available is (2, 3, 0)

Context of the need matrix is as follows:

Need [i] = Max [i] - Allocation [i]

Need for P1: (7, 5, 3) - (0, 1, 0) = 7, 4, 3
Need for P2: (3, 2, 2) - (3, 0, 2) = 0, 2, 0
Need for P3: (9, 0, 2) - (3, 0, 2) = 6, 0, 0
Need for P4: (2, 2, 2) - (2, 1, 1) = 0, 1, 1
Need for P5: (4, 3, 3) - (0, 0, 2) = 4, 3, 1

Process	Need		
	A	B	C
P1	7	4	3
P2	0	2	0
P3	6	0	0
P4	0	1	1
P5	4	3	1

Apply the Banker's Algorithm:

Available Resources of A, B and C are 2, 3, and 0.

Now we check if each type of resource request is available for each process.

Step 1:

For Process P1:

Need <= Available

7, 4, 3 <= 2, 3, 0 condition is **false**.

So, we examine another process, P2.

Step 2:

For Process P2:

Need <= Available

1, 2, 2 <= 2, 3, 0 condition **true**

New available = available + Allocation

(2, 3, 0) + (3, 0, 2) => 5, 3, 2

Similarly, we examine another process P3.

Step 3:

For Process P3:

P3 Need <= Available

6, 0, 0 < = 5, 3, 2 condition is **false**.

Similarly, we examine another process, P4.

Step 4:

For Process P4:

P4 Need <= Available

0, 1, 1 <= 5, 3, 2 condition is **true**

New Available resource = Available + Allocation

5, 3, 2 + 2, 1, 1 => 7, 4, 3

Similarly, we examine another process P5.

Step 5:

For Process P5:

P5 Need <= Available

4, 3, 1 <= 7, 4, 3 condition is **true**

New available resource = Available + Allocation

7, 4, 3 + 0, 0, 2 => 7, 4, 5

Now, we again examine for processes P1 and P3.

Step 6:

For Process P1:

P1 Need <= Available

7, 4, 3 <= 7, 4, 5 condition is **true**

New Available Resource = Available + Allocation

7, 4, 5 + 0, 1, 0 => 7, 5, 5

So, we examine another process P2.

Step 7:

For Process P3:

P3 Need <= Available

6, 0, 0 <= 7, 5, 5 condition is true

New Available Resource = Available + Allocation

7, 5, 5 + 3, 0, 2 => 10, 5, 7

Hence, P2 granted immediately and the safe sequence like P2, P4, P5, P1 and P3.

Ans. 4:

For granting the Request (3, 3, 0) by P5, first we have to check that **Request <= Available**, that is (3, 3, 0) <= (2, 3, 0),

Since the condition is false. So the request for (3, 3, 0) by process P5 cannot be granted.

Ans. 5:

For granting the Request (0, 2, 0) by P1, first we have to check that **Request <= Available**, that is (0, 2, 0) <= (2, 3, 0),

Since the condition is true. So the request for (0, 2, 0) by process P1 may be granted.

Allocation for P1 is (0, 3, 0)

Context of the need matrix is as follows:

Need [i] = Max [i] - Allocation [i]

Need for P1: (7, 5, 3) - (0, 3, 0) = 7, 2, 3

Process	**Need**		
	A	**B**	**C**
P1	7	2	3
P2	0	2	0
P3	6	0	0
P4	0	1	1
P5	4	3	1

Apply the Banker's Algorithm:

Available Resources of A, B and C are 2, 1, and 0.

For Process P1: 7, 2, 3<= 2, 1, 0 condition is **false**.

For Process P2: 0, 2, 0<= 2, 1, 0 condition is **false.**

For Process P3: 6, 0, 0<= 2, 1, 0 condition is **false**.

For Process P4: 0, 1, 1 <= 2, 1, 0 condition is **false.**

For Process P5: 4, 3, 1 <= 2, 1, 0 condition is **false.**

Hence, the state is unsafe, P1 cannot be granted immediately.

DEADLOCK DETECTION

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

- Apply an algorithm to examine the system's state to determine whether deadlock has occurred.
- Apply an algorithm to recover from the deadlock.

A deadlock detection algorithm is a technique used by an operating system to identify deadlocks in the system. This algorithm checks the status of processes and resources to determine whether any deadlock has occurred and takes appropriate actions to recover from the deadlock.

The algorithm employs several times varying data structures:

Available – A vector of length m indicates the number of available resources of each type.

Allocation – An n*m matrix defines the number of resources of each type currently allocated to a process. The column represents resource and rows represent a process.

Request – An n*m matrix indicates the current request of each process. If request[i][j] equals k then process P_i is requesting k more instances of resource type R_j.

The Bankers algorithm includes a **Safety Algorithm / Deadlock Detection Algorithm.** The algorithm for finding out whether a system is in a safe state can be described as follows:

Steps of Algorithm:

1. Let *Work* and *Finish* be vectors of length m and n respectively.
   Initialize *Work= Available*. For *i=0, 1, …., n-1*,
   if $Request_i$ = 0, then *Finish[i]* = true;
   otherwise, *Finish[i]*= false.
2. Find an index i such that both
   a) *Finish[i] == false*
   b) $Request_i$ <= *Work*
   If no such *i* exists go to step 4.
3. *Work= Work+* $Allocation_i$
   Finish[i]= true
   Go to Step 2.
4. If *Finish[i]== false* for some i, 0<=i<n, then the system is in a deadlocked state. Moreover, if *Finish[i]==false* the process P_i is deadlocked.

For example,

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

1. In this, Work = [0, 0, 0] &
   Finish = [false, false, false, false, false]

2. *i=0* is selected as both Finish[0] = false and [0, 0, 0]<=[0, 0, 0].
3. Work =[0, 0, 0]+[0, 1, 0] =>[0, 1, 0] &
   Finish = [true, false, false, false, false].

4. *i=2* is selected as both Finish[2] = false and [0, 0, 0]<=[0, 1, 0].
5. Work =[0, 1, 0]+[3, 0, 3] =>[3, 1, 3] &
   Finish = [true, false, true, false, false].

6. *i=1* is selected as both Finish[1] = false and [2, 0, 2]<=[3, 1, 3].
7. Work =[3, 1, 3]+[2, 0, 0] =>[5, 1, 3] &
   Finish = [true, true, true, false, false].

8. *i=3* is selected as both Finish[3] = false and [1, 0, 0]<=[5, 1, 3].
9. Work =[5, 1, 3]+[2, 1, 1] =>[7, 2, 4] &
   Finish = [true, true, true, true, false].

10. *i=4* is selected as both Finish[4] = false and [0, 0, 2]<=[7, 2, 4].
11. Work =[7, 2, 4]+[0, 0, 2] =>[7, 2, 6] &
    Finish = [true, true, true, true, true].

12. Since Finish is a vector of all true it means **there is no deadlock** in this example.

There are several algorithms for detecting deadlocks in an operating system, including:

1. Wait-For Graph:

A graphical representation of the system's processes and resources. A directed edge is created from a process to a resource if the process is waiting for that resource. A cycle in the graph indicates a deadlock.

2. Banker's Algorithm:

A resource allocation algorithm that ensures that the system is always in a safe state, where deadlocks cannot occur.

3. Resource Allocation Graph:

A graphical representation of processes and resources, where a directed edge from a process to a resource means that the process is currently holding that resource. Deadlocks can be detected by looking for cycles in the graph.

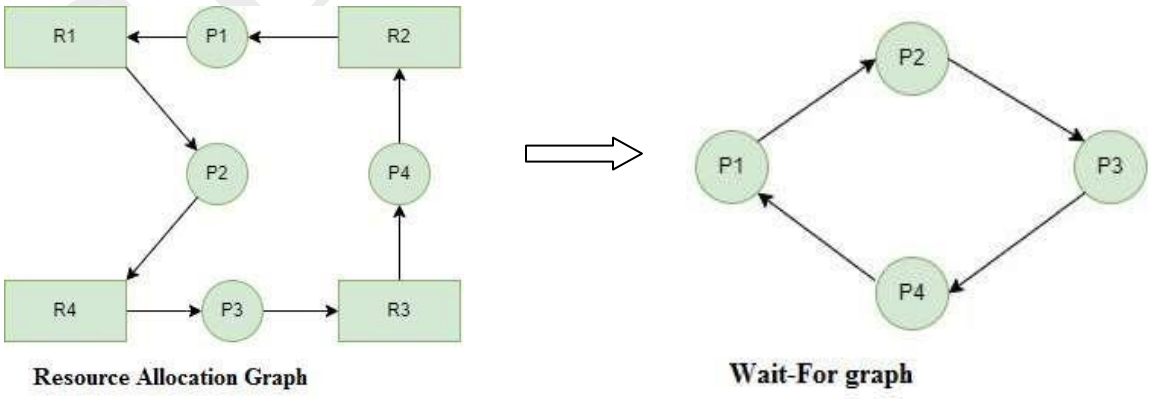
4. Detection by System Modeling:

A mathematical model of the system is created, and deadlocks can be detected by finding a state in the model where no process can continue to make progress.

5. Time stamping:

Each process is assigned a timestamp, and the system checks to see if any process is waiting for a resource that is held by a process with a lower timestamp.

These algorithms are used in different operating systems and systems with different resource allocation and synchronization requirements. The choice of algorithm depends on the specific requirements of the system and the trade-offs between performance, complexity and accuracy.



Resource Allocation Graph Wait-For graph

RECOVERY FROM DEADLOCK

The OS will use various recovery techniques to restore the system if it encounters any deadlocks. When a Deadlock Detection Algorithm determines that a deadlock has occurred in the system, the system must recover from that deadlock.

Approaches to Breaking a Deadlock

(a) Process Termination

To eliminate the deadlock, we can simply kill one or more processes. For this, we use two methods:

1. Abort all the Deadlocked Processes:

Aborting all the processes will certainly break the deadlock but at a great expense. The deadlocked processes may have been computed for a long time, and the result of those partial computations must be discarded and there is a probability of recalculating them later.

2. Abort one process at a time until the deadlock is eliminated:

Abort one deadlocked process at a time, until the deadlock cycle is eliminated from the system. Due to this method, there may be considerable overhead, because, after aborting each process, we have to run a deadlock detection algorithm to check whether any processes are still deadlocked.

(b) Resource Preemption

To eliminate deadlocks using resource preemption, we preempt some resources from processes and give those resources to other processes. This method will raise three issues –

1. Selecting a victim:

We must determine which resources and which processes are to be preempted and also in order to minimize the cost.

2. Rollback:

We must determine what should be done with the process from which resources are preempted. One simple idea is total rollback. That means aborting the process and restarting it.

3. Starvation:

In a system, it may happen that the same process is always picked as a victim. As a result, that process will never complete its designated task. This situation is called **Starvation** and must be avoided. One solution is that a process must be picked as a victim only a finite number of times.

PROCESS MANAGEMENT AND SYNCHRONIZATION

Process Synchronization is the coordination of execution of multiple processes in a multi-process system to ensure that they access shared resources in a controlled and predictable manner. It aims to resolve the problem of race conditions and other synchronization issues in a concurrent system.

The main objective of process synchronization is to ensure that multiple processes access shared resources without interfering with each other and to prevent the possibility of inconsistent data due to concurrent access. To achieve this, various synchronization techniques such as semaphores, monitors and critical sections are used.

On the basis of synchronization, processes are categorized as one of the following two types:

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

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

Race Condition

A race condition is a condition when there are many processes and every process shares the data with each other and accessing the data concurrently and the output of execution depends on a particular sequence in which they share the data and access.

(OR)

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

Several processes access and process the manipulations over the same data concurrently, then the outcome depends on the particular order in which the access takes place.

Example:

Let's say there are two processes P1 and P2 which share common variable (shared=10), both processes are present in ready – queue and waiting for its turn to be execute.

Suppose, Process P1 first come under execution, initialized as X=10 and increment it by 1 (ie. X=11), after then when CPU read line sleep(1), it switches from current process P1 to process P2 present in ready-queue. The process P1 goes in waiting state for 1 second.

Process 1	**Process 2**
int X = shared	int Y = shared
X++	Y--
sleep(1)	sleep(1)
shared = X	shared = Y

Now CPU execute the Process P2, initialized Y=10 and decrement Y by 1(ie. Y=9), after then when CPU read sleep(1), the current process P2 goes in waiting state and CPU remains idle for some time as there is no process in ready-queue.

After completion of 1 second of process P1 when it comes in ready-queue, CPU takes the process P1 under execution and execute the remaining line of code and shared=11.

After completion of 1 second of Process P2, when process P2 comes in ready-queue, CPU start executing the further remaining line of Process P2 and shared=9.

Note:

We are assuming the final value of common variable(shared) after execution of Process P1 and Process P2 is 10 (as Process P1 increment variable by 1 and Process P2 decrement variable by 1 and finally it becomes shared=10). But we are getting undesired value due to lack of proper synchronization.

Actual meaning of race-condition

- If the order of execution of process (first P1 -> then P2) then we will get the value of common variable (shared) = 9.
- If the order of execution of process (first P2 -> then P1) then we will get the final value of common variable (shared) =11.

Basically, Here the (value1 = 9) and (value2=11) are racing , If we execute these two process in our computer system then sometime we will get 9 and sometime we will get 10 as final value of common variable(shared). This phenomenon is called **Race-Condition.**

CRITICAL SECTION PROBLEM

A critical section is a code segment that can be accessed by only one process at a time. The critical section contains shared variables that need to be synchronized to maintain the consistency of data variables. So the critical section problem means designing a way for cooperative processes to access shared resources without creating data inconsistencies.

In the entry section, the process requests for entry in the **Critical Section.** Any solution to the critical section problem must satisfy three requirements:

- **Mutual Exclusion**: If a process is executing in its critical section, then no other process is allowed to execute in the critical section.
- **Progress**: If no process is executing in the critical section and other processes are waiting outside the critical section, then only those processes that are not executing in their remainder section can participate in deciding which will enter in the critical section next, and the selection can't be postponed indefinitely.
- **Bounded Waiting**: A bound must exist 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.

PETERSON'S SOLUTION

Peterson's Solution is a classical software-based solution to the critical section problem. In Peterson's solution, we have two shared variables:

- **boolean flag[i]**: Initialized to FALSE, initially no one is interested in entering the critical section
- **int turn**: The process whose turn is to enter the critical section.

```
// code for producer i
do
{
 flag[i] = true;
 turn = j;
 while (flag[j] == true && turn == j);
        critical section
 flag[i] = false;
        reminder section
 }while(TRUE);
```

```
// code for consumer j
do
{
 flag[j] = true;
 turn = i;
 while (flag[i] == true && turn == i);
        critical section
 flag[i] = false;
        reminder section
 }while(TRUE);
```

In the solution, i represents the Producer and j represents the Consumer. Initially, the flags are false. When a process wants to execute it's critical section, it sets its flag to true and turn into the index of the other process. This means that the process wants to execute but it will allow the other process to run first. The process performs busy waiting until the other process has finished it's own critical section. After this, the current process enters its critical section and adds or removes a random number from the shared buffer. After completing the critical section, it sets it's own flag to false, indicating it does not wish to execute anymore.

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.
- It is limited to 2 processes.
- Peterson's solution cannot be used in modern CPU architectures.

SEMAPHORES

Semaphore is a Hardware Solution. This Hardware solution is written or given to critical section problem. The Semaphore is just a normal integer. The Semaphore cannot be negative. The least value for a Semaphore is zero (0). The Maximum value of a Semaphore can be anything. The Semaphores usually have two operations. The two operations have the capability to decide the values of the semaphores.

The two Semaphore Operations are:

1. Wait ( )
2. Signal ( )

Wait Semaphore Operation

The Wait operation works on the basis of Semaphore or Mutex Value. If the Semaphore value is greater than zero, then the Process can enter the Critical Section Area.

If the Semaphore value is equal to zero then the Process has to wait.

If the process exits the Critical Section, then have to reduce the value of Semaphore.

Definition of wait()

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

Signal Semaphore Operation

The most important part is that this Signal Operation or V Function is executed only when the process comes out of the critical section. The value of semaphore cannot be incremented before the exit of process from the critical section.

Definition of signal()

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

There are two types of semaphores:

- **Binary Semaphores:**

  They can only be either 0 or 1. They are also known as mutex locks, as the locks can provide mutual exclusion. All the processes can share the same mutex semaphore that is initialized to 1. Then, a process has to wait until the lock becomes 0. Then, the process can make the mutex semaphore 1 and start its critical section. When it completes its critical section, it can reset the value of the mutex semaphore to 0 and some other process can enter its critical section.

- **Counting Semaphores:**

  They can have any value and are not restricted over a certain domain. They can be used to control access to a resource that has a limitation on the number of simultaneous accesses. The semaphore can be initialized to the number of instances of the resource. Whenever a process wants to use that resource, it checks if the number of remaining instances is more than zero, i.e., the process has an instance available. Then, the process can enter its critical section thereby decreasing the value of the counting semaphore by 1. After the process is over with the use of the instance of the resource, it can leave the critical section thereby adding 1 to the number of available instances of the resource.

CLASSICAL PROBLEMS OF SYNCHRONIZATION

The following problems of synchronization are considered as classical problems:

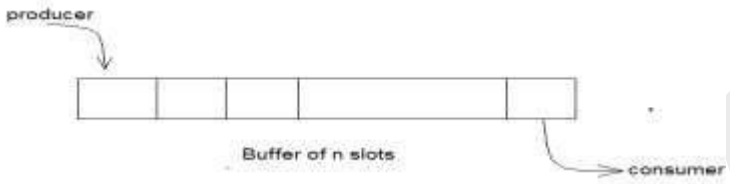
1. Bounded-buffer (or Producer-Consumer) Problem,

2. Dining-Philosophers Problem,

3. Readers and Writers Problem,

Bounded-buffer (or Producer-Consumer) Problem

Bounded Buffer problem is also called **producer consumer problem** and it is one of the classic problems of synchronization. This problem is generalized in terms of the Producer-Consumer problem. Solution to this problem is, creating two counting semaphores "full" and "empty" to keep track of the current number of full and empty buffers respectively. Producers produce a product and consumers consume the product, but both use of one of the containers each time.



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. There needs to be a way to make the producer and consumer work in an independent manner.

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

- **m**, 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.

The Producer Operation

```
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 atleast 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.

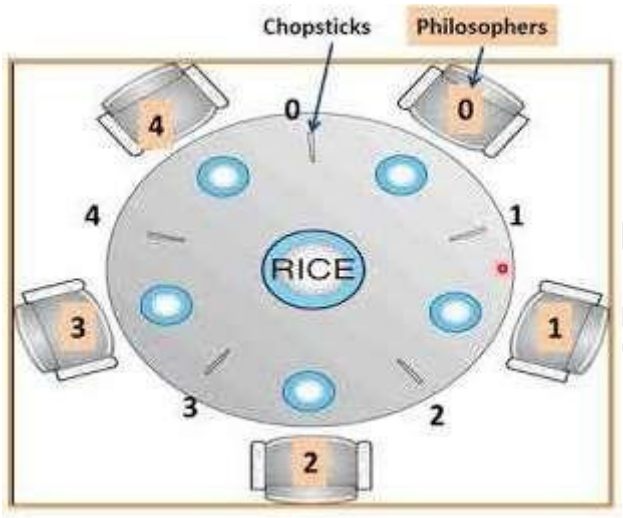
The Consumer Operation

```
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 atleast 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.

Dining-Philosophers Problem

The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pick up the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both. This problem involves the allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.



The design of the problem was to illustrate the challenges of avoiding deadlock, a deadlock state of a system is a state in which no progress of system is possible. Consider a proposal where each philosopher is instructed to behave as follows:

- The philosopher is instructed to think till the left fork is available, when it is available, hold it.
- The philosopher is instructed to think till the right fork is available, when it is available, hold it.
- The philosopher is instructed to eat when both forks are available.
- then, put the right fork down first
- then, put the left fork down next
- repeat from the beginning.

The structure of Philosopher i is as follows.

```
do
 {
  Wait( take_chopstick[i] );
  Wait( take_chopstick[(i+1) % 5] ) ;
  …
  EAT
  …
  Signal( put_chopstick[i] );
  Signal( put_chopstick[ (i+1) % 5] ) ;
  …
  THINK
 }  while(TRUE);
```

In the above code, first wait operation is performed on take_chopstick[i] and take_chopstick[(i+1) % 5]. This shows philosopher i have picked up the chopsticks from its left and right. The eating function is performed after that.

On completion of eating by philosopher i the, signal operation is performed on take_chopstick[i] and take_chopstick[(i+1) % 5]. This shows that the philosopher i have eaten and put down both the left and right chopsticks. Finally, the philosopher starts thinking again.

Let value of i = 0(initial value), Suppose Philosopher P_0 wants to eat, it will enter in Philosopher() function, and execute **Wait(take_chopstick[i]);** by doing this it holds **C0 chopstick** and reduces semaphore C0 to 0**,** after that it execute **Wait(take_chopstick[(i+1) % 5]);** by doing this it holds **C1 chopstick** (since i =0, therefore (0 + 1) % 5 = 1) and reduces semaphore C1 to 0.

Similarly, suppose now Philosopher P1 wants to eat, it will enter in Philosopher() function, and execute **Wait(take_chopstick[i]);** by doing this it will try to hold **C1 chopstick** but will not be able to do that**,** since the value of semaphore C1 has already been set to 0 by philosopher P0, therefore it will enter into an infinite loop because of which philosopher P1 will not be able to pick chopstick C1 whereas if Philosopher P2 wants to eat, it will enter in Philosopher() function, and execute **Wait(take_chopstickC[i]);** by doing this it holds **C2 chopstick** and reduces semaphore C2 to 0, after that, it executes **Wait(take_chopstickC[(i+1) % 5]);** by doing this it holds **C3 chopstick**(since i =2, therefore (2 + 1) % 5 = 3) and reduces semaphore C3 to 0.

Hence the above code is providing a solution to the dining philosopher problem, A philosopher can only eat if both immediate left and right chopsticks of the philosopher are available else philosopher needs to wait. Also at one go two independent philosophers can eat simultaneously (i.e., philosopher **P0 and P2, P1 and P3 & P2 and P4** can eat simultaneously as all are the independent processes and they are following the above constraint of dining philosopher problem)

The drawback of the above solution of the dining philosopher problem

- No two neighbouring philosophers can eat at the same point in time.
- This solution can lead to a deadlock condition. This situation happens if all the philosophers pick their left chopstick at the same time, which leads to the condition of deadlock and none of the philosophers can eat.

To avoid deadlock, some of the solutions are as follows :

- Maximum number of philosophers on the table should not be more than four, in this case, chopstick C4 will be available for philosopher P3, so P3 will start eating and after the finish of his eating procedure, he will put down his both the chopstick C3 and C4, i.e. semaphore C3 and C4 will now be incremented to 1. Now philosopher P2 which was holding chopstick C2 will also have chopstick C3 available, hence similarly, he will put down his chopstick after eating and enable other philosophers to eat.
- A philosopher at an even position should pick the right chopstick and then the left chopstick while a philosopher at an odd position should pick the left chopstick and then the right chopstick.
- Only in case if both the chopsticks (left and right) are available at the same time, only then a philosopher should be allowed to pick their chopsticks
- All the four starting philosophers (P0, P1, P2, and P3) should pick the left chopstick and then the right chopstick, whereas the last philosopher P4 should pick the right chopstick and then the left chopstick. This will force P4 to hold his right chopstick first since the right chopstick of P4 is C0, which is already held by philosopher P0 and its value is set to 0, i.e C0 is already 0, because of which P4 will get trapped into an infinite loop and chopstick C4 remains vacant. Hence philosopher P3 has both left C3 and right C4 chopstick available, therefore it will start eating and will put down its both chopsticks once finishes and let others eat which removes the problem of deadlock.

Readers and Writers Problem

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as readers and to the latter as writers. Precisely in OS we call this situation as the readers-writers problem. Problem parameters:

- One set of data is shared among a number of processes.
- Once a writer is ready, it performs its write. Only one writer may write at a time.
- If a process is writing, no other process can read it.
- If at least one reader is reading, no other process can write.
- Readers may not write and only read.

There are four types of cases that could happen here.

Case	Process 1	Process 2	Allowed/Not Allowed
Case 1	Writing	Writing	Not Allowed
Case 2	Writing	Reading	Not Allowed
Case 3	Reading	Writing	Not Allowed
Case 4	Reading	Reading	Allowed

Three variables are used: **mutex, wrt, readcnt**

1. Semaphore **mutex** is used to ensure mutual exclusion when **readcnt** is updated i.e. when any reader enters or exit from the critical section.
2. Semaphore **wrt** is used by both readers and writers.
3. **readcnt** tells the number of processes performing read in the critical section, initially 0 amd it is integer variable.

Functions for semaphore

wait() : decrements the semaphore value.

signal() : increments the semaphore value.

Reader process

- Reader requests the entry to critical section.
- If allowed:
  - it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the **wrt** semaphore to restrict the entry of writers if any reader is inside.
  - It then, signals mutex as any other reader is allowed to enter while others are already reading.
  - After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore "wrt" as now, writer can enter the critical section.
- If not allowed, it keeps on waiting.

```
do
{
  wait(mutex);        // Reader wants to enter the critical section
  readcnt++;          // The number of readers has now increased by 1

  if (readcnt==1)     // there is atleast one reader in the critical section
    wait(wrt);        //  no writer can enter if there is even one reader

  signal(mutex);      // other readers can enter where otherer is inside

   …..  perform READING

  wait(mutex);        // a reader wants to leave
  readcnt--;

if (readcnt == 0)     // no reader is left in the critical section,
    signal(wrt);      // writers can enter

  signal(mutex);      // reader leaves

} while(true);
```

Writer process

1. Writer requests the entry to critical section.
2. If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
3. It exits the critical section.

```
do
{
  wait(wrt);                    // writer requests for critical section

     …perform WRITING

  signal(wrt);                  // leaves the critical section

} while(true);
```

Thus, the semaphore „**wrt**„ is queued on both readers and writers in a manner such that preference is given to readers if writers are also there. Thus, no reader is waiting simply because a writer has requested to enter the critical section.

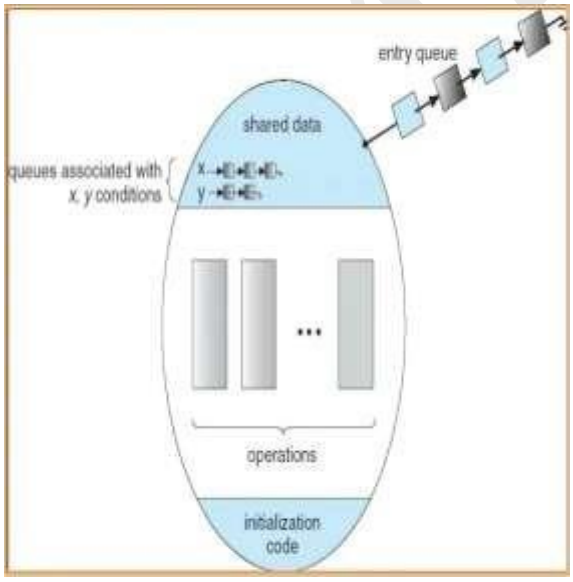
MONITOR

It is a synchronization technique that enables threads to mutual exclusion and the **wait()** for a given condition to become true. It is an abstract data type. It has a shared variable and a collection of procedures executing on the shared variable. A process may not directly access the shared data variables, and procedures are required to allow several processes to access the shared data variables simultaneously.

At any particular time, only one process may be active in a monitor. Other processes that require access to the shared variables must queue and are only granted access after the previous process releases the shared variables.

Syntax:

```
monitor
{
  //shared variable declarations
   data variables;
   Procedure P1() { ... }
   Procedure P2() { ... }
   .
   .
   .
   Procedure Pn() { ... }
   Initialization Code() { ... }
}
```



Advantages

- Mutual exclusion is automatic in monitors.
- Monitors are less difficult to implement than semaphores.
- Monitors may overcome the timing errors that occur when semaphores are used.
- Monitors are a collection of procedures and condition variables that are combined in a special type of module.

Disadvantages

- Monitors must be implemented into the programming language.
- The compiler should generate code for them.
- It gives the compiler the additional burden of knowing what operating system features is available for controlling access to crucial sections in concurrent processes.

Comparison between the Semaphore and Monitor

Features	Semaphore	Monitor
Definition	A semaphore is an integer variable that allows many processes in a parallel system to manage access to a common resource like a multitasking OS.	It is a synchronization process that enables threads to have mutual exclusion and the wait() for a given condition to become true.

Syntax	// Wait Operation
	wait(Semaphore S)
	{
	while (S<=0);
	S--;
	}
	// Signal Operation
	signal(Semaphore S)
	{
	S++;
	}
		Monitor
		{
		//shared variable declarations
		
		Procedure P1() { ... }
		Procedure P2() { ... }
		.
		.
		.
		Procedure Pn() { ... }
		Initialization Code() { ... }
		}

Basic	Integer variable	Abstract data type
Access	When a process uses shared resources, it calls the wait() method on S, and when it releases them, it uses the signal() method on S.	When a process uses shared resources in the monitor, it has to access them via procedures.
Action	The semaphore's value shows the number of shared resources available in the system.	The Monitor type includes shared variables as well as a set of procedures that operate on them.
Condition Variable	No condition variables.	It has condition variables.

UNIT – IV MEMORY MANAGEMENT and VIRTUAL MEMORY

Main Memory

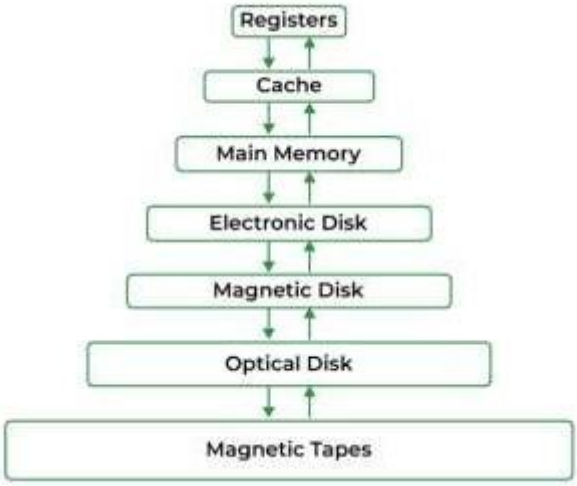
Main Memory is a large array of words or bytes, ranging in size from hundreds of thousands to billions.

Main memory is a repository of rapidly available information shared by the CPU and I/O devices.

Main memory is the place where programs and information are kept when the processor is effectively utilizing them.

Main memory is associated with the processor, so moving instructions and information into and out of the processor is extremely fast.

Main memory is also known as RAM (Random Access Memory). This memory is volatile. RAM loses its data when a power interruption occurs.



Memory Management

In a multiprogramming computer, the Operating System resides in a part of memory, and the rest is used by multiple processes.

The task of subdividing the memory among different processes is called Memory Management.

Memory management is a method in the operating system to manage operations between main memory and disk during process execution.

The main aim of memory management is to achieve efficient utilization of memory.

Requirement of Memory Management

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

Logical Address Space

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

Physical Address Space

An address seen by the memory unit (i.e the one loaded into the memory address register of the memory) is commonly known as a "Physical Address". A Physical address is also known as a Real address.

The set of all physical addresses corresponding to these logical addresses is known as Physical address space.

A physical address is computed by MMU. The run-time mapping from virtual to physical addresses is done by a hardware device Memory Management Unit (MMU). The physical address always remains constant.

Static and Dynamic Loading

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

(i) Static Loading

Static Loading is basically loading the entire program into a fixed address. It requires more memory space.

(ii) Dynamic Loading

The entire program and all data of a process must be in physical memory for the process to execute. So, the size of a process is limited to the size of physical memory. To gain proper memory utilization, dynamic loading is used. In dynamic loading, a routine is not loaded until it is called. All routines are residing on disk in a relocatable load format.

One of the advantages of dynamic loading is that the unused routine is never loaded. This loading is useful when a large amount of code is needed to handle it efficiently.

Static and Dynamic Linking

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

(i) Static Linking

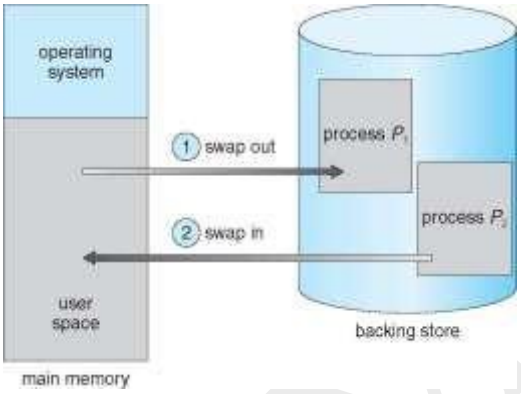
In static linking, the linker combines all necessary program modules into a single executable program. So there is no runtime dependency. Some operating systems support only static linking, in which system language libraries are treated like any other object module.

(ii) Dynamic Linking

The basic concept of dynamic linking is similar to dynamic loading. In dynamic linking, "Stub" is included for each appropriate library routine reference. A stub is a small piece of code. When the stub is executed, it checks whether the needed routine is already in memory or not. If not available then the program loads the routine into memory.

SWAPPING

When a process is executed it must have resided in memory. Swapping is a process of swapping a process temporarily into a secondary memory from the main memory, which is fast compared to secondary memory. A swapping allows more processes to be run and can be fit into memory at one time.



The main part of swapping is transferred time and the total time is directly proportional to the amount of memory swapped.

Swapping is also known as roll-out, or roll because if a higher priority process arrives and wants service, the memory manager can swap out the lower priority process and then load and execute the higher priority process. After finishing higher priority work, the lower priority process swapped back in memory and continued to the execution process.

Advantages

- If there is low main memory so some processes may has to wait for much long but by using swapping process do not have to wait long for execution on CPU.
- It utilize the main memory.
- Using only single main memory, multiple process can be run by CPU using swap partition.
- The concept of virtual memory start from here and it utilize it in better way.
- This concept can be useful in priority based scheduling to optimize the swapping process.

Disadvantages

- If there is low main memory resource and user is executing too many processes and suddenly the power of system goes off there might be a scenario where data get erase of the processes which are took parts in swapping.
- Chances of number of page faults occur
- Low processing performance

Example:

Suppose the user process's size is 2048KB and is a standard hard disk where swapping has a data transfer rate of 1Mbps. Calculate how long it will take to transfer from main memory to secondary memory.

User process size is 2048Kb

Data transfer rate is 1Mbps = 1024 kbps

Time = process size / transfer rate

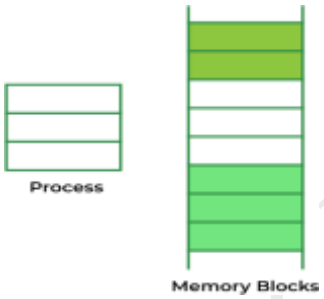
= 2048 / 1024 = 2 seconds or 2000 milliseconds

Now taking swap-in and swap-out time, the process will take 4000 ms or 4 seconds.

Contiguous Memory Allocation

The main memory should accommodate both the operating system and the different client processes. Therefore, the allocation of memory becomes an important task in the operating system. The memory is usually divided into two partitions: one for the resident operating system and one for the user processes.

We normally need several user processes to reside in memory simultaneously. Therefore, we need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory. In adjacent memory allotment, each process is contained in a single contiguous segment of memory.



Memory Allocation

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

- **Multiple partition allocation**

  A process is selected from the input queue and loaded into the free partition. When the process terminates, the partition becomes available for other processes.

- **Fixed partition allocation**

  The operating system maintains a table that indicates which parts of memory are available and which are occupied by processes.

  Initially, all memory is available for user processes and is considered one large block of available memory. This available memory is known as a "Hole".

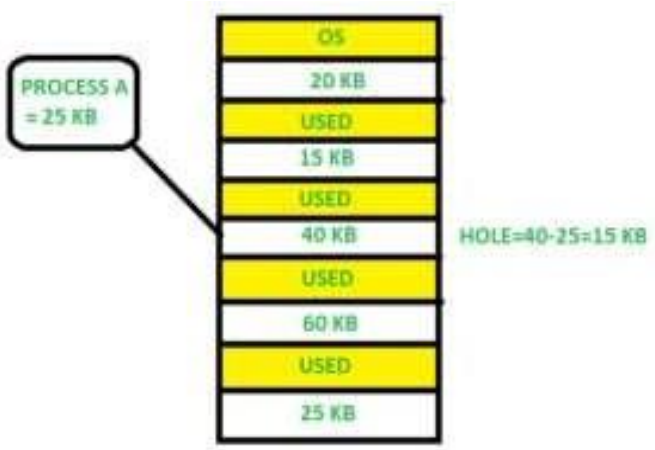
  When the process arrives and needs memory, we search for a hole that is large enough to store this process. If the requirement is fulfilled then we allocate memory to process, otherwise keeping the rest available to satisfy future requests.

While allocating a memory sometimes dynamic storage allocation problems occur, which concerns how to satisfy a request of size **n** from a list of free holes. There are some solutions to this problem:

First Fit

In the First Fit, the first available free hole fulfil the requirement of the process allocated.

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

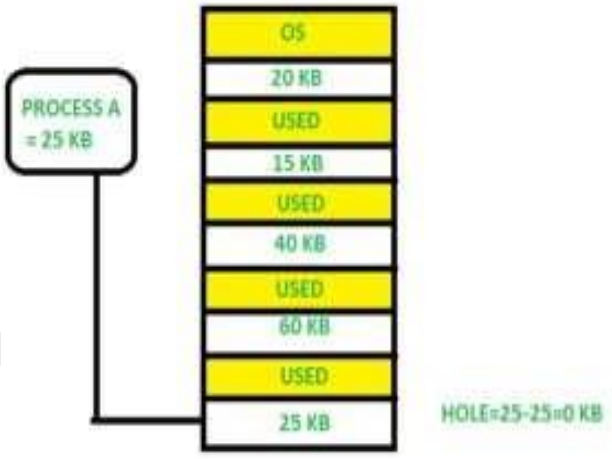


Best Fit

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

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

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

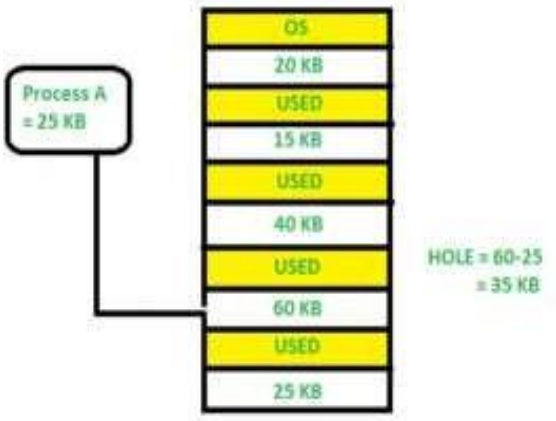


Worst Fit

In the Worst Fit, allocate the largest available hole to process. This method produces the largest leftover hole.

Here in this example, Process A (Size 25 KB) is allocated to the largest available memory block which is 60KB.

Inefficient memory utilization is a major issue in the worst fit.



FRAGMENTATION

Fragmentation is defined as when the process is loaded and removed after execution from memory, it creates a small free hole. These holes can not be assigned to new processes because holes are not combined or do not fulfill the memory requirement of the process.

To achieve a degree of multiprogramming, we must reduce the waste of memory or fragmentation problems. In the operating systems two types of fragmentation:

1. **Internal fragmentation**

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

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

2. **External fragmentation**

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

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

Both the first-fit and best-fit systems for memory allocation are affected by external fragmentation.

To overcome the external fragmentation problem **Compaction** is used. In the compaction technique, all free memory space combines and makes one large block. So, this space can be used by other processes effectively.

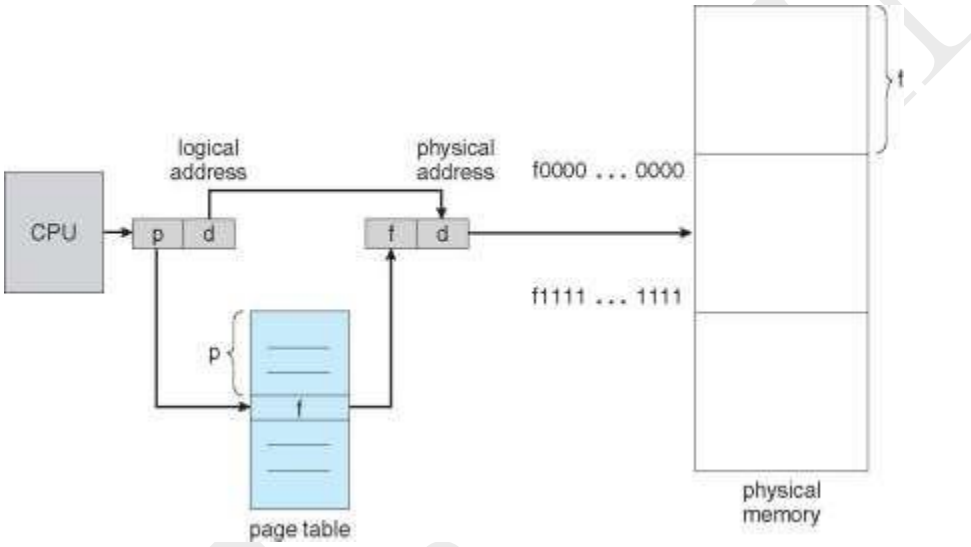
Another possible solution to the external fragmentation is to allow the logical address space of the processes to be non-contiguous, thus permitting a process to be allocated physical memory wherever the latter is available.

PAGING

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

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

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



The address generated by the CPU is divided into:

- **Page Number(p)**

  Number of bits required to represent the pages in Logical Address Space or Page number
- **Page Offset(d)**

  Number of bits required to represent a particular word in a page or page size of Logical Address Space or word number of a page or page offset.

Physical Address is divided into:

- **Frame Number(f)**

  Number of bits required to represent the frame of Physical Address Space or Frame number frame
- **Frame Offset(d)**

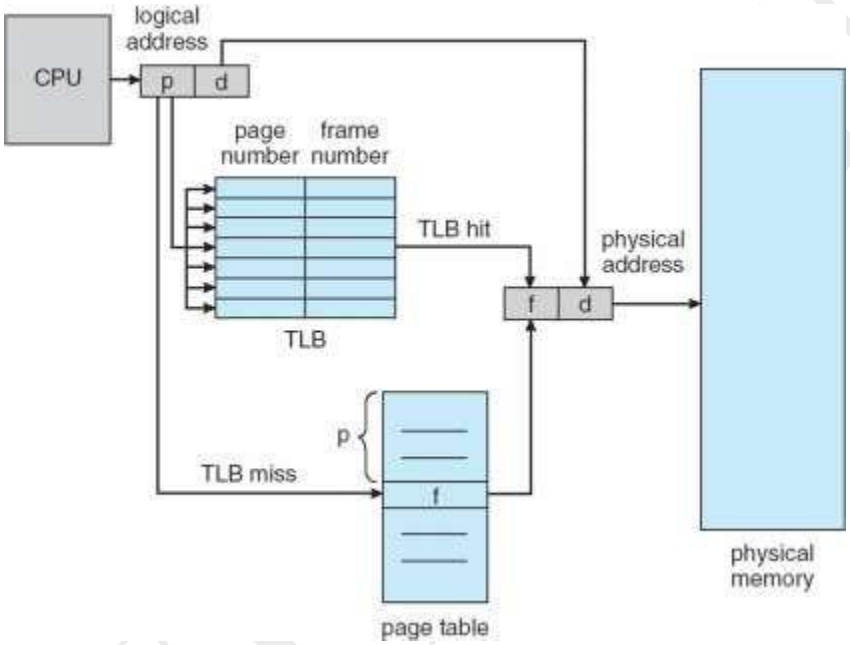
  Number of bits required to represent a particular word in a frame or frame size of Physical Address Space or word number of a frame or frame offset.

PAGING HARDWARE WITH TLB

The hardware implementation of the page table can be done by using dedicated registers. But the usage of the register for the page table is satisfactory only if the page table is small.

If the page table contains a large number of entries then we can use TLB (translation Look-aside buffer), a special, small, fast look-up hardware cache.

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



When a logical address is generated by the CPU, its page number is presented to the TLB. If the page number is found, its frame number is immediately available and is used to access memory.

If the page number is not in the TLB (known as a TLB miss), a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory. In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference. If the TLB is already full of entries, the operating system must select one for replacement. Replacement policies range from least recently used (LRU) to random. Furthermore, some TLBs allow entries to be wired down, meaning that they cannot be removed from the TLB. Typically, TLB entries for kernel code are often wired down.

The percentage of times that a particular page number is found in the TLB is called the **hit ratio**. An 80-percent hit ratio means that we find the desired page number in the TLB 80 percent of the time. If it takes 20 nanoseconds to search the TLB, and 100 nanoseconds to access memory, then a mapped memory access takes 120 nanoseconds when the page number is in the TLB. If we fail to find the page

number in the TLB (20 nanoseconds), then we must first access memory for the page table and frame number (100 nanoseconds), and then access the desired byte in memory (100 nanoseconds), for a total of 220 nanoseconds.

To find the effective memory-access time, we must weigh each case by its probability: (Where P is Hit ratio)

EAT(effective access time) = P x hit memory time + (1-P) x miss memory time.

= 0.80 x 120 + 0.20 x 220

= 140 nanoseconds.

In this example, we suffer a 40-percent slowdown in memory access time (from 100 to 140 ns).

For a 98-percent hit ratio, we have

EAT(effective access time)= P x hit memory time + (1-P) x miss memory time.

= 0.98 x 120 + 0.02 x 220

= 122 nanoseconds.

This increased hit rate produces only a 22-percent slowdown in access time.

Example:

What will be the EAT if hit ratio is 70%, time for TLB is 30ns and access to main memory is 90ns?

P = 70% = 70/100 = 0.7

Hit memory time = 30ns + 90ns = 120ns

Miss memory time = 30ns + 90ns + 90ns = 210ns

Therefore,

EAT = P x Hit + (1-P) x Miss

= 0.7 x 120 + 0.3 x 210

=840 + 63.0

=147 ns

SEGMENTATION

A process is divided into Segments. The chunks that a program is divided into which are not necessarily all of the exact sizes are called segments. Segmentation gives the user's view of the process which paging does not provide.

Here the user's view is mapped to physical memory. There is no simple relationship between logical addresses and physical addresses in segmentation.

A table stores the information about all such segments and is called **Segment Table**. It maps a two-dimensional Logical address into a one-dimensional Physical address. It's each table entry has:

Base Address:

It contains the starting physical address where the segments reside in memory.

Segment Limit:

Also known as segment offset. It specifies the length of the segment.

The address generated by the CPU is divided into:

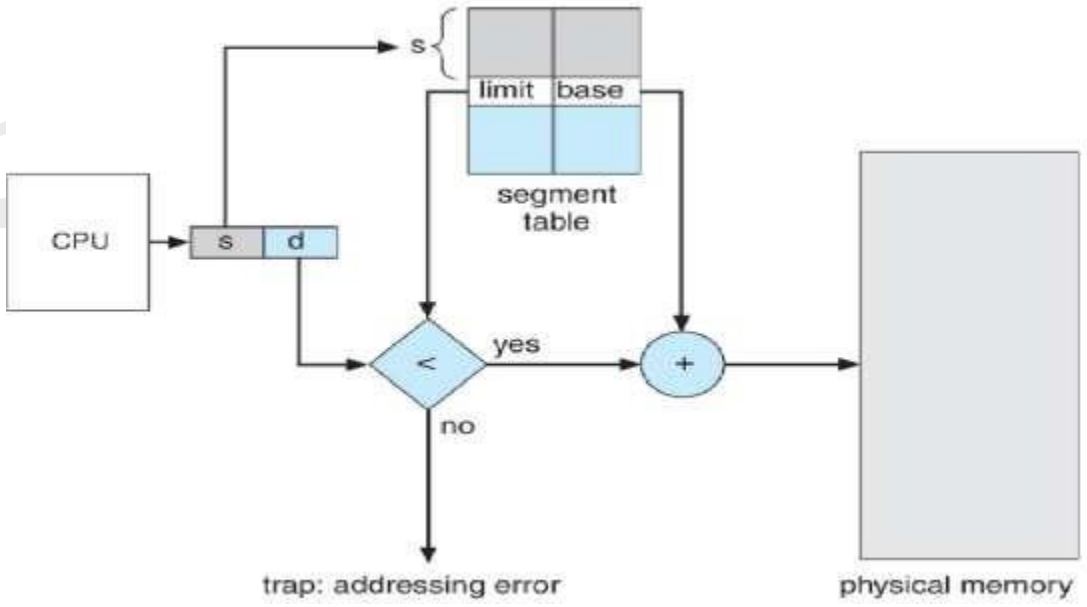
Segment number (s):

Number of bits required to represent the segment.

Segment offset (d):

Number of bits required to represent the size of the segment.

The **Segment number** is mapped to the segment table. The limit of the respective segment is compared with the offset. If the offset is less than the limit then the address is valid otherwise it throws an error as the address is invalid. In the case of valid addresses, the base address of the segment is added to the offset to get the physical address of the actual word in the main memory.

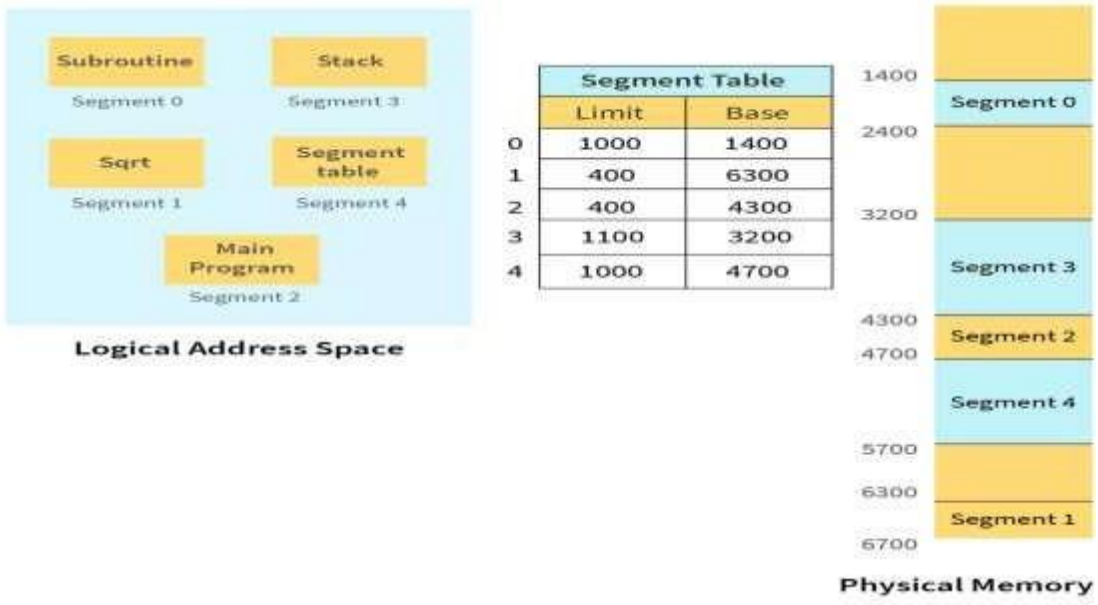


Example of Segmentation

Let us assume we have five segments namely: Segment-0, Segment-1, Segment-2, Segment-3, and Segment-4. Initially, before the execution of the process, all the segments of the process are stored in the physical memory space. We have a segment table as well. The segment table contains the beginning entry address of each segment (denoted by **base**). The segment table also contains the length of each of the segments (denoted by **limit**).

As shown in the image below, the base address of Segment-0 is 1400 and its length is 1000, the base address of Segment-1 is 6300 and its length is 400, the base address of Segment-2 is 4300 and its length is 400, and so on.

The pictorial representation of the above segmentation with its segment table is shown below.



SEGMENTATION WITH PAGING

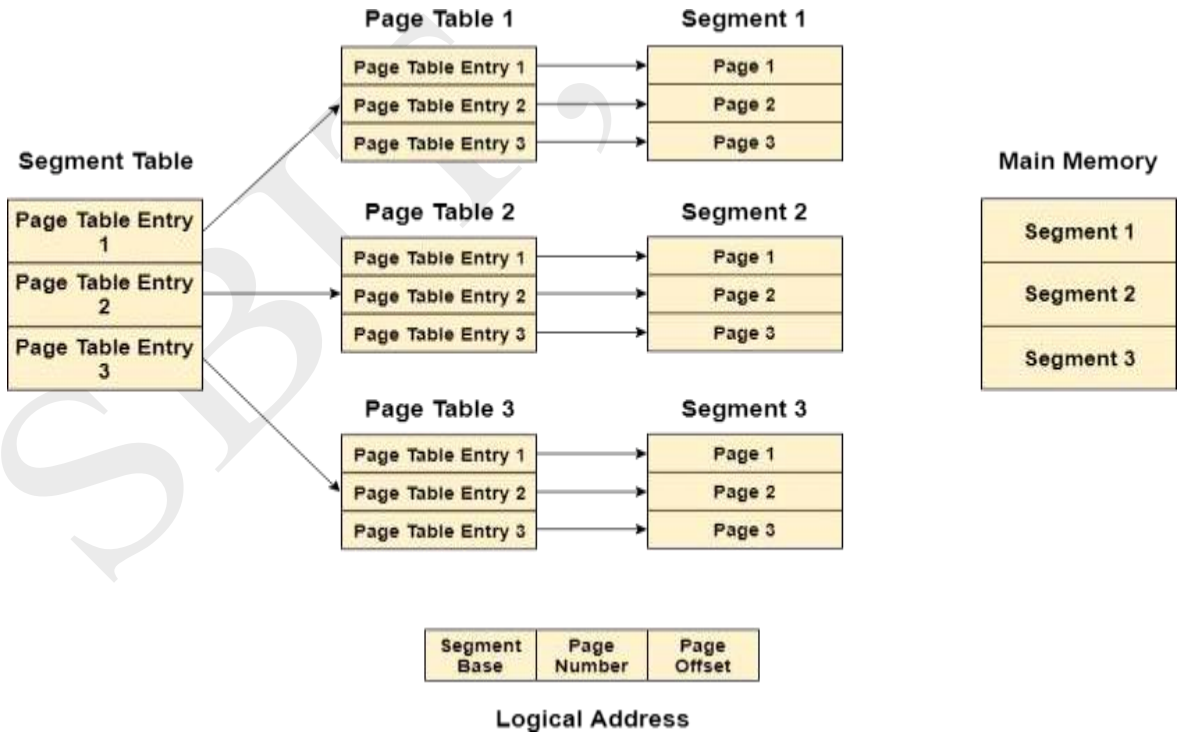
Pure segmentation is not very popular and not being used in many of the operating systems. However, Segmentation can be combined with Paging to get the best features out of both the techniques.

In Segmented Paging, the main memory is divided into variable size segments which are further divided into fixed size pages.

- Pages are smaller than segments.
- Each Segment has a page table which means every program has multiple page tables.
- The logical address is represented as Segment Number (base address), Page number and page offset.

Segment Number → It points to the appropriate Segment Number.
Page Number → It Points to the exact page within the segment
Page Offset → Used as an offset within the page frame

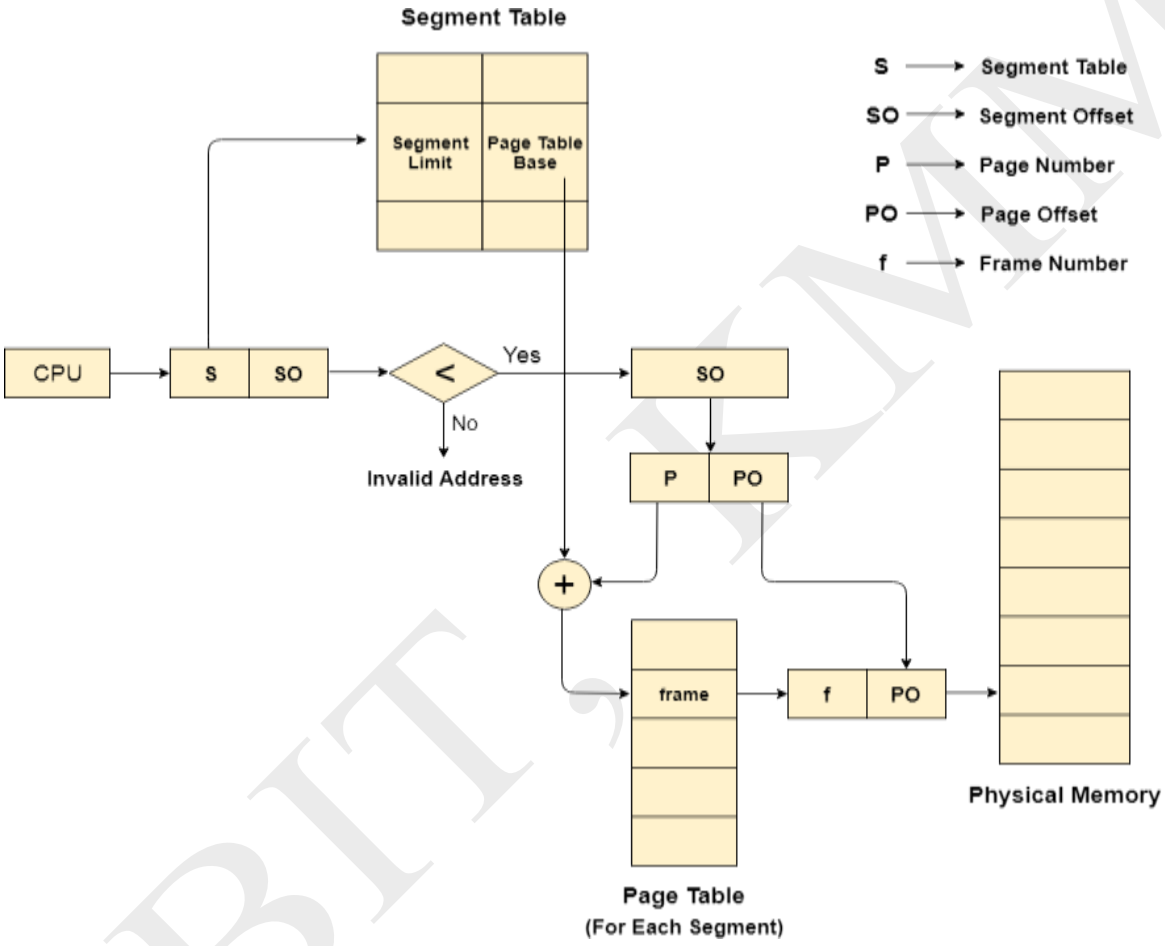
Each Page Table contains, the various information about every page of the segment. The Segment Table contains the information about every segment. Each segment table entry points to a page table entry and every page table entry is mapped to one of the page within a segment.



Translation of logical address to physical address

The CPU generates a logical address which is divided into two parts: Segment Number and Segment Offset. The Segment Offset must be less than the segment limit. Offset is further divided into Page number and Page Offset. To map the exact page number in the page table, the page number is added into the page table base.

The actual frame number with the page offset is mapped to the main memory to get the desired word in the page of the certain segment of the process.



Advantages of Segmented Paging

- It reduces memory usage.
- Page table size is limited by the segment size.
- Segment table has only one entry corresponding to one actual segment.
- External Fragmentation is not there.
- It simplifies memory allocation.

Disadvantages of Segmented Paging

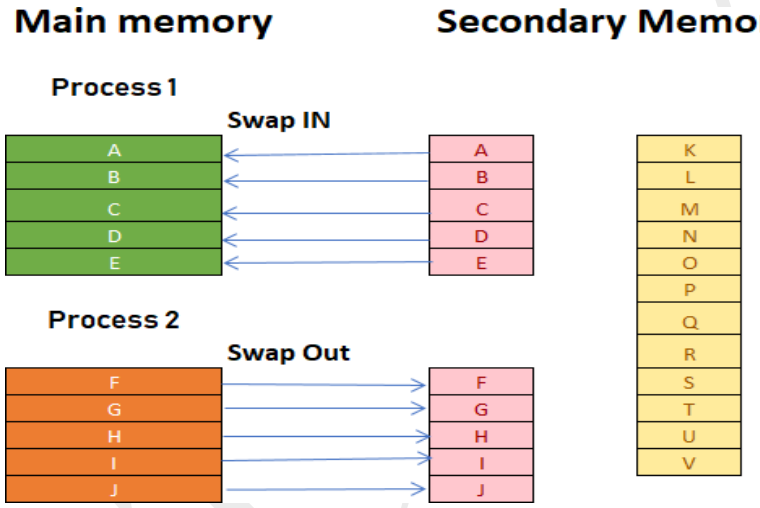
- Internal Fragmentation will be there.
- The complexity level will be much higher as compare to paging.
- Page Tables need to be contiguously stored in the memory.

DEMAND PAGING

Demand paging can be described as a memory management technique that is used in operating systems to improve memory usage and system performance. **Demand paging** is a technique used in virtual memory systems where pages enter main memory only when requested or needed by the CPU.

In demand paging, the operating system loads only the necessary pages of a program into memory at runtime, instead of loading the entire program into memory at the start.

A page fault occurred when the program needed to access a page that is not currently in memory. The operating system then loads the required pages from the disk into memory and updates the page tables accordingly. This process is transparent to the running program and it continues to run as if the page had always been in memory.



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

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

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

VIRTUAL MEMORY

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

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

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

All memory references within a process are logical addresses that are dynamically translated into physical addresses at run time. This means that a process can be swapped in and out of the main memory such that it occupies different places in the main memory at different times during the course of execution.

A process may be broken into a number of pieces and these pieces need not be continuously located in the main memory during execution. The combination of dynamic run-time address translation and the use of a page or segment table permits this.

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

PURE DEMAND PAGING

Pure demand paging is a specific implementation of demand paging. The operating system only loads pages into memory when the program needs them. In on-demand paging only, no pages are initially loaded into memory when the program starts, and all pages are initially marked as being on disk.

Benefits of the Demand Paging

So in the Demand Paging technique, there are some benefits that provide efficiency of the operating system.

- **Efficient use of physical memory**: Query paging allows for more efficient use because only the necessary pages are loaded into memory at any given time.
- **Support for larger programs:** Programs can be larger than the physical memory available on the system because only the necessary pages will be loaded into memory.

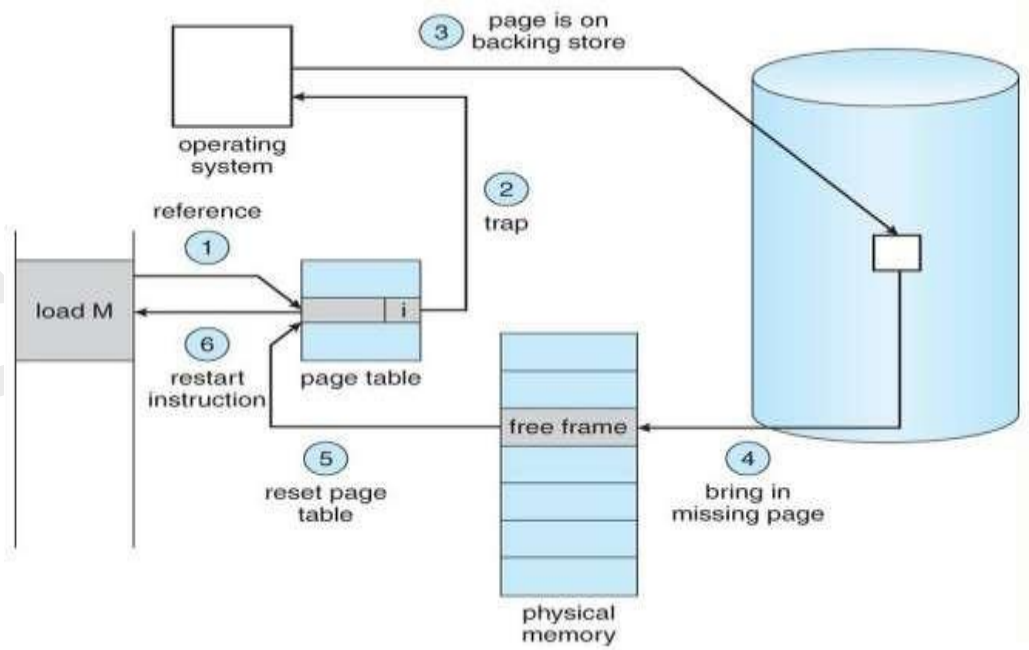
- **Faster program start:** Because only part of a program is initially loaded into memory, programs can start faster than if the entire program were loaded at once.
- **Reduce memory usage:** Query paging can help reduce the amount of memory a program needs, which can improve system performance by reducing the amount of disk I/O required.

Drawbacks of the Demand Paging

- **Page Fault Overload:** The process of swapping pages between memory and disk can cause a performance overhead, especially if the program frequently accesses pages that are not currently in memory.
- **Degraded performance:** If a program frequently accesses pages that are not currently in memory, the system spends a lot of time swapping out pages, which degrades performance.
- **Fragmentation:** Query paging can cause physical memory fragmentation, degrading system performance over time.
- **Complexity:** Implementing query paging in an operating system can be complex, requiring complex algorithms and data structures to manage page tables and swap space.

Working Process of Demand Paging

Suppose we want to run a process P which has four pages P0, P1, P2, and P3. Currently, in the page table, we have pages P1 and P3. So there are some steps that are followed in the working process of the demand paging in the operating system.



A Page Fault happens when you access a page that has been marked as invalid. The paging hardware would notice that the invalid bit is set while translating the

address across the page table, which will cause an operating system trap. The trap is caused primarily by the OS's failure to load the needed page into memory.

Procedure of page fault handling

1. Firstly, an internal table for this process to assess whether the reference was valid or invalid memory access.
2. If the reference becomes invalid, the system process would be terminated. Otherwise, the page will be paged in.
3. After that, the free-frame list finds the free frame in the system.
4. Now, the disk operation would be scheduled to get the required page from the disk.
5. When the I/O operation is completed, the process's page table will be updated with a new frame number, and the invalid bit will be changed. Now, it is a valid page reference.
6. If any page fault is found, restart these steps from starting.

Page Hit

When the CPU attempts to obtain a needed page from main memory and the page exists in main memory (RAM), it is referred to as a **"Page Hit"**.

Page Miss

If the needed page has not existed in the main memory (RAM), it is known as **"Page Miss"** or **"Page Fault".**

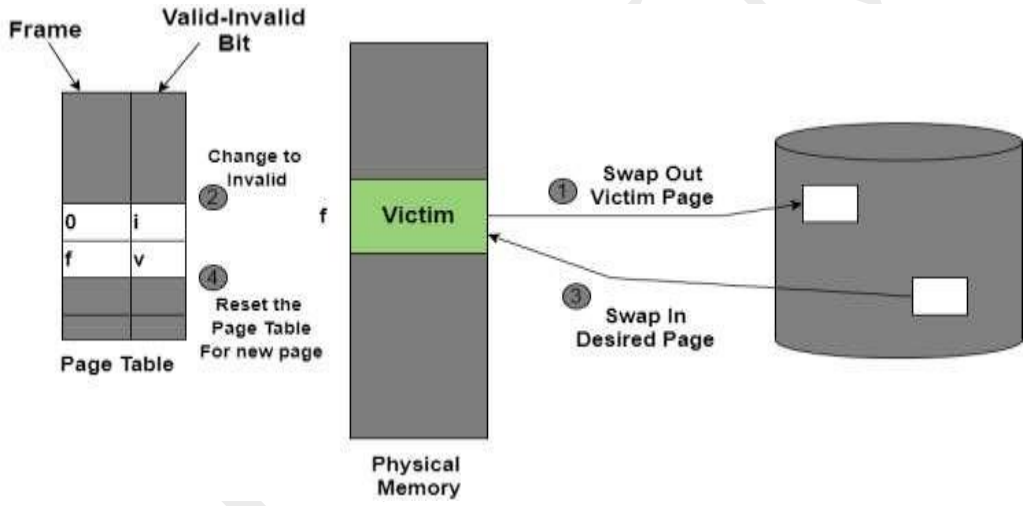
Page Fault Time

The time it takes to get a page from secondary memory and recover it from the main memory after loading the required page is known as **"Page Fault Time"**.

BASIC PAGE REPLACEMENT ALGORITHM

Page Replacement technique uses the following approach. If there is no free frame, then we will find the one that is not currently being used and then free it. A-frame can be freed by writing its content to swap space and then change the page table in order to indicate that the page is no longer in the memory.

1. First of all, find the location of the desired page on the disk.
2. Find a free Frame:
   a) If there is a free frame, then use it.
   b) If there is no free frame then make use of the page-replacement algorithm in order to select the victim frame.
   c) Then after that write the victim frame to the disk and then make the changes in the page table and frame table accordingly.
3. After that read the desired page into the newly freed frame and then change the page and frame tables.
4. Restart the process.



PAGE REPLACEMENT ALGORITHM

In an operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when a new page comes in.

A page fault happens when a running program accesses a memory page that is mapped into the virtual address space but not loaded in physical memory. Since actual physical memory is much smaller than virtual memory, page faults happen.

In case of a page fault, Operating System might have to replace one of the existing pages with the newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace.

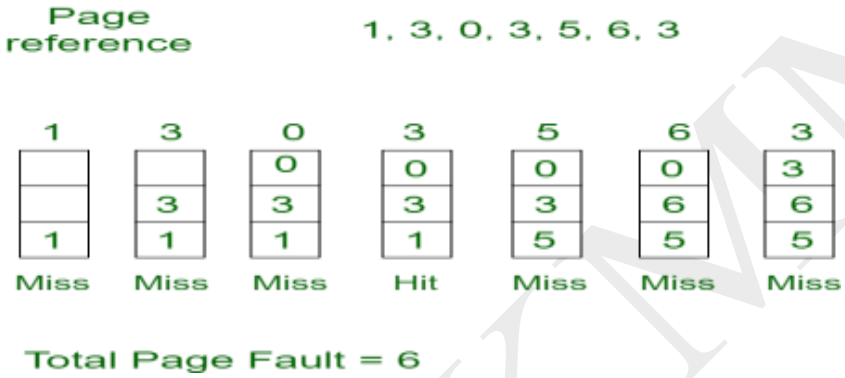
The target for all algorithms is to reduce the number of page faults.

(i) First-In-First-Out (FIFO) Page Replacement Algorithm:

This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

Example 1:

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



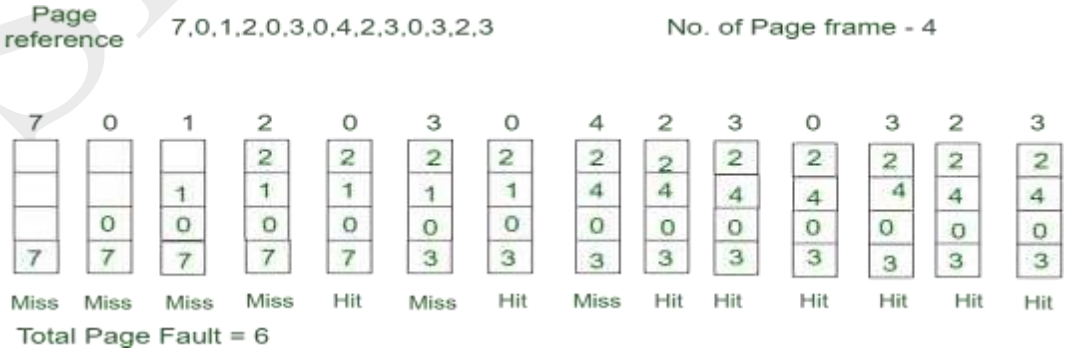
Initially, all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots —> **3 Page Faults.** When 3 comes, it is already in memory so **No Page Faults.** Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e 1. When 6 comes, it is also not available in memory so it replaces the oldest page slot i.e 3 and its cause **Page Fault.** Finally, when 3 come it is not available so it replaces 0 ie **page fault.**

(ii) Optimal Page replacement:

In this algorithm, pages are replaced which would not be used for the longest duration of time in the future.

Example:

Consider the page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 with 4 page frame.



Initially, all slots are empty, so when 7 0 1 2 are allocated to the empty slots ie **4 Page faults.** 0 is already there, so **No Page fault,** when 0 came**.** When 3 came it will take the place of 7 because it is not used for the longest duration of time in the future and its cause **Page fault.** 0 is already there, so **No Page fault.** 4 will takes place of 1 and its cause **Page Fault.** Now for the further page reference string, **No Page Fault** because they are already available in the memory.

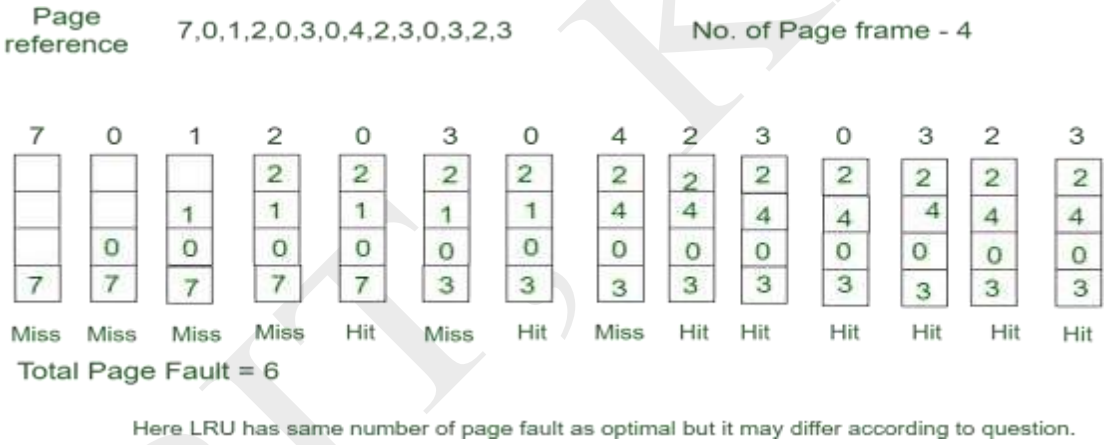
Optimal page replacement is perfect, but not possible in practice as the operating system cannot know future requests. The use of Optimal Page replacement is to set up a benchmark so that other replacement algorithms can be analyzed against it.

(iii) Least Recently Used:

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

Example:

Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 with 4 page frames.



Here LRU has same number of page fault as optimal but it may differ according to question.

Initially, all slots are empty, so when 7 0 1 2 are allocated to the empty slots, **4 Page faults.** 0 is already there, so **No Page fault.** When 3 came it will take the place of 7 because it is least recently used and its cause **Page fault**. 0 is already in memory, so **No Page fault**. 4 will takes place of 1 and its cause **Page Fault.** Now for the further page reference string **No Page fault** because they are already available in the memory.

Belady's Anomaly

Generally, on increasing the number of frames to a process virtual memory, its execution becomes faster as fewer page faults occur. Sometimes the reverse happens, i.e. more page faults occur when more frames are allocated to a process. This most unexpected result is termed **Belady's Anomaly**.

Belady's Anomaly is the name given to the phenomenon where increasing the number of page frames results in an increase in the number of page faults for a given memory access pattern.

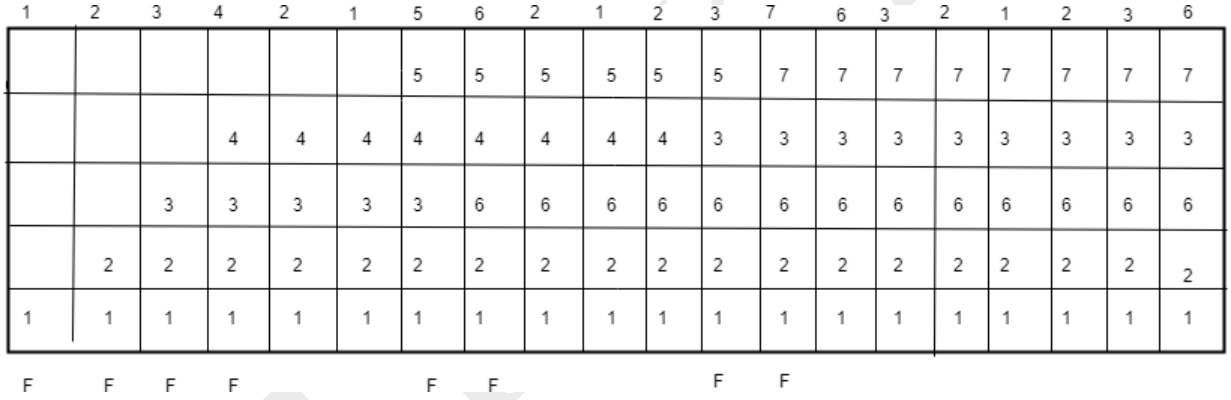
Question 1:

Consider the following page reference string:

1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6

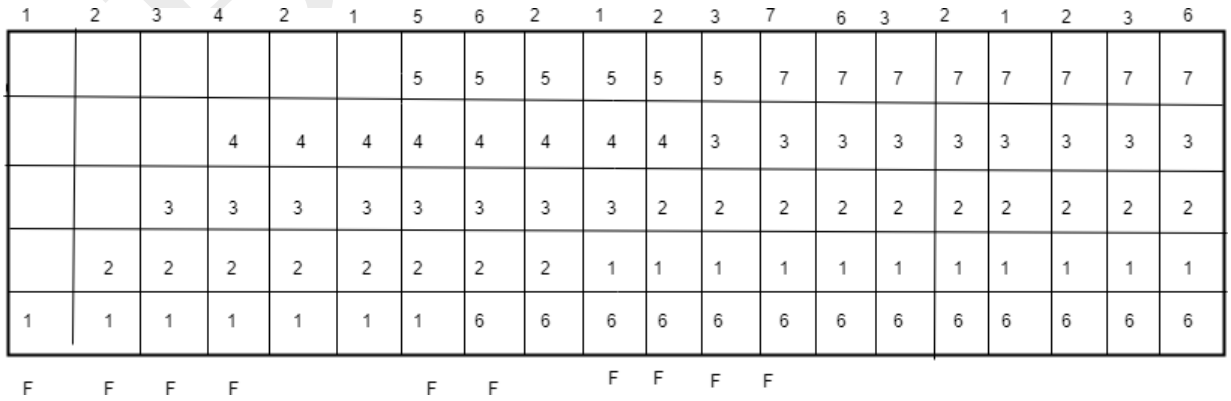
Calculate the number of page faults related to LRU, FIFO and optimal page replacement algorithms. Assume 5 page frames and all frames are initially empty.

In LRU:



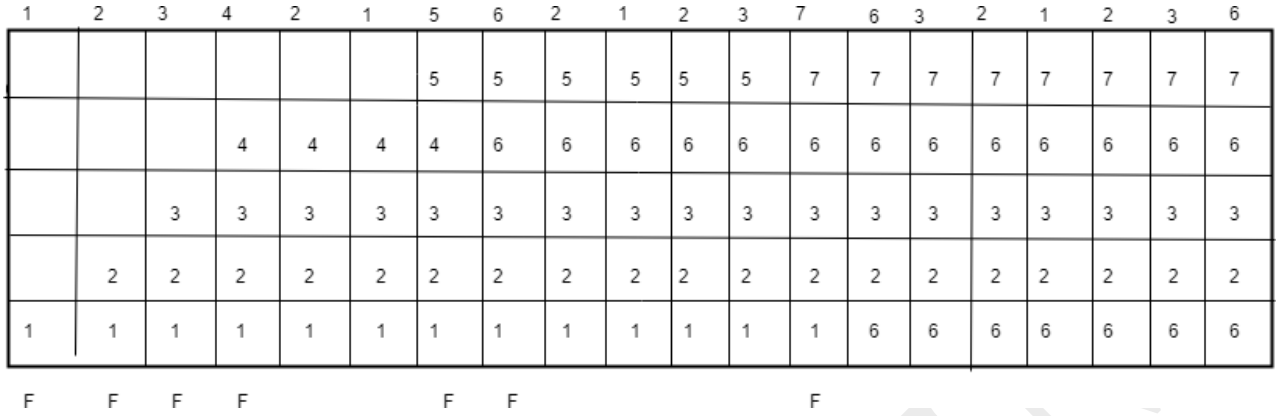
No. of Page fault = 8

In FIFO:



No. of Page fault = 10

In Optimal:



No. of Page fault = 7

Page Replacement Algorithm	No. of Page Fault
FIFO	10
LRU	8
Optimal	7

Question 2:

Consider the following page reference string:

1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.

How many page faults would occur for the following replacement algorithms, assuming one, two, three and four frames?

(i) LRU replacement

(ii) FIFO replacement

(iii) Optimal replacement

Remember that all frames are initially empty, so your first unique pages will all cost one fault each.

Solution

Frame : 1 No. of page fault is **20** for LRU, FIFO and Optimal page replacement.

Frame : 2

Ref. String	1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6	No.of Page Fault
FIFO	1	1	3	3	2	2	5	5	2	2		3	3	6	6	2	2		3	3	18
		2	2	4	4	1	1	6	6	1		1	7	7	3	3	1		1	6	
LRU	1	1	3	3	2	2	5	5	2	2		2	7	7	3	3	1		3	3	18
		2	2	4	4	1	1	6	6	1		3	3	6	6	2	2		2	6	
Optimal	1	1	3	4		1	5	6		1		3	3	3		3	1		1	6	15
		2	2	2		2	2	2		2		2	7	6		2	2		3	3	

Frame : 3

Ref. String	1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6	
FIFO	1	1	1	4		4	4	6	6	6		3	3	3		2	2		2		15
		2	2	2		1	1	1	2	2		2	7	7		7	1		1		
			3	3		3	5	5	5	1		1	1	6		6	6		6		
LRU	1	1	1	4		4	5	5	5	1		1	7	7		2	2			2	15
		2	2	2		2	2	6	6	6		3	3	3		3	3			3	
			3	3		1	1	1	2	2		2	2	6		6	1			6	
Optimal	1	1	1	1			1	1				3	3			3	3	3			11
		2	2	2			2	2				2	7			2	1	2			
			3	4			5	6				6	6			6	6	6			

Frame : 4

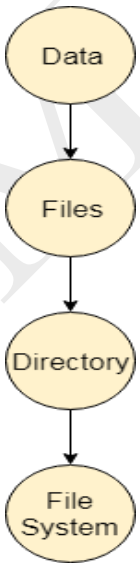
Ref. String	1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6	
FIFO	1	1	1	1			5	5	5	5		3	3	3		3	1		1		14
		2	2	2			2	6	6	6		6	7	7		7	7		3		
			3	3			3	3	2	2		2	2	6		6	6		6		
				4			4	4	4	1		1	1	1		2	2		2		
LRU	1	1	1	1			1	1				1	1	6			6				10
		2	2	2			2	2				2	2	2			2				
			3	3			5	5				3	3	3			3				
				4			4	6				6	7	7			1				
Optimal	1	1	1	1			1	1					7				1				8
		2	2	2			2	2					2				2				
			3	3			3	3					3				3				
				4			5	6					6				6				

UNIT-V FILE SYSTEM INTERFACE and OPERATIONS

System files and data are kept in the computer system's memory, and when these files are needed by an application, the operating system must have some way to read the memory and access the appropriate files.

FILE

A file can be defined as a data structure which stores the sequence of records. Files are stored in a file system, which may exist on a disk or in the main memory. Files can be simple (plain text) or complex (specially-formatted).



The collection of files is known as **Directory**. The collection of directories at the different levels, is known as **File System**.

ATTRIBUTES OF THE FILE

1. Name

Every file carries a name by which the file is recognized in the file system. One directory cannot have two files with the same name.

2. Identifier

Along with the name, Each File has its own extension which identifies the type of the file. For example, a text file has the extension **.txt,** A video file can have the extension **.mp4.**

3. Type

In a File System, the Files are classified in different types such as video files, audio files, text files, executable files, etc.

4. Location

In the File System, there are several locations on which, the files can be stored. Each file carries its location as its attribute.

5. Size

The Size of the File is one of its most important attribute. By size of the file, we mean the number of bytes acquired by the file in the memory.

6. Protection

The Admin of the computer may want the different protections for the different files. Therefore each file carries its own set of permissions to the different group of Users.

7. Time and Date

Every file carries a time stamp which contains the time and date on which the file is last modified.

OPERATIONS ON THE FILE

The various operations which can be implemented on a file such as read, write, open and close etc. are called file operations. These operations are performed by the user by using the commands provided by the operating system.

Some common operations are as follows:

1. Create

This operation is used to create a file in the file system. It is the most widely used operation performed on the file system. To create a new file of a particular type the associated application program calls the file system. This file system allocates space to the file. As the file system knows the format of directory structure, so entry of this new file is made into the appropriate directory.

2. Open

This operation is the common operation performed on the file. Once the file is created, it must be opened before performing the file processing operations. When the user wants to open a file, it provides a file name to open the particular file in the file system. It tells the operating system to invoke the open system call and passes the file name to the file system.

3. Write

This operation is used to write the information into a file. A system call write is issued that specifies the name of the file and the length of the data has to be written to the file. Whenever the file length is increased by specified value and the file pointer is repositioned after the last byte written.

4. Read

This operation reads the contents from a file. A Read pointer is maintained by the OS, pointing to the position up to which the data has been read.

5. Re-position or Seek

The seek system call re-positions the file pointers from the current position to a specific place in the file i.e. forward or backward depending upon the user's requirement. This operation is generally performed with those file management systems that support direct access files.

6. Delete

Deleting the file will not only delete all the data stored inside the file it is also used so that disk space occupied by it is freed. In order to delete the specified file the directory is searched. When the directory entry is located, all the associated file space and the directory entry is released.

7. Truncate

Truncating is simply deleting the file except deleting attributes. The file is not completely deleted although the information stored inside the file gets replaced.

8. Close

When the processing of the file is complete, it should be closed so that all the changes made permanent and all the resources occupied should be released. On closing it deallocates all the internal descriptors that were created when the file was opened.

9. Append

This operation adds data to the end of the file.

10. Rename

This operation is used to rename the existing file.

File Type

File type	Usual extension	Function
Executable	exe, com, bin	Read to run machine language program
Object	obj, o	Compiled, machine language not linked
Source Code	C, java, pas, asm, a	Source code in various languages
Batch	bat, sh	Commands to the command interpreter
Text	txt, doc	Textual data, documents
Word Processor	wp, tex, rrf, doc	Various word processor formats
Archive	arc, zip, tar	Related files grouped into one compressed file
Multimedia	mpeg, mov, rm	For containing audio/video information
Markup	xml, html, tex	It is the textual data and documents
Library	lib, a ,so, dll	It contains libraries of routines for programmers
Print or View	gif, pdf, jpg	It is a format for printing or viewing an ASCII or binary file.

FILE ACCESS METHODS

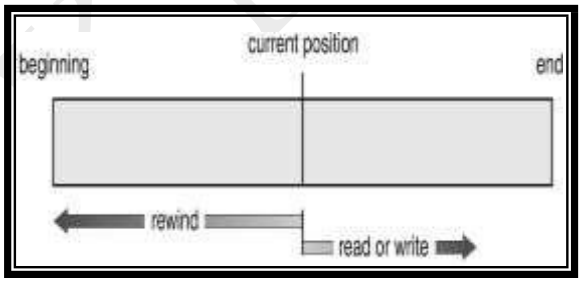
A file is a collection of bits/bytes or lines which are stored on secondary storage devices like a hard drive (magnetic disks).

File access methods in OS are nothing but techniques to read data from the system's memory. There are various ways in which we can access the files from the memory like:

- Sequential Access
- Direct/Relative Access, and
- Indexed Sequential Access.

1. Sequential Access

The operating system reads the file word by word in sequential access method of file accessing. A pointer is made, which first links to the file's base address. If the user wishes to read the first word of the file, the pointer gives it to them and raises its value to the next word. This procedure continues till the file is finished. It is the most basic way of file access.



The data in the file is evaluated in the order that it appears in the file and that is why it is easy and simple to access a file's data using sequential access mechanism. For example, editors and compilers frequently use this method to check the validity of the code.

Advantages

- The sequential access mechanism is very easy to implement.
- It uses lexicographic order to enable quick access to the next entry.

Disadvantages

- Sequential access will become slow if the next file record to be retrieved is not present next to the currently pointed record.
- Adding a new record may need relocating a significant number of records of the file.

2. Direct (or Relative) Access

A Direct/Relative file access mechanism is mostly required with the database systems. In the majority of the circumstances, we require filtered/specific data from the database, and in such circumstances, sequential access might be highly inefficient.

sequential access	implementation for direct access
reset	*cp = 0;*
read next	*read cp;* *cp = cp+1;*
write next	*write cp;* *cp = cp+1;*

Assume that each block of storage holds four records and that the record we want to access is stored in the tenth block. In such a situation, sequential access will not be used since it will have to traverse all of the blocks to get to the required record, while direct access will allow us to access the required record instantly.

The direct access mechanism requires the OS to perform some additional tasks but eventually leads to much faster retrieval of records as compared to the sequential access.

Advantages

- The files can be retrieved right away with direct access mechanism, reducing the average access time of a file.
- There is no need to traverse all of the blocks that come before the required block to access the record.

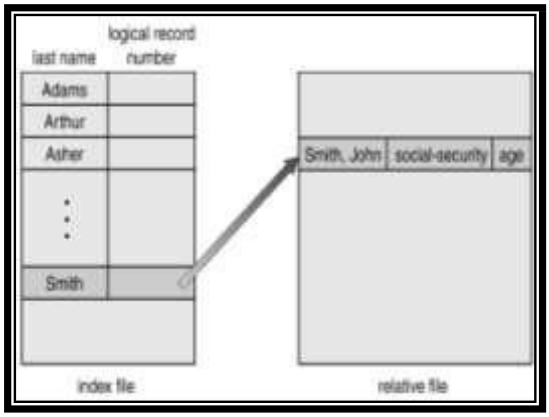
Disadvantages

- The direct access mechanism is typically difficult to implement due to its complexity.
- Organizations can face security issues as a result of direct access as the users may access/modify the sensitive information. As a result, additional security processes must be put in place.

3. Indexed Sequential Access

This method is practically similar to the pointer to pointer concept in which we store an address of a pointer variable containing address of some other variable/record in another pointer variable.

The indexes, similar to a book's index (pointers), contain a link to various blocks present in the memory. To locate a record in the file, we first search the indexes and then use the pointer to pointer concept to navigate to the required file.



Primary index blocks contain the links of the secondary inner blocks which contains links to the data in the memory.

Advantages

- If the index table is appropriately arranged, it accesses the records very quickly.
- Records can be added at any position in the file quickly.

Disadvantages of Indexed Sequential Access

- When compared to other file access methods, it is costly and less efficient.
- It needs additional storage space.

DIRECTORY STRUCTURE

Directory can be defined as the listing of the related files on the disk. The directory may store some or the entire file attributes.

To get the benefit of different file systems on the different operating systems, A hard disk can be divided into the number of partitions of different sizes. The partitions are also called volumes or mini disks.

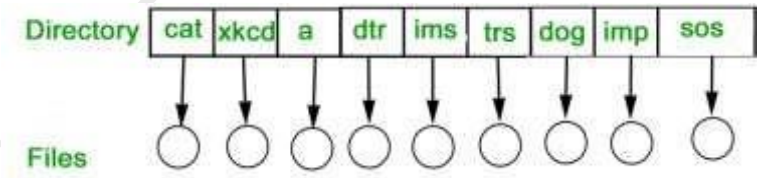
Each partition must have at least one directory in which, all the files of the partition can be listed. A directory entry is maintained for each file in the directory which stores all the information related to that file.

A directory can be viewed as a file which contains the Meta data of the bunch of files.

Every Directory supports a number of common operations on the file:

- File Creation
- Search for the file
- File deletion
- Renaming the file
- Traversing Files
- Listing of files

SINGLE LEVEL DIRECTORY



The simplest method is to have one big list of all the files on the disk. The entire system will contain only one directory which is supposed to mention all the files present in the file system. The directory contains one entry per each file present on the file system.

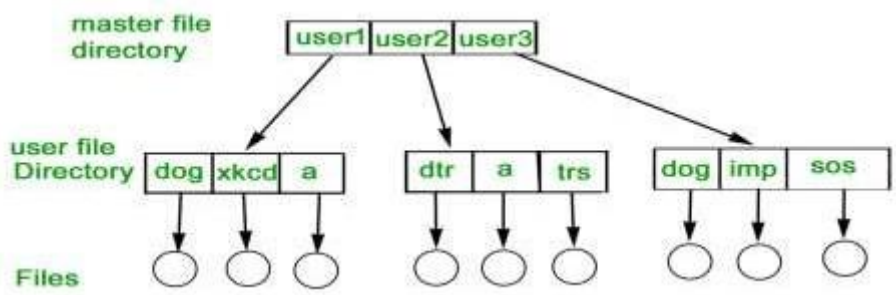
Advantages

1. Implementation is very simple.
2. If the sizes of the files are very small then the searching becomes faster.
3. File creation, searching, deletion is very simple since we have only one directory.

Disadvantages

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

TWO-LEVEL DIRECTORY



In two level directory systems, we can create a separate directory for each user. There is one master directory which contains separate directories dedicated to each user. For each user, there is a different directory present at the second level, containing group of user's file. The system doesn't let a user to enter in the other user's directory without permission.

Path name: Due to two levels there is a path name for every file to locate that file.

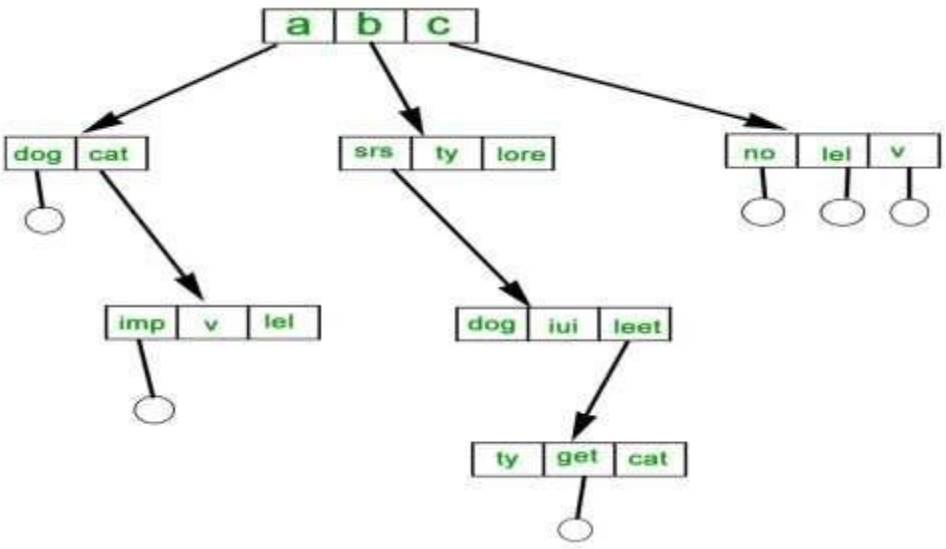
Advantage

- we can have the same file name for different users.
- Searching is efficient in this method.

TREE- STRUCTURED DIRECTORY

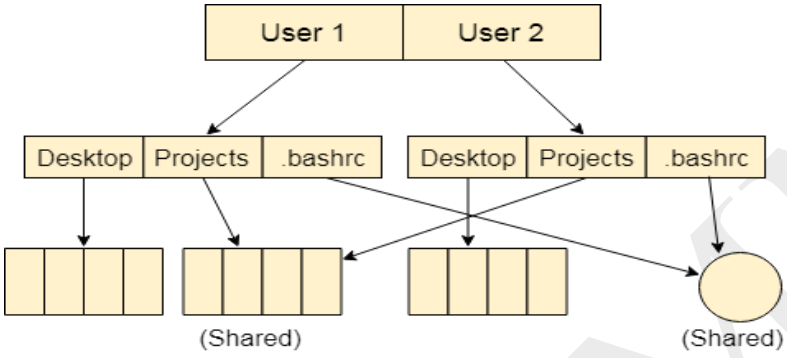
In Tree structured directory system, any directory entry can either be a file or sub directory. Tree structured directory system overcomes the drawbacks of two level directory system. The similar kind of files can now be grouped in one directory.

The directory is maintained in the form of a tree. Searching is efficient and also there is grouping capability. We have absolute or relative path name for a file.



ACYCLIC-GRAPH STRUCTURED DIRECTORY

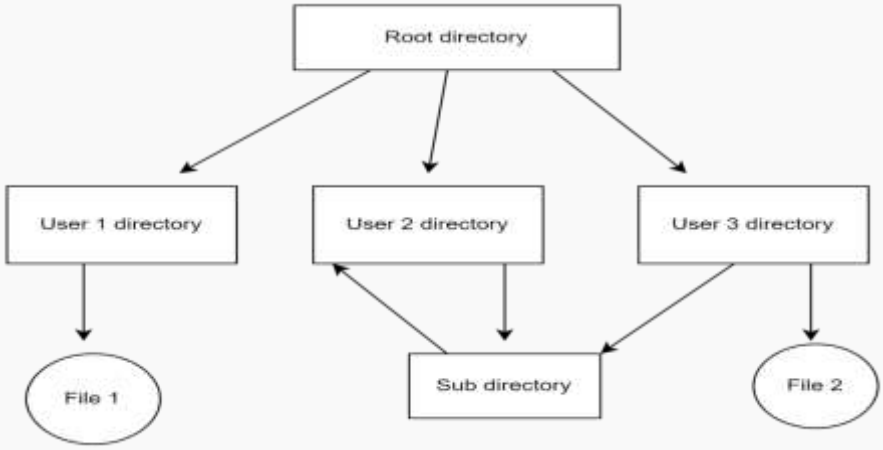
The tree structured directory system doesn't allow the same file to exist in multiple directories therefore sharing is major concern in tree structured directory system. We can provide sharing by making the directory an acyclic graph. In this system, two or more directory entry can point to the same file or sub directory. That file or sub directory is shared between the two directory entries.



Acyclic-Graph Structured Directory System

GENERAL-GRAPH DIRECTORY

This is an extension to the acyclic-graph directory. In the general-graph directory, there can be a cycle inside a directory.



In the above image, we can see that a cycle is formed in the user 2 directory. Although it provides greater flexibility, it is complex to implement this structure.

Advantages

- Compared to the others, the General-Graph directory structure is more flexible.
- Cycles are allowed in the directory for general-graphs.

Disadvantages

- It costs more than alternative solutions.
- Garbage collection is an essential step here.

PROTECTION IN FILE SYSTEM

In computer systems, a lot of user's information is stored, the objective of the operating system is to keep safe the data of the user from the improper access to the system.

Protection can be provided in number of ways. For a single laptop system, we might provide protection by locking the computer in a desk drawer or file cabinet. For multi-user systems, different mechanisms are used for the protection.

Types of Access

The files which have direct access of the any user have the need of protection. The files which are not accessible to other users doesn't require any kind of protection.

The mechanism of the protection provide the facility of the controlled access by just limiting the types of access to the file. Access can be given or not given to any user depends on several factors, one of which is the type of access required.

Several different types of operations can be controlled:

- **Read –** Reading from a file.
- **Write –** Writing or rewriting the file.
- **Execute –** Loading the file and after loading the execution process starts.
- **Append –** Writing the new information to the already existing file, editing must be end at the end of the existing file.
- **Delete –** Deleting the file which is of no use and using its space for the another data.
- **List –** List the name and attributes of the file.

Operations like renaming, editing the existing file, copying; these can also be controlled. There are many protection mechanism. each of them mechanism have different advantages and disadvantages and must be appropriate for the intended application.

Access Control

There are different methods used by different users to access any file. The general way of protection is to associate *identity-dependent access* with all the files and directories a list called access-control list (ACL) which specify the names of the users and the types of access associate with each of the user.

The main problem with the access list is their length. If we want to allow everyone to read a file, we must list all the users with the read access. This technique has two undesirable consequences:

Constructing such a list may be tedious and unrewarding task, especially if we do not know in advance the list of the users in the system.

Previously, the entry of the any directory is of the fixed size but now it changes to the variable size which results in the complicates space management. These

problems can be resolved by use of a condensed version of the access list. To condense the length of the access-control list, many systems recognize three classification of users in connection with each file:

- **Owner –** Owner is the user who has created the file.
- **Group –** A group is a set of members who has similar needs and they are sharing the same file.
- **Universe –** In the system, all other users are under the category called universe. The most common recent approach is to combine access-control lists with the normal general owner, group, and universe access control scheme. For example: Solaris uses the three categories of access by default but allows access-control lists to be added to specific files and directories when more fine-grained access control is desired.

Other Protection Approaches

The access to any system is also controlled by the password. If the use of password is random and it is changed often, this may be result in limit the effective access to a file.

The use of passwords has a few disadvantages:

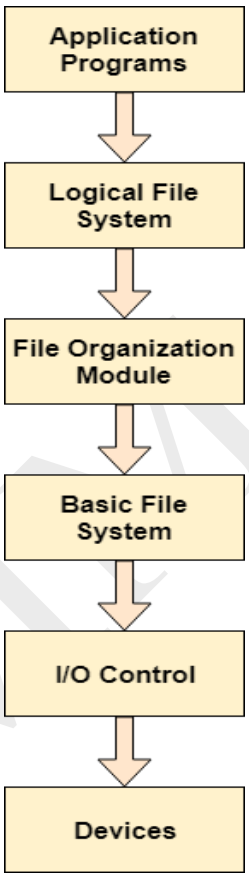
- The number of passwords are very large so it is difficult to remember the large passwords.
- If one password is used for all the files, then once it is discovered, all files are accessible; protection is on all-or-none basis.

FILE SYSTEM STRUCTURE

File System provide efficient access to the disk by allowing data to be stored, located and retrieved in a convenient way. A file System must be able to store the file, locate the file and retrieve the file.

Most of the Operating Systems use layering approach for every task including file systems. Every layer of the file system is responsible for some activities.

The image shown, elaborates how the file system is divided in different layers, and also the functionality of each layer.



- When an application program asks for a file, the first request is directed to the logical file system. The logical file system contains the Meta data of the file and directory structure. If the application program doesn't have the required permissions of the file then this layer will throw an error. Logical file systems also verify the path to the file.
- Generally, files are divided into various logical blocks. Files are to be stored in the hard disk and to be retrieved from the hard disk. Hard disk is divided into various tracks and sectors. Therefore, in order to store and retrieve the files, the logical blocks need to be mapped to physical blocks. This mapping is done by File organization module. It is also responsible for free space management.
- Once File organization module decided which physical block the application program needs, it passes this information to basic file system. The basic file system is responsible for issuing the commands to I/O control in order to fetch those blocks.
- I/O controls contain the codes by using which it can access hard disk. These codes are known as device drivers. I/O controls are also responsible for handling interrupts.

ALLOCATION METHODS

There are various methods which can be used to allocate disk space to the files. Selection of an appropriate allocation method will significantly affect the performance and efficiency of the system. Allocation method provides a way in which the disk will be utilized and the files will be accessed.

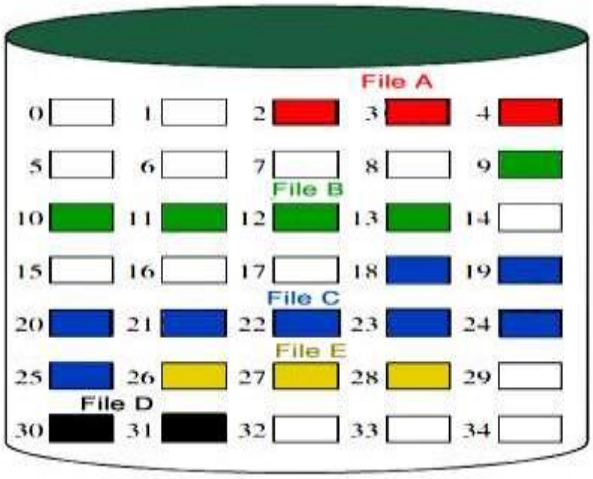
There are following methods which can be used for allocation.

1. Contiguous Allocation.
2. Linked Allocation
3. Indexed Allocation
4. Linked Indexed Allocation
5. Multilevel Indexed Allocation

Contiguous Allocation

A single continuous set of blocks is allocated to a file at the time of file creation. Thus, this is a pre-allocation strategy, using variable size portions. The file allocation table needs just a single entry for each file, showing the starting block and the length of the file. This method is best from the point of view of the individual sequential file.

Multiple blocks can be read in at a time to improve I/O performance for sequential processing. It is also easy to retrieve a single block. For example, if a file starts at block b, and the ith block of the file is wanted, its location on secondary storage is simply b+i-1.



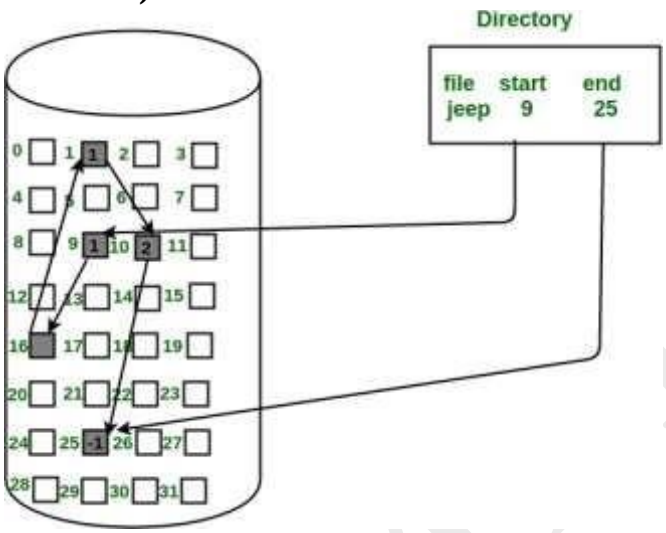
File allocation table

File name	Start block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

Disadvantage

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

Linked Allocation(Non-contiguous allocation)



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

Disadvantage

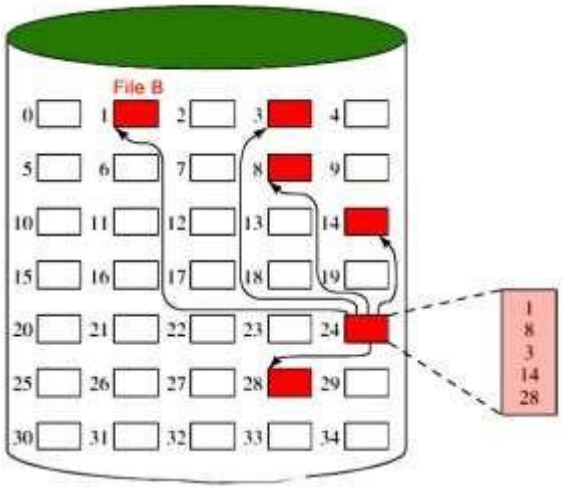
- Internal fragmentation exists in the last disk block of the file.
- There is an overhead of maintaining the pointer in every disk block.
- If the pointer of any disk block is lost, the file will be truncated.
- It supports only the sequential access of files.

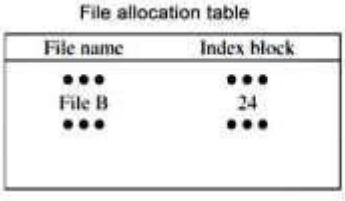
Indexed Allocation

It addresses many of the problems of contiguous and chained allocation. In this case, the file allocation table contains a separate one-level index for each file: The index has one entry for each block allocated to the file.

The allocation may be on the basis of fixed-size blocks or variable-sized blocks. Allocation by blocks eliminates external fragmentation, whereas allocation by variable-size blocks improves locality.

This allocation technique supports both sequential and direct access to the file and thus is the most popular form of file allocation.





FREE SPACE MANAGEMENT

It is not easy work for an operating system to allocate and de-allocate memory blocks (managing free space) simultaneously. The operating system uses various methods for adding free space and freeing up space after deleting a file. There are various methods using which a free space list can be implemented. We are going to explain them below-

1. Bitmap or Bit Vector :

A bit vector is a most frequently used method to implement the free space list. A bit vector is also known as a **Bit map**. It is a series or collection of bits in which each bit represents a disk block.

The values taken by the bits are either **1** or **0**. If the block bit is 1, it means the block is empty and if the block bit is 0, it means the block is not free. It is allocated to some files. Since all the blocks are empty initially so, each bit in the bit vector represents 0.

"Free block number" can be defined as that block which does not contain any value, i.e., they are free blocks.

The formula to find a free block number is :

[Block number = (number of bits per words)*(number of **0**-value word) + Offset of first **1** bit]

2. Linked List :

A **linked list** is another approach for free space management in an operating system. In it, all the free blocks inside a disk are linked together in a **linked list**. These free blocks on the disk are linked together by a pointer. These pointers of the free block contain the address of the next free block and the last pointer of the list points to null which indicates the end of the linked list.

This technique is not enough to traverse the list because we have to read each disk block one by one which requires I/O time.

The operating system can use this linked list to allocate memory blocks to processes as needed.

3. Grouping

The grouping technique is also called the **"modification of a linked list technique"**. In this method, first, the free block of memory contains the addresses of the **n-free** blocks. And the last free block of these **n** free blocks contains the addresses of the next **n** free block of memory and this keeps going on. This technique separates the empty and occupied blocks of space of memory.

4. Counting

In memory space, several files are created and deleted at the same time. For which memory blocks are allocated and de-allocated for the files. Creation of files occupy free blocks and deletion of file frees blocks.

When there is an entry in the free space, it consists of two parameters- **"address of first free disk block (a pointer)"** and **"a number 'n'"**.

SYSTEM CALLS

1. create

The create() function is used to create a new empty file in C. We can specify the permission and the name of the file which we want to create using the create() function. It is defined inside **<unistd.h>** header file and the flags that are passed as arguments are defined inside **<fcntl.h>** header file.

Syntax of create() in C

int **create**(char **filename*, mode_t *mode*);

Parameter

- **filename:** name of the file which you want to create
- **mode:** indicates permissions of the new file.

Return Value

- return first unused file descriptor (generally 3 when first creating use in the process because 0, 1, 2 fd are reserved)
- return -1 when an error

2. open

The open() function in C is used to open the file for reading, writing, or both. It is also capable of creating the file if it does not exist. It is defined inside **<unistd.h>** header file and the flags that are passed as arguments are defined inside **<fcntl.h>** header file.

Syntax of open() in C

int **open** (const char* *Path*, int *flags*);

Parameters

- **Path:** Path to the file which we want to open.
  - Use the **absolute path** beginning with "/" when you are **not working in the same directory** as the C source file.
  - Use **relative path** which is only the file name with extension, when you are **working in the same directory** as the C source file.
- **flags:** It is used to specify how you want to open the file. We can use the following flags.

Flags	Description
O_RDONLY	Opens the file in read-only mode.
O_WRONLY	Opens the file in write-only mode.
O_RDWR	Opens the file in read and write mode.

Flags	Description
O_CREAT	Create a file if it doesn't exist.
O_EXCL	Prevent creation if it already exists.
O_ APPEND	Opens the file and places the cursor at the end of the contents.
O_ASYNC	Enable input and output control by signal.
O_CLOEXEC	Enable close-on-exec mode on the open file.
O_NONBLOCK	Disables blocking of the file opened.
O_TMPFILE	Create an unnamed temporary file at the specified path.

3. close

The close() function in C tells the operating system that you are done with a file descriptor and closes the file pointed by the file descriptor. It is defined inside **<unistd.h>** header file.

Syntax of close() in C

int close(int fd);

Parameter

- **fd: F**ile descriptor of the file that you want to close.

Return Value

- **0** on success.
- **-1** on error.

4. read

From the file indicated by the file descriptor fd, the read() function reads the specified amount of bytes **cnt** of input into the memory area indicated by **buf**. The read() function is also defined inside the <unistd.h> header file.

Syntax of read() in C

size_t **read** (int *fd*, void* *buf*, size_t *cnt*);

Parameters

- **fd:** file descriptor of the file from which data is to be read.
- **buf:** buffer to read data from
- **cnt:** length of the buffer

Return Value

- return Number of bytes read on success
- return 0 on reaching the end of file
- return -1 on error
- return -1 on signal interrupt

5. write

Writes cnt bytes from buf to the file or socket associated with fd. cnt should not be greater than INT_MAX (defined in the limits.h header file). If cnt is zero, write() simply returns 0 without attempting any other action.

The write() is also defined inside **<unistd.h>** header file.

Syntax of write() in C

size_t **write** (int *fd*, void* *buf*, size_t *cnt*);

Parameters

- **fd:** file descriptor
- **buf:** buffer to write data to.
- **cnt:** length of the buffer.

Return Value

- returns the number of bytes written on success.
- return 0 on reaching the End of File.
- return -1 on error.
- return -1 on signal interrupts.

6. ioctl

- ioctl() is referred to as Input and Output Control.
- ioctl is a system call for device-specific input/output operations and other operations which cannot be expressed by regular system calls.

7. fork

- A new process is created by the fork() system call.
- A new process may be created with fork() without a new program being run-the new sub-process simply continues to execute exactly the same program that the first (parent) process was running.
- It is one of the most widely used system calls under process management.

8. exit

- The exit() system call is used by a program to terminate its execution.
- The operating system reclaims resources that were used by the process after the exit() system call.

9. exec

- A new program will start executing after a call to exec()
- Running a new program does not require that a new process be created first: any process may call exec() at any time. The currently running program is immediately terminated, and the new program starts executing in the context of the existing process.

10. wait

The **wait**() system call suspends execution of the current process until one of its children terminates. The call *wait(&status)* is equivalent to:

waitpid(-1, &status, 0);

11. waitpid

The **waitpid**() system call suspends execution of the current process until a child specified by *pid* argument has changed state. By default, **waitpid**() waits only for terminated children, but this behaviour is modifiable via the *options* argument, as described below.

The value of *pid* can be:

Tag	**Description**
< -1	meaning wait for any child process whose process group ID is equal to the absolute value of *pid.*
-1	meaning wait for any child process.
0	meaning wait for any child process whose process group ID is equal to that of the calling process.
> 0	meaning wait for the child whose process ID is equal to the value of *pid.*

DISK SCHEDULING ALGORITHMS

A process needs two type of time, CPU time and IO time. For I/O, it requests the Operating system to access the disk.

However, the operating system must be fare enough to satisfy each request and at the same time, operating system must maintain the efficiency and speed of process execution.

The technique that operating system uses to determine the request which is to be satisfied next is called disk scheduling.

Seek Time

Seek time is the time taken in locating the disk arm to a specified track where the read/write request will be satisfied.

Rotational Latency

It is the time taken by the desired sector to rotate itself to the position from where it can access the R/W heads.

Transfer Time

It is the time taken to transfer the data.

Disk Access Time

Disk access time is given as,

Disk Access Time = Rotational Latency + Seek Time + Transfer Time

Disk Response Time

It is the average of time spent by each request waiting for the IO operation.

Purpose of Disk Scheduling

The main purpose of disk scheduling algorithm is to select a disk request from the queue of IO requests and decide the schedule when this request will be processed.

Goal of Disk Scheduling Algorithm

- Fairness
- High throughout
- Minimal traveling head time

Disk Scheduling Algorithms

The list of various disks scheduling algorithm is given below. Each algorithm is carrying some advantages and disadvantages. The limitation of each algorithm leads to the evolution of a new algorithm.

- FCFS scheduling algorithm
- SSTF (shortest seek time first) algorithm
- SCAN scheduling
- C-SCAN scheduling
- LOOK Scheduling
- C-LOOK scheduling

FCFS Scheduling Algorithm

It is the simplest Disk Scheduling algorithm. It services the IO requests in the order in which they arrive. There is no starvation in this algorithm, every request is serviced.

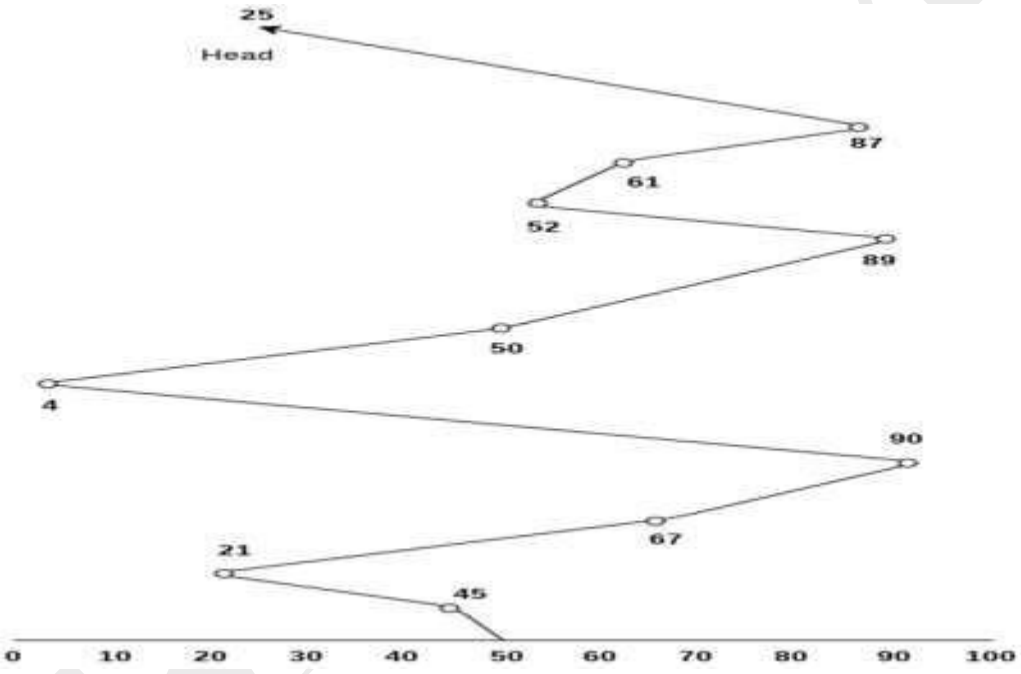
Disadvantages

- The scheme does not optimize the seek time.
- The request may come from different processes therefore there is the possibility of inappropriate movement of the head.

Example

Consider the following disk request sequence for a disk with 100 tracks 45, 21, 67, 90, 4, 50, 89, 52, 61, 87, 25. Head pointer starting at 50 and moving in left direction. Find the number of head movements in cylinders using FCFS scheduling.

Solution



Number of cylinders moved by the head

= (50-45)+(45-21)+(67-21)+(90-67)+(90-4)+(50-4)+(89-50)+(61-52)+(87-61)+(87-25)

= 5 + 24 + 46 + 23 + 86 + 46 + 49 + 9 + 26 + 62

= 376

SSTF Scheduling Algorithm

Shortest seek time first (SSTF) algorithm selects the disk I/O request which requires the least disk arm movement from its current position regardless of the direction.

It reduces the total seek time as compared to FCFS.

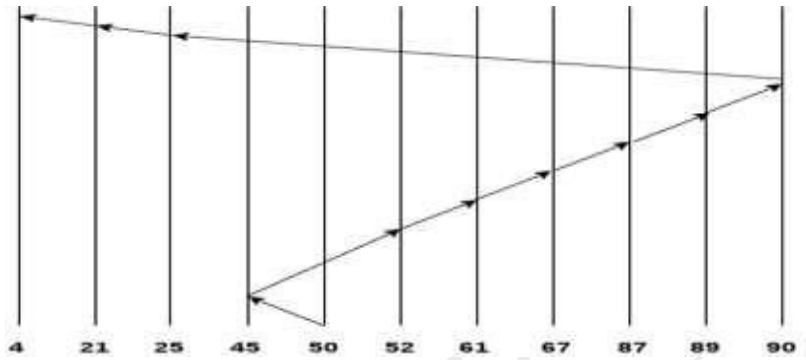
It allows the head to move to the closest track in the service queue.

Disadvantages

- It may cause starvation for some requests.
- Switching direction on the frequent basis slows the working of algorithm.
- It is not the most optimal algorithm.

Example

Consider the following disk request sequence for a disk with 100 tracks 45, 21, 67, 90, 4, 89, 52, 61, 87, 25. Head pointer starting at 50. Find the number of head movements in cylinders using SSTF scheduling.



Number of cylinders = 5 + 7 + 9 + 6 + 20 + 2 + 1 + 65 + 4 + 17 = 136

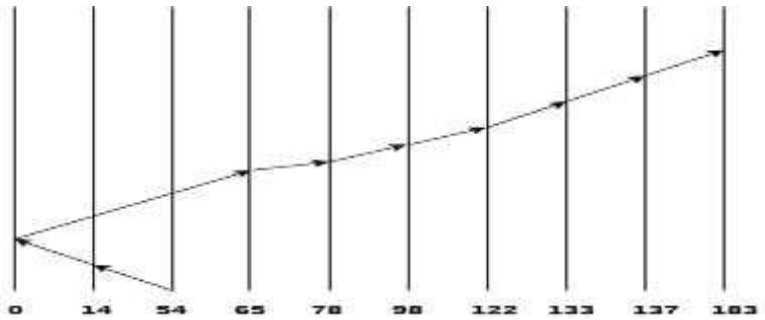
SCAN Algorithm

It is also called as Elevator Algorithm. In this algorithm, the disk arm moves into a particular direction till the end, satisfying all the requests coming in its path and then it turns back and moves in the reverse direction satisfying requests coming in its path.

It works in the way an elevator works, elevator moves in a direction completely till the last floor of that direction and then turns back.

Example

Consider the following disk request sequence for a disk with 100 tracks 98, 137, 122, 183, 14, 133, 65, 78. Head pointer starting at 54 and moving in left direction. Find the number of head movements in cylinders using SCAN scheduling.



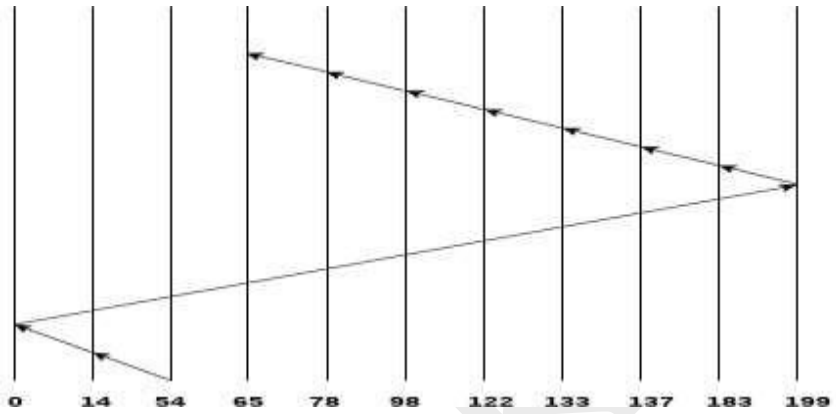
Number of Cylinders = 40 + 14 + 65 + 13 + 20 + 24 + 11 + 4 + 46 = 237

C-SCAN algorithm

In C-SCAN algorithm, the arm of the disk moves in a particular direction servicing requests until it reaches the last cylinder, then it jumps to the last cylinder of the opposite direction without servicing any request then it turns back and start moving in that direction servicing the remaining requests.

Example

Consider the following disk request sequence for a disk with 100 tracks 98, 137, 122, 183, 14, 133, 65, 78. Head pointer starting at 54 and moving in left direction.



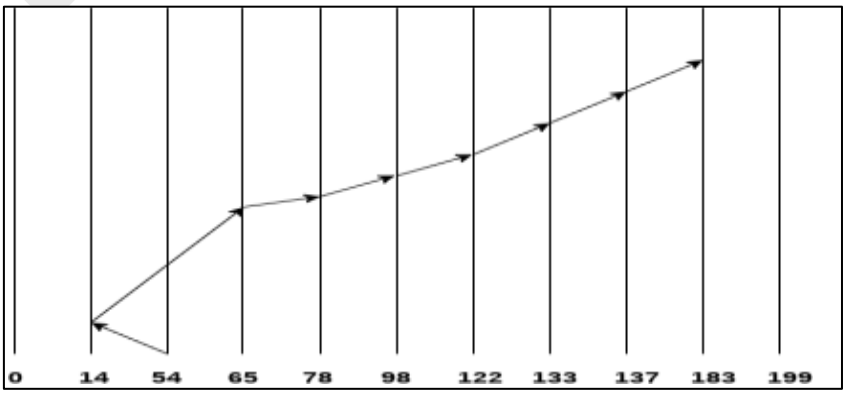
No. of cylinders crossed = 40 + 14 + 199 + 16 + 46 + 4 + 11 + 24 + 20 + 13 = 387

LOOK Scheduling

It is like SCAN scheduling Algorithm to some extant except the difference that, in this scheduling algorithm, the arm of the disk stops moving inwards (or outwards) when no more request in that direction exists. This algorithm tries to overcome the overhead of SCAN algorithm which forces disk arm to move in one direction till the end regardless of knowing if any request exists in the direction or not.

Example

Consider the following disk request sequence for a disk with 100 tracks 98, 137, 122, 183, 14, 133, 65, 78. Head pointer starting at 54 and moving in left direction.



Number of cylinders crossed = 40 + 51 + 13 + +20 + 24 + 11 + 4 + 46 = 209

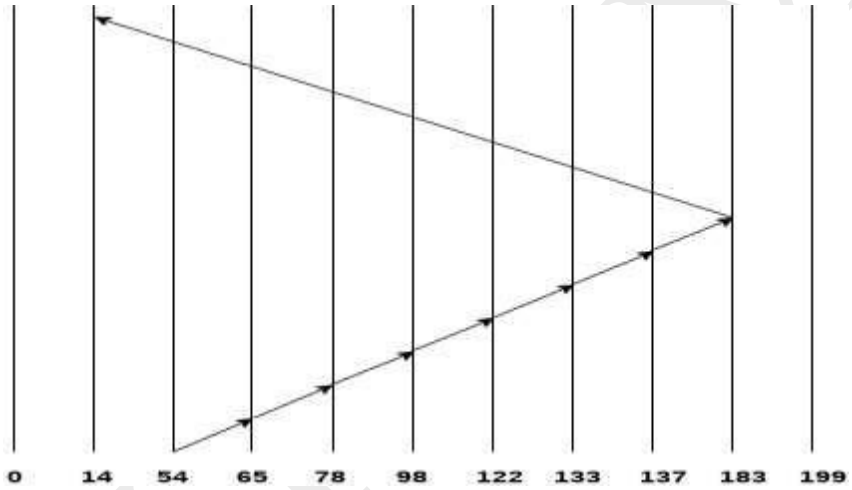
C LOOK Scheduling

C Look Algorithm is similar to C-SCAN algorithm to some extent. In this algorithm, the arm of the disk moves outwards servicing requests until it reaches the highest request cylinder, then it jumps to the lowest request cylinder without servicing any request then it again start moving outwards servicing the remaining requests.

It is different from C SCAN algorithm in the sense that, C SCAN force the disk arm to move till the last cylinder regardless of knowing whether any request is to be serviced on that cylinder or not.

Example

Consider the following disk request sequence for a disk with 100 tracks
98, 137, 122, 183, 14, 133, 65, 78. Head pointer starting at 54 and moving in left direction. Find the number of head movements in cylinders using C LOOK scheduling.



Number of cylinders crossed = 11 + 13 + 20 + 24 + 11 + 4 + 46 + 169 = 298