# Operating Systems Notes

**Contributor:** Abhishek Sharma

[Founder at TutorialsDuniya.com]

# Computer Science Notes

UNIT-I

## 1) Define OS and Explain about the Types of Operating System.

A) Definition: "**Operating System** is system software that works as an interface between a user and the computer hardware."

### Batch operating system

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

The problems with Batch Systems are as follows -

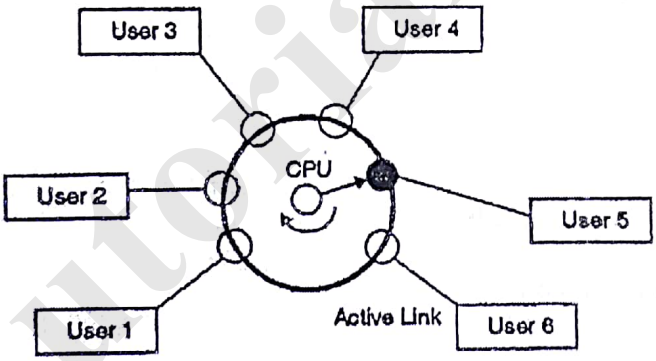
- Lack of interaction between the user and the job.
- CPU is often idle, because the speed of the mechanical I/O devices is slower than the CPU
- Difficult to provide the desired priority.

### Time-sharing operating systems

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

The main aim of Time-Sharing Systems, the objective is to minimize response time.

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




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

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

Advantages of Timesharing operating systems are as follows –

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

Disadvantages of Time-sharing operating systems are as follows –

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

3 **Distributed operating System:**

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

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

The advantages of distributed systems are as follows –

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

4 **Network operating System:**

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

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

The advantages of network operating systems are as follows –

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

The disadvantages of network operating systems are as follows –

- High cost of buying and running a server.

- Dependency on a central location for most operations.
- Regular maintenance and updates are required.

**Real Time operating System:**

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

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

There are two types of real-time operating systems.

Hard real-time systems

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

Soft real-time systems

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

## 2) Explain about the Operating System – Services.

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

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

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

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

**User Interface**

One is a **command-line interface(CLI)**, which uses text commands and a method for entering them. Most commonly, a **graphical user interface (GUI)** is used. Here, the interface is a window system with a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text.

### Program execution

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

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

- Loads a program into memory.
- Executes the program.
- Handles program's execution.
- Provides a mechanism for process synchronization.
- Provides a mechanism for process communication.
- Provides a mechanism for deadlock handling.

### I/O Operation

An I/O subsystem comprises of I/O devices and their corresponding driver software. Drivers hide the peculiarities of specific hardware devices from the users.

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

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

### File system manipulation

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

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

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

### Communication

In case of distributed systems which are a collection of processors that do not share memory, peripheral devices, or a clock, the operating system manages communications between all the processes. Multiple processes communicate with one another through communication lines in the network.

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

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

Error handling

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

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

Resource Management

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

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

Protection

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

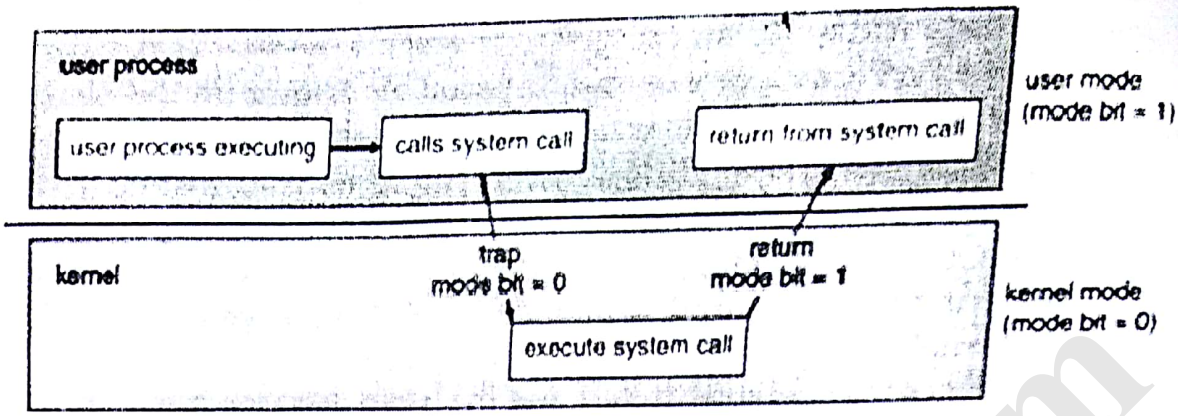
Protection refers to a mechanism or a way to control the access of programs, processes, or users to the resources defined by a computer system. Following are the major activities of an operating system with respect to protection –

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

**3) Define system call. And explain any 4 type with examples.**

A)

**system call Definition**: "system call is a request from an application program for the operating system to perform some hardware action on behalf of the application. System calls are initiated with a software interrupt assembly language instruction".




The system call provides an interface to the operating system services. Application developers often do not have direct access to the system calls, but can access them through an application programming interface (API). The functions that are included in the API invoke the actual system calls. By using the API, certain benefits can be gained:

- Portability: as long a system supports an API, any program using that API can compile and run.
- Ease of Use: using the API can be significantly easier than using the actual system call.
- Types of System Calls

There are 5 different categories of system calls:

1.process control, 2.file manipulation, 3.device manipulation, 4.information maintenance and 5.communication.

## 1 Process Control

A running program needs to be able to stop execution either normally or abnormally. When execution is stopped abnormally, often a dump of memory is taken and can be examined with a debugger.

Process control

1. end, abort
2. load, execute
3. create process, terminate process
4. get process attributes, set process attributes
5. wait for time
6. wait event, signal event
7. allocate and free memory

## 2 File Management

Some common system calls are *create*, *delete*, *read*, *write*, *reposition*, or *close*. Also, there is a need to determine the file attributes – *get* and *set* file attribute. Many times the OS provides an API to make these system calls.

- File management system calls

° create file, delete file
o open, close
° read, write, reposition

o get file attributes, set file attributes

## Device Management

Process usually require several resources to execute, if these resources are available, they will be granted and control returned to the user process. These resources are also thought of as devices. Some are physical, such as a video card, and others are abstract, such as a file.

User programs *request* the device, and when finished they *release* the device. Similar to files, we can *read*, *write*, and *reposition* the device.

- Device management system calls

o request device, release device
° read, write, reposition
o get device attributes, set device attributes

° logically attach or detach devices

## Information Management

Some system calls exist purely for transferring information between the user program and the operating system. An example of this is *time*, or *date*.

The OS also keeps information about all its processes and provides system calls to report this information.

- Information maintenance system calls

o get time or date, set time or date
o get system data, set system data
o get process, file, or device attributes

o set process, file, or device attributes

## Communication

There are two models of interprocess communication, the message-passing model and the shared memory model.

- Message-passing uses a common mailbox to pass messages between processes.
- Shared memory use certain system calls to create and gain access to create and gain access to regions of memory owned by other processes. The two processes exchange information by reading and writing in the shared data.
- Communication system calls
- o create, delete communication connection
- ° send, receive messages
- o transfer status information

o attach or detach remote devices

## 4 OPERATING SYSTEM OPERATIONS:

If there are no processes to execute, OS will sit idle and wait for some event to happen. Interrupts could be hardware interrupts or software interrupts. The OS is designed to handle both. A trap (or an exception) is a software generated interrupt caused either by an error (e.g. divide by zero) or by a specific request from a user program. A separate code segment is written in the OS to handle different types of interrupts.

These codes are known as interrupt handlers/ interrupt service routine. A properly designed OS ensures that an illegal program should not harm the execution of other programs. To ensure this, the OS operates in dual mode.

**Dual mode of operation**

The OS is design in such a way that it is capable of differentiating between the execution of OS code and user defined code. To achieve this OS need two different modes of operations this is thereby controlled by mode bit added to hardware of computer system as shown in Table 4.

| Mode Type | Definition | Mode Bit | Examples |
|---|---|---|---|
| User Mode | User Defined codes are executed | Mode Bit=1 | Creation of word document or in general user using any application program |
| Kernel Mode | OS system codes are executed (also known as supervisor, system, or privileged mode) | Mode Bit=0 | Handling interrupts-<br>Transferring control of a process from CPU to I/O on request |

**Table: User and Kernel Mode of Operating System**

**Transition from User to Kernel mode**

When a user application is executing on the computer system OS is working in user mode. On signal of system call via user application, the OS transits from user mode to kernel mode to service that request as shown below.

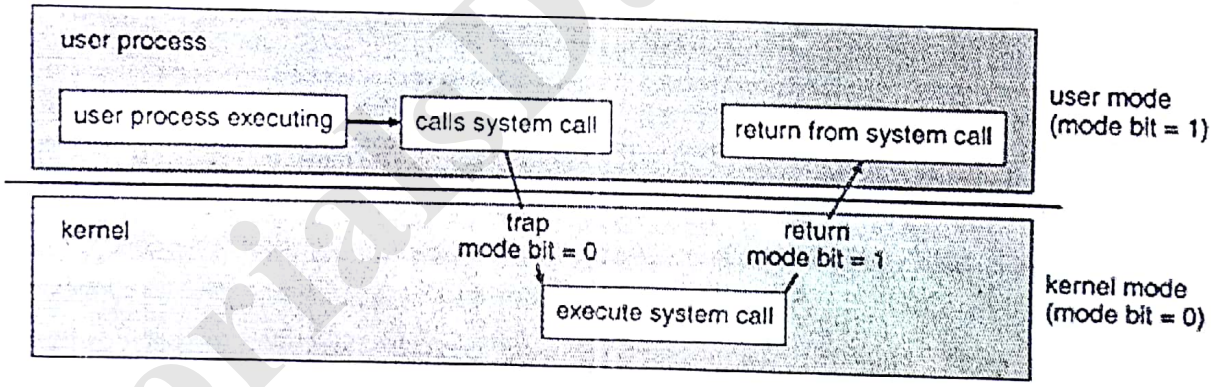



**Fig: Transition from user to kernel mode**

When the user starts the system the hardware starts in monitor/ kernel mode and loads the operating system. OS has the initial control over the entire system, when instructions are executed in kernel mode. OS then starts the user processes in user mode and on occurrence of trap, interrupt or system call again switch to kernel mode and gains control of the system. System calls are designed for the user programs through which user can ask OS to perform tasks reserved for operating system. System calls usually take the form of the trap. Once the OS service the interrupt it transfers control back to user program hence user mode by setting mode bit=1.

## Process Management

A program does nothing unless its instructions are executed by a CPU. A program in execution, as mentioned, is a process. compiler is a process, A word-processing is a process. A process needs certain resources—including CPU time, memory, files, and I/O devices—to accomplish its task. These resources

are either given to the process when it is created or allocated to it while it is running. a program is a *passive* entity, such as the contents of a file stored on disk, whereas a process is an *active* entity. A single-threaded process has one **program counter** specifying the next instruction to execute.
The operating system is responsible for the following activities in connection with process management:

- **Creating and deleting both user and system processes.**
- **Suspending and resuming processes.**
- **Providing mechanisms for process synchronization.**
- **Providing mechanisms for process communication.**
- **Providing mechanisms for deadlock handling.**

## Memory Management:

Operating System also Manages the Memory of the Computer System means Provide the Memory to the Process and Also Deallocate the Memory from the Process. And also defines that if a Process gets completed then this will deallocate the Memory from the Processes. Main memory is a repository of quickly accessible data shared by the CPU and I/O devices. The central processor reads instructions from main memory during the instruction-fetch cycle and both reads and writes data from main memory during the data-fetch cycle

The operating system is responsible for the following activities in connection with memory management:

- **Keeping track of which parts of memory are currently being used and by whom**
- **Deciding which processes (or parts thereof) and data to move into and out of memory**
- **Allocating and deallocating memory space as needed**

## File-System Management:

A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data. Operating System also Controls the all the Storage Operations means how the data or files will be Stored into memory the and how the Files will be Accessed by the users etc. All the Operations those are Responsible for Storing and Accessing the Files is determined by the Operating System Operating System also Allows us Creation of Files, Creation of Directories and Reading and Writing the data of Files and Directories and also Copy the contents of the Files and the Directories from One Place to Another Place.

Creating and deleting files

- Creating and deleting directories to organize files
- Supporting primitives for manipulating files and directories
- Mapping files onto secondary storage
- Backing up files on stable (nonvolatile) storage media

## Mass-Storage Management:

Most programs—including compilers, assemblers, word processors, editors, and formatters—are stored on a disk until loaded into memory and then use the disk as both the source and destination of their processing. Hence, the proper management of disk storage is of central importance to a computer system.
The operating system is responsible for the following activities in connection with disk management:

- Free-space management
- Storage allocation
- Disk scheduling

Because secondary storage is used frequently, it must be used efficiently. The entire speed of operation of a computer may hinge on the speeds of the disk subsystem and of the algorithms that manipulate that subsystem.

**Caching:** it is an important principle of computer systems. As all we know all information is kept in some storage such as **main memory**, as it is used, it is copied into a faster storage system-the **cache**-on a temporary basis. When we need a particular piece of information, we first check whether it is in cache or not, if it is in the cache we use it directly from the cache, otherwise we use the information from the source from where it is present and made a copy of that information in the cache assuming that we will need it again soon. By doing so we decreases the searching time of that particular information.

## Protection and Security:

**Protection** is a mechanism for controlling the access of processes or users to the resources defined by a computer system. Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by another subsystem that is malfunctioning. Protection and security require the system to be able to distinguish among all its users. Most operating systems maintain a list of user names and associated **user identifiers (user IDs).**

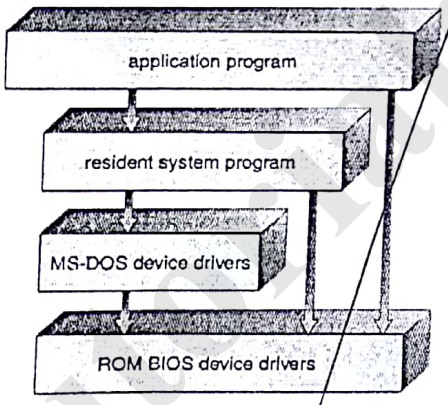
Group functionality can be implemented as a system-wide list of group names and **group identifiers.** A user can be in one or more groups, depending on operating-system design decisions. The user's group IDs are also included in every associated process and thread.

5 MS-DOS System Structure

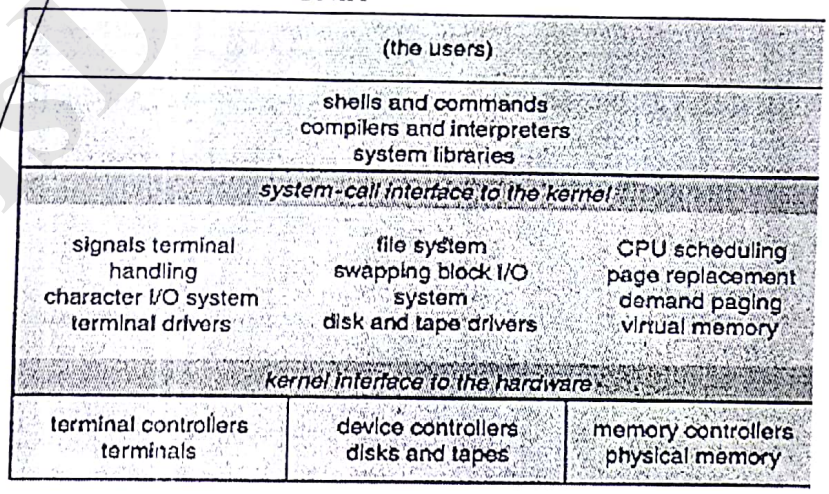
MS-DOS – written to provide the most functionality in the least space
1. not divided into modules
2. Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated

In MS-DOS, the interfaces and levels of functionality are not well separated. For instance, application programs are able to access the basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system crashes when user programs fail.

MS-DOS Layer Structure




UNIX




the original UNIX operating system. UNIX is another system that initially was limited by hardware functionality. Everything below the system call interface and above the physical hardware is the kernel. The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls.

UNIX -- limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts.

1. Systems programs
2. The kernel
3. Consists of everything below the system-call interface and above the physical hardware
4. Provides the file system, CPU scheduling, memory management, and other operating-system functions.

5.a large number of functions for one level.

## Layered Approach:

A system can be made modular in many ways. One method is the **layered approach,** in which the operating system is broken up into a number of layers (levels). The bottom layer (layer 0) id the hardware; the highest (layer N) is the user interface.

The main advantage of the layered approach is simplicity of construction and debugging. The layers are selected so that each uses functions (operations) and services of only lower-level layers. This approach simplifies debugging and system verification. The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware (which is assumed correct) to implement its functions. Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on. If an error is found during debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged. Thus, the design and implementation of the system is simplified.

Advantages of layered operating systems are:

1. It is decomposable and therefore effects separation of concerns and different abstraction levels
2. It allows good maintenance, where you can make changes without affecting layer interfaces

- Disadvantages of layered operating systems are:

1. It is difficult to exactly assign of functionalities to the correct and appropriate layer.
2. Because of having too many layers, performance of the system is degraded. )5

6

## About kernel:

Kernel is the core part of an operating system; it manages the system resources. Kernel is like a bridge between application and hardware of the computer. The Kernel can be classified further into two categories, Microkernel and Monolithic Kernel.

## Definition of Microkernel:

Microkernel being a kernel manages all system resources. But in a microkernel, the **user services** and the **kernel services** are implemented in different address space. The user services are kept in **user address space**, and kernel services are kept under **kernel address space**. This **reduces** the size of the kernel and further reduces the size of operating system.

In addition to the communication between application and hardware of the system, the microkernel provides minimal services of process and memory management. The communication between the client program/application and services running in user address space is established through message passing. They never interact directly. This reduces the speed of execution of microkernel.

In a microkernel, the user services are isolated from kernel services so if any user service fails it does not affect the kernel service and hence Operating system **remain unaffected**. This is one of the advantages in the microkernel. The microkernel is easily **extendable**. If the new services are to be added, they are added to user address space and hence, the kernel space do not require any modification. The microkernel is also easily portable, secure and reliable.

## Comparison Chart

| BASIS FOR COMPARISON | MICROKERNEL | MONOLITHIC KERNEL |
|---|---|---|
| Basic | In microkernel user services and | In monolithic kernel, both kernel, user services and kernel |

| BASIS FOR COMPARISON | MICROKERNEL | MONOLITHIC KERNEL |
|---|---|---|
| | services are kept in separate address space. | services are kept in the same address space. |
| Size | Microkernel are smaller in size. | Monolithic kernel is larger than microkernel. |
| Execution | Slow execution. | Fast execution. |
| Extendible | The microkernel is easily extendible. | The monolithic kernel is hard to extend. |
| Security | If a service crashes, it does effect on working of microkernel. | If a service crashes, the whole system crashes in monolithic kernel. |
| Code | To write a microkernel, more code is required. | To write a monolithic kernel, less code is required. |
| Example | QNX, L4Linux,Mac os x, Minix. | Linux, Windows (95,98), Solaris. |

**Multi programming:** Multiprogramming is the technique of running several programs at a time using timesharing. It allows a computer to do several things at the same time. Multiprogramming creates logical parallelism. The concept of multiprogramming is that the operating system keeps several jobs in memory simultaneously.

**Multitasking**

Multitasking has the same meaning of multiprogramming but in a more general sense, as it refers to having multiple (programs, processes, tasks, threads) running at the same time. This term is used in modern operating systems when multiple tasks share a common processing resource (e.g., CPU and Memory). At any time the CPU is executing one task only while other tasks waiting their turn. The

illusion of parallelism is achieved when the CPU is reassigned to another task (i.e. *process* or *thread context switching*).

There are subtle differences between multitasking and multiprogramming. A *task* in a multitasking operating system is not a whole application program but it can also refer to a "thread of execution" when one process is divided into sub-tasks. Each smaller task does not hijack the CPU until it finishes like in the older multiprogramming but rather a fair share amount of the CPU time called quantum.

## MODELS OF DISTRIBUTED SYSTEMS IN OS

### Client-Server

The client-server model is probably the most popular paradigm. The server is responsible for accepting, processing, and replying to requests. It is the producer. The client is purely the consumer. It requests the services of the server and accepts the results.

The basic web follows the client-server model. Your browser is the client. It requests web pages from a server (e.g., google.com), waits for results, and displays them for the user.

### Peer-To-Peer Model

The peer-to-peer model assumes that each entity in the network has equivalent functionality. In essence, it can play the role of a client or a server. Ideally, this reduces bottlenecks and enables each entity to contribute resources to the system. Unfortunately, it doesn't always work that way. enabling communication in such a system is challenging. First, peers must locate other peers in order to participate in the system. This is rarely done in a truly distributed or peer-to-peer fashion.

### Network Computers/Thin Clients

The network computer model assumes that the end user machine is a low-end computer that maintains a minimal OS. When it boots, it retrieves the OS and files/applications from a central server and runs applications locally. The thin client model is similar, though assumes that the process runs remotely and the client machine simply displays results (e.g., X-windows and VNC).

This model has been around for quite some time, but has recently received much attention. Google and Amazon are both promoting "cloud computing". Sun's Sun Ray technology also makes for an interesting demonstration. Though this model has yet to see success, it is beginning to look more promising.

### Mobile Devices

There is an increasing need to develop distributed systems that can run atop devices such as cell phones, cameras, and MP3 players. Unlike traditional distributed computing entities, which communicate over the Internet or standard local area networks, these devices often communicate via wireless technologies such as Bluetooth or other low bandwidth and/or short range mechanisms. As a result, the geographic location of the devices impacts system design. Moreover, mobile systems must take care to consider the battery constraints of the participating devices. System design for mobile ad hoc networks (MANETs), sensor networks, and delay/disruption tolerant networks (DTNs) is a very active area of research.

# TutorialsDuniya.com

**Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com**

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

**Please Share these Notes with your Friends as well**

OS II-UNIT

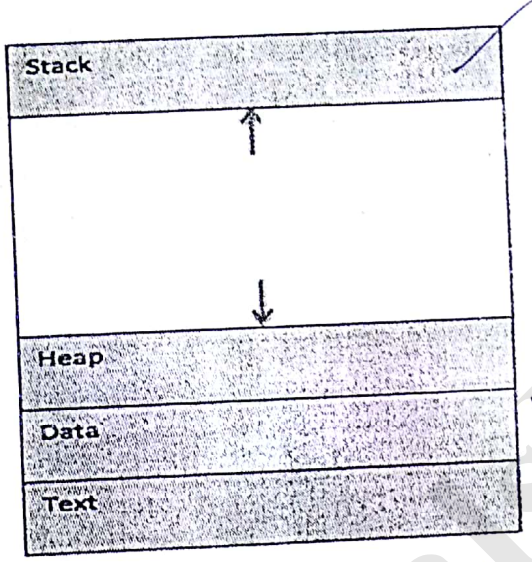
# 1) Process

**"A process is basically a program in execution. The execution of a process must progress in a sequential fashion."**

A process is defined as an entity which represents the basic unit of work to be implemented in the system.

To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program.

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




**Stack:** The process Stack contains the temporary data such as method/function parameters, return address and local variables.

**Heap:** This is dynamically allocated memory to a process during its run time.

**Text:** This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.

**Data:** This section contains the global and static variables.

## Program

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

```
1
```

```
#include <stdio.h>

int main() {
  printf("Hello, World! \n");
  return 0;
}
```

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

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

## 2) Process Life Cycle

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

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

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

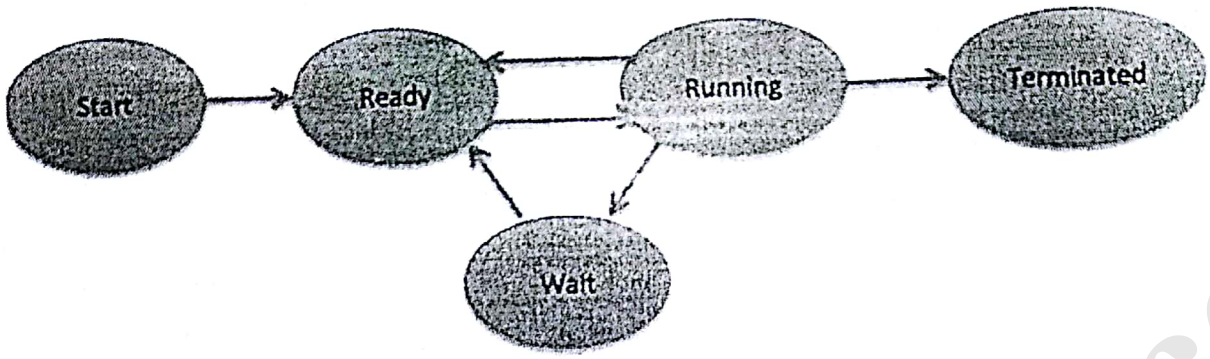
**Ready :** The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after **Start** state or while running it by but interrupted by the scheduler to assign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

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

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

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

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




## 3)Process Control Block (PCB)

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

**Process State:** The current state of the process i.e., whether it is ready, running, waiting, or whatever.

**Process privileges :** This is required to allow/disallow access to system resources.

**Process ID:** Unique identification for each of the process in the operating system.

**Pointer :** A pointer to parent process.

**Program Counter :** Program Counter is a pointer to the address of the next instruction to be executed for this process.

**CPU registers:** Various CPU registers where process need to be stored for execution for running state.

**CPU Scheduling Information :**Process priority and other scheduling information which is required to schedule the process.

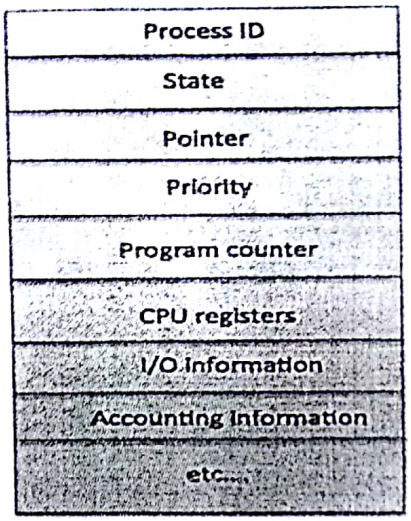
**Memory management information:** This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.

**Accounting information:** This includes the amount of CPU used for process execution, time limits, execution ID etc.

**IO status information :** This includes a list of I/O devices allocated to the process.

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

OS II-UNIT




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

## 4)OPERATING SYSTEM - PROCESS SCHEDULING

### Definition

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

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

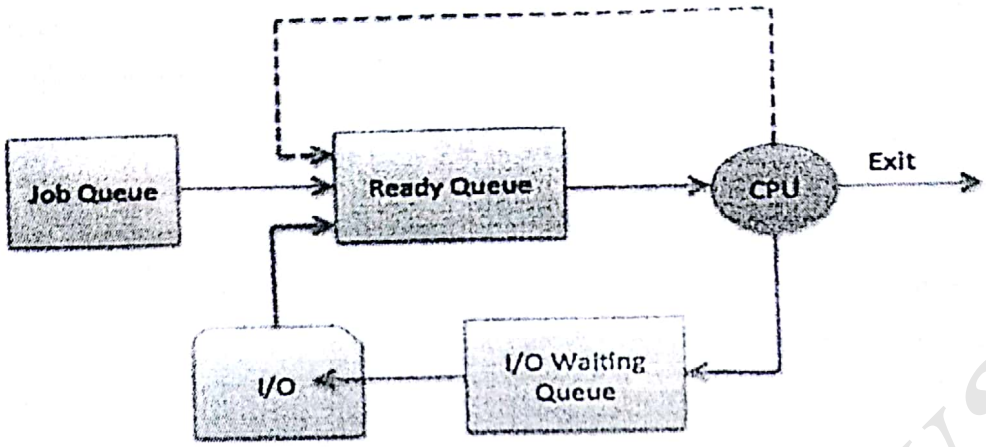
### Process Scheduling Queues

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

The Operating System maintains the following important process scheduling queues –

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

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




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

## Two-State Process Model

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

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

OS II-UNIT

# 5)Schedulers

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

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

## Long Term Scheduler

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

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

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

## Short Term Scheduler

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

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

## Medium Term Scheduler

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

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

## Comparison among Scheduler

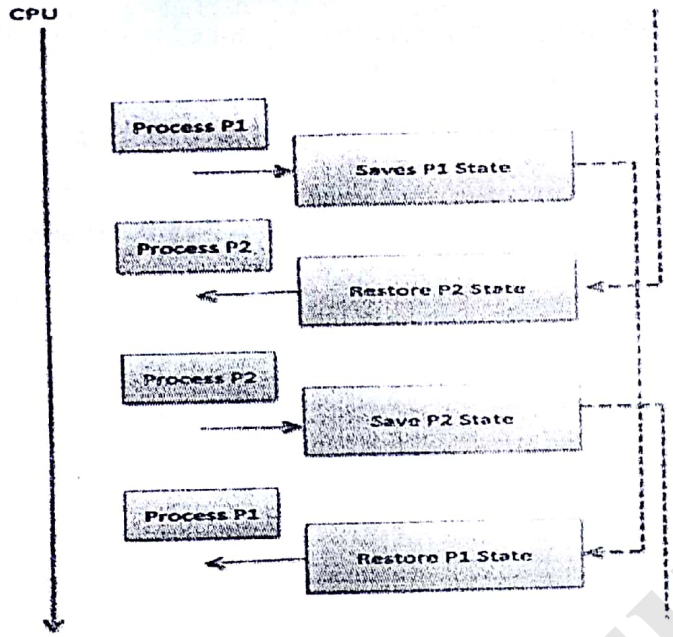
| S.N. | Long-Term Scheduler | Short-Term Scheduler | Medium-Term Scheduler |
|---|---|---|---|
| 1 | It is a job scheduler | It is a CPU scheduler | It is a process swapping scheduler. |
| 2 | Speed is lesser than short term scheduler | Speed is fastest among other two | Speed is in between both short and long term scheduler. |
| 3 | It controls the degree of multiprogramming | It provides lesser control over degree of multiprogramming | It reduces the degree of multiprogramming. |
| 4 | It is almost absent or minimal in time sharing system | It is also minimal in time sharing system | It is a part of Time sharing systems. |
| 5 | It selects processes from pool and loads them into memory for execution | It selects those processes which are ready to execute | It can re-introduce the process into memory and execution can be continued. |

## 6)Context Switch

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

When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block. After this, the state for

the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.




Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers.

Operating System - Multi-Threading

## What is Thread?

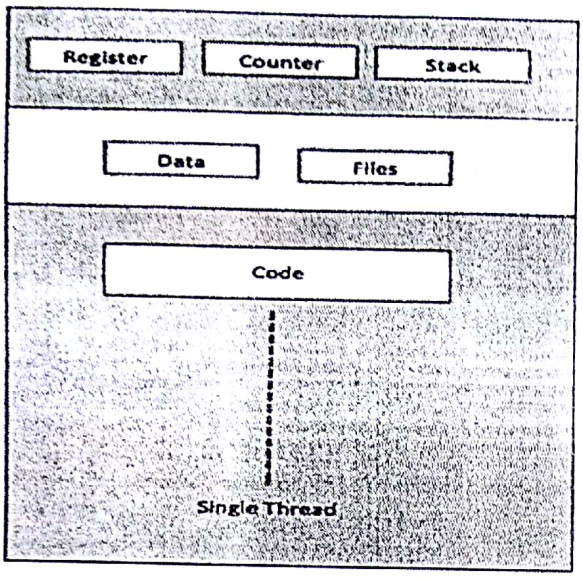
A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.

A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

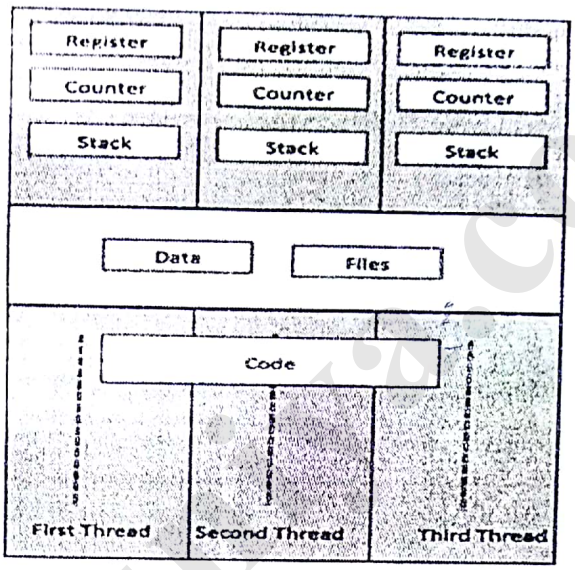
A thread is also called a **lightweight process**. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. Threads have been successfully used in implementing

network servers and web server. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors. The following figure shows the working of a single-threaded and a multithreaded process.




Single Process P with single thread




Single Process P with three threads

## 6)Difference between Process and Thread

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

| | | |
|---|---|---|
| | more resources. | resources. |
| 6 | In multiple processes each process operates independently of the others. | One thread can read, write or change another thread's data. |

## Advantages of Thread

- Threads minimize the context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- It is more economical to create and context switch threads.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

## 7)User Level thread Vs Kernel Level thread

| USER LEVEL THREAD | KERNEL LEVEL THREAD |
|---|---|
| User thread are implemented by users. | kernel threads are implemented by OS. |
| OS doesn't recognized user level threads. | Kernel threads are recognized by OS. |
| Implementation of User threads is easy. | Implementation of Kernel thread is complicated. |
| Context switch time is less. | Context switch time is more. |
| Context switch requires no hardware support. | Hardware support is needed. |
| If one user level thread perform blocking operation then entire process will be blocked. | If one kernel thread perform blocking operation then another thread can continue execution. |
| Example : Java thread, POSIX threads. | Example : Window Solaris. |

# 8)Multithreading Models

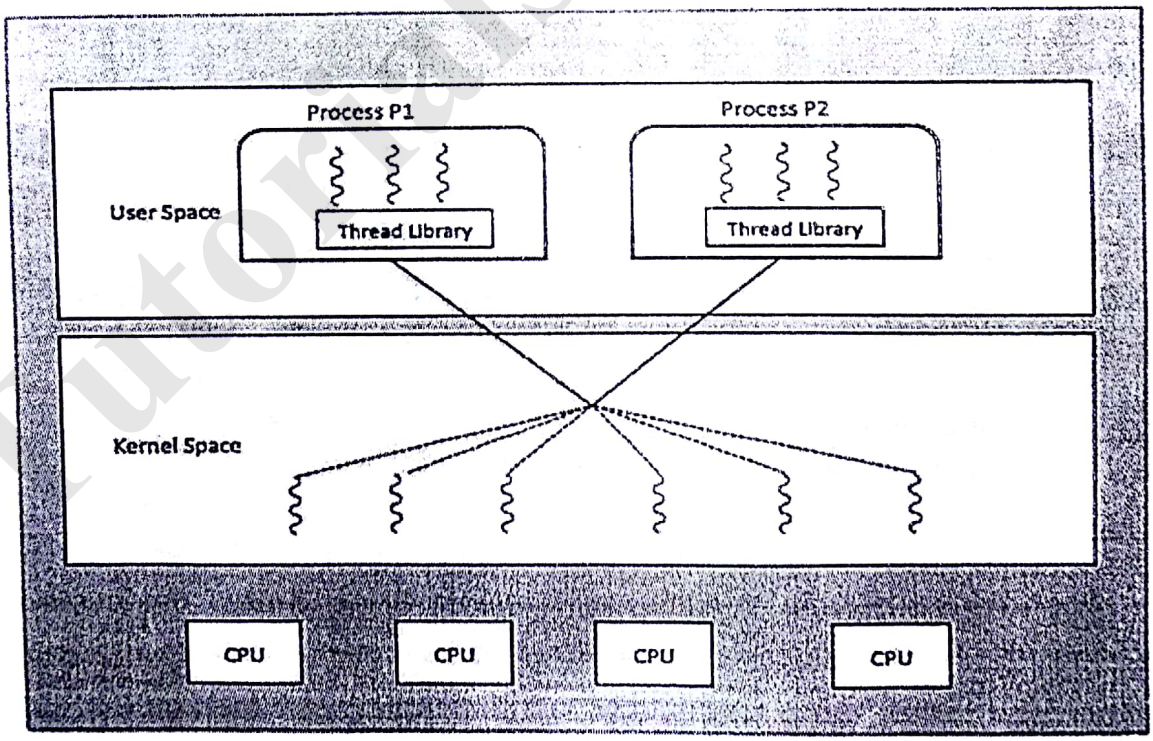
Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the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

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

## Many to Many Model

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

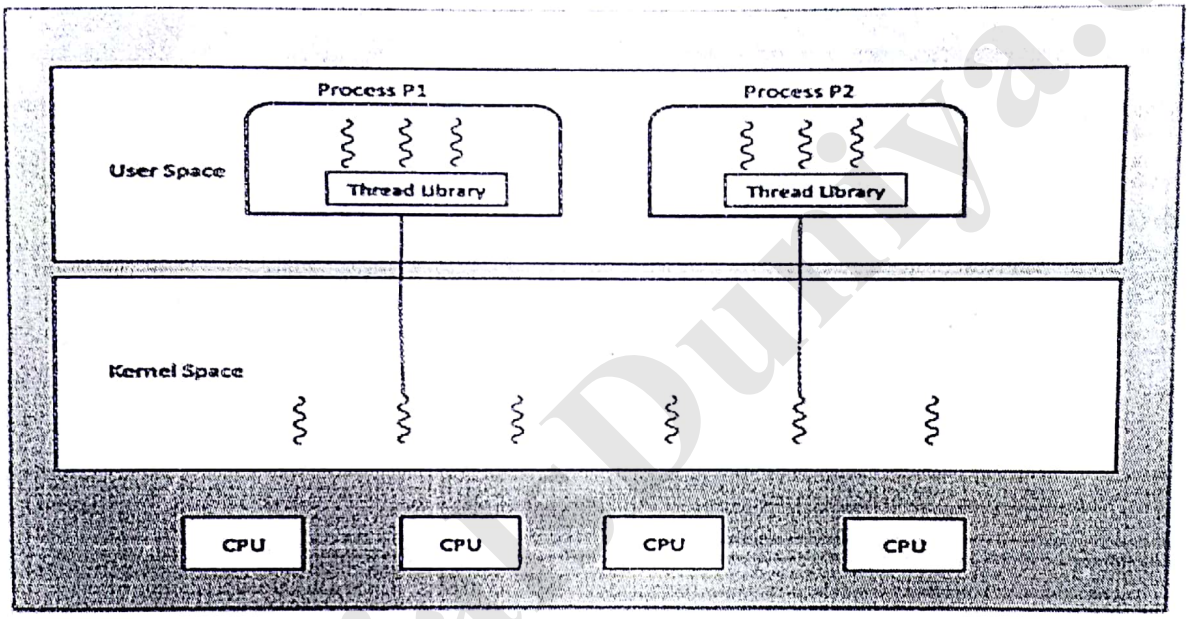
The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model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



## Many to One Model

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

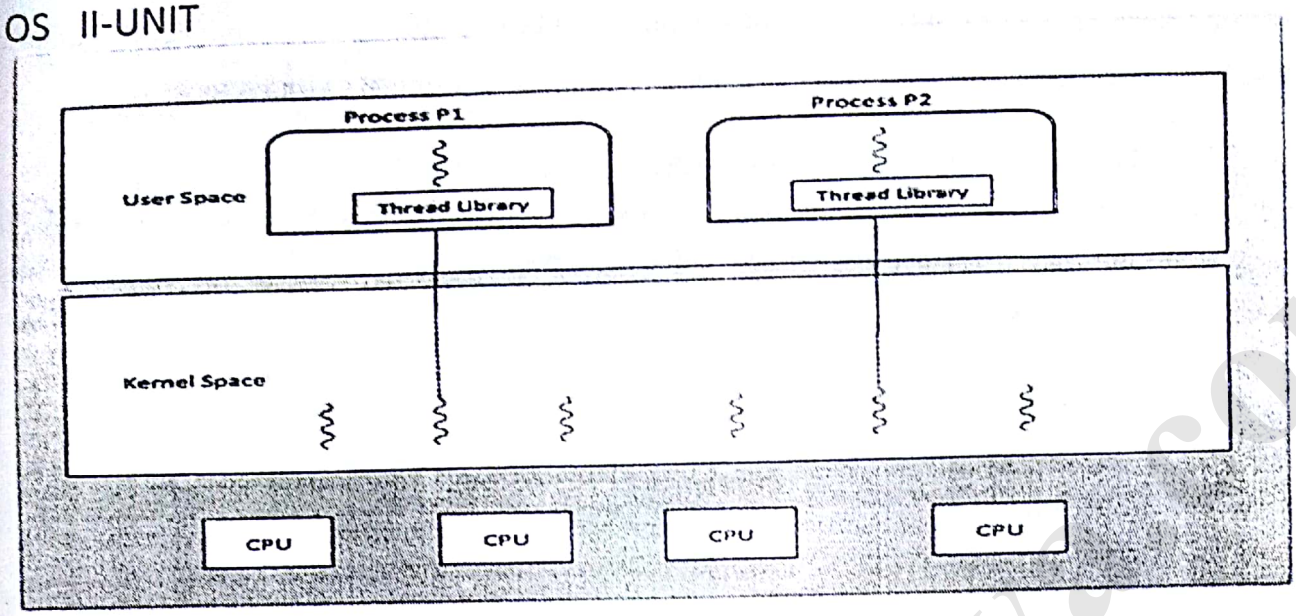
If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.




## One to One Model

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

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




## 9)THE ADVANTAGES OF MULTITHREADED PROGRAMMING:

**1. Responsiveness:**

Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. For instance, a multithreaded web browser could still allow user interaction in one thread while an image was being loaded in another thread.

**2. Resource sharing:**

By default, threads share the memory and the resources of the process to which they belong. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

**3. Economy:**

Allocating memory and resources for process creation is costly. Because threads share resources of the process to which they belong, it is more economical to create and context-switch threads. Empirically gauging the difference in overhead can be difficult, but in general it is much more time consuming to create and manage processes than threads. In Solaris, for example, creating a process is about thirty times slower than is creating a thread, and context switching is about five times slower.

**4. Utilization of multiprocessor architectures:**

The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors. A single threaded process can only

run on one CPU, no matter how many are available. Multithreading on a multi-CPU machine increases concurrency.

## 10)OPERATING SYSTEM SCHEDULING ALGORITHMS

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

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

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

## First Come First Serve (FCFS)

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

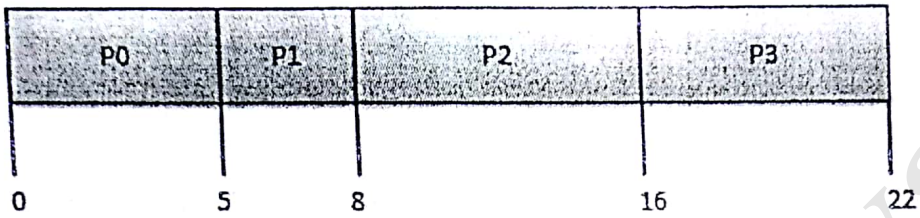
# TutorialsDuniya.com

**Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com**

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

**Please Share these Notes with your Friends as well**

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




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

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

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

## Shortest Job Next (SJN)

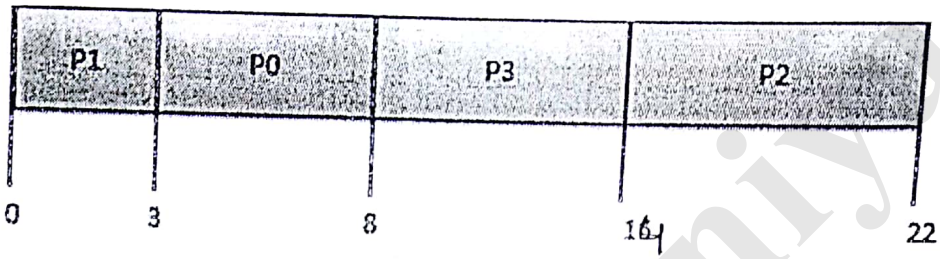
- This is also known as **shortest job first**, or SJF
- This is a non-preemptive, pre-emptive scheduling algorithm.
- Best approach to minimize waiting time.
- Easy to implement in Batch systems where required CPU time is known in advance.
- Impossible to implement in interactive systems where required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

W.T = Tat - B.T



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

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




Wait time of each process is as follows –

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

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

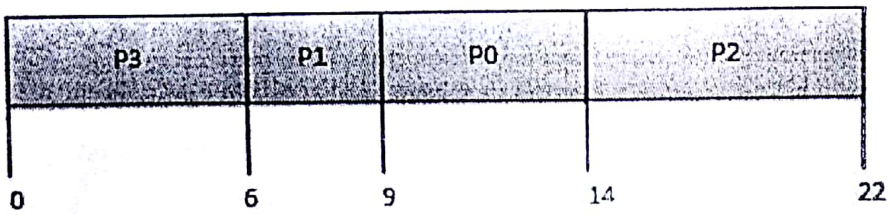
# Priority Based Scheduling

- Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.
- Each process is assigned a priority. Process with highest priority is to be executed first and so on.
- Processes with same priority are executed on first come first served basis.

- Priority can be decided based on memory requirements, time requirements or any other resource requirement.

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

at a point of time → comes into running state




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

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

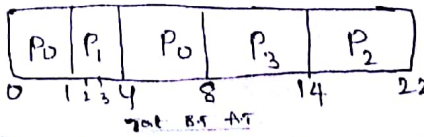
Average Wait Time: (9+5+12+0) / 4 = 6.5

## Shortest Remaining Time

- Shortest remaining time (SRT) is the preemptive version of the SJN algorithm.
- The processor is allocated to the job closest to completion but it can be preempted by a newer ready job with shorter time to completion.
- Impossible to implement in interactive systems where required CPU time is not known.

17

| Process | A.T | B.T |
|---|---|---|
| P0 | 0 | 5 |
| $P_1$ | 1 | 3 |
| $P_2$ | 2 | 8 |
| $P_3$ | 3 | 6 |




$P_0$ w.t = 8-5-0 = 3
$P_1$ w.t = 4-3-1 = 0
$P_2$ w.t = 22-8-2 = 12

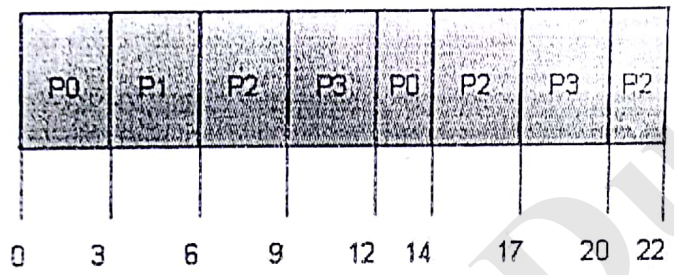
Avrg = $\frac{3+0+12+5}{4}$ = $\frac{20}{4}$ = 5

- It is often used in batch environments where short jobs need to give preference.

## Round Robin Scheduling

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

Quantum = 3




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

W.T = Tat - bT -

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

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

## Multiple-Level Queues Scheduling

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

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

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

## 11)about Inter Process Communication

A process can be of two type:

- Independent process.
- Co-operating process.

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes. Though one can think that those processes, which are running independently, will execute very efficiently but in practical, there are many situations when co-operative nature can be utilized for increasing computational speed, convenience and modularity. Inter process communication (IPC) is a mechanism which allows processes to communicate each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other using these two ways:

## 12) Inter Process Communication models:

1. Shared Memory
2. Message passing

The Figure 1 below shows a basic structure of communication between processes via shared memory method and via message passing.

An operating system can implement both method of communication. First, we will discuss the shared memory method of communication and then message passing. Communication between processes using shared memory requires processes to share some variable and it completely depends on how 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 other process, process1 generate information about certain computations or resources being used and keeps it as a record in shared memory.

When process2 need 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 other process as well as for delivering any specific information to other process.

Let's discuss an example of communication between processes using shared memory method.

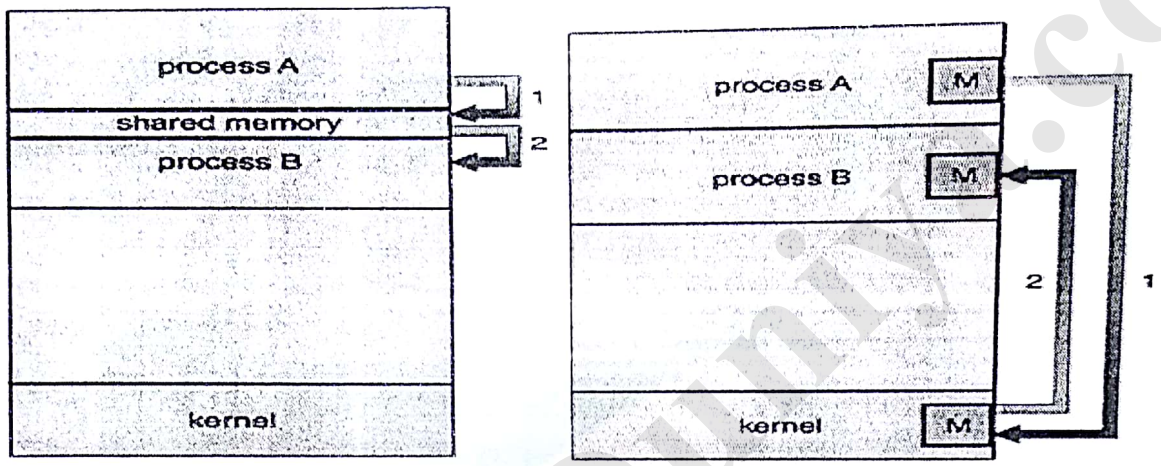



Figure 1: Shared Memory and Message Passing (Image Taken from "Operating System Cncepts" by Galvin et al.)

## Message-Passing Systems:

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and

is particularly useful in a distributed environment. A message-passing facility provides at least two operations: send(message) and receive(message). Messages sent by a process can be of either fixed or variable size.

- **Direct or indirect communication**

**Direct message passing**, The process which want to communicate must explicitly name the recipient or sender of communication.

e.g. **send(p1, message)** means send the message to p1.

similarly, **receive(p2, message)** means receive the message from p2.

In this method of communication, the communication link get established automatically, Symmetry and asymmetry between the sending and receiving can also be implemented i.e. either both process will name each other for sending and receiving the messages

or only sender will name receiver for sending the message and there is no need for receiver for naming the sender for receiving the message.

**Indirect message passing**, processes uses mailboxes (also referred to as ports) for sending and receiving messages. Each mailbox has a unique id and processes can communicate only if they share a mailbox. Link established only if processes share a common mailbox and a single link can be associated with many processes. The standard primitives used are : **send(A, message)** which means send the message to mailbox A. The primitive for the receiving the message also works in the same way e.g. **received (A, message)**.

**Synchronous and Asynchronous Message Passing:**

**Synchronization** Communication between processes takes place through calls to sendO and receive () primitives. There are different design options for implementing each primitive. Message passing may be either **blocking** or **nonblocking**— also known as **synchronous** and **asynchronous.**

- **Blocking send.** The sending process is blocked until the message is received by the receiving process or by the mailbox.
- **Nonblocking send.** The sending process sends the message and resumes operation.
- **Blocking receive.** The receiver blocks until a message is available.
- **Nonblocking receive.** The receiver retrieves either a valid message or a null.

Different combinations of send() and receive () are possible. When both sendQ and receive() are blocking, we have a **rendezvous** between the sender and the receiver. The solution to the producer-consumer problem becomes trivial when we use blocking sendO and r e c e i v e 0 statements. The producer merely invokes the blocking sendO call and waits until the message is delivered to either the receiver or the mailbox. Likewise, when the consumer invokes receive (), it blocks until a message is available.

**Buffering**

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

- **Zero capacity.** The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
- **Bounded capacity.** The queue has finite length *n;* thus, at most *n* messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue and the sender

can continue execution without waiting. The links capacity is finite, however. If the link is full, he sender must block until space is available in the queue.

- **Unbounded capacity.** The queues length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

### 13) STARVATION:

IT is a situation where a process does not get the resources it needs for a long time because the resources are being allocated to other processes. It generally occurs in a Priority based scheduling System. Where High Priority requests get processed first. Thus a request with least priority may never be processed. Priority scheduling leads to starvation.

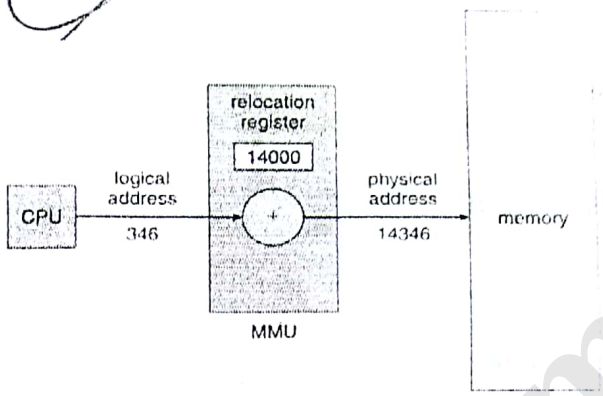
### 14) DIFFERENCE BETWEEN PREEMPTIVE AND NON-PREEMPTIVE SCHEDULING IN OS

| Preemptive Scheduling | Non-Preemptive Scheduling |
|---|---|
| Processor can be preempted to execute a different process in the middle of execution of any current process. | Once Processor starts to execute a process it must finish it before executing the other. It cannot be paused in middle. |
| CPU utilization is more compared to Non-Preemptive Scheduling. | CPU utilization is less compared to Preemptive Scheduling. |
| Waiting time and Response time is less. | Waiting time and Response time is more. |
| The preemptive scheduling is prioritized. The highest priority process should always be the process that is currently utilized. | When a process enters the state of running, the state of that process is not deleted from the scheduler until it finishes its service time. |
| If a high priority process frequently arrives in the ready queue, low priority process may starve. | If a process with long burst time is running CPU, then another process with less CPU burst time may starve. |
| Preemptive scheduling is flexible. | Non-preemptive scheduling is rigid. |
| Ex:- SRTF, Priority, Round Robin, etc. | Ex:- FCFS, SJF, Priority, etc. |

UNIT-III

Roll no:-16851-A052

1) **Memory management** is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution.




**Process logical Address Space vs physical address space:**

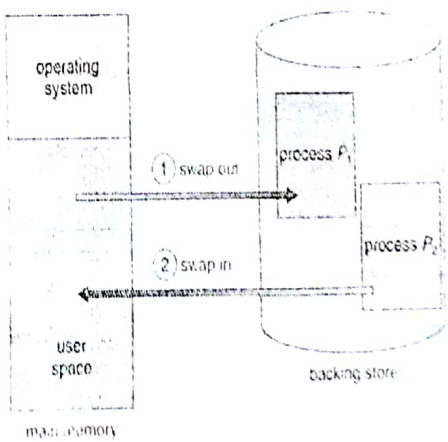
- The set of all logical addresses generated by the cpu is referred to as a **logical address space**. The set of all physical addresses corresponding to these logical addresses is referred to as a **physical address space.** The operating system takes care of mapping the logical addresses to physical addresses at the time of memory allocation to the program.
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes.
- Logical and physical addresses differ in execution-time address-binding scheme. The runtime mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device.
  - The value in the base register is added to every address generated by a user process, which is treated as offset at the time it is sent to memory. For example, if the base register value is 14000, then an attempt by the user to use address location 346 will be dynamically reallocated to location 14346.
  - The user program deals with virtual addresses; it never sees the real physical addresses.

2) Swapping: A process needs to be in memory for execution. But sometimes there is not enough main memory to hold all the currently active processes in a timesharing system. So, excess process are kept on disk and brought in to run dynamically. Swapping is the process of bringing in each process in main memory, running it for a while and then putting it back to the disk.

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

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

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




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

2048KB / 1024KB per second

= 2 seconds

= 2000 milliseconds

Now considering in and out time, it will take complete 4000 milliseconds plus other overhead where the process competes to regain main memory.

## 3) Contiguous and Noncontiguous Memory Allocation

1. The basic difference between contiguous and noncontiguous memory allocation is that contiguous allocation allocates **one single contiguous block of memory** to the process whereas, the noncontiguous allocation divides the **process into several blocks** and place them in the **different address space of the memory** i.e. in a noncontiguous manner.
2. In contiguous memory allocation, the process is stored in contiguous memory space; so there is **no overhead of address translation** during execution. But in noncontiguous memory allocation, there is **an overhead of address translation** while the process execution, as the process blocks are spread in the memory space.
3. Process stored in contiguous memory executes **faster** in comparison to process stored in noncontiguous memory space.
4. The solution for contiguous memory allocation is to **divide the memory space into the fixed-sized partition** and allocate a partition to a single process only. On the other hands, in non contiguous memory allocation, **a process is divided into several blocks** and each block is placed at **different places in memory** according to the availability of the memory.
5. In contiguous memory allocation, operating system has to maintain a **table** which indicates which partition is available for the process and which is occupied by the process. In noncontiguous memory allocation, a **table** is maintained for **each process** which indicates the base address of each block of the process placed in the memory space.

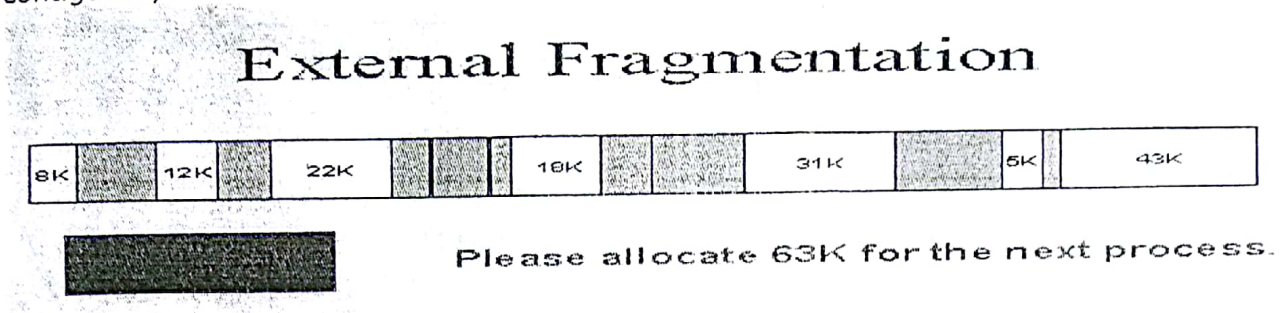
## 4) Fragmentation

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

Fragmentation is of two types:

**External fragmentation**

Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous, so it cannot be used.



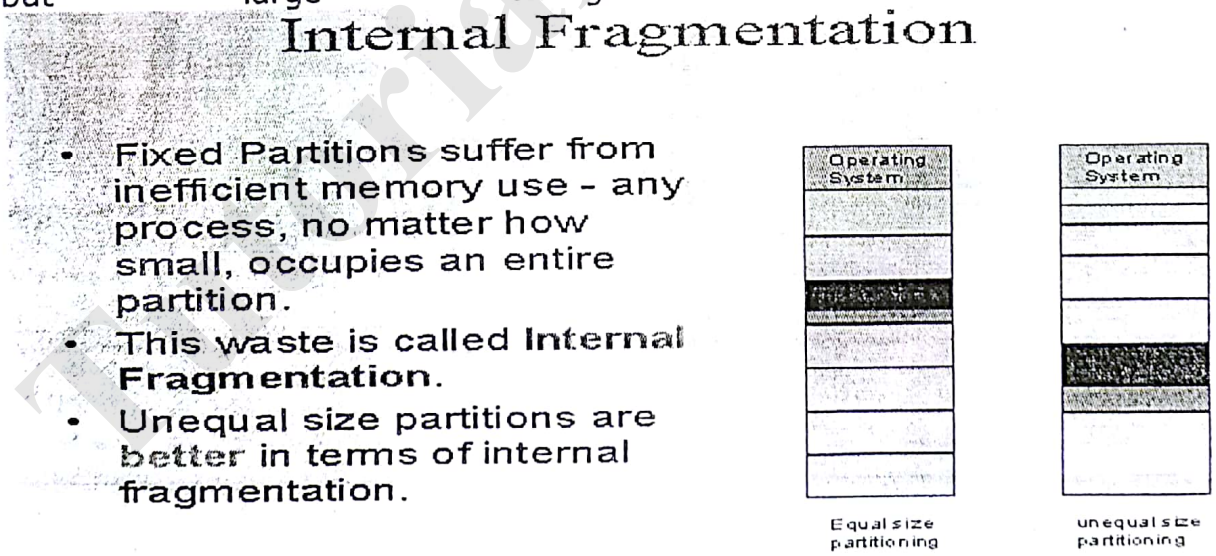

- Eventually, main memory forms holes too small to hold any process. This is external fragmentation.
- Total memory space may exist to satisfy a request but it is not contiguous.
- Compaction reduces external fragmentation by shuffling memory contents to place all free memory together in one large block.

**Internal fragmentation**

Memory block assigned to process is bigger. Some portion of memory is left unused, as it cannot be used by another process. The following diagram shows how fragmentation can cause waste of memory and a compaction technique can be used to create more free memory out of fragmented memory –

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

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




1. **First fit** - Search the list of holes until one is found that is big enough to satisfy the request, and assign a portion of that hole to that process. Whatever fraction of the hole not needed by the request is left on the free list as a smaller hole. Subsequent requests

may start looking either from the beginning of the list or from the point at which this search ended.

2. **Best fit** - Allocate the *smallest* hole that is big enough to satisfy the request. This saves large holes for other process requests that may need them later, but the resulting unused portions of holes may be too small to be of any use, and will therefore be wasted. Keeping the free list sorted can speed up the process of finding the right hole.

3. **Worst fit** - Allocate the largest hole available, thereby increasing the likelihood that the remaining portion will be usable for satisfying future requests.

## Segmentation:

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

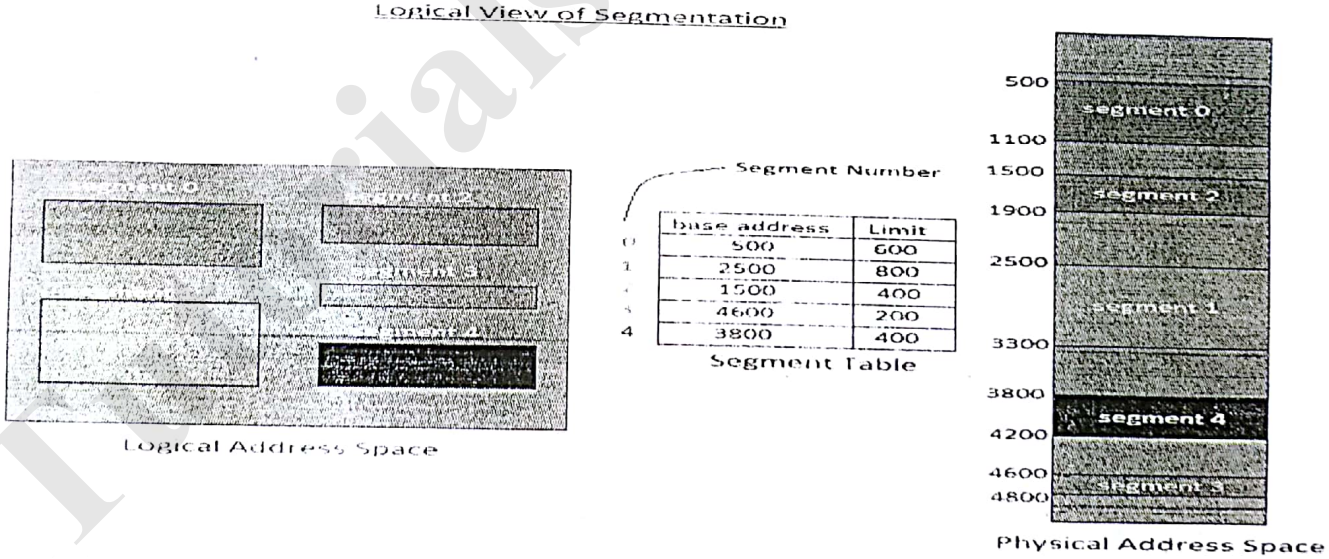
When a process is to be executed, its corresponding segmentation are loaded into non-contiguous memory though every segment is loaded into a contiguous block of available memory.

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

Memory Management technique in which memory is divided into variable sized chunks which can be allocated to processes. Each chunk is called a**Segment**.
A table stores the information about all such segments and is called **Segment Table.**

**Segment Table:** It maps two dimensional Logical address into one dimensional Physical address.
It's each table entry has

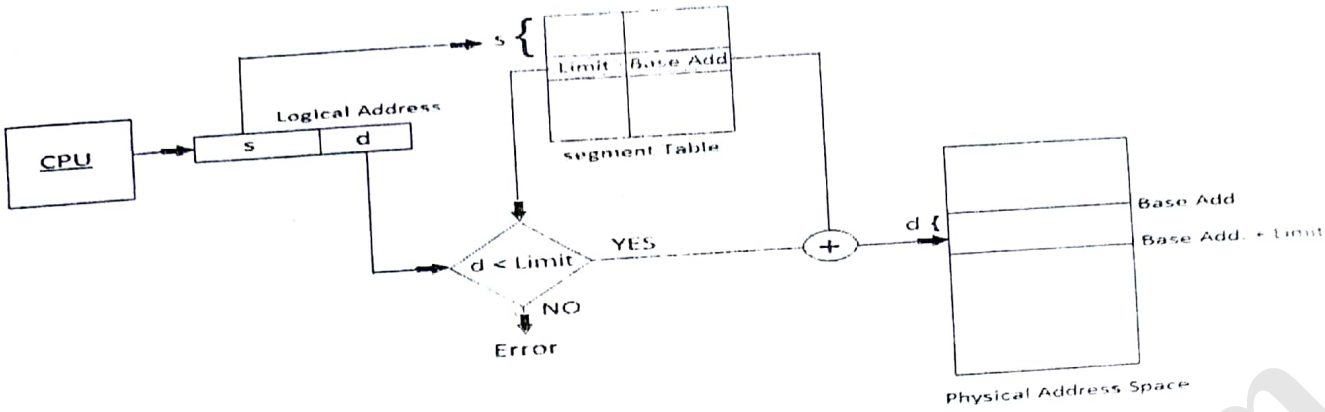
- **Base Address:** It contains the starting physical address where the segments reside in memory.
- **Limit:** It specifies the length of the segment.




**Translation of Two dimensional Logical Address to one dimensional Physical Address.**

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.




**Advantages of Segmentation:**

- No Internal fragmentation.
- Segment Table consumes less space in comparison to Page table in paging.

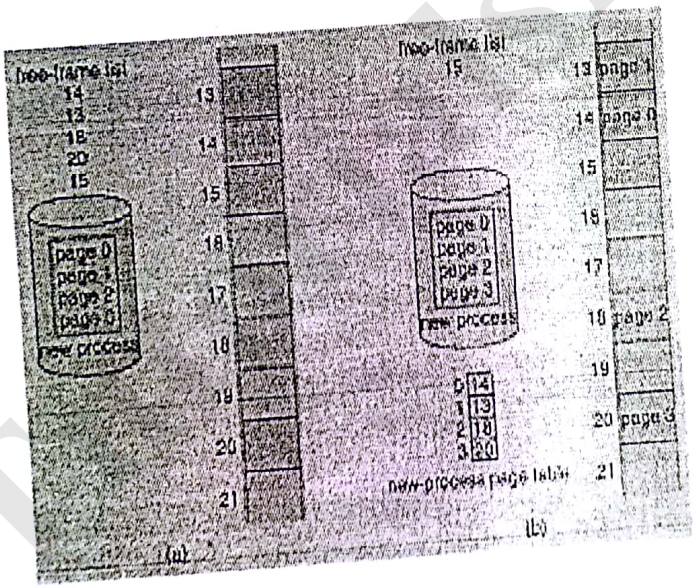
**Disadvantage of Segmentation:**

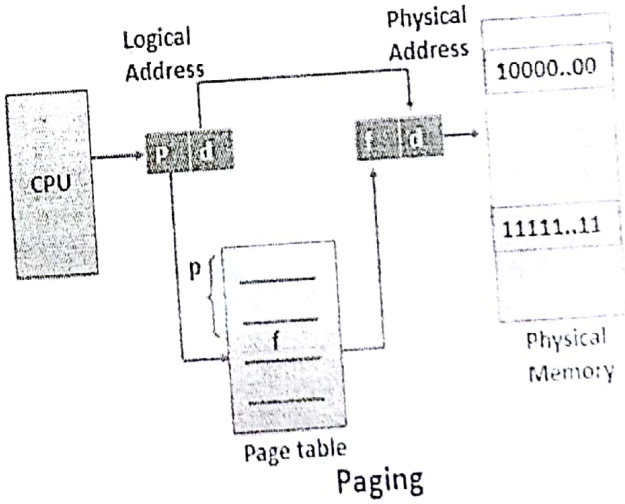
- As processes are loaded and removed from the memory, the free memory space is broken into little pieces, causing External fragmentation.

6) **Paging: "Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory."**

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

Paging is a memory management technique in which process address space is broken into blocks of the same size called **pages**. The size of the process is measured in the number of pages. Similarly, main memory is divided into small fixed-sized blocks of (physical) memory called **frames** and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.




Paging

Address Translation

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

Logical Address = Page number + page offset

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

Physical Address = Frame number + page offset

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

When the system allocates a frame to any page, it translates this logical address into a physical address and create entry into the page table to be used throughout execution of the program. When a process is to be executed, its corresponding pages are loaded into any available memory frames. Suppose you have a program of 8Kb but your memory can accommodate only 5Kb at a given point in time, then the paging concept will come into picture. When a computer runs out of RAM, the operating system (OS) will move idle or unwanted pages of memory to secondary memory to free up RAM for other processes and brings them back when needed by the program.

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

## Advantages and Disadvantages of Paging

Here is a list of advantages and disadvantages of paging –

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

## Virtual Memory:

"Virtual Memory is a storage allocation scheme in which secondary memory can be addressed as it were part of main memory."

This technique is useful as large virtual memory is provided for user programs when a very small physical memory is available say 4GB only.

**Benefits of having Virtual Memory :**

1. Large programs can be written, as virtual space available is huge compared to physical memory.
2. Less I/O required, leads to faster and easy swapping of processes.
3. More physical memory available, as programs are stored on virtual memory, so they occupy very less space on actual physical memory.
4. System libraries can be shared by mapping them into the virtual address space of more than one process.
5. Processes can also share virtual memory by mapping the same block of memory to more than one process.
6. Process pages can be shared during a fork( ) system call, eliminating the need to copy all of the pages of the original ( parent ) process.

# TutorialsDuniya.com

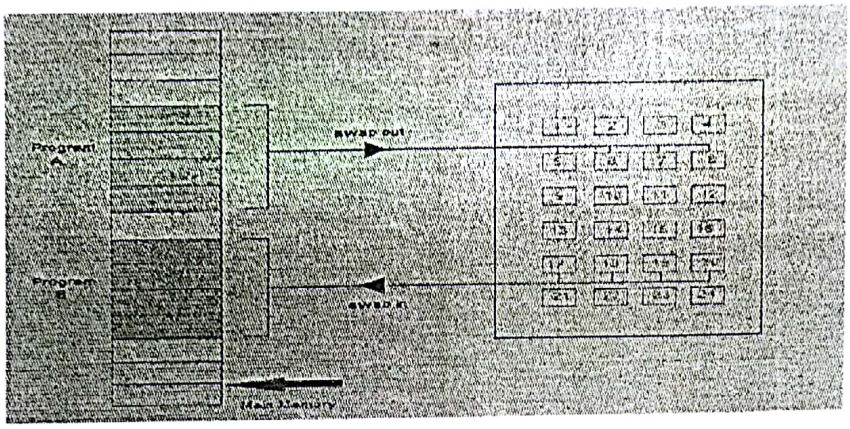
**Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com**

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

**Please Share these Notes with your Friends as well**

## Demand Paging:

The basic idea behind demand paging is that when a process is swapped in, its pages are not swapped in all at once. Rather they are swapped in only when the process needs them(On demand). This is termed as lazy swapper, although a pager is a more accurate term.



Initially only those pages are loaded which will be required the process immediately.

The pages that are not moved into the memory, are marked as invalid in the page table. For an invalid entry the rest of the table is empty. In case of pages that are loaded in the memory, they are marked as valid along with the information about where to find the swapped out page.

When the process requires any of the page that is not loaded into the memory, a page fault trap is triggered and following steps are followed,

1. The memory address which is requested by the process is first checked, to verify the request made by the process.
2. If its found to be invalid, the process is terminated.
3. In case the request by the process is valid, a free frame is located, possibly from a free-frame list, where the required page will be moved.
4. A new operation is scheduled to move the necessary page from disk to the specified memory location. ( This will usually block the process on an I/O wait, allowing some other process to use the CPU in the meantime. )
5. When the I/O operation is complete, the process's page table is updated with the new frame number, and the invalid bit is changed to valid.
6. The instruction that caused the page fault must now be restarted from the beginning.

There are cases when no pages are loaded into the memory initially, pages are only loaded when demanded by the process by generating page faults. This is called **Pure Demand Paging**.

The only major issue with Demand Paging is, after a new page is loaded, the process starts execution from the beginning. Its is not a big issue for small programs, but for larger programs it affects performance drastically.

## Page Replacement:

As studied in Demand Paging, only certain pages of a process are loaded initially into the memory. This allows us to get more number of processes into the memory at the same time. but what happens when a process requests for more pages and no free memory is available to bring them in. Following steps can be taken to deal with this problem :

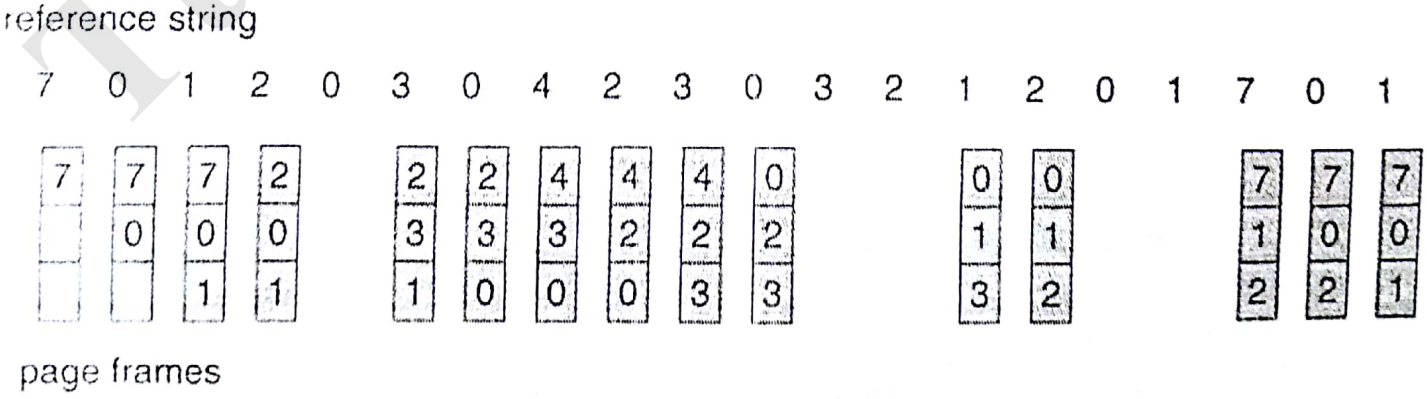
1. Put the process in the wait queue, until any other process finishes its execution thereby freeing frames.
2. Or, remove some other process completely from the memory to free frames.
3. Or, find some pages that are not being used right now, move them to the disk to get free frames. This technique is called **Page replacement** and is most commonly used. We have some great algorithms to carry on page replacement efficiently.

### *Basic Page Replacement*

- Find the location of the page requested by ongoing process on the disk.
- Find a free frame. If there is a free frame, use it. If there is no free frame, use a page-replacement algorithm to select any existing frame to be replaced, such frame is known as **victim frame.**
- Write the victim frame to disk. Change all related page tables to indicate that this page is no longer in memory.
- Move the required page and store it in the frame. Adjust all related page and frame tables to indicate the change.
- Restart the process that was waiting for this page.

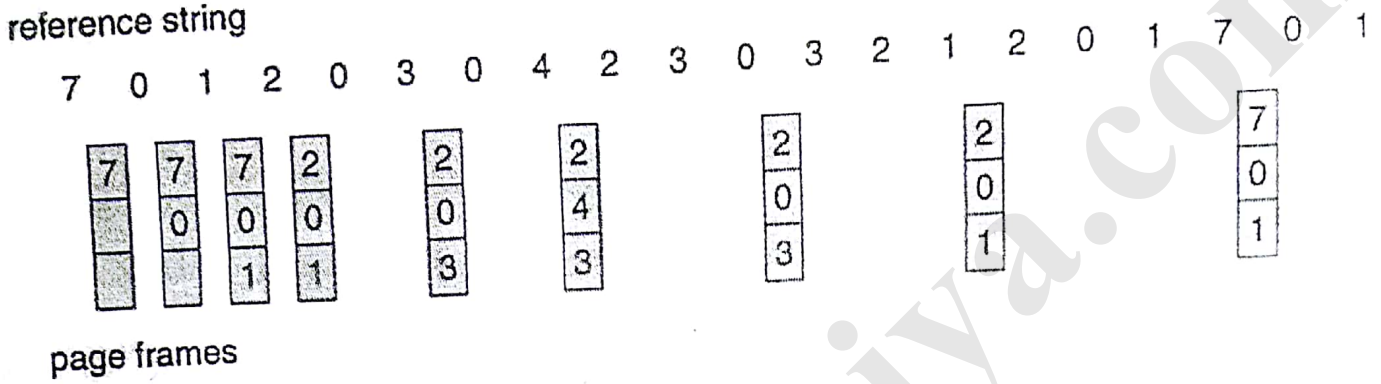
### FIFO Page Replacement

- A very simple way of Page replacement is FIFO (First in First Out)
- As new pages are requested and are swapped in, they are added to tail of a queue and the page which is at the head becomes the victim.
- Its not an effective way of page replacement but can be used for small systems.



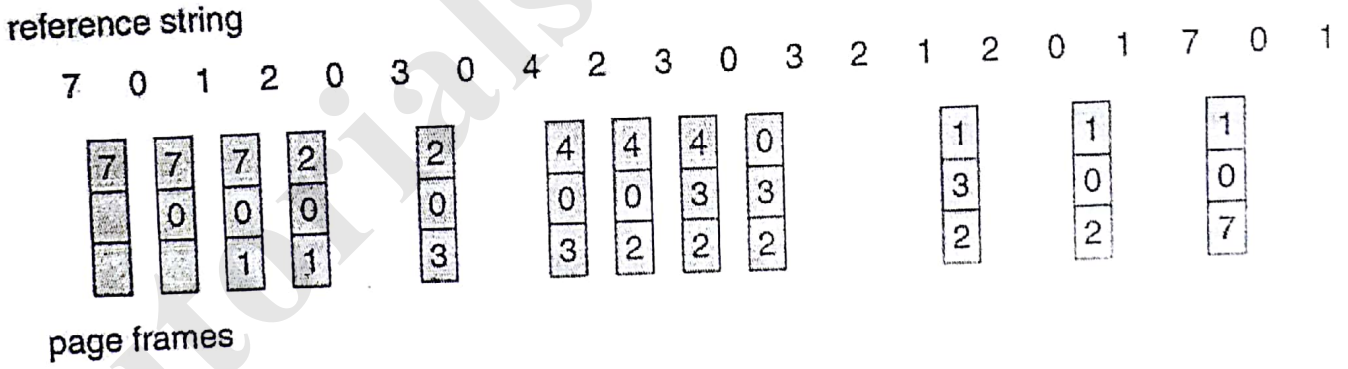
## (ii) Optimal Page algorithm

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




## (iii) Least Recently Used (LRU) algorithm

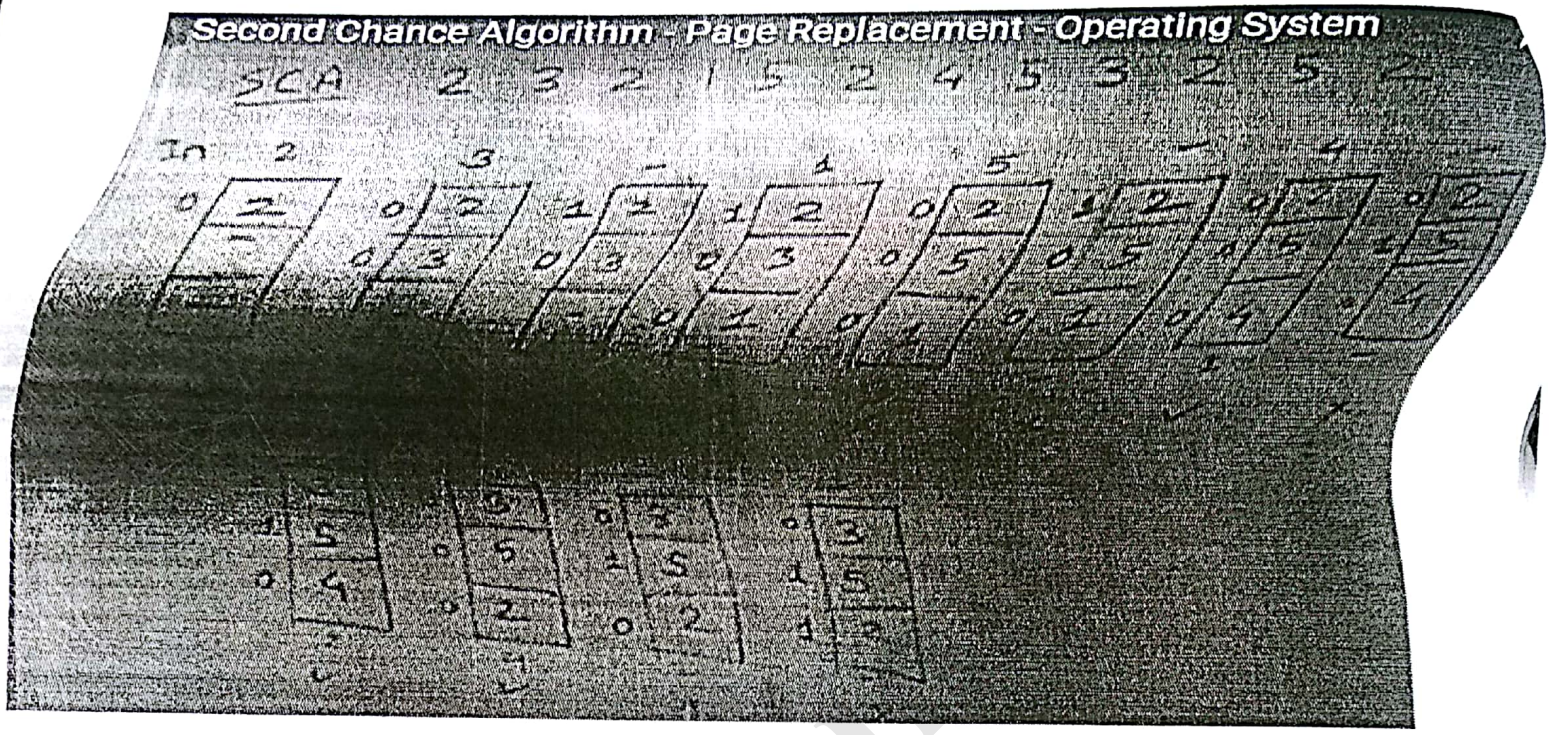
- "Replace the Page which has not been used for the longest time in main memory is the one which will be selected for replacement."
- Easy to implement, keep a list, replace pages by looking back into time.




(iv) **LRU-Approximation Page Replacement:**

- The second chance algorithm is essentially a FIFO, except the reference bit is used to give pages a second chance at staying in the page table.
- When a page must be replaced, the page table is scanned in a FIFO ( circular queue ) manner.
- If a page is found with its reference bit not set, then that page is selected as the next victim.
- If, however, the next page in the FIFO does have its reference bit set, then it is given a second chance:
- The reference bit is cleared, and the FIFO search continues.
- If some other page is found that did not have its reference bit set, then that page will be selected as the victim, and this page ( the one being given the second chance ) will be allowed to stay in the page table.
- If , however, there are no other pages that do not have their reference bit set, then this page will be selected as the victim when the FIFO search circles back around to this page on the second pass.

- If all reference bits in the table are set, then second chance degrades to FIFO, but als complete search of the table for every page-replacement.
- As long as there are some pages whose reference bits are not set, then any page referenced t enough gets to stay in the page table indefinitely.



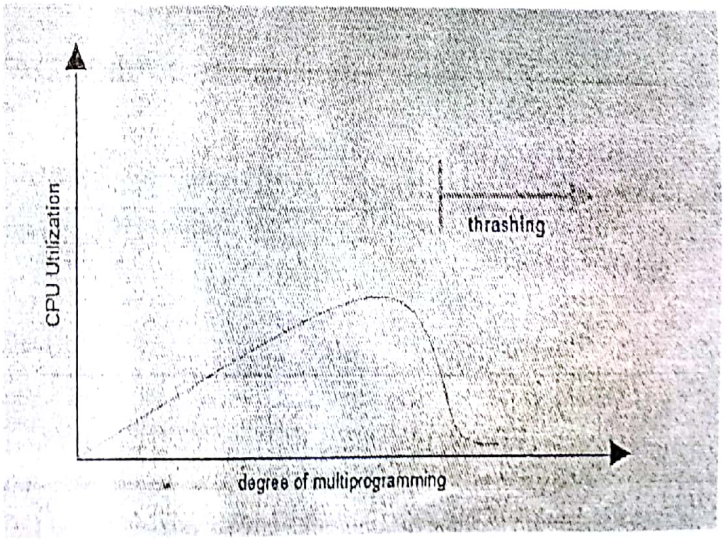

## *Thrashing:*

A process that is spending more time paging than executing is said to be thrashing. In other words it means, that the process doesn't have enough frames to hold all the pages for its execution, so it is swapping pages in and out very frequently to keep executing. Sometimes, the pages which will be required in the near future have to be swapped out.

Initially when the CPU utilization is low, the process scheduling mechanism, to increase the level of multiprogramming loads multiple processes into the memory at the same time, allocating a limited amount of frames to each process. As the memory fills up, process starts to spend a lot of time for the required pages to be swapped in, again leading to low CPU utilization because most of the processes are waiting for pages. Hence the scheduler loads more processes to increase CPU utilization, as this continues at a point of time the complete system comes to a stop.

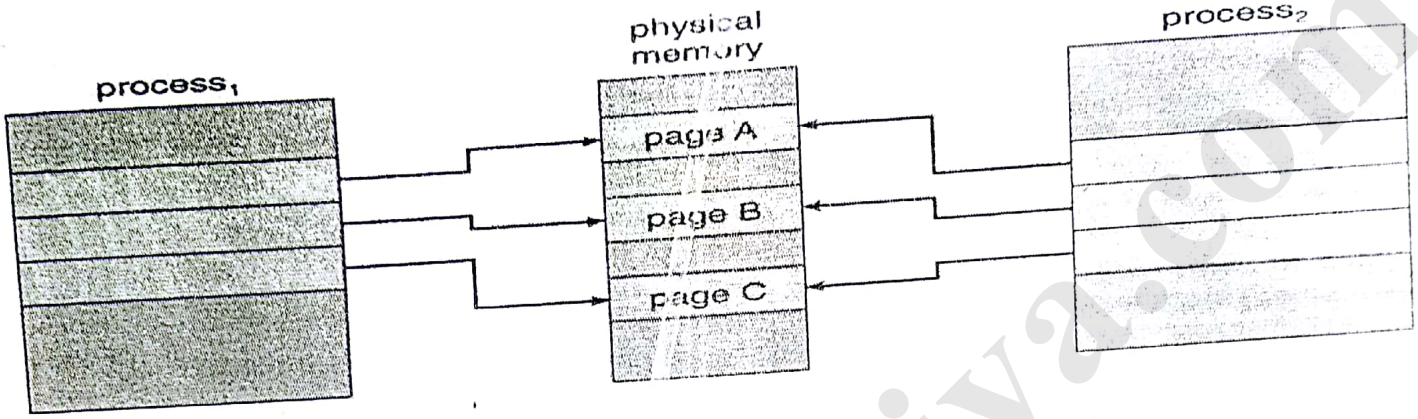
Causes of Thrashing :

1. **High degree of multiprogramming** : If the number of processes keeps on increasing in the memory than number of frames allocated to each process will be decreased. So, less number of frames will be available to each process. Due to this, page fault will occur more frequently and more CPU time will be wasted in just swapping in and out of pages and the utilization will keep on decreasing.
2. **Lacks of Frames**: If a process has less number of frames then less pages of that process will be able to reside in memory and hence more frequent swapping in and out will be required.



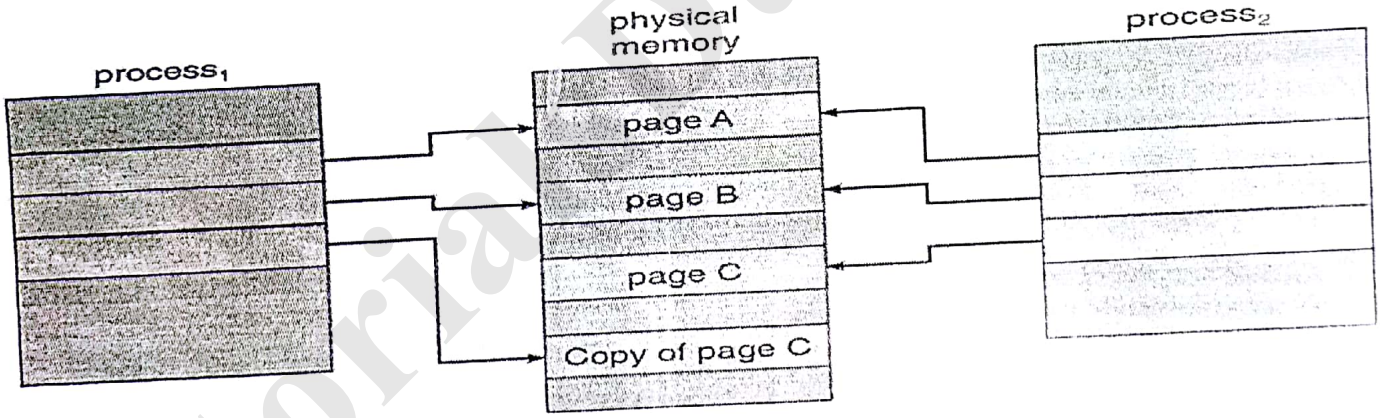
**Copy-on-Write**

- The idea behind a copy-on-write fork is that the pages for a parent process do not have to be actually copied for the child until one or the other of the processes changes the page. They can be simply shared between the two processes in the meantime, with a bit set that the page needs to be copied if it ever gets written to. This is a reasonable approach, since the child process usually issues an exec( ) system call immediately after the fork.




Before process 1 modifies page C.

- Obviously only pages that can be modified even need to be labeled as copy-on-write. Code segments can simply be shared.




After process 1 modifies page C.

# Comparison of Paging and Segmentation

| Paging | Segmentation |
|---|---|
| Main memory is partitioned into frames or blocks | Main memory is partationed into segments |
| The logical address space is divided into pages by MMU | The logical address space is divided into segments as specified by the program |
| The scheme suffers from internal fragmentation or page breaks | The scheme suffers from External fragmentation |
| OS maintains a free frame list , so that searching of free frame is not necessary | OS maintains the particulars of available memory |
| OS maintains a page map table for mapping between frames and pages | OS maintains a segment map table for mapping |
| This scheme does not support the users view of memory | This scheme support the users view of memory |
| Processor use the page number and displacement to calculate absolute address (p, d) | Processor use the segment number and displacement to calculate the absolute address (p, d) |

# UNIT-IV

## 1.Synchronization

A cooperating process is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages.

Concurrent access to shared data may result in data inconsistency. Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes. A Shared-memory solution to bounded-butter problem allows at most $n - 1$ items in buffer at the same time. A solution, where all $N$ buffers are used is not simple. Suppose that we modify the producer-consumer code by adding a variable *counter*, initialized to 0 and incremented each time a new item is added to the buffer.

The code for the producer process can be modified as follows:

```
while (true) {
    /* produce an item in next produced */
    while (counter == BUFFER SIZE)
        ; /* do nothing */
    buffer[in] = next produced;
    in = (in + 1) % BUFFER SIZE;
    counter++;
}
```

The code for the consumer process can be modified as follows:

```
while (true) {
    while (counter == 0)
        ; /* do nothing */
    next consumed = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    counter--;
    /* consume the item in next consumed */
}
```

Although the producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently.

Note that the statement "counter++" may be implemented in machine language as follows:

*register*1 = counter

*register*1 = *register*1 + 1

counter = *register*1

where *register*1 is one of the local CPU registers. Similarly, the statement "counter--" is implemented as follows:

*register*2 = counter

*register*2 = *register*2 – 1

counter = *register*2

If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved. Interleaving depends upon how the producer and consumer processes are scheduled.

Assume **counter** is initially 5. One interleaving of statements is:

producer: **register1 = counter** (*register1 = 5*)
producer: **register1 = register1 + 1** (*register1 = 6*)
consumer: **register2 = counter** (*register2 = 5*)
consumer: **register2 = register2 – 1** (*register2 = 4*)
producer: **counter = register1** (*counter = 6*)
consumer: **counter = register2** (*counter = 4*)

The value of **count** may be either 4 or 6, where the correct result should be 5.

**Race condition**: The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.

To prevent race conditions, concurrent processes must be **synchronized**.

## 2. The Critical-Section Problem

Consider a system consisting of *n* processes {*P*0, *P*1, . . . , *Pn*−1}. Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time.

The *critical-section problem* is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this

request is the [illegible]. The critical section may be followed by an exit section. The remaining code is the [illegible].

The general structure of a typical process *Pi* is

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

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

1. **Mutual exclusion.** If process *Pi* is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress.** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. **Bounded waiting.** There exists a bound, or limit, 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.

Two general approaches are used to handle critical sections in operating systems: preemptive kernels and nonpreemptive kernels. A preemptive kernel allows a process to be preempted while it is running in kernel mode. A nonpreemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

## 3. Synchronization Hardware

Generally any solution to the critical section problem requires lock.

```
do {
        acquire lock
            critical section
        release lock
            reminder section
} while (1);
```

Race conditions are prevented.

The critical-section problem could be solved simply in a single-processor environment if we could prevent interrupts from occurring while a shared variable was being modified. In this way, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption. Unfortunately, this solution is not as feasible in a multi processor environment. Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases

Two types of instructions are 1.test and set() 2. compare and swap()

The definition of the test and set() instruction

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

Mutual-exclusion implementation with test and set().

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

The definition of the compare and swap() instruction.

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

Mutual-exclusion implementation with the compare and swap() instruction.

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
} while (true);
```

## 4.Semaphores

A S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal().

The wait() operation was originally termed P , signal() was originally called V .

The definition of wait() is as follows:

```
wait(S) {
while (S <= 0)
; // busy wait
S--;
}
```

The definition of signal() is as follows:

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

**Two Types of Semaphores**

1. Counting semaphore – integer value can range over an unrestricted domain.
2. Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement.

   Counting semaphores is initialized to the number of resources available. Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal() operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

**Use semaphores to solve various synchronization problems**. For example, consider two concurrently running processes: *P*1 with a statement *S*1 and *P*2 with a statement *S*2. Suppose we require that *S*2 be executed only after *S*1 has completed. We can implement this scheme readily by letting *P*1 and *P*2 share a common semaphore synch, initialized to 0.

In process *P*1, we insert the statements

```
S1;
signal(synch);
```

In process *P*2, we insert the statements

```
wait(synch);
S2;
```

Because synch is initialized to 0, *P*2 will execute *S*2 only after *P*1 has invoked signal(synch), which is after statement *S*1 has been executed.

we define a semaphore as follows:

```
typedef struct {
int value;
struct process *list;
} semaphore;
```

Each semaphore has an integer value and a list of processes list. When a process must wait on a semaphore, it is added to the list of processes. A signal() operation removes one process from the list of waiting processes and awakens that process.

Now, the wait() semaphore operation can be defined as

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

signal() semaphore operation can be defined as

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

The block() operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.

**Critical Section of n Processes**

Shared data:

```
semaphore mutex; //initially mutex = 1
```

Process Pi:

```
do {
    wait(mutex);
     critical section

    signal(mutex);
    remainder section
} while (1);
```

## Deadlocks and Starvation

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

Let S and Q be two semaphores initialized to 1

| $P_0$ | $P_1$ |
|---|---|
| wait(S); | wait(Q); |
| wait(Q); | wait(S); |
| . | . |
| . | . |
| . | . |
| signal(S); | signal(Q); |
| signal(Q); | signal(S); |

Suppose that $P0$ executes wait(S) and then $P1$ executes wait(Q). When $P0$ executes wait(Q), it must wait until $P1$ executes signal(Q). Similarly, when $P1$ executes wait(S), it must wait until $P0$ executes signal(S). Since these signal() operations cannot be executed, $P0$ and $P1$ are deadlocked.

**Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

## Priority Inversion

A scheduling challenge arises when a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process—or a chain of lower-priority processes. Since kernel data are typically protected with a lock, the higher-priority process will have to wait for a lower-priority one to finish with the resource. The situation becomes more complicated if the lower-priority process is preempted in favor of another process with a higher priority.

An example, assume we have three processes—L, M, and H—whose priorities follow the order L < M < H. Assume that process H requires resource R, which is currently being accessed by process L. Ordinarily, process H would wait for L to finish using resource R.

However, now suppose that process M becomes runnable, thereby preempting process L. Indirectly, a process with a lower priority—processM—has affected how long process H must wait for L to relinquish resource R.

This problem is known as **priority inversion.** It occurs only in systems with more than two priorities. Typically these systems solve the problem by implementing a **priority-inheritance protocol.** According to this protocol, all processes that are accessing resources needed by a higher-

priority process inherit the higher priority until they are finished with the resources in questi When they are finished, their priorities revert to their original values.

In the example above, a priority-inheritance protocol would allow process L to temporarily inherit the priority of process H, thereby preventing process M from preempting its execution. When process L had finished using resource R, it would relinquish its inherited priority from H and assume its original priority. Because resource R would now be available, process H—not M—would run next.

## 5.Classic Problems of Synchronization

### 1.The Bounded-Buffer Problem

The producer and consumer-processes share the following data structures:

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

We assume that the pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

The structure of the producer process

```
do {
   . . .
   /* produce an item in next_produced */
   . . .
   wait(empty);
   wait(mutex);
   . . .
   /* add next_produced to the buffer */
   . . .
   signal(mutex);
   signal(full);
} while (true);
```

The structure of the consumer process.

```
do {
   wait(full);
   wait(mutex);
   . . .
   /* remove an item from buffer to next_consumed */
   . . .
   signal(mutex);
   signal(empty);
   . . .
   /* consume the item in next_consumed */
   . . .
} while (true);
```

# TutorialsDuniya.com

**Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com**

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

**Please Share these Notes with your Friends as well**

## 2.The Readers–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***. Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the readers–writers problem.

*The first* readers–writers problem, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting.

The *second* readers– writers problem requires that, once a writer is ready, that writer perform its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve.

In the solution to the first readers–writers problem, the reader processes share the following data structures:

```
Semaphore rwmutex = 1;
semaphore mutex = 1;
int read count = 0;
```

The semaphores mutex and rwmutex are initialized to 1; read count is initialized to 0. The semaphore rwmutex is common to both reader and writer processes. The mutex semaphore is used to ensure mutual exclusion when the variable read count is updated. The read count variable keeps track of how many processes are currently reading the object. The semaphore rw mutex functions as a mutual exclusion semaphore for the writers.

The code for a writer process is

```
do {
   wait(rw_mutex);
   . . .
   /* writing is performed */
   . . .
   signal(rw_mutex);
} while (true);
```

the code for a reader process is

```
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```

### 3.The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks

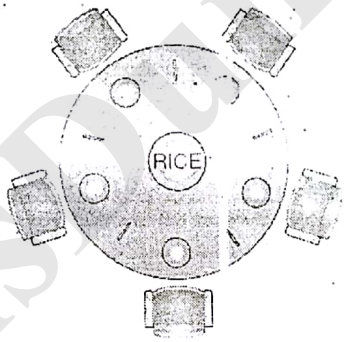



**Figure :**The situation of the dining philosophers.

When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbours). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again

The dining-philosophers problem is considered a classic synchronization Problem It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore. She releases her chopsticks by executing the signal() operation on the appropriate semaphores. Thus, the shared data are

semaphore chopstick[5];

The structure of philosopher *i* is ,

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for awhile */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
} while (true);
```

Although this solution guarantees that no two neighbours are eating simultaneously, it nevertheless must be rejected because it could create a deadlock. Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Several possible remedies to the deadlock problem are replaced by:

Allow at most four philosophers to be sitting simultaneously at the table.

- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even numbered philosopher picks up her right chopstick and then her left chopstick.

## 6. Monitors

Some of the problems of semaphore are

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

```
signal(mutex);
   ...
critical section
   ...
wait(mutex)
```

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

- Suppose that a process replaces signal(mutex) with wait(mutex). That is, it executes

```
wait(mutex);
...
critical section
...
wait(mutex);
```

In this case, a deadlock will occur.

- Suppose that a process omits the wait(mutex), or the signal(mutex), or both. In this case, either mutual exclusion is violated or a deadlock will occur

**monitor**-High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes

```
monitor monitor_name
{
    /* shared variable declarations */
    function P1 ( . . . ) {
    }
    function P2 ( . . . ) {
    }
    function Pn ( . . . ) {
    }
    initialization_code ( . . . ) {
    }
}
```

To allow a process to wait within the monitor, a **condition**-variable must be declared, as

**condition x, y;**

Condition variable can only be used with the operations **wait** and **signal**.

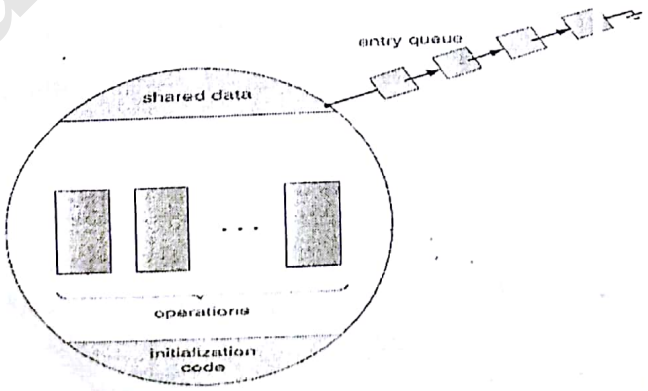
The operation **x.wait();**

means that the process invoking this operation is suspended until another process invokes

**x.signal();**

The **x.signal** operation resumes exactly one suspended process. If no process is suspended, then the **signal** operation has no effect.

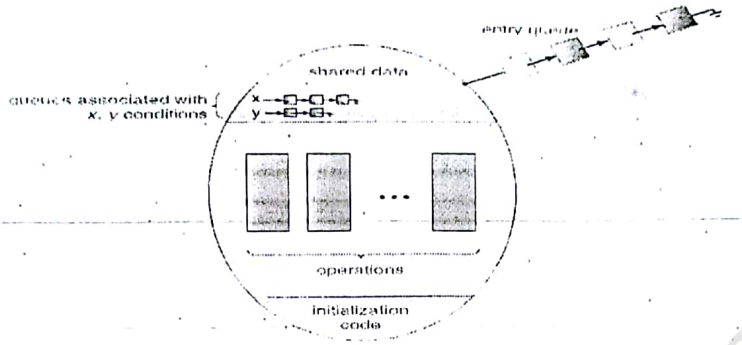
**Schematic view of a monitor.**



A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type *condition*:

condition x, y;

Monitor with condition variables




## Dining-Philosophers Solution Using Monitors

```
monitor dp
        {
                enum {thinking, hungry, eating} state[5];
                condition self[5];
                void pickup(int i)              // following slides
                void putdown(int i)     // following slides
                void test(int i)                // following slides
                void init() {
                        for (int i = 0; i < 5; i++)
                                state[i] = thinking;
                }
        }
void pickup(int i) {
                state[i] = hungry;
                test[i];
                if (state[i] != eating)
                        self[i].wait();
        }
        void putdown(int i) {
                state[i] = thinking;
                // test left and right neighbors
                test((i+4) % 5);
                test((i+1) % 5);
        }
void test(int i) {
                if ( (state[(I + 4) % 5] != eating) &&
                 (state[i] == hungry) &&
```

```
        (state[(i + 1) % 5] != eating)) {
            state[i] = eating;
            self[i].signal();
        }
    }
```

**Monitor Implementation Using Semaphores**

- Variables

```
semaphore mutex;  // (initially = 1)
semaphore next;   // (initially = 0)
int next-count = 0;
```

- Each external procedure *F* will be replaced by

```
wait(mutex);
  ...
  body of F;
  ...
if (next-count > 0)
        signal(next)
else
        signal(mutex);
```

- Mutual exclusion within a monitor is ensured.
- For each condition variable ***x***, we have:

```
semaphore x-sem; // (initially = 0)
int x-count = 0;
```

- The operation **x.wait** can be implemented as:

```
x-count++;
if (next-count > 0)
        signal(next);
else
        signal(mutex);
wait(x-sem);
x-count--;
```

- The operation x.signal can be implemented as:

```
if (x-count > 0) {
        next-count++;
        signal(x-sem);
        wait(next);
        next-count--;
}
```

## Deadlocks

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a **deadlock**.

A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

**Example :**

System has 2 tape drives.

$P_1$ and $P_2$ each hold one tape drive and each needs another one.

---

**Example**

semaphores *A* and *B*, initialized to 1

| $P_0$ | $P_1$ |
|---|---|
| *wait (A);* | *wait(B)* |
| *wait (B);* | *wait(A)* |

## 1.System Model

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources may be partitioned into several types each consisting of some number of identical instances. CPU cycles, files, and I/O devices (such as printers and DVD drives) are examples of resource types.

A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system.

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

1. **Request**. The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.

2. **Use**. The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).

3. **Release**. The process releases the resource.

## 2. Deadlock Characterization

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

### 2.1 Necessary Conditions

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

1. **Mutual exclusion**. At least one resource must be held in a non sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. **Hold and wait**. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

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

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

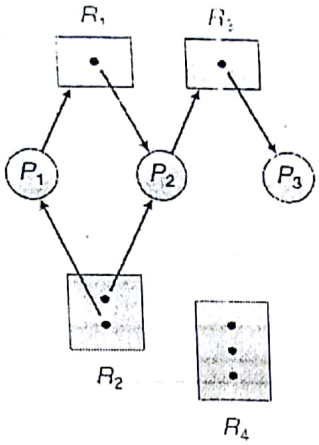
### 2.2 Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation graph**. This graph consists of a set of vertices $V$ and a set of edges $E$. The set of vertices $V$ is partitioned into two different types of nodes: $P = \{P1, P2, .... Pn\}$, the set consisting of all the active processes in the system, and $R = \{R1, R2, ..., Rm\}$, the set consisting of all resource types in the system.

A directed edge from process $Pi$ to resource type $Rj$ is denoted by $Pi \rightarrow Rj$; it signifies that process $Pi$ has requested an instance of resource type $Rj$ .A directed edge from resource type $Rj$ to Process $Pi$ is denoted by $Rj \rightarrow Pi$; it signifies that an instance of resource type $Rj$ has been allocated to process $Pi$.

A directed edge $Pi \rightarrow Rj$ is called a **request edge**; a directed edge $Rj \rightarrow Pi$ is called an **assignment edge**. Pictorially, we represent each process $Pi$ as a circle and each resource type $Rj$ as a rectangle. Since resource type $Rj$ may have more than one instance, we represent each such instance as a dot within the rectangle.

The resource-allocation graph example is




If the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

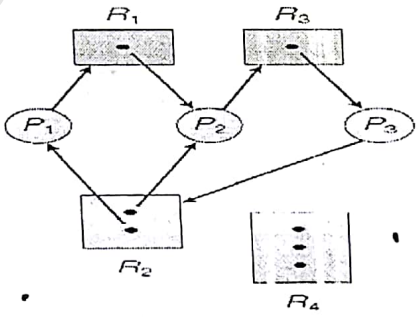
**Resource-allocation graph with a deadlock**

Suppose that process $P3$ requests an instance of resource type $R2$. Since no resource instance is currently available, we add a request edge $P3 \rightarrow R2$ to the graph . At this point, two minimal cycles exist in the system:

$P1 \rightarrow R1 \rightarrow P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P1$

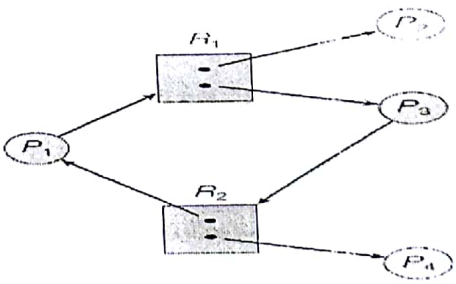
$P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P2$

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




**Resource-allocation graph with a cycle but no deadlock**

$P1 \rightarrow R1 \rightarrow P3 \rightarrow R2 \rightarrow P1$



## 3.Methods for Handling Deadlocks

We can deal with the deadlock problem in one of three ways:

- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will *never* enter a deadlocked state.
- We can allow the system to enter a deadlocked state, detect it, and recover.
- We can ignore the problem altogether and pretend that deadlocks never occur in the system.

## 4.Deadlock Prevention

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

### 1.Mutual Exclusion

The mutual exclusion condition must hold. That is, at least one resource must be **Non-sharable**. Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource. If several processes attempt to open a read only file at the same time, they can be granted simultaneous access to the file. however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.

### 2.Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.

One protocol that we can use requires each process to request and be allocated all its resources before it begins execution.

An alternative protocol allows a process to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

Both these protocols have two main disadvantages. First, resource utilization may be low, since resources may be allocated but unused for a long period.

Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

### 3.No Preemption

The third necessary condition for deadlocks is that there be no pre-emption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following

protocol. If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted.

Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process. If the resources are neither available nor held by a waiting process, the requesting process must wait.

**4.Circular Wait**

Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

To illustrate, we let consider $R=\{R1, R2, \ldots, Rm\}$ be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering.

For example, if the set of resource types $R$ includes tape drives, disk drives, and printers, then the function $F$ might be defined as follows:

$F$(tape drive) = 1

$F$(disk drive) = 5

$F$(printer) = 12

We can now consider the following protocol to prevent deadlocks: Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type—say, $Ri$. After that, the process can request instances of resource type $Rj$ if and only if $F(Rj) > F(Ri)$.

A process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer

## 5. Deadlock Avoidance

Require additional information about how resources are to be requested. Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.

Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.

## 5.1 Safe State

When a process requests an available resource. system must decide if immediate allocation leaves the system in a *safe state*.

A System is in safe state if there exists a safe sequence of all processes.

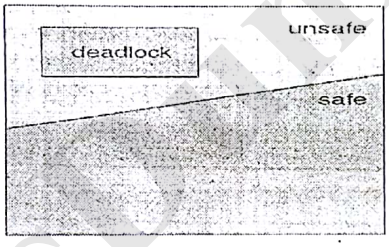
Sequence $<P_1, P_2, \ldots, P_n>$ is safe if for each $P_i$, the resources that $Pi$ can still request can be satisfied by currently available resources + resources held by all the $P_j$, with $j<I$.

If $P_i$ resource needs are not immediately available, then $P_i$ can wait until all $P_j$ have finished.

When $P_j$ is finished, $P_i$ can obtain needed resources, execute, return allocated resources, and terminate.

When $P_i$ terminates, $P_{i+1}$ can obtain its needed resources, and so on.

- If a system is in safe state ⇒ no deadlocks.
- If a system is in unsafe state ⇒ possibility of deadlock.
- Avoidance ⇒ ensure that a system will never enter an unsafe state.




**Example:**

We consider a system with twelve magnetic tape drives and three processes: *P*0, *P*1, and *P*2. Process *P*0 requires ten tape drives, process *P*1 may need as many as four tape drives, and process *P*2 may need up to nine tape drives.

Suppose that, at time *t*0, process *P*0 is holding five tape drives, process *P*1 is holding two tape drives, and process *P*2 is holding two tape drives. (Thus, there are three free tape drives.)

| | Maximum Needs | Current Needs |
|---|---|---|
| *P*0 | 10 | 5 |
| *P*1 | 4 | 2 |
| *P*2 | 9 | 2 |

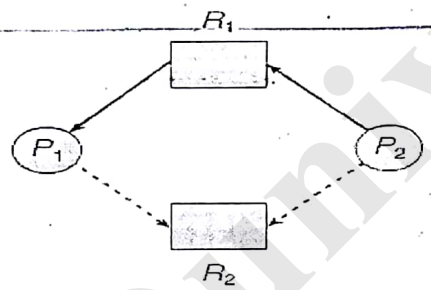
At time *t*0, the system is in a safe state. The sequence <*P*1, *P*0, *P*2> satisfies the safety condition.

## 5.2 Resource-Allocation-Graph Algorithm

A claim edge $Pi \rightarrow Rj$ indicates that process $Pi$ may request resource $Rj$ at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line. When process $Pi$ requests resource $Rj$, the claim edge $Pi \rightarrow Rj$ is converted to a request edge. Similarly, when a resource $Rj$ is released by $Pi$, the assignment edge $Rj \rightarrow Pi$ is reconverted to a claim edge $Pi \rightarrow Rj$.

The resources must be claimed a priori in the system. That is, before process $Pi$ starts executing, all its claim edges must already appear in the resource-allocation graph.

**Resource-allocation graph for deadlock avoidance**




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

If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. In that case, process $Pi$ will have to wait for its requests to be satisfied.

Suppose that $P2$ requests $R2$.Although$R2$ is currently free, we cannot allocate it to $P2$, since this action will create a cycle in the graph. If $P1$ requests $R2$, and $P2$ requests $R1$, then a deadlock will occur.

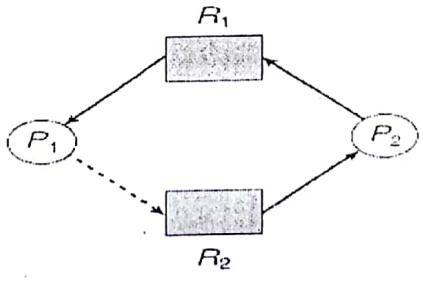



Fig: An unsafe state in a resource-allocation graph.

## 5.3 Banker's Algorithm

The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type.

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. We need the following data structures, where $n$ is the number of processes in the system and $m$ is the number of resource types:

- **Available**. A vector of length $m$ indicates the number of available resources of each type. If *Available*[$j$] equals $k$, then $k$ instances of resource type $Rj$ are available.
- **Max**. An $n \times m$ matrix defines the maximum demand of each process. If *Max*[$i$][$j$] equals $k$, then process $Pi$ may request at most $k$ instances of resource type $Rj$.
- **Allocation**. An $n \times m$ matrix defines the number of resources of each type currently allocated to each process. If *Allocation*[$i$][$j$] equals $k$, then process $Pi$ is currently allocated $k$ instances of resource type $Rj$.
- **Need**. An $n \times m$ matrix indicates the remaining resource need of each process. If *Need*[$i$][$j$] equals $k$, then process $Pi$ may need $k$ more instances of resource type $Rj$ to complete its task. Note that *Need*[$i$][$j$] equals *Max*[$i$][$j$]− *Allocation*[$i$][$j$].

### Safety Algorithm

For finding out whether or not a system is in a safe state.

1. Let ***Work*** and ***Finish*** be vectors of length $m$ and $n$, respectively. Initialize

   ***Work=Available*** and ***Finish***[$i$]=***false*** for $i$=0, 1, ..., $n$ − 1.

2. Find an index $i$ such that both

   a. ***Finish***[$i$] == ***false***

   b. ***Needi*** ≤ ***Work***

   If no such $i$ exists, go to step 4.

3. ***Work=Work+Allocationi***

   ***Finish***[$i$]=***true***

   Go to step 2.

4. If ***Finish***[$i$]==***true*** for all $i$, then the system is in a safe state.

### Resource-Request Algorithm

The algorithm for determining whether requests can be safely granted.

Let ***Requesti*** be the request vector for process $Pi$. If ***Requesti*** [$j$]==$k$, then process $Pi$ wants $k$ instances of resource type $Rj$. When a request for resources is made by process $Pi$, the following actions are taken:

# TutorialsDuniya.com

**Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com**

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

**Please Share these Notes with your Friends as well**

If $Request_i \leq Need_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.

If $Request_i \leq Available$, go to step 3. Otherwise, $P_i$ must wait, since the resources are not available.

3. Have the system pretend to have allocated the requested resources to process $P_i$ by modifying the state as follows:

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

If the resulting resource-allocation state is safe, the transaction is completed, and process $P_i$ is allocated its resources. However, if the new state is unsafe, then $P_i$ must wait for $Request_i$, and the old resource-allocation state is restored.

## 6. Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, the deadlock situation may occur. In this environment, the system may provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

### 6.1 Single Instance of Each Resource Type

Maintain *wait-for* graph. We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges. An edge from *Pi* to *Pj* in a wait-for graph implies that process *Pi* is waiting for process *Pj* to release a resource that *Pi* needs. An edge $Pi \rightarrow Pj$ exists in a wait-for graph if and only if the corresponding resource-allocation graph contains two edges $Pi \rightarrow Rq$ and $Rq \rightarrow Pj$ for some resource *Rq*.

Periodically invoke an algorithm that searches for a cycle in the graph. An algorithm to detect a cycle in a graph requires an order of $n^2$ operations, where *n* is the number of vertices in the graph.

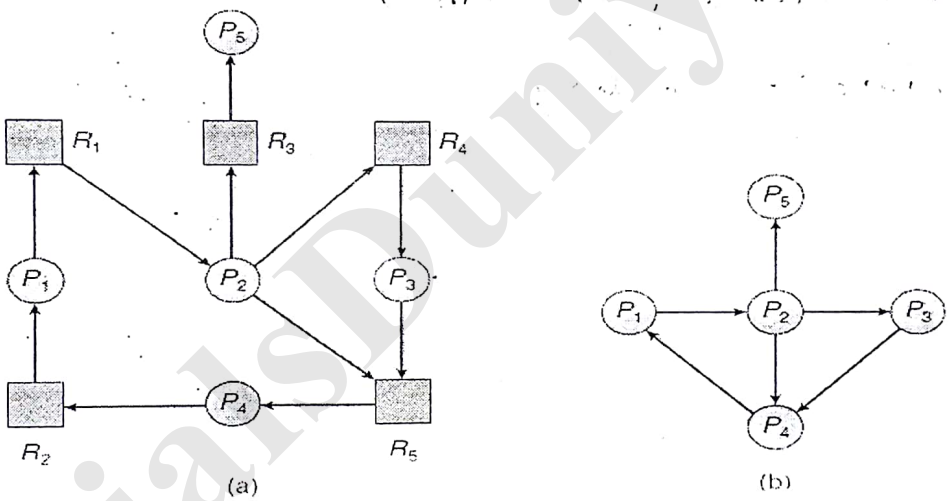



**Figure :** (a) Resource-allocation graph. (b) Corresponding wait-for graph.

### Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. The algorithm employs several time-varying data structures that are

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

**Allocation**. An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.

**Request**. An $n \times m$ matrix indicates the current request of each process. If *Request*[*i*][*j*] equals *k*, then process *Pi* is requesting *k* more instances of resource type *Rj*.

**Detection Algorithm**

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize

   *Work*=*Available*. For *i*=0, 1, ..., *n*-1, if *Allocationi* _ = 0, then

   *Finish*[*i*]=*false*. Otherwise, *Finish*[*i*]=*true*.

2. Find an index *i* such that both

   a. *Finish*[*i*] == *false*

   b. *Requesti* ≤ *Work*

   If no such *i* exists, go to step 4.

3. *Work*=*Work*+*Allocationi*

   *Finish*[*i*]=*true*

   Go to step 2.

4. If *Finish*[*i*] ==*false* for some *i*, $0 \le i < n$, then the system is in a deadlocked state. Moreover, if *Finish*[*i*] == *false*, then process *Pi* is deadlocked.

**Example:**

We consider a system with five processes *P*0 through *P*4 and three resource types *A*, *B*, and *C*. Resource type *A* has seven instances, resource type *B* has two instances, and resource type *C* has six instances. Suppose that, at time *T*0, we have the following resource-allocation state:

| | *Allocation* | *Request* | *Available* |
|---|---|---|---|
| | *A B C* | *A B C* | *A B C* |
| $P_0$ | 0 1 0 | 0 0 0 | 0 0 0 |
| $P_1$ | 2 0 0 | 2 0 2 | |
| $P_2$ | 3 0 3 | 0 0 0 | |
| $P_3$ | 2 1 1 | 1 0 0 | |
| $P_4$ | 0 0 2 | 0 0 2 | |

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence <*P*0, *P*2, *P*3, *P*1, *P*4> results in *Finish*[*i*] == *true* for all *i*.

Suppose now that process *P*2 makes one additional request for an instance of type *C*. The *Request* matrix is modified as follows:

| | *Request* |
|---|---|
| | *A B C* |
| $P_0$ | 0 0 0 |
| $P_1$ | 2 0 2 |
| $P_2$ | 0 0 1 |
| $P_3$ | 1 0 0 |
| $P_4$ | 0 0 2 |

We claim that the system is now deadlocked. Although We claim that the system is now deadlocked. Although we can reclaim the resources held by process *P*0, the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes *P*1, *P*2, *P*3, and *P*4.

**Detection-Algorithm Usage**

When should we invoke the detection algorithm? The answer depends on two factors:

1. How *often* is a deadlock likely to occur?
2. How *many* processes will be affected by deadlock when it happens?

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken. In addition, the number of processes involved in the deadlock cycle may grow.

## 7. Recovery from Deadlock

### 7.1 Process Termination

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

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.

Many factors may affect which process is chosen, including:

- What the priority of the process is
- How long the process has computed and how much longer the process will compute before completing its designated task
- How many and what types of resources the process has used (for example, whether the resources are simple to preempt)
- How many more resources the process needs in order to complete
- How many processes will need to be terminated
- Whether the process is interactive or batch

### 7.2 Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

If preemption is required to deal with deadlocks, then three issues need to be addressed:

- **Selecting a victim** – minimize cost.
- **Rollback** – return to some safe state, restart process for that state.
- **Starvation** – same process may always be picked as victim, include number of rollback in cost factor

Unit-7

# File System



## 1. File Concept:

Computer stores information on various storage media like magnetic disks, tapes and optical disks. Files are mapped by OS onto physical devices. File is named collection of related information that is recorded or stored on secondary storage. Commonly, files represents programs and data, where Data may be numeric, alphabetic or binary. Many different types of information may be stored in file: source, object, executable programs and so on

- Source file is sequence of sub-routines and functions each of which further organized as declarations followed by statements
- Object file is sequence of bytes organized into blocks understandable by system linker
- Executable file is series of code sections that loader can bring into memory and execute

### 1.1 File Attributes:

File attributes vary from one OS to another but typically consist of these:

- **Name** – symbolic file name is only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system, non human readable name
- **Type** – needed for systems that support different types of files
- **Location** – pointer to device and to the location of file on that device
- **Size** – current file size
- **Protection** – Access Control information that define who can do reading, writing, executing
- **Time, date, and user identification** – data for protection, security, and usage monitoring

### 1.2 File Operations:

- File is an **abstract data type**, operation that can be performed on files must be known to define file properly, these operations are provided by OS
- **Operations are:**
  - **Creating file**: first space in file system must be found for file and then entry for new file must be made in directory to create file
  - **Writing file**: make a system call specifying both name of file and information to be written, with given name system searches directory to find location and keeps write pointer to write into file
  - **Reading file**: make as system call that specifies name of file , with given name system reaches directory to find location and keeps read pointer to read contents from file
  - **Repositioning within file**: directory is searched for appropriate entry and current file position pointer is re-positioned to given value which is known as Seek
  - **Deleting file**: search directory for named file, releases all file space so that it can be reused by other files and erase directory entry

- Truncating file: user may want to erase contents of file but keep its attributes rather delete, this option lets file be resets to length zero and its file space released

- Other common operations:
  - Appending new information at the end of file
  - Renaming existing file
  - Copying file to another I/O device
  - Get & Set various attributes of file
- Some OS provide facilities for locking files, useful for files shared by several process
- File locks allows one process to lock file & prevent other process form granting access to it
- File locks provide functionality:
  - **Shared lock**: is reader lock in that several processes can acquire lock concurrently
  - **Exclusive lock**: is writer lock , only one process at a time can acquire lock
- OS may provide:
  - **Mandatory**: access is denied depending on locks held and requested
  - **Advisory** :processes can find status of locks and decide what to do

### 1.3 File Types:

- OS should recognize and support file types, then only it can operate on file in reasonable way.
- Common technique used is to include type as part of file name
- Name is split into two parts; **Name & Extension** separated by dot
- MS DOS recognizes only few extensions
- Application programs also use extensions to indicate file types: *.doc*

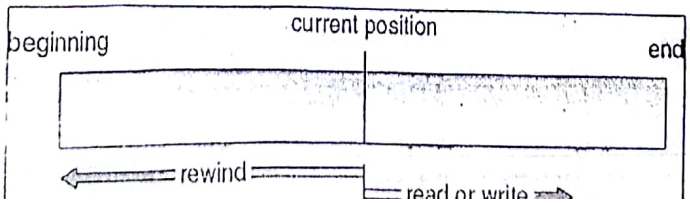
| file type | usual extension | function |
|---|---|---|
| executable | exe, com, bin or none | ready-to-run machine-language program |
| object | obj, o | compiled, machine language, not linked |
| source code | c, cc, java, pas, asm, a | source code in various languages |
| batch | bat, sh | commands to the command interpreter |
| text | txt, doc | textual data, documents |
| word processor | wp, tex, rtf, doc | various word-processor formats |
| library | lib, a, so, dll | libraries of routines for programmers |
| print or view | ps, pdf, jpg | ASCII or binary file in a format for printing or viewing |
| archive | arc, zip, tar | related files grouped into one file, sometimes compressed, for archiving or storage |
| multimedia | mpeg, mov, rm, mp3, avi | binary file containing audio or A/V information |

## 2. Access Methods

### 2.1 Sequential Access:

Simplest access method, file is accessed in order one record after other. Reads & Writes make up bulk of operations on file.

- Read Next: reads next portion of file, automatically advances file pointer
- Write Next: appends to end of file & advances to end of newly written data
- Tape model of file works as sequential access



### 2 Direct/ Relative Access:

ile is made up of **fixed length logical records** that allows to read or write records rapidly in no particular order

It is based on disk model of file, since disk allow random access to any file. Direct access files are used for immediate access of large amount information. Databases are often of this type, answer may contains in block of given query.

Ex: Airline reservation system, might store all info. of flight in particular block

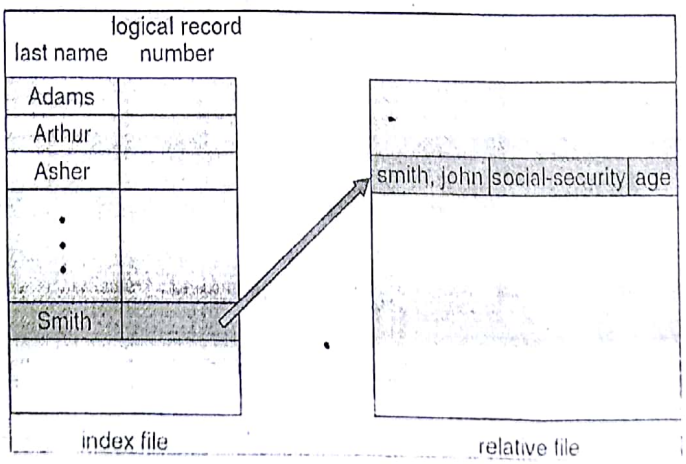
File operations includes block number N: **read N, write N**

| sequential access | implementation for direct access |
|---|---|
| *reset* | *cp = 0;* |
| *read next* | *read cp;*<br>*cp = cp + 1;* |
| *write next* | *write cp;*<br>*cp = cp + 1;* |

### 2.3 Other Access Methods:

Built on top of Direct access method, they involve contraction of index for file. **Index** contains pointer to various blocks in file like index in back of book. To find record in file, First search in index & then use pointer to access file directly

- Ex: retail price file might list Universal Product Codes (UPC) for items, with prices
- With large files, index file itself may become too large to kept in memory
  - One solution is to create index for index file
  - Primary index file will contain pointer to secondary index file, which points to actual data item
- Ex: IBM's indexed sequential access method (ISAM) uses small master index that points to disk blocks of secondary index
- Secondary index blocks point to actual file blocks
- Binary search is used to find block containing desired record
- Finally block is searched sequentially



## 3. Directory Structure:

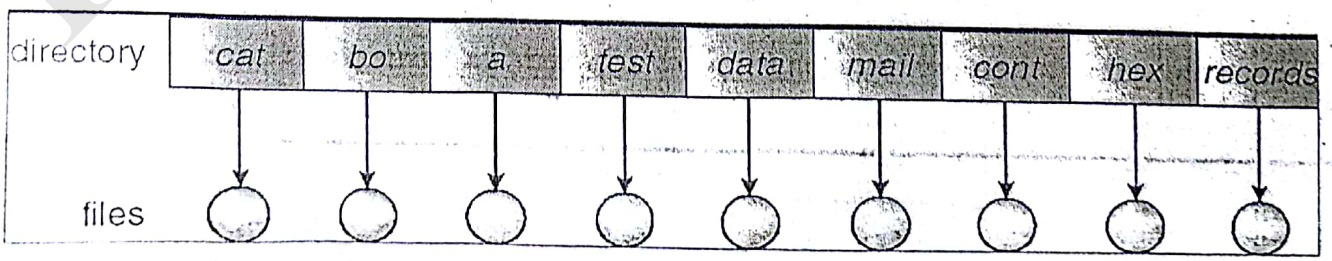
- Systems may have zero or more file systems & may be of varying types
- Ex: Solaries have few UFS file system, VFS file system, NFS file systems
- File system can be extensive, some systems store millions of files on tera bytes
- Organization of these data involves use of directories

### 3.1 Directory Overview:

- It can be viewed as symbol table that translates file names into their directory entries
- Directory itself can be organized in many ways: insert, delete entries, search form named entry, list all entries etc.
- Operations that are to be performed on directory:
  - **Search for a file**: search directory to find entry for particular file, files have symbolic names & similar names may indicate relationship between files
  - **Create a file**: new files need to create & add to directory
  - **Delete a file**: when file is no longer needed, remove it from directory
  - **List a directory**: able to list files in directory & contents of directory entry
  - **Rename a file**: file name represents its contents, able to change name when contents or use of file changes, it may allow to change its position in directory
  - **Traverse the file system**: for reliability, save contents & structure of entire file system at regular intervals, if file is no longer in use file can be copied to tape & disk space of that file released for reuse by another file .

### 1. Single Level Directory:

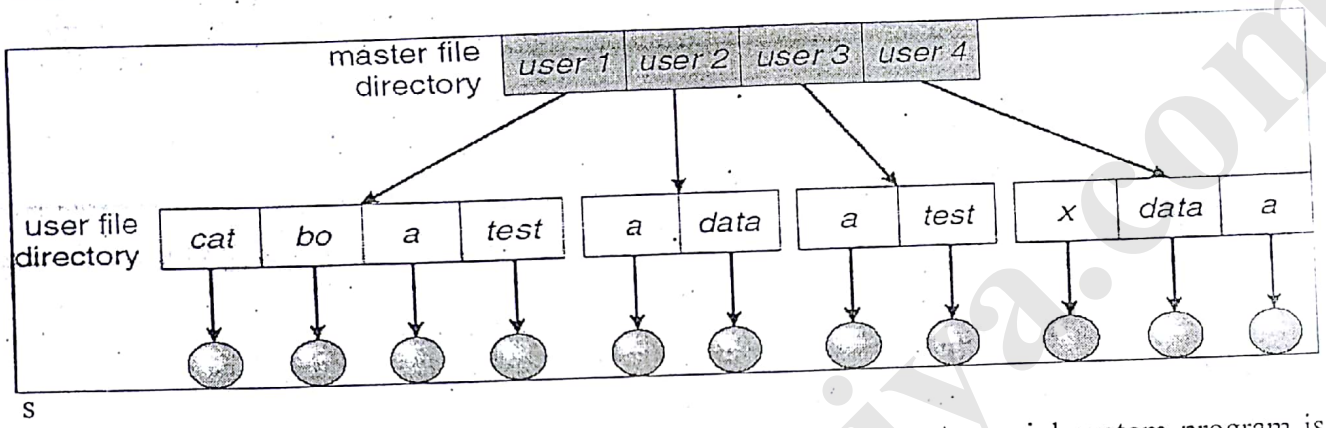
- All files are contained in same directory, which is easy to support & understand
- When number of files increases or when system has more than one user, it has significant limitations
- Since all files are in same directory, they must have unique names
- Length of file name is also limited
  - Ex: MS-DOS allows only 11 characters, UNIX allows 255 characters
- Single user on single level directory may find difficult to remember names of all files as number of files increases



## 2. Two Level Directory:

- Single level directory often leads to confusion of file names among different users.
- One solution is to create **Separate directory** for each user
- In two level directory structure

  Each user has his own **User File Directory (UFD).** When user job starts or user logs in, systems **Master File Directory (MFD)** is searched. MFD is indexed by user name or account number & each entry points to UFD of user. When user refers to particular file, only his own UFD is searched. Different users may have files with same name. To create or delete file OS searches in users UFD




s

  er directories themselves must be created or deleted as necessary. A special system program is run with appropriate user name & account information. It creates new UFD & adds entry for it to MFD

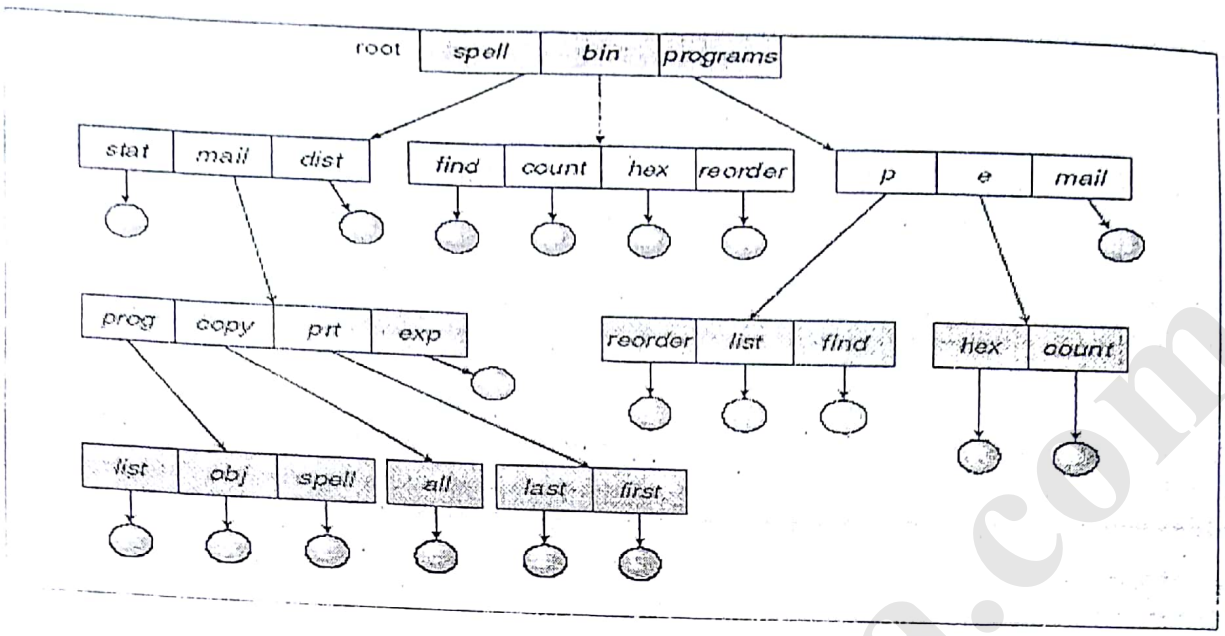
- Although two level directory solves name collision problem, it has some drawbacks
- **Advantages & Disadvantages:**
  - Isolation:
    - This structure effectively isolates one user from another
    - It is useful when users are completely independent
    - When users want to cooperate on some task by sharing files, it do not allows
    - If access it to be permitted, one user must have ability to name file in another users directory, we must give both user name & file name
- Two level directory can be thought of tree of height 2, root is MFD & Childs are UFD's
- Childs of UFD's are files, files are leaves of tree.

## 3. Tree Structured Directories:

Two level directory structure can be extended to tree of arbitrary height. This generalization allows users to create their own sub directories & to organize their files accordingly. Tree has root directory, every file in system has unique path name. Directory contains set of file or subdirectories, directory is simply another file but is treated in special way, All directories has same internal format

One bit in each directory entry define entry as **file (0) or sub directory (1).** special system call are used to create & delete directories Each process has **Current Directory**, contains most of files that are of

current interest When reference is made, it will be searched in current directory first If is not found, then user must either specify path name or change current directory.



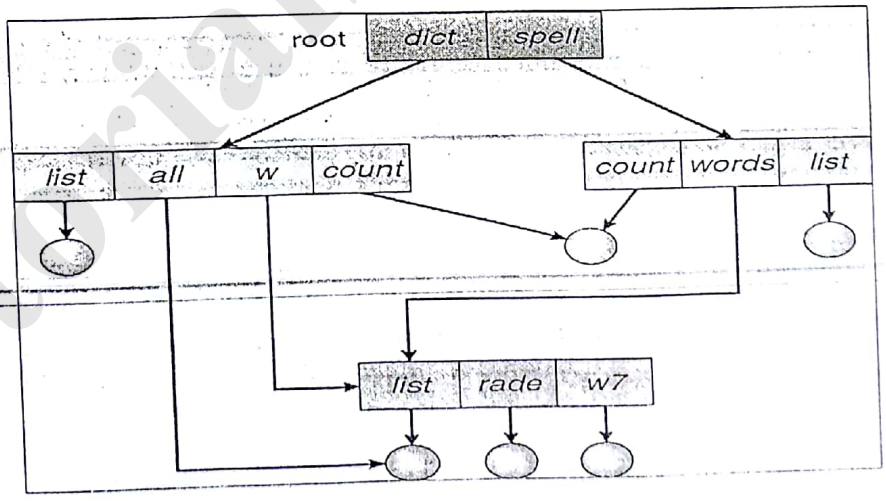

- Path names can be of tow types:
  - Absolute Path name:
  - Relative path name
- Difficulty in tree structured directory is **how to handle deletion of directory.** If directory is empty, its entry in directory that contains it can simply deleted. If directory is not empty then:
  - Do not delete until directory is empty Ex: MS-DOS
  - Provide option to delete all files & sub directories to be deleted or not Ex: UNIX
- With tree structure, users can be allowed to access in addition to their files of other users, path in tree structure is longer than two lever structure

## 4. Acyclic-Graph Directories:

- Consider two programmers who are working on joint project
  - Files associated with that project can be stored in sub directory, separating them from other projects & files of two programmers
  - Since both programmers are equally responsible for project, both want sub directory to be in their own directories
  - Common sub directory should be shared, Shared directory or file will exist in file system in two places at once
- A tree structure prohibits sharing of files or directories.
- An Acyclic Graph is a graph with no cycles allows directories to share sub directories & files. Acyclic graph is natural generalization of tree structured directory scheme
- With two copies, each programmer can view copy rather than original

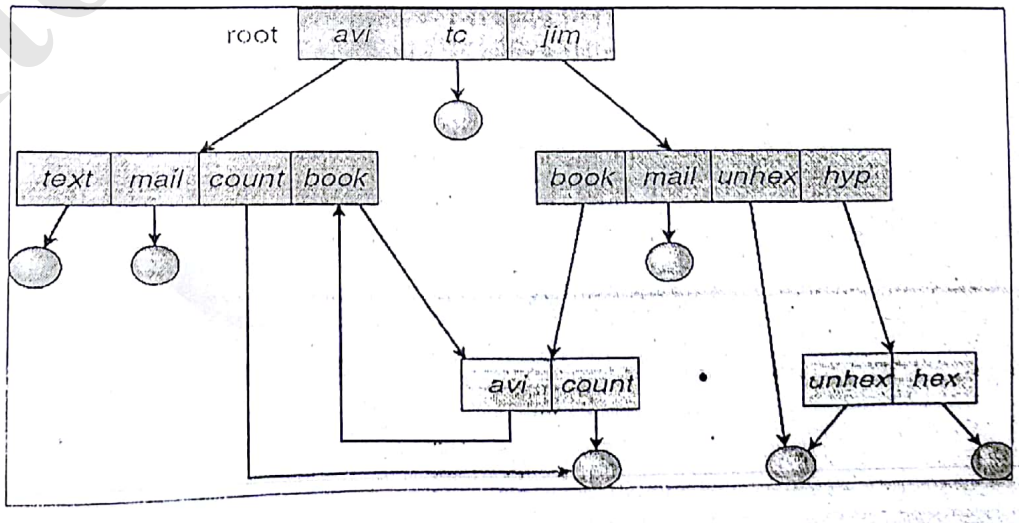
- If one programmer changes file, changes will not appear in others copy
- With shared files only one actual file exists, so any changes made by one person are immediately visible to other

- Even in single user, users file organization may require that some file be placed in different sub directories, Ex: program written for particular project should both in directory of all programs
- Simple ways is:
  - In UNIX common way to create new directory entry called Link, link is effectively pointer to another file or sub directory Ex: link may be absolute or relative path name
  - If directory entry is marked as link, then name of real file is included in link information
  - We resolve link by using that path name to locate real file
  - Links are easily identified by their format in directory entry & effectively named indirect pointers
- Another ways is:
  - To implement shared files is simply to duplicate all information about them in both sharing directories, both entries are identical & equal
  - Duplicate directory entries, make original & copy indistinguishable
  - Major problem is maintaining consistency when file is modified.
- Acyclic graph structure is more flexible than simple tree structure, but is more complex
  - A file may now have multiple absolute path names, distinct file names may refer to same file
  - 
  - When can space allocated to shared file be de-allocated & reused?
  - If file entry itself is deleted, space for file is de-allocated, leaving links dangling



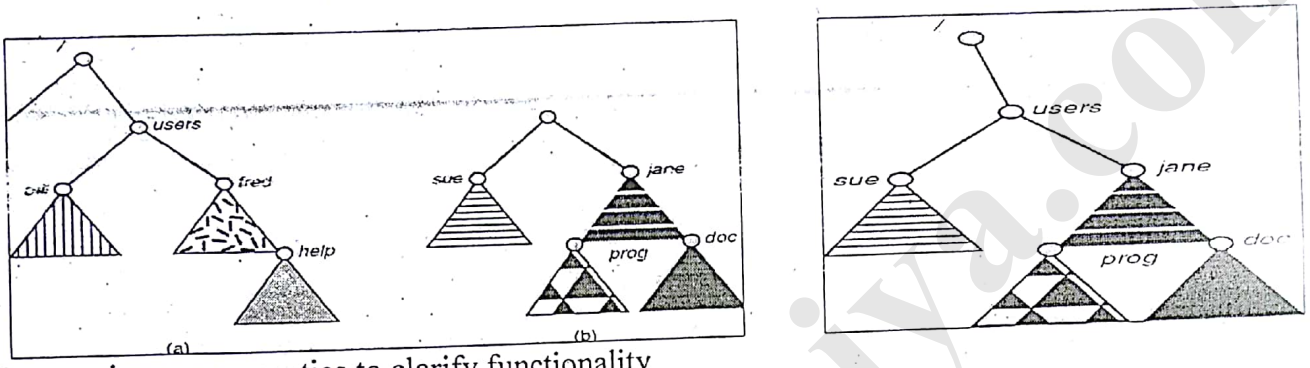
## 5. General Graph Directory

- Serious problem with acyclic graph structure is ensuring that there are no cycles. If we start with two level directory & allow users to create sub directories, a tree structured directory results. When we add links to existing tree structured directory, tree structure is destroyed resulting in simple graph structure
- Primary advantages of acyclic graph is Relative simplicity of algorithms to traverse graph & determine when there are no more references to file. If cycles are allowed to exist in directory, we likewise want to avoid searching any component twice, for reasons of correctness as well as performance
- Poorly designed algorithm might results in infinite loop continually searching through cycle & never terminating.
- one solution to limit arbitrarily number of directories that will be accessed during search Similar problem exists when we are trying to determine when file can be deleted. With acyclic graph directory structures, value of 0 in reference count means that there are no more references to file or directory & file can be deleted.
- When cycle exist, reference count may not be 0 even when it is no longer possible to refer to director or file
- This anomaly results from possibility of self referencing in directory structure
- In this case, we need to use garbage collection scheme to determine when last reference has been deleted & disk space can be reallocated. Garbage collection involves traversing entire file system making everything that can be accessed. Garbage collection for disk based file system is extremely time consuming
- Difficulty is to avoid cycles as new links are added to structure, how do we known when new link will complete cycle?
- There are algorithms to detect cycles in graphs, are computationally expensive when graph is on disk storage
- Simpler algorithm in special case of directories & links is to bypass links during directory traversal



## File System Mounting:

- A file system must be **mounted** before it can be accessed
- OS is given name of device and **Mount Point** – location within file structure where file system is to be attached, Typically mount point is an empty directory EX: in UNIX
- Next OS verifies that device contains valid file system by asking device driver to read device directory and verify the expected format
- Finally, OS notes in its directory structure that file system is mounted at the specified mount point
- It will enables OS to traverse its directory structure
- Consider the example:




- System impose semantics to clarify functionality
  - Ex: system may disallow mount over directory that contains files

## 5. File Sharing:

User oriented OS must accommodate need to share files. Once multiple users are allowed to share file.The challenge is to extend sharing to multiple file systems including remote

### Multiple Users:

- When OS accommodate multiple users, following issues are permitted
  - File sharing, File naming, File protection
- System must mediate file sharing, if we allow files to be shared by users
- For **Sharing & Protection**, it must maintain more file & directory attributes
- Most systems evolved to use concept of file **Owner & Group**
  - Owner can change attributes, grant access & have most control over it
  - Group defines sub set of users who can share access of file
- Owner & Group ID of given file are stored with other file attributes
- If a user requests file, User ID is checked with Owner attribute followed by Group attribute to grant permissions

### Remote File Systems:

- Networking allows sharing of resources spread over network
- Remote file sharing methods are changed with evolution of network:
  - **Manual transfer**: of file between machines via programs like FTP

- Distributed File System (DFS): remote directories are visible from local system
- World Wide Web: browser is needed to gain access to remote files

- FTP is used for both **Anonymous & Authenticated** access
- Anonymous access allows user to transfer files without having account on remote system Ex: WWW
- DFS involves much tighter integration between local & remote machines

Client Server Model:

- Remote file system allows computer to mount one or more file systems from one or more remote machines
- Machine containing file is **Server**, Machine seeking access is **Client**
- Server usually specifies available files on volume or directory
- Client can be specified by Network name or other identifier like IP address
- **Spoofing** is done, unauthorized client could be allowed to access server
- Secured encrypted keys are used to provide more security
- In Unix, Network File System (NFS) takes authentication via client networking
- In this user ID's of client & server should match other wise access denied
- Once remote file system is mounted, file access is allowed on behalf of user
- Server then apply standard access checks to determine user credential access
- If it is allowed, file handle will be returned to client

**Distributed Information Systems:**

- Known as Distributed Naming Services(DNS), provide unified access to info.
- It provides **Host name to Network address** translations for entire internet
- It provide user name, password, user ID, group ID for distributed facility
- Sun micro systems introduced yellow pages known as **Network Info. Service**
  - It centralizes storage of user, host names, printer information etc.
  - It uses unsecure authentication methods including pass words identify by host IP
  - NIS+ is much more secure replacement for NIS
- One distributed LADP directory is used by organization to store all user & resource information of other computers to result **Secure single sign in**

Failure Modes:

- Local file system can fail for variety of reasons
  - Failure of disk containing file system
  - Corruption of directory structure or disk management info. (meta data)
  - Disk, Cable, Host adapter failure
- Remote file system have even more failure modes
  - Complexity of network, more problems with interface interactions

# TutorialsDuniya.com

**Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com**

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

**Please Share these Notes with your Friends as well**

- Interrupts between hosts cause hardware failure, network implementation issues

**onsistency Semantics:**

- Consistency Semantics are
  - important criterion for evaluating any file system that supports file sharing
  - Used to specify how multiple users of system are to access shared file simultaneously
  - Typically implemented as code with file system
  - Directly related to Process Synchronization algorithms
- EX: performing atomic transaction to remote disk involve several network communications, disk reads & writes or both.

## 6. Protection:

- When information is stored in system, keep it safe from:
  - Physical Damage (Reliability)
  - Improper Access (Protection)
- Reliability is provided by duplicate copy of files
  - Many systems have programs that automatically copy disk files to tape
  - File System can be damaged by hardware problems, power failures, crashes, dirt, temperature extremes etc.
  - Files may be deleted accidentally
  - Bugs in file systems can also cause file content to lost
- Protection can be provided in many ways:
  - In Single user system, protect by physical remove of disk & locking
  - In Multi user system, other mechanisms are needed

**Types of Access:**

- Systems that do not permit access to files of other users do not need protection
- Complete protection is provided by Prohibiting access with **Controlled access**
- In Controlled access, access is permitted or denied depending on several factors one of which is type **of access** requested
- Types of operations may be controlled:
  - **Read**: read from file
  - **Write**: write or rewrite to file
  - **Execute**: load file into memory & execute it
  - **Append**: write new data at the end of file
  - **Delete**: delete file and free its space for possible reuse
  - **List**: list name & attributes of file

# TUTORIALSDUNIYA.COM

**Get FREE Compiled Books, Notes, Programs, Question Papers with Solution etc of the following subjects at https://www.tutorialsduniya.com**

- C and C++
- Java
- Data Structures
- Computer Networks
- Android Programming
- PHP Programming
- JavaScript
- Java Server Pages
- Python
- Microprocessor
- Artificial Intelligence
- Machine Learning
- Computer System Architecture
- Discrete Structures
- Operating Systems
- Algorithms
- DataBase Management Systems
- Software Engineering
- Theory of Computation
- Operational Research
- System Programming
- Data Mining
- Computer Graphics
- Data Science

---

- DU Programs: https://www.tutorialsduniya.com/programs
- TutorialsDuniya App: http://bit.ly/TutorialsDuniyaApp
- C++ Tutorial: https://www.tutorialsduniya.com/cplusplus
- Java Tutorial: https://www.tutorialsduniya.com/java
- JavaScript Tutorial: https://www.tutorialsduniya.com/javascript
- Python Tutorial: https://www.tutorialsduniya.com/python
- Kotlin Tutorial: https://www.tutorialsduniya.com/kotlin
- JSP Tutorial: https://www.tutorialsduniya.com/jsp

- Other operations like rename, copy, edit may also be controlled
- Protection is provided at only **Lower level**, all system may not have same type of protection ex: research group & large corporate computers

Access Control:

- Common approach is to make **access dependent on identity of user**
- Different users may need different types of access to file or directory
- Access Control List(ACL), most general scheme to implement identity dependent access to each file & directory
- ACL specifies user names & type of access allowed for each user
- When user request access to particular file:
  - OS checks access list associated with that file
  - If user is listed for the requested access, access is allowed
  - Otherwise, protection violation occurs and access denied
- **Advantage**: enables complex access methodologies
- **Problem**: with access lists is their length
  - If we want to allow everyone to read file, must list all users with read access
  - It requires list of users in system, constructing such list is waste
  - In general Directory is fixed size, now it must be variable size
- It can be resolved by **Condensed version of access list**
- Classification of users in connection with each file:
  - **Owner**: creator of file
  - **Group**: set of users who share the file
  - **Universe**: all other users
- Most common recent approach is combine access control lists with above classes
- Ex: User 1 able to use all operations, User2, 3 able to use only Read & Write, all others able to read only
- This can be achieved by creating a Group associated with the file, set access permissions as required
- Permissions & Access lists must be maintained tightly
- In case of UNIX:
  - Groups can be created or modified by the managers of system
  - In case of limited protection, only three fields are needed with 3 bits – RWX
  - More protection, can be included with 9 bits for 3 class of 3 fields each
- Windows XP users mange access control lists via GUI.

Other Protection Approaches:

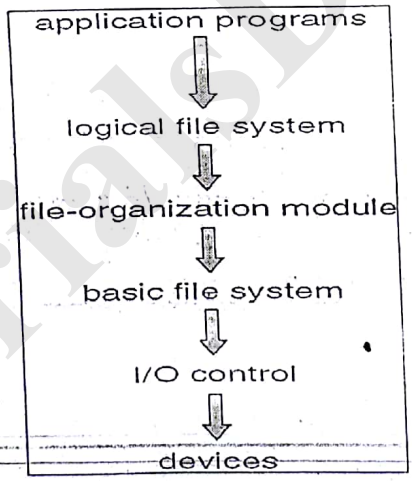
- Associate password with each file like computer password
- Disadvantages:

- Number of passwords that user needs to remember become large
- If only one password is used for all files & is discovered all files are accessible

- Some system allow user to associate password for sub directory rather than file
- Old OS's like DOS provide limited file protection
- In multilevel directory structure, we need to protect not only individual file but also collection of files in sub directories
- **Directory protection** is also required by restricting directory creation, deletion.

## 7. File System Structure:

- Disk provides bulk of secondary storage on which file system is maintained
- They have two characteristics:
  - Disk can be written in place
  - Disk can access directly any given block of information it contains
- Disk are performed in units of **Blocks** to improve I/O efficiently, transfer
- File system poses two quite different design problems:
  - Defining how file system should look to user i.e. defining file & its attributes, operations
  - Creating algorithms & data structures to map logical file system to physical storage
- File system itself composed of many different levels




- **I/O Control level**: consists of **Device drivers** and **Interrupt handlers** to transfer data between memory & disk
- **Basic File System**: issues only generic commands to appropriate device driver to read & write physical blocks on disk
- **File Organization Module**: knows about file and their logical & physical blocks
- **Logical File System**: manages meta data information, it includes all file system structure except actual data

- Logical file system manages directory structure to provide file organization module with information of given symbolic file name
- It maintains file structure via **File Control Block (FCB)**
- FCB contains information about file including ownership, permissions, and location of file contents
- Most OS support more than one file systems
- EX: Unix uses **UNIX file system (UFS)** which is based on Berkeley Fast File System (BFFS)
- Windows support **FAT, FAT32, NTFS**
- Linux file system is known as **External File System**

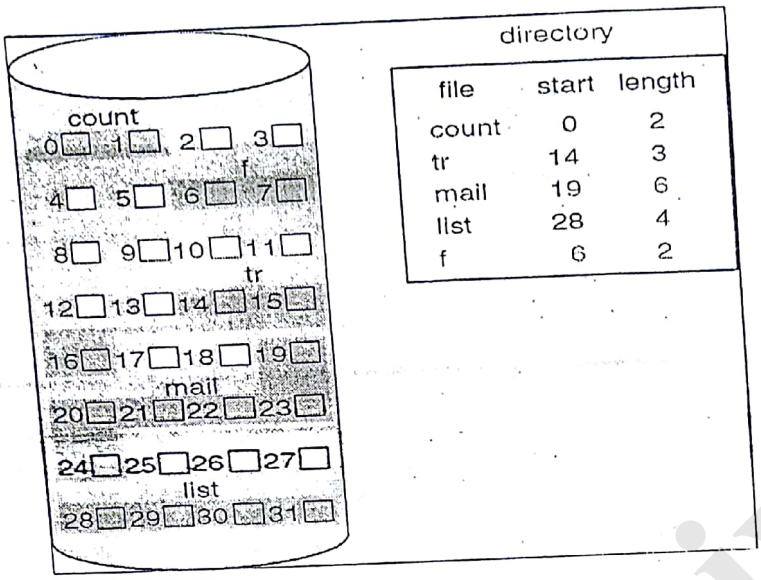
## 8. Allocation Methods:

- Many files are stored on same disk. Main problem is how to allocate space to files for effective utilization.
  - For effective utilization of disk space
  - Quick access of files
- Three major methods are as follows: contiguous, linked and indexed

### 8.1 Contiguous Allocation:

- Each file occupy set of contiguous blocks on disk, disk addresses define linear ordering on disk.
  Ex: IBM VM/CMS uses contiguous allocation
- Contiguous allocation of files is defined by **disk address & length of block.** Directory entry for each file indicates **address of starting block & length of area allocated for this file**
- Accessing is easy:
  - Sequential Access, disk address of last reference must be remembered
  - Direct Access, block *i* of file start at block *b* can access *b+i* directly
- Problems:
  - Finding space for new file
  - Dynamic storage allocation
  - External fragmentation
- Another problem is determining how much space is need for file.
- When file is created, total amount of space it will need must be found
- How does creator know size of file to be created?
  - Size of output file may be difficult to estimate
  - If we allocate too little space to file, file cannot be extended
  - With Best fit allocation, space on both sides of file may be in use & cannot make file longer
  - Two possibilities exist:
    - Terminate program, allocate more space, run program again, these repeated runs are costly

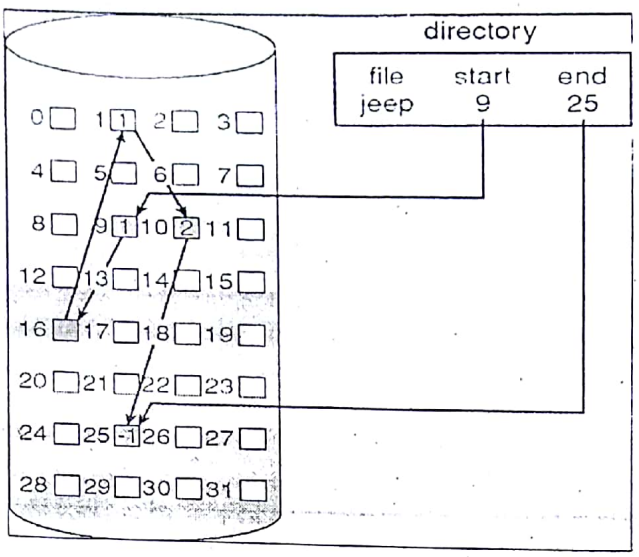
- Find larger hole, copy contents to file to new space, release previous space, time consuming
- Even if total amount of space need for file is known in advance, pre-allocation may inefficient for **slowly growing files**
  - To overcome external fragmentation compaction is used.




## 8.2 Linked Allocation:

- It solves all problems of contiguous allocation.
- Each file is linked list of disk blocks, disk blocks may be scattered any where on disk
- Directory contains pointer to first & last block of file
- Each block contains pointer to next block, pointers are not visible to user
- To create new file, simply create new entry in directory with **pointer to first disk block** initialized with **NIL, size is set to 0** to indicate **empty file**
- **Write to file** cause free space management to find free block & new block is written and linked to end of file
- **Reading file**, is simple by reading blocks with pointers from block to block
- There is no **External fragmentation**, any free block on free space is used. No need of **compaction**, file can grow as long as free blocks are available
- **Disadvantages:**
  - Inefficient for Direct access
    - It can be effectively used for **sequential access** files
    - To find $i^{th}$ block of file for **Direct access**, start at beginning of that file and follow pointers until we got $i^{th}$ block
  - Another disadvantage is **Space required for pointers.**
  - Reliability
    - If pointers were lost or damaged due to bug in OS or disk hardware failures
    - This error will result in linking into free space list or another file

- One solution is use **Doubly linked list** and store file name & relative block number in each block .

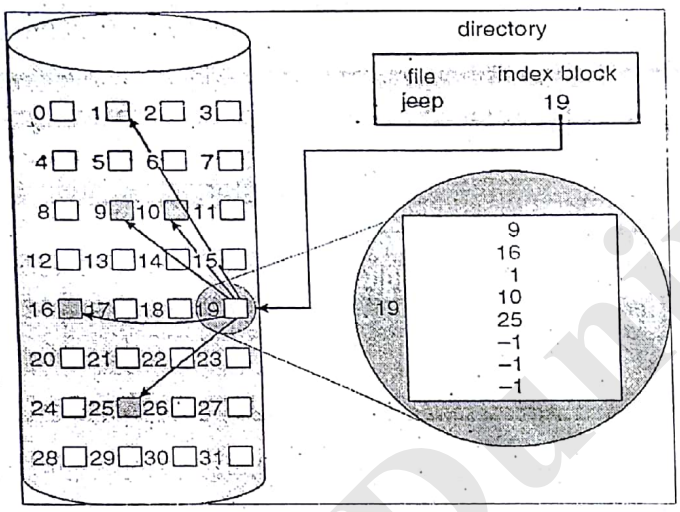



## 8.3 Indexed Allocation:

- Linked allocation solves External fragmentation, Size declaration problems. In absence of FAT, linked allocation can not support efficient direct access.
- Indexed allocation solves this problem by bringing all pointers together into one location known Index block. Each file has its **own index block**, which is **array of disk block addresses.** $i^{th}$ entry in index block points to $i^{th}$ block of file.
- Directory contains address of index block. To find or read $i^{th}$ block, use pointer in $i^{th}$ index block entry like paging.
- When pointer is created, all pointers in index block are set to NIL When $i^{th}$ block is first written, block is obtained form free space and its address is put in $i^{th}$ index block entry.
- Supports direct access with out suffering from external fragmentation
- Indexed allocation suffer from **wasted space**
- Pointer overhead of index block is generally greater than pointer overhead of linked allocation
- An entire index block must be allocated, even if only one or two pointers will be non NIL
- Every file must have index block, so index block to be as small as possible
- If index block is too small, it will not be able to hold enough pointers for large file, mechanism for this purpose include following:
- **Linked Scheme:**
  - Index block is normally one disk block, can read & write directly by it self
  - To allow for large files, we may link together several index blocks
  - Index block contains small header giving name of file & set of first 100 disk block address
  - Next address is pointer to another index block
- **Multilevel Index:**
  - Use first level index block to point to set of second level index blocks

- Which in turn point to file blocks
- To access block OS uses first level index to find second level index to find desired block
- N levels of index can be implemented

- **Combined Scheme:**
  - UFS uses this approach, which will keep first **15 pointers of index block** in files inode
  - First **12 pointers** point to **direct blocks**, contains address of data block
  - Next **3 pointers** points to **indirect blocks**
    - First indirect block pointer is address of **Single indirect block**
    - Single indirect block is an index block, containing addresses of block
  - Combined Scheme: UNIX (4K bytes per block)




## 9. Free Space Management:

Since disk space is limited, we need to reuse space from deleted files for new files if possible. To keep track of free disk space, system maintains **Free space list.** It records all free disk blocks those are not allocated to some file or directory. To create file, search free space list for required amount of space & allocate that space to new file

### 9.1 Bit Vector:

- Frequently, Free space list is implemented as Bit map or Bit vector
- Each block is represented by 1 bit, if block is free, the bit is 1, it block is allocated, bit is 0
- Main advantage of this approach is
  - Its relative simplicity
  - its efficiency in finding first free block or N consecutive free blocks on disk
- Many computers supply bit manipulation instructions that can be used effectively for that purpose
- One technique for finding first free block on system that uses bit vector to allocate disk space is to sequentially check each word in bit map to see whether that value is not 0
- First non 0 word is scanned for first 1 bit, which is location of first free block

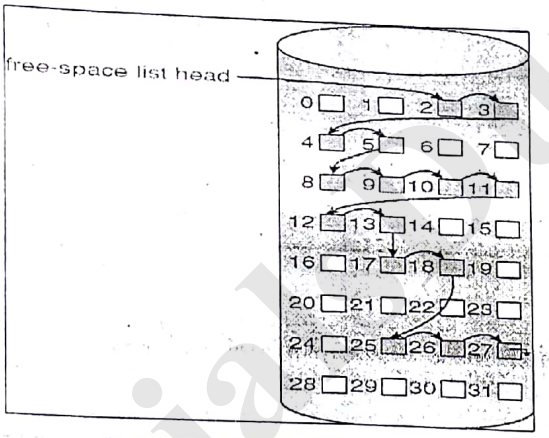
- Calculation of block number is:

  (number of bits per word)x(number of 0 value words)+offset of first 1 bit

- Hardware features driving software functionality
  - Bit vectors are inefficient unless entire vector is kept in main memory
  - Keeping it in main memory is possible for smaller disks but not for larger disks.

## 9.2 Linked List:

- Linking together all free disk blocks, keeping pointer to first free block in special location on disk & caching it in memory
- First block contain pointer to next free disk block & so on
- However, it is not efficient
  - To traverse list, must read each block
  - Which requires substantial I/O time
- Usually OS simply needs to free block so that it can allocate that block to file, so first block in free list is used
- FAT method incorporates free block accounting into allocation data structures, no separate method is needed




## 9.3 Grouping:

- Modification of free list approach is to store address of N free blocks in first free block
- First N-1 of these blocks are actually free, last block contains addresses of another N free blocks
- Address of large number of free blocks can now be found quickly

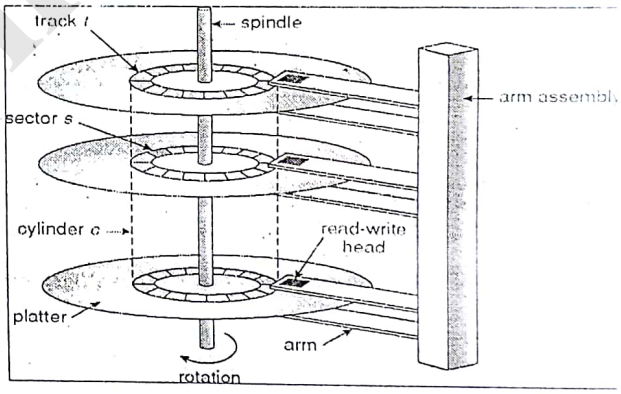
## 9.4 Counting:

- Several contiguous blocks may be allocated or freed simultaneously, particularly when space is allocated with contiguous allocation algorithm or through clustering
- Rather than keeping list of N free disk addresses, we can keep address of first free block & number N of free contiguous blocks that follow first block. Each entry in free space list then consists of disk address & count. Although each entry requires more space than simple disk address, overall list will be shorter as long as count is greater than 1.

## erview of Mass Storage Structure:

### agic Disks:

- Magnetic disks provide bulk of secondary storage of modern computers
  - **Platter** is flat circular disk with two surfaces covered with magnetic material
  - Data is stored by recording it magnetically on these platters
  - **Read-Write head** flies just above each surface of ever platter, they are attached to **Disk arm** that moves all heads as unit
  - Surface of platter is logically divided into circular **Tracks**, which are sub divided into **sectors**, Set of tracks that are at one arm position make up **Cylinder**
  - Most drives rotate at 60 to 200 times per second
  - Disk speed has two parts(typically in MB per Second):
    - **Transfer rate** is rate at which data flow between drive and computer
    - **Positioning time (random-access time)** is time to move disk arm to desired cylinder (**seek time**) and time for desired sector to rotate under the disk head (**rotational latency**)
  - Disk platter is coated with protective layer to avoid contact between head & disk
- Sometimes head will damage magnetic surface called as **Head crash,** head crash normally cannot be repaired
- Disks can be **removable**, removable disks have one platter like **Floppy disk**.
- Floppy disk is inexpensive with **soft plastic case & flexible platter**
- Head of Floppy disk sits directly on disk surface, disk will be rotate slowly




- Disk Drive is attached to computer by set of wires known as **I/O bus**
  - Busses vary, including **EIDE, ATA, SATA, USB, Fibre Channel, SCSI**
  - **Controllers**: special processors used for Data transfer on bus
    - **Host controller** is controller at computer end of bus
    - **Disk controller** is built into each disk drive for Disk I/O
  - Computer places command on host controller, then sent to disk controller
  - Disk controller operates disk drive hardware to carry out command

### Magnetic Tapes:

- It is relatively permanent & can hold large quantities of data with slow access
- In addition, random access on magnetic tape is hundred times slower than magnetic disk
- Tapes are used mainly for backup for storage of infrequently used information
- Moving to correct spot on tape can take minutes, it can write data fastly

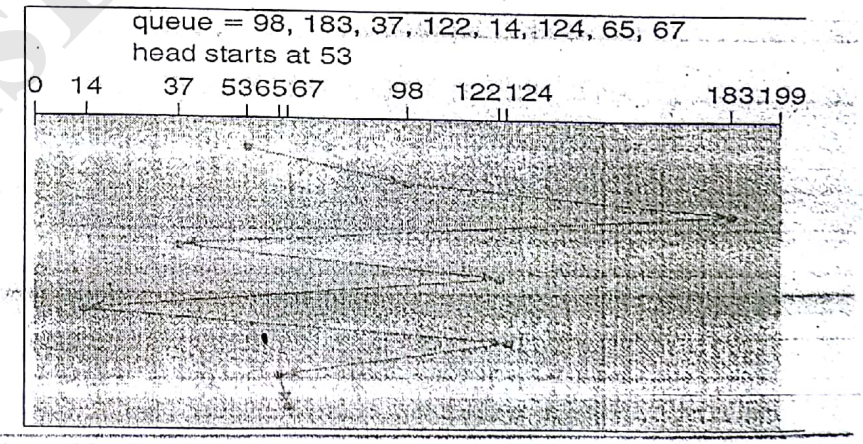
- Tape capabilities vary generally, depending on particular kind of tape drive
- Types & their drivers are usually categorized by width

## Disk Scheduling:

- One of responsibilities of OS is to use hardware efficiently
- For disk drives, **fast access time & large bandwidth** should be achieved
- Access time has two components:
  - **Seek time**: is time for disk arm to move head to cylinder containing desired sector
  - **Rotation Latency**: is time for disk to rotate desired sector to disk head
- Disk bandwidth is total number bytes transferred divided by total time between first request of service & completion of last transfer
- Disk Scheduling is used to improve both Access time & Bandwidth
- I/O is done by **system call.** it specifies several pieces of information:
  - Whether this operation is input or output
  - What is the disk address for transfer
  - What is the memory address for transfer
  - What is the number of sector to be transfer
- If the desired disk drive & controller are available, request can be serviced
- If it is busy, request will be placed in queue of pending requests of drive

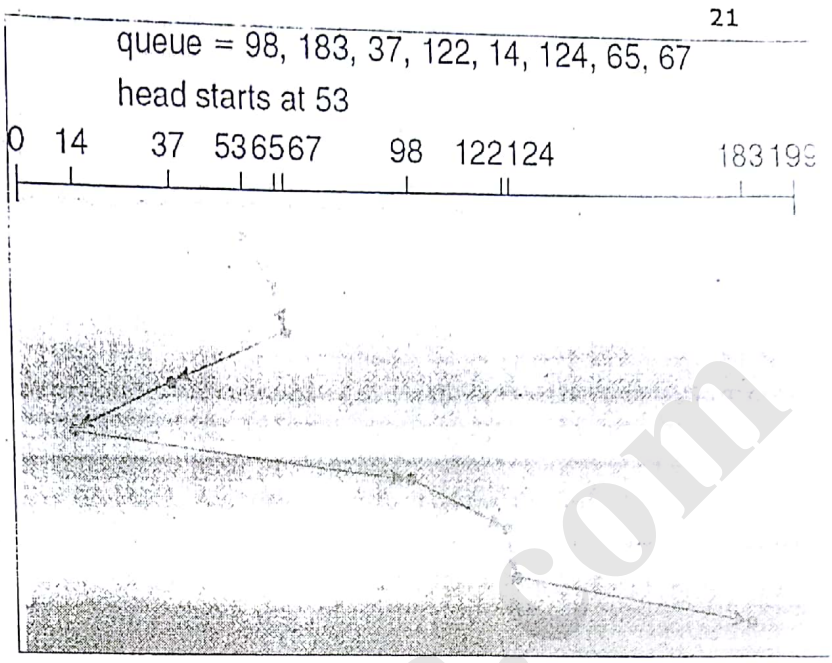
### FCFS Scheduling:

- Simplest form of disk scheduling is **First Come First Served(FCFS)** algorithm
- It is fair, but does not provide fastest service
- Ex: consider disk queue with request for I/O for blocks on cylinders
  98, 183, 37, 122, 14, 124, 65, 67



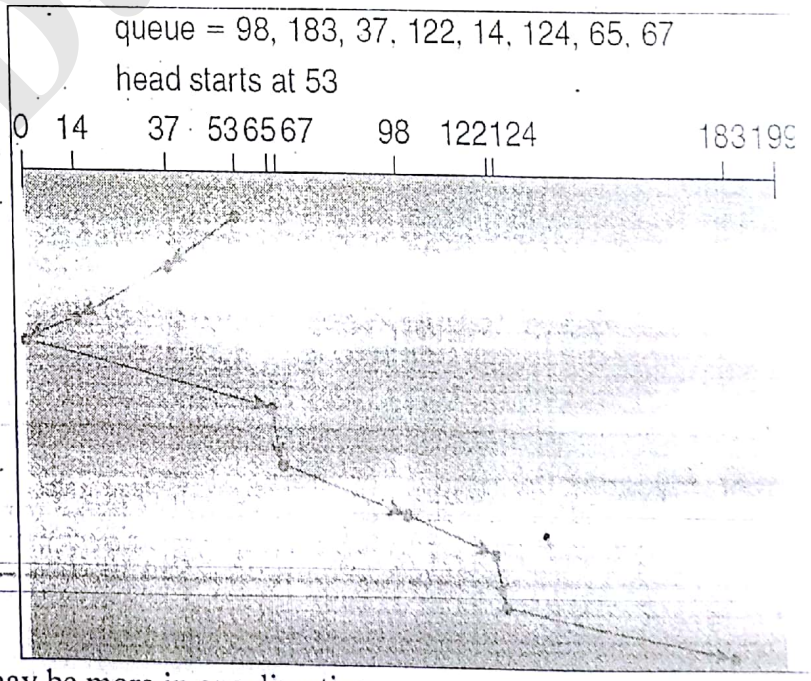

- **Problem:**
  - Wild swing from 122 to 14 & then back to 124
  - Will performance of head movement

**F Scheduling:**




- It serves all requests close to the **current head position**
- **Shortest Seek Time First (SSTF)** algorithm will select request with **minimum seek time** from current head position
- It chooses pending request closes to current head position
- It gives improved performance than FCFS algorithm
- SSTF is essentially a form of SJF scheduling
- Ex: 98, 183, 37, 122, 14, 124, 65, 67
- **Problem:**
  - Similar as SJF it also had **Starvation** problem
  - the request with largest seek time will always for the shortest seek time requests

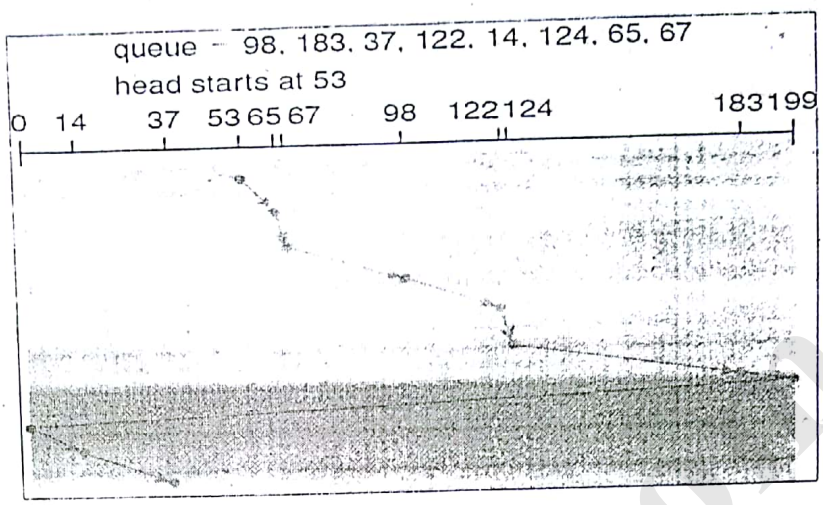
**SCAN Scheduling:**




- Disk arm starts at one end of the disk & moves toward other end
- It serves requests as it reaches each cylinder until it get to other end of disk
- At other end, direction of head movement is reversed & continues serving requests
- Head continuously scans back & forth across disk
- It is known **Elevator Algorithm**
- Ex: 98, 183, 37, 122, 14, 124, 65, 67
- **Problem:**
  - Density of requests in one direction may be more in one direction
  - If heaviest density requests are at other end of disk, they have to wait for longest time

C-SCAN Scheduling:

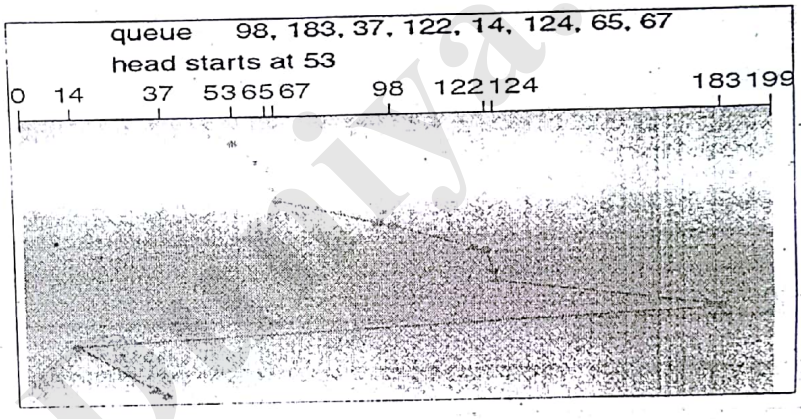
- Circular SCAN is designed to provide more **uniform wait time**
- In this head moves from one end to other end of disk serves requests along the way
- When head reaches other end, it immediately returns to beginning of disk without serving any request on return trip



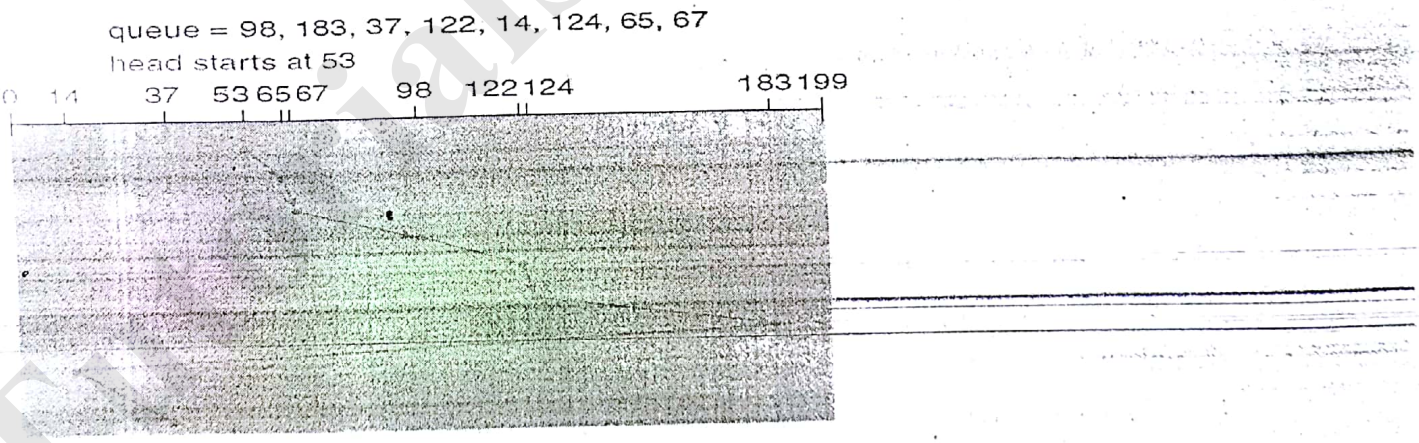

- It will treat cylinders as circular list that wraps around from final cylinder to first one

LOOK Scheduling:

- Both SCAN & C-SCAN move disk arm across full width of disk
- In LOOK, Arm goes only as far as final request in each direction
- It reverses direction immediately, without first going all way to end of disk




C-LOOK Scheduling




Selection of Disk scheduling algorithm:

- How do we choose best one?
  - SSTF is common and has natural appeal
  - SCAN & C-SCAN perform better for systems that place heavy load on disk
- Performance depends heavily on **number & types of requests**
- Requests for disk service greatly **influenced by file allocation method**

# OPERATING SYSTEMS (JNTUK-R16)

## UNIT-6; LINUX SYSTEM & ANDROID SOFTWARE PLAT FORM

SYLLABUS:

- Linux System
  - Components of Linux
  - Interprocess Communication
  - Synchronization
  - Interrupt
  - Exception
  - System Call
- Android Software Platform
  - Android Architecture
  - Operating System Services
  - Android Runtime
  - Application Development
  - Application Structure
  - Application Process Management

## PART-I: LINUX SYSTEM:

### Introduction to Linux Operating System:

- Linux is an open source operating system which is powerful and easy to implement.
- It can be easily installed on the system.
- It can run on various platforms and can be shared and distributed freely.
- The first version of LINUX was released in August 1991 through internet, not an official version.
- Another version 0.02 was published officially on October 5th 1991.
- This version allows GNO programs such as gcc, bash.
- Finally, the first version (0.01) was released publicly in March 1994.
- The most widely used LINUX operating systems are,
  - Arch Linux
  - CentOS
  - Debian and Raspbian
  - Fedora
  - Gentoo Linux
  - Linux Mint
  - Mageia
  - openSUSE
  - Ubuntu
  - Redhat

### 1. Components of Linux:

- The various components of Linux are,
  a) Process
  b) Thread
  c) Interprocess communication

d) Synchronization
e) Bootstrap
f) Time Management
g) Kernel Module

**a) Process:**

- A Process is a program in execution.
- A Process contains Program code and a set of data
- The process attributes are,
  - Identifier
  - Process Sate
  - Priority
  - Program Counter
  - Memory Pointer
  - Context Data
  - I/O status Information
  - Accounting Information

**Process Control Block:**

- All Process information available in Process Control Block
- PCB includes
  - Process Identification
  - Process State Information
  - Process Control Information

**b) Thread:**

- A Thread can be thought as a basic unit of CPU utilization.
- It is also called as light-weight process.
- Multiple threads can be created by a particular process.
- Each thread shares code, data segments and open files of that process with other threads
- However, each of them has their separate register set values and stacks.

**c) Inter-process communication:**

Refer Topic-2

**d) Synchronization:**

Refer Topic-3

**e) Bootstrap:**

- Bootstrapping is used to initialize the operating system as required by the user.
- It is needed at startup of every system because there is no operating system when the system boots.
- Bootstrapping is carried out in four steps for X386-based systems
  - It performs POST (Power On Self Test)
  - It uses BIOS (Basic Input Output System)
  - It loads the data using boot loader
  - It finally loads the Operating Systems

**f) Time Management:**

- In Linux Systems, there are two different types of times.

*) Current Date and Time:

- All the systems including Linux based systems maintain the current date and time using RTC (Real Time Clock) device.
- It maintains the time even when the system is in off state.

*) Interval Timer:

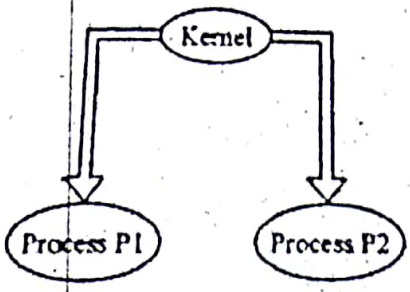
- All the computer systems also maintains a PIT (Programmable Interval Timer) which issues interrupt signals based on a specific frequency set by the user.
- The default value for linux system is 1 ms.
- For every PIT execution, the kernel performs
  - Incrementing of clock
  - Incrementing of current date and time
  - Checking of failed or expired timers
  - Checking of active timer with its expiry time

g) Kernel Module:

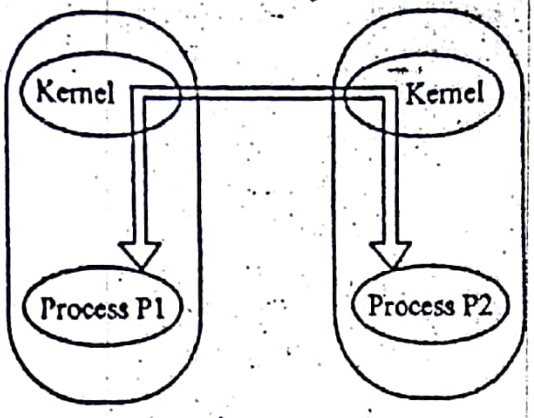
- Kernel Modules are used for module management in Linux systems.
- Examples of kernel modules are,
  - File systems
  - Device drivers
  - Network protocols etc
- In Linux the kernel module can be brought into action using insmod command.
- It is executed only after resolving all the pending requests present in the module program.
- However, various modules can be loaded and unloaded automatically.

## 2. Interprocess Communication (IPC):

- Inter Process Communication (IPC) is a mechanism that involves communication of one process with another process.
- This usually occurs only in one system.
- Communication can be of two types –
  - Between related processes initiating from only one process, such as parent and child processes.
  - Between unrelated processes, or two or more different processes.
- The design goal of IPC is to allow processes to share data among them, without interfering with one another.
- The IPC between two processes on a single computer is depicted as,




- The IPC between two processes on two different computers is depicted as,




**Methods of IPC:**

- The different methods of IPC are,

a) Signals
b) Pipes
c) FIFOs
d) Semaphores
e) Message Queues
f) Shared Memory

**Signals:**

- Signal is a mechanism to communication between multiple processes by way of signaling.
- This means a source process will send a signal (recognized by number) and the destination process will handle it accordingly.

**Pipes:**

- Communication between two related processes.
- The mechanism is half duplex meaning the first process communicates with the second process.
- To achieve a full duplex i.e., for the second process to communicate with the first process another pipe is required.

**FIFOs:**

- Communication between two unrelated processes.
- FIFO is a full duplex, meaning the first process can communicate with the second process and vice versa at the same time.

**Semaphores:**

- Semaphores are meant for synchronizing access to multiple processes.
- When one process wants to access the memory (for reading or writing), it needs to be locked (or protected) and released when the access is removed.
- This needs to be repeated by all the processes to secure data.

**Message Queues:**

- Communication between two or more processes with full duplex capacity.
- The processes will communicate with each other by posting a message and retrieving it out of the queue.
- Once retrieved, the message is no longer available in the queue.

**Shared Memory:**

- Communication between two or more processes is achieved through a shared piece of memory among all processes.

- The shared memory needs to be protected from each other by synchronizing access to all the processes.

## 3. Synchronization:

- Synchronization is used to ensure that the behavior of resource is in accordance with the expected behavior in a shared resource environment.
- Linux systems allow multiple kernel paths to access multiple data structures at the same time.
- The results generated from these executions are based on the way of interleaving their operations.
- To ensure appropriate behavior, these executions need to be controlled.
- The schemes used for performing synchronization in Linux systems are
    a) Interrupt Disabling
    b) Atomic Operations
    c) Kernel Semaphore
    d) Spinlock
    e) Seqlock
    f) Disabling Kernel Preemption
    g) Kernel Path Synchronization
    h) Deadlock Handling

### a) Interrupt Disabling:

- This scheme is used to avoid race conditions that can be occur in a shared resource environment.
- In this scheme, the interrupt circuit is disabled when the processor is about to execute critical section.
- When the execution is completed, the circuit is enabled.
- During this, the non-maskable interrupts are ignored and are not disabled.
- Instead, they are allocated to CPU.
- In X386 systems, cli instruction is used for disabling the interrupts whereas sti instruction is used for enabling it.

### b) Atomic Operations:

- In this scheme, certain atomic operations such as test-and-set, read-modify-write are used in order to avoid the race conditions.
- This typically happens when there exist trivial memory cell or integer variable in the critical section.
- In Linux based systems, atomic_t variable used and can be accessed by various operations.
- These operations are implemented with X386 systems.
- In these systems, the following instructions are considered as memory barriers.
    - I/O port instructions
    - Lock byte prefixed instructions
    - Instructions that perform write operations on registers such as debug, system and control
    - Iret instructions

### c) Kernel Semaphore:

- In this scheme, the kernel paths are synchronized

- The kernel semaphores acts as objects whose type is struct semaphore.
- Linux uses up and down atomic operations which are used before and after the execution of critical section respectively.
- The services are served in FIFO order.
- The Linux also uses read/ write semaphores which are used to improve concurrent execution.
- The atomic operations used for this purpose include down_read, down_write, up_read, up_write.

d) Spinlock:

- This scheme is preferred in multiprocessor systems where semaphore-based synchronization cannot be used.
- This is because, the semaphore-based synchronization fails when two CPUs accesses the kernel data structures concurrently where one is kept in waiting and another one has completed its critical section execution.
- In this scheme, the waiting queue is removed and the down operation iteratively checks the value of integer until a spin lock is detected.
- At this position, the preemption is disabled to avoid freezing of the entire system.
- This scheme is preferred only for multi processor systems.

e) Seqlock:

- This scheme is similar to read/ write spinlock.
- In this scheme, higher pricrity is allocated to the write operations thereby allowing them to enter the critical section even if the users are actively executing critical section.
- When writes collide with read operations, the read operations need to be performed again.

f) Disabling Kernel Preemption:

- In this scheme, the kernel disables preemption in the situations where kernel executions cannot be preempted.
- An alternative to this method is to interrupt the process which is currently executing the kernel use exceptions or interrupt.
- However this process can lead to race conditions among all the kernel paths associated with a single process.

g) Kernel Path Synchronization:

- This scheme considers kernel paths as separate agents so that they can be easily synchronized.
- Once the execution of kernel is started with a kernel path, it cannot be interrupted until the critical section is completed.
- Due to this, another execution with same critical path cannot be started.
- To allow this, local interrupts need to be disable along with a spinlock.

h) Deadlock Handling:

- Linux does not support automatic dead lock detection and prevention.
- Therefore, it is the responsibility of kernel developers to implement the synchronization process is such a way that it avoids deadlock.

## 4. Interrupt:

- When the interrupt handler is in a running state, the interrupt signal corresponding to the interrupt vector gets disable for a specific period of time.
- Also there are chances that an interrupt serviced by a process is not accessed for the process.
- We observe that interrupt handler is allowed to run in arbitrary process context.
- At the same time these handlers are not entitled to carry out any sort of blocking operation or process-specific operation.
- In order to accelerate the device utilization, the speed of certain entities of interrupt service is fasten up such as critical service parts are made faster while the execution of non-critical parts is delayed.
- For instance, when the packet arrives across the Ethernet, it is received by the critical part while the deferrable part simply analyzes the packet and proceeds further.
- According to Linux, an interrupt service is divided into two parts.
    i) Top Half which is executed by the Kernel immediately
    ii) Bottom Half which is executed later

i) Top Half which is executed by the Kernel immediately:

- This part is divided into two parts
- Critical Part
- Non-critical Part

Critical Part:

- It is executed while INTR interrupts disabled.

Non-critical Part:

- It is executed while INTR interrupts enabled.

- Some of the jobs performed by this part are,
    - Acknowledging the PC
    - Updation of data structures used by interrupting device and CPU

ii) Bottom Half which is executed later:

- Here interrupts are enabled and kernel maintains interrupt service routines in a form of queue.
- The execution of bottom half is not interrupted by other bottom half.
- However, it gets interrupted by any additional top half.
- Moreover the Linux 2.6 improves concurrency in multiprocessor systems by using various types of deferrable tasks which includes,
    - Softirg
    - Tasklet
- Both are executed in a serial manner on a single CPU.

Softirq:

- There are 32 softirqs which are allocated statically.
- Ideally, only 6 are used.
- They can run concurrently on multiple CPUs.

Tasklet:

- They cannot run concurrently on multiple CPUs.

- However, tasklets of different types can run.

## 5. Exception:

- The mechanism for address space switching is analogous for both interrupt as well as exception.
- The approach of kernel to both interrupt and exception is different.
- In essence, the system call in Linux is taken as exception (software interrupt)
- The kernel uses trap gate descriptors to connect exception.
- The structure of exception handlers is as follows,
  - Storage of registers on stack of kernel
  - Handling of exception in C function
  - Execution of ref_from_exception function
- On the other hand, the Linux categorizes the exceptions into the following,
  - Errors in program (division by zero)
  - Page fault, which is a kernel design choice
- Ideally, the services offered by exceptions are very fast in the category of division by zero.
- It operates by sending a POSIX signal to the process in a running state.
- While in the later case, Linux retrieves the process from the exception.

## 6. System Call;

- An application process attains the requested services provided by the kernel by means of invoking system calls.
- A software interrupt is a type of interrupt accountable for initiating system call in X386 microprocessor and it is caused by an exception of special instruction "int OX80" present in the instruction set.
- Each and every system call need to be transmitted or passed via an interrupt vector OX80.
- Subsequently, a process need to perform the following actions in order to make a system call.
  - Initiating registers so as to pass arguments to the system call
  - Triggering the software interrupt
- For instance, arguments like identifying the call type and system call number are passed to the system call via EAX register.
- On each occurrence of software interrupt, the processor is not only responsible for altering mode of the operation to kernel but, also responsible for reading the starting address of generic system call handler system_call that stores in trap gate descriptor.
- After reading the address, the processor jumps to the specific address and sets the segment selector in CS, DS, ES and SS registers as follows.

| | | |
|---|---|---|
| Code Segment (CS) Register | : | The segment selector points to kernel mode |
| Data Segment (DS) Register | : | The segment selector points to kernel DS |
| Extra Segment (ES) Register | : | The segment selector points to kernel DS |
| Stack Segment (SS) Register | : | The segment selector points to user segment |

# PART-II: ANDROID SOFTWARE PLATFORM:

## 7. Android:

- Android is a complete set of software for mobile devices such as tablet computers, notebooks, smartphones, electronic book readers, set-top boxes etc.
- It contains a linux-based Operating System, middleware and key mobile applications.
- It can be thought of as a mobile operating system.
- But it is not limited to mobile only. It is currently used in various devices such as mobiles, tablets, televisions etc.
- Android is a software package and linux based operating system for mobile devices such as tablet computers and smartphones.
- It is developed by Google and later the OHA (Open Handset Alliance).
- Java language is mainly used to write the android code even though other languages can be used.
- The goal of android project is to create a successful real-world product that improves the mobile experience for end users.

Android Versions:

| VERSION | CODE NAME |
|---|---|
| 1.5 | Cupcake |
| 1.6 | Donut |
| 2.1 | Eclair |
| 2.2 | Froyo |
| 2.3 | Gingerbread |
| 3.1 and 3.3 | Honeycomb |
| 4.0 | Ice Cream Sandwitch |
| 4.1, 4.2 and 4.3 | Jelly Bean |
| 4.4 | KitKat |
| 5.0 | Lollipop |
| 6.0 | Marshmallow |
| 7.0 | Nougat |
| 8.0 | Oreo |

Features of Android:

i) Storage:

- Android uses SQLite for storing the data.
- SQLite is a light-weight relational database.

ii) Connectivity:

- Android supports various connections such as,
  - GSM/EDGE
  - Bluetooth
  - CDMA
  - UMTS
  - EV-DO
  - WiMax
  - WiFi
  - LTE

iii) Multi-touch:

- It provides multi-touch screens.

iv) Messaging:

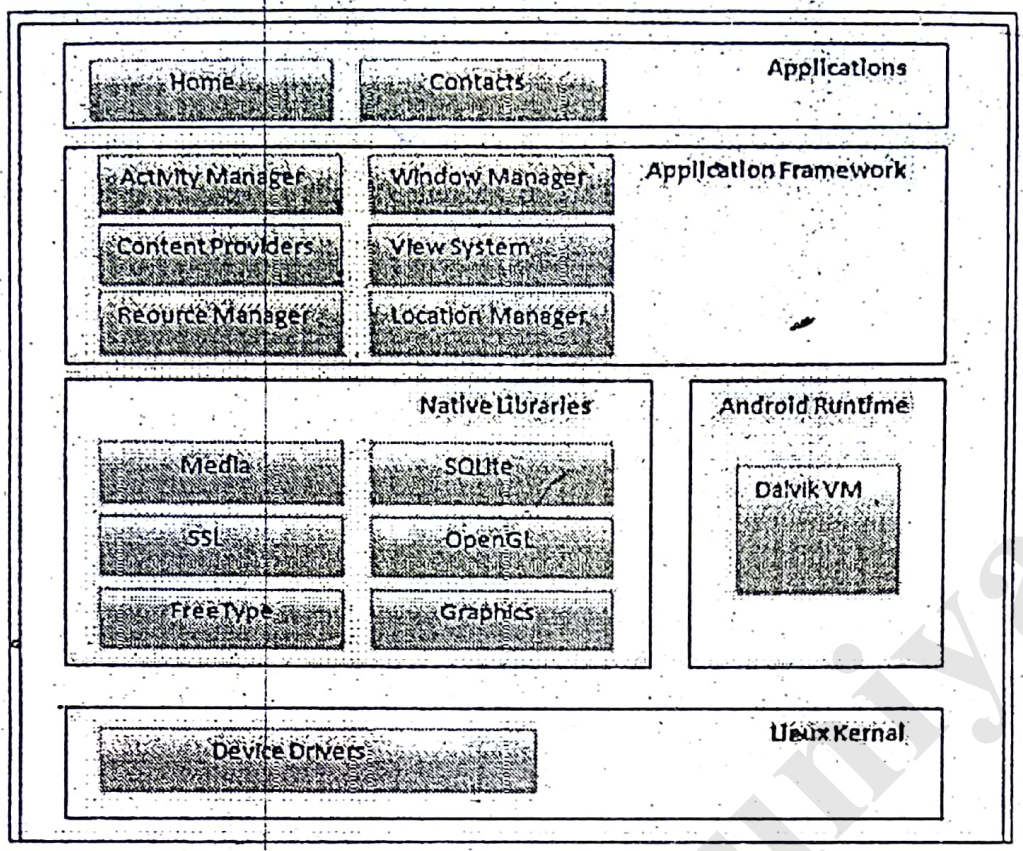
- Android supports SMS for text messages and MMS for video messaging

v) Flash:

- Android provides support for flash by using flash 10.1 for version 2.3
- The Other features are
  - Browser
  - Multi-tasking
  - Tethering (hotspot)
  - Camera
  - GPS
  - Digital Compass
  - Proximity Sensor
  - Accelerometer Sensor

## 8. Android Architecture:

- Android architecture or Android software stack is categorized into five parts:
  a) Linux kernel
  b) native libraries (middleware),
  c) Android Runtime
  d) Application Framework
  e) Applications
- Android architecture is depicted as,




**a) Linux kernel:**

- It is the heart of android architecture that exists at the root of android architecture.
- Linux kernel is responsible for device drivers, power management, memory management, device management and resource access.

**b) Native Libraries:**

- On the top of linux kernel, their are Native libraries such as WebKit, OpenGL, FreeType, SQLite, Media, C runtime library (libc) etc.
- The WebKit library is responsible for browser support, SQLite is for database, FreeType for font support, Media for playing and recording audio and video formats.

**c) Android Runtime:**

- In android runtime, there are core libraries and DVM (Dalvik Virtual Machine) which is responsible to run android application.
- DVM is like JVM but it is optimized for mobile devices.
- It consumes less memory and provides fast performance.

**d) Android Framework:**

- On the top of Native libraries and android runtime, there is android framework.
- Android framework includes Android API's such as UI (User Interface), telephony, resources, locations, Content Providers (data) and package managers.
- It provides a lot of classes and interfaces for android application development.

**e) Applications:**

- On the top of android framework, there are applications.
- All applications such as home, contact, settings, games, browsers are using android framework that uses android runtime and libraries.
- Android runtime and native libraries are using linux kernal.

## 8. Operating System Services:

- In Android, the operating system services are,
  - i) Memory Management
  - ii) Process/ Thread Management
  - iii) Power Management
  - iv) Network
  - v) Security
  - vi) Driver Management
  - vii) Interprocess communication

## 9. Android Runtime:

- The core part of android software platform is a runtime system.
- It comprises two main concepts, i.e., first is Dalvik VM and second are core libraries corresponding to the features of java programming language sustained by android.
- We observe that, every single android applications runs as a VM with in an individual local process.
- Also, an instance of Dalvik VM is present with in the process address space.
- One or many Dalvik VMs can be supported by android platform effectively just by sharing the code and data
- The java applications are then converted into Dalvik executable .exe format files, the translation of java application file into class file is done using standard java compilers.
- This java .class files are are converted into .dex files using dx tool.
- The Dalvik VM consists of platform independent byte code, where java byte code from .class files are converted into Dalvik byte code in .dex files.
- The Dalvik VM uses the Linux for implementing security and protection through uid (user identifier) and gid(group identifier)

## 10. Application Development:

- The android is not considered as application development platform.
- This implies that, the development of the application is done at some other place and subsequently installed in android platform for execution.
- Ideally the android has SDK, which provides tools and APIs required for creating android applications.
- A virtual mobile device called emulator is used for testing and debugging the applications.
- The guide for the development of the application can be obtained from http://developer/android.com/guide.
- This makes the development process easy and fast as large numbers of APIs are available across the site.
- Every single android application is a java application.
- They invoke android APIs to acquire services from the android platform.

ii) Process Lifecycle:

- In android, the process lifecycle states are,

Visible:

- In this state, the activity is visible on the screen but not foreground and also not in interaction with user.

Foreground:

- In this state, the activity is in interaction with user and present on top of the screen.

Service:

- In this state, a service element is run by the process.

Background:

- In this state, the activity is paused and not made visible to the user.

Empty:

- In this state, no activity is performed by the process.

iii) IPC:

Refer Topic-2

iv) Security:

- The resources of the system cannot be used by an application without the user permission (during installation).
- For instance, if a mobile application requires vibration and the user do not permit, then the application cannot access the vibration system.

## Short Answer Questions:

1. What is Inter-process Communication? List out different methods of IPC.

2. Write short notes on Spinlock and Seqlock.

3. Discuss about Operating System Services.

4. Explain about atomic operations.

5. Discuss about the classification of X386 interrupt descriptors

X386 interrupt descriptors types:

- These are 3 types
  - i) Task Gate Descriptor
  - ii) Interrupt Gate Descriptor
  - iii) Trap Gate Descriptor

i) Task Gate Descriptor:

- This type of class includes TSS selector which runs during interruptions.

ii) Interrupt Gate Descriptor:

- This type of class includes a segment selector and offset for interrupt handler.
- Apart from this, IF bit in EFLAGS is processed by the processor for disabling INTR.

iii) Trap Gate Descriptor:

- This type of class is same as that of interrupt gate descriptor but with a difference is that instead of processing the IF bit it remains fixed.

6. Discuss about application development

7. What are the different types of elements used in application structure?

8. Discuss about operations performed by activity management

9. Explain different states of process life cycle

10. Write short notes on exception handling in Linux

## Long Answer Questions:

1. Give a brief overview on Linux operating system and Discuss the different components of Linux.

2. Explain about IPC in Linux.

3. Explain about Synchronization supported by Linux

4. Explain about a) Interrupt b) Exception c) System call

5. What is android? Discuss its versions and features.

6. Discuss in detail about architecture of android software platform

7. Explain about a) Android Runtime b) Application Development c) Application Structure

8. Explain about application process management.

## 11. Application Structure:

- The applications of an android platform are capable of providing its services to other applications by sharing the elements.
- This can be achieved when the system initiates the requested application and the requested element in a process.
- However the application that is intended for using the services must be permitted by the application that is providing the service.
- The application constitutes of numerous entry points and one or more elements of the following types,
  - Service Element
  - Activity Element
  - Content provider Element
  - Broadcast Receiver Element

Service Element:

- It can be defined as a task which runs in background for unlimited duration in presence of other activities.
- For example, a user can play a song in background while doing other activities.
- Using the android.app.service class, service object can be instantiated.

Activity Element:

- It is a visual user interface through a window.
- The view object specifies visual content corresponding to a window, such that every viw manages a rectangular area within the window.

Content provider Element:

- This type of element assists other applications to make use of data from their own application.
- The class of this element extends the class android.content.contentprovider.

Broadcast Receiver Element:

- This element as the name implies, receives the broadcast messages and respond to them correspondingly.
- The typical response includes notifying the user, initiating the activity.
- The object for this element is instantiated from the class android.content.BroadCastReceiver.

## 12. Application Process Management:

- Application process management of an android has 4 phases
  - i) Activity Management
  - ii) Process Lifecycle
  - iii) IPC
  - iv) Security

**i) Activity Management:**

- An application possesses in different activities where every activity is an object of the class android.app.Activity.
- This object consists of hooks for the following operations,

Create:

- This deals with the creation of an activity.

- This is done by calling the onCreate( ) hook.

Start:
- This deals with the initialization of activity.
- It is done by calling the onStart( ) hook.

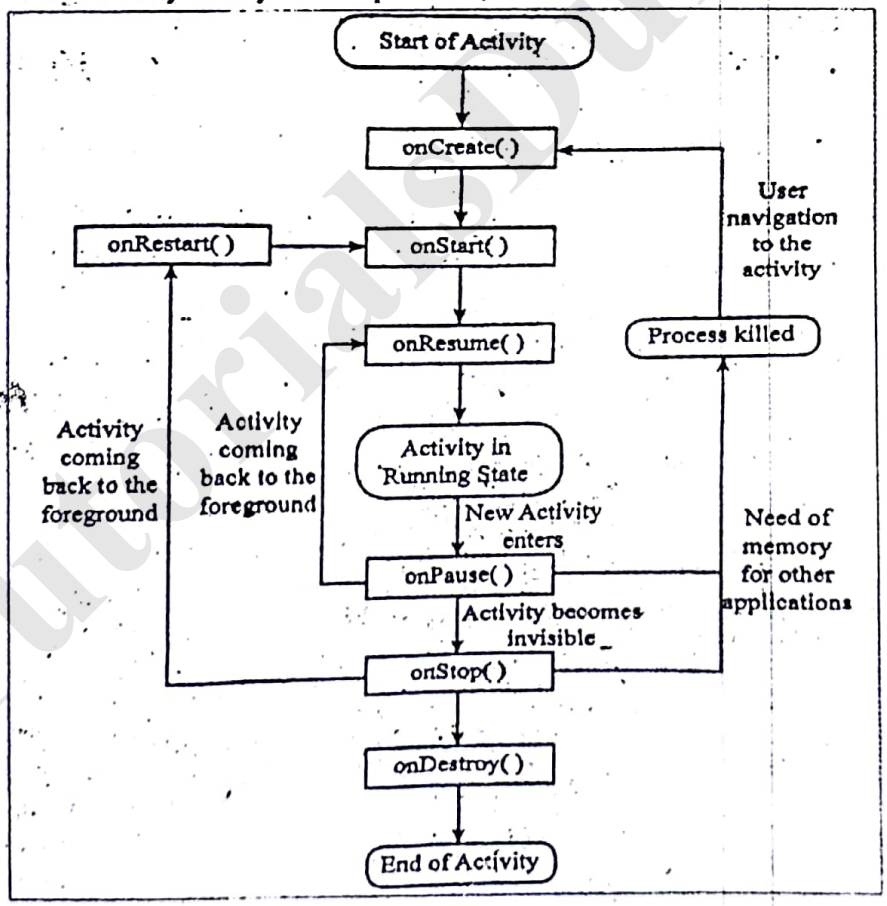
Resume:
- This deals with the activity interacting with its user.
- It is carried out by calling the onResume( ) hook.

Pause:
- This deals with stopping an activity for temporary purpose and resuming some other activity for completion.
- It is carried out by calling the onPause( ) hook

Stop:
- This deals with the activity becoming invisible to the user.
- It is carried out by calling the onStop( ) hook.

Destroy:
- This deals with the activity being destroyed either manually or automatically by the system.
- It is carried out by calling the onDestroy( ) hook.

Restart:
- This deals with the activity getting restarted after it was stopped.
- It is carried out by calling the onRestart( ) hook.
- The Activity Lifecycle is depicted as,



# TutorialsDuniya.com

**Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com**

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

**Please Share these Notes with your Friends as well**