



Operating System (3140702) Handbook

Operating System (Gujarat Technological University)



Scan to open on Studocu

Lecture note

Operating System (3140702)

Year: 2019-2020

UNIT 1:Introduction

What is Operating System?

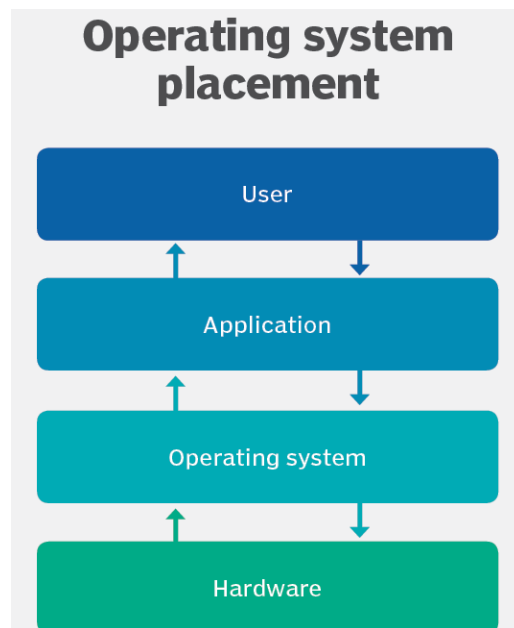
An **operating system** or **OS** is a software program that enables the computer hardware to communicate and operate with the computer software.

An operating system (OS), in its most general sense, is software that allows a user to run other applications on a computing device. While it is possible for a software application to interface directly with hardware, the vast majority of applications are written for an OS, which allows them to take advantage of common libraries and not worry about specific hardware details.

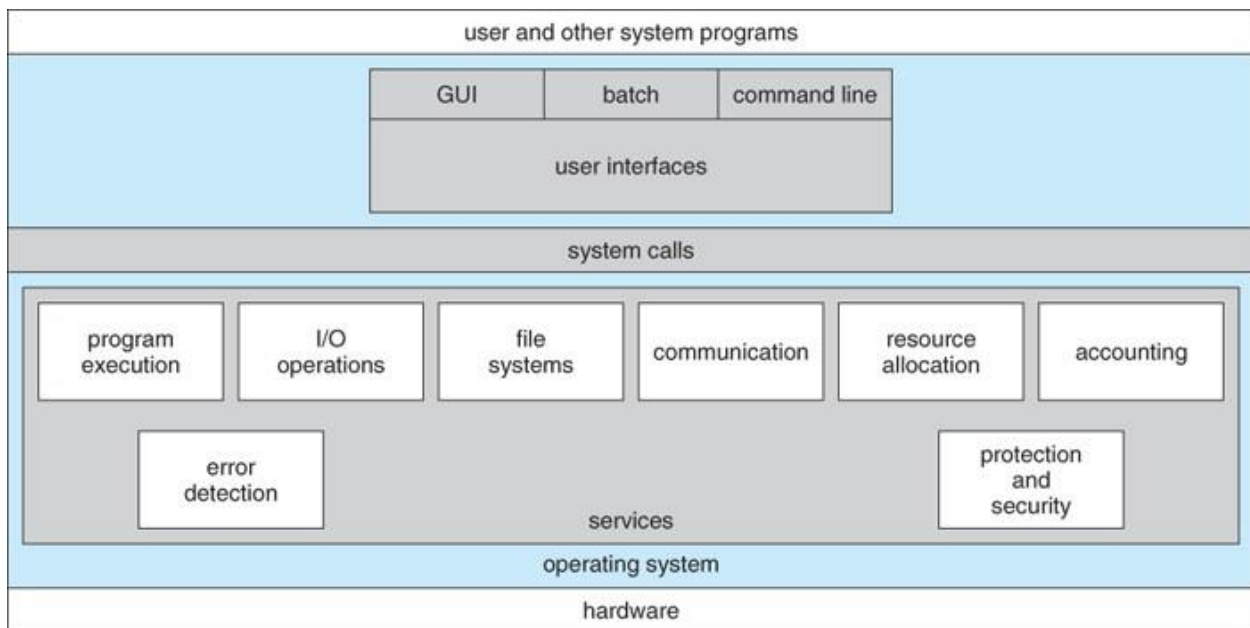
The operating system manages a computer's hardware resources, including:

- Input devices such as a keyboard and mouse.
- Output devices such as display monitors, printers and scanners.
- Network devices such as modems, routers and network connections.
- Storage devices such as internal and external drives.

The OS also provides services to facilitate the efficient execution and management of, and memory allocations for, any additional installed software application programs.



Operating System Services



An operating system is an interface which provides services to both the user and to the programs. It provides an environment for the program to execute. It also provides users with the services of how to execute programs in a convenient manner. The operating system provides some services to program and also to the users of those programs. The specific services provided by the OS are of course different.

Following are the common services provided by an operating system:

1. Program execution
2. I/O operations
3. File system manipulation
4. Communication
5. Error detection
6. Resource allocation
7. Protection

1) Program Execution

- An operating system must be able to load many kinds of activities into the memory and to run it. The program must be able to end its execution, either normally or abnormally.

- A process includes the complete execution of the written program or code. There are some of the activities which are performed by the operating system:
 - The operating system Loads program into memory
 - It also Executes the program
 - It Handles the program's execution
 - It Provides a mechanism for process synchronization
 - It Provides a mechanism for process communication

2) I/O Operations

- The communication between the user and devices drivers are managed by the operating system.
- I/O devices are required for any running process. In I/O a file or an I/O devices can be involved.
- I/O operations are the read or write operations which are done with the help of input-output devices.
- Operating system give the access to the I/O devices when it required.

3) File system manipulation

The collection of related information which represent some content is known as a file. The computer can store files on the secondary storage devices. For long-term storage purpose. examples of storage media include magnetic tape, magnetic disk and optical disk drives like CD, DVD.

- A file system is a collection of directories for easy understand and usage. These directories contain some files. There are some major activities which are performed by an operating system with respect to file management.
- The operating system gives an access to the program for performing an operation on the file.
- Programs need to read and write a file.
- The user can create/delete a file by using an interface provided by the operating system.
- The operating system provides an interface to the user creates/ delete directories.
- The backup of the file system can be created by using an interface provided by the operating system.

4) Communication

In the computer system, there is a collection of processors which do not share memory peripherals devices or a clock, the operating system manages communication between all the processes. Multiple processes can communicate with every process through communication lines in the network. There are some major activities that are carried by an operating system with respect to communication.

- Two processes may require data to be transferred between the process.
- Both the processes can be on one computer or a different computer, but are connected through a computer network.

5) Error handling

An error is one part of the system that may cause malfunctioning of the complete system. The operating system constantly monitors the system for detecting errors to avoid some situations. This gives relief to the user of the worry of getting an error in the various parts of the system causing malfunctioning.

The error can occur anytime and anywhere. The error may occur anywhere in the computer system like in CPU, in I/O devices or in the memory hardware. There are some activities that are performed by an operating system:

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

6) Resource management

When there are multiple users or multiple jobs running at the same time resources must be allocated to each of them. There are some major activities that are performed by an operating system:

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

7) Protection

The owners of information stored in a multi-user computer system want to control its use. When several disjoint processes execute concurrently it should not be possible for any process to interfere with another process. Every process in the computer system must be secured and controlled.

What is System call?

A system call

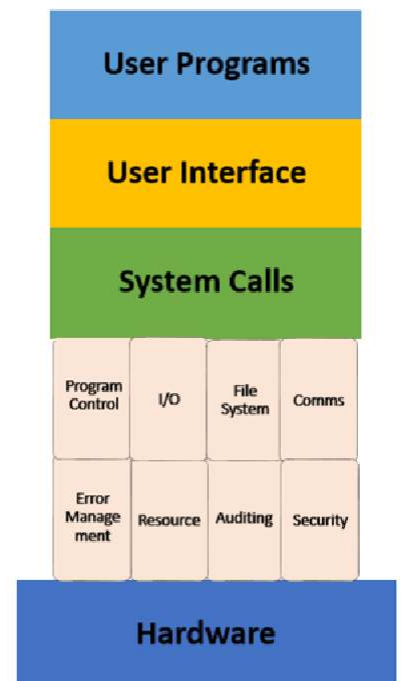
is a mechanism that provides the interface between a process and the operating system. It is a programmatic method in which a computer program requests a service from the kernel of the OS.

System call offers the services of the operating system to the user programs via API (Application Programming Interface). System calls are the only entry points for the kernel system. Example of System Call

For example if we need to write a program code to read data from one file, copy that data into another file. The first information that the program requires is the name of the two files, the input and output files.

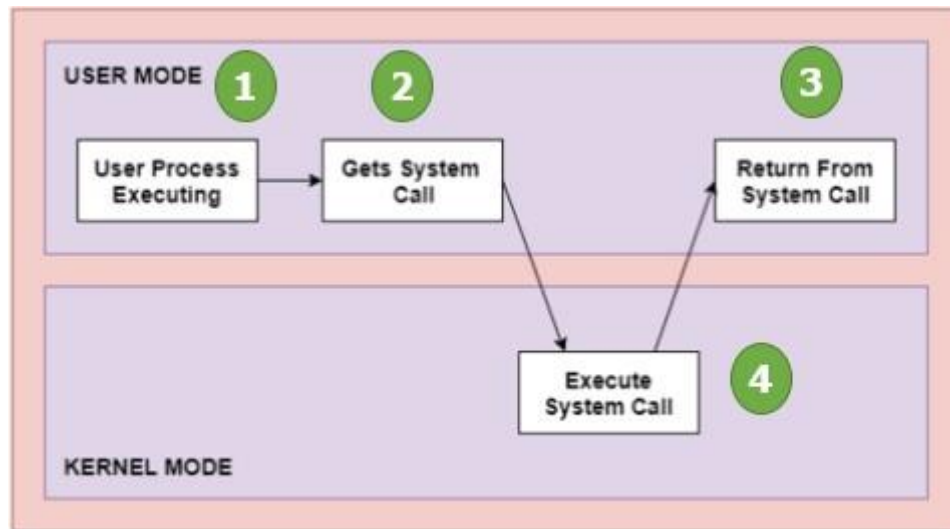
In an interactive system, this type of program execution requires some system calls by OS.

- First call is to write a prompting message on the screen
- Second, to read from the keyboard, the characters which define the two files.



Prepared By: Prof. Neha Prajapati

How System Call Works?



As you can see in the above-given diagram.

Step 1) The processes executed in the user mode till the time a system call interrupts it.

Step 2) After that, the system call is executed in the kernel-mode on a priority basis.

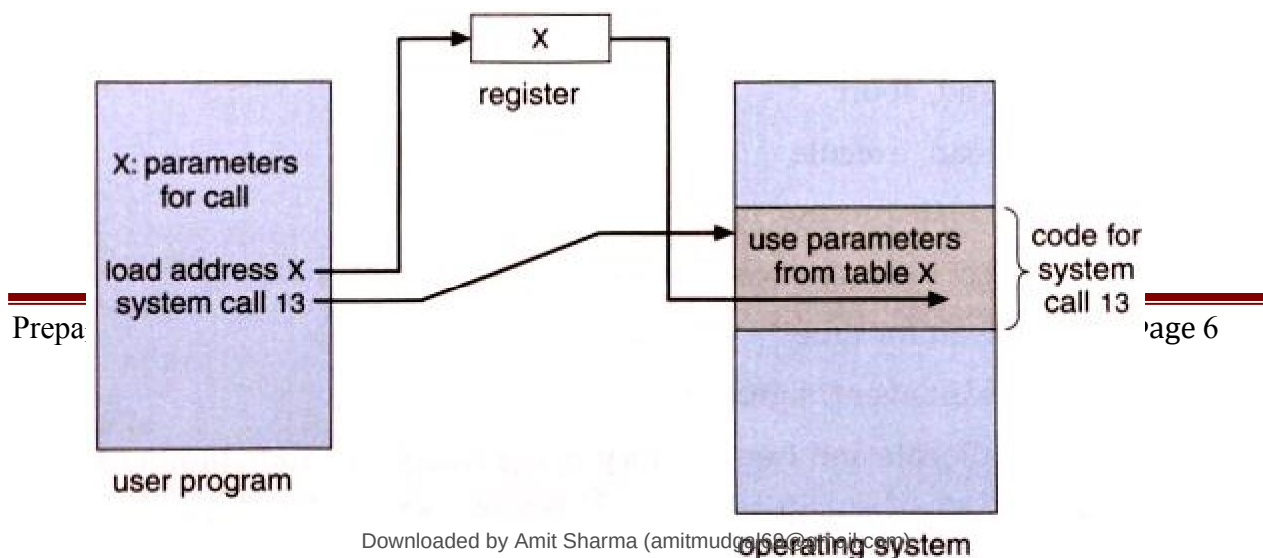
Step 3) Once system call execution is over, control returns to the user mode.,

Step 4) The execution of user processes resumed in Kernel mode.

System Call Parameters

Three general methods exist for passing parameters to the OS:

1. Parameters can be passed in registers.
2. When there are more parameters than registers, parameters can be stored in a block and the block address can be passed as a parameter to a register.
3. Parameters can also be pushed on or popped off the stack by the operating system.



Why do you need System Calls in OS?

Following are situations which need system calls in OS:

- Reading and writing from files demand system calls.
- If a file system wants to create or delete files, system calls are required.
- System calls are used for the creation and management of new processes.
- Network connections need system calls for sending and receiving packets.
- Access to hardware devices like scanner, printer, need a system call.

Types of System calls

Here are the five types of system calls used in OS:

- Process Control
- File Management
- Device Management
- Information Maintenance
- Communications

Process Control

This system calls perform the task of process creation, process termination, etc.

Functions:

- End and Abort
- Load and Execute
- Create Process and Terminate Process
- Wait and Signed Event
- Allocate and free memory

File Management

File management system calls handle file manipulation jobs like creating a file, reading, and writing, etc.

Functions:

- Create a file
- Delete file
- Open and close file
- Read, write, and reposition
- Get and set file attributes

Device Management

Device management does the job of device manipulation like reading from device buffers, writing into device buffers, etc.

Functions

- Request and release device
- Logically attach/ detach devices
- Get and Set device attributes

Information Maintenance

It handles information and its transfer between the OS and the user program.

Functions:

- Get or set time and date
- Get process and device attributes

Communication:

These types of system calls are specially used for interprocess communications.

Functions:

- Create, delete communications connections
- Send, receive message
- Help OS to transfer status information
- Attach or detach remote devices

Rules for passing Parameters for System Call

Here are general common rules for passing parameters to the System Call:

- Parameters should be pushed on or popped off the stack by the operating system.
- Parameters can be passed in registers.
- When there are more parameters than registers, it should be stored in a block, and the block address should be passed as a parameter to a register.

Important System Calls Used in OS

wait()

In some systems, a process needs to wait for another process to complete its execution. This type of situation occurs when a parent process creates a child process, and the execution of the parent process remains suspended until its child process executes.

The suspension of the parent process automatically occurs with a wait() system call. When the child process ends execution, the control moves back to the parent process.

fork()

Processes use this system call to create processes that are a copy of themselves. With the help of this system Call parent process creates a child process, and the execution of the parent process will be suspended till the child process executes.

exec()

This system call runs when an executable file in the context of an already running process that replaces the older executable file. However, the original process identifier remains as a new process is not built, but stack, data, heap, data, etc. are replaced by the new process.

kill():

The kill() system call is used by OS to send a termination signal to a process that urges the process to exit. However, a kill system call does not necessarily mean killing the process and can have various meanings.

exit():

The exit() system call is used to terminate program execution. Specially in the multi-threaded environment, this call defines that the thread execution is complete. The OS reclaims resources that were used by the process after the use of exit() system call.

Categories	Windows	Unix
Process control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
Device manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
File manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	Open() Read() write() close()
Information maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	Pipe() shm_open() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup ()	Chmod() Umask() Chown()

Views Of Operating System

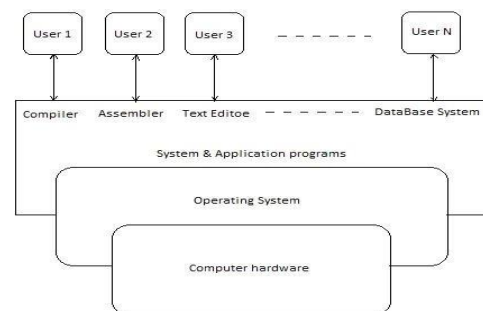
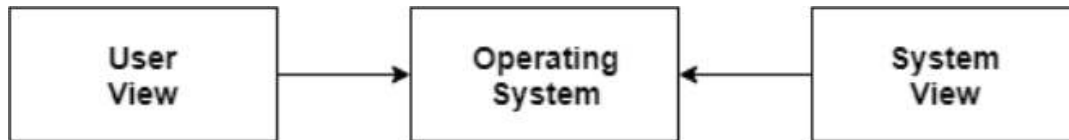


Figure 1: Abstract View of the Components of Computer System

An operating system is a construct that allows the user application programs to interact with the system hardware. Operating system by itself does not provide any function but it provides an atmosphere in which different applications and programs can do useful work.

The operating system can be observed from the point of view of the user or the system. This is known as the user view and the system view respectively. More details about these are given as follows:



1.) User View

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

- If the user is using a personal computer, the operating system is largely designed to make the interaction easy. Some attention is also paid to the performance of the system, but there is no need for the operating system to worry about resource utilization. This is because the personal computer uses all the resources available and there is no sharing.
- If the user is using a system connected to a mainframe or a minicomputer, the operating system is largely concerned with resource utilization. This is because there may be multiple terminals connected to the mainframe and the operating system makes sure that all the resources such as CPU, memory, I/O devices etc. are divided uniformly between them.
- If the user is sitting on a workstation connected to other workstations through networks, then the operating system needs to focus on both individual usage of resources and sharing through the network. This happens because the workstation exclusively uses its own resources but it also needs to share files etc. with other workstations across the network.
- If the user is using a handheld computer such as a mobile, then the operating system handles the usability of the device including a few remote operations. The battery level of the device is also taken into account.

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

2.) System View

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

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

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

Objectives of Operating System

The three primary objectives of Operating System use.

1)Hiding The details of H/W

2)Allocating the Resources to Processes

3)Providing an Effective User Interface

(i) Hiding The details of H/W

Making a computer system convenient to use i.e. hides details of Hardware resources from the programmer and provides him with a convenient interface of using computer system.

It acts as an intermediary between hardware and software providing a high level interface to low level hardware and making it easier for the software to access the use of those resources.

Concept of abstraction allows an os to hide the H/W Details

There are various Reasons that brought the need abstraction

Firstly, the code required to manage the peripheral devices is not standardized; therefor OS provides device drivers as subroutines that perform the tasks on behalf of the program.

Secondly, the OS maintain the H/W abstraction to hide the communication between hardware and software.

Thirdly ,

the OS converts the computer h/w into different sections each of which contain different programs to execute the process accesses the H/W through abstraction.

(ii) Allocating the Resources to Processes(Manage Resources)

This involves performing such tasks as keeping track of who is using which resource, granting resource requests, accounting for resource usage, and mediating conflicting requests from different programs and users

(iii) Providing an Effective User Interface

To provide users a convenient interface to use the computer system.

Types Of Operating System

1) Distributed Operating System

The motivation behind developing distributed operating systems is the availability of powerful and inexpensive microprocessors and advances in communication technology.

These advancements in technology have made it possible to design and develop distributed systems comprising of many computers that are inter connected by communication networks. The main benefit of distributed systems is its low price/performance ratio.

Advantages Distributed Operating System

1. As there are multiple systems involved, user at one site can utilize the resources of systems at other sites for resource-intensive tasks.
2. Fast processing.
3. Less load on the Host Machine.

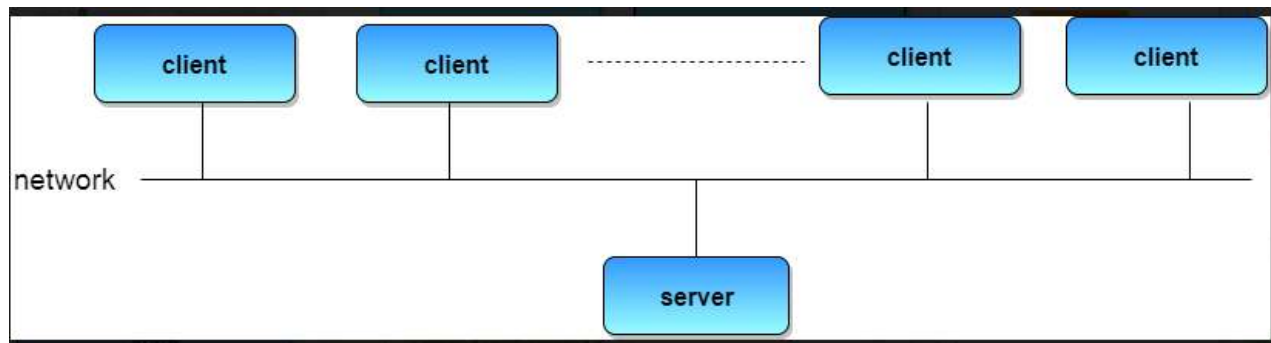
Types of Distributed Operating Systems

Following are the two types of distributed operating systems used:

1. Client-Server Systems
2. Peer-to-Peer Systems

Client-Server Systems

Centralized systems today act as **server systems** to satisfy requests generated by **client systems**. The general structure of a client-server system is depicted in the figure below:



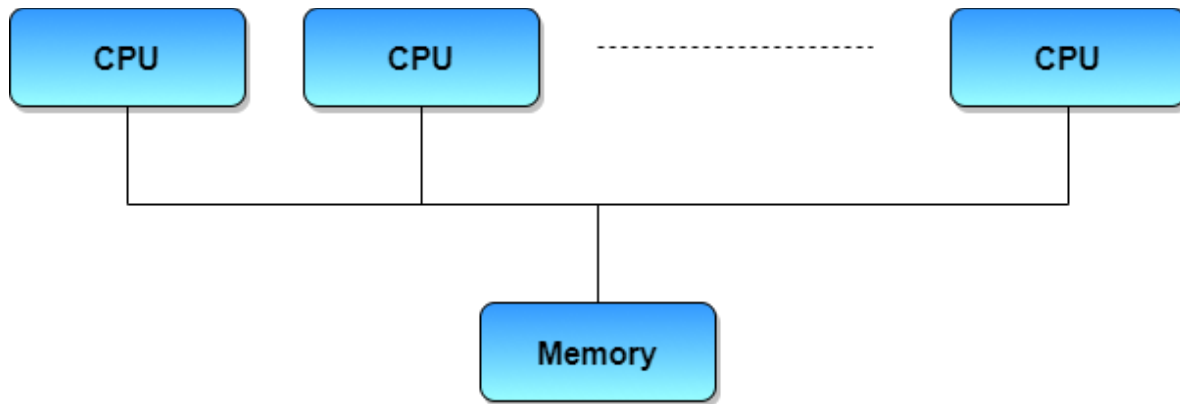
Server Systems can be broadly categorized as: **Compute Servers** and **File Servers**.

- **Compute Server systems**, provide an interface to which clients can send requests to perform an action, in response to which they execute the action and send back results to the client.
- **File Server systems**, provide a file-system interface where clients can create, update, read, and delete files.

Peer-to-Peer Systems

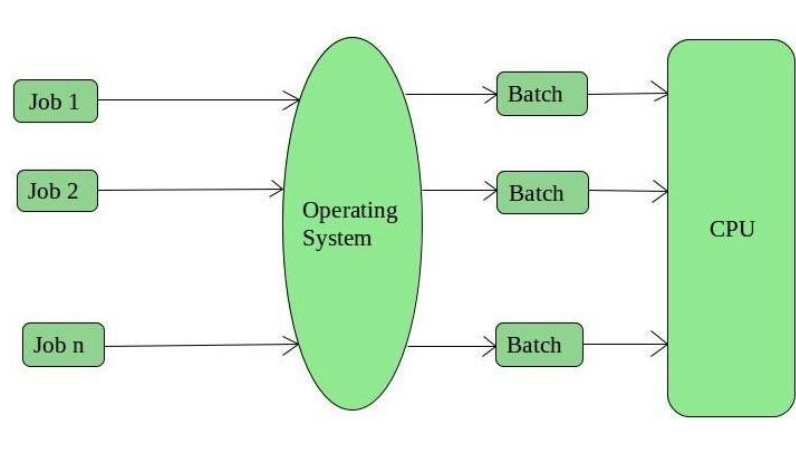
The growth of computer networks - especially the Internet and World Wide Web (WWW) – has had a profound influence on the recent development of operating systems. When PCs were introduced in the 1970s, they were designed for **personal** use and were generally considered standalone computers. With the beginning of widespread public use of the Internet in the 1990s for electronic mail and FTP, many PCs became connected to computer networks.

In contrast to the **Tightly Coupled** systems, the computer networks used in these applications consist of a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory. The processors communicate with one another through various communication lines, such as high-speed buses or telephone lines. These systems are usually referred to as loosely coupled systems (or distributed systems). The general structure of a client-server system is depicted in the figure below:



2)Batch Operating System –

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



Advantages of Batch Operating System:

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

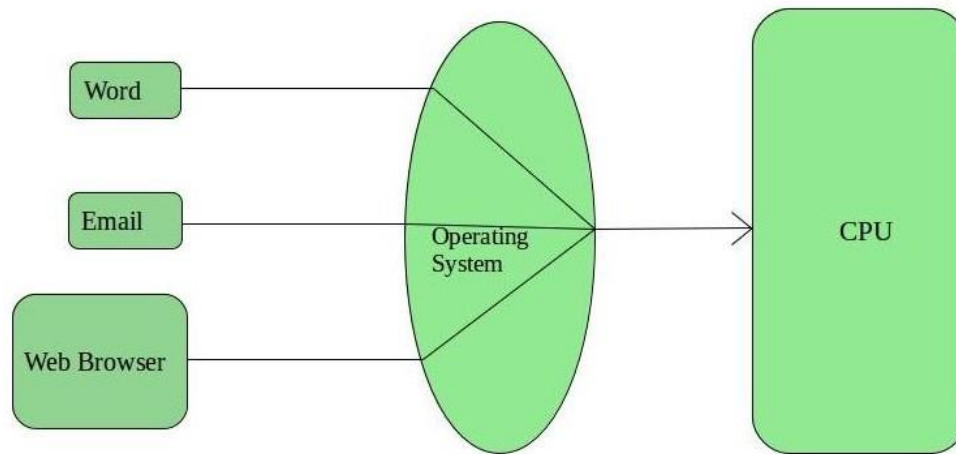
Disadvantages of Batch Operating System:

- The computer operators should be well known with batch systems
- Batch systems are hard to debug
- It is sometime costly
- The other jobs will have to wait for an unknown time if any job fails

Examples of Batch based Operating System: Payroll System, Bank Statements etc.

3)Time-Sharing Operating Systems –

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



Advantages of Time-Sharing OS:

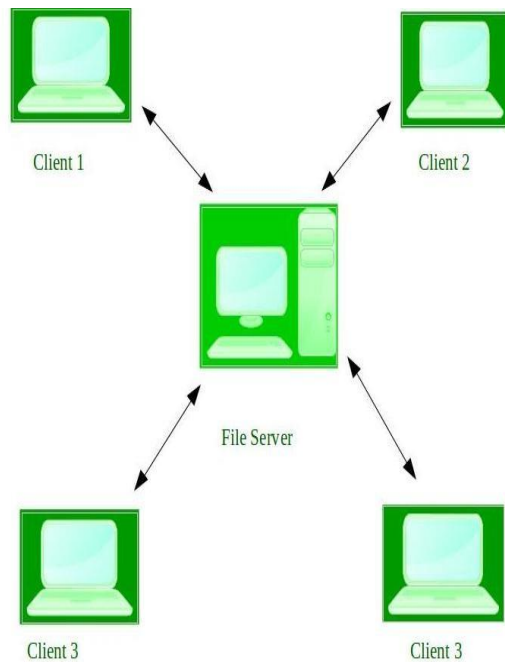
- Each task gets an equal opportunity
- Less chances of duplication of software
- CPU idle time can be reduced

Disadvantages of Time-Sharing OS:

- Reliability problem
- One must have to take care of security and integrity of user programs and data
- Data communication problem

4)Network Operating System –

These systems run on a server and provide the capability to manage data, users, groups, security, applications, and other networking functions. These type of operating systems allow shared access of files, printers, security, applications, and other networking functions over a small private network. One more important aspect of Network Operating Systems is that all the users are well aware of the underlying configuration, of all other users within the network, their individual connections etc. and that's why these computers are popularly known as **tightly coupled systems**.



Advantages of Network Operating System:

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

Disadvantages of Network Operating System:

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

Examples of Network Operating System are: Microsoft Windows Server 2003, Microsoft Windows Server 2008, UNIX, Linux, Mac OS X, Novell NetWare, and BSD etc.

5) Real-Time Operating System –

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

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

Two types of Real-Time Operating System which are as follows:

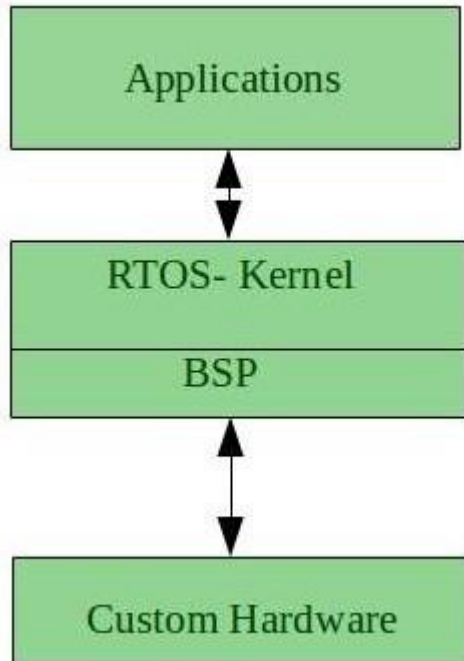
- **Hard Real-Time Systems:**

These OSs are meant for the applications where time constraints are very strict and even

the shortest possible delay is not acceptable. These systems are built for saving life like automatic parachutes or air bags which are required to be readily available in case of any accident. Virtual memory is almost never found in these systems.

- **Soft Real-Time Systems:**

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



Advantages of RTOS:

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

Disadvantages of RTOS:

- **Limited Tasks:** Very few tasks run at the same time and their concentration is very less on few applications to avoid errors.

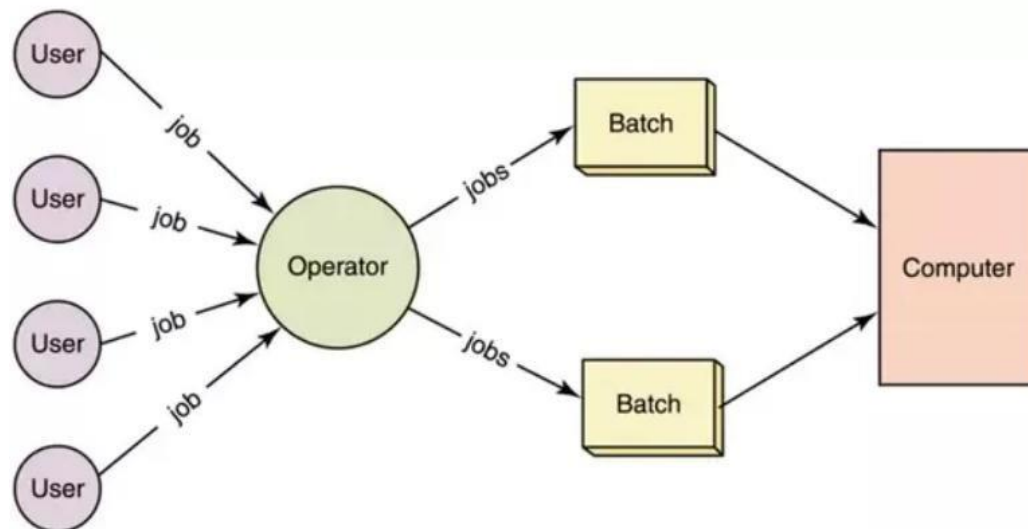
- **Use heavy system resources:** Sometimes the system resources are not so good and they are expensive as well.
- **Complex Algorithms:** The algorithms are very complex and difficult for the designer to write on.
- **Device driver and interrupt signals:** It needs specific device drivers and interrupt signals to response earliest to interrupts.
- **Thread Priority:** It is not good to set thread priority as these systems are very less prone to switching tasks.

Examples of Real-Time Operating Systems are: Scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots, air traffic control systems, etc.

6) Batch Operating System

In this, jobs which are of similar type are grouped together and treated as a **batch**. Now, they are stored on the Punch card (a stiff paper in which digital data is stored and represented using some specific sequence of holes) which will be submitted to the system for processing. The system will then perform all the required operations in a **sequence**. So, we consider this as a type of Serialprocessing.

Eg: Bank Statements



Advantages:

1. Suppose a job takes a very long time(1 day or so). Then, such processes can be performed even in the **absence** of humans.
2. They doesn't require any **special** hardware and system support to input data.

Disadvantages:

1. It is very difficult to **debug** batch systems.
2. Lack of **interaction** between user and operating system.

3. Suppose an error occurs in one of the jobs of a batch. Then, all the **remaining** jobs get affected i.e; they have to wait until the error is resolved.

7) Single Programming OS

Single program os only allow a single program to execute at a single point of time.

Program must be executed completely before the execution of the next program.

8)Multi User OS

A **multi-user operating system (OS)** is a computer system that allows multiple users that are on different computers to access a single system's OS resources simultaneously, as shown in Figure 1. Users on the system are connected through a network. The OS shares resources between users, depending on what type of resources the users need. The OS must ensure that the system stays well-balanced in resources to meet each user's needs and not affect other users who are connected. Some examples of a multi-user OS are Unix, Virtual Memory System (VMS) and mainframe OS.

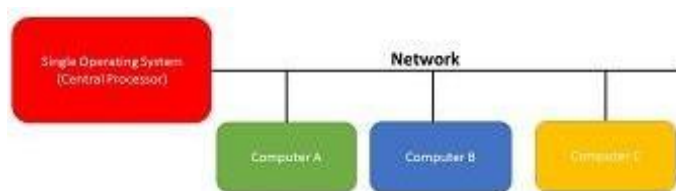


Figure 1 - Multi-user OS Handling Three Different Computers on the Network

Multi-user operating systems were originally used for time-sharing and batch processing on mainframe computers. These types of systems are still in use today by large companies, universities, and government agencies, and are usually used in servers, such as the Ubuntu Server edition (18.04.1 LTS) or Windows Server 2016. The server allows multiple users to access the same OS and share the hardware and the kernel, performing tasks for each user concurrently.

9) Multi Programming OS

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

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

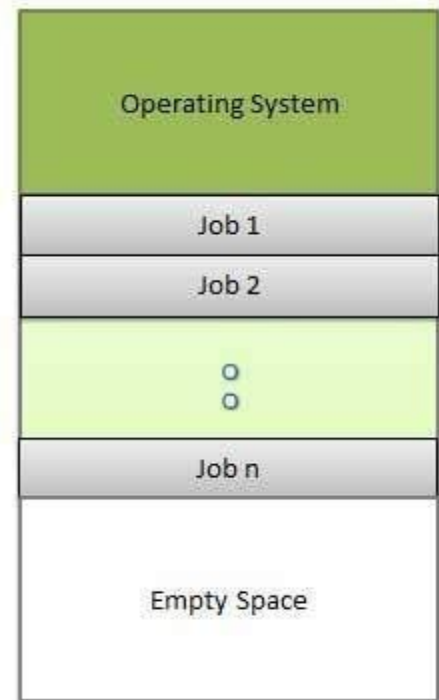
- An OS does the following activities related to multiprogramming. The operating system keeps several jobs in memory at a time.
- This set of jobs is a subset of the jobs kept in the job pool.
- The operating system picks and begins to execute one of the jobs in the memory.
- Multiprogramming operating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

Advantages

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

Disadvantages

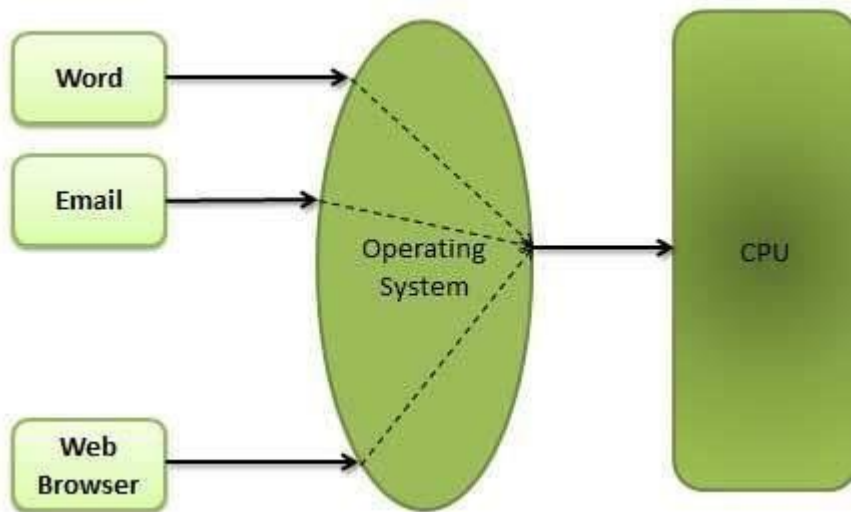
- CPU scheduling is required.
- To accommodate many jobs in memory, memory management is required.



10) Multitasking

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

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



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

11) Multi Processing OS

In a uni-processor system, only one process executes at a time. Multiprocessing is the use of two or more CPUs (processors) within a single Computer system. The term also refers to the ability of a system to support more than one processor within a single computer system. Now since there are multiple processors available, multiple processes can be executed at a time. These multi processors share the computer bus, sometimes the clock, memory and peripheral devices also.

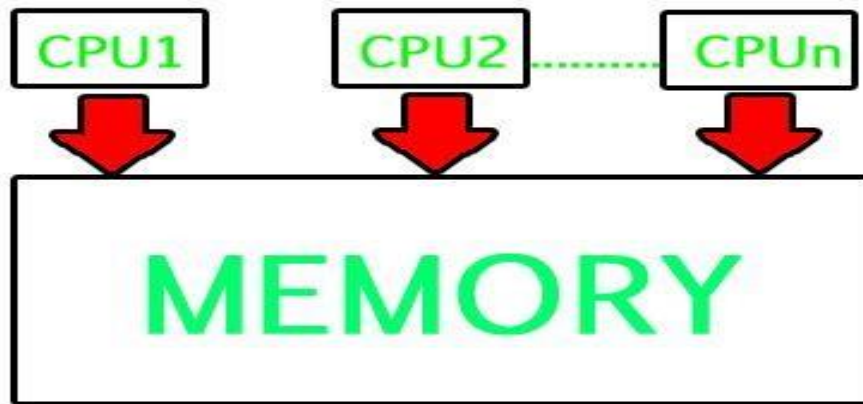
Multi processing system's working –

- With the help of multiprocessing, many processes can be executed simultaneously. Say processes P1, P2, P3 and P4 are waiting for execution. Now in a single processor system, firstly one process will execute, then the other, then the other and so on.
- But with multiprocessing, each process can be assigned to a different processor for its execution. If its a dual-core processor (2 processors), two processes can be executed

simultaneously and thus will be two times faster, similarly a quad core processor will be four times as fast as a single processor.

Why use multi processing –

- The main advantage of multiprocessor system is to get more work done in a shorter period of time. These types of systems are used when very high speed is required to process a large volume of data. Multi processing systems can save money in comparison to single processor systems because the processors can share peripherals and power supplies.
- It also provides increased reliability in the sense that if one processor fails, the work does not halt, it only slows down. e.g. if we have 10 processors and 1 fails, then the work does not halt, rather the remaining 9 processors can share the work of the 10th processor. Thus the whole system runs only 10 percent slower, rather than failing altogether.



Multiprocessing refers to the hardware (i.e., the CPU units) rather than the software (i.e., running processes). If the underlying hardware provides more than one processor then that is multiprocessing. It is the ability of the system to leverage multiple processors' computing power.

Difference between Multi programming and Multi processing –

- A System can be both multi programmed by having multiple programs running at the same time and multiprocessing by having more than one physical processor. The difference between multiprocessing and multi programming is that Multiprocessing is basically executing multiple processes at the same time on multiple processors, whereas multi programming is keeping several programs in main memory and executing them concurrently using a single CPU only.
- Multiprocessing occurs by means of parallel processing whereas Multi programming occurs by switching from one process to other (phenomenon called as context switching).

UNIT 2: PROCESS MANAGEMENT

1.0 Process Management

■ **After writing your program**

1. Save it

2. Compile Program

To check errors and to obtain machine understandable program(.exe)

3. Run the Program

To obtain result(output)

■ **Difference between Program and Processing**

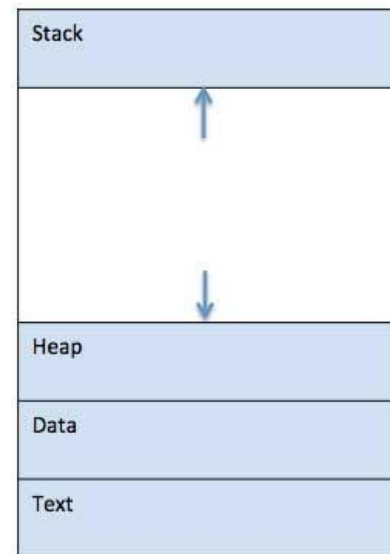
BASIS FOR COMPARISON	PROGRAM	PROCESS
Basic	Program is a set of instruction.	When a program is executed, it is known as process.
Nature	Passive	Active
Lifespan	Longer	Limited
Required resources	Program is stored on disk in some file and does not require any other resources.	Process holds resources such as CPU, memory address, disk, I/O etc.

■ **Process**

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

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

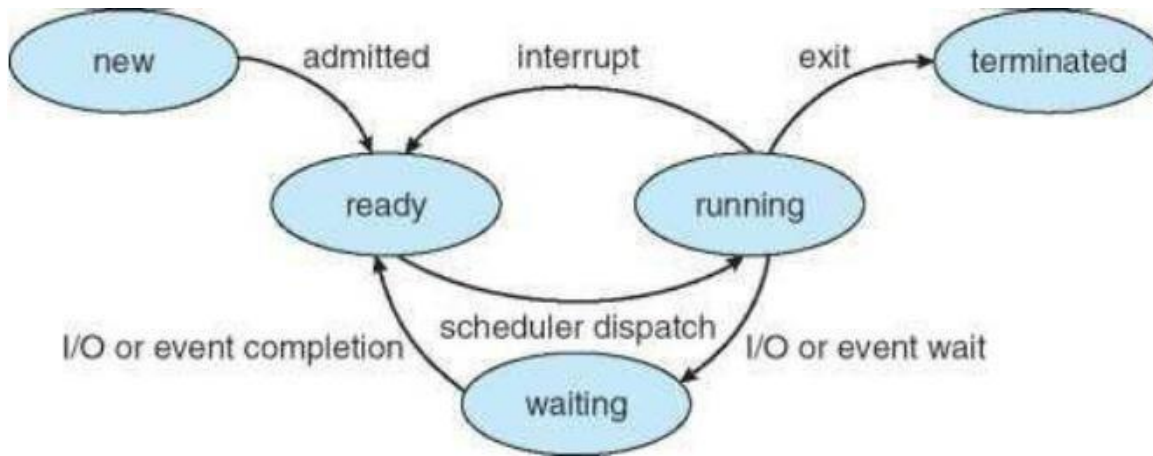
When a program is loaded into the memory and it becomes a process, it can be divided into four sections — stack, heap, text and data. The following image shows a **simplified layout of a process inside main memory** —



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

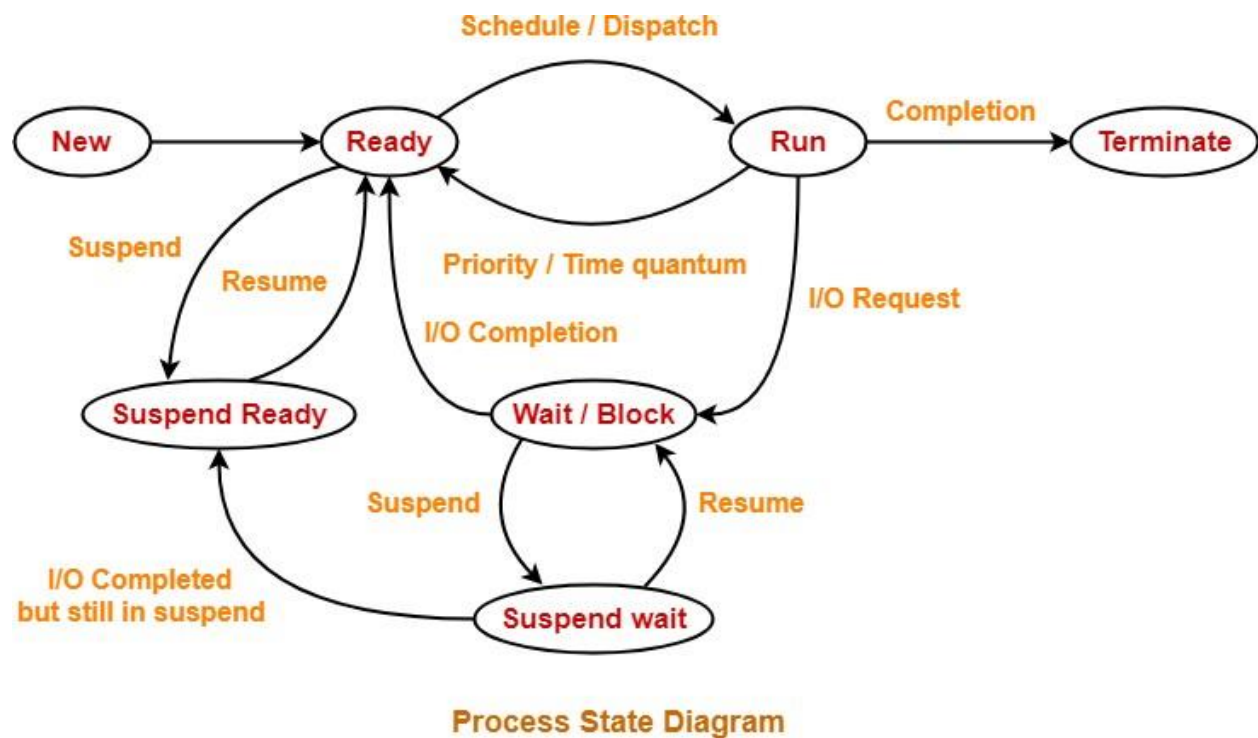
■ process State

- There are mainly Five states of process



S.N.	State & Description
1	Start This is the initial state when a process is first started/created.
2	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.
3	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.
4	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.
5	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.

Suspended Processes



Reasons for Suspension of process

Reason	Comment
Swapping	The OS needs to release sufficient main memory to bring in a process that is ready to execute.
Other OS Reason	OS suspects process of causing a problem.
Interactive User Request	e.g. debugging or in connection with the use of a resource.
Timing	A process may be executed periodically (e.g., an accounting or system monitoring process) and may be suspended while waiting for the next time.
Parent Process Request	A parent process may wish to suspend execution of a descendent to examine or modify the suspended process, or to coordinate the activity of various descendants.

Table 3.3 Reasons for Process Suspension

1. swapping 2. Timing 3. Interactive User Request 4. Parent Process Request

Process Control Block(PCB)

Process Control Block is a data structure that contains information of the process related to it. The process control block is also known as a task control block, entry of the process table, etc.

It is very important for process management as the data structuring for processes is done in terms of the PCB. It also defines the current state of the operating system.

While creating a process the operating system performs several operations. To identify these process, it must identify each process, hence it assigns a process identification number (PID) to each process. As the operating system supports multi-programming, it needs to keep track of all the processes. For this task, the process control block (PCB) is used to track the process's execution status. Each block of memory contains information about the process state, program counter, stack pointer, status of opened files, scheduling algorithms, etc. All these information is required and must be saved when the process is switched from one state to another. When the process made transitions from one state to another, the operating system must update information in the process's PCB.

A process control block (PCB) contains information about the process, i.e. registers, quantum, priority, etc. The process table is an array of PCB's, that means logically contains a PCB for all of the current processes in the system.



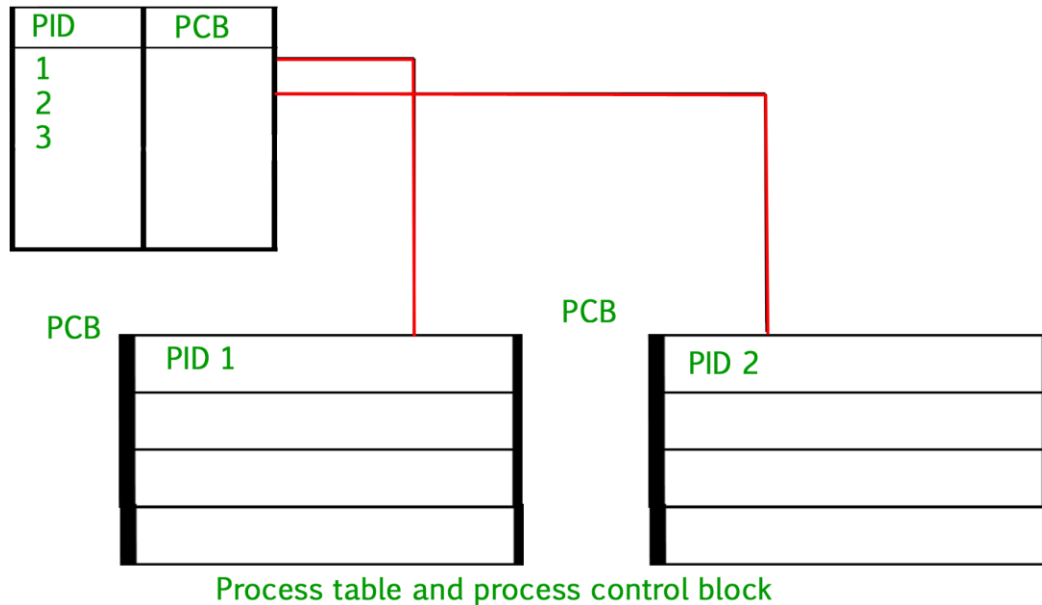
Process Control Block

- **Process state** – It stores the respective state of the process.
- **Pointer** – It is a stack pointer which is required to be saved when the process is switched from one state to another to retain the current position of the process.
- **Process number** – Every process is assigned with a unique id known as process ID or PID which stores the process identifier.
- **Program counter** – It stores the counter which contains the address of the next instruction that is to be executed for the process.
- **Register** – These are the CPU registers which includes: accumulator, base, registers and general purpose registers.
- **Memory limits** – This field contains the information about memory management system used by operating system. This may include the page tables, segment tables etc.

- **Open files list** – This information includes the list of files opened for a process.

- **Miscellaneous accounting and status data** – This field includes information about the amount of CPU used, time constraints, jobs or process number, etc.

The process control block stores the register content also known as execution content of the processor when it was blocked from running. This execution content architecture enables the operating system to restore a process's execution context when the process returns to the running state. When the process made transitions from one state to another, the operating system update its information in the process's PCB. The operating system maintains pointers to each process's PCB in a process table so that it can access the PCB quickly.



OPERATING SYSTEM - PROCESS SCHEDULING

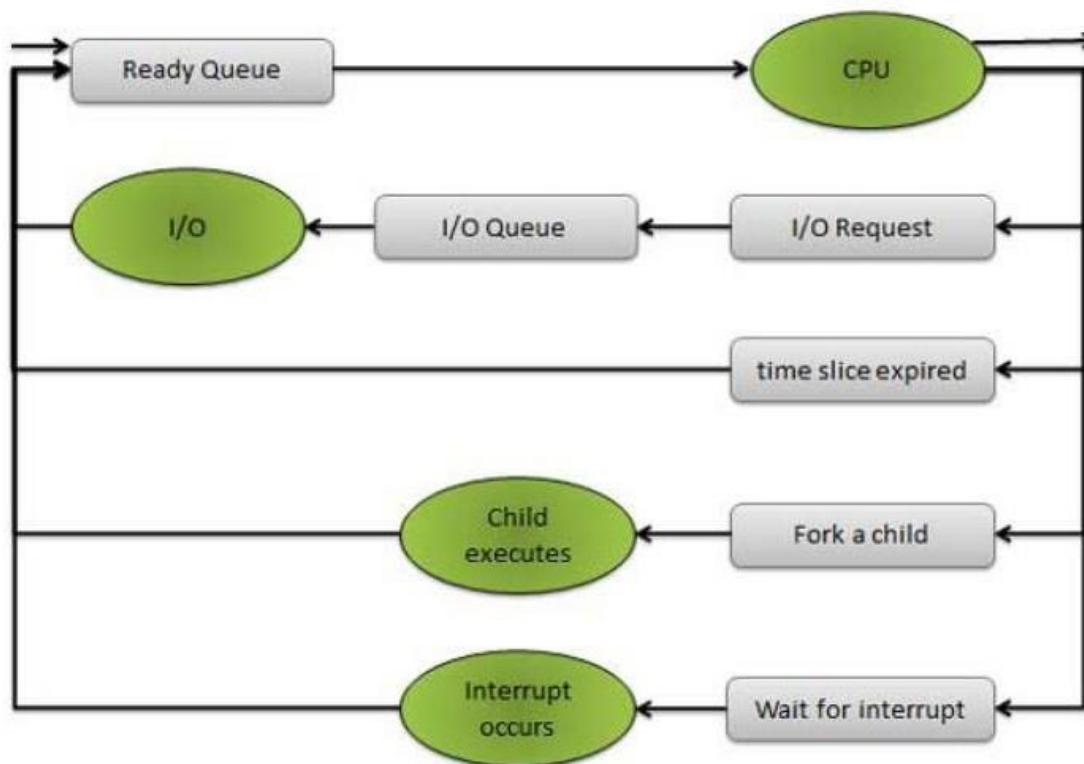
Definition:

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 system. Such operating systems allow more than one process to be loaded into the executable memory at a time and loaded process shares the CPU using time multiplexing.

Scheduling Queues :

Scheduling queues refers to queues of processes or devices. When the process enters into the system, then this process is put into a job queue. This queue consists of all processes in the

system. The operating system also maintains other queues such as device queue. Device queue is a queue for which multiple processes are waiting for a particular I/O device. Each device has its own device queue. This figure shows the queuing diagram of process scheduling. Queue is represented by rectangular box. The circles represent the resources that serve the queues. The arrows indicate the process flow in the system. Queues are of two types Ready queue Device queue. A newly arrived process is put in the ready queue. Processes wait in ready queue for allocating the CPU. Once the CPU is assigned to a process, then that process will execute. While executing the process, any one of the following events can occur.



The process could issue an I/O request and then it would be placed in an I/O queue. The process could create new sub process and will wait for its termination. The process could be removed forcibly from the CPU, as a result of interrupt and put back in the ready queue.

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

Schedulers

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

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

Long Term Scheduler

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

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

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

Short Term Scheduler

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

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

Medium Term Scheduler

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

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

Comparison among Scheduler

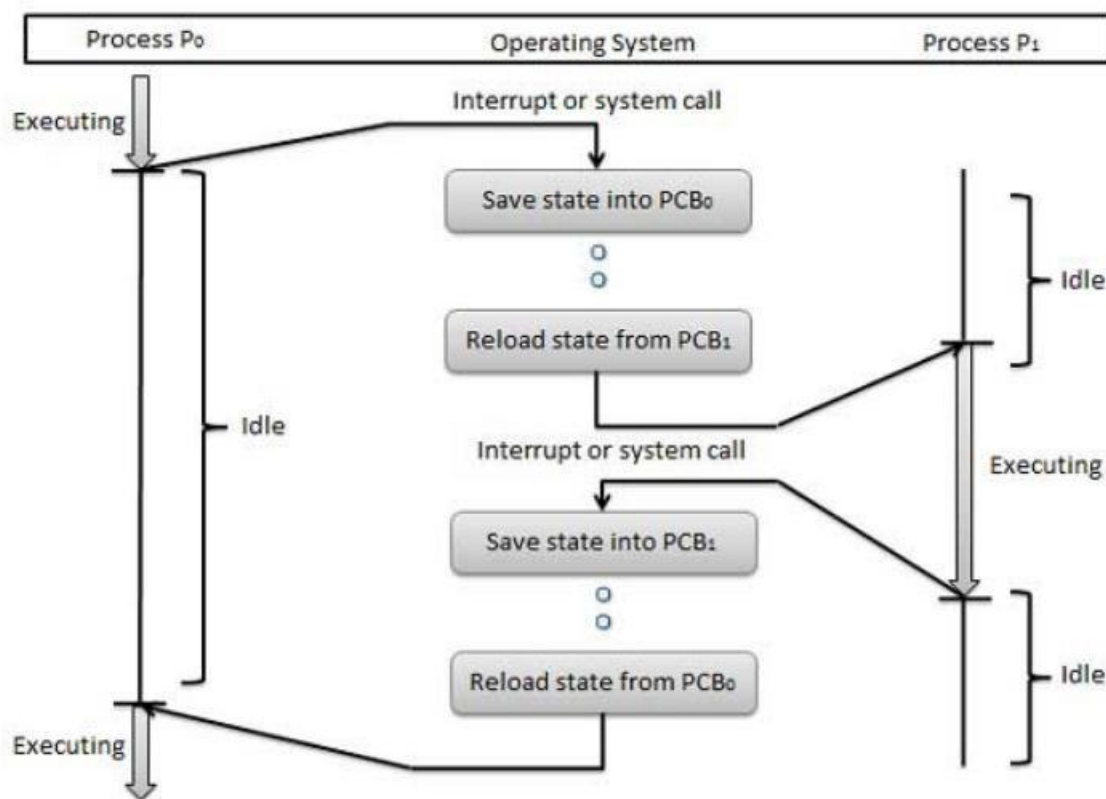
S.N.	Long-Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
2	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
3	It controls the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

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. When the process is switched, the following information is stored for later use.

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

Scheduling Criteria

Scheduling can be defined as a set of policies and mechanisms which controls the order in which the work to be done is completed. The scheduling program which is a system software concerned with scheduling is called the scheduler and the algorithm it uses is called the scheduling algorithm.

Various criteria or characteristics that help in designing a good scheduling algorithm are:

- **CPU Utilization** – A scheduling algorithm should be designed so that CPU remains busy as possible. It should make efficient use of CPU.
- **Throughput** – Throughput is the amount of work completed in a unit of time. In other words throughput is the processes executed to number of jobs completed in a unit of time. The scheduling algorithm must look to maximize the number of jobs processed per time unit.
- **Response time** – Response time is the time taken to start responding to the request. A scheduler must aim to minimize response time for interactive users.
- **Turnaround time** – Turnaround time refers to the time between the moment of submission of a job/ process and the time of its completion. Thus how long it takes to execute a process is also an important factor.

Turnaround Time = Waiting Time + Burst Time(Execution Time)

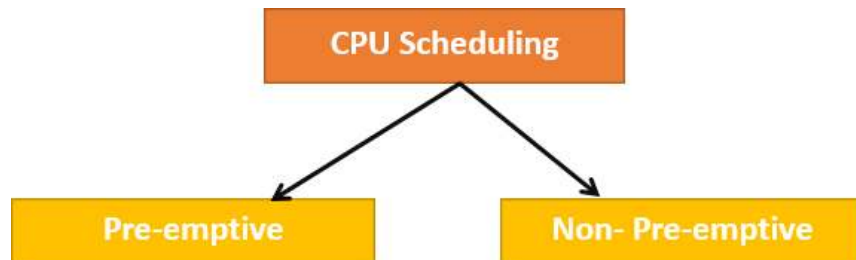
- **Waiting time** – It is the time a job waits for resource allocation when several jobs are competing in multiprogramming system. The aim is to minimize the waiting time.

Waiting time = Response Time - Arrival Time

- **Fairness** – A good scheduler should make sure that each process gets its fair share of the CPU.

Scheduling Algorithms

CPU Scheduling is a process of determining which process will own CPU for execution while another process is on hold. The main task of CPU scheduling is to make sure that whenever the CPU remains idle, the OS at least select one of the processes available in the ready queue for execution. The selection process will be carried out by the CPU scheduler. It selects one of the processes in memory that are



Preemptive Scheduling

In Preemptive Scheduling, the tasks are mostly assigned with their priorities. Sometimes it is important to run a task with a higher priority before another lower priority task, even if the lower priority task is still running. The lower priority task holds for some time and resumes when the higher priority task finishes its execution.

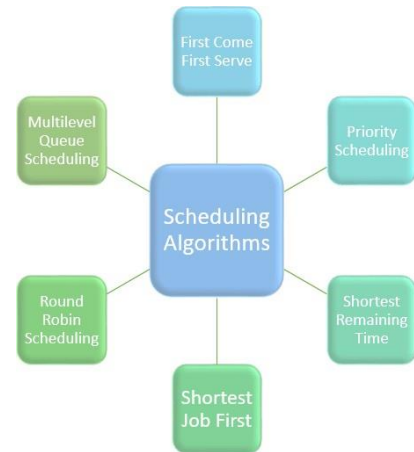
Non-Preemptive Scheduling

In this type of scheduling method, the CPU has been allocated to a specific process. The process that keeps the CPU busy will release the CPU either by switching context or terminating. It is the only method that can be used for various hardware platforms. That's because it doesn't need special hardware (for example, a timer) like preemptive scheduling.

2

Types of CPU scheduling Algorithm There are mainly six types of process scheduling algorithms

1. First Come First Serve (FCFS)
2. Shortest-Job-First (SJF) Scheduling
3. Shortest Remaining Time
4. Priority Scheduling
5. Round Robin Scheduling
6. Multilevel Queue Scheduling



■ Threads

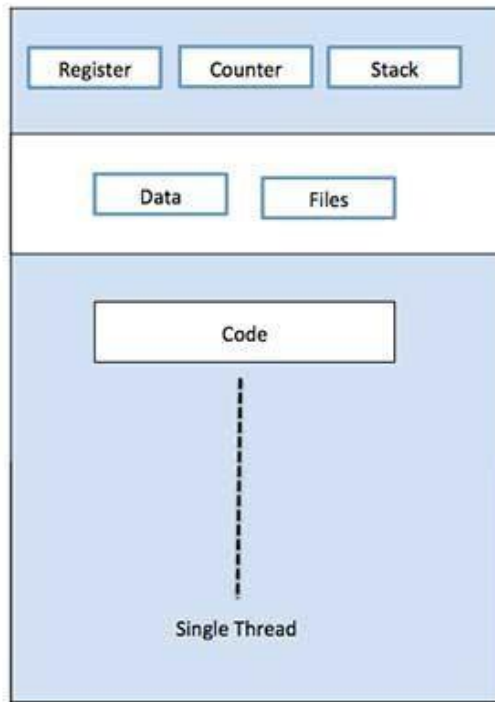
What is Thread?

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.

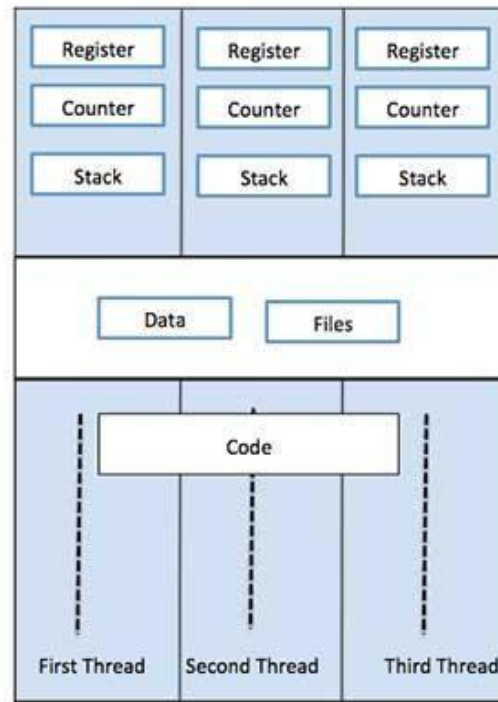
A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

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

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

Advantages of Thread

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

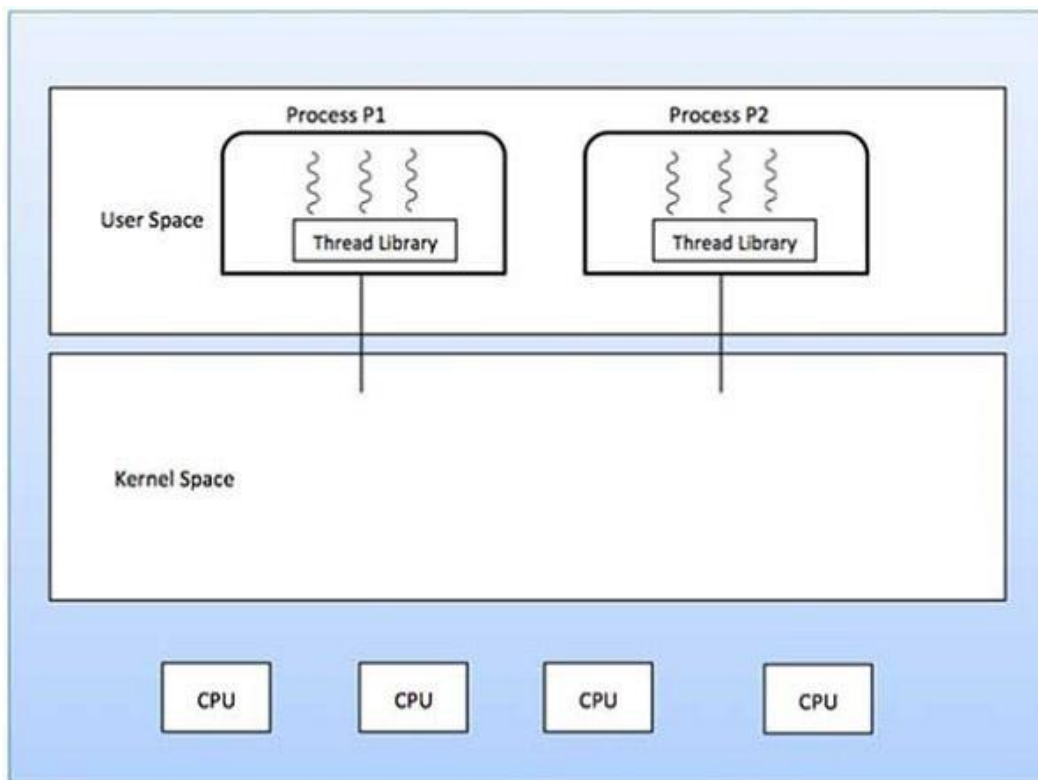
Types of Thread

Threads are implemented in following two ways –

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

1) User Level Threads

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



Advantages

- Thread switching does not require Kernel mode privileges.

- User level thread can run on any operating system.
- Scheduling can be application specific in the user level thread.
- User level threads are fast to create and manage.

Disadvantages

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

2) Kernel Level Threads

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

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

Advantages

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

Disadvantages

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

Multi threading Models

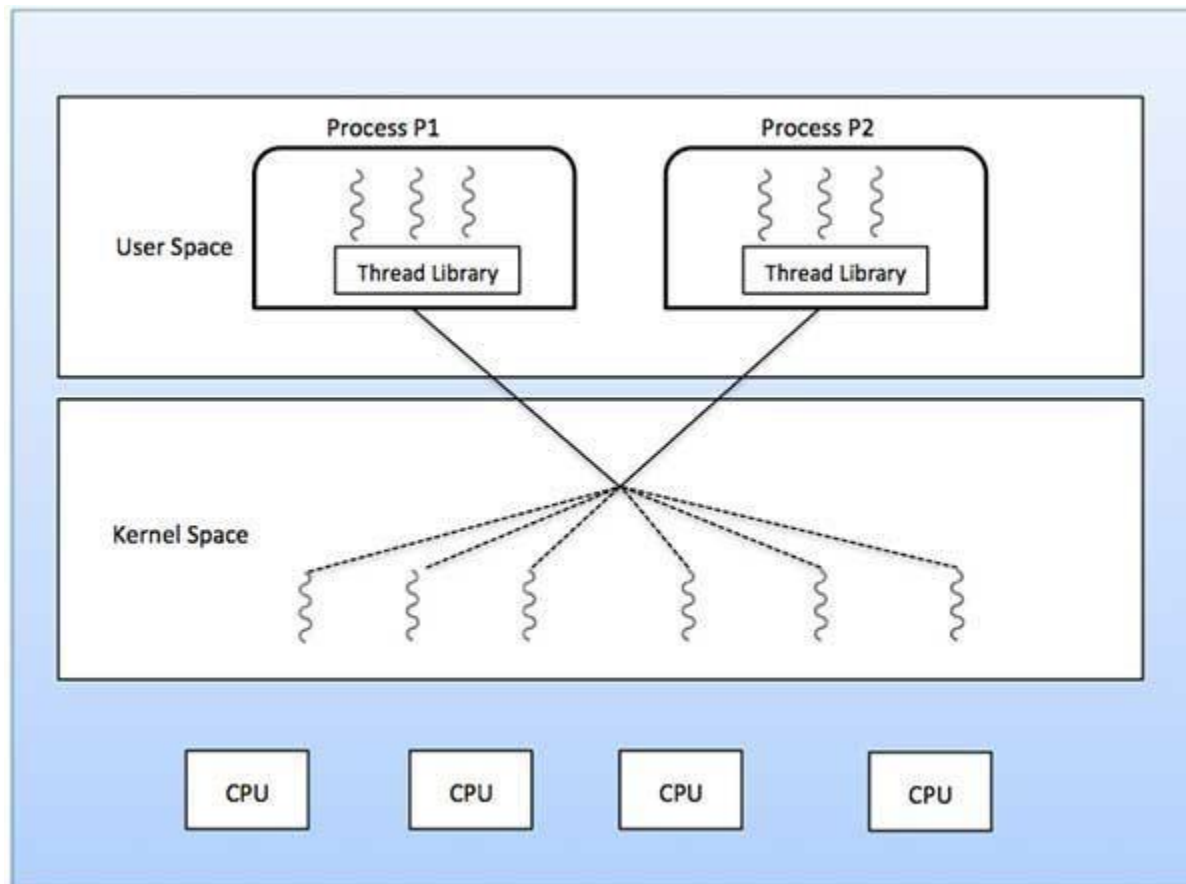
Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the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

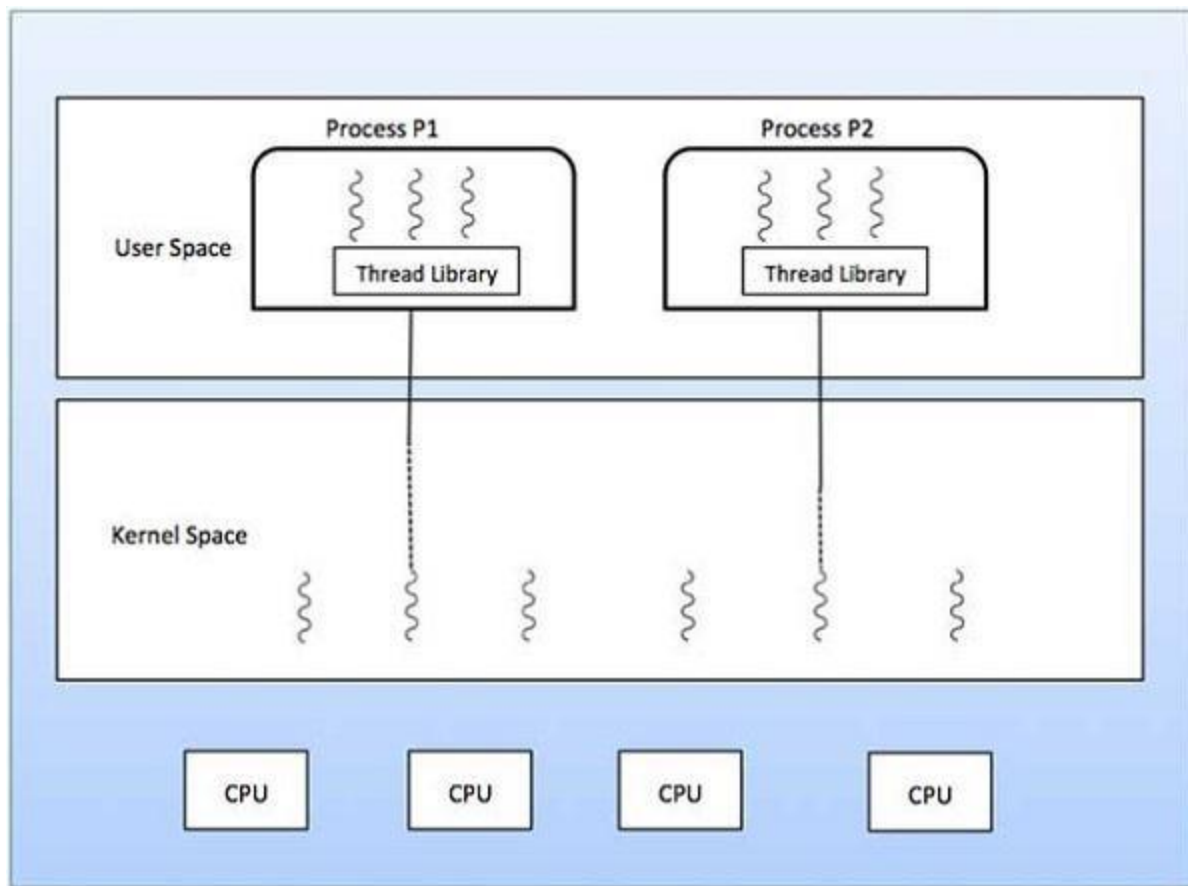
The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model 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.

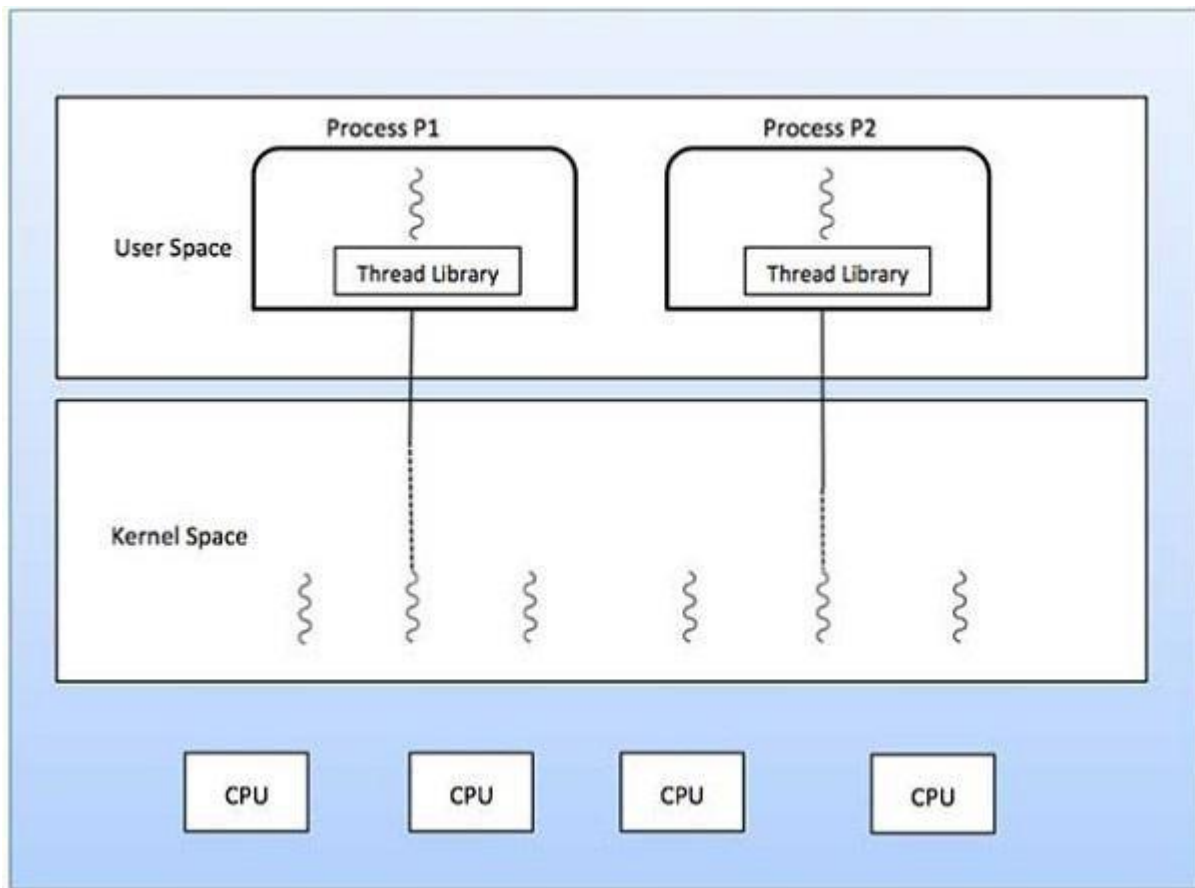
If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.



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.



Thread Life Cycle



- **New** – A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
- **Runnable** – After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting** – Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed Waiting** – A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated (Dead)** – A runnable thread enters the terminated state when it completes its task or otherwise terminates.
- **Blocked State:** The thread is waiting for an event to occur or waiting for an I/O device.
- **Sleep:** A sleeping thread becomes ready after the designated sleep time expires.

UNIT 3 & 4: Concurrency and IPC

Race Condition

A race condition is a situation that may occur inside a critical section. This happens when the result of multiple thread execution in critical section differs according to the order in which the threads execute.

Race conditions in critical sections can be avoided if the critical section is treated as an atomic instruction. Also, proper thread synchronization using locks or atomic variables can prevent race conditions.

Suppose for a moment that two processes need to perform a bit [flip](#) at a specific memory location. Under normal circumstances the operation should work like this:

Process 1	Process 2	Memory Value
Read value		0
Flip value		1
	Read value	1
	Flip value	0

In this example, Process 1 performs a bit flip, changing the memory value from 0 to 1. Process 2 then performs a bit flip and changes the memory value from 1 to 0.

If a race condition occurred causing these two processes to overlap, the sequence could potentially look more like this:

Process 1	Process 2	Memory Value
Read value		0
	Read value	0
Flip value		1
	Flip value	1

In this example, the bit has an ending value of 1 when its value should be 0. This occurs because Process 2 is unaware that Process 1 is performing a simultaneous bit flip.

If Order of Execution is changed then the result is also changed and it generates Inconsistency.

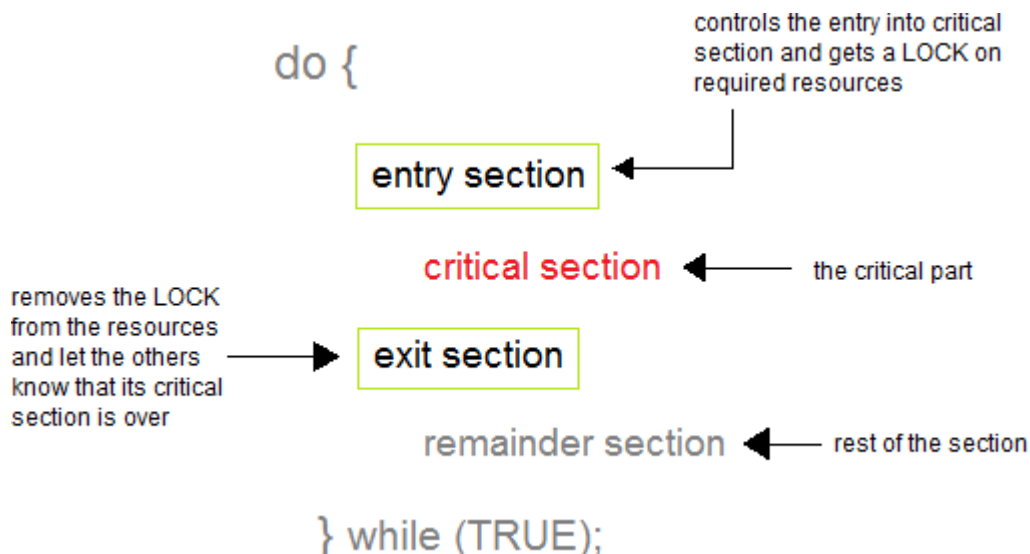
Process Synchronization

Process Synchronization means sharing system resources by processes in a such a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data. Maintaining data consistency demands mechanisms to ensure synchronized execution of cooperating processes.

Process Synchronization was introduced to handle problems that arose while multiple process executions. Some of the problems are discussed below.

Critical Section Problem

A Critical Section is a code segment that accesses shared variables and has to be executed as an atomic action. It means that in a group of cooperating processes, at a given point of time, only one process must be executing its critical section. If any other process also wants to execute its critical section, it must wait until the first one finishes.



Solution to Critical Section Problem

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

1. Mutual Exclusion

Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.

2. Progress

If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.

Progress Criteria :

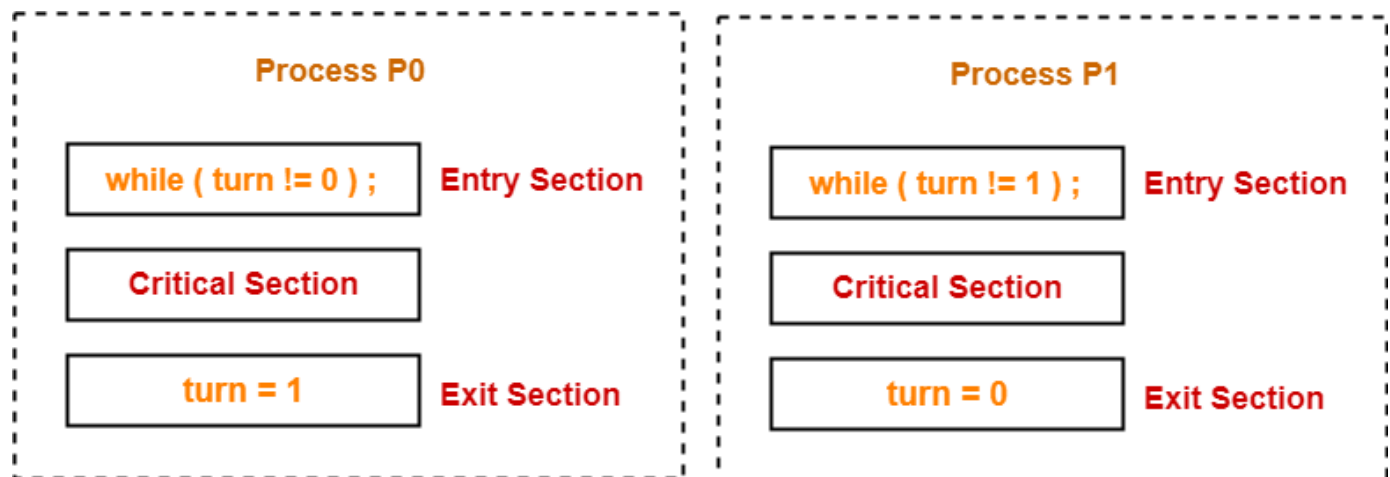
1. Strict alteration(If there is no strict alteration than its satisfy the progress)
2. Deadlock (If there is no deadlock than it iits satisfy the progress)

3. Bounded Waiting

After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted. So after the limit is reached, system must grant the process permission to get into its critical section.

CASE 1 :Turn Variable-

- Turn variable is a synchronization mechanism that provides synchronization among two processes.
- It uses a turn variable to provide the synchronization.



Initially, turn value is set to 0.

- Turn value = 0 means it is the turn of process P_0 to enter the critical section.
- Turn value = 1 means it is the turn of process P_1 to enter the critical section.

Working-

This synchronization mechanism works as explained in the following scenes-

Scene-01:

- Process P_0 arrives.
- It executes the $turn \neq 0$ instruction.
- Since turn value is set to 0, so it returns value 0 to the while loop.
- The while loop condition breaks.
- Process P_0 enters the critical section and executes.
- Now, even if process P_0 gets preempted in the middle, process P_1 can not enter the critical section.
- Process P_1 can not enter unless process P_0 completes and sets the turn value to 1.

Scene-02:

- Process P_1 arrives.
- It executes the $turn \neq 1$ instruction.
- Since turn value is set to 0, so it returns value 1 to the while loop.
- The returned value 1 does not break the while loop condition.
- The process P_1 is trapped inside an infinite while loop.
- The while loop keeps the process P_1 busy until the turn value becomes 1 and its condition breaks.

Scene-03:

- Process P_0 comes out of the critical section and sets the turn value to 1.
- The while loop condition of process P_1 breaks.
- Now, the process P_1 waiting for the critical section enters the critical section and executes.
- Now, even if process P_1 gets preempted in the middle, process P_0 can not enter the critical section.
- Process P_0 can not enter unless process P_1 completes and sets the turn value to 0.

The characteristics of this synchronization mechanism are-

- It ensures mutual exclusion.
- It follows the strict alternation approach.

Strict Alternation Approach

In strict alternation approach,

- Processes have to compulsorily enter the critical section alternately whether they want it or not.
- This is because if one process does not enter the critical section, then other process will never get a chance to execute again.

- It does not guarantee progress since it follows strict alternation approach.
- It ensures bounded waiting since processes are executed turn wise one by one and each process is guaranteed to get a chance.
- It ensures processes does not starve for the CPU.
- It is architectural neutral since it does not require any support from the operating system.
- It is **deadlock** free.
- It is a busy waiting solution which keeps the CPU busy when the process is actually waiting.

CASE 2 :Flag Variable-

-Algorithm 2

- Shared variables
 - **boolean flag[2];** // “interest” bits
initially flag [0] = flag [1] = false.
 - **flag [i] = true $\Rightarrow$ P_i** declares interest in entering its critical section
- Process P_i // where the “other” process is P_j

```
do {  
    flag[i] = true;    // declare your own interest  
    while (flag[ j]) ; //wait if the other guy is interested  
    critical section  
    flag [i] = false;  // declare that you lost interest  
    remainder section // allows other guy to enter  
} while (1);
```

Mutual Exclusion : Satisfy

Progress : Strict alteration (not) but deadlock is occurred hence, progress is not satisfy.

CASE 3 : PETERSON SOLUTION-

The algorithm uses two variables, `flag` and `turn`. A `flag[n]` value of `true` indicates that the process `n` wants to enter the critical section.. Entrance to the critical section is granted for process `P0` if `P1` does not want to enter its critical section or if `P1` has given priority to `P0` by setting `turn` to 0.

```
bool flag[2] = {false, false};
int turn;
```

```
P0:    flag[0] = true;
P0_gate: turn = 1;
        while (flag[1] == true && turn == 1)
        {
            // busy wait
        }
        // critical section
        ...
        // end of critical section
        flag[0] = false;
```

```
P1:    flag[1] = true;
P1_gate: turn = 0;
        while (flag[0] == true && turn == 0)
        {
            // busy wait
        }
        // critical section
        ...
        // end of critical section
        flag[1] = false;
```

The algorithm satisfies the three essential criteria to solve the critical section problem, provided that changes to the variables `turn`, `flag[0]`, and `flag[1]` propagate immediately and atomically. The while condition works even with preemption.

The three criteria are mutual exclusion, progress, and bounded waiting.

Since `turn` can take on one of two values, it can be replaced by a single bit, meaning that the algorithm requires only three bits of memory.

Mutual exclusion

`P0` and `P1` can never be in the critical section at the same time: If `P0` is in its critical section, then `flag[0]` is true. In addition, either `flag[1]` is false (meaning `P1` has left its critical section), or `turn` is 0 (meaning `P1` is just now trying to enter the critical section, but graciously waiting), or `P1` is at label `P1_gate` (trying to enter its critical section, after setting `flag[1]` to true but before setting `turn` to 0 and busy waiting). So if both processes are in their critical sections then we conclude that the state must satisfy `flag[0]` and `flag[1]` and `turn = 0` and `turn = 1`. No state can satisfy both `turn = 0` and `turn = 1`, so there can be no state where both processes are in their critical sections. (This recounts an argument that is made rigorous in.)

Progress

Progress is defined as the following: 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 making the decision as to which process will enter its critical section next. Note that for a process or thread, the remainder sections are parts of the code that are not related to the critical section. This selection cannot be postponed indefinitely. process cannot immediately re-enter the critical section if the other process has set its flag to say that it would like to enter its critical section.

Bounded waiting

Bounded waiting, or bounded bypass means that the number of times a process is bypassed by another process after it has indicated its desire to enter the critical section is bounded by a function of the number of processes in the system.:11 In Peterson's algorithm, a process will never wait longer than one turn for entrance to the critical section.

Semaphore

Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization.

The definitions of wait and signal are as follows –

Wait

The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.

```
wait(S)
{
    while (S<=0);

    S--;
}
```

Signal

The signal operation increments the value of its argument S.

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

Types of Semaphores

There are two main types of semaphores i.e. counting semaphores and binary semaphores. Details about these are given as follows:

- **Counting Semaphores**

These are integer value semaphores and have an unrestricted value domain. These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources. If the resources are added, semaphore count automatically incremented and if the resources are removed, the count is decremented.

- **Binary Semaphores**

The binary semaphores are like counting semaphores but their value is restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores.

Advantages of Semaphores

Some of the advantages of semaphores are as follows:

- Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.
- There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
- Semaphores are implemented in the machine independent code of the microkernel. So they are machine independent.

Disadvantages of Semaphores

Some of the disadvantages of semaphores are as follows –

- Semaphores are complicated so the wait and signal operations must be implemented in the correct order to prevent deadlocks.
- Semaphores are impractical for last scale use as their use leads to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.
- Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.

Reader Writer Problem

The readers-writers problem relates to an object such as a file that is shared between multiple processes. Some of these processes are readers i.e. they only want to read the data from the object and some of the processes are writers i.e. they want to write into the object.

The readers-writers problem is used to manage synchronization so that there are no problems with the object data. For example - If two readers access the object at the same time there is no problem. However if two writers or a reader and writer access the object at the same time, there may be problems.

To solve this situation, a writer should get exclusive access to an object i.e. when a writer is accessing the object, no reader or writer may access it. However, multiple readers can access the object at the same time.

This can be implemented using semaphores. The codes for the reader and writer process in the reader-writer problem are given as follows:

Reader Process

The code that defines the reader process is given below:

```
wait (mutex);
rc ++;
if (rc == 1)
wait (wrt);
signal(mutex);

.
.  READ THE OBJECT
.
wait(mutex);
rc --;
if (rc == 0)
signal (wrt);
signal(mutex);
```

In the above code, mutex and wrt are semaphores that are initialized to 1. Also, rc is a variable that is initialized to 0. The mutex semaphore ensures mutual exclusion and wrt handles the writing mechanism and is common to the reader and writer process code.

The variable rc denotes the number of readers accessing the object. As soon as rc becomes 1, wait operation is used on wrt. This means that a writer cannot access the object anymore. After the read operation is done, rc is decremented. When rc becomes 0, signal operation is used on wrt. So a writer can access the object now.

Writer Process

The code that defines the writer process is given below:

```
wait(wrt);

.
.  WRITE INTO THE OBJECT
.
signal(wrt);
```

If a writer wants to access the object, wait operation is performed on wrt. After that no other writer can access the object. When a writer is done writing into the object, signal operation is performed on wrt.

Producer Consumer Problem

The producer consumer problem is a synchronization problem. There is a fixed size buffer and the producer produces items and enters them into the buffer. The consumer removes the items from the buffer and consumes them.

A producer should not produce items into the buffer when the consumer is consuming an item from the buffer and vice versa. So the buffer should only be accessed by the producer or consumer at a time.

The producer consumer problem can be resolved using semaphores. The codes for the producer and consumer process are given as follows:

Producer Process

The code that defines the producer process is given below:

```
do {  
    .  
    . PRODUCE ITEM  
    .  
    wait(empty);  
    wait(mutex);  
    .  
    . PUT ITEM IN BUFFER  
    .  
    signal(mutex);  
    signal(full);  
} while(1);
```

In the above code, mutex, empty and full are semaphores. Here mutex is initialized to 1, empty is initialized to n (maximum size of the buffer) and full is initialized to 0. The mutex semaphore ensures mutual exclusion. The empty and full semaphores count the number of empty and full spaces in the buffer.

After the item is produced, wait operation is carried out on empty. This indicates that the empty space in the buffer has decreased by 1. Then wait operation is carried out on mutex so that consumer process cannot interfere.

After the item is put in the buffer, signal operation is carried out on mutex and full. The former indicates that consumer process can now act and the latter shows that the buffer is full by 1.

Consumer Process

The code that defines the consumer process is given below:

```

wait(full);
wait(mutex);
.
.  REMOVE ITEM FROM BUFFER
.
signal(mutex);
signal(empty);
.
.  CONSUME ITEM
.
} while(1);

```

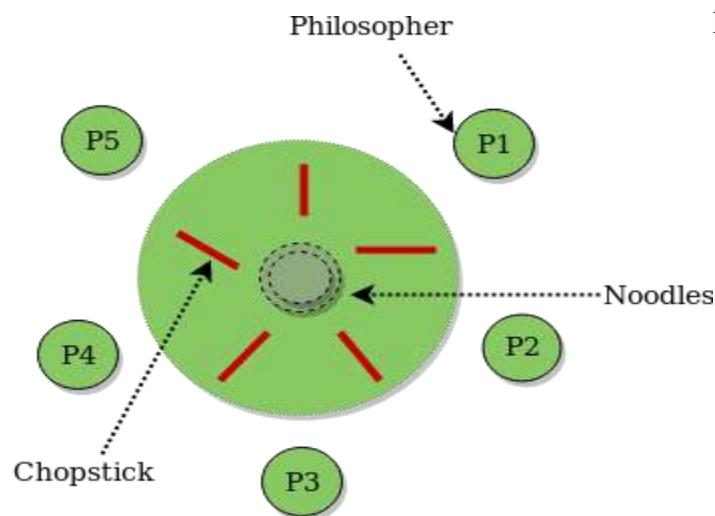
The wait operation is carried out on full. This indicates that items in the buffer have decreased by 1. Then wait operation is carried out on mutex so that producer process cannot interfere.

Then the item is removed from buffer. After that, signal operation is carried out on mutex and empty. The former indicates that consumer process can now act and the latter shows that the empty space in the buffer has increased by 1.

Dining Philosophers Problem

The dining philosophers problem states that there are 5 philosophers sharing a circular table and they eat and think alternatively. There is a bowl of rice for each of the philosophers and 5 chopsticks. A philosopher needs both their right and left chopstick to eat. A hungry philosopher may only eat if there are both chopsticks available. Otherwise a philosopher puts down their chopstick and begin thinking again.

The dining philosopher is a classic synchronization problem as it demonstrates a large class of concurrency control problems.



Solution of Dining Philosophers Problem

A solution of the Dining Philosophers Problem is to use a semaphore to represent a chopstick. A chopstick can be picked up by executing a wait operation on the semaphore and released by executing a signal semaphore.

The structure of the chopstick is shown below:

semaphore chopstick [5];

Initially the elements of the chopstick are initialized to 1 as the chopsticks are on the table and not picked up by a philosopher.

The structure of a random philosopher i is given as follows:

```
do {  
    wait( chopstick[i] );  
    wait( chopstick[ (i+1) % 5] );  
    .  
    . EATING THE RICE  
    .  
    signal( chopstick[i] );  
    signal( chopstick[ (i+1) % 5] );  
    .  
    . THINKING  
    .  
} while(1);
```

In the above structure, first wait operation is performed on chopstick[i] and chopstick[(i+1) % 5]. This means that the philosopher i has picked up the chopsticks on his sides. Then the eating function is performed.

After that, signal operation is performed on chopstick[i] and chopstick[(i+1) % 5]. This means that the philosopher i has eaten and put down the chopsticks on his sides. Then the philosopher goes back to thinking.

Difficulty with the solution

The above solution makes sure that no two neighboring philosophers can eat at the same time. But this solution can lead to a deadlock. This may happen if all the philosophers pick their left chopstick simultaneously. Then none of them can eat and deadlock occurs.

Some of the ways to avoid deadlock are as follows:

- There should be at most four philosophers on the table.
- An even philosopher should pick the right chopstick and then the left chopstick while an odd philosopher should pick the left chopstick and then the right chopstick.
- A philosopher should only be allowed to pick their chopstick if both are available at the same time.

Monitors

Monitors are a synchronization construct that were created to overcome the problems caused by semaphores such as timing errors.

Monitors are abstract data types and contain shared data variables and procedures. The shared data variables cannot be directly accessed by a process and procedures are required to allow a single process to access the shared data variables at a time.

This is demonstrated as follows:

```
monitor monitorName
```

```
{  
    data variables;
```

```
    Procedure P1(... )
```

```
{  
}
```

```
    Procedure P2(... )
```

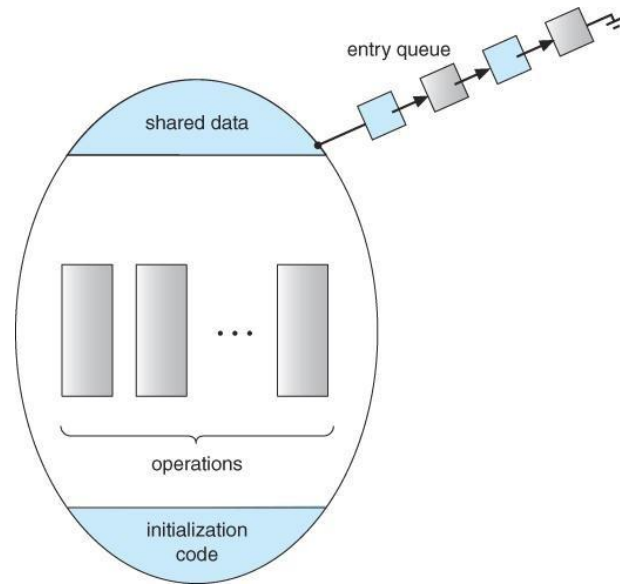
```
{  
  
}
```

```
    Procedure Pn(... )
```

```
{  
  
}
```

```
    Initialization Code(. ..)
```

```
{  
  
}  
}
```



Only one process can be active in a monitor at a time. Other processes that need to access the shared variables in a monitor have to line up in a queue and are only provided access when the previous process release the shared variables.

Monitor use two queues

1) Monitor Queue (Waiting queue -Outside Monitor Queue)

2)Conditional Variable Queue

The *condition variable* is a synchronization primitive that provides a queue for threads waiting for a resource. A thread tests to see if the resource is available. If it is available, it uses it. Otherwise it adds itself to the queue of threads waiting for the resource. When a thread has finished with a resource, it wakes up exactly one thread from the queue (or none, if the queue is empty). In the case of a sharable resource, a broadcast can be sent to wake up all sleeping threads.

Proper use of condition variables provides for safe access both to the queue and to test the resource even with concurrency. The implementation of condition variables involves several mutexes.

Condition variables support three operations:

- wait - add calling thread to the queue and put it to sleep
- signal - remove a thread from the queue and wake it up
- broadcast - remove and wake-up all threads on the queue

When using condition variables, an additional mutex must be used to protect the critical sections of code that test the lock or change the locks state.

Producer Consumer Problem using Monitors

The following PSEUDO CODE shows a solution to the producer–consumer problem using MONITORS. Since mutual exclusion is implicit with monitors, no extra effort is necessary to protect the critical section. In other words, the solution shown below works with any number of producers and consumers without any modifications. It is also noteworthy that it is less likely for a programmer to write code that suffers from race conditions when using monitors than when using semaphores.

monitor ProducerConsumer

```
{
    int itemCount = 0;
    condition full;
    condition empty;

    procedure add(item)
    {
        if (itemCount == BUFFER_SIZE)
        {
            wait(full);
        }

        putItemIntoBuffer(item);
        itemCount = itemCount + 1;
```

```

        if (itemCount == 1)
        {
            notify(empty);
        }
    }

    procedure remove()
    {
        if (itemCount == 0)
        {
            wait(empty);
        }

        item = removeItemFromBuffer();
        itemCount = itemCount - 1;

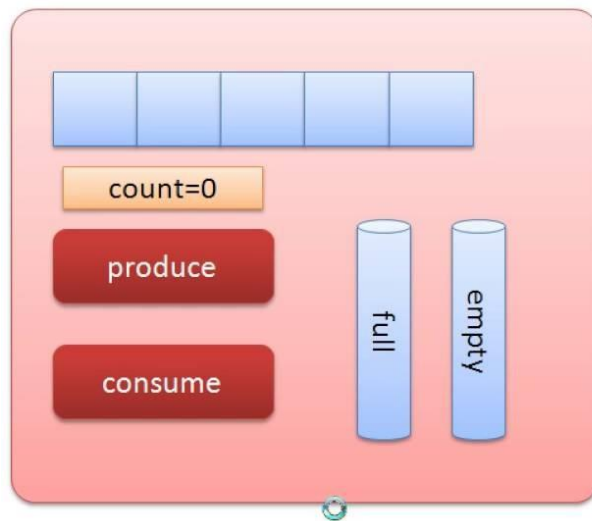
        if (itemCount == BUFFER_SIZE - 1)
        {
            notify(full);
        }

        return item;
    }
}

procedure producer()
{
    while (true)
    {
        item = produceItem();
        ProducerConsumer.add(item);
    }
}

procedure consumer()
{
    while (true)
    {
        item = ProducerConsumer.remove();
        consumeItem(item);
    }
}

```

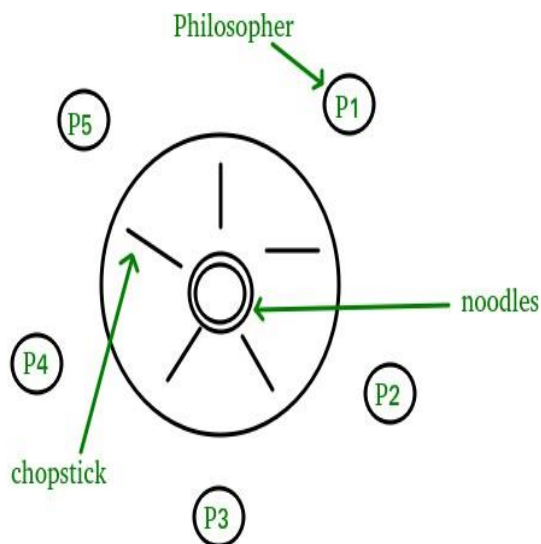


Producer

Consumer

Dining-Philosophers using Monitors

Dining-Philosophers Problem – N philosophers seated around a circular table



- There is one chopstick between each philosopher
- A philosopher must pick up its two nearest chopsticks in order to eat
- A philosopher must pick up first one chopstick, then the second one, not both at once

We need an algorithm for allocating these limited resources(chopsticks) among several processes(philosophers) such that solution is free from deadlock and free from starvation.

There exist some algorithm to solve Dining – Philosopher Problem, but they may have deadlock situation. Also, a deadlock-free solution is not necessarily starvation-free. Semaphores can result in deadlock due to programming errors. Monitors alone are not sufficiency to solve this, we need monitors with *condition variables*

Monitor-based Solution to Dining Philosophers

We illustrate monitor concepts by presenting a deadlock-free solution to the dining-philosophers problem. Monitor is used to control access to state variables and condition variables. It only tells when to enter and exit the segment. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.

To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:

THINKING – When philosopher doesn't want to gain access to either fork.

HUNGRY – When philosopher wants to enter the critical section.

EATING – When philosopher has got both the forks, i.e., he has entered the section.

Philosopher i can set the variable $state[i] = EATING$ only if her two neighbors are not eating ($state[(i+4) \% 5] \neq EATING$) and ($state[(i+1) \% 5] \neq EATING$).

// Dining-Philosophers Solution Using Monitors

monitor DP

```
{
    status state[5];
    condition self[5];

    // Pickup chopsticks
    Pickup(int i)
    {
        // indicate that I'm hungry
        state[i] = hungry;

        // set state to eating in test()
        // only if my left and right neighbors
        // are not eating
        test(i);

        // if unable to eat, wait to be signaled
        if (state[i] != eating)
```

```

        self[i].wait;
    }

    // Put down chopsticks
    Putdown(int i)
    {

        // indicate that I'm thinking
        state[i] = thinking;

        // if right neighbor R=(i+1)%5 is hungry and
        // both of R's neighbors are not eating,
        // set R's state to eating and wake it up by
        // signaling R's CV
        test((i + 1) % 5);
        test((i + 4) % 5);
    }

    test(int i)
    {

        if (state[(i + 1) % 5] != eating
            && state[(i + 4) % 5] != eating
            && state[i] == hungry) {

            // indicate that I'm eating
            state[i] = eating;

            // signal() has no effect during Pickup(),
            // but is important to wake up waiting
            // hungry philosophers during Putdown()
            self[i].signal();
        }
    }

```

```

    }

    init()
    {

        // Execution of Pickup(), Putdown() and test()
        // are all mutually exclusive,
        // i.e. only one at a time can be executing

for
    i = 0 to 4

        // Verify that this monitor-based solution is
        // deadlock free and mutually exclusive in that
        // no 2 neighbors can eat simultaneously
        state[i] = thinking;
    }
} // end of monitor

```

Above Program is a monitor solution to the dining-philosopher problem.

We also need to declare
condition self[5];

This allows philosopher *i* to delay herself when she is hungry but is unable to obtain the chopsticks she needs. We are now in a position to describe our solution to the dining-philosophers problem. The distribution of the chopsticks is controlled by the monitor Dining Philosophers. Each philosopher, before starting to eat, must invoke the operation `pickup()`. This act may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the `putdown()` operation. Thus, philosopher *i* must invoke the operations `pickup()` and `putdown()` in the following sequence:

```

DiningPhilosophers.pickup(i);
    ...
    eat
    ...
DiningPhilosophers.putdown(i);

```

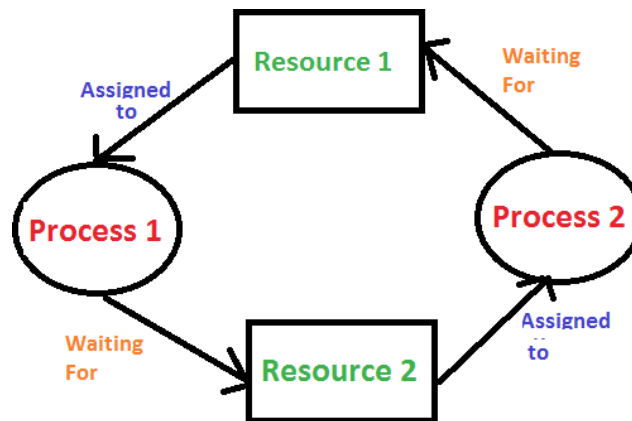
It is easy to show that this solution ensures that **no two neighbors** are eating simultaneously and that no deadlocks will occur. We note, however, that it is possible for a philosopher to starve to death.

UNIT 5: Deadlock

What is Deadlock?

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

Consider an example when two trains are coming toward each other on same track and there is only one track, none of the trains can move once they are in front of each other. Similar situation occurs in operating systems when there are two or more processes hold some resources and wait for resources held by other(s). For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.



Deadlock can arise if following four conditions hold simultaneously (Necessary Conditions)

A deadlock situation on a resource can arise if and only if all of the following conditions hold simultaneously in a system:

1. **Mutual exclusion:** At least one resource must be held in a non-shareable mode. Otherwise, the processes would not be prevented from using the resource when necessary. Only one process can use the resource at any given instant of time.
2. **Hold and wait or resource holding:** a process is currently holding at least one resource and requesting additional resources which are being held by other processes.
3. **No preemption:** a resource can be released only voluntarily by the process holding it.

4. **Circular wait:** each process must be waiting for a resource which is being held by another process, which in turn is waiting for the first process to release the resource. In general, there is a set of waiting processes, $P = \{P_1, P_2, \dots, P_N\}$, such that P_1 is waiting for a resource held by P_2 , P_2 is waiting for a resource held by P_3 and so on until P_N is waiting for a resource held by P_1

Methods for handling deadlock

This strategy involves designing a system that violates one of the four necessary conditions required for the occurrence of deadlock.

- This ensures that the system remains free from the deadlock.

The various conditions of deadlock occurrence may be violated as-

1. Mutual Exclusion-

- To violate this condition, all the system resources must be such that they can be used in a shareable mode.
- In a system, there are always some resources which are mutually exclusive by nature.
- So, this condition can not be violated.

2. Hold and Wait-

This condition can be violated in the following ways-

Approach-01:

In this approach,

- A process has to first request for all the resources it requires for execution.
- Once it has acquired all the resources, only then it can start its execution.
- This approach ensures that the process does not hold some resources and wait for other resources.

Drawbacks-

The drawbacks of this approach are-

- It is less efficient.
- It is not implementable since it is not possible to predict in advance which resources will be required during execution.

Approach-02:

In this approach,

- A process is allowed to acquire the resources it desires at the current moment.
- After acquiring the resources, it start its execution.
- Now before making any new request, it has to compulsorily release all the resources that it holds currently.
- This approach is efficient and implementable.

Approach-03:

In this approach,

- A timer is set after the process acquires any resource.
- After the timer expires, a process has to compulsorily release the resource.

3. No Preemption-

- This condition can be violated by forceful preemption.
- Consider a process is holding some resources and request other resources that can not be immediately allocated to it.
- Then, by forcefully preempting the currently held resources, the condition can be violated.

A process is allowed to forcefully preempt the resources possessed by some other process only if-

- It is a high priority process or a system process.
- The victim process is in the waiting state.

4. Circular Wait-

- This condition can be violated by not allowing the processes to wait for resources in a cyclic manner.
- To violate this condition, the following approach is followed-

Approach-

- A natural number is assigned to every resource.
- Each process is allowed to request for the resources either in only increasing or only decreasing order of the resource number.
- In case increasing order is followed, if a process requires a lesser number resource, then it must release all the resources having larger number and vice versa.
- This approach is the most practical approach and implementable.
- However, this approach may cause starvation but will never lead to deadlock.

3 Deadlock Avoidance-

- This strategy involves maintaining a set of data using which a decision is made whether to entertain the new request or not.
- If entertaining the new request causes the system to move in an unsafe state, then it is discarded.
- This strategy requires that every process declares its maximum requirement of each resource type in the beginning.
- The main challenge with this approach is predicting the requirement of the processes before execution.
- **Banker's Algorithm** is an example of a deadlock avoidance strategy.

Banker's Algorithms

Following Data structures are used to implement the Banker's Algorithm:

Let ' n ' be the number of processes in the system and ' m ' be the number of resources types.

Available :

- It is a 1-d array of size ' m ' indicating the number of available resources of each type.
- $Available[j] = k$ means there are ' k ' instances of resource type R_j

Max :

- It is a 2-d array of size ' $n*m$ ' that defines the maximum demand of each process in a system.
- $Max[i, j] = k$ means process P_i may request at most ' k ' instances of resource type R_j .

Allocation :

- It is a 2-d array of size ' $n*m$ ' that defines the number of resources of each type currently allocated to each process.
- $Allocation[i, j] = k$ means process P_i is currently allocated ' k ' instances of resource type R_j

Need :

- It is a 2-d array of size ' $n*m$ ' that indicates the remaining resource need of each process.
- $Need[i, j] = k$ means process P_i currently need ' k ' instances of resource type R_j for its execution.
- $Need[i, j] = Max[i, j] - Allocation[i, j]$

Allocation_i specifies the resources currently allocated to process **P_i** and **Need_i** specifies the additional resources that process **P_i** may still request to complete its task.

Banker's algorithm consists of Safety algorithm and Resource request algorithm

Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state can be described as follows

1) Let Work and Finish be vectors of length 'm' and 'n' respectively.

Initialize: Work = Available

Finish[i] = false; for i=1, 2, 3, 4....n

2) Find an i such that both

a) Finish[i] = false

b) Need_i <= Work

if no such i exists goto step (4)

3) Work = Work + Allocation[i]

Finish[i] = true

goto step (2)

4) if Finish [i] = true for all i

then the system is in a safe state

Resource-Request Algorithm

Let Request_i be the request array for process P_i. Request_i [j] = k means process P_i wants k instances of resource type R_j. When a request for resources is made by process P_i, the following actions are taken:

1) If $Request_i \leq Need_i$

Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.

2) If $Request_i \leq Available$

Goto 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:

$Available = Available - Request_i$

$Allocation_i = Allocation_i + Request_i$

$Need_i = Need_i - Request_i$

Example:

Considering a system with five processes P_0 through P_4 and three resources of type A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t_0 following snapshot of the system has been taken:

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Question1. What will be the content of the Need matrix?

Need [i, j] = Max [i, j] – Allocation [i, j]

So, the content of Need Matrix is:

Process	Need		
	A	B	C
P ₀	7	4	3
P ₁	1	2	2
P ₂	6	0	0
P ₃	0	1	1
P ₄	4	3	1

Question2. Is the system in a safe state? If Yes, then what is the safe sequence?

Applying the Safety algorithm on the given system,

Step 1 of Safety Algo

m=3, n=5

Work = Available

Work =

3	3	2
---	---	---

Finish =

false	false	false	false	false
-------	-------	-------	-------	-------

Step 2

For i=3

Need₃ = 0, 1, 1

Finish [3] = false and Need₃ < Work

So P₃ must be kept in safe sequence

Step 3

Work = Work + Allocation₃

Work =

7	5	5
---	---	---

Finish =

true	true	false	true	true
------	------	-------	------	------

Step 2

For i=0

Need₀ = 7, 4, 3

Finish [0] is false and Need₀ > Work

So P₀ must wait

Step 3

Work = Work + Allocation₃

Work =

7	4	3
---	---	---

Finish =

false	true	false	true	false
-------	------	-------	------	-------

Step 2

For i=2

Need₂ = 6, 0, 0

Finish [2] is false and Need₂ > Work

So P₂ must wait

Step 2

For i=1

Need₁ = 1, 2, 2

Finish [1] is false and Need₁ < Work

So P₁ must be kept in safe sequence

Step 2

For i=4

Need₄ = 4, 3, 1

Finish [4] is false and Need₄ < Work

So P₄ must be kept in safe sequence

Step 3

Work = Work + Allocation₂

Work =

10	5	7
----	---	---

Finish =

true	true	true	true	true
------	------	------	------	------

Step 3

Work = Work + Allocation₄

Work =

7	4	5
---	---	---

Finish =

false	true	false	true	true
-------	------	-------	------	------

Step 4

Finish [i] = true for 0 ≤ i ≤ n

Hence the system is in Safe state

Step 2

For i=0

Need₀ = 7, 4, 3

Finish [0] is false and Need < Work

So P₀ must be kept in safe sequence

The safe sequence is P₁, P₃, P₄, P₀, P₂

Question3. What will happen if process P₁ requests one additional instance of resource type A and two instances of resource type C?

A B C
Request₁ = 1, 0, 2

To decide whether the request is granted we use Resource Request algorithm

Step 1
1, 0, 2 1, 2, 2 ✓
Request₁ < Need₁

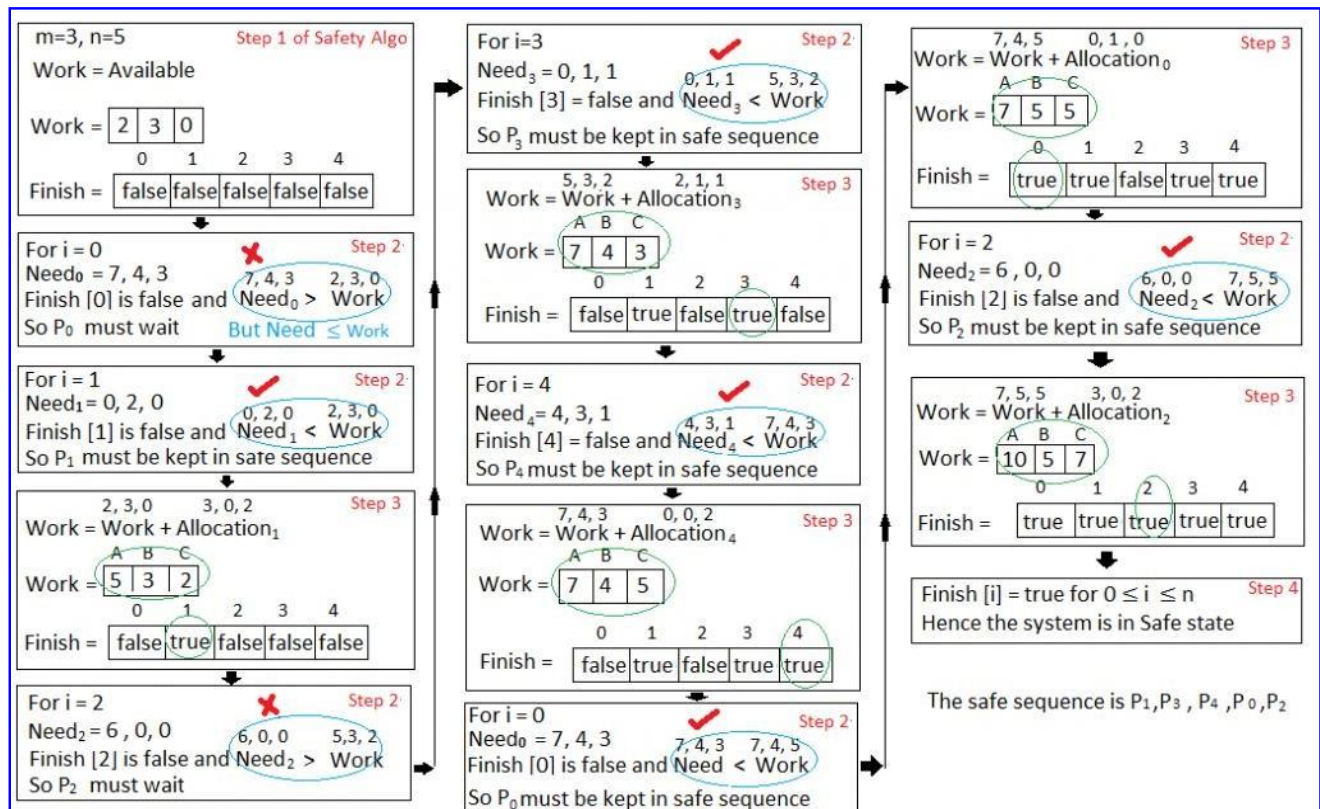
Step 2
1, 0, 2 3, 3, 2 ✓
Request₁ < Available

Step 3

Available = Available – Request₁
Allocation₁ = Allocation₁ + Request₁
Need₁ = Need₁ - Request₁

Process	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P ₀	0	1	0	7	4	3	2	3	0
P ₁	3	0	2	0	2	0			
P ₂	3	0	2	6	0	0			
P ₃	2	1	1	0	1	1			
P ₄	0	0	2	4	3	1			

We must determine whether this new system state is safe. To do so, we again execute Safety algorithm on the above data structures.



Hence the new system state is safe, so we can immediately grant the request for process P₁.

Following is the SAFE Sequence

P1 -> P3 -> P4 -> P0 -> P2

4Deadlock Detection and Recovery-

- This strategy involves waiting until a deadlock occurs.
- After deadlock occurs, the system state is recovered.
- The main challenge with this approach is detecting the deadlock.

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

- Apply an algorithm to examine state of system to determine whether deadlock has occurred or not.
- Apply an algorithm to recover from the deadlock. For more refer- Deadlock Recovery

Deadlock Detection Algorithm/ Bankers Algorithm:

The algorithm employs several time varying data structures:

- **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 a process. Column represents resource and resource represent process.
- **Request-** An $n \times m$ matrix indicates the current request of each process. If $request[i][j]$ equals k then process P_i is requesting k more instances of resource type R_j .

We treat rows in the matrices Allocation and Request as vectors, we refer them as $Allocation_i$ and $Request_j$.

Steps of Algorithm:

1. Let $Work$ and $Finish$ be vectors of length m and n respectively. Initialize $Work = Available$. For $i=0, 1, \dots, n-1$, if $Request_i = 0$, then $Finish[i] = true$; otherwise, $Finish[i] = false$.
2. Find an index i such that both
 - a) $Finish[i] == false$
 - b) $Request_i \leq Work$If no such i exists go to step 4.
3. $Work = Work + Allocation_i$
 $Finish[i] = true$
Go to Step 2.

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

For example,

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

- In this, $Work = [0, 0, 0]$ &
 $Finish = [false, false, false, false, false]$
- $i=0$ is selected as both $Finish[0] = false$ and $[0, 0, 0] \leq [0, 0, 0]$.
- $Work = [0, 0, 0] + [0, 1, 0] \Rightarrow [0, 1, 0]$ &
 $Finish = [true, false, false, false, false]$.
- $i=2$ is selected as both $Finish[2] = false$ and $[0, 0, 0] \leq [0, 1, 0]$.
- $Work = [0, 1, 0] + [3, 0, 3] \Rightarrow [3, 1, 3]$ &
 $Finish = [true, false, true, false, false]$.
- $i=1$ is selected as both $Finish[1] = false$ and $[2, 0, 2] \leq [3, 1, 3]$.
- $Work = [3, 1, 3] + [2, 0, 2] \Rightarrow [5, 1, 5]$ &
 $Finish = [true, true, true, false, false]$.
- $i=3$ is selected as both $Finish[3] = false$ and $[1, 0, 0] \leq [5, 1, 5]$.
- $Work = [5, 1, 5] + [2, 1, 1] \Rightarrow [7, 2, 6]$ &
 $Finish = [true, true, true, true, false]$.
- $i=4$ is selected as both $Finish[4] = false$ and $[0, 0, 2] \leq [7, 2, 6]$.
- $Work = [7, 2, 6] + [0, 0, 2] \Rightarrow [7, 2, 8]$ &
 $Finish = [true, true, true, true, true]$.
- Since $Finish$ is a vector of all true it means **there is no deadlock** in this example.

5 Deadlock Recovery

When a Deadlock Detection Algorithm determines that a deadlock has occurred in the system, the system must recover from that deadlock. There are two approaches of breaking a Deadlock:

1. Process Termination:

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

- **(a). Abort all the Deadlocked Processes:**

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

- **(b). Abort one process at a time until deadlock is eliminated:**

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

2. Resource Preemption:

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

- **(a). Selecting a victim:**

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

- **(b). Rollback:**

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

- **(c). Starvation:**

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

Resource Allocation Graph (RAG).

As Banker's algorithm using some kind of table like allocation, request, available all that thing to understand what is the state of the system. Similarly, if you want to understand the state of the system instead of using those table, actually tables are very easy to represent and understand it, but then still you could even represent the same information in the graph. That graph is called **Resource Allocation Graph (RAG).**

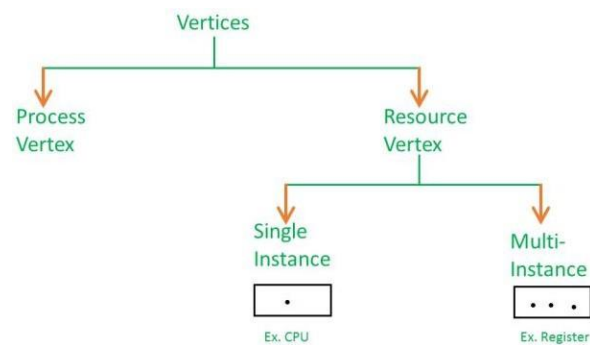
So, resource allocation graph is explained to us what is the state of the system in terms of **processes and resources**. Like how many resources are available, how many are allocated and what is the request of each process. Everything can be represented in terms of the diagram. One of the advantages of having a diagram is, sometimes it is possible to see a deadlock directly by using RAG, but then you might not be able to know that by looking at the table. But the tables

are better if the system contains lots of process and resource and Graph is better if the system contains less number of process and resource. We know that any graph contains vertices and edges. So RAG also contains vertices and edges. In RAG vertices are two type –

1. Process vertex – Every process will be represented as a process vertex. Generally, the process will be represented with a circle.

2. Resource vertex – Every resource will be represented as a resource vertex. It is also two type

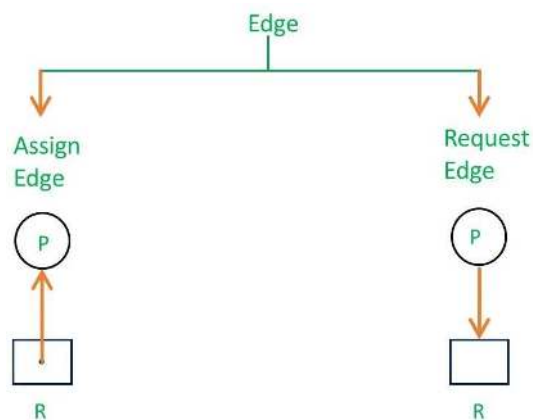
- **Single instance type resource** – It represents as a box, inside the box, there will be one dot. So the number of dots indicate how many instances are present of each resource type.
- **Multi-resource instance type resource** – It also represents as a box, inside the box, there will be many dots.



Now coming to the edges of RAG. There are two types of edges in RAG –

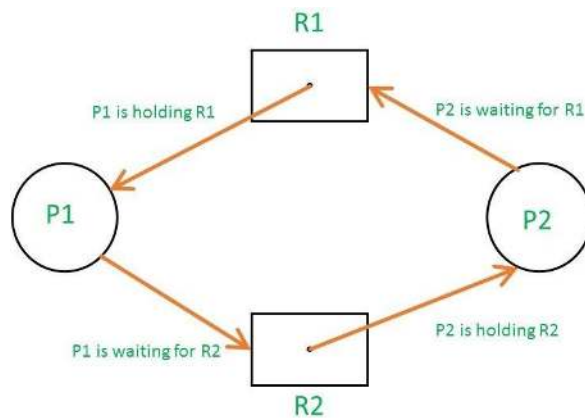
1. Assign Edge – If you already assign a resource to a process then it is called Assign edge.

2. Request Edge – It means in future the process might want some resource to complete the execution, that is called request edge.



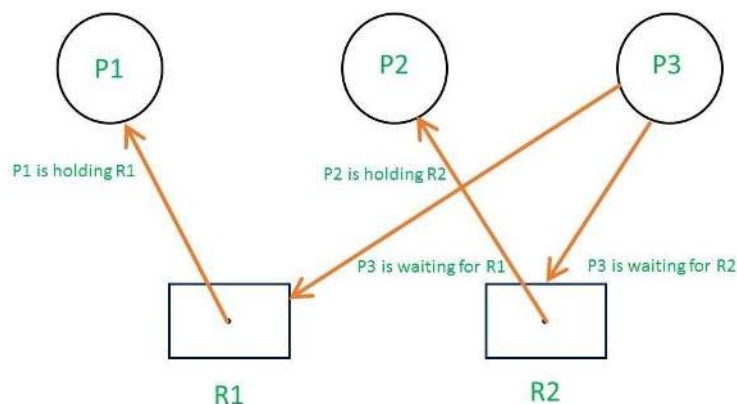
So, if a process is using a resource, an arrow is drawn from the resource node to the process node. If a process is requesting a resource, an arrow is drawn from the process node to the resource node.

Example 1 (Single instances RAG) –



SINGLE INSTANCE RESOURCE TYPE WITH DEADLOCK

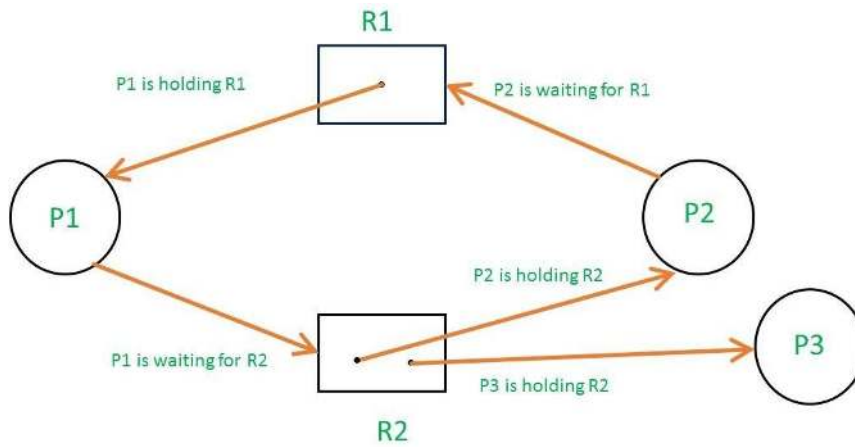
If there is a cycle in the Resource Allocation Graph and each resource in the cycle provides only one instance, then the processes will be in deadlock. For example, if process P1 holds resource R1, process P2 holds resource R2 and process P1 is waiting for R2 and process P2 is waiting for R1, then process P1 and process P2 will be in deadlock.



SINGLE INSTANCE RESOURCE TYPE WITHOUT DEADLOCK

Here's another example, that shows Processes P1 and P2 acquiring resources R1 and R2 while process P3 is waiting to acquire both resources. In this example, there is no circular dependency. So cycle in single-instance resource type is the sufficient condition for deadlock.

Example 2 (Multi-instances RAG) –



MULTI INSTANCES WITHOUT DEADLOCK

From the above example, it is not possible to say the RAG is in a safe state or in an unsafe state. So to see the state of this RAG, let's construct the allocation matrix and request matrix.

Process	Allocation		Request	
	Resource		Resource	
	R1	R2	R1	R2
P1	1	0	0	1
P2	0	1	1	0
P3	0	1	0	0

- The total number of processes are three; P1, P2 & P3 and the total number of resources are two; R1 & R2.

Allocation matrix –

- For constructing the allocation matrix, just go to the resources and see to which process it is allocated.

- R1 is allocated to P1, therefore write 1 in allocation matrix and similarly, R2 is allocated to P2 as well as P3 and for the remaining element just write 0.

Request matrix –

- In order to find out the request matrix, you have to go to the process and see the outgoing edges.
- P1 is requesting resource R2, so write 1 in the matrix and similarly, P2 requesting R1 and for the remaining element write 0.

So now available resource is = (0, 0).

Checking deadlock (safe or not) –

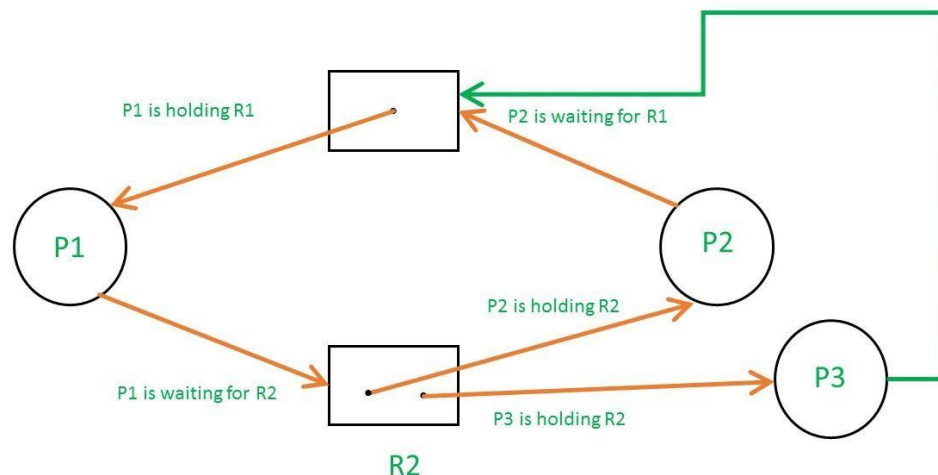
Available =	0	0	(As P3 does not require any extra resource to complete the execution and after completion P3 release its own resource)
P3	0	1	

New Available =	0	1	(As using new available resource we can satisfy the requirement of process P1 and P1 also release its previous resource)
P1	1	0	

New Available =	1	1	(Now easily we can satisfy the requirement of process P2)
P2	0	1	

New Available =	1	2	
-----------------	---	---	--

So, there is no deadlock in this RAG. Even though there is a cycle, still there is no deadlock. Therefore in multi-instance resource cycle is not sufficient condition for deadlock.



MULTI INSTANCES WITH DEADLOCK

Above example is the same as the previous example except that, the process P3 requesting for resource R1.

So the table becomes as shown in below.

Process	Allocation		Request	
	Resource		Resource	
	R1	R2	R1	R2
P1	1	0	0	1
P2	0	1	1	0
P3	0	1	1	0

So, the Available resource is = (0, 0), but requirements are (0, 1), (1, 0) and (1, 0). So you can't fulfill any one requirement. Therefore, it is in deadlock.

Therefore, every cycle in a multi-instance resource type graph is not a deadlock, if there has to be a deadlock, there has to be a cycle. So, in case of RAG with multi-instance resource type, the cycle is a necessary condition for deadlock, but not sufficient.

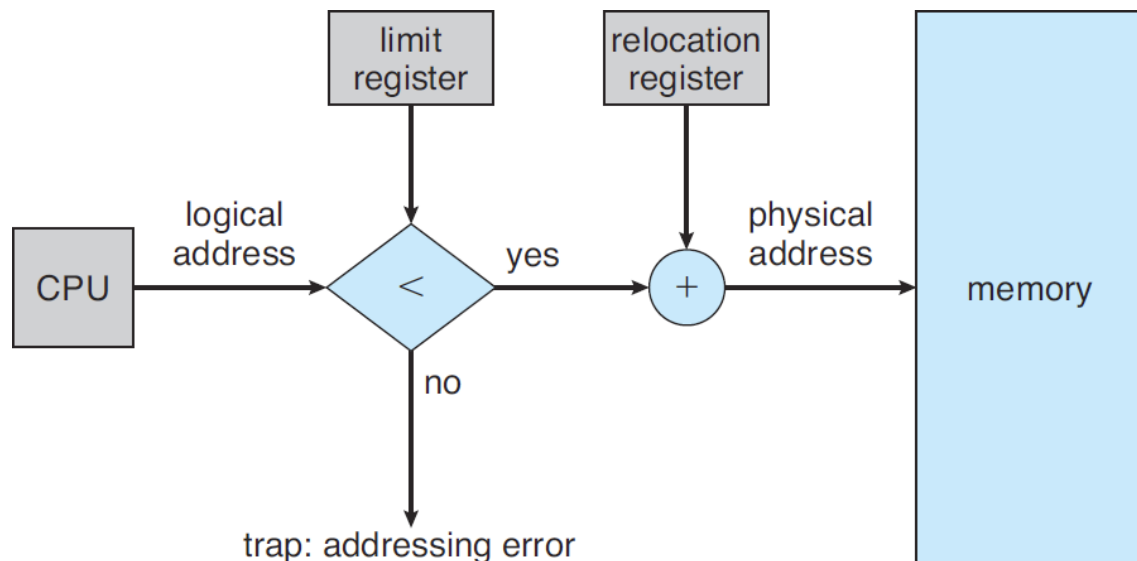
UNIT 6: Memory Management

What is Memory Management?

Answer:

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

Address Translation From Logical Address to Physical Address



Logical Address is generated by CPU while a program is running. The logical address is virtual address as it does not exist physically, therefore, it is also known as Virtual Address. This address is used as a reference to access the physical memory location by CPU. The term Logical Address Space is used for the set of all logical addresses generated by a program's perspective. The hardware device called Memory-Management Unit is used for mapping logical address to its corresponding physical address.

Physical Address identifies a physical location of required data in a memory. The user never directly deals with the physical address but can access by its corresponding logical address. The user program generates the logical address and thinks that the program is running in this logical address but the program needs physical memory for its execution, therefore, the logical address

must be mapped to the physical address by MMU before they are used. The term Physical Address Space is used for all physical addresses corresponding to the logical addresses in a Logical address space.

For Example

1. CPU will generate logical address for eg: 346
2. MMU will generate relocation register(base register) for eg:14000
3. In Memory physical address is located eg:(346+14000= 14346)

Instruction-execution cycle Follows steps:

1. First instruction is fetched from memory e.g. ADD A,B
2. Then these instructions are decoded i.e., Addition of A and B
3. And further loading or storing at some particular memory location takes place.

Basic Hardware

As main memory and registers are built into processor and CPU can access these only. So every instructions should be written in direct access storage devices.

1. If CPU access instruction from register then it can be done in one CPU clock cycle as registers are built into CPU.
2. If instruction resides in main memory then it will be accessed via memory bus that will take lot of time. So remedy to this add fast memory in between CPU and main memory i.e. adding cache for transaction.
3. Now we should insure that process resides in legal address.
4. Legal address consists of base register(holds smallest physical address) and limit register(size of range).

For example:

Base register = 300040

limit register = 120900

then legal address = (300040+120900)= 420940(inclusive).

legal address = base register+ limit register

What is the contiguous and non contiguous memory allocation ?

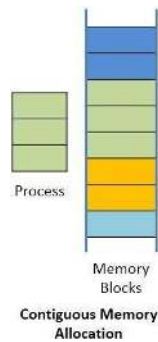
Answer:

Memory is a large array of bytes, where each byte has its own address. The memory allocation can be classified into two methods contiguous memory allocation and non-contiguous memory allocation. The major difference between Contiguous and Noncontiguous memory allocation is that the **contiguous memory allocation** assigns the consecutive blocks of memory to a process requesting for memory whereas, the **noncontiguous memory allocation** assigns the separate memory blocks at the different location in memory space in a nonconsecutive manner to a process requesting for memory. We will discuss some more differences between contiguous and non-contiguous memory allocation with the help of comparison chart shown below.

Definition of Contiguous Memory Allocation

The operating system and the user's processes both must be accommodated in the main memory. Hence the main memory is **divided into two** partitions: at one partition the operating system resides and at other the user processes reside. In usual conditions, the several user processes must reside in the memory at the same time, and therefore, it is important to consider the allocation of memory to the processes.

The Contiguous memory allocation is one of the methods of memory allocation. In contiguous memory allocation, when a process requests for the memory, a **single contiguous section of memory blocks** is assigned to the process according to its requirement.



The contiguous memory allocation can be achieved by dividing the memory into the fixed-sized **partition** and allocate each partition to a single process only. But this will cause the degree of multiprogramming, bounding to the number of fixed partition done in the memory. The contiguous memory allocation also leads to the **internal fragmentation**. Like, if a fixed sized memory block allocated to a process is slightly larger than its requirement then the left over memory space in the block is called internal fragmentation. When the process residing in the partition terminates the partition becomes available for the another process.

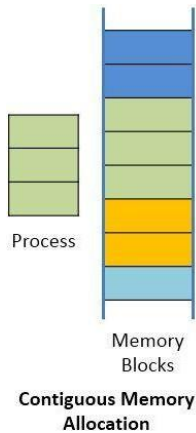
BASIS THE COMPARISON	CONTIGUOUS MEMORY ALLOCATION	NONCONTIGUOUS MEMORY ALLOCATION
Basic	Allocates consecutive blocks of memory to a process.	Allocates separate blocks of memory to a process.
Overheads	Contiguous memory allocation does not have the overhead of address translation while execution of a process.	Noncontiguous memory allocation has overhead of address translation while execution of a process.
Execution rate	A process executes faster in contiguous memory allocation	A process executes quite slower comparatively in noncontiguous memory allocation.
Solution	The memory space must be divided into the fixed-sized partition and each partition is allocated to a single process only.	Divide the process into several blocks and place them in different parts of the memory according to the availability of memory space available.
Table	A table is maintained by operating system which maintains the list of available and occupied partition in the	A table has to be maintained for each process that carries the base addresses of each block which has been acquired by a process in memory.

Definition of Contiguous Memory Allocation

The operating system and the user's processes both must be accommodated in the main memory. Hence the main memory is **divided into two** partitions: at one partition the operating system resides and at other the user processes reside. In usual conditions, the several user processes must reside in the memory at the same time, and therefore, it is important to consider the allocation of memory to the processes.

The Contiguous memory allocation is one of the methods of memory allocation. In contiguous memory allocation, when a process requests for the memory, a **single contiguous section of memory blocks** is assigned to the process according to its requirement.

The contiguous memory allocation can be achieved by dividing the memory into the fixed-sized **partition** and allocate each partition to a single process only. But this will cause the degree of multiprogramming, bounding to the number of fixed partition done in the memory. The contiguous memory allocation also leads to the **internal fragmentation**. Like, if a fixed sized memory block allocated to a process is slightly larger than its requirement then the left over memory space in the block is called internal fragmentation. When the process residing in the partition terminates the partition becomes available for the another process.



Definition Non-Contiguous Memory Allocation

The Non-contiguous memory allocation allows a process to **acquire the several memory blocks at the different location in the memory** according to its requirement. The noncontiguous memory allocation also **reduces the memory wastage** caused due to internal and external fragmentation. As it utilizes the memory holes, created during internal and external fragmentation.

Paging and segmentation are the two ways which allow a process's physical address space to be non-contiguous. In non-contiguous memory allocation, the process is divided into **blocks** (pages or segments) which are placed into the different area of memory space according to the availability of the memory.

The noncontiguous memory allocation has an advantage of reducing memory wastage but, but it **increases the overheads** of address translation. As the parts of the process are placed in a different location in memory, it **slows the execution** of the memory because time is consumed in address translation.

What is paging?

Answer:

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

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

Example:

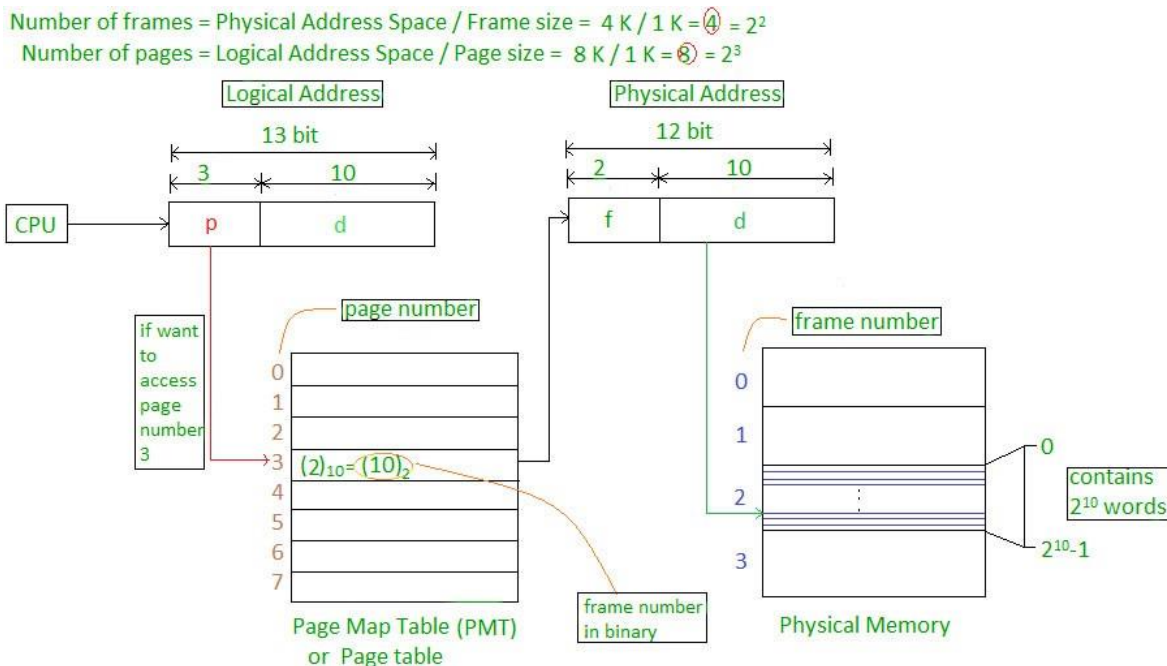
- If Logical Address = 31 bit, then Logical Address Space = 2^{31} words = 2 G words (1 G = 2^{30})
- If Logical Address Space = 128 M words = $2^7 * 2^{20}$ words, then Logical Address = $\log_2 2^{27} = 27$ bits
- If Physical Address = 22 bit, then Physical Address Space = 2^{22} words = 4 M words (1 M = 2^{20})
- If Physical Address Space = 16 M words = $2^4 * 2^{20}$ words, then Physical Address = $\log_2 2^{24} = 24$ bits

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

- The Physical Address Space is conceptually divided into a number of fixed-size blocks, called **frames**.
- The Logical address Space is also splitted into fixed-size blocks, called **pages**.
- Page Size = Frame Size

Let us consider an example:

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



Address generated by CPU is divided into

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

Physical Address is divided into

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

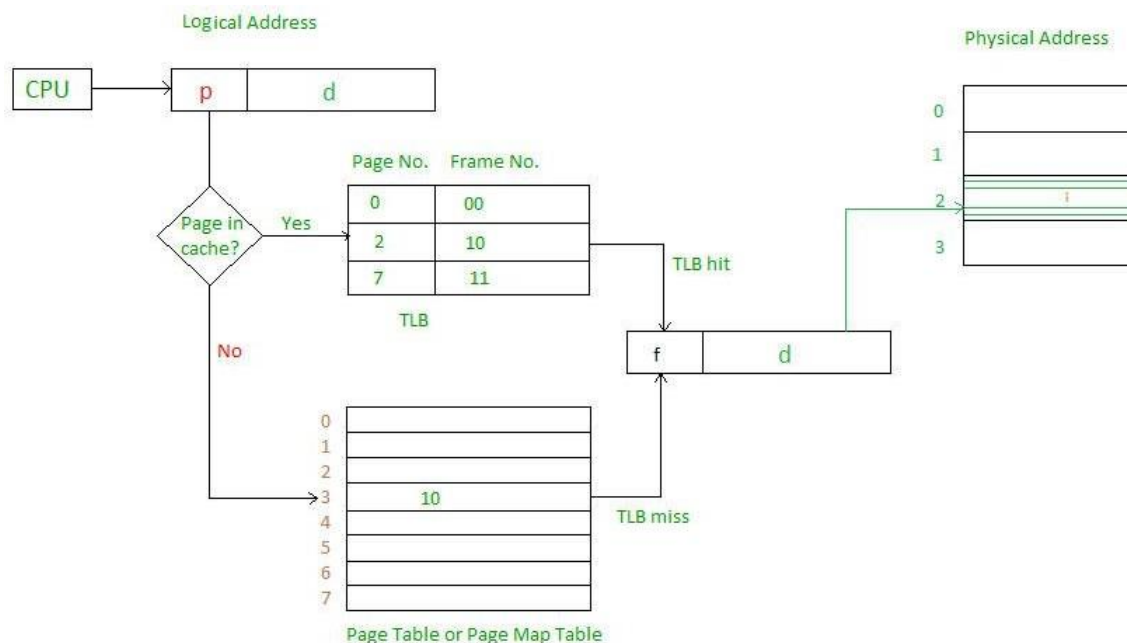
What is TLB?

Answer:

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

The TLB is associative, high speed memory.

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



Main memory access time = m

If page table are kept in main memory,

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

TLB access time = c

TLB hit ratio = x, then miss ratio = (1-x)

When hit occurs

Effective access time = hit ratio * (c+m) + miss ratio * (c+m+m)

For page table access

for main memory access

What is Virtual Memory?

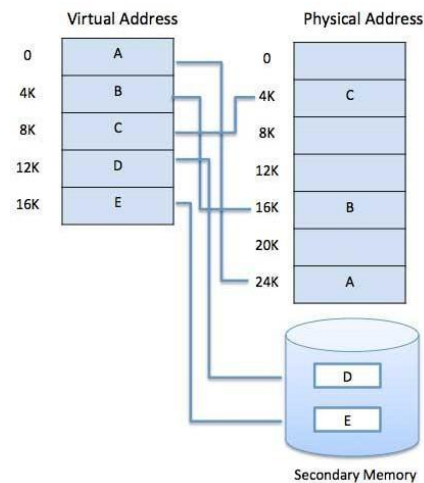
A computer can address more memory than the amount physically installed on the system. This extra memory is actually called **virtual memory** and it is a section of a hard disk that's set up to emulate the computer's RAM.

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

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

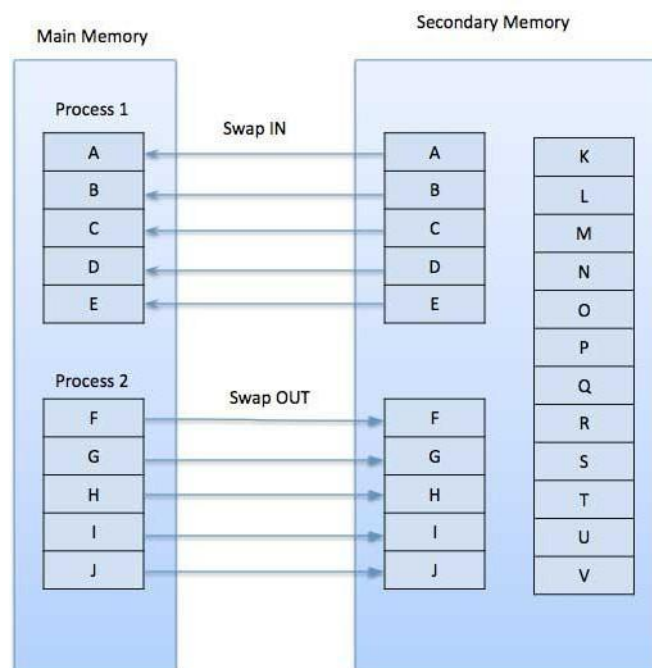
- User written error handling routines are used only when an error occurred in the data or computation.
- Certain options and features of a program may be used rarely.
- Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.
- The ability to execute a program that is only partially in memory would counter many benefits.
- Less number of I/O would be needed to load or swap each user program into memory.
- A program would no longer be constrained by the amount of physical memory that is available.
- **Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.**

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



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

What is Demand Paging?



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

executing the new program after loading the first page and fetches that program's pages as they are referenced.

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

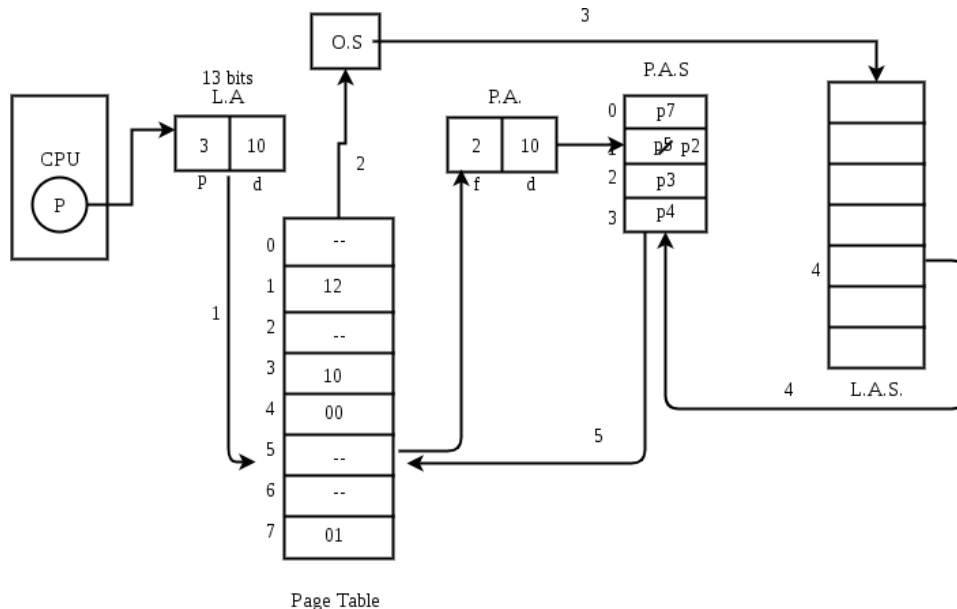
Advantages

Following are the advantages of Demand Paging –

- Large virtual memory.
- More efficient use of memory.
- There is no limit on degree of multiprogramming.

Disadvantages

- Number of tables and the amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.



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

6. The signal will be sent to the CPU to continue the program execution and it will place the process back into ready state.

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

Page Replacement algorithm.

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

Page Fault – A page fault happens when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory.

Since actual physical memory is much smaller than virtual memory, page faults happen. In case of page fault, Operating System might have to replace one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

Page Replacement Algorithms :

- **First In First Out (FIFO) –**

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

Page reference stream:																
1	2	3	2	1	5	2	1	6	2	5	6	3	1	3	6	1
2	4	3														
1	1	1	1	1	2	2	3	5	1	6	6	2	5	5	3	3
	2	2	2	2	3	3	5	1	6	2	2	5	3	3	1	1
		3	3	3	5	5	1	6	2	5	5	3	1	1	6	6
*	*	*			*		*	*	*	*		*	*		*	*
FIFO																
Total 14 page faults																

Optimal Page replacement –

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.

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

Page reference stream:

1 2 3 2 1 5 2 1 6 2 5 6 3 1 3 6 1 2 4 3

1	1	1	1	1	1	1	1	6	6	6	6	6	6	6	6	2	2	2
	2	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1	4	4
		3	3	3	5	5	5	5	5	5	5	3	3	3	3	3	3	3
*	*	*			*			*				*	*			*	*	

Optimal

Total 9 page faults

Least Recently Used (LRU) algorithm

- Page which has not been used for the longest time in main memory is the one which will be selected for replacement.

• E

Page reference stream:

1 2 3 2 1 5 2 1 6 2 5 6 3 1 3 6 1 2 4 3

1	1	1	1	3	2	1	5	2	1	6	2	5	6	6	1	3	6	1	2
	2	2	3	2	1	5	2	1	6	2	5	6	3	1	3	6	1	2	4
		3	2	1	5	2	1	6	2	5	6	3	1	3	6	1	2	4	3
*	*	*			*			*		*		*	*			*	*	*	

LRU

Total 11 page faults

e

nt, keep a list, replace pages by looking back into time.

UNIT 7 : I/O MANAGEMENT

Draw the block diagram for DMA. Write steps for DMA data transfer.

A device controller need not necessarily control a single device. It can usually control multiple I/O devices. It comes in the form of an electronic circuit board that plugs directly into the system bus, and there is a cable from the controller to each device it controls. The cables coming out of the controller are usually terminated at the back panel of the main computer box in the form of connectors known as ports.

elow illustrates how I/O devices are connected to a computer system through device controllers. Please note the following points in the diagram:

- Each I/O device is linked through a hardware interface called I/O Port.
- Single and Multi-port device controls single or multi-devices.
- The communication between I/O controller and Memory is through bus only in case of Direct Memory Access (DMA), whereas the path passes through the CPU for such communication in case of non-DMA.

Using device controllers for connecting I/O devices to a computer system instead of connecting them directly to the system bus has the following advantages:

- A device controller can be shared among multiple I/O devices allowing many I/O devices to be connected to the system.

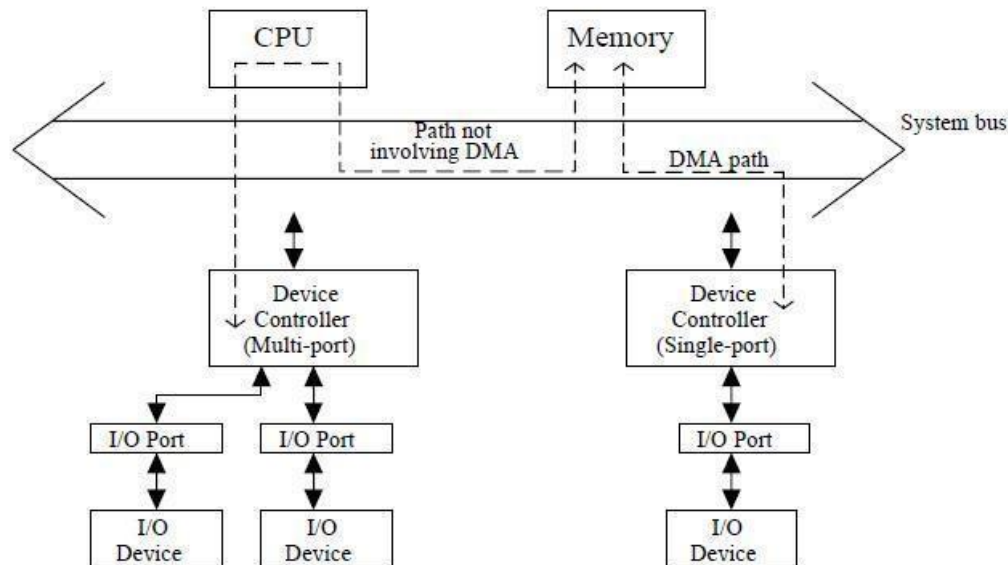


Figure 2: Connecting I/O Devices using Device Controller

- I/O devices can be easily upgraded or changed without any change in the computer system.

- I/O devices of manufacturers other than the computer manufacturer can be easily plugged in to the computer system. This provides more flexibility to the users in buying I/O devices of their choice.

Write short note: RAID levels.

RAID, or “Redundant Arrays of Independent Disks” is a technique which makes use of a combination of multiple disks instead of using a single disk for increased performance, data redundancy or both. The term was coined by David Patterson, Garth A. Gibson, and Randy Katz at the University of California, Berkeley in 1987.

Why data redundancy?

Data redundancy, although taking up extra space, adds to disk reliability. This means, in case of disk failure, if the same data is also backed up onto another disk, we can retrieve the data and go on with the operation. On the other hand, if the data is spread across just multiple disks without the RAID technique, the loss of a single disk can affect the entire data.

Key evaluation points for a RAID System

- **Reliability:** How many disk faults can the system tolerate?
- **Availability:** What fraction of the total session time is a system in uptime mode, i.e. how available is the system for actual use?
- **Performance:** How good is the response time? How high is the throughput (rate of processing work)? Note that performance contains a lot of parameters and not just the two.
- **Capacity:** Given a set of N disks each with B blocks, how much useful capacity is available to the user?

RAID is very transparent to the underlying system. This means, to the host system, it appears as a single big disk presenting itself as a linear array of blocks. This allows older technologies to be replaced by RAID without making too many changes in the existing code.

RAID-0 (Striping)

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

In the figure, blocks “0,1,2,3” form a stripe.

- Instead of placing just one block into a disk at a time, we can work with two (or more) blocks placed into a disk before moving on to the next one.

Disk 0	Disk 1	Disk 2	Disk 3
0	3	4	6
1	3	5	7
8	10	12	14
9	11	13	15

Evaluation:

- Reliability: 0
There is no duplication of data. Hence, a block once lost cannot be recovered.
- Capacity: $N*B$
The entire space is being used to store data. Since there is no duplication, N disks each having B blocks are fully utilized.

RAID-1 (Mirroring)

- More than one copy of each block is stored in a separate disk. Thus, every block has two (or more) copies, lying on different disks.

Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

The above figure shows a RAID-1 system with mirroring level 2.

- RAID 0 was unable to tolerate any disk failure. But RAID 1 is capable of reliability.

Evaluation:

Assume a RAID system with mirroring level 2.

- Reliability: 1 to $N/2$

1 disk failure can be handled for certain, because blocks of that disk would have duplicates on some other disk. If we are lucky enough and disks 0 and 2 fail, then again this can be handled as the blocks of these disks have duplicates on disks 1 and 3. So, in the best case, $N/2$ disk failures can be handled.

- Capacity: $N*B/2$
Only half the space is being used to store data. The other half is just a mirror to the already stored data.

RAID-4 (Block-Level Striping with Dedicated Parity)

- Instead of duplicating data, this adopts a parity-based approach.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

In the figure, we can observe one column (disk) dedicated to parity.

- Parity is calculated using a simple XOR function. If the data bits are 0,0,0,1 the parity bit is $XOR(0,0,0,1) = 1$. If the data bits are 0,1,1,0 the parity bit is $XOR(0,1,1,0) = 0$. A simple approach is that even number of ones results in parity 0, and an odd number of ones results in parity 1.

C1	C2	C3	C4	Parity
0	0	0	1	1
0	1	1	0	0

Assume that in the above figure, C3 is lost due to some disk failure. Then, we can recompute the data bit stored in C3 by looking at the values of all the other columns and the parity bit. This allows us to recover lost data.

- **Evaluation:**
 - Reliability: 1
RAID-4 allows recovery of at most 1 disk failure (because of the way parity works). If more than one disk fails, there is no way to recover the data.
 - Capacity: $(N-1)*B$
One disk in the system is reserved for storing the parity. Hence, $(N-1)$ disks are made available for data storage, each disk having B blocks.

- **RAID-5 (Block-Level Striping with Distributed Parity)**
- This is a slight modification of the RAID-4 system where the only difference is that the parity rotates among the drives.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

In the figure, we can notice how the parity bit “rotates”.

- This was introduced to make the random write performance better.
- **Evaluation:**
- Reliability:1
RAID-5 allows recovery of at most 1 disk failure (because of the way parity works). If more than one disk fails, there is no way to recover the data. This is identical to RAID-4.
- Capacity:(N-1)*B
Overall, space equivalent to one disk is utilized in storing the parity. Hence, (N-1) disks are made available for data storage, each disk having B blocks.

Explain of Controller in I/O transfer.

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

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

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

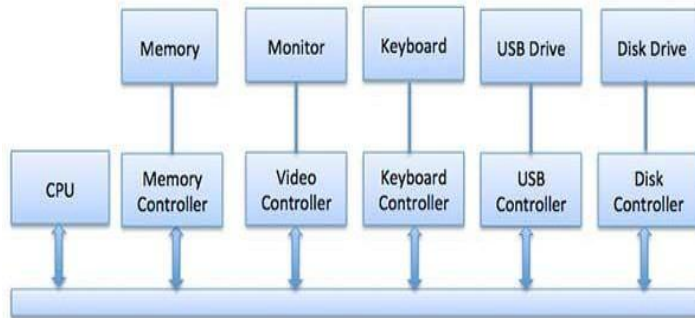
Device Controllers

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

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

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

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



Synchronous vs asynchronous I/O

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

Communication to I/O Devices

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

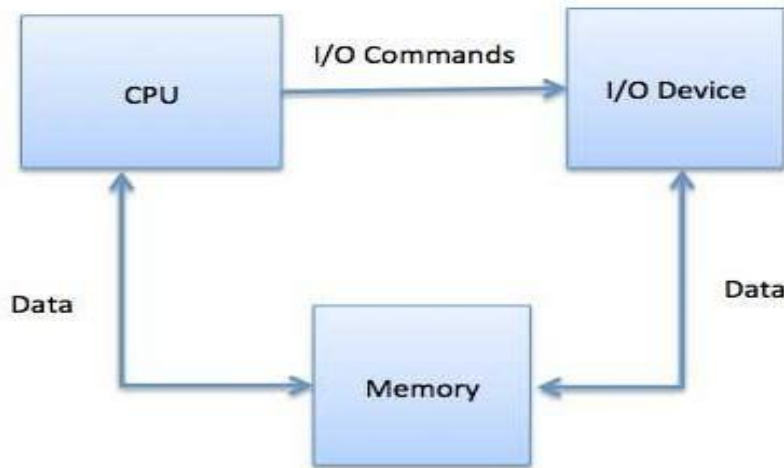
- Special Instruction I/O
- Memory-mapped I/O
- Direct memory access (DMA)

Special Instruction I/O

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

Memory-mapped I/O

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



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

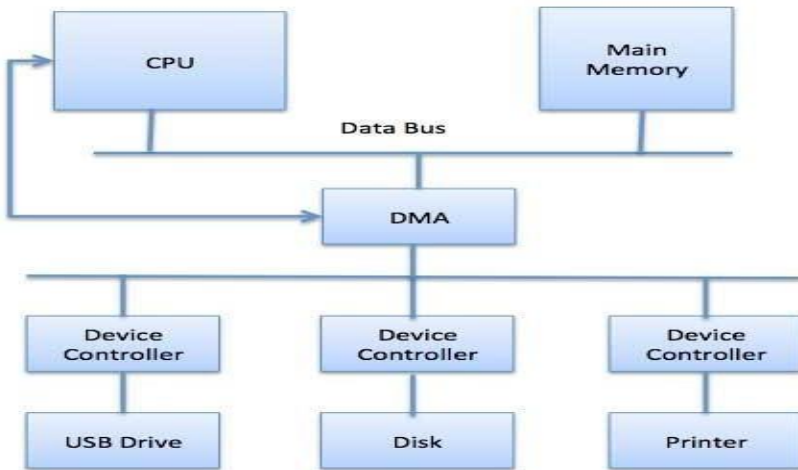
The advantage to this method is that every instruction which can access memory can be used to manipulate an I/O device. Memory mapped IO is used for most high-speed I/O devices like disks, communication interfaces.

6Direct Memory Access (DMA)

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

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

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



The operating system uses the DMA hardware as follows –

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

Polling vs Interrupts I/O

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

Polling I/O

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

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

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

Interrupts I/O

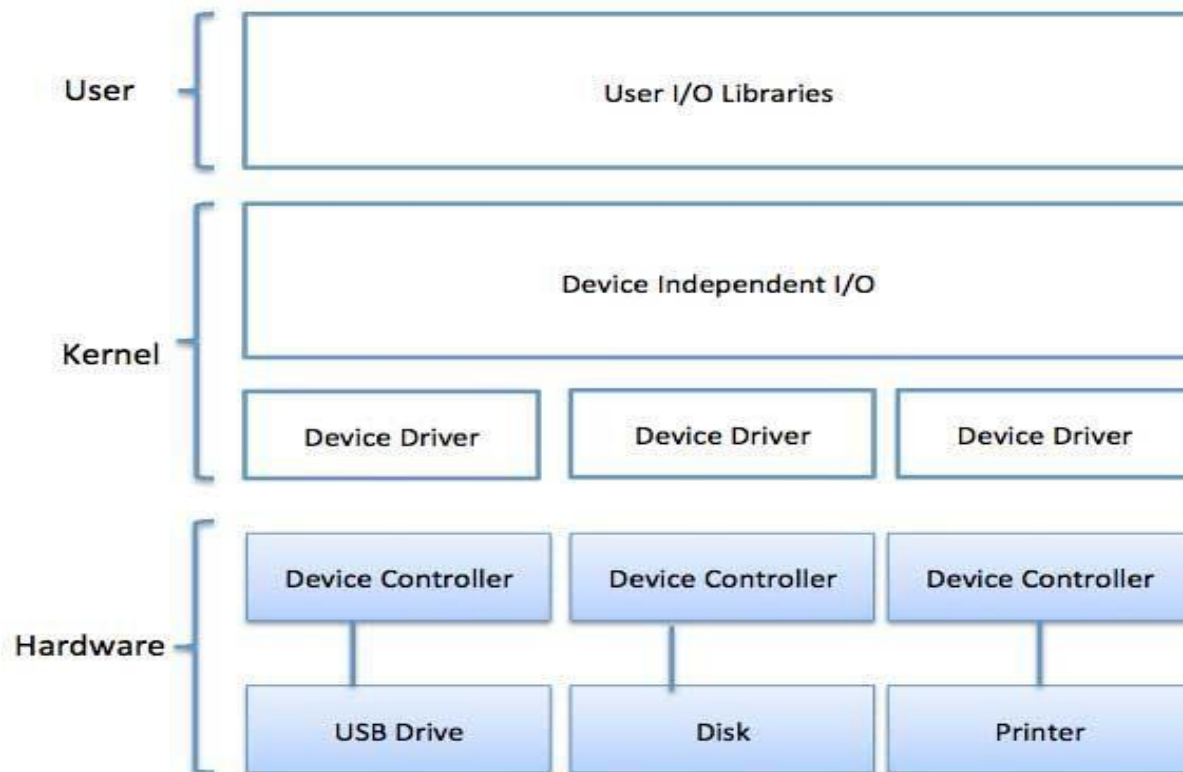
An alternative scheme for dealing with I/O is the interrupt-driven method. An interrupt is a signal to the microprocessor from a device that requires attention.

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

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

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

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



Device Drivers

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

A device driver performs the following jobs –

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

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

Interrupt handlers

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

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

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

Device-Independent I/O Software

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

- Uniform interfacing for device drivers
- Device naming - Mnemonic names mapped to Major and Minor device numbers
- Device protection
- Providing a device-independent block size
- Buffering because data coming off a device cannot be stored in final destination.
- Storage allocation on block devices
- Allocation and releasing dedicated devices
- Error Reporting

User-Space I/O Software

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

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

Kernel I/O Subsystem

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

- **Scheduling** – Kernel schedules a set of I/O requests to determine a good order in which to execute them. When an application issues a blocking I/O system call, the request is placed on the queue for that device. The Kernel I/O scheduler rearranges the order of the queue to improve the overall system efficiency and the average response time experienced by the applications.
- **Buffering** – Kernel I/O Subsystem maintains a memory area known as **buffer** that stores data while they are transferred between two devices or between a device with an application operation. Buffering is done to cope with a speed mismatch between the

producer and consumer of a data stream or to adapt between devices that have different data transfer sizes.

- **Caching** – Kernel maintains cache memory which is region of fast memory that holds copies of data. Access to the cached copy is more efficient than access to the original.
- **Spooling and Device Reservation** – A spool is a buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams. The spooling system copies the queued spool files to the printer one at a time. In some operating systems, spooling is managed by a system daemon process. In other operating systems, it is handled by an in kernel thread.
- **Error Handling** – An operating system that uses protected memory can guard against many kinds of hardware and application errors.

Disk Scheduling Algorithms

Disk scheduling is done by operating systems to schedule I/O requests arriving for the disk. Disk scheduling is also known as I/O scheduling.

Disk scheduling is important because:

- Multiple I/O requests may arrive by different processes and only one I/O request can be served at a time by the disk controller. Thus other I/O requests need to wait in the waiting queue and need to be scheduled.
- Two or more request may be far from each other so can result in greater disk arm movement.
- Hard drives are one of the slowest parts of the computer system and thus need to be accessed in an efficient manner.

There are many Disk Scheduling Algorithms but before discussing them let's have a quick look at some of the important terms:

- **Seek Time**: Seek time is the time taken to locate the disk arm to a specified track where the data is to be read or write. So the disk scheduling algorithm that gives minimum average seek time is better.
- **Rotational Latency**: Rotational Latency is the time taken by the desired sector of disk to rotate into a position so that it can access the read/write heads. So the disk scheduling algorithm that gives minimum rotational latency is better.
- **Transfer Time**: Transfer time is the time to transfer the data. It depends on the rotating speed of the disk and number of bytes to be transferred.
- **Disk Access Time**: Disk Access Time is:

$$\text{Disk Access Time} = \text{Seek Time} + \text{Rotational Latency} + \text{Transfer Time}$$

Disk Response Time: Response Time is the average of time spent by a request waiting to perform its I/O operation. *Average Response time* is the response time of the all requests. *Variance Response Time* is measure of how individual request are serviced with respect to average response time. So the disk scheduling algorithm that gives minimum variance response time is better.

FCFS Disk Scheduling Algorithms

Prerequisite: [Disk scheduling algorithms.](#)

Given an array of disk track numbers and initial head position, our task is to find the total number of seek operations done to access all the requested tracks if **First Come First Serve (FCFS)** disk scheduling algorithm is used.

First Come First Serve (FCFS)

FCFS is the simplest [disk scheduling algorithm](#). As the name suggests, this algorithm entertains requests in the order they arrive in the disk queue. The algorithm looks very fair and there is no starvation (all requests are serviced sequentially) but generally, it does not provide the fastest service.

Algorithm:

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. 'head' is the position of disk head.
2. Let us one by one take the tracks in default order and calculate the absolute distance of the track from the head.
3. Increment the total seek count with this distance.
4. Currently serviced track position now becomes the new head position.
5. Go to step 2 until all tracks in request array have not been serviced.

Example:

Input:

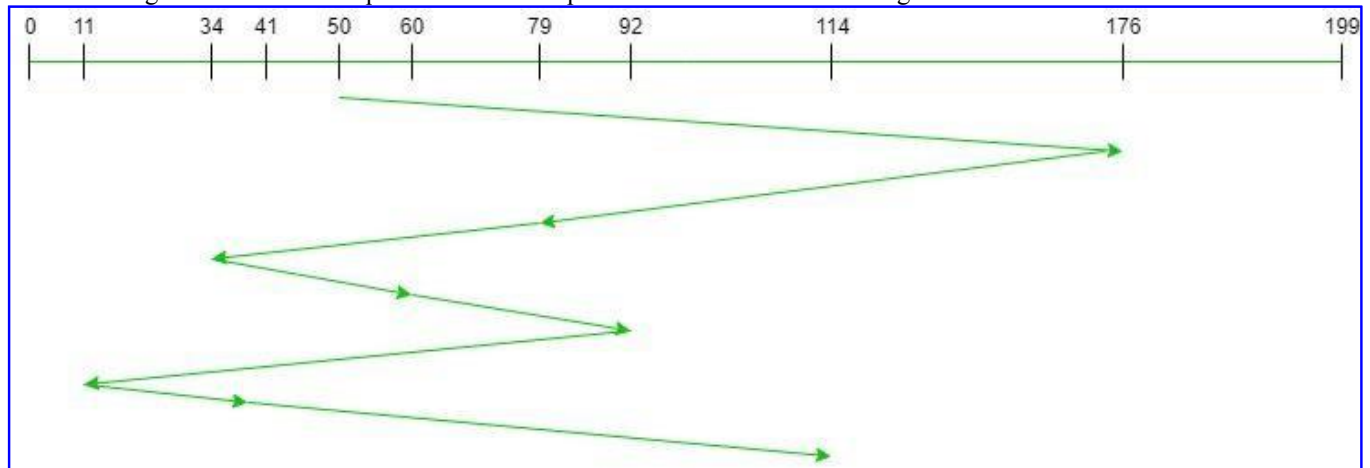
Request sequence = {176, 79, 34, 60, 92, 11, 41, 114}
Initial head position = 50

Output:

Total number of seek operations = 510
Seek Sequence is

176
79
34
60
92
11
41
114

The following chart shows the sequence in which requested tracks are serviced using FCFS.



Therefore, the total seek count is calculated as:

$$\begin{aligned}
 &= (176-50) + (176-79) + (79-34) + (60-34) + (92-60) + (92-11) + (41-11) + (114-41) \\
 &= 510
 \end{aligned}$$

Program for SSTF disk scheduling algorithm

Given an array of disk track numbers and initial head position, our task is to find the total number of seek operations done to access all the requested tracks if **Shortest Seek Time First (SSTF)** is a disk scheduling algorithm is used.

Shortest Seek Time First (SSTF) –

Basic idea is the tracks which are closer to current disk head position should be serviced first in order to *minimise the seek operations*.

Algorithm –

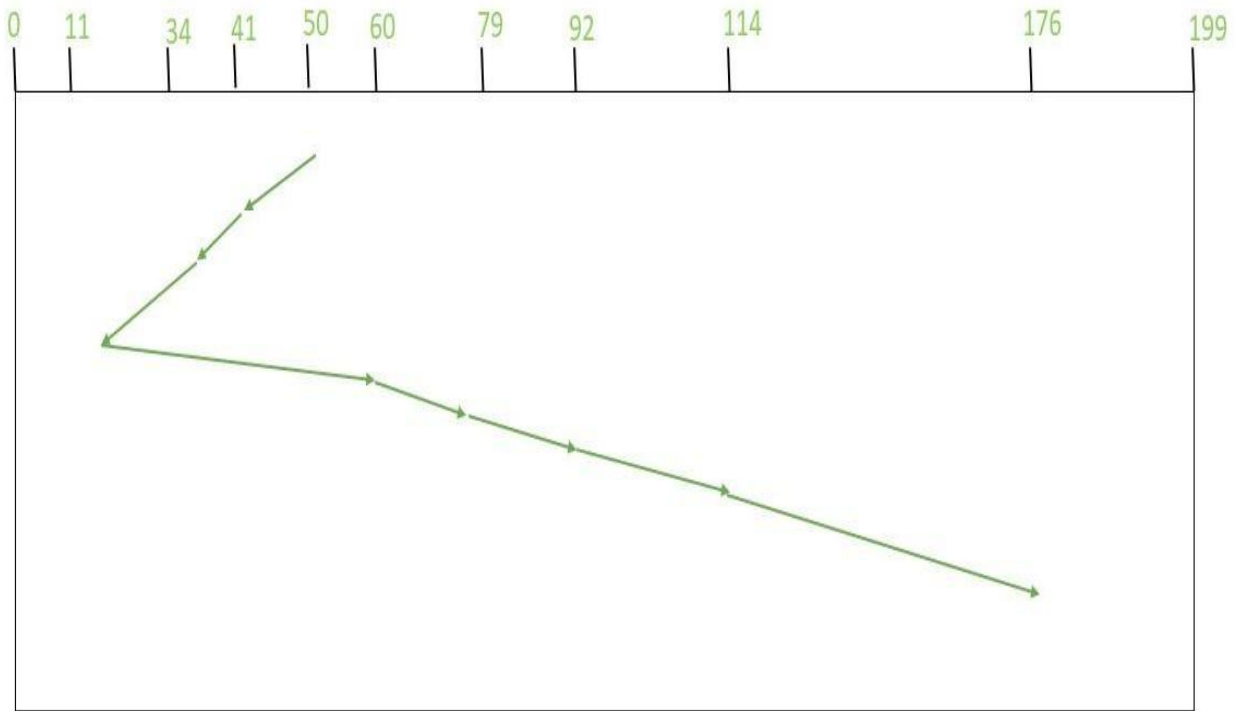
1. Let Request array represents an array storing indexes of tracks that have been requested. 'head' is the position of disk head.
2. Find the positive distance of all tracks in the request array from head.
3. Find a track from requested array which has not been accessed/serviced yet and has minimum distance from head.
4. Increment the total seek count with this distance.
5. Currently serviced track position now becomes the new head position.
6. Go to step 2 until all tracks in request array have not been serviced.

Example –

Request sequence = {176, 79, 34, 60, 92, 11, 41, 114}

Initial head position = 50

The following chart shows the sequence in which requested tracks are serviced using SSTF.



Therefore, total seek count is calculates as:

$$\begin{aligned}
 &= (50-41) + (41-34) + (34-11) + (60-11) + (79-60) + (92-79) + (114-92) + (176-114) \\
 &= 204
 \end{aligned}$$

C-SCAN Disk Scheduling Algorithm

Given an array of disk track numbers and initial head position, our task is to find the total number of seek operations done to access all the requested tracks if C-SCAN disk scheduling algorithm is used.

What is C-SCAN (Circular Elevator) Disk Scheduling Algorithm?

Circular SCAN (C-SCAN) scheduling algorithm is a modified version of SCAN disk scheduling algorithm that deals with the inefficiency of SCAN algorithm by servicing the requests more uniformly. Like SCAN (Elevator Algorithm) C-SCAN moves the head from one end servicing all the requests to the other end. However, as soon as the head reaches the other end, it immediately returns to the beginning of the disk without servicing any requests on the return trip (see chart below) and starts servicing again once reaches the beginning. This is also known as the “Circular Elevator Algorithm” as it essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.

Algorithm:

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. 'head' is the position of disk head.
2. The head services only in the right direction from 0 to size of the disk.
3. While moving in the left direction do not service any of the tracks.
4. When we reach at the beginning(left end) reverse the direction.
5. While moving in right direction it services all tracks one by one.
6. While moving in right direction calculate the absolute distance of the track from the head.
7. Increment the total seek count with this distance.
8. Currently serviced track position now becomes the new head position.
9. Go to step 6 until we reach at right end of the disk.
10. If we reach at the right end of the disk reverse the direction and go to step 3 until all tracks in request array have not been serviced.

Examples:**Input:**

Request sequence = {176, 79, 34, 60, 92, 11, 41, 114}

Initial head position = 50

Output:

Initial position of head: 50

Total number of seek operations = 190

Seek Sequence is

60

79

92

114

176

199

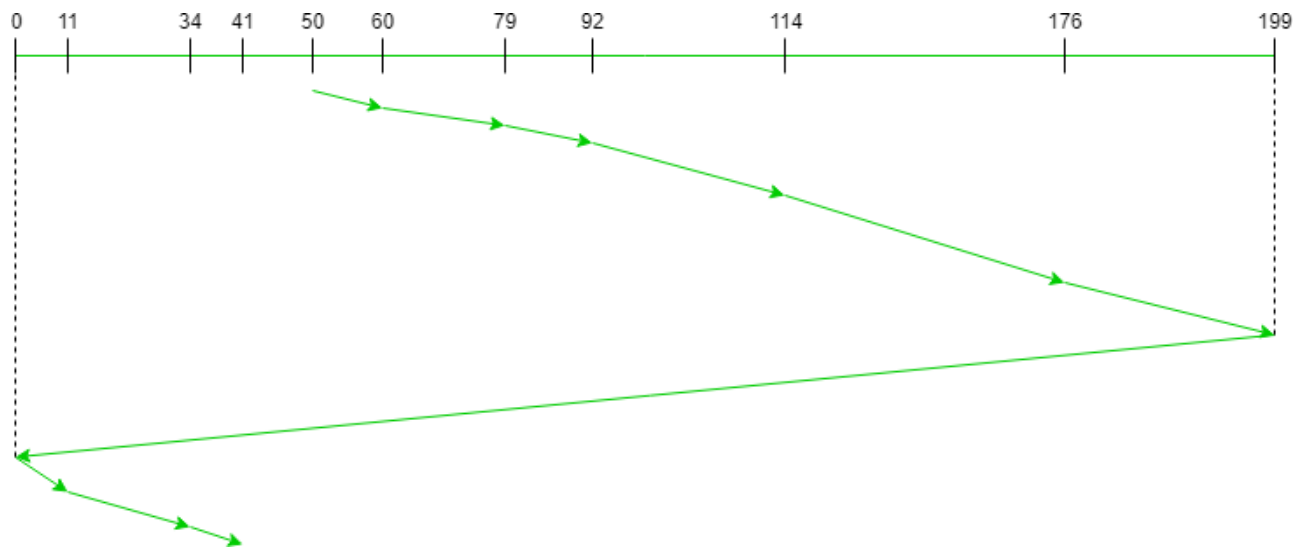
0

11

34

41

The following chart shows the sequence in which requested tracks are serviced using SCAN.



Therefore, the total seek count is calculated as:

$$\begin{aligned}
 &= (60-50) + (79-60) + (92-79) \\
 &\quad + (114-92) + (176-114) + (199-176) + (199-0) \\
 &\quad + (11-0) + (34-11) + (41-34)
 \end{aligned}$$

LOOK Disk Scheduling Algorithm:

Given an array of disk track numbers and initial head position, our task is to find the total number of seek operations done to access all the requested tracks if *LOOK* disk scheduling algorithm is used. Also, write a program to find the seek sequence using *LOOK* disk scheduling algorithm.

LOOK **Disk** **Scheduling** **Algorithm:**
 LOOK is the advanced version of SCAN (elevator) disk scheduling algorithm which gives slightly better seek time than any other algorithm in the hierarchy (*FCFS* → *SRTF* → *SCAN* → *C-SCAN* → *LOOK*). The LOOK algorithm services request similarly as SCAN algorithm meanwhile it also “looks” ahead as if there are more tracks that are needed to be serviced in the same direction. If there are no pending requests in the moving direction the head reverses the direction and start servicing requests in the opposite direction.

The main reason behind the better performance of LOOK algorithm in comparison to SCAN is because in this algorithm the head is not allowed to move till the end of the disk.

Algorithm:

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. ‘head’ is the position of disk head.
2. The initial direction in which head is moving is given and it services in the same direction.

3. The head services all the requests one by one in the direction head is moving.
4. The head continues to move in the same direction until all the request in this direction are not finished.
5. While moving in this direction calculate the absolute distance of the track from the head.
6. Increment the total seek count with this distance.
7. Currently serviced track position now becomes the new head position.
8. Go to step 5 until we reach at last request in this direction.
9. If we reach where no requests are needed to be serviced in this direction reverse the direction and go to step 3 until all tracks in request array have not been serviced.

Examples:

Input:

Request sequence = {176, 79, 34, 60, 92, 11, 41, 114}

Initial head position = 50

Direction = right (We are moving from left to right)

Output:

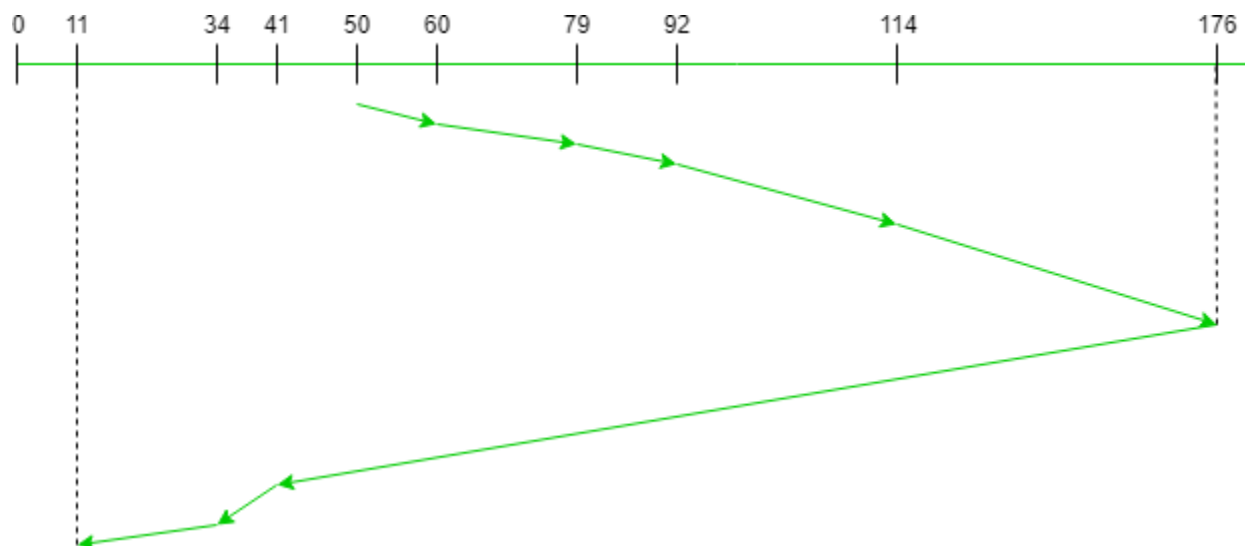
Initial position of head: 50

Total number of seek operations = 291

Seek Sequence is

60
79
92
114
176
41
34
11

The following chart shows the sequence in which requested tracks are serviced using LOOK.



Therefore, the total seek count is calculated as:

$$\begin{aligned}
 &= (60-50)+(79-60)+(92-79) \\
 &\quad +(114-92)+(176-114) \\
 &\quad +(176-41)+(41-34)+(34-11)
 \end{aligned}$$

C-LOOK Disk Scheduling Algorithm

Given an array of disk track numbers and initial head position, our task is to find the total number of seek operations done to access all the requested tracks if **C-LOOK** disk scheduling algorithm is used. Also, write a program to find the seek sequence using **C-LOOK** disk scheduling algorithm.

C-LOOK (Circular LOOK) Disk Scheduling Algorithm:

C-LOOK is an enhanced version of both **SCAN** as well as **LOOK** disk scheduling algorithms. This algorithm also uses the idea of wrapping the tracks as a circular cylinder as C-SCAN algorithm but the seek time is better than C-SCAN algorithm. We know that C-SCAN is used to avoid starvation and services all the requests more uniformly, the same goes for C-LOOK.

In this algorithm, the head services requests only in one direction(either left or right) until all the requests in this direction are not serviced and then jumps back to the farthest request on the other direction and service the remaining requests which gives a better uniform servicing as well as avoids wasting seek time for going till the end of the disk.

Algorithm-

1. Let Request array represents an array storing indexes of the tracks that have been requested in ascending order of their time of arrival and **head** is the position of the disk head.
2. The initial direction in which the head is moving is given and it services in the same direction.
3. The head services all the requests one by one in the direction it is moving.
4. The head continues to move in the same direction until all the requests in this direction have been serviced.
5. While moving in this direction, calculate the absolute distance of the tracks from the head.
6. Increment the total seek count with this distance.
7. Currently serviced track position now becomes the new head position.
8. Go to step 5 until we reach the last request in this direction.
9. If we reach the last request in the current direction then reverse the direction and move the head in this direction until we reach the last request that is needed to be serviced in this direction without servicing the intermediate requests.
10. Reverse the direction and go to step 3 until all the requests have not been serviced.

Examples:

Input:

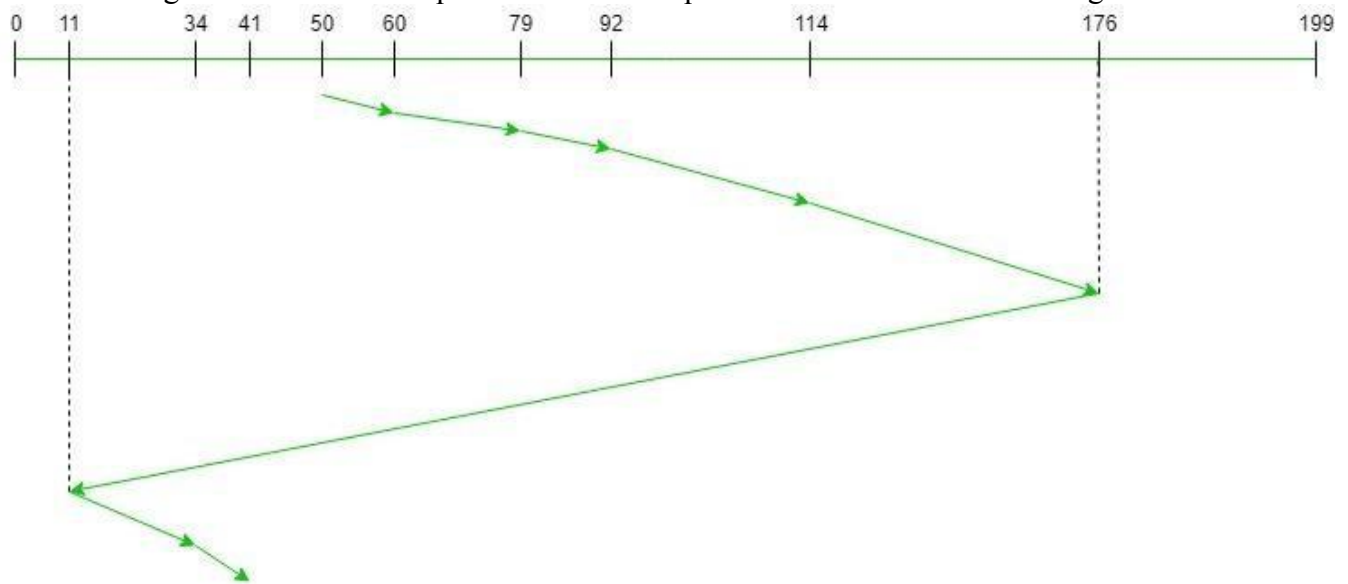
Request sequence = {176, 79, 34, 60, 92, 11, 41, 114}

Initial head position = 50
Direction = right (Moving from left to right)

Output:

Initial position of head: 50
Total number of seek operations = 156
Seek Sequence is
60
79
92
114
176
11
34
41

The following chart shows the sequence in which requested tracks are serviced using C-LOOK.



Therefore, the total seek count = $(60 - 50) + (79 - 60) + (92 - 79) + (114 - 92) + (176 - 114) + (176 - 11) + (34 - 11) + (41 - 34) = 321$

UNIT 8: Security And Protection

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

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

Authentication

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

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

One Time passwords

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

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

Program Threats

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

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

System Threats

System threats refers to misuse of system services and network connections to put user in trouble. System threats can be used to launch program threats on a complete network called as program attack. System threats creates such an environment that operating system resources/ user files are misused. Following is the list of some well-known system threats.

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

Computer Security Classifications

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

S.N.	Classification Type & Description
1	Type A Highest Level. Uses formal design specifications and verification techniques. Grants a high degree of assurance of process security.
2	Type B Provides mandatory protection system. Have all the properties of a class C2 system. Attaches a sensitivity label to each object. It is of three types. ▣ B1 – Maintains the security label of each object in the system. Label is used for making decisions to access control. ▣ B2 – Extends the sensitivity labels to each system resource, such as storage objects, supports covert channels and auditing of events. ▣ B3 – Allows creating lists or user groups for access-control to grant access or revoke access to a given named object.
3	Type C Provides protection and user accountability using audit capabilities. It is of two types. ▣ C1 – Incorporates controls so that users can protect their private information and keep other users from accidentally reading / deleting their data. UNIX versions are mostly C1 class. ▣ C2 – Adds an individual-level access control to the capabilities of a C1 level system.
4	Type D Lowest level. Minimum protection. MS-DOS, Window 3.1 fall in this category.

Goals of Protection

- Obviously to prevent malicious misuse of the system by users or programs. See chapter 15 for a more thorough coverage of this goal.
- To ensure that each shared resource is used only in accordance with system *policies*, which may be set either by system designers or by system administrators.
- To ensure that errant programs cause the minimal amount of damage possible.

- Note that protection systems only provide the *mechanisms* for enforcing policies and ensuring reliable systems. It is up to administrators and users to implement those mechanisms effectively.

Principles of Protection

- The **principle of least privilege** dictates that programs, users, and systems be given just enough privileges to perform their tasks.
- This ensures that failures do the least amount of harm and allow the least of harm to be done.
- For example, if a program needs special privileges to perform a task, it is better to make it a SGID program with group ownership of "network" or "backup" or some other pseudo group, rather than SUID with root ownership. This limits the amount of damage that can occur if something goes wrong.
- Typically each user is given their own account, and has only enough privilege to modify their own files.
- The root account should not be used for normal day to day activities - The System Administrator should also have an ordinary account, and reserve use of the root account for only those tasks which need the root privileges

Domain of Protection

- A computer can be viewed as a collection of *processes* and *objects* (both HW & SW).
- The **need to know principle** states that a process should only have access to those objects it needs to accomplish its task, and furthermore only in the modes for which it needs access and only during the time frame when it needs access.
- The modes available for a particular object may depend upon its type.

Domain Structure

- A **protection domain** specifies the resources that a process may access.
- Each domain defines a set of objects and the types of operations that may be invoked on each object.
- An **access right** is the ability to execute an operation on an object.
- A domain is defined as a set of $\langle \text{object}, \{ \text{access right set} \} \rangle$ pairs, as shown below. Note that some domains may be disjoint while others overlap.

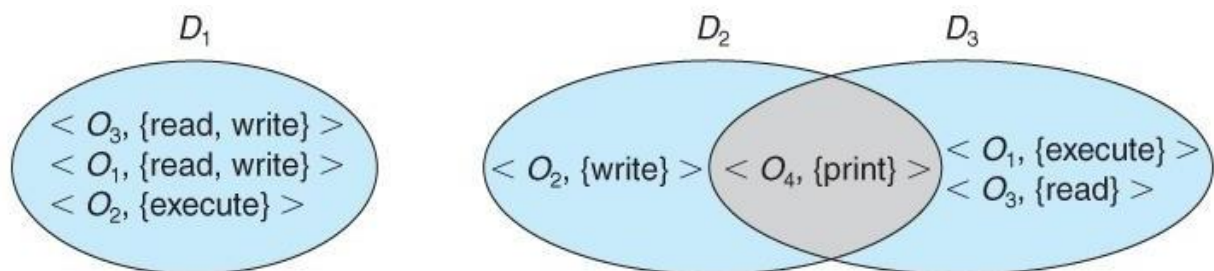


Figure 8.1- System with three protection domains.

- The association between a process and a domain may be *static* or *dynamic*.
 - If the association is static, then the need-to-know principle requires a way of changing the contents of the domain dynamically.
 - If the association is dynamic, then there needs to be a mechanism for **domain switching**.
- Domains may be realized in different fashions - as users, or as processes, or as procedures. E.g. if each user corresponds to a domain, then that domain defines the access of that user, and changing domains involves changing user ID.

An Example: UNIX

- UNIX associates domains with users.
- Certain programs operate with the SUID bit set, which effectively changes the user ID, and therefore the access domain, while the program is running. (and similarly for the SGID bit.) Unfortunately this has some potential for abuse.
- An alternative used on some systems is to place privileged programs in special directories, so that they attain the identity of the directory owner when they run. This prevents crackers from placing SUID programs in random directories around the system.
- Yet another alternative is to not allow the changing of ID at all. Instead, special privileged daemons are launched at boot time, and user processes send messages to these daemons when they need special tasks performed.

An Example: MULTICS

- The MULTICS system uses a complex system of rings, each corresponding to a different protection domain, as shown below:

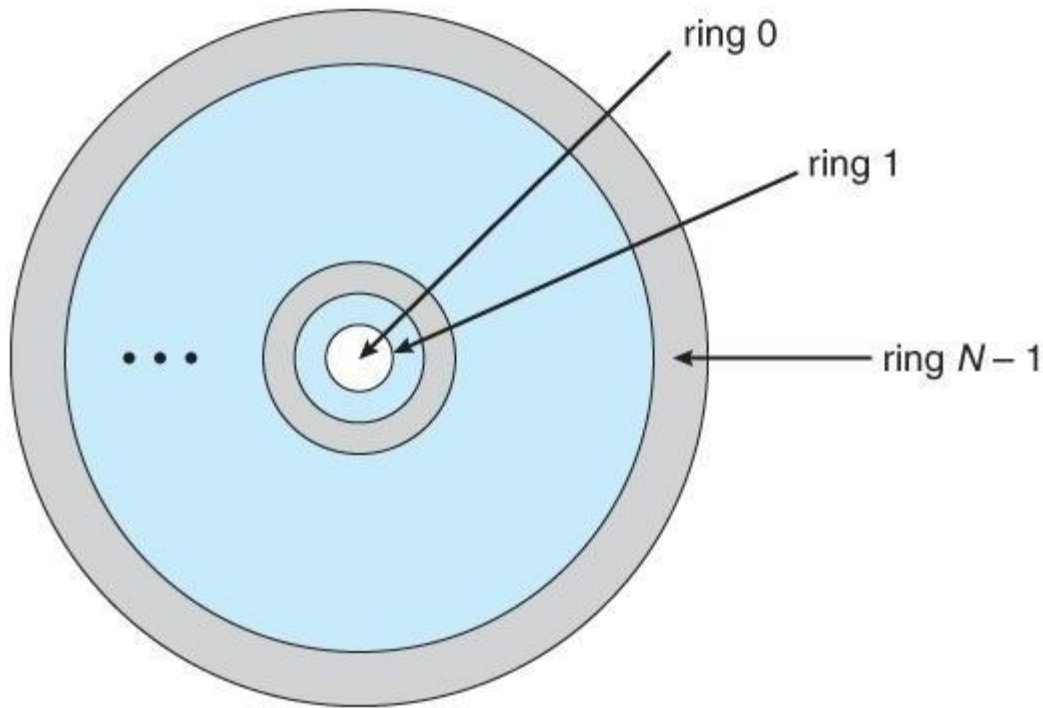


Figure 8.2 - MULTICS ring structure.

- Rings are numbered from 0 to 7, with outer rings having a subset of the privileges of the inner rings.
- Each file is a memory segment, and each segment description includes an entry that indicates the ring number associated with that segment, as well as read, write, and execute privileges.
- Each process runs in a ring, according to the *current-ring-number*, a counter associated with each process.
- A process operating in one ring can only access segments associated with higher (farther out) rings, and then only according to the access bits. Processes cannot access segments associated with lower rings.
- Domain switching is achieved by a process in one ring calling upon a process operating in a lower ring, which is controlled by several factors stored with each segment descriptor:
 - An **access bracket**, defined by integers $b1 \leq b2$.
 - A **limit** $b3 > b2$
 - A **list of gates**, identifying the entry points at which the segments may be called.
- If a process operating in ring i calls a segment whose bracket is such that $b1 \leq i \leq b2$, then the call succeeds and the process remains in ring i .
- Otherwise a trap to the OS occurs, and is handled as follows:
 - If $i < b1$, then the call is allowed, because we are transferring to a procedure with fewer privileges. However if any of the parameters being passed are of segments below $b1$, then they must be copied to an area accessible by the called procedure.

- If $i > b_2$, then the call is allowed only if $i \leq b_3$ and the call is directed to one of the entries on the list of gates.
- Overall this approach is more complex and less efficient than other protection schemes.

Access Matrix

- The model of protection that we have been discussing can be viewed as an **access matrix**, in which columns represent different system resources and rows represent different protection domains. Entries within the matrix indicate what access that domain has to that resource.

object domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Figure 8.3 - Access matrix.

- Domain switching can be easily supported under this model, simply by providing "switch" access to other domains:

object domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Figure 8.4 - Access matrix of Figure 14.3 with domains as objects.

- The ability to **copy** rights is denoted by an asterisk, indicating that processes in that domain have the right to copy that access within the same column, i.e. for the same object. There are two important variations:
 - If the asterisk is removed from the original access right, then the right is **transferred**, rather than being copied. This may be termed a **transfer** right as opposed to a **copy** right.
 - If only the right and not the asterisk is copied, then the access right is added to the new domain, but it may not be propagated further. That is the new domain does not also receive the right to copy the access. This may be termed a **limited copy** right, as shown in Figure 14.5 below:

object domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

object domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

Figure 8.5 - Access matrix with copy rights.

- The **owner** right adds the privilege of adding new rights or removing existing ones:

object domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

(a)

object domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

(b)

Figure 8.6 - Access matrix with owner rights.

- Copy and owner rights only allow the modification of rights within a column. The addition of **control rights**, which only apply to domain objects, allow a process operating in one domain to affect the rights available in other domains. For example in the table below, a process operating in domain D2 has the right to control any of the rights in domain D4.

object domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

Figure 8.7 - Modified access matrix of Figure 14.4

Implementation of Access Matrix

Global Table

- The simplest approach is one big global table with $\langle \text{domain, object, rights} \rangle$ entries.
- Unfortunately this table is very large (even if sparse) and so cannot be kept in memory (without invoking virtual memory techniques.)
- There is also no good way to specify groupings - If everyone has access to some resource, then it still needs a separate entry for every domain.

Access Lists for Objects

- Each column of the table can be kept as a list of the access rights for that particular object, discarding blank entries.
- For efficiency a separate list of default access rights can also be kept, and checked first.

Capability Lists for Domains

- In a similar fashion, each row of the table can be kept as a list of the capabilities of that domain.
- Capability lists are associated with each domain, but not directly accessible by the domain or any user process.
- Capability lists are themselves protected resources, distinguished from other data in one of two ways:
 - A **tag**, possibly hardware implemented, distinguishing this special type of data. (other types may be floats, pointers, booleans, etc.)
 - The address space for a program may be split into multiple segments, at least one of which is inaccessible by the program itself, and used by

the operating system for maintaining the process's access right capability list.

A Lock-Key Mechanism

- Each resource has a list of unique bit patterns, termed locks.
- Each domain has its own list of unique bit patterns, termed keys.
- Access is granted if one of the domain's keys fits one of the resource's locks.
- Again, a process is not allowed to modify its own keys.

Comparison

- Each of the methods here has certain advantages or disadvantages, depending on the particular situation and task at hand.
- Many systems employ some combination of the listed methods.

Access Control

- **Role-Based Access Control, RBAC**, assigns privileges to users, programs, or roles as appropriate, where "privileges" refer to the right to call certain system calls, or to use certain parameters with those calls.
- RBAC supports the principle of least privilege, and reduces the susceptibility to abuse as opposed to SUID or SGID programs.

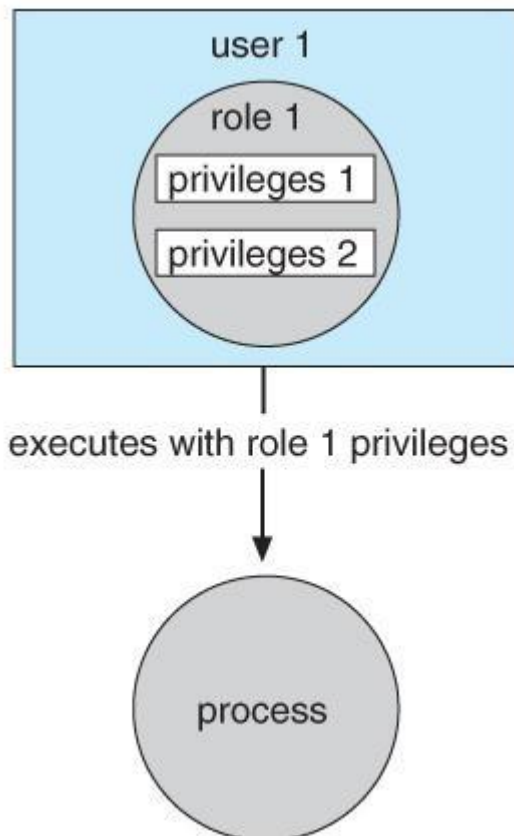


Figure 8.8 - Role-based access control in Solaris 10.

14.7 Revocation of Access Rights

- The need to revoke access rights dynamically raises several questions:
 - Immediate versus delayed - If delayed, can we determine when the revocation will take place?
 - Selective versus general - Does revocation of an access right to an object affect *all* users who have that right, or only some users?
 - Partial versus total - Can a subset of rights for an object be revoked, or are all rights revoked at once?
 - Temporary versus permanent - If rights are revoked, is there a mechanism for processes to re-acquire some or all of the revoked rights?
- With an access list scheme revocation is easy, immediate, and can be selective, general, partial, total, temporary, or permanent, as desired.
- With capabilities lists the problem is more complicated, because access rights are distributed throughout the system. A few schemes that have been developed include:
 - Reacquisition - Capabilities are periodically revoked from each domain, which must then re-acquire them.
 - Back-pointers - A list of pointers is maintained from each object to each capability which is held for that object.
 - Indirection - Capabilities point to an entry in a global table rather than to the object. Access rights can be revoked by changing or invalidating the table entry, which may affect multiple processes, which must then re-acquire access rights to continue.
 - Keys - A unique bit pattern is associated with each capability when created, which can be neither inspected nor modified by the process.
 - A master key is associated with each object.
 - When a capability is created, its key is set to the object's master key.
 - As long as the capability's key matches the object's key, then the capabilities remain valid.
 - The object master key can be changed with the set-key command, thereby invalidating all current capabilities.
 - More flexibility can be added to this scheme by implementing a **list** of keys for each object, possibly in a global table.

Capability-Based Systems (Optional)

14.8.1 An Example: Hydra

- Hydra is a capability-based system that includes both system-defined **rights** and user-defined rights. The interpretation of user-defined rights is up to the specific user programs, but the OS provides support for protecting access to those rights, whatever they may be
- Operations on objects are defined procedurally, and those procedures are themselves protected objects, accessed indirectly through capabilities.
- The names of user-defined procedures must be identified to the protection system if it is to deal with user-defined rights.

- When an object is created, the names of operations defined on that object become **auxiliary rights**, described in a capability for an **instance** of the type. For a process to act on an object, the capabilities it holds for that object must contain the name of the operation being invoked. This allows access to be controlled on an instance-by-instance and process-by-process basis.
- Hydra also allows **rights amplification**, in which a process is deemed to be **trustworthy**, and thereby allowed to act on any object corresponding to its parameters.
- Programmers can make direct use of the Hydra protection system, using suitable libraries which are documented in appropriate reference manuals.

14.8.2 An Example: Cambridge CAP System

- The CAP system has two kinds of capabilities:
 - *Data capability*, used to provide read, write, and execute access to objects. These capabilities are interpreted by microcode in the CAP machine.
 - *Software capability*, is protected but not interpreted by the CAP microcode.
 - Software capabilities are interpreted by protected (privileged) procedures, possibly written by application programmers.
 - When a process executes a protected procedure, it temporarily gains the ability to read or write the contents of a software capability.
 - This leaves the interpretation of the software capabilities up to the individual subsystems, and limits the potential damage that could be caused by a faulty privileged procedure.
 - Note, however, that protected procedures only get access to software capabilities for the subsystem of which they are a part. Checks are made when passing software capabilities to protected procedures that they are of the correct type.
 - Unfortunately the CAP system does not provide libraries, making it harder for an individual programmer to use than the Hydra system.

14.9 Language-Based Protection (Optional)

- As systems have developed, protection systems have become more powerful, and also more specific and specialized.
- To refine protection even further requires putting protection capabilities into the hands of individual programmers, so that protection policies can be implemented on the application level, i.e. to protect resources in ways that are known to the specific applications but not to the more general operating system.

14.9.1 Compiler-Based Enforcement

- In a compiler-based approach to protection enforcement, programmers directly specify the protection needed for different resources at the time the resources are declared.
- This approach has several advantages:
 1. Protection needs are simply declared, as opposed to a complex series of procedure calls.
 2. Protection requirements can be stated independently of the support provided by a particular OS.
 3. The means of enforcement need not be provided directly by the developer.
 4. Declarative notation is natural, because access privileges are closely related to the concept of data types.
- Regardless of the means of implementation, compiler-based protection relies upon the underlying protection mechanisms provided by the underlying OS, such as the Cambridge CAP or Hydra systems.
- Even if the underlying OS does not provide advanced protection mechanisms, the compiler can still offer some protection, such as treating memory accesses differently in code versus data segments. (E.g. code segments can't be modified, data segments can't be executed.)
- There are several areas in which compiler-based protection can be compared to kernel-enforced protection:
 1. **Security.** Security provided by the kernel offers better protection than that provided by a compiler. The security of the compiler-based enforcement is dependent upon the integrity of the compiler itself, as well as requiring that files not be modified after they are compiled. The kernel is in a better position to protect itself from modification, as well as protecting access to specific files. Where hardware support of individual memory accesses is available, the protection is stronger still.
 2. **Flexibility.** A kernel-based protection system is not as flexible to provide the specific protection needed by an individual programmer, though it may provide support which the programmer may make use of. Compilers are more easily changed and updated when necessary to change the protection services offered or their implementation.
 3. **Efficiency.** The most efficient protection mechanism is one supported by hardware and microcode. Insofar as software based protection is concerned, compiler-based systems have the advantage that many checks can be made off-line, at compile time, rather than during execution.
- The concept of incorporating protection mechanisms into programming languages is in its infancy, and still remains to be fully developed. However the general goal is to provide mechanisms for three functions:
 1. Distributing capabilities safely and efficiently among customer processes. In particular a user process should only be able to access resources for which it was issued capabilities.

2. Specifying the **type** of operations a process may execute on a resource, such as reading or writing.
3. Specifying the **order** in which operations are performed on the resource, such as opening before reading.

14.9.2 Protection in Java

- Java was designed from the very beginning to operate in a distributed environment, where code would be executed from a variety of trusted and untrusted sources. As a result the Java Virtual Machine, JVM incorporates many protection mechanisms
- When a Java program runs, it load up classes dynamically, in response to requests to instantiates objects of particular types. These classes may come from a variety of different sources, some trusted and some not, which requires that the protection mechanism be implemented at the resolution of individual classes, something not supported by the basic operating system.
- As each class is loaded, it is placed into a separate protection domain. The capabilities of each domain depend upon whether the source URL is trusted or not, the presence or absence of any digital signatures on the class (Chapter 15), and a configurable policy file indicating which servers a particular user trusts, etc.
- When a request is made to access a restricted resource in Java, (e.g. open a local file), some process on the current **call stack** must specifically assert a privilege to perform the operation. In essence this method **assumes responsibility** for the restricted access. Naturally the method must be part of a class which resides in a protection domain that includes the capability for the requested operation. This approach is termed **stack inspection**, and works like this:
 - When a caller may not be trusted, a method executes an access request within a `doPrivileged( )` block, which is noted on the calling stack.
 - When access to a protected resource is requested, `checkPermissions( )` inspects the call stack to see if a method has asserted the privilege to access the protected resource.
 - If a suitable `doPrivileged` block is encountered on the stack before a domain in which the privilege is disallowed, then the request is granted.
 - If a domain in which the request is disallowed is encountered first, then the access is denied and a `AccessControlException` is thrown.
 - If neither is encountered, then the response is implementation dependent.
- In the example below the untrusted applet's call to `get( )` succeeds, because the trusted URL loader asserts the privilege of opening the specific URL `lucent.com`. However when the applet tries to make a direct call to `open( )` it fails, because it does not have privilege to access any sockets.

protection domain:	untrusted applet	URL loader	networking
socket permission:	none	*.lucent.com:80, connect	any
class:	gui: ... get(url); open(addr); ...	get(URL u): ... doPrivileged { open('proxy.lucent.com:80'); } <request u from proxy> ...	open(Addr a): ... checkPermission (a, connect); connect (a); ...

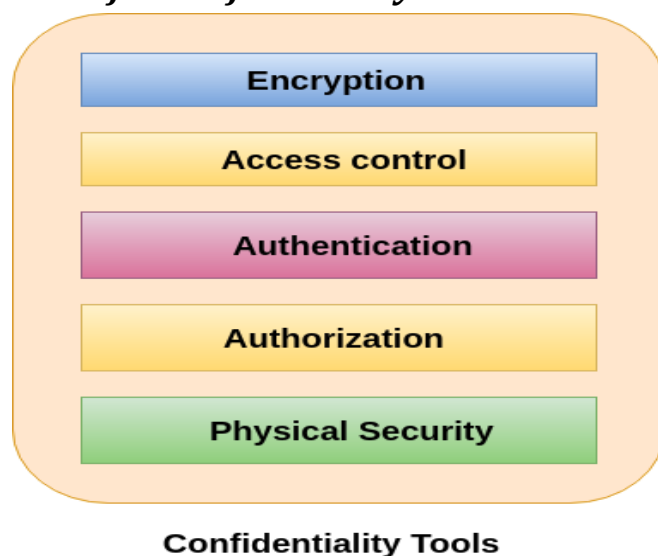
Figure 14.9 - Stack inspection.

Discuss Some Security Goals.

1. Confidentiality

Confidentiality is roughly equivalent to privacy and avoids the unauthorized disclosure of information. It involves the protection of data, providing access for those who are allowed to see it while disallowing others from learning anything about its content. It prevents essential information from reaching the wrong people while making sure that the right people can get it. Data encryption is a good example to ensure confidentiality.

Tools for Confidentiality



Encryption

Encryption is a method of transforming information to make it unreadable for unauthorized users by using an algorithm. The transformation of data uses a secret key (an encryption key) so that the transformed data can only be read by using another secret key (decryption key). It protects sensitive data such as credit card numbers by encoding and transforming data into unreadable cipher text. This encrypted data can only be read by decrypting it. Asymmetric-key and symmetric-key are the two primary types of encryption.

Access control

Access control defines rules and policies for limiting access to a system or to physical or virtual resources. It is a process by which users are granted access and certain privileges to systems, resources or information. In access control systems, users need to present credentials before they can be granted access such as a person's name or a computer's serial number. In physical systems, these credentials may come in many forms, but credentials that can't be transferred provide the most security.

Authentication

An authentication is a process that ensures and confirms a user's identity or role that someone has. It can be done in a number of different ways, but it is usually based on a combination of-

- something the person has (like a smart card or a radio key for storing secret keys),
- something the person knows (like a password),
- something the person is (like a human with a fingerprint).

Authentication is the necessity of every organizations because it enables organizations to keep their networks secure by permitting only authenticated users to access its protected resources. These resources may include computer systems, networks, databases, websites and other network-based applications or services.

Authorization

Authorization is a security mechanism which gives permission to do or have something. It is used to determine a person or system is allowed access to resources, based on an access control policy, including computer programs, files, services, data and application features. It is normally preceded by authentication for user identity verification. System administrators are typically assigned permission levels covering all system and user resources. During authorization, a system verifies an authenticated user's access rules and either grants or refuses resource access.

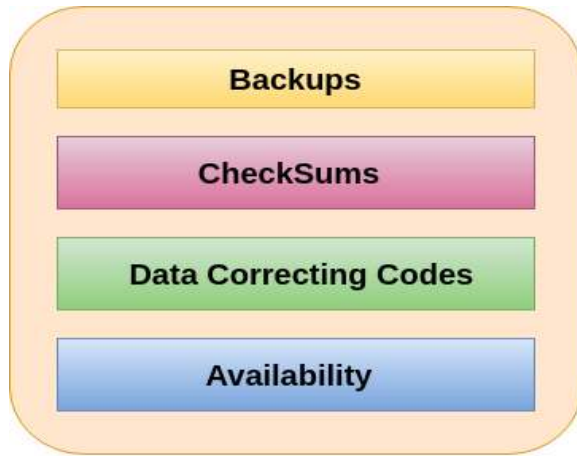
Physical Security

Physical security describes measures designed to deny the unauthorized access of IT assets like facilities, equipment, personnel, resources and other properties from damage. It protects these assets from physical threats including theft, vandalism, fire and natural disasters.

2. Integrity

Integrity refers to the methods for ensuring that data is real, accurate and safeguarded from unauthorized user modification. It is the property that information has not be altered in an unauthorized way, and that source of the information is genuine.

Tools for Integrity



Integrity Tools

Backups

Backup is the periodic archiving of data. It is a process of making copies of data or data files to use in the event when the original data or data files are lost or destroyed. It is also used to make copies for historical purposes, such as for longitudinal studies, statistics or for historical records or to meet the requirements of a data retention policy. Many applications especially in a Windows environment, produce backup files using the .BAK file extension.

Checksums

A checksum is a numerical value used to verify the integrity of a file or a data transfer. In other words, it is the computation of a function that maps the contents of a file to a numerical value. They are typically used to compare two sets of data to make sure that they are the same. A checksum function depends on the entire contents of a file. It is designed in a way that even a small change to the input file (such as flipping a single bit) likely to results in different output value.

Data Correcting Codes

It is a method for storing data in such a way that small changes can be easily detected and automatically corrected.

3. Availability

Availability is the property in which information is accessible and modifiable in a timely fashion by those authorized to do so. It is the guarantee of reliable and constant access to our sensitive data by authorized people.

Tools for Availability

- Physical Protections
- Computational Redundancies

Physical Protections

Physical safeguard means to keep information available even in the event of physical challenges. It ensure sensitive information and critical information technology are housed in secure areas.

Computational redundancies

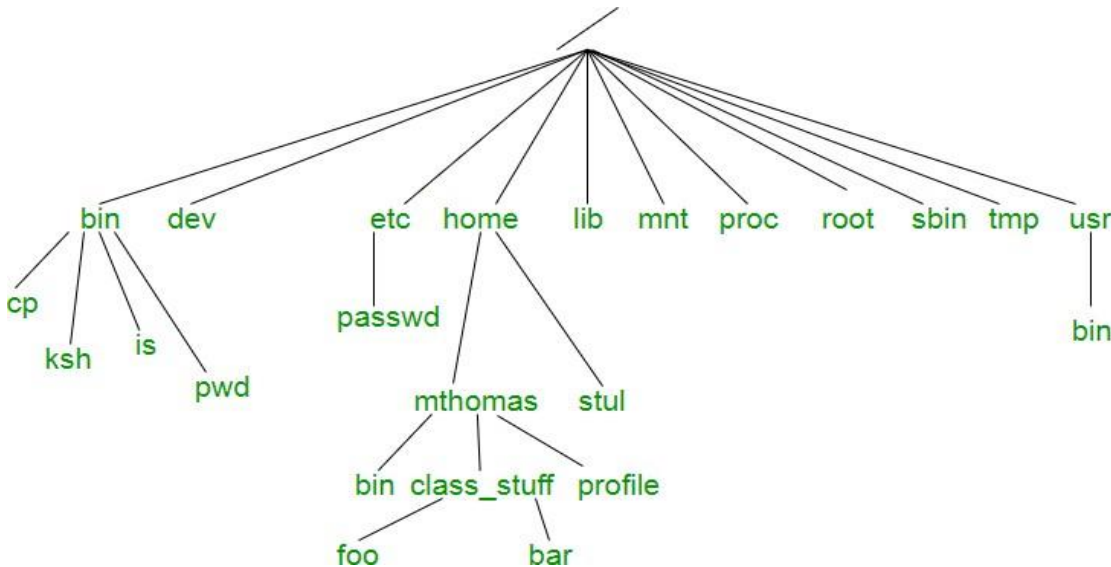
It is applied as fault tolerant against accidental faults. It protects computers and storage devices that serve as fallbacks in the case of failures.

Unit 9: UNIX AND LINUX Operating System's

Unix File System

Unix file system is a logical method of **organizing and storing** large amounts of information in a way that makes it easy to manage. A file is a smallest unit in which the information is stored. Unix file system has several important features. All data in Unix is organized into files. All files are organized into directories. These directories are organized into a tree-like structure called the file system.

Files in Unix System are organized into multi-level hierarchy structure known as a directory tree. At the very top of the file system is a directory called “root” which is represented by a “/”. All other files are “descendants” of root.



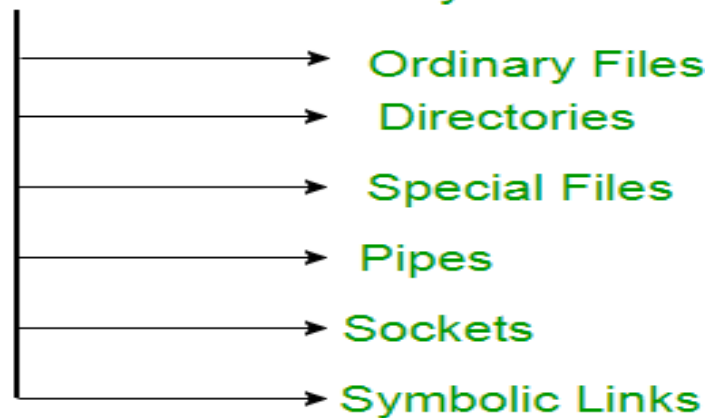
Directories or Files and their description –

- **/** : The slash / character alone denotes the root of the filesystem tree.
- **/bin** : Stands for “binaries” and contains certain fundamental utilities, such as ls or cp, which are generally needed by all users.
- **/boot** : Contains all the files that are required for successful booting process.
- **/dev** : Stands for “devices”. Contains file representations of peripheral devices and pseudo-devices.
- **/etc** : Contains system-wide configuration files and system databases. Originally also contained “dangerous maintenance utilities” such as init, but these have typically been moved to /sbin or elsewhere.
- **/home** : Contains the home directories for the users.
- **/lib** : Contains system libraries, and some critical files such as kernel modules or device drivers.
- **/media** : Default mount point for removable devices, such as USB sticks, media players, etc.
- **/mnt** : Stands for “mount”. Contains filesystem mount points. These are used, for example, if the system uses multiple hard disks or hard disk partitions. It is also often used for remote (network) filesystems, CD-ROM/DVD drives, and so on.
- **/proc** : procfs virtual filesystem showing information about processes as files.
- **/root** : The home directory for the superuser “root” – that is, the system administrator. This account’s home directory is usually on the initial filesystem, and hence not in /home (which may be a mount point for another filesystem) in case specific maintenance needs to be performed, during which other filesystems are not available. Such a case could occur, for example, if a hard disk drive suffers physical failures and cannot be properly mounted.

- **/tmp** : A place for temporary files. Many systems clear this directory upon startup; it might have tmpfs mounted atop it, in which case its contents do not survive a reboot, or it might be explicitly cleared by a startup script at boot time.
- **/usr** : Originally the directory holding user home directories, its use has changed. It now holds executables, libraries, and shared resources that are not system critical, like the X Window System, KDE, Perl, etc. However, on some Unix systems, some user accounts may still have a home directory that is a direct subdirectory of /usr, such as the default as in Minix. (on modern systems, these user accounts are often related to server or system use, and not directly used by a person).
- **/usr/bin** : This directory stores all binary programs distributed with the operating system not residing in /bin, /sbin or (rarely) /etc.
- **/usr/include** : Stores the development headers used throughout the system. Header files are mostly used by the **#include** directive in C/C++ programming language.
- **/usr/lib** : Stores the required libraries and data files for programs stored within /usr or elsewhere.
- **/var** : A short for “variable.” A place for files that may change often – especially in size, for example e-mail sent to users on the system, or process-ID lock files.
- **/var/log** : Contains system log files.
- **/var/mail** : The place where all the incoming mails are stored. Users (other than root) can access their own mail only. Often, this directory is a symbolic link to /var/spool/mail.
- **/var/spool** : Spool directory. Contains print jobs, mail spools and other queued tasks.
- **/var/tmp** : A place for temporary files which should be preserved between system reboots.

Types of Unix files – The UNIX files system contains several different types of files :

Classification of Unix File System :



1. Ordinary files – An ordinary file is a file on the system that contains data, text, or program instructions.

- Used to store your information, such as some text you have written or an image you have drawn. This is the type of file that you usually work with.
- Always located within/under a directory file.
- Do not contain other files.
- In long-format output of ls -l, this type of file is specified by the “-” symbol.

2. Directories – Directories store both special and ordinary files. For users familiar with Windows or Mac OS, UNIX directories are equivalent to folders. A directory file contains an entry for every file and subdirectory that it houses. If you have 10 files in a directory, there will be 10 entries in the directory. Each entry has two components.

(1) The Filename

(2) A unique identification number for the file or directory (called the inode number)

- Branching points in the hierarchical tree.

- Used to organize groups of files.
- May contain ordinary files, special files or other directories.
- Never contain “real” information which you would work with (such as text). Basically, just used for organizing files.
- All files are descendants of the root directory, (named /) located at the top of the tree.

In long-format output of `ls -l` , this type of file is specified by the “d” symbol.

3. Special Files – Used to represent a real physical device such as a printer, tape drive or terminal, used for Input/Output (I/O) operations. **Device or special files** are used for device Input/Output(I/O) on UNIX and Linux systems. They appear in a file system just like an ordinary file or a directory.

On UNIX systems there are two flavors of special files for each device, character special files and block special files :

- When a character special file is used for device Input/Output(I/O), data is transferred one character at a time. This type of access is called raw device access.
- When a block special file is used for device Input/Output(I/O), data is transferred in large fixed-size blocks. This type of access is called block device access.

For terminal devices, it’s one character at a time. For disk devices though, raw access means reading or writing in whole chunks of data – blocks, which are native to your disk.

- In long-format output of `ls -l`, character special files are marked by the “c” symbol.
- In long-format output of `ls -l`, block special files are marked by the “b” symbol.

4. Pipes – UNIX allows you to link commands together using a pipe. The pipe acts a temporary file which only exists to hold data from one command until it is read by another. A Unix pipe provides a one-way flow of data. The output or result of the first command sequence is used as the input to the second command sequence. To make a pipe, put a vertical bar (|) on the command line between two commands. For example: **who | wc -l**

In long-format output of `ls -l` , named pipes are marked by the “p” symbol.

5. Sockets – A Unix socket (or Inter-process communication socket) is a special file which allows for advanced inter-process communication. A Unix Socket is used in a client-server application framework. In essence, it is a stream of data, very similar to network stream (and network sockets), but all the transactions are local to the filesystem.

In long-format output of `ls -l`, Unix sockets are marked by “s” symbol.

6. Symbolic Link – Symbolic link is used for referencing some other file of the file system. Symbolic link is also known as Soft link. It contains a text form of the path to the file it references. To an end user, symbolic link will appear to have its own name, but when you try reading or writing data to this file, it will instead reference these operations to the file it points to. If we delete the soft link itself , the data file would still be there. If we delete the source file or move it to a different location, symbolic file will not function properly.

In long-format output of `ls -l` , Symbolic link are marked by the “l” symbol (that’s a lower case L).

1. ls

ls is a utility for listing the files in a directory.

Most used options are:

- a** all files (include files with . prefix)
- l** long detail (provide file statistics)
- t** order by creation time
- u** sort by access time (or show when last accessed together with -l)
- S** order by size
- r** reverse order
- F** mark directories with /, executables with *, symbolic links with @, local sockets with =, named pipes (FIFOs) with | Other options include:
- Fx --color** color-coded listing
- s** show file sizes
- Z** show the file/directory with there SELinux context (only on SELinux enable tree)
- h** "human readable"; show file sizes in kilobytes and megabytes (-h can be used together with -l or -s)

```
ls *st* : list files that contain st in name
ls > list : output list to file named "list"
ls | more : fit listing to screen
```

Examples

```
$ ls
fish hello.txt
$ ls -l
-rw-r--r--  1 username  groupname    0 Apr 11 00:09 fish
-rw-r--r--  1 username  groupname   11 Apr 11 00:10 hello.txt
```

Several systems have an alias **ll** which does the same as **ls -l**:

```
$ ll
```

```
-rw-r--r-- 1 username  groupname    0 Apr 11 00:09 fish
-rw-r--r-- 1 username  groupname    11 Apr 11 00:10 hello.txt
```

Be careful with the -F option. Here is one example:

```
$ ls -F /usr/X11R6/bin/X*
```

```
/usr/X11R6/bin/X@      /usr/X11R6/bin/Xnest*  /usr/X11R6/bin/Xprt*
/usr/X11R6/bin/Xmark*  /usr/X11R6/bin/Xorg*   /usr/X11R6/bin/Xvfb*
```

We do not know yet if there is a symbolic link "X" and an executable "Xmark" or if "X@" and "Xmark*" are just the names of normal files. (Though "@" and "*" are not much found in filenames, they are possible.) So we check by dropping the -F:

```
$ ls /usr/X11R6/bin/X*
```

```
/usr/X11R6/bin/X      /usr/X11R6/bin/Xnest  /usr/X11R6/bin/Xprt
/usr/X11R6/bin/Xmark  /usr/X11R6/bin/Xorg   /usr/X11R6/bin/Xvfb
```

2. mkdir

mkdir is a utility for creating a directory.

Examples

```
$ mkdir newdirectoryname
```

The -p option also makes parent-directories as needed. Instead of:

```
$ mkdir foo
$ cd foo
$ mkdir bar
```

you can just do:

```
$ mkdir -p foo/bar
```

3.cd

cd changes the **current** directory of the shell. This current directory will be used by other programs launched from the shell.

Because "cd" changes the state of the shell, it is a *shell built-in command*. In contrast, most commands are separate programs which the shell starts.

Examples

Change to 'foobar' directory:

```
$ cd foobar
```

Change to your home directory, `cd` command used without an option will drop you back into your home directory.

```
$ cd
```

`~` (tilde) stores the path to your home directory, this command has same effect as the previous one.

```
$ cd ~
```

Change to parent directory:

```
$ cd ..
```

Change to the previous directory:

```
$ cd -
```

Tips:

The "CDPATH" might only work in some shells. For example, **ksh** has it.

By setting "CDPATH" environment variable in your shell you can take advantage of shell command completion facility.

```
$ echo $CDPATH
```

```
./usr/local:/usr/share/doc
```

If you have the `$CDPATH` set, then you press 'TAB' key and get possible path completions

```
$ cd bas [TAB]
```

```
base-config/ base-files/ base-passwd/ bash/      bastille/
```

4. pwd

pwd (for Print Working Directory) shows the current directory that you are in.

Though "pwd" is often available as an external program (like /bin/pwd), many shells offer an equivalent version as a shell builtin command. Like any external command, "pwd" would inherit the current directory from the shell or other program that starts it.

Examples

```
$ pwd
/home/username
```

You can change the directory, you can also

```
$ cd /usr
$ pwd
/usr
```

You can also use "pwd" in scripts. If you have enough experience with scripting, then you would know that the next line complains if the current directory is /home/username.

```
$ test "x$(pwd)" = x/home/username || echo wrong directory
```

5. cp

cp copies a file

Most used options are:

- r** copies directories (recursively)
- p** preserves permissions, ownership, and timestamps
- i** prompt before overwrite
- v** verbose, show filenames as they are being copied

Examples

Makes a copy of file 'debian' and call it 'Debian' (assuming 'Debian' is not already a directory)

```
$ cp -i debian Debian
```

Makes a copy of file 'debian' and put it at /tmp/debian

```
$ cp -i debian /tmp/debian
```

Same as the previous command (the filename defaults to be the same).

```
$ cp -i debian /tmp
```

Makes a copy of directory 'mydir' (and all its contents) and put it at /tmp/mydir

```
$ cp -ir mydir/ /tmp
```

Copy multiple files to directory /tmp

```
$ cp -i foo bar baz /tmp
```

5. mv

mv move and/or rename files

Examples

Rename file 'unix' to 'Unix' (assuming "Unix" is not a directory)

```
$ mv -i unix Unix
```

Move file Unix from your home directory to /tmp.

```
$ mv -i ~/Unix /tmp/Unix
```

Same as the previous command (the filename defaults to be the same).

```
$ mv -i ~/Unix /tmp
```

Move file Unix from your home directory to /tmp, and rename it to 'unix'.

```
$ mv -i ~/Unix /tmp/unix
```

Move multiple files to directory /tmp

```
$ mv -i foo bar baz /tmp
```

6.rm

rm deletes a file from the filesystem, like the "del" command in DOS.

The GNU long options (like --directory) are available on Linux, but not most other systems.

Some useful options are:

-d, --directory

unlink FILE, even if it is an empty directory (some systems let superuser unlink non-empty directories too)

-f, --force

ignore nonexistent files, never prompt

-i, --interactive

prompt before any removal

-P

(*BSD only) overwrite file before deletion

-r, -R, --recursive

remove the contents of directories recursively (the force option must often be used to successfully run rm recursively)

-v, --verbose

(GNU only) explain what is being done

--help

(GNU only) display help and exit

--version

(GNU only) output version information and exit

Examples:

The usage of "rm" is considered potentially more dangerous than equivalents in other operating systems because of the way the shell parses wildcards and names of special directories and in its non-verbose actions.

Here is a classic example. Instead of deleting files that end with .o ("*.o") it deletes all files in the directory ("*") and also a file called .o. There is an unwanted space between the asterisk and the period.

```
$ rm *.o
rm: cannot remove `o': No such file or directory
```

To remove a file whose name starts with a '-', for example '-foo', use one of these commands:

```
$ rm -- -foo
$ rm ./-foo
```

It might be useful to create an alias such as "remove" which moves the files to a local "trash" file so you can go there and recover files you accidentally "remove"d.

Secure deletion of files:

Note that if you use rm to remove a file, it is usually possible to recover the contents of that file since rm does not remove it from the hard disk. It simply removes the file systems link to it.

On *BSD systems, the -P option overwrites the data with the file before removing it.

```
$ rm -P secretfile
```

7.rmdir[\[edit\]](#)

rmdir is a utility for deleting empty directories.

Examples

```
$ rmdir directoryname
```

If the directory is not empty, the correct way to remove the directory and all its contents recursively is to use

```
$ rm -r directoryname
```

8.cat

Original purpose of this command is to concatenate the files (horizontally). It accepts any number of files as its arguments and outputs the contents of each file, completely, one after another, on the screen (the standard output device). Assume that you have three files, *file1*, *file2* and *file3*, with the following contents in them, respectively:

file1: These are the contents of file1

file2: This text originates in file2

file3: file3 holds this text

The output of the following command:

```
cat file1 file2 file3
```

would be:

These are the contents of file1

This text originates in file2

file3 holds this text

Process Utilities

1.ps :Process status

This command stands for 'Process Status'. It is similar to the "Task Manager" that pop-ups in a Windows Machine when we use Cntrl+Alt+Del. This command is similar to 'top' command but the information displayed is different.

To check all the processes running under a user, use the command -

```
ps ux
```

```
home@VirtualBox:~$ ps ux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
home     1114   0.0   0.8  46548  8512 ?        Ssl   Sep03   0:00 gnome-sess
home     1151   0.0   0.0   3856   140 ?        Ss    Sep03   0:00 /usr/bin/s
home     1154   0.0   0.0   3748   484 ?        S     Sep03   0:00 /usr/bin/d
home     1155   0.1   0.2   6656  3036 ?        Ss    Sep03   0:18 //bin/dbus
home     1157   0.0   0.2   9148  2368 ?        S     Sep03   0:00 /usr/lib/g
home     1162   0.0   0.2  31588  2296 ?        Ssl   Sep03   0:00 /usr/lib/g
home     1174   0.0   1.4 132472 14884 ?        Sl    Sep03   0:03 /usr/lib/g
```

You can also check the process status of a single process, use the syntax -

```
ps PID
```

```
guru99@VirtualBox:~$ ps 1268
  PID TTY          STAT       TIME COMMAND
 1268 ?            S<l        0:02 /usr/bin/pulseaudio --start --log-target=syslog
```

2. *Kill*

This command **terminates running processes** on a Linux machine.

To use these utilities you need to know the PID (process id) of the process you want to kill

Syntax -

```
kill PID
```

To find the PID of a process simply type

```
pidof Process name
```

Let us try it with an example.

```
home@VirtualBox:~$ pidof Photoshop.exe
1525
home@VirtualBox:~$ kill 1525
```

3. *NICE*

Linux can run a lot of processes at a time, which can slow down the speed of some high priority processes and result in poor performance.

To avoid this, you can tell your machine to prioritize processes as per your requirements.

This priority is called Niceness in Linux, and it has a value between -20 to 19. The lower the Niceness index, the higher would be a priority given to that task.

The default value of all the processes is 0.

To start a process with a niceness value other than the default value use the following syntax

```
nice -n 'Nice value' process name
```

```
home@VirtualBox:~$ nice -n 19 banshee
```

If there is some process already running on the system, then you can 'Renice' its value using syntax.

```
renice 'nice value' -p 'PID'
```

To change Niceness, you can use the 'top' command to determine the PID (process id) and its Nice value. Later use the renice command to change the value.

Let us understand this by an example.

Checking the niceness value of the process 'banshee'

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
3293	home	20	0	277m	64m	35m	S	96.4	6.4	9:56.72	banshee

Renicing the value to -20

```
home@VirtualBox:~$ sudo renice -20 -p 3293
[sudo] password for home:
3293 (process ID) old priority 0, new priority -20
```

The value changed to -20

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
3293	home	0	-20	277m	64m	35m	S	95.2	6.4	3:32.95	banshee

4. DF

This utility reports the free disk space(Hard Disk) on all the file systems.

```
guru99@guru99-VirtualBox:~$ df
Filesystem      1K-blocks    Used Available Use% Mounted on
/dev/sda1        7837756 2921376   4523216  40% /
udev             246488      4    246484    1% /dev
tmpfs            101512     752    100760    1% /run
none              5120        0      5120    0% /run/lock
none             253776      76    253700    1% /run/shm
```

If you want the above information in a readable format, then use the command

```
'df -h'
```

```
guru99@guru99-VirtualBox:~$ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda1        7.5G  2.8G  4.4G  40% /
udev             241M  4.0K  241M   1% /dev
tmpfs            100M  752K   99M   1% /run
none             5.0M    0   5.0M   0% /run/lock
none             248M  76K  248M   1% /run/shm
```

Other Commands

1.grep

The grep filter searches a file for a particular pattern of characters, and displays all lines that contain that pattern. The pattern that is searched in the file is referred to as the regular expression (grep stands for globally search for regular expression and print out).

Syntax:

grep [options] pattern [files]

Options Description

- c** : This prints only a count of the lines that match a pattern
- h** : Display the matched lines, but do not display the filenames.
- i** : Ignores, case for matching
- l** : Displays list of a filenames only.
- n** : Display the matched lines and their line numbers.
- v** : This prints out all the lines that do not matches the pattern
- e exp** : Specifies expression with this option. Can use multiple times.
- f file** : Takes patterns from file, one per line.
- E** : Treats pattern as an extended regular expression (ERE)
- w** : Match whole word
- o** : Print only the matched parts of a matching line, with each such part on a separate output line.

Sample Commands

Consider the below file as an input.

```
$cat > geekfile.txt
```

```
unix is great os. unix is opensource. unix is free os.
```

```
learn operating system.
```

```
Unix linux which one you choose.
```

```
uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
```

1. Case insensitive search : The -i option enables to search for a string case insensitively in the give file. It matches the words like “UNIX”, “Unix”, “unix”.

```
$grep -i "UNix" geekfile.txt
```

Output:

```
unix is great os. unix is opensource. unix is free os.
```

```
Unix linux which one you choose.
```

```
uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
```

2. Displaying the count of number of matches : We can find the number of lines that matches the given string/pattern

```
$grep -c "unix" geekfile.txt
```

Output:

```
2
```

3. Display the file names that matches the pattern : We can just display the files that contains the given string/pattern.

```
$grep -l "unix" *
```

or

```
$grep -l "unix" f1.txt f2.txt f3.txt f4.txt
```

Output:

```
geekfile.txt
```

4. Checking for the whole words in a file : By default, grep matches the given string/pattern even if it found as a substring in a file. The -w option to grep makes it match only the whole words.

```
$ grep -w "unix" geekfile.txt
```

Output:

```
unix is great os. unix is opensource. unix is free os.
```

```
uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
```

5. Displaying only the matched pattern : By default, grep displays the entire line which has the matched string. We can make the grep to display only the matched string by using the -o option.

```
$ grep -o "unix" geekfile.txt
```

Output:

```
unix
```

```
unix
```

```
unix
```

```
unix
```

```
unix
```

```
unix
```

6. Show line number while displaying the output using grep -n : To show the line number of file with the line matched.

```
$ grep -n "unix" geekfile.txt
```

Output:

```
1:unix is great os. unix is opensource. unix is free os.
```

```
4:uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
```

7. Inverting the pattern match : You can display the lines that are not matched with the specified search sting pattern using the -v option.

```
$ grep -v "unix" geekfile.txt
```

Output:

```
learn operating system.
```

```
Unix linux which one you choose.
```

8. Matching the lines that start with a string : The ^ regular expression pattern specifies the start of a line. This can be used in grep to match the lines which start with the given string or pattern.

```
$ grep "^unix" geekfile.txt
```

Output:

```
unix is great os. unix is opensource. unix is free os.
```

9. Matching the lines that end with a string : The \$ regular expression pattern specifies the end of a line. This can be used in grep to match the lines which end with the given string or pattern.

```
$ grep "os$" geekfile.txt
```

10.Specifies expression with -e option. Can use multiple times :

```
$grep -e "Agarwal" -e "Aggarwal" -e "Agrawal" geekfile.txt
```

11. -f file option Takes patterns from file, one per line.

```
$cat pattern.txt
```

```
Agarwal
```

```
Aggarwal
```

```
Agrawal
```

```
$grep -f pattern.txt geekfile.txt
```

2.cut

The cut command in UNIX is a command for cutting out the sections from each line of files and writing the result to standard output. It can be used to cut parts of a line by **byte position, character and field**. Basically the cut command slices a line and extracts the text. It is necessary to specify option with command otherwise it gives error. If more than one file name is provided then data from each file is **not precedes** by its file name.

Syntax:

```
cut OPTION... [FILE]...
```

Let us consider two files having name **state.txt** and **capital.txt** contains 5 names of the Indian states and capitals respectively.

```
$ cat state.txt
```

```
Andhra Pradesh
```

```
Arunachal Pradesh
```

```
Assam
```

Bihar

Chhattisgarh

Without any option specified it displays error.

\$ cut state.txt

cut: you must specify a list of bytes, characters, or fields

Try 'cut --help' for more information.

3.finger

Finger command is used in Linux and Unix-like system to check the information of any currently logged in users from the terminal. It is a command-line utility that can provide users login time, tty (name), idle time, home directory, shell name, etc.

Finger package is not installed by default in most Linux and Ubuntu, other Debian flavored systems. In this tutorial, we will check how to install and use finger command in Linux.

Finger Command Syntax

The following command shows the syntax of finger command:

```
finger [-lmsp] [user1 user2 ..... ]
```

4.suid

SUID is a special file permission for executable files which enables other users to run the file with effective permissions of the file owner. Instead of the normal `x` which represents execute permissions, you will see an `s` (to indicate SUID) special permission for the user.

This below example command will find all files with SUID set in the current directory using -perm (print files only with permissions set to 4000) option.

```
$ find . -perm /4000
```

```

aaronkilik@tecmint ~ $ find . -perm /4000
./backup.sh
./update.sh
./diskusage.sh
aaronkilik@tecmint ~ $ ls -l backup.sh
-rwxrwx--- 1 aaronkilik aaronkilik 0 Aug  3 22:06 backup.sh
aaronkilik@tecmint ~ $ ls -l update.sh
-rwxrws--- 1 aaronkilik aaronkilik 0 Aug  3 22:06 update.sh
aaronkilik@tecmint ~ $ ls -l diskusage.sh
-rwxrws--- 1 aaronkilik aaronkilik 20 Aug  3 22:04 diskusage.sh
aaronkilik@tecmint ~ $

```

Find Files with SUID Permissions

You can use the [ls command](#) with `-l` option (for long listing) to view the permissions on the listed files as shown in the image above.

5.wc

wc stands for **word count**. As the name implies, it is mainly used for counting purpose.

- It is used to find out **number of lines, word count, byte and characters count** in the files specified in the file arguments.
- By default it displays **four-columnar output**.
- First column shows number of lines present in a file specified, second column shows number of words present in the file, third column shows number of characters present in file and fourth column itself is the file name which are given as argument.

Syntax:

wc [OPTION]... [FILE]...

Let us consider two files having name **state.txt** and **capital.txt** containing 5 names of the Indian states and capitals respectively.

```

$ cat state.txt
Andhra Pradesh
Arunachal Pradesh
Assam
Bihar
Chhattisgarh

```

```

$ cat capital.txt
Hyderabad
Itanagar
Dispur
Patna
Raipur

```

Passing only one file name in the argument.

```
$ wc state.txt
5 7 63 state.txt
OR
$ wc capital.txt
5 5 45 capital.txt
```

6.chmod

In Unix-like operating systems, the **chmod** command is used to change the access mode of a file. The name is an abbreviation of **change mode**.

Syntax :

```
chmod [reference][operator][mode] file...
```

The references are used to distinguish the users to whom the permissions apply i.e. they are list of letters that specifies whom to give permissions. The references are represented by one or more of the following letters:

Reference	Class	Description
u	owner	file's owner
g	group	users who are members of the file's group
o	others	users who are neither the file's owner nor members of the file's group
a	all	All three of the above, same as ugo

The operator is used to specify how the modes of a file should be adjusted. The following operators are accepted:

Operator	Description
+	Adds the specified modes to the specified classes
-	Removes the specified modes from the specified classes

= The modes specified are to be made
the exact modes for the specified
classes

Note : Putting blank space(s) around operator would make the command fail.
The modes indicate which permissions are to be granted or removed from the specified classes.
There are three basic modes which correspond to the basic permissions:

r Permission to read the file.
w Permission to write (or delete) the file.
x Permission to execute the file, or, in
the case of a directory, search it.

Types of permissions which we will be changing using `chmod` command :
In linux terminal, to see all the permissions to different files, type `ls -l` command which lists the files in the working directory in long format.

7.man

man command in Linux is used to display the user manual of any command that we can run on the terminal. It provides a detailed view of the command which includes NAME, SYNOPSIS, DESCRIPTION, OPTIONS, EXIT STATUS, RETURN VALUES, ERRORS, FILES, VERSIONS, EXAMPLES, AUTHORS and SEE ALSO.

Every manual is divided into the following sections:

- Executable programs or shell commands
- System calls (functions provided by the kernel)
- Library calls (functions within program libraries)
- Games
- Special files (usually found in `/dev`)
- File formats and conventions eg `/etc/passwd`
- Miscellaneous (including macro packages and conventions), e.g. `groff(7)`
- System administration commands (usually only for root)
- Kernel routines [Non standard]

Syntax :

```
$man [OPTION]... [COMMAND NAME]...
```

Options and Examples

1. No Option: It displays the whole manual of the command.

Syntax :

```
$ man [COMMAND NAME]
```

Example:

```
$ man printf
```

Output:

In this example, manual pages of the command '*printf*' are simply returned.

8.wall

wall command in Linux system is used to write a message to all users. This command displays a message, or the contents of a file, or otherwise its standard input, on the terminals of all currently logged in users. The lines which will be longer than 79 characters, wrapped by this command. Short lines are whitespace padded to have 79 characters. A carriage return and newline at the end of each line is put by *wall* command always. Only the superuser can write on the terminals of users who have chosen to deny messages or are using a program which automatically denies messages. Reading from a file is refused when the invoker is not superuser and the program is *suid(set-user-ID)* or *sgid(set-group-ID)*.

Syntax:

```
wall [-n] [-t timeout] [message | file]
```

Options:

- **wall -n:** This option will suppress the banner.

```
wall -n
```
- **wall -t:** This option will abandon the write attempt to the terminals after timeout seconds. This timeout needs to be a positive integer. The by default value is 300 seconds, which is a legacy from the time when peoples ran terminals over modem lines.

Example:

```
wall -t 30
```

- **wall -V :** This option display version information and exit.

```
wall -V
```

- **wall -h :** This option will display help message and exit.

```
wall -h
```

9.sort

SORT command is used to sort a file, arranging the records in a particular order. By default, the sort command sorts file assuming the contents are ASCII. Using options in sort command, it can also be used to sort numerically.

- SORT command sorts the contents of a text file, line by line.
- sort is a standard command line program that prints the lines of its input or concatenation of all files listed in its argument list in sorted order.
- The sort command is a command line utility for sorting lines of text files. It supports sorting alphabetically, in reverse order, by number, by month and can also remove duplicates.
- The sort command can also sort by items not at the beginning of the line, ignore case sensitivity and return whether a file is sorted or not. Sorting is done based on one or more sort keys extracted from each line of input.
- By default, the entire input is taken as sort key. Blank space is the default field separator.

The sort command follows these features as stated below:

1. Lines starting with a number will appear before lines starting with a letter.
2. Lines starting with a letter that appears earlier in the alphabet will appear before lines starting with a letter that appears later in the alphabet.
3. Lines starting with a lowercase letter will appear before lines starting with the same letter in uppercase.

Examples

Suppose you create a data file with name file.txt

Command :

```
$ cat > file.txt
```

```
abhishek
```

```
chitransh
```

```
satish
```

```
rajan
```

```
naveen
```

```
divyam
```

```
harsh
```

Sorting a file : Now use the sort command Syntax :

\$ sort filename.txt

Command:

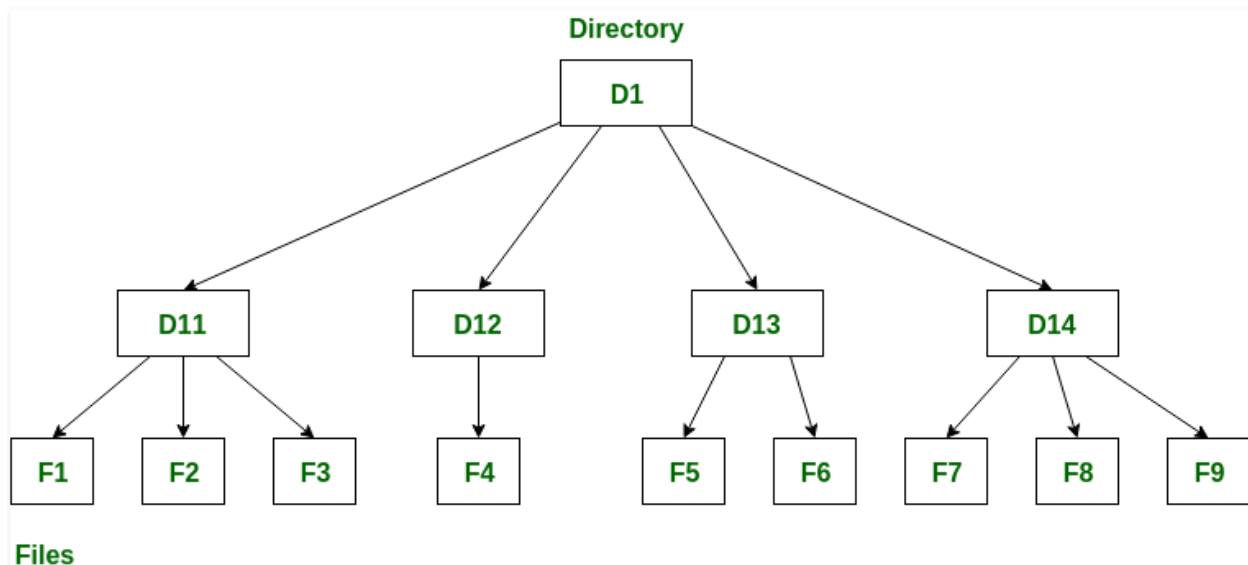
```
$ sort file.txt
```

Output :

abhishek
chitransh
divyam
harsh
naveen
rajan
satish

Directory Structure

A **directory** is a container that is used to contain folders and file. It organizes files and folders into a hierarchical manner.

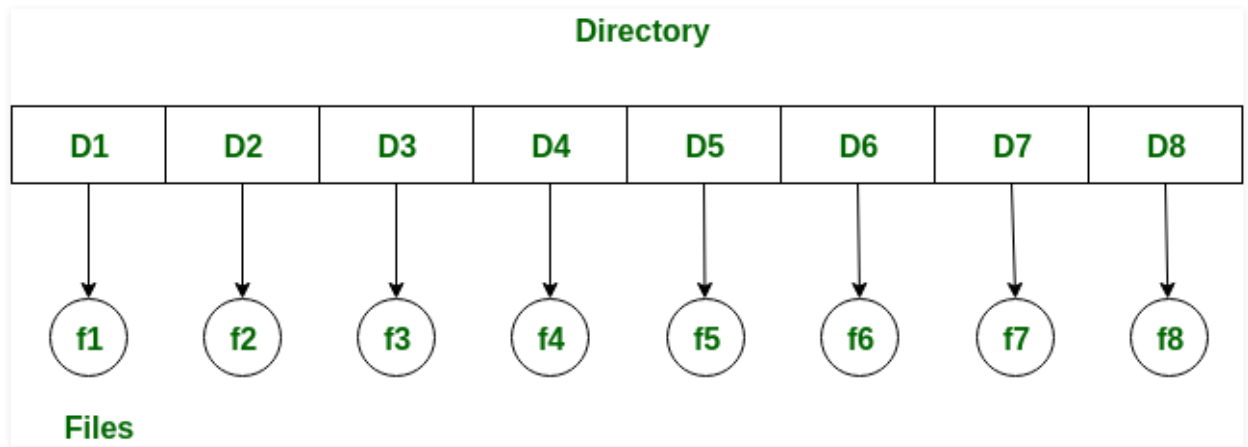


There are several logical structures of a directory, these are given below.

1. Single-level directory –

Single level directory is simplest directory structure. In it all files are contained in same directory which make it easy to support and understand.

A single level directory has a significant limitation, however, when the number of files increases or when the system has more than one user. Since all the files are in the same directory, they must have the unique name. If two users call their dataset test, then the unique name rule is violated.



Advantages:

- Since it is a single directory, so its implementation is very easy.
- If the files are smaller in size, searching will become faster.
- The operations like file creation, searching, deletion, updating are very easy in such a directory structure.

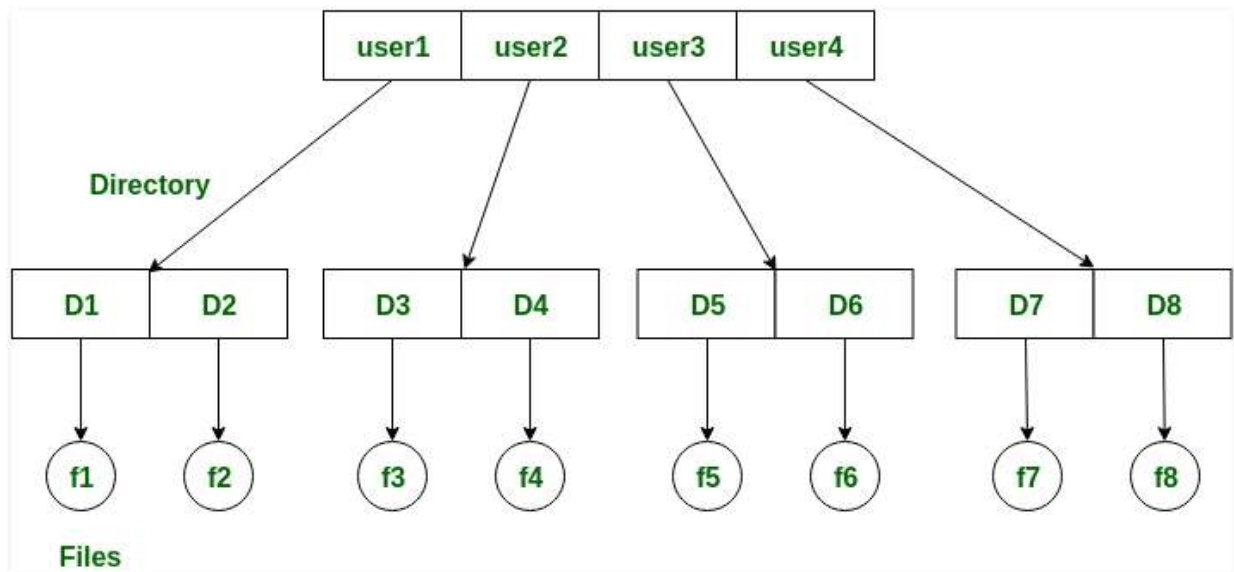
Disadvantages:

- There may chance of name collision because two files can not have the same name.
- Searching will become time taking if the directory is large.
- In this can not group the same type of files together.

2. Two-level directory –

As we have seen, a single level directory often leads to confusion of files names among different users. the solution to this problem is to create a separate directory for each user. In the two-level directory structure, each user has there own *user files directory (UFD)*. The UFDs has similar structures, but each lists only the files of a single user. system's *master file directory (MFD)* is searches whenever a new user id=s logged in. The MFD is indexed by username or account number, and each entry points to the UFD for that

user.



Advantages:

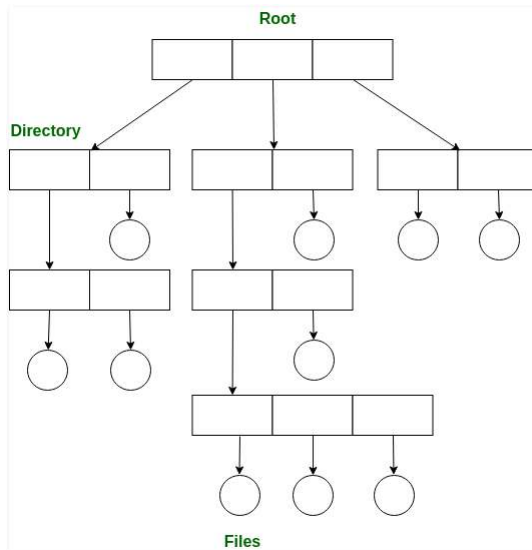
- We can give full path like /User-name/directory-name/.
- Different users can have same directory as well as file name.
- Searching of files become more easy due to path name and user-grouping.

Disadvantages:

- A user is not allowed to share files with other users.
- Still it not very scalable, two files of the same type cannot be grouped together in the same user.

3. Tree-structured directory –

Once we have seen a two-level directory as a tree of height 2, the natural generalization is to extend the directory structure to a tree of arbitrary height. This generalization allows the user to create there own subdirectories and to organize on their files accordingly.



A tree structure is the most common directory structure. The tree has a root directory, and every file in the system have a unique path.

Advantages:

- Very generalize, since full path name can be given.
- Very scalable, the probability of name collision is less.
- Searching becomes very easy, we can use both absolute path as well as relative.

Disadvantages:

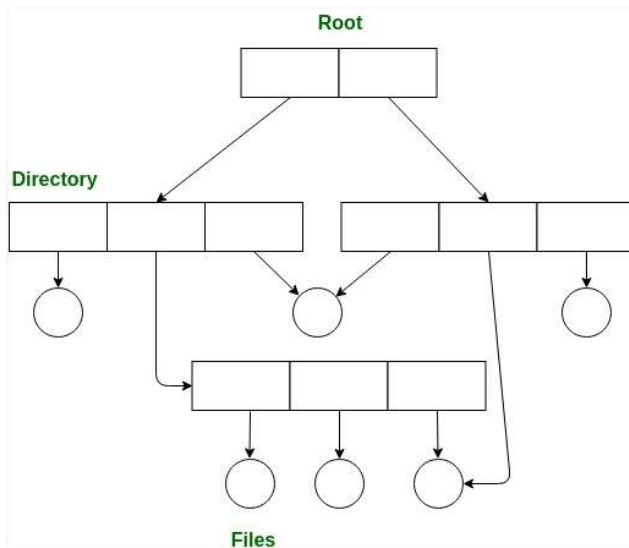
- Every file does not fit into the hierarchical model, files may be saved into multiple directories.
- We can not share files.
- It is inefficient, because accessing a file may go under multiple directories.
-

4. Acyclic graph directory –

An acyclic graph is a graph with no cycle and allows to share subdirectories and files. The same file or subdirectories may be in two different directories. It is a natural generalization of the tree-structured directory.

It is used in the situation like when two programmers are working on a joint project and they need to access files. The associated files are stored in a subdirectory, separating them from other projects and files of other programmers, since they are working on a joint project so they want the subdirectories to be into their own directories. The common subdirectories should be shared. So here we use Acyclic directories.

It is the point to note that shared file is not the same as copy file . If any programmer makes some changes in the subdirectory it will reflect in both subdirectories.



Advantages:

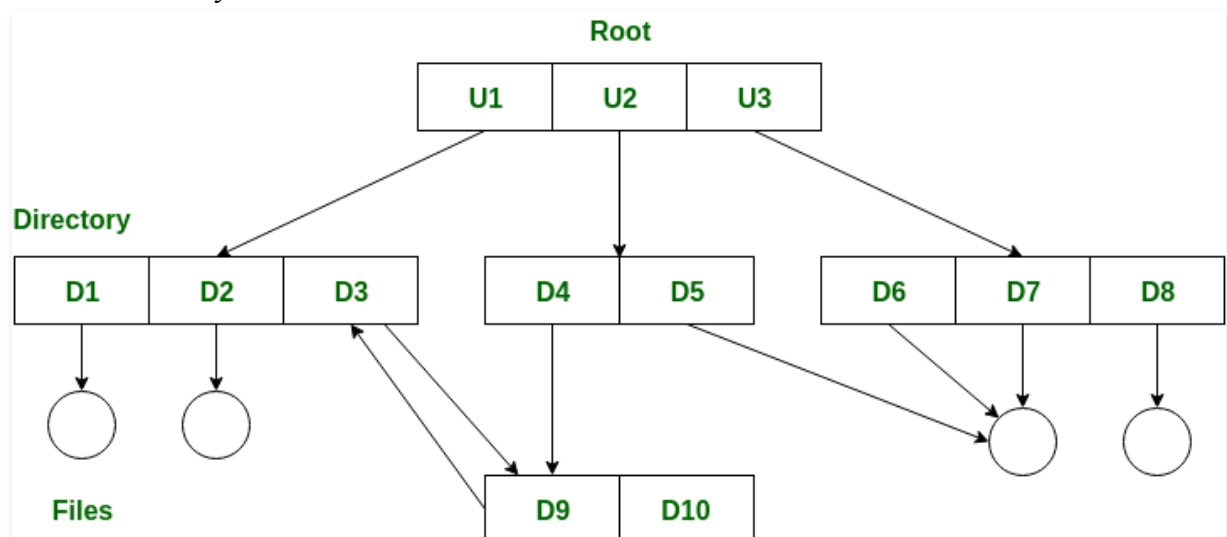
- We can share files.
- Searching is easy due to different-different paths.

Disadvantages:

- We share the files via linking, in case of deleting it may create the problem,
- If the link is softlink then after deleting the file we left with a dangling pointer.
- In case of hardlink, to delete a file we have to delete all the reference associated with it.

5. General graph directory structure –

In general graph directory structure, cycles are allowed within a directory structure where multiple directories can be derived from more than one parent directory. The main problem with this kind of directory structure is to calculate total size or space that has been taken by the files and directories.



Advantages:

- It allows cycles.
- It is more flexible than other directories structure.

Disadvantages:

- It is more costly than others.
- It needs garbage collection.

What is I-Node? Explain in Detail.

In Unix based operating system each file is indexed by an **Inode**. Inode are special disk blocks they are created when the file system is created. The number of Inode limits the total number of files/directories that can be stored in the file system.

The Inode contains the following information:

- Administrative information (permissions, timestamps, etc).
- A number of direct blocks (typically 12) that contains to the first 12 blocks of the files.
- A single indirect pointer that points to a disk block which in turn is used as an index block, if the file is too big to be indexed entirely by the direct blocks.
- A double indirect pointer that points to a disk block which is a collection of pointers to disk blocks which are index blocks, used if the file is too big to be indexed by the direct and single indirect blocks.
- A triple indirect pointer that points to an index block of index blocks of index blocks.

Inode Total Size:

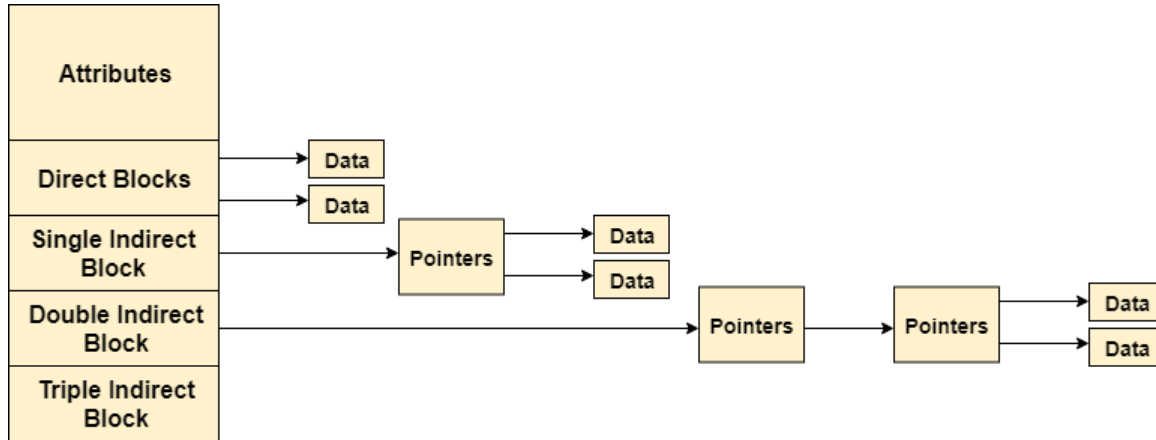
- Number of disk block address possible to store in 1 disk block = (Disk Block Size / Disk Block Address).
- Small files need only the direct blocks, so there is little waste in space or extra disk reads in those cases. Medium sized files may use indirect blocks. Only large files make use of the double or triple indirect blocks, and that is reasonable since those files are large anyway. The disk is now broken into two different types of blocks: **Inode and Data Blocks**.
- There must be some way to determine where the Inodes are, and to keep track of free Inodes and disk blocks. This is done by a **Superblock**. Superblock is located at a fixed position in the file system. The Superblock is usually replicated on the disk to avoid catastrophic failure in case of corruption of the main Superblock.
- Index allocation schemes suffer from some of the same performance problems. As does linked allocation. For example, the index blocks can be cached in memory, but the data blocks may be spread all over a partition.

In UNIX based operating systems, each file is indexed by an Inode. Inode are the special disk block which is created with the creation of the file system. The number of files or directories in a file system depends on the number of Inodes in the file system.

An Inode includes the following information

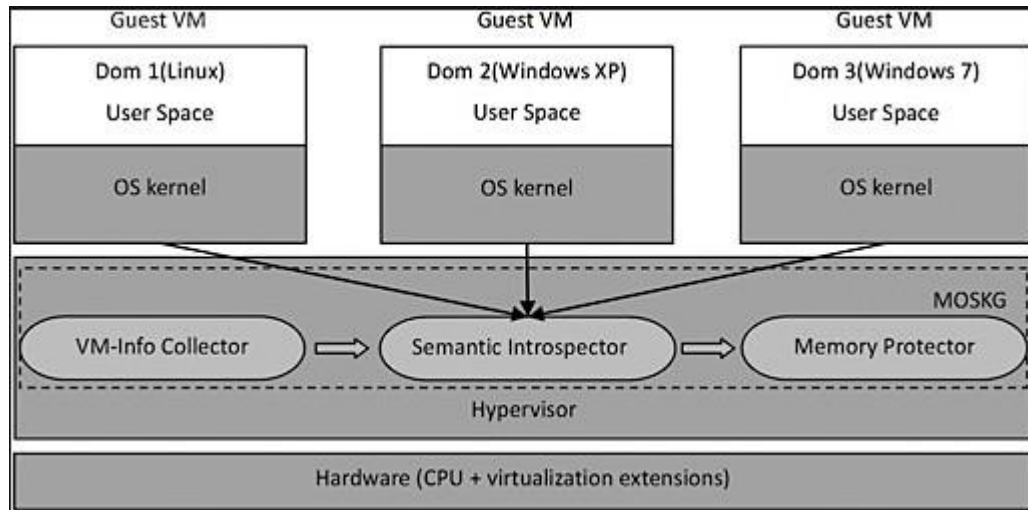
1. Attributes (permissions, time stamp, ownership details, etc) of the file
2. A number of direct blocks which contains the pointers to first 12 blocks of the file.

3. A single indirect pointer which points to an index block. If the file cannot be indexed entirely by the direct blocks then the single indirect pointer is used.
4. A double indirect pointer which points to a disk block that is a collection of the pointers to the disk blocks which are index blocks. Double index pointer is used if the file is too big to be indexed entirely by the direct blocks as well as the single indirect pointer.
5. A triple index pointer that points to a disk block that is a collection of pointers. Each of the pointers is separately pointing to a disk block which also contains a collection of pointers which are separately pointing to an index block that contains the pointers to the file blocks.



UNIT 10 : Virtualization Concepts

Virtual Machines



Virtual Machine abstracts the hardware of our personal computer such as CPU, disk drives, memory, NIC (Network Interface Card) etc, into many different execution environments as per our requirements, hence giving us a feel that each execution environment is a single computer. For example, VirtualBox.

When we run different processes on an operating system, it creates an illusion that each process is running on a different processor having its own virtual memory, with the help of CPU scheduling and virtual-memory techniques. There are additional features of a process that cannot be provided by the hardware alone like system calls and a file system. The virtual machine approach does not provide these additional functionalities but it only provides an interface that is same as basic hardware. Each process is provided with a virtual copy of the underlying computer system.

We can create a virtual machine for several reasons, all of which are fundamentally related to the ability to share the same basic hardware yet can also support different execution environments, i.e., different operating systems simultaneously.

The main drawback with the virtual-machine approach involves disk systems. Let us suppose that the physical machine has only three disk drives but wants to support seven virtual machines. Obviously, it cannot allocate a disk drive to each virtual machine, because virtual-machine software itself will need substantial disk space to provide virtual memory and spooling. The solution is to provide virtual disks.

Users are thus given their own virtual machines. After which they can run any of the operating systems or software packages that are available on the underlying machine. The virtual-machine software is concerned with multi-programming multiple virtual machines onto a physical machine, but it does not need to consider any user-support software. This arrangement can

provide a useful way to divide the problem of designing a multi-user interactive system, into two smaller pieces.

Advantages:

1. There are no protection problems because each virtual machine is completely isolated from all other virtual machines.
2. Virtual machine can provide an instruction set architecture that differs from real computers.
3. Easy maintenance, availability and convenient recovery.

Disadvantages:

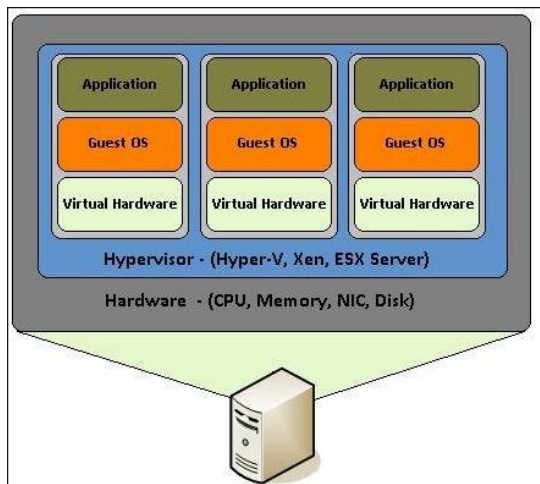
1. When multiple virtual machines are simultaneously running on a host computer, one virtual machine can be affected by other running virtual machines, depending on the workload.
2. Virtual machines are not as efficient as a real one when accessing the hardware.

Virtualization

Operating system based Virtualization refers to an operating system feature in which the kernel enables the existence of various isolated user-space instances. The installation of virtualization software also refers to Operating system-based virtualization. It is installed over a pre-existing operating system and that operating system is called the host operating system.

In this virtualization, a user installs the virtualization software in the operating system of his system like any other program and utilize this application to operate and generate various virtual machines. Here, the virtualization software allows direct access to any of the created virtual machine to the user. As the host OS can provide hardware devices with the mandatory support, operating system virtualization may affect compatibility issues of hardware even when the hardware driver is not allocated to the virtualization software.

Virtualization software is able to convert hardware IT resources which require unique software for operation into virtualized IT resources. As the host OS is a complete operating system in itself, many OS based services are available as organizational management and administration tools can be utilized for the virtualization host management.



Some major operating system-based services are mentioned below:

1. Backup and Recovery.
2. Security Management.
3. Integration to Directory Services.

Various major operations of Operating System Based Virtualization are described below:

1. Hardware capabilities which can be employed, such as the network connection and CPU.
2. Connected peripherals with which it can interact with, such as webcam, printer, keyboard, or Scanners.
3. Data which can be read or written, such as files, folders and network shares.

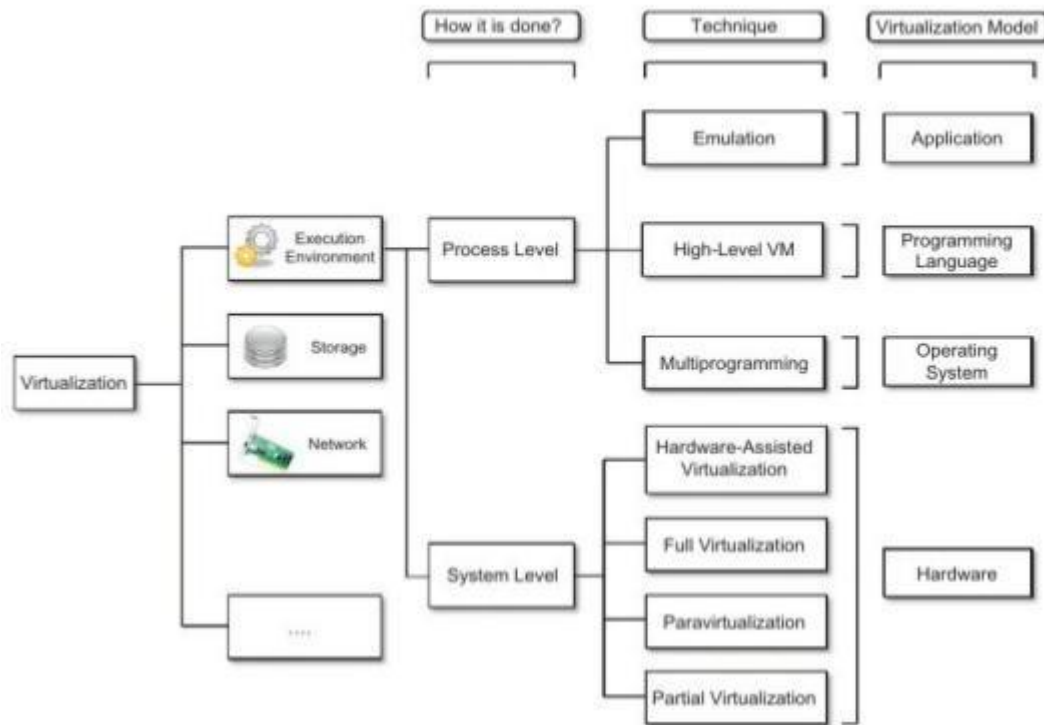
The Operating system may have the capability to allow or deny access to such resources based on which program requests them and the user account in the context of which it runs. OS may also hide these resources, which leads that when computer program computes them, they do not appear in the enumeration results. Nevertheless, from a programming perspective, the computer program has interacted with those resources and the operating system has managed an act of interaction.

With operating-system-virtualization, or containerization, it is probable to run programs within containers, to which only parts of these resources are allocated. A program which is expected to perceive the whole computer, once run inside a container, can only see the allocated resources and believes them to be all that is available. Several containers can be formed on each operating system, to each of which a subset of the computer's resources is allocated. Each container may include many computer programs. These programs may run parallel or distinctly, even interrelate with each other.

Operating system-based virtualization can raise demands and problems related to performance overhead, such as:

1. The host operating system employs CPU, memory, and other hardware IT resources.
2. Hardware-related calls from guest operating systems need to navigate numerous layers to and from the hardware, which shrinkage overall performance.
3. Licenses are frequently essential for host operating systems, in addition to individual licenses for each of their guest operating systems.

Taxonomy of virtualization



Hypervisors

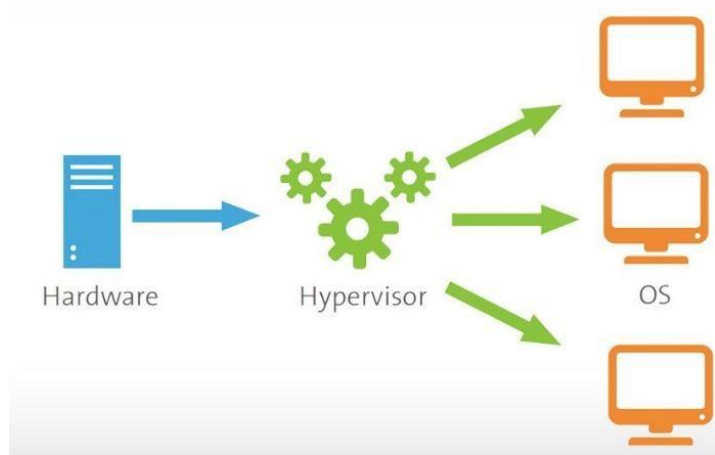
Hypervisor is a form of virtualization software used in Cloud hosting to divide and allocate the resources on various pieces of hardware. The program which provides partitioning, isolation or abstraction is called a virtualization hypervisor. Hypervisor is a hardware virtualization technique that allows multiple guest operating systems (OS) to run on a single host system at the same time. A hypervisor is sometimes also called a virtual machine manager (VMM).

Types of Hypervisor –

TYPE-1

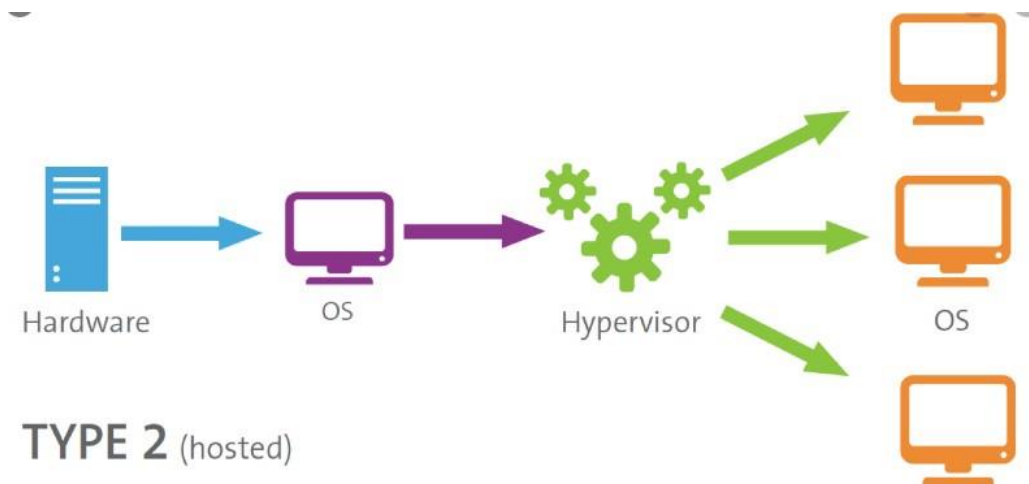
Hypervisor runs directly on the underlying host system. It is also known as “Native Hypervisor” or “Bare metal hypervisor”. It does not require any base server operating system. It has direct access to hardware resources. Examples of Type 1 hypervisors include VMware ESXi, Citrix XenServer, and Microsoft Hyper-V hypervisor.

Hypervisor:



TYPE-2 Hypervisor:

A Host operating system runs on underlying host system. It is also known as ‘Hosted Hypervisor’. Basically a software installed on an operating system. Hypervisor asks operating system to make hardware calls. Example of Type 2 hypervisor include VMware Player or Parallels Desktop. Hosted hypervisors are often found on endpoints like PCs.



Choosing the right hypervisor

Type 1 hypervisors offer much better performance than Type 2 ones because there’s no middle layer, making them the logical choice for mission-critical applications and workloads. But that’s not to say that hosted hypervisors don’t have their place – they’re much simpler to set up, so they’re a good bet if, say, you need to deploy a test environment quickly. One of the best ways to determine which hypervisor meets your needs is to compare their performance metrics. These include CPU overhead, amount of maximum host and guest memory, and support for virtual processors. The following factors should be examined before choosing a suitable hypervisor:

1. Understand your needs: The company and its applications are the reason for the data center (and your job). Besides your company’s needs, you (and your co-workers in IT) also have your own needs. Needs for a virtualization hypervisor are:

- a. Flexibility
- b. Scalability
- c. Usability
- d. Availability
- e. Reliability
- f. Efficiency
- g. Reliable support

2. The cost of a hypervisor: For many buyers, the toughest part of choosing a hypervisor is striking the right balance between cost and functionality. While a number of entry-level solutions are free, or practically free, the prices at the opposite end of the market can be staggering. Licensing frameworks also vary, so it's important to be aware of exactly what you're getting for your money.

3. Virtual machine performance: Virtual systems should meet or exceed the performance of their physical counterparts, at least in relation to the applications within each server. Everything beyond meeting this benchmark is profit.

4. Ecosystem: It's tempting to overlook the role of a hypervisor's ecosystem – that is, the availability of documentation, support, training, third-party developers and consultancies, and so on – in determining whether or not a solution is cost-effective in the long term.

5. Test for yourself: You can gain basic experience from your existing desktop or laptop. You can run both VMware vSphere and Microsoft Hyper-V in either VMware Workstation or VMware Fusion to create a nice virtual learning and testing environment.

HYPERVISOR REFERENCE MODEL

There are 3 main modules coordinate in order to emulate the underlying hardware:

- 1. Dispatcher
- 2. Allocator
- 3. Interpreter

DISPATCHER:

The dispatcher behaves like the entry point of the monitor and reroutes the instructions of the virtual machine instance to one of the other two modules.

ALLOCATOR:

The allocator is responsible for deciding the system resources to be provided to the virtual machine instance. It means whenever virtual machine tries to execute an instruction that results in changing the machine resources associated with the virtual machine, the allocator is invoked by the dispatcher.

INTERPRETER:

The interpreter module consists of interpreter routines. These are executed, whenever virtual machine executes a privileged instruction.

Choosing the right hypervisor

Type 1 hypervisors offer much better performance than Type 2 ones because there's no middle layer, making them the logical choice for mission-critical applications and workloads. But that's not to say that hosted hypervisors don't have their place – they're much simpler to set up, so they're a good bet if, say, you need to deploy a test environment quickly. One of the best ways to determine which hypervisor meets your needs is to compare their performance metrics.

These include CPU overhead, amount of maximum host and guest memory, and support for virtual processors. The following factors should be examined before choosing a suitable hypervisor:

1. Understand your needs: The company and its applications are the reason for the data center (and your job). Besides your company's needs, you (and your co-workers in IT) also have your own needs. Needs for a virtualization hypervisor are:

- a. Flexibility
- b. Scalability
- c. Usability
- d. Availability
- e. Reliability
- f. Efficiency
- g. Reliable support

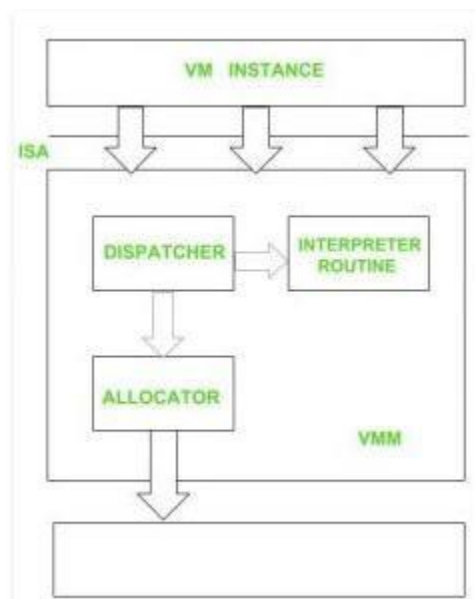
2. The cost of a hypervisor: For many buyers, the toughest part of choosing a hypervisor is striking the right balance between cost and functionality. While a number of entry-level solutions are free, or practically free, the prices at the opposite end of the market can be staggering. Licensing frameworks also vary, so it's important to be aware of exactly what you're getting for your money.

3. Virtual machine performance: Virtual systems should meet or exceed the performance of their physical counterparts, at least in relation to the applications within each server. Everything beyond meeting this benchmark is profit.

4. Ecosystem: It's tempting to overlook the role of a hypervisor's ecosystem – that is, the availability of documentation, support, training, third-party developers and consultancies, and so on – in determining whether or not a solution is cost-effective in the long term.

5. Test for yourself: You can gain basic experience from your existing desktop or laptop. You can run both VMware vSphere and Microsoft Hyper-V in either VMware Workstation or VMware Fusion to create a nice virtual learning and testing environment.

HYPERVISOR REFERENCE MODEL



There are 3 main modules coordinate in order to emulate the underlying hardware:

1. Dispatcher
2. Allocator
3. Interpreter

DISPATCHER:

The dispatcher behaves like the entry point of the monitor and reroutes the instructions of the virtual machine instance to one of the other two modules.

ALLOCATOR:

The allocator is responsible for deciding the system resources to be provided to the virtual machine instance. It means whenever virtual machine tries to execute an instruction that results in changing the machine resources associated with the virtual machine, the allocator is invoked by the dispatcher.

INTERPRETER:

The interpreter module consists of interpreter routines. These are executed, whenever virtual machine executes a privileged instruction.

Full Virtualization:

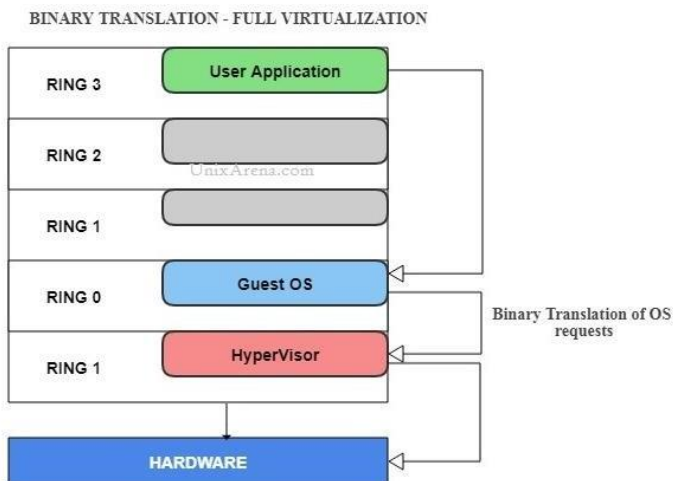
Virtual machine simulates hardware to allow an unmodified guest OS to be run in isolation. There are two types of Full virtualizations in the enterprise market. On both full virtualization types, guest operating system's source information will not be modified.

- Software assisted full virtualization
- Hardware-assisted full virtualization

Software Assisted – Full Virtualization (BT – Binary Translation)

It completely relies on binary translation to trap and virtualize the execution of sensitive, non-virtualizable instructions sets. It emulates the hardware using the software instruction sets. Due to binary translation, it is often criticized for performance issues. Here is the list of software which will fall under software assisted (BT).

- VMware Workstation (32-bit guests)
- Virtual PC
- Virtual Box (32-bit guests)
- VMware Server

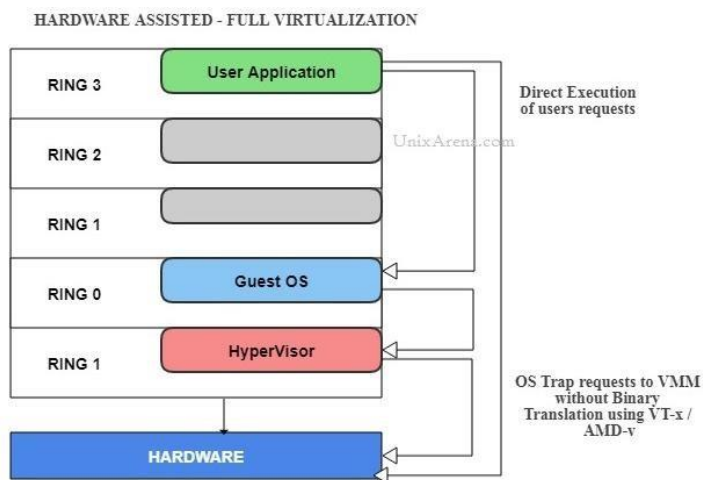


Binary Translation – Full Virtualization

Binary Translation – Full Virtualization

Hardware-Assisted – Full Virtualization (VT)

Hardware-assisted full virtualization eliminates the binary translation and it directly interrupts with hardware using the virtualization technology which has been integrated on X86 processors since 2005 (Intel VT-x and AMD-V). Guest OS's instructions might allow a virtual context execute privileged instructions directly on the processor, even though it is virtualized.



Hardware-assisted virtualization – Hypervisor

2 Paravirtualization:

Paravirtualization works differently from the full virtualization. It doesn't need to simulate the hardware for the virtual machines. The hypervisor is installed on a physical server (host) and a guest OS is installed into the environment. Virtual guests aware that it has been virtualized,

unlike the full virtualization (where the guest doesn't know that it has been virtualized) to take advantage of the functions. In this virtualization method, guest source codes will be modified with sensitive information to communicate with the host. Guest Operating systems require extensions to make API calls to the hypervisor. In full virtualization, guests will issue a hardware calls but in paravirtualization, guests will directly communicate with the host (hypervisor) using the drivers. Here is the list of products which supports paravirtualization.

- Xen
- IBM LPAR
- Oracle VM for SPARC (LDOM)

Xen supports both Full virtualization and Para-virtualization

3 Hybrid Virtualization: (Hardware Virtualized with PV Drivers)

In Hardware assisted full virtualization, Guest operating systems are unmodified and it involves many VM traps and thus high CPU overheads which limit the scalability. Paravirtualization is a complex method where guest kernel needs to be modified to inject the API. By considering these issues, engineers have come with hybrid paravirtualization. It's a combination of both Full & Paravirtualization. The virtual machine uses paravirtualization for specific hardware drivers (where there is a bottleneck with full virtualization, especially with I/O & memory intense workloads), and the host uses full virtualization for other features. The following products support hybrid virtualization.

- Oracle VM for x86
- Xen
- VMware ESXi

The following diagram will help you to understand how VMware supports both full virtualization and hybrid virtualization. RDMA uses the paravirtual driver to bypass VMkernel in hardware-assisted full virtualization.

4 OS level Virtualization:

Operating system-level virtualization is widely used. It also knowns "containerization". Host Operating system kernel allows multiple user spaces aka instance. In OS-level virtualization, unlike other virtualization technologies, there will be very little or no overhead since it uses the host operating system kernel for execution. Oracle Solaris zone is one of the famous containers in the enterprise market. Here is the list of other containers.

- Linux LCX
- Docker
- AIX WPAR Solaris Containers – OS-level virtualization Example

Memory Virtualization

Memory Virtualization Beyond CPU virtualization, the next critical component is memory virtualization. This involves sharing the physical system memory and dynamically allocating it to virtual machines. Virtual machine memory virtualization is very similar to the virtual memory support provided by modern operating systems. Applications see a contiguous address space that is not necessarily tied to the underlying physical memory in the system. The operating system keeps mappings of virtual page numbers to physical page numbers stored in page tables. All modern x86 CPUs include a memory management unit (MMU) and a translation lookaside buffer (TLB) to optimize virtual memory performance.

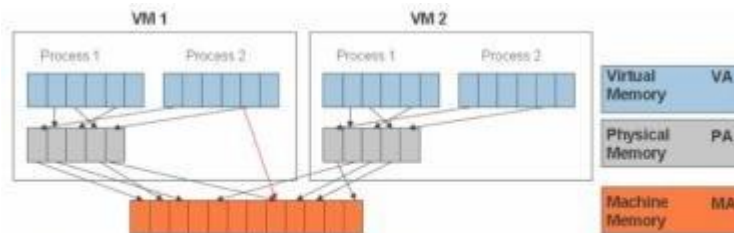


Figure 8 – Memory Virtualization

To run multiple virtual machines on a single system, another level of memory virtualization is required. In other words, one has to virtualize the MMU to support the guest OS. The guest OS continues to control the mapping of virtual addresses to the guest memory physical addresses, but the guest OS cannot have direct access to the actual machine memory. The VMM is responsible for mapping guest physical memory to the actual machine memory, and it uses shadow page tables to accelerate the mappings. As depicted by the red line in Figure 8, the VMM uses TLB hardware to map the virtual memory directly to the machine memory to avoid the two levels of translation on every access. When the guest OS changes the virtual memory to physical memory mapping, the VMM updates the shadow page tables to enable a direct lookup. MMU virtualization creates some overhead for all virtualization approaches, but this is the area where second generation hardware assisted virtualization will offer efficiency gains.

Device and I/O Virtualization

Device and I/O Virtualization The final component required beyond CPU and memory virtualization is device and I/O virtualization. This involves managing the routing of I/O requests between virtual devices and the shared physical hardware. Software based I/O virtualization and management, in contrast to a direct pass-through to the hardware, enables a rich set of features and simplified management. With networking for example, virtual NICs and switches create virtual networks between virtual machines without the network traffic consuming bandwidth on the physical network, NIC teaming allows multiple physical NICs to appear as one and failover transparently for virtual machines, and virtual machines can be seamlessly relocated to different systems using VMotion while keeping their existing MAC addresses. The key to effective I/O virtualization is to preserve these virtualization benefits while keeping the added CPU utilization to a minimum. The hypervisor virtualizes the physical hardware and presents each virtual

machine with a standardized set of virtual devices as seen in Figure 9. These virtual devices effectively emulate well-known hardware and translate the virtual machine requests to the system hardware. This standardization on consistent device drivers also helps with virtual machine standardization and portability across platforms as all virtual machines are configured to run on the same virtual hardware regardless of the actual physical hardware in the system.

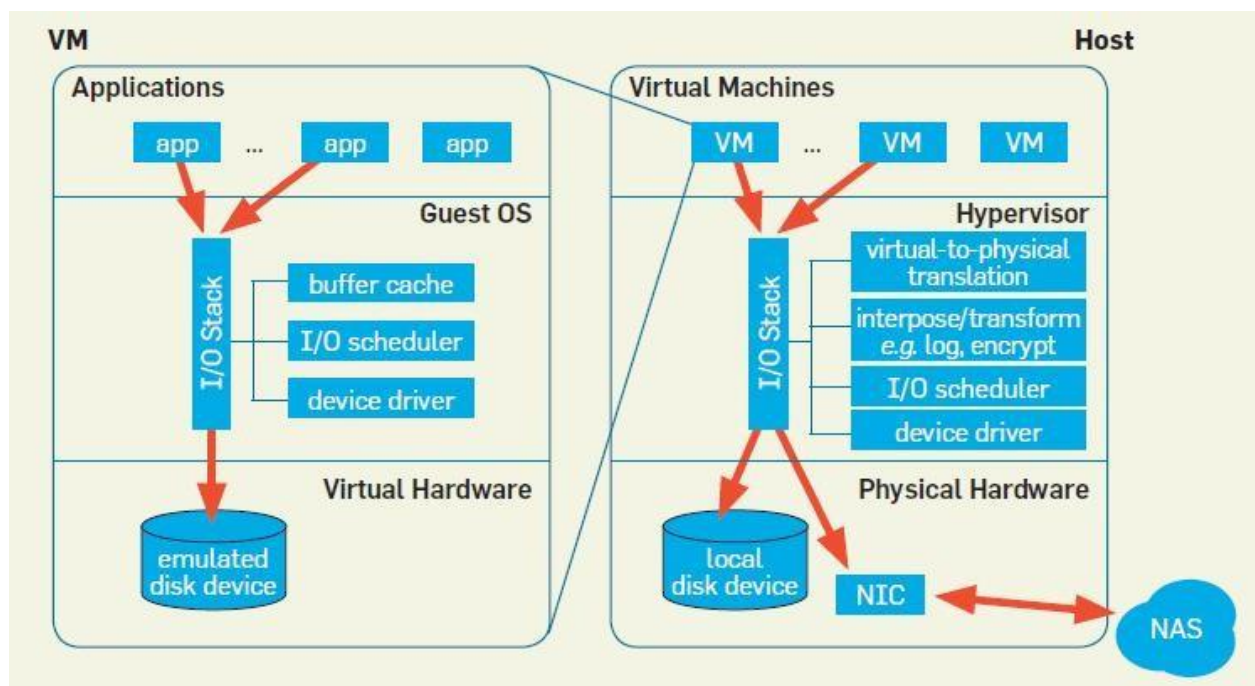


Figure: I/O Virtualization