Unit 1

Introduction to Operating System

1. Types and structure of operating system

Introduction and History of Operating System

An operating system is the program that manages all the application programs in a computer system. An OS provides the means for proper use of the resources in the operation of computer system. An OS is similar to government of a country. Like a government, it performs no useful function by itself but it simply provides an environment within which other programs can do useful work.

An **operating system** (**OS**) is a collection of software that manages computer hardware resources and provides common services for computer programs. The operating system is an essential component of the system software in a computer system.

Operating Systems have been developed over the past 40 years for two main purposes:

- First, operating systems attempt to schedule computational activities to ensure good performance of the computing system i.e. Resource manager
- Second, they provide a convenient environment for the development and execution of the programs, user machine.

OS: A Resource Manager:

A computer system has many resources that may be required to solve a problem: CPU time, memory space, files storage space, I/O devices, and so on. The OS acts as the manager of these resources. Many resources can be conflicted while requested by various users; the OS is responsible how to allocate them to specific programs and users so that it can operate the computer system efficiently and fairly.

When a computer has multiple users, the need for managing and protecting the memory, I/O devices and other devices is even greater.

- The primary task of OS is to keep track of who is using which resource, to grant resource requests, to mediate conflicting requests from different programs etc.
- Users often need to share not only hardware, but information (File, Database) as well.

Resource management includes multiplexing resources in 2 ways, they are

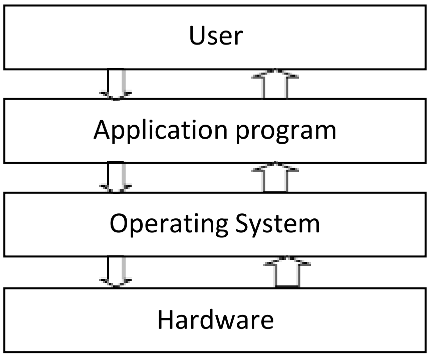
- In Time
- In Space

When Resource is Time Multiplexed

- –Who goes next and for how long‖- is the main task for time resource manager in OS.
- Different programs or user take turns using it.
- Multiple programs will run by allocating the CPU through OS.
- When another program gets to use the CPU, the OS will choose.
- Sharing the printer is the ideal example for time multiplexing

When Resource is Space Multiplexed

- Main memory is normally divided up among several running programs
- OS assume enough memory to hold multiple programs
- OS hold several programs in memory at once instead of completing one by one.
- The issues of fairness, protection and so on will solve by OS.
- Hard disk is another resource manager for space multiplexed.



OS: Virtual Machine:

A computer system consists of one or more processors, main memory and many types of I/O devices such as disk, tapes, terminals, network interfaces, etc. Writing programs for using these hardware resources correctly and efficiently is an extremely difficult job, requires depth knowledge of functions of such resources. Hence to make computer systems usable by a large number of users, OS provides a mechanism to shield programmers and other users from the complexity of hardware resources. This problem is solved by putting a layer of software on top of the bare hardware. This layer of hardware manages all hardware resources of the system, and presents the user with an interface or virtual machine that is easier, safer and efficient to program and use i.e. an OS hides details of hardware resources from programmers and other users. It provides a high level interface to low-level hardware resources, making it easier for programmers and other users to use a computer system.

In above figure, OS layer surrounds hardware resources. Then a layer of other system software (such as compiler, interpreter, editors, utilities, etc) and a set of application software (such as commercial data processing applications, scientific and engineering applications, entertainment and educational applications, etc) surrounds OS layer. Finally end users view the computer in terms of the user interfaces of the application programs.

1.1 Operating system concepts and functionalities

System Calls

A **system call** is how a program requests a service from an operating system's kernel. This may include hardware related services (e.g. accessing the hard disk), creating and executing new processes, and communicating with integral kernel services (like scheduling). System calls provide an essential interface between a process and the operating system.

A system call is a mechanism that is used by the application program to request a service from the operating system. They use a machine-code instruction that causes the processor to change mode. An example would be from supervisor mode to protected mode. This is where the operating system performs actions like accessing hardware devices or the memory management unit. Generally the operating system provides a library that sits between the operating system and normal programs.

System calls provide the interface between a process and the operating system. Most operations interacting with the system require permissions not available to a user level process, e.g. I/O performed with a device present on the system, or any form of communication with other processes requires the use of system calls. System calls can be roughly grouped into five major categories:

1. Process Control
   - load
   - execute
   - create process
   - terminate process
   - get/set process attributes
   - wait for time, wait event, signal event
   - allocate, free memory

2. File management
   - create file, delete file
   - open, close
   - read, write, reposition
   - get/set file attributes

3. Device Management
   - request device, release device
   - read, write, reposition
   - get/set device attributes
   - logically attach or detach devices

4. Information Maintenance
    - get/set time or date
    - get/set system data
    - get/set process, file, or device attributes

5. Communication
    - create, delete communication connection
    - send, receive messages
    - transfer status information
    - attach or detach remote devices

1.1.1 The Shell

Shell and the kernel are the parts of this Operating system. These both parts are used for performing any operation on the system. When a user gives his command for performing any operation, then the request will goes to the shell parts, the shell parts is also called as the interpreter which translate the human program into the machine language and then the request will be transferred to the kernel that means shell is just as the interpreter of the commands which converts the request of the user into the machine language.

Kernel is also known as heart of operating system and the every operation is performed by using the kernel**,** when the kernel receives the request from the shell then this will process the request and display the results on the screen. The various types of operations those are performed by the kernel are as followings:-

- It controls the state of the process means it checks whether the process is running or process is waiting for the request of the user.
- provides the memory for the processes those are running on the system means kernel runs the allocation and de-allocation process , first when we request for the service then the kernel will provides the memory to the process and after that it also release the memory which is given to a process.
- The kernel also maintains a time table for all the processes those are running means the kernel also prepare the schedule time means this will provide the time to various process of the CPU and the kernel also puts the waiting and suspended jobs into the different memory area.
- When a kernel determines that the logical memory doesn‘t fit to store the programs. Then he uses the concept of the physical memory which will store the programs into temporary manner i.e. virtual memory.
- Kernel also maintains all the files those are stored into the computer system and the kernel also stores all the files into the system as no one can read or write the files without any permission. So that the kernel system also provides us the facility to use the passwords and also all the files are stored into the particular manner.

As we have learned there are many programs or functions those are performed by the kernel but the functions those are performed by the kernel will never be shown to the user. And the functions of the kernel are transparent to the user.

1.2 Operating system structure

An operating system might have many structures. According to the structure of the operating system; operating systems can be classified into many categories. Some of the main structures used in operating systems are:

1. Monolithic architecture of operating system

It is the oldest architecture used for developing operating system. Operating system resides on kernel for anyone to execute. System call is involved that means switching from user mode to kernel mode and transfer control to operating system shown as event

1) Many CPU has two modes, kernel mode, for the operating system in which all instruction are allowed and user mode for user program in which I/O devices and certain other instruction are not

allowed. Two operating systems, then examines the parameter of the call to determine which system call is to be carried out shown in event

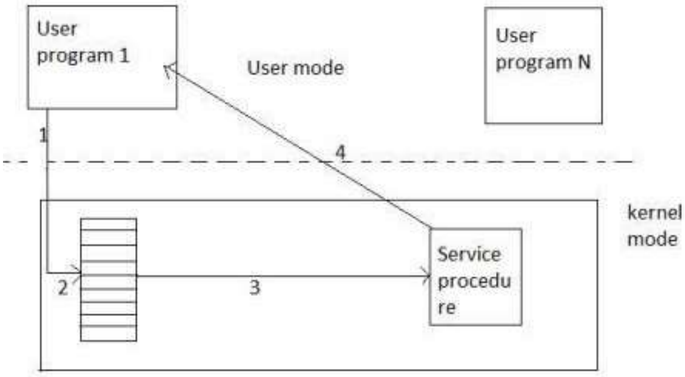


fig:- monolithic structure of os

2) Next, the operating system index‘s into a table that contains procedure that carries out system call. This operation is shown in events.
3) Finally, it is called when the work has been completed and the system call is finished, control is given back to the user mode as shown in event 4.

2. Layered Architecture of operating system

The layered Architecture of operating system was developed in 60‘s in this approach; the operating system is broken up into number of layers. The bottom layer (layer 0) is the hardware layer and the highest layer (layer n) is the user interface layer as shown in the figure.

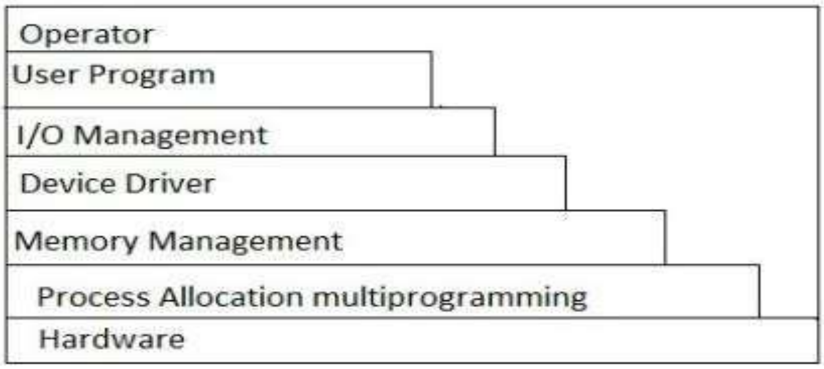


fig:- layered Architecture

The layered are selected such that each user functions and services of only lower level layer. The first layer can be debugged with out any concern for the rest of the system. It user basic hardware to implement this function once the first layer is debugged., it‘s correct functioning can be assumed while the second layer is debugged & soon . If an error is found during the debugged of particular layer, the layer must be on that layer, because the layer below it already debugged. Because of this design of the system is simplified when operating system is broken up into layer.

Os/2 operating system is example of layered architecture of operating system another example is earlier version of Windows NT.

The main disadvantage of this architecture is that it requires an appropriate definition of the various layers & a careful planning of the proper placement of the layer.

The system had 6 layers and the layers are as follows:

1) Layer 0 deals with allocation of the processor, switching between processes when interrupts occurred or timers expired. Above layer 0, the system consisted of sequential processes, each of which could be programmed without having to worry about the fact that multiple processes were running on a single processor i.e. the Layer 0 provided the basic multiprogramming of the CPU.

2) Layer 1 deal with the memory management. It allocated space for process in main memory and on a 512K word drum used for holding parts of processes (pages) for which there was no room in main memory. Above layer 1, processes did not have to worry about whether they were in memory or on the drum; Layer 1 software took care of making sure pages were brought into memory whenever they were needed.

Layer	Function
5	The Operator
4	User Programs
3	Input / Output management
2	Operator – process communication (Device driver)
1	Memory and drum management
0	Processor allocation and multiprogramming

3) Layer 2 handled communication between each process and the operator console. Above this layer each process effectively had its own operator console.
4) Layer 3 took care of managing I/O devices and buffering the information streams to and from them. Above layer 3, each process could deal with abstract I/O devices with nice properties, instead of real devices with many peculiarities.
5) Layer 4 was where the user programs were found. They did not have to worry about process, memory, console or I/O management.
6) The system operator process was located in layer 5.

3. **Virtual memory architecture of operating system**

Virtual machine is an illusion of a real machine. It is created by a real machine operating system, which make a single real machine appears to be several real machine. The architecture of virtual machine is shown above.

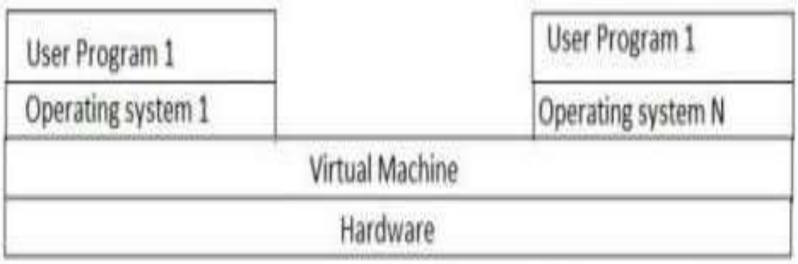


fig:- virtual memory architecture of os

The best example of virtual machine architecture is IBM 370 computer. In this system each user can choose a different operating system. Actually, virtual machine can run several operating systems at once, each of them on its virtual machine. Its multiprogramming shares the resource of a single machine in different manner.

The concepts of virtual machine are:-

a) Control program (cp):- cp creates the environment in which virtual machine can execute. It gives to each user facilities of real machine such as processor, storage I/0 devices.
b) Conversation monitor system (cons, CMS):- cons is a system application having features of developing program. It contains editor, language translator, and various application packages.
c) Remote spooling communication system (RSCS):- provide virtual machine with the ability to transmit and receive file in distributed system.
d) IPCS (interactive problem control system):- it is used to fix the virtual machine software problems

The idea of a virtual machine is used now days in a different context: Running old MS-DOS programs on a Pentium. When designing the Pentium and its software, both Intel and Microsoft realized that there would be a big demand for running old software on new hardware. For this reason, Intel provided a virtual 8086 mode on the Pentium. In this mode, the machine acts like an 8086, including 16-bit addressing with a 1 MB limit.
This mode is used by windows, and other OS for running old MS-DOS programs.

4. **Client/server architecture of operating system**

A trend in modern operating system is to move maximum code into the higher level and remove as much as possible from operating system, minimizing the work of the kernel. The basic approach is to implement most of the operating system functions in user processes to request a service, such as request to read a particular file, user send a request to the server process, server checks the parameter and finds whether it is valid or not, after that server does the work and send back the answer to client server model works on request- response technique i.e. Client always send request to the side in order to perform the task, and on the other side, server gates complementing that request send back response. The figure below shows client server architecture.

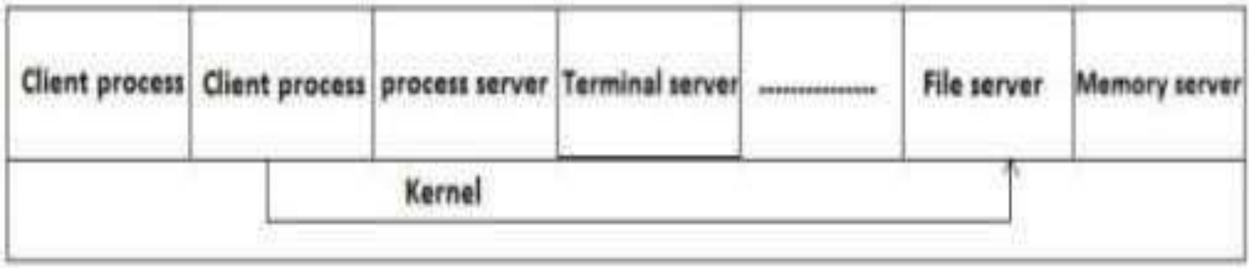


fig:- The client- server model

In this model, the main task of the kernel is to handle all the communication between the client and the server by splitting the operating system into number of ports, each of which only handle some specific task i.e. file server, process server, terminal server and memory service. Another advantage of the client-server model is it‘s adaptability to user in distributed system. If the client communicates with the server by sending it the message, the client need not know whether it was send a ……. Is the network to a server on a remote machine? As in case of client, same thing happen and occurs in client side that is a request was send and a reply come back.

1.3 Types and evolution of operating system

Operating systems have been evolving through the years. The history of OS is closely tied to the architecture of the computers on which they run.

Serial Processing (First Generation, 1945-55)

In earlier computer system, from the late 1940s to the mid 1950s, the programmer interacted directly with the computer hardware, because there was no operating system. A single group of people designed, built, programmed, operated and maintained each machine. These machines were run from a console consisting of display lights, toggle switches, some form of input device and a printer. Programs in machine code were loaded with the input device (e.g. card reader). If an error halted the program, the error condition was indicated by the lights. The programmer could proceed to examine register and main memory to determine the cause of the error. If the program proceeded to a normal completion, the output appeared on the printer.
Those earlier system has two major problems:
Scheduling: Most installation used a sign-up sheet to reserve machine time. Typically, a user could sign up for a multiple of half an hour. Let us consider, if a user might sign up for an hour ad finish in 45 minutes, this will result in wasted computer idle time. On the other hand if the job doesn‘t finish in the allotted time, and be forced to stop before resolving the problem.

Setup time: a single program could involve loading the compiler and the high level language program (source program) into memory, saving the compiled program (object program), and then loading and linking together the object program and common function. Each of these steps could involve mounting or dismounting tapes or setting up card decks. If an error occurred the hapless user typically had to go back to the beginning of the setup sequence that means considerable amount of time was spent, just in setting up the program to run.

Early machine were very expensive and therefore it was important to maximize machine use. The wasted time caused by scheduling and setup time was unacceptable.

Simple batch system (Second Generation, 1955-65)

To overcome the problems of serial processing batch operating system was developed. The first batch operating system was developed in the mid 1950s by General Motors for use on an IBM701.

In such system, a user doesn‘t have direct access to the machine. The user submits the job on cards or tape to a computer operator, who batches the jobs together sequentially and places the entire batch on an input device for the use by the monitor (a piece of software that controls sequence of events). Each program is constructed to branch back to the monitor when it completes processing, at which point the monitor automatically begins loading the next program.

In such system the processor is often idle. The problem is that I/O devices are slow compared to the processor.

Multiprogramming (Third Generation, 1965 – 1980)

Multiprogramming allows the processor to handle multiple batch jobs at a time, multiprogramming can be used to handle multiple interactive jobs.

Time Sharing

The basic technique for a time sharing system is to have multiple users simultaneously using the system through terminals, with the OS interleaving the execution of each user program in a short burst, or quantum of computation.

The first time sharing operating system to be developed was Compatible Time sharing System (CTSS), developed at MIT by a MAC (Machine Aided Cognition, Multiple Access Computers) project. It was developed for the IBM 70 in 1961 and later transferred to an IBM 7094.

Type of Operating System Services

1) **Program Execution**

The purpose of computer system is to allow the users to execute programs in an efficient manner. The operating system provides an environment where the user can conveniently run these programs. The user does not have to worry about the memory allocation or de-allocation or any other thing because these things are taken care of by the operating system.

To run a program, the program is required to be loaded into the RAM first and then to assign CPU time for its execution. Operating system performs this function for the convenience of the user. It also performs other important tasks like allocation and de-allocation of memory, CPU scheduling etc.

2) **I/O Operations**

Each program requires an input and after processing the input submitted by user it produces output. This involves the use of I/O devices. The operating system hides the user from all these details of underlying hardware for the I/O. So the operating system makes the users convenient to run programs by providing I/O functions. The I/O service cannot be provided by user-level programs and it must be provided by the operating system.

3) **File System Manipulation**

While working on the computer, generally a user is required to manipulate various types of files like as opening a file, saving a file and deleting a file from the storage disk. This is an important task that is also performed by the operating system.

Thus operating system makes it easier for the user programs to accomplish their task by providing the file system manipulation service. This service is performed by the 'Secondary Storage Management' a part of the operating system.

4) **Communication**

Operating system performs the communication among various types of processes in the form of shared memory. In multitasking environment, the processes need to communicate with each other and to exchange their information. These processes are created under a hierarchical structure where the main process is known as parent process and the sub processes are known as child processes.

5) **Error Detection**

Operating system also deals with hardware problems. To avoid hardware problems the operating system constantly monitors the system for detecting the errors and fixing these errors (if found). The main function of operating system is to detect the errors like bad sectors on hard disk, memory overflow and errors related to I/O devices. After detecting the errors, operating system takes an appropriate action for consistent computing.

This service of error detection and error correction cannot be handled by user programs because it involves monitoring the entire computing process. These tasks are too critical to be handed over to the user programs. A user program, if given these privileges; can interfere with the corresponding operation of the operating systems.

6) **Resource allocation**

In the multitasking environment, when multiple jobs are running at a time, it is the responsibility of an operating system to allocate the required resources (like as CPU, main memory, tape drive or secondary storage etc.) to each process for its better utilization. For this purpose various types of algorithms are implemented such as process scheduling, CPU scheduling, disk scheduling etc.

7) **Accounting**

Operating system keeps an account of all the resources accessed by each process or user. In multitasking, accounting enhances the system performance with the allocation of resources to each process ensuring the satisfaction to each process.

8) **Protection System**

If a computer system has multiple users and allows the concurrent execution of multiple processes, then the various processes must be protected from one another's activities.

Operating System Services as Resource Manager

Operating system works as a resource manager to manage the resources efficiently in a computer such as processor, memory, input/ output devices etc. To decide about which resources are used by which running programs and how to administer them, is known as the resource management. Operating system assigns the computer resources to processes for an efficient use; therefore, it plays an important role as a resource manager while ensuring the user satisfaction.

To manage the computer resources most effectively the OS decides which program should run at what time, how much memory should be allocated for an execution of a program, where to save the file so that disk space can be optimally utilized etc. Below given are some of the important services performed by operating system as a resource manager:

1. **Process Management**

In multiprocessing environment, operating system allows more than one application (or process) to run simultaneously. Process management is a part of an operating system which manages the processes in such a way that system performance can be enhanced. The operating system deals with other types of activities also that includes user programs and system programs like as printer spooling virtual memory, swapping etc.

A process is an activity that needs certain resources to complete its task. Various computer resources are CPU time, main memory, and I/O devices. These resources are allocated to the processes and based on decision that which process should be assigned for the allocation of resource and this decision is taken by process management implementing the process scheduling algorithm.

It is important to note that a process is not a program. A process is only ONE instant of a program in execution. There are many processes running the same program.

The five major activities of an operating system in regard to process management are:

- Creation and deletion of user and system processes.
- Suspension and re-activation of processes.
- A mechanism for process synchronization.
- A mechanism for process communication.
- A mechanism for deadlock handling.

2. Main-Memory Management

Memory management is the most important part of an operating system that deals directly with both the primary (known as main memory) memory and secondary memory. The main memory is a large array of bytes and each byte has its own address. Main memory provides the storage for a program that can be accessed directly by the CPU for its exertion. So for a program to be executed, the primary task of memory management is to load the program into main memory.

Memory management performs mainly two functions, these are:

- Each process must have enough memory in which it has to execute.
- The different locations of memory in the system must be used properly so that each and every process can run most effectively.

Operating system loads the instructions into main memory then picks up these instructions and makes a queue to get CPU time for its execution. The memory manager tracks the available memory locations which one is available, which is to be allocated or de-allocated. It also takes decision regarding which pages are required to swap between the main memory and secondary memory. This activity is referred as virtual memory management that increases the amount of memory available for each process.

The major activities of an operating system in regard to memory-management are:

- Keep track of which part of memory are currently being used and by whom.
- Decide which processes should be loaded into memory when the memory space is free.
- Allocate and de-allocate memory spaces as and when required.

3. File Management

A file is a collection of related information defined by its creator. Computer can store files on the disk (secondary storage), which provide long term storage. Some examples of storage media are magnetic tape, magnetic disk and optical disk. Each of these media has its own properties like speed, capacity, and data transfer rate and access methods.

A file system is normally organized into directories to make ease of their use. These directories may contain files and other directories. Every file system is made up of similar directories and subdirectories. Microsoft separates its directories with a back slash and its file names aren't case sensitive whereas Unix-derived operating systems (including Linux) use the forward slash and their file names generally are case sensitive.

The main activities of an operating system in regard to file management are:

- creation and deletion of files/ folders
- support of manipulating files/ folders,

- mapping of files onto secondary storage and
- Taking back up of files.

4. **I/O device Management**

Input/ Output device, management is a part of an operating system that provides an environment for the better interaction between system and the I/O devices (such as printers, scanners tape drives etc.). To interact with I/O devices in an effective manner, the operating system uses some special programs known as device driver. The device drivers take the data that operating system has defined as a file and then translate them into streams of bits or a series of laser pulses (in regard with laser printer).

A device driver is a specific type of computer software that is developed to allow interaction with hardware devices. Typically this constitutes an interface for, communicating with the I/O device, through the specific computer bus or communication subsystem that the hardware is connected with. The device driver is a specialized hardware dependent computer program that enables another program, typically an operating system to interact transparently with a hardware device, and usually provides the required interrupt handling necessary for the time dependent hardware interfacing.

5. **Secondary-Storage Management**

A computer system has several levels of storage such as primary storage, secondary storage and cache storage. But primary storage and cache storage can not be used as a permanent storage because these are volatile memories and its data are lost when power is turned off Moreover, the main memory is too small to accommodate all data and programs. So the computer system must provide secondary storage to backup the main memory. Secondary storage consists of tapes drives, disk drives, and other media.

The secondary storage management provides an easy access to the file and folders placed on secondary storage using several disk scheduling algorithms.

The four major activities of an operating system in regard to secondary storage management are:

- Managing the free space available on the secondary-storage device .
- Allocation of storage space when new files have to be written .
- Scheduling the requests for memory access.
- Creation and deletion of files.

6. **Network Management**

An operating system works as a network resource manager when multiple computers are in a network or in a distributed architecture. A distributed system is a collection of processors that do not share memory, peripheral devices, or a clock. The processors communicate with one another through communication lines called network, the communication-network design must consider routing and network strategies, and the problems with network and security.

Most of today's networks are based on client-server configuration. A client is a program running on the local machine requesting to a server for the service, whereas a server is a program running on the remote machine providing service to the clients by responding their request.

7. **Protection (User Authentication)**

Protection (or security) is the most demanding feature of an operating system. Protection is an ability to authenticate the users for an illegal access of data as well as system.

Operating system provides various services for data and system security by the means of passwords, file permissions and data encryption. Generally computers are connected through a network or Internet link, allowing the users for sharing their files accessing web sites and transferring their files over the network. For these situations a high level security is expected.

At the operating system level there are various software firewalls. A firewall is configured to allow or deny traffic to a service running on top of the operating system. Therefore by installing the firewall one can work with running the services, such as telnet or ftp, and not to worry about Internet threats because the firewall would deny all traffic trying to connect to the service on that port.

If a computer system has multiple users and allows the concurrent execution of multiple processes, then the various processes must be protected from one another's activities. Protection refers to mechanism for controlling the access of programs, processes, or users to the resources defined by a computer system.

8. **Command Interpreter System**

A command interpreter is an interface of the operating system with the user. The user gives commands which are executed by operating system (usually by turning them into system calls). The main function of a command interpreter is to get and execute the user specified command.

Command-Interpreter is usually not a part of the kernel, since multiple command interpreters may be supported by an operating system, and they do not really need to run in kernel mode. There are two main advantages of separating the command interpreter from the kernel.

If you want to change the way the command interpreter looks, i.e., you want to change the interface of command interpreter, then you can do that if the command interpreter is separate from the kernel. But if it is not then you cannot change the code of the kernel and will not be able to modify the interface.

If the command interpreter is a part of the kernel; it is possible for an unauthenticated process to gain access to certain part of the kernel. So it is advantageous to have the command interpreter separate from kernel.

Process and Threads

Process: Introduction

A process is defined as an entity which represents the basic unit of work to be implemented in the system i.e. a process is a program in execution. The execution of a process must progress in a sequential fashion. In general, a process will need certain resources such as the CPU time, memory, files, I/O devices and etc. to accomplish its task. As a process executes, it changes state. The state of a process is defined by its current activity. Each process may be in one of the following states: New state, ready state, waiting state, running state, and finished (Exit) state.

Five state Model

1. **New:** A process that has just been created but has not yet been admitted to the pool of executable processes by the OS.
2. **Ready:** Process that is prepared to execute when given the opportunity. That is, they are not waiting on anything except the CPU availability.
3. **Running:** the process that is currently being executed.
4. **Blocked:** A process that cannot execute until some event occurs, such as the completion of an I/O operation.
5. **Exit:** a process that has been released from the pool of executable processes by the OS, either because it halted or because it aborted for some reason A process that has been released by OS either after normal termination or after abnormal termination (error).

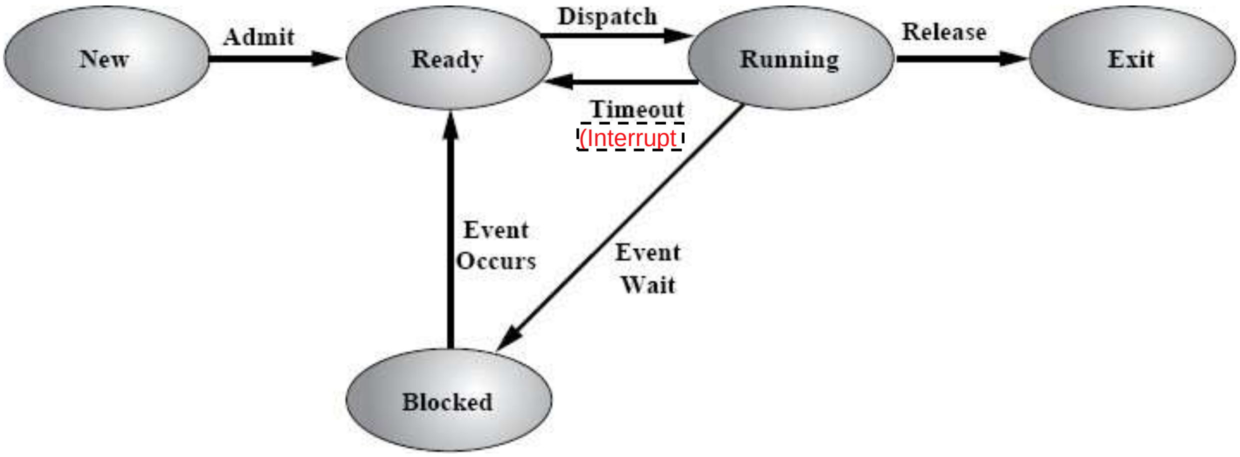


Fig: 5 state process model

Transition between the states:

1. The New state corresponds to a process that has just been defined or newly born process. For example, if a new user attempts to log on to a time sharing system or if a new batch job is submitted for execution.

2. **New to Ready:** process has been loaded into main memory and is awaiting execution on a CPU. There may be many "ready" processes at any one point of the system's execution—for example, in a one-processor system, only one process can be executing at any one time, and all other "concurrently executing" processes will be waiting for execution.

3. **Ready to Running:** The OS selects one of the processes in the ready state for execution.

4. **Running to Exit:** The currently running process is terminated by the OS if the process indicates that it is completed or if it aborts i.e. the process is terminated when it reaches a natural completion point. At this point process is no longer eligible for execution and all of the memory and resources associated with it are de-allocated so they can be used by other processes.

5. **Running to Waiting:** A process may be blocked due to various reasons such as when a particular process has tired the CPU time allocated to it or it is waiting for an event to occur. A process is put in blocked or waiting state if it requests something for which it must wait. For example, a process may request a service from the OS and the OS is not prepared to perform immediately. It can request a resource such as file or a shared section of virtual memory that is not immediately available.

6. **Waiting to Ready:** A process in the blocked state is moved to the ready state when the event for which it has been waiting occurs.

The state of process from ready to running, running to waiting and waiting to ready will be continued till the process terminates. This process is shown in above figure.

Process Control Block (PCB)

A process control block or PCB is a data structure (a table) that holds information about a process. Every process or program that runs needs a PCB. When a user requests to run a particular program, the operating system constructs a process control block for that program. It is also known as Task Control Block (TCB). It contains many pieces of information associated with a specific process i.e. it simply serves as the respiratory for any information that may vary from process to process. The process control block typically contains:

- An ID number that identifies the process
- Pointers to the locations in the program and its data where processing last occurred
- Register contents
- States of various flags and switches
- Pointers to the upper and lower bounds of the memory required for the process
- A list of files opened by the process
- The priority of the process
- The status of all I/O devices needed by the process

Process state
Process ID number
Program Counter
CPU registers
Memory limits
List of open files
………

Fig: Process Control Block

The information stored in the Process Control Block in given below

- **Process State:** The state may be new, ready, running, and waiting, halted, and so on.
- **Program Counter:** the counter indicates the address of the next instruction to be executed for this process.
- **CPU register:** The registers vary in number and type, depending on the computer architecture. They include accumulator, index registers, stack pointers, and general purpose registers. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.
- **CPU Scheduling information:** This information includes a process priority, pointers to scheduling queues, and other scheduling parameters.
- **Memory management information:** this information include the value of the base and limit registers, the page table, or the segment tables, depending on the memory system used by the OS.
- **Accounting information:** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers and so on.
- **I/O status information:** This information includes the list of I/O devices allocated to the process, a list of open files and so on.

Role of PCB:

The PCB is most important and central data structure in an OS. Each PCB contains all the information about a process that is needed by the OS. The blocks are read and/or modified by virtually every module in the OS, including those involved with scheduling, resource allocation, interrupt processing and performance monitoring and analysis that mean PCB defines the state of OS.

The PCB contains the information about the process. It is the central store of information that allows the operating system to locate all the key information about a process. When CPU switches from one process to another, the operating system uses the Process Control Block (PCB) to save the state of process and uses this information when control returns back process is terminated, the Process Control Block (PCB) released from the memory.

A context switch is the computing process of storing and restoring the state (context) of a CPU such that multiple processes can share a single CPU resource. The context switch is an essential feature of a multitasking operating system. Context switches are usually computationally intensive and much of the design of operating systems is to optimize the use of context switches. There are three scenarios where a context switch needs to occur: multitasking, interrupt handling, user and kernel mode switching. In a context switch, the state of the first process must be saved somehow, so that, when the scheduler gets back to the execution of the first process, it can restore this state and continue. The state of the process includes all the registers that the process may be using, especially the program counter, plus any other operating system specific data that may be necessary. Often, all the data that is necessary for state is stored in one data structure, called a process control block.

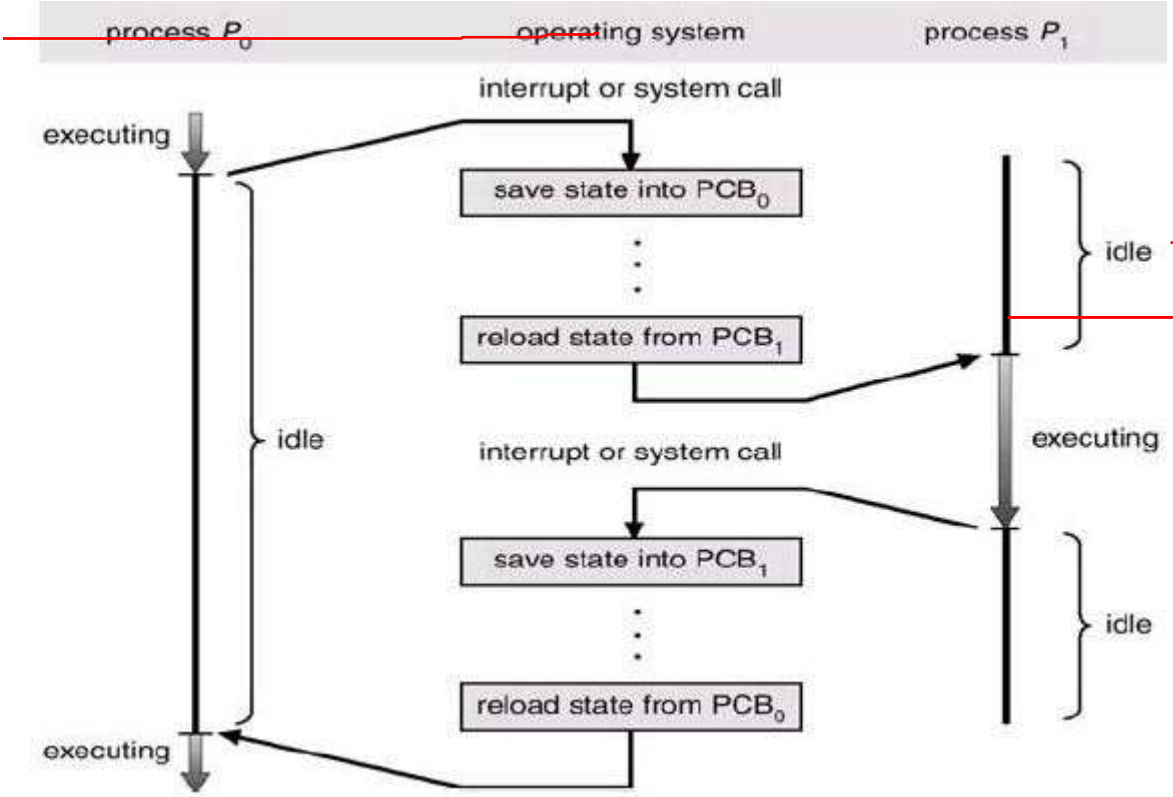


Fig: Showing CPU switches from process to process (Role of PCB)

Interprocess Communication (IPC)

Processes executing concurrently (parallel) in the OS may be either independent process or cooperating process. A process is **independent** if it cannot or be affected by other processes executing in the system. Any process that does not share data with any other process is independent. A process is **cooperating** if it can affect or be affected by other processes executing in the system. Any process that shares data with other process is a cooperating process.

Advantages of process cooperation:

- Information sharing- such as shared files.
- Computation speed-up – to run a task faster, we must break it into subtasks, each of which will be executing in parallel. This speed up can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).
- Modularity – construct a system in a modular function (i.e., dividing the system functions into separate processes).
- Convenience – one user may have many tasks to work on at one time. For example, a user may be editing, printing, and compiling in parallel.

Inter-Process Communication (IPC) is a set of techniques for the exchange and synchronization of data among two or more threads in one or more processes. Interprocess communication is useful for creating *cooperating* processes. Processes may be running on one or more computers connected by a network. IPC techniques are divided into methods for message passing, synchronization, shared memory, and remote procedure calls (RPC). The method of IPC used may vary based on the bandwidth and latency of communication between the threads, and the type of data being communicated.

The common Linux shells all allow redirection. For example

```
$ ls | pr | lpr
```

It pipes the output from the **ls** command listing the directory's files into the standard input of the **pr** command which paginates them. Finally the standard output from the pr command is piped into the standard input of the **lpr** command which prints the results on the default printer. Pipes then are unidirectional byte streams which connect the standard output from one process into the standard input of another process. Neither process is aware of this redirection and behaves just as it would normally. It is the shell which sets up these temporary pipes between the processes.

There are two fundamental mechanisms for interprocess communication.

Message Passing

In the message passing model, communication takes place by means of messages exchanged between the cooperating processes that mean it provides mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. It is particularly useful in distributed environment, where communicating processes may reside on different computers connected by a network. For example, a chat program used on World Wide Web could be designed so that chat participants communicate with one other by exchanging messages.

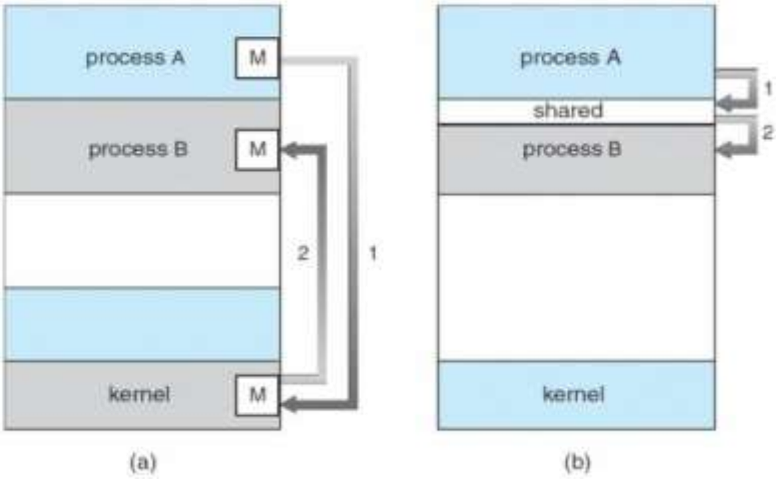
A message passing facility provides at least two operations: **send (message)** and **receive (message)**. Messages send by a process can be of either fixed size or variable size. If process A and B wants to communicate, they must send messages to and receive messages from each other and a communication link must exist between them. The sender typically uses send () system call to send messages, and the receiver uses receive () system call to receive messages. The communication link can be established in varieties of ways and some methods for logically implementing a link and the send () / receive operations are:

- Direct or indirect communication
- Synchronous or asynchronous communication
- Automatic or explicit buffering.

Message passing is useful for exchanging smaller amount of data, because no conflicts needed be avoided. It is easier to implement than shared memory for intercomputer communication.

Shared Memory

Interprocess communication using shared memory requires communicating processes to establish a region of shared memory. Shared memory is a memory that may be simultaneously accessed by multiple programs with intent to provide communication among them. Typically, a shared memory region resides in the address space of the process creating the shared memory segment. Other processes that wish to communicate using shared memory segment must attach it to their address space. Now they can exchange information by reading and writing data in the shared areas. Since both processes can access the shared memory area like regular working memory, this is a very fast way of communication. On the other hand, it is less powerful, as for example the communicating processes must be running on the same machine, and care must be taken to avoid issues if processes sharing memory are running on separate CPUs and the underlying architecture is not cache coherent. Since shared memory is inherently non-blocking, it can‘t be used to achieve synchronization.



It allows maximum speed and convenience of communication. It is faster than message passing, because the message passing system are typically implemented using system calls and thus requires the more time consuming task of kernel intervention but in shared memory systems, system calls are only required only to establish shared memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and assistance of kernel is not required.

Concurrent Processes:

Concurrent processes come into conflict with each other when they are competing for the use of the same resource such as I/O devices, memory, processor time, and the clock. With competing processes, there is no exchange of information between the competing processes. However the execution of one process may affect the behavior of those competing processes. In particular, if two processes both wish to a single resource, then one process will be allocated that resource by the operating system and the other will have to wait. In extreme case, the blocked process may never get access to the resource and hence will never successfully terminate. In competing processes, three control problems must be faced and they are:

- Mutual exclusion
- Deadlock
- Starvation.

Critical Regions and Race Conditions: From

a) Mutual exclusion

Mutual exclusion (Mutex) is the concept of restricting access to shared resource when corrupt it. It is the OS's responsibility to make sure that this does not happen. There are many methods that can be used to implement mutual exclusion such as semaphore, monitor, etc. the mutual exclusion has following properties:

- Safety: No two processes must use the shared resource at the same time (should not be in the critical region at the same time) .
- Liveliness: There should not be deadlock and a process comes out of the critical section after some time.
- Fairness: A process wanting to use critical section must only wait some time

Mutual exclusion (often abbreviated to **mutex**) algorithms are used in concurrent programming to avoid the simultaneous use of a common resource, such as a global variable, by pieces of computer code called critical section. A critical section is a piece of code in which a process or thread accesses a common resource. The critical section by itself is not a mechanism or algorithm for mutual exclusion. A program, process, or thread can have the critical section in it without any mechanism or algorithm which implements mutual exclusion.

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

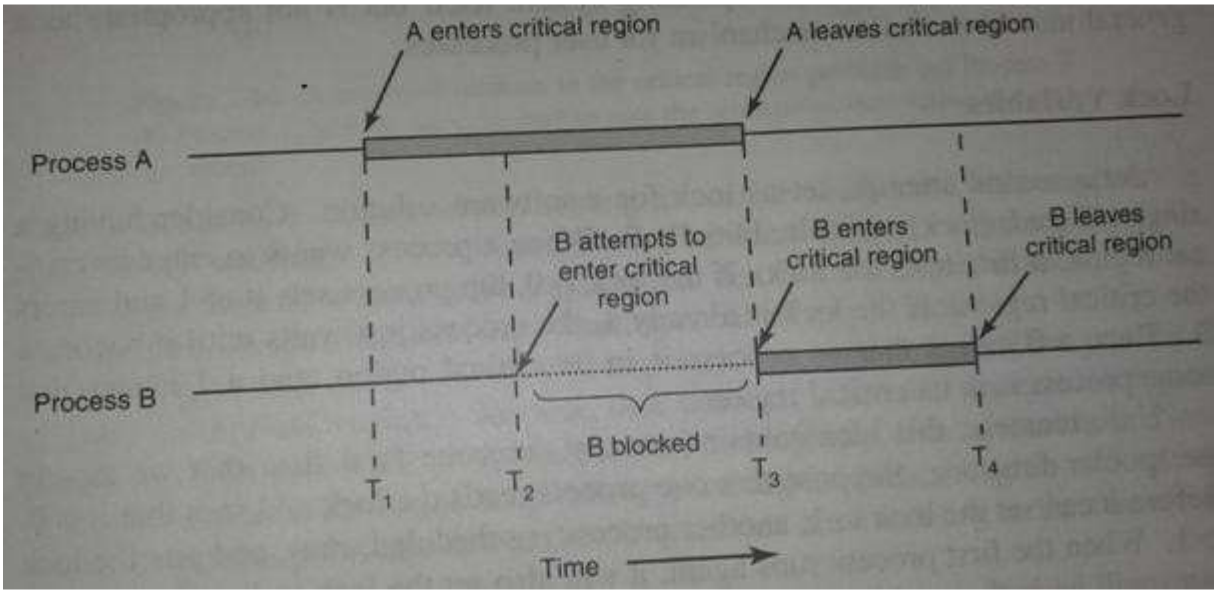


Fig: Mutual exclusion using critical region

Here process A enters its critical region at time T_1. A little later, at time T_2 process B attempts to enter it critical region but fails because another process is already in its critical region and only one process can enter at its critical region. So the process B is temporarily suspended until time T_3. When process A leaves its critical region, immediately process B enters in its critical region. Eventually B leaves at time T_4 and are back to the original situation with no processes in their critical regions.

There are various proposals for achieving mutual exclusion:

Disabling interrupts:

It is the simplest solution to disable all interrupts to each process just after entering its critical section and enable them just before leaving it. The CPU is only switched from process to process as a result of clock or other interrupts. If the interrupts are disabled clock interrupts can't occur and the CPU can't be switched to another process. So, once a process has disabled interrupts, it can examine and update the shared memory without fear that any other process will interfere.

However this approach is not attractive because if one of the process disabled the interrupt but it never turned on the interrupt, then that will be the end of the system.

Peterson's Algorithm

Peterson's solution is shown in following diagram.

```
#define FALSE 0
#define TRUE 1
#define N 2
int turn;
int interested[N];                /*All values initially 0 (FALSE)*/
void enter_region(int process)    /*Process is 0 or 1*/
{
int other;                        /*Number of other process*/
other = 1 - process;              /*the opposite of process*/
interested[process] = TRUE;       /*Show that you are interested*/
turn = process;                   /*Set Flag*/
while(turn== process && interested[other] == TRUE)       /*Null statement*/
}

void leave_region(int process)    /*Process who is leaving*/
{
intereste[process] = FALSE;       /*Indicate departure from critical region*/
}
```

Fig: Peterson's solution for achieving mutual exclusion

Before using the share variable (i.e. before entering its critical region), each process calls `enter_region` with its own process number 0 or 1, as a parameter. This call will cause it to wait, if need be until it is safe to enter. After it has finished with the shared variables, the process calls `leave_region()` to indicate that it is done and to allow the other process to enter.

Initially, neither process is in its critical region. Now process 0 calls `enter_region()`. It indicates its interest by setting its array element and sets turn to 0. Since process 1 is not interested, enter_region() return immediately. It process 1 now calls `enter_region`, it will hang there until interested[0] goes to FALSE, an event that only happens when process 0 calls `leave_reion` to exit the critical region.

When both process calls `enter_region()` almost simultaneously. Both will store their process in turn. Whichever store is done last is the one that counts; the first one is lost. Suppose that process 1 stores last, so turn is 1. When both processes come to the while statement, process 0 executes it zero times and enters its critical region. Process 1 loop and does not enter in its critical region.

TSL (Test and Set Lock)

TSL instruction is used for reading from a location or writing to a location in the memory and then saving a non-zero value at an address in memory. It is implemented in the hardware and is used in a system with multiple processors. When one processor is accessing the memory, no other processor can access the same memory location until TSL instruction is finished. Locking of the memory bus in this way is done in the hardware.

It reads the contents of memory word LOCK into register RX (`TSL RX, LOCK`) and then stores a non- zero value at the memory address LOCK. The operations of reading the word and storing into it are guaranteed to be invisible and no other processor can access the memory word until the instruction is finished. The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

To use the TSL instruction, a shared variable LOCK is used to access to shared memory. When LOCK is 0, any process may set it to 1 using the TSL instruction and then the read or write the shared memory. When it is done, the process sets LOCK back to 0 using an ordinary move instruction.

It is shown in following assembly language.

Enter_region:

```
   TSL REGISTER, LOCK         //Copy LOCK to register and set LOCK to 1
   CMP REGISTER, #0           // Was LOCK Zero?
   JNE ENTER_REGION           // If it was non zero, LOCK was set, so loop
   RET                        //Return to caller; entered to critical region
```

Leave_region:

```
   MOVE LOCK, #0              //Store a 0 in LOCK
   RET                        // Return to caller
```

The first instruction copies old value of LOCK to the register and then sets LOCK to 1. The old value is compared with 0. If it is non zero, the lock was already set, so the program just goes back to the beginning and tests it again. Sooner or later it will become 0 and the subroutine returns, with the lock set. The program just stores a 0 in LOCK.

Lock

It is a software solution. In this approach shared variables are locked (1) or unlocked (0). When a process wants to enter its critical section, it first checks the lock. If the lock is 0, the process sets it to 1 and enters in the critical region. If the lock is already 1, the process just waits until it becomes 0. Thus, a 0 means that no process is in its critical region, and a 1 means that some process is in its critical region.

This also has a problem. Suppose that one process reads the lock and sees that it is 0. Before it ca set lock to 1, another process is scheduled, runs and sets the lock to 1. When the first process runs again, it will also sets the lock to 1, and two process will enter in their critical regions at the same time. Now the race will occur, if the second process modifies the lock just after the first process has finished its second check.

b) Deadlock

Consider two processes, P1 and P2, and two critical resources, R1 and R2. Suppose that each process needs access to both resources to perform part of its function. Then it is possible to have the following situations: R1 is assigned by the OS to P2, and R2 is assigned to P1. Each process is waiting for one of the two resources. Neither will release the resources that is already owns until it has acquired the other resources and performed its critical section. In such case both processes are **deadlocked**.

c) Starvation.

Suppose that three processes, P1, P2, and P3, each requires periodic access to resource R. consider the situation in which P1 is in possession of the resource, and both P2 and P3 are delayed, waiting for that resource. When P1 exits its critical section, either P2 or P3 should be allowed access to R. assume that P3 is granted access and that before it completes its critical section; P1 again requires access. If P1 is granted access after P3 has finished, and if P1 and P3 repeatedly grant access to each other, then P2 may indefinitely be denied access to the resources, even though there is no deadlock situation. Such problem is known as **starvation.**

Critical Section (Critical Region):

Suppose two or more processes require access to a single non-sharable resource such as a printer. During the course of execution, each process will sending commands to the I/O device, receives status information, sending data, and / or receiving data. Such resource is called **critical resource** and the portion of the program that uses it is called **critical section** of the program. However only one program at a time be allowed in its critical section.

When two or more processes access shared data, often the data must be protected during access. Typically, a process that reads data from a shared queue cannot read it whilst the data is currently being written or its value being changed. Where a process is considered that it cannot be interrupted at the same time as performing a critical function such as updating data, it is prevented from being interrupted by the operating system till it has completed the update. During this time, the process is said to be in its **critical section** that means the part of the program where the shared memory is accessed is called **critical section** or **critical region**. Once the process has written the data, it can then be interrupted and other processes can also run.

Problems occur only when both tasks attempt to read and write the data at the same time. The answer is simple, lock the data structure whilst accessing (semaphores or interrupts disabled). There is no need for data locking if both processes only read at same time. Critical sections of a process should be small so that they do not take long to execute and thus other processes can run.

Semaphore:

A semaphore is hardware or a software tag variable whose value indicates the status of a common resource. Its purpose is to lock the resource being used. A process which needs the resource will check the semaphore for determining the status of the resource followed by the decision for proceeding. In multitasking operating systems, the activities are synchronized by using the semaphore techniques.
There are 2 types of semaphores:

- Binary semaphores
- Counting semaphores

Binary Semaphore:

- They are used to acquire locks.
- Binary semaphores have 2 methods associated with it and they are: Up and down / lock, unlock.
- Binary semaphores can take only 2 values (0/1).
- When a resource is available, the process in charge set the semaphore to 1 else 0.

Counting Semaphore:

- Typically, it is used to allocate resources from a pool of identical resources.
- Counting Semaphore may have value to be greater than one.

Synchronization of Semaphore:
Sometimes a process may need to wait for some other process to finish before it can continue. In this instance, the two processes need to be synchronized together. There are a number of ways in which this can be done. A common method in operating systems is to use a variable called a **semaphore** that only one process can own at a time. There are two calls associated with a semaphore, one to lock it and one to unlock it. When a process attempts to lock a semaphore, it will be successful if the semaphore is free. If the semaphore is already locked, the process requesting the lock will be blocked and remain blocked till the process that has the semaphore unlocks it. When that happens, the process that was blocked will be unblocked and the semaphore can then be locked by it.

System semaphores are used by the operating system to control system resources. A program can be assigned a resource by getting a semaphore (via a system call to the operating system). When the resource is no longer needed, the semaphore is returned to the operating system, which can then allocate it to another program.

Solving the producer-consumer problem using semaphore:

```
#define N 100 //Number of slots in the buffer
Typedef int semaphore
Semaphore mutex = 1; //controls access to critical region
Semaphore empty = N; // Counts empty buffer slots
Semaphore full = 0;       // Counts full buffer slots
```

```
void producer (void)
{
int item;
while (TRUE)                          // TRUE is the constant
    {
    item = produce_item();            // generates something to put in buffer
    down(&empty);                     // decrement empty count
    down(&mutex);                     // Enter critical region
    insert_item(item);                // put new item in buffer
    up(&mutex);                       // Leave the critical region
    up(&full);                        // Increment count of full slots
    }
}

void consumer (void)
{
int item;
while (TRUE)                            // TRUE is the constant
   {
   down(&empty);                        // decrement full count
   down(&mutex);                        // Enter critical region
   item = remove_item(item);            // take item from buffer
   up(&mutex);                          // Leave the critical region
   up(&empty);                          // Increment count of empty slots
   consume_item(item);                  // do something with the item
   }
}
```

Fig: The producer-consumer problem using semaphore

This solution uses three semaphores: *full, empty* and mutex. The full semaphore is used for counting the number of slots that are full, empty for counting the number of empty slots that are empty and mutex to make sure the producer and consumer do not access the buffer at the same time. full is initially 0, empty is initially equal to the number of slots in the buffer, and mutex is initial 1. Semaphore that are initialized to 1, and are used by two or more processes to ensure that only one of them can enter its critical region at the same time. If each process does a down just before entering its critical region and up just after leaving it, mutual exclusion is guaranteed.

Monitor

A monitor is a collection of procedures, variables and data structure that are all grouped together in a special kind of module or package. Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures form procedures declare outside the monitor.

```
Monitor example
integer I;
condition c;
procedure producer (x);
.
.
.
end;
procedure consumer (x);
.
.
.
end;
end monitor;
```

Fig: a monitor

Only one process can be active in a monitor at any instant. When a process calls a monitor procedure, the first few instructions of the procedure will check to see if any other process is currently active within the monitor. If so, the calling process will be suspended until the other process has left the monitor. If no other process is using the monitor, the calling process may enter.

Race conditions

In some OS, processes that are working together may share some common storage that each can read and write. The shared storage may be in main memory (kernel) or it may be a shared file. When a process wants to print a file, it enters the file name in a special spooler directory. Another process, the printer daemon, periodically checks to see if so are any files to be printed, and if so removes their name from the directory.

A race condition occurs when two processes (or threads) access the same variable/resource without doing any synchronization that means when several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which access takes place is called a **race condition**.

- One process is doing a coordinated update of several variables

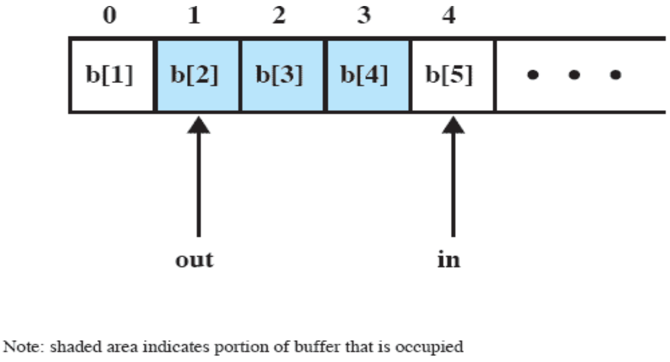
- The second process observing one or more of those variables will see inconsistent results
- Final outcome dependent on the precise timing of two processes

Example

- One process is changing the balance in a bank account while another is simultaneously observing the account balance and the last activity date
- Now, consider the scenario where the process changing the balance gets interrupted after updating the last activity date but before updating the balance
- If the other process reads the data at this point, it does not get accurate information (either in the current or past time)

Producer/Consumer Problem

- One or more producers are generating data and placing these in a buffer
- One or more consumers are taking items out of the buffer one at time
- Only one producer or consumer may access the buffer at any one time
- Producer can‘t add data into full buffer and consumer can‘t remove data from empty buffer



Note: shaded area indicates portion of buffer that is occupied

Figure 5.8 Infinite Buffer for the Producer/Consumer Problem

- producer:

```
while (true)
   {
   /* produce item v */
   b[in] = v;
   in++;
   }
```

- consumer:

```
while (true)
   {
   while (in <= out)
      /*do  nothing */;
   w = b[out];
   out++;
   /* consume item w */
   }
```

Classical IPC problem

The Dining Philosophers' Problem

It is a classical synchronization problem.

The problem can be stated as: five philosophers are seated around a circular table. Each philosopher has a plate of spaghetti. The spaghetti is so slipper that a philosopher needs two forks to eat it. Between each pair of plates is one fork. The layout of the table is shown in following diagram.

Each philosopher does two things: think and eat. The philosopher thinks for a while, and then stops thinking and becomes hungry. When the philosopher becomes hungry, he/she cannot eat until he/she owns the forks to his/her left and right. When the philosopher is done eating he/she puts down the fork and begins thinking again.

The challenge in the dining philosophers problem is to design a protocol so that the philosophers do not deadlock (i.e. every philosopher has a fork), and so that no philosopher starves (i.e. when a philosopher is hungry, he/she eventually gets the forks). Additionally, we should try to minimize the time that philosophers spent waiting to eat.

Solutions:

- When a philosopher wants to eat, he/she checks both forks. If they are free, then he/she eats. Otherwise, he/she waits on a condition variable. Whenever a philosopher finishes eating, he/she checks to see if his neighbors want to eat and are waiting. If so, then he/she calls signal on their condition variables so that they can recheck the chopsticks and eat if possible.
  A problem with this solution is starvation. After a few seconds, philosophers 0 and 2 get to eat, then 1 and 3, and then 0 and 2 again and so on. Philosopher 4 never gets to eat, because there is never a time when 0 and 3 are both not eating.

- We must guarantee that no philosopher may starve. For example, suppose you maintain a queue of philosophers. When a philosopher is hungry, he/she gets put onto the tail of the queue. A philosopher may eat only if he/she is at the head of the queue, and if the chopsticks are free.

The Readers ad Writers Problem

```
semaphore mutex = 1;                  // Controls access to the reader count
semaphore db = 1;                     // Controls access to the database
int reader_count;                     // The number of reading processes accessing the data

Reader()
{
  while (TRUE) {                      // loop forever
     down(&mutex);                           // gain access to reader_count
     reader_count = reader_count + 1;        // increment the reader_count
     if (reader_count == 1)
         down(&db);                          // if this is the first process to read the
database,
                                             // a down on db is executed to prevent access to the
                                             // database by a writing process
     up(&mutex);                             // allow other processes to access reader_count
     read_db();                              // read the database
     down(&mutex);                           // gain access to reader_count
     reader_count = reader_count - 1;        // decrement reader_count
     if (reader_count == 0)
         up(&db);                            // if there are no more processes reading from the
                                             // database, allow writing process to access the
data
     up(&mutex);                             // allow other processes to access
reader_countuse_data();
                                             // use the data read from the database (non-
critical)
}

Writer()
{
  while (TRUE) {                      // loop forever
     create_data();                          // create data to enter into database (non-critical)
     down(&db);                              // gain access to the database
     write_db();                             // write information to the database
     up(&db);                                // release exclusive access to the database
}
```

A data object is to be shared among several concurrent processes. Some of these processes may want to only to read the content of the shared object, whereas others may want to update the shared object. We distinguish between these two types of processes by referring to those processes that are interested in only reading as **readers**, and to the rest as **writers**. Obviously, if two readers access the shared data object simultaneously, no adverse effects will result.

The readers-writers problem has several variations, all involving priorities. The simplest one, referred to as the **first** readers-writers problem, requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. The **second** readers-writers problem requires that, once a writer is ready, that writer performs its write as soon as possible. Here, I adopt the first readers-writers problem. At a given time, there is only one writer and any number of readers. When a writer is writing, the readers can not enter into the database. The readers need to wait until the writer finishes writing on the database. Once a reader succeeds in reading the database, subsequent readers can enter into the critical section (in this case, say, and database) without waiting for the precedent reader finish to read. On the other hand, a writer who arrives later than the reader who is reading currently is required to wait the last reader finish to read. Only when the last reader finishes reading, the writer can enter into the critical section and is able to write on the database.

The Sleeping Barber Problem

There is one barber in the barber shop, one barber chair and n chairs for waiting customers. If there are no customers, the barber sits down in the barber chair and takes a nap. An arriving customer must wake the barber. Subsequent arriving customers take a waiting chair if any are empty or leave if all chairs are full. This problem addresses race conditions.

This solution uses three semaphores, one for customers (counts waiting customers), one for the barber (idle - 0 or busy - 1) and a mutual exclusion semaphore, mutex. When the barber arrives for work, the barber procedure is executed blocking the barber on the customer semaphore until a customer arrives. When a customer arrives, the customer procedure is executed which begins by acquiring mutex to enter a critical region. Subsequent arriving customers have to wait until the first customer has released mutex. After acquiring mutex, a customer checks to see if the number of waiting customers is less than the number of chairs. If not, mutex is released and the customer leaves without a haircut. If there is an available chair, the waiting counter is incremented, the barber is awaken, the customer releases mutex, the barber grabs mutex, and begins the haircut. Once the customer's hair is cut, the customer leaves. The barber then checks to see if there is another customer. If not, the barber takes a nap.

The two principal elements of the problem seem to me to be

- the synchronization of the customers with barbers - i.e. customers get their hair cut at approximately the same time that barber cuts their hair, and
- The restriction of the number of customers connected to the shop. A solution to the problem should not make any assumptions about the timing of actions; arbitrary delays may occur at any point in any process.

Number of chairs in the waiting room is known in advance and never changes. Waiting queue can not overflow because any customer that sees no free chairs turns around and leaves barbershop. So we can represent waiting queue as circular array of fixed size. Two indices are used to represent it. Head points to next request to be serviced if not null. Tail points to next free slot where request can be put.

Barber waits next in line (the one head index points to) non null request to get serviced. To mark slot as free for itself it nullifies it once obtained reference to request. Next head index is advanced to make this slot available for use by customers. Once it completed with execution it notifies waiting customer that it is free to go by changing done flag.

Customer first checks if there are free slots. Then it competes with other customers for a free slot (the one tail index points to). If successful it puts request that combines action and done flag into waiting queue array with the index value of just advanced tail. Once successfully queued request customer waits until value in the done flag is not changed.

```
#define CHAIRS 5
typedef int semaphore;
semaphore customers = 0;
semaphore barbers=0;
semaphore mutex=1;
int waiting=0;

void barber(void)
{
while (TRUE)
   {
   P(customers);          /*go to sleep if no customers*/
   P(mutex);
   waiting=waiting-1;
   V(barbers);
   V(mutex);
   cut_hair();
   }
}
```

```
void customer(void)
{
P(mutex);
if (waiting lessthan CHAIRS)
{
  waiting=waiting+1;
V(customers);
  V(mutex);
  P(barbers);
get_haircut();
}
else
{
V(mutex);
}
}
```

The barber shop has a barber, a barber chair, and n chairs for waiting customers. When a barber finishes cutting a customer's hair, the barber fetches another customer from the waiting room if there is a customer, or sits down in the barber chair and falls asleep if the waiting room is empty. When a customer arrives, he/she wakes up the barber, if the barber is sleeping. If a customer arrives while the barber is cutting a customer's hair, he/she sits down if there is an empty chair for waiting customers. If there are no empty chairs, the customer must give up and leave the shop.

Producer consumer problem:

- It is also known as bounded buffer problem. It is a multi process synchronization problem.
- The problem describes two processes, the producer and consumer, who share common fixed size buffer.
- Producer: the producer's job is to generate a piece of data, put into the buffer and start again.
- Consumer: the consumer is consuming the data (i.e. removing it from the buffer) one piece at a time

Problem:

- The problem is to make sure that the producer won't try to add data into the buffer if it is full and that the customer won't try to remove data from an empty buffer.

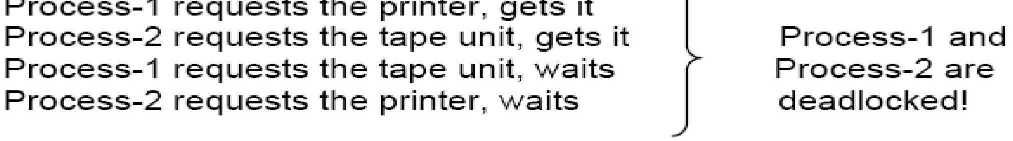
Solution:

- Producer either go to sleep or discard data if the buffer is full
- The next time the Consumer removes the item from the buffer and it notifies the producer who starts to fill the buffer again.
- Consumer can go to sleep if it finds the buffer to be empty.
- The next time the producer puts data into the buffer , it weeks up the sleeping customer

Deadlock: Introduction:

In a multiprogramming environment, several processes may compete for finite number of resources. A process request resources; if the resources are not available at that time, the process enters in a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called **deadlock** i.e. a deadlock is a situation in which two or more transactions are waiting for one another to give up locks. A set of processes is in a deadlock state if every process in the set is waiting for an event (release) that can only be caused by some other process in the same set.

Example:



Process-1 requests the printer, gets it
Process-2 requests the tape unit, gets it
Process-1 requests the tape unit, waits
Process-2 requests the printer, waits

Process-1 and
Process-2 are
deadlocked!

Necessary Conditions:

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

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

1. **Mutual exclusion**
   At least one resource must be held in a non-sharable mode i.e. only one process at a time can use the resource. If any other process requests this resource, then that process must wait for the resource to be released.

2. **Hold and wait**
   A process must be holding at least one resource and waiting for at least one resource that is currently being held by some other process.

3. **No preemption**
   Resources cannot be preempted i.e. once a process is holding a resource (i.e. once its request has been granted), then that resource cannot be taken away from that process until the process voluntarily releases it.

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

Resource Allocation Graph

Deadlock can also be described in terms of a directed graph called resource allocation graph. This graph consists of a set of vertices V and set of edges E. The set of vertices V is portioned into two different types of nodes:

- The set of all actives process P = {P1, P2, P3, … , Pn} in the system and
- The set consisting of all resource types in the system.

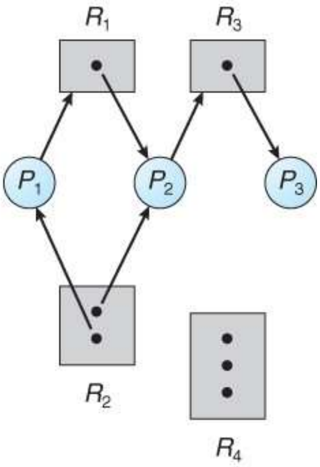


Fig: Resource allocation Graph

Request Edges - A set of directed arcs from Pi to Rj, indicating that process Pi has requested Rj, and is currently waiting for that resource to become available.

Assignment Edges - A set of directed arcs from Rj to Pi indicating that resource Rj has been allocated to process Pi, and that Pi is currently holding resource Rj.

In above diagram, there are:
The sets P, R, and E:

- P = {P1, P2, P3}
- R = {R1, R2, R3, R4}
- E = { P1 → R1, P2 → R3, R1 → P2, R3 → P3, R2 → P1, R2 → P2}

Resources Instances:

- One instance of resource R1 and R3
- Two instance of resource R2
- Three instances of resource R4

Process states:

- Process P1 is holding an instance of resources type R2 and is waiting for an instance of resource type R1.
- Process P2 is holding an instance of resources type R1 and an instance of R2 is waiting for an instance of resource type R3.
- Process P3 is holding an instance of resources type R3

If a resource-allocation graph contains no cycles, then the system is not deadlocked. If the graph does not contain a cycle, then a deadlock may exist.

If each process type has exactly one instance, then a cycle implies that a deadlock has occurred.
If the cycle involves only a set of resources types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.
In following diagram, there are two minimal cycles

P1 → R1 → P2 → R3 → P3 → R2 → P1
P2 → R3 → P3 → R2 → P2

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

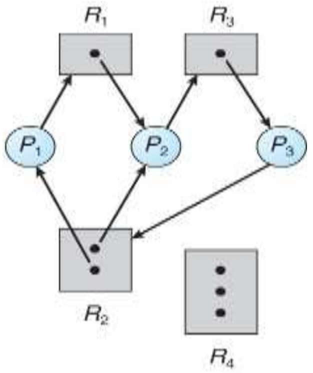


Fig: Resource allocation graph with deadlock

If each resource type has several instances, then a cycle does not necessarily implies that a deadlock has occurred. In this case, a cycle graph is necessary but not a sufficient condition existence of deadlock.

Let us consider following graph. It has: P1 → R1 → P3 → R2 → P1
There is no deadlock. The process P4 may release its instances of resource type R2 and that resource can be allocated to P3, breaking the cycle.

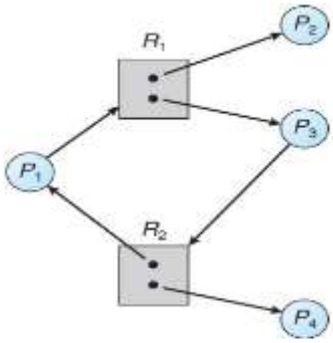


Fig: Resource allocation graph with a cycle but no deadlock

Note:

- If a Resource allocation graph does not have a cycle, the system is not in a deadlock state.
- If there is a cycle, the system may or not be in a deadlock state.

Methods for Handling Deadlocks

There are three ways of handling deadlocks:

- Deadlock prevention
- Deadlock avoidance
- Deadlock detection and recovery.

1. **Deadlock prevention**

Different protocols are used to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state. To prevent the system from deadlocks, one of the four conditions that may create a deadlock should be discarded. The methods for those conditions are as follows:

a) **Mutual Exclusion:**

The mutual exclusion condition must hold for non shareable Sources. For example, a printer cannot be simultaneously shared by several processes. Read only files are good example of shareable resources. If several process attempts to open a read only file at the same time, they can simultaneously access to the file. A process never needs to wait or a shareable resource. In general, we do not have systems with all resources being sharable. Some resources like printers, processing units are non-sharable. So it is not possible to prevent deadlocks by denying mutual exclusion.

b) **Hold and Wait:**

- One protocol to ensure that hold-and-wait condition never occurs says each process must request and get all of its resources before it begins execution.
- Another protocol is ―Each process can request resources only when it does not occupy any resources.‖

Example:

The difference between two protocols:

- Let us consider a process that copies data from DVD drive to a file on disk, sorts the file and then prints the result to a printer. If all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file and printer. It will hold the printer for its entire execution, even though it needs the printer at the end.
- The second method allows the process to request initially only the DVD drive and disk file. It copies from DVD drive to disk and then releases both the DVD drive and the disk file. The process must then again request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminate.

Both these protocols have two main disadvantages:

- Resource utilization may be low, since resources may be allocated but unused for a long period.
- Starvation is possible. A process that needs several popular resources my may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

c) **No Preemption:**

There should be no preemption of resources that have already been allocated. To ensure that this condition does not hold we can use following protocols:

- One protocol is ―If a process that is holding some resources and requests another resource and that resource cannot be immediately allocated to it (i.e. the process must wait), then all resources the process is currently holding are preempted i.e. it must release all resources that are currently allocated to it.‖ The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it regain its old resources, as well as the new ones that is requesting.
- Another protocol is ‒When a process requests some resources, if they are available, allocate them. If a resource it requested is not available, then we check whether it is being used or it is allocated to some other process that is waiting for other additional resources. If that resource is not being used, then the OS preempts it from the waiting process and allocate it to the requesting process. If that resource is used, the requesting process must wait.‖

  This protocol can be applied to resources whose states can easily be saved and restored such as CPU registers, memory space). It cannot be applied to resources like printers.

d) **Circular Wait:**

One protocol to ensure that the circular wait condition never holds is ―Impose a linear ordering of all resource types.‖ Then, each process can only request resources in an increasing order of priority.

For example, set priorities for r1 = 1, r2 = 2, r3 = 3, and r4 = 4. With these priorities, if process P wants to use r1 and r3, it should first request r1, then r3.

Another protocol is ―Whenever a process requests a resource rj, it must have released all resources rk with priority (rk) ≥ priority (rj).

For example:
Let us consider R = {R1, R2, R3, … , Rn} be the set of resource types. For example, if the set of resources types R includes: tape drives, disk drives, and printer, then the function F might me defined as follows:

F (tape drive) = 1
F (disk drive) = 5
F (printer) = 12

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

2. Deadlock avoidance

Deadlock can be avoided if certain information about processes is available to the operating system before allocation of resources, such as which resources a process will consume in its lifetime. For every resource request, the system sees whether granting the request will mean that the system will enter an *unsafe* state, meaning a state that could result in deadlock. The system then only grants requests that will lead to *safe* states. In order for the system to be able to determine whether the next state will be safe or unsafe, it must know in advance at any time:

- resources currently available
- resources currently allocated to each process
- resources that will be required and released by these processes in the *future*

Deadlock avoidance algorithm can be classified into following two methods:

a) Safe state
b) Resource allocation graph algorithm

Safe state:

A state is safe if the system can allocate resources to each process in some order and still avoid a deadlock.
Let us consider a system with 12 magnetic tape drives and 3 processes P_0, P_1, and P_2. The process P_0 requires 10 tape drives; P_1 may need 4 tape drives and process P_2 needs up to 9 tape drives. Suppose at time T_0, process P_0 is holding 5 tape drives, process P_1 is holding 2 tape drives and process P_2 is holding 2 tape drives. It is shown in following table

	Maximum Needs	Current needs
P0	10	5
P1	4	2
P2	9	2

At the time T_0, the system is in safe state. The sequence < P_1, P_0, P_2> satisfies the safety condition. Process P_1 can immediately be allocated all its tape drives and then return them (the system will have 5 available tape drives), then process P_0 can get all its tape drives and return them (the system will have 10 available tape drives) and finally process P_2 can get all tape drives and return them (the system will then have all 12 tape drives available).

A system can go from a safe state to unsafe state. Suppose that, at time T_1, process P_2 requests and is allocated one more tape drive. The system is no longer in safe state. At this state only process P_1 can be allocated all its tape drives. When it returns them, the system will have only four available tape drives. Since process P_0 is allocated 5 tape drives but has a maximum of 10, it may request 5 more tape drives. If it does so, it will have to wait, because they are unavailable. Similarly, process P_2 may request 6 additional tape drives and have to wait, resulting in deadlock.

The mistake was in granting the request from process P_2 for 1 more tape drive. If the process P2 wait until either of the other processes had finished and released its resources, then we could have avoided the deadlock.

Banker's Algorithm:

The resource allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type. Banker's algorithm is applicable for such system but is less efficient than the resource-allocation graph. This algorithm requires prior information about the maximum number of each resource class that each process may request. Using this information we can define a deadlock avoidance algorithm.

When a new process enters the system, it must declare the maximum number of instances of each resources type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Several data structures must be maintained to implement the banker's algorithm. Some of them are, where ***n*** is the number of processes in the system and ***m*** is the number of resources types:

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

Note: Need [i][j] = ~~matri~~max [i][j] – Allocation [i][j]

Safety Algorithm:
This algorithm can be described as follows:

1. Let Work and Finish be vectors of length m and n respectively. Intitizlize Work = Available and Finish[i] = false for i = 0, 1, 2, … , n – 1
2. Find an index i such that both
   a. Finish[i] = = false
   b. $Need_i \leq Work$
   If no such i exists, go to step 4.
3. Work = Work + Allocation
   Finish[i] = true
   Go to step 2
4. If Finish[i] == true for all i, then the system is in safe state.

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

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise and error condition, since the process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise, Pi must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:

   Available = Available – Request
   Allocation = Allocation + Request;
   $Need_i$ = $Need_i$ – $Request_i$;

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

Example:
Let us consider a system with 5 processes P_0 to P_4 and three resource types A, B, and C. Resource type A has 10 instances, resource type B has 5 instances, and resource type C has 7 instances. Suppose that, at time T_0, the following snapshots of the system has been taken:

	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			

Then content of the matrix Need is defined to be Max – Allocation and is as follows:

	Allocation		
	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

The system is currently in safe state. The sequence <P_1, P_3, P_4, P_2, P_0> satisfies the safety criteria.

Suppose now that process P_1 requests 1 additional instance of resource type A and 2 instances of resource type C, so *$Request_1$ = (1, 0, 2)*. To decide whether this request can be immediately granted, we first check that *$Request_i$* ≤ *Available* that is, that (1, 0, 2) (3, 3, 2), which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:

	Allocation			Max			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 execute safety algorithm and find that the sequence < P_1, P_3, P_4, P_0, P_2> satisfies the safety requirement. Hence, we can immediately grant the request of process P_1.

5. **Deadlock detection and recovery.**
   If a system does not employ a protocol to ensure that deadlocks will never occur, then a deadlock-detection and recovery scheme is employed.
   Detection:
   A deadlock detection algorithm must be invoked to determine whether a deadlock has occurred i.e.it determines the existence of the deadlock.
   Single instance of each resource type:
   If all resources ahs only a single instance, then we can define a deadlock-detection algorithm by a ***wait-for graph***. This graph can be obtained by resource allocation graph by removing the resource nodes and collapsing the appropriate edges.

More precisely, an edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs. An edge $P_i \rightarrow P_j$ exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ for some resource R_q.
It is shown in following diagram

Fig: (a) Resource allocation graph (b) corresponding wait-for graph

Several instance of a resource type:
The wait-for graph is not applicable for a resource-allocation system with multiple instances of each resource type. For this banker's algorithm is applicable.

Let us consider a system with 5 processes P_0 to P_4 and three resource types A, B, and C. Resource type A has 7 instances, resource type B has 2 instances, and resource type C has 6 instances. Suppose that, at time T_0, we have following resource allocation state:

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

Now suppose that process P_2 makes 1 additional request for an instance of type C. the *Request* matrix is modified as follows:

	Request		
	A	B	C
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

Now the system is deadlocked. Although we reclaim the resources held by process P_0, the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

Detection-algorithm usage:
If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until deadlock can be broken. In addition, the number of processes involved I the deadlock cycle may grow.
Deadlocks occur only when some process makes a request that cannot be granted immediately. Then we can invoke the deadlock detection algorithm every time a request for allocation cannot be granted immediately. In this case, we can identify not only the deadlock set of processes but also the specific process that "caused" the deadlock.

Recovery:

Detection algorithm determines the existence of the deadlock. There are two options for breaking a deadlock: simply abort one or more processes to break the circular wait and to preempt some resources from one or more of the deadlocked processes. It is described in following section:

A. Process termination

Following methods are used to eliminate deadlocks by aborting a process.

Abort all deadlocked process: This method will break the deadlock cycle. . Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state. Similarly, if the process was in the midst of printing data on a printer, the system must reset the printer to a correct state before printing the next job.

Abort one process at a time until the deadlock cycle is eliminated: This method incurs considerable overhead, since after each process is aborted a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked. Many factors may affect which process is chosen, such as:

- What is the priority of the process?
- How long the process has computed and how much longer the process will compute before completing its designated task.
- How many and what types of resources the process has used
- How many more resources the process needs in order to complete
- How many processes will need to be terminated?
- Whether the process is interactive or batch.

B. Resource preemption

In this method, some resources are preempted from other processes and give these resources to other processes until the deadlock cycle is broken. For preemption following issues are needed to be addressed:

- **Selecting a victim:** for process termination, we must determine the order of preemption to minimize cost. Cost factors includes different parameters such as the number of resources a deadlock process is holding, amount of time the process has consumed during its execution.

- **Roolback:** Rollback means to abort the process and the restart it. In general, it is difficult to determine what a safe state is; the simple solution is a total rollback. This method requires the system to keep more information about the state of all running processes.

- **Starvation:** Generally cost factor is a base for selecting victim process but it may happen that the same process is always picked as a victim and as a result, this process never completes its designated task that means starvation. We have to ensure that a process can be picked as a victim for only a finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

Thread

Introduction

A thread is a separate part of a process i.e. a thread is the smallest unit of processing that can be performed in an OS. A process can consist of several threads, each of which execute separately. It is a flow of control within a process. It is also known as ***light weight process***. For example, one thread could handle screen refresh and drawing, another thread printing, another thread the mouse and keyboard. This gives good response times for complex programs. Windows NT is an example of an operating system which supports multi-threading.

A thread is a basic unit of CPU utili~~zation; it comprises~~ a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to same process its code section, data section, and other operating resources, such as open files and signals. A traditional (or heavy weight) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.

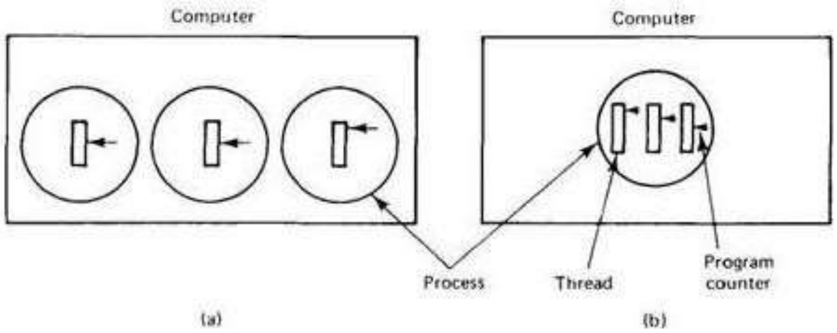


Fig: (a) Three processes with one thread each (b) A process with 3 threads

Multithreading:

Multithreading is the ability of a program or an operating system process to manage its use by more than one user at a time and to even manage multiple requests by the same user without having to have multiple copies of the programming running in the computer

A multithread process contains several different flow of control within the same address space. When multiple threads are running concurrently then this is known as ***multithreading***, which is similar to multitasking. Multithreading allows sub-processes to run concurrently.

Basically, an operating system with multitasking capabilities will allow programs (or processes) to run seemingly at the same time. On the other hand, a single program with multithreading capabilities will allow individual sub-processes (or threads) to run seemingly at the same time.

One example of multithreading is when we are downloading a video while playing it at the same time. Multithreading is also used extensively in computer-generated animations.

Thread Usage

Threads were invented to allow parallelism to be combined with sequential execution and blocking system calls. Some examples of situations where we might use threads:

- Doing lengthy processing: When a windows application is calculating it cannot process any more messages. As a result, the display cannot be updated.
- Doing background processing: Some tasks may not be time critical, but need to execute continuously.
- Concurrent execution on multiprocessors
- Manage I/O more efficiently: some threads wait for I/O while others compute
- It is mostly used in large server applications

Advantages of Threads

Res

Advantage of multithread includes:

- **Resource sharing**

  Process can only share resources using shared memory or message passing method. Such techniques are explicitly arranged by the programmer. However, threads share the memory and resources of the process to which they are belong by default. It (Sharing of code and data) allows an application to have several different threads of activity within the same address space.

- **Responsiveness**

  It may allow a program to continue running even if part of it is blocked or is performing lengthy operations, which increases the responsiveness to the user.

- **Economy**

  Allocation of memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads. It is much more time consuming to create and manage processes than threads.

- **Scalability**

  In multiprocessor architecture threads may run in parallel on different processors. A single threaded process can only run on one processor. Multithreading on a multi-CPU machine increases parallelism.

User Space and Kernel Space Threads

User level threads are threads that are visible to the programmer and unknown to the kernel. Such threads are supported above the kernel and managed without kernel support.

The operating system directly supports and manages kernel level threads. In general, user level threads are faster to create and manage than are kernel threads, as no intervention from the kernel is required. Windows XP, Linux, Mac OS X supports kernel threads.

Difference between User Level & Kernel Level Thread

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

Multithreading Model

A relationship must exist between user threads and kernel threads. There are three models related to user and kernel threads and they are:

- **Many- to- one model:** it maps many user threads to single kernel threads.
- **One - to one model:** it mach each user thread to a corresponding thread
- **Many - to - many model:** It maps many user threads to a smaller or equal number of kernel threads.

1. Many to one model:

It maps many user threads to single kernel threads. Thread management is done by the thread library in user space, so it if efficient but the entire process will block if a thread makes a blocking system call. Because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.

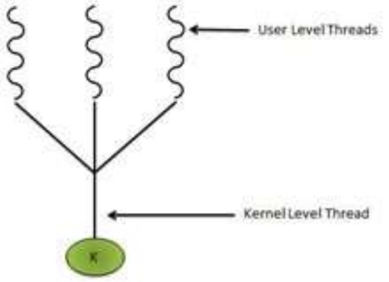


Fig: Many-to-one model

2. One to one model:

It mach each user thread to a corresponding kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call i.e. it also allows multiple threads to run in parallel on multiprocessor. The drawback is that creating a user thread requires creating the corresponding kernel thread and the overhead of creating the corresponding kernel thread can burden the performance of an application.

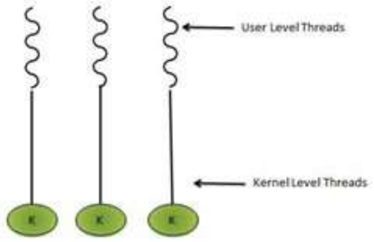


Fig: One-to-one model

3. Many- to - many model:

It multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine. This model allows the developer to create as many user threads as the user wishes and the corresponding kernel threads can run in parallel on a multiprocessor. In such case true concurrency is not gained because the kernel can schedule only one thread at a time.

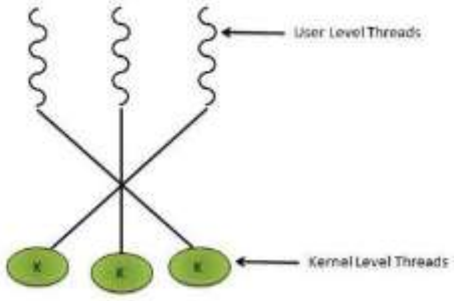


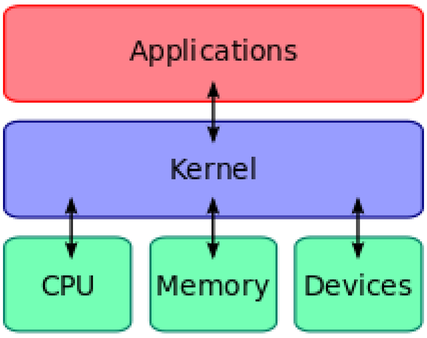
Fig: Many-to-many model

Difference between Process and Thread

S.N	Process	Thread
1	It is heavy weight	It is light weight, which takes lesser resources than a process
2	Process switching needs interaction with OS	Thread switching does not need interaction with OS
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, 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.

Unit 3
Kernel

Introduction and Architecture of a Kernel



It is the core of the operating system and has complete control over everything that occurs in the system. The kernel is a fundamental part of a modern computer's operating system. The kernel is the core of an operating system that manages input/output requests from software and translates them into data processing instructions for the central processing unit and other electronic components of a computer. Since there are many programs, and resources are limited, the kernel also decides when and how long a program should run. This is called scheduling

Accessing the hardware directly can be very complex, since there are many different hardware designs for the same type of component. Kernels usually implement some level of hardware abstraction to hide the underlying complexity from applications and provide a clean and uniform interface. This helps application programmers to develop programs without having to know how to program for specific devices. The kernel relies upon software drivers that translate the generic command into instructions specific to that device.

The kernel's primary function is to manage the computer's hardware and resources and allow other programs to run and use these resources.

It is the software responsible for running programs and providing secure access to the machine's hardware. Since there are many programs, and resources are limited, the kernel also decides when and how long a program should run. This is called scheduling. Accessing the hardware directly can be very complex.

Typically, the resources consist of:

- The **Central Processing Unit:** This is the most central part of a computer system, responsible for *running* or *executing* programs. The kernel takes responsibility for deciding at any time which of the many running programs should be allocated to the processor or processors (each of which can usually run only one program at a time)
- The **computer's memory:** Memory is used to store both program instructions and data. Typically, both need to be present in memory in order for a program to execute. Often multiple programs will want access to memory, frequently demanding more memory than the computer has available. The kernel is responsible for deciding which memory each process can use, and determining what to do when not enough is available.
- **Any Input / Output (I/O):** devices present in the computer, such as keyboard, mouse, disk drives, USB devices, printers, displays, network adapters, etc. The kernel allocates requests from applications to perform I/O to an appropriate device (or subsection of a device, in the case of files on a disk or windows on a display) and provides convenient methods for using the device (typically abstracted to the point where the application does not need to know implementation details of the device).

A kernel is a central component of an operating system. It acts as an interface between the user applications and the hardware. The sole aim of the kernel is to manage the communication between the software (user level applications) and the hardware (CPU, disk memory etc) i.e. the kernel's primary function is to manage the computer's hardware and resources and allow other programs to run and use these resources. Kernels also usually provide methods for synchronization and communication between processes called inter-process communication (IPC). The main tasks of the kernel are:

- Process management
- Device management
- Memory management
- Interrupt handling

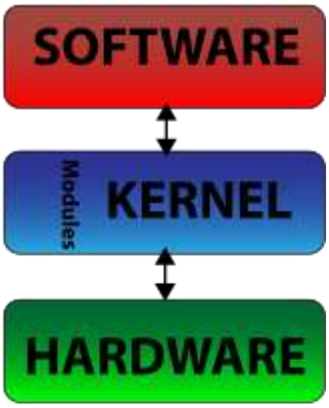
- I/O communication
- File system...etc

Types of Kernels

Kernels may be classified mainly in two categories

- Monolithic
- Micro Kernel

1. **Monolithic Kernel**



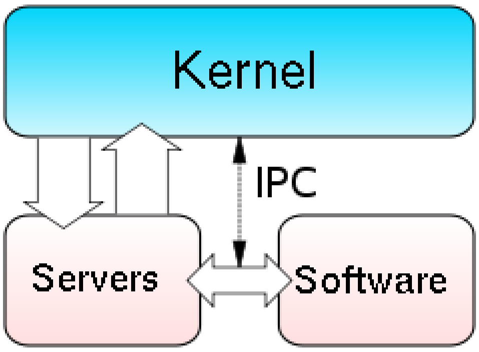
It provides rich and powerful abstractions of the underlying hardware. A Monolithic kernel executes all the operating system instructions in the same address space to improve the performance of the system. In this type of kernel architecture, all the basic system services like process and memory management, interrupt handling etc were packaged into a single module in kernel space. This type of architecture led to some serious drawbacks like

- Size of kernel, which was huge.
- Kernels often become very large and difficult to maintain.
- Monolithic kernels are not portable
- Since the modules run in the same address space, a bug can bring down the entire system.

Monolithic kernels contain all the operating system core functions and the device drivers (such as disk drives, video cards and printers). A monolithic kernel is one single program that contains all of the code necessary to perform every kernel related task. Every part which is to be accessed by most programs which cannot be put in a library is in the kernel space: Device drivers, Scheduler, Memory handling, File systems, Network stacks. Many system calls are provided to applications, to allow them to access all those services.

Linux follows the monolithic modular approach

2. **Micro Kernel**



A micro kernel runs most of the operating system's background process in user space to make the operating system more modular and, therefore, easier to maintain. It provides a small set of simple hardware abstractions and use applications called servers to provide more functionality. This architecture allows some basic services like device driver management, protocol stack, file system etc to run in user space. This reduces the kernel code size and also increases the security and stability of OS as we have the bare minimum code running in kernel. So, if suppose a basic service like network service crashes due to buffer overflow, then only the networking service's memory would be corrupted, leaving the rest of the system still functional.

The microkernel approach consists of defining a simple abstraction over the hardware, with a set of primitives or system calls to implement minimal OS services such as memory management, multitasking, and inter-process communication. Other services, including those normally provided by the kernel, such as networking, are implemented in user-space programs, referred to as *servers*. Microkernels are easier to maintain than monolithic kernels, but the large number of system calls and context switches might slow down the system because they typically generate more overhead than plain function calls.

Context Switching (Kernel Mode and User Mode)

A **context switch** is the computing process of storing and restoring state (context) of a CPU so that execution can be resumed from the same point at a later time. This enables multiple processes to share a single CPU. The context switch is an essential feature of a multitasking operating system. A multitasking operating system is one in which multiple processes execute on a single CPU seemingly simultaneously and without interfering with each other. This illusion of *concurrency* is achieved by means of context switches that are occurring in rapid succession (tens or hundreds of times per second). These context switches occur as a result of processes voluntarily relinquishing their time in the CPU or as a result of the *scheduler* making the switch when a process has used up its CPU *time slice*. Switching from one process to another requires a certain amount of time for doing the administration - saving and loading registers and memory maps, updating various tables and list etc. It is also known as *process switch* or a *task switch.*

Context switching can be described in the kernel performing the following activities with regard to processes (including threads) on the CPU:

1. Suspending the progression of one process and storing the CPU's *state* (i.e., the context) for that process somewhere in memory,
2. Retrieving the context of the next process from memory and restoring it in the CPU's registers and
3. Returning to the location indicated by the program counter in order to resume the process.

There are three situations where a context switch needs to occur. They are:

- Multitasking
- Interrupt handling
- User and kernel level switching

1. Multitasking

Most commonly, within some scheduling scheme, one process needs to be switched out of the CPU so another process can run. Within a preemptive multitasking operating system, the scheduler allows every task to run for some certain amount of time, called its time slice.

If a process does not voluntarily give up the CPU (for example, by performing an I/O operation), a timer interrupt fires, and the operating system schedules another process for execution instead. This ensures that the CPU cannot be monopolized by any one processor-intensive application.

2. Interrupt handling

Modern architectures are interrupt-driven. This means that if the CPU requests data from a disk, for example, it does not need to busy-wait until the read is over, it can issue the request and continue with some other execution; when the read is over, the CPU can be *interrupted* and presented with the read. For interrupts, a program called an interrupt handler is installed, and it is the interrupt handler that handles the interrupt from the disk.

The kernel services the interrupts in the context of the interrupted process even though it may not have caused the interrupt. The interrupted process may have been executing in user mode or in kernel mode. The kernel saves enough information so that it can later resume execution of the interrupted process and services the interrupt in kernel mode. The kernel does not spawn or schedule a special process to handle interrupts.

3. User and kernel level switching

Kernel space is strictly reserved for running privileged kernel, kernel extensions, and most device drivers. In contrast, ***user space*** is the memory area where application software and some drivers execute. Context switches can occur only in kernel mode. Kernel mode is a privileged mode of the CPU in which only the kernel runs and which provides access to all memory locations and all other system resources. Other programs, including applications, initially operate in user mode, but they can run portions of the kernel code via system calls.

Kernel is the heart of OS which manages the core features of an OS while if some useful applications and utilities are added over the kernel, then the complete package becomes an OS. So, it can easily be said that an operating system consists of a kernel space and a user space. There are two execution modes for the CPU for the execution of task and they are:

- User mode and
- Kernel mode

User Mode:
It is a *non-privileged* mode for user programs in which each process starts out. It is non-privileged in that it is forbidden for processes in this mode to access those portions of memory (i.e., RAM) that have been allocated to the kernel or to other programs. The kernel is not a process, but rather a controller of processes, and it alone has access to all resources on the system.

When a *user mode process* (i.e., a process currently in user mode) wants to use a service that is provided by the kernel, it must switch temporarily into kernel mode, which has *root* (i.e., administrative) privileges, including *root access permissions* (i.e., permission to access any memory space or other resources on the system). When the kernel has satisfied the process's request, it restores the process to user mode. This change in mode is termed a ***mode switch***.

Kernel Mode:
Kernel mode is also referred to as *system mode*. When the CPU is in kernel mode, it is assumed to be executing *trusted* software, and thus it can execute any instructions and reference any memory addresses. Kernel is *trusted* software, but all other programs are considered *untrusted* software. Thus, all user mode software must request use of the kernel by means of a system call in order to perform privileged instructions, such as process creation or *input/output* operations.
A machine runs in kernel mode for:

- After machine boot
- Interrupt handler
- System call
- exception

* The Process Switch is always done in Kernel Mode

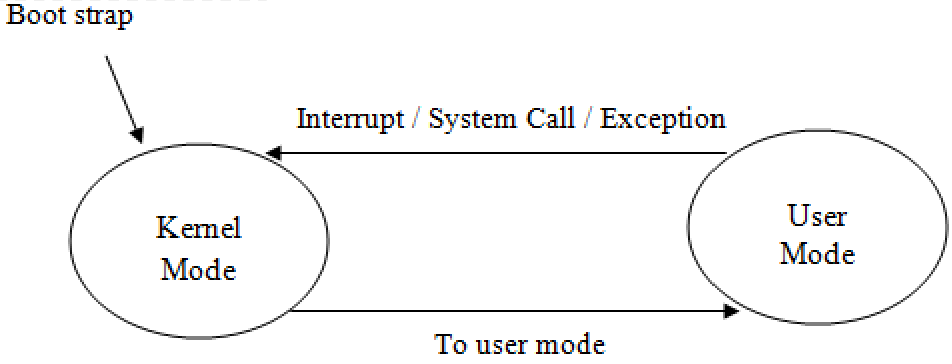


Fig: User Mode – Kernel Model

First Level Interrupt Handling (FLIH)

Interrupt handler is a section of a computer program or of the operating system that takes control when an interrupt is received and performs the operations required to service the interrupt. It is also known as Interrupt Service Routine (ISR). An interrupt handler is a low-level counterpart of event handlers. These handlers are initiated by either hardware interrupts or software interrupts, and is used for servicing hardware devices and transitions between protected modes of operation such as system calls.
Interrupt handlers are divided into two parts:

- First-Level Interrupt Handler (FLIH) and
- Second-Level Interrupt Handlers (SLIH)

First-Level Interrupt Handler (FLIH)

FLIH is also known as *hard interrupt handlers* or *fast interrupt handlers*. It is a software or hardware routine that is activated by interrupt signals sent by peripheral devices and decides, based on the relative importance of the interrupts, how they should be handled. The job of a FLIH is to quickly service the interrupt and schedule the execution of a SLIH for further long-lived interrupt handling.

It performs following task:

- save registers of current process in PCB
- Determine the source of interrupt
- Initiate service of interrupt

Second-Level Interrupt Handlers (SLIH)

SLIH is also known as *slow/soft interrupt handlers*. A SLIH completes long interrupt processing tasks similarly to a process. SLIH either have a dedicated kernel thread for each handler, or are executed by a pool of kernel worker threads. These threads sit on a run queue in the operating system until processor time is available for them to perform processing for the interrupt. SLIHs may have a long-lived execution time, and thus are typically scheduled similarly to threads and processes.

Kernel Implementation of Processes

A kernel process is a process that is created in the kernel protection domain and always executes in the kernel protection domain. Kernel processes can be used in subsystems, by complex device drivers, and by system calls. They can also be used by interrupt handlers to perform asynchronous processing not available in the interrupt environment. Kernel processes can also be used as device managers where asynchronous input/output (I/O) and device management is required.

Introduction to Kernel Processes

A kernel process (kproc) exists only in the kernel protection domain and differs from a user process in the following ways:

- It is created using the **creatp** and **initp** kernel services.
- It executes only within the kernel protection domain and has all security privileges.
- It can call a restricted set of system calls and all applicable kernel services.
- It has access to the global kernel address space (including the kernel pinned and pageable heaps), kernel code, and static data areas.
- It must poll for signals and can choose to ignore any signal delivered, including a **kill** signal.
- Its text and data areas come from the global kernel heap.
- It cannot use application libraries.
- It has a process-private region containing only the **u-block** (user block) structure and possibly the kernel stack.
- Its *parent process* is the process that issued the **creatp** kernel service to create the process.
- It can change its parent process to the **init** process and can use interrupt disable functions for serialization.
- It can use locking to serialize process-time access to critical data structures.
- It can only be a 32–bit process in the 32–bit kernel.
- It can only be a 64–bit process in the 64–bit kernel.

Sites used

http://www.linfo.org/context_switch.html

Introduction

Technical terms used in scheduling:

- **Ready queue:** The processes waiting to be assigned to a processor are put in a queue called *ready queue.*
- **Burst time:** The time for which a process holds the CPU is known as *burst time.*
- **Arrival time:** *Arrival Time* is the time at which a process arrives at the ready queue.
- **Turnaround time:** The interval from the time of submission of a process to the time of completion is the turnaround time.
- **Waiting time:** *Waiting time* is the amount of time a process has been waiting in the ready queue.
- **Response Time:** Time between submission of requests and first response to the request.
- **Throughput:** number of processes completed per unit time.
- **Dispatch latency** – It is the time it takes for the dispatcher to stop one process and start another running
- **Context switch:** A context switch is the computing process of storing and restoring the state (context) of a CPU so that execution can be resumed from the same point at a later time. This enables multiple processes to share a single CPU.

Optimal scheduling algorithm will have minimum waiting time, minimum turnaround time and minimum number of context switches.

Introduction:

In a single-processor system, only one process can run at a time and other must wait until the CPU is free and can be rescheduled. The objective of multiprogramming is to have some process running at all times to maximize CPU utilization. CPU Scheduling is the process used to maximize the CPU utilization. CPU scheduling is the basis of multi-programmed operating system. By switching the CPU among processes the OS can make the computer more productive.

Scheduling is the process of controlling and prioritizing processes to send to a processor for execution. The goal of scheduling is maintaining a constant amount of work for the processor, eliminating high and lows in the workload and making sure each process is completed within a reasonable time frame. It is an internal operating system program, called the scheduler performs this task.
CPU scheduling is done in the following four circumstances:

- When a process switches from the running state to the waiting state
- When a process switches from the running state to the ready state (For example, when an interrupt occurs)
- When process switches from the waiting state to the ready state (For example, at completion of I/O)
- When a process terminates.

Scheduling Objectives and Criteria

Objectives:

1. **Efficient utilization of resources**. In general, we want to avoid having a resource sit idle when there is some process/thread that could use it.
   Example: Suppose that the system contains a mix of CPU bound and IO bound processes (or threads). In this case, if the CPU bound processes/threads are allowed to ―hog‖ the CPU, then utilization of the IO devices will be low, since the primary processes that use them will have to spend a lot of time waiting for the CPU in order to be able to generate work for the IO devices to do.
2. **Maximizing throughput**. The throughput of a system is the number of processes (or threads) that actually complete in a period of time.
3. Minimizing average **turnaround time.**
4. **Minimizing response time.** The response time is the time between when an interactive users submits a request and when the system begins to respond to the request. (Response time is only an issue when the user is interactive).

Criteria:

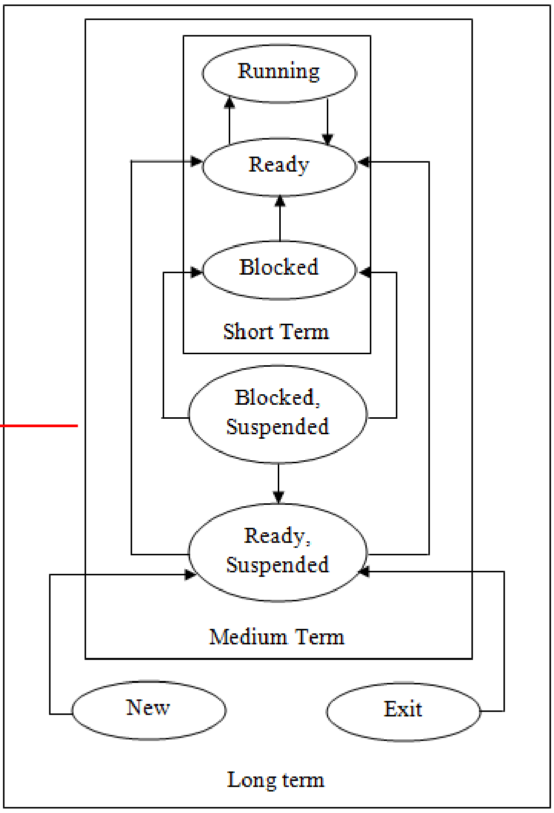
There are many possible criteria:

- **CPU Utilization:** Keep CPU utilization as high as possible i.e. keep the CPU as busy as possible. CPU utilization can range from 0 to 100 percents. In real system, it should range from 40% (for a light loaded system) to 90% (for a heavily used system)
- **Throughput:** It is number of processes completed per unit time. For long processes, this rate may be 1 process per hour; for short transactions, it may be 10 processes per second.
- **Turnaround Time:** The time interval from the submission of a process to the time completion is *turnaround time* i.e. amount of time to execute a particular process. It is the sum of the periods spends waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O. The turnaround time is generally limited by the speed of the output device.
- **Waiting Time:** Amount of time spent ready to run but not running i.e. the amount of time a process has been waiting in the ready queue. The CPU scheduling algorithm does not affect the amount of time that a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue.
- **Response Time:** It is time between submission of requests and first response to the request not the time it takes to output the response.
- **Scheduler Efficiency:** The scheduler doesn't perform any useful work, so any time it takes is pure overhead. So, need to make the scheduler very efficient.

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time and response time.

Scheduling Levels (Process scheduler)

Scheduler are responsible for management of jobs, such as allocating resources needed for any specific job, partitioning of jobs schedule parallel executing of tasks, data management, event correlation, and etc. The **process scheduler** is the component of the operating system that is responsible for deciding whether the currently running process should continue running and, if not, which process should run next.



Types of schedulers: A multi programmed system may include as many as three types of scheduler:

1. **The long-term (high-level) scheduler**:
   - It admits new processes to the system. Long-term scheduling may be necessary because each process requires a portion of the available memory to contain its code and data.
   - It determines which programs are admitted to the system for processing. Thus, it controls the degree of multiprogramming.
   - Once admitted job or program becomes a process and is added to the queue for the short term scheduler.
   - This scheduler dictates what processes are to run on a system, and the degree of concurrency to be supported at any one time.
   - A long-term scheduler is executed infrequently - only when a new job arrives. Thus, it can be fairly sophisticated.
   - Long term schedulers are generally found on batch systems, but are less common on timeshared systems or single user systems.

- Long-term scheduling performs a gate-keeping function. It decides whether there's enough memory, or room, to allow new programs or jobs into the system.
- When a job gets past the long-term scheduler, it's sent on to the medium-term scheduler.

2. **Medium-term (intermediate) scheduling**
   - The mid-term scheduler temporarily removes processes from main memory and places them on secondary memory (such as a disk drive) or vice versa. This is commonly referred to as "swapping out" or "swapping in"
   - The mid-term scheduler may decide to swap out a process which has not been active for some time, or a process which has a low priority, or a process which is page faulting frequently, or a process which is taking up a large amount of memory in order to free up main memory for other processes, swapping the process back in later when more memory is available, or when the process has been unblocked and is no longer waiting for a resource.
   - It controls which processes are actually resident in memory, as opposed to being swapped out to disk.
   - A medium-term scheduler (if it exists) is executed more frequently. To minimize overhead, it cannot be too complex.
   - The medium-term scheduler makes the decision to send a job on or to sideline it until a more important process is finished. Later, when the computer is less busy or has less important jobs, the medium-term scheduler allows the suspended job to pass.

3. **Short term scheduler:**
   - The short term scheduler determines the assignment of the CPU to ready processes i.e. it decides which of the ready, in-memory processes are to be executed (allocated a CPU) next following a clock interrupt, an IO interrupt, an operating system call or another form of signal
   - It is also known as *dispatcher*.
   - It makes scheduling decisions much more frequently than the long-term or mid-term schedulers
   - The short-term scheduler takes jobs from the "ready" line and gives them the green light to run. It decides which of them can have resources and for how long.
   - The short-term scheduler runs the highest-priority jobs first and must make on-the-spot decisions.
   - For example, when a running process is interrupted and may be changed, the short-term scheduler must recalibrate and give the highest-priority job the green light.
   - This scheduler can be preemptive, or non-preemptive.

Preemptive Versus Non-Preemptive Scheduling

CPU Scheduling is of two types depending on nature and they are:

a. Preemptive scheduling and
b. Non-preemptive scheduling.

Preemptive Scheduling

- Tasks are usually assigned with priorities. At times it is necessary to run a certain task that has a higher priority before another task although it is running. Therefore, the running task is interrupted for some time and resumed later when the priority task has finished its execution. This is called preemptive scheduling.
- It allows the process to be interrupted in the midst of its execution i.e. forcibly removes process from the CPU to allocate that CPU to another process.
- The preemptive scheduling is prioritized. The highest priority process should always be the process that is currently utilized.
- Example: Round Robin scheduling

Non –Preemptive Scheduling

- Under non-preemptive, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.
- It does not allow the process to be interrupted in the midst of its execution i.e. it is unable to "force" processes off the CPU.
- It is also known as *Cooperative* scheduling or *voluntary*.
- This scheduling method was used by Microsoft Windows 3.x.
- Example: FCFS scheduling

Scheduling Algorithm Optimization Criteria:

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

Scheduling Techniques

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU i.e. The objective of scheduling algorithms is to assign the CPU to the next ready process based on some predetermined policy. There are many different CPU scheduling algorithms. Some of them are:

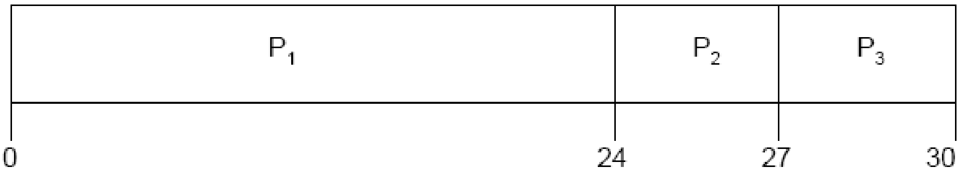
First in First Served (FCFS) Scheduling

It is simplest CPU scheduling algorithm. The FCFS scheduling algorithm is non preemptive i.e. once the CPU has been allocated t a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.
In this technique, the process that requests the CPU first is allocated the CPU first i.e. when a process enters the ready queue; its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.
The average waiting time under this technique is often quite long.

Consider the following set of processes that arrives at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

Suppose that the processes arrive in the order: *P1, P2,* and *P3*
The Gantt chart for the schedule is:

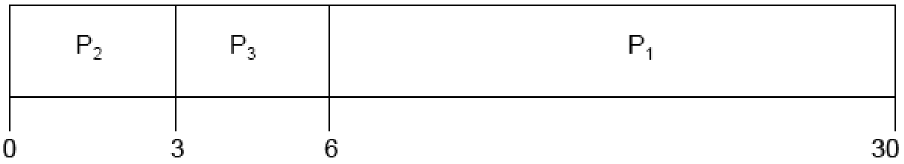


- Waiting time for *P1* = 0; *P2* = 24; *P3* = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

Suppose that the processes arrive in the order:

P2, P3, and P1

The Gantt chart for the schedule is:



- Waiting time for *P1* = 6; *P2* = 0; *P3* = 3
- Average waiting time: (6 + 0 + 3)/3 = 3 milliseconds

Shortest Job First (SJF) Scheduling

This technique is associated with the length of the next CPU burst of a process. When the CPU is available, it is assigned to the process that has smallest next CPU burst. If the next bursts of two processes are the same, FCFS scheduling is used.

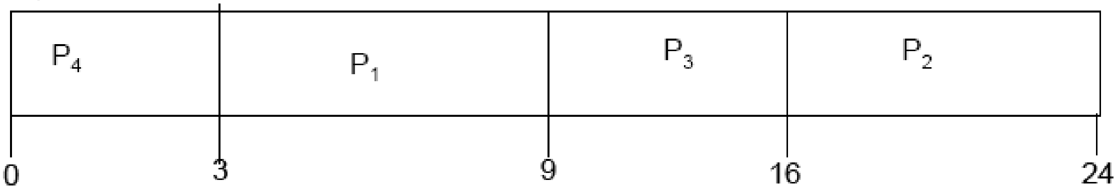
The SJF algorithm is optimal i.e. it gives the minimum average waiting time for a given set of processes.

The real difficulty with this algorithm knows the length of next CPU request.

Let us consider following set of process with the length of the CPU burst given in milliseconds.

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

SJF scheduling chart



The waiting time for process P1= 3, P2 = 16, P3 = 9 and P4 = 0 milliseconds, Thus

- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7 milliseconds

A preemptive SJF algorithm will preempt the currently executing, where as a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is also known **Shortest-Remaining-time (SRT) First Scheduling.**

Let us consider following four processes with the length of the CPU burst given in milliseconds.

Process	Arrival time	Burst time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Gantt chart is:

P1	P2	P4	P1	P3

0 1 5 10 17 26

Process P1 is started at time 0, since it is the only process in the queue. Process P2 arrives at time 1. The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds), so process P1 is preempted, and process P2 is scheduled.

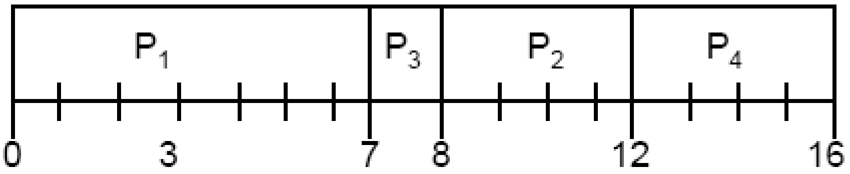
The average waiting time is:

= (P1 + P2 + P3 + P4) / 4
= [(10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)]/4
= (9 + 0 + 15 + 2)/4
= 26 / 4
= 6.5 milliseconds

Example: Non-Preemptive SJF

Process	Arrival time	Burst time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

- SJF (non-preemptive) Gant Chart



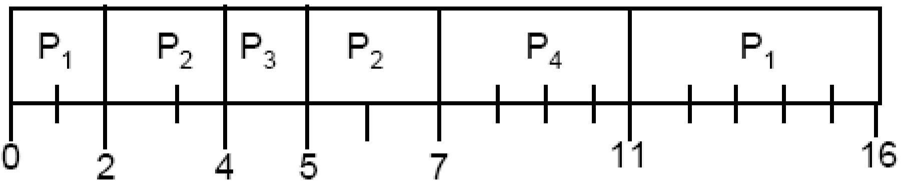
- Average waiting time

= (P1 + P2 + P3 + P4) / 4
= [(0 – 0) + (8 – 2) + (7 – 4) + (12 – 5)] / 4
= (0 + 6 + 3 + 7) / 4
= 4 milliseconds

Example: Preemptive SJF

Process	Arrival time	Burst time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

- SJF (preemptive) (Shortest Remaining Time First, SRTF) Gantt chart



- Average waiting time

$$= [(11 - 2) + (5 - 4) + (4 - 4) + (7 - 5)] / 4$$
$$= (9 + 1 + 0 + 2) / 4$$
$$= 3 \text{ milliseconds}$$

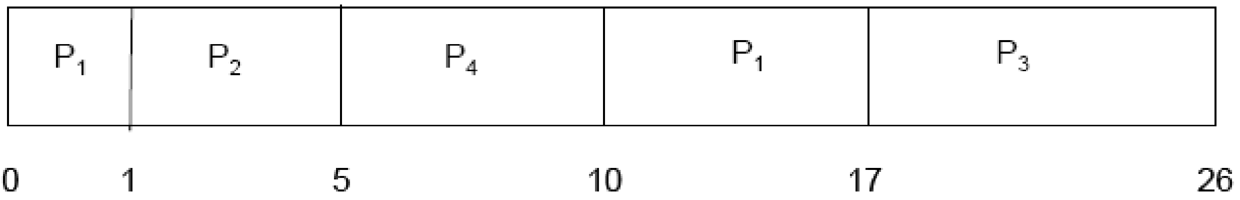
Shortest Remaining Time (SRT) Scheduling

It is a preemptive version of SJF algorithm where the remaining processing time is considered for assigning CPU to the next process.

- Now we add the concepts of varying arrival times and preemption to the analysis

Process	Arrival time	Burst time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 milliseconds

Priority Scheduling

A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal priorities are scheduled in FCFS order. Priorities are generally indicated by some fixed range of numbers and there is no general method of indicating which is the highest or lowest priority, it may be either increasing or decreasing order.

Priority can be defined either internally or externally.

- Internally defined priorities use some measurable quantity to compute the priority of a process. For example, time limits, memory requirements, the number of open files and the ratio of average I/O burst to average CPU burst has been used in computing priorities.
- External priorities are set by criteria outside the OS, such as importance of process, the type and amount of funds being paid for computer user, and other political factors.

Priority scheduling can be either preemptive or non-preemptive. When a process arrives at the ready queue, its priority is compared with the priority of currently running process.

- A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
- A non-preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

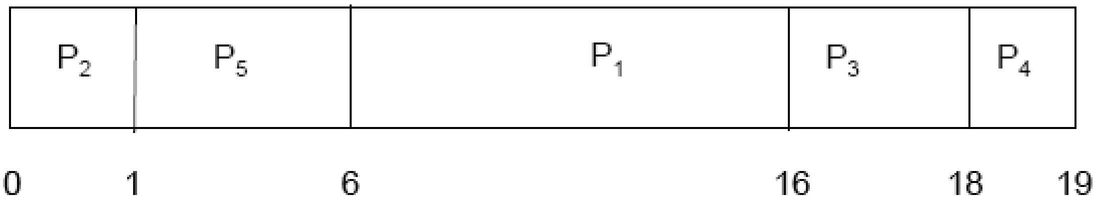
A major problem of such scheduling algorithm is **indefinite blocking** or **starvation.** A process that is ready to run but waiting for the CPU can be considered blocked. Such scheduling can leave some low priority process waiting indefinitely. The solution for this problem is **aging**. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.

Consider following set of processes, assumed to have arrived at time 0 in order P1, P2, …, P5 with the length of the CPU burst given in milliseconds

Example:

Process	Burst time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Priority scheduling Gantt chart



Average waiting time

= (6 + 0 + 16 + 18 + 1) / 5
= 41 / 5
= 8.2 milliseconds

Round Robin Scheduling

Round Robin (RR) scheduling algorithm is designed especially for time sharing systems. It is similar to FCFS scheduling but preemption is added to enable the system to switch between processes. A small unit of time called **time quantum** or **time slice** is defined. A time quantum is generally from 10 to 100 milliseconds in length. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

The process may have a CPU burst less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the OS. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

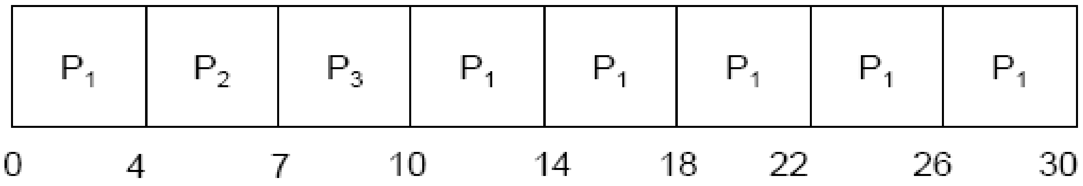
The average waiting time under the RR policy is often long.

Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds.

Example 1: Quantum time = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

The Gantt chart is:



The process P1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P2. Process P2 does not need 4 milliseconds, so it quits before its time quantum expires. The CPU is then given to the next process, process P3. Once each process has received 1 time quantum time, the CPU is returned to the process for an additional time quantum.

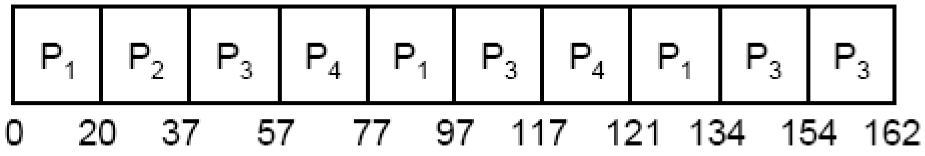
Average time:

= (P1 + P2 + P3) / 3
= [(10 – 4) + 4 + 7] / 3
= 17 / 3 = 5.66 milliseconds

Example 2: quantum = 20

Process	Burst time	Waiting time for each process
P1	53	0 + (77 – 20) + (121 – 97) = 81
P2	17	20
P3	68	37 + (97 – 57) + (134 – 117) = 94
P4	24	57 + (117 – 77) = 97

Gantt chart



- Average Waiting Time
  = (P1 + P2 + P3 + P4) / 4
  = [{0 + (77 – 20) + (121 – 97)} + 20 + {37 + (97-57) + (134 – 117)} + {57 + (117 – 77)}] / 4
  = (81+20+94+97) / 4 = 73 milliseconds

If there are ***n*** processes in the ready queue and the time quantum is ***q***, then each process gets ***1/n*** of the CPU time in chunks of at most ***q*** time units. Each process must wait no longer than ***(n – 1) x q*** time units until its next time quantum. For example, there are 5 processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.

The performance of the RR algorithm depends heavily on the size of the time quantum.

- If the time quantum is extremely large, the RR policy is similar to FCFS policy.
- If the time quantum is extremely small (say 1 millisecond), the RR approach is called **processor sharing** and creates the appearance that each of n processes has its own processor running at 1/n speed of the real processor.

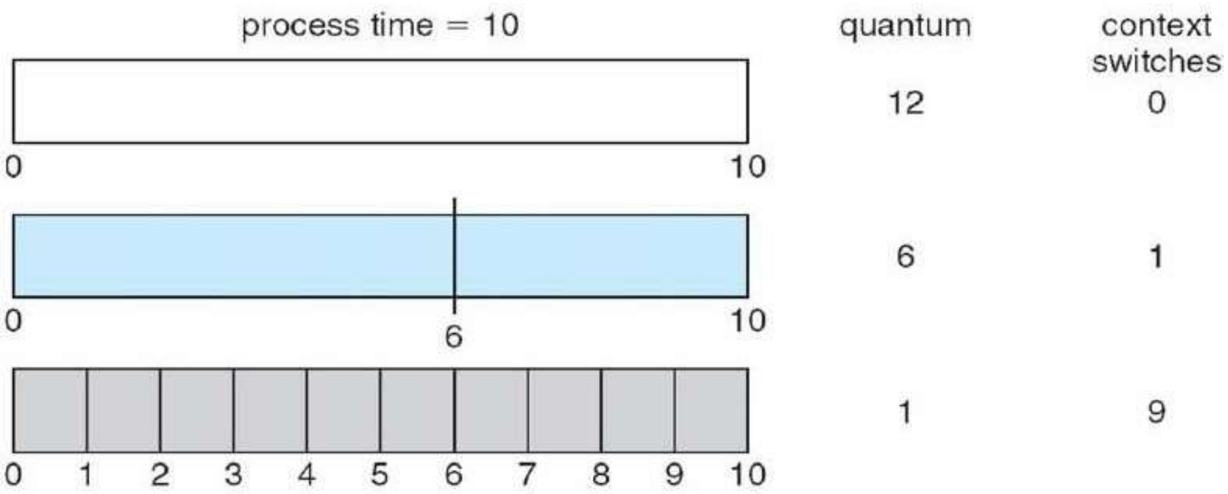
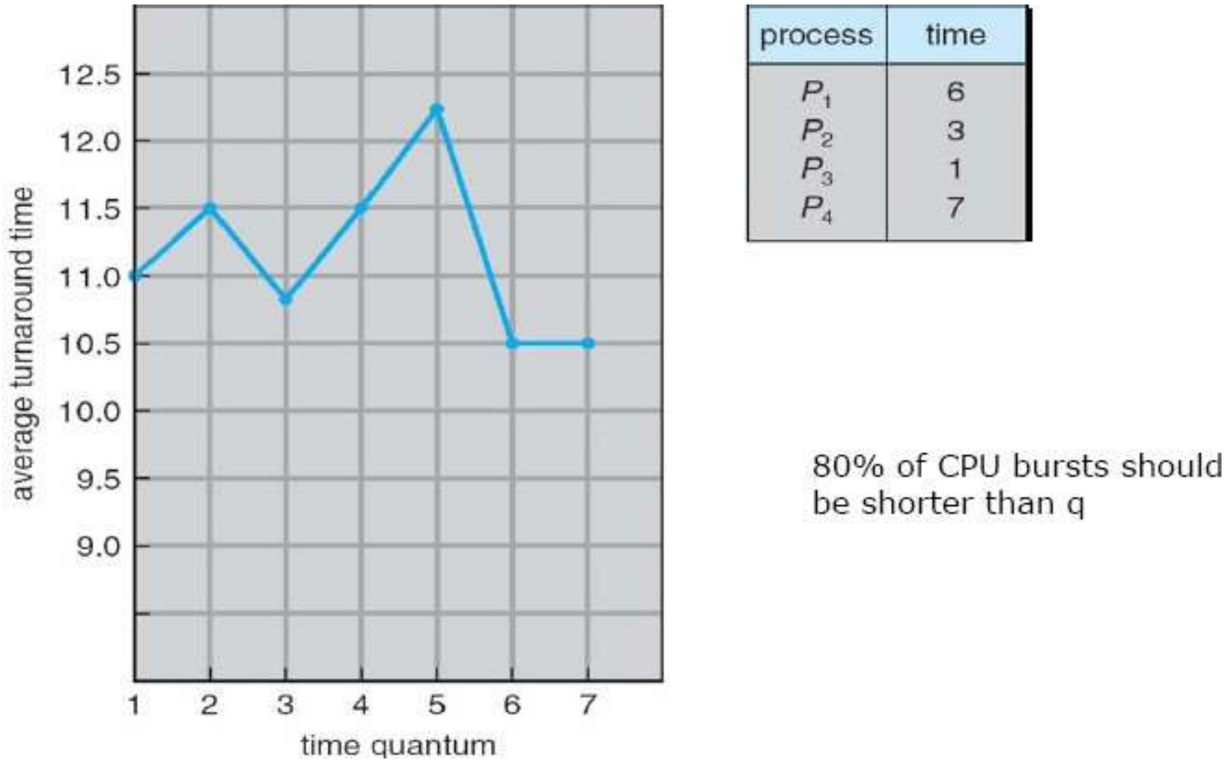


Fig: Quantum time and context switching

Turnaround time varies with the time quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts should be shorter than q

Fig: Quantum time and Turnaround Time

Highest-Response Ration Next (HRN) Scheduling

Highest Response Ratio Next (HRRN) scheduling is a non-preemptive discipline, in which the priority of each job is dependent on its estimated run time, and also the amount of time it has spent waiting. Jobs gain higher priority the longer they wait, which prevents indefinite postponement (process starvation).
It selects a process with the largest ratio of waiting time over service time. This guarantees that a process does not starve due to its requirements.
In fact, the jobs that have spent a long time waiting compete against those estimated to have short run times.

```
Priority = waiting time + estimated runtime / estimated runtime
                              (Or)
Ratio = (waiting time + service time) / service time
```

Advantages

- Improves upon SPF scheduling

- Still non-preemptive
- Considers how long process has been waiting
- Prevents indefinite postponement

Disadvantages

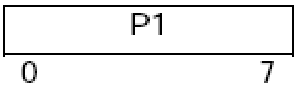
- Does not support external priority system. Processes are scheduled by using internal priority system.

Example: Consider the Processes with following Arrival time, Burst Time and priorities

Process	Arrival time	Burst time	Priority
P1	0	7	3 (High)
P2	2	4	1 (Low)
P3	3	4	2

Solution: HRRN

At time 0 only process p1 is available, so p1 is considered for execution



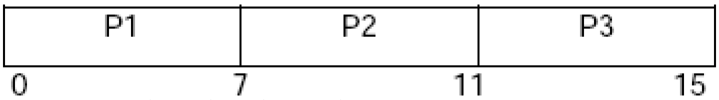
Since it is Non-preemptive, it executes process p1 completely. It takes 7 ms to complete process p1 execution.

Now, among p2 and p3 the process with highest response ratio is chosen for execution.

Ratio for p2 = (5 + 4) / 4 = 2.25

Ratio for p3 = (4 + 4) / 4 = 2

As process p2 is having highest response ratio than that of p3. Process p2 will be considered for execution and then followed by p3.



Average waiting time = 0 + (7 - 2) + (11 - 3) / 3 = 4.33

Average Turnaround time = 7 + (11 - 2) + (15 - 3) / 3 = 9.33

Multilevel Queues

A multilevel queue scheduling algorithm partitions the ready queues into several separate queues. The processes are permanently assigned to one queue, based on some property of the process, such as memory size, process priority or process type. Each queue has its own scheduling algorithm. For example, separate queue might be used for foreground and background processes. The foreground queue might be scheduled by RR algorithm, while the background queue is scheduled by FCFS algorithm.

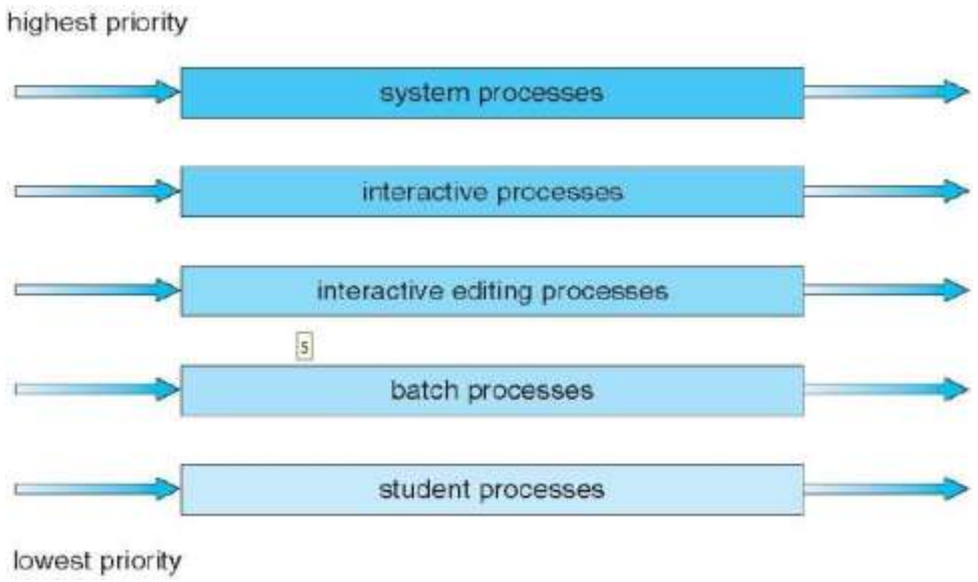


Fig: Multilevel queue scheduling

Let us consider following five queues with its priority.

1. System processes
2. Interactive processes
3. Interactive editing processes
4. Batch processes
5. Student processes.

Each queue has absolute priority over lower priority queues. No process can in the batch queue, could run unless the queues for system process, interactive process, and interactive editing processes were all empty. If an interactive editing process entered in the ready queue while a batch process was running, the batch process would be preempted.

In such scheduling algorithm, processes are permanently assigned to a queue when they enter the system. If there are separate queues foreground and background processes, for example, process do not move from one queue to another, since processes do not change their foreground and background nature. So it is inflexible.

Multilevel Feedback Queues

The multileveled feedback scheduling algorithm allows a process to move between queues. This method separates processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O bound and interactive processes in the higher-priority queues i.e. a process that waits too long in a lower-priority queue may be moved to a higher-priority queue which prevents starvation.

For example, consider a multilevel feedback queue scheduler with 3 queues as in following figure, numbered from 0 to 2. The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly processes in queue 2 will only be executed if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.

A process entering the ready queue is put in queue 0, a process in queue 0 is given a time quantum of 8 milliseconds. If it doesn't finish within this time, it is moved to the tail of the queue1. If the queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it doesn't complete, it is preempted and is put into queue 2. Process in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.

Multilevel-feedback-queue scheduler defined by the following parameters:

- ✓ The number of queues
- ✓ The scheduling algorithms for each queue
- ✓ The method used to determine when to upgrade a process
- ✓ The method used to determine when to demote a process
- ✓ method used to determine which queue a process will enter when that process needs service

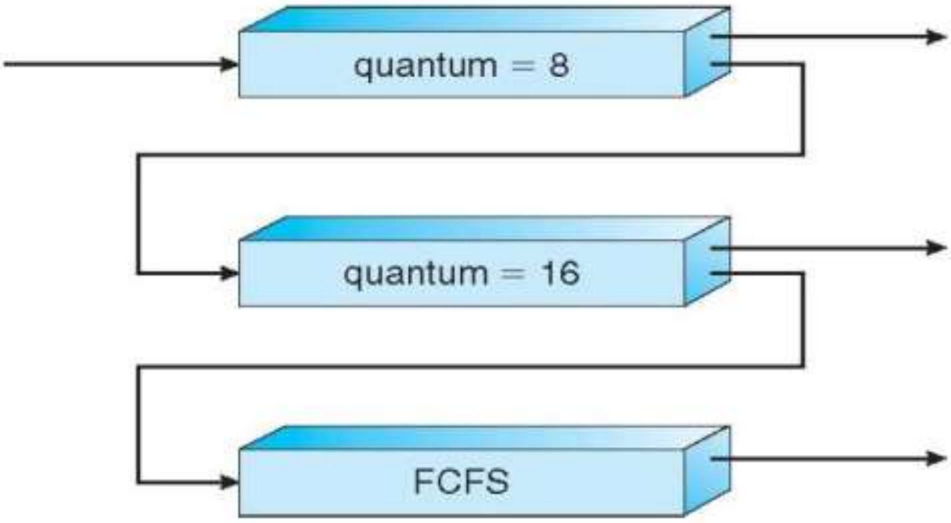


Fig: Multilevel feedback queue

Unit 5
Memory Management

Introduction: Memory Management

Memory is the physical device which is used to store programs or data on a temporary or permanent basis for use in a computer or other digital electronics device. There are various types of memories in a computer system and are accessed by various processes for their execution.

It is most important and most complex task of an operating system. Memory management involves treating main memory as a resource to be allocated to and shared among number of active processes i.e. it is the act of managing computer memory. It involves providing ways to allocate portions of memory to programs at their request, and freeing it for reuse when no longer needed. To efficiently use the processor and the I/O facilities, it is desirable to maintain as many processes in main memory as possible.

Memory management is sub module of an OS, whose responsibility is to

- Allocate and de-allocate memory space as requested.
- Keep track of which parts of memory are currently being used and by whom.
- Efficient utilization when the memory resource is heavily competed.

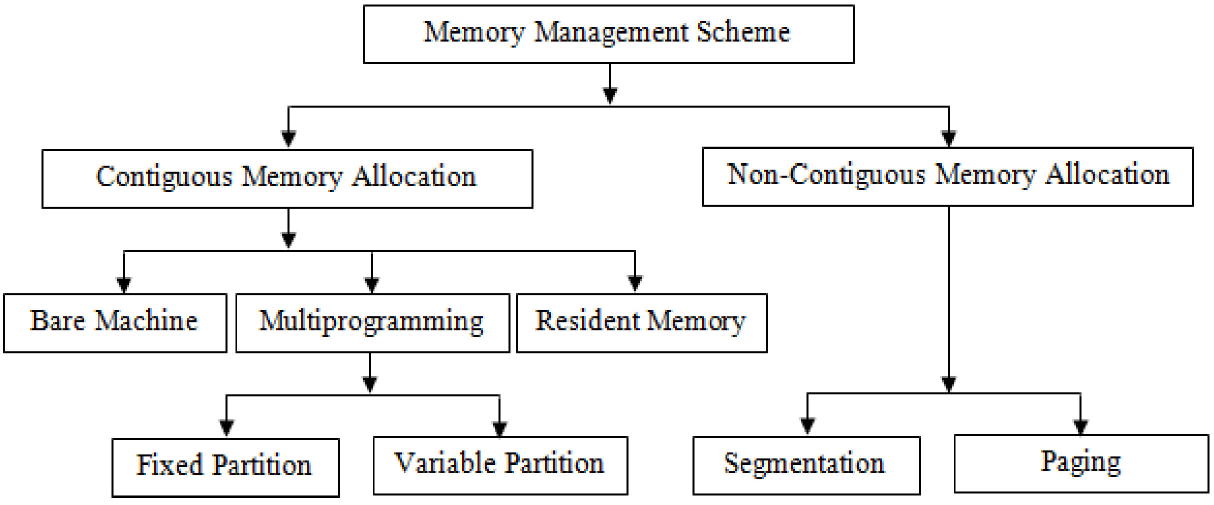


Fig: types of memory management

The memory management subsystem provides:

Large Address Spaces

The operating system makes the system appear as if it has a larger amount of memory than it actually has. The virtual memory can be many times larger than the physical memory in the system,

Protection

Each process in the system has its own virtual address space. These virtual address spaces are completely separate from each other and so a process running one application cannot affect another. Also, the hardware virtual memory mechanisms allow areas of memory to be protected against writing. This protects code and data from being overwritten by rogue applications.

Memory Mapping

Memory mapping is used to map image and data files into a processes address space. In memory mapping, the contents of a file are linked directly into the virtual address space of a process.

Fair Physical Memory Allocation

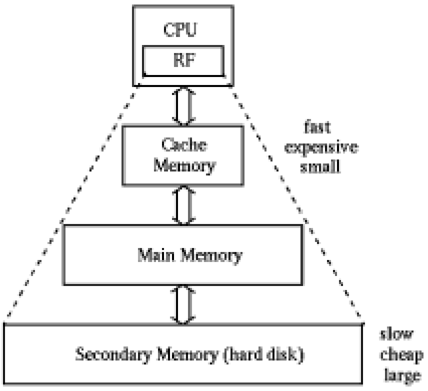
The memory management subsystem allows each running process in the system a fair share of the physical memory of the system,

Shared Virtual Memory

Although virtual memory allows processes to have separate (virtual) address spaces, there are times when we need processes to share memory. For example there could be several processes in the system running the bash command shell. Rather than have several copies of bash, one in each processes virtual address space, it is better to have only one copy in physical memory and all of the processes running bash share it. Dynamic libraries are another common example of executing code shared between several processes.

Storage Organization, Hierarchy and Management

Memory hierarch is a ranking of computer memory devices, with devices having the fastest access time at the top of the hierarchy, and devices with slower access times but larger capacity and lower cost at lower levels.



The Memory Hierarhcy

The memories in the hierarchy are as follows:

- At the top level of the memory hierarchy are the CPU's **general purpose registers**. The registers provide the fastest access to data possible on the 80x86 CPU. The register file is also the smallest memory object in the memory hierarchy.
- The **Level One Cache (L1)** system is the next highest performance subsystem in the memory hierarchy. On the 80x86 CPUs, the Level One Cache is provided on-chip by Intel and cannot be expanded. The size is usually quite small (typically between 4Kbytes and 32Kbytes), though much larger than the registers available on the CPU chip. Although the Level One Cache size is fixed on the CPU and we cannot expand it, the cost per byte of cache memory is much lower than that of the registers because the cache contains far more storage than is available in all the combined registers.
- The **Level Two Cache (L2)** is present on some CPUs. The Level Two Cache is generally much larger than the level one cache (e.g., 256 or 512KBytes versus 16 Kilobytes). On systems where the Level Two Cache is external, many system designers let the end user select the cache size and upgrade the size. For economic reasons, external caches are actually more expensive than caches that are part of the CPU package.
- Below the Level Two Cache system in the memory hierarchy the main memory subsystem is available. This is the general-purpose, relatively low-cost memory found in most computer systems. *Main memory* or *internal memory* is the only one directly accessible to the CPU. The CPU continuously reads instructions stored there and executes them as required.
- *Secondary memory* also known as external memory or auxiliary storage is different from primary memory in that it is not directly accessible by the CPU. The data stored in such memories are permanently stored and are not lost even the power cut off i.e. non-volatile in nature. Such memories are relatively inexpensive than others.

Operation on memory hierarchy:

Moving data between the registers and the rest of the memory hierarchy is strictly a program function. The program, of course, loads data into registers and stores register data into memory using instructions like MOV. It is strictly the programmer's or compiler's responsibility to select an instruction sequence that keeps heavily referenced data in the registers as long as possible.

Most transparent memory subsystem accesses always take place between one level of the memory hierarchy and the level immediately below or above it. For example, the CPU rarely accesses main memory directly. Instead, when the CPU requests data from memory, the Level One Cache subsystem takes over. If the requested data is in the cache, then the Level One Cache subsystem returns the data and that's the end of the memory access. On the other hand if the data is not present in the level one cache, then it passes the request

Fetch Strategy
Placement Strategy
Replacement

on down to the Level Two Cache subsystem. If the Level Two Cache subsystem has the data, it returns this data to the Level One Cache, which then returns the data to the CPU. Note that requests for this same data in the near future will come from the Level One Cache rather than the Level Two Cache since the Level One Cache now has a copy of the data.

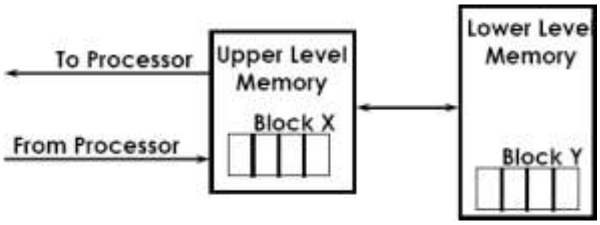
If neither the Level One nor Level Two Cache subsystems have a copy of the data, then the memory subsystem goes to main memory to get the data. If found in main memory, then the memory subsystems copy this data to the Level Two Cache which passes it to the Level One Cache which gives it to the CPU. Once again, the data is now in the Level One Cache, so any references to this data in the near future will come from the Level One Cache.

If the data is not present in main memory, but is present in Virtual Memory on some storage device, the operating system takes over, reads the data from disk (or other devices, such as a network storage server) and places this data in main memory. Main memory then passes this data through the caches to the CPU. Because of locality of reference, the largest percentage of memory accesses takes place in the Level One Cache system. The next largest percentage of accesses occurs in the Level Two Cache subsystems. The most infrequent accesses take place in Virtual Memory.

At any given time, data are copied between only two adjacent levels:

- Upper level and
- Lower level

The upper level are the one closer to the processor which is smaller, faster, uses more expensive technology where as the lower level are the one away from the processor which are bigger, slower, uses less expensive technology



Storage allocation

Many files are stored on the same disk. The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly. **Storage allocation** is the assignment of particular areas of a magnetic disk to particular data or instructions. An allocation method refers to how disk block are allocated for files.

There are various methods. Some of major methods are:

- Contiguous memory allocation
- Linked allocation
- Indexed allocation, etc.

Contiguous versus Noncontiguous storage allocation

- Each file occupies a set of contiguous blocks on the disk
- Simple – only starting location (block) and length (number of blocks) are required
- Random access
- Wasteful of space (dynamic storage allocation problem)
- Files cannot grow.

Contiguous storage allocation

Contiguous storage allocation is allocation technique in which each program occupied a single contiguous memory block i.e. each logical object is placed in a set of memory locations with strictly consecutive addresses. In these systems, the technique of multiprogramming was not possible. 1-D array is an example of contiguous memory allocation, where one array element is immediately followed by the next.

Contiguous memory allocation is one of the oldest memory allocation schemes. When a process needs to execute, memory is requested by the process. The size of the process is compared with the amount of contiguous main memory available to execute the process. If sufficient contiguous memory is ~~found~~, the process is allocated memory to start its execution. Otherwise, it is added to a queue of waiting processes until sufficient free contiguous memory is available.

The contiguous memory allocation scheme can be implemented in operating systems with the help of two registers, known as the **base** and **limit registers**. When a process is executing in main memory, its base register contains the starting address of the memory location where the process is executing, while the amount of bytes consumed by the process is stored in the limit register. A process does not directly refer to the actual address for a corresponding memory location. Instead, it uses a relative address with respect to its base register. All addresses referred by a program are considered as virtual addresses. The CPU generates the logical or virtual address, which is converted into an actual address with the help of the memory management unit (MMU). The base address register is used for address translation by the MMU.

Thus, a physical address is calculated as follows:
Physical Address = Base register address + Logical address/Virtual address.

The address of any memory location referenced by a process is checked to ensure that it does not refer to an address of a neighboring process. This processing security is handled by the underlying operating system.

Features of contiguous storage allocation:

- It is oldest technique.
- Contiguous disk space allocation is very simple to implement.
- The entire file can be read from the disk in a single operation i.e. data can be accessed more quickly.
- The degree of multiprogramming is reduced due to processes waiting for free memory.
- The method of ***contiguous storage allocation*** can cause disk fragmentation. If file is removed and blocks become free. It is also not possible to compact the disk on the spot to compress the hole.

Noncontiguous storage allocation

In non-contiguous storage allocation, a program is divided into several blocks that may be placed in different parts of main memory i.e. it implies that a single logical object may be placed in non-consecutive sets of memory locations. It is more difficult for an operating system to control non-contiguous storage allocation. The benefit is that if main memory has many small holes available instead of a single large hole, then operating system can often load and execute a program that would otherwise need to wait.

There are two mechanisms that are used to manage non-contiguous memory allocation and they are:

- Paging (System view) and
- Segmentation (User view)

Logical and Physical Memory

Logical address: An address generated by the CPU is commonly referred to as a logical address,
Physical address: It is also known as absolute address. An address seen by the memory unit i.e. the one loaded into the memory address register of the memory is commonly referred as a physical memory.

Logical address space: The set of all logical addresses generated by a program is a *logical address space*;
Physical address space: The set of all physical address corresponding to these logical addresses is a *physical address space.*
Memory-management Unit (MMU): The run-time mapping from virtual to physical addresses is done by a hardware device called the memory-management unit (MMU).

Fragmentation

It is a technique of partition of memory into different blocks or pages while a process are loaded or removed from the memory. The free memory space is broken into little pieces; such types of pieces may or may not be of any use to be allocated individually to any process. This may give rise to term memory waste or fragmentation. In other words, Fragmentation refers to the condition of a disk in which files are divided into pieces scattered around the disk. Fragmentation occurs naturally when we use a disk frequently, creating, deleting, and modifying files. Fragmentation occurs in a dynamic memory allocation system when many of the free blocks are too small to satisfy any request. This is entirely invisible to users, but it can slow down the speed at which data is accessed because the disk must search through different parts of the disk to put together a single file.

There are two types of fragmentation and they are:

- External fragmentation
- Internal fragmentation

External fragmentation: it happens when a dynamic memory allocation algorithm allocates some memory and small pieces are left over that cannot be effectively used. If too much external fragmentation occurs, the amount of usable memory is drastically reduced. Total memory space exists to satisfy a request, but it is not contiguous.

Internal Fragmentation: Internal fragmentation is the space wasted inside of allocated memory blocks because of restriction on the allowed sizes of allocated blocks. Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

Fragmentation can be done by two methods and they are:

1. Fixed partition multiprogramming
2. Variable partition multiprogramming.

Fixed Partition Multiprogramming (Static)

Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This leaves holes of unused memory inside the partitions: *internal fragmentation.*

In most schemes for memory management, we can assume that the OS occupies some fixed portion of main memory and that the rest of main memory is available for use by multiple processes. The simplest scheme for managing the available memory is partition it into regions with fixed boundaries.

Following diagram shows two alternatives for fixed partitioning.

- Equal-size of partition
- Unequal-size of partition

a) Equal-size of partition

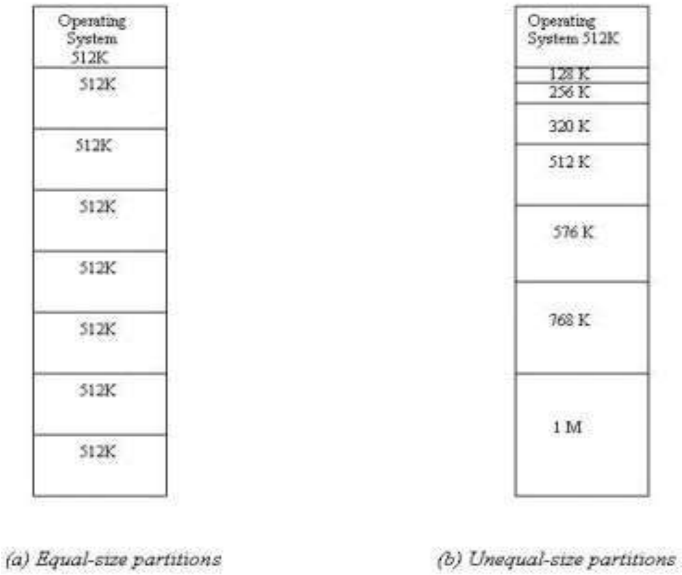
In this case, any process whose size is less than or equal to the partition size can be loaded into any available partition. If all partitions are full and no resident process is in the ready or running state, the OS can swap a process out of any of the partitions and load in another process.

There are two difficulties with the use of such equal-size fixed partitions:

- A program may be too big to fit into a partition.
- Internal fragmentation (unused space within partitions)

b) Unequal-size of partition

In this method each process is assigned to the smallest partition within which it will fit. In this case a scheduling queue is needed for each partition, to hold swapped – out processes destined for that partition. The advantage of this approach is that processes are always assigned in such a way as to minimize wasted memory within a partition. This technique seems optimum for individual and not whole. When it is time to load a process into main memory, the smallest available partition that will hold the process is selected. If all partitions are occupied, then a swapping decision must be made. Preference may be given to the swapping out of the smallest partition that will hold the incoming process.

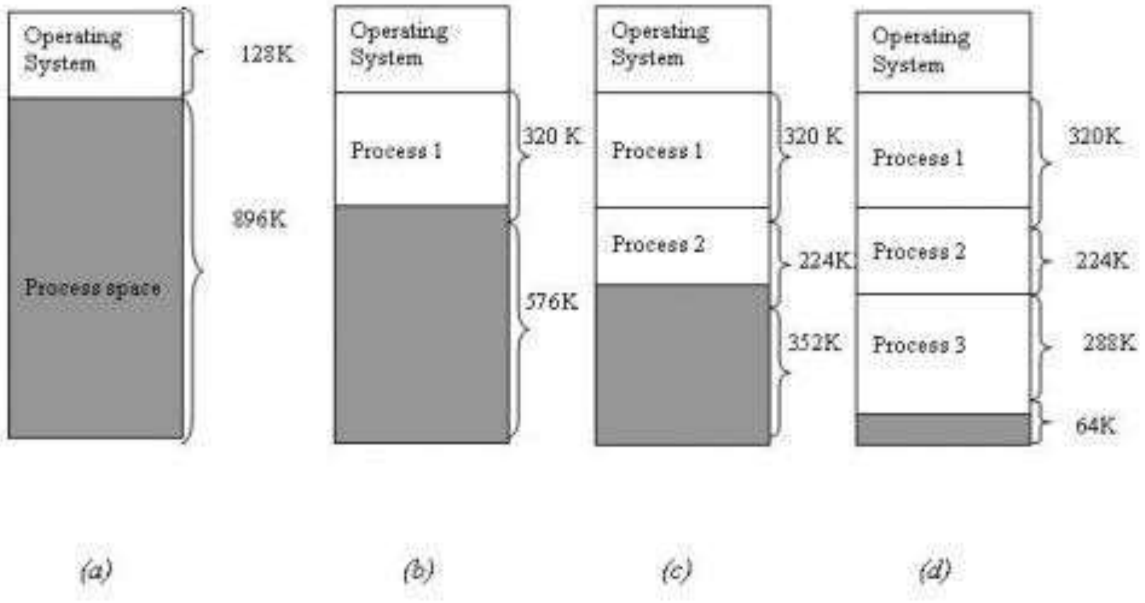


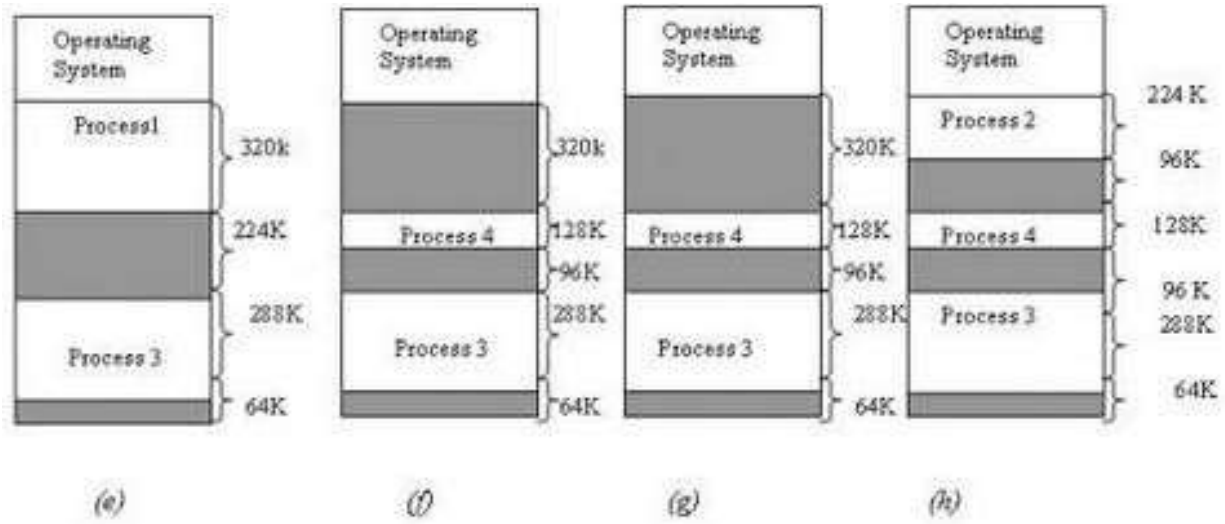
(a) Equal-size partitions (b) Unequal-size partitions

Variable Partition Multiprogramming (Dynamic)

In such scheme, partitions are of variable length and number. When a process is brought into main memory, it is allocated exactly as much memory as it requires and no more. Let us consider 1MB of memory is shown in following diagram.

- Initially, main memory is empty except for the OS (fig a).
- The first three processes are loaded in, starting where the OS ends, and occupy just enough space for each process (fig b, c and d). This leaves a "hole" at the end of memory that is too small for a fourth process. At some point, none of the processes in memory is ready.
- The OS therefore swaps out process 2 (fig e), which leaves sufficient room to load a new process, process 4 (fig f). Because process 4 is smaller than process 2, another small hole is created.
- Later, a point is reached at which none of the processes in main memory is ready, but process 2 in the Ready, suspend state is available. Because there is insufficient room in memory for process 2, the OS swaps process 1 out (fig g) and swaps process 2 back in (fig h).
- This phenomena is called ***external fragmentation*** i.e. the memory that is external to all partitions becomes increasingly fragmented.





We can overcome from such external fragmentation by following techniques:

Compaction: it is time consuming procedure, wasteful of processor time. It involves shifting a program in memory in such a way that the program does not notice the change. This consideration requires that logical addresses be relocated dynamically, at execution time. If addresses are relocated only at load time we cannot support compact storage.

Placement algorithm

There are 3 placement algorithms: first fit, next fit and best fit

First fit memory allocation:

- find the first unused block of memory that can contain the process,
- Faster in making allocation of memory space

Best fit memory allocation:

- chooses the block that is closest in size to the request i.e. the smallest partition fitting the requirements
- Results in least wasted space
- Internal fragmentation reduced but not eliminated.
- Slower in making allocation

Next fit:

- Starts searching from last allocated block and choose the next available block that is large enough, for the rest available block when a new job arrives.

Worst fit:

- Allocates the largest free available block to the new job
- Opposite of best fit

Solved Numerical:

1. For the following partition of 100K, 500K, 200K, 300K and 600K (in order) place the processes 212K, 417K, 122K, and 426K in order according to the: Best fit and worst fit

Sol^n: Here, the given processes are:

P1 = 212K P2 = 417K P3 = 122K P4 = 426K

i. Best fit

Operating System

	100K
P2	500K
P3	200K
P1	300K
P4	600K

Fig: Best Fit

ii. Worst fit

Operating System	
	100K
P2	500K
	200K
P3	300K
P1	600K

Fig: Worst Fit

2. For the following partition of 100K, 500K, 300K, 50K, and 600K in order place the process 212K, 417K, 122K and 40K in order to: First fit and Next fit.

Sol^{n}: Here, the given processes are:

P1 = 212K   P2 = 417K   P3 = 122K   P4 = 40K

i. Best fit

Operating System	
P4 (40K)	100K
P1 (212K)	500K
P3 (122K)	300K
	50K
P2 (417K)	600K

Fig: Best Fit

ii. Next fit

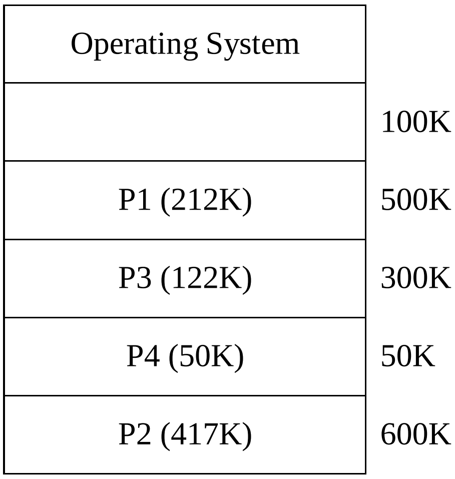


Fig: Next Fit

Swapping:

Each and every process must be in main memory for execution. However, a process can be transferred temporarily out of memory to a backing storage device for certain time period and then brought back into main memory for continued execution. This process of transferring a process from main memory to backing storage and vice-versa is known as **Swapping.** Transformation of process from main memory to backing memory is called *swapped* out and from backing memory to main memory is known as *swapped in.* This scheme allows more processes to be run than can be fit into memory at one time.

For example:

> Let us consider a multiprogramming environment with a round-robin CPU-scheduling algorithm. When a quantum expires, the memory management will start to swap out the process that just finished and to swap another process into the memory space that has been freed. Normally, a process that is swapped out will be swapped back into the same memory space it occupied previously.

Swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants services, the memory manager can swap out the lower-process and then load and execute the higher-priority process. When the higher-priority process finishes, the lower-priority process can be swapped backed in and continued. Sometimes this variant of swapping is called ***roll out, roll in.***

However, all the process cannot be swapped. If we want to swap a process, we must be sure that the process is completely idle. For example, if a process is waiting for an I/O operation then it cannot be swapped to free up memory.

Swapping process can be shown in following figure.

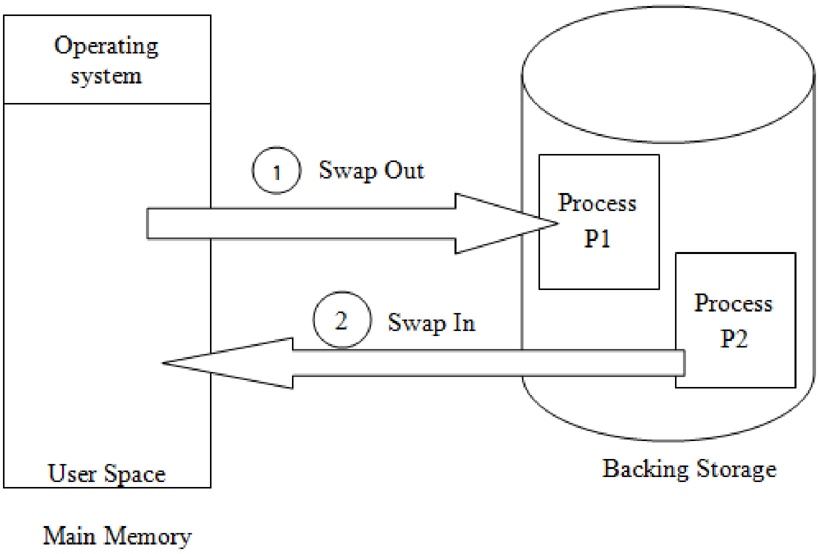


Fig: Swapping of two processes using a disk as a backing storage

Virtual Memory

Introduction

The main memory is considered as the physical memory in which many programs wants to reside. However, the size of such memory is limited and it cannot hold or load all the programs simultaneously. To overcome such types of problem, the overloaded inactive programs are swapped out from the main memory to hard disk for certain period of time. Such technique is known as virtual memory i.e. **Virtual memory** is a memory management system in a computer that temporarily stores inactive parts of the content of RAM on a disk, restoring it to RAM when quick access to it is needed.

Virtual memory permits software to use additional memory by utilizing the hard disc drive (HDD) as temporary storage. This technique involves the manipulation and management of memory by allowing the loading and execution of larger programs or multiple programs simultaneously. It is often considered more cost effective than purchasing additional RAM and enables the system to run more programs. However the process of mapping data back and forth between the hard drive and the RAM takes longer than accessing it directly from the memory i.e. depending upon virtual memory slows the computer.

Paging

Paging is one of the memory management schemes by which a computer can store and retrieve data from secondary storage for use in main memory. Paging is an important part of virtual memory implementation in most operating systems, allowing them to use disk storage for data that does not fit into physical RAM.

In this system, physical memory is broken down into fixed-sized blocks called **page frames** or simply **frames** and the logical (virtual) memory is broken down into the blocks of same sized blocks called **pages**. When a process is to be executed, its pages are loaded into any available memory frames form their source. The backing store is also divided into fixed-sized blocks that are of the same sizes as the memory frames.

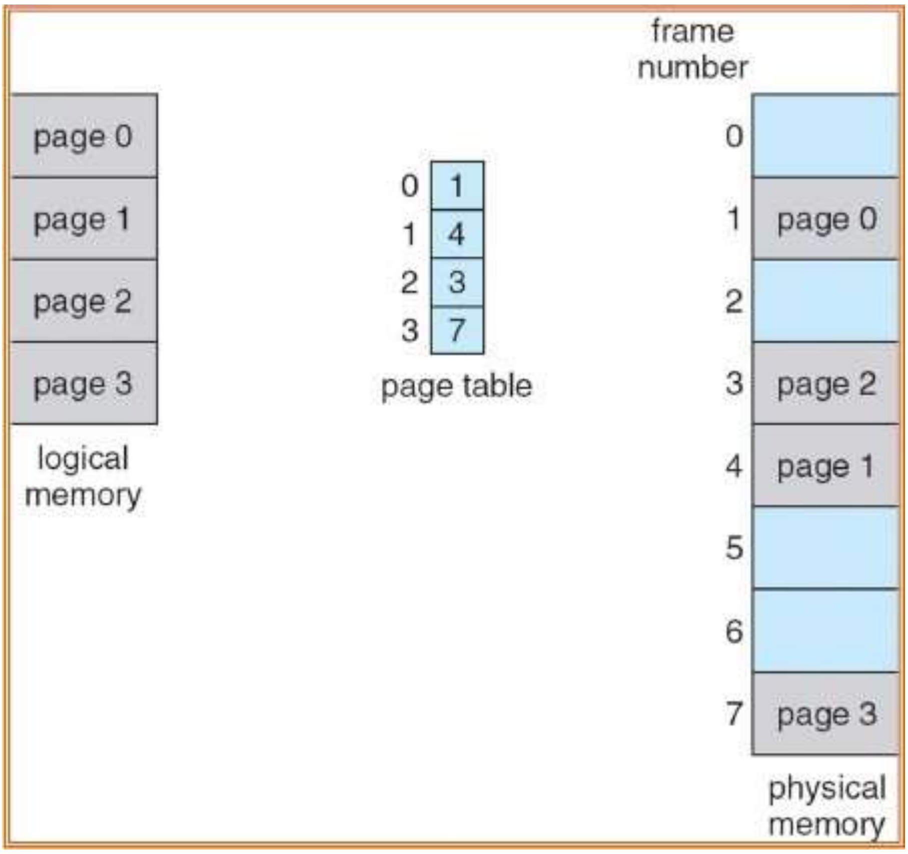


Fig: Paging concept

Page table:

A **page table** is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual address and physical address i.e. it is used to map virtual page number to physical page number. There are several types of page tables that are best suited for different requirements. Some of them are:

- Inverted page table
- Multilevel page table
- Virtualized page table
- Nested page table

TLB (Translation Look Aside Buffers)

A **translation lookaside buffer** (**TLB**) is a cache that memory management hardware uses to improve virtual address translation speed i.e. it acts as a cache for the page table. It is also referred to as the *address translation cache*. A TLB has a fixed number of slots that contain page table entries, which map virtual addresses to physical addresses. A typical TLB contains anywhere from 64 to 256 entries.

TLB is a table used in a virtual memory system that lists the physical address page number associated with each virtual address page number. A TLB is used in conjunction with a cache whose tags are based on virtual addresses. The virtual address is presented simultaneously to the TLB and to the cache so that cache access and the virtual-to- physical address translation can proceed in parallel. If the requested address is not cached then the physical address is used to locate the data in main memory. The alternative would be to place the translation table between the cache and main memory so that it will only be activated once there was a cache miss.

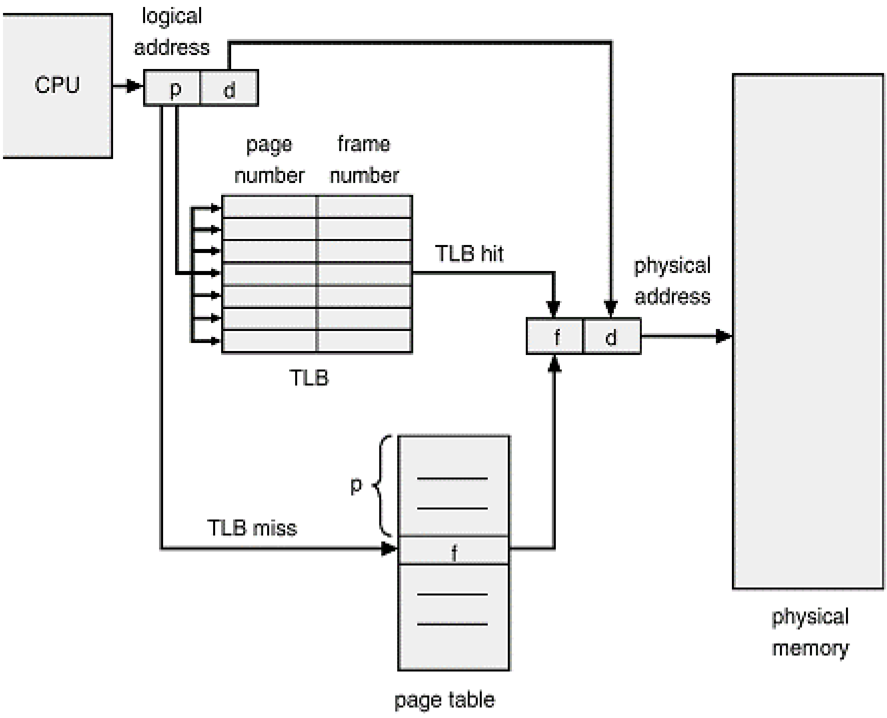
A translation lookaside buffer (TLB) has a fixed number of slots containing the following entries:
Page table entries, which map virtual addresses to

- Physical addressees

- intermediate table addresses

Segment table entries, which map virtual addresses to

- segment addresses
- intermediate table addresses
- page table addresses.



The organization of hardware of paging with TLB is shown in above diagram.

Whenever a program performs a memory reference, the virtual address sent to the TLB to determine if it contains a translation for the address i.e. the program will first examine the TLB. If the desired page table entry is present or mapped i.e. a TLB hit occurs, then the frame number is retrieved and the real address is formed. If desired page table entry is not present in TLB i.e. a TLB miss occurs, then the processor uses the page number to index the process page table and examine the corresponding page entry. Hence TLB is a memory cache which as a fast execution processing element.

Page Fault

The main functions of paging are performed when a program tries to access pages that are not currently mapped to physical memory (RAM). This situation is known as a **page fault**. The operating system must then take control and handle the page fault, in a manner invisible to the program. Therefore, the operating system must:

- Determine the location of the data in secondary storage.
- Obtain an empty page frame in RAM to use as a container for the data.
- Load the requested data into the available page frame.
- Update the page table to refer to the new page frame.
- Return control to the program, transparently retrying the instruction that caused the page fault.

If there is not enough available RAM when obtaining an empty page frame, a page replacement algorithm is used to choose an existing page frame for eviction.

Thrashing

Thrashing is a condition in which excessive paging operations are taking place i.e. it refer to any situation in which multiple processes are competing for the same resource, and the excessive swapping back and forth between connections causes a slowdown. This causes the performance of the computer to degrade or collapse.

Thrashing is often caused:

- when the system does not have enough memory,

- the system swap file is not properly configured, or
- too much is running on the computer and it has low system resources.

To resolve hard drive thrashing, a user can do any of the below.

- Increase the amount of RAM in the computer.
- Decrease the amount of programs being run on the computer.
- Adjust the size of the swap file.

Page Replacement Algorithm

Page Replacement Algorithm

When a page fault occurs, the operating system has to choose a page to remove from memory to make room for the page that has to be brought in. The page replacement is done by swapping the required pages from backup storage to main memory and vice-versa. A page replacement algorithm is evaluated by running the particular algorithm on a string of memory references and computes the page faults.

The basic page replacement method:

1. Find the location of the desired page on the disk.
2. Find a free frame:
   - If there is a free frame, use it.
   - If there is no free frame, use a page replacement algorithm to select a victim frame.
   - Write the victim frame to the disk; change the page and frame tables accordingly.
3. Read the desired page into the newly free frame; change the page and frame tables.
4. Restart the user process.

Example:

Let us consider the reference as (for a memory with 3 frames):

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

First in First out (FIFO) Algorithm

It is simplest page replacement algorithm. In this algorithm the oldest page is chosen as a victim. A FIFO quest is created to hold all pages in memory. We replace the page at the head of the queue and when a page is brought into memory, we insert it at the tail of the queue.

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*
7	**0**	**1**	**2**	**0**	**3**	**0**	**4**	**2**	**3**	**0**	**3**	**2**	**1**	**2**	**0**	**1**	**7**	**0**	**1**
7	7	7	2		2	2	4	4	4	0			0	0			7	7	7
	0	0	0		3	3	3	2	2	2			1	1			1	0	0
		1	1		1	0	0	0	3	3			3	2			2	2	1

Fig: FIFO page replacement algorithm

There are 15 faults.

Optimal Page (OPT) Replacement Algorithm

It has lowest page fault rate among all f the algorithms. This algorithm states that - "Replace the page that will not be used for the longest period of time" i.e. future knowledge of reference string is required

Working method of optimal page replacement algorithm:

- The first 3 (three) references cause faults that fill the 3 (three) empty frames.
- The reference to page 2 replaces page 7, because 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14.
- The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again.

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*
7	**0**	**1**	**2**	**0**	**3**	**0**	**4**	**2**	**3**	**0**	**3**	**2**	**1**	**2**	**0**	**1**	**7**	**0**	**1**
7	7	7	2		2		2			2			2				7		
	0	0	0		0		4			0			0				0		
		1	1		3		3			3			1				1		

Fig: Optimal page replacement algorithm

There are 9 page faults.

Difference between FIFO and OPT algorithm:

The key distinction between the FIFO and OPT algorithms (other than looking backward versus forward in time) is that

- the FIFO algorithm uses the time when a page was brought into memory
- Whereas the OPT algorithm uses the time when a page is to be used.

Least Recently Used (LRU) Replacement Algorithm

In this technique:

- It is necessary to maintain a linked list of all pages in memory, with the most recently used page at the front and the least recently used page at the rear.
- The page that has *not been used* for the longest period of time is replaced.
- The difficulty is that the list must be updated on every memory reference.
- Finding a page in the list, deleting it, and then moving it to the front is a very time consuming operation

Working method of LRU page replacement algorithm:

- Notice that the first 5 faults are the same as those for optimal replacement.
- When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used.
- When it then faults for page 2, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory.

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*
7	**0**	**1**	**2**	**0**	**3**	**0**	**4**	**2**	**3**	**0**	**3**	**2**	**1**	**2**	**0**	**1**	**7**	**0**	**1**
7	7	7	2		2		4	4	4	0			1		1		1		
	0	0	0		0		0	0	3	3			3		0		0		
		1	1		3		3	2	2	2			2		2		7		

Fig: LRU replacement algorithm

There are 12 page faults.

Not Recently Used (NRU) Page Replacement Algorithm

Two status bit associated with each page. **R** is set whenever the page is referenced (read or written). **M** is set when the page is written to (i.e., modified).

When a page fault occurs, the operating system inspects all the pages and divides them into four categories based on the current values of their R and M bits:

- Class 0: not referenced, not modified.
- Class 1: not referenced, modified.
- Class 2: referenced, not modified.
- Class 3: referenced, modified.

The NRU algorithm removes a page at random from the lowest numbered non-empty class.

Second Chance Page (SCP) Replacement Algorithm

A simple modification to FIFO that avoids the problem of heavily used page. It inspects the R bit if it is 0, the page is both old and unused, and so it is replaced immediately. If the R bit is 1, the bit is cleared, the page is put onto the end of the list of pages, and its load time is updated as though it had just arrived in memory. Then the search continues.

Clock Page (CP) Replacement Algorithm

This algorithm keeps all the page frames on a circular list in the form of a clock, as shown in following figure. The hand of the clock points to the oldest page.

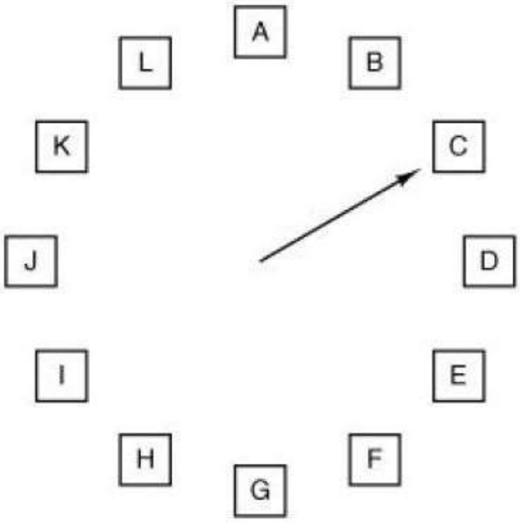


Fig: CP page replacement algorithm

When a page fault occurs, the page being pointed to by the hand is inspected. If its R bit is 0, the page is expelled, the new page is inserted into the clock in its place, and the hand is advanced one position. If R is 1, it is cleared and the hand is advanced to the next page. This process is repeated until a page is found with R = 0.

Segmentation

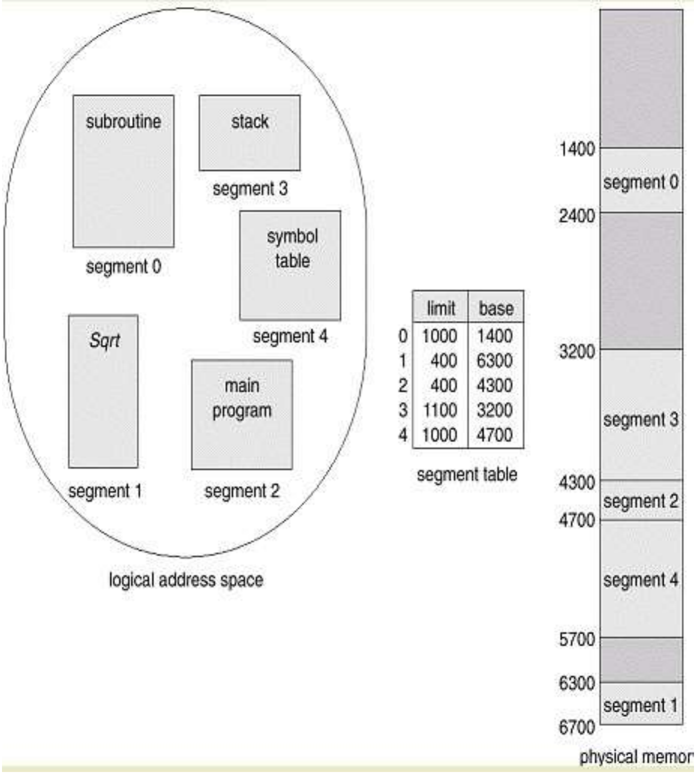
Segmentation:

Segmentation is a memory management method that supports the user view of memory i.e. it is a technique for breaking memory up into logical pieces.

Segmentation is a Memory Management technique in which memory is divided into variable sized chunks which can be allocated to processes. Each chunk is called a segment. Each segment consists of linear sequence of sequence of addresses starting from 0 to maximum value depending upon the size of segment. A table stores the information about all such segments and is called Global Descriptor Table (GDT). A GDT entry is called Global Descriptor. It is the common way to achieve memory protection.

The size of a segment can increase or decrease depending upon the data stored or nature of operation performed on that segment. This change of size doesn't affect other segments in the memory.

Segmentation can be implemented with or without paging



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Fig: Segmentation

Advantages of segmentation:

- Can share data in a controlled way with appropriate protection mechanisms.
- Can move segments independently.
- Can put segments on disk independently.
- No internal fragmentation

Disadvantages of segmentation:

- externa Fragmentation and complicated memory management
- The maximum size of segment is limited by the size of memory.

Implementation of Pure Segmentation

The implementation of segment differs from paging. Paging is fixed size and segment is not. Following figure shows the example of segmentation.

- Initially there are 5 segments.
- Now segment 1 is removed and segment 7 which is smaller is put in the place, as shown in the figure (B), between segment 0 and segment 2 and there is an unused area i.e. holes.
- Then segment 4 is replaced by segment 5 as shown in figure (C).
- Segment 3 is replaced by segment 6 as shown in figure (D).

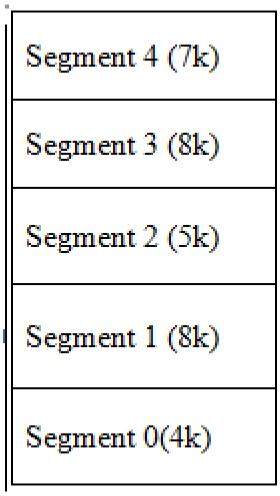


Fig (A)

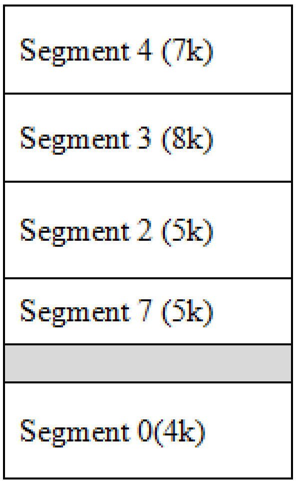


Fig (B)

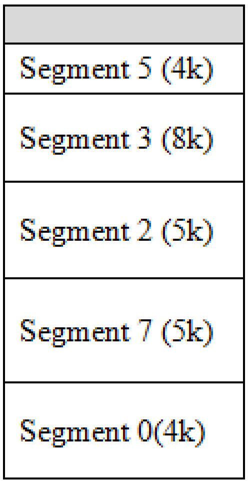


Fig (C)

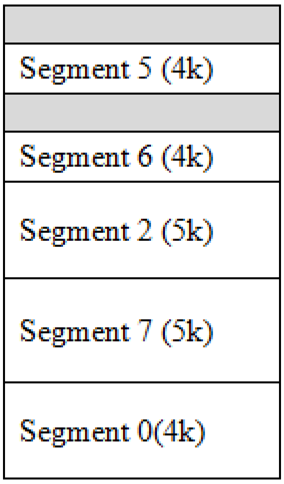


Fig (D)

Comparison between paging and segmentation:

SN	Paging	Segmentation
1	• Block replacement easy • Fixed-length blocks	• Block replacement hard • Variable-length blocks • Need to find contiguous, variable-sized, unused part of main memory
2	Invisible to application programmer	Visible to application programmer.
3	No external fragmentation, But there is Internal Fragmentation unused portion of page.	No Internal Fragmentation, But there is external Fragmentation unused portion of main memory.
4	Units of code and date are broken into separate pages.	Keeps blocks of code or data as a single units
5	segmentation is a logical unit visible to the user's program and id of arbitrary size	paging is a physical unit invisible to the user's view and is of fixed size
6	Segmentation maintains multiple address spaces per process.	Paging maintains one address space.
7	No sharing of procedures between users is facilitated.	sharing of procedures between users is facilitated

Unit 6
Input Output

Introduction

In a computer system there are varieties of input and output device with different speed and function. An input/output management module is responsible to manage and control such types of devices. I/O refers to the communication between information processing system, such as a computer, and the outside world possibly a human, or another information processing system. Inputs are the signals or data received by the system, and outputs are the signals or data sent from it. Input output management is also one of the primary responsibilities of an operating system.

Principles of I/O Hardware

I/O Devices

Input/output device is used to enter information and instructions into a computer for storage or processing and to deliver the processed data to a human operator or, in some cases, a machine controlled by the computer. It is also known as computer peripheral. I/O devices can be roughly divided into two categories:

- Block devices and
- Character devices

Block devices:

A device which stores information in fixed size blocks, each one with its own address is termed as block device. Common block size would be range from 512 bytes to 32,768 bytes. A block device can be read or write independently of all the other ones.
Example: Disks are the most common block devices.

Character devices:

A character device delivers or accepts a stream of characters, without regard to any block structure. It is not addressable and does not have any seek operation.
Example: printer, mouse, etc

Device Controllers

- An electronic device in the form of chip or circuit board that controls functioning of the I/O device is called the *device controller* or *I/O Controller* or *adopter*. The operating system directly deals with the device controller.
- It is the hardware that controls the communication between the system and the peripheral drive unit.
- Any device connected to the computer is connected by a plug and socket, and the socket is connected to a device controller.
- Each device controller has a local buffer and a command register. It communicates with the CPU by interrupts.
- A device's controller plays an important role in the operation of that device; it functions as a bridge between the device and the operating system.
- I-O devices generally contain two parts: *mechanical* and *electrical* part. This electrical part is known as a device controller and can take the form of a chip on personal computers and mechanical part is a device.
- It takes care of low level operations such as error checking, moving disk heads, data transfer, and location of data on the device.
- There are many device controllers in a computer system. Example: serial port controller, SCSI bus controller, disk controller

Function of device controllers:

- Stops and starts the activity of the peripheral device.
- Generate error checking code.
- Checks the error in the data received from the interface.

- Abort that command which have errors.
- Retry the command having an error
- Receives the control signals from the interface unit
- Convert the format of the data
- Check the status of the device.

Memory-mapped I/O

It is the scheme that assigns specific memory location to I/O devices. For example, in memory mapped display text character drives its data from a specific location. Thus communication to and from I/O device can be same as reading and writing to memory address that responds to the I/O devices.

DMA (Direct Memory Access)

DMA Stands for "Direct Memory Access." DMA is a method of transferring data from the computer's RAM to another part of the computer without processing it using the CPU. While most data that is input or output from your computer is processed by the CPU, some data does not require processing, or can be processed by another device. In these situations, DMA can save processing time and is a more efficient way to move data from the computer's memory to other devices.

In other words, direct memory access (DMA) is a method that allows an input/output (I/O) device to send or receive data directly to or from the main memory, bypassing the CPU to speed up memory operations. The process is managed by a chip known as a DMA controller (DMAC).

For example, a sound card may need to access data stored in the computer's RAM, but since it can process the data itself, it may use DMA to bypass the CPU. Video cards that support DMA can also access the system memory and process graphics without needing the CPU

DMA is an essential feature of all modern computers, as it allows devices to transfer data without subjecting the CPU to a heavy overhead. With DMA, the CPU gets freed from this overhead and can do useful tasks during data transfer.

A DMA Controller is a device, which takes over the system bus to directly transfer information from one part of the system to another. The DMA controller is told to make a transfer either by the CPU or some special circumstances; then the DMA controller makes a request to gain control of the bus from the CPU, other processors, or controllers which might currently be using bus; these other devices then relinquish control of the bus by putting their lines into tri-state condition; they then grant the bus to the DMA controller; and finally, the DMA controller takes over the bus, generating its own address and control signals for the bus and causing the transfer of information.

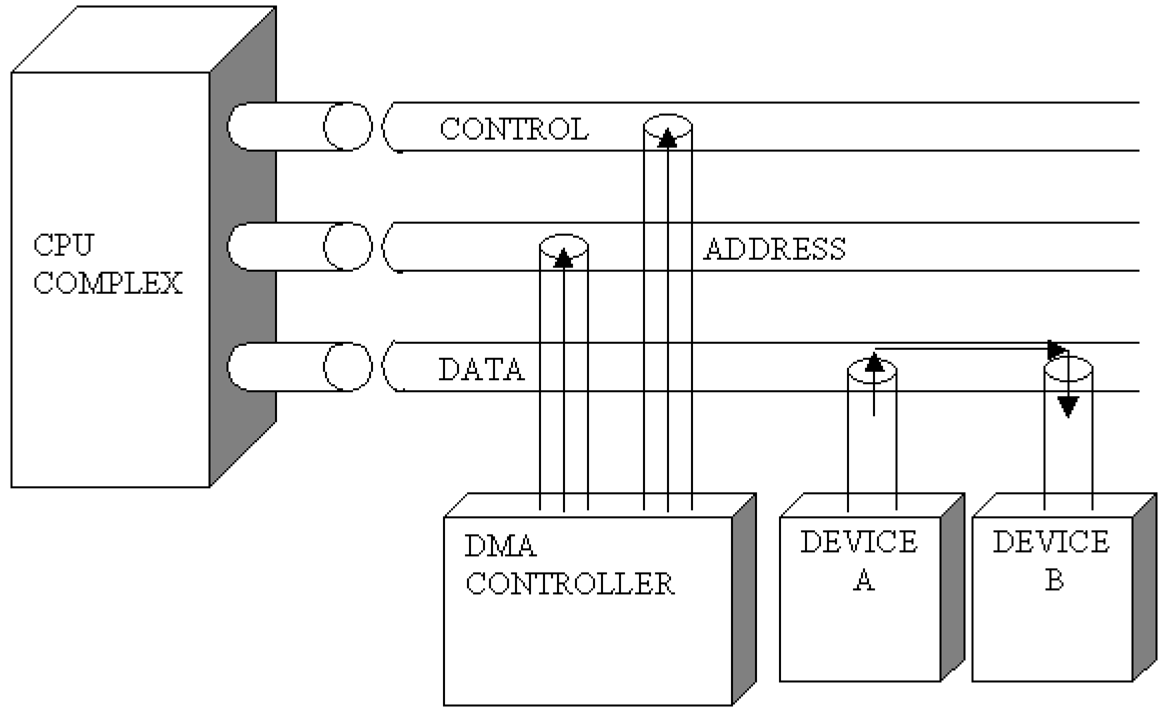


Figure 1: DMA transfer

Working of DMA

- Transfers a block of data directly to or from memory
- An interrupt is sent when the task is complete
- The processor is only involved at the beginning and end of the transfer

Principles of I/O Software

Principle of I/O Software

The basic idea of I/O software is to organize the software as a series of a layer. The lower layer contains the information of the hardware and the upper layer contains the interface to the user. The main concept of I/O software is called as device independence which that the program can be written or match in such a way that it can be read on hard disk without modifying the program for each different deice.

The main important job for I/O software is error handling; it controls the error by using control program (error handling).

The concept of I/O software is sharable and dedicated device. Some I/O devices such as hard disk can be used by multiple users at a same time. No problem is occurs even if multiple user open one particular file at one time.

Some command is used in the I/O software are as follows:

- Polled I/O and Interrupt driven I/O
- Character user interface and graphical user interface

Polled I/O versus Interrupt Driven I/O

Most input and output devices are much slower than the CPU—so much slower that it would be a terrible waste of the CPU to make it wait for the input devices. There are two mechanisms: **Polling** and **Interrupt driven I/O**

Pooling:

Polling means determining the status of an I/O device with direct action taken by the CPU i.e. the mechanism for a CPU to check on the status of an I/O device is by reading a memory address which is associated with an I/O device (using memory mapped I/O) is called *polling*. To poll an I/O device, the CPU typically uses memory-mapped I/O. I/O devices are assigned memory addresses which aren't used by memory at all, but are instead, special addresses that the I/O device recognizes. The CPU can then check the status of the device by reading a byte or word at some pre-determined address for that particular I/O device. Occasionally, polling is the right thing to do, especially if devices are quick enough.

- The main advantage of polling is that the CPU determines how often it needs to poll.
- The problem with polling is that the CPU operates at a much faster speed than most I/O devices. Thus, a CPU can get into a busy wait, checking the device many times, even though the device is very slow.

Example:
Let us consider a game where a basketball player is asked to make as many free-throws as possible in one minute, but the clock is down the hall in another room. The player must run down the hall and check if the minute has passed, and if not, go back to the gym and try to make another shot, then run down the hall to check the clock and run back to take another shot. The player spends much of the time simply checking (polling) the clock.

Interrupts

An interrupt causes the CPU to "stop" and determine what device is interrupting. Here the CPU works on its given tasks continuously. When an input is available, such as when someone types a key on the keyboard, then the CPU is interrupted from its work to take care of the input data. The CPU can work continuously on a task without checking the input devices, allowing the devices themselves to interrupt it as necessary. This requires some extra "smarts" in the form of electronic circuitry at the I/O devices so that they can interrupt the CPU.

Character User Interface and Graphical User Interface

Goals of I/O Software

The main goal of I/O Software has been broadly classified into four layers:

- Interrupt Handler
- Device Driver
- Device dependent operating system software
- User level interface

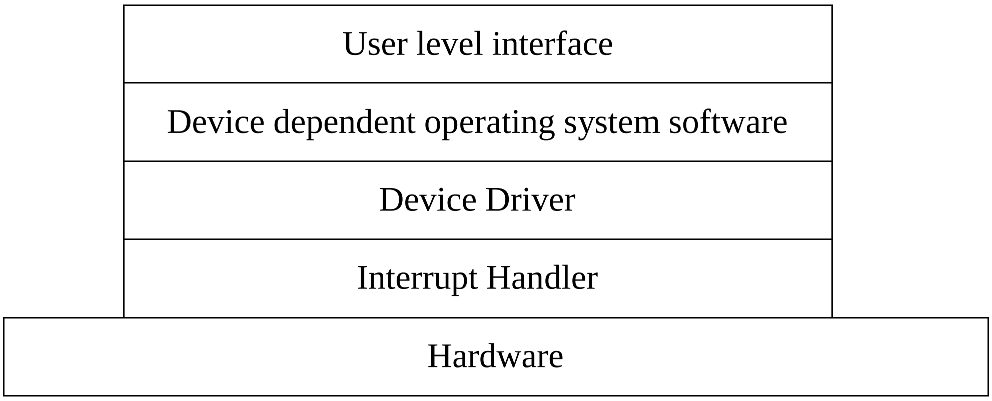


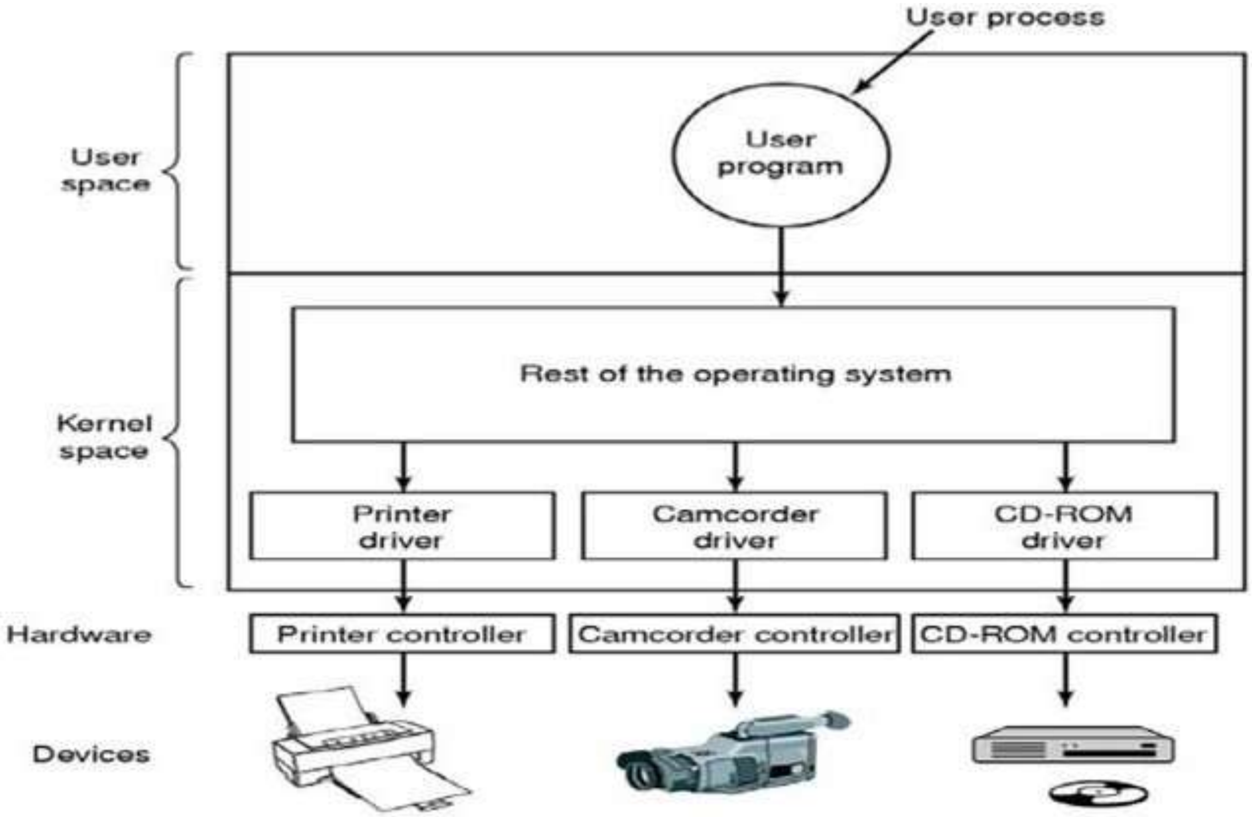
Fig: Layer of I/O software system

1. **Interrupt Handler**
   It is the bottom layer of I/O software. When an event occurs the micro-controller generates a *hardware interrupt*. The interrupt forces the micro-controller's program counter to jump to a specific address in program memory. This special memory address is called the *interrupt vector*. At this memory location we install a special function known as an *interrupt service routine* (ISR) which is also known as an *interrupt handler*. So when a hardware interrupt is generated, program execution jumps to the interrupt handler and executes the code in that handler. When the handler is done, then program control returns the micro-controller to the original program it was executing. So a hardware interrupt allows a micro-controller to interrupt an existing program and react to some external hardware event.

2. **Device Driver**

   A **device driver** is a computer program that operates or controls a particular type of device that is attached to a computer. A driver provides a software interface to hardware devices, enabling operating systems and other computer programs to access hardware functions without needing to know precise details of the hardware being used.

   A driver is a small piece of software that tells the operating system and other software how to communicate with a piece of hardware.



3. **Device independent operating system software**

   Device independence is the process of making a software application is able to function on a wide variety of devices i.e. the capability of a program, operating system or programming language to work on varieties of computers or peripherals, despite their electronic variation.

   Function of device-independent I/O software:
   - Uniform interfacing for device drivers
   - Buffering
   - Error reporting
   - Allocating and releasing dedicate devices
   - Providing a device-independent block size

4. **User level interface**

   It is the upper most layer of I/O system. This layer mainly represents the outermost user interface that occurs between the operating system and end user. A user interface is that portion of an interactive computer system that communicates with the user. Proper design of a user interface can make a significant difference in training time, performance speed, error rates, user satisfaction, and the user's retention of knowledge of operations over time.

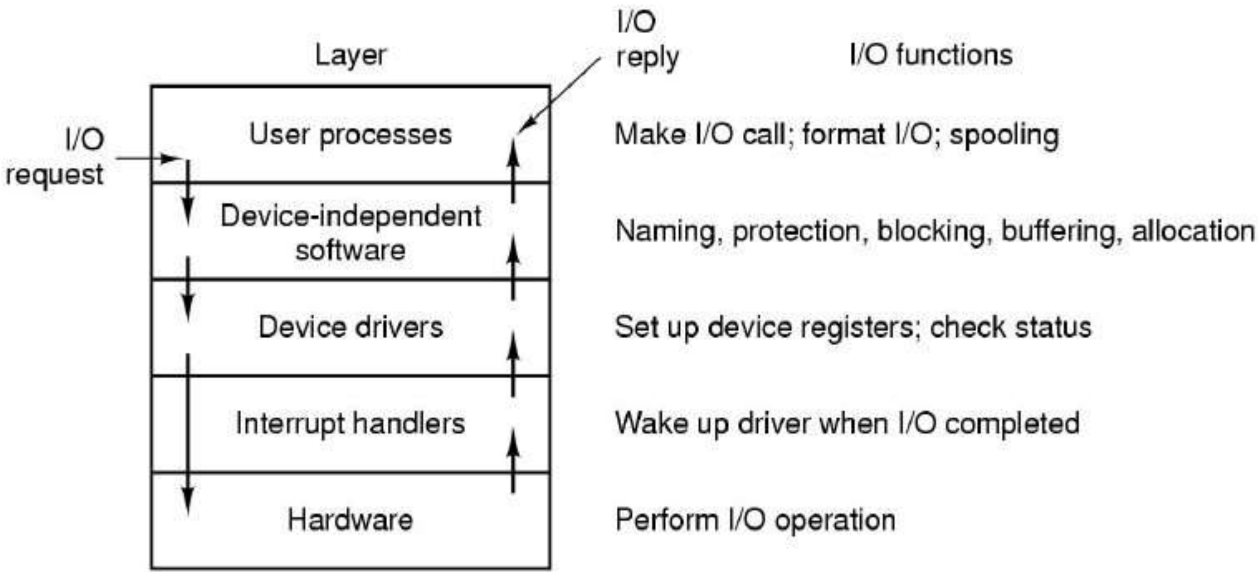

Layer
I/O reply
I/O functions
I/O request
User processes
Make I/O call; format I/O; spooling
Device-independent software
Naming, protection, blocking, buffering, allocation
Device drivers
Set up device registers; check status
Interrupt handlers
Wake up driver when I/O completed
Hardware
Perform I/O operation

Fig: Layer of I/O System

Disk

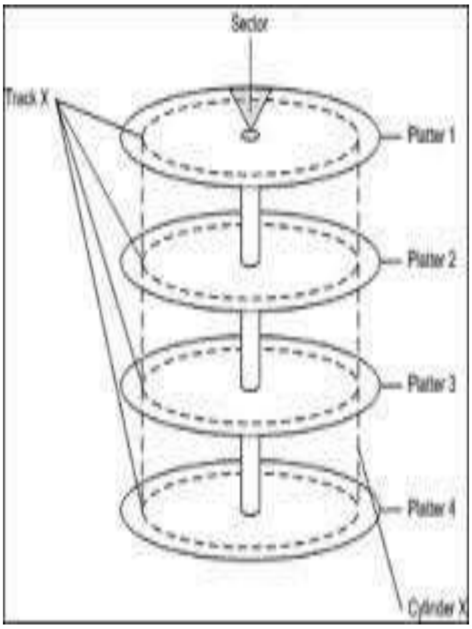
Disk Hardware

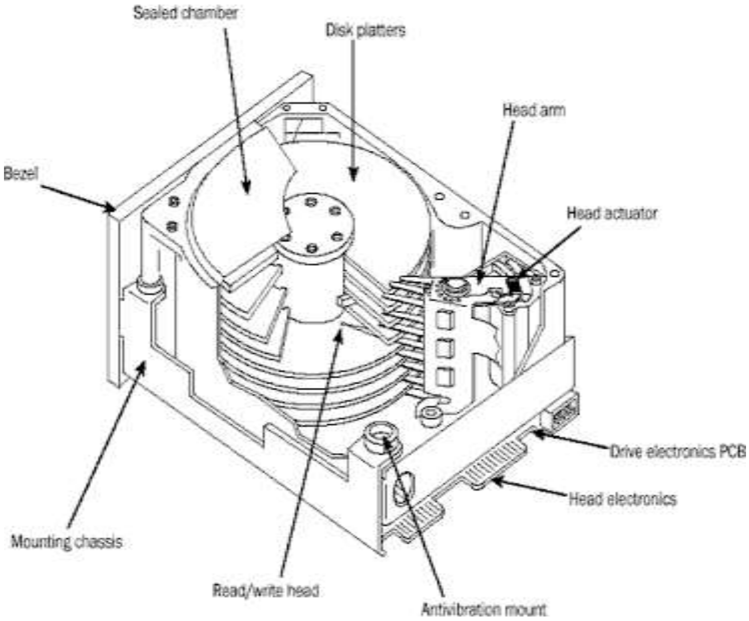
A hard disk is part of a computer system that stores and provides relatively quick access to large amounts of data i.e. a disk drive is a randomly addressable and rewritable storage device. A hard disk is really a set of stacked "disks," each of which has data recorded electromagnetically in concentric circles or "tracks" on the disk. A "head" records (writes) or reads the information on the tracks. Two heads, one on each side of a disk, read or write the data as the disk spins. Each read or write operation requires that data be located, which is an operation called a "seek." A hard disk/drive unit comes with a set rotation speed varying from 4500 to 7200 rpm.

Technical terms related to Disk Drive:

- Platter: Hard drives are normally composed of multiple disks called platters. These platters are stacked on top of each other. It is the platter that actually stores the data. It consists of a substrate coated with magnetic media. The substrate is there to act as a rigid support for the magnetic media. The magnetic layer is protected by a thin layer of carbon and a lubrication layer to prevent damage in case the head comes in contact with the platter surface. Typically, both sides of the platter have magnetic media on which to store data.
- **Spindle/Motor:** The platters are attached at the center to a rod or pin called a spindle that is directly attached to the shaft of the motor that controls the speed of rotation.
- **Head-Actuator Assembly:** This assembly consists of an actuator, the arms, the sliders and the read/write heads. The actuator is the device that moves the arms containing read/write heads across the platter surface in order to store and retrieve information. The head arms move between the platters to access and store data. At the end of each arm is a head slider, which consists of a block of material that holds the head. The read/write heads convert the electronic 0s and 1s in the magnetic fields on the disks.
- **Logic Board:** Logic boards consisting of chips, memory and other components control the disk speed and direct the actuator in all its movements. It also performs the process of transferring data from the computer to the magnetic fields on the disk.
- **Tracks:** A track is a concentric ring on the disk where data is stored.
- **Cylinders:** On drives that contain multiple platters, all the tracks on all the platters that are at the same distance from the center are referred to as a cylinder. The data from all the tracks in the cylinder can be read by simply switching between the different heads, which is much faster than physically moving the head between the different tracks on a single disk.

- **Sectors:** Tracks are further broken down into sectors, which are the smallest units of storage on a disk, typically 512 bytes. A larger number of sectors are recorded on the outer tracks, and progressively fewer toward the center. Data can also be read faster from the outer tracks and sectors than the inner ones.
- **Clusters:** Sectors are grouped together into clusters. As far as the operating system is concerned, all the sectors in one cluster are a single unit.
- **Extents:** A set of contiguous clusters storing a single file or part of a file is called an extent. It is best to keep the number of extents for any file to a minimum, as each extent requires a separate input/output (I/O) operation. Reducing the number of extents in a file is achieved using a process known as defragmentation. This greatly improves performance.
- The s***eek time*** is the time for the disk arm to move the heads to the cylinder containing the desired sector.
- The ***rotational latency*** is the additional time for the disk to rotate the desired sector to the disk head.
- The ***disk bandwidth*** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.





Disk Management is used to manage the drives (hard disk drives, optical disk drives, flash drives) installed in a computer. Disk Management can be used to partition drives, format drives, assign drive letters, delete partition, and much more.

Disk Scheduling (Disk Arm scheduling algorithms)

The major responsibility of an OS is also to use the hardware efficiently. For a multiprogramming system with many processes, the disk queue may often have several pending requests. Thus, when one request is completed, the OS chooses which pending request to service next. Such decision is made by *disk scheduling algorithm*. Using such scheduling algorithms we can improve both the access time and bandwidth of servicing disk I/O request in a good order.

There are various disk scheduling algorithms and some of them are:

- FCFS (FIFO)
- SSTF (Shortest Seek Time First) or CCF (Closet Cylinder First)
- SCAN algorithm
- C-SCAN algorithm (Circular SCAN)

1. FCFS (FIFO) (First Come First Serve)

- The simplest form of disk scheduling is the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service.
- Consider, for example, a disk queue with requests for I/O to blocks on cylinders in that order;
  `98,  183, 37,  122, 14,  124, 65,  67`

Working:

- ➢ If the disk head is initially at cylinder 53,
- ➢ It will first move from 53 to 98,
- ➢ then to 183, 37, 122, 14, 124, 65, and finally to 67,
- ➢ For a total head movement of 640 cylinders.
- ➢ This schedule is diagrammed in

- The wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule.
- If the requests for cylinders 37 and 14 could be serviced together, before or after the requests at 122 and 124, the total head movement could be decreased substantially, and performance could be thereby improved.

2. SSTF (Shortest Seek Time First) or CCF (Closet Cylinder First)

- It seems reasonable to service all the requests close to the current head position before moving the head far away to service other requests. This assumption is the basis for the shortest-seek-time-first (SSTF) algorithm.
- The SSTF algorithm selects the request with the minimum seek time from the current head position.
- Since seek time increases with the number of cylinders traversed by the head, SSTF chooses the pending request closest to the current head position.
- For our example request queue
  `53, 65, 67, 37, 14, 98, 122, 124, 183`
- This scheduling method results in a total head movement of only 236 cylinders-little more than one-third of the distance needed for FCFS scheduling of this request queue. This algorithm gives a substantial improvement in performance.
- SSTF scheduling is essentially a form of shortest-job-first (SJF) scheduling; and like SJF scheduling, it may cause starvation of some requests (steady supply of shorter seek time requests).
- Although the SSTF algorithm is a substantial improvement over the FCFS algorithm, it is not optimal. Consider;
  `53, 37, 14, 65, 67, 98, 122, 124, 183`
- This strategy reduces the total head movement to 208 cylinders.

3. SCAN algorithm

- In the SCAN algorithm, the disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk.
- At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk.
- The SCAN algorithm is sometimes called the elevator algorithm, since the disk arms behaves just like an elevator in a building, first servicing all the requests going up and then reversing to service requests the other way.
- For our example request queue, we need to know the direction of head movement in addition to the head's current position, 53
- 37, 14, 65, 67, 98, 122, 124, 183

4. C-SCAN algorithm (Circular SCAN)

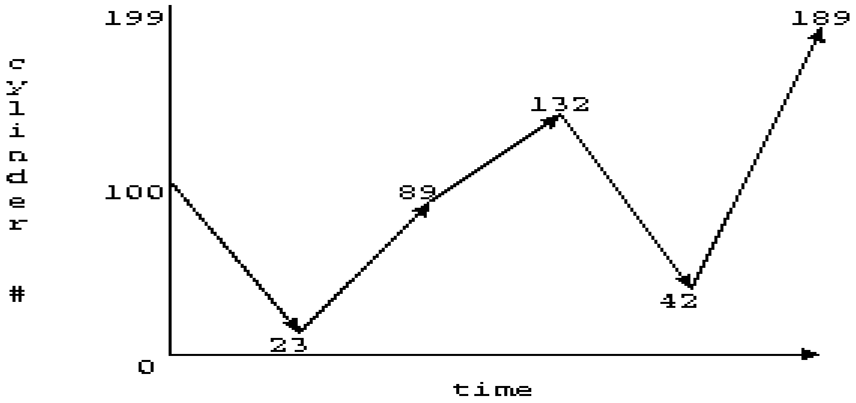
- Circular SCAN (C-SCAN) scheduling is a variant of SCAN designed to provide a more uniform wait time.
- Like SCAN, CSCAN moves the head from one end of the disk to the other, servicing requests along the way.

- When the head reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip.

Examples of Disk Scheduling Algorithms

Work Queue: 23, 89, 132, 42, 187. There are 200 cylinders numbered from 0 - 199 and the disk head starts at number 100

FCFS



Total time is estimated by total arm motion

= | 100 – 23 | + | 23 – 89 | + | 89 – 132 | + | 132 – 42 | + | 42 – 187 |

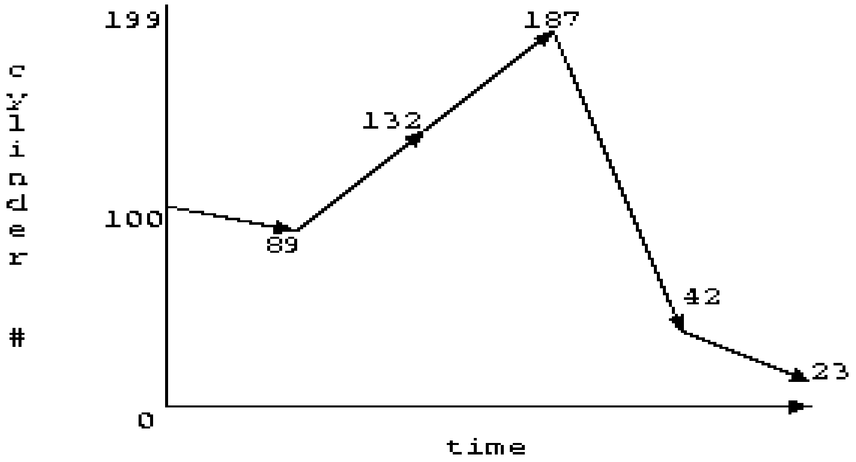
= 77 + 66 + 43 + 90 + 145

= 421

Average time = total time / number of seek

= 421 / 5 = 84.20

SSTF



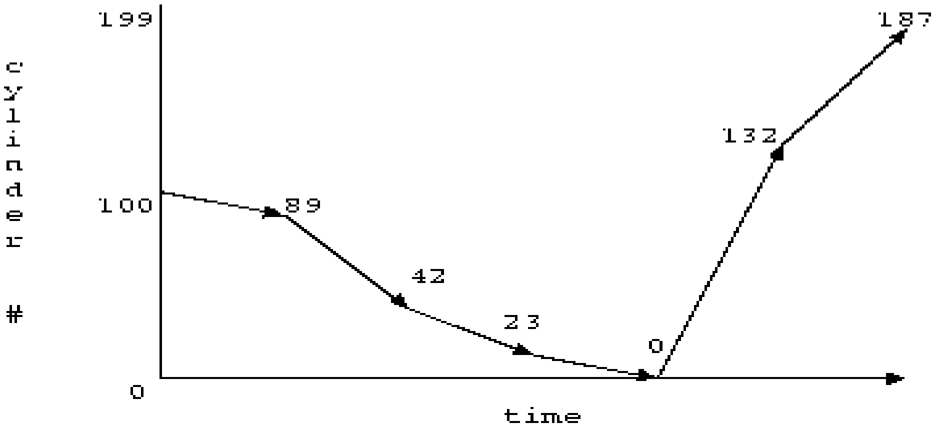
Total time is estimated by total arm motion

= | 100 – 89 | + | 89 – 132 | + | 132 – 187 | + | 187 – 42 | + | 42 – 23 |

= 11 + 43 + 55 + 145 + 19

= 273

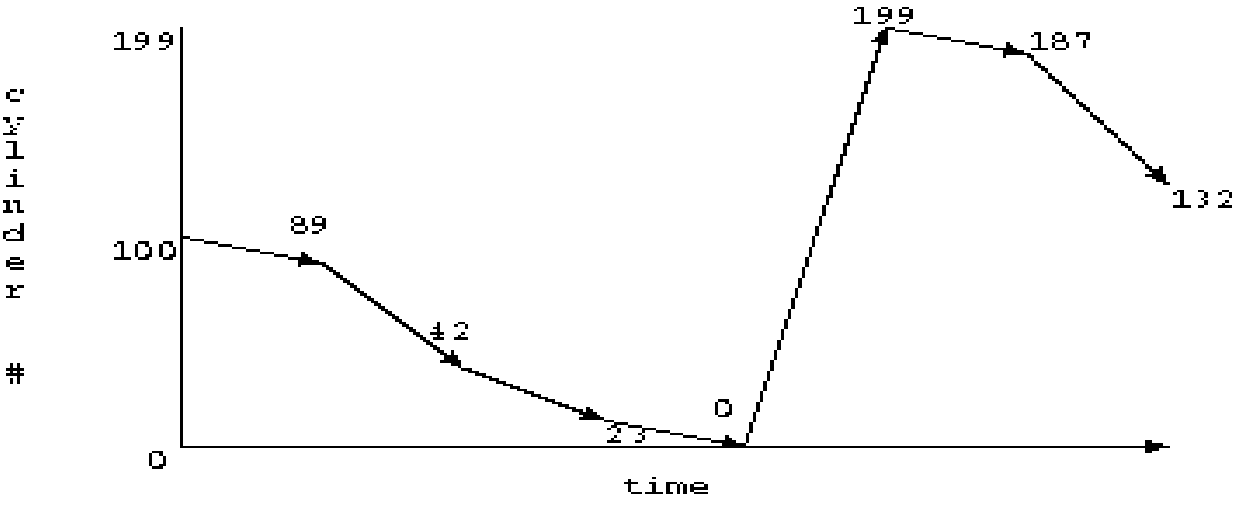
SCAN: assume we are going inwards (i.e., towards 0)



Total time is estimated by total arm motion

= | 100 – 89 | + | 89 – 42 | + | 42 – 23 | + | 23 – 0 | + | 0 – 132 + | 132 – 187 |
= 11 + 47 + 19 + 23 + 132 + 55
= 287

C-SCAN



Total time is estimated by total arm motion

= | 100 – 89 | + | 89 – 42 | + | 42 – 23 | + | 23 – 0 | + | 0 – 199 + | 199 – 187 | + | 187 – 132 |
= 11 + 47 + 19 + 23 + 199 + 12 + 55
= 366

Seek Time, Transfer Time:

When the disk drive is operating, the disk is rotating at constant speed. To read or write, the head must be positioned at the desired track and at the beginning of the desired sector on that track. Track selection involves moving the head in a moveable-head system, the time it takes to position the head at the track is known as **seek time.** In either case, once the track is selected, the disk controller waits until the appropriate sector rotates to line up with the head. The time it takes for the beginning of the sector to reach the head is known as **rotational delay** or **rotational latency.** The sum of seek time if any and the rotational delay is the **access time**, the time it takes to get into position to read or write.

Once the head is in position, the Read or Write operation is then performed as the sector moves under the head; this is the data transfer portion of the operation.

Seek time:

Seek time is the time required to move the disk arm to the required track. The seek time consists of two key components: the initial startup time and the time taken to traverse the cylinders that have to be crossed once the access arm is up to speed. We can approximate seek time with the linear formula

$$T_s = m \times n + s$$

Where,
T_s = estimated seek time
n = number of tracks traversed
m = constant that depends on the disk drive
s = startup time

For example, an expensive Winchester disk on a personal computer might be approximated by `m = 0.3 ms` and `s = 20 ms`, whereas a larger, more expensive disk drive might have `m = 0.1 ms` and `s = 3 ms`.

Rotational Delay:
Disk, other than floppy disk, typically rotate at 3600 rpm, which is one revolution per 16.7 ms. Thus on the average rotational delay will be 8.3 ms. Floppy disks rotate much more slowly, typically between 300 and 600 rmp. Thus the average delay will be 100 and 200 ms.

Transfer Time:
The transfer time to or from the disk depends on the rotation speed of the disk in following fashion.

$$T = \frac{b}{rN}$$

Where, T = transfer time
b = number of bytes to be transferred
N = number of bytes on a track
r = rotation speed in revolution per second

Thus, the total average access time can be expressed as

$$Ta = Ts + \frac{1}{2r} + \frac{b}{rN}$$

Where, T_s is the average seeking time.

Unit 7
File System

Introduction: File System

A file is a collection of related information that is recorded on secondary storage. Commonly, file represents programs and data. Data files may be numeric, alphabetic or binary. In general, a file is a sequence of bits, bytes, lines or records. There are many different types of files: *data files, text files, program files, directory files*, and so on. Different types of files store different types of information. For example, program files store programs, whereas text files store text.

A **file system** is used to control how data is stored and retrieved. The file system consists of two distinct parts:

- A collection of ***files***, each storing related data and
- A ***directory structure***, which organizes and provides information about all the files in the system.

The file system permits users to create data collections, called files, with desirable properties, such as

- **Long-term existence:** Files are stored on disk or other secondary storage for long term and do not disappear when a user logs off.
- **Sharable between processes:** Files have names and can be shared among different processes with associated access permissions that permit controlled sharing.
- **Structure:** Depending on the file system, a file can have an internal structure that is convenient for particular applications. In addition, files can be organized into hierarchical or more complex structure to reflect the relationships among files.

A **file management system** is that set of system software that provides services to users and applications related to the use of files i.e. a user or application may access files through the file management system. It has following objectives:

- To meet the data-management needs and requirements of the user, which includes storage of data
- To optimize performance, both from the system point of view (through put) and the users point of view (response time)
- To provide I/O support for a varieties of storage device.
- To minimize or eliminate the potential for lost or destroy of data.
- To provide I/O support for multiple users in case of multiple-user system.

The minimal sets of requirements of file management system are:

- Each user should be able to create, delete and change files.
- May have controlled access to other user's files.
- Control what types of accesses are allowed to the user's files.
- Should be able to restructure the user's files in a form appropriate to the problem
- Should be able to move data between files.
- Should be able to backup and recover the user's files in case of damage.
- Should be able to access the user's files by a symbolic name.

A Window Operating system supports following file systems:

File System	Meaning
FAT	The MS-DOS operating system introduced the File Allocation Table system of keeping track of file entries and free clusters. Filenames where restricted to eight characters with an addition three characters signifying the file type. The FAT tables were stored at the beginning of the storage space.
FAT32	An updated version of the FAT system designed for Windows 98. It supports file compression and long filenames

NTFS	Windows NT introduced the NT File System, designed to be more efficient at handling files than the FAT system. It spreads file tables throughout the disk, beginning at the center of the storage space. It supports file compression and long filenames

File Organization

A file is a collection of data, usually stored on disk. A file enables us to divide our data into meaningful groups, for example, we can use one file to hold all of a company's product information and another to hold all of its personnel information.

The term "file organization" refers to the way in which data is stored in a file and the method by which it can be accessed i.e. File organization refers primarily to the logical arrangement (structuring) of data in a file system. The physical organization of the file on secondary storage depends upon the blocking strategy and the file allocation strategy.

While selecting a file organization, following criteria's is important:

- Rapid access for effective information retrieval
- Ease of update to aid in having up-to-date information.
- Economy of storage to reduce storage capacity.
- Simple maintenance to reduce cost and potential for error
- Reliability to assure confidence in the data.

There are three file organizations: sequential, relative and indexed.

- Sequential
- Relative and
- Indexed

1. **Sequential**

   A sequential file is one in which the individual records can only be accessed sequentially, that is, in the same order as they were originally written to the file. New records are always added to the end of the file. The order of the records is fixed. The records are stored and sorted in physical, contiguous blocks within each block the records are in sequence.

   Records in these files can only be read or written sequentially.

   Three types of sequential file are supported by this COBOL system:

   - Record sequential
   - Line sequential
   - Printer sequential

2. **Relative**

   A relative file is a file in which each record is identified by its ordinal position within the file (record 1, record 2 and so on). This means that records can be accessed randomly as well as sequentially. Because records can be accessed randomly, access to relative files is fast.

   Within a relative file are numbered positions, called *cells*. These cells are of fixed equal length and are consecutively numbered from 1 to *n*, where 1 is the first cell, and *n* is the last available cell in the file. Each cell either contains a single record or is empty. Records in a relative file are accessed according to cell number. A cell number is a record's relative record number; its location relative to the beginning of the file.

3. **Indexed**

   An indexed file is a file in which each record includes a primary key. To distinguish one record from another, the value of the primary key must be unique for each record. Records can then be accessed randomly by specifying the value of the record's primary key. Indexed file records can also be accessed sequentially. As well as a primary key, indexed files can contain one or more additional keys known as alternate keys. The value of a record's alternate key(s) does not have to be unique.

Blocking and Buffering

Block: It is a smallest amount of data that can be read from or written to secondary storage at one time i.e. any chunk of data that can be treated as a unit (for reading, writing, organizing). The process of grouping several components into one block is called **blocking**. **Clustering** is a grouping file components according to access behavior

Factors affecting block size:

- size of available main memory
- space reserved for programs (and their internal data space) that use the files
- size of one component of the block
- characteristics of the external storage device used

File Descriptor (FD)

File descriptor is an abstract indicator for accessing a file i.e. it is an index for an entry in a kernel-resident array data structure containing the details of open files. In other words a file descriptor is an unsigned integer used by a process to identify an open file. In POSIX (an OS) this data structure is called a ***file descriptor table***, and each process has its own file descriptor table. The process passes the file descriptor to the kernel through a system call, and the kernel will access the file on behalf of the process. The process itself cannot read or write the file descriptor table directly.

On Linux, the set of file descriptors open in a process can be accessed under the path:

/proc/PID/fd/

where PID is the process identifier.

In simple words, when we open a file, the operating system creates an entry to represent that file and store the information about that opened file. So if there are 10 files opened then there will be 10 entries in OS (somewhere in kernel). These entries are represented by integers like (0, 1, 2.). This entry number is the file descriptor i.e. it is a unique integer that identifies an open *file* within a process.

File descriptor values: when the shell runs a program, it opens three files with file descriptors 0, 1, and 2. The default assignments for these descriptors are as follows:

0	Represents standard input.
1	Represents standard output.
2	Represents standard error.

These default file descriptors are connected to the terminal, so that if a program reads file descriptor 0 and writes file descriptors 1 and 2, the program collects input from the terminal and sends output to the terminal. As the program uses other files, file descriptors are assigned in ascending order.

File Naming

A filename is a string used to uniquely identify a file stored on the file system of a computer. Files are an abstraction mechanism. They provide a way to store information on the disk and read it back later. This must be done in such a way as to shield the user from the details of how and where the information is stored, and how the disks actually work. Probably the most important characteristic of any abstraction mechanism is the way the objects being managed are named, when a process creates a file; it gives the file a name. When the process terminates, the file continues to exist and can be accessed by other processes using its name.

The name of a file is called **filename**. All files have names. Different operating system imposes different restrictions on filenames. Many system, allow a filename extension that consists of one or more characters following the proper filename. The *filename extension* usually indicates what type of file it is. Every

filename has two parts: *primary filename* and *secondary name* and both are separated by dot (.) symbol. Primary file names are given by the user and the secondary name are optional, generally by the system. The secondary filename indicates the nature of file i.e. indicates the type of file.
Example: balance_sheet.xls, operating_system.doc, etc
Some extension names are listed as follows:

Extension name	Full forms	Meanings
File.doc	Document	Word document file
File.xls	Excel	Excel spreadsheet file
File.ppt	PowerPoint	PowerPoint file
File.bak	Backup	Backup file
File.c		C source Program
File.gif	Graphical interchange format	Photo file
File.jpg	Joint photographic group	Photo file
File.jpeg	Joint photographic expert group	Photo file
File.txt	Text	General Text format files
File.pdf	Portable document format	Portable document format
File.ps	PostScript	PostScript
File.zip		Compressed archive

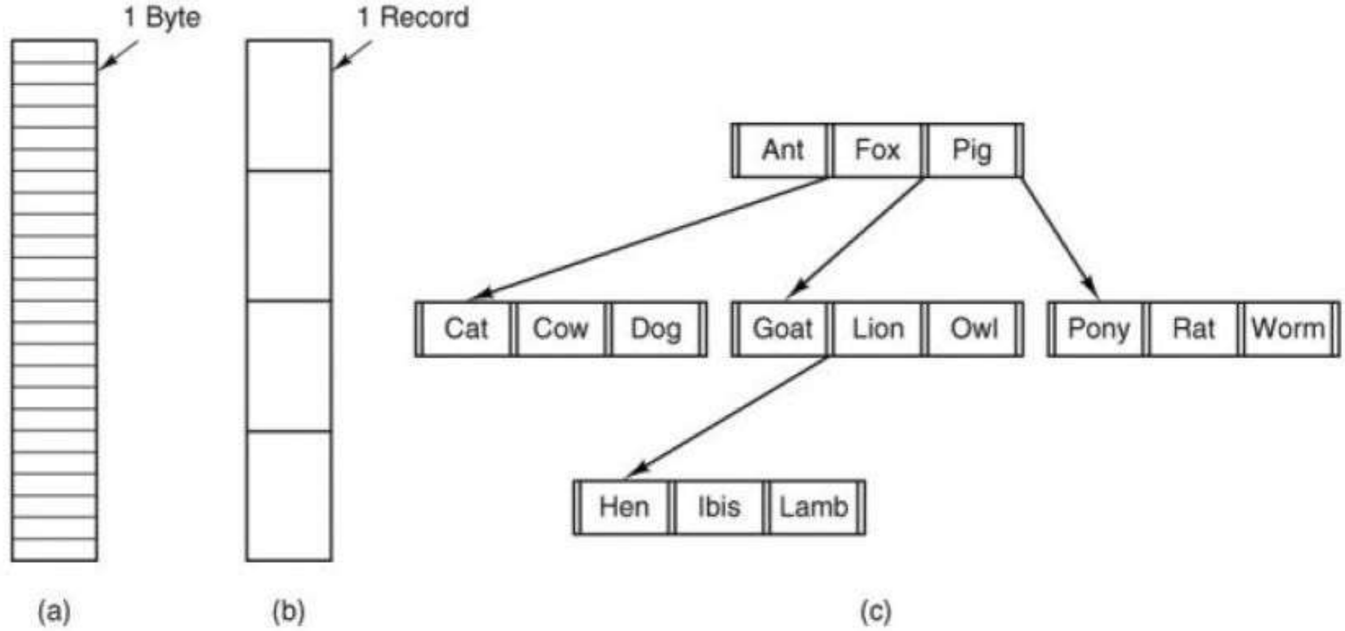
Rules for naming a file:

Most operating systems prohibit the use of certain characters in a filename and impose a limit on the length of a filename. File naming is an important component of file management. Here are the basic rules and recommendations.

- Filename should consist different symbols such as alphabets (Capital or small), digits (0 to 9) or combination of both.
- Only use hyphens and underscores. Avoid any other punctuation marks, accented letters, non-Latin letters, and other non-standard characters such as forward and back slashes, colon, semi-colon, asterisks, angle brackets or brackets.
- Number of characters in a filename (primary name) should not exceed 8 characters (in DOS) or 256 characters (in GUI mode OS). However it should not too length.
- File names should end in a three-letter file extension preceded by a period, such as .CR2, .JPG, .TIFF, etc.

File Structure

The file system structure is the most basic level of organization in an operating system. Almost all of the ways an operating system interacts with its users, applications, and security model are dependent upon the way it organizes files on storage devices.



a) **Byte sequence**
It is just an unstructured sequence of bytes. In effect, the operating system does not know or care what is in the file. All it sees are bytes. Any meaning must be imposed by user-level programs. Both UNIX and Windows 98 use this approach.

b) **Record sequence**
In this model, a file is a sequence of fixed-length records, each with some internal structure. Central to the idea of a file being a sequence of records is the idea that the read operation returns one record and the write operation overwrites or appends one record.

c) **Tree**
In this organization, a file consists of a tree of records, not necessarily all the same length, each containing a key field in a fixed position in the record. The tree is sorted on the key field, to allow rapid searching for a particular key.

File Types

File types refer to classifying the content of the *file*, such as a program, text *file* There are three basic types of files recognized by the system and they are:

- **Regular files**
- **Directory files**
- **Special files**

a) **Regular (ordinary) files**
Regular files are the most common files and are used to contain or store data. Regular files are in the form of text files or binary files. Ordinary File may belong to any type of Application for example notepad, paint, C Program, Songs etc. So all the Files those are created by a user are Ordinary Files. Ordinary Files are used for Storing the information about the user Programs. With the help of Ordinary Files we can store the information which contains text, database, any image or any other type of information.
Text file: Text files are regular files that contain information stored in ASCII format text and are readable by the user and can display and print these files.

Binary files: Binary files are regular files that contain information readable by the computer. Binary files might be executable files that instruct the system to complete a job. Commands and programs are stored in executable, binary files. Special compiling programs are required to translate ASCII text into its equivalent binary code.

b) Directory files

The Files those are Stored into the Particular Directory or Folder. For Example a Folder Name Songs which Contains Many Songs So that all the Files of Songs are known as Directory Files.

Directory files contain information that the system needs to access all types of files, but directory files do not contain the actual file data. As a result, directories occupy less space than a regular file and give the file system structure flexibility and depth. Each directory entry represents either a file or a subdirectory. Each entry contains the name of the file and the file's index node reference number (*i-node number*). The i-node number points to the unique index node assigned to the file. The i-node number describes the location of the data associated with the file. Directories are created and controlled by a separate set of commands.

c) Special files These files are used to model the IO

Types:
Character Special
Files Block Special

The Special Files are those which are not created by the user. Or The Files those are necessary to run a System. The Files those are created by the System. Means all the Files of an operating system or Window, are refers to Special Files. There are Many Types of Special Files, System Files, or windows Files, Input output Files. All the System Files are Stored into the System by using. sys Extension.

File Attributes

Every file has a name and its data. In addition, all operating systems associate other information with each file, for example, the date and time the file was created and the files size, these extra items are called file's **attributes.** Some time it is also known as **metadata**. The list of attributes varies from system to system.

Attribute	Meaning	Attribute	Meaning
Protection	Who can access the files and in what way	Lock flags	0 for unlocked; nonzero for locked
Password	Needed to access the file	Record length	Number of bytes in a record
Creator	ID of the person who created the file	Key position	Offset of the key within each record
Owner	Current owner	Key length	Number of bytes in the key field
Read only flag	flag 0 for read/write; 1 for read only	Creation time	Date and time the file was created
Hidden flag	0 for normal; 1 for do not display in listings	Time of last access	Date and time the file was last accessed
System flag	0 for normal files; 1 for system file	Time of last change	Date and time the file was last changed
Archive flag	0 for has been backed up; 1 for needs to be backed up	Current size	Number of bytes in the file
Random access flag	0 for sequential access only; 1 for random access	Maximum size	Number of bytes in the file may grow to
Temporary flag	0 for normal; 1 for delete file on process exit		

File Operations

Files exist to store information and allow it to be retrieved later. Different systems provide different operations to allow storage and retrieval. The most common system calls relating to files.

1. **Create.** The file is created with no data i.e. an empty file. The purpose of the call is to announce that the file is coming and to set some of the attributes.

2. **Delete**. A file has to be deleted from the system when the file is not needed any more. This will free up disk space.

3. **Open.** Before using a file, a process must open it. The purpose of the open call is to allow the system to fetch the attributes and list of disk addresses into main memory for rapid access on later calls.

4. **Append.** This call is a restricted form of write. It can only add data to the end of the file.

5. **Close.** When all the accesses are finished, the attributes and disk addresses are no longer needed, so the file should be closed to free up some internal table space. Many systems encourage this by imposing a maximum number of open files on processes.

6. **Read.** Data are read from file. Usually, the bytes come from the current position. The caller must specify how much data are needed and must also provide a buffer to put them in.

7. **Write.** Data are written to the file, again, usually at the current position. If the current position is the end of the file, the file's size increases. If the current position is in the middle of the file, existing data are overwritten and lost forever.

8. **Seek**. This approach repositions the file pointer to a specific place in the file i.e. it is a method that specifies from where to take the data in random access files. After this call has completed, data can be read from, or written to, that position.

9. **Get attributes**. Processes often need to read file attributes to do their work. For example, the UNIX make program is commonly used to manage software development projects consisting of many source files. When make is called, it examines the modification times of the entire source and object files and arranges for the minimum number of compilations required to bring everything up to date. To do its job, it must look at the attributes, namely, the modification times.

10. **Set attributes.** Some of the attributes are user settable and can be changed after the file has been created. So this system call provides such facility. Most of the flags also fall in this category. Generally it is to protect a file.

11. **Rename.** It is used to change the name of an existing file. It is done infrequently. It is not always strictly necessary, because the file can usually be copied to a new file with the new name, and the old file then deleted.

12. **Lock.** It is used to lock a file or a part of a file to prevent multiple simultaneous accesses by different process. For an airline reservation system, for instance, locking the database while making a reservation prevents reservation of a seat for two different travelers.

File Access Methods

When a file is used then the stored information in the file must be accessed and read into the memory of a computer system. Various mechanisms are provided to access a file from the operating system.

- Sequential access
- Direct Access
- Index Access

1. **Sequential Access:**
   It is the simplest access mechanism, in which information's stored in a file are accessed in an order such that one record is processed after the other. For example editors and compilers usually access files in this manner.

2. **Direct Access:**

It is an alternative method for accessing a file, which is based on the disk model of a file, since disk allows random access to any block or record of a file. For this method, a file is viewed as a numbered sequence of blocks or records which are read / written in an arbitrary manner, i.e. there is no restriction on the order of reading or writing. It is well suited for Database management System.

3. **Index Access:**
   In this method an index is created which contains a key field and pointers to the various blocks. To find an entry in the file for a key value, we first search the index and then use the pointer to directly access a file and find the desired entry. With large files, the index file itself may become too large to be keeping in memory. One solution is to create an index for the index file. The primary index file would contain pointers to secondary index files, which would point to the actual data items.

File Sharing

File sharing is the practice of sharing or offering access to digital information or resources, including documents, multimedia (audio/video), graphics, computer programs, images and e-books. It is the private or public distribution of data or resources in a network with different levels of sharing privileges.

File sharing is the public or private sharing of computer data or space in a network with various levels of access privilege.
File sharing allows a number of people to use the same file or file by some combination of being able to read or view it, write to or modify it, copy it, or print it. Typically, a file sharing system has one or more administrators. Users may all have the same or may have different levels of access privilege. File sharing can also mean having an allocated amount of personal file storage in a common file system.

File sharing can be done using several methods. The most common techniques for file storage, distribution and transmission include the following:

- Removable storage devices
- Centralized file hosting server installations on networks
- World Wide Web-oriented hyperlinked documents
- Distributed peer-to-peer networks

However there are two main issues while sharing files and they are: ***access rights*** and ***the management of simultaneous access.***

a) **Access Right (Access Privilege)**
   The file system should provide a flexible tool for allowing extensive file sharing among users. There should be a number of options so that the way in which a particular file is accessed can be controlled. Typically, users or group of users are granted certain access rights to a file. Some access rights that can be assigned to a particular user for a particular file are:

   - ***None:*** It means that the user may not even learn the existence of the file i.e. the user is not allowed to read the user directory that includes this file.
   - ***Knowledge:*** It means that a user can determine that the file exists and who is the owner. The user is then able to request the owner for additional access rights.
   - ***Execution:*** The user can load and execute a program but cannot copy it.
   - ***Reading:*** The user can read the file for any purpose, including copying and execution.
   - ***Appending:*** The user can add data to the file, often only at the end, but cannot modify or delete any of the file's contents. This right is useful in collecting data from a number of sources.
   - ***Updating:*** The user can modify, delete, and add to the file's data. Updating normally includes writing the file initially, rewriting it completely or in part, and removing all or a portion of the data.
   - ***Changing Protection:*** The user can change the access rights granted to other users. Typically this right is held only by the owner of the file.
   - ***Deletion:*** The user can delete the file form the file system.

Access can be provided to the following class of the user:

- ***Specific users:*** Individual users who are designated by user ID.
- ***User groups:*** A set of users who are not individually defined.
- ***All:*** All users who have access to this system. These are public files.

b) Simultaneous access:

When access is granted to append or update a file to more than one user, the operating system or file management system must have a mechanism to control the simultaneous access to the file. A brute-force approach is to allow a user to lock the entire file when it is to be updated.

File control Block (FCB)

File control blocks (FCB) are data structures that hold information about a file. When an operating system needs to access a file, it creates an associated file control block to manage the file.
The structure of the file control block differs between operating systems, but most file control blocks include the following parts

- Filename
- Location of file on secondary storage
- Length of file
- Date and time or creation or last access

ACM (Access Control Matrix)

The *access control matrix* is a matrix with each subject represented by a row, and each object represented by a column. An access control matrix is a tool that can describe the current protection state. The access matrix model is the policy for user authentication, and has several implementations such as access control lists (ACLs) and capabilities. It is used to describe which users have access to what objects.

The access matrix model consists of four major parts:

- A list of objects (file, piece of hardware, process)
- A list of subjects (processes and rights)
- A function T which returns an object's type
- The matrix itself, with the objects making the columns and the subjects making the rows

The access controls provided with an operating system typically authenticate principals using some mechanism such as passwords or Kerberos, and then mediate their access to files, communications ports, and other system resources. Their effect can often be modeled by a matrix of access permissions, with columns for files and rows for users. We‘ll write r for permission to read, w for permission to write, x for permission to execute a program, and (–) for no access at all, as shown in Figure below.

	Operating system	**Accounts program**	**Accounting data**	**Audit trail**
Surya	rwx	rwx	rw	r
Daman	x	X	rw	-
Sunita	rx	R	r	r

In above example:

- Surya is the system administrator, and has universal access (except to the audit trail, which even he should only be able to read).
- Daman, the manager, needs to execute the operating system and application, but only through the approved interfaces — he mustn‘t have the ability to tamper with them. She also needs to read and write the data.

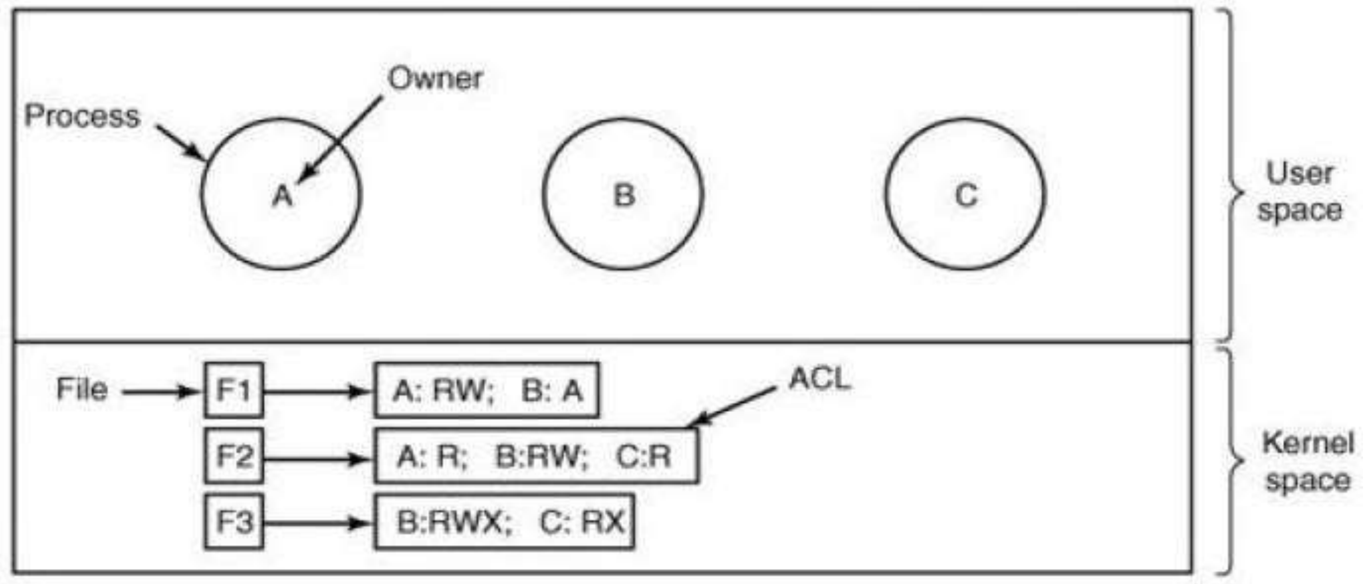
- Sunita, the auditor, can read everything.

Access control matrices (whether in two or three dimensions) can be used to implement protection mechanisms, as well as just model them. But they do not scale well. For instance, a bank with 50,000 staff and 300 applications would have an access control matrix of 15 million entries. This is inconveniently large. It might not only impose a performance problem but also be vulnerable to administrators' mistakes. We will usually need a more compact way of storing and managing this information. The two main ways of doing this are to use groups or roles to manage the privileges of large sets of users simultaneously, or to store the access control matrix either by columns (access control lists) or rows (capabilities, sometimes known as ―tickets‖) or certificates.

ACL (Access Control List)

In multi-user operating systems, files may be accessed by multiple users. Permission rights associated with folders (directories) and files are used to protect or restrict access to files. An ACL consists of a set of ACL entries. An ACL entry specifies the access permissions on the associated object for an individual user or a group of users as a combination of reading write and execute permissions.

In other words, The Access Control List (ACL) is the list of a system object's (file, folder or other network resources) security information that defines access rights for resources like users, groups, processes or devices. The object's security information is known as a permission, which controls resource access to view or modify system object contents. The most common privileges include the ability to read a file (or all the files in a directory), to write to the file or files, and to execute the file. Each ACL has one or more access control entries (ACEs) consisting of the name of a user or group of users. Generally, the system administrator or the object owner creates the access control list for an object.



Each file has an ACL associated with it.

- File F1 has two entries in its ACL (separated by a semicolon). The first entry says that any process owned by user "A" may read and write the file. The second entry says that any process owned by user "B" may read the file. All other accesses by these users and all accesses by other users are forbidden. Note that the rights are granted by user, not by process. As far as the protection system goes, any process owned by user "A" can read and write file F1. It does not matter if there is one such process or 100 of them. It is the owner, not the process ID that matters.

- File F2 has three entries in its ACL: "A", "B", and "C" can all read the file, and in addition B can also write it. No other accesses are allowed.

- File F3 is apparently an executable program, since "B" and "C" can both read and execute it. B can also write it.

Directories

Directory is a location for storing files in our computer i.e. it is a *filing cabinet* of data storage device in which data (file) is grouped and listed in a hierarchical manner for allowing direct user access to any file. It is used to organize files and folders into a hierarchical structure. The directory contains information about the files, including attributes, location, and ownership. It keeps track of file. The directory is itself a file, owned by the operating system and accessible by various file management routines.

A directory contains following information's:

- ***Basic information's:*** file name, file type, file organization
- ***Address information:*** volume (storage device e.g. C:, D:, etc), starting address, size used (current size of the file in bytes, words, or blocks), size allocated (maximum size of the file)
- ***Access control information's:*** Owner, Access information (user name, password), permitted actions (Read, Write, Execute, transmitting over a network i.e. share)
- ***Usage information's:*** date created, identity of creator, date last read access, identity of last reader, date last modified, identity of last modifier, date of last backup, current usage (information about current activity such as process or processes that have the file open, whether it is locked bya a process, etc)

Directory Hierarchy

Files are grouped into directories, and directories are organized in a hierarchy. It is also known as directory tree. At the top of the hierarchy is the –root‖ directory, symbolized by –/‖. It can be shown in following figure.

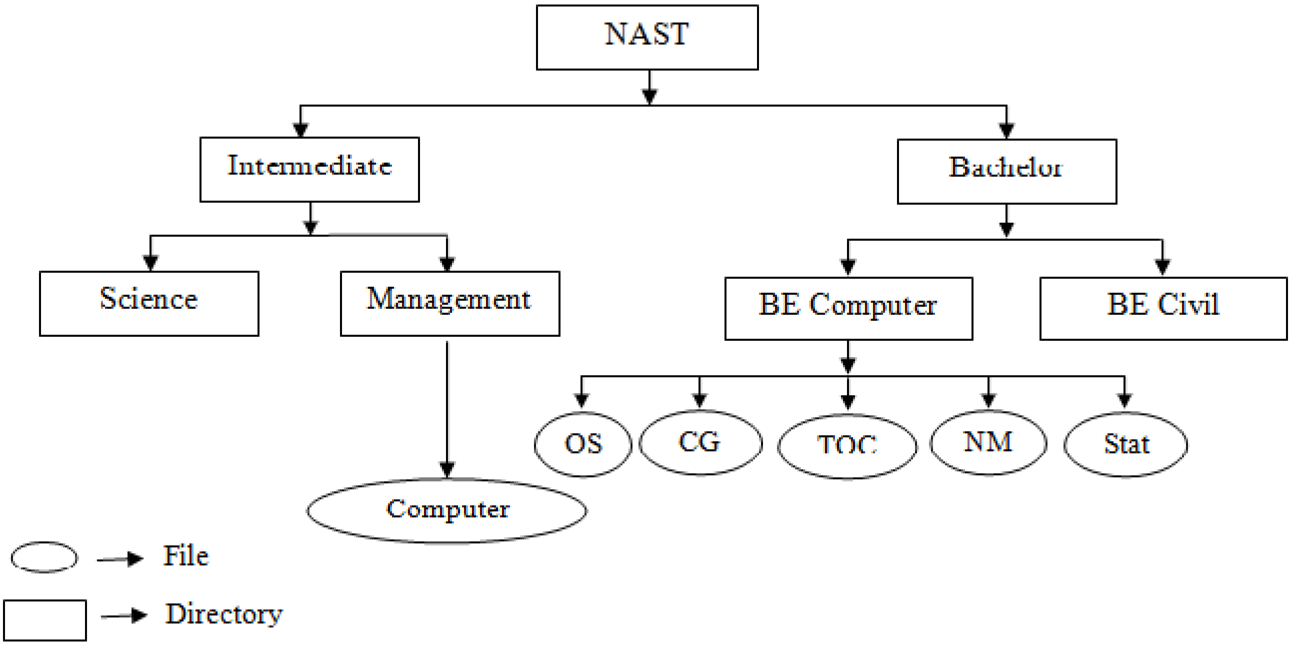


Fig: Directory Hierarchy

In above figure, "NAST" is a root directory which contains two sub directories named: *Intermediate* and *Bachelor*. At the bottom of "BE Computer" directory, there are 5 files: OS, CG, TOC, NM and Stat.

All subdirectory names and file names within a directory must be unique. However, names within different directories can be the same. Creating an arbitrary number of sub directories provides a powerful structuring tool for users to organize their work. The use of tree-structured directory minimizes the difficulty in assigning unique names. Any file in the system can be located by following a path from the root, or master directory down various branches until the file is reached.

Path names:

A path is the route through a file system to a particular file. A pathname (or *path name*) is the specification of that path. Each operating system has its own format for specifying a pathname. The path name of a file describes that file's place within the file-system hierarchy. There are two common methods: absolute path name and relative path name.

- **Absolute path name**
  An absolute pathname is a pathname that starts from the root path and specifies all directories between the root path and the specified directory or file.
  Example: D:\NAST\Bachelor\BE Computer/OS
  It means that the root directory contains a subdirectory *NAST*, which in turn contains a subdirectory *Bachelor*, which in turn again contains another sub directory *BE Computer* which contains the file *OS*.
- **Relative path name**
  A relative pathname is a pathname that specifies some abbreviations for traversing up the directory structure and down to another directory or file in the same directory tree. In most file systems the . . (Two periods) abbreviation indicates "go up one directory level". Relative path do not begin with "\". It specifies location relative to our current working directory and it can be used as a shorter way to specify a file name.
  Example: ..\NAST\Bachelor\BE computer\OS

Directory Operation

The system call form managing directories may vary from system to system. Some of the basic operations on directory are listed below:

1. ***Create:*** When a directory is created, it is empty i.e. it doesn‘t contain any files or sub directories.
2. ***Delete:*** A created directory can be deleted from the system in order to free up the storage. However only empty directory are deleted.
3. ***Open:*** Before a directory can be read, it must be opened, it should be opened. A listing program opens the directory and the files inside it.
4. ***Close:*** When a directory has been read, it should be closed to free up internal table space.
5. ***Read:*** This call returns the next entry in an open directory.
6. ***Rename:*** It can be renamed like as files, because in many cases it is assumed as a file
7. ***Link:*** Linking is a technique that allows a file to appear in more than one directory. This system call specifies an existing file and a path name, and creates a link from the existing file to the name specified by the path. In this way, the same file may appear in multiple directories.
8. ***Unlink:*** A directory entry is removed. If the file is unlinked then it only present in one directory i.e. normal case.

File System Implementation

Contiguous Allocation

It is simplest allocation scheme to store each file as a contiguous run of the disk blocks. It requires each file to occupy a set of contiguous addresses on a disk. It sores each file as a contiguous run of disk blocks. Thus on a disk with 1-KB blocks, a 50-KB file would be allocated 50 consecutive blocks. Both sequential and direct access is supported by the contiguous allocation method.

Advantages:

Contiguous disk space allocation has two significant advantages.

- **It is simple to implement** because keeping track of where a file's blocks are is reduced to remembering two numbers: the disk address of the first block and the number of blocks in the file. Given the number of the first block, the number of any other block can be found by a simple addition.
- **The read performance is excellent** because the entire file can be read from the disk in a single operation. Only one seek is needed (to the first block). After that, no more seeks or rotational delays are needed so data come in at the full bandwidth of the disk.

Thus contiguous allocation is simple to implement and has high performance.

Disadvantage:

- In time, the disk becomes fragmented, consisting of files and holes. It needs compaction to avoid this.

Example of contiguous allocation: CD and DVD ROMs

Linked List Allocation

This method keeps each file as a linked list of disk blocks as shown in the following figure. The first word of each block is used as a pointer to the next one. The rest of the block is for data.

Advantage:

- Unlike contiguous allocation, every disk block can be used in this method. No space is lost to disk fragmentation.

Disadvantage:

- It can be used only for sequential access files. To find the ith block of a file, we must start at the beginning of that file, and follow the pointers until we get the ith block. It is inefficient to support direct access capability for linked allocation of files.
- Another problem of linked list allocation is reliability. Since the files are linked together with the pointer scattered all over the disk. Consider what will happen if a pointer is lost or damaged.

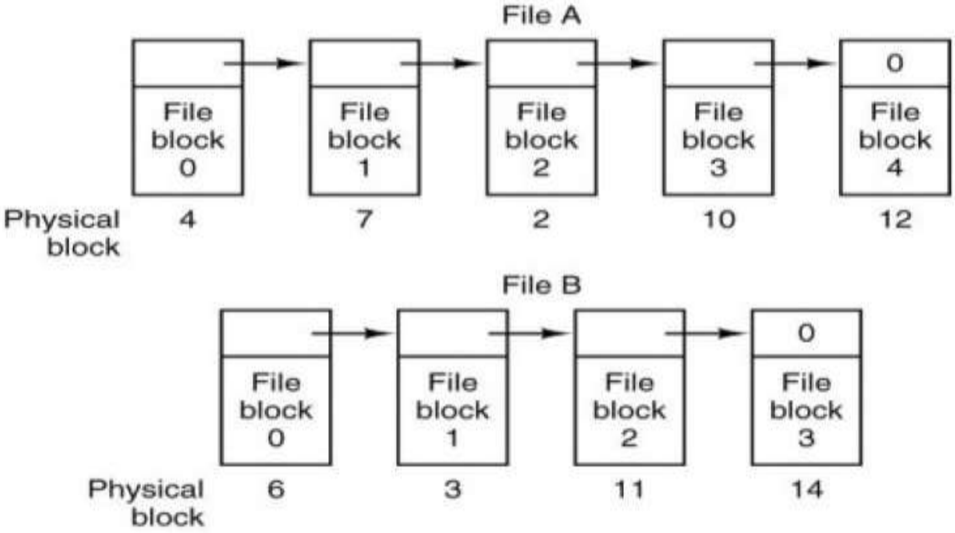
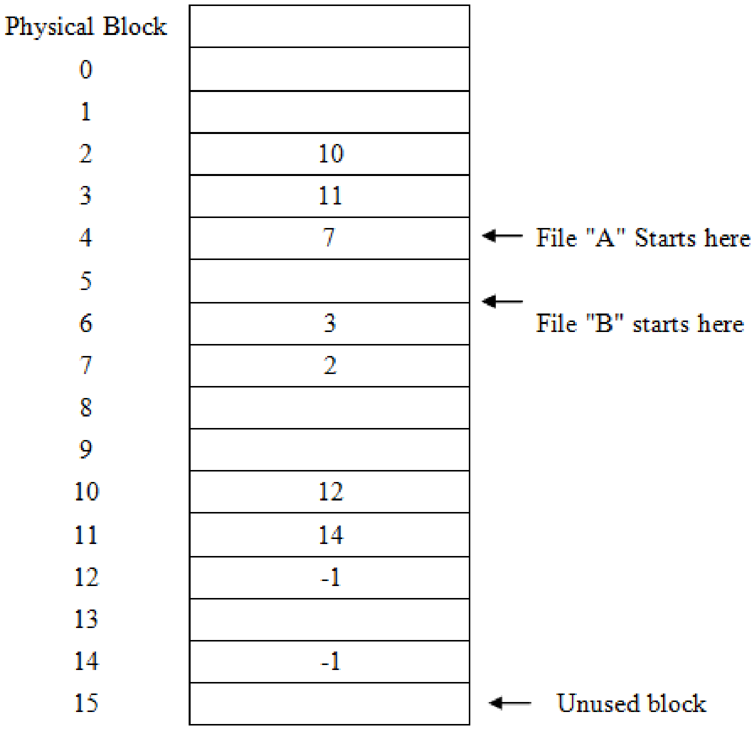


Fig: Storing a file as a linked list of disk blocks.

Linked List allocation using a table in memory:

File "**A**" uses disk blocks 4, 7, 2, 10, and 12 and file **"B"** uses disk blocks 6, 3, 11, and 14, in order. Using the following table we can start with block 4 and follow the chain all the way to the end. Same process is done with block 6 (File "B"). Both chains are terminated with a special marker (e.g. -1) that is not a valid block number. Such table in the main memory is called FAT (File Allocation Table).



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

Fig: Linked list allocation using a FAT in main memory

I-nodes

It is one of the modern method for keeping tracks of each block belongs to which file is to associate with each file within table, called i-node table. I-node is a unique number given to a file in an operating system i.e. it is a data structure that stores all the information's about a file except its name and its actual data. When a file is created both name and an I node number is assigned to it, which is unique within the file system. Both name and their corresponding I node numbers are stored as entries in their directory that appears to the user to contain the file i.e. the directory associates file names and i-nodes.

Whenever a user or a program refers to a file by name, the operating system uses that name to look up the corresponding I node, which then enables the system to obtain the information it needs about the file to perform further operations.

A i-node contains following metadata:

- Size of file (in bytes) and its physical location
- The file's owner and group
- The file's access permissions (read, write, execute)
- Timestamps telling when the i-node was created, last modified and last accessed and
- A reference count telling how many hard links points to the i-node.

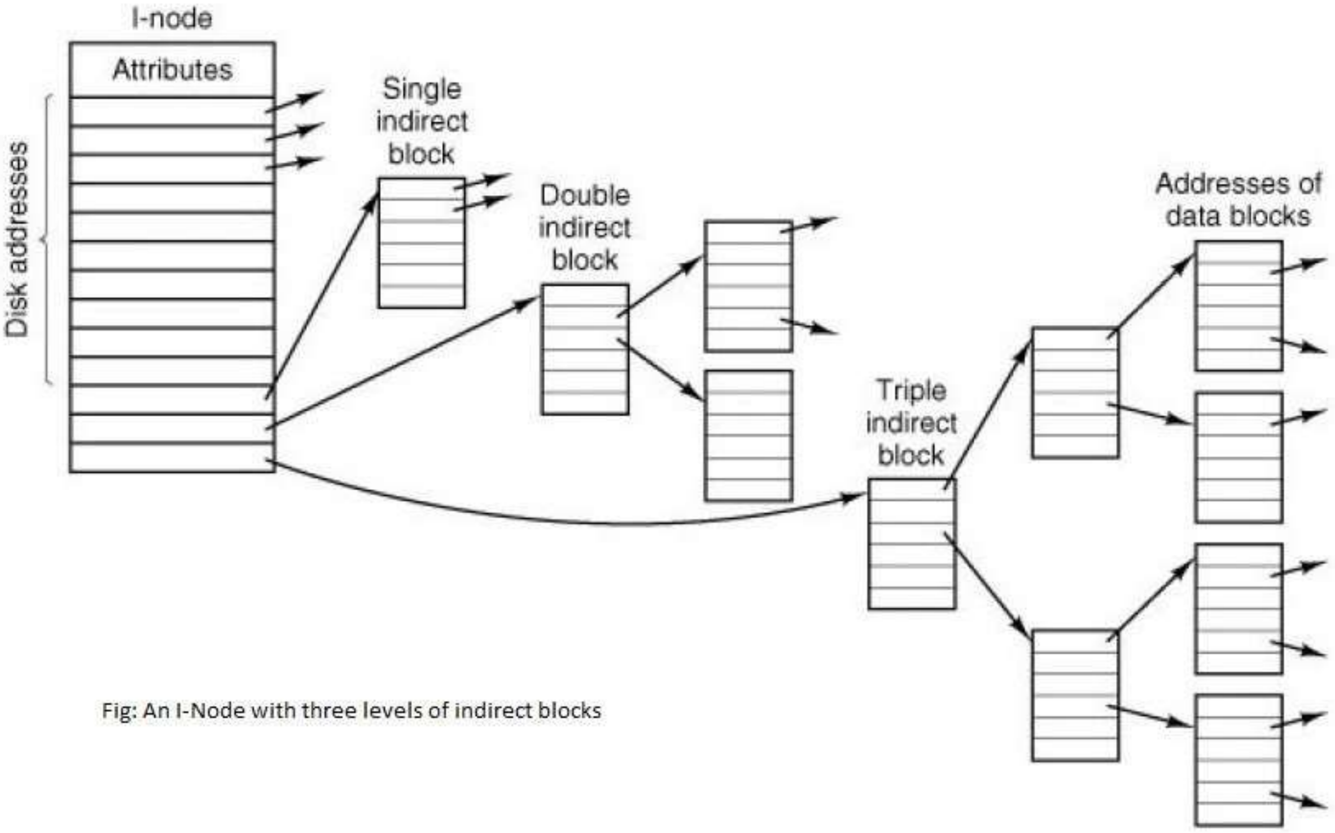


Fig: An I-Node with three levels of indirect blocks

It solves the external fragmentation and size declaration problems of contiguous allocation.

Security and Multi-media Files

Security

Files are securable objects, access to them is regulated by the access-control model that governs access to all other securable objects in Windows.

Files with sensitive information are stored everywhere these days. Servers, workstations, backup tapes, USB drives, laptops etc, they all contain sensitive and mission critical information. No file is secure by default, and it usually sits on a user‘s workstation or USB drive, waiting to be used. What if that media was lost or stolen? The information on that media is then compromised and can severely damage the company. With federal laws and regulations in place, it would be quite expensive too.

File encryption: has been around for as long as computers have. Yet, the use of file encryption almost never goes beyond the very large corporations that have many pending patents, a large database of customer & client information, etc. Regardless of the size of your organization, your information is your most valuable asset. If you lose or compromise that information, it could damage your company severely.

Multi-media files

A file that is capable of holding two or more multimedia elements such as text, audio, video, image, animations. Multimedia is usually recorded and played, displayed, or accessed. Multimedia presentations may be viewed by person on stage, projected, transmitted, or played locally with a media player.

Multimedia data and information must be stored in a disk file using formats similar to image file formats. Multimedia formats, however, are much more complex than most other file formats because of the wide variety of data they must store. Such data includes text, image data, audio and video data, computer animations, and other forms of binary data, such as Musical Instrument Digital Interface (MIDI), control information, and graphical fonts. Typical multimedia formats do not define new methods for storing these types of data. Instead, they offer the ability to store data in one or more existing data formats that are already in general use.

Some extension of media files:

.asx, .wax, .wvx, .wmx, mp3, mp4, etc

Security in Windows 2000

Windows 2000 is an operating system for use on both client and server computers. Windows 2000 security is based on a simple model of authentication and authorization. *Authentication* identifies the user when s/he logs on and makes network connections to services. Once identified, the user is authorized access to a specific set of network resources based on permissions. *Authorization* takes place through the mechanism of access control, using entries stored in Active Directory as well as access control lists that define permissions for objects including printers, files, and network file and print shares.

The key features of Windows 2000 distributed security services are:

- Smart Card authentication support: This card stores a user's public key, private key, and certificate. These cards are a secure way to protect and control a user's keys, instead of storing them on a computer.
- Kerberos for network authentication: It is the primary security protocol that allows users to use a single logon to access all resources. The protocol verifies both the identity of the user and the integrity of the session data.
- File encryption and certificate services
- Strong user authentication and authorization.
- Secure communications between internal and external resources.
- The ability to set and manage required security policies.
- Automated security auditing.
- Interoperability with other operating systems and security protocols.
- An extensible architecture to support application development that uses Windows 2000 security features.

To immediately secure your Windows 2000 system, we have to take following three steps:

- **Install anti-virus software**
  We have to install third party antivirus software.

- **Install a personal firewall**
  A personal firewall protects our machine against Internet attacks and random network scans.

- **Run Windows Update and Enable Automatic Updates**
  We should run Windows Update on our system to install all Critical and Recommended update

Unit 8
Distributed Operating System

Introduction

Distributed system is 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 networks, such as high speed buses or telephone lines. The processors in a distributed system may vary in size and function. These processors are also known as *sites*, *nodes*, *computers*, *machines*, and *hosts*.

Distributed Operating System is a model where distributed applications are running on multiple computers linked by communications. A distributed operating system is an extension of the network operating system that supports higher levels of communication and integration of the machines on the network.
Example: Let us suppose a large bank with hundreds of branch offices all over the world. Each office has a master computer to store local accounts and handle local transactions. In addition, each computer has the ability to talk to all other branch computers and with a central computer at headquarters. If transactions can be done without regard to where a customer or account is, and the users do not notice any difference between this system and the old centralized mainframe that it replaced, it too would be considered a distributed system.

Advantages and Disadvantages of Distributed Operating System

Advantages:

- ***Resource sharing:***
  A distributed operating system provides mechanism for sharing files at remote sites, processing information in a distributed database, printing files at remote sites, using remote specialized hardware devices, and performing other operations. In such system a number of different sites are connected to one other. So a user at one site may be able to use the resources available at another. Example, a user at site "A" may be using a laser printer located at another site "B". Similarly a user at "B" may access a file that resides at "A".

- ***Computation speedup:***
  Distributed system can perform load sharing. In such system a particular computation can be partitioned into sub computations that can run concurrently and then it allows us to distribute the sub computations among the various sites to run concurrently. Thus this system increases computation speed.

- ***Reliability:***
  In such system, if one site in the distributed system fails then the remaining sites can continue operating their task.
  The failure of a site must be detected by the system and appropriate action may be needed to recover from the failure. If the function of the failed site can be taken over by the other site, the system must ensure that the transfer of function occurs correctly. Finally, when the failed site recovers or repaired, mechanism must be available to integrate it back into the system smoothly.

- ***Communication:***
  In this system several sites are connected through communication networks through which users can exchange information's. At a low level, messages (such as file transfer, remote login, mail, remote procedure calls, etc) are passed between systems.

Disadvantages:

- ***Complexity:*** It is complex system software from the user, designers view. It is harder to understand what will happen at any given case and even harder to design software to handle even understood complexities. It also requires more complex synchronization.

- ***Difficulties of allocating resources:*** In this system, local machine may have inadequate resources for a task. It may arise resource conflicts while using remote resources.

- ***Security:*** There is no centralized control. So there is lack of security problem.

- ***Heterogeneity Problems:*** Such system consists of different kinds of heterogeneous systems (hardware and software). So there might be problem with data formats, executable formats, software versioning, and different operating system.

Hardware and Software Concepts

		Data Stream	
		Single	Multiple
Instruction Stream	Single	**SISD**	**SIMD**
	Multiple	**MISD**	**MIMD**

Flynn proposed the following categories of computer systems:

1. **Single instruction single data (SISD) stream:** A single processor executes a single instruction stream to operate on data stored in a single memory. All traditional uni-processor computers fall in this category, from personal computers to mainframes.

2. **Single instruction multiple data (SIMD) stream:** This type refers to array processors with one instruction unit that fetches an instruction, and then commands many data units to carry it out in parallel, each with its own data. These machines are useful for computation that repeat the same calculation on many sets of data. Vector and array processors fall into this category.

3. **Multiple instruction single data (MISD) stream:** A sequence of data is transmitted to a set of processors, each of which executes a different instruction sequence. This structure has never been implemented. No known computers fit in this model.

4. **Multiple instruction multiple data (MIMD) stream**: A set of processors simultaneously execute different instruction sequences on different data sets, which essentially means a group of independent computers, each with its own program counter, program and data. All distributed systems are MIMD.

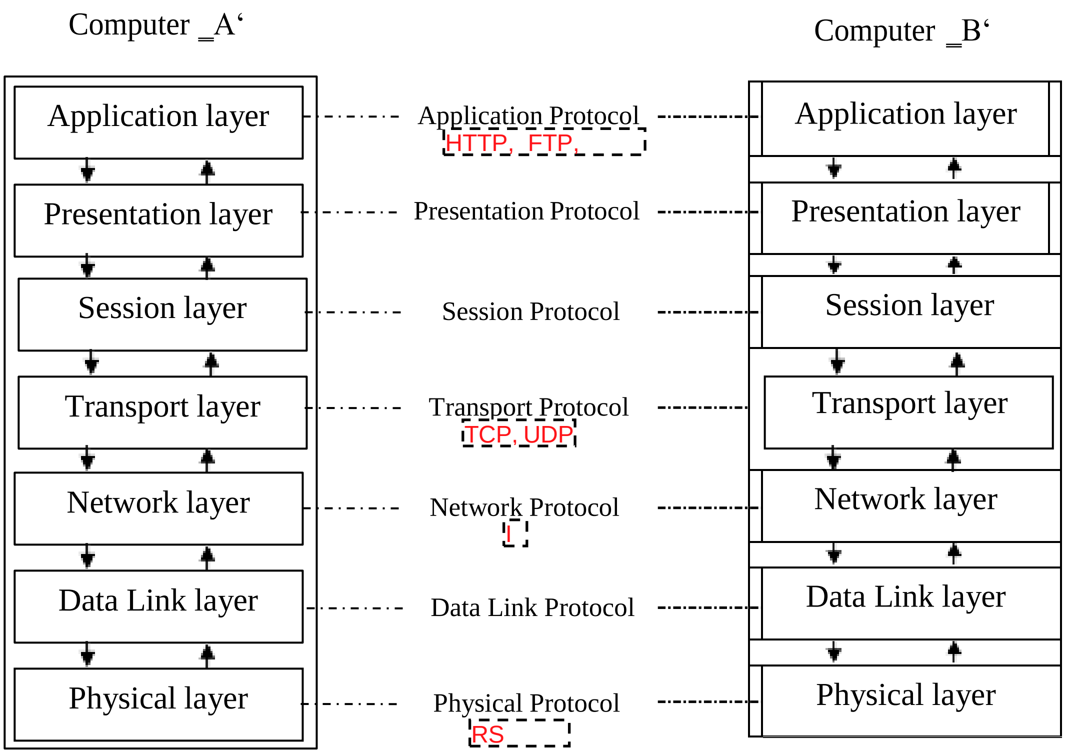
Communication in Distributed Systems

OSI is the system adopted for data communication through internet because there are different sets of technologies, protocols, devices and software‘s used for transmitting data and there is need of a common language, which is understood by al the devices, protocols, technologies and software in the network. OSI provides such type of facility by which all the devices attached in the network speak the same language.

The OSI model was released in 1984, a standard network model. It describes how information makes its way from application programs through a network medium to another application program in another computer.

All communication in distributed system is based on message passing. When a process "A" wants to communicate with a process "B", it first builds a message in its own address space. Then it executes a system call that causes the operating system to fetch the message and send it over the network to "B". To make it easier to deal with numerous levels and issues involved in communication, the international Standards Organization has developed a reference model, that clearly identifies the various level involved,

gives them standard names and points out which level should do which job. This model is called OSI (Open System Interconnection Reference Model)
It consists of seven different layers.



1) Application layer

- It is close to the user.
- It provides services to application programs.

Electronic

- It synchronizes the sending and receiving application.
- It determines whether the sufficient resources of the intended communication exist.
- It establishes agreement on procedures for error recovery and control of data integrity.

2) Presentation layer

- It ensures that information sent by the application layer of one system will be readable by the application layer of another system
- It provides common format for transmitting data across various systems, so that data can be understood.
- It provides facilities to convert message data into a form which is meaningful to communicating application layer entities.
- It may perform such transformation as encoding, decoding, cod conversion, compression and decompression, encryption and decryption on the message data.

3) Session layer

- It provides means of establishing, maintaining and termination a dialogue or session between two end users.
- It allows the two parties to authenticate each before establish a dialog session between them
- It specifies dialog type- one way, two way alternative or two way simultaneous (simplex, half duplex and duplex) and imitates a dialog session.
- It also provides priority management service which is useful for giving priority to important and time bound message over normal, less important message.
- E.g. of session layer protocols and interfaces are: NFS (Network File System), RPC (Remote Procedure Call), SAL, ASP (Apple Talk session protocol)

4) **Transport layer**

- It accepts message of arbitrary length form the session layer, segments them into packets, submits them to the network layer for transmission and finally reassembles the packets at the destination.

errorless delivery of packets is done by this

- It includes mechanisms for handling lost and out of sequence packets. For this transport layer record a sequence number in each packet and uses the sequence number for detecting loss packets and for ensuring that message reconstructed n the correct sequences.
- The two most popular transport layer protocols are: TCP (Transport Control Protocol) and UDP (User Datagram Protocol)

5) **Network layer**

routin

- It is responsible for setting up a logical path between two nodes for communication to take place.
- It encapsulates frames into packets, which can be transmitted from one node to another node by using a high level addressing and routing scheme of the network layer.
- Two popular network layer protocols are: X.25 protocol and Internet Protocol (IP)

6) **Data link layer**

- It is the responsibility of the data link layer to detect and correct any errors in the transmitted data.
- It partitions the raw bit stream into frames, so that error detection and correction can be performed independently for each frame.
- It also performs flow control of frames between two sites to ensure that a sender does not overwhelm the receiver by sending frames at a faster than the receiver can process.
- It is concerned with network topology.
- It is sub-divided into two sub-layers: Logical Link control (LLC) and Media Access Control (MAC) layer.

7) **Physical layer**

- It is responsible for transmitting raw bit streams between tow nodes.
- It also deals with the mechanical details such as size and shape of connecting plugs, the number of pins in the plugs and function of each pin. Etc.
- RS232-c is a popular physical layer standard for serial communication lines.

ATM (Asynchronous Transfer Mode)

The –asynchronous‖ in ATM means ATM devices do not send and receive information at fixed speeds or using a timer, but instead negotiate transmission speeds based on hardware and information flow reliability. The –transfer model‖ in ATM refers to the fixed-size cell structure used for packaging information.

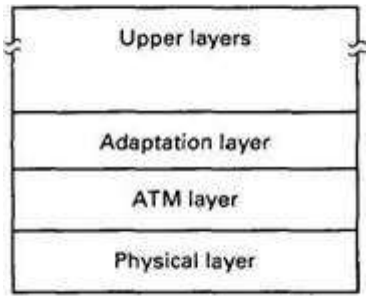
The ATM model is that a sender first establishes a connection (i.e., a virtual circuit) to the receiver or receivers. During connection establishment, a route is determined from the sender to the receiver(s) and routing information is stored in the switches along the way. Using this connection, packets can be sent, but they are chopped up by the hardware into small, fixed-sized units called **cells.** The cells for a given virtual circuit all follow the path stored in the switches. When the connection is no longer needed, it is released and the routing information washed out from the switches.

This scheme has a number of advantages over traditional packet and circuit switching. The most important one is that a single network can now be used to transport an arbitrary mix of voice, data, broadcast television, videotapes, radio, and other information efficiently, replacing what were previously separate networks (telephone, X.25, cable TV, etc.). New services, such as video conferencing for businesses, will also use it. In all cases, what the network sees is cells; it does not care what is in them. This integration represents an enormous cost saving and simplification that will make it possible for each home and business to have a single wire (or fiber) coming in for all its communication and information needs. It will also make possible new applications, such as video-on-demand, teleconferencing, and access to thousands of remote data bases.

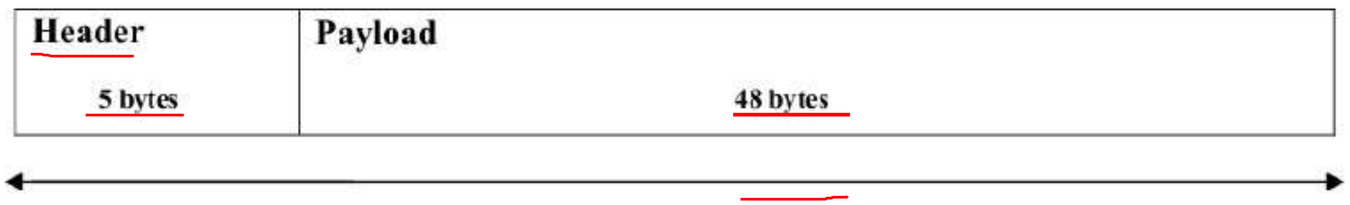
Cell switching lends itself well to multicasting (one cell going to many destinations), a technique needed for transmitting broadcast television to thousands of houses at the same time. Conventional circuit switching, as used in the telephone system, cannot handle this. Broadcast media, such as cable TV can, but they cannot handle point-to-point traffic without wasting bandwidth (effectively broadcasting every message). The advantage of cell switching is that it can handle both point-to-point and multicasting efficiently.

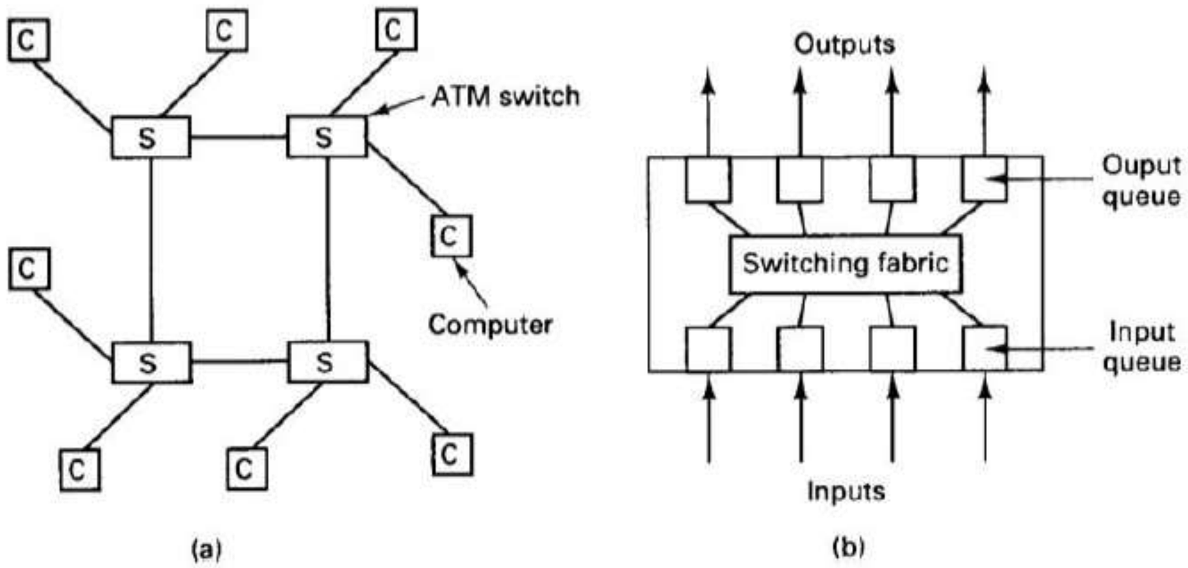
Fixed-size cells allow rapid switching, something much harder to achieve with current store-and-forward packet switches. They also eliminate the danger of a small packet being delayed because a big one is hogging a needed line. With cell switching, after each cell is transmitted , a new one can be sent, even a new one belonging to a different packet.

ATM has its own protocol hierarchy, as shown in Fig. 2-4. The physical layer has the same functionality as layer 1 in the OSI model. The ATM layer deals with cells and cell transport, including routing, so it covers OSI layer 2 and part of layer 3. However, unlike OSI layer 2, the ATM layer does not recover lost or damaged cells. The adaptation layer handles breaking packets into cells and reassembling them at the other end, which does not appear explicitly in the OSI model until layer 4. The service offered by the adaptation layer is not a perfectly reliable end-to-end service, so transport connections must be implemented in the upper layers, for example, by using ATM cells to carry TCP/IP traffic.



ATM transfers information in fixed-size units called cells. Each cell consists of 53 octets, or bytes as shown in following diagram.





Client-Server Model

Client/server architecture is a computing model in which the server hosts, delivers and manages most of the resources and services to be consumed by the client. This type of architecture has one or more client computers connected to a central server over a network or Internet connection. Servers are powerful computers or processes dedicated to managing file servers, print servers or network server. Clients are PCs or workstations on which users run applications. Clients rely on servers for resources, such as files, devices, and even processing power.

Client/server architecture is a producer-consumer computing architecture where the server acts as the producer and the client as a consumer. The server provides different kinds of computing services to the client on demand. These services can include applications access, storage, file sharing, printer access and/or direct access to the server‘s raw computing power.

Client/server architecture works when the client computer sends a resource or process request to the server over the network connection, which is then processed and delivered to the client. A server computer can manage several clients simultaneously, whereas one client can be connected to several servers at a time, each providing a different set of services. In its simplest form, the Internet is also based on client/server architecture where the Web server serves many simultaneous users with Web page and or website data.

Characteristics of clients and servers:

Client's characteristics:-

- Always initiates requests to servers.
- Waits for replies.
- Receives replies.
- Usually connects to a small number of servers at one time.
- Usually interacts directly with end-users using any user interface such as graphical user interface.

Server characteristics:-

- Always wait for a request from one of the clients.
- Serve clients requests then replies with requested data to the clients.
- A server may communicate with other servers in order to serve a client request.

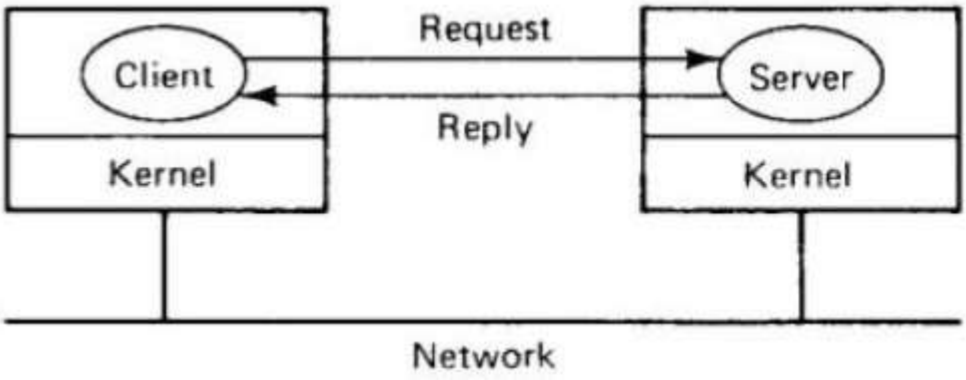


Fig: Client-Server Model

Advantages of client-server:

- **Centralization access:** Data management is much easier because files are in one location i.e. resources and data security is controlled through the server. This allows fast backup and efficient error management.
- **Flexibility:** New technology as well as number of clients can be easily integrated into the system
- **Scalability:** Any element can be upgraded when needed
- **Interoperability:** all components of the system such as clients, network servers etc work together. The server network is designed to serve requests from clients quickly. All the data are processed on the server and only result are returned to the client. This reduces the amount of network traffic between server and clients, improving network performance.

Disadvantage of client-server:

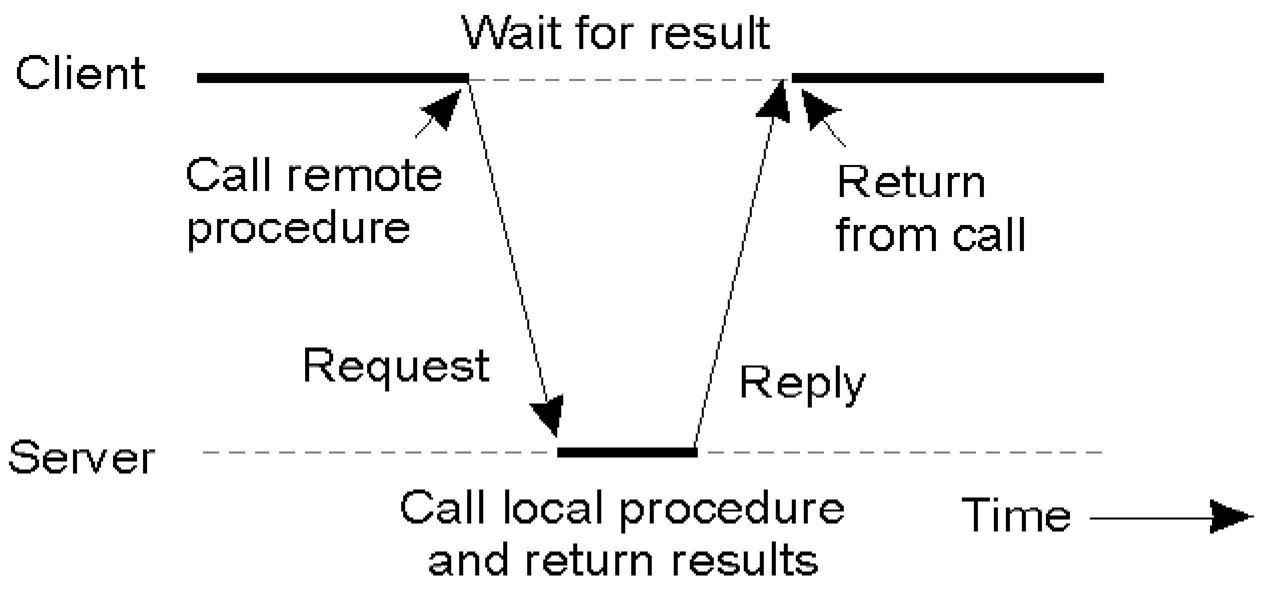
- It is more expensive to install and need a lot of maintenance and for the purpose highly skilled personnel are also required.
- If a server fails, it is possible that the system suffers heavy delay or completely breaks down, which can potentially block hundred of clients form working with their data or their applications
- High data traffic problem is created by broadcasting.
- As the number of simultaneous client request to given server increases, the server can become overloaded.

RPC (Remote Procedure Call)

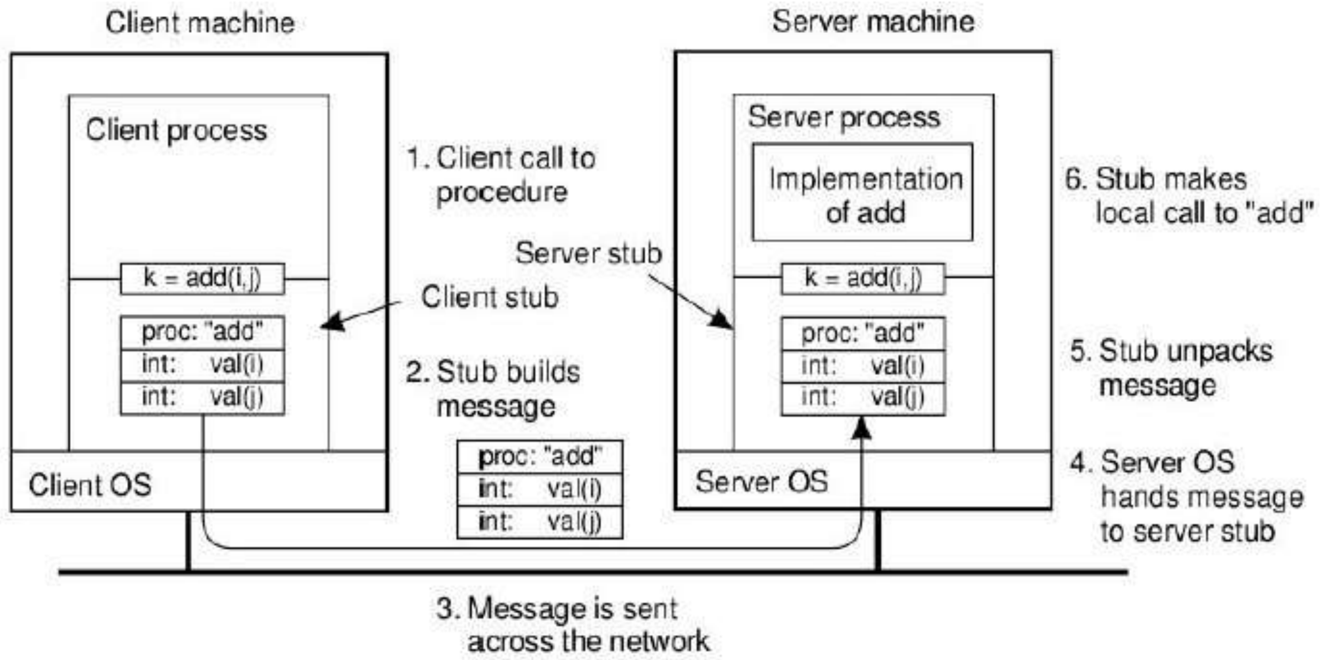
A remote procedure call (RPC) is an inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction. That is, the programmer writes essentially the same code whether the subroutine is local to the executing program, or remote. When the software in question uses object-oriented principles, RPC is called remote invocation or remote method invocation.

Principle of RPC:

- Client makes procedure call (just like a local procedure call) to the client stub
- Server is written as a standard procedure
- Stubs take care of packaging arguments and sending messages
- Packaging parameters is called marshalling
- Stub compiler generates stub automatically from specs in an Interface Definition Language (IDL)
- Simplifies programmer task



Example of RPC:



A remote procedure call occurs in the following steps:

1. The client procedure calls the client stub in the normal way.
2. The client stub builds a message and calls the local operating system.
3. The client‘s OS sends the message to the remote OS.
4. The remote OS gives the message to the server stub.
5. The server stub unpacks the parameters and calls the server.
6. The server does the work and returns the result to the stub.
7. The server stub packs it in a message and calls its local OS.
8. The server‘s OS sends the message to the client‘s OS.
9. The client‘s OS gives the message to the client stub.
10. The stub unpacks the result and returns to the client.

Group Communication

Group is a number of processes which cooperate to provide a service. Group is dynamic i.e. it can be created and destroyed. Group communication is coordination among processes of a group.

Group communication is needed by:

- Highly available servers (client-server)

- Database Replication
- Multimedia Conferencing
- Online Games
- Cluster management

Group communication offers improved efficiency and convenience because

- It delivers a single message to multiple receivers by taking advantages of a network's multicast capability, so it reduces sender and network overhead.
- It provides a high level communication abstraction to simplify user programs in interacting with a group of receivers and
- It hides from application the internal coordination of a group

There are three types of group communications:

- One to one communication
- Inter-and intra-group communications
- Many-to-many communication (group-to-group or multicast) communication

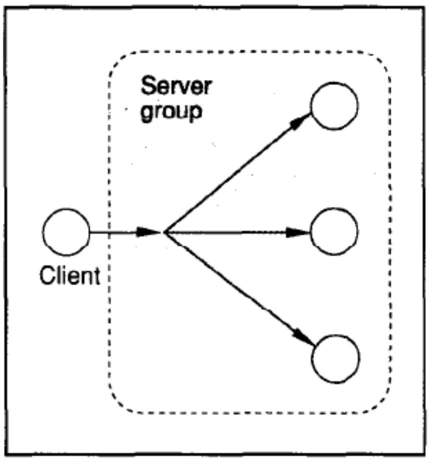


Figure 1. One-to-many communications.

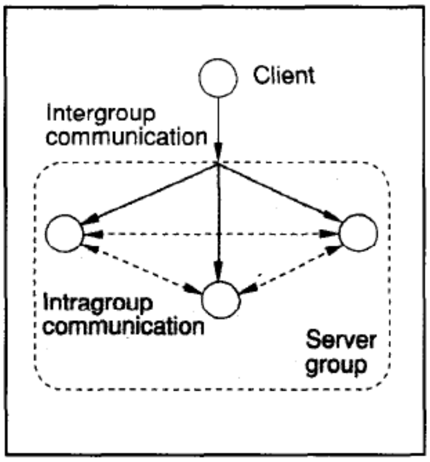


Figure 2. Inter- and intragroup communications.

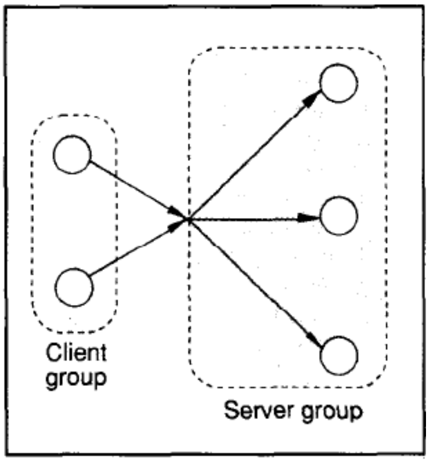


Figure 3. Many-to-many (group-to-group) communications.

Unit 9
Case Study

DOS Operating System

Introduction: MS-DOS

MS-DOS stands for Microsoft Disk Operating system. It is a non-graphical command line operating system. MS-DOS originally written by Tim Paterson and introduced by Microsoft in August 1981 and was last updated in 1994 when MS-DOS 6.22 was released. MS-DOS allows the user to navigate, open, and otherwise manipulate files on their computer from a command line.

Advantages:

- It is very light weight.
- It allows direct access to most hardware.
- It doesn't have overhead of multitasking features
- It is relatively small in size and require less memory, so it takes less time to load
- It is universally compatible with all PC i.e. No sophisticated hardware is needed to run.

Disadvantages:

- It is a *single tasking* operating system. It can support only one user and one program at a time.
- It has a *character-based interface*. So it is harder to use by the user as compared to graphical based interface.

DOS commands

Any instruction given to the computer to perform a specific task is called command. The DOS has several commands, each for a particular task and these are stored in DOS directory on the disk. The commands are of two types:

- Internal command
- External command

a) **Internal Commands:** These are in built commands of MS-DOS i.e. these are stored in Command interpreter file (COMMAND.COM). These commands reside in the memory as long as the machine is at the system prompt (C :\>) level. To use these commands no extra /external file is required.
E.g. DATE, TIME, DIR, VER etc.

b) **External commands:** These are separate program (.com) files that reside in DOS directory and when executed behave like commands. An external command has predefined syntax.
For e.g. HELP, DOSKEY, BACKUP, RESTORE, FORMAT etc.

Basic DOS commands:

a) General commands:

- TIME: sets or displays the system time.
- DATE: Sets or displays system date.
- TYPE: Displays the contents of at the specified file.
- PROMPT: Customizes the DOS command prompt.

b) File management commands:

- COPY: Copies one or more files from source disk/drive to the specified disk/drive.
- XCOPY: Copies files and directories, including lower-level directories if they exists.
- DEL: Removes specified files from specified disk/drive.
- REN: Changes the name of a file(Renaming).
- ATTRIB: Sets or shows file attributes (read, write, hidden, Archive).

- BACKUP: Stores or back up one or more files/directories from source disk/drive to other destination disk/drive.
- RESTORE: Restores files that were backed up using BACKUP command.
- EDIT: Provides a full screen editor to create or edit a text file.
- FORMAT: Formats a disk/drive for data storage and use.

c) Directory commands :

- DIR: To list all or specific files of any directory on a specified disk.
- MD: To make directory or subdirectory on a specified disk/drive.
- CD or CHDIR: Change DOS current working directory to specified directory on specified disk or to check for the current directory on the specified or default drive.
- RMDIR or RD: Removes a specified sub-directory only when it is empty. This command cannot remove root directory (C :\) or current working directory.
- TREE: Displays the entire directory paths found on the specified drive.
- PATH: Sets a sequential search path for the executables files, if the same are not available in the current directory.

System Configurations

Autoexe.bat

It stands automatic executable batch file. This file contains a pre stored sequences of commands that the system executed automatically each time. This is a special batch program that is automatically executed when the system is started. It can be used to define keys, define the path that MS-DOS uses to find files, display messages on the screen etc. It will be executed only if it exists in the root directory or the diskette from which the system is loaded. Each time the system is started, MS-DOS executes the commands stored in AUTOEXEC.BAT file.

Config.sys

This is a system file which is used to configure DOS environment. This file contains reference to device drivers which are loaded when OS takes control of the computer. Using this file new device driver and other software's can be easily installed i.e. this device drivers are required for configuring operating system for running special devices.

MSDOS.sys

It contains the file management and the disk buffering management capabilities. It keeps track of all the disk access of an application program and remains permanently in memory.

IO.sys:

It is the basic I/O system. This program provides interface between the hardware devices and software of the system. It takes care of the keyboard input, character output to monitor, output to printer and time of the day.

Command.com

This file contains command processor. The command processor interprets user commands. Internal commands are the part of this system file where as the external commands are loaded into memory from disk whenever needed. It is also called command interpreter. It is the program that displays the system prompt and handles user interface by executing the command typed in by the user using keyboard.

Filing and Disk Management

A *file system* is the structure in which files are named, stored, and organized.

- Primary - From which you can boot an OS, such as MS-DOS or Win2K Server. Can only have 4 per disk.

- Extended – serves to overcome limit of 4 primary partitions, is not bootable. Can only have 1 per disk.
- System – contains O/S boot files. Can only exists on a primary
- Boot – contains O/S system files. Can exist on a primary or extended.
- Partition needs to be formatted with a file system after it‘s created, and it can then be assigned a drive letter.
- At least 1 partition needs to be marked active , that‘s where our computer will look for the hardware-specific files to start the operating system

Memory Management

Memory is organized in MS-DOS as follows:

- Interrupt vector table
  MS-DOS interrupt handlers use memory addresses in the 1st Mb of physical memory while your program and its data are loaded beyond the 1st Mb
- Optional extra space
- IO.Sys
- MSDOS.Sys
- Buffer, control areas, and installed device drivers
- External commands or utilities
- User attack fro .com files
- Transient path of command

UNIX / Linux Operating System

UNIX / Linux Operating System:

The UNIX operating system is a stable, multi-user, multi-tasking system for servers, desktops and laptops. The original UNIX was developed at AT&T's and Bell Labs research center by Ken Thompson, Dennis Ritchie, and others in 1960s. UNIX systems also have a graphical user interface (GUI) similar to Microsoft Windows which provides an easy to use environment.

UNIX was one of the first operating systems to be written in a high-level programming language, namely C. UNIX operating systems are widely used in servers, workstations and mobile devices.

There are many different versions of UNIX, although they share common similarities. The most popular varieties of UNIX are Sun Solaris, GNU/Linux, and MacOS X.

Linux was originally developed as a free and open source operating system for Intelx86-based personal computers. Now it is a leading operating system on servers and other systems such as mainframe computers and super computers.

The UNIX operating system is made up of three parts; the kernel, the shell and the programs.

The kernel

The kernel of UNIX is the hub of the operating system: it allocates time and memory to programs and handles the file store and communications in response to system calls.

As an illustration of the way that the shell and the kernel work together, suppose a user types rm myfile (which has the effect of removing the file **myfile**). The shell searches the filestore for the file containing the program rm, and then requests the kernel, through system calls, to execute the program rm on **myfile**. When the process rm myfile has finished running, the shell then returns the UNIX prompt % to the user, indicating that it is waiting for further commands.

The shell

The shell acts as an interface between the user and the kernel. When a user logs in, the login program checks the username and password, and then starts another program called the shell. The shell is a command line interpreter (CLI). It interprets the commands the user types in and arranges for them to be carried out. The commands are themselves programs: when they terminate, the shell gives the user another prompt (% on our systems).

Features of UNIX:

- It contains a shell that provides interface between user and system.
- It contains kernel that is used to control the whole operation.
- All users enter the UNIX system by typing a login name and password.
- UNIX system creates and destroys process frequently.
- It supports paging and swapping for memory management
- The file system is in hierarchical form.
- The processes on the UNIX are scheduled on the basis of multi-level queue structure or priority based algorithm.

UNIX Layer: (UNIX environment)

User
Application
Shell
Kernel
Hardware

Fig: UNIX Layers

The UNIX operating system is broken up into a number of layers (levels) and each layer uses the function and services of the lower level layers. The layers are:

1. **Hardware:**
   - It is the bottom layer (layer 0).
   - The Unix operating system will run on 386 or 486 computers with a minimum of 2 megabytes of RAM and 40 megabytes of disk space

2. **Kernel**
   - It is next layer i.e. layer 1.
   - It controls the execution of commands.
   - It is the heart of UNIX system.
   - The kernel interacts with hardware and initiates most tasks such as memory management, task scheduling and file management.
   - It is machine dependent.

3. **Shell:**
   - The shell is the utility that processes our requests i.e. it is a program that inputs command into UNIX.
   - When you type in a command at our terminal, the shell interprets the command and calls the program that we request.
   - It acts as an interface between the user and the kernel
   - The C Shell, Bourne Shell and Korn Shell are the most popular shells used in the UNIX environment.

4. Application (Graphical User Interface)

- Application is the outermost level of UNIX layer.
- It is simply the window display environment that we see on the screen when using a workstation.
- We can customize this environment to suit your tastes as well

File Systems and Disk Management

- A UNIX file system is a sequence of objects or more byte contains arbitrary information.
- No distinction is made between ASCII files and other kinds of files.
- File name is limited up to 256 characters.
- All the ASCII character except null character are allowed in the file name.
- Extension may be of any length and files may have multiple extension like program.z
- Files can be grouped together in directories which can contain sub-directories, i.e. the fiel system is hierarchical in UNIX.

Filter

When a program takes its input from another program, performs some operation on that input, and writes the result to the standard output, it is referred to as a *filter* i.e. a *filter* is a small and (usually) specialized program in UNIX like operating system that transforms plain text (i.e., human readable) data in some meaningful way and that can be used together with other filters and pipes to form a series of operations that produces highly specific results.

Filters read data from standard input and write to standard output. Standard input is the source of data for a program, and by default it is text typed in at the keyboard. However, it can be redirected to come from a file or from the output of another program. Standard output is the destination of output from a program, and by default it the display screen. This means that if the output of a command is not redirected to a file or another device (such as a printer) or piped to another filter for further processing, it will be sent to the monitor where it will be displayed.

Numerous filters are included on Unix-like systems, a few of which are:

awk, cat, comm, csplit, cut, diff, expand, fold,
rep, head, join, less, more, paste, sed, sort,
spell, tail, tr, unexpand, uniq and wc.

Example:

```
ls /sbin | grep mk | sort -r | head -3
```

The *ls* command lists the contents of /sbin and pipes its output to the filter *grep*, which searches for all files and directories that contain the letter sequence *mk* in their names. grep then pipes its output to the *sort* filter, which, with its *-r* option, sorts it in reverse alphabetical order. sort, in turn, pipes its output to the *head* filter. The default behavior of head is to read the first ten lines of text or output from another command, but the *-3* option here tells it read only the first three. head thus writes the first three results from sort (i.e., the last three filenames or directory names from /sbin that contain the string mk) to the display screen.

Pipelining

A *pipe* is a form of redirection that is used in Linux and other UNIX like operating system to send the output of one program to another program for further processing.

Redirection is the transferring of standard output to some other destination, such as another program, a file or a printer, instead of the display monitor (which is its default destination). Standard output, sometimes abbreviated *stdout*, is the destination of the output from command line (i.e., all-text mode) programs in Unix-like operating systems.

Pipes are used to create what can be visualized as *a pipeline of commands*, which is a temporary direct connection between two or more simple programs. This connection makes possible the performance of some highly specialized task that none of the constituent programs could perform by themselves. A command is merely an instruction provided by a user telling a computer to do something, such as launch a program. The command line programs that do the further processing are referred to as filters.

This direct connection between programs allows them to operate simultaneously and permits data to be transferred between them continuously rather than having to pass it through temporary text files or through the display screen and having to wait for one program to be completed before the next program begins.

A pipe is designated in commands by the vertical bar character. The general syntax for pipes is:

```
    command_1 | command_2 [| command_3 . . . ]
```

This chain can continue for any number of commands or programs.

Examples

1. The output of the *ls* command (which is used to *list* the contents of a directory) is commonly piped to the the less (or more) command to make the output easier to read, i.e.,
   ```
   ls -al | less
   ```
   or
   ```
   ls -al | more
   ```
   ls reports the contents of the current directory in the absence of any arguments (i.e., input data in the form of the names of files or directories).
   - The *-l* option tells ls to provide detailed information about each item, and
   - The *-a* option tells ls to include all files, including hidden files (i.e., files that are normally not visible to users). Because ls returns its output in alphabetic order by default, it is not necessary to pipe its output to the sort command (unless it is desired to perform a different type of sorting, such as reverse sorting, in which case sort's *-r* option would be used).

2. The following example employs a pipe to combine the `ls` and the `wc` (i.e., *word count*) commands in order to show how many *file system objects* (i.e., files, directories and links) are in the current directory:
   ```
   ls | wc -l
   ```

Sockets

In UNIX and some other operating system, socket is a software object that connects an application to a network protocol. In UNIX, for example, a program can send and receive TCP/IP messages by opening a socket and reading and writing data to and from the socket. This simplifies program development because the programmer need only worry about manipulating the socket and can rely on the operating system to actually transport messages across the network correctly. Socket in this sense is completely a software object, not a physical component.

Sockets are used for communication, particularly over a network. Sockets were originally developed by the BSD branch of UNIX systems, but they are generally portable to other Unix-like systems:

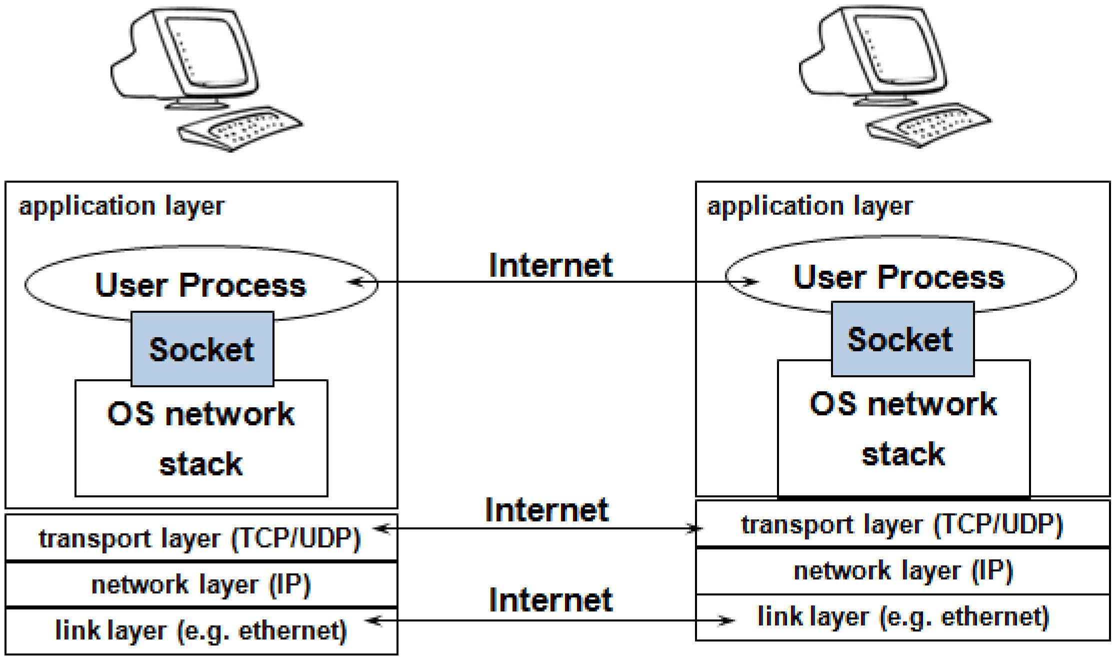


Fig: UNIX - Sockets

Security in Linux/UNIX

Linux security model is closely related to special UNIX security mechanism. The security

User accounts

Every UNIX-like system includes a root account, which is the only account that may directly carry out administrative functions. All of the other accounts on the system are unprivileged. This means these accounts have no rights beyond access to files marked with appropriate permissions, and the ability to launch network services.

File permission

Every file and directory on a UNIX system has three sets of file permissions that determine how it may be accessed, and by whom:

- The permissions for the *owner*, the specific account that is responsible for the file
- The permissions for the *group* that may use the file
- The permissions that apply to all *other* accounts

Each set may have none or more of the following permissions on the item:

- *read*
- *write*
- *execute*

The majority of files on a UNIX-like system is owned by the root account, and has permissions that restrict or block access from all other accounts.

System firewall

The *netfilter* framework included in the Linux kernel restricts incoming and outgoing network connections according to a set of rules that have been defined by the administrator. Several Linux distributions configure firewall rules by default, and offer utilities for managing simple firewall configurations.

Encrypted storage

We can create one or more *encrypted volumes* to store our sensitive files. Each volume is a single file which may enclose other files and directories of our choice. To access the contents of the volume, we must provide the correct decryption password to a utility, which then makes the volume available as if it were a directory or drive. The contents of an encrypted volume cannot be read when the volume is not mounted. We may store or copy encrypted volume files as our wish without affecting their security.

In extreme cases, you may decide to encrypt an entire disk partition that holds or caches data, so that none of the contents may be read by unauthorized persons. On Linux you may use either LUKX, CryptoFS or EncFS to encrypt disks

Application Isolation

The most common UNIX-like operating systems provide several methods of limiting the ability of a program to affect either other running programs, or the host system itself.

- *Mandatory Access Control (MAC)* supplements the normal UNIX security facilities of a system by enforcing absolute limits that cannot be circumvented by any program or account.
- *Virtualization* enables you to assign a limited set of hardware resources to a virtual machine, which may be monitored and backed up by separate processes on the host system.
- The *chroot* utility runs programs within a specified working directory, and prevents them from accessing any other directory on that system.

The administrator may setup guest operating systems in virtual environments for specific tasks, and restrict these guests far more than would be possible for a multi-purpose system.

Shell

A UNIX shell is a command line interpreter that provides a traditional interface for the UNIX operating system. The user interacts with the system by typing commands in the shell. The shell hides the details of the underlying operating system behind the shells interface. The different types of UNIX shell are:

- Bourne shell
- C-shell
- Korn shell

Bourne shell

- It was one of the major shells used in early version of the UNIX operating system.
- It was written by Stephan Bourne at Bell laboratory.
- Every UNIX system has at least one shell compatible with Bourne shell and it was default shell in many UNIX.
- The Bourne Shell has neither the interactive features, nor the complex programming constructs.
- It doesn't contain command history feature i.e. the ability to store commands in a buffer, then modified and reused
- It doesn't contain line editing feature i.e. the ability to modify the current or previous command lines with a text editor
- It also doesn't contain restricted shells i.e. security feature providing a controlled environment with limited capabilities

C-shell

- The C-Shell was developed by Bill Joy of Sun-micro system.
- The C-shell has syntax similar to the C programming language.
- The C shell is a command processor typically run in a text window, allowing the user to type commands.
- The C shell program name is *csh* , and the shell prompt is % symbol.

Korn shell

- This shell was provided by T-Korn
- It is enhancement of the Bourne shell including features from the C-shell.

Memory Management

Memory management is one of the most important services of UNIX kernel. In a computer system, CPU must cooperate with the memory to accomplish any computing. The main memory has limited space and cannot contain all the programs on the disk. However, a process cannot execute if it is not brought in memory. Thus, the memory management becomes quite important, especially when the sizes of application programs become fairly large.

UNIX is intended to be machine independent. So it's memory management scheme is also vary from one system to another.

Actually it has two separate memory management scheme.

The paging system provides a virtual memory capability that allocates page frames in main memory to process and also allocates page frames to disk block buffers. Although this is an effective memory management scheme for user process and disk I/O.

For paged virtual memory, UNIX makes use of a number of data structure having following table:

Page table entry	Disk block description	Performance data table	Swap use table

Fig: UNIX memory management format.

As in a single process operating system only one process at a time can be running, there is just one program sharing the memory, except the operating system. The operating system may be located at the lower-addressed space of the memory and the user program at the rest part. Thus, the memory management is quite simple i.e. there is not too much work to do for the memory management in the single process operating system. The memory management just handles how to load the program into the ser memory space from the disk when a program is typed in by a user and leaves the process management to accomplish the program execution. When a new program name is typed in by the user after the first one finishes, the memory management also loads it into the same space and overwrite the first one

In multiprocessing operating systems, there are many processes that represent different programs to execute simultaneously, which must be put in different areas of the memory. Multiprogramming increase the CPU utilization, but needs complex schemes to divide and manage the memory space for several processes in order to avoid the processes interfering with each other when executing and make their execution just like single process executing in the system.

It may be the simplest scheme to divide the physical memory into several fixed areas with different sizes. When a task arrives, the memory management should allocate it the smallest area that is large enough to hold it and mark this area as used. When the task finishes, the management should de-allocate the area and mark it as free for the later task. A data structure is necessary to hold the information for each size-fixed area, including its size, location and use state. There are two biggest disadvantages for this scheme

- The fixed sizes cannot meet the needs of the number of increasing of the tasks brought in the system simultaneously, however it can be handled by swapping and
- The size growing of application program and it can be handled by paging.

Windows 2000

Windows 2000:

Microsoft Windows 2000, Win2K in short, is the Windows NT series 32 bit Windows operating system released by Microsoft in 1999.

There are four versions of Windows2000 **(English version)**

- Windows 2000 professional

- Windows server
- Windows Advanced server
- Windows 2000 Datacenter server

File System and Disk Management

Windows 2000 systems can support the following file systems:

- FAT, FAT32
- NTFS (New Technology File System)
- CDFS (Compact Disk File System)
- UDF (Universal Disk Format for DVDs)
- EFS (Encrypting File System) It runs as a service and is used to encrypt and decrypt files on an NTFS file system for security purposes. The EFS is not a file system like NTFS since it does not create partitions and control the placement of file data, it only is used to control the encryption of data.

FAT file system:

- Used with DOS
- It can only support partitions up to 4 G.
- No spaces are allowed in the file name.

FAT 32 or VFAT file system:

Virtual File Allocation Table (VFAT) introduced by Windows 95 which allows long file names. VFAT is not natively supported by Windows 2000.

- FAT32 file systems support partitions up to 32GB.
- Filenames up to 255 characters long.
- Filenames begin with a letter and exclude " / \ [ ] : ; | = , ^ * ?
- The last part is the extension but spaces can be used
- It supports file attributes used by DOS such as read-only, archive, system, and hidden.
- Won't support running POSIX applications.

FAT partitions provide no local security, only share level security across a network

NTFS file system:

Windows 2000 NTFS file systems are newer than Windows NT NTFS file systems. In order for Windows NT and Windows 2000 to use the Windows 2000 file system together, the Windows NT system must have service pack 4 or later installed.

- Filenames up to 255 characters long
- Filenames preserve case but are not case sensitive.
- Filenames exclude " / \ < > : | * ?
- Supports built in file compression as a file attribute. Compression is applied to files in a folder if that folder has its compression attribute set. Also optionally sub folders and their contents may be compressed.
- Provides automatic transaction tracking of disk activity for correcting corrupt or failed operations.
- Supports auditing.
- Provides sector sparing.
- There is a recycle bin for each user.
- Windows 16 bit and DOS environments can't use this file system.
- A master file table is used to save individual file, boot sector, disk structure, and file recovery information.
- Provides file logging ability and file recovery.
- Supports POSIX.
- Supports file sharing with Macintosh clients.

- Supports file encryption with the Encrypting File System (EFS) on Windows 2000.
- Allows volumes on remote computers or local computers to be mounted as though they are part of the same partition they are mounted on. This feature is available on Windows 2000.
- Removable media formatted in NTFS can be changed and accessed without rebooting the system in Windows 2000 (not NT).

CDFS

- It stands for Compact Disk File System.
- The file system that supports compact disks (CDs)

UDF

- It stands for Universal Disk Format.
- The file system that supports DVDs.

File system and windows system

Operating System	NTFS	FAT32	FAT	CDFS	UDF	HPFS
Windows 2000	Yes	Yes	Yes	Yes	Yes	No
Windows NT 4.0	Yes	No	Yes	Yes	Yes	No
Windows NT 3.51	Yes	No	Yes	Yes	No	Yes
Windows 98	No	Yes	Yes	Yes	Yes	No
Windows 95	No	Yes	Yes	Yes	Yes	No
Windows 3,x & WFW	No	No	Yes	Yes	No	No
OS/2	No	No	Yes	Yes	No	Yes
MS-DOS	No	No	Yes	Yes	No	No

Networking

Security

Authentication is performed by the system to be sure the user is really who they claim to be. Authentication may be done at and for a local computer or at a global level for a domain using domain controllers across the network. Windows 2000 supports the following types of authentication:

- Kerberos V5 (RFC 1510) - An internet standard authentication protocol which is the default protocol for Windows 2000 computers within a domain. This is not used for computers in different forests.
- Windows NT LAN Manager (NTLM) - Used to authenticate users from Windows 95, 98, and NT systems. Windows 2000 Active Directory must be operating in mixed mode to use this authentication method.
- Secure Sockets Layer/Transport Layer Security (SSL/TLS) - Requires certificate servers and is used to authenticate users that are logging onto secure web sites.
- Smart card

AMOEBA

AMOEBA:

Amoeba is a **distributed operating system.** It collects a huge variety of single machines connected over a (fast) network to one, huge computer. It was originally developed at the Vrije Universiteit in Amsterdam by

Andrew Tanenbaum and many more. Amoeba was always designed to be used, so it was deemed essential to achieve extremely high performance. Currently, it's the fastest distributed operating system.

Amoeba is a general-purpose distributed operating system. It is designed to take a collection of machines and make them act together as a single integrated system. In general, users are not aware of the number and location of the processors that run their commands, or of the number and location of the file servers that store their files. To the casual user, an Amoeba system looks like a single old-fashioned time-sharing system.

Amoeba was a distributed operating system developed by Andrew S. Tanenbaum and others at the Vrije University. The aim of the Amoeba project is to build a timesharing system that makes an entire network of computers appear to the user as a single machine.

The goal of the Amoeba project was to construct an operating system for networks of computers that would present the network to the user as if it were a single machine. An Amoeba network would consists of a number of workstations connected to a "pool" of processors, and executing a program from a terminal would cause it to run on any of the available processors, with the operating system providing load balancing.

Amoeba was a microkernel-based operating system. It offered multithread programs and a remote procedure call (RPC) mechanism for communication between threads, potentially across the network; even kernel-threads would use this RPC mechanism for communication. Each thread was assigned a 48-bit number called its "port", which would serve as its unique, network-wide "address" for communication.

The basic design goals of Amoeba are:

- Distribution—Connecting together many machines
- Parallelism—Allowing individual jobs to use multiple CPUs easily
- Transparency—Having the collection of computers act like a single system
- Performance—Achieving all of the above in an efficient manner

The system Architecture of Amoeba

Amoeba implements a universal distributed Client-Server-Model. Basically the whole system needs only three Functions to do all the work: The transaction call from the Client, and the GetRequest and PutReply functions on the Server side.

An Amoeba System consists of four principle components:

- Workstations
- Pool Processors
- Specialized Servers (File server...)
- Gateways

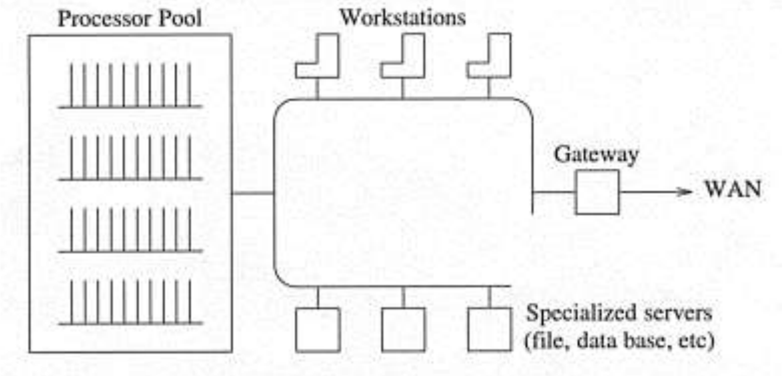


Fig: System architecture of AMOEBA

a) **Processor Pool**

Amoeba is designed as a collection of micro kernels. Thus the Amoeba system consists of many CPU's connected over a network. Each CPU owns his own local Memory in the range from 2MB to several 100MB. A huge number of Processor's build the so called Processor pool. This group of CPUs can be dynamically allocated as needed by the system and the users. Specialized servers, called Run server, distribute processes in a fair manner to these machines.

b) **Workstations:**

Workstations allow the users to gain access to the Amoeba system. There is typically one workstation per user, and the workstations are mostly diskless; only a workstation kernel must be booted. Amoeba supports X-Windows and UNIX-emulation.

c) **Specialized server:**

At heart of the Amoeba system are several specialized servers that carry out and synchronize the fundamental operations of the kernel. Some servers are: directory server, file server, boot server, etc.

- Amoeba has a directory server (called SOAP) that is the naming service for all objects used in the system. SOAP provides a way to assign ASCII names to an object so it's easier to manipulate (by humans). The directory server can replicate files without fearing their change.
- Amoeba has of course a file server (called the Bullet Server) that implements a stable high speed file service. High speed is achieved by using a large buffer cache. Since the files are first created in cache, and are only written to disk when they are closed, all the files can be stored contiguously. And file server crashes normally don't result in an inconsistent file system. The Bullet server uses the virtual disk server to perform I/O to disk, so it's possible that the file server run as a normal user program!
- The Boot server controls all global system servers (outside the kernel): start, check and poll, restart if crashed.

d) **Gateway**

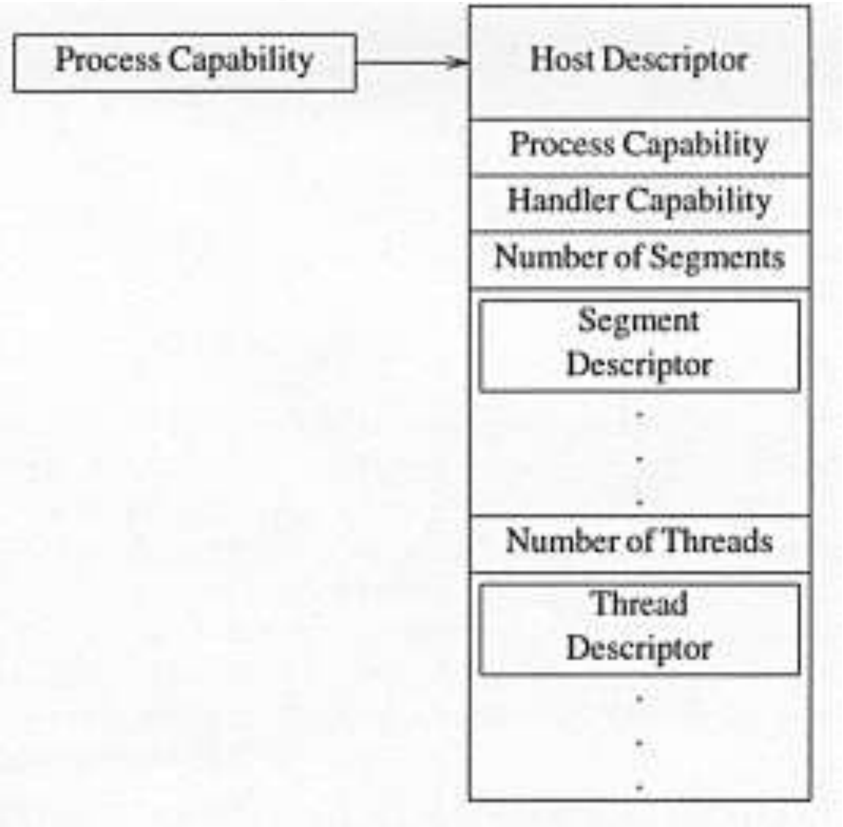
Memory management

Process Management

Communication in AMOEBA

Process management

A process is an object in Amoeba. Information about the processes in Amoeba are contained in capabilities and in a data structure called a process descriptor, which is used for process creation and stunned processes (and process migration). The process descriptor consists of four components:



- The host descriptor provides the requirements for the system where the process must run, by describing what machine it can be run
- The capabilities include the capability of the process which every client needs, and the capability of a handler, which deals signals and process exit
- The segment component describes the layout of the address space (see below)
- The thread component describes the state of each of the threads (see below) in the process and their state information's (IP, Stack, ...)

Amoeba supports a simple thread model. When a process starts up, it has at least one thread. The number of threads is dynamic. During execution, the process can create additional threads. And existing threads can terminate. All threads are managed by the kernel. The advantage of this design is that when a thread does a RPC, the kernel can block that thread and schedule another one in the same process if one is ready!

Three methods are provided for thread synchronization:

- Mutexes
- Semaphores
- Signals

A Mutex is like a binary semaphore. It can be in one of two states, locked or unlocked. Trying to lock an unlocked Mutex causes it to become locked. The calling thread continues. Trying to lock a Mutex that is already locked causes the calling thread to block until another thread unlocks the Mutex. The second way threads can synchronize is by counting Semaphores. These are slower than Mutex, but there are times when they are needed. A semaphore can't be negative. Try down a zero semaphore causes the calling thread to block until another thread do a up operation on the semaphore.

Signals are asynchronous interrupts sent from one thread to another in the same process. Signals can be raised, caught, or ignored. Asynchronous interrupts between processes use the stun mechanism.

Memory management

Amoeba supplies a simple memory management based on segments. Each process owns at least three segments:

- Text/Code segment
- Stack segement for the main thread/process
- Data segment

Each further thread gets its own stack segment, and the process can allocated arbitrary additional data segments.

All segments are page protected by the underlying MMU, the kernel segments, too.

Communication

All processes, the kernel too, communicate with a standardized RPC (Remote procedure call) interface. There are only three functions to reach this goal:

- trans(req_header, req_buf, req_size, rep_header,rep_buf, rep_size)
  - ➔ do a transaction to another server

- getreq(req_header, req_buf, req_size)
  - ➔ get a client request

- putrep(rep_header, rep_buf, rep_size)
  - ➔ send a reply to the client

The first function is used by client to send a message to a server, and get a reply from the server on this request. The reply and request buffers are generic memory buffers (char). The reply and request headers are simple data structures to describe the request and the capability of the server. On the other side, the server calls within an infinite loop the getreq function. Each time a client sends this server (determined by a server port - see capabilities for details) a message, the getreq function returns with the client data filled in the request buffer, if any. The request header contains information's about the client request.

Because the client expects a reply, the server must send a reply (either with or without reply data) using the reply function.

Based on the article: Amoeba: An Overview of a Distributed Operating System by Eric W. Lund - March 29, 1998, Rochester Institute of Technolog and informations about Amoeba taken from a web site from the University of Halle. Additional parts are taken from the Amoeba tribute site from Stephen Wagner. Furthermore the classics: The Amoeba kernel, Andrew S. Tanenbaum, M.F. Kaashoek.