DIPLOMA IN INFORMATION COMMUNICATION TECHNOLOGY

STUDY NOTES

Operating Systems

MODULE I

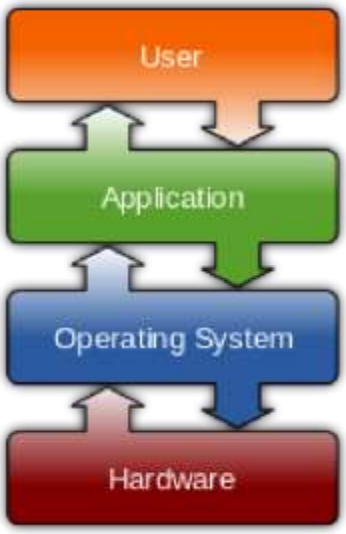
Contents

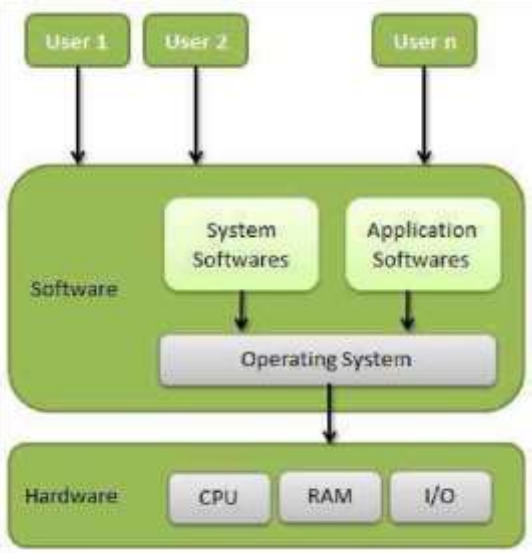
CHAPTER 1: INTRODUCTION TO OPERATING SYSTEM

Introduction to Operating system

An **operating system** (**OS**) is system software that manages computer hardware and software resources and provides common services for computer programs. All computer programs, excluding firmware, require an operating system to function.

An **operating system** is a program that acts as an interface between the user and the computer hardware and controls the execution of all kinds of application programs and assistant system software programs (i.e. Utilities).





Basic importance of the operating system

1) Operating system behaves as a resource manager. It utilizes the computer in a cost effective manner. It keeps account of different jobs and the where about of their results and locations in the memory.
2) Operating system schedules jobs according to their priority passing control from one program to the next. The overall function of job control is especially important when there are several uses (a multi user environment)
3) Operating system makes a communication link user and the system and helps the user to run application programs properly and get the required output
4) Operating system has the ability to fetch the programs in the memory when required and not all the Operating system to be loaded in the memory at the same time, thus giving the user the space to work in the required package more conveniently and easily

5) Operating system helps the user in file management. Making directories and saving files in them is a very important feature of operating system to organize data according to the needs of the user
6) Multiprogramming is a very important feature of operating system. It schedules and controls the running of several programs at ones
7) It provides program editors that help the user to modify and update the program lines
8) Debugging aids provided by the operating system helps the user to detect and rename errors in programs
9) Disk maintenance ability of operating system checks the validity of data stored on diskettes and other storage to make corrections to erroneous data

Operating Systems Terminology's

Processes: A **process** is an instance of a program running in a computer.
A program in the execution is called a Process. Process is not the same as program. A process is more than a program code (Program Code + Data + Execution status).

Files: A collection of data or information that has a name, called the *filename*. Almost all information stored in a computer must be in a file. 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 system call is a way for programs to interact with the operating system. A computer program makes a system call when it makes a request to the operating system's kernel. System calls are used for hardware services, to create or execute a process, and for communicating with kernel services, including application and process scheduling.

Shell AND kernel
A **shell** is a software interface that's often a command line interface that enables the user to interact with the computer. Some examples of shells are MS-DOS Shell, command.com, csh, ksh, and sh. Below is a picture and example of what a Terminal window with an open shell. **A Kernel** is first section of the operating system to load into memory. As the center of the operating system, the kernel needs to be small, efficient and loaded into a protected area in the memory; so as not to be overwritten**.** It can be responsible for such things as disk drive management, interrupt handler, file management, memory management, process management, etc.

Virtual Machines: A virtual machine (VM) is a software program or operating system that not only exhibits the behavior of a separate computer, but is also capable of performing tasks such as running applications and programs like a separate computer. A virtual machine, usually known as

a guest is created within another computing environment referred as a "host." Multiple virtual machines can exist within a single host at one time.

The History of Operating Systems

The first operating system was created by General Motors in 1956 to run a single IBM mainframe computer. Other IBM mainframe owners followed suit and created their own operating systems. As you can imagine, the earliest operating systems varied wildly from one computer to the next, and while they did make it easier to write programs, they did not allow programs to be used on more than one mainframe without a complete rewrite.

In the 1960s, IBM was the first computer manufacturer to take on the task of operating system development and began distributing operating systems with their computers. However, IBM wasn't the only vendor creating operating systems during this time. Control Data Corporation, Computer Sciences Corporation, Burroughs Corporation, GE, Digital Equipment Corporation, and Xerox all released mainframe operating systems in the 1960s as well.

In the late 1960s, the first version of the Unix operating system was developed. Written in C, and freely available during it's earliest years, Unix was easily ported to new systems and rapidly achieved broad acceptance. Many modern operating systems, including Apple OS X and all Linux flavors, trace their roots back to Unix.

Microsoft Windows was developed in response to a request from IBM for an operating system to run its range of personal computers. The first OS built by Microsoft wasn't called Windows, it was called MS-DOS and was built in 1981 by purchasing the 86-DOS operating system from Seattle Computer Products and modifying it to meet IBM's requirements. The name Windows was first used in 1985 when a graphical user interface was created and paired with MS-DOS.

Apple OS X, Microsoft Windows, and the various forms of Linux (including Android) now command the vast majority of the modern operating system market.

The First Generation (1940's to early 1950's)

When electronic computers where first introduced in the 1940's they were created without any operating systems. All programming was done in absolute machine language, often by wiring up plugboards to control the machine's basic functions. During this generation computers were generally used to solve simple math calculations, operating systems were not necessarily needed.

The Second Generation (1955-1965)

The first operating system was introduced in the early 1950's, it was called GMOS and was created by General Motors for IBM's machine the 701. Operating systems in the 1950's were called single-stream batch processing systems because the data was submitted in groups. These new machines were called mainframes, and they were used by professional operators in large

computer rooms. Since there was such as high price tag on these machines, only government agencies or large corporations were able to afford them.

The Third Generation (1965-1980)

By the late 1960's operating systems designers were able to develop the system of multiprogramming in which a computer program will be able to perform multiple jobs at the same time.The introduction of multiprogramming was a major part in the development of operating systems because it allowed a CPU to be busy nearly 100 percent of the time that it was in operation. Another major development during the third generation was the phenomenal growth of minicomputers, starting with the DEC PDP-1 in 1961. The PDP-1 had only 4K of 18bit words, but at $120,000 per machine (less than 5 percent of the price of a 7094), it sold like hotcakes. These microcomputers help create a whole new industry and the development of more PDP's. These PDP's helped lead to the creation of personal computers which are created in the fourth generation.

The Fourth Generation (1980-Present Day)

The fourth generation of operating systems saw the creation of personal computing. Although these computers were very similar to the minicomputers developed in the third generation, personal computers cost a very small fraction of what minicomputers cost. A personal computer was so affordable that it made it possible for a single individual could be able to own one for personal use while minicomputers where still at such a high price that only corporations could afford to have them. One of the major factors in the creation of personal computing was the birth of Microsoft and the Windows operating system. The windows Operating System was created in 1975 when Paul Allen and Bill Gates had a vision to take personal computing to the next level. They introduced the MS-DOS in 1981 although it was effective it created much difficulty for people who tried to understand its cryptic commands. Windows went on to become the largest operating system used in techonology today with releases of Windows 95, Windows 98, WIndows XP (Which is currently the most used operating system to this day), and their newest operating system Windows 7. Along with Microsoft, Apple is the other major operating system created in the 1980's. Steve Jobs, co founder of Apple, created the Apple Macintosh which was a huge success due to the fact that it was so user friendly. Windows development throughout the later years were influenced by the Macintosh and it created a strong competition between the two companies. Today all of our electronic devices run off of operating systems, from our computers and smartphones, to ATM machines and motor vehicles. And as technology advances, so do operating systems.

Operating System Structure

An operating system might have many structure. 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

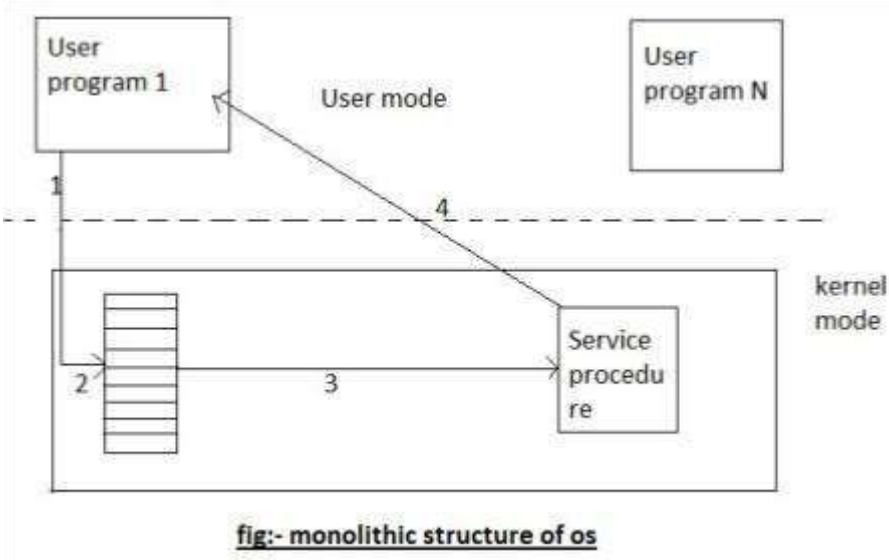


fig:- monolithic structure of os

It is the oldest architecture used for developing operating system. Operating system resides on kernel for anyone to execute. System call is involved i.e. 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 system then examines the parameter of the call to determine which system call is to be carried out shown in event 2. Next, the operating system index‟s into a table that contains procedure that carries out system call. This operation is shown in events. 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.

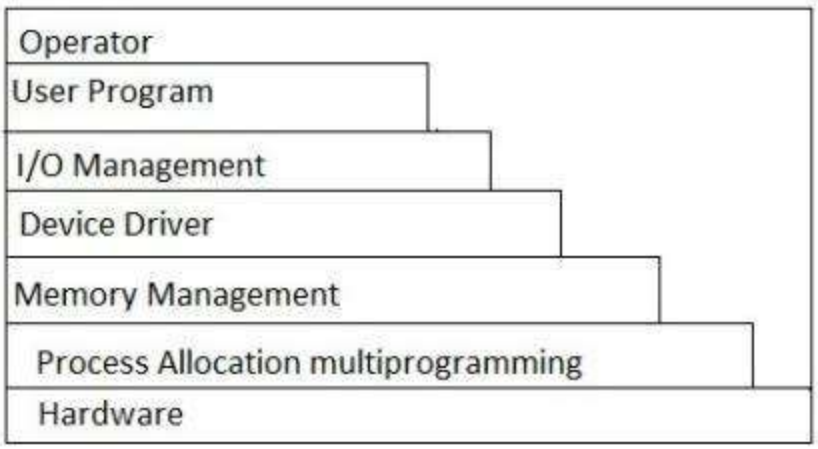


fig:- layered Architecture

The layered are selected such that each user functions and services of only lower level layer. The first layer can be debugged wit 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.

3. Virtual memory architecture of operating system

fig:- virtual memory architecture of os

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.
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):- 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.

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, minimising 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.

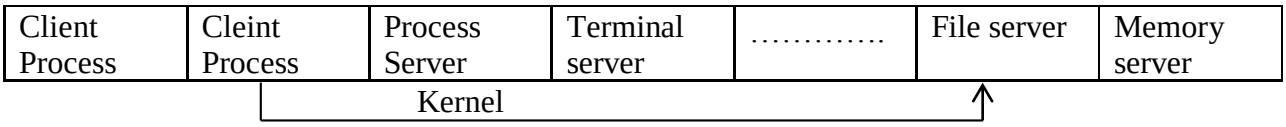


Fig: The client-server model

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.

Operating System – Types

Batch Operating System

The users of a batch operating system do not interact with the computer directly. Each user prepares his job on an off-line device like punch cards and submits it to the computer operator. To speed up

processing, jobs with similar needs are batched together and run as a group. The programmers leave their programs with the operator and the operator then sorts the programs with similar requirements into batches.

The problems with Batch Systems are as follows:

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

Time-sharing Operating Systems

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

The main difference between Multiprogrammed Batch Systems and Time-Sharing Systems is that in case of Multiprogrammed batch systems, the objective is to maximize processor use, whereas in Time-Sharing Systems, the objective is to minimize response time.

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

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

Advantages of Timesharing operating systems are as follows:

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

Disadvantages of Time-sharing operating systems are as follows:

Problem of reliability

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

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

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

The advantages of distributed systems are as follows:

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

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

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

The advantages of network operating systems are as follows:

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

The disadvantages of network operating systems are as follows:

- High cost of buying and running a server.
- Dependency on a central location for most operations. Regular maintenance and updates are required.

Real-Time Operating System

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

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

There are two types of real-time operating systems.

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

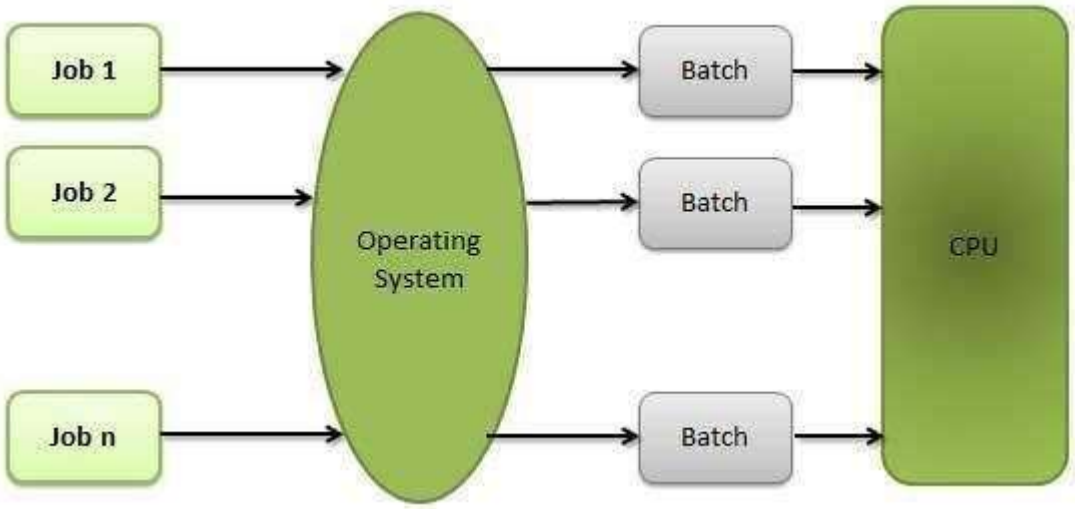
2) Soft real-time systems: Soft real-time systems are less restrictive. A critical real-time task gets priority over other tasks and retains the priority until it completes. Soft real-time systems have limited utility than hard real-time systems. For example, multimedia, virtual reality, Advanced Scientific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

Operating System – Properties

Batch Processing

Batch processing is a technique in which an Operating System collects the programs and data together in a batch before processing starts. An operating system does the following activities related to batch processing:

- The OS defines a job which has predefined sequence of commands, programs and data as a single unit.
- The OS keeps a number a jobs in memory and executes them without any manual information.
- Jobs are processed in the order of submission, i.e., first come first served fashion.
- When a job completes its execution, its memory is released and the output for the job gets copied into an output spool for later printing or processing.



Advantages

- Batch processing takes much of the work of the operator to the computer.
- Increased performance as a new job get started as soon as the previous job is finished, without any manual intervention.

Disadvantages

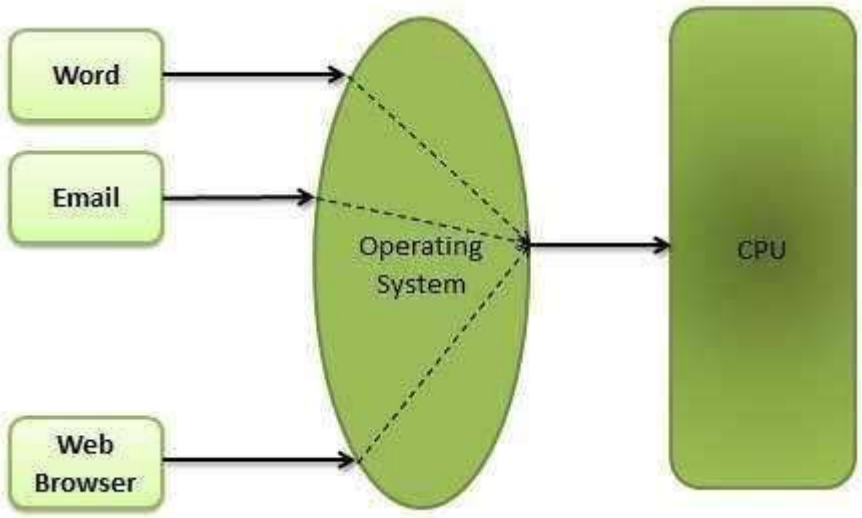
- Difficult to debug programs.
- A job could enter an infinite loop.
- Due to lack of protection scheme, one batch job can affect other pending jobs.

Multitasking

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

- The user gives instructions to the operating system or to a program directly, and receives an immediate response.

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



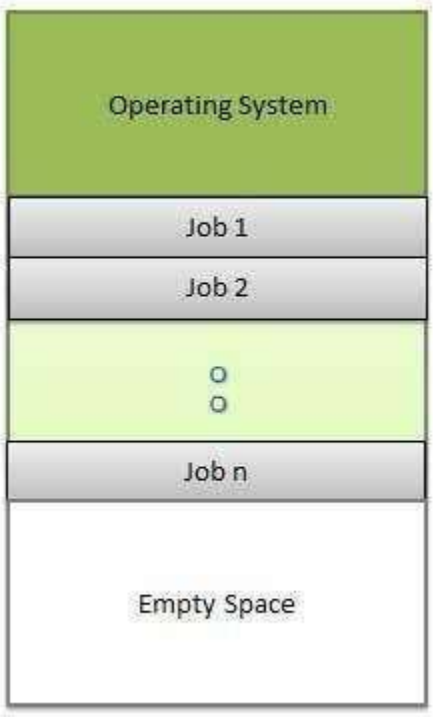
- A program that is loaded into memory and is executing is commonly referred to as a **process**.
- When a process executes, it typically executes for only a very short time before it either finishes or needs to perform I/O.
- Since interactive I/O typically runs at slower speeds, it may take a long time to complete. During this time, a CPU can be utilized by another process.
- The operating system allows the users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user.

- As the system switches CPU rapidly from one user/program to the next, each user is given the impression that he/she has his/her own CPU, whereas actually one CPU is being shared among many users.

Multiprogramming

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

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



An OS does the following activities related to multiprogramming.

- The operating system keeps several jobs in memory at a time.

- This set of jobs is a subset of the jobs kept in the job pool.
- The operating system picks and begins to execute one of the jobs in the memory.
- Multiprogramming operating systems monitor the state of all active programs and system resources using memory management programs to ensures that the CPU is never idle, unless there are no jobs to process. Advantage

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

Disadvantages

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

Interactivity

Interactivity refers to the ability of users to interact with a computer system. An Operating system does the following activities related to interactivity:

- Provides the user an interface to interact with the system.
- Manages input devices to take inputs from the user. For example, keyboard.
- Manages output devices to show outputs to the user. For example, Monitor.

The response time of the OS needs to be short, since the user submits and waits for the result.

Real-Time Systems

Real-time systems are usually dedicated, embedded systems. An operating system does the following activities related to real-time system activity.

- In such systems, Operating Systems typically read from and react to sensor data.
- The Operating system must guarantee response to events within fixed periods of time to ensure correct performance.

Distributed Environment

A distributed environment refers to multiple independent CPUs or processors in a computer system. An operating system does the following activities related to distributed environment:

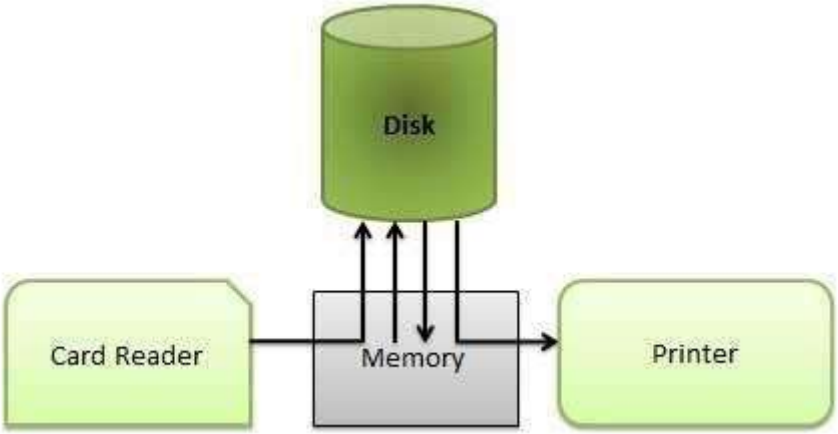
- The OS distributes computation logics among several physical processors.
- The processors do not share memory or a clock. Instead, each processor has its own local memory.
- The OS manages the communications between the processors. They communicate with each other through various communication lines.

Spooling

Spooling is an acronym for simultaneous peripheral operations on line. Spooling refers to putting data of various I/O jobs in a buffer. This buffer is a special area in memory or hard disk which is accessible to I/O devices.

An operating system does the following activities related to distributed environment:

- Handles I/O device data spooling as devices have different data access rates.
- Maintains the spooling buffer which provides a waiting station where data can rest while the slower device catches up.
- Maintains parallel computation because of spooling process as a computer can perform I/O in parallel fashion. It becomes possible to have the computer read data from a tape, write data to disk and to write out to a tape printer while it is doing its computing task.



Advantages

- The spooling operation uses a disk as a very large buffer.

Spooling is capable of overlapping I/O operation for one job with processor operations for another job.

Job control

job control refers to the control of multiple tasks or jobs on a computer system, ensuring that they each have access to adequate resources to perform correctly, that competition for limited resources does not cause a deadlock where two or more jobs are unable to complete, resolving such situations where they do occur, and terminating jobs that, for any reason, are not performing as expected.

Job control language (JCL)

Short for **Job Control Language**, **JCL** is a scripting language that is used to communicate with the operating system. Using JCL, a user can submit a statement to the operating system, which it then uses to execute a job. JCL also enables the user to view resources needed to run a job and minor control details. A language used to construct statements that identify a particular job to be run and specify the job's requirements to the operating system under which it will run.

A programming language used to specify the manner, timing, and other requirements of execution of a task or set of tasks submitted for execution, especially in background, on a multitasking computer; a programming language for controlling job execution.

Command language

Sometimes referred to as a **command script**, a **command language** is a language used for executing a series of commands instructions that would otherwise be executed at the prompt(text or symbols used to represent the system's readiness to perform the next). A good example of a command language is Microsoft Windows batch files(A **batch file** or **batch job** is a collection, or list, of commands that are processed in sequence often without requiring user input or intervention). Although command languages are useful for executing a series of commands, their functionality is limited to what is available at the command line which can make them easier to learn.

Advantages of command languages

Very easy for all types of users to write.

- Do not require the files to be compiled.
- Easy to modify and make additional commands.
- Very small files.
- Do not require any additional programs or files that are not already found on the operating system.

Disadvantages of command languages

- Can be limited when comparing with other programming languages or scripting languages.
- May not execute as fast as other languages or compiled programs.
- Some command languages often offer little more than using the commands available for the operating system used.

CHAPTER 2: PROCESS MANAGEMENT

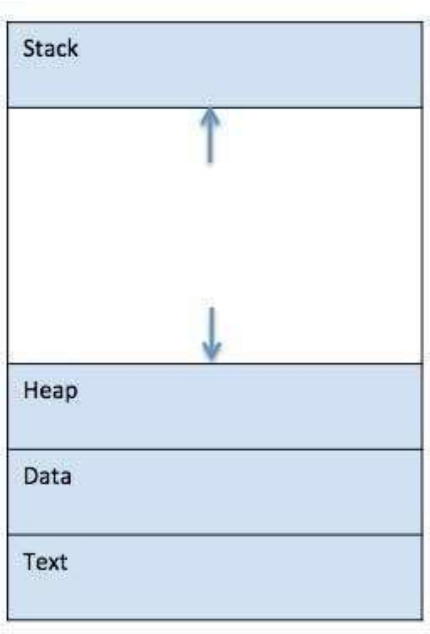
Definition of a Process and terms

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

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

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

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



S.N.	Component & Description
1	**Stack:** The process Stack contains the temporary data such as method/function parameters, return address, and local variables.

2	**Heap** This is a dynamically allocated memory to a process during its runtime.
3	**Text** This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.
4	**Data** This section contains the global and static variables.

The Process Model

Process models are processes of the same nature that are classified together into a model. Thus, a process model is a description of a process at the type level. Since the process model is at the type level, a process is an instantiation of it. The same process model is used repeatedly for the development of many applications and thus, has many instantiations. One possible use of a process model is to prescribe how things must/should/could be done in contrast to the process itself which is really what happens. A process model is roughly an anticipation of what the process will look like. What the process shall be will be determined during actual system development.

The goals of a process model are to be:

- Descriptive o Track what actually happens during a process
  - Take the point of view of an external observer who looks at the way a process has been performed and determines the improvements that must be made to make it perform more effectively or efficiently.
- Prescriptive o Define the desired processes and how they should/could/might be performed.
  - Establish rules, guidelines, and behavior patterns which, if followed, would lead to the desired process performance. They can range from strict enforcement to flexible guidance.
- Explanatory o Provide explanations about the rationale of processes.
  - Explore and evaluate the several possible courses of action based on rational arguments. o Establish an explicit link between processes and the requirements that the model needs to fulfill. o Pre-defines points at which data can be extracted for reporting purposes.

Process Levels

A **process** hierarchy is defined by its **levels** and the information given in these **levels**. It is key to have a defined information base on each **level** (e.g. a **process** step is always performed by a specific role instead of an abstract organizational unit), otherwise **process** levels are realized in **threads**.

Threads

Despite of the fact that a thread must execute in process, the process and its associated threads are different concept. Processes are used to group resources together and threads are the entities scheduled for execution on the CPU.

A thread is a single sequence stream within in a process. Because threads have some of the properties of processes, they are sometimes called *lightweight processes*. In a process, threads allow multiple executions of streams. In many respect, threads are popular way to improve application through parallelism. The CPU switches rapidly back and forth among the threads giving illusion that the threads are running in parallel. Like a traditional process i.e., process with one thread, a thread can be in any of several states (Running, Blocked, Ready or Terminated). Each thread has its own stack. Since thread will generally call different procedures and thus a different execution history. This is why thread needs its own stack. An operating system that has thread facility, the basic unit of CPU utilization is a thread. A thread has or consists of a program counter (PC), a register set, and a stack space. Threads are not independent of one other like processes as a result threads shares with other threads their code section, data section, OS resources also known as task, such as open files and signals.

Processes Vs Threads

As we mentioned earlier that in many respect threads operate in the same way as that of processes. Some of the similarities and differences are:

Similarities

- Like processes threads share CPU and only one thread active (running) at a time.
- Like processes, threads within a processes, threads within a processes execute sequentially.
- Like processes, thread can create children.
- And like process, if one thread is blocked, another thread can run.

Differences

- Unlike processes, threads are not independent of one another.
- Unlike processes, all threads can access every address in the task .
- Unlike processes, thread are design to assist one other. Note that processes might or might not assist one another because processes may originate from different users.

Why Threads?

Following are some reasons why we use threads in designing operating systems.

1. A process with multiple threads make a great server for example printer server.
2. Because threads can share common data, they do not need to use interprocess communication.
3. Because of the very nature, threads can take advantage of multiprocessors.

Threads are cheap in the sense that

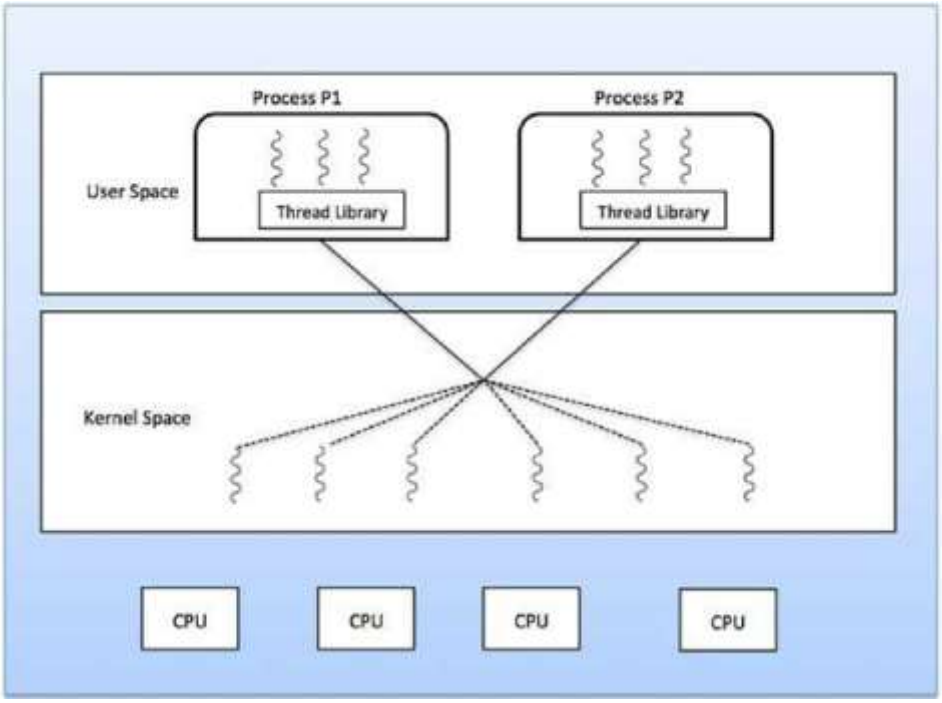
1. They only need a stack and storage for registers therefore, threads are cheap to create.
2. Threads use very little resources of an operating system in which they are working. That is, threads do not need new address space, global data, program code or operating system resources.
3. Context switching are fast when working with threads. The reason is that we only have to save and/or restore PC, SP and registers.

But this cheapness does not come free - the biggest drawback is that there is no protection between threads.

Thread levels (User-level & Kernel-level)

User-Level Threads

User-level threads implement in user-level libraries, rather than via systems calls, so thread switching does not need to call operating system and to cause interrupt to the kernel. In fact, the kernel knows nothing about user-level threads and manages them as if they were single-threaded processes.



Advantages:

The most obvious advantage of this technique is that a user-level threads package can be implemented on an Operating System that does not support threads. Some other advantages are

- User-level threads does not require modification to operating systems.

- Simple Representation:
  Each thread is represented simply by a PC, registers, stack and a small control block, all stored in the user process address space.
- Simple Management:
  This simply means that creating a thread, switching between threads and synchronization between threads can all be done without intervention of the kernel.
- Fast and Efficient:
  Thread switching is not much more expensive than a procedure call.

Disadvantages:

- There is a lack of coordination between threads and operating system kernel. Therefore, process as whole gets one time slice irrespect of whether process has one thread or 1000 threads within. It is up to each thread to relinquish control to other threads.
- User-level threads requires non-blocking systems call i.e., a multithreaded kernel. Otherwise, entire process will blocked in the kernel, even if there are runable threads left in the processes. For example, if one thread causes a page fault, the process blocks.

Kernel-Level Threads

In this method, the kernel knows about and manages the threads. No runtime system is needed in this case. Instead of thread table in each process, the kernel has a thread table that keeps track of all threads in the system. In addition, the kernel also maintains the traditional process table to keep track of processes. Operating Systems kernel provides system call to create and manage threads.

Advantages:

- Because kernel has full knowledge of all threads, Scheduler may decide to give more time to a process having large number of threads than process having small number of threads.
- Kernel-level threads are especially good for applications that frequently block.

Disadvantages:

- The kernel-level threads are slow and inefficient. For instance, threads operations are hundreds of times slower than that of user-level threads.
- Since kernel must manage and schedule threads as well as processes. It require a full thread control block (TCB) for each thread to maintain information about threads. As a result there is significant overhead and increased in kernel complexity.

Multithreading Models

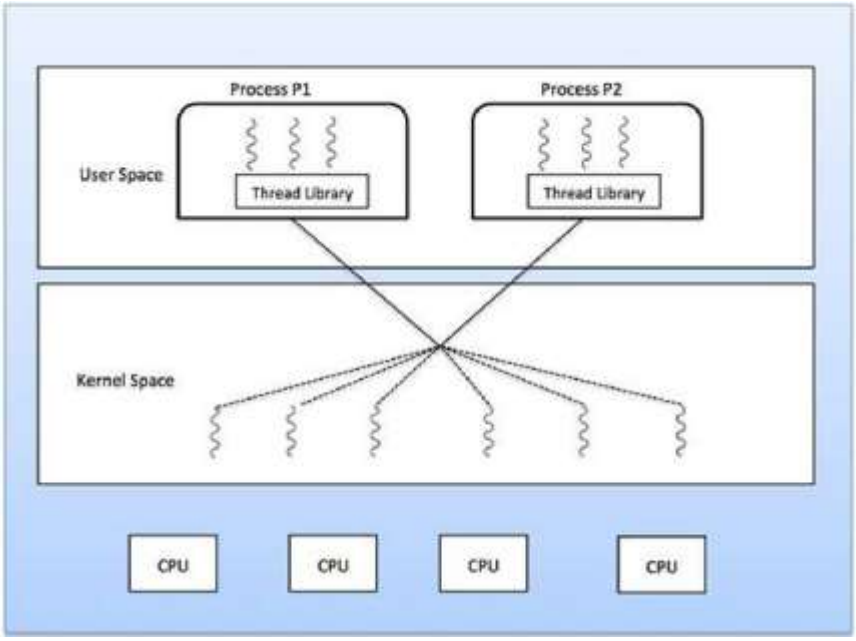
Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the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

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

Many to Many Model

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

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

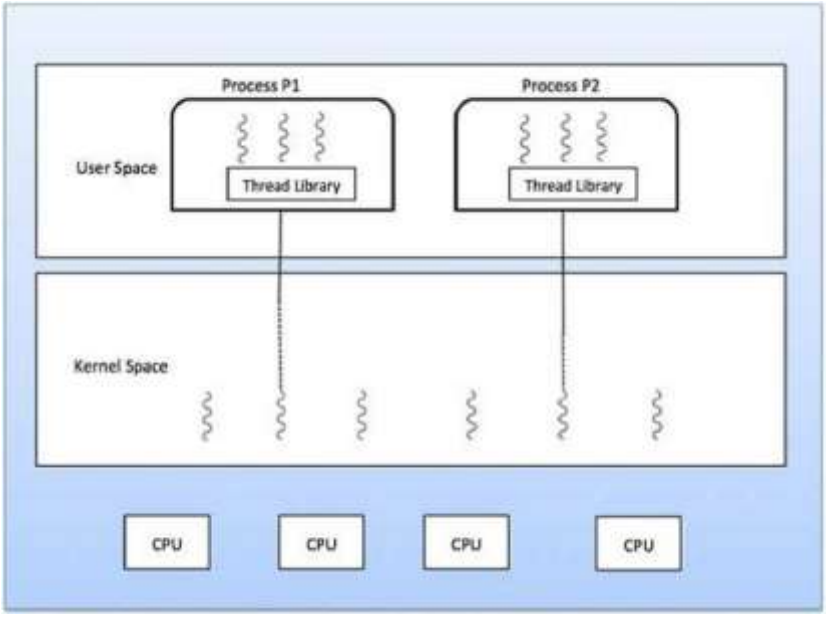


Many to One Model

Many-to-one model maps many user level threads to one Kernel-level thread. Thread management is done in user space by the thread library. When thread makes a blocking system call, the entire

process will be blocked. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

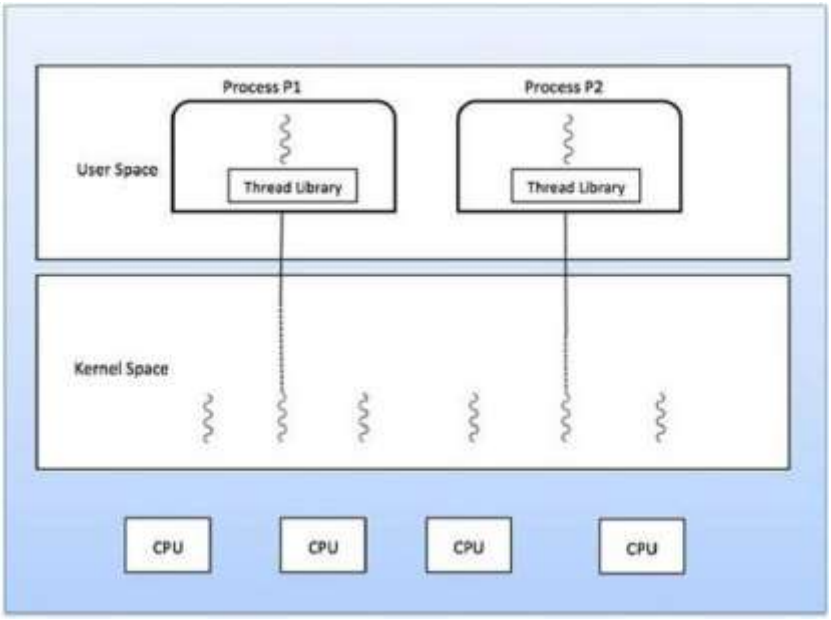
If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.



One to One Model

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

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



Advantages of Threads over Multiple Processes

- **Context Switching** Threads are very inexpensive to create and destroy, and they are inexpensive to represent. For example, they require space to store, the PC, the SP, and the general-purpose registers, but they do not require space to share memory information, Information about open files of I/O devices in use, etc. With so little context, it is much faster to switch between threads. In other words, it is relatively easier for a context switch using threads.
- **Sharing** Treads allow the sharing of a lot resources that cannot be shared in process, for example, sharing code section, data section, Operating System resources like open file etc.

Disadvantages of Threads over Multiprocesses

- **Blocking** The major disadvantage if that if the kernel is single threaded, a system call of one thread will block the whole process and CPU may be idle during the blocking period.
- **Security** Since there is, an extensive sharing among threads there is a potential problem of security. It is quite possible that one thread over writes the stack of another thread (or damaged shared data) although it is very unlikely since threads are meant to cooperate on a single task.

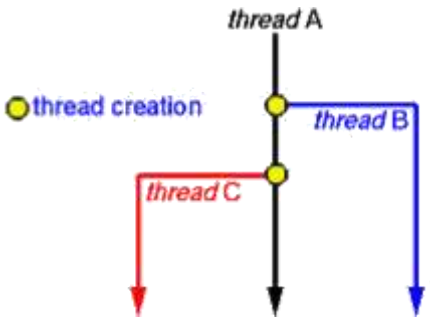
Application that Benefits from Threads

A proxy server satisfying the requests for a number of computers on a LAN would be benefited by a multi-threaded process. In general, any program that has to do more than one task at a time could benefit from multitasking. For example, a program that reads input, process it, and outputs could have three threads, one for each task.

Application that cannot benefit from Threads

Any sequential process that cannot be divided into parallel task will not benefit from thread, as they would block until the previous one completes. For example, a program that displays the time of the day would not benefit from multiple threads.

Resources used in Thread Creation and Process Creation



When a new thread is created it shares its code section, data section and operating system resources like open files with other threads. But it is allocated its own stack, register set and a program counter.

The creation of a new process differs from that of a thread mainly in the fact that all the shared resources of a thread are needed explicitly for each process. So though two processes may be running the same piece of code they need to have their own copy of the code in the main memory to be able to run. Two processes also do not share other resources with each other. This makes the creation of a new process very costly compared to that of a new thread.

Context Switch

To give each process on a multiprogrammed machine a fair share of the CPU, a hardware clock generates interrupts periodically. This allows the operating system to schedule all processes in main memory (using scheduling algorithm) to run on the CPU at equal intervals. Each time a clock interrupt occurs, the interrupt handler checks how much time the current running process has used. If it has used up its entire time slice, then the CPU scheduling algorithm (in kernel) picks a different process to run. Each switch of the CPU from one process to another is called a context switch.

Major Steps of Context Switching

- The values of the CPU registers are saved in the process table of the process that was running just before the clock interrupt occurred.
- The registers are loaded from the process picked by the CPU scheduler to run next.

In a multiprogrammed uniprocessor computing system, context switches occur frequently enough that all processes appear to be running concurrently. If a process has more than one thread, the Operating System can use the context switching technique to schedule the threads so they appear to execute in parallel. This is the case if threads are implemented at the kernel level. Threads can also be implemented entirely at the user level in run-time libraries. Since in this case no thread scheduling is provided by the Operating System, it is the responsibility of the programmer to yield the CPU frequently enough in each thread so all threads in the process can make progress.

Action of Kernel to Context Switch Among Threads

The threads share a lot of resources with other peer threads belonging to the same process. So a context switch among threads for the same process is easy. It involves switch of register set, the program counter and the stack. It is relatively easy for the kernel to accomplished this task.

Action of kernel to Context Switch Among Processes

Context switches among processes are expensive. Before a process can be switched its process control block (PCB) must be saved by the operating system. The PCB consists of the following information:

- The process state.
- The program counter, PC.
- The values of the different registers.
- The CPU scheduling information for the process.
- Memory management information regarding the process.
- Possible accounting information for this process.
- I/O status information of the process.

When the PCB of the currently executing process is saved the operating system loads the PCB of the next process that has to be run on CPU. This is a heavy task and it takes a lot of time.

Process States Life Cycle

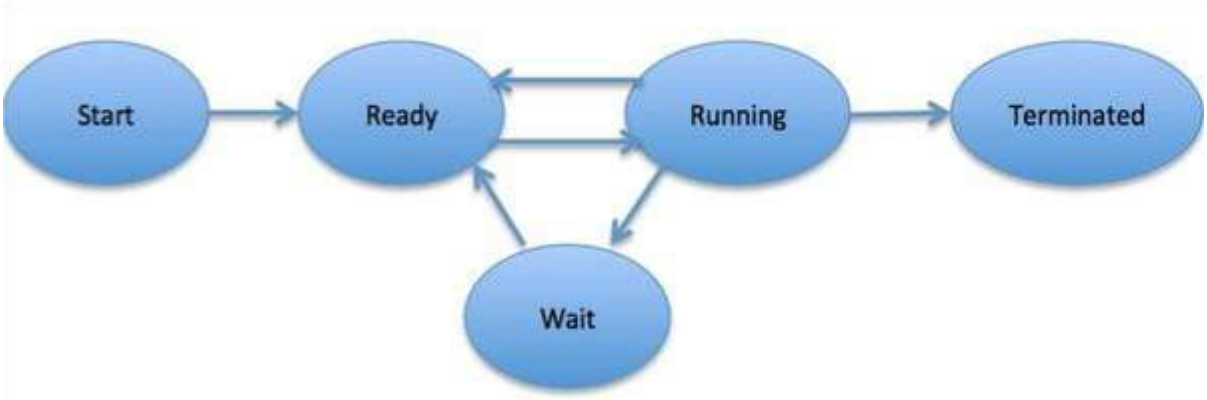
When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardized.

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

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

2	**Ready** The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after **Start** state or while running it by but interrupted by the scheduler to assign CPU to some other process.
3	**Running** Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.
4	**Waiting** Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.
5	**Terminated or Exit** Once the process finishes its execution, or it is terminated by the
	operating system, it is moved to the terminated state where it waits to be removed from main memory.

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



Process Control Block (PCB)

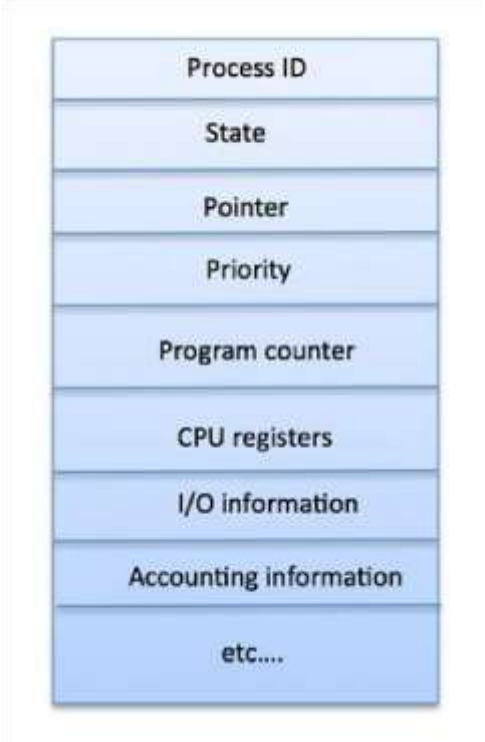
A Process Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID). A PCB keeps all the information needed to keep track of a process as listed below in the table −

S.N.	Information & Description

7	**Memory management information:** This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.
8	
9	**CPU registers:** Various CPU registers where process need to be stored for execution for running state.
10	**Accounting information:** This includes the amount of CPU used for process execution, time limits, execution ID etc.
	IO status information: This includes a list of I/O devices allocated to the process.

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

1	**Process State:** The current state of the process i.e., whether it is ready, running, waiting, or whatever.
2	**Process privileges:** This is required to allow/disallow access to system resources.
3	**Process ID:** Unique identification for each of the process in the operating system.
4	**Pointer:** A pointer to parent process.
5	**Program Counter:** Program Counter is a pointer to the address of the next instruction to be executed for this process.
6	



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

Inter-process communication

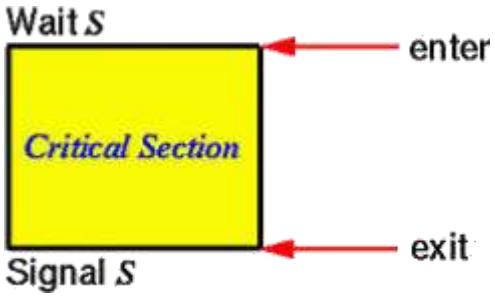
Inter-process communication or **inter-process communication** (**IPC**) refers specifically to the mechanisms an operating system provides to allow the processes to manage shared data.

Race Conditions

In operating systems, processes that are working together share some common storage (main memory, file etc.) that each process can read and write. When two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called race conditions. Concurrently executing threads that share data need to synchronize their operations and processing in order to avoid race condition on shared data. Only one „customer" thread at a time should be allowed to examine and update the shared variable.
Race conditions are also possible in Operating Systems. If the ready queue is implemented as a linked list and if the ready queue is being manipulated during the handling of an interrupt, then interrupts must be disabled to prevent another interrupt before the first one completes. If interrupts are not disabled than the linked list could become corrupt.

Critical Section

How to avoid race conditions?



The key to preventing trouble involving shared storage is find some way to prohibit more than one process from reading and writing the shared data simultaneously. That part of the program where the shared memory is accessed is called the *Critical Section*. To avoid race conditions and flawed results, one must identify codes in *Critical Sections* in each thread. The characteristic properties of the code that form a *Critical Section* are

- Codes that reference one or more variables in a "read-update-write" fashion while any of those variables is possibly being altered by another thread.
- Codes that alter one or more variables that are possibly being referenced in "read-updata-write" fashion by another thread.

- Codes use a data structure while any part of it is possibly being altered by another thread.
- Codes alter any part of a data structure while it is possibly in use by another thread.

Here, the important point is that when one process is executing shared modifiable data in its critical section, no other process is to be allowed to execute in its critical section. Thus, the execution of critical sections by the processes is mutually exclusive in time.

Mutual Exclusion

A way of making sure that if one process is using a shared modifiable data, the other processes will be excluded from doing the same thing.

Formally, while one process executes the shared variable, all other processes desiring to do so at the same time moment should be kept waiting; when that process has finished executing the shared variable, one of the processes waiting; while that process has finished executing the shared variable, one of the processes waiting to do so should be allowed to proceed. In this fashion, each process executing the shared data (variables) excludes all others from doing so simultaneously. This is called Mutual Exclusion.

Note that mutual exclusion needs to be enforced only when processes access shared modifiable data - when processes are performing operations that do not conflict with one another they should be allowed to proceed concurrently.

Mutual Exclusion Conditions

If we could arrange matters such that no two processes were ever in their critical sections simultaneously, we could avoid race conditions. We need four conditions to hold to have a good solution for the critical section problem (mutual exclusion).

- No two processes may at the same moment inside their critical sections.
- No assumptions are made about relative speeds of processes or number of CPUs.
- No process should outside its critical section should block other processes.
- No process should wait arbitrary long to enter its critical section.

Proposals for Achieving Mutual Exclusion

The mutual exclusion problem is to devise a pre-protocol (or entry protocol) and a post-protocol (or exist protocol) to keep two or more threads from being in their critical sections at the same time. Tanenbaum examine proposals for critical-section problem or mutual exclusion problem.

Problem

When one process is updating shared modifiable data in its critical section, no other process should allowed to enter in its critical section.
Proposal 1 -Disabling Interrupts (Hardware Solution)

Each process disables all interrupts just after entering in its critical section and re-enable all interrupts just before leaving critical section. With interrupts turned off the CPU could not be switched to other process. Hence, no other process will enter its critical and mutual exclusion achieved.

Conclusion
Disabling interrupts is sometimes a useful interrupts is sometimes a useful technique within the kernel of an operating system, but it is not appropriate as a general mutual exclusion mechanism for users process. The reason is that it is unwise to give user process the power to turn off interrupts.

Proposal 2 - Lock Variable (Software Solution)

In this solution, we consider a single, shared, (lock) variable, initially 0. When a process wants to enter in its critical section, it first test the lock. If lock is 0, the process first sets it to 1 and then enters the critical section. If the lock is already 1, the process just waits until (lock) variable becomes 0. Thus, a 0 means that no process in its critical section, and 1 means hold your horses - some process is in its critical section.

Conclusion
The flaw in this proposal can be best explained by example. Suppose process A sees that the lock is 0. Before it can set the lock to 1 another process B is scheduled, runs, and sets the lock to 1. When the process A runs again, it will also set the lock to 1, and two processes will be in their critical section simultaneously.

Proposal 3 - Strict Alteration

In this proposed solution, the integer variable 'turn' keeps track of whose turn is to enter the critical section. Initially, process A inspect turn, finds it to be 0, and enters in its critical section. Process B also finds it to be 0 and sits in a loop continually testing 'turn' to see when it becomes 1.Continuously testing a variable waiting for some value to appear is called the *Busy-Waiting*.

Conclusion

Taking turns is not a good idea when one of the processes is much slower than the other. Suppose process 0 finishes its critical section quickly, so both processes are now in their noncritical section. This situation violates above mentioned condition 3.

Using Systems calls 'sleep' and 'wakeup'

Basically, what above mentioned solution do is this: when a processes wants to enter in its critical section , it checks to see if then entry is allowed. If it is not, the process goes into tight loop and waits (i.e., start busy waiting) until it is allowed to enter. This approach waste CPUtime.

Now look at some interprocess communication primitives is the pair of steep-wakeup.

- Sleep: It is a system call that causes the caller to block, that is, be suspended until some other process wakes it up.
- Wakeup: It is a system call that wakes up the process.

Both 'sleep' and 'wakeup' system calls have one parameter that represents a memory address used to match up 'sleeps' and 'wakeups' .

The Bounded Buffer Producers and Consumers

The bounded buffer producers and consumers assumes that there is a fixed buffer size i.e., a finite numbers of slots are available.

Statement

To suspend the producers when the buffer is full, to suspend the consumers when the buffer is empty, and to make sure that only one process at a time manipulates a buffer so there are no race conditions or lost updates.

As an example how sleep-wakeup system calls are used, consider the producer-consumer problem also known as bounded buffer problem.

Two processes share a common, fixed-size (bounded) buffer. The producer puts information into the buffer and the consumer takes information out.
Trouble arises when

1. The producer wants to put a new data in the buffer, but buffer is already full.
   Solution: Producer goes to sleep and to be awakened when the consumer has removed data.
2. The consumer wants to remove data the buffer but buffer is already empty.
   Solution: Consumer goes to sleep until the producer puts some data in buffer and wakes consumer up.

Conclusion

This approaches also leads to same race conditions we have seen in earlier approaches. Race condition can occur due to the fact that access to 'count' is unconstrained. The essence of the problem is that a wakeup call, sent to a process that is not sleeping, is lost.

Semaphore & Monitor Definition of Semaphore

Being a process synchronization tool, **Semaphore** is an **integer variable S.** This integer variable S is initialized to the **number of resources** present in the system. The value of semaphore S can be modified only by two functions **wait**() and **signal**() apart from initialization.

The wait() and signal() operation modifies the value of the semaphore S indivisibly. Which means when a process is modifying the value of the semaphore, no other process can simultaneously modify the value of the semaphore. Further, the operating system distinguishes the semaphore in two categories Counting semaphores and Binary semaphore.

In **Counting Semaphore**, the value of semaphore S is initialized to the number of resources present in the system. Whenever a process wants to access the shared resources, it performs **wait**() operation on the semaphore which **decrements** the value of semaphore by one. When it releases the shared resource, it performs a **signal**() operation on the semaphore which **increments** the value of semaphore by one. When the semaphore count goes to **0**, it means **all resources are occupied** by the processes. If a process need to use a resource when semaphore count is 0, it executes wait() and get **blocked** until a process utilizing the shared resources releases it and the value of semaphore becomes greater than 0.

In **Binary semaphore**, the value of semaphore ranges between 0 and 1. It is similar to mutex lock, but mutex is a locking mechanism whereas, the semaphore is a signalling mechanism. In binary semaphore, if a process wants to access the resource it performs wait() operation on the semaphore and **decrements** the value of semaphore from 1 to 0. When process releases the resource, it performs a **signal**() operation on the semaphore and increments its value to 1. If the value of the semaphore is 0 and a process want to access the resource it performs wait() operation and block itself till the current process utilizing the resources releases the resource.

Definition of Monitor

To overcome the timing errors that occurs while using semaphore for process synchronization, the researchers have introduced a high-level synchronization construct i.e. the **monitor type**. A monitor type is **an abstract data type** that is used for process synchronization.

Being an abstract data type monitor type contains the **shared data variables** that are to be shared by all the processes and some programmer-defined **operations** that allow processes to execute in mutual exclusion within the monitor. A process can **not directly access** the shared data variable in

the monitor; the process has to access it **through procedures** defined in the monitor which allow only one process to access the shared variables in a monitor at a time.

The syntax of monitor is as follow:

```
1. monitor monitor_name
2. {
3. //shared variable declarations
4. procedure P1 ( . . . ) {
5. }
6. procedure P2 ( . . . ) { 7. }
8. procedure Pn ( . . . ) {
9. }
10. initialization code ( . . . ) {
11. }
12. }
```

A monitor is a construct such as only one process is active at a time within the monitor. If other process tries to access the shared variable in monitor, it gets blocked and is lined up in the queue to get the access to shared data when previously accessing process releases it.

Conditional variables were introduced for additional synchronization mechanism. The conditional variable **allows a process to wait inside the monitor** and allows a waiting process to resume immediately when the other process releases the resources.

The **conditional variable** can invoke only two operation **wait**() and **signal**(). Where if a process **P invokes a wait()** operation it gets suspended in the monitor till other process **Q invoke signal()** operation i.e. a signal() operation invoked by a process resumes the suspended process.

Key Differences Between Semaphore and Monitor

1. The basic difference between semaphore and monitor is that the **semaphore** is an **integer variable S** which indicate the number of resources available in the system whereas, the **monitor** is the **abstract data type** which allows only one process to execute in critical section at a time.
2. The value of semaphore can be modified by **wait()** and **signal()** operation only. On the other hand, a monitor has the shared variables and the procedures only through which shared variables can be accessed by the processes.
3. In Semaphore when a process wants to access shared resources the process performs **wait**() operation and block the resources and when it release the resources it performs

signal() operation. In monitors when a process needs to access shared resources, it has to access them through procedures in monitor.

4. Monitor type has **condition variables** which semaphore does not have.

Process scheduling

Definition

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

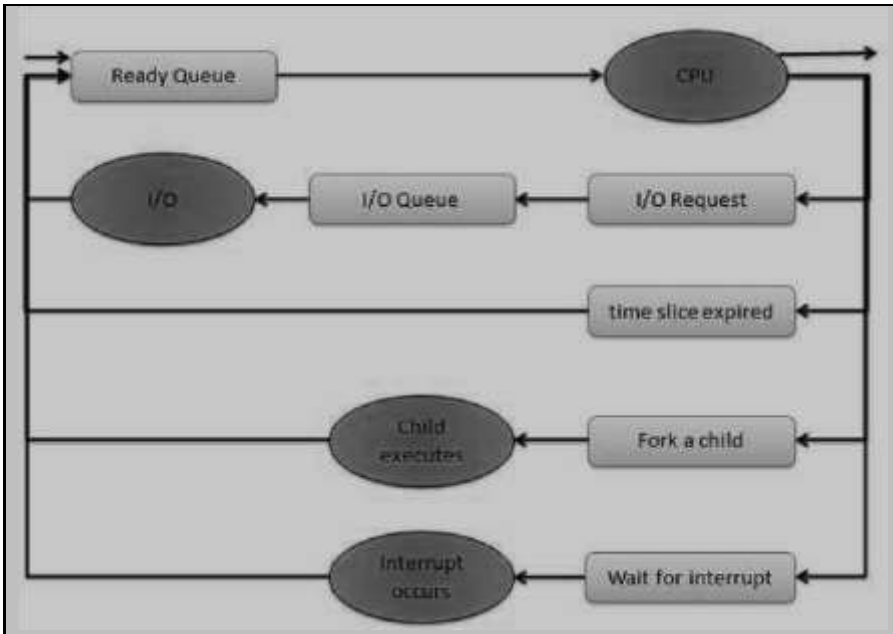
Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

Process Scheduling Queues

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

Note: Scheduling queues refers to queues of processes or devices. When the process enters into the system, then this process is put into a job queue. This queue consists of all processes in the system. The operating system also maintains other queues such as device queue. Device queue is a queue for which multiple processes are waiting for a particular I/O device. Each device has its own device queue.
This figure shows the queuing diagram of process scheduling.
- Queue is represented by rectangular box.
- The circles represent the resources that serve the queues.
- The arrows indicate the process flow in the system.

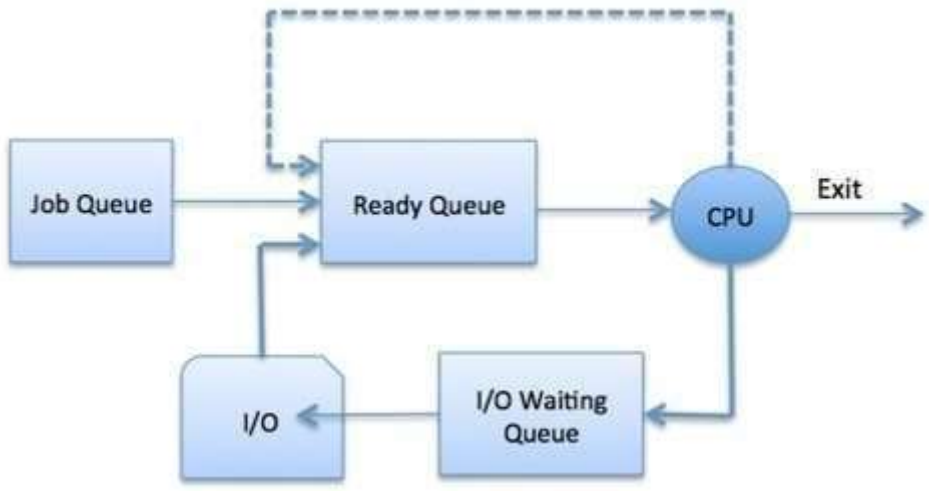
A newly arrived process is put in the ready queue. Processes waits in ready queue for allocating the CPU. Once the CPU is assigned to a process, then that process will execute. While executing the process, any one of the following events can occur.
- The process could issue an I/O request and then it would be placed in an I/O queue.
- The process could create new sub process and will wait for its termination.



The process could be removed forcibly from the CPU, as a result of interrupt and put back in the ready queue.

The Operating System maintains the following important process scheduling queues:

- **Job queue -** This queue keeps all the processes in the system.
- **Ready queue -** This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
- **Device queues -** The processes which are blocked due to unavailability of an I/O device constitute this queue.



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

Two-State Process Model

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

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

Process scheduling and Job scheduling

Job scheduling is the process of allocating system resources to many different tasks by an operating system (OS). The system handles prioritized **job** queues that are awaiting CPU time and it should determine which **job** to be taken from which queue and the amount of time to be

allocated for the **job**. This type of scheduling makes sure that all jobs are carried out fairly and on time.

Job scheduling is performed using job schedulers. Job schedulers are programs that enable scheduling and, at times, track computer "batch" jobs, or units of work like the operation of a payroll program. Job schedulers have the ability to start and control jobs automatically by running prepared job-control-language statements or by means of similar communication with a human operator. Generally, the present-day job schedulers include a graphical user interface (GUI) along with a single point of control.

Organizations wishing to automate unrelated IT workload could also use more sophisticated attributes from a job scheduler, for example:

- Real-time scheduling in accordance with external, unforeseen events
- Automated restart and recovery in case of failures
- Notifying the operations personnel
- Generating reports of incidents
- Audit trails meant for regulation compliance purposes

In-house developers can write these advanced capabilities; however, these are usually offered by providers who are experts in systems-management software.

In scheduling, many different schemes are used to determine which specific job to run. Some parameters that may be considered are as follows:

- Job priority
- Availability of computing resource
- License key if the job is utilizing a licensed software
- Execution time assigned to the user
- Number of parallel jobs permitted for a user
- Projected execution time
- Elapsed execution time
- Presence of peripheral devices
- Number of cases of prescribed events

What is the difference between job scheduling and process scheduling?

Jobs or processes are the same. Whereas, Job scheduler and CPU scheduler are two different terms. Job scheduler is also called long term scheduler and CPU/Process scheduler is called short term scheduler

Schedulers

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

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

Long-Term Scheduler

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

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

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

Short-Term Scheduler

It is also called as **CPU scheduler**. Its main objective is to increase system performance in accordance with the chosen set of criteria. It is the change of ready state to running

S.N.	Long-Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
2	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
3	It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
4	It is almost absent or minimal in time sharing	It is also minimal in time sharing system	It is a part of Time sharing systems.

	system		
5	It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

Medium-Term Scheduler

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

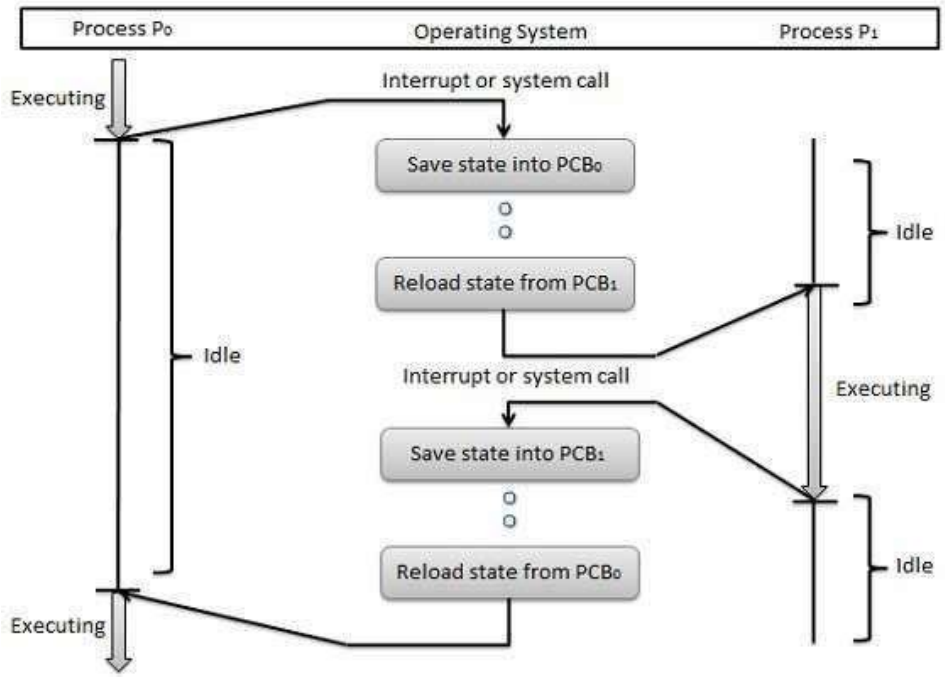
A running process may become suspended if it makes an I/O request. A suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other processes, the suspended process is moved to the secondary storage. This process is called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

Comparison among Schedulers

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

Context Switch

A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.
When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.



Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers. When the process is switched, the following information is stored for later use.

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

Process Scheduling Algorithms

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

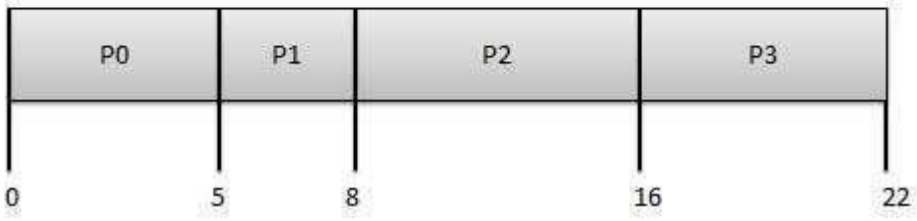
- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-Next (SJN) Scheduling
- Priority Scheduling
- Shortest Remaining Time
- Round Robin(RR) Scheduling
- Multiple-Level Queues Scheduling

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

First Come, First Served (FCFS)

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

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



Wait time of each process is as follows:

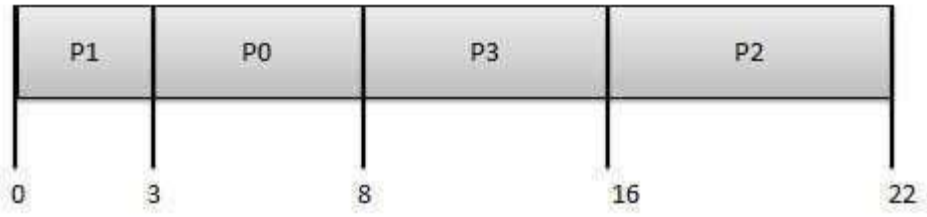
Process	Wait Time : Service Time - Arrival Time
P0	0 - 0 = 0
P1	5 - 1 = 4
P2	8 - 2 = 6
P3	16 - 3 = 13

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

Shortest Job Next (SJN)

- This is also known as **shortest job first**, or SJF.
- This is a non-preemptive scheduling algorithm.
- Best approach to minimize waiting time.
- Easy to implement in Batch systems where required CPU time is known in advance.
- Impossible to implement in interactive systems where the required CPU time is not known.
- The processer should know in advance how much time a process will take.

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



Wait time of each process is as follows:

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

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

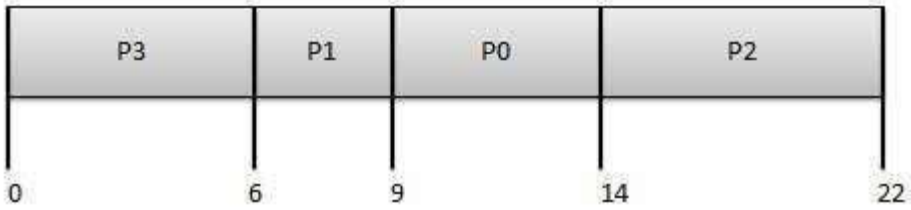
Priority Based Scheduling

- Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems. □

□

- Each process is assigned a priority. Process with highest priority is to be executed first and so on. □
- Processes with same priority are executed on first come first served basis. □
- Priority can be decided based on memory requirements, time requirements or any other resource requirement. □

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



Wait time of each process is as follows:

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

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

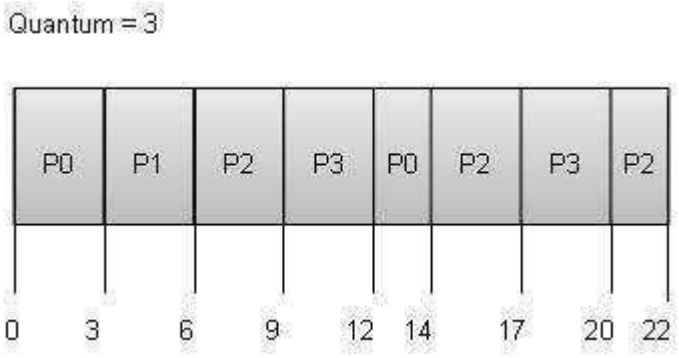
Shortest Remaining Time

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

Round Robin Scheduling

- Round Robin is a preemptive process scheduling algorithm.
- Each process is provided a fix time to execute; it is called a **quantum**.
- Once a process is executed for a given time period, it is preempted and other process executes for a given time period.

- Context switching is used to save states of preempted processes.



Wait time of each process is as follows:

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

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

Multiple-Level Queues Scheduling

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

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

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

Deadlock

Introduction

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a wait state. It may happen that waiting processes will never again change state,

because the resources they have requested are held by other waiting processes. This situation is called deadlock.

If a process requests an instance of a resource type, the allocation of any instance of the type will satisfy the request. If it will not, then the instances are not identical, and the resource type classes have not been defined properly.
A process must request a resource before using it, and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task.
Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. **Request**: If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
2. **Use:** The process can operate on the resource.
3. **Release**: The process releases the resource

Deadlock Characterization

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

Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system: **1. Mutual exclusion:** At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. Hold and wait: There must exist a process that is holding at least one resource and is waiting to acquire additional resources that are currently being held by other processes.

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

4. Circular wait: There must exist a set $\{P_0, P_1, ..., P_n\}$ of waiting processes such that P0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …., P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a system resourceallocation graph. The set of vertices V is partitioned into two different types of nodes P = $\{P_1, P_2,$ … $P_n\}$ the set consisting of all the active processes in the system; and R = $\{R_1, R_2, …, R_1\}$, the set consisting of all resource types in the system.
A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$, it signifies that process Pi requested an instance of resource type R_j and is currently waiting for that resource. A directed

edge from resource type R_j to process P_i is denoted by R_j_ P_i it signifies that an instance of resource type R_j has been allocated to process P_i. A directed edge P_i_ R_j is called a request edge; a directed edge R_j _ Pi is called an assignment edge.
When process Pi requests an instance of resource type R_j, a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is instantaneously transformed to an assignment edge. When the process no longer needs access to the, resource it releases the resource, and as a result the assignment edge is deleted.

Definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If, on the other hand, the graph contains the cycle, then a deadlock must exist.
If each resource type has several instances, 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.
A set of vertices *V* and a set of edges *E*. V is partitioned into two types: o *P* = $\{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.

o $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system. request edge – directed edge P_1 R_j
assignment edge – directed edge R_j P_i

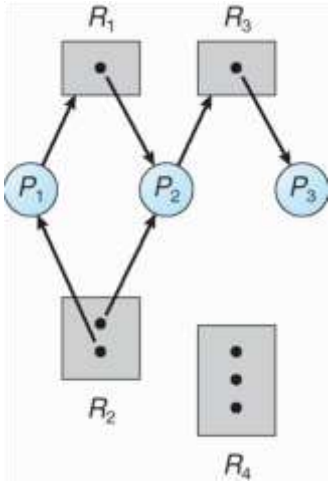


Fig. Resource Allocation Graph

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

Suppose that process P_3 requests an instance of resource type R_2 Since no resource instance is currently available, a request edge $P_3 \rightarrow R_2$ is added to the graph. At this point, two minimal cycles exist in the system:

P1 → R1 → P2 → R3 → P3 → R2 → P1

P2 → R3 → P3 → R2 → P2

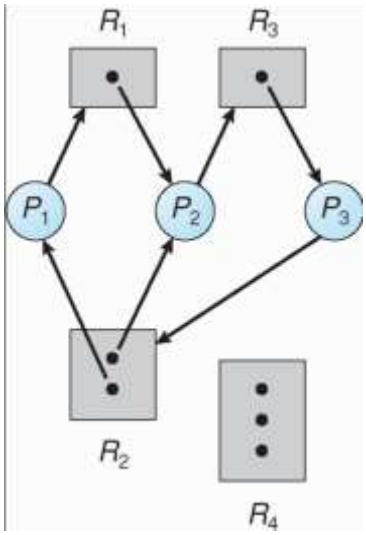


Fig. Resource Allocation Graph with Deadlock

Processes P_1, P_2, and P_3 are deadlocked. Process P_2 is waiting for the resource R_3, which is held by process P_3. Process P_3, on the other hand, is waiting for either process P_1 or process P_2 to release resource R_2. In addition, process P_I is waiting for process P_2 to release resource R_1.

Method for Handling Deadlock //Detection

There are three different methods for dealing with the deadlock problem:

- We can use a protocol to ensure that the system will never enter a deadlock state.
- We can allow the system to enter a deadlock state and then recover. • We can ignore the problem all together, and pretend that deadlocks never occur in the system. This solution is the one used by most operating systems, including UNIX.

Deadlock avoidance, on the other hand, requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, we can decide for each request whether or not the process should wait. Each request requires that the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process, to decide whether the current request can be satisfied or must be delayed. If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur If a system does not ensure that a deadlock will never occur, and also does not provide a mechanism for deadlock detection and recovery, then we may arrive at a

situation where the system is in a deadlock state yet has no way of recognizing what has happened.

1. Deadlock Prevention

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

a) Mutual Exclusion

The mutual-exclusion condition must hold for non-sharable resources. For example, a printer cannot be simultaneously shared by several processes. Sharable resources, on the other hand, do not require mutually exclusive access, and thus cannot be involved in a deadlock. **b) Hold and Wait**

1. When whenever a process requests a resource, it does not hold any other resources. One protocol that be used requires each process to request and be allocated all its resources before it begins execution.

2. An alternative protocol allows a process to request resources only when the process has none. A process may request some resources and use them. Before it can request any additional resources, however it must release all the resources that it is currently allocated here are two main disadvantages to these protocols. First, resource utilization may be low, since many of the resources may be allocated but unused for a long period. In the example given, for instance, we can release the tape drive and disk file, and then again request the disk file and printer, only if we can be sure that our data will remain on the disk file. If we cannot be assured that they will, then we must request all resources at the beginning for both protocols.

Second, starvation is possible. **c)**

No Preemption

If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are preempted. That is this resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting. **d) Circular Wait**

Circular-wait condition never holds is to impose a total ordering of all resource types, and to require that each process requests resources in an increasing order of enumeration.

Let $R = \{R_1, R_2, ..., R_n\}$ be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering. Formally, we define a one-to-one function F: R _ N, where N is the set of natural numbers.

2. Deadlock Avoidance

Prevent deadlocks requests can be made. The restraints ensure that at least one of the necessary conditions for deadlock cannot occur, and, hence, that deadlocks cannot hold. Possible side effects

of preventing deadlocks by this, melted, however, are Tow device utilization and reduced system throughput.
An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested. For example, in a system with one tape drive and one printer, we might be told that process P will request first the tape drive, and later the printer, before releasing both resources. Process Q on the other hand, will request first the printer, and then the tape drive. With this knowledge of the complete sequence of requests and releases for each process we can decide for each request whether or not the process should wait.
A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular wait condition. The resource allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes. **a. Safe State**
A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence. A sequence of processes $<P_1, P_2, .. P_n>$ is a safe sequence for the current allocation state if, for each Pi the resources that P_j can still request can be satisfied by the currently available resources plus the resources held by all the P_j, with $j < i$. In this situation, if the resources that process P_i needs are not immediately available, then Pi can wait until all P_j have finished. When they have finished, Pi can obtain all of its needed resources, complete its designated task return its allocated resources, and terminate. When P_i terminates, $P_i + 1$ can obtain its needed resources, and so on.

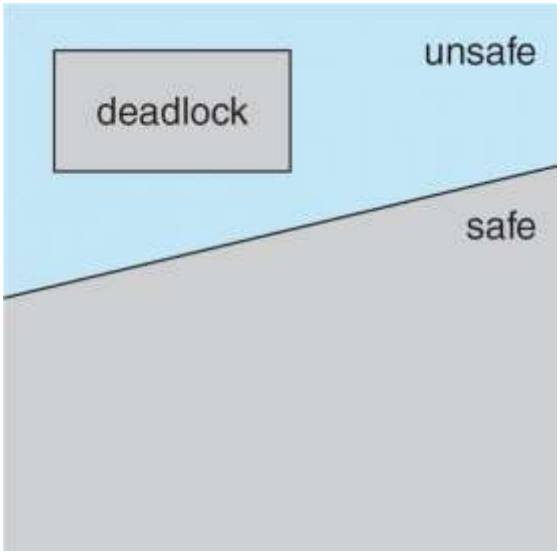


Fig. Safe, Unsafe & Deadlock State

If no such sequence exists, then the system state is said to be unsafe.

b. Resource-Allocation Graph Algorithm

Suppose that process Pi requests resource R_j. The request can be granted only if converting the request edge $P_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow P_i$ does not result in the formation of a cycle in the resource-allocation graph. **c. Banker's Algorithm**

The resource-allocation graph algorithm is not applicable to a resource-allocation system with multiple instances of each resource type. The deadlock-avoidance algorithm that we describe next is applicable to such a system, but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the banker's algorithm.

3. Deadlock Detection

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

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

a) Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph. We obtain this graph from the resource-allocation graph by removing the nodes of type resource and collapsing the appropriate edges.

b) Several Instances of a Resource Type

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

The algorithm used are :

- **Available**: A vector of length m indicates the number of available resources of each type.
- **Allocation**: An n x m matrix defines the number of resources of each type currently allocated to each process.
- **Request**: An n x m matrix indicates the current request of each process. If Request [i, j] = k, then process P, is requesting k more instances of resource type R_j.

c) Detection-Algorithm Usage

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken.

4. Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives exist. One possibility is to inform the operator that a deadlock has spurred, and to let the operator deal with the deadlock manually. The other possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to preempt some resources from one or more of the deadlocked processes. **a) Process Termination**

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

- **Abort all deadlocked processes**: This method clearly will break the dead – lock cycle, but at a great expense, since these processes may have computed for a long time, and the results of these partial computations must be discarded, and probably must be recomputed.
- **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 processes are still deadlocked. **b) Resource Preemption**

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until he deadlock cycle is broken.
The three issues are considered to recover from deadlock

1. **Selecting a victim**
2. **Rollback**
3. **Starvation**

Summary

A deadlocked state occurs when two or more processes are waiting indefinitely for an event that can be caused only one of the waiting processes. There are three principal methods for dealing with deadlocks:
Use some protocol to prevent or avoid deadlocks, entering that the system will never enter a deadlocked state.
Allow the system to enter a deadlocked state, detect it, and then recover.
Ignore the problem altogether and pretend that deadlocks never occur in the system.

Deadlock prevention is a set of methods for ensuring that at least one of the necessary condition cannot hold. Deadlock avoidance requires additional information about how resources are to be requested. Deadlock avoidance algorithm dynamically examines the resource allocation state to ensure that a circular wait condition can never exist. Deadlock occurs only when some process makes a request that cannot be granted immediately.

Description of Error Diagnosis

Diagnostic error can be defined as a *diagnosis* that is missed, wrong or delayed, as detected by some subsequent definitive test or finding. In computing, operating system failure result to error messages or sound that is subject to the type of failure.

Diagnosis System Error Logs/beeps and other critical errors can occur when your Windows operating system becomes corrupted. Opening programs will be slower and response times will lag. When you have multiple applications running, you may experience crashes and freezes. There

can be numerous causes of this error including excessive startup entries, registry errors, hardware/RAM decline, fragmented files, unnecessary or redundant program installations and so on.

Diagnosis System Error Logs/beeps repair tool

There are many reasons why Diagnosis System Error Logs/beeps, including having malware, spyware, or programs not installing properly. You can have all kinds of system conflicts, registry errors, and Active X errors. Reimage specializes in Windows repair. It scans and diagnoses, then repairs, your damaged PC with technology that not only fixes your Windows Operating System, but also reverses the damage already done with a full database of replacement files.

A Scan (approx. 5 minutes) into your PC‟s Windows Operating System detects problems divided into 3 categories – Hardware, Security and Stability. At the end of the scan, you can review your PC‟s Hardware, Security and Stability in comparison with a worldwide average. You can review a summary of the problems detected during your scan.

Windows Errors

A Windows error is an error that happens when an unexpected condition occurs or when a desired operation has failed. When you have an error in Windows, it may be critical and cause your programs to freeze and crash or it may be seemingly harmless yet annoying.

Blue Screen of Death

A stop error screen or bug check screen, commonly called a blue screen of death (also known as a BSoD, bluescreen), is caused by a fatal stystem error and is the error screen displayed by the Microsoft Windows family of operating systems upon encountering a critical error, of a nonrecoverable nature, that causes the system to “crash”.

Damaged DLLs

One of the biggest causes of DLL‟s becoming corrupt/damaged is the practice of constantly installing and uninstalling programs. This often means that DLL‟s will get overwritten by newer versions when a new program is installed, for example. This causes problems for those applications and programs that still need the old version to operate. Thus, the program begins to malfunction and crash.

Freezing Computer

Computer hanging or freezing occurs when either a program or the whole system ceases to respond to inputs. In the most commonly encountered scenario, a program freezes and all windows belonging to the frozen program become static. Almost always, the only way to recover from a system freeze is to reboot the machine, usually by power cycling with an on/off or reset button.

Virus Damage

Once your computer has been infected with a virus, it‟s no longer the same. After removing it with your anti-virus software, you‟re often left with lingering side-effects. Technically, your computer might no longer be infected, but that doesn‟t mean it‟s error-free. Even simply removing a virus can actually harm your system.

Operating System Recovery

Reimage repairs and replaces all critical Windows system files needed to run and restart correctly, without harming your user data. Reimage also restores compromised system settings and registry values to their default Microsoft settings. You may always return your system to its pre-repair condition.

Reimage patented technology, is the only PC Repair program of its kind that actually reverses the damage done to your operating system. The online database is comprised of over 25,000,000 updated essential components that will replace any damaged or missing file on a Windows operating system with a healthy version of the file so that your PC‟s performance, stability & security will be restored and even improve. The repair will deactivate then quarantine all Malware found then remove virus damage. All System Files, DLLs, and Registry Keys that have been corrupted or damaged will be replaced with new healthy files from our continuously updated online database.

CHAPTER 3: MEMORY MANAGEMENT

Introduction to Memory management

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

Memory management Objective

- ✓ To provide a detailed description of various ways of organizing memory hardware.
- ✓ To discuss various memory-management techniques, including paging and segmentation.
- ✓ To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging.

Memory management Concepts

Process Address Space

The process address space is the set of logical addresses that a process references in its code. For example, when 32-bit addressing is in use, addresses can range from 0 to 0x7fffffff; that is, 2^31 possible numbers, for a total theoretical size of 2 gigabytes.

The operating system takes care of mapping the logical addresses to physical addresses at the time of memory allocation to the program. There are three types of addresses used in a program before and after memory is allocated:

S.N.	Memory Addresses & Description
1	**Symbolic addresses** The addresses used in a source code. The variable names, constants, and instruction labels are the basic elements of the symbolic address space.
2	**Relative addresses** At the time of compilation, a compiler converts symbolic addresses into relative addresses.
3	**Physical addresses** The loader generates these addresses at the time when a program is loaded into main memory.

Virtual and physical addresses are the same in compile-time and load-time addressbinding schemes. Virtual and physical addresses differ in execution-time address-binding scheme.

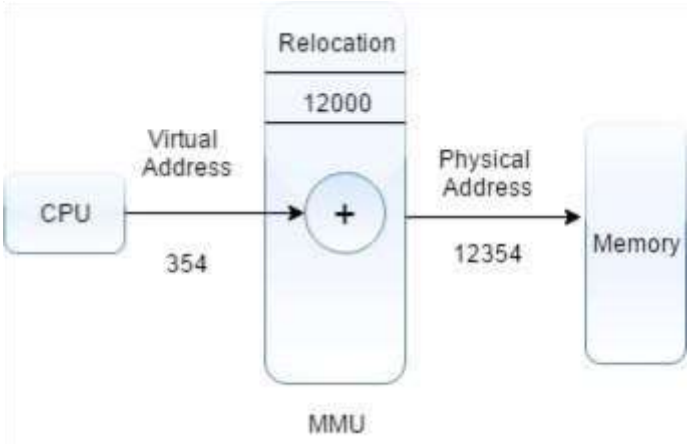
The set of all logical addresses generated by a program is referred to as a **logical address space**. The set of all physical addresses corresponding to these logical addresses is referred to as a **physical address space**.

The runtime mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device. MMU uses the following mechanism to convert virtual address to physical address.

- The value in the base register is added to every address generated by a user process, which is treated as offset at the time it is sent to memory. For example, if the base register value is 10000, then an attempt by the user to use address location 100 will be dynamically reallocated to location 10100.
- The user program deals with virtual addresses; it never sees the real physical addresses.

The address generated by CPU is called logical (or virtual) address space. Processes are always uses virtual address space and they do not see physical address. Logical address space is set of logical addresses that generated by a program.

The physical address is address that seen by memory unit and used to access memory units. Virtual addresses are mapped with physical addresses by memory management unit.



Memory management unit (MMU) is a hardware device that maps virtual addresses to physical addresses.

Static vs Dynamic Loading

The choice between Static or Dynamic Loading is to be made at the time of computer program being developed. If you have to load your program statically, then at the time of compilation, the complete programs will be compiled and linked without leaving any external program or module dependency. The linker combines the object program with other necessary object modules into an absolute program, which also includes logical addresses.

If you are writing a Dynamically loaded program, then your compiler will compile the program and for all the modules which you want to include dynamically, only references will be provided and rest of the work will be done at the time of execution.

At the time of loading, with **static loading**, the absolute program (and data) is loaded into memory in order for execution to start.

If you are using **dynamic loading**, dynamic routines of the library are stored on a disk in relocatable form and are loaded into memory only when they are needed by the program.

Static vs Dynamic Linking

As explained above, when static linking is used, the linker combines all other modules needed by a program into a single executable program to avoid any runtime dependency.

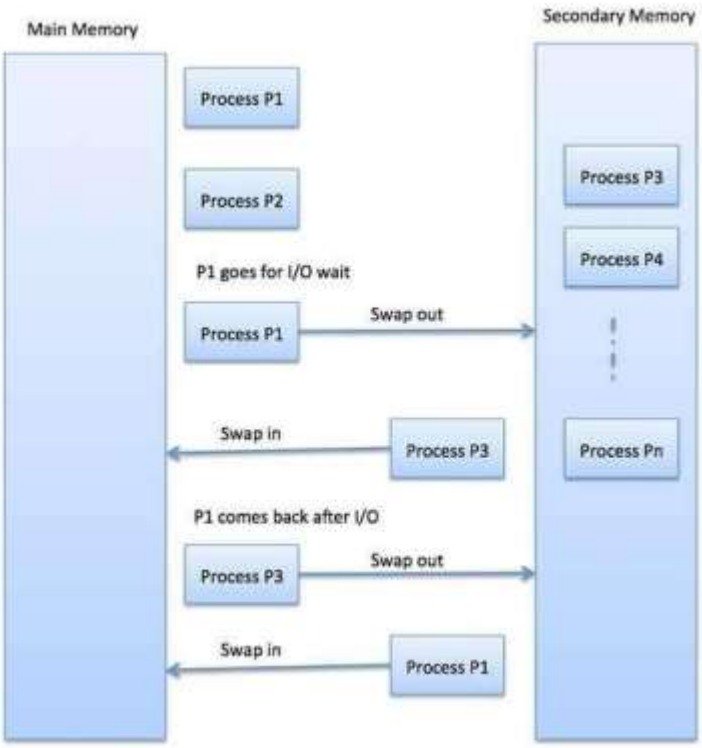
When dynamic linking is used, it is not required to link the actual module or library with the program, rather a reference to the dynamic module is provided at the time of compilation and linking. Dynamic Link Libraries (DLL) in Windows and Shared Objects in Unix are good examples of dynamic libraries.

Memory allocation technique

Swapping

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

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



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

Let us assume that the user process is of size 2048KB and on a standard hard disk where

swapping will take place has a data transfer rate around 1 MB per second. The actual transfer of the 1000K process to or from memory will take

```
2048KB / 1024KB per second
= 2 seconds
= 200 milliseconds
```

Now considering in and out time, it will take complete 400 milliseconds plus other overhead where the process competes to regain main memory.

Memory Allocation

Main memory usually has two partitions:

- **Low Memory** -- Operating system resides in this memory.

- **High Memory** -- User processes are held in high memory.

Operating system uses the following memory allocation mechanism.

S.N.	Memory Allocation & Description
1	**Single-partition allocation** In this type of allocation, relocation-register scheme is used to protect user processes from each other, and from changing operatingsystem code and data. Relocation register contains value of smallest physical address whereas limit register contains range of logical addresses. Each logical address must be less than the limit register.
2	**Multiple-partition allocation** In this type of allocation, main memory is divided into a number of fixed-sized partitions where each partition should contain only one process. When a partition is free, a process is selected from the input queue and is loaded into the free another process.

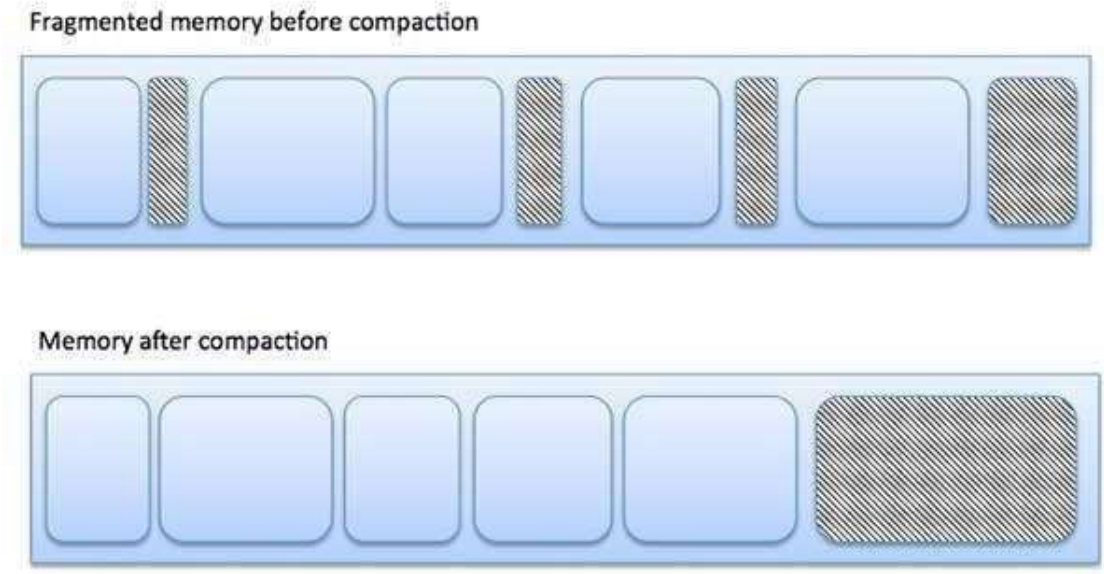
Fragmentation

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

Fragmentation is of two types:

S.N.	Fragmentation & Description
1	**External fragmentation** Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous, so it cannot be used.
2	**Internal fragmentation** Memory block assigned to process is bigger. Some portion of memory is left unused, as it cannot be used by another process.

The following diagram shows how fragmentation can cause waste of memory and a compaction technique can be used to create more free memory out of fragmented memory:



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

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

Contiguous Allocation

The main memory must accommodate both the operating system and the various user processes. The memory is usually divided into two partitions, one for the resident operating system, and one for the user processes.
To place the operating system in low memory. Thus, we shall discuss only me situation where the operating system resides in low memory. The development of the other situation is similar.

Common Operating System is placed in low memory.

1. Single-Partition Allocation

If the operating system is residing in low memory, and the user processes are executing in high memory. And operating-system code and data are protected from changes by the user processes. We also need protect the user processes from one another. We can provide this 2 protection by using relocation registers.

The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation = 100,040 and limit = 74,600). With relocation and limit registers, each logical address must be less than the limit register; the MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory.

The relocation-register scheme provides an effective way to allow the operating system size to change dynamically.

2. Multiple-Partition Allocation

One of the simplest schemes for memory allocation is to divide memory into a number of fixedsized partitions. Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. When a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.

The operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes, and is considered as one large block, of available memory, a hole. When a process arrives and needs memory, we search for a hole large enough for this process.

For example, assume that we have 2560K of memory available and a resident operating system of 400K. This situation leaves 2160K for user processes. FCFS job scheduling, we can immediately allocate memory to processes P1, P2, P3. Holes size 260K that cannot be used by any of the remaining processes in the input queue. Using a round-robin CPU-scheduling with a quantum of 1 time unit, process will terminate at time 14, releasing its memory. Memory allocation is done using Round-Robin Sequence as shown in fig. When a process arrives and needs memory, we search this set for a hole that is large enough for this process. If the hole is too large, it is split into two: One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, we merge these adjacent holes to form one larger hole.

This procedure is a particular instance of the general dynamic storage-allocation problem, which is how to satisfy a request of size n from a list of free holes.

There are many solutions to this problem. The set of holes is searched to determine which hole is best to allocate, first-fit, best-fit, and worst-fit are the most common strategies used to select a free hole from the set of available holes.

First-fit: Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
Best-fit: Allocate the smallest hole that is big enough. We must search the entire list, unless the list is kept ordered by size. This strategy-produces the smallest leftover hole.

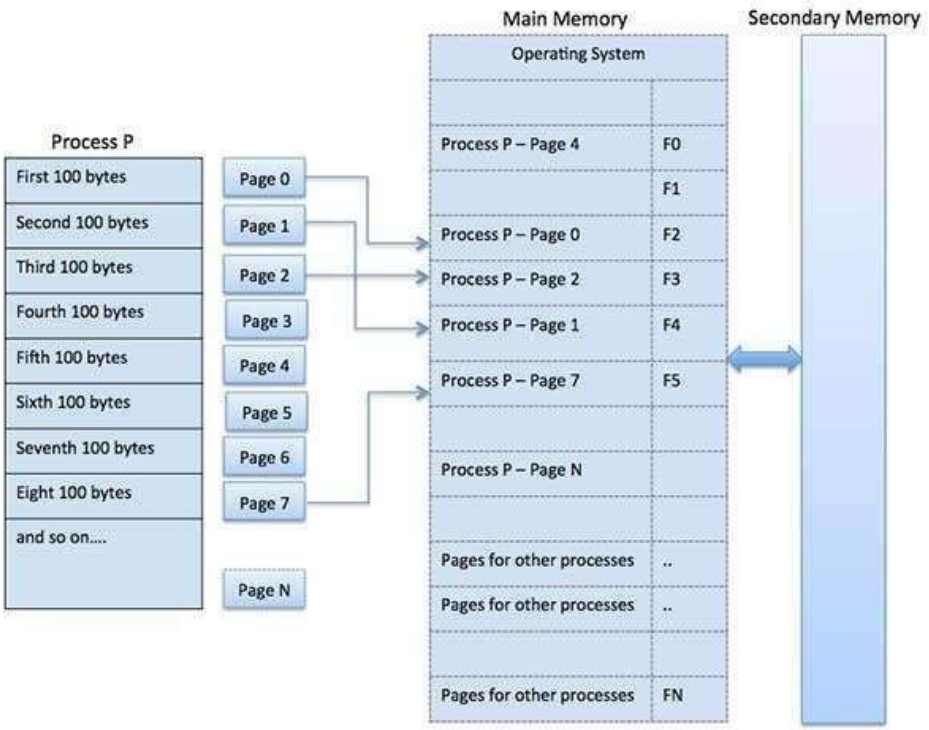
Worst-fit: Allocate the largest hole. Again, we must search the entire list unless it is sorted by size. This strategy produces the largest leftover hole which may be more useful than the smaller leftover hole from a best-t approach.

Paging

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

Paging is a memory management technique in which process address space is broken into blocks of the same size called **pages** (size is power of 2, between 512 bytes and 8192 bytes). The size of the process is measured in the number of pages.

Similarly, main memory is divided into small fixed-sized blocks of (physical) memory called **frames** and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.



Address Translation

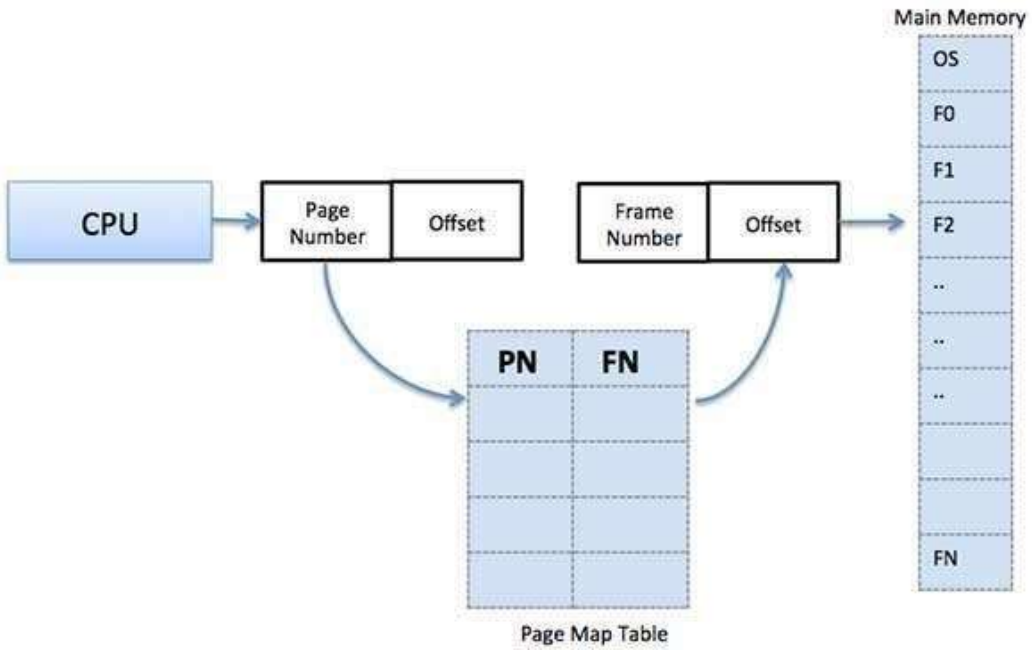
Page address is called **logical address** and represented by **page number** and the **offset**.

```
Logical Address = Page number + page offset
```

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

```
Physical Address = Frame number + page offset
```

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



When the system allocates a frame to any page, it translates this logical address into a physical address and creates entry into the page table to be used throughout execution of the program.

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

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

Advantages and Disadvantages of Paging

Here is a list of advantages and disadvantages of paging:

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

RAM.

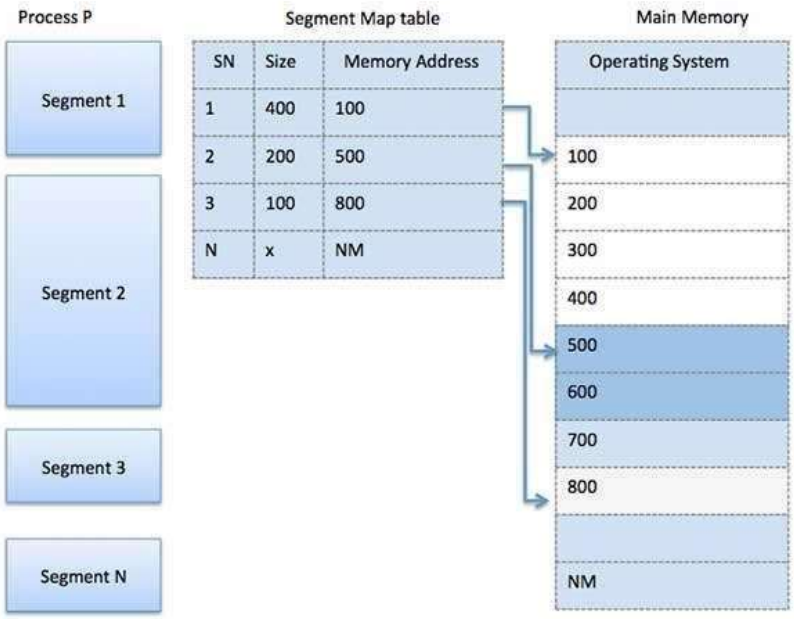
2. Segmentation

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

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

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

A program segment contains the program's main function, utility functions, data structures, and so on. The operating system maintains a **segment map table** for every process and a list of free memory blocks along with segment numbers, their size and corresponding memory locations in main memory. For each segment, the table stores the starting address of the segment and the length of the segment. A reference to a memory location includes a value that identifies a segment and an offset.



Virtual Memory

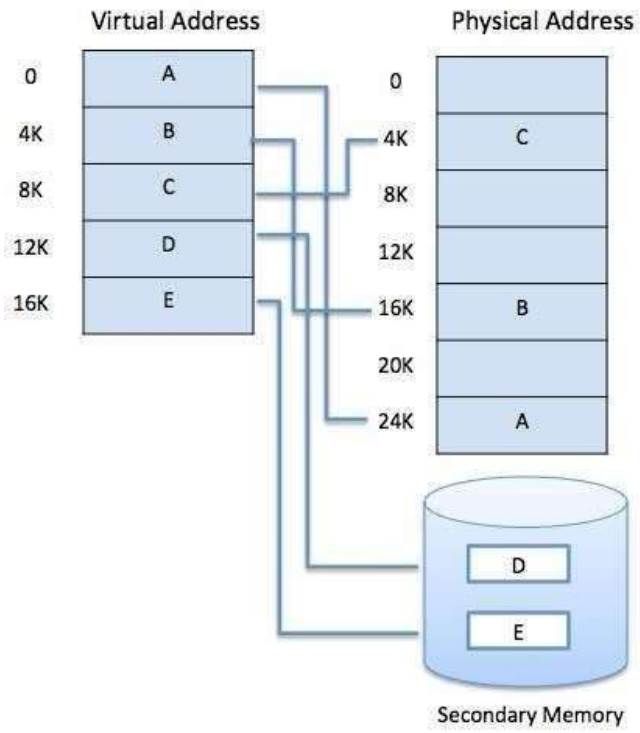
Basic Concept of virtual memory

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

The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory serves two purposes. First, it allows us to extend the use of physical memory by using disk. Second, it allows us to have memory protection, because each virtual address is translated to a physical address.
Following are the situations, when entire program is not required to be loaded fully in main memory.

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

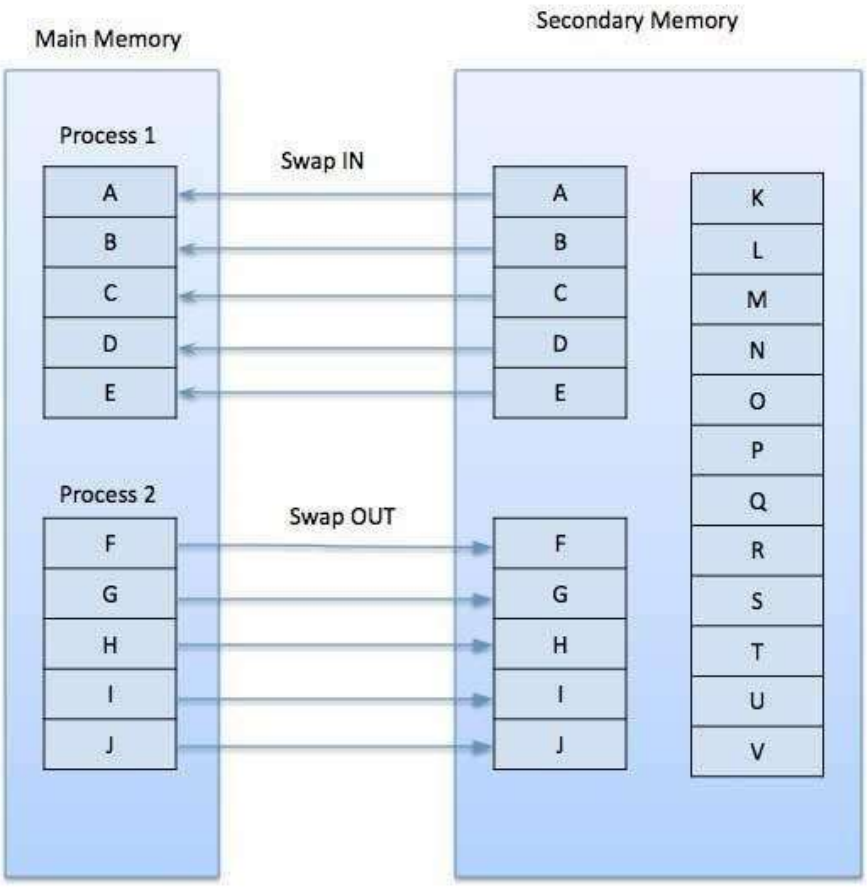
Modern microprocessors intended for general-purpose use, a memory management unit, or MMU, is built into the hardware. The MMU's job is to translate virtual addresses into physical addresses. A basic example is given below:



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

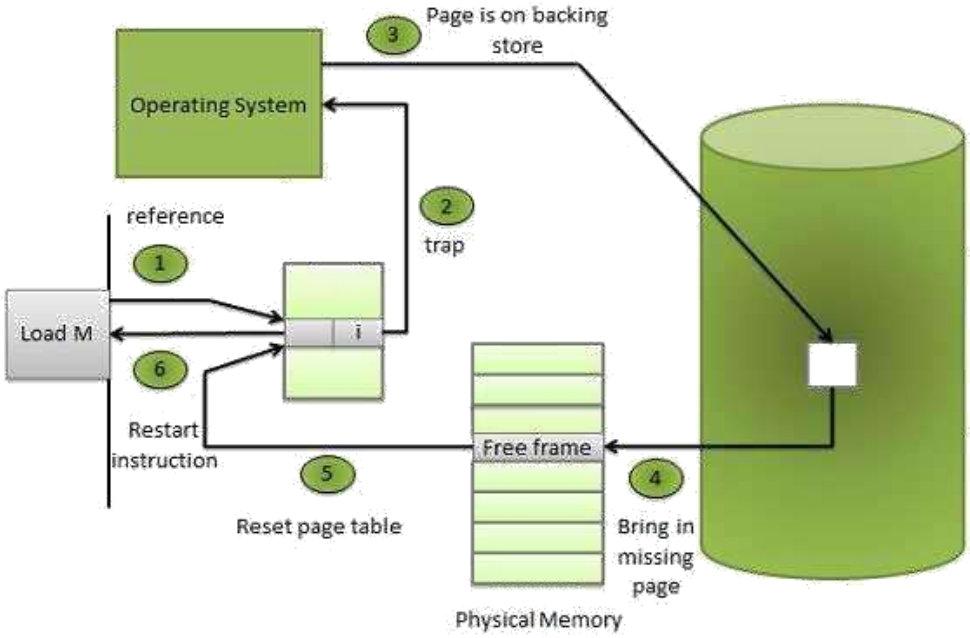
Demand Paging

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



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

Page fault can be handled as following



Step	Description
Step 1	Check an internal table for this process, to determine whether the reference was a valid or it was an invalid memory access.
Step 2	If the reference was invalid, terminate the process. If it was valid, but page have not yet brought in, page in the latter.
Step 3	Find a free frame.
Step 4	Schedule a disk operation to read the desired page into the newly allocated frame
Step 5	When the disk read is complete, modify the internal table kept with the process and the page table to indicate that the page is now in memory.
Step 6	Restart the instruction that was interrupted by the illegal address trap. The process can now access the page as though it had always been in memory. Therefore, the operating system reads the desired page into memory and restarts the process as though the page had always been in memory.

Advantages

Following are the advantages of Demand Paging:

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

Disadvantage

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

Page Replacement Algorithm

Page replacement algorithms are the techniques using which an Operating System decides which memory pages to swap out, write to disk when a page of memory needs to be allocated. Paging happens whenever a page fault occurs and a free page cannot be used for allocation purpose accounting to reason that pages are not available or the number of free pages is lower than required pages.

When the page that was selected for replacement and was paged out, is referenced again, it has to read in from disk, and this requires for I/O completion. This process determines the quality of the page replacement algorithm: the lesser the time waiting for page-ins, the better is the algorithm.

A page replacement algorithm looks at the limited information about accessing the pages provided by hardware, and tries to select which pages should be replaced to minimize the total number of page misses, while balancing it with the costs of primary storage and processor time of the algorithm itself. There are many different page replacement algorithms. We evaluate an algorithm by running it on a particular string of memory reference and computing the number of page faults.

Reference String

The string of memory references is called reference string. Reference strings are generated artificially or by tracing a given system and recording the address of each memory reference. The latter choice produces a large number of data, where we note two things.

- For a given page size, we need to consider only the page number, not the entire address.
- If we have a reference to a page **p**, then any immediately following references to page **p** will never cause a page fault. Page p will be in memory after the first reference; the immediately following references will not fault.
- For example, consider the following sequence of addresses - 123,215,600,1234,76,96

- If page size is 100, then the reference string is 1,2,6,12,0,0 □

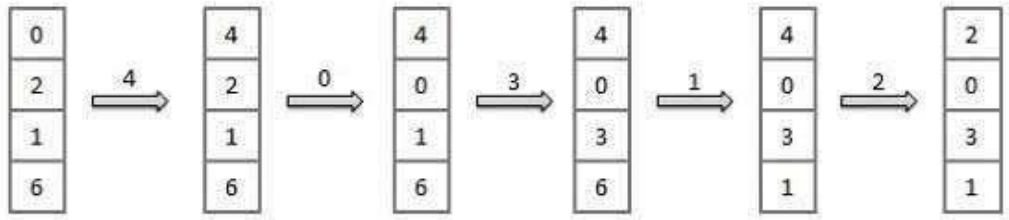
First In First Out (FIFO) Algorithm

- Oldest page in main memory is the one which will be selected for replacement. □
- Easy to implement, keep a list, replace pages from the tail and add new pages at the head. □

□

Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses : x x x x x x x x x



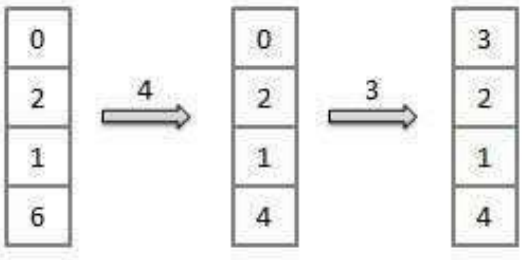
Fault Rate = 9 / 12 = 0.75

Optimal Page Algorithm

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

Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses : x x x x x x



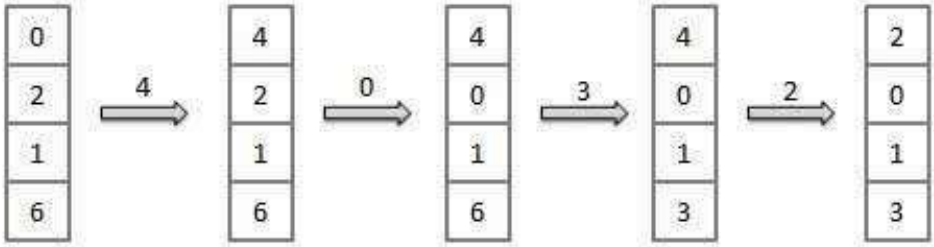
Fault Rate = 6 / 12 = 0.50

Least Recently Used (LRU) Algorithm

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

Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses : x x x x x x x x



Fault Rate = 8 / 12 = 0.67

Page Buffering Algorithm

- To get a process start quickly, keep a pool of free frames.
- On page fault, select a page to be replaced.
- Write the new page in the frame of free pool, mark the page table and restart the process.
- Now write the dirty page out of disk and place the frame holding replaced page in free pool.

Least Frequently Used (LFU) Algorithm

- The page with the smallest count is the one which will be selected for replacement.
- This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again.

Most Frequently Used (MFU) Algorithm

- This algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

Segmented paging and Paged segmentation?

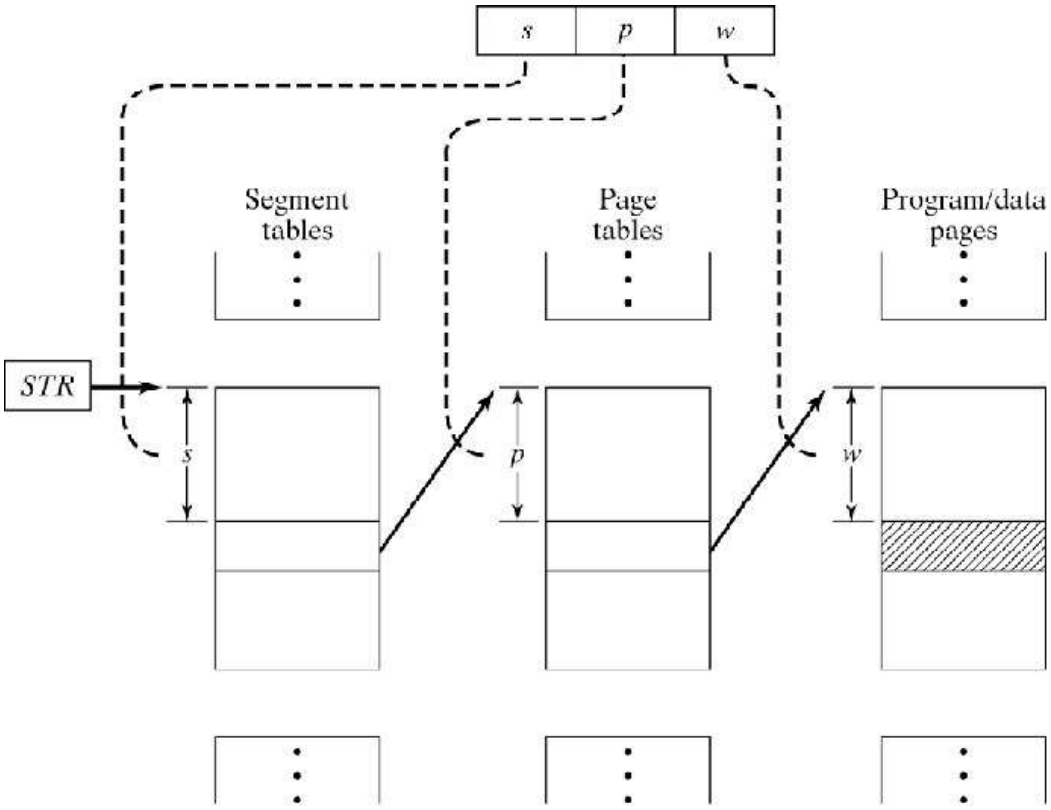
- **Segmented paging**- Segments are divided into pages. Implementation requires STR(segment table register) and PMT(page map table).In this scheme, each virtual address consists of a *segment number*, *page number* within that segment and an *offset* within that page.The segment

number indexes into segment table which yields the base address of the page table for that segment. The page number indexes into the page table, each of which entry is a page frame. Adding the PFN(page frame number) and the offset results in the physical address. Hence addressing can be described by the following function :

va = (s, p, w) where, va is the virtual address, |s| determines number of segments (size of ST), |p| determines number of pages per segment (size of PT), |w| determines page size.

```
address_map(s, p, w)
{
  pa = *(*(STR+s)+p)+w;
return pa; }
```

The diagram is here:



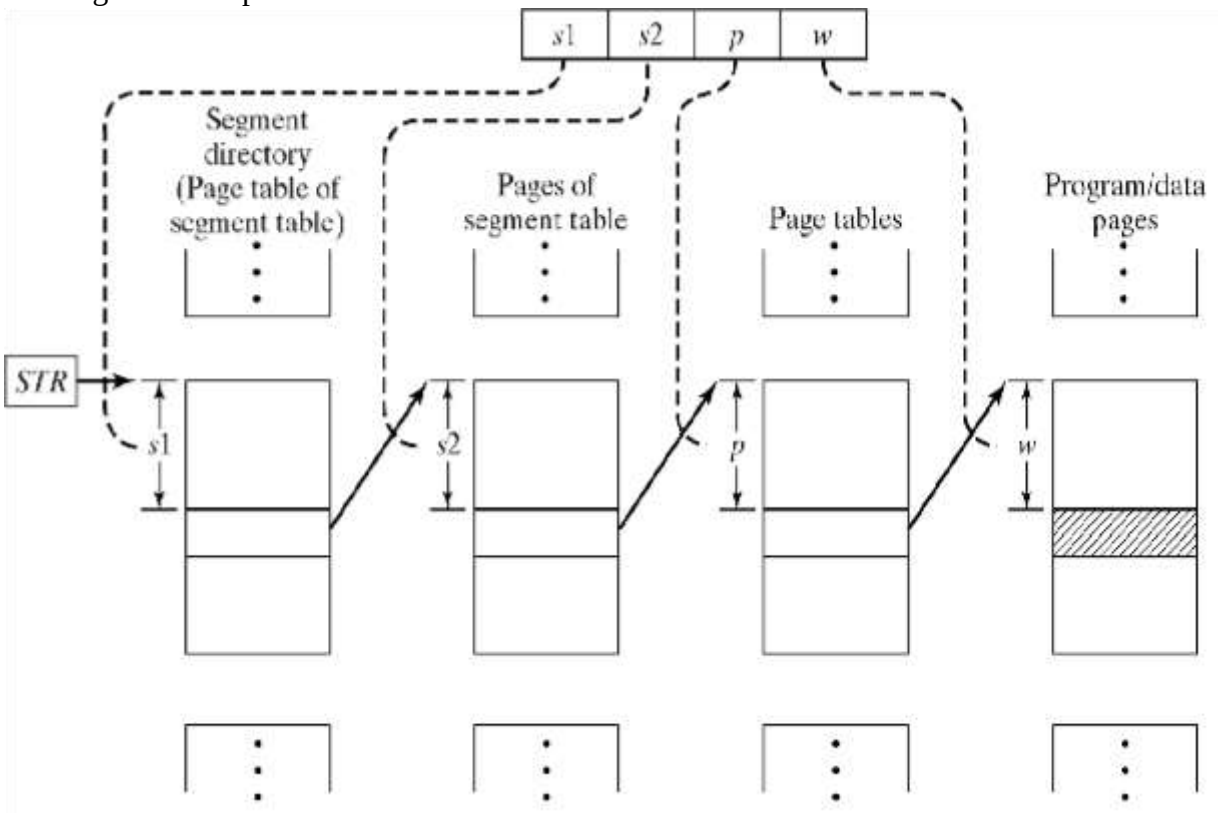
- **Paged Segmentation**- Sometimes segment table or page table may too large to keep in physical memory (they can even reach MBs).Therefore, the segment table is divided into pages too and thus a page table of ST pages is created. The *segment number* is broken into *page no.*

(s1) and *page offset(s2)* of page table of ST pages. So, the virtual address can be described as:

va = (s1, s2, p, w)

```
address_map
(s1, s2, p, w)
{
pa = *(*(*(STR+s1)+s2)+p)+w;
return pa; }
```

The diagram description is here:

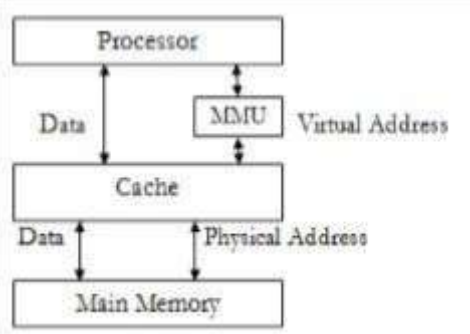


Summary of Virtual Memory (Paging and Segmentation)

- In the most computer system, the physical main memory is not as large as address space of the processor.

- When we try to run a program, if it do not completely fit into the main memory the parts of its currently being executed are stored in main memory and remaining portion is stored in secondary storage device such as HDD.
- When a new part of program is to be brought into main memory for execution and if the memory is full, it must replace another part which is already is in main memory.
- As this secondary memory is not actually part of system memory, so for CPU, secondary memory is Virtual Memory.
- Techniques that automatically more program and data blocks into physical memory when they are required for execution are called virtual memory
- Virtual Memory is used to logically extend the size of main memory.
- When Virtual Memory is used, the address field is virtual address.
- A special hardware unit knows as MMU translates Virtual Address into Physical Address.

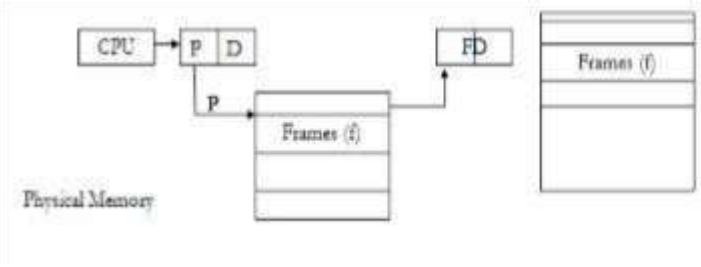
Memory Management Unit



- Address Translation is done by two techniques o Paging o Segmentation

1. Paging:

- Physical memory is divided into fixed size block know as Frames.
- Logical Memory is divided into blocks of same size knows as Pages.
- When a process is to be executed, its pages are loaded into available memory - -- Paging Hardware :

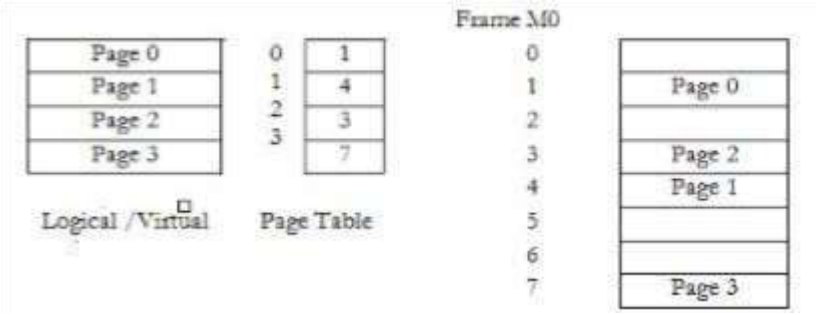


Physical Memory

Page Table (Base Register)

- Every address generated by CPU is divided into two parts :
  - Page Number (P) o Displacement/Offset (d)
- The page number is used as index into a page table from page table contains base address (f) of each page in physical memory.
- This base address (f) is combined with the page offset (d) to define the physical memory address.

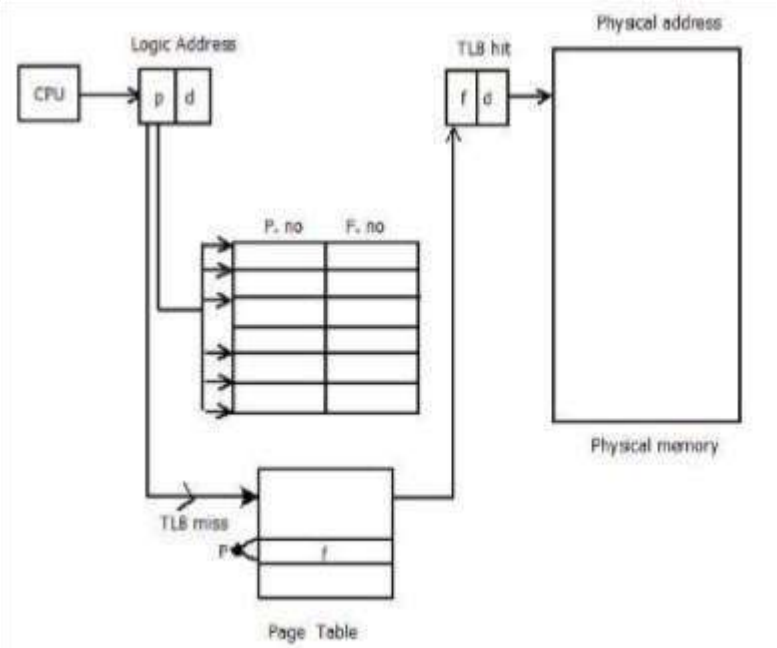
Paging Model of Logical & Physical Memory:



Disadvantages:

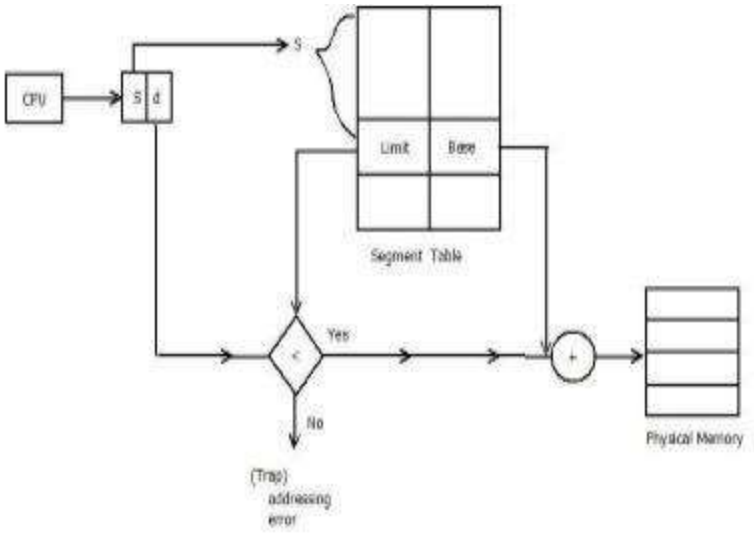
- This approach slow down the memory access by faster of 2.
- So the solution to this problem is to use special small fast cache know as translation Look Aside Buffer (TLB)

Paging Hardware with TLB:



- The TLB contains only few of page table entries.
- When logical address is generated by CPU its page number is used to index the TLB.
- If page number is found, frame number is obtained and we can access memory and is a TLB Hit.
- If page is not in TLB, then it is TLB miss and Reference is masked to page task.
- If TLB is already FULL of entries, then Operating System must select Replacement Policy.
- If the required page is not in main memory, the page must be brought from secondary to main memory.

2. Segmentation:



- The mapping is done with help of segment table. Each entry of segment table has base and limits.
- The segment base contains starting physical address where resides in memory whereas limit specifies length of the segments.
- The segment number is used as index for segment table.
- The offset must be between O and limits.
- If it is not less than limit then it is trapped (addressing the error).
- If it is less than limit, then add it to segment base to produce address in physical memory.

CHAPTER 4: DEVICE (1/0) MANAGEMENT

Objectives of device (I/O) management

Objectives of the I/O Management in Operating System:

The objectives of the I/O Management module are as follows:

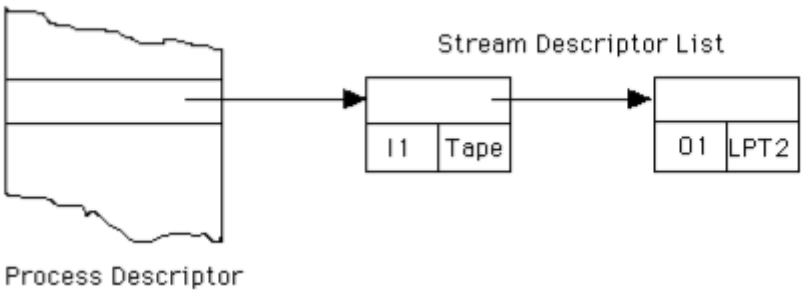
1. Generality and Device Independence - Uniform Treatment of all Devices:

I/O devices are typically quite complex mechanically and electronically. Much of this complexity is related to the electronic engineering and is of no interest to the user or the programmer. The average user is not aware of the complexities of positioning the heads on a disk drive, reading the signal from the disk surface, waiting for the required sector to rotate into position etc.

Users and programmers may be unaware of this complexity. The I/O management module must try to treat all external devices uniformly. This is achieved by virtual device. A virtual device is a special kind of file that is associated with a particular external device. Users create a virtual device of a given type, and operating system associates a physical device with it when the device is required for the first time. All virtual devices in a system are stored in a liked list.

Details

- So users of Peripherals can use a standard way of interacting with different devices.
- OS needs to smooth out all the differences between devices.
- This is achieved by the Virtual Device. Users interact with Virtual Devices called streams. User creates a stream of a given type, and OS associates it with a physical device the first time the device is required.
- Info about stream, and physical device, needs to be held in Process Descriptor - PD points to a stream descriptor list.



- Stream is opened when OS associates stream with Phy. Device. Stream is closed either explicitly by process or when process terminates

Advantages

- User can interact with devices using standard set of instructions.
- User does not need to know which specific Phy. Device will be used by a process
- Easy to change device from, say, Tape to Disk, without having to rewrite large chunks of code.

Device Handlers

- So far we've looked at ways in which users can use I/O devices in a standard way.
- Need to look at ways in which OS can operate I/O devices in a uniform manner.
- Device Handlers are software that interface between OS and I/O devices.
- When I/O instruction is received, Device Handler converts generic instruction and internal character character code into format required by specific device.
- To convert generic instructions in specific commands, device handler needs to know the device's characteristics. These are held in the device's device descriptor. □ The descriptor contains:

the device identification the instructions

which operate the device pointers to

character translation table device's

current status process id of process

currently using device

- All device descriptors are linked to form a device structure pointed at from Central Table.

Note:
Device independence is the process of making a software application be able to function on a wide variety of devices regardless of the local hardware on which the software is used.
Character independence is the process of making a software application make devices format and share data uniformly.

2. Efficiency:

Perhaps the most significant characteristic of the I/O system is. the speed disparity between it and the processor. I/O devices involve mechanical operations. They cannot compete with the microsecond or nanosecond speed of the processor and memory. The I/O management module must try to minimize the disparity by the use of techniques like buffering and spooling.

Principles of device (I/O) Hardware

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

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

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

Device Controllers

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

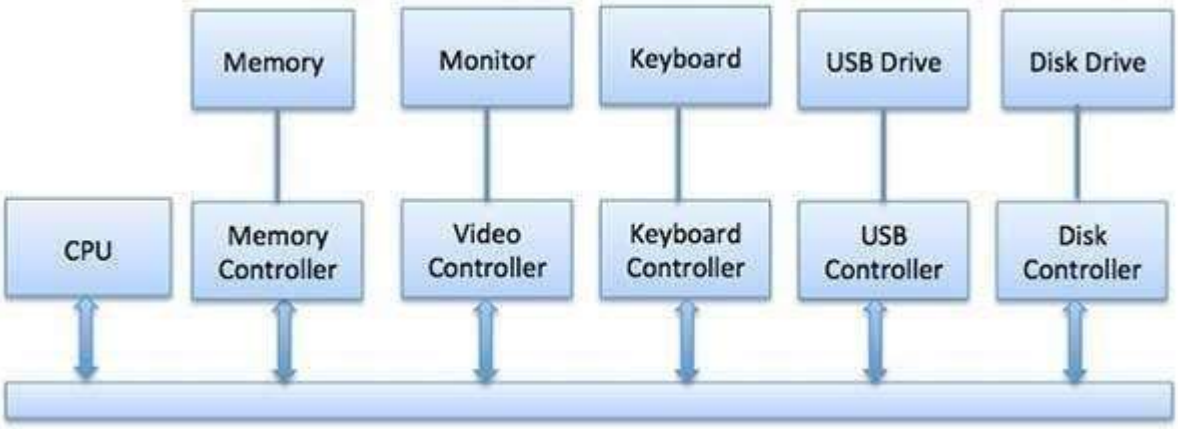
The Device Controller works like an interface between a device and a device driver. I/O units (Keyboard, mouse, printer, etc.) typically consist of a mechanical component and an electronic component where electronic component is called the device controller.

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

Any device connected to the computer is connected by a plug and socket, and the socket is connected to a device controller. Following is a model for connecting the CPU, memory,

controllers, and I/O devices where CPU and device controllers all use a common bus for communication.

Synchronous vs Asynchronous I/O



- **Synchronous I/O** — In this scheme CPU execution waits while I/O proceeds

- **Asynchronous I/O** — I/O proceeds concurrently with CPU execution

Communication to I/O Devices

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

- Special Instruction I/O

- Memory-mapped I/O

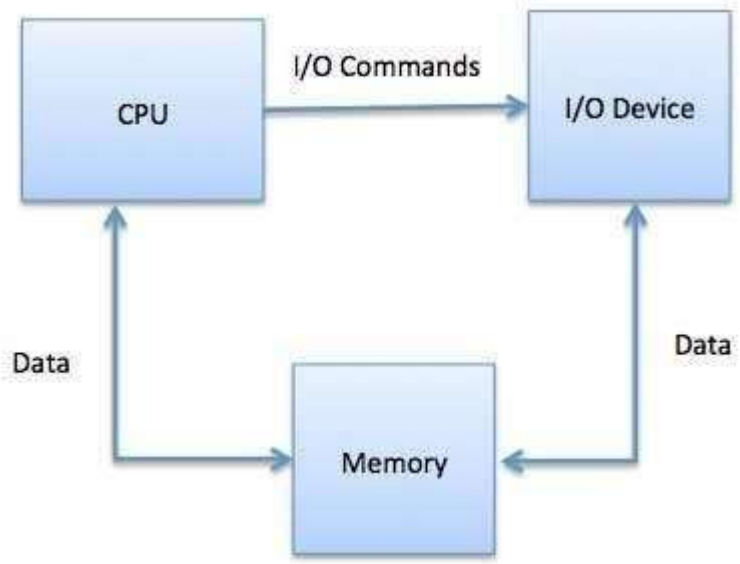
- Direct memory access (DMA)

Special Instruction I/O

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

Memory-mapped I/O

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



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

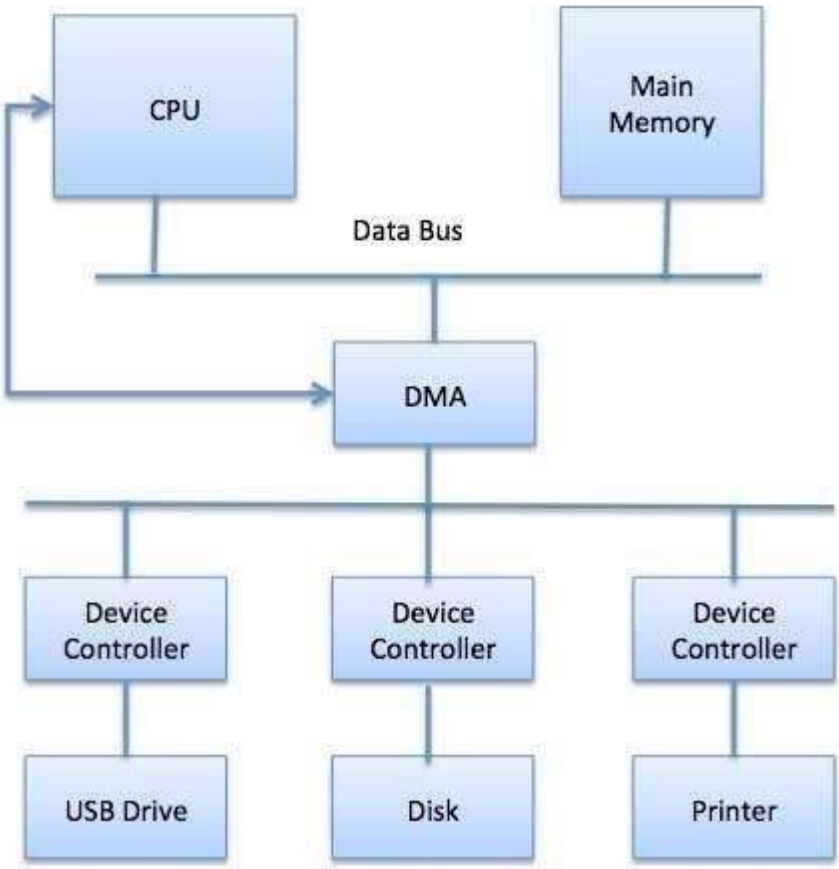
The advantage to this method is that every instruction which can access memory can be used to manipulate an I/O device. Memory mapped IO is used for most high-speed I/O devices like disks, communication interfaces.

Direct Memory Access (DMA)

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

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

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



The operating system uses the DMA hardware as follows:

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

Polling vs Interrupts I/O

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

Polling I/O

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

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

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

Interrupts I/O

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

A device controller puts an interrupt signal on the bus when it needs CPU's attention when

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

Principles of I/O Software

Goals of the I/O Software

Device independence

The principle of device independence works very well with respect to random access information storage devices. Essentially no level of software, from the OS upward cares what kind of storage device it's reading from/ writing to.

It works fairly well in equating character input devices (e.g. keyboard) with disks and character output devices (e.g. a character-oriented window) with disk; which is why redirection can be done. For the most part, but with significant exceptions, software that reads files from disk can be applied to take its input from the keyboard and vice versa.

For other devices "device independence" works with respect to some "systemsy" features such as naming the device, handling the interrupts, and protection. At the user level, device independence for such devices is not generally an issue. At the level of user code, printing on the printer does the speaker, and nothing is gained by trying to make it do so. There is pretty much no application that you are going to output sometimes on the printer and other times on the speaker.

There are some devices that don't look like any other device at any level. The interface to the monitor is through video RAM; entirely unlike the interface to any other output device. The interface to the clock consists wholly of the interrupt handler; again, there's no resemblance to any other device.

Uniform naming

Recall that we discussed the value of the name space implemented by file systems. There is no dependence between the name of the file and the device on which it is stored. So a file called IAmStoredOnAHardDisk might well be stored on a floppy disk.

Error handling

There are several aspects to error handling including: detection, correction (if possible) and reporting.

1. Detection should be done as close to where the error occurred as possible before more damage is done (fault containment). This is not trivial.

2. Correction is sometimes easy, for example ECC (error correcting code) memory does this automatically (but the OS wants to know about the error so that it can schedule replacement of the faulty chips before unrecoverable double errors occur).
   Other easy cases include successful retries for failed ethernet transmissions. In this example, while logging is appropriate, it is quite reasonable for no action to be taken.

3. Error reporting tends to be awful. The trouble is that the error occurs at a low level but by the time it is reported the context is lost. Unix/Linux in particular is horrible in this area.

Creating the illusion of synchronous I/O

- I/O *must* be asynchronous for good performance. That is the OS *cannot* simply wait for an I/O to complete. Instead, it proceeds with other activities and responds to the notification when the I/O has finished.
- Users (mostly) want no part of this. The code sequence
- Read X
- Y <-- X+1 Print Y
  Should print a value one greater than that read. But if the assignment is performed before the read completes, the wrong value is assigned.

- Performance junkies sometimes do want the asynchrony so that they can have another portion of their program executed while the I/O is underway. That is they implement a mini-scheduler in their application code.

Buffering

- Often needed to hold data for examination prior to sending it to its desired destination.
- But this involves copying and takes time.

Sharable vs dedicated devices

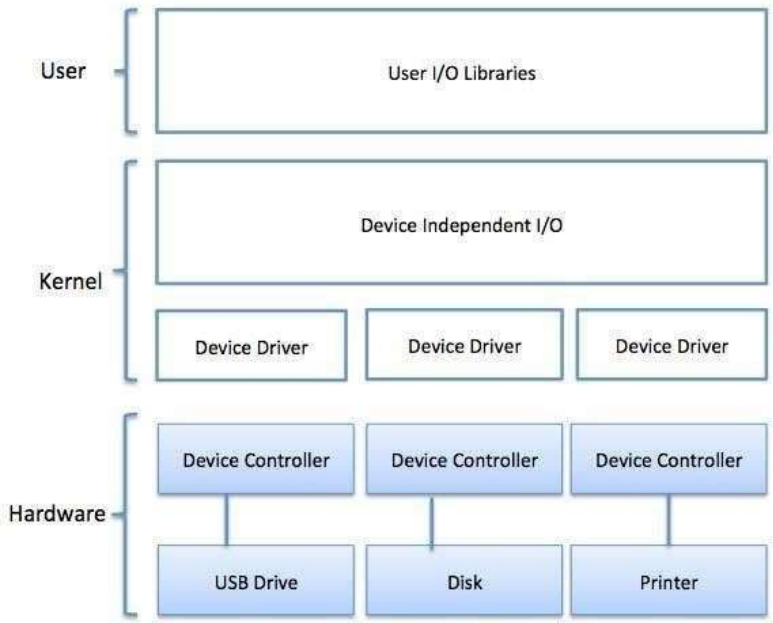
For devices like printers and tape drives, only one user at a time is permitted. These are called **serially reusable** devices. Devices like disks and Ethernet ports can be shared by processes running concurrently.

Introduction to I/O software I/O software is often organized in the following layers:

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

- **Hardware**: This layer includes actual hardware and hardware controller which interact with the device drivers and makes hardware alive.

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



Device Drivers

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

A device driver performs the following jobs:

- To accept request from the device independent software above to it.

- Interact with the device controller to take and give I/O and perform required error handling

- Making sure that the request is executed successfully

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

Interrupt handlers

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

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

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

Device-Independent I/O Software

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

- Uniform interfacing for device drivers

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

User-Space I/O Software

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

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

Kernel I/O Subsystem

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

- **Scheduling** - Kernel schedules a set of I/O requests to determine a good order in which to execute them. When an application issues a blocking I/O system call, the request is placed on the queue for that device. The Kernel I/O scheduler rearranges the order of the queue to improve the overall system efficiency and the average response time experienced by the applications.
- **Buffering** - Kernel I/O Subsystem maintains a memory area known as **buffer** that stores data while they are transferred between two devices or between a device with an application operation. Buffering is done to cope with a speed mismatch between the producer and consumer of a data stream or to adapt between devices that have different data transfer sizes.
- **Caching** - Kernel maintains cache memory which is region of fast memory that holds copies of data. Access to the cached copy is more efficient than access to the original.

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

Disks and disk operations

Overview of Mass-Storage Structure

1) Traditional **magnetic disks** have the following basic structure:
   - One or more ***platters*** in the form of disks covered with magnetic media. ***Hard disk*** platters are made of rigid metal, while "***floppy***" disks are made of more flexible plastic.
   - Each platter has two working ***surfaces.*** Older hard disk drives would sometimes not use the very top or bottom surface of a stack of platters, as these surfaces were more susceptible to potential damage.
   - Each working surface is divided into a number of concentric rings called ***tracks.*** The collection of all tracks that are the same distance from the edge of the platter, (i.e. all tracks immediately above one another in the following diagram) is called a ***cylinder***.
   - Each track is further divided into ***sectors,*** traditionally containing 512 bytes of data each, although some modern disks occasionally use larger sector sizes. (Sectors also include a header and a trailer, including checksum information among other things. Larger sector sizes reduce the fraction of the disk consumed by headers and trailers, but increase internal fragmentation and the amount of disk that must be marked bad in the case of errors.)
   - The data on a hard drive is read by read-write ***heads.*** The standard configuration (shown below) uses one head per surface, each on a separate ***arm***, and controlled by a common ***arm assembly*** which moves all heads simultaneously from one cylinder to another. (Other configurations, including independent read-write heads, may speed up disk access, but involve serious technical difficulties.)
   - The storage capacity of a traditional disk drive is equal to the number of heads (i.e. the number of working surfaces), times the number of tracks per surface, times the number of sectors per track, times the number of bytes per sector. A particular physical block of data is specified by providing the head-sector-cylinder number at which it is located.

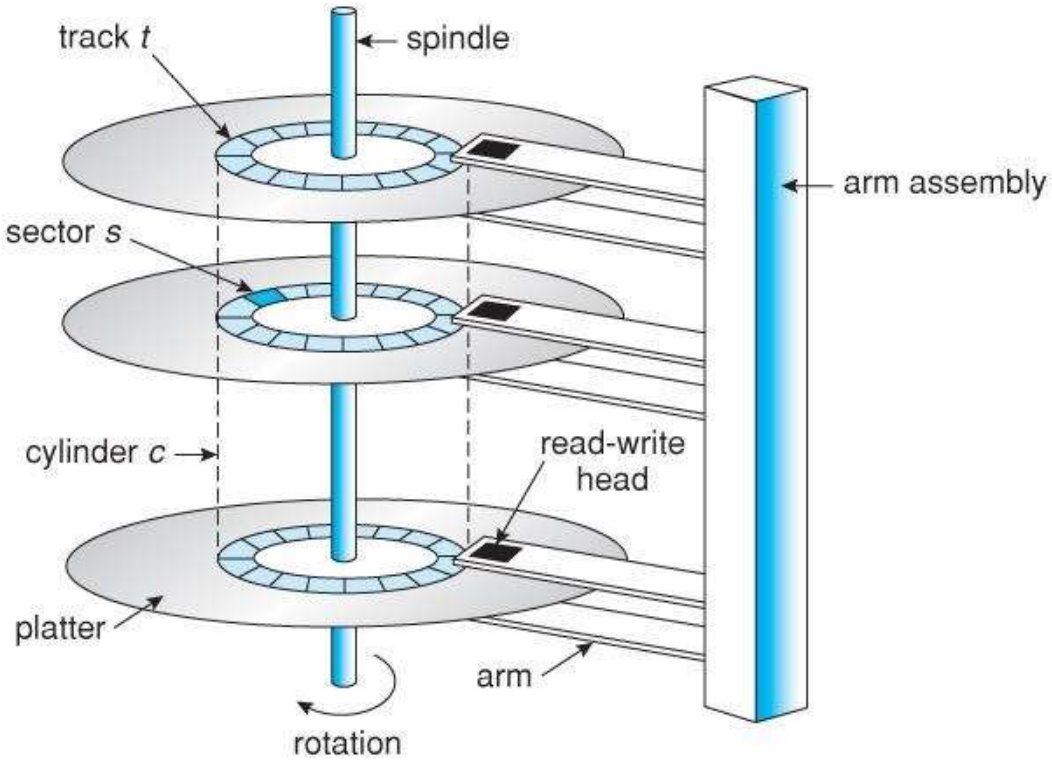


Figure - Moving-head disk mechanism.

2) In operation the disk rotates at high speed, such as 7200 rpm (120 revolutions per second.) The rate at which data can be transferred from the disk to the computer is composed of several steps:
   - The ***positioning time***, a.k.a. the ***seek time*** or ***random access time*** is the time required to move the heads from one cylinder to another, and for the heads to settle down after the move. This is typically the slowest step in the process and the predominant bottleneck to overall transfer rates.
   - The ***rotational latency*** is the amount of time required for the desired sector to rotate around and come under the read-write head.This can range anywhere from zero to one full revolution, and on the average will equal one-half revolution. This is another physical step and is usually the second slowest step behind seek time. (For a disk rotating at 7200 rpm, the average rotational latency would be 1/2 revolution / 120 revolutions per second, or just over 4 milliseconds, a long time by computer standards.
   - The ***transfer rate***, which is the time required to move the data electronically from the disk to the computer. (Some authors may also use the term transfer rate to refer to the overall

transfer rate, including seek time and rotational latency as well as the electronic data transfer rate.)

3) Disk heads "fly" over the surface on a very thin cushion of air. If they should accidentally contact the disk, then a ***head crash*** occurs, which may or may not permanently damage the disk or even destroy it completely. For this reason it is normal to ***park*** the disk heads when turning a computer off, which means to move the heads off the disk or to an area of the disk where there is no data stored.
4) Floppy disks are normally ***removable***. Hard drives can also be removable, and some are even ***hotswappable***, meaning they can be removed while the computer is running, and a new hard drive inserted in their place.
5) Disk drives are connected to the computer via a cable known as the ***I/O Bus.*** Some of the common interface formats include Enhanced Integrated Drive Electronics, EIDE; Advanced Technology Attachment, ATA; Serial ATA, SATA, Universal Serial Bus, USB; Fiber Channel, FC, and Small Computer Systems Interface, SCSI.
6) The ***host controller*** is at the computer end of the I/O bus, and the ***disk controller*** is built into the disk itself. The CPU issues commands to the host controller via I/O ports. Data is transferred between the magnetic surface and onboard ***cache*** by the disk controller, and then the data is transferred from that cache to the host controller and the motherboard memory at electronic speeds.

Disk Structure

Disk provide bulk of secondary storage of computer system. The disk can be considered the one I/O device that is common to each and every computer. Disks come in many size and speeds, and information may be stored optically or magnetically. Magnetic tape was used as an early secondary storage medium, but the access time is much slower than for disks. For backup, tapes are currently used.

Modern disk drives are addressed as large one dimensional arrays of logical blocks, where the logical block is the smallest unit of transfer. The actual details of disk I/O operation depends on the computer system, the operating system and the nature of the I/O channel and disk controller hardware.

The basic unit of information storage is a sector. The sectors are stored on a flat, circular, media disk. This media spins close to one or more read/write heads. The heads can move from the inner portion of the disk to the outer portion.

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 movable head system or electronically selecting one head on a fixed head system. These characteristics are common to floppy disks, hard disks, CD-ROM and DVD.

Disk Performance Parameters

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 movable-head system or electronically selecting one head on a fixed-head system. On a movable-head system, the time it takes to position the head at the track is known as **seek time.**

When 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 the seek time, if any, and the rotational delay equals the **access time,** which is 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; the time required for the transfer is the **transfer time.**

Seek Time Seek time is the time required to move the disk arm to the required track. It turns out that this is a difficult quantity to pin down. The seek time consists of two key components: the initial startup time and the time taken to traverse the tracks that have to be crossed once the access arm is up to speed.

Ts = m x n + s

Rotational Delay Disks, other than floppy disks, rotate at speeds ranging from 3600 rpm up to, as of this writing, 15,000 rpm; at this latter speed, there is one revolution per 4 ms. Thus, on the average, the rotational delay will be 2 ms. Floppy disks typically rotate at between 300 and 600 rpm. Thus the average delay will be between 100 and 50 ms.

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

T= b/rN

where
T = transfer time b = number of bytes
to be transferred N = number of
bytes on a track
r = rotation speed, in revolutions per second
Thus the total average access time can be expressed as
Ta = Ts + where Ts is the
average seek time.

Disk Scheduling

The amount of head needed to satisfy a series of I/O request can affect the performance. If desired disk drive and controller are available, the request can be serviced immediately. If a device or

controller is busy, any new requests for service will be placed on the queue of pending requests for that drive. When one request is completed, the operating system chooses which pending request to service next.

Different types of scheduling algorithms are as follows.

1. First Come, First Served scheduling algorithm(FCFS).

2. Shortest Seek Time First (SSTF) algorithm

3. SCAN algorithm

4. Circular SCAN (C-SCAN) algorithm

5. Look Scheduling Algorithm

1. First Come, First Served scheduling algorithm(FCFS).

The simplest form of scheduling is first-in-first-out (FIFO) scheduling, which processes items from the queue in sequential order. This strategy has the advantage of being fair, because every request is honored and the requests are honored in the order received. With FIFO, if there are only a few processes that require access and if many of the requests are to clustered file sectors, then we can hope for good performance.

Priority With a system based on priority (PRI), the control of the scheduling is outside the control of disk management software.

Last In First Out ln transaction processing systems, giving the device to the most recent user should result. In little or no arm movement for moving through a sequential file. Taking advantage of this locality improves throughput and reduces queue length.

2. Shortest Seek Time First (SSTF) algorithm

The SSTF policy is to select the disk I/O request the requires the least movement of the disk arm from its current position. **Scan** With the exception of FIFO, all of the policies described so far can leave some request unfulfilled until the entire queue is emptied. That is, there may always be new requests arriving that will be chosen before an existing request.

The choice should provide better performance than FCFS algorithm.

Under heavy load, SSTF can prevent distant request from ever being serviced. This phenomenon is known as starvation. SSTF scheduling is essentially a from of shortest job first scheduling. SSTF scheduling algorithm are not very popular because of two reasons.

1. Starvation possibly exists.
2. it increases higher overheads.

3. SCAN scheduling algorithm

The scan algorithm has the head start at track 0 and move towards the highest numbered track, servicing all requests for a track as it passes the track. The service direction is then reserved and the scan proceeds in the opposite direction, again picking up all requests in order.

SCAN algorithm is guaranteed to service every request in one complete pass through the disk. SCAN algorithm behaves almost identically with the SSTF algorithm. The SCAN algorithm is sometimes called elevator algorithm.

4. C SCAN Scheduling Algorithm

The C-SCAN policy restricts scanning to one direction only. Thus, when the last track has been visited in one direction, the arm is returned to the opposite end of the disk and the scan begins again.

This reduces the maximum delay experienced by new requests.

5. LOOK Scheduling Algorithm

Start the head moving in one direction. Satisfy the request for the closest track in that direction when there is no more request in the direction, the head is traveling, reverse direction and repeat. This algorithm is similar to innermost and outermost track on each circuit.

Disk Management

Operating system is responsible for disk management. Following are some activities involved. **1. Disk Formatting**

Disk formatting is of two types.

a) Physical formatting or low level formatting.

b) Logical Formatting

Physical Formatting

- Disk must be formatted before storing data.
- Disk must be divided into sectors that the disk controllers can read/write.
- Low level formatting files the disk with a special data structure for each sector.
- Data structure consists of three fields: header, data area and trailer.
- Header and trailer contain information used by the disk controller.
- Sector number and Error Correcting Codes (ECC) contained in the header and trailer.
- For writing data to the sector – ECC is updated.
- For reading data from the sector – ECC is recalculated.
- Low level formatting is done at factory.

Logical Formatting

- After disk is partitioned, logical formatting used.
- Operating system stores the initial file system data structures onto the disk.

2. Boot Block

When a computer system is powered up or rebooted, a program in read only memory executes. Diagnostic check is done first.

Stage 0 boot program is executed.

Boot program reads the first sector from the boot device and contains a stage-1 boot program.

May be boot sector will not contain a boot program.

PC booting from hard disk, the boot sector also contains a partition table.

The code in the boot ROM instructs the disk controller to read the boot blocks into memory and then starts executing that code.

Full boot strap program is more sophisticated than the bootstrap loader in the boot ROM.

Swap Space Management

Swap space management is low level task of the operating system. The main goal for the design and implementation of swap space is to provide the best throughput for the virtual memory system.

1. Swap-Space Use

The operating system needs to release sufficient main memory to bring in a process that is ready to execute. Operating system uses this swap space in various way. Paging systems may simply store pages that have been pushed out of main memory. Unix operating system allows the use of multiple swap space are usually put on separate disks, so the load placed on the I/O system by paging and swapping can be spread over the systems I/O devices.

2. Swap Space Location

Swap space can reside in two places:

1. Separate disk partition
2. Normal file System

If the swap space is simply a large file within the file system, normal file system routines can be used to create it, name it and allocate its space. This is easy to implement but also inefficient. External fragmentation can greatly increase swapping times. Catching is used to improve the system performance. Block of information is cached in the physical memory, and by using special tools to allocate physically continuous blocks for the swap file.

Swap space can be created in a separate disk partition. No file system or directory structure is placed on this space. A separate swap space storage manager is used to allocate and deallocate the blocks. This manager uses algorithms optimized for speed. Internal fragmentation may increase. Some operating systems are flexible and can swap both in raw partitions and in file system space.

Stable Storage Implementation

The write ahead log, which required the availability of stable storage.

By definition, information residing in stable storage is never lost.

To implement such storage, we need to replicate the required information on multiple storage devices (usually disks) with independent failure modes.

We also need to coordinate the writing of updates in a way that guarantees that a failure during an update will not leave all the copies in a damaged state and that, when we are recovering from failure, we can force all copies to a consistent and correct value, even if another failure occurs during the recovery.

Disk Reliability

Good performance means high speed, another important aspect of performance is reliability.

A fixed disk drive is likely to be more reliable than a removable disk or tape drive.

An optical cartridge is likely to be more reliable than a magnetic disk or tape.

A head crash in a fixed hard disk generally destroys the data, whereas the failure of a tape drive or optical disk drive often leaves the data cartridge unharmed.

Summary

Disk drives are the major secondary storage I/O devices on most computers. Most secondary devices are either magnetic disks or magnetic tapes. Modern disk drives are structured as large one dimensional arrays of logical disk blocks. Disk scheduling algorithms can improve the effective bandwidth, the average response time, and the variance response time. Algorithms such as SSTF, SCAN, C-SCAN. LOOK, and CLOOK are designed to make such improvements through strategies for disk queue ordering.
Performance can be harmed by external fragmentation. The operating system manages block. First, a disk must be low level formatted to create the sectors on the raw hardware, new disks usually come preformatted. Then, disk is partitioned, file systems are created, and boot blocks are allocated to store the system bootstrap program. Finally when a block is corrupted, the system must have a way to lock out that block or to replace it logically with a space. Because of efficient swap space is a key to good performance, systems usually bypass the file system and use raw disk access for paging I/O. Some systems dedicate a raw disk partition to swap space, and others use a file within the file system instead.

Computer clocking system

Introduction to system clocking

A **clock** may refer to any of the following:

1. In general, the **clock** refers to a microchip that regulates the timing and speed of all computer functions. Within this chip is a crystal that vibrates at a specific frequency when electricity is applied. The shortest time any computer is capable of performing is one clock, or one vibration of the clock chip. The speed of a computer processor is measured in clock speed, for example, 1

MHz is one million cycles, or vibrations, a second. 2 GHz is two billion cycles, or vibrations, a second.

2. Another name for the **internal clock** or **real-time clock (RTC).**

3. A **system clock** or **system timer** is a continuous pulse that helps the computer clock keep the correct time.

The hardware and software clocks

A personal computer has a battery driven hardware clock. The battery ensures that the clock will work even if the rest of the computer is without electricity. The hardware clock can be set from the BIOS setup screen or from whatever operating system is running.

The Linux kernel keeps track of time independently from the hardware clock. During the boot, Linux sets its own clock to the same time as the hardware clock. After this, both clocks run independently. Linux maintains its own clock because looking at the hardware is slow and complicated.

The kernel clock always shows universal time. This way, the kernel does not need to know about time zones at all. The simplicity results in higher reliability and makes it easier to update the time zone information. Each process handles time zone conversions itself (using standard tools that are part of the time zone package).

The hardware clock can be in local time or in universal time. It is usually better to have it in universal time, because then you don't need to change the hardware clock when daylight savings time begins or ends (UTC does not have DST). Unfortunately, some PC operating systems, including MS-DOS, Windows, and OS/2, assume the hardware clock shows local time. Linux can handle either, but if the hardware clock shows local time, then it must be modified when daylight savings time begins or ends (otherwise it wouldn't show local time).

Computer terminals

Computer Terminal Hardware

A computer terminal is an electronic or electromechanical hardware device that is used for entering data into, and displaying data from, a computer or a computing system. ... A terminal that depends on the host computer for its processing power is called a "dumb terminal" or thin client.

Data input-output device, usually made up of a monitor (display), keyboard, mouse, or touch screen. It is the point at which a user is connected to and communicate with a computer or a website through a network. Three basic types of terminals are (1) Dumb terminal: has no built-in

data processing capabilities and serves only to send and receive data, (2) Smart terminal: has limited data processing capabilities, and (3) Intelligent terminal: has substantial data processing capabilities due to inbuilt processor and memory.

Memory-Mapped I/O

Think of a disk controller and a read request. The goal is to copy data from the disk to some portion of the central memory. How do we do this?

- The controller contains a microprocessor and memory and is connected to the disk (by a cable).
- When the controller asks the disk to read a sector, the contents come to the controller via the cable and are stored by the controller in its memory.
- The question is how does the OS, which is running on another processor, let the controller know that a disk read is desired and how is the data eventually moved from the controller's memory to the general system memory.
- Typically the interface the OS sees consists of some device registers located on the controller.
  - These are memory locations into which the OS writes information such as sector to access, read vs. write, length, where in system memory to put the data (for a read) or from where to take the data (for a write).
  - There is also typically a device register that acts as a ``go button".
  - There are also devices registers that the OS reads, such as status of the controller, errors found, etc.
- So now the question is how does the OS read and write the device register o With **Memory-mapped I/O** the device registers appear as normal memory. All that is needed is to know at which address each device regester appears. Then the OS uses normal load and store instructions to write the registers.
  - Some systems instead have a special ``I/O space" into which the registers are mapped and require the use of special I/O space instructions to accomplish the load and store. From a conceptual point of view there is no difference between the two models.

Summary

Memory-mapped I/O uses a section of memory for I/O. The idea is simple. Instead of having "real" memory (i.e., RAM) at that address, place an I/O device.

Thus, communicating to an I/O device can be the same as reading and writing to memory addresses devoted to the I/O device. The I/O device merely has to use the same protocol to communicate with the CPU as memory uses.

Some ISAs use special I/O instructions. However, the signals generated by the CPU for I/O instructions and for memory-mapped I/O are nearly the same. Usually, there's just one special I/O pin that lets you know whether its a memory address or an I/O address. Other than that, they behave nearly identically.

Input/Output software

I/O (input/output) software, describes any operation by program that transfers data to storage or from computer storage to display or other outputting means. The I/O devices work with respective software to handle inputting and outputting.
Typical I/O devices are printers, hard disks, keyboards, and mouses. In fact, some devices are basically input-only devices (keyboards and mouses); others are primarily output-only devices (printers); and others provide both input and output of data (hard disks, diskettes, writable CDROMs).

Virtual devices

A virtual device, in operating systems, refers to a device file that has no associated hardware. A virtual device mimics a physical hardware device when, in fact, it exists only in software form. Therefore, it makes the system believe that a particular hardware exists when it really does not.

A virtual device is also known as a virtual peripheral.

Objective of Virtual devices

Virtual devices are generally used to fix an error in the operating system. For example, a bug or virus can be detected by supposing an external device is monitoring it.

History of Virtual devices

Initially, the command *mknod* was used to produce the character and block devices that populate the "/dev/" directory. But now the *udev* device manager automatically creates and destroys device nodes in the virtual file system. The supposed hardware (virtual device) is detected by the kernel, but, actually, it is only a file/directory.

Virtual Device Types

For each type of device, there is a set of the generic commands.
For example, for char device one set of commands and for block device there can be another set.
Types of Physical and Virtual devices in a system may be as follows: char, block, loop back device, file, pipe, socket, RAM disk, sound, video and media.

Virtual device driver

Definition: A virtual-device driver is the component of a device driver that communicates directly between an application and memory or a physical device.

- Virtual device driver controls the flow of data
- Allows more than one application to access the same memory or physical device without conflict.

Char Device: For example, a device to which one character is sent at one time or is read from it at one time.

For example, mouse, keyboard, keypad, timer.

Block Device: For example, a device to which one block of characters is sent at one time or is read from it at one time. For example, printer, disk

Loop-back Device: A device to which one character or set of characters are sent, and those are echoed back to same.

Copy Device: A device using which a set of characters are sent, and those are returned to another device. For example, disk_copy device when characters are copied from one disk to another or a keyboard-cum-display device. Keyboard input is sent to a buffer and display unit uses that buffer for display.

Virtual Devices

Besides the physical devices of a system, drivers are also used in a systems for virtual devices.

- Physical device drivers and virtual device drivers have analogies.
- Like physical device, virtual device drivers may also have functions for device connect or open, read, write and close.

Driver

A memory block can have data buffers for input and output in analogy to buffers at an IO device and can be accessed from a *char* driver or *block* or *pipe* or *socket* driver.

Virtual Device Examples

- Pipe device: A device from to which the blocks of characters are send from one end and accessed from another ends in FIFO mode (first-in first-out) after a connect function is executed to connect two ends.
- Socket device: A device from to which (a) the blocks of characters are send from one end with a set of the port (application) and sender addresses, (b) accessed from another end port (application) and receiver addresses, (c) access is in FIFO mode (first-in first-out) only after a connect function is executed to connect two sockets.
- File device: A device from which the blocks of characters are accessed similar to a disk in a tree like format (folder, subfolder,...). For example, named files at the memory stick.

- RAM disk Device: A set of RAM memory blocks used like a disk, which is accessed by defining addresses of directory, subdirectory, second level subdirectory, folder and subfolder

Difference between various types of virtual devices

- Pipe needs one address at an end,
- Socket one addresses and one port number at an end, and
- File and disk can have multiple addresses. Reading and writing into a file is from or to current cursor address in the currently open folder.
- Just as a file is sent *read* call, a device must be sent a driver command when its input buffer(s) is to be read**.**
- Just as a file is sent *write* call, a device needs to be sent a driver command when its output buffer is to be written.

CHAPTER 5: FILE MANAGEMENT

File management

Definition: File management describes the fundamental methods for naming, storing and handling files. **File management is the storing, naming, sorting and handling computer files.**

Objectives of File Management systems:

1) Guarantees data in the file is valid
2) Optimizes performance in terms of throughput & response time
3) Provide I/O support for storage device type
4) Provide I/O support for multiple users
5) Meet user requirements for data operations

File system

A file is a named collection of related information that is recorded on secondary storage such as magnetic disks, magnetic tapes and optical disks. In general, a file is a sequence of bits, bytes, lines or records whose meaning is defined by the files creator and user.

File Concept

- A file is a collection of similar records. The file is treated as a single entity by users and applications and may be referred by name. Files have unique file names and may be created and deleted. Restrictions on access control usually apply at the file level.

- A file is a container for a collection of information. The file manager provides a protection mechanism to allow users administrator how processes executing on behalf of different users can access the information in a file. File protection is a fundamental property of files because it allows different people to store their information on a shared computer.

- File represents programs and data. Data files may be numeric, alphabetic, binary or alpha numeric. Files may be free form, such as text files. In general, file is sequence of bits, bytes, lines or records.

- A file has a certain defined structure according to its type.
  1 Text File
  2 Source File
  3 Executable File
  4 Object File

File Structure

A File Structure should be according to a required format that the operating system can understand.

- A file has a certain defined structure according to its type.
- A text file is a sequence of characters organized into lines.
- A source file is a sequence of procedures and functions.
- An object file is a sequence of bytes organized into blocks that are understandable by the machine.
- When an operating system defines different file structures, it also contains the code to support these file structure. Unix, MS-DOS support minimum number of file structure.

File Structure Four terms are used for files
• Field
• Record
• Database
A field is the basic element of data. An individual field contains a single value. A record is a collection of related fields that can be treated as a unit by some application program.
A file is a collection of similar records. The file is treated as a singly entity by users and applications and may be referenced by name. Files have file names and maybe created and deleted. Access control restrictions usually apply at the file level.
A database is a collection of related data. Database is designed for use by a number of different applications. A database may contain all of the information related to an organization or project, such as a business or a scientific study. The database itself consists of one or more types of files. Usually, there is a separate database management system that is independent of the operating system.

File Attributes

File attributes vary from one operating system to another. The common attributes are,

- **Name** – only information kept in human-readable form.
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing

- **Time, date, and user identification** – data for protection, security, and usage monitoring Information about files are kept in the directory structure, which is maintained on the disk

File Operations

Any file system provides not only a means to store data organized as files, but a collection of functions that can be performed on files. Typical operations include the following:

Create: A new file is defined and positioned within the structure of files.

Delete: A file is removed from the file structure and destroyed.

Open: An existing file is declared to be "opened" by a process, allowing the process to perform functions on the file.

Close: The file is closed with respect to a process, so that the process no longer may perform functions on the file, until the process opens the file again. **Read:** A process reads all or a portion of the data in a file.

Write: A process updates a file, either by adding new data that expands the size of the file or by changing the values of existing data items in the file.

File Types – Name, Extension

A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts : a name and an extension. Following table gives the file type with usual extension and function.

File Type	Usual Extension	Function
Executable	exe, com, bin	Read to run machine language program.
Object	obj, o	Compiled, machine language, not linked
Source Code	c, cc, java, pas asm, a	Source code in various language
Text	txt, doc	Textual data, documents

File type refers to the ability of the operating system to distinguish different types of file such as text files source files and binary files etc. Many operating systems support many types of files. Operating system like MS-DOS and UNIX have the following types of files:

Ordinary files

- These are the files that contain user information.
- These may have text, databases or executable program.
- The user can apply various operations on such files like add, modify, delete or even remove the entire file.

Directory files

- These files contain list of file names and other information related to these files.

Special files

- These files are also known as device files.
- These files represent physical device like disks, terminals, printers, networks, tape drive etc.

These files are of two types:

- **Character special files** - data is handled character by character as in case of terminals or printers.
- **Block special files** - data is handled in blocks as in the case of disks and tapes.

File Management Systems:

A file management system is that set of system software that provides services to users and applications in the use of files. following objectives for a file management system:

- To meet the data management needs and requirements of the user which include storage of data and the ability to perform the aforementioned operations.
- To guarantee, to the extent possible, that the data in the file are valid.
- To optimize performance, both from the system point of view in terms of overall throughput.
- To provide I/O support for a variety of storage device types.
- To minimize or eliminate the potential for lost or destroyed data.
- To provide a standardized set of I/O interface routines to use processes.

TO provide I/O support for multiple users, in the case of multiple-user systems **File System Architecture.** At the lowest level, **device drivers** communicate directly with peripheral devices or their controllers or channels. A device driver is responsible for starting I/O operations on a device and processing the completion of an I/O request. For file operations, the typical devices controlled are disk and tape drives. Device drivers are usually considered to be part of the operating system.

The I/O control, consists of device drivers and interrupt handlers to transfer information between the memory and the disk system. A device driver can be thought of as a translator.
The basic file system needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk.
The file-organization module knows about files and their logical blocks, as well as physical blocks. By knowing the type of file allocation used and the location of the file, the fileorganization module can translate logical block addresses to physical block addresses for the basic file system to transfer. Each file's logical blocks are numbered from 0 (or 1) through N, whereas the physical blocks containing the data usually do not match the logical numbers, so a translation is needed to locate each block. The file-organization module also includes the freespace manager, which tracks unallocated and provides these blocks to the file organization module when requested.

The logical file system uses the directory structure to provide the file-organization module with the information the latter needs, given a symbolic file name. The logical file system is also responsible for protection and security.
To create a new file, an application program calls the logical file system. The logical file system knows the format of the directory structures. To create a new file, it reads the appropriate directory into memory, updates it with the new entry, and writes it back to the disk. Once the file is found the associated information such as size, owner, access permissions and data block locations are generally copied into a table in memory, referred to as the open-file fable, consisting of information about all the currently opened files.
The first reference to a file (normally an open) causes the directory structure to be searched and the directory entry for this file to be copied into the table of opened files. The index into this table is returned to the user program, and all further references are made through the index rather than with a symbolic name.

The name given to the index varies. Unix systems refer to it as a file descriptor, Windows/NT as a file handle, and other systems as a file control block.
Consequently, as long as the file is not closed, all file operations are done on the open-file table. When the file is closed by all users that have opened it, the updated file information is copied back to the disk-based directory structure.

File-System Mounting

As a file must be opened before it is used, a file system must be mounted before it can be available to processes on the system. The mount procedure is straight forward. The stem is given the name of the device, and the location within the file structure at which to attach the file system (called the mount point).

The operating system verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format. Finally, the operating system notes in its directory structure that a file system is mounted at the specified mount point. This scheme enables the operating system to traverse its directory structure, switching among file systems as appropriate.

File Access Mechanisms

File access mechanism refers to the manner in which the records of a file may be accessed. There are several ways to access files:

- Sequential access
- Direct/Random access
- Indexed sequential access

Sequential access

A sequential access is that in which the records are accessed in some sequence, i.e., the information in the file is processed in order, one record after the other. This access method is the most primitive one. Example: Compilers usually access files in this fashion.

Direct/Random access

- Random access file organization provides, accessing the records directly.
- Each record has its own address on the file with by the help of which it can be directly accessed for reading or writing.
- The records need not be in any sequence within the file and they need not be in adjacent locations on the storage medium.

Indexed sequential access

- This mechanism is built up on base of sequential access.
- An index is created for each file which contains pointers to various blocks.

- Index is searched sequentially and its pointer is used to access the file directly.

Space Allocation

Files are allocated disk spaces by operating system. Operating systems deploy following three main ways to allocate disk space to files.

- Contiguous Allocation
- Linked Allocation

- Indexed Allocation

Contiguous Allocation

- Each file occupies a contiguous address space on disk.

- Assigned disk address is in linear order.

- Easy to implement.

- External fragmentation is a major issue with this type of allocation technique.

Linked Allocation

- Each file carries a list of links to disk blocks.

- Directory contains link / pointer to first block of a file.

- No external fragmentation

- Effectively used in sequential access file.

- Inefficient in case of direct access file.

Indexed Allocation

- Provides solutions to problems of contiguous and linked allocation.

- An index block is created having all pointers to files.
- Each file has its own index block which stores the addresses of disk space occupied by the file.

- Directory contains the addresses of index blocks of files.