Introduction to Operating System
PCSC-301 (For UG students)

(Class notes and reference books are required to complete this study)

Release Date: 27.12.2014

Operating System – Objectives and Functions

An OS is a program that controls the execution of application programs and acts as an interface between applications and the computer hardware. It can be thought of as having three objectives:

• **Convenience:** An OS makes a computer more convenient to use.

• **Efficiency:** An OS allows the computer system resources to be used in an efficient manner.

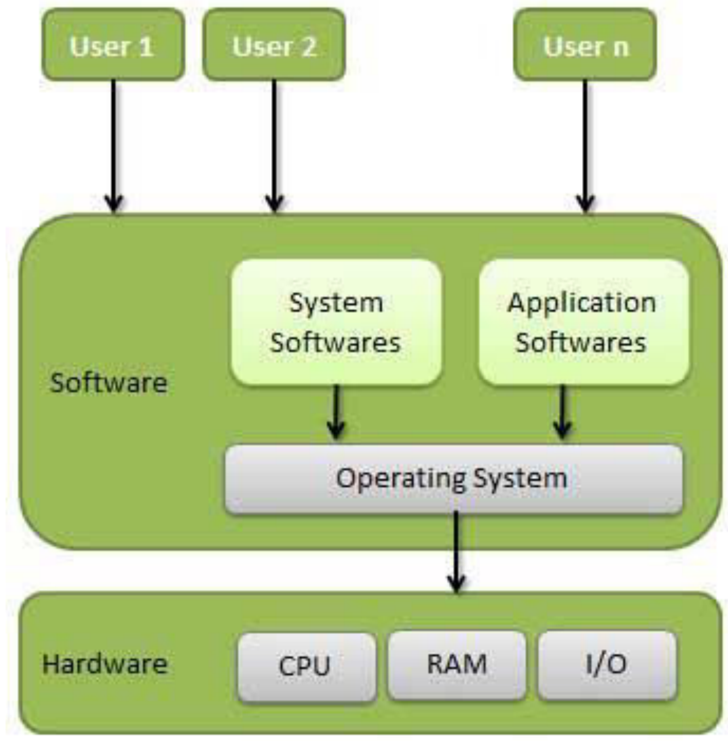
• **Ability to evolve:** An OS should be constructed in such a way as to permit the effective development, testing, and introduction of new system functions without interfering with service.

Let us examine these three aspects of an OS in turn.

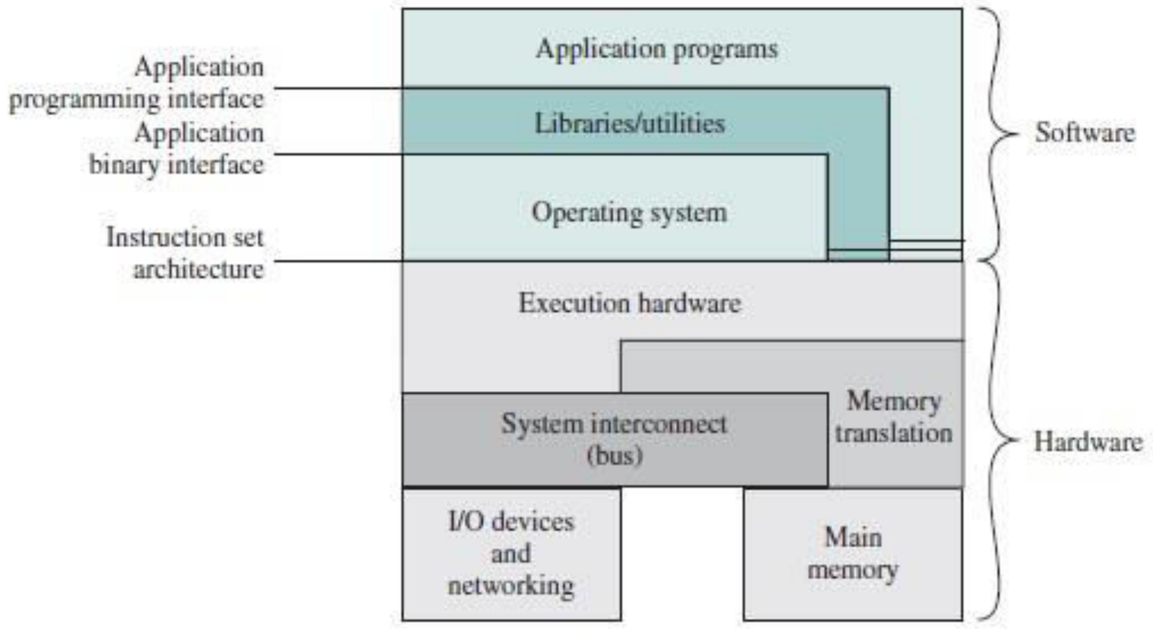
1. The Operating System as a User/Computer Interface

The hardware and software used in providing applications to a user can be viewed in a layered or hierarchical fashion, as depicted in Figure below. The user of those applications, the end user, generally is not concerned with the details of computer hardware. Thus, the end user views a computer system in terms of a set of applications.

An application can be expressed in a programming language and is developed by an application programmer. If one were to develop an application program as a set of machine instructions that is completely responsible for controlling the computer hardware, one would be faced with an overwhelmingly complex undertaking. To ease this chore, a set of system programs is provided. Some of these programs are referred to as utilities, or library programs. These implement frequently used functions that assist in program creation, the management of files, and the control of I/O devices. A programmer will make use of these facilities in developing an application, and the application, while it is running, will invoke the utilities to perform certain functions. The most important collection of system programs comprises the OS. The OS masks the details of the hardware from the programmer and provides the programmer with a convenient interface for using the system. It acts as mediator, making it easier for the programmer and for application programs to access and use those facilities and services.



Details of interfaces:


Application programming interface
Application binary interface
Instruction set architecture
Application programs
Libraries/utilities
Operating system
Execution hardware
Memory translation
System interconnect (bus)
I/O devices and networking
Main memory
Software
Hardware

Computer Hardware and Software Structure

Briefly, the OS typically provides services in the following areas:

• **Program development:** The OS provides a variety of facilities and services, such as editors and debuggers, to assist the programmer in creating programs. Typically, these services are in the form of utility programs that, while not strictly part of the core of the OS, are supplied with the OS and are referred to as application program development tools.

• **Program execution:** A number of steps need to be performed to execute a program. Instructions and data must be loaded into main memory, I/O devices and files must be initialized, and other resources must be prepared. The OS handles these scheduling duties for the user.

• **Access to I/O devices:** Each I/O device requires its own peculiar set of instructions or control signals for operation. The OS provides a uniform interface that hides these details so that programmers can access such devices using simple reads and writes.

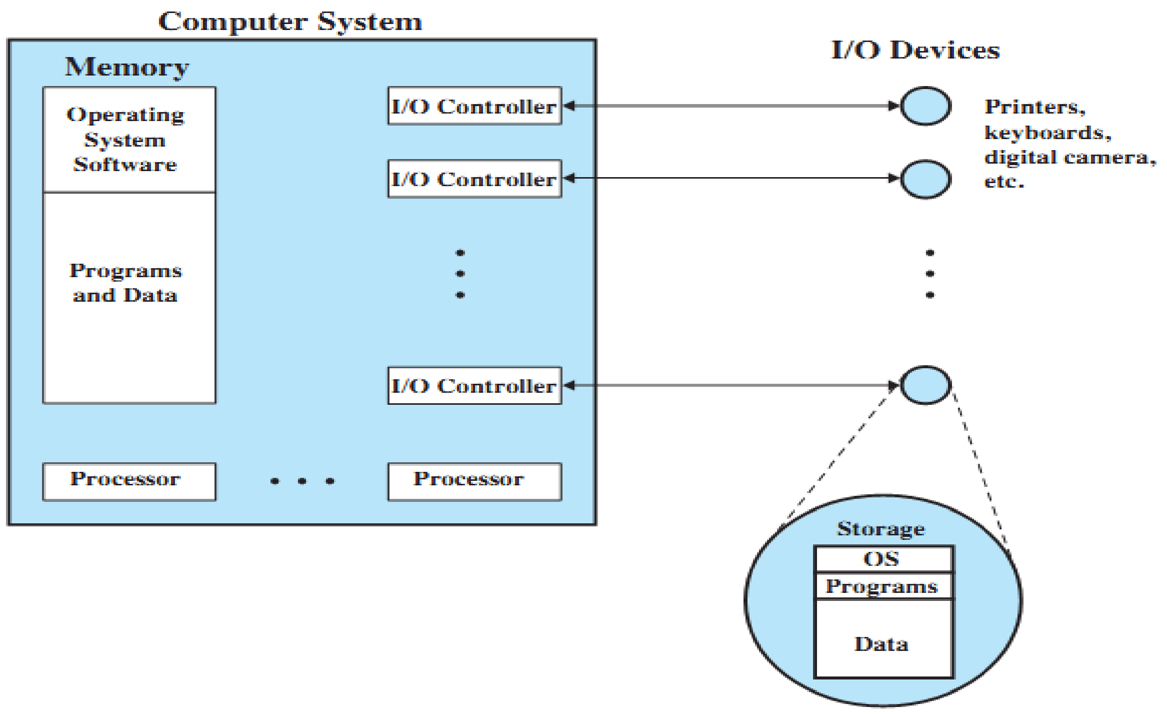
• **Controlled access to files:** For file access, the OS must reflect a detailed understanding of not only the nature of the I/O device (disk drive, tape drive) but also the structure of the data contained in the files on the storage medium. In the case of a system with multiple users, the OS may provide protection mechanisms to control access to the files.

• **System access:** For shared or public systems, the OS controls access to the system as a whole and to specific system resources. The access function must provide protection of resources and data from unauthorized users and must resolve conflicts for resource contention.

• **Error detection and response:** A variety of errors can occur while a computer system is running. These include internal and external hardware errors, such as a memory error, or a device failure or malfunction; and various software errors, such as division by zero, attempt to access forbidden memory location, and inability of the OS to grant the request of an application. In each case, the OS must provide a response that clears the error condition with the least impact on running applications. The response may range from ending the program that caused the error, to retrying the operation, to simply reporting the error to the application.

• **Accounting:** A good OS will collect usage statistics for various resources and monitor performance parameters such as response time. On any system, this information is useful in anticipating the need for future enhancements and in tuning the system to improve performance. On a multiuser system, the information can be used for billing purposes.

2. The Operating System as Resource Manager



The figure above suggests the main resources that are managed by the OS. A portion of the OS is in main memory. This includes the **kernel** , or **nucleus** , which contains the most frequently used functions in the OS and, at a given time, other portions of the OS currently in use. The remainder of main memory contains user programs and data. The memory management hardware in the processor and the OS jointly control the allocation of main memory, as we shall see. The OS decides when an I/O device can be used by a program in execution and controls access to and use of files. The processor itself is a resource, and the OS must determine how much processor time is to be devoted to the execution of a particular user program. In the case of a multiple-processor system, this decision must span all of the processors.

3. Ease of Evolution of an Operating System

A major OS will evolve over time for a number of reasons:

• **Hardware upgrades plus new types of hardware:** For example, early versions of UNIX and the Macintosh OS did not employ a paging mechanism because they were run on processors without paging hardware. 1 Subsequent versions of these operating systems were modified to exploit paging capabilities. Also, the use of graphics terminals and page-mode terminals instead of line-at-a-time scroll mode terminals affects OS design. For example, a graphics terminal typically allows the user to view several applications at the same time through "windows" on the screen. This requires more sophisticated support in the OS.

• **New services:** In response to user demand or in response to the needs of system managers, the OS expands to offer new services. For example, if it is found to be difficult to maintain good performance for users with existing tools, new measurement and control tools may be added to the OS.

• **Fixes:** Any OS has faults. These are discovered over the course of time and fixes are made. Of course, the fix may introduce new faults.

CPU States

The first four states in the list below are found on most every Unix system. Other state names may appear on different platforms, and some of these are also listed below.

Idle	Nothing to do
User	Running a user's process
Kernel	Handling a kernel call, fault, or interrupt
Nice	Running a user's niced process
Wait	Waiting on some form of i/o
iowait	Waiting on user i/o
Swap	Waiting on swapping or paging i/o

I/O Channels or I/O Processors

The disparity between the I/O devices and the CPU motivated the development of I/O Processors (also called I/O channels).

Function: provide data flow path between I/O devices and memory.

Characteristics:
- Simple (minimal processing capabilities).
- Specialized, and not too fast, and much less expensive than conventional CPU.

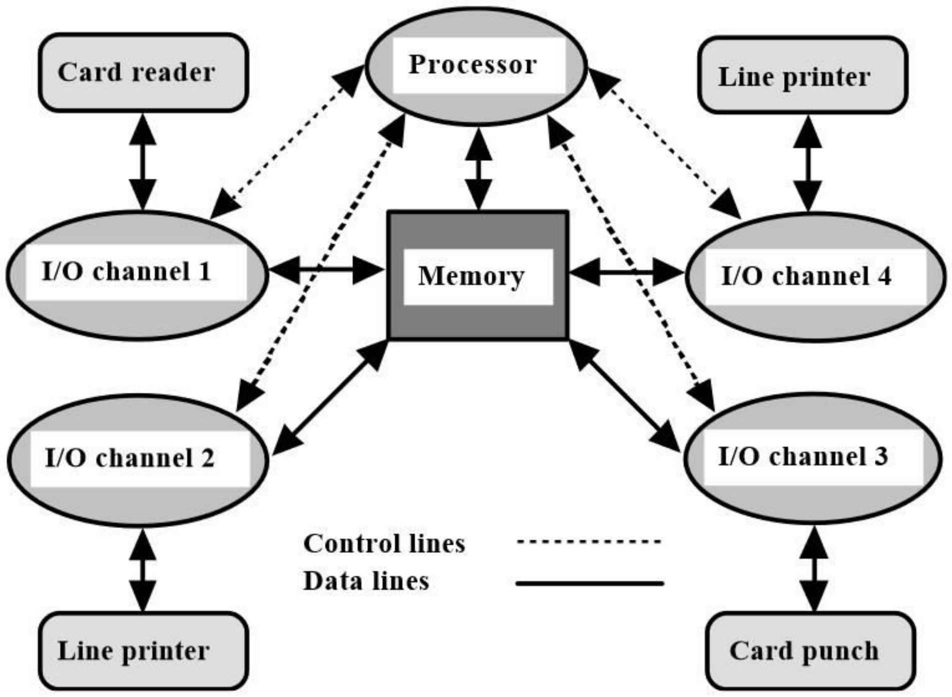
Computer systems that use channel I/O have special hardware components that handle all input/output operations in their entirety independently of the systems' CPU(s). The CPU of a system that uses channel I/O typically has only one machine instruction in its repertoire for input and output; this instruction is used to pass input/output commands to the specialized I/O hardware in the form of channel programs. I/O thereafter proceeds without intervention from the CPU until an event requiring notification of the operating system occurs, at which point the I/O hardware signals an interrupt to the CPU.

A channel is an independent hardware component that coordinates all I/O to a set of controllers or devices. It is not merely a medium of communication, despite the name; it is a *programmable* device that handles all details of I/O after being given a list of I/O operations to carry out (the channel program).

Each channel may support one or more controllers and/or devices, but each channel program may only be directed at one of those connected devices. A channel program contain lists of commands to the channel itself and to the controller and device to which it is directed. Once the operating system has prepared a complete list of channel commands, it executes a single I/O machine instruction to initiate the channel program; the channel thereafter assumes control of the I/O operations until they are completed.

It is possible to develop very complex channel programs, including testing of data and conditional branching within that channel program. This flexibility frees the CPU from the overhead of starting, monitoring, and managing individual I/O operations. The specialized channel hardware, in turn, is dedicated to I/O and can carry it out more efficiently than the CPU (and entirely in parallel with the CPU). Channel I/O is not unlike the Direct Memory Access (DMA) of microcomputers, only more complex and advanced. Most mainframe operating systems do not fully exploit all the features of channel I/O.

On large mainframe computer systems, CPUs are only one of several powerful hardware components that work in parallel. Special input/output controllers (the exact names of which vary from one manufacturer to another) handle I/O exclusively, and these in turn are connected to hardware channels that also are dedicated to input and output. There may be several CPUs and several I/O processors. The overall architecture optimizes input/output performance without degrading pure CPU performance. Since most real-world applications of mainframe systems are heavily I/O-intensive business applications, this architecture helps provide the very high levels of throughput that distinguish mainframes from other types of computer.



Types of I/O channels

Memory Hierarchy

The design constraints on a computer's memory can be summed up by three questions: How much? How fast? How expensive? i.e. capacity, speed and cost.
The question of how much is somewhat open ended. If the capacity is there, applications will likely be developed to use it. The question of how fast is, in a sense, easier to answer. To achieve greatest performance, the memory must be able to keep up with the processor. That is, as the processor is executing instructions, we would not want it to have to pause waiting for instructions or operands. The final question must also be considered. For a practical system, the cost of memory must be reasonable in relationship to other components. As might be expected, there is a trade-off among the three key characteristics of memory: namely, capacity, access time, and cost. A variety of technologies are used to implement memory systems, and across this spectrum of technologies, the following relationships hold:

• Faster access time, greater cost per bit

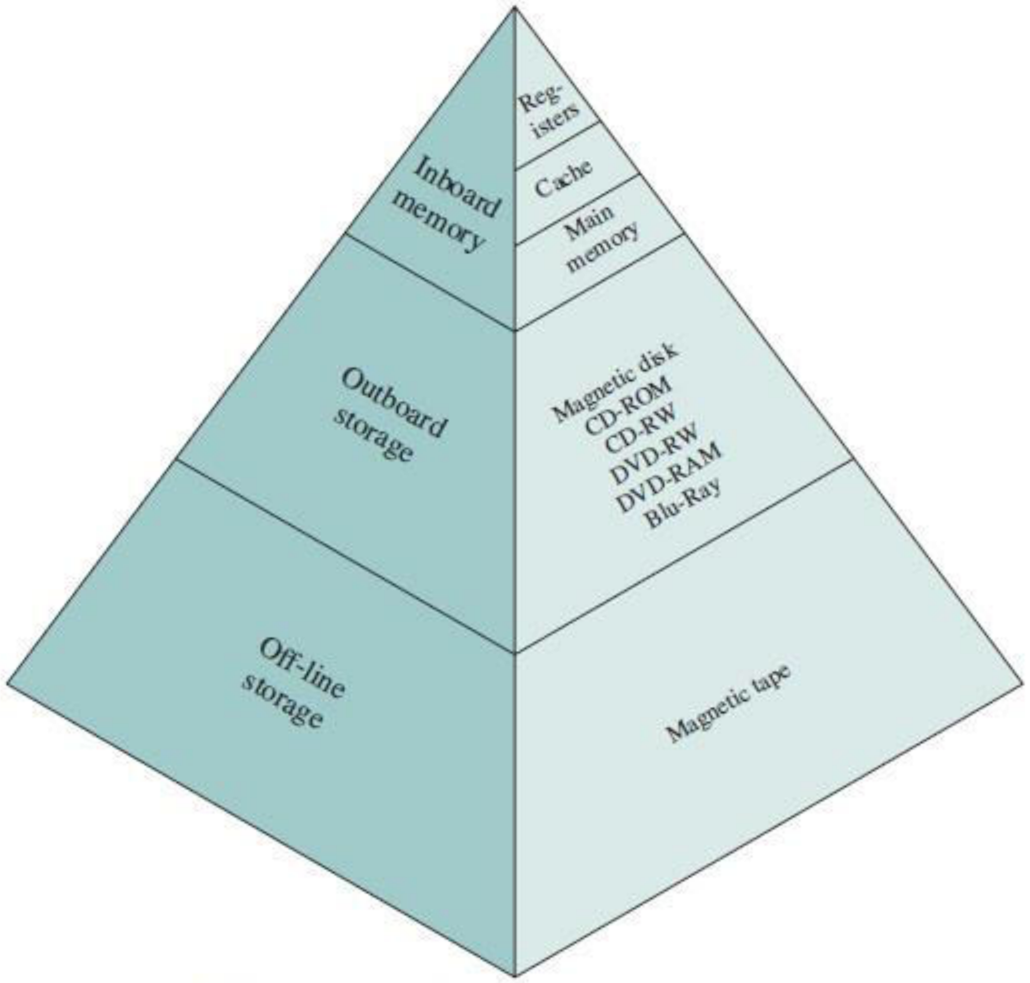
• Greater capacity, smaller cost per bit

• Greater capacity, slower access speed

The dilemma facing the designer is clear. The designer would like to use memory technologies that provide for large-capacity memory, both because the capacity is needed and because the cost per bit is low. However, to meet performance requirements, the designer needs to use expensive, relatively lower-capacity memories with fast access times. The way out of this dilemma is to not rely on a single memory component or technology, but to employ a **memory hierarchy** . A typical hierarchy is illustrated in Figure given below. As one goes down the hierarchy, the following occur:

a. Decreasing cost per bit
b. Increasing capacity
c. Increasing access time
d. Decreasing frequency of access to the memory by the processor

Thus, smaller, more expensive, faster memories are supplemented by larger, cheaper, slower memories. The key to the success of this organization is the decreasing frequency of access at lower levels. We will examine this concept in greater detail later in this chapter, when we discuss the cache, and when we discuss virtual memory later in this book. A brief explanation is provided at this point.
Suppose that the processor has access to two levels of memory. Level 1 contains 1,000 bytes and has an access time of 0.1 μs; level 2 contains 100,000 bytes and has an access time of 1 μs. Assume that if a byte to be accessed is in level 1, then the processor accesses it directly. If it is in level 2, then the byte is first transferred to level 1 and then accessed by the processor. For simplicity, we ignore the time required for the processor to determine whether the byte is in level 1 or level 2.



The Memory Hierarchy

Types of OS

Simple Batch Systems

Early computers were very expensive, and therefore it was important to maximize processor utilization. The wasted time due to scheduling and setup time was unacceptable. To improve utilization, the concept of a batch OS was developed. It appears that the first batch OS (and the first OS of any kind) was developed in the mid-1950s by General Motors for use on an IBM 701 [WEIZ81]. The concept was subsequently refined and implemented on the IBM 704 by a number of IBM customers. By the early 1960s, a number of vendors had developed batch operating systems for their computer systems. IBSYS, the IBM OS for the 7090/7094 computers, is particularly notable because of its widespread influence on other systems. The central idea behind the simple batch-processing scheme is the use of a piece of software known as the **monitor.** With this type of OS, the user no longer has direct access to the processor. Instead, the user submits the job on cards or tape to a computer operator, who batches the jobs together sequentially and places the entire batch on an input device, for use by the monitor. Each program is constructed to branch back to the monitor when it completes processing, at which point the monitor automatically begins loading the next program.

The problems with Batch Systems are following.

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

Multiprogramming Operating Systems

To overcome the problem of underutilization of CPU and main memory, the multiprogramming was introduced. The multiprogramming is interleaved execution of multiple jobs by the same computer.

In multiprogramming system, when one program is waiting for I/O transfer; there is another program ready to utilize the CPU. So it is possible for several jobs to share the time of the CPU. But it is important to note that multiprogramming is not defined to be the execution of jobs at the same instance of time. Rather it does mean that there are a number of jobs available to the CPU (placed in main memory) and a portion of one is executed then a segment of another and so on.

Multiprogramming operating systems are fairly sophisticated compared to single-program, or **uniprogramming**, systems. To have several jobs ready to run, they must be kept in main memory, requiring some form of **memory management.** In addition, if several jobs are ready to run, the processor must decide which one to run; this decision requires an algorithm for scheduling means extra CPU time to process on this scheduling.

Multitasking Operating Systems

Multitasking, in an operating system, is allowing a user to perform more than one computer task (such as the operation of an application program) at a time. The operating system is able to keep track of where you are in these tasks and go from one to the other without losing information. Microsoft Windows 2000 / 7 / 8, IBM's OS/390, and Linux etc. are examples of operating systems that can do multitasking (almost all of today's operating systems can). When you open your Web browser and then open Word at the same time, you are causing the operating system to do multitasking.

Multitasking has the same meaning as multiprogramming in the general sense as both refer to having multiple (programs, processes, tasks, threads) running at the same time. Multitasking is the term used in modern operating systems when multiple tasks share a common processing resource (CPU and Memory). At any point in time the CPU is executing one task only while other tasks waiting their turn. The illusion of parallelism is achieved when the CPU is reassigned to another task (context switch). There are few main differences between multitasking and multiprogramming. A task in a multitasking operating system is not a whole application program (recall that programs in modern operating systems are divided into logical pages). Task can also refer to a thread of execution when one process is divided into sub tasks (will talk about multi threading later). The task does not hijack the CPU until it finishes like in the older multiprogramming model but rather have a fair share amount of the CPU time called quantum (will talk about time sharing later in this article). Just to make it easy to remember, multitasking and multiprogramming refer to a similar concept (sharing CPU time) where one is used in modern operating systems while the other is used in older operating systems. Multitasking always refers to multiprogramming but vice versa is not true.

Being able to do multitasking doesn't mean that an unlimited number of tasks can be juggled at the same time. Each task consumes system storage and other resources. As more tasks are started, the system may slow down or begin to run out of shared storage.

Time-sharing Operating Systems

With the use of multiprogramming, batch processing can be quite efficient. However, for many jobs, it is desirable to provide a mode in which the user interacts directly with the computer. Indeed, for some jobs, such as transaction processing, an interactive mode is essential.

Today, the requirement for an interactive computing facility can be, and often is, met by the use of a dedicated personal computer or workstation. That option was not available in the 1960s, when most computers were big and costly. Instead, time sharing was developed. Just as multiprogramming allows the processor to handle multiple batch jobs at a time, multiprogramming can also be used to handle multiple interactive jobs. In this latter case, the technique is referred to as **time-sharing,** because processor time is shared among multiple users. In a time-sharing system, multiple users simultaneously access the system through terminals, with the OS interleaving the execution of each user program in a short burst or quantum of computation. Thus, if there are *n* users actively requesting service at one time, each user will only see on the average 1/ *n* of the effective computer capacity, not counting OS overhead. However, given the relatively slow human reaction time, the response time on a properly designed system should be similar to that on a dedicated computer. Both batch processing and time sharing use multiprogramming.

One of the first time-sharing operating systems to be developed was the Compatible Time-Sharing System (CTSS) [CORB62], developed at MIT by a group known as Project MAC (Machine-Aided Cognition, or Multiple-Access Computers). The system was first developed for the IBM 709 in 1961 and later transferred to an IBM 7094.

Recall that in a single processor system, parallel execution is an illusion. One instruction from one process at a time can be executed by the CPU even though multiple processes reside in main memory. Imagine a restaurant with only one waiter and few customers. There is no way for the waiter to serve more than one customer at a time but if it happens that the waiter is fast enough to rotate on the tables and provide food quickly then you get the feeling that all customers are being served at the same time. This is the example of time sharing when CPU time (or waiter time) is being shared between processes (customers). Multiprogramming and multitasking operating systems are nothing but time sharing systems. In multiprogramming though the CPU is shared between programs it is not the perfect example on CPU time sharing because one program keeps running until it blocks however in a multitasking (modern operating system) time sharing is best manifested because each running process takes only a fair amount of the CPU time called quantum time. Even in a multiprocessing system when we have more than one processor still each processor time is shared between running processes. Time-sharing is implemented to have a better response time.

Compatible Time-Sharing System or CTSS, was demonstrated in November 1961. CTSS has a good claim to be the first time-sharing system and remained in use until 1973. Another contender for the first demonstrated time-sharing system was PLATO II, created by Donald Bitzer.

Real-time Operating System

It is a multitasking operating system that aims at executing real-time applications. Real-time operating systems often use specialized scheduling algorithms so that they can achieve a deterministic nature of behavior. The main object of real-time operating systems is their quick and predictable response to events. They either have an event-driven design or a time-sharing one. An event-driven system switches between tasks based of their priorities while time-sharing operating systems switch tasks based on clock interrupts.

An RTOS performs all general purpose tasks of an OS, but is also specially designed to run applications with very precise timing and a high degree of reliability. This can be especially important in measurement and automation systems where downtime is costly or a program delay could cause a safety hazard.

To be considered "real-time", an operating system must have a known maximum time for each of the critical operations that it performs (or at least be able to guarantee that maximum most of the time). Some of these operations include OS calls and interrupt handling. Operating systems that can absolutely guarantee a maximum time for these operations are commonly referred to as "hard real-time", while operating systems that can only guarantee a maximum most of the time are referred to as "soft real-time". In practice, these strict categories have limited usefulness - each RTOS solution demonstrates unique performance characteristics and the user should carefully investigate these characteristics.

Windows CE, OS-9, Symbian and LynxOS are some of the commonly known real-time operating systems.

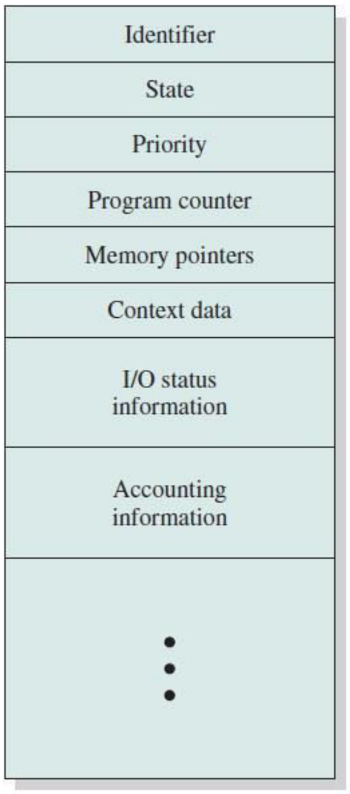
Process

What is a process?

• A program in execution
• An instance of a program running on a computer
• The entity that can be assigned to and executed on a processor
• A unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system resources.

We can also think of a process as an entity that consists of a number of elements. Two essential elements of a process are **program code** (which may be shared with other processes that are executing the same program) and a **set of data** associated with that code. Let us suppose that the processor begins to execute this program code, and we refer to this executing entity as a process. At any given point in time, *while the program is executing,* this process can be uniquely characterized by a number of elements which can be found in a Process Control Block.

Process Control Block



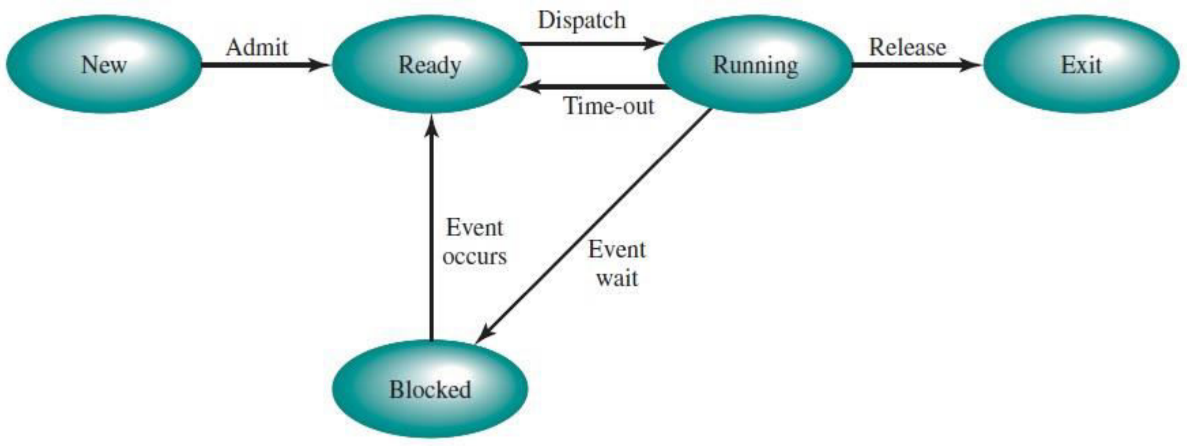
Simplified Process Control Block

The information in the preceding list is stored in a data structure, typically called a **process control block** (Figure above), that is created and managed by the OS. The significant point about the process control block is that it contains sufficient information so that it is possible to interrupt a running process and later resume execution as if the interruption had not occurred. The process control block is the key tool that enables the OS to support multiple processes and to provide for multiprocessing. When a process is interrupted, the current values of the program counter and the processor registers (context data) are saved in the appropriate fields of the corresponding process control block, and the state of the process is changed to some other value, such as *blocked* or *ready* (described subsequently). The OS is now free to put some other process in the running state. The program counter and context data for this process are loaded into the processor registers and this process now begins to execute.

We can say that a process consists of program code and associated data plus a process control block. For a single-processor computer, at any given time, at most one process is executing and that process is in the *running* state.

• **Identifier:** A unique identifier associated with this process, to distinguish it from all other processes.
• **State:** If the process is currently executing, it is in the running state.
• **Priority:** Priority level relative to other processes.
• **Program counter:** The address of the next instruction in the program to be executed.
• **Memory pointers:** Includes pointers to the program code and data associated with this process, plus any memory blocks shared with other processes.
• **Context data:** These are data that are present in registers in the processor while the process is executing.
• **I/O status information:** Includes outstanding I/O requests, I/O devices (e.g., disk drives) assigned to this process, a list of files in use by the process, and so on.
• **Accounting information:** May include the amount of processor time and clock time used, time limits, account numbers, and so on.

Process states



Five-State Process Model

• **Running:** The process that is currently being executed. For this chapter, we will assume a computer with a single processor, so at most one process at a time can be in this state.
• **Ready:** A process that is prepared to execute when given the opportunity.

• **Blocked/Waiting:** A process that cannot execute until some event occurs, such as the completion of an I/O operation.
• **New:** A process that has just been created but has not yet been admitted to the pool of executable processes by the OS. Typically, a new process has not yet been loaded into main memory, although its process control block has been created.
• **Exit:** A process that has been released from the pool of executable processes by the OS, either because it halted or because it aborted for some reason.

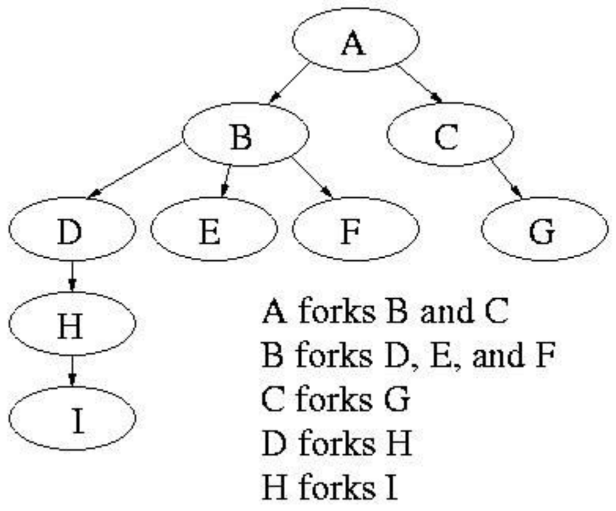
The types of events that lead to each state transition for a process; the possible transitions are as follows:
• **Null**: **New:** A new process is created to execute a program. This event occurs for any specific reason.
• **New** : **Ready:** The OS will move a process from the New state to the Ready state when it is prepared to take on an additional process. Most systems set some limit based on the number of existing processes or the amount of virtual memory committed to existing processes. This limit assures that there are not so many active processes as to degrade performance.
• **Ready** : **Running:** When it is time to select a process to run, the OS chooses one of the processes in the Ready state. This is the job of the scheduler or dispatcher. Scheduling is explored in Part Four.
• **Running**: **Exit:** The currently running process is terminated by the OS if the process indicates that it has completed, or if it aborts.
• **Running** : **Ready:** The most common reason for this transition is that the running process has reached the maximum allowable time for uninterrupted execution; virtually all multiprogramming operating systems impose this type of time discipline. There are several other alternative causes for this transition, which are not implemented in all operating systems. Of particular importance is the case in which the OS assigns different levels of priority to different processes. Suppose, for example, that process A is running at a given priority level, and process B, at a higher priority level, is blocked. If the OS learns
that the event upon which process B has been waiting has occurred, moving B to a ready state, then it can interrupt process A and dispatch process B. We say that the OS has **preempted** process A. 6 Finally, a process may voluntarily release control of the processor. An example is a background process that performs some accounting or maintenance function periodically.
• **Running**: **Blocked:** A process is put in the Blocked state if it requests something for which it must wait. A request to the OS is usually in the form of a system service call; that is, a call from the running program to a procedure that is part of the operating system code. For example, a process may request a service from the OS that the OS is not prepared to perform immediately. It can request a resource, such as a file or a shared section of virtual memory,
that is not immediately available. Or the process may initiate an action, such as an I/O operation, that must be completed before the process can continue. When processes communicate with each other, a process may be blocked when it is waiting for another process to provide data or waiting for a message from another process.
• **Blocked**: **Ready:** A process in the Blocked state is moved to the Ready state when the event for which it has been waiting occurs.
• **Ready** : **Exit:** For clarity, this transition is not shown on the state diagram. In some systems, a parent may terminate a child' process at any time. Also, if a parent terminates, all child processes associated with that parent may be terminated.
• **Blocked**: **Exit:** The comments under the preceding item apply.

Process Hierarchy

Modern general purpose operating systems permit a user to create and destroy processes.

In unix this is done by the fork system call, which creates a child process, and the exit system call, which terminates the current process.
After a fork both parent and child keep running (indeed they have the same program text) and each can fork off other processes.
A process tree results. The root of the tree is a special process created by the OS during startup.
A process can choose to wait for children to terminate. For example, if C issued a wait() system call it would block until G finished.

Old or primitive operating system like MS-DOS are not multiprogrammed so when one process starts another, the first process is automatically blocked and waits until the second is finished.



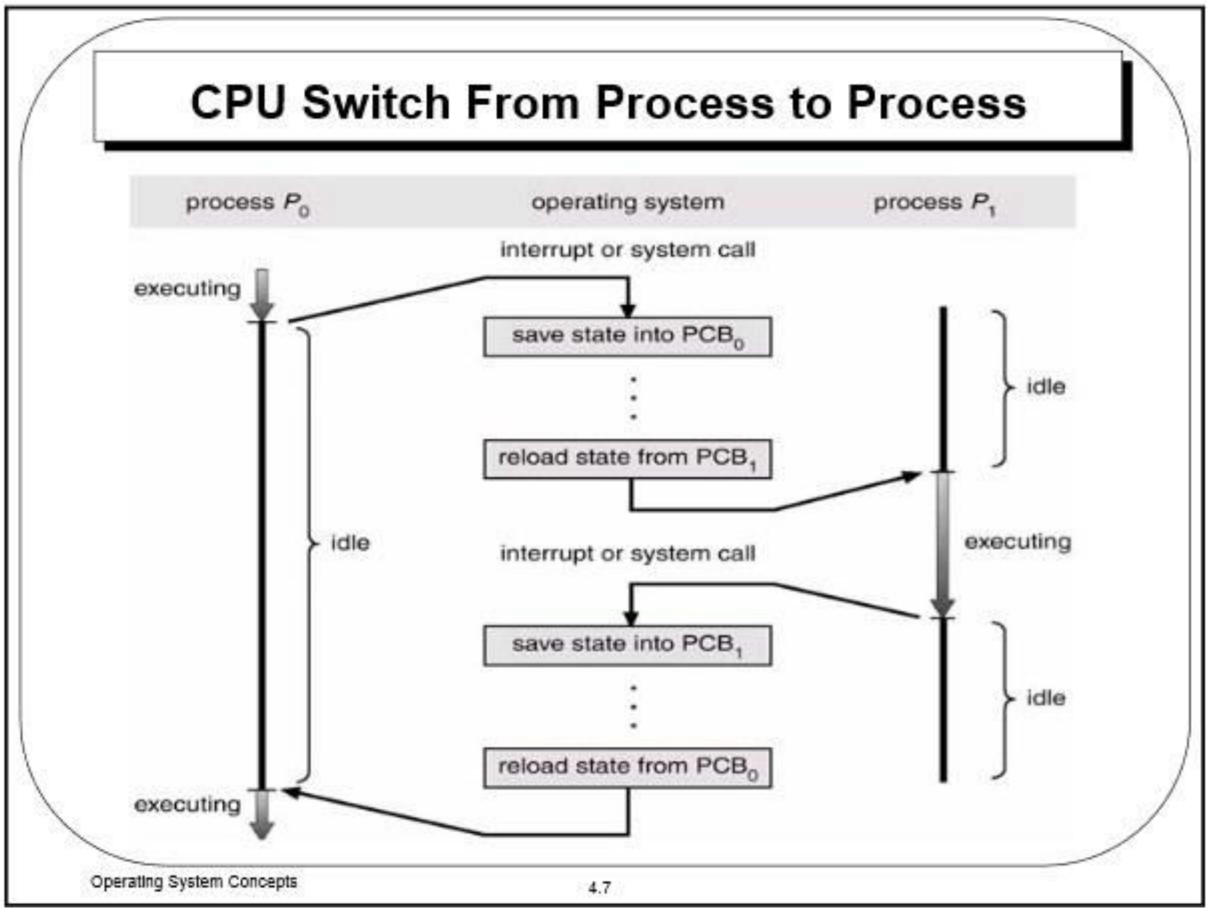
Context Switching

Context Switch - When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process.

Context-Switch Time is overhead; the system does no useful work while switching.

Context-Switch Time depends on hardware support.

Context-Switch Speed varies from machine to machine depending on memory speed, number of registers copied. The speed ranges from 1 to 1000 microsecond



Schedulers

Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue (i.e, selects processes from pool (disk) and loads them into memory for execution).

Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU (i.e, selects from among the processes that are ready to execute, and allocates the CPU to one of them).

Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast).

Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow).

The long-term scheduler controls the degree of multiprogramming (the number of processes in memory)

Medium-term scheduler – to remove processes from memory and reduce the degree of multiprogramming (the process is swapped out and swapped in by the medium-term scheduler).

Independent and Cooperating Process

Independent process cannot affect or be affected by the execution of another process.

Cooperating process can affect or be affected by the execution of another process.

Any process that shares data with other processes is a **cooperating process**.

CPU Scheduling

Scheduling criteria is also called as scheduling methodology. Key to multiprogramming is scheduling. Different CPU scheduling algorithm have different properties .The criteria used for comparing these algorithms include the following:

CPU Utilization: Keep the CPU as busy as possible. It range from 0 to 100%. In practice, it ranges from 40 to 90%.

Throughput: Throughput is the rate at which processes are completed per unit of time.

Turnaround time: This is the how long a process takes to execute a process. It is calculated as the time gap between the submission of a process and its completion.

Waiting time: Waiting time is the sum of the time periods spent in waiting in the ready queue.

Response time: Response time is the time it takes to start responding from submission time. It is calculated as the amount of time it takes from when a request was submitted until the first response is produced.

Fairness: Each process should have a fair share of CPU.

Non-preemptive Scheduling:

In non-preemptive mode, once if a process enters into running state, it continues to execute until it terminates or blocks itself to wait for Input/Output or by requesting some operating system service.

Preemptive Scheduling:

In preemptive mode, currently running process may be interrupted and moved to the ready state by the operating system.

When a new process arrives or when an interrupt occurs, preemptive policies may incur greater overhead than non-preemptive version but preemptive version may provide better service.

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

Scheduling Algorithms

Scheduling algorithms or scheduling policies are mainly used for short-term scheduling. The main objective of short-term scheduling is to allocate processor time in such a way as to optimize one or more aspects of system behavior.

For these scheduling algorithms assume only a single processor is present. Scheduling algorithms decide which of the processes in the ready queue is to be allocated to the CPU is basis on the type of scheduling policy and whether that policy is either preemptive or non-preemptive. For scheduling arrival time and service time are also will play a role.

List of scheduling algorithms are as follows:

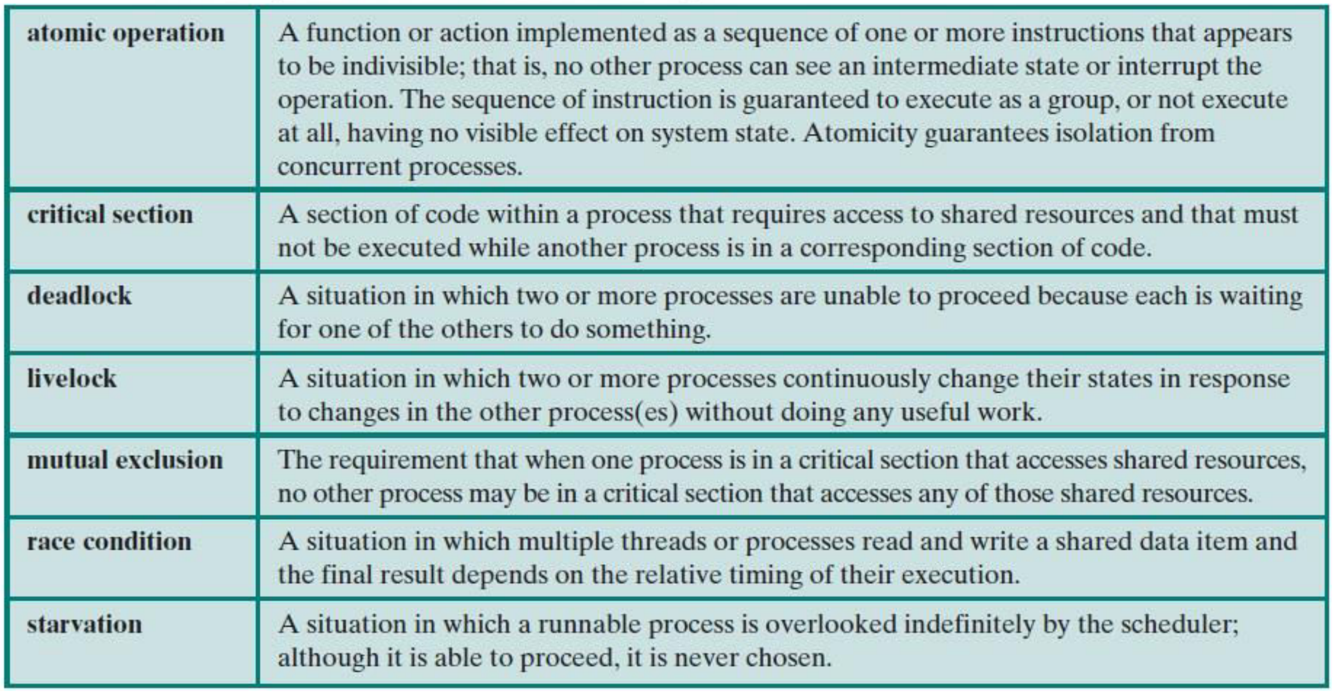
Non Preemptive	**Preemptive**
First-come, first-served scheduling (FCFS)	Round-Robin (RR)
Shortest Job First Scheduling (SJF)	Shortest Remaining time (SRT)

Concurrency

The central themes of operating system design are all concerned with the management of processes and threads:
• **Multiprogramming:** The management of multiple processes within a uniprocessor system
• **Multiprocessing** : The management of multiple processes within a multiprocessor
• **Distributed processing:** The management of multiple processes executing on multiple, distributed computer systems. The recent proliferation of clusters is a prime example of this type of system.

Fundamental to all of these areas, and fundamental to OS design, is concurrency. Concurrency encompasses a host of design issues, including communication among processes, sharing of and competing for resources (such as memory, files, and I/O access), synchronization of the activities of multiple processes, and allocation of processor time to processes. We shall see that these issues arise not just in multiprocessing and distributed processing environments but even in single-processor multiprogramming systems.

Some Key Terms Related to Concurrency



atomic operation	A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes.
critical section	A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.
deadlock	A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
livelock	A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.
mutual exclusion	The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
race condition	A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.
starvation	A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

Process Interaction

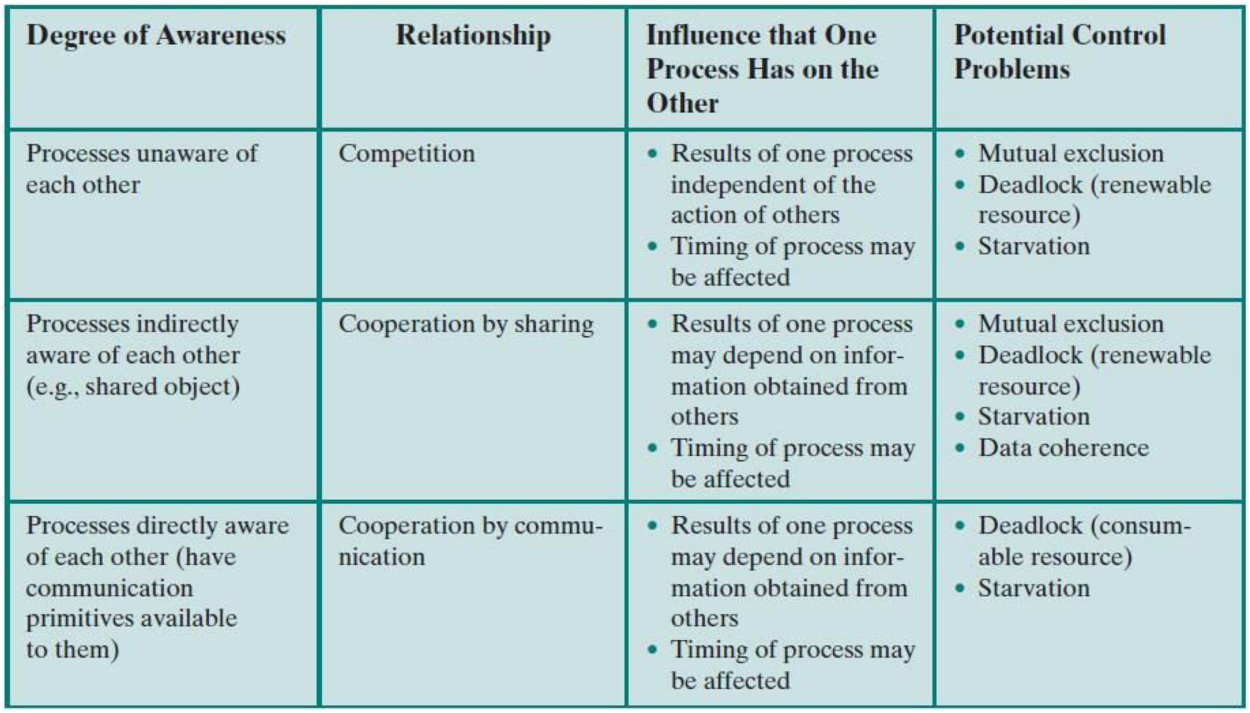
We can classify the ways in which processes interact on the basis of the degree to which they are aware of each other's existence. Table 5.2 lists three possible degrees of awareness plus the consequences of each:

• **Processes unaware of each other:** These are independent processes that are not intended to work together. The best example of this situation is the multiprogramming of multiple independent processes. These can either be batch jobs or interactive sessions or a mixture. Although the processes are not working together, the OS needs to be concerned about **competition** for resources. For example, two independent applications may both want to access the same disk or file or printer. The OS must regulate these accesses.

• **Processes indirectly aware of each other:** These are processes that are not necessarily aware of each other by their respective process IDs but that share access to some object, such as an I/O buffer. Such processes exhibit **cooperation** in sharing the common object.

• **Processes directly aware of each other:** These are processes that are able to communicate with each other by process ID and that are designed to work jointly on some activity. Again, such processes exhibit **cooperation.** Conditions will not always be as clear-cut as suggested in Table follows. Rather, several processes may exhibit aspects of both competition and cooperation. Nevertheless, it is productive to examine each of the three items in the preceding list separately and determine their implications for the OS.

Process Interaction



Degree of Awareness	Relationship	Influence that One Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	• Results of one process independent of the action of others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation • Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Deadlock (consumable resource) • Starvation

Race Condition

A race condition occurs when multiple processes or threads read and write data items so that the final result depends on the order of execution of instructions in the multiple processes. Let us consider two simple examples.

As an example, suppose that two processes, P1 and P2, share the global variable a. At some point in its execution, P1 updates a to the value 1, and at some point in its execution, P2 updates a to the value 2. Thus, the two tasks are in a race to write variable a. In this example, the “loser” of the race (the process that updates last) determines the final value of a.

For our second example, consider two processes, P3 and P4, that share global variables b and c, with initial values b = 1 and c = 2. At some point in its execution, P3 executes the assignment b = b + c, and at some point in its execution, P4 executes the assignment c = b + c. Note that the two processes update different variables. However, the final values of the two variables depend on the order in which the two processes execute these two assignments. If P3 executes its assignment statement first, then the final values are b = 3 and c = 5. If P4 executes its assignment statement first, then the final values are b = 4 and c = 3.

Problems of Concurrency – Mutual exclusion, Deadlock & Starvation

COMPETITION AMONG PROCESSES FOR RESOURCES: Concurrent processes come into conflict with each other when they are competing for the use of the same resource. In its pure form, we can describe the situation as follows. Two or more processes need to access a resource during the course of their execution. Each process is unaware of the existence of other processes, and each is to be unaffected by the execution of the other processes. It follows from this that each process should leave the state of any resource that it uses unaffected. Examples of resources include I/O devices, memory, processor time, and the clock. There is no exchange of information between the competing processes. However, the execution of one process may affect the behavior of competing processes. In particular, if two processes both wish access to a single resource, then one process will be allocated that resource by the OS, and the other will have to wait. Therefore, the process that is denied access will be slowed down. In an extreme case, the blocked process may never get access to the resource and hence will never terminate successfully.

In the case of competing processes three control problems must be faced. First is the need for **mutual exclusion.** Suppose two or more processes require access to a single non-sharable resource, such as a printer. During the course of execution, each process will be sending commands to the I/O device, receiving status information, sending data, and/or receiving data. We will refer to such a resource as a **critical resource,** and the portion of the program that uses it as a **critical section** of the program. It is important that only one program at a time be allowed in its critical section. We cannot simply rely on the OS to understand and enforce this restriction because the detailed requirements may not be obvious. In the case of the printer, for example, we want any individual process to have control of the printer while it prints an entire file. Otherwise, lines from competing processes will be interleaved.
The enforcement of mutual exclusion creates two additional control problems. One is that of **deadlock.** For example, consider two processes, P1 and P2, and two resources, R1 and R2. Suppose that each process needs access to both resources to perform part of its function. Then it is possible to have the following situation: the OS assigns R1 to P2, and R2 to P1. Each process is waiting for one of the two resources. Neither will release the resource that it already owns until it has acquired the other resource and performed the function requiring both resources. The two processes are deadlocked.

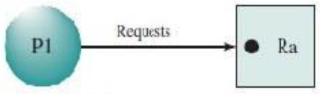
A final control problem is **starvation.** Suppose that three processes (P1, P2, P3) each require periodic access to resource R. Consider the situation in which P1 is in possession of the resource, and both P2 and P3 are delayed, waiting for that resource. When P1 exits its critical section, either P2 or P3 should be allowed access to R. Assume that the OS grants access to P3 and that P1 again requires access before P3 completes its critical section. If the OS grants access to P1 after P3 has finished, and subsequently alternately grants access to P1 and P3, then P2 may indefinitely be denied access to the resource, even though there is no deadlock situation.

Deadlock

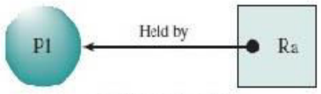
Deadlock can be defined as the *permanent* blocking of a set of processes that either compete for system resources or communicate with each other. A set of processes is deadlocked when each process in the set is blocked awaiting an event (typically the freeing up of some requested resource) that can only be triggered by another blocked process in the set. Deadlock is permanent because none of the events is ever triggered. Unlike other problems in concurrent process management, there is no efficient solution in the general case.

Resource Allocation Graph

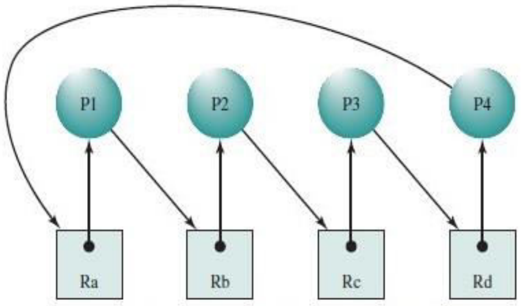
A useful tool in characterizing the allocation of resources to processes is the **resource allocation graph,** introduced by Holt [HOLT72]. The resource allocation graph is a directed graph that depicts a state of the system of resources and processes, with each process and each resource represented by a node. A graph edge directed from a process to a resource indicates a resource that has been requested by the process but not yet granted (Figure given below). Within a resource node, a dot is shown for each instance of that resource. Examples of resource types that may have multiple instances are I/O devices that are allocated by a resource management module in the OS. A graph edge directed from a reusable resource node dot to a process indicates a request that has been granted (Figure given below); that is, the process has been assigned one unit of that resource. A graph edge directed from a consumable resource node dot to a process indicates that the process is the producer of that resource. Figure C shows an example deadlock. There is only one unit each of resources Ra and Rb. Process P1 holds Rb and requests Ra, while P2 holds Ra but requests Rb. Figure d has the same topology as Figure c, but there is no deadlock because multiple units of each resource are available. The resource allocation graph of Figure e corresponds to the deadlock situation in Figure b. Note that in this case; we do not have a simple situation in which two processes each have one resource the other needs. Rather, in this case, there is a circular chain of processes and resources that results in deadlock.



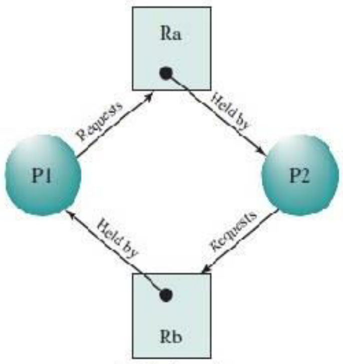
(a) Resource is requested



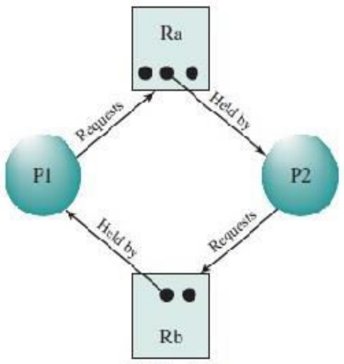
(b) Resource is held



E Resource Allocation Graph for Figure 6.1b



(c) Circular wait



(d) No deadlock

Necessary conditions for Deadlock

NECESSARY CONDITIONS

ALL of these four **must** happen simultaneously for a deadlock to occur:

Mutual exclusion

One or more than one resource must be held by a process in a non-sharable (exclusive) mode.

Hold and Wait

A process holds a resource while waiting for another resource.

No Preemption

There is only voluntary release of a resource - nobody else can make a process give up a resource.

Circular Wait

Process A waits for Process B waits for Process C .... waits for Process A.

No preemption:

a) Release any resource already being held if the process can't get an additional resource.
b) Allow preemption - if a needed resource is held by another process, which is also waiting on some resource, steal it. Otherwise wait.

Circular wait:

a) Number resources and only request in ascending order.
b) EACH of these prevention techniques may cause a decrease in utilization and/or resources. For this reason, prevention isn't necessarily the best technique.
c) Prevention is generally the easiest to implement.

Mutual exclusion:

a) Automatically holds for printers and other non-sharables.
b) Shared entities (read only files) don't need mutual exclusion (and aren't susceptible to deadlock.)
c) Prevention not possible, since some devices are intrinsically non-sharable.

Hold and wait:

a) Collect all resources before execution.
b) A particular resource can only be requested when no others are being held. A sequence of resources is always collected beginning with the same one.
c) Utilization is low, starvation possible.

Deadlock Prevention

Do not allow one of the four conditions those may invoke deadlock to prevent deadlock:

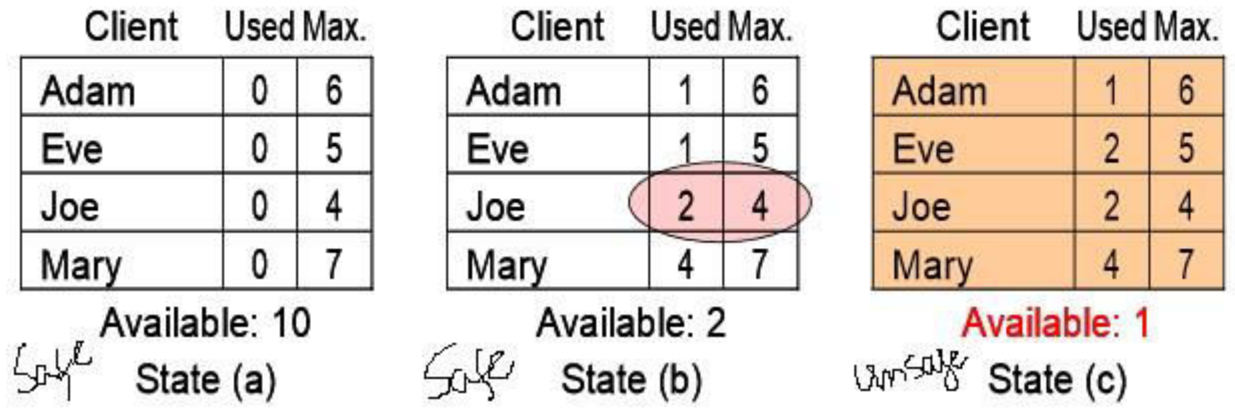
1. No preemption
2. Circular wait
3. Mutual Exclusion
4. Hold and wait

Banker's Algorithm – Deadlock Avoidance

Banker's behavior
(example of one resource type with many instances):

- Clients are asking for loans up-to an agreed limit
- The banker knows that not all clients need their limit simultaneously
- All clients must achieve their limits at some point of time but not necessarily simultaneously
- After fulfilling their needs, the clients will pay-back their loans

Example: The banker knows that all 4 clients need 22 units together, but he has only total 10 units



Client	Used	Max.
Adam	0	6
Eve	0	5
Joe	0	4
Mary	0	7

Available: 10

Safe State (a)

Client	Used	Max.
Adam	1	6
Eve	1	5
Joe	2	4
Mary	4	7

Available: 2

Safe State (b)

Client	Used	Max.
Adam	1	6
Eve	2	5
Joe	2	4
Mary	4	7

Available: 1

Unsafe State (c)

As per Banker's following points are to be maintained:

- Always keep so many resources that satisfy the needs of at least one client
- Multiple instances.
- Each process must a priori claim maximum use.
- When a process requests a resource it may have to wait.
- When a process gets all its resources it must return them in a finite amount of time.

Interprocess Communication - IPC

Mechanism for processes to communicate and to synchronize their actions:

Message system – processes communicate with each other without resorting to shared variables

IPC facility provides two operations:

- send(message) – message size fixed or variable
- receive(message)

If P and Q wish to communicate, they need to:

- establish a communication link between them
- exchange messages via send/receive

Implementation of communication link:

- physical (e.g., shared memory, hardware bus)
- logical (e.g., logical properties)

Examples of interprocess and interthread communication facilities include:

- Data transfer:
  - Pipes (named, dynamic – shell or process generated)
  - shared buffers or files
  - TCP/IP socket communication (named, dynamic - loop back interface or network interface)
  - D-Bus is an IPC mechanism offering one to many broadcast and subscription facilities between processes. Among other uses, it is used by graphical frameworks such as KDE and GNOME.
- Shared memory
  - Between Processes
  - Between Threads (global memory)
- Messages
  - OS provided message passing:

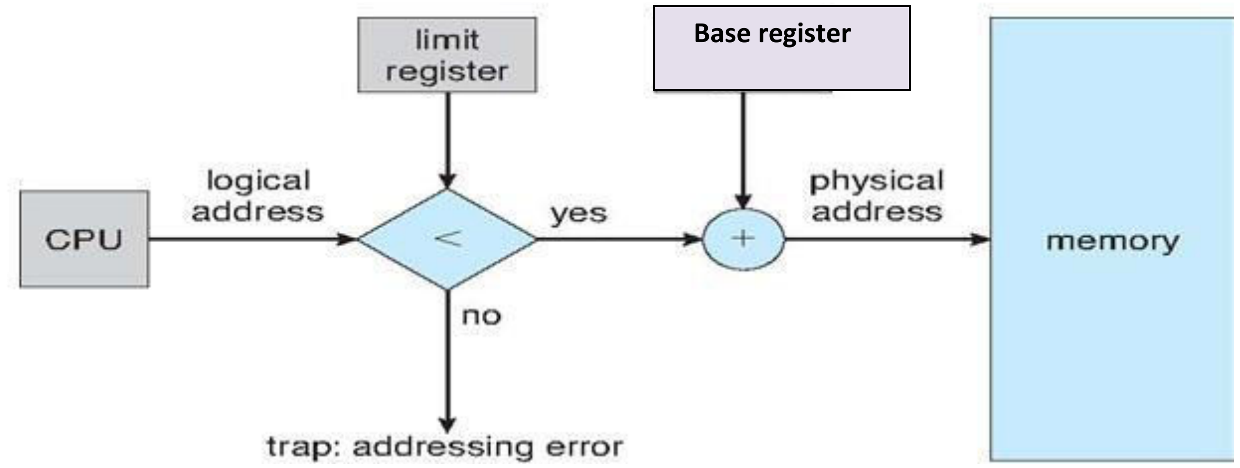
  send() / receive():

  - Signals
  - Via locks, semaphores or monitors

Base Register – Limit Register

What is the Base register and what is the Limit register?

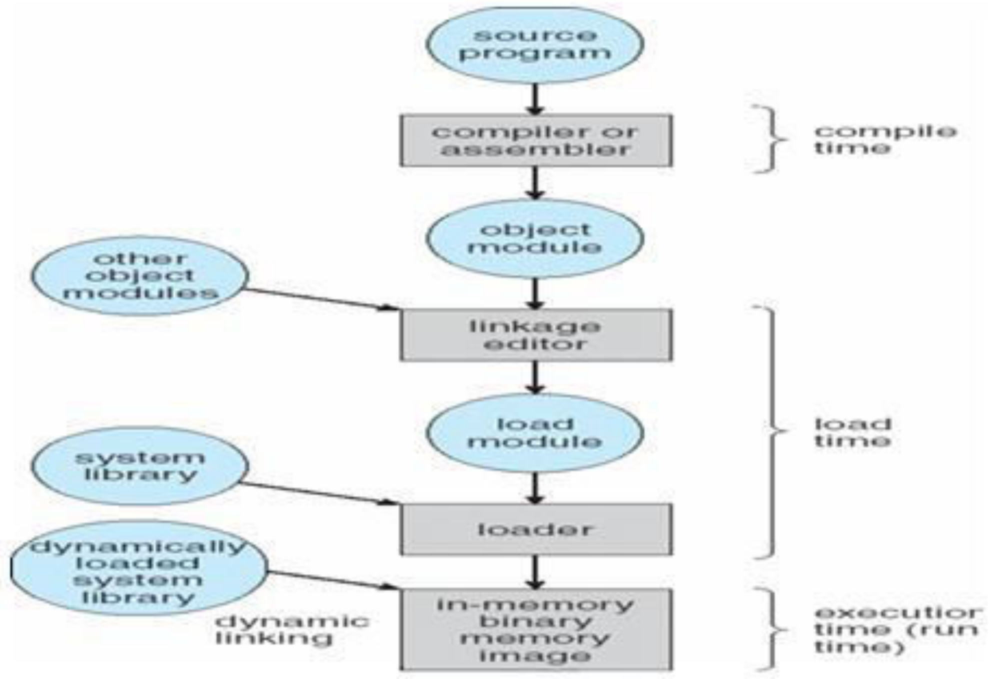
- **Base register:** Specifies the smallest legal physical memory address.
- **Limit register:** Specifies the size of the range.
- A pair of base and limit registers specifies the logical address space.
- The base and limit registers can be loaded only by the **operating system.**
- Ex: If the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).



Address Binding

Address binding can occur at three different stages:

- **Compile Time:** if you know at compile time where the process will reside in memory, then absolute code can be generated.
- **Load Time:** if it is not know at compile time where the process will reside in memory, then the compiler must generate relocatable code and the final binding is delayed until the load time.
- **Execution Time:** if the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

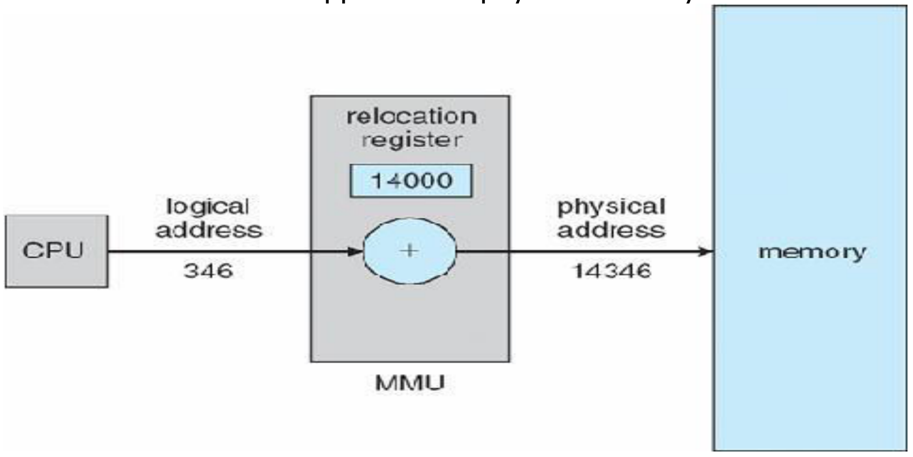


Logical Memory and Physical Memory

- **Logical Memory Address:** Addresses generated by the CPU.
- **Physical Memory Address:** Addresses seen by the memory unit.
- Logical and Physical addresses are the same compile-time and load-time address binding schemes, and differ in the execution-time address-binding scheme.

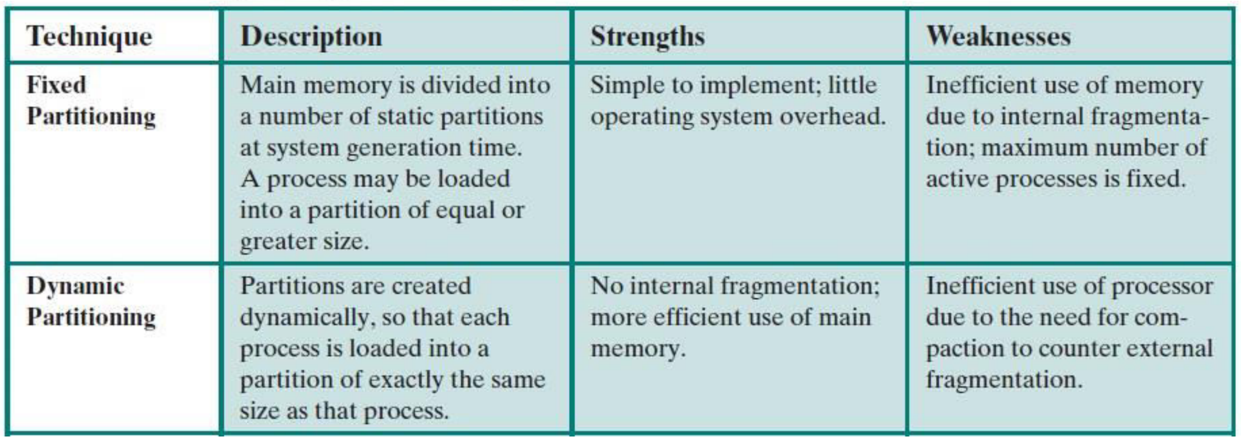
What is the Memory Management Unit (MMU)?

- **Memory Management Unit (MMU):** It is a hardware device that maps the logical address to physical address.
- The value in the MMU relocation register is added to every logical address generated by the CPU to be mapped into a physical memory address.



Memory Partitioning

Memory Management Techniques



Technique	Description	Strengths	Weaknesses
Fixed Partitioning	Main memory is divided into a number of static partitions at system generation time. A process may be loaded into a partition of equal or greater size.	Simple to implement; little operating system overhead.	Inefficient use of memory due to internal fragmentation; maximum number of active processes is fixed.
Dynamic Partitioning	Partitions are created dynamically, so that each process is loaded into a partition of exactly the same size as that process.	No internal fragmentation; more efficient use of main memory.	Inefficient use of processor due to the need for compaction to counter external fragmentation.

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

One possibility is to make use of equal-size partitions. In this case, any process whose size is less than or equal to the partition size can be loaded into any available partition. If all partitions are full and no process is in the Ready or Running state, the operating system can swap a process out of any of the partitions and load in another process, so that there is some work for the processor. There are two difficulties with the use of equal-size fixed partitions:
• A program may be too big to fit into a partition. In this case, the programmer must design the program with the use of overlays so that only a portion of the program need be in main memory at any one time. When a module is needed that is not present, the user's program must load that module into the program's partition, **overlaying** whatever programs or data are there.

Dynamic Partitioning
To overcome some of the difficulties with fixed partitioning, an approach known as dynamic partitioning was developed. Again, this approach has been supplanted by more sophisticated memory management techniques. An important operating system that used this technique was IBM's mainframe operating system, OS/MVT (Multiprogramming with a Variable Number of Tasks). With dynamic partitioning, the partitions are of variable length and number. When a process is brought into main memory, it is allocated exactly as much memory as it requires and no more.

Dynamic Partitioning strategies to allocate processes into memory

To satisfy a request of size n from a list of free holes, we use one of the following strategies:

- **First-Fit:** Allocate the process to the first big enough hole.

- **Best-Fit:** Allocate the process to the smallest hole that is big enough to accommodate the process. The must search the entire list unless ordered by size and produce the smallest leftover hole.

- **Worst-Fit:** Allocate the process to the largest hole. The must search the entire list unless ordered by size and produce the largest leftover hole.

Page Description Table

The OS obviously needs to keep track of both partitions and free memory. Once created, a partition defined by its base address and size. Those attributes remain essentially unchanged for as long as the related partition exists. In addition for the purpose of process switching and swapping, it is important to know which partition belongs to a given process. An enhanced version of Partition Description Table is required to keep track of partitions in the case of dynamic partitioning. To imply any dynamic allocation strategies the PDT data is very much required. For each and every allocation of process into memory PDT access is must. The structure of PDT can be designed as given below:

Partition SL No	Base (in K)	Size (in K)	Status
1	0	100	Allocated
2	100	120	Allocated
3	220	300	Not allocated
……			

Fragmentation

In computer memory/ storage, **fragmentation** is a phenomenon in which storage space is used inefficiently, reducing capacity or performance and often both. The exact consequences of fragmentation depend on the specific system of storage allocation in use and the particular form of fragmentation. In many cases, fragmentation leads to storage space being "wasted", and in that case the term also refers to the wasted space itself.

Fragmentation can be one of the following:

- **External Fragmentation:** Total memory space exists to satisfy a request but is not contiguous. It can be resulted from best-fit and first-fit in the case of dynamic memory allocation.

- **Internal Fragmentation:** Allocated memory may be slightly larger than the required memory resulted in a size difference that is memory internal to a partition but is not used in the case of fixed-sized partitioning.

The general approach to avoiding external fragmentation is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size.

How to reduce the external fragmentation?
To reduce the external fragmentation by:

- **Compaction**: Shuffle memory contents to place all free memory together in one large block. It is possible only if relocation is dynamic, and is done at execution time.
- Another possible solution: **paging and segmentation.**

Non contiguous Memory Allocation types

Non-Contiguous Memory Allocation:

1. Paging 2. Segments 3. Segments with paging.

Virtual Memory

A storage allocation scheme in which secondary memory can be addressed as though it were part of main memory. The addresses a program may use to reference memory are distinguished from the addresses the memory system uses to identify physical storage sites, and program-generated addresses are translated automatically to the corresponding machine addresses. The size of virtual storage is limited by the addressing scheme of the computer system and by the amount of secondary memory available and not by the actual number of main storage locations.

Virtual Memory involves separation of user logical memory from physical memory.

- Simulating more random access memory (RAM) than actually exists, allowing the computer to run larger programs and multiple programs concurrently.
- A common function in most every OS and hardware platform, virtual memory uses the hard disk to temporarily hold what was in real memory.

It can be implemented as:

- Demand paging.
- Demand segmentation.