Chapter 1 Types and structure of Operating System

Introduction and History of Operating System

- 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 programs.
- An operating system (OS) is a collection of software that manages computer hardware resources and provides common services for computer programs.
- The operating system is a vital component of the system software in a computer system.
- An operating system falls under the category of system software that performs all the fundamental tasks like file management, memory handling, process management, handling the input/output and governing and managing the peripheral devices like disk drives, networking hardware, printers, etc.

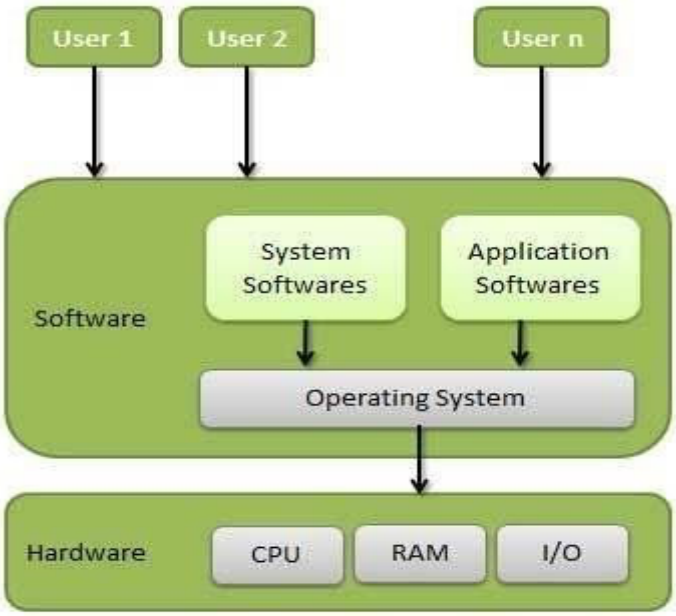


Fig: Operating System

Some examples of OS are Linux, Windows, OS X, Solaris, OS/400, Chrome OS, etc.

Objective/Goals of OS

An Operating System has a special program which controls the execution of application programs. OS acts as an intermediary among applications and the hardware components. OS can be thought of as having three objectives. These are:

- **Convenience**: It makes a computer more suitable to use.
- **Efficiency**: It provides the computer system resources with efficiency and in easy to use format.
- **Ability to develop**: It should be built in such a way that it permits the efficient development, testing and installation of new system functions without interfering with service.

Operating system as an extended machine

- Operating systems perform two basically unrelated functions: providing a clean abstract set of resources instead of the messy hardware to application programmers and managing these hardware resources.
- The architecture (instruction set, memory, I/O, and bus structure) of most computers at the machine level language is primitive and awkward to program, especially for input/output operations.
- Users do not want to be involved in programming of storage devices.
- Operating System provides a simple, high-level abstraction such that these devices contain a collection of named files.
- Such files consist of useful piece of information like a digital photo, e mail messages, or web page.
- Operating System provides a set of basic commands or instructions to perform various operations such as read, write, modify, save or close.
- Dealing with them is easier than directly dealing with hardware.
- Thus, Operating System hides the complexity of hardware and presents a beautiful interface to the users.
- Just as the operating system shields (protect from an unpleasant experience) the programmer from the disk hardware and presents a simple file-oriented interface, it also conceals a lot of unpleasant business concerning interrupts, timers, memory management, and other low-level features.
- In each case, the abstraction offered by the operating system is simpler and easier to use than that offered by the underlying hardware.
- In this view, the function of the operating system is to present the user with the equivalent of an **extended machine** or **virtual machine** that is easier to work with than the underlying hardware.

OS as a Resource Manager

- The concept of an operating system as providing abstractions to application programs is a top-down view. Alternatively, bottom-up view holds that the OS is there to manage all pieces of a complex system. Also, A computer consists of a set of resources such as processors, memories, timers, disks, printers and many others. The Operating System manages these resources and allocates them to specific programs.
- **As a resource manager**, the Operating system provides the controlled allocation of the processors, memories, I/O devices among various programs.
- Moreover, multiple user programs are running at the same time. The processor itself is a resource and the Operating System decides how much processor time should be given for the execution of a particular user program.
- The operating system also manages memory and I/O devices when multiple users are working.
- The primary task of OS is to keep the track of which programs are using which resources, to grant resource requests, to account for usage, and to resolve conflicting requests from different programs and users.
- Also, **Resource management** includes multiplexing (sharing) resources in two ways: in time and in space.

History of Operating Systems

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

When electronic computers were 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 plug boards to control the machine's basic functions. During these 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 18-bit 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 technology 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 was 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 smart phones, to ATM machines and motor vehicles. And as technology advances, so do operating systems.

Operating System Concepts and Functionalities

Operating System Concepts

Process

- A process is an instance of a program in execution.
- A program becomes process when executable file is loaded in the main memory.
- A process is defined as an entity which represents the basic unit of work to be implemented in the system.

Files

- 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.

System Calls

- In computing, a **system call** is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on.
- 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 call **provides** the services of the operating system to the user programs via Application Program Interface (API).
- It provides an interface between a process and operating system to allow user-level processes to request services of the operating system. System calls are the only entry points into the kernel system. All programs needing resources must use system calls.

1.2.3 The Shell

- The shell is the outermost layer of the operating system. Shells incorporate a programming language to control processes and files, as well as to start and control other programs. The shell manages the interaction between you and the operating system by prompting you for input, interpreting that input for the operating system, and then handling any resulting output from the operating system.
- Shells provide a way for you to communicate with the operating system. This communication is carried out either interactively (input from the keyboard is acted upon immediately) or as a shell script. A shell script is a sequence of shell and operating system commands that is stored in a file.

Function of Operating System

Following are some of important functions of an operating System:

- **Processor management** which involves putting the tasks into order and pairing them into manageable size before they go to the CPU.
- **Memory management** which coordinates data to and from RAM (random-access memory) and determines the necessity for virtual memory.
- **Device management** which provides interface between connected devices.
- **Storage management** which directs permanent data storage.
- **Application** which allows standard communication between software and your computer.
- **User interface** which allows you to communicate with your computer.
- It provides **file management** which refers to the way that the operating system manipulates stores, retrieves and saves data.
- **Error Handling** is done by the operating system. It takes preventive measures whenever required to avoid errors.
- **Security** − By means of password and similar other techniques, it prevents unauthorized access to programs and data.
- **Control over system performance** − Recording delays between request for a service and response from the system.
- **Coordination between other software and users** − Coordination and assignment of compilers, interpreters, assemblers and other software to the various users of the computer systems.

Operating System Structure

An operating system might have many structures. According to the structure of the operating system; operating systems can be classified into many categories.

Some of the main structures used in operating systems are:

Monolithic architecture of operating system

It is the oldest architecture used for developing operating system. Operating system resides on kernel for anyone to execute. System call is involved 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 is allowed and user mode for user program in which I/O devices and certain other instructions are not allowed. Two operating systems then examine the parameter of the call to determine which system call is to be carried out shown in event 2. Next, the operating system indexes 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.

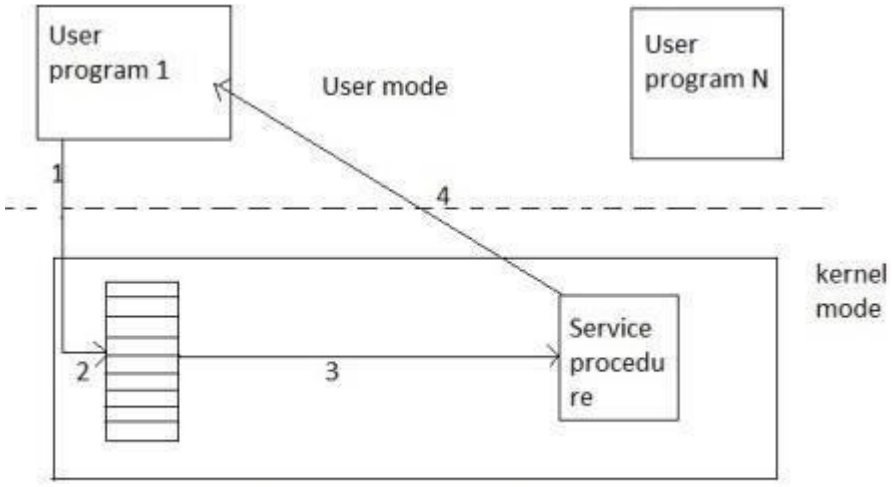


fig:- monolithic structure of os

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.

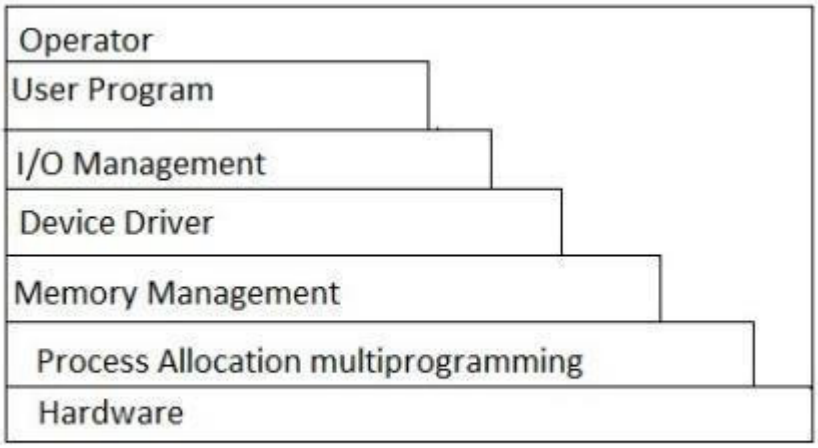


fig:- layered Architecture

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.

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 machines. 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.

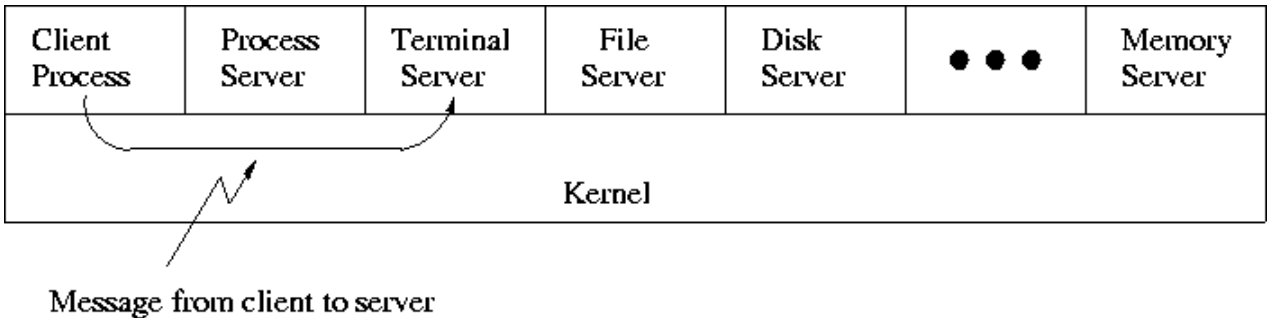
Its multiprogramming shares the resource of a single machine in different manner.

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.

Client/Server architecture of operating system

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



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 happens and occurs in client side that is a request were send and a reply come back.

Kernel mode and User mode

In any modern operating system, the CPU is actually spending time in two very distinct modes to make sure it works correctly:

Kernel Mode

In Kernel mode, the executing code has complete and unrestricted access to the underlying hardware. It can execute any CPU instruction and reference to any memory address. Kernel mode is generally reserved for the lowest-level, most trusted functions of the operating system. Crashes in kernel mode are catastrophic; they will halt the entire PC.

User Mode

In User mode, the executing code has no ability to directly access hardware or reference memory. Code running in user mode must delegate to system APIs to access hardware or memory. Due to the protection afforded by this sort of isolation, crashes in user modeare always recoverable. Most of the code running on your computer will execute in user mode.

Necessity of Dual Mode (User Mode and Kernel Mode) in Operating System

The lack of a dual mode i.e. user mode and kernel mode in an operating system can cause serious problems. Some of these are:

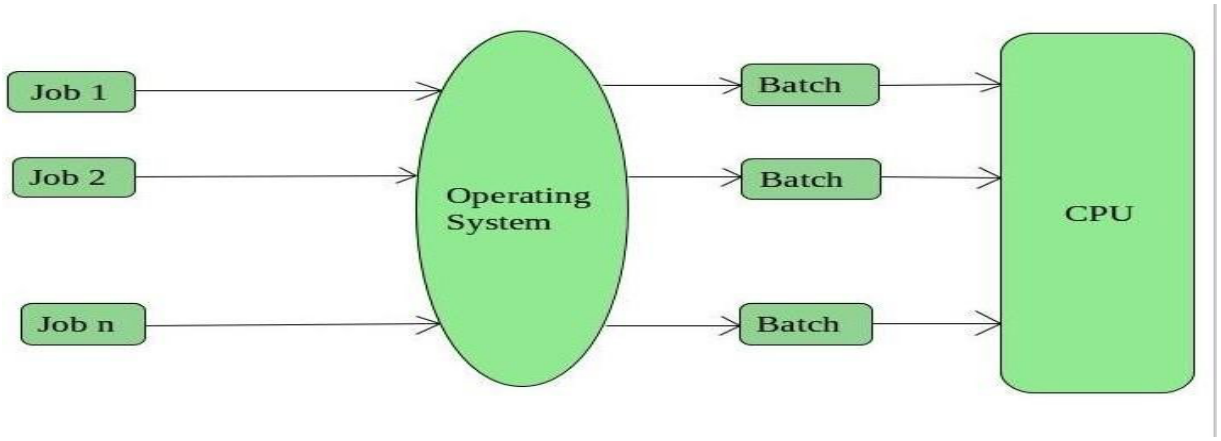
- A running user program can accidentally wipe out the operating system by overwriting it with user data.
- Multiple processes can write in the same system at the same time, with disastrous results.

Types and evolution of Operating System

Operating systems are there from the very first computer generation and they keep evolving with time. Some of the important types of operating systems which are most commonly used are:

i. Batch Operating System

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



Advantages of Batch Operating System:

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

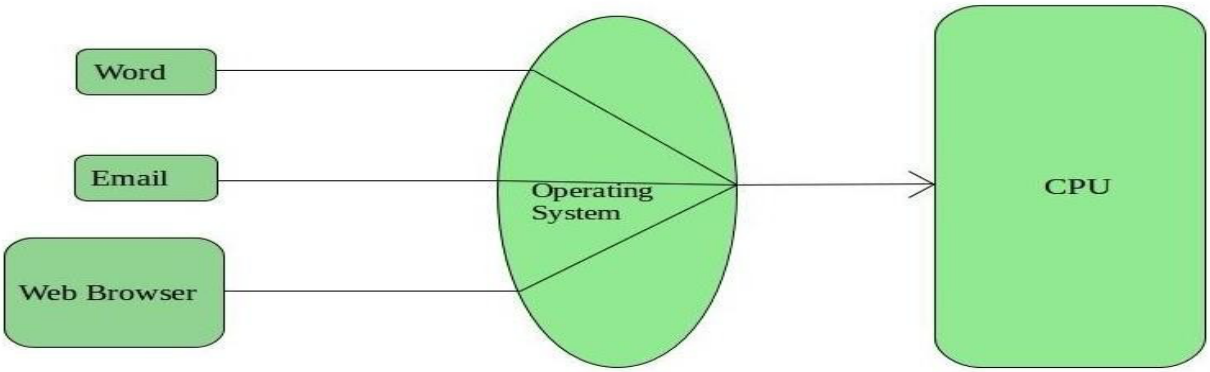
Disadvantages of Batch Operating System:

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

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

ii. Time-Sharing Operating Systems

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



Advantages of Time-Sharing OS:

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

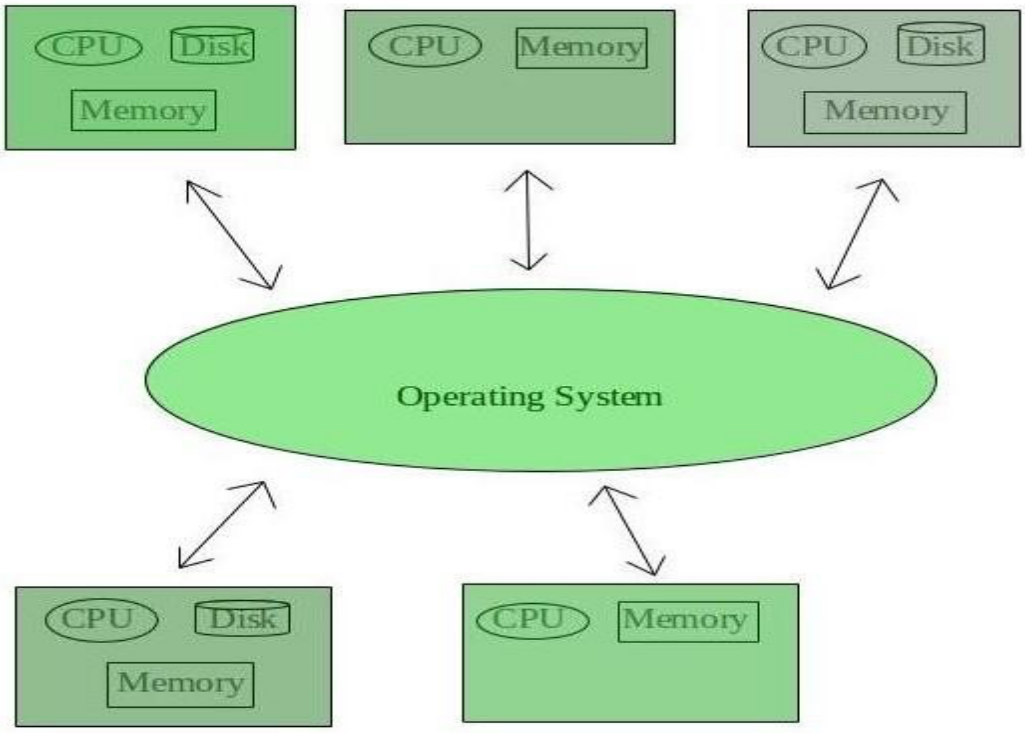
Disadvantages of Time-Sharing OS:

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

Examples of Time-Sharing OSs are: Multicast, Unix etc.

iii. Distributed Operating System

These types of operating system are a recent advancement in the world of computer technology and are being widely accepted all-over the world and, that too, with a great pace. Various autonomous interconnected computers communicate each other using a shared communication network. Independent systems possess their own memory unit and CPU. These are referred as **loosely coupled systems** or distributed systems. These systems processors differ in sizes and functions. The major benefit of working with these types of operating system is that it is always possible that one user can access the files or software which are not actually present on his system but on some other system connected within this network i.e., remote access is enabled within the devices connected in that network.



Advantages of Distributed Operating System:

- Failure of one will not affect the other network communication, as all systems are independent from each other
- Electronic mail increases the data exchange speed
- Since resources are being shared, computation is highly fast and durable
- Load on host computer reduces
- These systems are easily scalable as many systems can be easily added to the network
- Delay in data processing reduces

Disadvantages of Distributed Operating System:

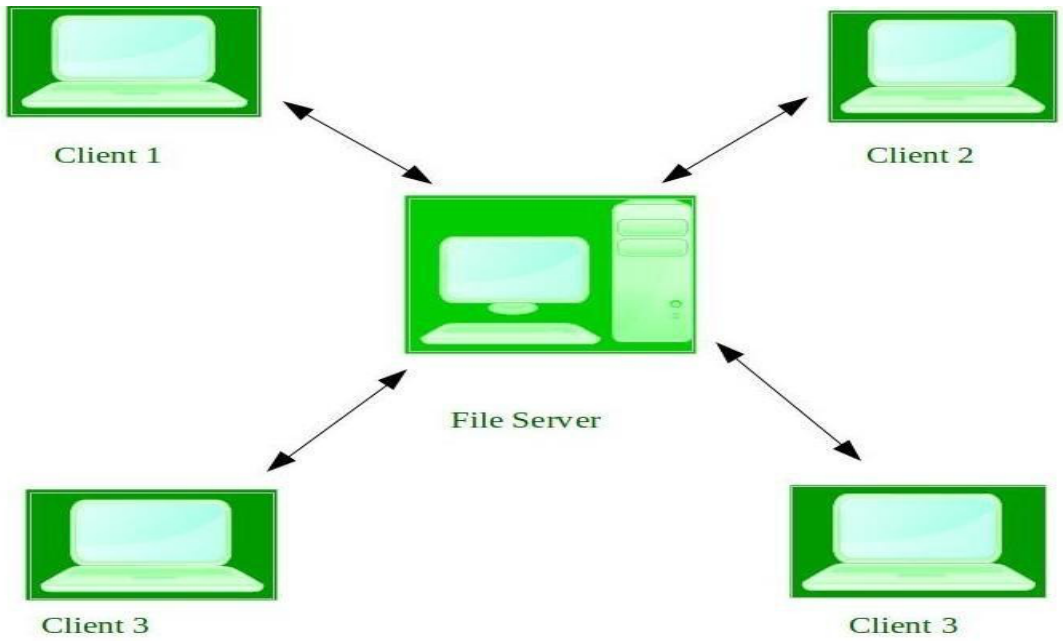
- Failure of the main network will stop the entire communication
- To establish distributed systems the language which are used are not well defined yet
- These types of systems are not readily available as they are very expensive. Not only that the underlying software is highly complex and not understood well yet.

Examples of Distributed Operating System are- LOCUS, AMOEBA etc.

iv. Network Operating System

This system run on a server and provides the capability to manage data, users, groups, security, applications, and other networking functions. These types of operating systems allows shared access of files, printers, security, applications, and other networking functions over a small private

network. One more important aspect of Network Operating Systems is that all the users are well aware of the underlying configuration, of all other users within the network, their individual connections etc. and that's why these computers are popularly known as **tightly coupled systems**.



Advantages of Network Operating System:

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

Disadvantages of Network Operating System:

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

Examples: Microsoft Windows Server 2003, Microsoft Windows Server 2008, UNIX, Linux, Mac OS X, Novell NetWare, and BSD etc.

v. Parallel Operating System

- Parallel operating systems are used to interface multiple networked computers to complete tasks in parallel.
- The architecture of the software is often a UNIX-based platform, which allows it to coordinate distributed loads between multiple computers in a network.
- Parallel operating systems are able to use software to manage all of the different resources of the computers running in parallel, such as memory, caches, storage space, and processing power.
- Parallel operating systems also allow a user to directly interface with all of the computers in the network

- A parallel operating system works by dividing sets of calculations into smaller parts and distributing them between the machines on a network. To facilitate communication between the processor cores and memory arrays, routing software has to either share its memory by assigning the same address space to all of the networked computers, or distribute its memory by assigning a different address space to each processing core.

vi. 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.

Advantages of Real Time Operating System:

- Maximum Consumption
- Task Shifting
- Error Free
- 24-7 systems

Disadvantages of Real Time Operating System:

- Limited Tasks
- Use heavy system resource
- Complex Algorithms
- Expensive

There are two types of real-time operating systems.

I. 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.

II. 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.

Multitasking, multiprogramming and multiprocessing

Multitasking

Both the memory and CPU time is shared among the tasks It performs multiple tasks concurrently. It gives a illusion that all the processes or tasks are performed simultaneously. For example, in your desktop you can listen to music, download things simultaneously. The concept is that the CPU time is shared equally among the processes or tasks. Again, take an example that 3 tasks are loaded intothe main memory. Consider that the time allocated for each program is 5 micro seconds. Now the first task is executed. After 5 microseconds irrespective whether the task gets completed or not it gets switched to the second task. Similarly, after 5 seconds it goes to the third task. Thus, the CPU time is shared equally among the tasks.

Multiprogramming

The concept of multiprogramming is that more than one program that is to be executed by the processor is loaded into the memory.

Say we have 2 programs loaded into the memory. The first program that is loaded is getting executed. At one point of time, it requires input from the user or waiting for some data. During the waiting time, the CPU is idle. Instead of wasting the time, the CPU will now begin to execute the second program. Meanwhile the first program once it receives the required data, will again get the CPU time and get executed blocking or pausing the execution of the second program. After the completion of the first program, the second program is executed from where it was paused. The concept was introduced to maximize the CPU usage.

Multiprocessing

The term multiprocessing is introduced around the modern times when they started to use more than one processor in a single computer (Remember the terms like dual core, quad core, octa core processor). These processors share some things in common like memory, peripherals etc. By sharing the memory and peripherals they are able execute different task simultaneously.

In general,
Multiprogramming: takes place in a system where it has a single processor.
Multitasking: single processor or sometimes multiprocessor
Multiprocessing: multiprocessor.

Chapter 2 Processes and Threads

Process Concepts

Program

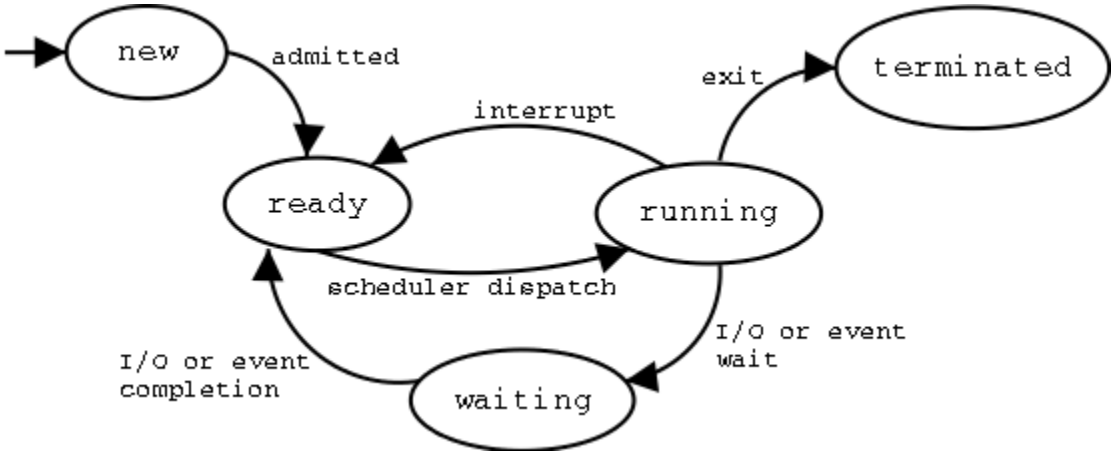
A program is a piece of code which may be a single line or millions of lines. A computer program is usually written by a computer programmer in a programming language. A computer program is a collection of instructions that performs a specific task when executed by a computer. When we compare a program with a process, we can conclude that a process is a dynamic instance of a computer program.

Process

A process is an instance of a program in execution. A program becomes process when executable file is loaded in the main memory. A process is defined as an entity which represents the basic unit of work to be implemented in the system. Each process has its own address space and process control block (PCB).

Process states and its transition

When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardized. In general, a process can have one of the following five states at a time.



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

- **Running:** Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.
- **Waiting:** Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.
- **Terminated or Exit:** Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory.

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). The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates. A PCB keeps all the information needed to keep track of a process as listed in the figure:

- **Process ID**: Unique identification for each of the process in the operating system.
- **Process State**: The current state of the process i.e., whether it is ready, running, waiting, or whatever.
- **Pointer**: A pointer to parent process.
- **Priority:** Contains the priority numbers of each process.
- **Program Counter**: Program Counter is a pointer to the address of the next instruction to be executed for this process.

- **CPU registers**: Various CPU registers where process need to be stored for execution for running state.
- **IO status information**: This includes a list of I/O devices allocated to the process.
- **Accounting information**: This includes the amount of CPU used for process execution, time limits, execution ID etc
- **CPU Scheduling Information**: Process priority and other scheduling information which is required to schedule the process.
- **Memory management information**: This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.

Operation of process

The operations of process carried out by an operating system are primarily of two types:

i. Process Creation

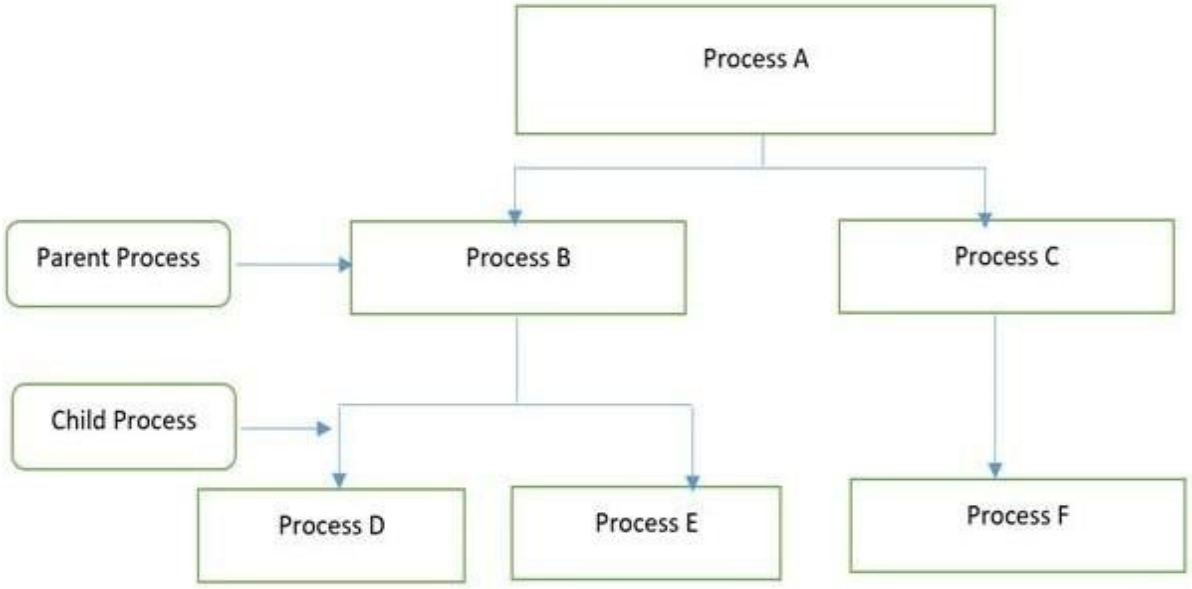
Process creation is a task of creating new processes. There are different situations in which a new process is created. There are different ways to create new process. A new process can be created at the time of initialization of operating system or when system calls such as **fork ()** are initiated by other processes. The process, which creates a new process using system calls, is called parent process while the new process that is created is called child process. The child processes can create new processes using system calls. A new process can also create by an operating system based on the request received from the user.

The process creation is very common in running computer system because corresponding to every task that is performed there is a process associated with it. For instance, a new process is created every time a user logs on to a computer system, an application program such a MS Word is initiated, or when a document printed.

ii. Process termination

Process termination is an operation in which a process is terminated after the execution of its last instruction. This operation is used to terminate or end any process. When a process is terminated, the resources that were being utilized by the process are released by the operating system. When a child process terminates, it sends the status information back to the parent process before terminating. The child process can also be terminated by the parent process if the task performed by the child process is no longer needed. In addition, when a parent process terminates, it has to terminate the child process as well became a child process cannot run when its parent process has been terminated.

The termination of a process when all its instruction has been executed successfully is called normal termination. However, there are instances when a process terminates due to some error. This termination is called as abnormal termination of a process.



The above figure shows the hierarchical structure of processes.

Concurrent processing

Concurrent processing is a computing model in which multiple processors execute instructions simultaneously for better performance. Tasks are broken down into subtasks that are then assigned to separate processors to perform simultaneously, instead of sequentially as they would have to be carried out by a single processor. Concurrent processing is sometimes said to be synonymous with parallel processing.

Parallel Processing

Parallel processing, one form of multiprocessing, is a situation in which two or more processors operate in unison. Two or more CPUs are executing instructions simultaneously. The Processor Manager has to coordinate the activity of each processor as well as synchronize cooperative interaction among the CPUs.
There are two primary benefits to parallel processing systems:

- **Increased reliability**
  - ✓ The availability of more than one CPU
  - ✓ If one processor fails, then the others can continue to operate and absorb the load.
- **Faster processing**

The increased processing speed is often achieved because sometimes Instructions can be processed in parallel, two or more at a time in one of several ways:

- ✓ Some systems allocate a CPU to each program or job
- ✓ Others allocate a CPU to each working set or parts of it
- ✓ Others subdivide individual instructions so that each subdivision can be processed simultaneously.

Race Condition

A situation where several processes/threads access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition. A race condition occurs when two or more threads can access shared data and they try to change it at the same time.

Possibilities of race:

- many concurrent process **read the same data**.
- one process **reading** and another process **writing** same data.
- two or more process **writing same data**.

Example 1

- P1 and P2 share global variables b and c with initial values b=1 and c=2
- P1 has an assignment operation, b=b+c
- P2 has an assignment operation c=b+c
- If P1 an P2 are running concurrently, value of b and c would be 3 and 5 if P1 executes its assignment first or it could be 4 and 3 if P2 executes its assignment first.

Example 2

Single process	output
Read(X)	**X=5**
X= X + 5	**X=10**

Case 1:

Process 1	Process 2	X
Read(x)		5
X=X+5		10
	Read(X)	10
	X=X+5	15

Case 2:

Process 1	Process 2	X
	Read(x)	5
	X=X+5	10
Read(x)		10
X=X+5		15

Case 3:

Process 1	Process 2	X
Read(x)		5
	Read(x)	5
X=X+5		10
	X=X+5	10

To guard against the race conditions, we need to synchronize the processes in such a way that only one process at a time can be manipulating shared variable or file. This is called mutual exclusion (i.e. critical section is executed as an atomic instruction).

Inter-process Communication

IPC is the mechanism for processes to communicate and to synchronize their actions or to allow processes to exchange information.

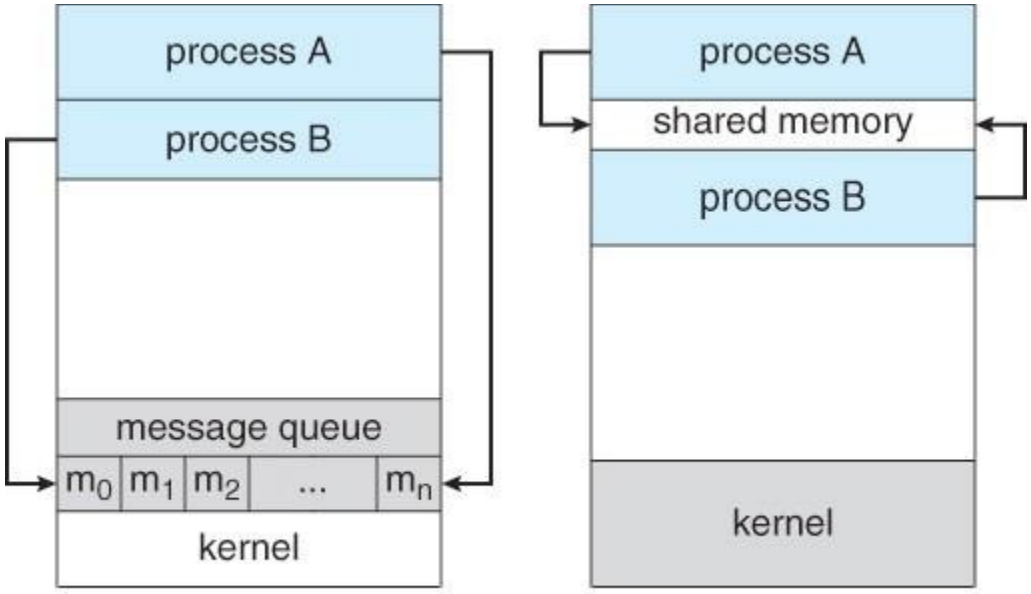
There are numerous reasons for providing an environment or situation which allows process co-operation:

- **Information sharing**: Since some users may be interested in the same piece of information (for example, a shared file), you must provide a situation for allowing concurrent access to that information.
- **Computation speedup**: If you want a particular work to run fast, you must break it into sub-tasks where each of them will get executed in parallel with the other tasks. Note that such a speed-up can be attained only when the computer has compound or various processing elements like CPUs or I/O channels.
- **Modularity**: You may want to build the system in a modular way by dividing the system functions into split processes or threads.
- **Convenience**: Even a single user may work on many tasks at a time. For example, a user may be editing, formatting, printing, and compiling in parallel.

Working together with multiple processes, require an inter-process communication (IPC) method which will allow them to exchange data along with various information. There are two primary models of inter-process communication:

i. shared memory and
ii. message passing.

In the shared-memory model, a region of memory which is shared by cooperating processes gets established. Processes can be then able to exchange information by reading and writing all the data to the shared region. In the message-passing form, communication takes place by way of messages exchanged among the cooperating processes.
The two communications models are contrasted in the figure below:



Message passing is mechanism for processes to communicate and to synchronize their actions without resorting to a shared variable.

This method of Inter Process Communication (IPC) uses two primitives:

- send (destination, message);
- receive (source, message);

- Send calls sends a message to a given destination.
- *Receive* receives a message from a given source. If no message is available, receiver can block until one arrives, or it can return immediately with error code.

If processes P and Q needs to communicate, they need to

1. establish a ***communication link*** *(physical or logical)* between them

2. exchange message via send/receive

Processes must name each other explicitly:

1. send(P, message) : send a message to process P
2. receive(Q, message) : receive a message from process Q

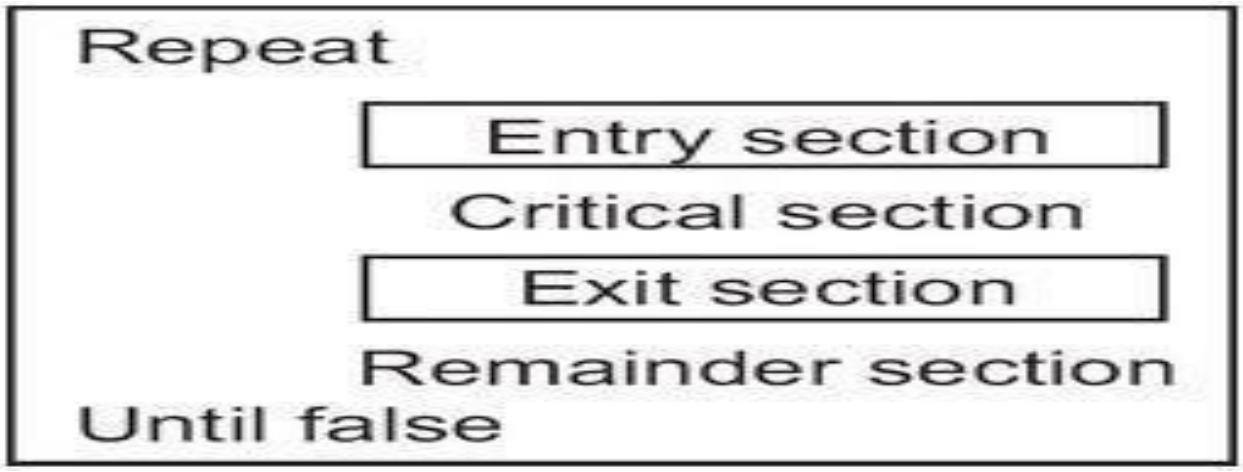
Properties of direct communication link

1. A link is associated with exactly one pair of communicating processes
2. Between each pair there exists exactly one link
3. The link may be unidirectional, but is usually bidirectional

<u>Critical Section/region</u>

- Critical section is the area of the code/instruction that should be executed atomically.
- Critical section is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution.

- If a process wants to access any kind of common shareable variable, common shareable table, common shareable files in its then the process is trying to enter a critical section.



Requirements for Critical Section

1. **Mutual Exclusion**

At a time only one process can enter/ access to the critical section.

2. **Progress**

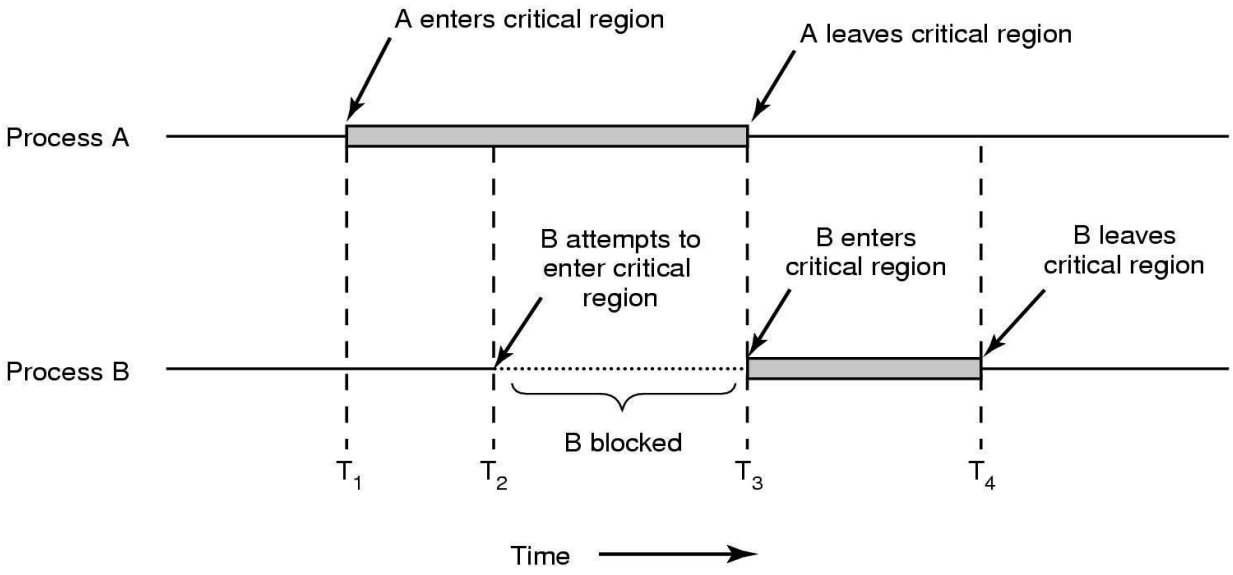
If CS is vacant/free and some process is tried to enter CS, no **infinite time** should be taken for the selection of process to go into the CS. Processes on the remainder section also get chance to enter CS.

3. **Bounded Waiting**

Let a particular process request to enter a CS, by OS allowing other process to enter CS. In such case there must be upper bound that the process gets denied to enter CS.

Mutual Exclusion

- A mutual exclusion (mutex) is a program object that prevents simultaneous access to a shared resource.
- Mutual exclusion is a property of concurrency control, which is instituted for the purpose of preventing race conditions.
- It is the requirement that one thread of execution never enter its critical section at the same time that another concurrent thread of execution enters its own critical section



Mutual Exclusion can be achieved using following techniques:

i. **Mutual Exclusion with busy waiting**

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

Process 1

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Process 2

- In this proposed solution, the integer variable 'turn' keeps track of whose turn is to enter the critical section.
- Initially turn = 0. Process 1 inspects turn, finds it to be 0 and enters in its critical section. Process 2 also finds it to be 0 and therefore 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**.
- When process 1 exit from the critical region is set to turn to 1 and now process 2 can find it to be 1 and enters into the critical region.
- Also, in this way both the process gets the alternate turn to enter in the critical region.
- Taking turns is not a good idea when one of the processes is much slower than the other.
- Can used only when the waiting period is expected to be short.

ii. Semaphore

- Semaphore is a protected integer value that can facilitate the processes to enter to a critical section.
- It ensures only one process enters to a CS at a time in a mutually exclusive way.
- Perform **atomic operation** i.e. no partial execution

Operations

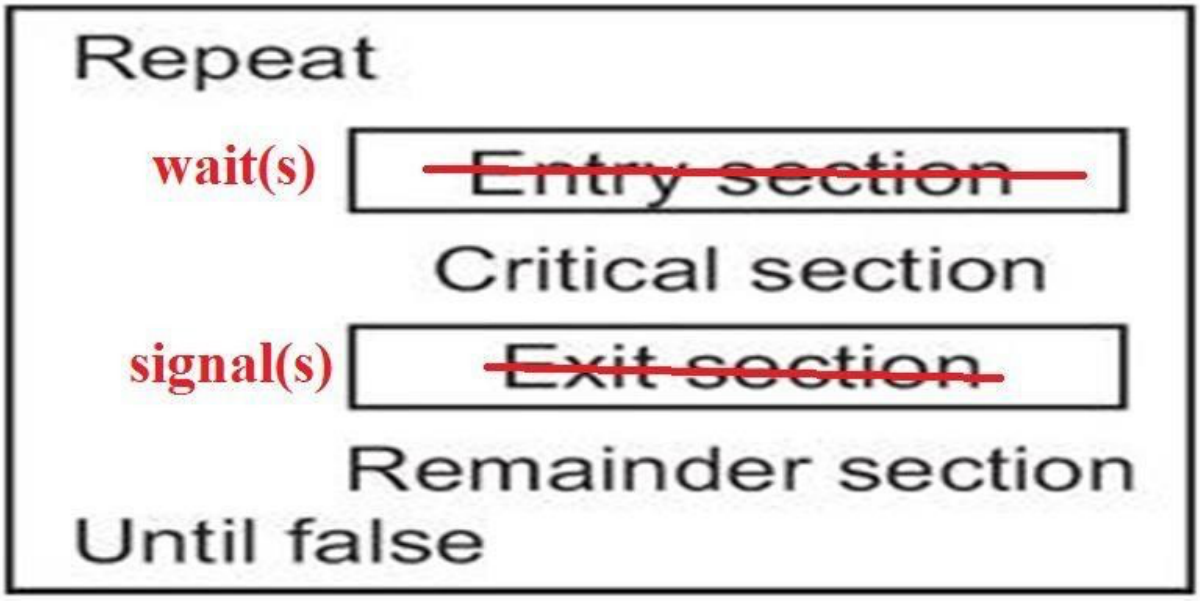
i. wait(s)

while (s = = 0); do no operation

s=s-1

ii. signal(s)

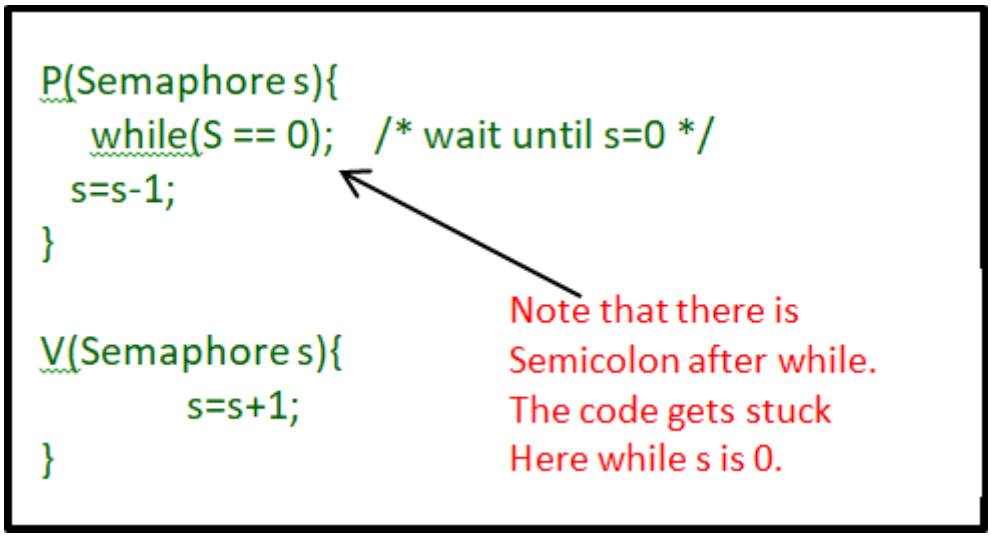
s=s+1



Semaphore is a simply a variable. This variable is used to solve critical section problem and to achieve process synchronization in the multi-processing environment.

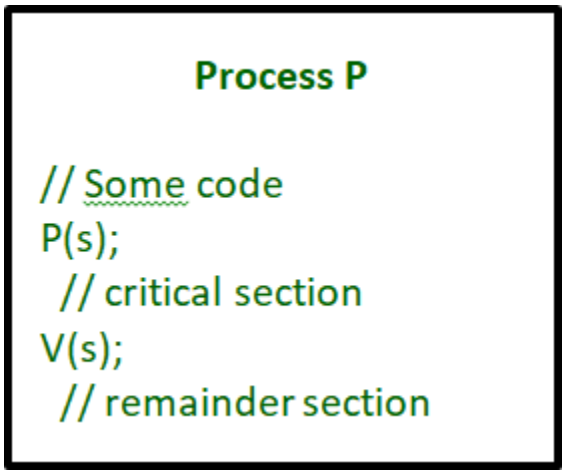
The two most common kinds of semaphores are counting semaphores and binary semaphores. Counting semaphore can take non-negative integer values and Binary semaphore can take the value 0 & 1. only.

Now let us see how it do so. First look at two operations which can be used to access and change the value of semaphore variable.



Some point regarding P and V operation

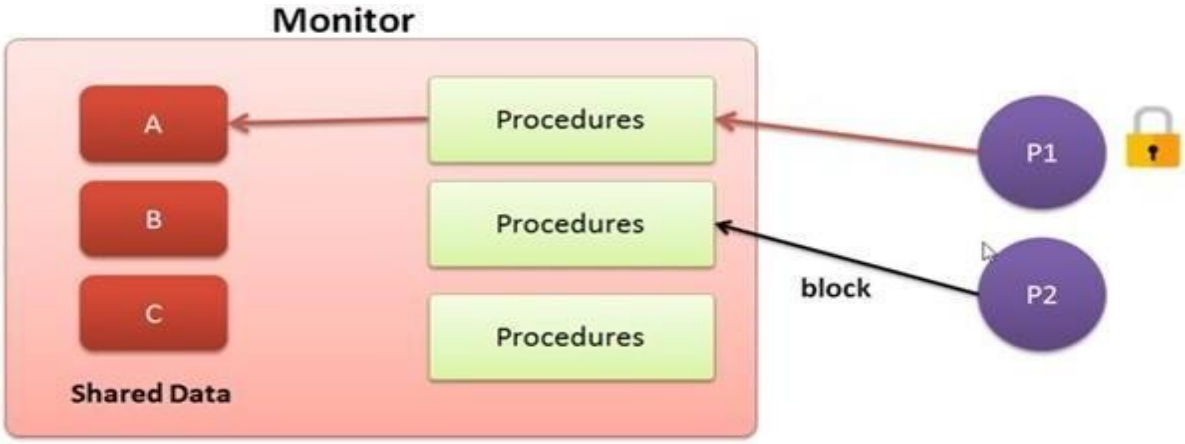
1. P operation is also called wait, sleep or down operation and V operation is also called signal, wake-up or up operation.
2. Both operations are atomic and semaphore(s) is always initialized to one.
3. A critical section is surrounded by both operations to implement process synchronization. See below image. Critical section of Process P is in between P and V operation.

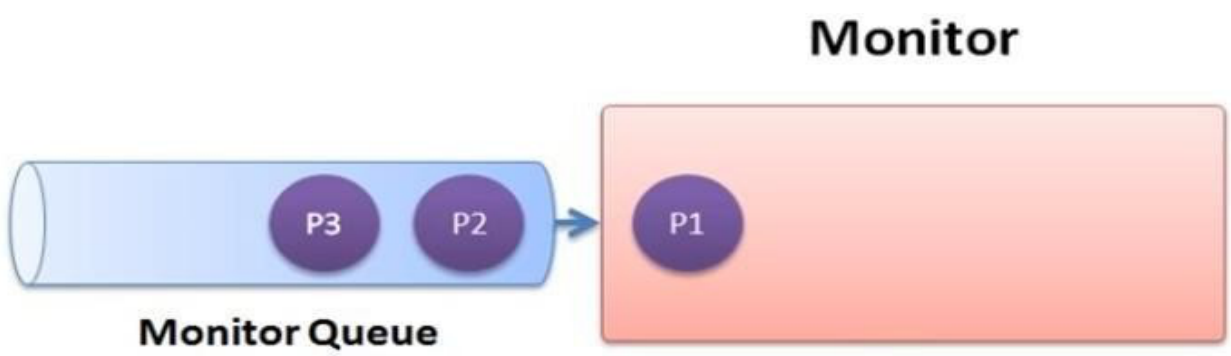


iii. Monitor

- A monitor is a collection of procedures, variables and data structures that are grouped together in a special kind of module or package
- Once the process enters in monitor, the process gets locked and remaining process are kept in queue
- Monitor is a type, or abstract data type, encapsulates private data with public methods to operate on that data

- Processes may call the procedures in a monitor whenever they want to but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor





iv. Peterson's Algorithm

Peterson's algorithm is used for mutual exclusion and allows two processes to share a single-use resource without conflict. It uses only shared memory for communication. Peterson's formula originally worked only with two processes, but has since been generalized for more than two.

In this algorithm turn variables (turn) and status flags (flag) are conditions or variables that are used in Peterson's algorithm. Because of these two conditions, and because of waiting for a turn only if other flags are set, the need to clear and reset flags is avoided. After a flag is set, the turn is immediately given away when using Peterson's algorithm.

Mutual exclusion, no progress and bounded waiting are three essential criteria used to solve the critical section problem when using the algorithm.

Algorithm

```
flag[0]=true;
turn=1;
while(flag[1]==true&&turn==1);
// Critical Section
flag[0]=false
```

v. Dekker's Algorithm

Dekker's algorithm will allow only a single process to use a resource if two processes are trying to use it at the same time. The highlight of the algorithm is how it solves this problem. It succeeds in preventing the conflict by enforcing mutual exclusion, meaning that only one process may use the resource at a time and will wait if another process is using it. This is achieved with the use of two "flags" and a "token". The flags indicate whether a process wants to enter the critical section (CS) or not; a value of 1 means TRUE that the process wants to enter the CS, while 0, or FALSE, means the opposite. The token, which can also have a value of 1 or 0, indicates priority when both processes have their flags set to TRUE.

This algorithm can successfully enforce mutual exclusion but will constantly test whether the critical section is available and therefore wastes significant processor time. It creates the problem known as lockstep synchronization, in which each thread may only execute in strict synchronization. It is also non-expandable as it only supports a maximum of two processes for mutual exclusion.

Algorithm

```
flag[0]=true;
while(flag[1]==true)
{
   if(turn!=0)
   {
      flag[0]=false;
      while(turn!=0);
      flag[0]=true;
   }
}
// Critical Section
turn=1;
flag[0]=false;
```

vi. Strict Alternation

- In this approach, we have an integer variable named turn, which keeps track of whose turn is it to enter critical section
- Say, in a two process scenario, Process 0 runs when turn=0 and process 1 runs when turn=1

```
while(TRUE){
        while(turn!=0);
        critical_region();
        turn=1;
        noncritical_region();
        }
```

Fig: Process 0

```
while(TRUE){
        while(turn!=1);
        critical_region();
        turn=0;
        noncritical_region();
        }
```

Fig: Process 1

- Turn is initially set to 0
- Process 0 examines turn to be 0 and enters the critical region.

- Process 1 also finds turn to be 0 so it sits in a tight loop continually testing turn to see when it becomes 1. Continuously testing a variable until some value appears is called busy waiting
- When process 0 leaves the critical region, it setsturn to 1 so that process 1 can now enter the critical region.

Problem

If process 0 finishes its work in non-critical region fast enough after it set, turn to 1 and process 0 needs to enter critical section again before process 1 then it cannot do so. So, process 1 which is running outside critical region is blocking process 0.

vii. Lock Variable

- In this approach, we will have a single shared variable called lock
- If lock=0, it means that no process is in the critical region while lock=1 means that a process is currently working in the critical region
- When a process wants to enter into its critical section, it first tests the lock
- If the lockis 0, the process sets it to 1 and enters the critical section
- If the lock is already 1, the process just waits until it becomes zero

What is the problem with this approach??

- two process may be in critical region at the same time
- a process sees lock =0 and enters critical region, but before it can set lock to 1, another process is scheduled, it sees lock=0 and enters critical region. Both process enter critical section, both assigning 1 to lock

viii. TSL (Test and Lock)

- This proposal to solve critical section problem requires a little help from hardware, it uses the following instruction

  TSL Rx, LOCK
- It reads the content of memory word lock into register RX and stores a non-zero value at the memory address lock
- This is an atomic instruction, that is operation of reading a word and storing into it are guaranteed to be indivisible
- When lock is 0, any process may set it to 1 through TSL instruction and read or write the shared memory
- When it is done, the process sets lock back to 0 using an ordinary MOVE instruction.

```
enter_region:
        TSL REGISTER, LOCK
        CMP REGISTER, #0
        JNE enter_region
        RET
Leave_region:
MOVE LOCK,#0
RET
```

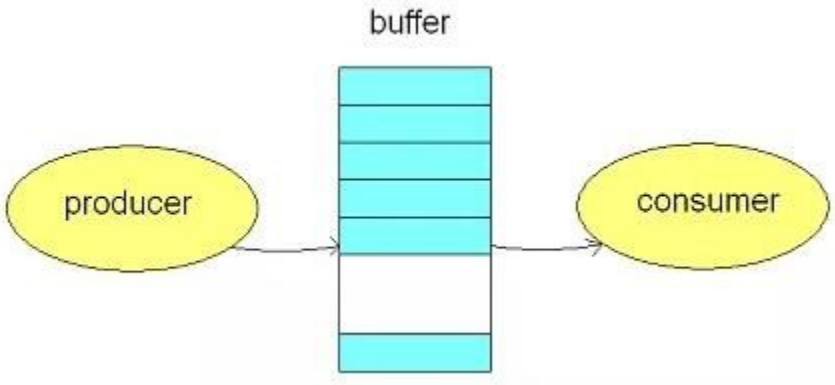
Producer-consumer problem

Producer-consumer problem (also known as the bounded-buffer problem) is a multiprocessing synchronization problem.

The problem describes two processes, the producer and the consumer, who share a common, fixed size buffer.

Producer: - The producer's job is to generate a piece of data, put it into the buffer and start again.

Consumer: - The consumer is consuming the data (i.e., removing it from the buffer) one piece at a time.



Problem

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

Solution to Producer-consumer problem

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

Sleep and wakeup

- When a process wants to enter its critical region, it checks to see if entry is allowed, if it is not, the process just sits in a tight loop waiting, this approach wastes CPU time.
- We could use an IPC primitives that blocks a process instead of wasting CPU time when they are not allowed to enter into their critical section.
- Such a simple primitive pair is **sleep and wakeup**
- Essentially, when a process is not permitted to access its critical section, it uses a system call known as Sleep, which causes that process to block.
- The process will not be scheduled to run again, until another process uses the Wakeup system call.
- Wakeup is called by a process when it leaves its critical section if any other processes have blocked.

```
BufferSize = 3;
 count = 0;
 Producer()
{
 int item;
 WHILE (true){
 make_new(item);                    // create a new item to put in the buffer
 IF(count==BufferSize) Sleep();     // if the buffer is full, sleep
 put_item(item);                    // put the item in the buffer
count = count + 1;                  // increment count of items
IF (count==1)
Wakeup(Consumer);                   // if the buffer was previously empty, wake the
consumer
} }
Consumer()
{
 Int item;
 WHILE(true)
 {
 IF(count==0) Sleep();              // if the buffer is empty, sleep
remove_item(item);                  // take an item from the buffer
count = count - 1;                   // decrement count of items
IF(count==N-1)
Wakeup(Producer);                    // if buffer was previously full, wake the producer
Consume_item(item);                 // consume the item
}}
```

Problem with the above Solution

The problem with this solution is that it contains a race condition that can lead into a deadlock. Consider the following scenario:

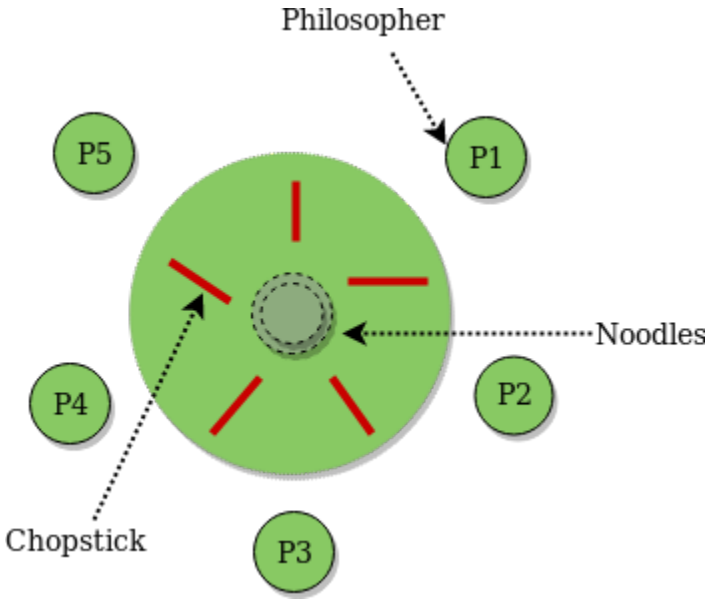
- ✓ The consumer has just read the variable itemCount, noticed it's zero and is just about to move inside the if-block.
- ✓ Just before calling sleep, the consumer is interrupted and the producer is resumed.
- ✓ The producer creates an item, puts it into the buffer, and increases itemCount.
- ✓ Because the buffer was empty prior to the last addition, the producer tries to wake up the consumer.
- ✓ Unfortunately, the consumer wasn't yet sleeping, and the wakeup call is lost. When the consumer resumes, it goes to sleep and will never be awakened again. This is because the consumer is only awakened by the producer when itemCount is equal to 1.
- ✓ The producer will loop until the buffer is full, after which it will also go to sleep.
- ✓ Since both processes will sleep forever, we have run into a deadlock. This solution therefore is unsatisfactory.

Classical IPC problems

i. The Dining Philosophers Problem

The dining philosophers' problem is a classic synchronization problem which is used to evaluate situations where there is a need of allocating multiple resources to multiple processes**.**

Consider there are five philosophers sitting around a circular dining table. The dining table has five chopsticks and a bowl of noodles in the middle as shown in the below figure. At any instant, a philosopher is either eating or thinking. When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.



From the problem statement, it is clear that a philosopher can think for an indefinite amount of time. But when a philosopher starts eating, he has to stop at some point of time. The philosopher is in an endless cycle of thinking and eating.

An array of five semaphores, stick[5], for each of the five chopsticks.

The code for each philosopher looks like:

```
while(TRUE)
{
   wait(stick[i]);
   /*
      mod is used because if i=5, next
      chopstick is 1 (dining table is circular)
   */
   wait(stick[(i+1) % 5]);
   /* eat */
   signal(stick[i]);
   signal(stick[(i+1) % 5]);
   /* think */}
```

When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down.

But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever. The possible solutions for this are:

- A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.
- Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

Code for solution to dining philosopher problem

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#define N 5
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (phnum + 4) % N
#define RIGHT (phnum + 1) % N
int state[N];
int phil[N] = { 0, 1, 2, 3, 4 };
sem_t mutex;
sem_t S[N];
void test(int phnum)
{
if (state[phnum] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
                // state that eating
                state[phnum] = EATING;
                sleep(2);
        printf("Philosopher %d takes fork %d and %d\n", phnum + 1, LEFT + 1, phnum + 1);
                printf("Philosopher %d is Eating\n", phnum + 1);
                // sem_post(&S[phnum]) has no effect
                // during takefork
                // used to wake up hungry philosophers
                // during putfork
                sem_post(&S[phnum]);
        }
}
```

```
// take up chopsticks
void take_fork(int phnum)
{
	sem_wait(&mutex);
	// state that hungry
	state[phnum] = HUNGRY;
	printf("Philosopher %d is Hungry\n", phnum + 1);
	// eat if neighbours are not eating
	test(phnum);
	sem_post(&mutex);
	// if unable to eat wait to be signalled
	sem_wait(&S[phnum]);
	sleep(1);
}
// put down chopsticks
void put_fork(int phnum)
{
	sem_wait(&mutex);
	// state that thinking
	state[phnum] = THINKING;
printf("Philosopher %d putting fork %d and %d down\n", phnum + 1, LEFT + 1, phnum + 1);
	printf("Philosopher %d is thinking\n", phnum + 1);
	test(LEFT);
	test(RIGHT);
	sem_post(&mutex);
}
void * philospher(void* num)
{
	while (1) {
		int* i = num;
		sleep(1);
		take_fork(*i);
		sleep(0);
		put_fork(*i);
	}
}
int main()
{
	int i;
	pthread_t thread_id[N];
	// initialize the semaphores
	sem_init(&mutex, 0, 1);
	for (i = 0; i < N; i++)
		sem_init(&S[i], 0, 0);
	for (i = 0; i < N; i++) {
		// create philosopher processes
```

```
            pthread_create(&thread_id[i], NULL, philospher, &phil[i]);
            printf("Philosopher %d is thinking\n", i + 1);
        }
        for (i = 0; i < N; i++)
            pthread_join(thread_id[i], NULL);
}
```

ii. Readers and writers' problem

Consider a situation where we have a file shared between many people.

- If one of the people tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her.
- However, if some person is reading the file, then others may read it at the same time.

Precisely in OS we call this situation as the readers-writers problem.

Problem parameters:

- One set of data is shared among a number of processes
- Once a writer is ready, it performs its write. Only one writer may write at a time
- If a process is writing, no other process can read it
- If at least one reader is reading, no other process can write
- Readers may not write and only read

Solution when Reader has the Priority over Writer

Here priority means, no reader should wait if the share is currently opened for reading.
hree variables are used: mutex, wrt, readcnt to implement solution

- **semaphor**e mutex, wrt; // semaphore mutex is used to ensure mutual exclusion when readcnt is updated i.e. when any reader enters or exit from the critical section and semaphore wrt is used by both readers and writers
- **int** readcnt; // readcnt tells the number of processes performing read in the critical section, initially 0

Functions for sempahore :

- wait() : decrements the semaphore value.
- signal() : increments the semaphore value.

Writer process:

1. Writer requests the entry to critical section.
2. If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
3. It exits the critical section.

```
do {
   // writer requests for critical section
   wait(wrt);
   // performs the write
   // leaves the critical section
   signal(wrt);
} while(true);
```

Reader process:

1. Reader requests the entry to critical section.
2. If allowed:
   - it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the wrt semaphore to restrict the entry of writers if any reader is inside.
   - It then, signals mutex as any other reader is allowed to enter while others are already reading.
   - After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore ‐wrt‖ as now, writer can enter the critical section.
3. If not allowed, it keeps on waiting.

```
do {
  // Reader wants to enter the critical section
  wait(mutex);
  // The number of readers has now increased by 1
  readcnt++;
  // there is atleast one reader in the critical section
  // this ensure no writer can enter if there is even one reader
  // thus we give preference to readers here
  if (readcnt==1)
    wait(wrt);
  // other readers can enter while this current reader is inside
  // the critical section
  signal(mutex);
  // current reader performs reading here
  wait(mutex); // a reader wants to leave
  readcnt--;
// that is, no reader is left in the critical section,
  if (readcnt == 0)
     signal(wrt); // writers can enter
  signal(mutex); // reader leaves
} while(true);
```

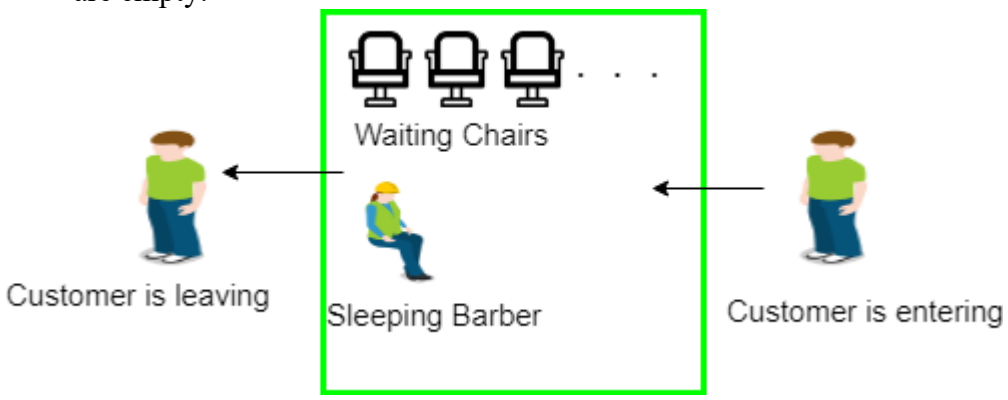
Thus, the semaphore _wrt_ is queued on both readers and writers in a manner such that preference is given to readers if writers are also there. Thus, no reader is waiting simply because a writer has requested to enter the critical section.

iii. Sleeping Barber problem

Problem: The analogy is based upon a hypothetical barber shop with one barber. There is a barber shop which has one barber, one barber chair, and n chairs for waiting for customers if there are any to sit on the chair.

- If there is no customer, then the barber sleeps in his own chair.
- When a customer arrives, he has to wake up the barber.

- If there are many customers and the barber is cutting a customer‘s hair, then the remaining customers either wait if there are empty chairs in the waiting room or they leave if no chairs are empty.

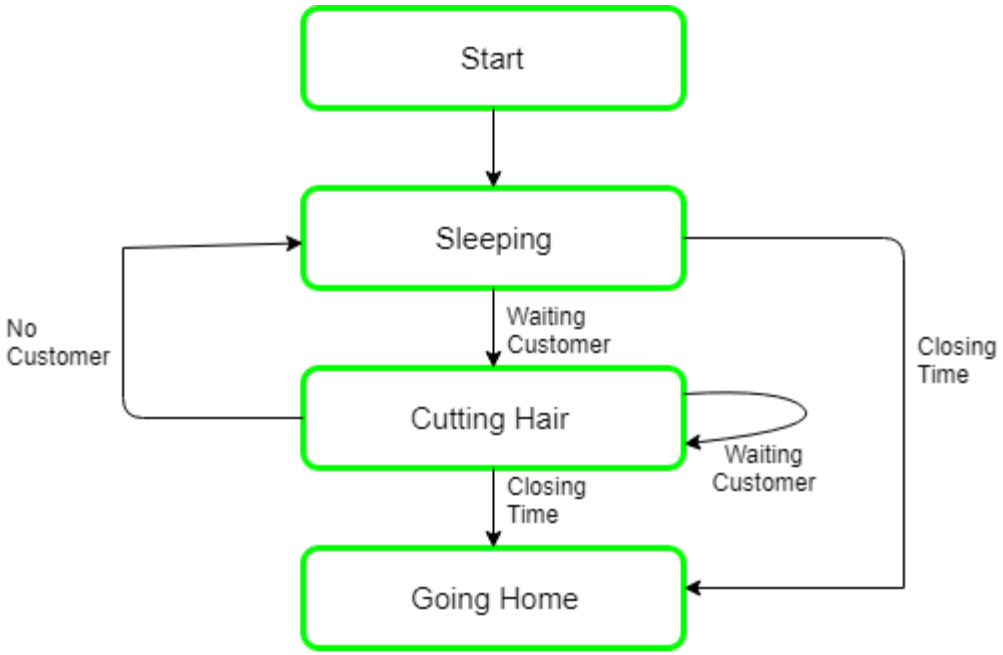


Solution: The solution to this problem includes three semaphores. First is for the customer which counts the number of customers present in the waiting room (customer in the barber chair is not included because he is not waiting). Second, the barber 0 or 1 is used to tell whether the barber is idle or is working, And the third mutex is used to provide the mutual exclusion which is required for the process to execute. In the solution, the customer has the record of the number of customers waiting in the waiting room if the number of customers is equal to the number of chairs in the waiting room then the upcoming customer leaves the barbershop.
When the barber shows up in the morning, he executes the procedure barber, causing him to block on the semaphore customers because it is initially 0. Then the barber goes to sleep until the first customer comes up.

When a customer arrives, he executes customer procedure the customer acquires the mutex for entering the critical region, if another customer enters thereafter, the second one will not be able to anything until the first one has released the mutex. The customer then checks the chairs in the waiting room if waiting customers are less then the number of chairs then he sits otherwise he leaves and releases the mutex.

If the chair is available then customer sits in the waiting room and increments the variable waiting value and also increases the customer‘s semaphore this wakes up the barber if he is sleeping.

At this point, customer and barber are both awake and the barber is ready to give that person a haircut. When the haircut is over, the customer exits the procedure and if there are no customers in waiting room barber sleeps.



Algorithm for Sleeping Barber problem:

```
Semaphore Customers = 0;
Semaphore Barber = 0;
Mutex Seats = 1;
int FreeSeats = N;
Barber {
        while(true) {
                        /* waits for a customer (sleeps). */
                        down(Customers);
                        /* mutex to protect the number of available seats.*/
                        down(Seats);
                        /* a chair gets free.*/
                        FreeSeats++;
                        /* bring customer for haircut.*/
                        up(Barber);
                        /* release the mutex on the chair.*/
                        up(Seats);
                        /* barber is cutting hair.*/
        }
}
Customer {
        while(true) {
                        /* protects seats so only 1 customer tries to sit
                        in a chair if that's the case.*/
                        down(Seats);
                        if(FreeSeats > 0) {
                                /* sitting down.*/
```

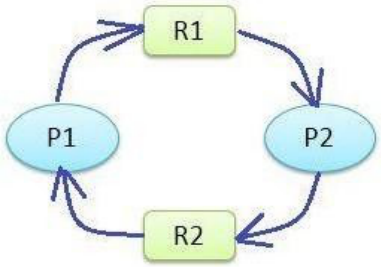
```
                FreeSeats--;
                /* notify the barber. */
                up(Customers);
                /* release the lock */
                up(Seats);
                /* wait in the waiting room if barber is busy. */
                down(Barber);
                // customer is having hair cut
        } else {
                /* release the lock */
                up(Seats);
                // customer leaves
}}}
```

Deadlock and indefinite postponement

Definition

Deadlock is a situation where the several processes in CPU compete for the finite number of resources available within the CPU. Each process holds a resource and wait to acquire a resource that is held by some other process. All the processes wait for resources in a circular fashion.

In the figure below, you can see that Process P1 has acquired resource R2 that is requested by process P2 and Process P1 is requesting for resource R1 which is again held by R2.



So process P1 and P2 form a deadlock.

Deadlock is a common problem in multiprocessing operating systems, distributed systems, and also in parallel computing systems.

Formal definition:

‑A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.‖
Because all the processes are waiting, none of them will ever cause any of the events that could wake up any of the other member in the set, and all the processes continue to wait forever.
That is, none of the processes can-

- run
- release resources
- be awakened

Types of resource

1. **Preemptable resources**
   - these resources can be taken away from the processes owning it with no ill effects
   - For example: memory
2. **Non-Preemptable resources**
   - Resources that cannot be taken away from its current owner without causing the computation to fail.
   - For example: scanner, printer etc.

In general, deadlocks involve non-preemptable resource

Conditions of deadlock

A deadlock situation can arise if and only if the following four conditions hold simultaneously in a system-

- **Mutual Exclusion**: At least one resource is 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.
- **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.
- **No Preemption**: Resources cannot be preempted; that is, a resource can only be released voluntarily by the process holding it, after the process has completed its task.
- **Circular Wait**: There must exist a set $\{p_0, p_1,.....p_n\}$ of waiting processes such that p_0 is waiting for a resource which is held by p_1, p_1 is waiting for a resource which is held by p_2,..., p_{n-1} is waiting for a resource which is held by p_n and p_n is waiting for a resource which is held by p_0.

NOTE: All four of these conditions must be present for a deadlock to occur. If one of them is absent, no deadlock is possible.

Deadlock modeling

- The four Coffman conditions can be modeled using directed graphs
- The graph has two kind of nodes: *circles* for processes and *squares* for resources
- A directed arc from a resource node (square) to a process node (circle) means that the resource is being held by the process

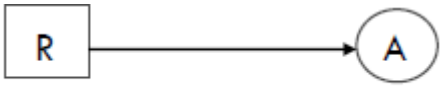


Fig: Process A is holding the resource R

- A directed arc from a process to a resource means that the process is currently requesting that resource

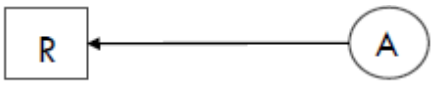


Fig: Process A is requesting the resource R

Handling Deadlock

- Deadlock detection
- Recovery from deadlock
- Deadlock prevention
- Deadlock avoidance

Deadlock detection and Recovery

Deadlock detection

Deadlock detection with one resource of each type

- For a system with one instance of each resource, we can detect deadlock by constructing a resource allocation graph.
- If the graph contains one or more cycles, a deadlock exists. Any process that is part of the cycle is deadlocked.
- If no cycle exists, the system is not deadlocked

Example:

- Consider a complex system with 7 processes, A through G, and 6 resources, R through W.
- The state of which resources are currently owned and which ones are currently being requested is as follows:

1. Process A holds R and wants S.
2. Process B holds nothing but wants T.
3. Process C holds nothing but wants S.
4. Process D holds U and wants S and T.
5. Process E holds T and wants V.
6. Process F holds W and wants S.
7. Process G holds V and wants U.

From this, we can construct resource graph as follows:

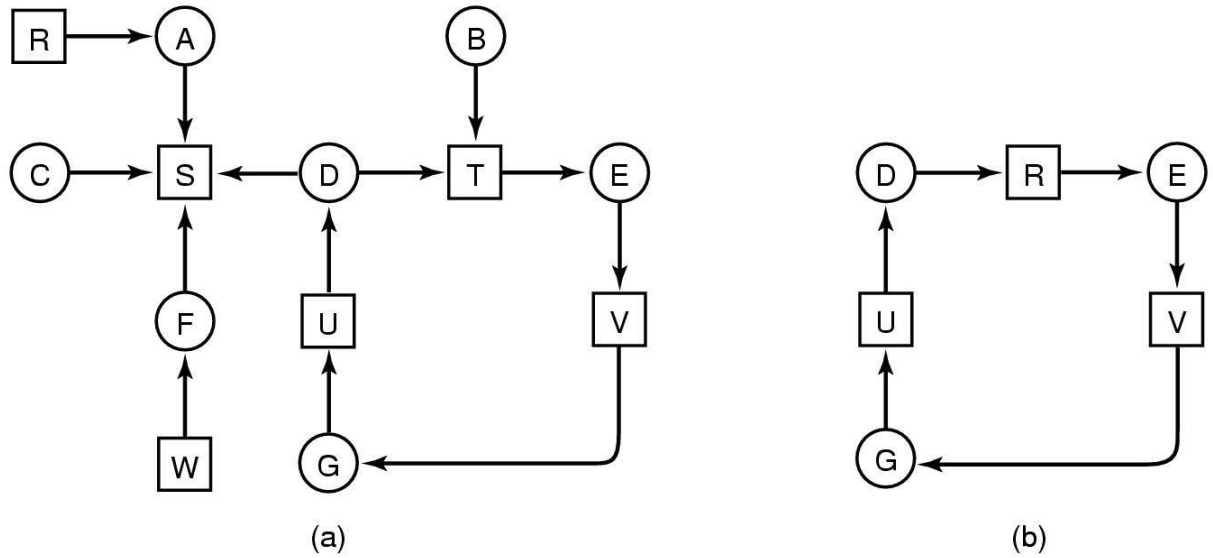


Figure: (a) A resource graph. (b) A cycle extracted from (a).

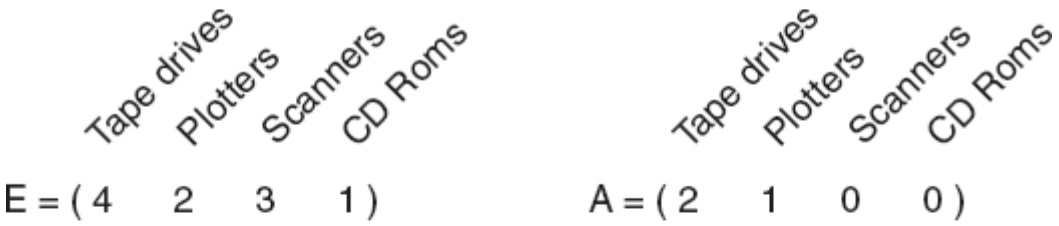
Processes D, E and G are all deadlocked.
Processes A, C and F are not deadlocked because S can be allocated to any one of them, which thenfinishes and take by other two processes in turn.

Deadlock Detection with multiple resource of each type

In this case, we use a matrix based algorithm for detecting deadlocks among n processes (P1,…Pn)
E: Existing resource vector (E1,E2….Em) ; we have m different resource
For example: if class 1 is printer then E1=2 means we have 2 printers
A: Available resource vectors
For example: if A1=0, no printers are available, both printers have been assigned
C: Current Allocation matrix
C_{ij} is the number of instances of resource j that are held by process i
R: Request matrix
R_{ij} is the number of instances of resource j that process i wants.

Deadlock detection algorithm

1. Look for an unmarked process, *Pi* , for which the i-th row of *R* is less than or equal to *A*.
2. If such process is found, add the *i-th* row of *C* to *A*, mark the process, and go back to step 1.
3. If no such process exists, the algorithm terminates.



Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Here, requests represented by 1st and 2nd rows of matrix R cant not be satisfied (Compare A with each Rs) . But 3rd one can be satisfied.

- So process 3 runs and eventually returns A=(2 2 2 0).
- At this point process 2 can run and return A=(4 2 2 1).
- Now process 1 can run and there is no deadlock.
- **What happen** if process 2 needs 1 CD-ROM drive, 2 Tape drives and 1 Plotter (i.e. 2nd row of R becomes (2 1 1 1))?
- Entire system gets deadlock or not???

Deadlock Recovery

Traditional operating system such as Windows doesn't deal with deadlock recovery as it is time and space consuming process. Real time operating systems use Deadlock recovery.

i. Recovery through preemption

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

Issues to be addresses:

a. Selecting a victim
b. Rollback
c. Starvation

ii. Recovery through rollback

- Processes are checkpoint periodically
- Checkpoint refers to save the state of a process by writing the state to a file
- Checkpoint contains not only the memory image but also the resource state (i.e. the resource allocated to the process)

- To be more effective, new checkpoints must not replace the old checkpoints but should be written to a different file
- When a deadlock is detected, we first determine the resource causing the deadlock and then rollback the process which currently holds that resource to a point in time before it acquired the resource by starting one of its earlier checkpoints

iii. Recovery through killing process

The simplest way to recover from deadlock is to kill one or more processes.

Which process to kill?

a. Kill all the deadlocked processes

Sure to break deadlock but can be **expensive** as some of the processes may have computed for a long time and killing the process makes the process to do all those computations once again after it has been started again

b. Abort one process at a time until the deadlock cycle is eliminated.

This method **incurs considerable overhead**, since, after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Deadlock Prevention

Deadlock prevention simply involves techniques to attack the four conditions that may cause deadlock (*Coffman conditions*).

i. Prevention from Mutual exclusion

- The resource should not be assigned to a single process until absolutely necessary.
- For example: instead of assigning a printer to a process, we can create a spool (send (data that is intended for printing or processing on a peripheral device) to an intermediate store.), so that several processes can generate output at the same time

ii. Preventing the Wait and Hold condition

- The process should request all of its required resources at the beginning.
- If every resource is available, the process runs or else it waits but does not hold any of the resource.
- This way hold and wait condition is eliminated but it will lead to low device utilization.
- For example, if a process requires printer at a later time and we have allocated printer before the start of its execution printer will remained blocked till it has completed its execution.

iii. Preventing the non-preemption condition

- Preempt resources from process when resources required by other high priority process.

iv. Preventing Circular Wait

- Each resource will be assigned with a numerical number. A process can request for the resources only in increasing order of numbering.
- For Example, if P1 process is allocated R5 resources, now next time if P1 ask for R4, R3 lesser than R5 such request will not be granted, only request for resources more than R5 will be granted.

Deadlock Avoidance

- We cannot always assume that a process will ask for all its required resource at once as we did in earlier topic: deadlock detection
- The system must decide whether granting a resource is safe or not and make the allocation if it is safe
- So, we need an algorithm to avoid deadlock by making right choices all the time given that some information is available in advance

Safe and unsafe states

- A state is said to be safe if there is some scheduling order in which every process can run up to completion even if all of them suddenly request their maximum number of resources immediately
- 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 *<P1, P2, ..., Pn>* is a safe sequence for the current allocation state if, for each *Pi,* the resource requests that *Pi* can still make can be satisfied by the currently available resources plus the resources held by all *P*
- A safe state is a non-deadlock state.
- Conversely, a deadlocked state is an unsafe state.
- Not all unsafe states are deadlocks; however an unsafe state *may* lead to a deadlock.
- As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states

Note: Safe and unsafe states can be determined by using Bankers algorithm.

The Banker's algorithm

The Banker's algorithm is a resource allocation and deadlock avoidance algorithm developed by Edsger Dijkstra. Resource allocation state is defined by the number of available and allocated resources and the maximum demand of the processes. When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.

Algorithm

1) Find a row in the Need matrix which is less than the Available vector. If such a row exists, then the process represented by that row may complete with those additional resources. If no such row exists, eventual deadlock is possible.
2) Double check that granting these resources to the process for the chosen row will result in a safe state. Looking ahead, pretend that that process has acquired all its needed resources, executed, terminated, and returned resources to the Available vector. Now the value of the Available vector should be greater than or equal to the value it was previously.
3) Repeat steps 1 and 2 until
    a) all the processes have successfully reached pretended termination (this implies that the initial state was safe); or
    b) deadlock is reached (this implies the initial state was unsafe).

Basic Facts:

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

Some terminologies

a) Available

It represents the number of available resources of each type.

b) Max

It represents the maximum number of instances of each resource that a process can request.

c) Allocation

It represents the number of resources of each type currently allocated to each process.

d) Need

It indicates the remaining resource needs of each process.

Bankers Algorithms for single resource

Example 1: State whether the given processes are in deadlock or not. Given that resource instance is 10.

process	Allocated	Maximum
A	3	9
B	2	4
C	2	7

Solution,

Calculating need resources, using

Need = Maximum - Allocated we get,

process	Allocated	Maximum	Need
A	3	9	6
B	2	4	2
C	2	7	5

Here currently total allocation = 3+2+2 = 7

So free = total available – current allocation = 10 – 7 = 3

- **Step 1**

With current free resources process B can be executed, since need of B ≤ Free i.e 2 ≤ 3 So B executes. After execution of B it release the resources allocated by it.

Total free resource becomes, free = current free + Allocation by B = (1+2+2) = 5

- **Step 2**

Now, with current free resources process C can be executed, since need of C ≤ Free i.e 5 ≤ 5 So C executes. After execution of C it release the resources allocated by it.

Total free resource becomes, free = current free + Allocation by C = (5+2) = 7

- **Step 3**

With current free resources process, A can be executed, since need of A ≤ Free i.e 6 ≤ 7 So A executes. After execution of A, it releases the resources allocated by it.
Total free resource becomes, free = current free + Allocation by A = (1+6+3) = 10

Here all the process runs successfully hence they are is safe state and occurs no deadlock.
Safe sequence is: **B→C→A**

Example 2: State whether the given processes are in deadlock or not. Given that resource instance is 10.

process	Allocated	Maximum
A	4	9
B	2	4
C	2	7

Solution,
Calculating need resources, using
Need = Maximum - Allocated we get,

process	Allocated	Maximum	Need
A	4	9	5
B	2	4	2
C	2	7	5

Here currently total allocation = 4+2+2 = 8
So free = total available – current allocation = 10 – 8 = 2

- **Step 1**

With current free resources process B can be executed, since need of B ≤ Free i.e 2 ≤ 2 So B executes. After execution of B it release the resources allocated by it.
Total free resource becomes, free = current free + Allocation by B = (2+2) = 4

With current free resources none of the processes can be further be executed hence process are unsafe and occurs deadlock.

Bankers Algorithm for multiple resources

Example 1: A system has four process P1, P2, P3 and P4 and three resources R1, R2 and R3 with existing resources E = (15, 9, 5). After allocating resources to all the processes available resources becomes A = (3, 2, 0). State whether the process is safe or not using banker‘s algorithm. If safe, write the safe sequence.

Process	Allocation			Maximum			Need		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	3	0	1	3	2	2	0	2	1
P2	5	4	1	6	8	2	1	4	1
P3	2	2	0	3	2	4	1	0	4
P4	2	1	3	4	2	3	2	1	0

Note: If Need is not given, it can be calculated using Need = Maximum – Allocation

Solution:

We have A = (3, 2, 0)

Step 1: With current available resources A = (3, 2, 0) P4 can be executed.

Since need of P4 ≤ A i.e. (2,1,0) ≤ (3,2,0) so P4 executes

After complete execution of P4 it releases the resources which is allocated by it. Now total current available resources A becomes, A = previous free + Allocation by P4

$$A = (3,2,0) + (2,1,3) = (5,3,3)$$

Step 2: With current available resources A = (5, 3, 3) P1 can be executed.

Since need of P1 ≤ A i.e. (0,2,1) ≤ (5,3,3) so P1 executes

After complete execution of P1 it releases the resources which is allocated by it. Now total current available resources A becomes, A = previous free + Allocation by P1

$$A = (5,3,3) + (3,0,1) = (8,3,4)$$

Step 3: With current available resources A = (8, 3, 4) P3 can be executed.

Since need of P3 ≤ A i.e. (1,0,4) ≤ (8,3,4) so P3 executes

After complete execution of P3 it releases the resources which is allocated by it. Now total current available resources A becomes, A = previous free + Allocation by P3

$$A = (8,3,4) + (2,2,0) = (10,5,4)$$

Step 4: With current available resources A = (10, 5, 4) P2 can be executed.

Since need of P2 ≤ A i.e. (1,4,1) ≤ (10,5,4) so P2 executes

After complete execution of P2 it releases the resources which is allocated by it. Now total current available resources A becomes, A = previous free + Allocation by P2

$$A = (10,5,4) + (5,4,1) = (15,9,5)$$

Here all the process runs hence they are in safe state and occurs no deadlock.

Safe sequence is: **P4→P1→P3→P2**

Example 2:

Assume we have the following resources:

- 5 tape drives
- 2 graphic displays
- 4 printers
- 3 disks

We can create a vector representing our total resources: Total = (5, 2, 4, 3).

Consider we have already allocated these resources among four processes as demonstrated by the following matrix named **Allocation.**

Process Name	Tape Drives	Graphics	Printers	Disk Drives
Process A	2	0	1	1
Process B	0	1	0	0
Process C	1	0	1	1
Process D	1	1	0	1

The vector representing the allocated resources is the sum of these columns:
Allocated = (4, 2, 2, 3).
We also need a matrix to show the number of each resource still needed for each process; we call this matrix **Need.**

Process Name	Tape Drives	Graphics	Printers	Disk Drives
Process A	1	1	0	0
Process B	0	1	1	2
Process C	3	1	0	0
Process D	0	0	1	0

The vector representing the available resources will be the sum of these columns subtracted from the Allocated vector: **Available** = (1, 0, 2, 0).

Following the algorithm sketched above,

Iteration 1:

Examine the Need matrix. The only row that is less than the Available vector is the one for Process D.

Need(Process D) = (0, 0, 1, 0) < (1, 0, 2, 0) = **Available**

If we assume that Process D completes, it will turn over its currently allocated resources, incrementing the Available vector.

(1, 0, 2, 0) Current value of Available

\+ (1, 1, 0, 1) Allocation (Process D)

` ` ` ` ` ` ` ` ` ` ` ` ` ` ` ` `

(2, 1, 2, 1) **Updated** value of **Available**

Iteration 2:

Examine the Need matrix, ignoring the row for Process D. The only row that is less than the Available vector is the one for Process A.

Need(Process A) = (1, 1, 0, 0) < (2, 1, 2, 1) = Available

If we assume that Process A completes, it will turn over its currently allocated resources, incrementing the Available vector.

(2, 1, 2, 1) Current value of Available

\+ (2, 0, 1, 1) Allocation (Process A)

` ` ` ` ` ` ` ` ` ` ` ` ` ` ` `

(4, 1, 3, 2) **Updated** value of **Available**

Iteration 3:

Examine the Need matrix without the row for Process D and Process A. The only row that is less than the Available vector is the one for Process B.

Need(Process B) = (0, 1, 1, 2) < (4, 1, 3, 2) = **Available**

If we assume that Process B completes, it will turn over its currently allocated resources, incrementing the Available vector.

(4, 1, 3, 2) Current value of Available

\+ (0, 1, 0, 0) Allocation (Process B)

` ` ` ` ` ` ` ` ` ` ` ` ` ` ` `

(4, 2, 3, 2) **Updated** value of **Available**

Iteration 4:

Examine the Need matrix without the rows for Process A, Process B, and Process D. The only row left is the one for Process C, and it is less than the Available vector.

Need(Process C) = (3, 1, 0, 0) < (4, 2, 3, 2) = Available

If we assume that Process C completes, it will turn over its currently allocated resources, incrementing the Available vector.

(4, 2, 3, 3) Current value of Available

\+ (1, 0, 1, 1) Allocation (Process C)

` ` ` ` ` ` ` ` ` ` ` ` ` ` ` `

(5, 2, 4, 3) **Updated** value of **Available**

Notice that the final value of the Available vector is the same as the original Total vector, showing the total number of all resources:

Total = (5, 2, 4, 2) < (5, 2, 4, 2) = Available

This means that the initial state represented by the Allocation and Need matrices is a safe state.

The safe sequence that assures this safe state is <D, A, B, C>.

Note: The Banker's algorithm can also be used in the detection of deadlock.

Disadvantages of the Banker's Algorithm

- It requires the number of processes to be fixed; no additional processes can start while it is executing.
- It requires that the number of resources remain fixed; no resource may go down for any reason without the possibility of deadlock occurring.
- It allows all requests to be granted in finite time, but one year is a finite amount of time.
- Similarly, all of the processes guarantee that the resources loaned to them will be repaid in a finite amount of time. While this prevents absolute starvation, some pretty hungry processes might develop.
- All processes must know and state their maximum resource need in advance.

Issues related to deadlock

- **Two-phase locking**

In databases and transaction processing, two-phase locking (2PL) is a concurrency control method that guarantees serializability. It is also the name of the resulting set of database transaction schedules (histories). The protocol utilizes locks, applied by a transaction to data, which may block (interpreted as signals to stop) other transactions from accessing the same data during the transaction's life. By the 2PL protocol, locks are applied and removed in two phases:

- **Expanding phase**: locks are acquired and no locks are released.
- **Shrinking phase**: locks are released and no locks are acquired.

Two types of locks are utilized by the basic protocol: Shared and Exclusive locks. Refinements of the basic protocol may utilize more lock types. Using locks that block processes, 2PL may be subject to deadlocks that result from the mutual blocking of two or more transactions.

- **Non-resource deadlock**

In non-resource deadlock, deadlock occurs without the involvement of any resources.
Say you have two nodes in a network that communicate and have a 3 steps handshake:

- node1 sends a message to node2 and waits for a response
- node2 receives the message and sends back the response to node1 and waits
- but the response is lost on the network due to a temporary disruption

Both nodes are waiting for each other => deadlock

- **Starvation**

Starvation is the phenomena in which a process is not able to acquire the desired resources (like processor, I/O device etc) for a very long time to progress with its execution.

This can happen due to drawbacks of scheduling algorithms. Scheduling algorithms are used to decide to which process the resource(s) needs to be given next.
Example:

- In Shortest Job First Algorithm, if there is a process with a rather large CPU burst waiting for its execution and a stream of processes with smaller CPU burst keep entering the ready queue (a queue consisting of those processes which are ready to execute), then the CPU will always be allocated to the process with the shorter CPU burst and there is a probability

that the process with the large CPU burst might never get the CPU to complete its execution and starve.

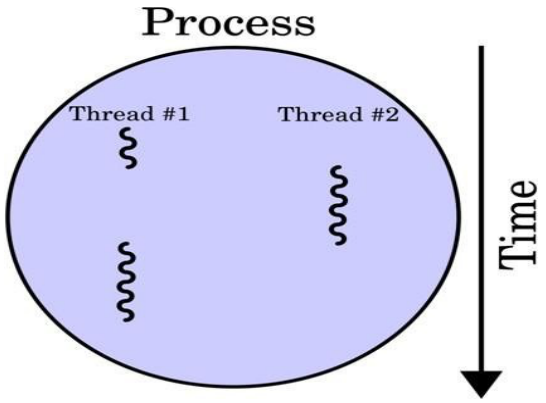
- In Priority Scheduling, a constant stream of high priority processes might starve one or more lower priority process(es) as the CPU will always be allocated to the process with highest priority.

Deadlock vs Starvation

Deadlock	Starvation
Deadlock occurs when none of the processes in the set is able to move ahead due to occupancy of the required resources by some otherprocess	Starvation occurs when a process waits for an indefinite period of time to get the resource it requires.
It is also known as Circular waiting	It is also known lived lock.
In deadlocked, requested resources are blocked by the other processes.	In starvation, the requested resources are continuously used by high priority processes.
These four conditions that lead to deadlock: mutual exclusion, hold and wait, no-preemption and circular wit	Uncontrolled management of resources, Process priorities , Use of random selection leads to starvation.
No process can make progress once it occurs.	Apart from the victim process other processes can progress or proceed
Requires external intervention to solve deadlock.	May or may not require external intervention to solve starvation.

Threads

A thread is the smallest unit of programmed instructions that can be managed independently and processing that can be performed by an operating system. A thread is also called a **lightweight process**. Multiple threads can be existing within a same process that share common resources such as memory. If a process has multiple threads of control, it can perform more than one task at a time. Threads within a process share the same virtual memory space but each has a separate stack,and possibly "thread-local storage" if implemented. They are lightweight because a context switch is simply a case of switching the stack pointer and program counter and restoring other registers, whereas a process context switch involves switching the MMU context as well.



For example: a web browser can have a thread to display images or text while another thread retrieves data from the network.
Also in a word processor, a **background thread** may **check spelling and grammar** while a **foreground thread** processes **user input** (keystrokes), while yet a **third thread loads images** from the hard drive, and a **fourth** does **periodic automatic backups** of the file being edited.

Advantages of using Threads

- Threads are easy and inexpensive to create. Threads are 10 times as faster to create than process (studies done by Mach developers)
- It takes less time to terminate the thread than process.
- Use of thread provides concurrency within process.
- Thread minimizes context switching time (i.e. switching between two threads within thesame process is faster)
- Threads enhance efficiency in communication between different executing programs. (Threads within the same process share data and code memory which allows them to communicate without involving the kernel)
- Utilization of multiprocessor architectures to a greater scale and efficiency

Types/ model of Threads

1. **User Level Threads**

- Thread management is done in **user space** of main memory by a **user-level thread library.**
- The user-level or programmer takes the information from thread library which contains the code about creation, destroy, schedule, execution and restoring of threads.
- Kernel don't know about these threads
- User Level Threads are faster to create and manage

Advantages:

1. 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.
2. User-level threads do not require modification to operating systems.
3. **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.
4. **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.

5. Fast and Efficient: Thread switching is not much more expensive than a procedure call.

Disadvantages:

1. User-Level threads are not a perfect solution as with everything else, they are a tradeoff. Since, User-Level threads are invisible to the OS they are not well integrated with the OS. As a result, Os can make poor decisions like scheduling a process with idle threads, blocking a process whose thread initiated an I/O even though the process has other threads that can run and unscheduling a process with a thread holding a lock. Solving this requires communication between between kernel and user-level thread manager.
2. 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.
3. User-level threads require non-blocking systems call i.e., a multithreaded kernel. Otherwise, entire process will block in the kernel, even if there are runnable threads left in the processes. For example, if one thread causes a page fault, the process blocks.

2. Kernel Level Threads

- Thread that are defined and managed by directly kernel of Operating system is kernel level threads
- Kernel performs thread creation, scheduling and management
- Kernel threads are used in internal working of operating system
- Kernel threads are slower to create and manage

Advantages:

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

Disadvantages:

1. The kernel-level threads are slow and inefficient. For instance, threads operations are hundreds of times slower than that of user-level threads.
2. Since kernel must manage and schedule threads as well as processes. It requires 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.

User implementation of threads

Implement the entire threads package in user space without the kernel knowing about them. When one thread gets blocked by a system call, all user level threads get blocked. As far as the kernel is concerned, it is managing ordinary, single-threaded processes. A user-level threads package can be implemented on an operating system that does not support threads.

The threads run on top of a run-time system, which is a collection of procedures that manage threads. When threads are managed in user space, each process needs its own private thread table to keep track of the threads in that process. This table is analogous to the kernel‘s process table, except that it keeps tracks only of the per-thread properties, such as each thread‘s program counter, stack pointer, registers, state, and so forth. The thread table is managed by the run-time system. When a thread is moved to ready state or blocked state, the information needed to restart it is stored

in the thread table, exactly the same way as the kernel stores information about processes in the process table.

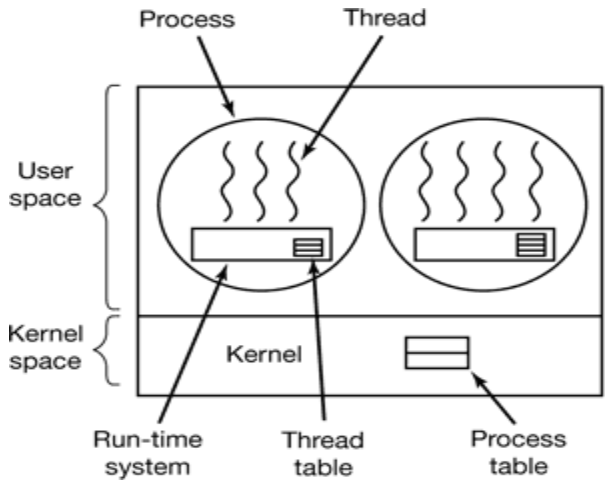


Figure: A thread package managed by user space

Kernel implementation of threads

The kernel's thread table holds each thread registration, state and other information. The information is the same as with the use-level threads, but now kept in the kernel instead of the user space (inside the run-time system). This information is the subset of information that traditional kernels maintain about their single-threaded process, that is, the process state. In addition, the kernel also maintains the traditional process table to keep track of processes.

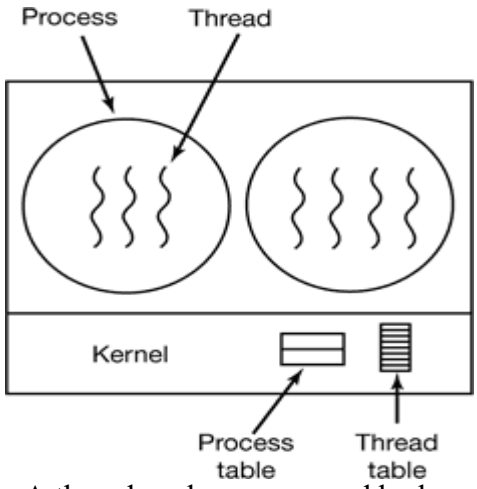


Figure: A thread package managed by kernel

Thread usage

Threads provide a convenient way of allowing an application to maximize its usage of CPU resources in a system, especially in a multiple processor configuration. In a Remote Desktop Services environment, however, multiple users can be running multithreaded applications, and all of the threads for all of the users compete for the central CPU resources of that system. With this in mind, you should tune and balance application thread usage for a multiuser, multiprocessor Remote Desktop Services environment. While a poorly designed multithreaded application with idle or wasted threads might perform adequately on a client computer, the same application might not perform well on a multiuser Remote Desktop Services server.

process vs Threads

	Process	**Thread**
Definition	An executing instance of a program is called a process.	A thread is a subset of the process.
Process	It has its own copy of the data segment of the parent process.	It has direct access to the data segment of its process.
Communication	Processes must use inter-process communication to communicate with sibling processes.	Threads can directly communicate with other threads of its process.
Overheads	Processes have considerable overhead.	Threads have almost no overhead.
Creation	New processes require duplication of the parent process.	New threads are easily created.
Changes	Any change in the parent process does not affect child processes.	Any change in the main thread may affect the behavior of the other threads of the process.
Memory	Run in separate memory spaces.	Run in shared memory spaces.
Dependence	Processes are independent.	Threads are dependent.

Chapter 3 Kernel

Introduction and Architecture of Kernel

A kernel is the core component of an operating system. Using interprocess communication and system calls, it acts as a bridge between applications and the data processing performed at the hardware level.

When an operating system is loaded into memory, the kernel loads first and remains in memory until the operating system is shut down again. The kernel is responsible for low-level tasks such as disk management, task management and memory management. A computer kernel interfaces between the three major computer hardware components, providing services between the application/user interface and the CPU, memory and other hardware I/O devices.

The kernel provides and manages computer resources, allowing other programs to run and use these resources. The kernel also sets up memory address space for applicatns, loads files with application code into memory, sets up the execution stack for programs and branches out to particular locations inside programs for execution.

The kernel is responsible for:

- Process management for application execution
- Memory management, allocation and I/O
- Device management through the use of device drivers
- System call control, which is essential for the execution of kernel services

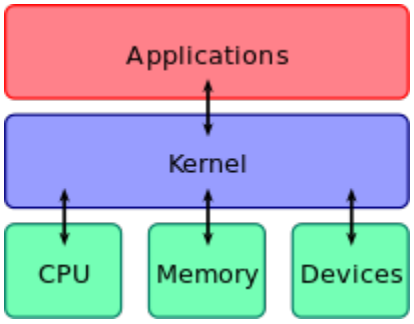


Fig: Architecture of Kernel

Types of kernel

Kernels can be broadly classified mainly in two categories:

- ✓ Monolithic Kernel
- ✓ Micro Kernel.

1. Monolithic Kernels

Earlier in this type of kernel architecture, all the basic system services like a process and memory management, interrupt handling etc were packaged into a single module in kernel space. This type of architecture led to some serious drawbacks like:

- The Size of the kernel, which was huge.
- Poor maintainability, which means bug fixing or addition of new features resulted in recompilation of the whole kernel which could consume hours.

In a modern-day approach to monolithic architecture, the kernel consists of different modules which can be dynamically loaded and unloaded. This modular approach allows easy extension of OS's capabilities. With this approach, maintainability of kernel became very easy as only the concerned module needs to be loaded and unloaded every time there is a change or bug fix in a particular module. So, there is no need to bring down and recompile the whole kernel for the smallest bit of change. Also, stripping of a kernel for various platforms (say for embedded devices etc) became very easy as we can easily unload the module that we do not want. Linux follows the monolithic modular approach.

2. **Microkernels**

This architecture majorly caters to the problem of ever growing size of kernel code which we could not control in the monolithic approach. This architecture allows some basic services like device driver management, protocol stack, file system etc to run in user space. This reduces the kernel code size and also increases the security and stability of OS as we have the bare minimum code running in kernel. So, if suppose a basic service like network service crashes due to buffer overflow, then only the networking service's memory would be corrupted, leaving the rest of the system still functional.

In this architecture, all the basic OS services which are made part of user space are made to run as servers which are used by other programs in the system through inter process communication (IPC). eg: we have servers for device drivers, network protocol stacks, file systems, graphics, etc. Microkernel servers are essentially daemon programs like any others, except that the kernel grants some of them privileges to interact with parts of physical memory that are otherwise off limits to most programs. This allows some servers, particularly device drivers, to interact directly with hardware. These servers are started at the system start-up.

Other types of kernels include:

- **Hybrid Kernel**: This is a mix between the 2 above. The kernel is larger than hybrid but smaller than monolithic. What you normally get is a stripped-down monolithic kernel that has the majority of device drivers removed but still all of the system services within the kernel space. The device drivers will be attached to the kernel as required when starting up or running. These kernels are typically found on desktops, your Windows, Mac and Linux OS flavors.

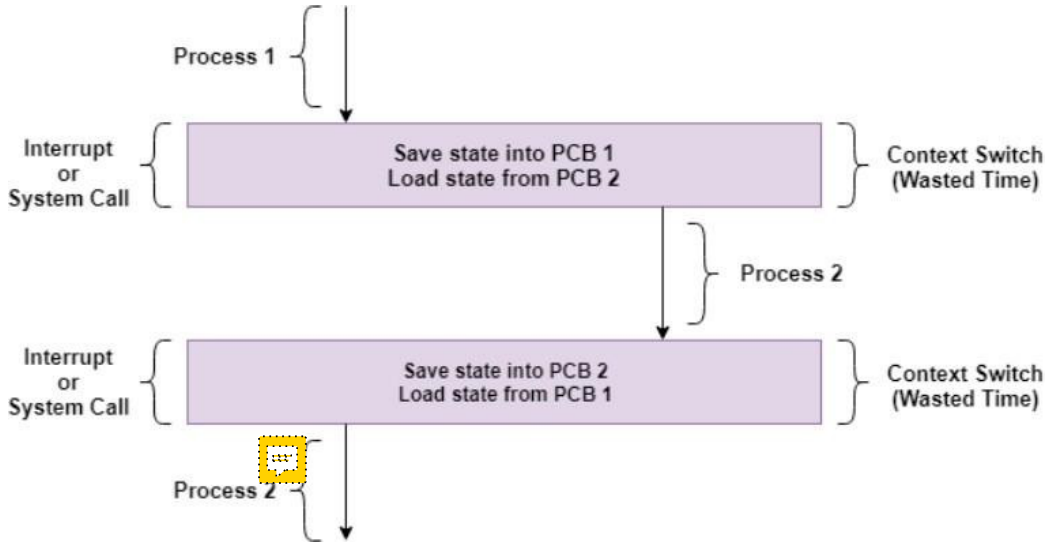
- **Nano kernel**. This kernel type only offers hardware abstraction, there are no services and the kernel space is at a minimum. A Nano kernel forms the basis of a hypervisor upon which you may emulate multiple systems via virtualization. Nano kernels are also very good for embedded projects.

- **Exo-Kernel**. This kernel is the smallest kernel. It offers process protection and resource handling and nothing else. The programmer using this kernel is responsible for correctly accessing the device they wish to use.

Context Switching (Kernel mode and User mode)

A context switch is a procedure that a computer's CPU (central processing unit) follows to change from one task (or process) to another while ensuring that the tasks do not conflict. Context Switching involves storing the context or state of a process so that it can be reloaded when required and execution can be resumed from the same point as earlier. This is a feature of a multitasking operating system and allows a single CPU to be shared by multiple processes.

A diagram that demonstrates context switching is as follows:



In the above diagram, initially Process 1 is running. Process 1 is switched out and Process 2 is switched in because of an interrupt or a system call. Context switching involves saving the state of Process 1 into PCB1 and loading the state of process 2 from PCB2. After some time again a context switch occurs and Process 2 is switched out and Process 1 is switched in again. This involves saving the state of Process 2 into PCB2 and loading the state of process 1 from PCB1.

When does context switching take place?

- **Multitasking:** In a multitasking environment, a process is switched out of the CPU so another process can be run. The state of the old process is saved and the state of the new process is loaded. On a pre-emptive system, processes may be switched out by the scheduler.
- **Interrupt Handling:** The hardware switches a part of the context when an interrupt occurs. This happens automatically. Only some of the context is changed to minimize the time required to handle the interrupt.
- **User and Kernel Mode Switching:** A context switch may take place when a transition between the user mode and kernel mode is required in the operating system.

Context Switching Steps:

The steps involved in context switching are as follows:

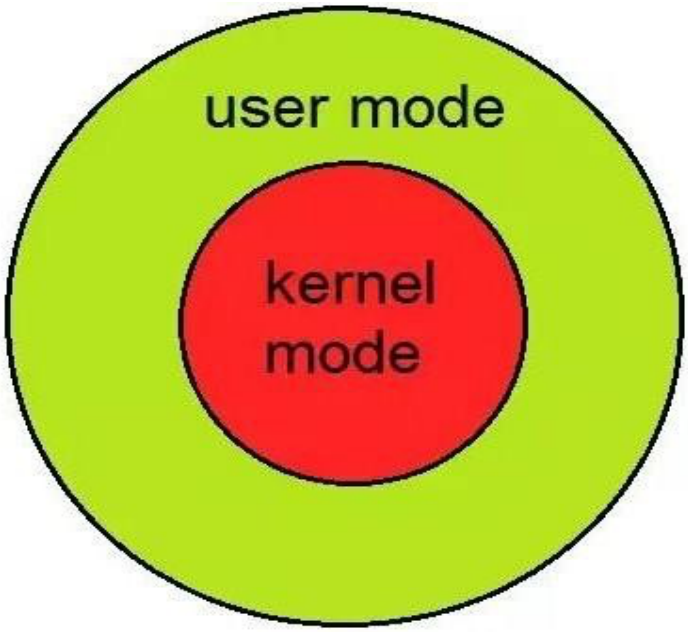
- Save the context of the process that is currently running on the CPU. Update the process control block and other important fields.
- Move the process control block of the above process into the relevant queue such as the ready queue, I/O queue etc.
- Select a new process for execution.
- Update the process control block of the selected process. This includes updating the process state to running.
- Update the memory management data structures as required.
- Restore the context of the process that was previously running when it is loaded again on the processor. This is done by loading the previous values of the process control block and registers.

Kernel Mode

- When CPU is in kernel mode, the code being executed can access any memory address and any hardware resource.
- It is a very privileged and powerful mode.
- If a program crashes in kernel mode, the entire system will be halted.
- In Kernel mode, the executing code has complete and unrestricted access to the underlying hardware.
- core operating system components run in kernel mode.

User Mode

- When CPU is in user mode, the programs don‘t have access to memory and hardware resources.
- In user mode, if any program crashes, only that particular program is halted. That means the system will be in a safe state even if a program in user mode crashes.
- Most programs in an OS run in user mode.
- In Kernel mode, the executing code has complete and unrestricted access to the underlying hardware.
- Applications run in user mode



First Level Interrupt Handling

The operating system includes routine called FLIH to process each different class of interrupt. When an interrupt occurs, the operating system saves the status of the interrupted process and routes control to the appropriate First level interrupt handler, this is done by context switching. There are six first level interrupt handler.

1. **SVC (Supervisor Call) FLIH**: These are initiated by the running process executing the SVC instruction. SVC is a user generated request for a particular system service such as performing I/O request, obtaining more storage. SVC mechanism help to keep the Operating System secure from user process. A user must request a service through a SVC. The OS is thus aware of all user attempts to cross its boarder and it may refuse certain request if the user does not have appropriate privileges.
2. **I/O interrupts**: These are initiated by the input/output hardware. They signal the CPU if the current status of the device is changed. I/O interrupt are caused when an I/O operation completes, when an I/O error occurs or when a device is ready.
3. **External interrupts**: These are caused on the receipt of a signal from another processor on a multiprocessor system.
4. **Restart FLIH:** These occur when the operator presses the restart button or when a restart instruction arrives from another processor on a multiprocessor system.
5. **Program Check FLIH**: These are caused by the wide range of problem that may occur as a program machine language is executed. These problems include division by zero, arithmetic overflow, data in incorrect format, attempts to execute and invalid operation code, attempt to reference a memory location beyond the limit of real memory. Many systems give user the option to specify their own routine to be executed when a program check interrupt occurs.
6. **Machine Check FLIH**: These are caused by malfunctioning of hardware.

Kernel Implementation of processes

Most of the operating system executes within the environment of a user process. We need two modes: User mode and Kernel mode. Some portion of the operating system operate as a separate process known as kernel process. Process creation in Unix is made by means of kernel system call fork ().

When a process issues a fork request, the operating system perform the following function:

i. It allocates a slot in process table for a new process.
ii. It assigns a unique process ID to a child process.
iii. It assigns the child process to a ready to run state.
iv. It returns the ID number of the child process to parent process and zero value to a child process.

All this work is accomplished in kernel mode in the parent process. When a kernel has completed these functions, it can do one of the following routines

i. **Stay in parent process**: Control return to the user mode at the point of the fork call of the parent.
ii. **Transfer control to the child process**: The child process begins executing at the same point in the code as the parent, at the return from fork call.
iii. **Transfer control to another process**: Both the child and the parent process are left in ready to run state.

Example:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
  // make two process which run same
   // program after this instruction
   fork();
   printf("I am Nabraj!");
   return 0;
}
```

Output:
I am Nabraj!
I am Nabraj!

Number of times I am Nabraj! printed is equal to number of process created.
Total Number of Processes = 2^n where n is number of fork system calls.
So here n = 1, 2^1 = 2

Chapter 4 Scheduling

Introduction

Scheduling is the method by which task specified by some means is assigned to resources that complete the task. 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 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.

Objective and criteria of process scheduling

- Max CPU utilization [Keep CPU as busy as possible]
- Fair allocation of CPU.
- Max throughput [Number of processes that complete their execution per time unit]
- Min turnaround time [Time taken by a process to finish execution]
- Min waiting time [Time a process waits in ready queue]
- Min response time [Time when a process produces first response]

Scheduling Level

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 levels:

- 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.

Short Term Scheduler

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

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

Medium Term Scheduler

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

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

Comparison among Scheduler

	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

Preemptive vs. Non-Preemptive Scheduling

1. Preemptive Scheduling:

Preemptive scheduling is used when a process switches from running state to ready state or from waiting state to ready state. The resources (mainly CPU cycles) are allocated to the process for the limited amount of time and then is taken away, and the process is again placed back in the ready queue if that process still has CPU burst time remaining. That process stays in ready queue till it gets next chance to execute.

Algorithms based on preemptive scheduling are: Round Robin (RR), Shortest Job First (SJF basically non preemptive) and Priority (non-preemptive version), etc.

2. Non-Preemptive Scheduling:

Non-preemptive Scheduling is used when a process terminates, or a process switch from runningto waiting state. In this scheduling, once the resources (CPU cycles) are allocated to a process, the process holds the CPU till it gets terminated or it reaches a waiting state. In case of non-preemptive scheduling does not interrupt a process running CPU in middle of the execution. Instead, it waits till the process complete its CPU burst time and then it can allocate the CPU to another process.

Algorithms based on preemptive scheduling are: Shortest Remaining Time First (SRTF), Priority (preemptive version), etc.

Comparison of preemptive and Non preemptive scheduling

BASIS	PREEMPTIVE SCHEDULING	NON-PREEMPTIVE SCHEDULING
Basic	The resources are allocated to a process for a limited time.	Once resources are allocated to a process, the process holds it till it completes its burst time Or switches to waiting state.
Interrupt	Process can be interrupted in Between.	Process cannot be interrupted till it terminates Or switches to waiting state.
Starvation	If a high priority process frequently arrives in the ready Queue, low priority process may starve.	If a process with long burst time is running CPU, then another process with less CPU burst time may starve.
Overhead	Preemptive scheduling has Overheads of scheduling the processes.	Non-preemptive scheduling does not have overheads.
Flexibility	Preemptive scheduling is flexible.	Non-preemptive scheduling is rigid.
Cost	Preemptive scheduling is cost Associative.	Non-preemptive scheduling is not cost Associative.

Scheduling Techniques

Different time with respect to a process

- **Arrival Time(AT):** Time at which the process arrives in the ready queue.
- **Burst Time(BT):** Time required by a process for CPU execution.
- **Completion Time(CT):** Time at which process completes its execution.
- **Turn Around Time(TAT):** The time interval from the time of submission of a process to the time of the completion of the process.
  OR,
  Time Difference between completion time and arrival time. i.e. **TAT= CT - AT**
- **Waiting Time(WT):** The total time waiting in a ready queue.
  OR**,**
  Time Difference between turnaround time and burst time. i.e.**WT = TAT - BT**
- **Response time(RT):** Amount of time from when a request was submitted until the first response is produced.
- **Quantum size:** It is the specific time interval used to prevent any one process monopolizing the system i.e. can be run exclusively.

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms**.** The rule or set of rules that is followed to schedule the process is known as process scheduling algorithm**.** The algorithms are either non-preemptive or preemptive**.**

Some of the algorithms are:

i. First Come First Serve (FCFS) Scheduling

- It is a non-preemptive scheduling algorithm.
- The process which arrives first, gets executed first or the process which requests the CPU first, gets the CPU allocated first.
- First Come First Serve, is just like FIFO(First in First out) Queue data structure, where the data element which is added to the queue first, is the one who leaves the queue first.
- A perfect real life example of FCFS scheduling is **buying tickets at ticket counter**.

Advantages

- FCFS algorithm doesn't include any complex logic, it just puts the process requests in a queue and executes it one by one. Hence, FCFS is pretty simple and easy to implement.
- Eventually, every process will get a chance to run, so starvation doesn't occur.

Disadvantages

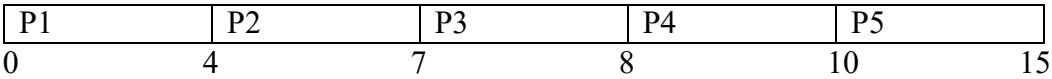
- There is no option for pre-emption of a process. If a process is started, then CPU executes the process until it ends.
- Because thereis no pre-emption, if a process executes for a long time, the processes in the back of the queue will have to wait for a long time before they get a chance to be executed.

Example 1: Consider the following set of processes having their arrival time and CPU-burst time. Find the average waiting time and turnaround time using FCFS.

Process	Arrival Time (ms)	Burst Time (ms)
P1	0	4
P2	1	3
P3	2	1
P4	3	2
P5	4	5

Solution:

Preparing Gantt chart using FCFS we get,



We know,

TAT (Turnaround Time) = CT (Completion Time) – AT (Arrival Time) and
WT (Waiting Time) = TAT (Turnaround Time) – BT (Burst Time)

Now finding TAT and WT,

Process	AT	BT	CT	TAT	WT
P1	0	4	4	4	0
P2	1	3	7	6	3
P3	2	1	8	6	5
P4	3	2	10	7	5
P5	4	5	15	11	6
				∑TAT= 34	**∑WT= 19**

Average Turnaround time(ATAT) = $\frac{\sum}{} = \frac{34}{5} = 6.8$ ms

Average Waiting Time (AWT) = $\frac{\sum}{} = \frac{18}{5} = 3.6$ ms

Example 2: Consider the given processes with arrival and burst time given below. Schedule the given processes using FCFS and find average waiting time (AWT) and average turn-around time (ATAT).

PROCESSES	ARRIVAL TIME	BRUST TIME
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2
F	3	4

SOLUTION:

Applying FCFS Gantt chart for the given processes becomes:

A	B	F	C	D	E
0	3	9	13	17	22	24

Now, from Gantt chart,

Process	Arrival time (AT)	Burst time (BT)	Completion time (CT)	Turnaround time TAT = CT – AT	Waiting time WT=TAT-BT
A	0	3	3	3	0
B	2	6	9	7	1
C	4	4	17	13	9
D	6	5	22	16	11
E	8	2	24	16	14
F	3	4	13	10	6
				∑ TAT=65	**∑ WT=41**

Average turn-around time = ∑(TAT) / n = 65/6 = 10.83

Average waiting time = ∑(WT) / n =41/6 = 6.83

Example 3: Consider the processes P1, P2, P3, P4 given in the below table, arrives for execution in the same order, with Arrival Time 0, and given Burst Time, find the average waiting time using the FCFS scheduling algorithm.

Process	Burst Time (ms)
P1	21
P2	3
P3	6
P4	2

Solution:

P1	P2	P3	P4

0 21 24 30 32

Now from Gantt chart,

Process	Arrival Time (ms)	Burst Time	Completion Time	Turnaround Time TAT=CT-AT	Waiting Time WT = TAT - BT
P1	0	21	21	21	0
P2	0	3	24	24	21
P3	0	6	30	30	24
P4	0	2	32	32	30

Average WT = (0+21+24+30)/4 = 18.75

ii. Shortest Job First (SJF) Scheduling

- It is a non-preemptive scheduling algorithm.
- It is also known as shortest job next (SJN)
- It is a scheduling policy that selects the waiting process with the smallest execution time to execute next.

Advantages.

- The throughput is increased because more processes can be executed in less amount of time.
- Having minimum average waiting time among all scheduling algorithms.

Disadvantages:

- It is practically infeasible as Operating System may not know burst time and therefore may not sort them.
- Longer processes will have more waiting time, eventually they'll suffer starvation.

Example 1: Consider the given processes with arrival and burst time given below. Schedule the given processes using SJF and find average waiting time (AWT) and average turnaround time(ATAT).

PROCESSES	ARRIVAL TIME	BRUST TIME
A	0	7
B	2	5
C	3	1
D	4	2
E	5	3

Solution:

According to the given question the Gantt chart for SJF given processes becomes:

A	C	D	E	B

0 7 8 10 13 18

Now,

Calculating of TAT and WT using: **TAT= CT – AT and WT = TAT - BT**

Process	Arrival time	Burst time	Completion time	Turn-around Time	Waiting time
A	0	7	7	7	0
B	2	5	18	16	11
C	3	1	8	5	4
D	4	2	10	6	4
E	5	3	13	8	5
				∑TAT=42	∑WT=24

Average turn-around time = ∑(TAT) / n = 42/5 = 10.83

Average waiting time = ∑(WT) / n =24/5 = 4.8

Example 2: Five processes P1,P2,P3,P4 and P5 having burst time 5,3,4,3,1 arrives at CPU . Find the Average TAT and Average WT using Shortest job first scheduling algorithm.

PROCESSES	BRUST TIME (ms)
P1	5
P2	3
P3	4
P4	3
P5	1

Solution

Since there is no any arrival time so it is assumed all the processes arrive initially. i.e. AT=0

The Gantt chart for the above processes becomes:

P5	P4	P2	P3	P1

0 1 4 7 11 16

Now, using Gantt chart above and formula,

TAT = (CT-AT) and WT = (TAT-BT)

Process	Arrival time	Burst time	CT	TAT	WT
P1	0	5	16	16	11
P2	0	3	7	7	4
P3	0	4	11	11	7
P4	0	3	4	4	1
P5	0	1	1	1	0
				∑TAT=39	∑WT=23

Average turn-around time = ∑(TAT) / n = 39/5 = 7.8 ms

Average waiting time = ∑(WT) / n =23/5 = 4.6 ms

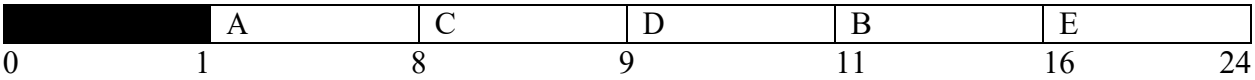
Example 3: Consider the given processes with arrival and burst time given below. Schedule the given processes using SJF and find waiting time and turn-around time.

PROCESSES	ARRIVAL TIME	BRUST TIME
A	1	7
B	2	5
C	3	1
D	4	2
E	5	8

Solution

The Gantt chart for the above processes becomes:

Here no process arrives at time= 0, so CPU stay in idle position.



Now, using Gantt chart above and formula,

TAT = (CT-AT) and WT = (TAT-BT)

Process	Arrival time	Burst time	CT	TAT	WT
A	1	7	8	7	0
B	2	5	16	14	9
C	3	1	9	6	5
D	4	2	11	7	5
E	5	8	24	19	11

iii. Shortest Remaining Time (SRT) Scheduling

- It is preemptive scheduling algorithm
- It is also called Preemptive shortest job first or Shortest remaining time next or Shortest remaining time first
- In this scheduling algorithm, the process with the smallest amount of time remaining until completion is selected to execute. Since the currently executing process is the one with the shortest amount of time remaining by definition, and since that time should onlyreduce as

execution progresses, processes will always run until they complete or a new process is added that requires a smaller amount of time.

Advantage:

- Short processes are handled very quickly.
- The system also requires very little overhead since it only makes a decision when a process completes or a new process is added.
- When a new process is added the algorithm only needs to compare the currently executing process with the new process, ignoring all other processes currently waiting to execute.

Disadvantage:

- Like shortest job first, it has the potential for process starvation.
- Long processes may be held off indefinitely if short processes are continually added.

Example 1: Consider the following processes with arrival time and burst time. Schedule them using SRT.

PROCESSES	ARRIVAL TIME(AT)	BRUST TIME (BT)
A	0	7
B	1	5
C	2	3
D	3	1
E	4	2
F	5	1

Solution:

Gantt chart for the above processes becomes:

A	B	C	D	C	C	F	E	B	A

0 1 2 3 4 5 6 7 9 13 19

Now, TAT = (CT-AT) and WT = (TAT-BT)

Process	Arrival time	Burst time	CT	TAT	WT
A	0	3	19	19	12
B	2	6	13	12	7
C	4	5	6	4	1
D	6	4	4	1	0
E	8	2	9	5	3
F	5	1	7	2	1
				∑TAT=43	**∑WT=24**

Average turn-around time = ∑(TAT) / n = 43/6 = 7.16 units
Average waiting time = ∑(WT) / n =24/6 = 4 units

iv. Priority Scheduling

- In this scheduling Priority is assigned for each process.
- Process with highest priority is executed first and so on.
- Processes with same priority are executed in FCFS manner.

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

Advantages of Priority Scheduling:

- The priority of a process can be selected based on memory requirement, time requirement or user preference. For example, a high-end game will have better graphics that means theprocess which updates the screen in a game will have higher priority so as to achieve bettergraphics performance.

Disadvantages:

- A second scheduling algorithm is required to schedule the processes which have same priority.
- In preemptive priority scheduling, a higher priority process can execute ahead of an already executing lower priority process. If lower priority process keeps waiting for higher priority processes, starvation occurs.

Priority scheduling is of both types: preemptive and non-preemptive.

a. Non-Preemptive Priority Scheduling

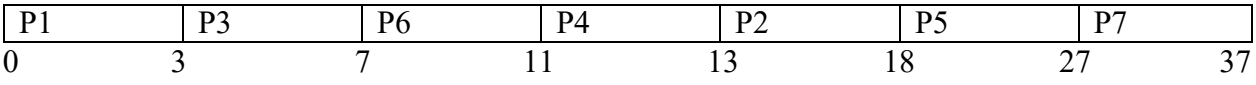
- In the Non-Preemptive Priority scheduling, The Processes are scheduled according to thepriority number assigned to them.
- Once the process gets scheduled, it will run till the completion.
- Generally, the lower the priority number, the higher is the priority of the process.

Example 1: There are 7 processes P1, P2, P3, P4, P5, P6 and P7. Their priorities, Arrival Time and burst time are given in the table. Find waiting time and response time using non preemptive priority scheduling. Also find average WT.

Process	Priority	Arrival Time	Burst Time
P1	2	0	3
P2	6	2	5
P3	3	1	4
P4	5	4	2
P5	7	6	9
P6	4	5	4
P7	10	7	10

Solution:

Gantt chart for the process is:



From the Gantt chart prepared, we can determine the completion time of every process. The turnaround time, waiting time and response time will be determined.

Turn Around Time = Completion Time - Arrival Time

Waiting Time = Turn Around Time - Burst Time

Response time = Time at which process get first response – Arrival Time

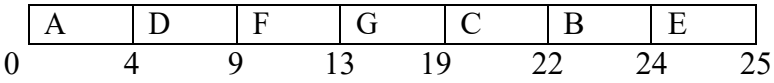
Process	Priority	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time	Response Time
P1	2	0	3	3	3	0	0
P2	6	2	5	18	16	11	13
P3	3	1	4	7	6	2	3
P4	5	4	2	13	9	7	11
P5	7	6	9	27	21	12	18
P6	4	5	4	11	6	2	7
P7	10	7	10	37	30	20	27

Average Waiting Time = (0+11+2+7+12+2+20)/7 = 54/7 units

Example 2: Consider the given processes below. Schedule the algorithm using non preemptive priority, schedule the given processes. Assume 12 as highest priority and 2 as lowest priority.

Process	AT	BT (ms)	PRIORITY
A	0	4	2(lowest)
B	1	2	4
C	2	3	6
D	3	5	10
E	4	1	3
F	5	4	12(highest)
G	6	6	9

Solution:



b. Preemptive Priority Scheduling

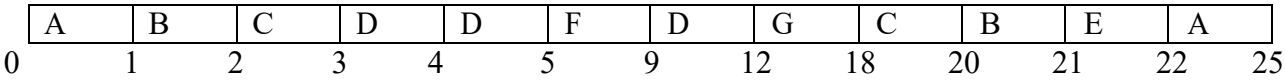
In Preemptive Priority Scheduling, at the time of arrival of a process in the ready queue, its Priority is compared with the priority of the other processes present in the ready queue as well as with the one which is being executed by the CPU at that point of time. The One with the highest priority among all the available processes will be given the CPU next.

Example 1: Consider the given processes below. Schedule the algorithm using preemptive priority algorithm and hence find AWT and ATAT. Assume 12 as highest priority and 2 as lowest priority.

Process	AT	BT (ms)	PRIORITY
A	0	4	2(lowest)
B	1	2	4
C	2	3	6
D	3	5	10
E	4	1	3
F	5	4	12(highest)
G	6	6	9

Solution:

The Gantt chart for the given processes becomes: -



Now,

Process	AT	BT	PRIORITY	CT	TAT	WT
A	0	4	2	25	25	21
B	1	2	4	21	20	18
C	2	3	6	20	18	15
D	3	5	10	12	9	4
E	4	1	3	22	18	17
F	5	4	12	9	4	0
G	6	6	9	18	12	6
					∑TAT=106	**∑WT=81**

Average turn-around time= (∑TAT=106)/7 = 15.14 ms

Average waiting time will be = (∑WT=81)/7 =11.57 ms

Example 2: There are 7 processes P1, P2, P3, P4, P5, P6 and P7 given. Their respective priorities, Arrival Times and Burst times are given in the table below. Find average WT using preemptive priority scheduling.

Process Id	Priority	Arrival Time	Burst Time
P1	2(L)	0	1
P2	6	1	7
P3	3	2	3
P4	5	3	6
P5	4	4	5
P6	10(H)	5	15
P7	9	15	8

Solution:

Gantt chart for above process is:

P1	P2	P6	P7	P2	P4	P5	P3

0 1 5 20 28 31 37 42 45

Now,

Process	Priority	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time
P1	2	0	1	1	1	0
P2	6	1	7	31	30	23
P3	3	2	3	45	43	40
P4	5	3	6	37	34	28
P5	4	4	5	42	38	33
P6	10(H)	5	15	20	15	0
P7	9	6	8	28	22	14

v. Round Robin Scheduling

- CPU is assigned to the process for a fixed amount of time.
- This fixed amount of time is called as time quantum or time slice.
- After the time quantum expires, the running process is preempted and sent to the ready queue.
- Then, the processor is assigned to the next arrived process.
- It is always preemptive scheduling algorithm

Advantages

- It gives the best performance in terms of average response time.
- It is best suited for time sharing system, client server architecture and interactive system.

Disadvantages

- It leads to starvation for processes with larger burst time as they have to repeat the cycle many times.
- Its performance heavily depends on time quantum.
- Priorities cannot be set for the processes.

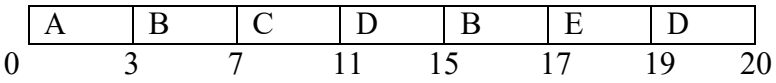
Example 1: Consider the following processes with AT and BT given. Find waiting time and turnaround time using round robin with time quantum = 4**.**

PROCESSES	ARRIVAL TIME(AT)	BRUST TIME (BT)
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Solution:

Ready Queue: ABCDBED

Gantt chart:



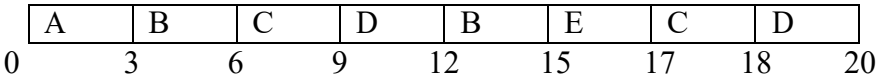
Now,

Process	Arrival time	Burst time	CT	TAT	WT
A	0	3	3	3	0
B	2	6	17	15	9
C	4	5	11	7	2
D	6	4	20	14	10
E	8	2	19	11	9

Example 2: Draw the Gantt chart of above processes using round robin with quantum 3 and 2.Solution:

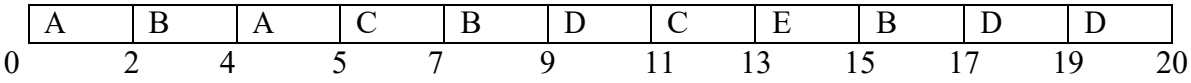
TQ=3

Ready queue: ABCDBECD



Now, TQ = 2

Ready queue: ABACBDCEBDD



vi. HRRN (Highest Response Ratio Next)

- Highest Response Ratio Next (HRNN) is one of the most optimal scheduling algorithms.
- This is a non-preemptive algorithm in which, the scheduling is done on the basis of an extra parameter called Response Ratio.
- A Response Ratio is calculated for each of the available jobs and the Job with the highest response ratio is given priority over the others.

Response Ratio is calculated by the given formula,

Response Ratio = (WT+BT)/BT

Where,

WT → Waiting Time

BT → Service Time or Burst Time

Advantages-

- It performs better than SJF Scheduling.
- It not only favors the shorter jobs but also limits the waiting time of longer jobs.

Disadvantages-

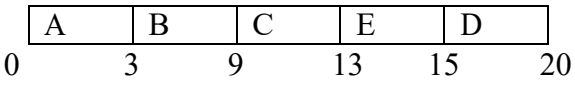
- It cannot be implemented practically. This is because burst time of the processes cannot be known in advance.

Example: Consider the following processes with AT and BT given. Find waiting time and turnaround time using HRRN.

PROCESSES	ARRIVAL TIME(AT)	BRUST TIME (BT)
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Solution:

Gantt chart:



Here A and B can be executed in FIFO. At, time = 9 C, D, E arrives so RR need to be calculated.

Response Ratio of C $(RR)_C = \frac{5+4}{4} = 2.25$

Response Ratio of D $(RR)_D = \frac{3+5}{5} = 1.6$

Response Ratio of E $(RR)_E = \frac{1+2}{2} = 1.5$

Since $(RR)_C$ is larger than other hence C runs.

Again,

Response Ratio of D $(RR)_D = \frac{7+4}{4} = 2.4$

Response Ratio of E $(RR)_E = \frac{5+2}{2} = 3.5$

Since $(RR)_E$ is larger than other hence E runs.

Now from the Gantt chart:

Process	Arrival time (AT)	Burst time (BT)	Completion time (CT)	Turnaround time TAT = CT – AT	Waiting time WT=TAT-BT
A	0	3	3	3	0
B	2	6	9	7	1
C	4	4	13	9	5
D	6	5	20	14	9
E	8	2	15	7	5

vii. Real Time Scheduling

- A real time system is one in which time plays an essential role since processing should be done within the defined time limit otherwise the system will fail.
- Generally, in this type of system, one or more physical devices external to the computer generates the real time data (stimuli) and the computer must react appropriately to them within fixed amount of time.
- The algorithm that are used to schedule such real time system periodically is known as Real Time Scheduling Algorithm.
- E.g., the computer attached with Compact Disc Player gets the bits as they come of the drive and must start converting them into music with a very tight time interval.
- If the calculation takes too long the music sounds peculiar.
- Other example includes: Auto pilot aircraft, Robot control in automated factory, Patient monitoring factory (ICU) etc.
- Real Time scheduling algorithm includes:

i. Rate Monotonic Algorithm (RM)

This is a fixed priority algorithm and follows the philosophy that higher priority is given to tasks with the higher frequencies. Likewise, the lower priority is assigned to tasks with the lower frequencies. The scheduler at any time always chooses the highest priority task for execution. By approximating toa reliable degree, the execution times and the time that it takes for system handling functions, the behavior of the system can be determined apriority.

ii. Earliest Deadline First Algorithm (EDF)

This algorithm takes the approach that tasks with the closest deadlines should be meted out the highest priorities. Likewise, the task with the latest deadline has the lowest priority. Due to this approach, the processor idle time is zero and so 100% utilization can be achieved.

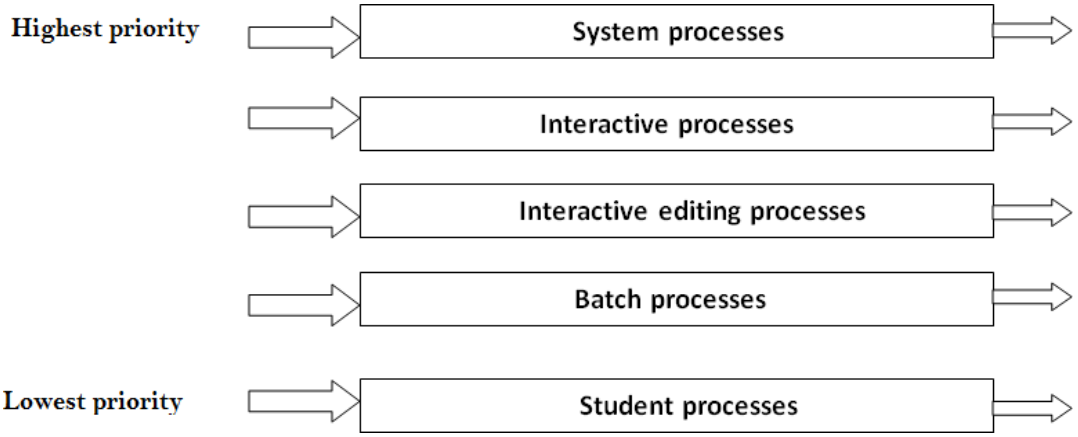
viii. Multilevel Queue

- In the multilevel queue scheduling algorithm partition the ready queue has divided into seven separate queues. Based on some priority of the process; like memory size, process priority, or process type these processes are permanently assigned to one queue.
- Each queue has its own scheduling algorithm. For example, some queues are used for the foreground process and some for the background process.

- The foreground queue can be scheduled by using a round-robin algorithm while the background queue is scheduled by a first come first serve algorithm.
- It is said that there will be scheduled between the queues which are easily implemented as a fixed- priority preemptive scheduling.

Let us take an example of a multilevel queue scheduling algorithm with five queues:

- System process
- Interactive processes
- Interactive editing processes
- Batch processes
- Student processes



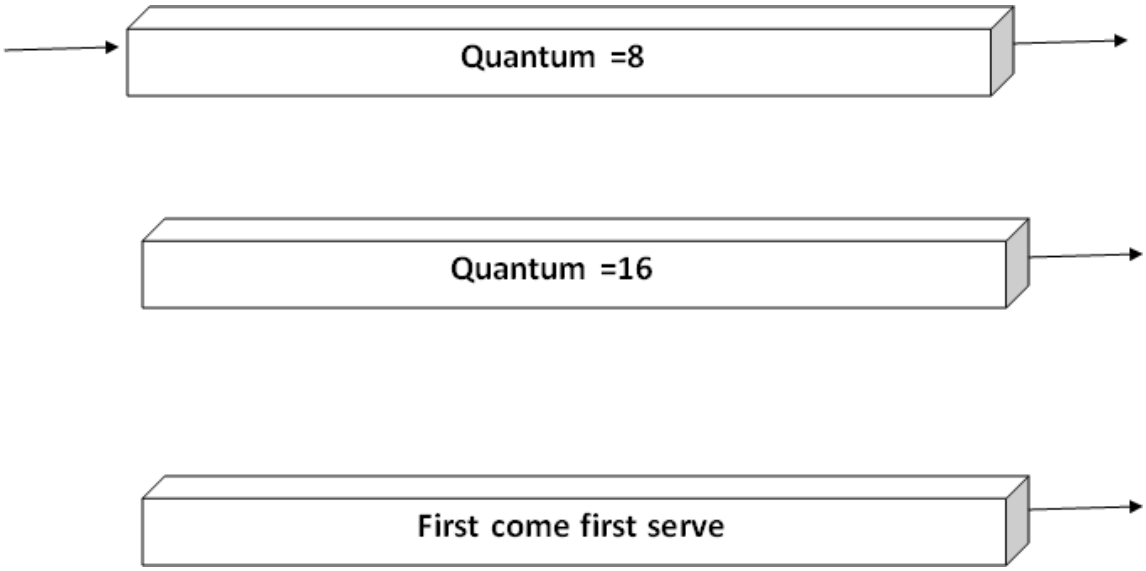
ix. Multilevel Feedback Queue

- In multilevel queue scheduling algorithm processes are permanently stored in one queue in the system and do not move between the queue.
- There is some separate queue for foreground or background processes but the processes do not move from one queue to another queue and these processes do not change their foreground or background nature, this type of arrangement has the advantage of low scheduling but it is inflexible in nature.
- Multilevel feedback queue scheduling it allows a process to move between the queue. This the process is separate with different CPU burst time.
- If a process uses too much CPU time, then it will be moved to the lowest priority queue.
- This idea leaves I/O bound and interactive processes in the higher priority queue.
- Similarly, the process which waits too long in a lower priority queue may be moved to a higher priority queue. This form of aging prevents starvation.

The multilevel feedback queue scheduler has the following parameters:

- The number of queues in the system.
- The scheduling algorithm for each queue in the system.
- The method used to determine when the process is upgraded to a higher-priority queue.
- The method used to determine when to demote a queue to a lower - priority queue.

- The method used to determine which process will enter in queue and when that process needs service.



Chapter 5 Memory Management

Introduction

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.

The part of OS that manages the memory hierarchy is called memory manager.

- Keeps track of used/unused part of memory.
- Allocate/de-allocate memory to processes.
- Manages swapping between main memory and disk.
- Protection and sharing of memory

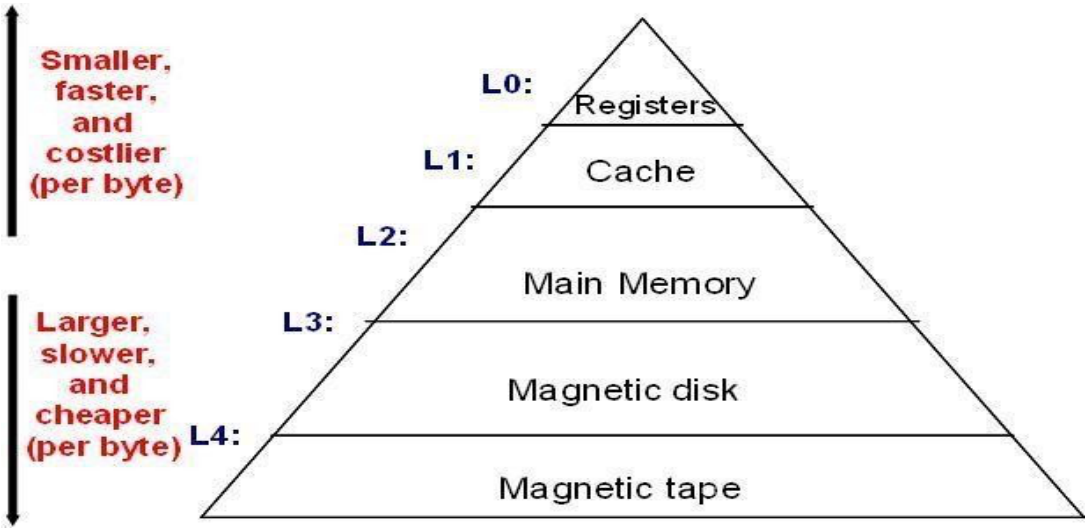
Memory Organization

Memory unit is the collection of storage units or devices together. The memory unit stores the binary information in the form of bits. Generally, memory/storage is classified into 2 categories:

- **Volatile Memory:** This loses its data, when power is switched off.
- **Non-Volatile Memory:** This is a permanent storage and does not lose any data when power is switched off.

Memory hierarchy

The Memory Hierarchy consists of all storage devices employed in a computer system from slow but high capacity Auxiliary memory to relatively faster Main memory to an even smaller and faster Cache memory" accessible to the high speed processing logic. Memory hierarchy separates computer storage into a hierarchy based on response time.



i. **Internal register** is for holding the temporary results and variables. Accessing data from these registers is the fastest way of accessing memory.

ii. **Cache** is used by the CPU for memory which is being accessed over and over again. Instead of pulling it every time from the main memory, it is put in cache for fast access. It is also a smaller memory, however, larger than internal register.

iii. **Main memory or RAM (Random Access Memory):** It is a type of the computer memory and is a hardware component. It can be increased provided the operating system can handle it.

iv. **Hard disk:** A hard disk is a hardware component in a computer. Data is kept permanently in this memory. Memory from hard disk is not directly accessed by the CPU, hence it is slower. As compared with RAM, hard disk is cheaper per bit.

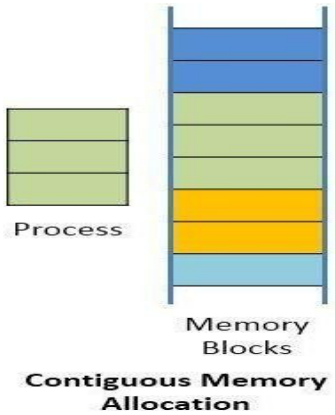
v. **Magnetic tape:** Magnetic tape memory is usually used for backing up large data. When the system needs to access a tape, it is first mounted to access the data. When the data is accessed, it is then un-mounted. The memory access time is slower in magnetic tape and it usually takes few minutes to access a tape.

Storage Allocation

The process of allocating the block of memory to a process is known as storage allocation. Memory is a large array of bytes, where each byte has its own address. The memory allocation can be classified into two methods:

i. Contiguous Memory Allocation

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



Contiguous Memory Allocation

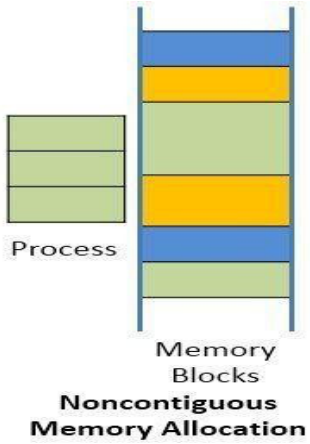
The contiguous memory allocation can be achieved by dividing the memory into the fixed- sized **partition** and allocate each partition to a single process only. But this will cause the degree

Of multiprogramming, bounding to the number of fixed partitions done in the memory. The contiguous memory allocation also leads to the **internal fragmentation**. Like, if a fixed sized memory block allocated to a process is slightly larger than its requirement then the left over memory space in the block is called internal fragmentation. When the process residing in the partition terminates the partition becomes available for another process.

In the variable partitioning scheme, the operating system maintains a **table** which indicates, which partition of the memory is free and which occupied by the processes. The contiguous memory allocation fastens the execution of a process by reducing the overheads of address translation.

ii. Non-contiguous memory

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



Noncontiguous Memory Allocation

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

Partitioning

Partitioning is the process of breaking the main memory into number of blocks.

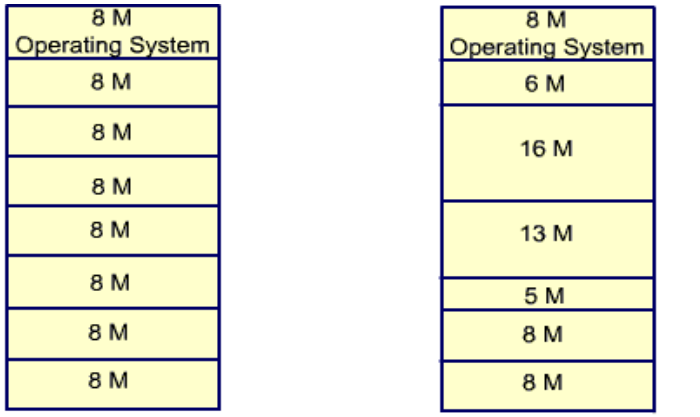
Partitioning is of two types:

1. Fixed/Static Partition Allocation
2. Variable/Dynamic Partition Allocation

Fixed/Static Partition Allocation

The simplest way to manage the memory available for the processes is to partition it into regions with fixed boundaries.

The partition may be equal sized or unequal sized

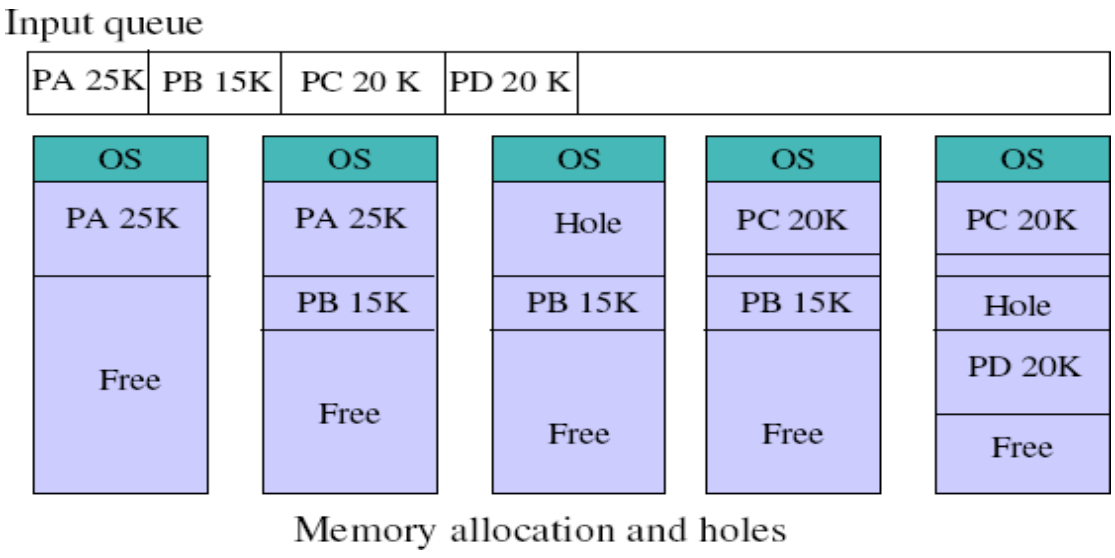


(a) Equal Size Partitions (b) Un-equal Size Partitions

- A process whose size is less than or equal to the partition size can be loaded into that partition
- Main memory utilization is extremely inefficient.
- Any program, no matter how small, occupies an entire partition. This phenomenon, in which there is wasted space internal to a partition due to the fact that the block of data loaded is smaller than the partition, is referred to as **internal fragmentation.**
- Inefficient use of Memory due to internal fragmentation.

Dynamic partitioning

- To overcome some of the difficulties with fixed partitioning, an approach known as dynamic partitioning was developed
- 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
- Holes (pieces of free memory) might be created during dynamic partitioning.
- Given a new process, the OS must decide which hole to use for new process



Memory allocation and holes

- Frequent loading and unloading of programs causes free space to be broken into little pieces
- Eventually it leads to a situation in which there are a lot of small holes in memory. As time goes on memory becomes more and more fragmented, and memory utilization declines. This phenomenon is referred to as **external fragmentation**.
- External fragmentation exists when there is enough memory to fit a process in memory, but the space is not contiguous.

Fragmentation

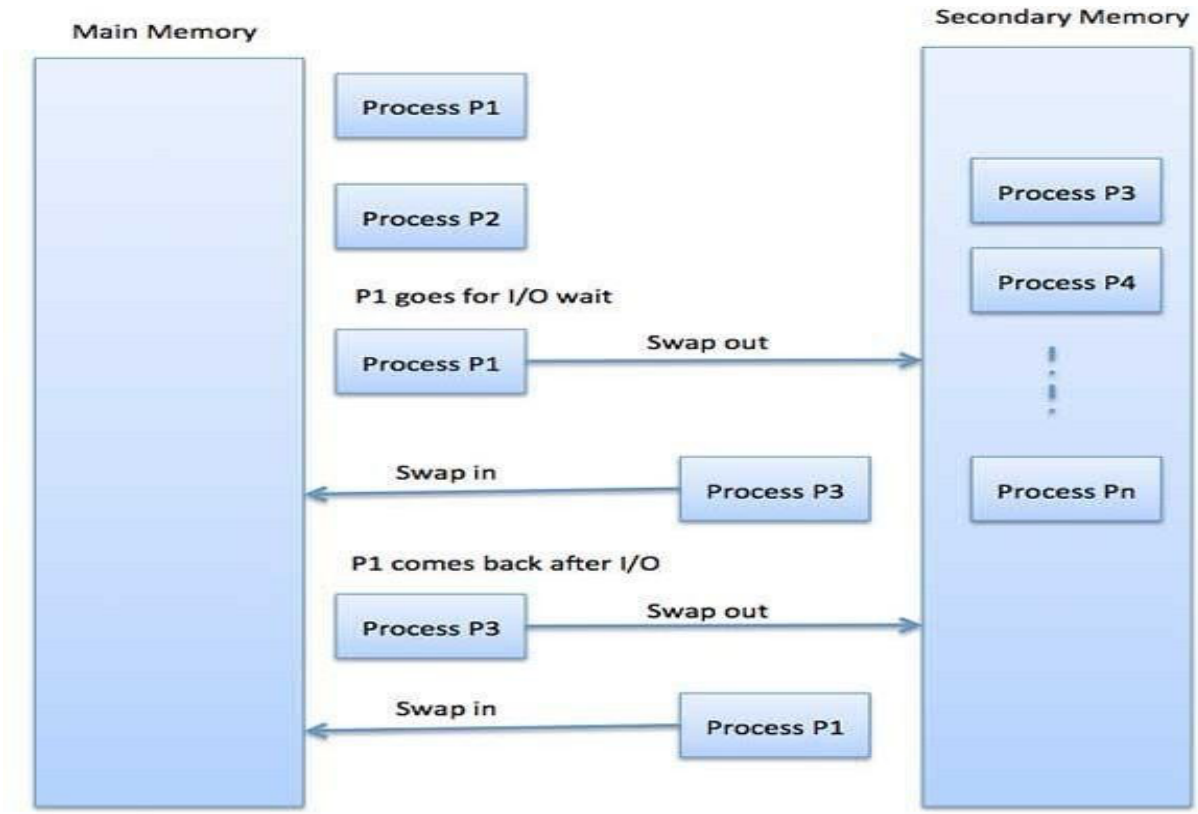
- Fragmentation refers to the condition of a disk in which files are divided into pieces scattered around the disk.
- Fragmentation occurs naturally when you use a disk frequently, creating, deleting, and modifying files.
- At some point, the operating system needs to store parts of a file in noncontiguous clusters.
- This is entirely invisible to users, but it can slow down the speed at which data isaccessed because the disk drive must search through different parts of the disk to put together a single file.
- Fragmentation is of two types: Internal and External fragmentation.

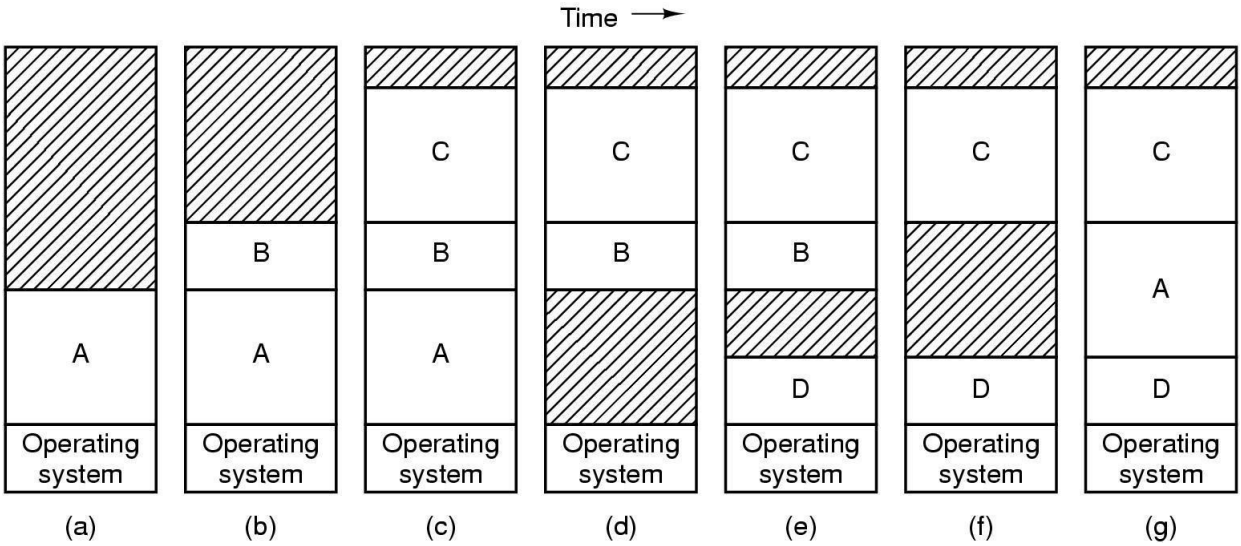
Internal vs. External fragmentation

- External fragmentation exists when there is enough memory to fit a process in memory but the space is not contiguous. Internal fragmentation exists when memory internal to a partition is wasted.
- External fragmentation exists between the blocks of allocated memory and internal fragmentation is within the allocated block.
- Internal Fragmentation occurs when a fixed size memory allocation technique is used. External fragmentation occurs when a dynamic memory allocation technique is used.

Swapping

- Swapping is a mechanism in which a process can be swapped temporarily out of main memory (or move) to secondary storage (disk) and make that memory available to other processes.
- Backing store is a usually a hard disk drive or any other secondary storage which fast in access and large enough to accommodate copies of all memory images for all users. It must be capable of providing direct access to these memory images
- At some later time, the system swaps back the process from the secondary storage tomain memory.
- Similar to round-robin CPU-scheduling algorithm, when a quantum expires, the memory manager will swap out that process to swap another process into the memory space that has been freed
- 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.





Relocation and protection

Relocation

- Programmers typically do not know in advance which other programs will be resident in main memory at the time of execution of their Program
- Active processes need to be able to be swapped in and out of main memory in order to maximize processor utilization

- Specifying that a process must be placed in the same memory region when it is swapped back in would be limiting
    - ✓ may need to relocate the process to a different area of memory

Protection

- Processes need to acquire permission to reference memory locations for reading or writing purposes
- Location of a program in main memory is unpredictable
- Memory references generated by a process must be checked at run time
- Mechanisms that support relocation also support protection

Logical and Physical Memory

Logical memory enables the user to use large amount of memory to store data. It defines way to organize the physical memory such as RAM and cache. This enables the Operating System to arrange memory into a logical manner such as assigning a logical address.

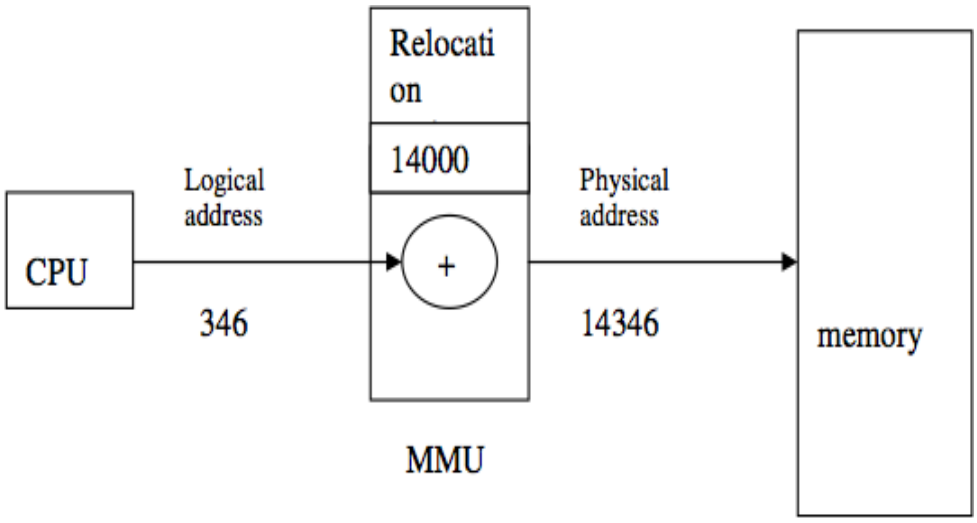
Physical memory refers to the actual RAM of the system, which usually takes the form of cards attached onto the motherboard. Also called primary memory, it is the only storage type directly accessibly to the CPU and holds the instructions of programs to execute. Physical memory is linearly addressable; memory addresses increase in a linear fashion and each byte is directly addressable.

Logical address

An address generated by the CPU is commonly known as Logical address. When we give the problem to the computer then our computer passes the problem to the processor through logical address, which we are not seen, this address called logical address. And the set of all logical addresses generated by a program is known as logical address Space.

Physical address

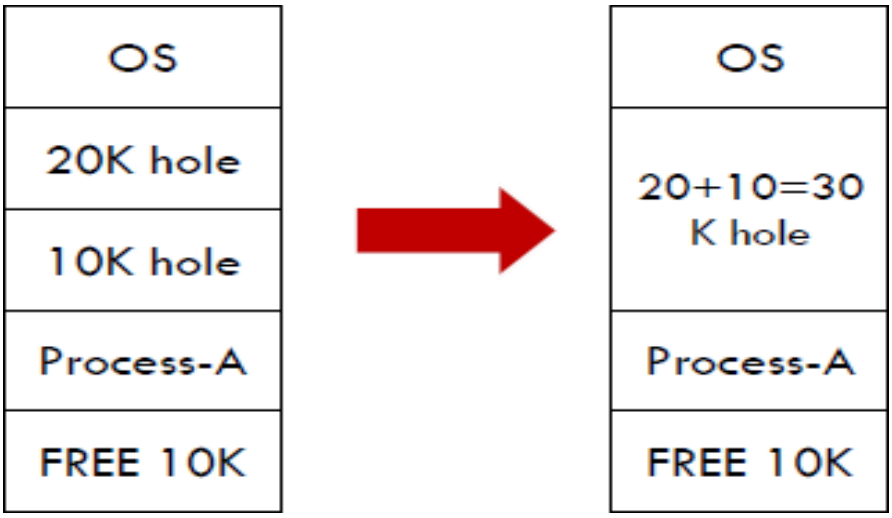
When our processor create process and solve our problem then we store data in secondary memory through address called physical address. In simple word the address seen by the memory unit that is one loaded into the memory address register of the memory is commonly refereed as the Physical Address. And the set of all physical addresses corresponding to these logical addresses is physical address space.



Coalescing and Compaction

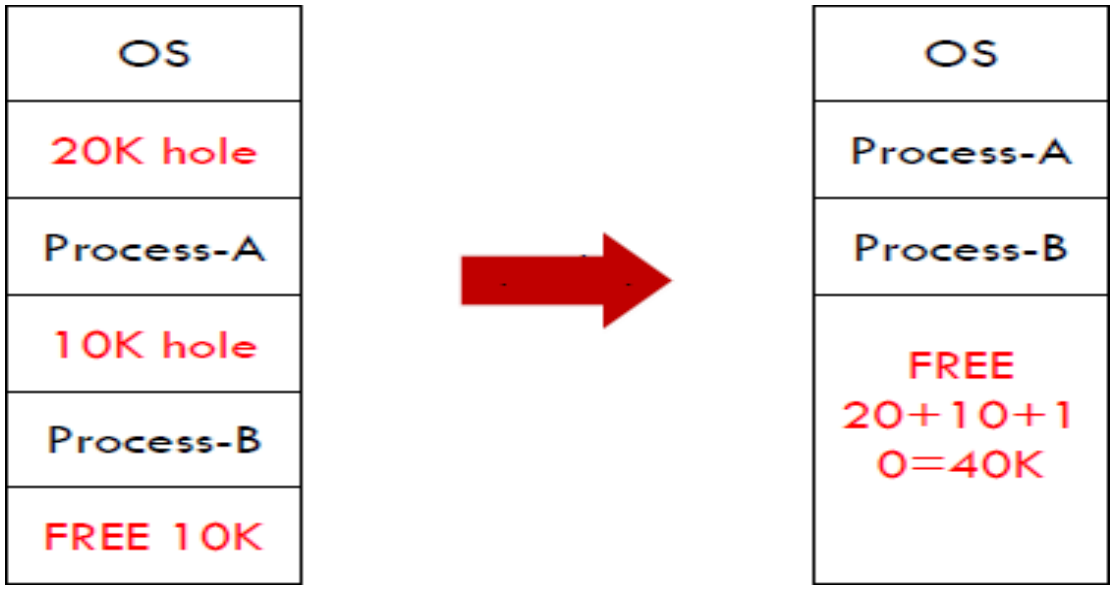
Coalescing

The process of merging two adjacent holes to form a single larger hole is called coalescing.



Compaction

Even when holes are coalesced, no individual hole may be large enough to hold the job, although the sum of holes is larger than the storage required for a process. It is possible to combine all the holes into one big one by moving all the processes downward as far as possible; this technique is called memory compaction.



MEMORYALLOCATION POLICIES

First-Fit

The memory manager allocates the first hole that is big enough. It stops the searching as soon as it finds a free hole that is large enough.

Advantages: It is a fast algorithm because it searches as little as possible.

Disadvantages: Not good in terms of storage utilization

Best-Fit

- Allocate the smallest hole that is big enough for a process
- The OS must search the entire list of holes or store the list sorted by size of the hole
- Best fit try to find a hole that is close to the actual size needed.

Advantages: more storage utilization than first fit.

Disadvantages: slower than first fit because it requires searching whole list at time.

Worst - Fit

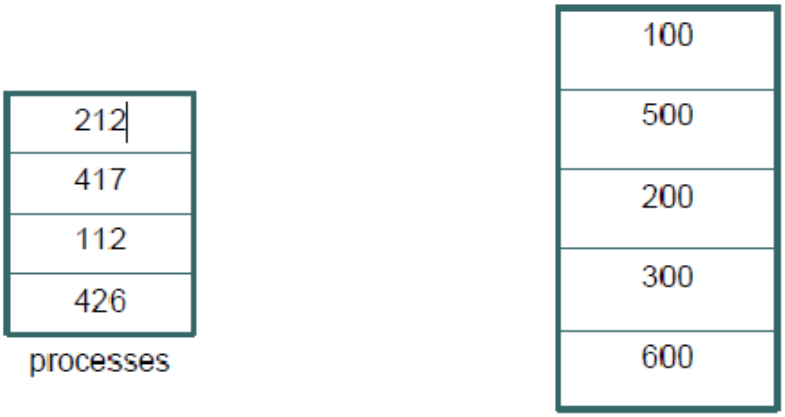
- Allocate the largest hole.
- It searches the entire list, and takes the largest hole, rather than creating a tiny hole
- It produces the largest leftover hole, which may be more useful.

Advantages: some time it has more storage utilization than first fit and best fit.

Disadvantages: not good for both performance and utilization.

Q: Given the memory partitions of 100K, 500K, 200K, 300K and 600K (in order), how would each of the First-fit, Best-fit, and Worst-fit algorithms place processes of 212K, 417K,112K, and 426K (in order)?

Which algorithm makes the most efficient use of memory?



- First-fit: search the list of available memory and allocate the first block that is big enough

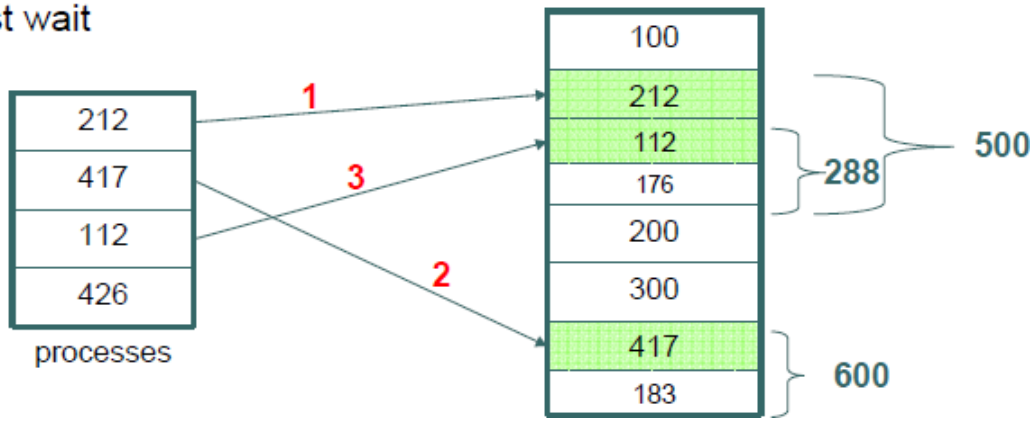
Processes placement:

212 K → 500 K partition

417 K → 600 K partition

112 K → 288 K partition (New partition 288 K = 500 K-212 K)

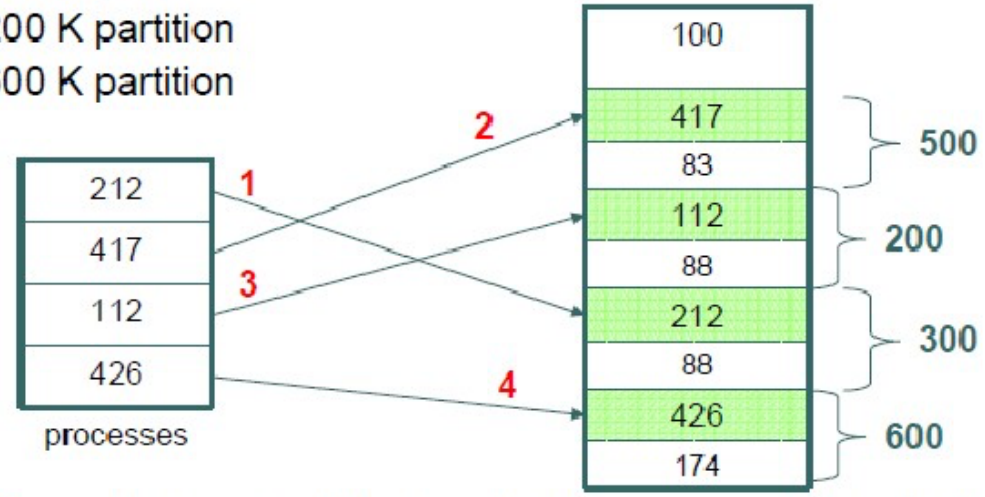
426 K must wait



- Best-fit: search the entire list of available memory and allocate the smallest block that is big enough

Processes placement:

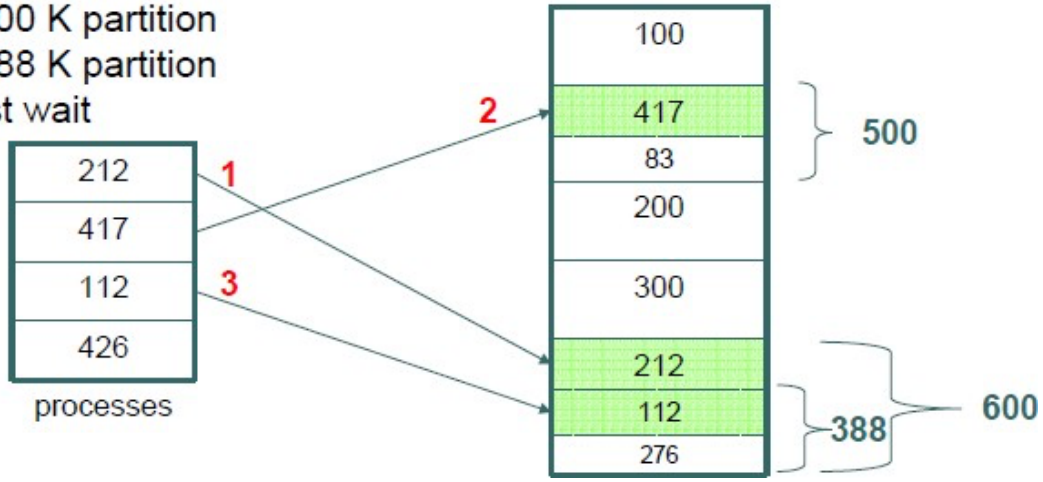
212 K → 300 K partition
417 K → 500 K partition
112 K → 200 K partition
426 K → 600 K partition



- Worst-fit: search the entire list of available memory and allocate the largest block. The justification for this scheme is that the leftover block produced would be larger and potentially more useful than that produced by the best-fit approach

Processes placement:

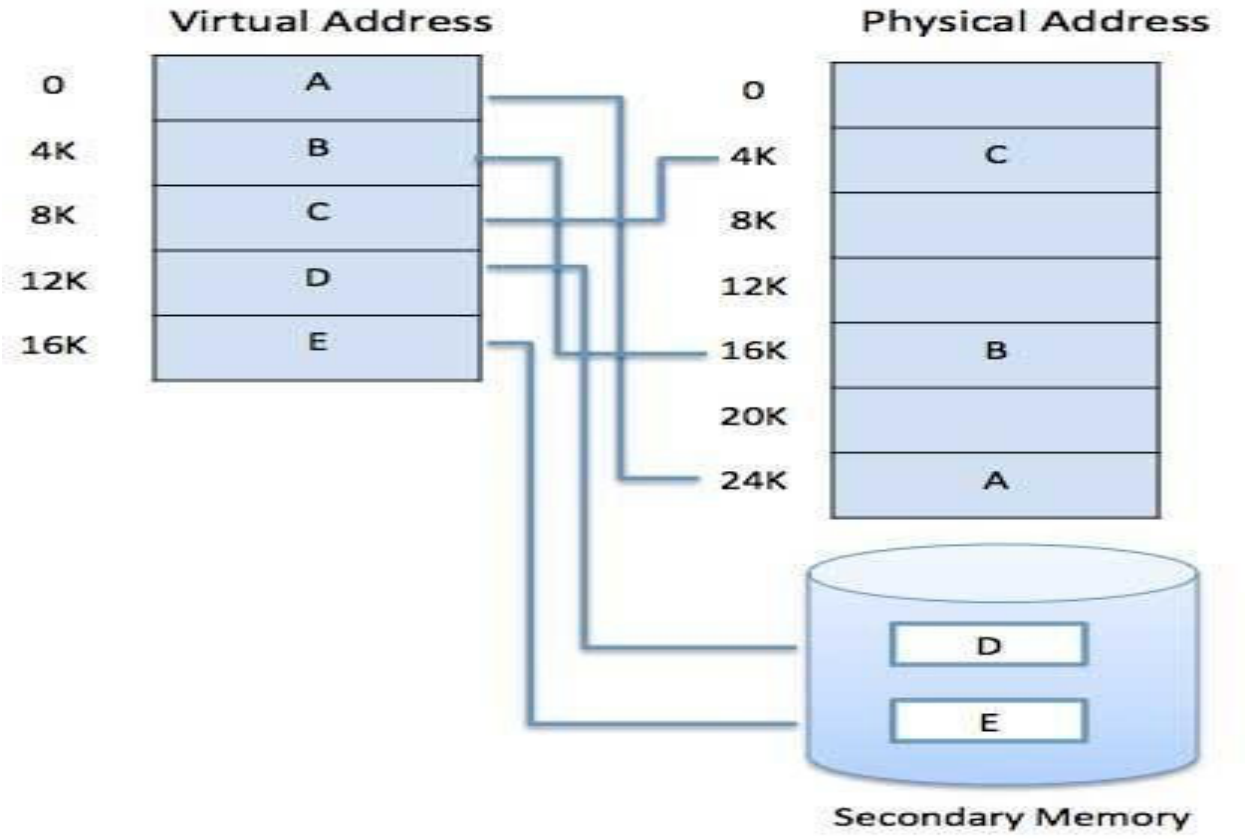
212 K → 600 K partition
417 K → 500 K partition
112 K → 388 K partition
426 K must wait



Virtual memory

- Virtual memory is a memory management technique that allows programs larger than physical memory to be executed.

- A computer can address more memory than the amount physically installed on the system. This extra memory is actually called **virtual memory.**
- The run time mapping from virtual address to physical address is done by hardware device called memory management unit (MMU).
- Virtual Memory is a space where large programs can store themselves in form of pages while their execution and only the required pages or portions of processes are loaded into the main memory



- Virtual memory takes the advantage of the fact that processes typically do not use all the allocated memory. In other words, most processes never need all their pages at once.

PAGING

- **Paging** is a memory management scheme by which a computer stores and retrieves data from secondary storage for use in main memory.
- Paging is a process of reading data from, and writing data to the secondary storage.
- Partition memory into small equal fixed-size chunks and divide each process into the same size chunks.
- The chunks of a process are called ***pages.***
- The chunks of memory are called ***frames.***

- The basic idea behind paging is to only keep those part of the processes in memory that are actually being used
- Paging is non-contiguous memory allocation technique.
- The basic idea behind paging is to only keep those part of the processes in memory that are actually being used
- Paging:
    - ❖ divides processes to a fixed sized **page**
    - ❖ Then selectively allocates pages to **frames** in memory, and
    - ❖ manages (moves, removes, reallocates) pages in memory
  - The logical memory of the process is still contiguous but the pages need not be allocated contiguously in memory
  - By dividing the memory into fixed sized pages, we can eliminate external fragmentation
  - However, paging dose not eliminate internal fragmentation (e.g 4KB frame size and 15 KBof process)
  - **Summary**: breaking process memory into fixed sized pages and map them to a physical frame of memory

OS needs to do two things

1. Mapping the pages to frame, maintain a ***page table***
2. Translate the virtual address (contiguous) into physical address (not so easy as the physical address for a process is scattered all over the memory)

Address generated by CPU (logical address) is divided into:

- Page number **p** is an index into a *page table* that contains base address of each page in physical memory.
- Page offset **d** is a *displacement*, combined with base address to define the physical memory address that is sent to the memory unit.

Page Fault

- When the page (data) requested by a program is not available in the memory, it is called as a page fault. This usually results in the application being shut down.
- A page is a fixed length memory block used as a transferring unit between physical memory and an external storage. A page fault occurs when a program accesses a page that has been mapped in address space, but has not been loaded in the physical memory.

Page hit

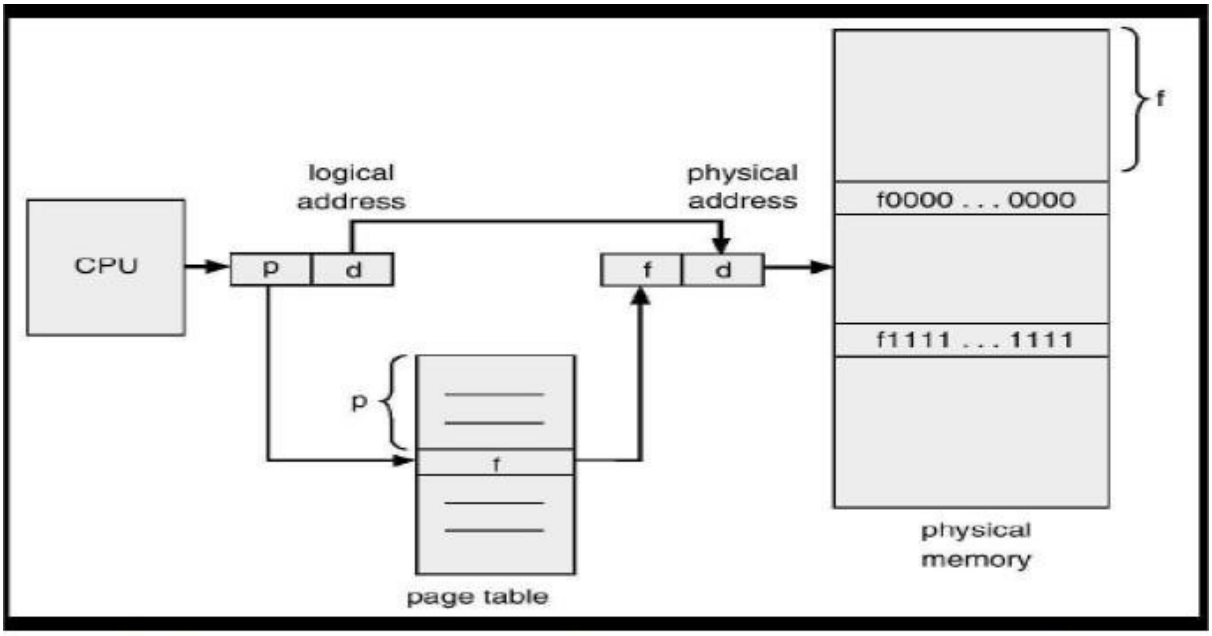
Whenever a process refers to a page that is present in memory, such condition is known as **page hit.**

Page table

A page table is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses

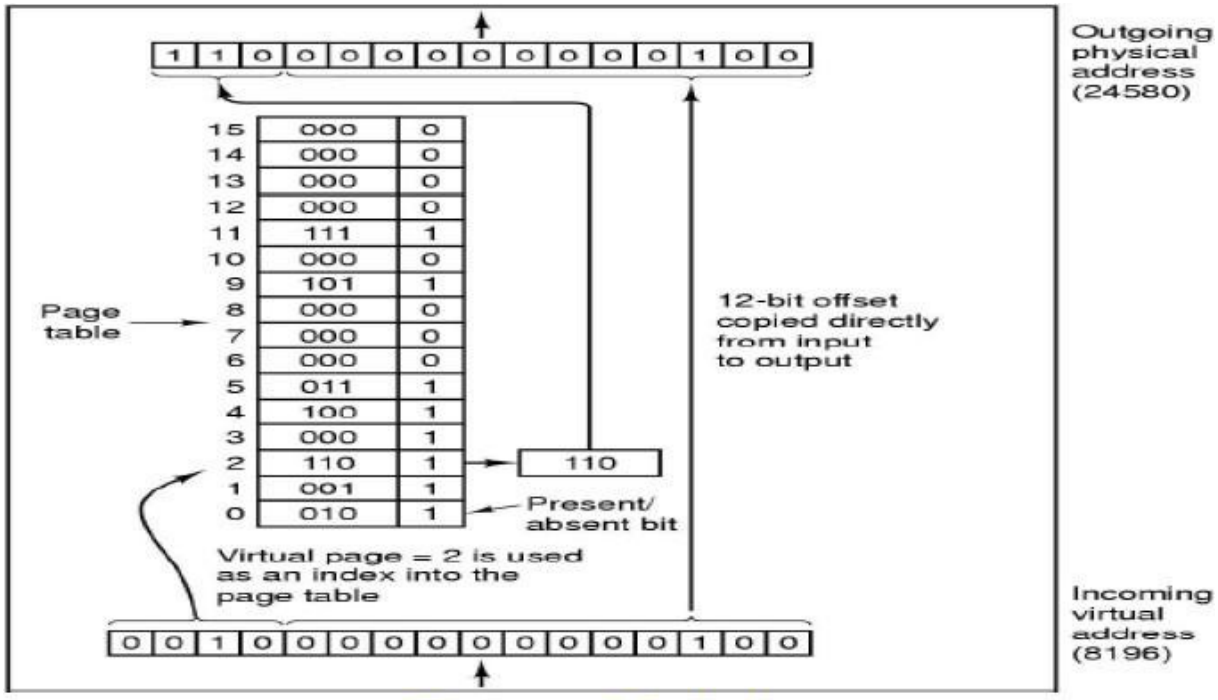
Frame

Frame is a fixed sized block in physical memory space.



Address Translation Example

Fig: Address translation process in paging



Page Tables

Fig: Example of Address translation process in paging

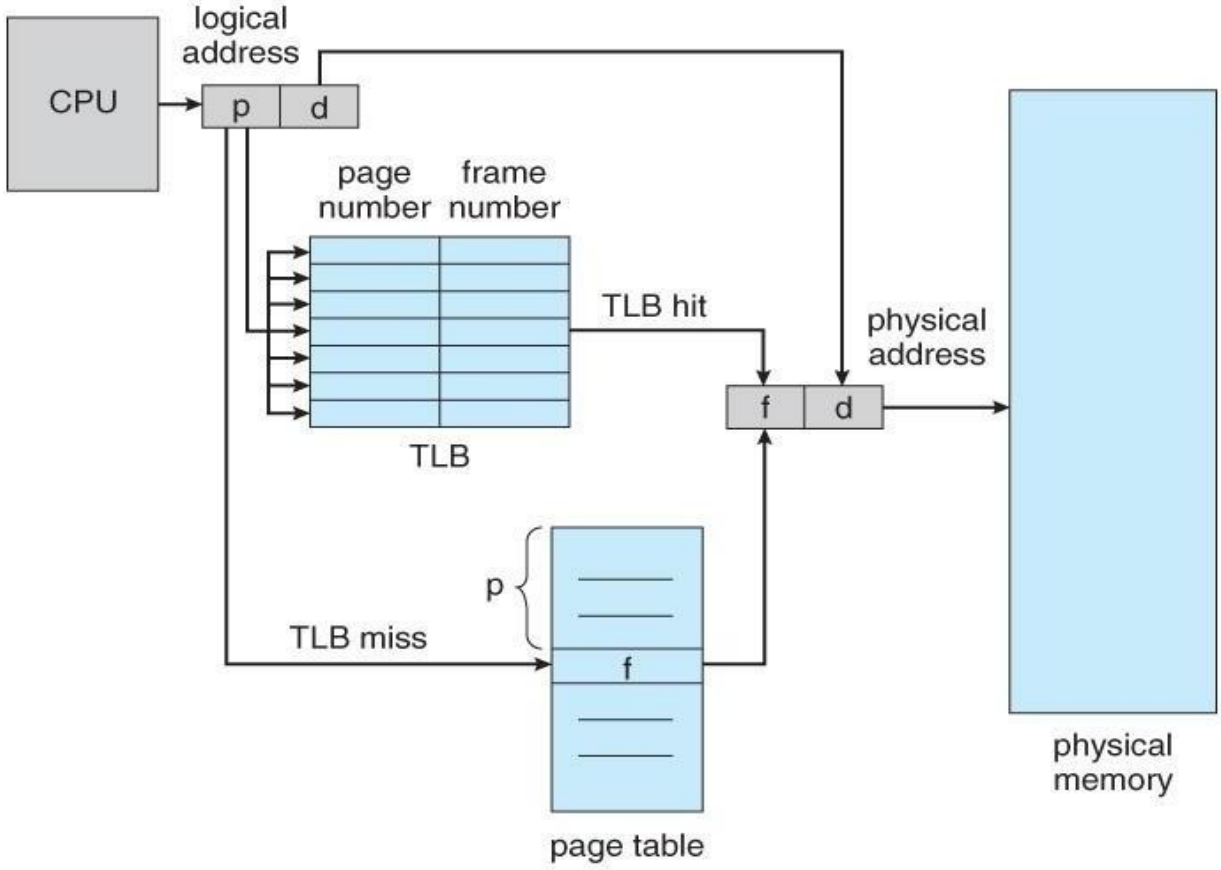
Advantages and Disadvantages of paging

- Paging reduces external fragmentation, but still suffer from internal fragmentation.

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

Translation look-aside buffer (TLB

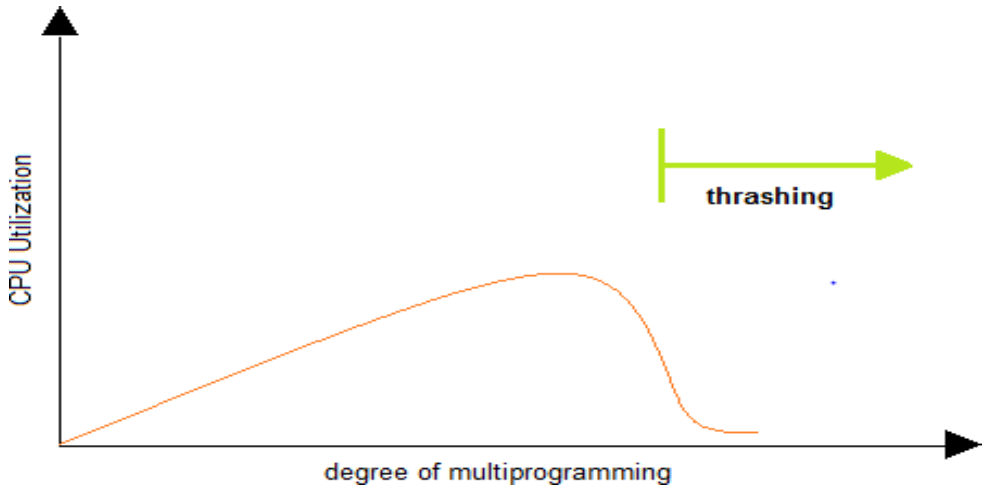
- A translation look-aside buffer (TLB) is a memory cache that stores recent translations of virtual memory to physical addresses for faster retrieval.
- When a virtual memory address is referenced by a program, the search starts in the CPU. First, instruction caches are checked.
- If the required memory is not in these very fast caches, the system has to look up the memory's physical address.
- At this point, TLB is checked for a quick reference to the location in physical memory.
- When an address is searched in the TLB and not found, the physical memory must be searched with a memory page crawl operation. As virtual memory addresses are translated, values referenced are added to TLB.
- When a value can be retrieved from TLB, speed is enhanced because the memory address is stored in the TLB on processor. Most processors include TLBs to increase the speed of virtual memory operations through the inherent latency-reducing proximity as well as the high-running frequencies of current CPU's



Thrashing

Thrashing is term referred to a high paging activity. A process is thrashing if it is spending more of its time paging than actually executing.

Consider a process that currently needs some more frames. Currently all the frames are occupied. So, it will have to swap with certain pages; however, since all the pages are actively being used, there would be need for an immediate page replacement after this. Such a scenario is called thrashing. It results in serious performance problems.



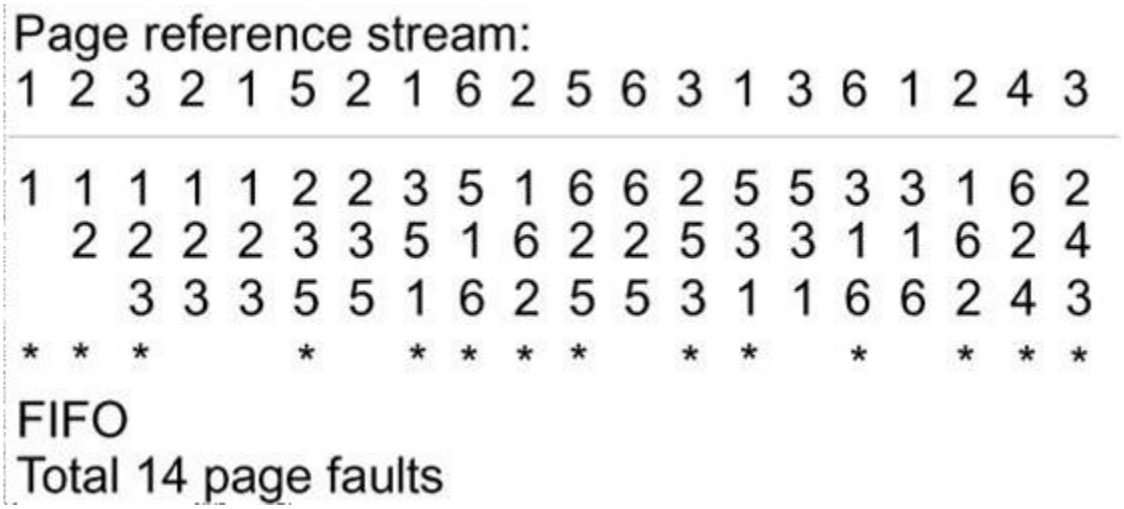
Page replacement algorithm

In operating systems, that use paging for memory management, page replacement algorithm are needed to decide which page needed to be replaced when new page comes in. Whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace.

Some of the algorithm used for page replacement algorithm are:

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.

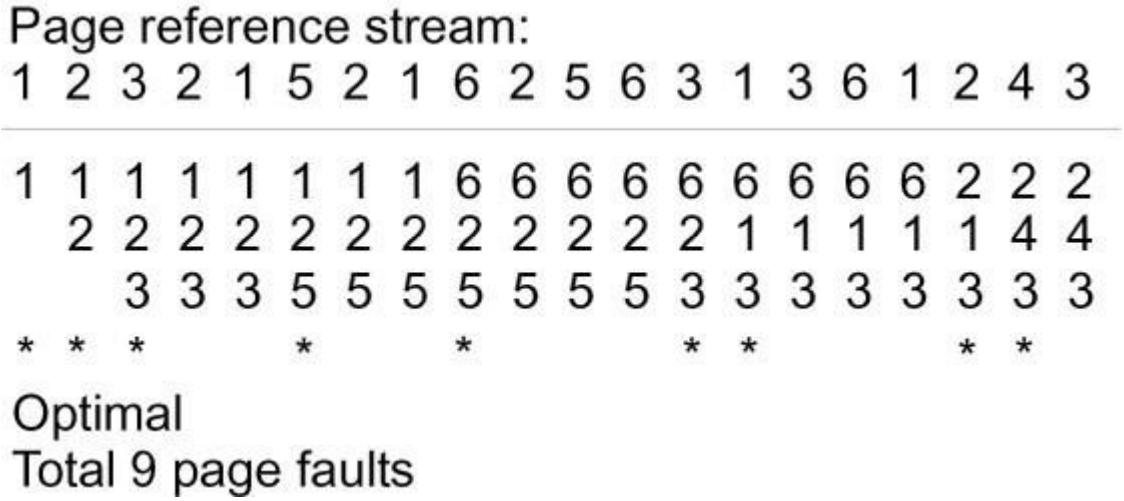


Belady's Anomaly

Normally, one would expect that with the total number of frames increasing, the number of page fault decreases. However, for FIFO, there are cases where this generalization fails. This is called **Belady's Anomaly.**

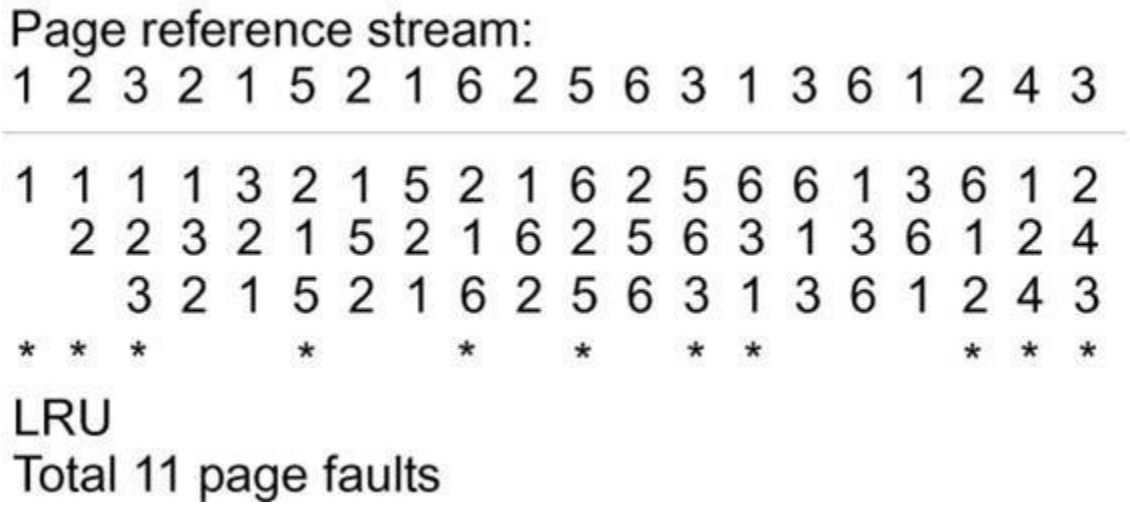
Optimal Page algorithm

- 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.



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.



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.

The Not Recently Used

The not recently used (NRU) page replacement algorithm is an algorithm that favours keeping pages in memory that have been recently used. This algorithm works on the following principle: when a page is referenced, a referenced bit is set for that page, marking it as referenced. Similarly, when a page is modified (written to), a modified bit is set. The setting of the bits is usually done by the hardware, although it is possible to do so on the software level as well.

At a certain fixed time, interval, a timer interrupt triggers and clears the referenced bit of all the pages, so only pages referenced within the current timer interval are marked with a referenced bit. When a page needs to be replaced, the operating system divides the pages into four classes:

3. referenced, modified

2. referenced, not modified

1. Not referenced, modified

0. Not referenced, not modified

Although it does not seem possible for a page to be modified yet not referenced, this happens when a class 3 page has its referenced bit cleared by the timer interrupt. The NRU algorithm picks a random page from the lowest category for removal. So out of the above four page categories, the NRU algorithm will replace a not-referenced, not-modified page if such a page exists. Note that this algorithm implies that a modified but not-referenced (within the last timer interval) page is less important than a not-modified page that is intensely referenced.

Random

Random replacement algorithm replaces a random page in memory. This eliminates the overhead cost of tracking page references. Usually it fares better than FIFO, and for looping memory references it is better than LRU, although generally LRU performs better in practice

Clock

Clock is a more efficient version of FIFO than Second-chance because pages don't have to be constantly pushed to the back of the list, but it performs the same general function as Second-Chance. The clock algorithm keeps a circular list of pages in memory, with the "hand" (iterator) pointing to the last examined page frame in the list. When a page fault occurs and no empty frames exist, then the R (referenced) bit is inspected at the hand's location. If R is 0, the new page is put in place of the page the "hand" points to. Otherwise, the R bit is cleared, then the clock hand is incremented and the process is repeated until a page is replaced.

Second-chance

A modified form of the FIFO page replacement algorithm, known as the Second-chance page replacement algorithm, fares relatively better than FIFO at little cost for the improvement. It works by looking at the front of the queue as FIFO does, but instead of immediately paging out that page, it checks to see if its referenced bit is set. If it is not set, the page is swapped out. Otherwise, the referenced bit is cleared, the page is inserted at the back of the queue (as if it were a new page) and this process is repeated. This can also be thought of as a circular queue. If all the pages have their referenced bit set, on the second encounter of the first page in the list, that page will be swapped out, as it now has its referenced bit cleared. If all the pages have their reference bit cleared, then second chance algorithm degenerates into pure FIFO.

As its name suggests, Second-chance gives every page a "second-chance" – an old page that has been referenced is probably in use, and should not be swapped out over a new page that has not been referenced.

Working set page replacement algorithm

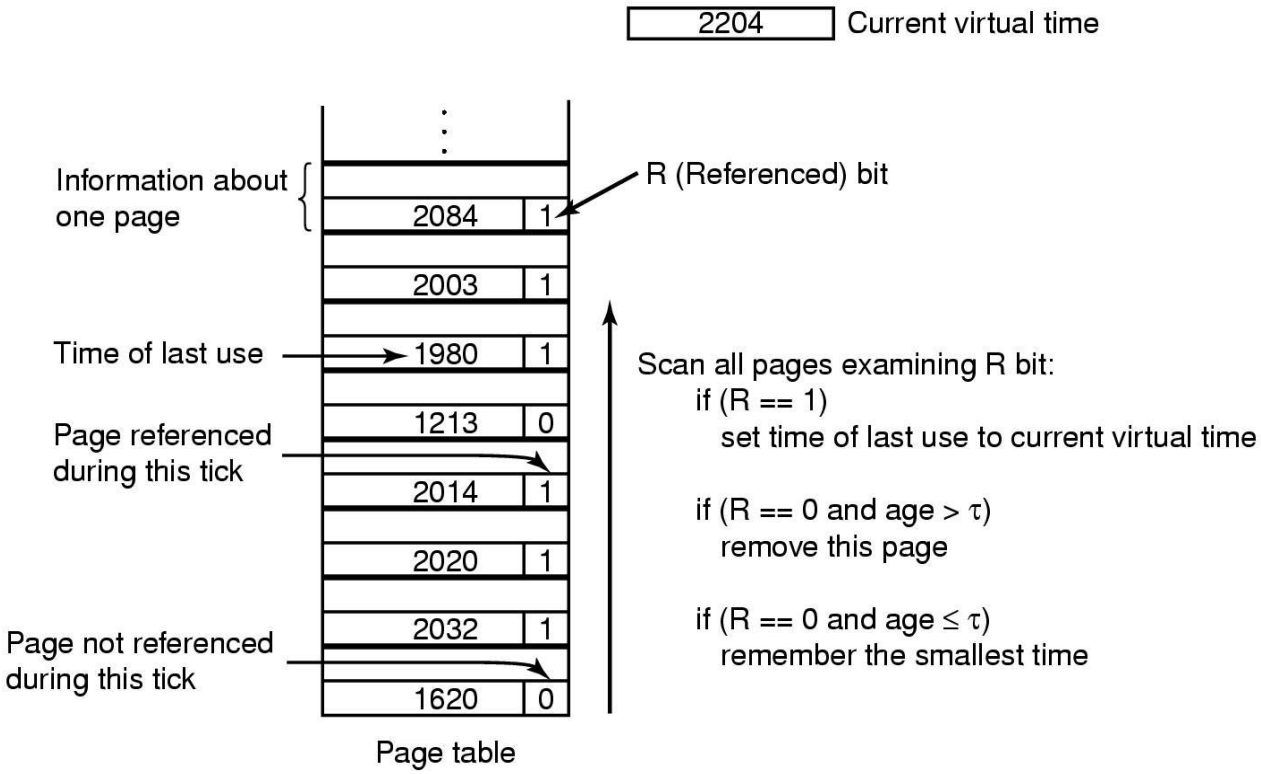
The algorithm works as follows:

The hardware is assumed to set the R and M bits. Similarly, a periodic clock interrupt is assumed to cause software to run that clears the Referenced bit on every clock tick. On every page fault, the page table is scanned to look for a suitable page to evict.

As each entry is processed, the R bit is examined. If it is 1, the current virtual time is written into the Time of last use field in the page table, indicating that the page was in use at the time the fault occurred. Since the page has been referenced during the current clock tick, it is clearly in the working set and is not a candidate for removal (t is assumed to span multiple clock ticks).

If R is 0, the page has not been referenced during the current clock tick and may be a candidate for removal. To see whether or not it should be removed, its age, that is, the current virtual time minus its Time of last use is computed and compared to t. If the age is greater than t, the page is no longer in the working set. It is reclaimed and the new page loaded here. The scan continues updating the remaining entries, however.

However, if R is 0 but the age is less than or equal to t, the page is still in the working set. The page is temporarily spared, but the page with the greatest age (smallest value of Time of last use) is noted. If the entire table is scanned without finding a candidate to evict, that means that all pages are in the working set. In that case, if one or more pages with R = 0 were found, the one with the greatest age is evicted. In the worst case, all pages have been referenced during the current clock tick (and thus all have R = 1), so one is chosen at random for removal, preferably a clean page, if one exists.



WS Clock page replacement algorithm

- ✓ An implementation of the working set algorithm
- ✓ All pages are kept in a circular list (ring)
- ✓ As pages are added, they go into the ring
- ✓ The ―clock hand‖ advances around the ring
- ✓ Each entry contains ―time of last use‖
- ✓ Upon a page fault...
  - ❖ If Reference Bit = 1...
    - Page is in use now. Do not evict.
    - Clear the Referenced Bit.
    - Update the ―time of last use‖ field.
  - ❖ If Reference Bit = 0

If the age of the page is less than T...

- Page is in the working set.
- Advance the hand and keep looking

If the age of the page is greater than T...

- If page is clean
  - Reclaim the frame and we are done!
- If page is dirty
  - Schedule a write for the page
  - Advance the hand and keep looking

Segmentation

- Segmentation is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions. Each segment is actually a different logical address space of the program.
- When a process is to be executed, its corresponding segmentation are loaded into non-contiguous memory though every segment is loaded into a contiguous block of available memory.
- Segmentation memory management works very similar to paging but here segments are of variable-length whereas 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.

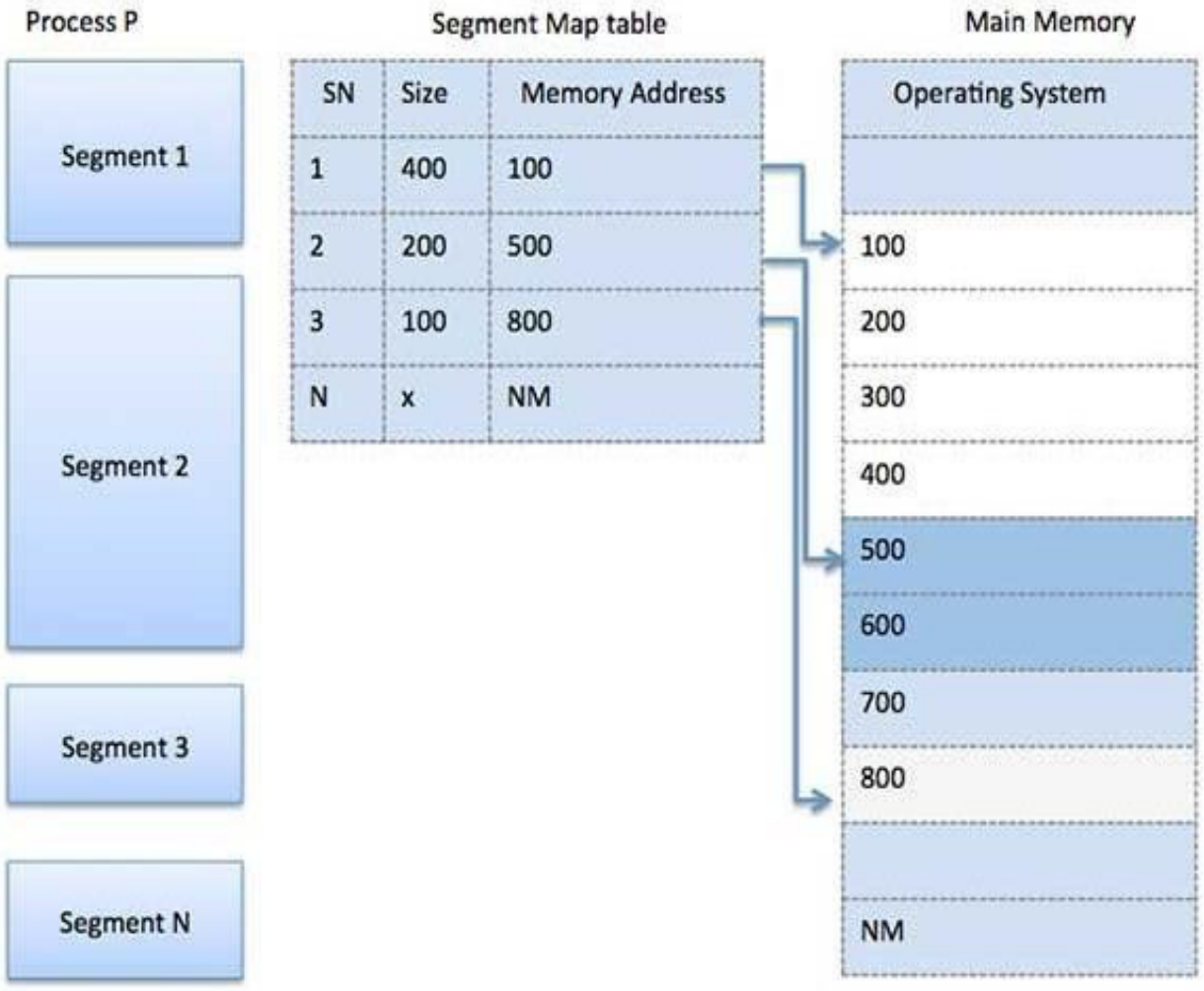


Fig: Segmentation

- Segmentation is a technique to break memory into logical pieces where each piece represents a group of related information.
- For example, data segments or code segment for each process, data segment for operating system and so on.
- Segmentation can be implemented using or without using paging.
- Unlike paging, segment are having varying sizes and thus eliminates internal fragmentation. External fragmentation still exists but to lesser extent.
- Because of the use of unequal size segments, segmentation is similar to dynamic partitioning
- Segmentation eliminates internal fragmentation but like dynamic partitioning, it suffers from external fragmentation
- Like paging, a logical address using segmentation consists of two parts:
    1. Segment number
    2. Offset

Address translation in segmentation

- Base address is fetched
- Certain limit is set initially
- Offset must be in between 0 and limit of that segment
- If third step is complete without error, then add base with offset.

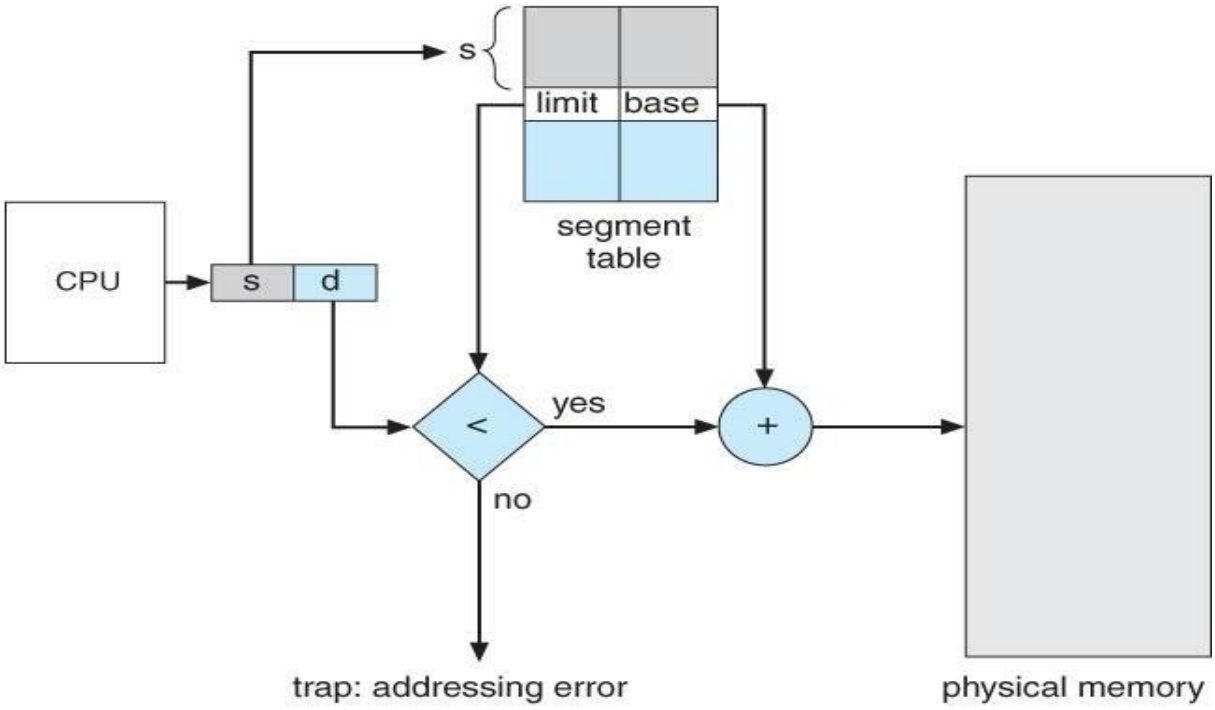


Fig: Address translation process in segmentation

ADVANTAGE OF SEGMENTATION

- It simplifies the handling of growing data structure (OS can grow or shrink the segments as needed)
- It allows programs to be altered and recompiled independently, without requiring the entire set of programs to be relinked and reloaded
- It lends itself to sharing among a process. A programmer can place a utility program or a useful table of data in a segment that can be referenced by other processes
- It lends itself to protection. Because a segment can be constructed to contain a well-defined set of programs or data, the programmer/system administrator can assign access privileges in a convenient fashion.

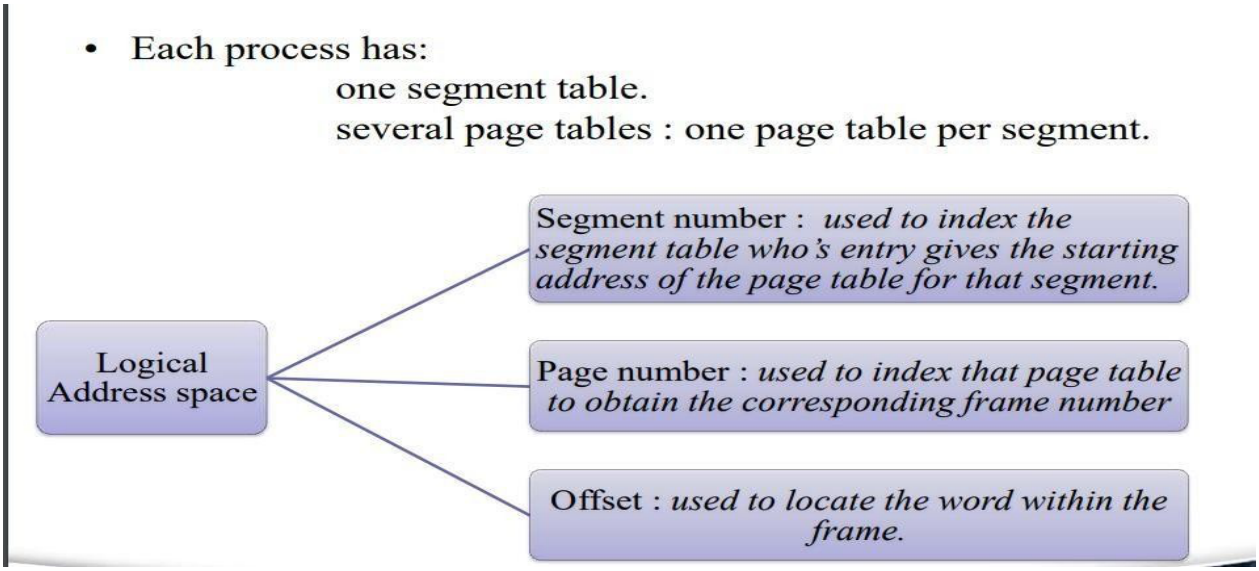
PAGING vs SEGMENTATION

COMPARISON	PAGING	SEGMENTATION
Basic	A page is of fixed block size.	A segment is of variable size.
Fragmentation	Paging may lead to internal fragmentation.	Segmentation may lead to external fragmentation.
Address	The user specified address is divided by CPU into a page number and offset.	The user specifies each address by two quantities a segment number and the offset (Segment limit).
Size	The hardware decides the page size.	The user specifies the segment size.
Table	Paging involves a page table that contains base address of each page.	Segmentation involves the segment table that contains segment number and offset (segment length).

Segmentation with paging

- It allows to use segmentation from user‘s point of view and also to divide each segment into pages of fixed size.
- In a combined paging/segmentation system, a user‘s address space is broken up into a number of segments. Each segment is broken up into a number of fixed-sized pages which are equal in length to a main memory frame
- Segmentation is visible to the programmer
- Paging is transparent to the programmer
- Segments can be of different lengths, so it is harder to find a place for a segment in memory than a page. With segmented virtual memory, we get the benefits of virtual memory but we still have to do dynamic storage allocation of physical memory. In order to avoid this, it is possible to combine segmentation and paging into a two-level virtual memory system.

Each segment descriptor points to page table for that segment. This give some of the advantages of paging (easy placement) with some of the advantages of segments (logical division of the program).



How to get address?

- The segment number indexes into the segment table which yields the base address of the page table for that segment.
- Check the remainder of the address (page number and offset) against the limit of the segment.
- Use the page number to index the page table. The entry is the frame. (The rest of this is just like paging.)
- Concat the frame and the offset to get the physical address.

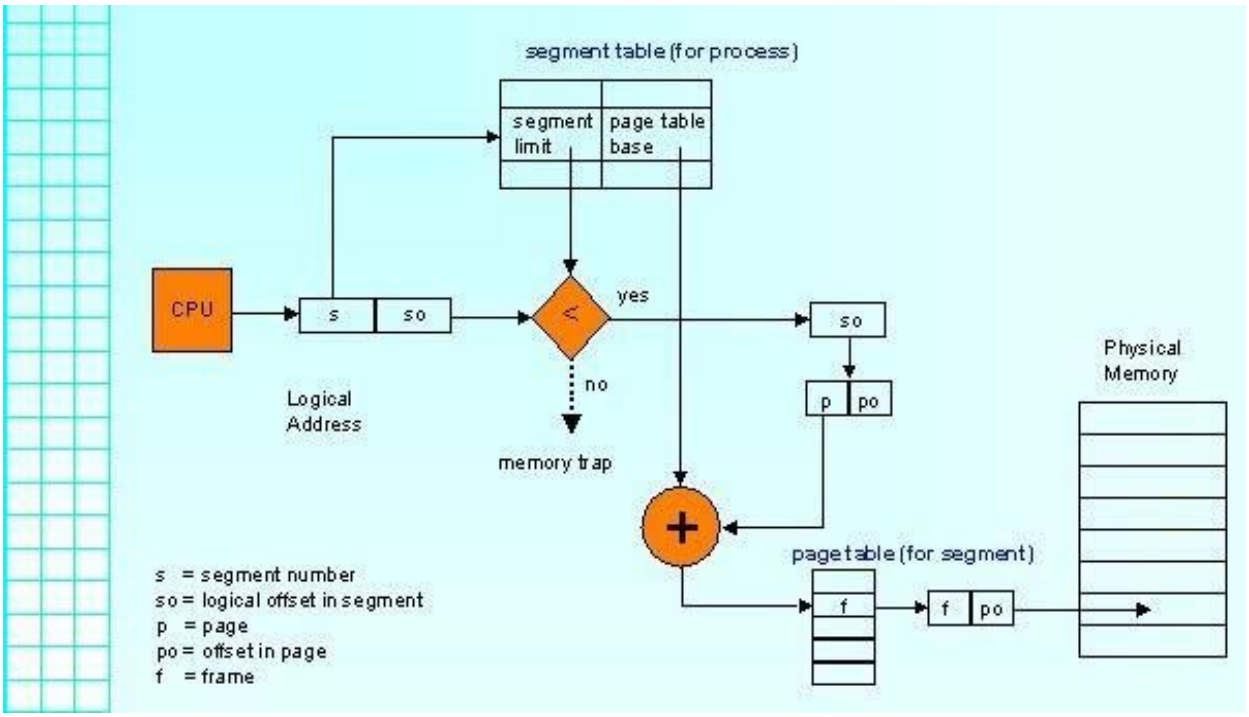


Fig: Address Translation Segmentation with paging

Advantages

- Reduces memory usage as opposed to pure paging
  - Page table size limited by segment size
  - Segment table has only one entry per actual segment
- Share individual pages by copying page table entries.
- Share whole segments by sharing segment table entries, which is the same as sharing the page table for that segment.
- Most advantages of paging still hold
  - Simplifies memory allocation
  - Eliminates external fragmentation.
- In general, this system combines the efficiency in paging with the protection and sharing capabilities of the segmentation