# EDU ENGINEERING

PIONEER OF ENGINEERING NOTES

## TAMIL NADU'S BEST EDTECH PLATFORM FOR ENGINEERING

**CONNECT WITH US**

**WEBSITE:** www.eduengineering.net

**TELEGRAM:** @eduengineering

**INSTAGRAM:** @eduengineering

- **Regular Updates for all Semesters**
- **All Department** Notes AVAILABLE
- **Handwritten** Notes AVAILABLE
- **Past Year** Question Papers AVAILABLE
- **Subject wise** Question Banks AVAILABLE
- **Important Questions** for Semesters AVAILABLE
- Various **Author Books** AVAILABLE

# UNIT-II

# PROCESS MANAGEMENT

1) Processes
2) Process Concept
3) Process Scheduling
4) Operations on Process
5) Inter-process Communication
6) CPU Scheduling
7) Scheduling Criteria
8) Scheduling algorithm
9) Multiple-Processor scheduling.
10) Real-time Scheduling
11) Threads
12) Overview
13) Multithreading models
14) Threading issues
15) Process Synchronization
16) The critical-section Problem.
17) Synchronization Hardware.
18) Mutex locks
19) Semaphores.
20) Classic problems of synchronization.
21) Critical regions.
22) Monitors
23) Deadlocks
24) System models
25) Deadlock characterization
26) Methods for handling deadlocks
27) Deadlock Prevention
28) Deadlock avoidance
29) Deadlock detection
30) Recovery from deadlock.

## (1) Process :

Early Computer Systems allowed only one Program to be executed at a time. This Program had complete control of the system and had access to all the System's resources.

In contrast, current-day computer systems allow multiple Programs to be loaded into memory and executed concurrently. Process is a program which is in execution. A Process is the unit of work in a modern time-sharing system. By switching the CPU between processes, the operating system can make the computer more productive.

## (2) Process Concept :

An operating system executes a variety of Programs.

* Batch system - jobs
* Time Shared Systems - User programs (or) tasks.

We will use the terms job and Process almost interchangeably.

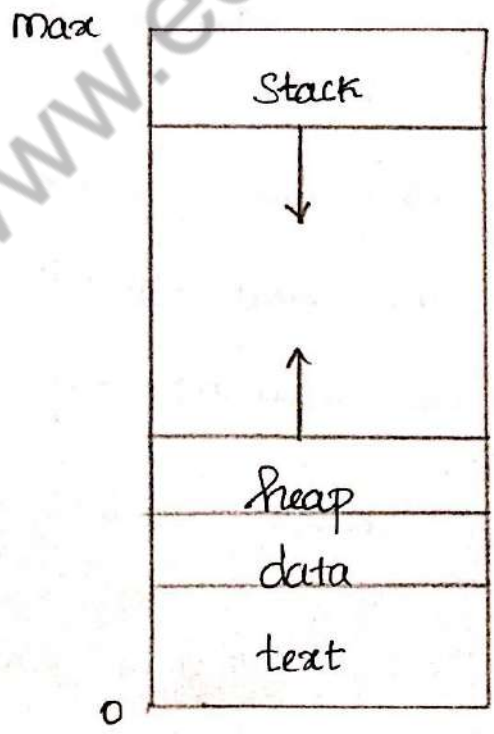
Process - It is a Program in execution (informal definition)

Program is passive entity stored on disk (executable file), Process is active.

Program becomes process when executable file loaded into memory.

* Execution of Program started via GUI, command line entry of its name, etc.
* One Program can be several processes.
* Consider multiple users executing the same Program.
* In memory, a process consists of multiple Parts:
  * Program code, also called text section
  * Current activity including
    Program counter
    Processor registers
  * Stack containing temporary data
    function parameters return address, local variables.
  * Data section containing global variables.
  * Heap containing memory dynamically allocated during run time.

Process in memory.



We emphasize that a program by itself is not a Process; a Program is a passive entity, such as a file containing a list of instructions stored on disk (often called an executable file), whereas a Process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources.

A Program becomes process, when an executable file is loaded into memory. Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in prog.exe or a.out)

* Process state :

As a process executes, it changes state. The state of a process is defined in part by the current activity of that process. Each process may be in one of the following states;

New - The Process is being created.

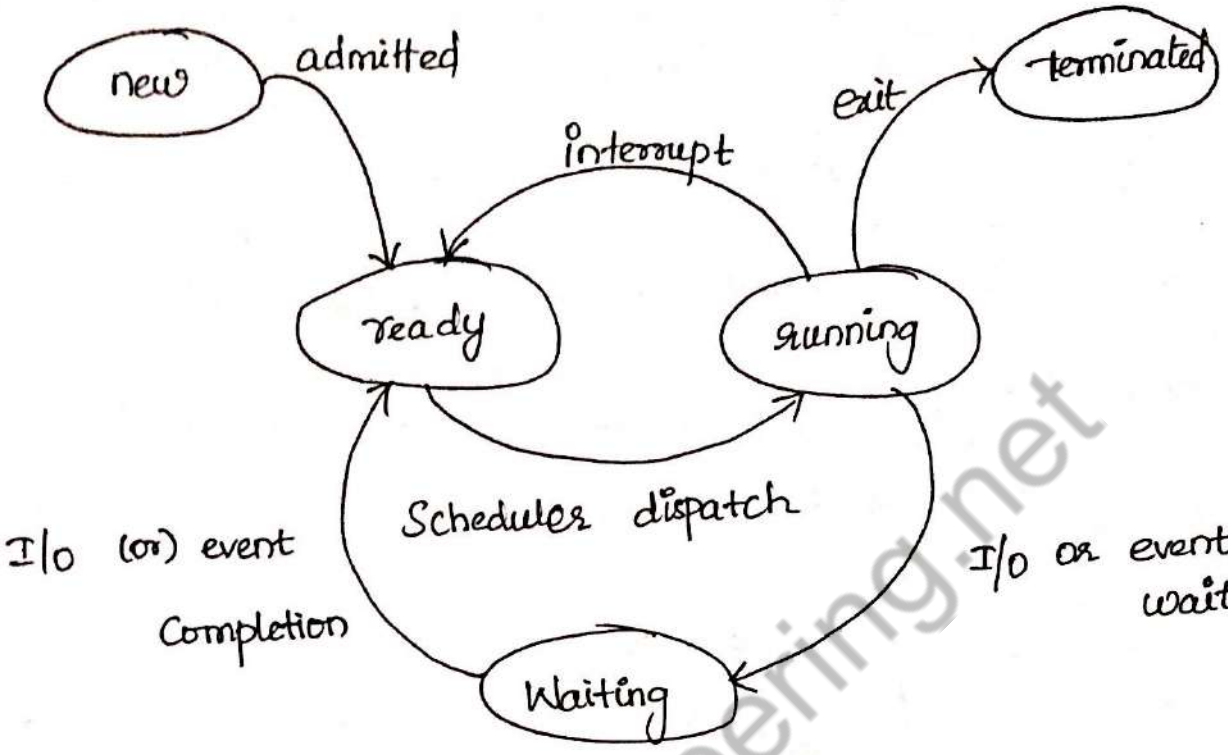
Running - Instructions are being executed.

Waiting - The Process is waiting for some event to occur.

Ready - The Process is waiting to be assigned to a processor.

Terminated - The Process has finished execution.

Diagram of Process state.




It is important to realize that only one Process can be running on any Processor at any instant. Many Processes may be ready and waiting, however.

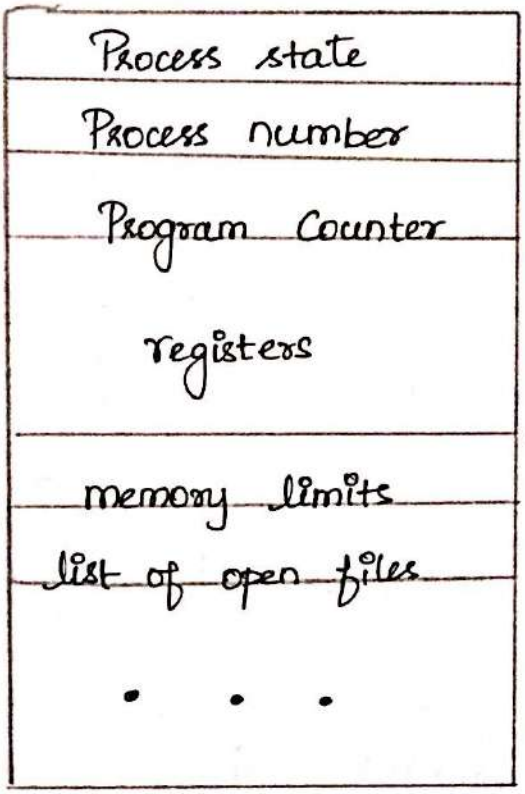
* Process control Block (PCB) :

Each Process is represented in the operating System by a Process control Block (PCB) - also called a task control block. It contains many pieces of information associated with a specific Process, including these :

* Process state
* Program counter
* CPU register
* CPU - Scheduling information.
* Memory - Management information.
* Accounting information.
* I/O status information.

"Data structure maintained by OS for every Process is called PCB"

## Process control Block - PCB




* Each information of Process is controlled by PCB, and each Process information is Stored in PCB, and Process ID.
* PCB is identified by an Integer called Process ID (PID)

Program state : The state may be new, ready, running, waiting, halted and so on.,

Program Counter : The counter indicates the address of the next instruction to be executed for this Process.

CPU registers : The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the Program counter, this state information must be saved when an interrupt occurs, to allow the Process to be continued correctly afterward. (temporary storage device)

CPU - Scheduling information :

This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

Memory - Management information :

This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.

Accounting information :

This information includes the amount of CPU and real time used, time limits, account numbers, job (or) process numbers, and so on.

I/O status information :

This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

In brief, the PCB simply serves as the repository for any information that may vary from process to process.

*Threads :

The process model discussed so far has implied that a process is a program that performs a single thread of execution. for eg., when a process is running a word-processor program, a single thread of instructions is being executed. Many modern OS have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time.

## (3) Process Scheduling :

The objective of multiprogramming is to have some Process running at all times, to maximize (cpu utilization.)

The objective of timesharing is to switch the cpu among processes so frequently that users can interact with each program while it is running. To meet these objectives, the Process scheduler selects an available Process for program execution on the cpu. for a single-Processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the cpu is free and can be rescheduled.

- * Scheduling Queues
- * Schedulers
- * Context switch

* Scheduling Queues :

Job Queue - set of all process in the system.

Ready Queue - set of all process residing in main memory, ready and waiting to execute.

Device Queue - set of Processes waiting for an I/o device process migrate among the various queues.

A common representation of Process scheduling is a Queueing diagram. Two types of Queues are Present : the ready Queue and a set of device Queues. The circles represent the resources that serve the Queues, and the arrows indicate the flow of Processes in the System. A new Process is initially put in the ready Queue. It waits there until it is selected for

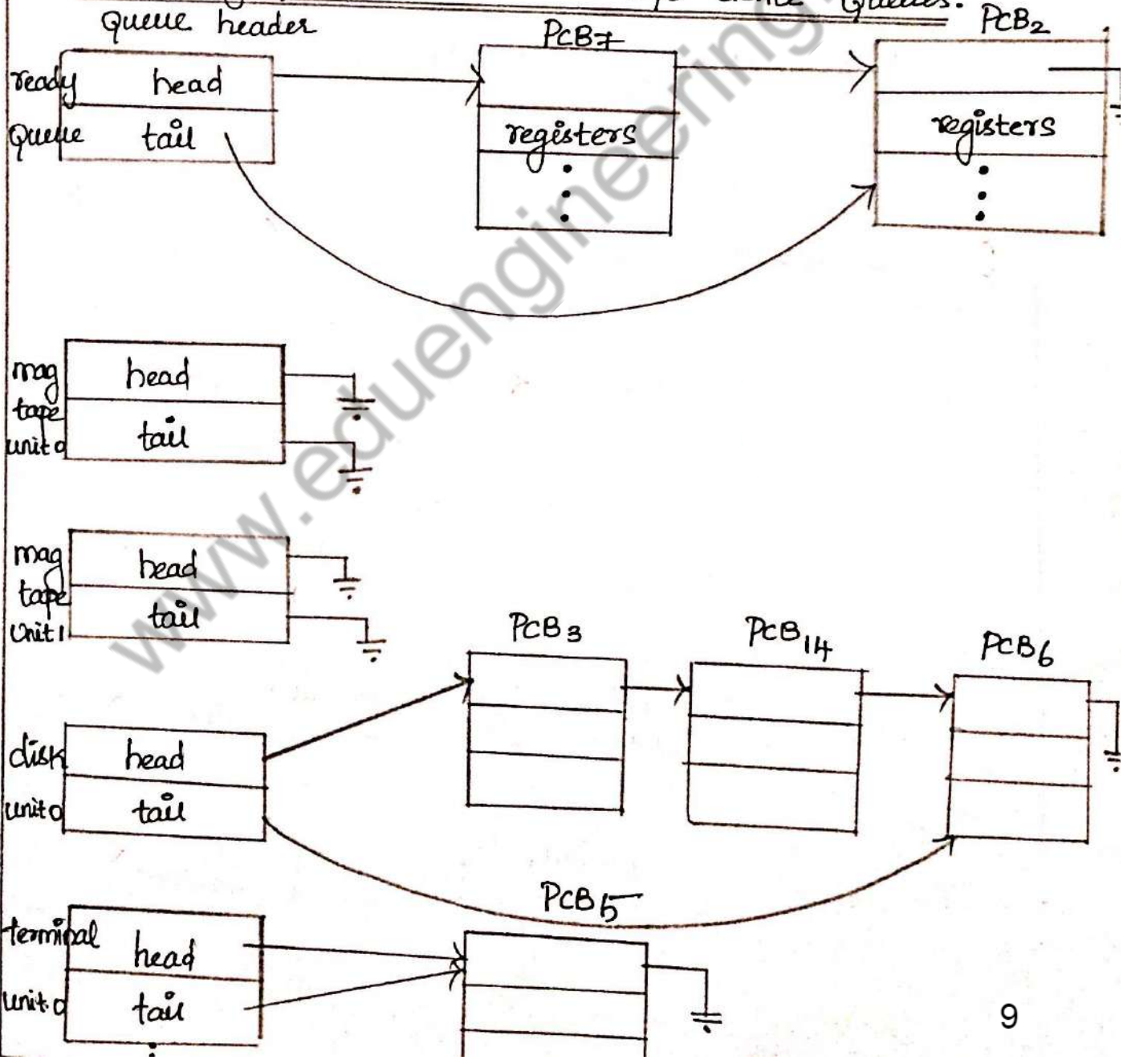
execution (or) dispatched. Once the process is allocated the cpu and is executing, one of several events could occur.

* The Process could issue an I/o request and then be placed in an I/o Queue.

* The Process could create a new child process and wait for the child's termination.

* The Process could be removed forcibly from the cpu, as a result of an interrupt, and be put back in the Ready Queue.

The ready Queue and various I/o device Queues.



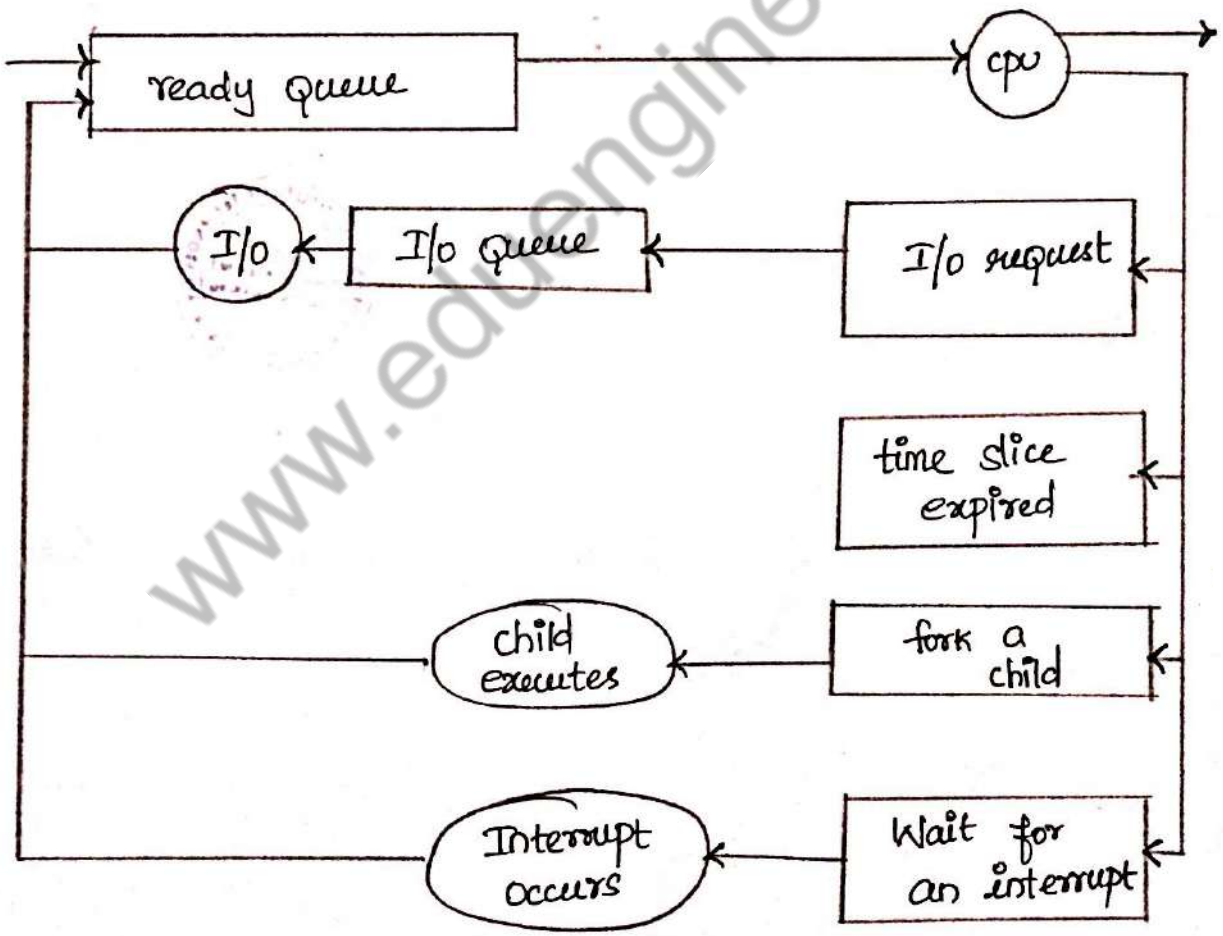
* Schedulers : [N/D-19]

(*) Long-term scheduler (or job scheduler) – selects which Processes should be brought into the ready Queue.

(*) Short-term scheduler (or cpu scheduler) - selects which Process should be executed next and allocates cpu.

The operating system, must select for scheduling Purposes, Processes from these Queues in some fashion. The selection Process is carried out by the appropriate scheduler.

Queueing diagram representation of Process scheduling.



The primary distinction between these two schedulers lies in frequency execution. The short-term schedulers must select a new process a new process for the cpu frequently.

A Process may execute for only a few milliseconds before waiting for an I/o request. Often, the short-term scheduler executes at least once every 100 milliseconds. Because of the short time between executions, the short-term scheduler must be fast. If it takes 10 milliseconds to decide to execute a Process for 100 milliseconds, then $10/(100+10) = 9$ Percent of the cpu is being used (wasted) simply for scheduling the work.

The long-term scheduler executes much less frequently; minutes may separate the creation of one new process and the next. The long-term scheduler controls the degree of multiprogramming (the number of Processes in memory).

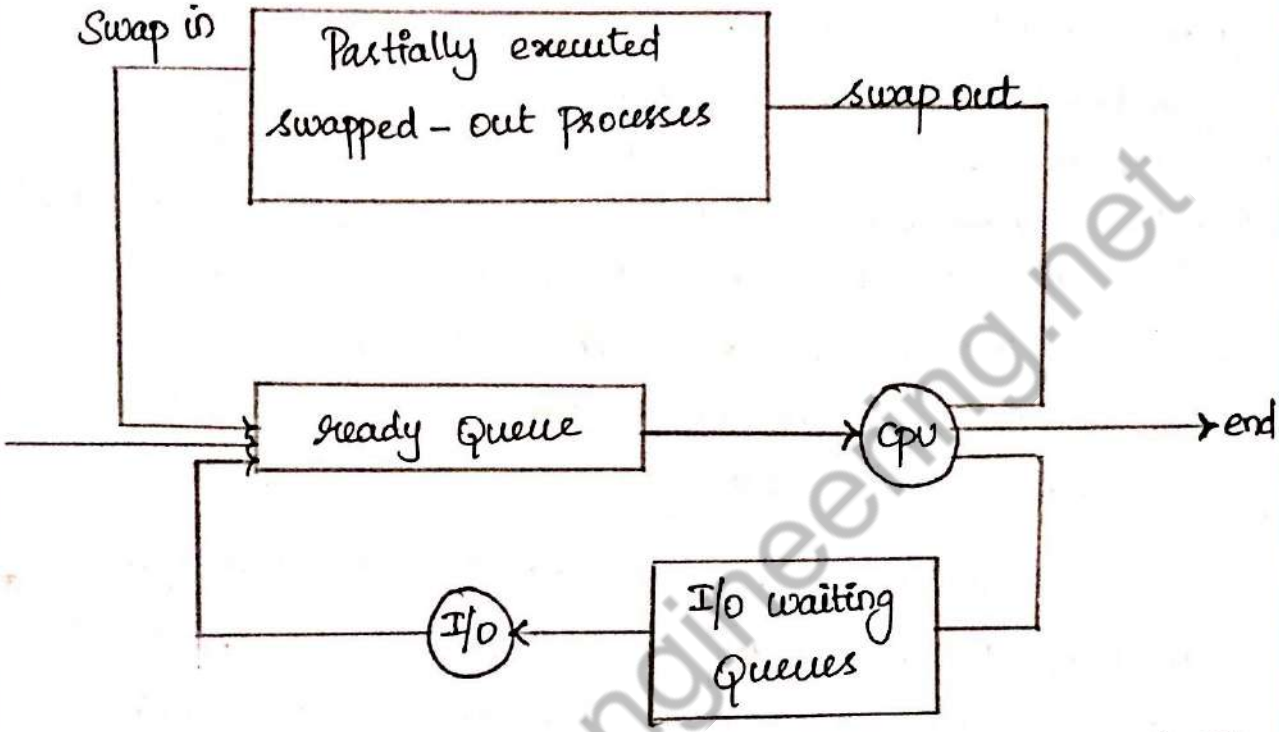
If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

Processes can be described as either:

- I/o bound process - spends more time doing I/o than computations, many short cpu bursts.

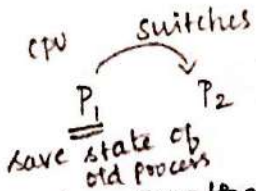
CPU-bound Process - Spends more time doing computations, few very long CPU bursts.

Addition of medium-term scheduling to the Queueing diagram.




Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of Scheduling. The key idea behind a medium-term Scheduler is that sometimes it can be advantageous to remove a process from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming.

* Context Switch :




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

Execution :

* Parent and children execute ~~&~~ concurrently.
* Parent ~~and~~ ~~th~~ waits until children terminate.

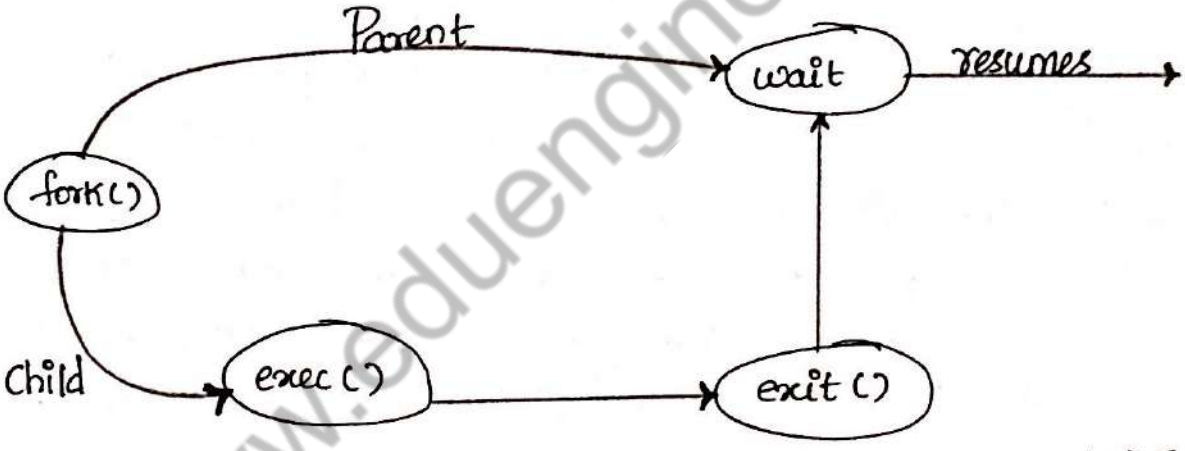
Address space :

* child duplicate of parent.
* child has a program loaded into it.

UNIX examples :

* fork system call create new process.
* exec System call used after a fork to replace the process' memory space with a new program.

Process creation using fork() system call.




When a process creates a new process, two possibilities exist in terms of execution.

1) The parent continues to execute concurrently with its children.

2) The Parent waits until some (or) all of its children have terminated.

* Context - switch time is overhead; the system does no useful work while switching.
* Time dependent on hardware support.

## (4) Operations on Processes.

The Processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination.

✓ * Process Creation (fork())
✓ * Process termination (exit())
*

* Process creation:

A Process may create several new processes, via a Create-process system call, during the course of execution. The creating process is called a parent process, and the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a tree of Processes.

Most operating systems (including UNIX and the Windows family of operating systems) identify Processes according to a unique Process identifier (or pid), which is typically an integer number.

Resource sharing:
→ Parent and children share all resources.
→ Children share subset of Parent's resources.
→ Parent and child share no resources.

There are also two possibilities in terms of the address space of the new process:

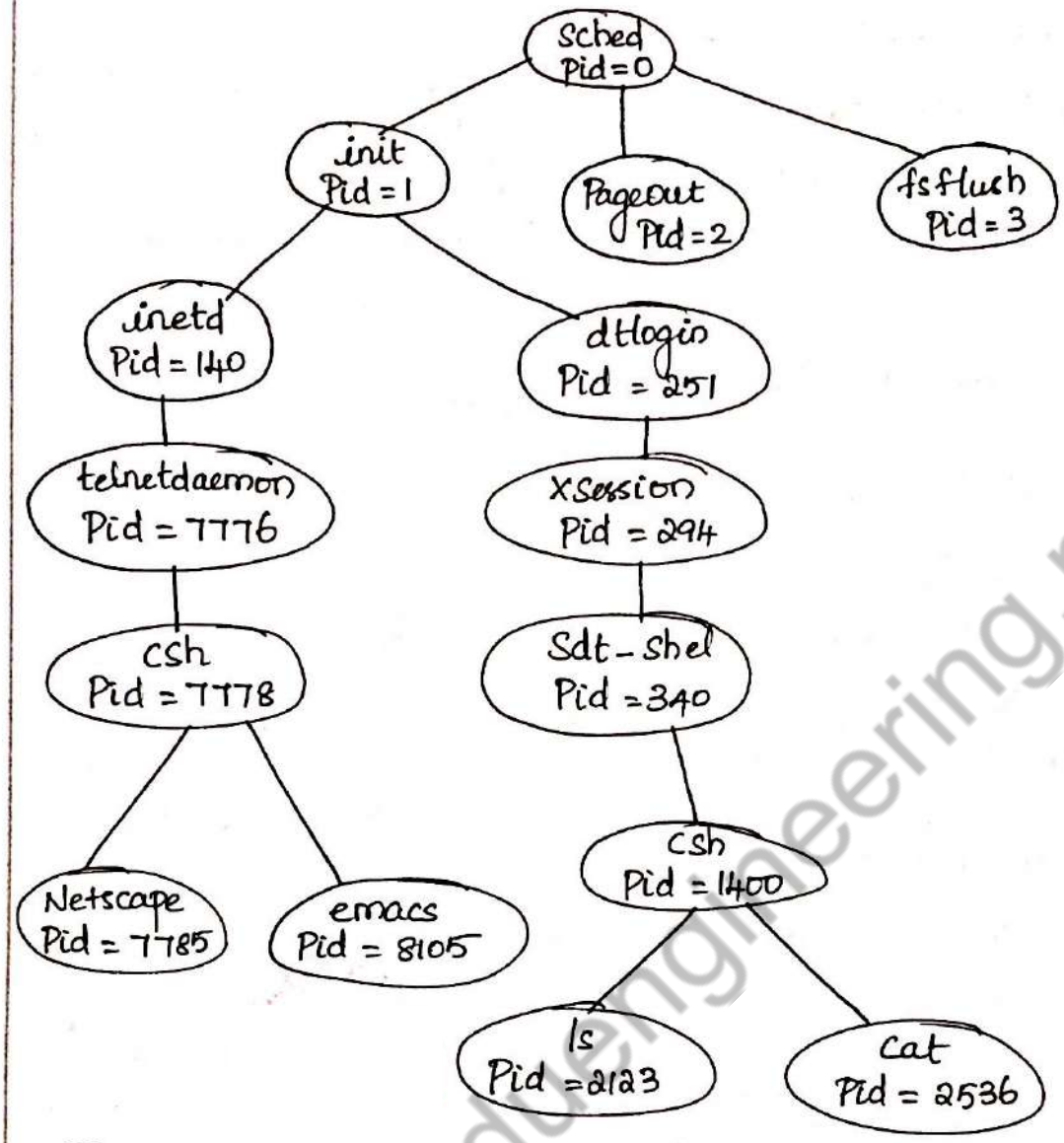
1) The child process is a duplicate of the parent process (it has the same program and data as the parent).

2) The child process has a new program loaded into it.

C Program Forking Separate Process.

```
int main ()
{
   Pid_t pid;
   /* fork another process */
  Pid = fork ();
  if (Pid <0)
{
  /* error occurred */
 fprintf (stderr, "fork failed");
}  exit (-1);
   else if (Pid ==0)
{
   error /* Child Process */
 execlp ("bin/ls", "ls", NULL);
}
   else
  {  /* Parent process */
  /* Parent will wait for the child to complete */
   wait (NULL);
   Printf ("child Complete");
    exit (0);
  }
}
```

A tree of processes on a typical Solaris system.




In general, a process will need certain resources (cpu time, memory, files, I/o devices), to accomplish its task. When process creates a subprocess, that subprocess may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process.

The parent may have to partition its resources among its children, or it may be able to share some resources among several of its children.

* Process Termination : Parent - exit()
child - wait()

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the (exit() system call.) At that point, the process may return a status value (typically an integer) to its parent process via the wait() system call.

A process can cause the termination of another process via an appropriate system call. Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Note that a parent needs to know the identities of its children. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.

✓ A parent may terminate the execution of one of its children for a variety of reasons, such as these;

* The child has exceeded its usage of some of the resources that it has been allocated.

* The task assigned to the child is no longer required.

* The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

✓ If a process terminates either normally or abnormally then all its children must also be terminated. This phenomenon, refered to as cascading termination, is normally initiated by the operating system.

Eg., To illustrate process execution and termination, consider that, in UNIX, we can terminate a process by using the exit() system call; its parent process may wait for the termination of a child process by using the wait() system call.

The wait() system call returns the process identifier of a terminated child so that the parent can tell which of its children has terminated.

## 5) Interprocess communication (IPC)

Processes executing concurrently in the operating system may be either independent processes (or) cooperating processes.

A process is independent if it cannot affect (or) be affected by the other processes executing in the system. Any process that does not share data with any other process is independent. A process is cooperating if it can affect or be affected by the other processes executing in the system.

✓ Clearly, any process that shares data with other processes is a cooperating process.

There are several reasons for providing an environment that allows process cooperation:

* Information sharing.
* Computation speedup.
* Modularity.
* Convenience.

Information sharing :

Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.

Computation speedup :

If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).

Modularity :

We may want to construct the system in a modular fashion, dividing the system functions into separate Processes or threads.

Convenience :

Even an individual user may work on many tasks at the same time. for instance, a user may be editing, Printing, and compiling in parallel.

Mechanism for processes to communicate and to Synchronize their actions.

Message system - Process Communicate with each other without resorting to shared variables.

Ipc facility provides two operations :

Send (message) - message size fixed (or) variable.

receive (message).

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

→ establish a communication link between them.

→ exchange messages via send / receive.

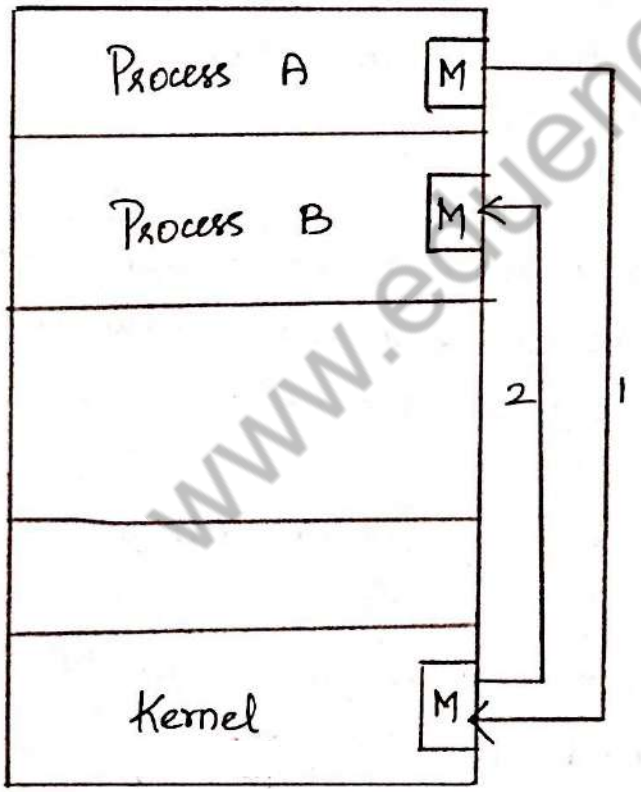
Implementation of communication link

→ Physical (eg., shared memory, hardware bus)

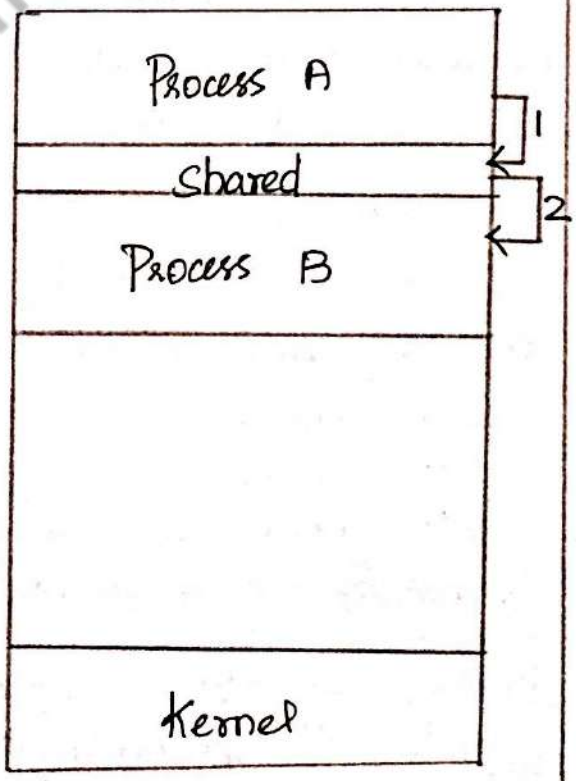
→ Logical (eg., logical properties).

Communication models :-

(a) Message passing




(b) shared Memory



Ipc mechanism will allow them to exchange data and information. These are two fundamentals models of Ipc

* Shared memory
* Message passing.

In shared-memory model, a region of memory that is shard by cooperating process is established.

In message passing model, communication takes place by means of messages exchanged between the cooperating Processes.

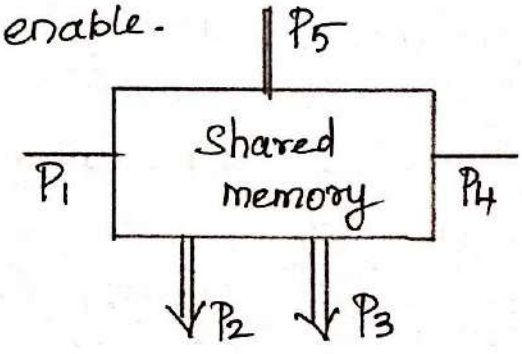
Ipc is a mechanism by which two (or) more process communicate with each other through message passing mechanism without using shared addressed space.

"Ipc means how two process are communicate to each other".

✓* Sharing of data (Memory)
✓* Message passing Mechanism.

Sharing of Data :-

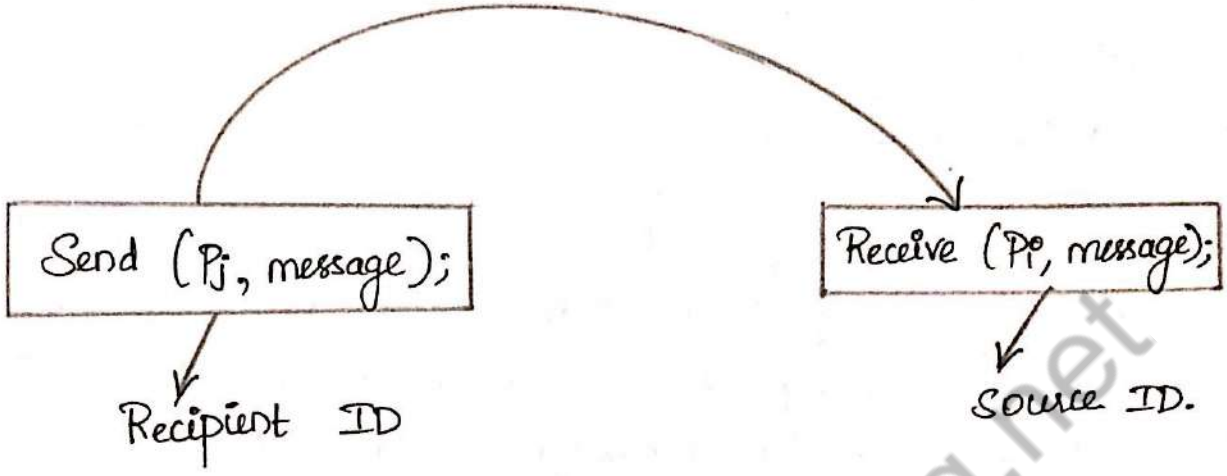
Two (or) more Processes share their data with each other by which Ipc is enable.




The disadvantage of this mechanism is that your data is shared among all the user.

Message passing Mechanism:

It send the message to the recipient and recipient receive the message from the source.




Message passing communication:

There are two types of message passing communication are used.

→ Direct Communication.
→ Indirect Communication.

Direct Communication:

Sometime is called "Symmetric Naming Convention". Here sender and receiver both knows the identity (ID) of each other.


Direct Communication use two types of naming convention.

1) Asymmetric Naming Convention
2) Symmetric Naming Convention.

Symmetric Naming Convention:

Both Sender & receiver ID's are known.

Eg.,

```
Send (recipient ID, Message); // Sender end
Receiver (Source ID, message); // Receiver end.
```

Asymmetric Naming convention:

Sender knows the ID of recipient but receiver only receive the message without knowing Sender ID.

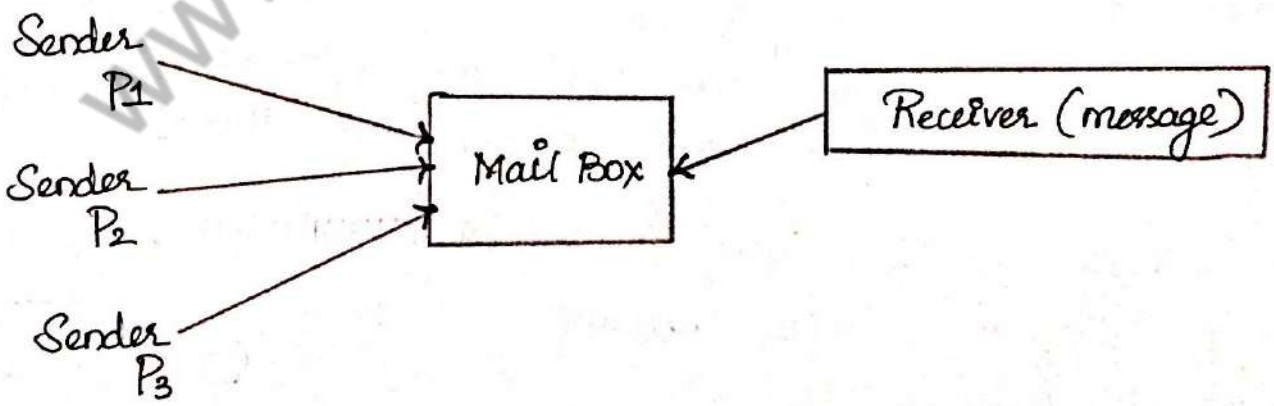
Eg.,

```
Send (recipient ID, message); //sender end
Receiver (Message); // receiver end.
```

Indirect communication:

In this communication we doesn't know the identity of receiver and sender. Receiver receive the message from that mail box. The system calls are as following.

Mailbox own by the receiver.



## Synchronous and Asynchronous Message passing.

* It is based on blocking and non blocking method.

* Block refers synchronous message passing and non blocking refers asynchronous message passing.

There are 4 types of message passing.

1) Blocking Send : Sender process is block until the receiver receive the message.

2) Non-Blocking Send : Sender process continuously Send the message without looking whether receiver receive the message or not.

3) Blocking Receive : Receiver process is block till sender process send the message.

4) Non Blocking receive : Receiver continuously receive the message without any restriction.

* Buffering in Ipc :

Buffering is a storage facility which are work in between communication between two (or) more devices.

In Ipc messages are not directly transfer to the receiver, it transfer through buffer mechanism.

There are '3' types of buffering techniques;

(1) Zero capacity buffering.
(2) Bounded buffer.
(3) Unbounded buffer.

Buffering Techniques :

(1) Zero capacity : There is no memory used take and deliver the message file that sending process is block.

(2) These type of buffering has buffer size =0.

Bounded Buffer : These type of buffer have finite length.

Unbounded Buffer : These type of buffer have infinite length. thus, any number of messages can wait in it. The Sender never blocks.

The zero-capacity case is sometimes referred to as a message system with no buffering; the other cases are referred to as systems with automatic buffering.

## (6) Cpu Scheduling :

→ Cpu Scheduling is the basis of multiprogrammed operating systems.

→ The objective of multiprogramming is to have some Process running at all times, in order to maximize cpu Utilization.

→ Scheduling is a fundamental operating-system function.

→ Almost all computer resources are scheduled before use.

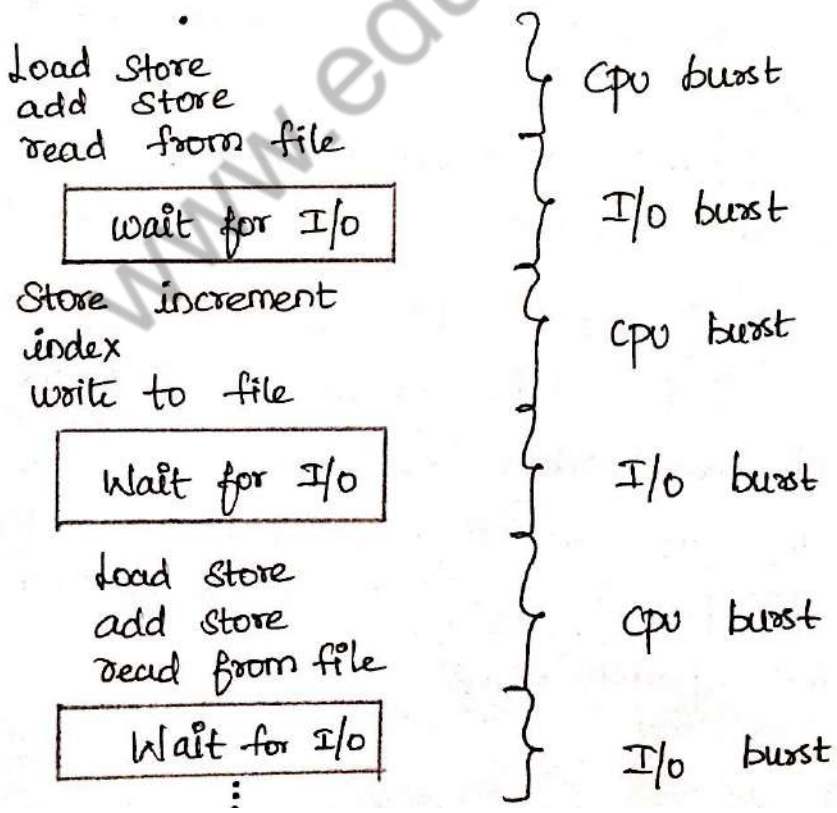
Cpu Scheduling is a process which allows one Process to use the cpu while execution of another Process is on hold (in waiting state) due to Unavailability of any resources like I/o etc., Thereby making full use of cpu, The aim of cpu scheduling is to make the system efficient, fast and fair.

Cpu- I/o Burst cycle.
Cpu Scheduler
Pre emptive scheduling
Non-Preemptive scheduling
Dispatcher.

## Cpu- I/o Burst cycle:

* Process execution consists of a cycle of cpu execution and I/o wait.
* Processes alternate between these two states.
* Processes execution begins with a cpu burst.
* That is followed by an I/o burst, then another cpu burst, then another I/o burst, and so on.
* Eventually, the last cpu burst will end with a system request to terminate execution, rather than with another I/o burst.

Alternating sequence of cpu and I/o bursts.



CPU Scheduler:

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed.

The selection process is carried out by the short-term scheduler or CPU scheduler. The ready queue is not necessarily a first-in, first-out (FIFO) queue. It may be a FIFO queue, a priority queue, a tree, or simply an unordered linked list.

Preemptive Scheduling:

CPU scheduling decisions may take place under the following four circumstances:

1) When a process switches from the running state to the waiting state.
2) When a process switches from the running state to the ready state.
3) When a process switches from the waiting state to the ready state.
4) When a process terminates.

Under 1 & 4 scheduling scheme is non-preemptive. Otherwise the scheduling scheme is preemptive

Non-Preemptive Scheduling:

→ In non preemptive scheduling, once the CPU has been allocated a process, the process keeps the CPU until it releases the CPU either by termination or by switching to the waiting state.

→ This scheduling method is used by the Microsoft windows environment.

(6) Dispatcher :

The dispatcher is the module that gives control of the cpu to the process selected by the short-term scheduler. The time it takes for the dispatcher to stop one process and start another running is known as dispatch latency.

This function involves :

1) Switching context
2) Switching to user mode
3) Jumping to the proper location in the user program to restart that program.

(7) Scheduling Criteria :

Many criteria have been suggested for comparing cpu-scheduling algorithms. The criteria include the following.

1) cpu Utilization
2) Throughput
3) Turnaround time
4) Waiting time
5) Response time

cpu Utilization :

We want to keep the cpu as busy as possible. Conceptually, cpu utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).

Throughput :

If the cpu is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be

One process per hour; for short transactions, it may be ten processes per second.

Turnaround time :

from the point of view of a particular Process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time.

Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the cpu, and doing I/o.

Waiting time :

The cpu scheduling algorithm does not affect the amount of time during which a Process executes or does I/o ; it affects only the amount of time that a Process spends waiting in the ready Queue.

Waiting time is the sum of the Periods spent waiting in the ready Queue.

Response time :

In an interactive system, turnaround time may not be the best criterion. The time from the submission of a request until the first response is produced.

Response time is the time it takes to start responding, not the time it takes to output the response.

## (8) Scheduling algorithms :

cpu scheduling deals with the problem of deciding which of the processes in the ready Queue is to be allocated the cpu. There are many different cpu-scheduling algorithms.

* First-come, First-Served Scheduling (FCFS)
* Shortest-Job-First Scheduling
* Priority scheduling
* Round-Robin scheduling.
* Multilevel Queue scheduling.
* Multilevel feedback Queue scheduling.

* First-come, First-Served scheduling :-

The process that requests the cpu first is allocated the cpu first. FCFS policy is easily managed by FIFO queue.

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

| Process | Burst time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

Arrival time : Time at which the process arrives in the ready queue.

completion time : Time at which process completes its execution.

Burst time : Time required by a process for cpu execution.

Turn around time : Time difference between completion time and arrival time.

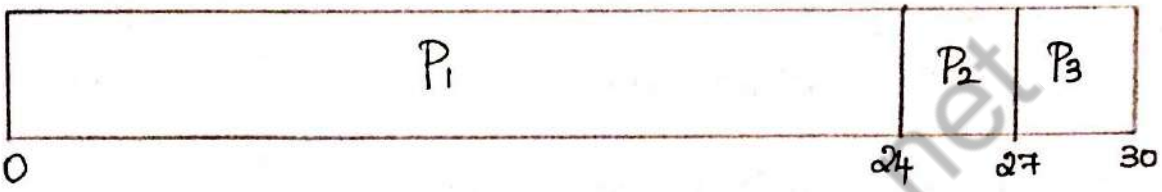
Turn around time = Completion time - Arrival time.

Waiting time : Time difference between turn around time and burst time.

Waiting time = Turn around time - Burst time.

If the processes arrive in the order $P_1$, $P_2$, $P_3$, and are served in FCFS order, we get the result shown in the following Gantt chart.

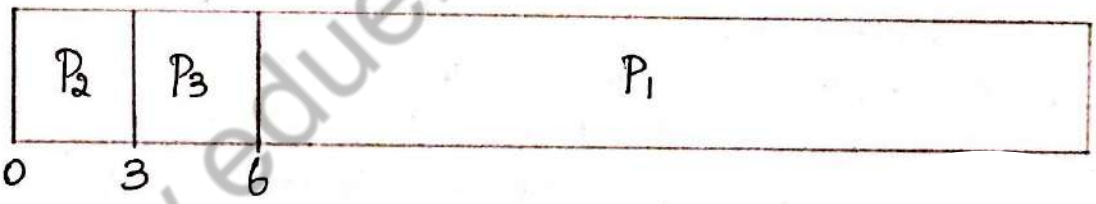
Gantt chart is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes.




The waiting time is 0 milliseconds for process $P_1$, 24 ms for $P_2$, 27 ms for $P_3$.

∴ Average Waiting time = $(0 + 24 + 27)/3 = 51/3$
$= 17$ ms.

Eg., If the process arrives in the order $P_2$, $P_3$, $P_1$, then




Waiting time :- $P_2 \rightarrow 0$ ms
$P_3 \rightarrow 3$ ms
$P_1 \rightarrow 6$ ms.

Average Waiting time :- $(0+3+6)/3 = 9/3$
$= 3$ milliseconds.

Convoy effect is phenomenon associated with the First come First serve (FCFS) algorithm, in which the whole OS slows down due to few slow processes.

FCFS algorithm is non-preemptive scheduling algorithm. Once cpu time has been allocated to a process, other processes can get cpu time only after the current process has finished. This property of FCFS scheduling leads to the situation called convoy effect.

To avoid convoy effect, Preemptive scheduling like Round Robin scheduling can be used - as the smaller processes don't have to wait much for cpu time - making their execution faster and leading to less resources sitting idle.

* Shortest - Job - First Scheduling :

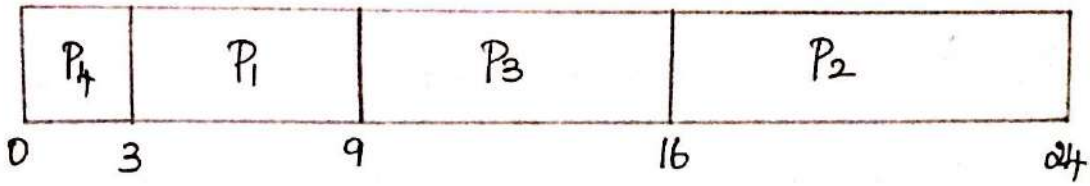
A different approach to cpu scheduling is the SJF scheduling algorithm. This algorithm associates with each process the length of the processes next cpu burst. When the cpu is available, it is assigned to the process that has the smallest next cpu burst. If the next cpu bursts of two processes are the same, FCFS scheduling is used to break the tie.

Note that a more appropriate term for this scheduling method would be the "shortest - next - cpu - burst algorithm", because scheduling depends on the length of the next cpu burst of a process, rather than its total length.

Eg ; With process and cpu burst time in milliseconds.

| Process | Burst time |
|---|---|
| $P_1$ | 6 |
| $P_2$ | 8 |
| $P_3$ | 7 |
| $P_4$ | 3 |

Using SJF Scheduling, the Gantt chart will be as.,




Waiting time :

$P_1 \rightarrow 3$ milliseconds
$P_2 \rightarrow 16$ milliseconds
$P_3 \rightarrow 9$ milliseconds
$P_4 \rightarrow 0$ millisecond.

Average waiting time ;

$\Rightarrow (3+16+9+0)/4 = 28/4 = 7$ milliseconds.

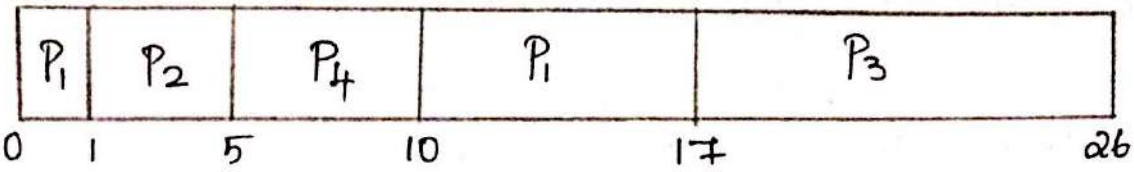
A Preemptive SJF algorithm will preempt the currently executing process, whereas a non-preemptive SJF algorithm will allow the currently running process to finish its cpu burst.

Preemptive SJF scheduling is sometimes called Shortest-remaining-time-first scheduling.

Eg., Consider the following 4 processes, with the length of the cpu burst given in milliseconds.

| Process | Arrival Time | Burst Time |
|---|---|---|
| $P_1$ | 0 | 8 |
| $P_2$ | 1 | 4 |
| $P_3$ | 2 | 9 |
| $P_4$ | 3 | 5 |

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting Preemptive SJF schedule is as depicted in the following Gantt chart.




Process $P_1$ is started at time 0, since it is the only process in the queue.

Process $P_2$ arrives at time 1. The remaining time for process $P_1$ (8-1 = 7 milliseconds) is larger than the time required by process $P_2$ (4 milliseconds), so Process $P_1$ is preempted, and $P_2$ is scheduled. The average waiting time for this example is

$[(10-1) + (1-1) + (17-2) + (5-3)] = 26/4 = 6.5$ ms.

Non-preemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

* Priority Scheduling :

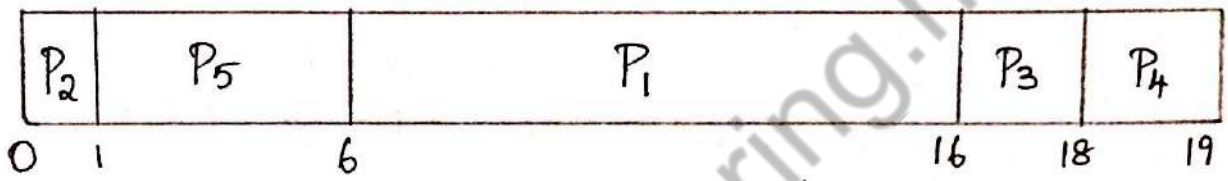
The SJF algorithm is a special case of the general priority scheduling algorithm. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. The larger the CPU burst, the lower the priority, and vice-versa.

Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 (or) 0 to 4095. However, there is no general agreement on whether 0 is the highest (or) lowest priority. Some system use low numbers to represent low priority and others use low numbers for high priority.

But we assume the low numbers represent highest priority.

| Process | Burst time | Priority |
|---|---|---|
| $P_1$ | 10 | 3 |
| $P_2$ | 1 | 1 |
| $P_3$ | 2 | 4 |
| $P_4$ | 1 | 5 |
| $P_5$ | 5 | 2 |

Gantt chart :




Waiting time = $(6+0+16+18+1)/5$

Avg = $41/5 = 8.2$ milliseconds.

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

A preemptive priority scheduling algorithm will preempt the cpu if the priority of the newly arrived process is higher than the priority of the currently running process.

A non preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

* Round - Robin Scheduling:

The RR scheduling algorithm is designed especially for time-sharing systems. It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a time Quantum (or) time slice is defined. A time Quantum is generally from 10 to 100 milliseconds in length.

To implement RR scheduling, we keep the ready Queue as a FIFO Queue of Processes. New Processes are added to the tail of the ready Queue. The cpu scheduler picks the first process from the ready Queue, sets a timer to interrupt after 1 time Quantum, and dispatches the processes.

Only two things happen. The process may have a cpu burst of less than 1 time Quantum. In this case, the process itself will release the cpu voluntarily.

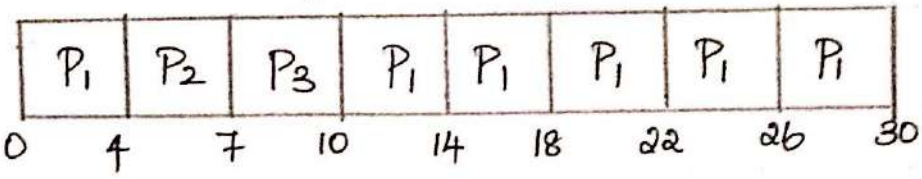
Otherwise, if the cpu burst of the currently running process is longer than 1 time Quantum, the timer will go off and will cause an interrupt to the Operating System.

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

| Process | Burst time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

If we use a time Quantum of 4 milliseconds, then Process $P_1$ gets the 1st 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time Quantum, and the CPU is given to the next process in the Queue, Process $P_2$. It doesn't need 4 ms, so it quits before its time Quantum expires, and so on..

The resulting RR schedule is as follows;




Waiting time :-

$P_1 \rightarrow$ 6 ms (10-4)

$P_2 \rightarrow$ 4 ms

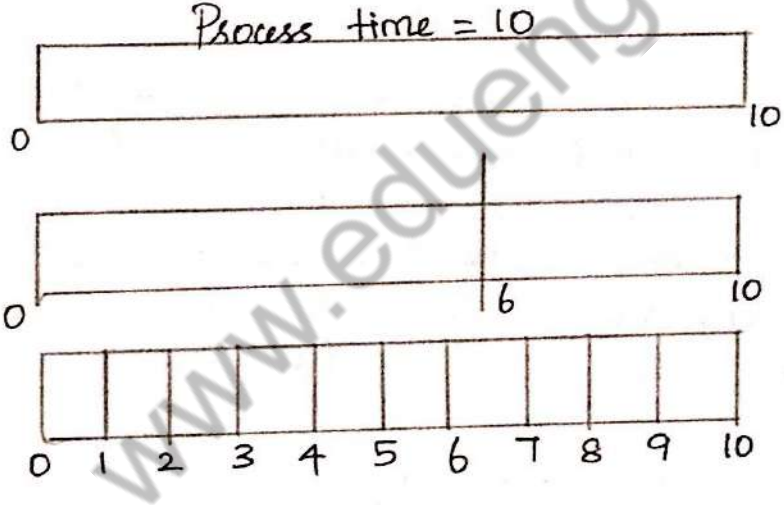
$P_3 \rightarrow$ 7 ms.

Average Waiting time is;

$(6+4+7)/3 = 17/3$

$= 5.66$ ms//.

How a smaller time Quantum increases context switches.




| Quantum | Context switches |
|---|---|
| 12 | 0 |
| 6 | 1 |
| 1 | 9 |

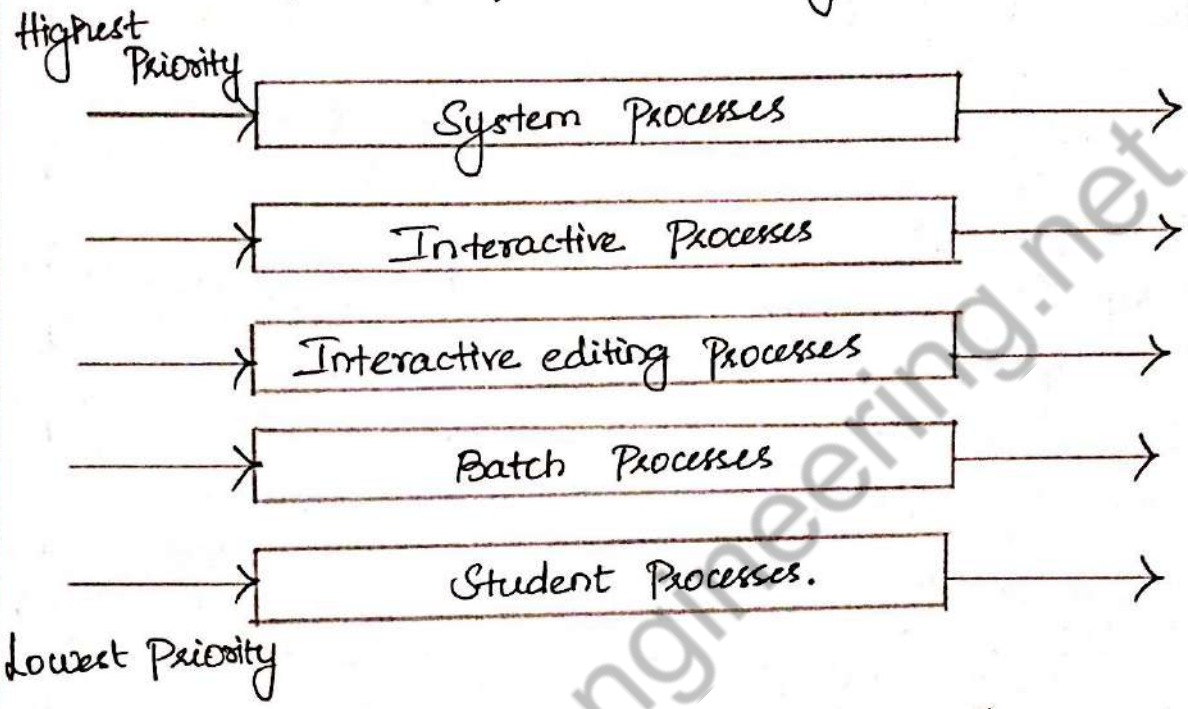
for eg., that we have only one process of 10 time units. If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead.

If the Quantum is 6 time units, however, the Process requires 2 Quanta, resulting in a context switch. If the time Quantum is 1 time unit, then 9 context switches will occur, slowing the execution of the Process.

* Multilevel Queue scheduling:

It partitions the ready Queue into several separate Queues. The Processes are Permanently assigned to one Queue, generally based on some property of the Process, such as memory size, Process Priority, or Process type. Each Queue has its own scheduling algorithm.

Multilevel Queue scheduling:

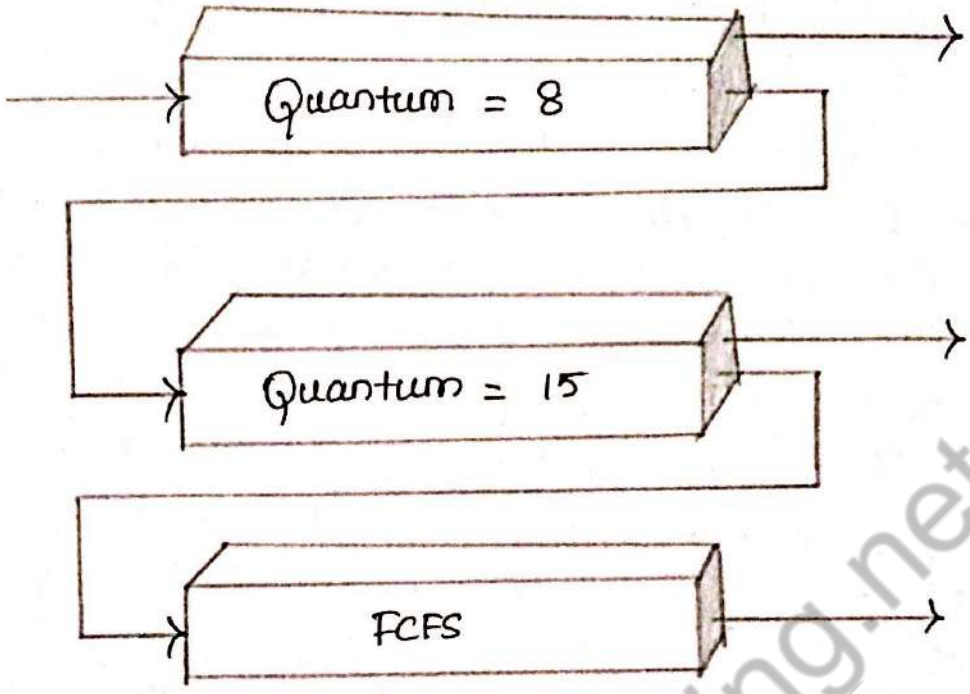



for instance, in the foreground (interactive) Queue – background (batch) Queue example, the foreground Queue can be given 80 percent of the CPU time for RR scheduling among its Processes, whereas the background Queue receives 20 percent of the CPU to give to its Processes on an FCFS basis.

* Multilevel feedback Queue scheduling:

It allows a process to move between Queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority Queue. This schemes leaves I/O bound and Interactive Processes in the higher-priority Queues.

Multilevel feedback Queues.




In general, a multilevel feedback Queue scheduler is defined by the following parameters.

1) The number of Queues.
2) The scheduling algorithm for each Queue.
3) The method used to determine when to upgrade a Process to a higher Priority Queue.
4) The method used to determine when to demote a Process to a lower Priority Queue.
5) The method used to determine when Queue a process will enter when that Process needs service.

Eg., Consider a multilevel feedback Queue scheduler with three Queues, numbered from 0 to 2. The scheduler 1st executes all Process in Queue 0. Only when Queue 0 is empty will it execute Processes in Queue 1. Similarly, Processes in Queue 2 will only be executed if Queues 0 and 1 are empty. A Process that arrives for Queue 1 will

Preempt a process in Queue 2. A process in Queue 1 will Preempt a process ~~in Queue 2~~ arriving for Queue 0.

A Process entering the ready queue is put in Queue 0. A Process in Queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of Queue 1. If Queue 0 is empty, the Process at the head of Queue 1 is given a Quantum of 16 milliseconds. If it does not complete, it is Preempted and is put into Queue 2. Processes in Queue 2 are run on as FCFS basis, but are run only when Queues 0 and 1 are empty.

## (9) Multiple - Processor scheduling :

In multiple - Processor scheduling multiple CPU's are available and hence Load sharing becomes Possible. However multiple Processor scheduling is more complex as compared to single Processor scheduling. In multiple Processor scheduling there are cases when the Processors are identical ie., Homogeneous, in terms of their functionality, we can use any Processor available to run any Process in the Queue.

Approaches to Multiple - Processor scheduling.
- Processor affinity.
- Load Balancing.
- Multicore Processors.
- Virtualization and scheduling.

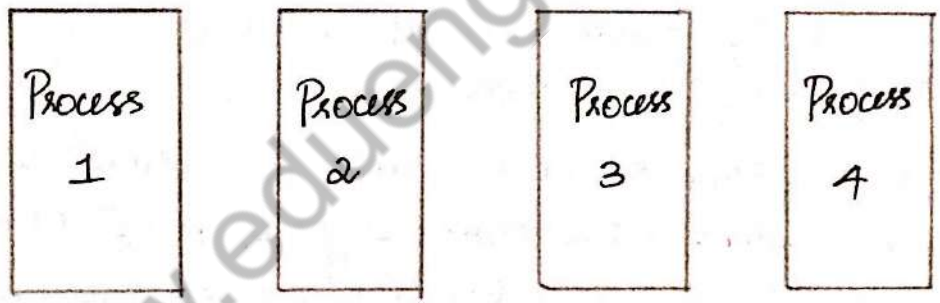
* Approaches to Multiple-Processor scheduling :-

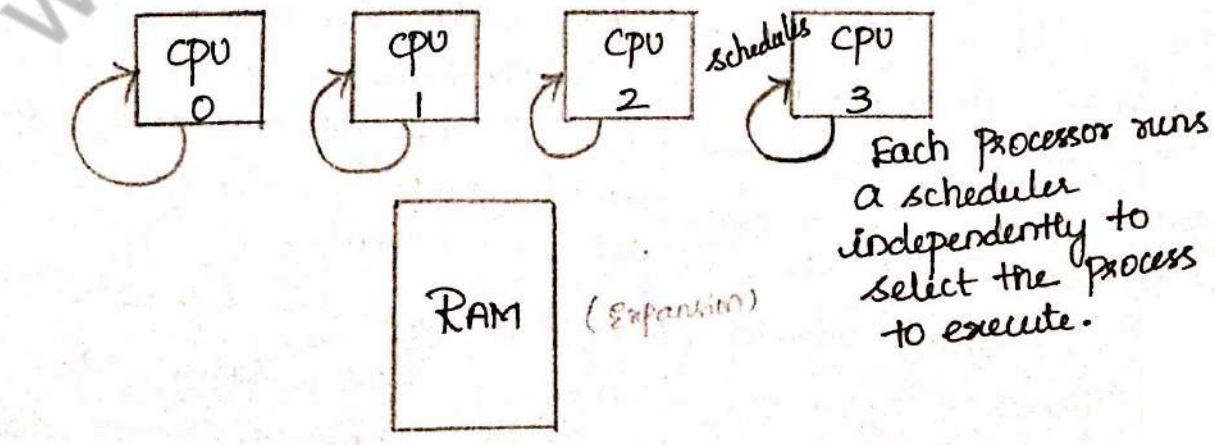
One approach is when all the scheduling decisions and I/O Processing are handled by a single Processor which is called the Master Server and the Other Processors executes only the user code. This entire scenario is called Asymmetric multiprocessing.

A Second approach uses Symmetric multiprocessing where each Processor is self scheduling. All Processes may be in a common ready Queue (or) each Processor may have its own private Queue for ready Processes.

The scheduling Proceeds further by having the Scheduler for each processor examine the ready Queue and select a process to execute.

Multiprocessor scheduling.



* Processor Affinity: It means a process has an affinity for the processor on which it is currently running.

When a process runs on a specific processor there are certain effects on the cache memory. If the process migrates to another processor, the contents of the cache memory must be invalidated for the first processor and the cache for the second processor must be repopulated. Because of high cost of invalidating and repopulating caches, most of the SMP (symmetric multiprocessing) systems try to avoid migration of processes from one processor to another and try to keep a process running on the same processor. This is known as Processor affinity.

- Soft Affinity.
- Hard Affinity.

Soft Affinity: When an operating system has a policy of attempting to keep a process running on the same processor but not guaranteeing it will do so, this situation is called soft affinity.

Hard Affinity: Some systems such as Linux also provide some system calls that support Hard Affinity which allows a process to migrate between processors.

* Load Balancing:

It is the phenomena which keeps the workload evenly distributed across all processors in an SMP systems. Load Balancing is necessary only on systems where each processor has its own private queue of process which are eligible to execute.

Load Balancing is unnecessary because once a processor becomes idle it immediately extracts a runnable process from the common run queue.

On SMP it is important to keep the workload balanced among all Processors to fully utilize the benefits of having more than one Processor else one (or) more than one Processor ~~else one~~ will sit idle while other Processors have high workloads along with lists of Processors awaiting the cpu.

There are two general approaches to load balancing;

* Push migration
* Pull migration.

In Push migration a task routinely checks the load on each processor and if it finds an imbalance then it evenly distributes load on each processors by moving the Processes from overloaded to idle or less busy Processors.

In Pull migration when an idle Processor pulls a waiting task from a busy Processor for its execution.

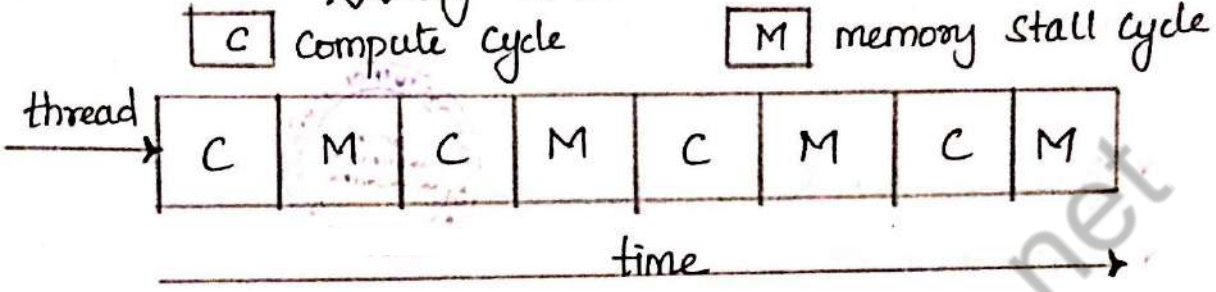
* Multi-core Processors :-

In multi-core processors multiple Processor cores are places on the same physical chip. Each core has a register set to maintain its architectural state and thus appears to the operating system as a separate physical processor.

SMP systems that use multicore Processors are faster and consume less power than systems in which each processor has its own physical chip.

However multicore Processors may complicate the scheduling problems. When processor accesses memory then it spends a significant amount of time waiting for the data to become available. This situation is called Memory stall.

Memory stall.




It occurs for various reasons such as cache miss, which is accessing the data that is not in the cache memory. In such cases the processor can spend upto fifty Percent of its time waiting for data to become available from the memory.

To solve this Problem recent hardware designs have implemented multithreaded Processor cores in which two or more hardware threads are assigned to each core. Therefore if one thread stalls while waiting for the memory, core can switch to another thread.

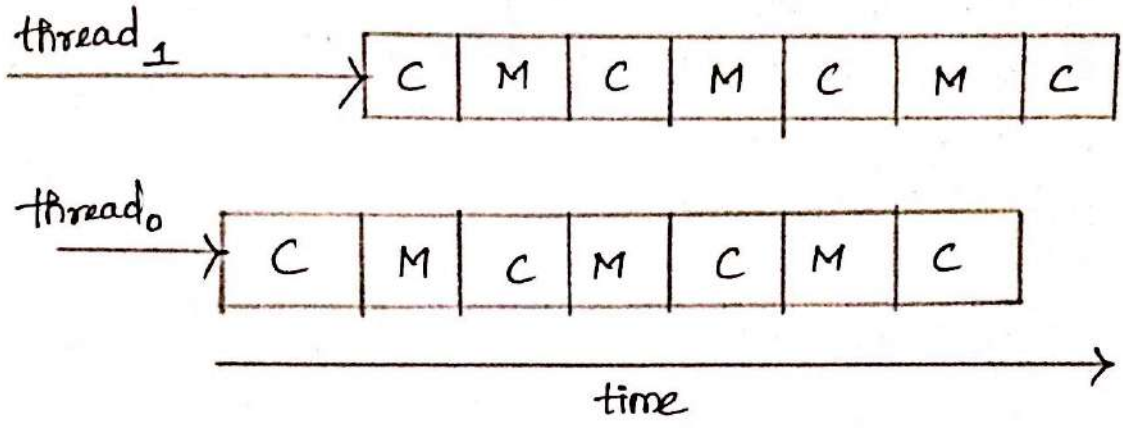
'2' Ways to multi-thread a Processor.

→ Coarse - Grained Multithreading.
→ Fine - Grained Multithreading.

Coarse - Grained Multithreading :- A thread executes on a processor until a long latency event such as a memory stall occurs.

Fine - Grained Multithreading :- It switches between threads at a much finer level mainly at the boundary of an instruction cycle.

Multithreaded multicore system:




Virtualization and ~~Threading~~ Scheduling :-

In this type of multiple - processor scheduling even a single cpu system acts like a multiple-Processor system. Most virtualized environments have one host or operating system and many guest operating systems.

The Host OS creates and manages the virtual machines and each virtual machine has a guest OS installed and applications running within that guest.

Each guest OS may be assigned for specific use cases, applications, and users, including time sharing or even real-time operation.

Virtualization can thus undo the good scheduling - algorithm efforts of the operating systems within virtual machines.

(10) Real-time Scheduling :-

Tasks or processes attempt to control or react to events that take place in the outside world. These event occur in real time and process must be able to keep up with them.

Real time systems examples are

a) control of laboratory experiments.
b) Process control in industrial plants.
c) Robotics.
d) Air traffic control.
e) Telecommunications.
f) Military command and control systems.

Classification of real time task:

1) Hard real time
2) Soft real time
3) Aperiodic task
4) Periodic task

A hard real time task must meet its deadline, otherwise it will cause undesirable damage or a fatal error to the system.

A soft real time: Deadline is desirable but not mandatory, work is continued even if deadline missed.

Aperiodic task has deadline or constraint for start (or) finish times (or) both.

Periodic task: Requirement may be stated as once per period T (or) exactly T units apart.

Real time OS characteristics:

(i) Deterministic
(ii) Responsiveness
(iii) User control
(iv) Reliability
(v) Fail soft operation.

(i) Deterministic:

* Operations are performed at fixed, Predetermined times or within Predetermined time intervals.

* Concerned with how long the operating system delays before acknowledging an interrupt and there is sufficient capacity to handle all the requests within the required time.

(ii) Responsiveness:

* How long, after acknowledgement, it takes the Operating system to service interrupt.

* Includes amount of time to begin execution of the interrupt.

* Includes the amount of time to perform the interrupt.

(iii) User Control:

* It is essential to allow the user fine grained control over task priority. The user should be able to distinguish between hard and softs tasks and to specify relative priorities within each class.

* It also allow the user to specify the use of paging or process swapping.

(iv) Reliability:

* A Real time system is responding to and controlling events in real time. So loss or degradation of performance may have catastrophic consequences, ranging from financial loss to major equipment damage and even loss of life.

* A processor failure in a multiprocessor normal time system may result in a reduced level of service until the failed processor is repaired (or) replaced.

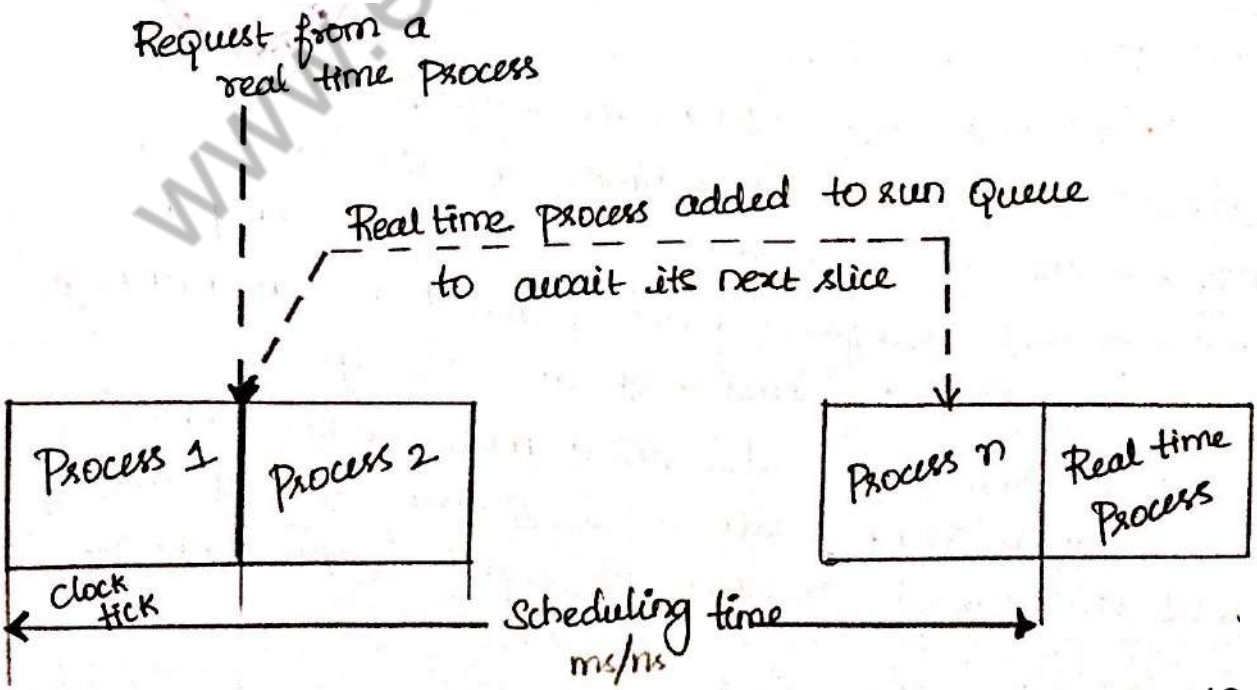
Features of Real time OS:

1) Fast context switch
2) small size.
3) Ability to respond to external interrupts quickly.
4) Multitasking with Ipc tools such as semaphores, signals and events.
5) Preemptive scheduling based on priority.
6) Special alarms and timeouts.
7) Minimization of intervals during which interrupts are disabled.
8) Use of special sequential files that can accumulate data at a fast rate.

Classes of Real time scheduling :-

In a preemptive scheduler that uses simple round robin scheduling; a real time task would be added to the ready queue to awaits its next time slice.

RR preemptive scheduler.



Classes of Algorithms :-

1) Static table driven approaches.
2) Static priority driven Preemptive approaches.
3) Dynamic planning based approaches.
4) Dynamic best effort approaches.

⇒ Static table driven approach is applicable to tasks that are periodic. These perform a static analysis of feasible schedules of dispatching. The result of the analysis is scheduled that determines, at run time, when a task must begin execution.

⇒ In static priority driven preemptive scheduling, static analysis is performed. But no schedule is drawn up. The analysis is used to assign priority to tasks, so that a traditional Priority driven Preemptive scheduler can be used. One example of this approach is the rate monotonic algorithm, which assigns static priorities to tasks based on their periods.

⇒ In dynamic Planning based scheduling, feasibility is determined at run time rather than off line prior to the start of execution.

Dynamic best effort approaches : No feasibility analysis is performed. The system tries to meet all deadlines and aborts any started process whose deadline is missed.

(11) Threads [MD-19]

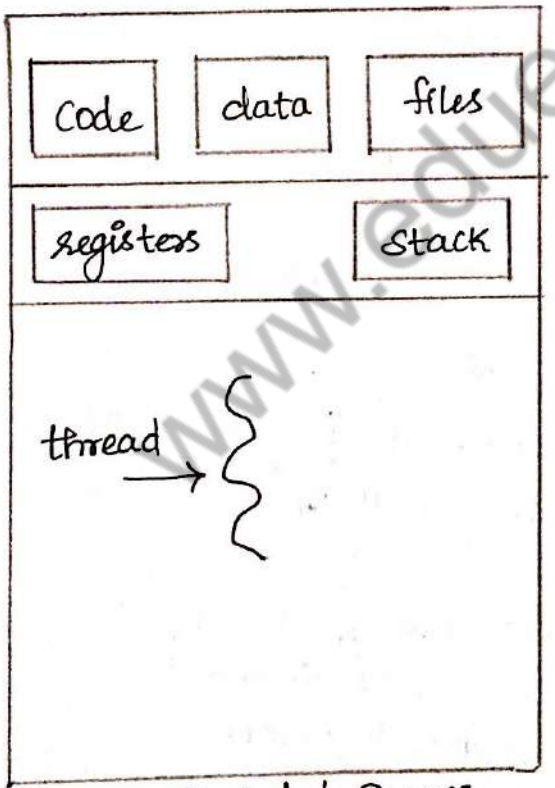
Overview.

* Motivation
* Benefits
* Multicore Programming

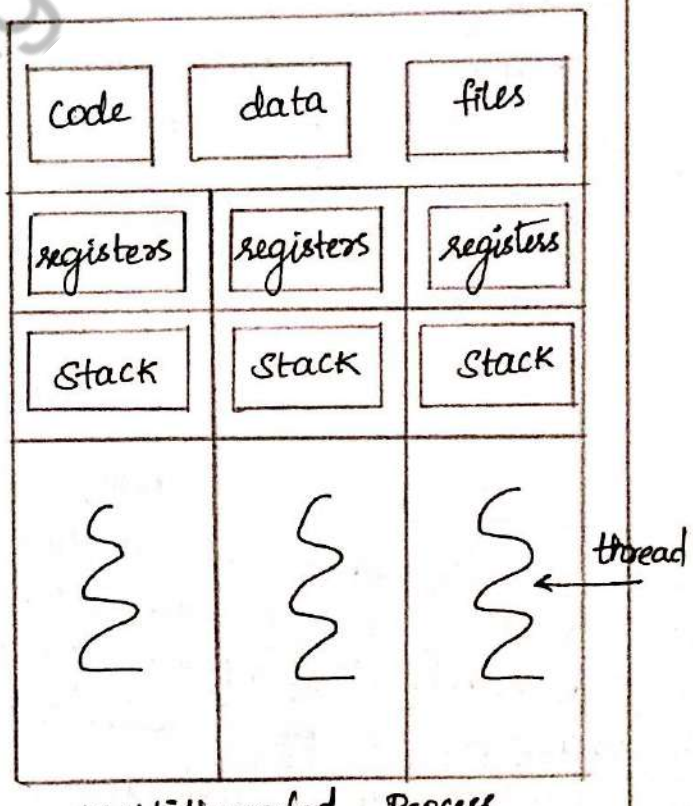
A thread is a basic unit of CPU utilization; it comprises a thread ID, a Program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating system resources, such as open files and signals.

A traditional (or) Heavy weight process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.

Single threaded and multithreaded Process.




Single-threaded Process

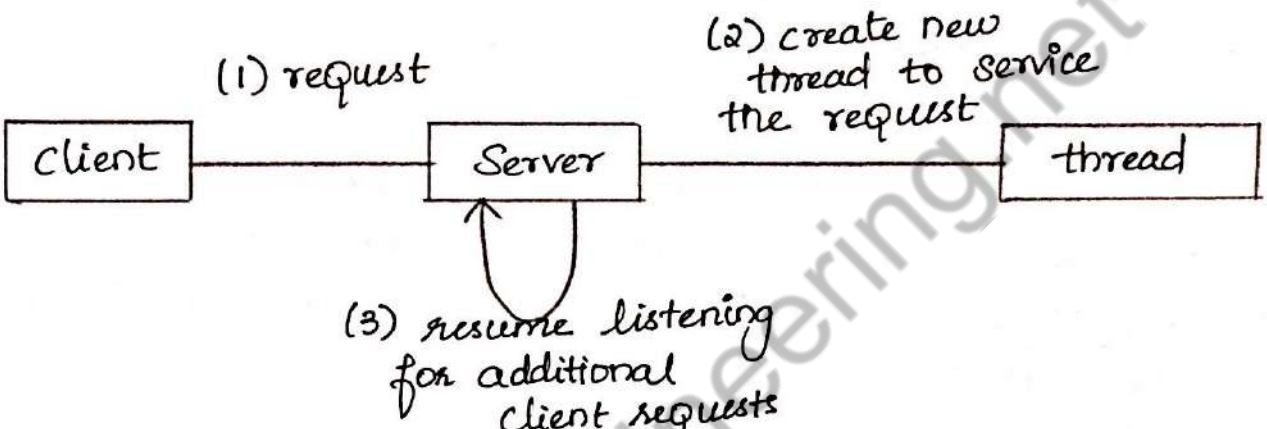



Multithreaded Process

* Motivation:

Many software packages that run on modern desktop PCs are multithreaded. Threads also play vital role in remote Procedure call (RPC), systems. finally, most operating system kernels are now multithreaded; several threads operate in the kernel, and each thread Performs a specific task, such as managing devices (or) interrupt handling.

Multithreaded Server architecture.




Eg., Solaris creates a set of threads in the kernel Specifically for interrupt handling. Linux uses a kernel thread for managing the amount of free memory in the system.

* Benefits :-

The benefits of multithreaded programming can be broken down into four major categories.

(1) Responsiveness
(2) Resource sharing
(3) Economy
(4) Scalability.

(1) Responsiveness :

Multithreading an interactive application may allow a program to continue running even if part

of it is blocked (or) is performing a lengthy operation, thereby increasing responsiveness to the user.

(2) Resource sharing :

Processes can only share resources through techniques such as shared memory and message passing.

(3) Economy :

Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads.

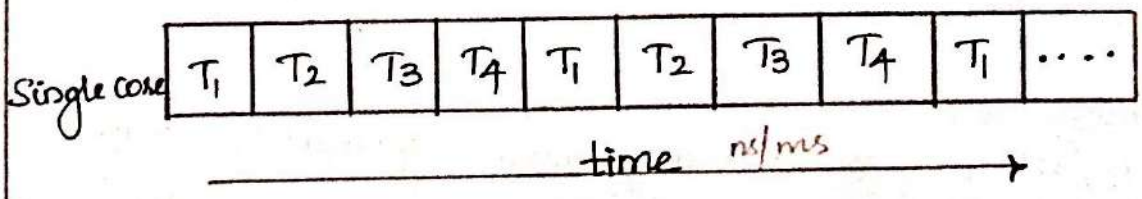
(4) Scalability :

The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores.

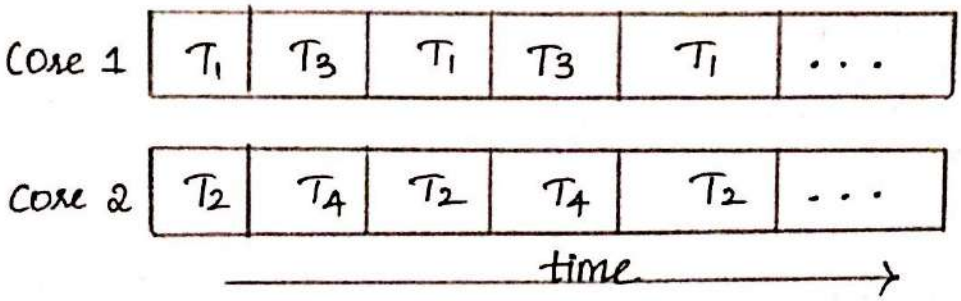
* Multicore Programming :

Earlier in the history of computer design, in response to the need for more computing performance, single-cpu systems evolved into multi-cpu systems. A more recent, similar trend in system design is to place multiple computing cores on a single chip. Each core appears as a separate processor to the operating whether the cores appear across cpu chips or within cpu chips, we call these systems as multicore (or) multiprocessor systems.

Concurrent execution on a single-core system.



Concurrent execution on a multiple system.




Multithreaded Programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency. Consider an application with four threads. On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time because the processing core is capable of executing only one thread at a time. On a system with multiple cores, however, concurrency means that the threads can run in parallel, because the system can assign a separate thread to each core. Notice the distinction between parallelism and concurrency in this discussion.

A system is parallel if it can perform more than one task simultaneously. In contrast, a concurrent system supports more than one task by allowing all the tasks to make progress.

## (13) Multithreading models:

A relationship must exist between user threads and kernel threads. In this section, we look at three common ways of establishing such a relationship.

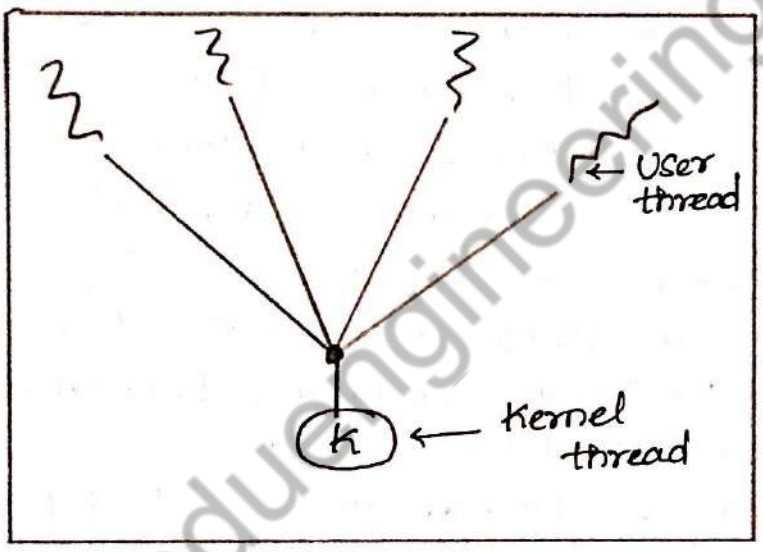
* Many-to-one
* one-to-one
* many-to-many.

Many-to-one:

→ Many user-level threads mapped to single kernel thread.

→ Examples:

→ Solaris Green Threads.
→ GNU portable Threads.



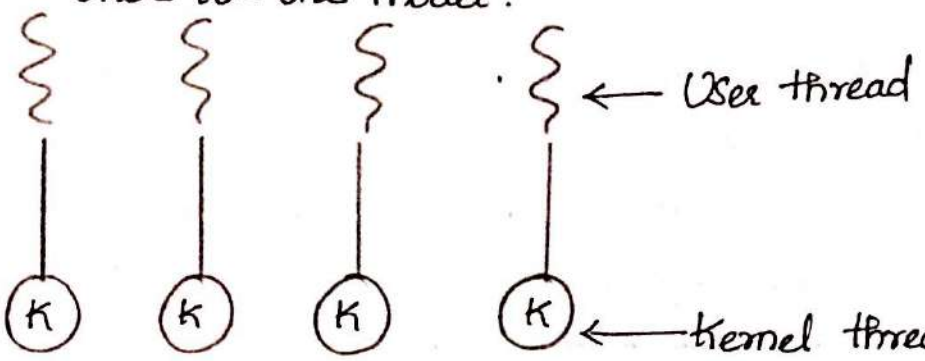

Thread management is done by the thread library in user space, so it is efficient; but the entire Process will block if a thread makes a blocking system call. Also, because (only one thread can access the kernel at a time,) multiple threads are unable to run in parallel on multiprocessors.

* one-one model:

Each user-level thread maps to kernel thread.

Eg; Windows NT/XP/2000
Linux
Solaris 9 and later

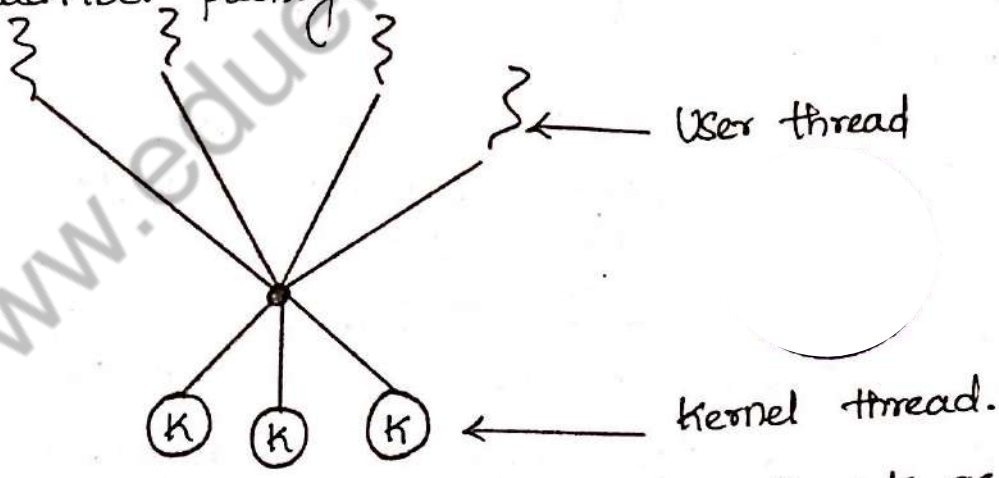
One-to-one model:




(It allows multiple threads to run in Parallel on multiprocessors.) The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.

* Many-to-many model:

(Allows many user level threads to be mapped to many kernel threads.) Allows the operating system to create a sufficient number of kernel threads. Solaris prior to version 9. Windows NT/2000 with the ThreadFiber package.




Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also called as two-level model.

## 14) Threading Issues :

There are a variety of issues to consider with multithreaded Programming.

- Semantics of fork() and exec() system calls.
- Thread cancellation.
  * Asynchronous (or) deferred.
- Signal handling.
  * Synchronous and asynchronous
- Thread pooling.
- Thread - specific data.
  * Create facility needed for data private to thread.

- Semantics of fork() and exec() :-

• Recall that when fork() is called, a separate, duplicate process is created.

• How should fork() behave in a multithreaded Program?
  - Should all threads be duplicated?
  - Should only the threads that made the call to fork() be duplicated?

• In some systems, different versions of fork() exist depending on the desired behavior.
  - Some UNIX Systems have fork1() and forkall()

✓ fork1() only duplicates the calling thread.

✓ forkall() duplicates all of the threads in a Process.

- In a POSIX - compliant system, fork() behaves the same as fork1().

Semantics of fork() and exec()

- The exec() system call continues to behave as expected.

Replaces the entire process that called it, including all threads.

If planning to call exec() after fork(), then there is no need to duplicate all of the threads in the calling process.

* All threads in the child process will be terminated when exec() is called.

* Use fork1(), rather than forkall() if using in conjunction with exec()

* Thread Cancellation :

(It is the act of terminating a thread before it has completed.)

Eg., clicking the stop button on your web browser will stop the thread that is rendering the web page.

The thread to be cancelled is called the target thread. Threads can be cancelled in a couple of ways

* Asynchronous Cancellation :

Terminates the target thread immediately. Thread may be in the middle of writing data... not so good.

* Deferred Cancellation : It allows the target thread to periodically check if it should be cancelled. Allows thread to terminate itself in an orderly fashion.

* Signal Handling:

Signals are used in UNIX systems to notify a process that a particular event has occurred.

CTRL-C is an example of an asynchronous signal that might be sent to a process.

* An asynchronous signal is one that is generated from outside the process that receives it.

Divide by 0 is an example of a synchronous signal that might be sent to a process.

* A synchronous signal is delivered to the same process that caused the signal to occur.

All signals follows the same basic pattern:

* A signal is generated by particular event.
* The signal is delivered to a process.
* The signal is handled by a signal handler (all signals are handled exactly once).

Signal handling is straightforward in a single-threaded process.

* The one (and only) thread in the process receives and handles the signal.

In a multithreaded program, where should signals be delivered?

Options:

(1) Deliver the signal to the thread to which the signal applies.
(2) Deliver the signal to every thread in the process.
(3) Deliver the signal only to certain threads in the process.
(4) Assign a specific thread to receive all signals for the process.

Option 1: Deliver the signal to the thread to which the signal applies.

Most likely option when handling synchronous signals (eg., only the thread that attempts to divide by zero needs to know of the error).

Option 2: Deliver the signal to every thread in the process.

likely to be used in the event that the process is being terminated. (eg., a CTRL-C is sent to terminate the process, all threads need to receive this signal and terminate).

* Thread pools:

⇒ In applications where threads are repeatedly being created/destroyed "thread pools" might provide a performance benefit.

Eg., A server that spawns a new thread each time a client connects to the system and discards that thread when the client disconnects.

⇒ A thread pool is a group of threads that have been pre-created and are available to do work as needed.

- ✓ Threads may be created when process starts.
- ✓ A thread may be kept in a Queue until it is needed.
- ✓ After a thread finishes, it is placed back into a Queue until it is needed again.
- Avoids the extra time needed to spawn new threads when they are needed.

Advantages of thread pools:

- Typically faster to service a request with an existing thread than create a new thread (Performance benefit).
- Bounds the number of threads in a process.

* Thread-specific data: In some applications it may be useful for each thread to have its own copy of data.

→ Also referred as Thread-local storage (or) Thread-static variables.

In C#

```
Class FooBar {
    [ThreadStatic] static int foo;
}
```

Eg., Windows XP.

Implements threads using the one-to-one thread model.

Also implements a fiber that uses a many-to-many model.

Linux oftentimes uses the term task rather than Process or thread.

## (15) Process Synchronization:

1) Concurrent access to shared data may result in data inconsistency.

2) Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

3) Shared memory solution to bounded-buffer problem allows atmost n-1 items in buffer at the same time. A solution, where all N buffers are used is not simple.

4) Suppose that we modify the Producer-consumer code by adding a variable counter, initialized to 0 and increment it each time a new item is added to the buffer.

5) Race condition : The situation where several Processes access - and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.

6) To Prevent race conditions, concurrent Processes must be Synchronized.

## (16) Critical Section Problem :

Consider a system consisting of n Processes $\{P_0, P_1, \ldots P_{n-1}\}$. Each Process has a segment of code, called a critical section, in which the Process may be changing common variables, updating a table, writing a file and so on.

The important feature of the system is that, when one Process is executing in its critical section, no other process is to be allowed to execute in its critical section. ie., no two Processes are executing in their critical sections at the same time.

Each Process must request Permission to enter into critical section. The section of code Implementing this request is the entry section, and followed by an exit section. The remaining code is the remainder section.

The general structure of a typical process $P_i$ is shown.

```
do
{
    entry section
        critical section
    exit section
        remainder section
} while (TRUE);
```

A Solution to the critical section problem must satisfy the following three requirements.

* Mutual Exclusion
* Progress
* Bounded waiting.

Mutual Exclusion : If Process $P_i$ is executing in its critical Section, then no other processes can be executing in their critical sections.

Progress : If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

Bounded Waiting : There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Two general approaches are used to handle critical sections in operating systems:

* Preemptive kernels
* non-preemptive kernels.

A Preemptive Kernel allows a process to be Preempted while it is running in kernel mode.

A non-preemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

17) Synchronization Hardware:

Many systems Provide hardware support for critical sections. Many systems provide special hardware instructions that allows us either to test and modify the content of a word set. If two test and Set instructions are executed simultaneously, they will be executed sequentially in some arbitrary order. The Process enter its critical section only if waiting [i] == false and key == false.

Eg;

```
While (true)
{
    waiting [i] = true;
    key = true;
    while (waiting [i] && key)
    key = testandset (lock);
    waiting [i] = false;
    Critical section
    j = j+1;
    // waiting [j] = false;
}
```

Uniprocessors:

- could disable interrupts.
- currently running code would execute without Preemption.
- generally too inefficient on multiprocessor system.
  * Modern machine provide special atomic hardware instructions.
  * Atomic – Non interruptable.

- Either test memory word and set value.
- (or) swap contents of two memory words.

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (TRUE);
```

(18) Mutex Locks:

In computer programming, a mutual exclusion object (mutex) is a program object that allows multiple program threads to share the same resources, such as file access, but not simultaneously.

Strictly speaking, a mutex is locking mechanism used to synchronize access to a resource. Only one task (can be a thread or process based on OS abstraction) can acquire the mutex. It means there is ownership associated with mutex, and only the owner can release the lock (mutex).

The mutex is set to unlock when the data is no longer needed or the routine is finished.

## (19) Semaphores :

The hardware-based solutions to the critical-section problem presented are complicated for application programmers to use. To overcome this difficulty, we can use a synchronization tool called a Semaphore.

A Semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations :

✓ wait () and signal ().

The wait () operation was originally termed "P"
The signal () operation was originally termed "V".

The definition of wait () is as follows ;

```
wait (s) {
        while S <= 0
          ; // no-op
      S--;
}
```

The definition of signal () is as follows :

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

When one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

* Usage
* Implementation
* Deadlocks and Starvation
* Priority Inversion

* Usage : operating systems often distinguish between counting and binary semaphores. The value of a counting semaphores can range over an unrestricted domain. The value of a binary semaphores can range only between 0 and 1. On some systems, binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion.

Semaphores also used to solve various synchronization Problems, for example, consider two concurrently running Processes : $P_1$ with a statement $S_1$ and $P_2$ with a statement $S_2$. Suppose we require that $S_2$ be executed only after $S_1$ has completed.

We can implement this scheme readily by letting $P_1$ and $P_2$ share a common semaphore synch, initialized to 0, and by inserting the statements.

```
S1;
signal (synch);
```

in Process $P_1$ and the statements

```
wait (synch);
S2;
```

in Process $P_2$. Because synch is initialized to 0, $P_2$ will execute $S_2$ only after $P_1$ has invoked signal (synch), which is after statement $S_1$ has been executed.

Implementation:

The main disadvantage of the semaphore definition given here is that it requires busy waiting. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This conditional looping is clearly a problem in a real multiprogramming system, where a single cpu is shared by many processes.

Mutual-exclusion implementation with semaphores.

```
do {
    wait (mutex);
    // critical section
    signal (mutex);
    // remainder section
} while (TRUE);
```

The semaphore discussed so far requires a busy waiting. That is if a process is in critical-section, the other process that tries to enter its critical-section must loop continuously in the entry code.

To overcome the busy waiting problem, the definition of the semaphore operations wait and signal should be modified.

⇒ When a process executes the wait operation and finds that the semaphore value is not positive, the process can block itself. The block

Operation places the process into a waiting Queue associated with the semaphore.

⇒ A Process that is blocked waiting on a semaphore should be restarted when some other Process executes a signal operation. The blocked Process should be restarted by a wakeup operation which put that process into ready Queue.

To Implement the semaphore, we define a semaphore as a record as;

```
typedef struct {
    int value;
    struct process *L;
} Semaphore;
```

* Deadlock and Starvation:

The implementation of a semaphore with a waiting Queue may result in a situation where two (or) more processes are waiting indefinitely for an event that can be caused only by one of the waiting Processes.

The event in Question is the execution of a signal() operation. When such a state is reached, these Processes are said to be deadlocked.

To illustrate this, we consider a system consisting of two processes, $P_0$ and $P_1$, each accessing two semaphores, S and Q, set to the value 1;

| $P_0$ | $P_1$ |
|---|---|
| wait (s); | wait (Q); |
| wait (Q); | wait (s); |
| : | : |
| signal (s); | signal (Q); |
| signal (Q); | signal (s); |

Suppose that $P_0$ executes wait(S) and then $P_1$ executes wait(Q). When $P_0$ executes wait(Q), it must wait until $P_1$ executes signal(Q). Similarly, when $P_1$ executes wait(S), it must wait until $P_0$ executes signal(S). Since these signal() operations cannot be executed, $P_0$ and $P_1$ are deadlocked.

Another problem related to deadlock is indefinite blocking or starvation, a situation where a process wait indefinitely within the semaphore. Indefinite blocking may occur if we add or remove processes from the list associated with a semaphore in LIFO order.

Types of Semaphores.

* Counting Semaphore - any positive integer value.
* Binary Semaphore - integer value can range only between 0 and 1.

* Priority Inversion:

A scheduling challenge arises when a higher-priority process needs to read (or) modify kernel data that are currently being accessed by a lower-priority process - or a chain of lower-priority processes. This problem is known as priority Inversion.

It occurs only in systems with more than two priorities, so one solution is to have only two priorities.

(20) Classical Problems of Synchronization : [MD-19]

Here, we Present a number of synchronization Problems as examples of a large class of concurrency Control problems. These Problems are used for testing nearly every newly pooposed synchronization scheme. In our solutions to the poblems, we use semaphores for synchronization.

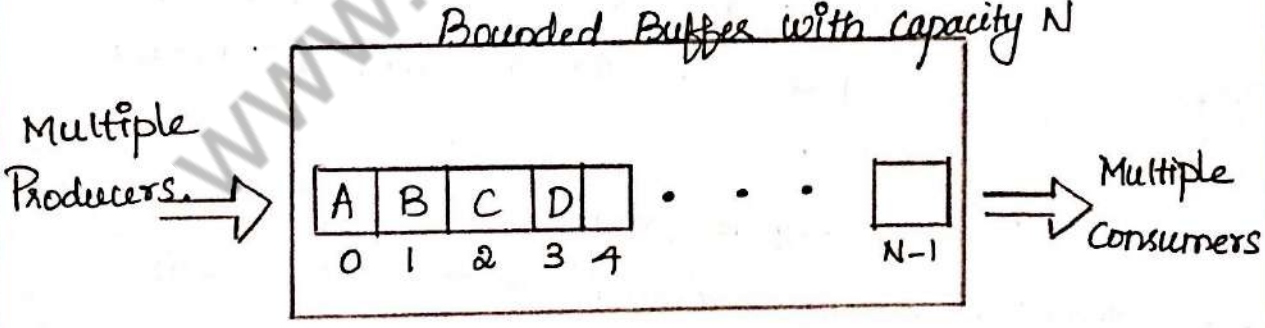
* The Bounded - Buffer poblem.
* The Readers - Writers poblem.
* The Dining - philosophers poblem.

* The Bounded - Buffer problem :

The bounded buffer Problems (ie the Producer - Consumer problem) is a classic example of Concurrent access to a shared resource. A bounded buffer lets multiple Producers and multiple Consumers Share a single buffer.

Producers must block if the "Buffer is full".

Consumers must block if the "Buffer is Empty".



We assume that the pool consists of n buffers, each capable of holding one item.

The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

The structure of the Producer process :-

```
do {
    ...
    // Produce an item in nextp.
    ...
    wait (empty);
    Wait (mutex);
    ...
    // add nextp to buffer
    ...
    signal (mutex);
    Signal (full);
} while (TRUE);
```

We can interpet this code as the Producer Producing full buffers for the Consumer (or) as the Consumer Producing Empty buffers for the Producer.

The structure of the Consumer process :

```
do {
    wait (full);
    wait (mutex);
    ...
    // remove an item from buffer to nextc
    ...
    signal (mutex);
    signal (Empty);
    ...
    // Consumes the item in nextc
    ...
} while (TRUE);
```

* The Readers - Writers Problem :

Readers : Only reads the data set they do not perform any updates.

Writers : Can both read and write.

Problem : Whenever allows multiple readers to read at the same time. Only one single writer can access the shared data at same time.

R-W and W-W ⇒ leads to crash.

Shared data :-

```
dat set;
Semaphore mutex = 1
Semaphore wrt   = 1
Integer readcount = 0.
```

Structure of writer Process : — read / write

```
do {
    wait (wrt);
    ...
    // writing is Performed
    ...
    signal (wrt);
} while (TRUE);
```

Structure of reader process :

```
do {
    wait (mutex);
    readcount ++;
    if (readcount == 1)
      wait (wrt);
    signal (mutex);
    ...
    // reading is Performed
    ...
```

```
wait (mutex);
readcount --;
if (readcount == 0)
  signal (wrt);
  signal (mutex);
} while (TRUE);
```

Reader - Writer locks are most useful in the following situations :

(i) In applications where it is easy to identify which processes only read shared data and which processes only write shared data.

(ii) In applications that have more readers than writers. This is because reader - writer locks generally require more overhead to establish than semaphores (or) mutual exclusion locks. The increased concurrency of allowing multiple readers compensates for the overhead involved in setting up the reader- writer lock.

* The Dining - philosophers problem :

Consider 5 philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.

In center of table is a bowl of rice, and the table is laid with 5 single chopsticks.

→ When philospher thinks, she does not interact with her colleagues.

→ from time to time, a philosopher gets hungry

and tries to pick up the two chopsticks that are closet to her (the chopsticks that are between her and her left and right neighbors).

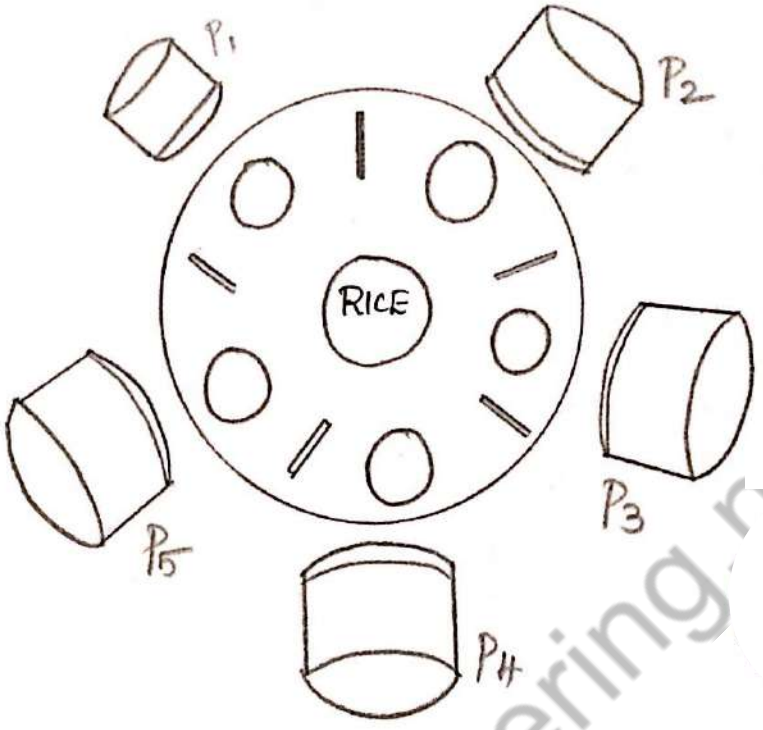
A philosopher may pick up only one chopstick at a time. obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

Problem : Develop an algorithm where no philosopher Staves. ie., every philosopher should eventually get a chance to eat.

The structure of philosopher i.

```
Chopstick [5] initialized to 1
Structure of philosopher i;
do {
    wait (chopstick [i]);
    wait (chopstick [(i+1) %. 5]);
    ...
    // eat
    signal (chopstick [i]);
    signal (chopstick [(i+1) %. 5]);
    ...
    // think
    ...
} while (TRUE);
```

## Solution of the dining philosophers




One simple solution is to represent each chopstick with a semaphore.

Shared data : State [5] (thinking, hungry, eating)

Self [5] : Semaphore : { init to all 0 }

mutex : Semaphore { init to 1 }

left : (i+4) %. 5 { left neighbour }

right : (i+1) %. 5, { right neighbour }

```
Philosopher (i)
{
  State [i] = hungry;
  wait (mutex);
  test [i];
  Signal (mutex);
  if (state [i] != Eating)
  wait (self [i]);
  . . . . .
```

```
. . . Eating . . .
wait (mutex);
Putdown (i);
```

```
void test (int i)
{
  if ((state (left) != eating) && (state [i] == hungry)
                && (state (right) != eating)
  {
     state [i] = Eating;
     signal (self (i));
  }

void putdown (int i)
  {
     state [i] = thinking;
     wait (self (i));
  // test left & right neighbours
     test (left);
     test (right);
  }
```

Several possible remedies to the deadlock problem are listed next.

* Allow at most four philosophers to be sitting simultaneously at the table.
* Allow a philosopher to pick up her chopsticks only if both chopsticks are available.
* Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

(21) Critical regions :-

→ Regions referring to the same shared variable excluded each other in time.

→ When a process tries to execute the region statement, the Boolean expression B is evaluated. If B is true, statement S is executed. If it is false, the process is delayed until B becomes true and no other process is in the region associated with v.

Limitations :

→ Conditional critical regions are still distributed among the program code.

→ They are more difficult to implement efficiently than semaphores.

(22) Monitors :-

Monitors are based on abstract data types. A monitor is a programming language construct that provides equivalent functionality to that of semaphores but is easier to control. A monitor consists of procedures, the shared object and administrative data.

Characteristics of a monitor :-

(1) Only one process can be active within the monitor at a time.

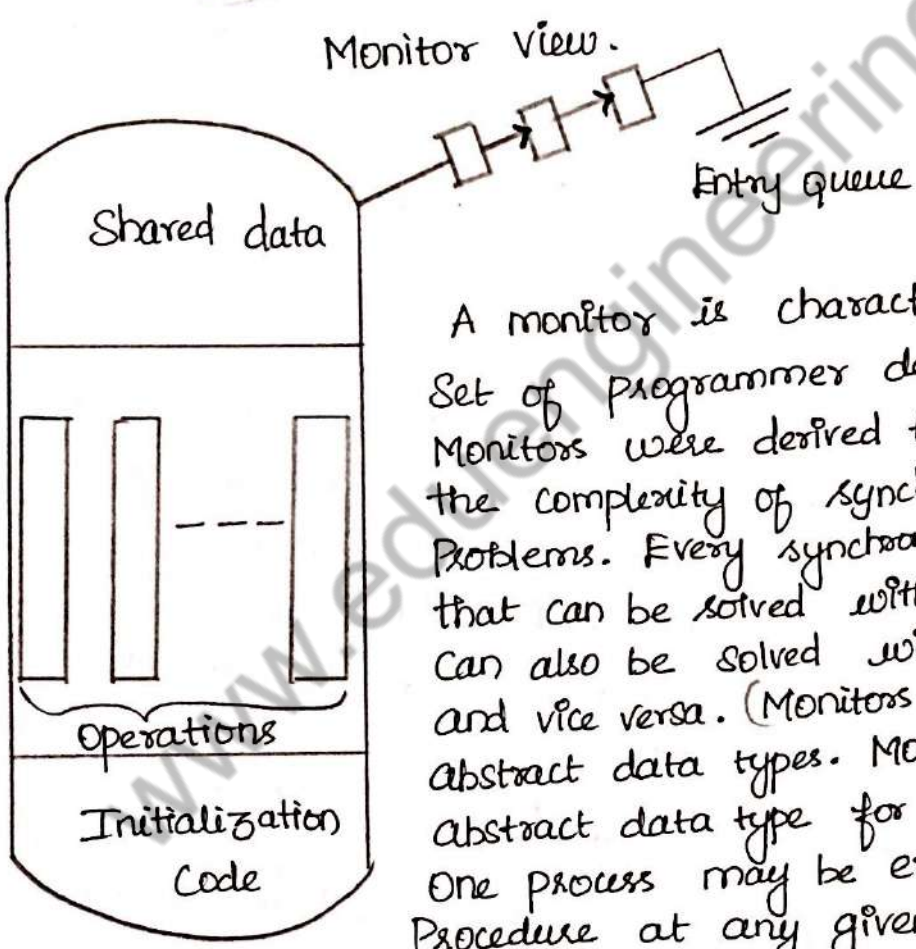
(2) The local data variables are accessible only by the monitor's procedures and not by any external procedure.

(3) A process enters the monitor by invoking one of its procedures.

(4) Monitor provides high-level of synchronization. The synchronization of process is accomplished via two special operations namely, wait and signal, which are executed within the monitors procedures.

(5) Monitors are a high level data abstraction tool combining three features:

(1) Shared data
(2) Operation on data
(3) Synchronization, scheduling.




A monitor is characterised by a set of programmer defined operators. Monitors were derived to simplify the complexity of synchronization problems. Every synchronization problem that can be solved with monitors can also be solved with semaphores, and vice versa. (Monitors are based on abstract data types. Monitor is an abstract data type for which only one process may be executing a procedure at any given time. Processes during enter to monitor when it is already in use must wait. This waiting is automatically managed by the monitor.) The above diagram shows the schematic view of a monitor.

A monitor is a software module consisting of one (or) more procedures, an initialization sequence and local data.

Syntax of monitor;

```
Monitor  monitor_name
{
   declaration of shared variable
   Procedure body P1()
   {
      ---
      ---
   }
   P2()
  {
    procedure body
      ---
      ---
   }
      :
      :
  Pn()
{
     Procedure body
   }  ---
      { initialization code
   }  }
```

The monitor construct has been implemented in a number of programming languages. since monitors are a language feature, they are implemented with the help of a compiler. In response to the keywords monitor, condition, signal, wait and notify, the compiler insert little bits of code in the program. The data variables in the monitor can be accessed by only one process at a time. A shared data structure can be protected by placing it in a monitor. The data inside the monitor may be either global to all Procedures within the monitors (or) local to a specific procedure.

A monitor supports synchronization by the use of condition variables that are contained within the monitor and accessible only within the monitor. Two condition variables are;

1) X.wait() : Suspend execution of the calling process on condition X. The monitor is now available for use by another process.

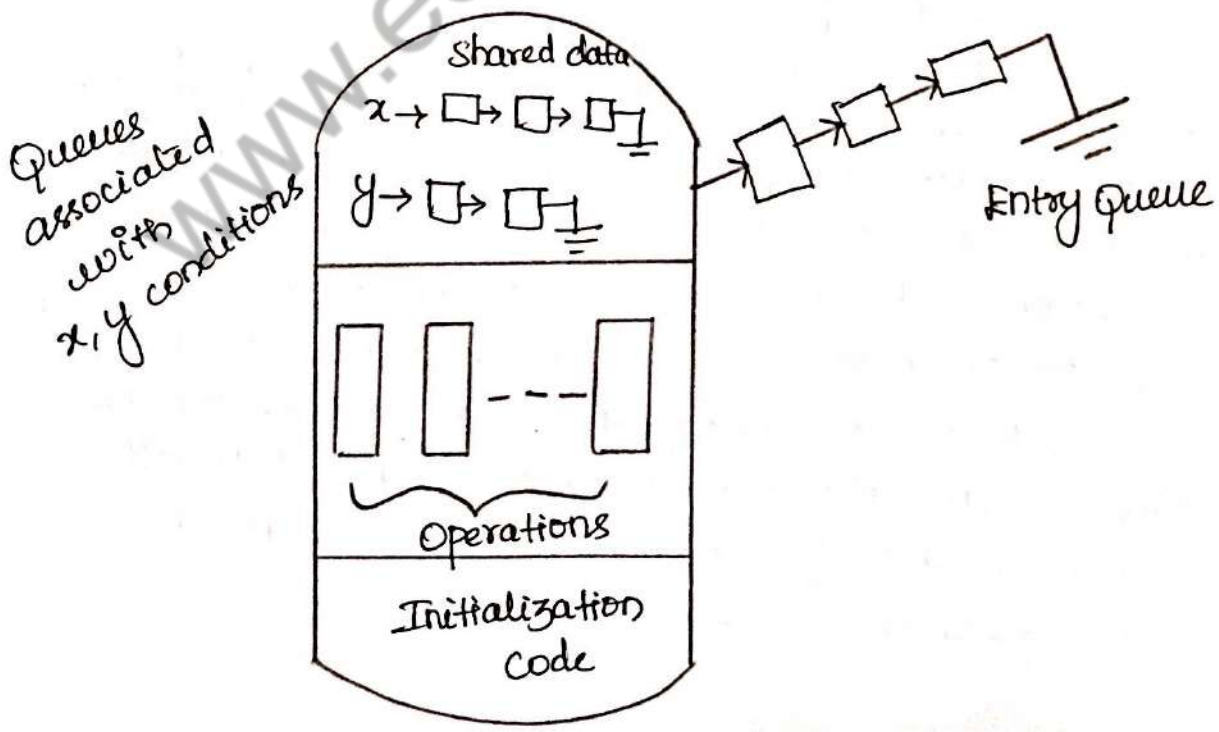
2) X.signal() : Resume execution of some process suspended after a X.wait on the same condition. This operation resumes exactly one suspended process.

A condition variable is like a semaphore, with two differences;

(1) A semaphore counts the number of excess up operations, but a signal operation on a condition variable has no effect unless some process is waiting. A wait on a condition variable always blocks the calling process.

(2) A wait on a condition variable automatically does an up on the monitor mutex and blocks the caller.

Monitor with condition variables.



Monitor with Condition variables

Syntax:

```
Interface condition {
    Public void x.signal()
    Public void x. wait();
}
```

Each condition variable is associated with some logical condition on the state of the monitor. Consider what happens when a consumer is blocked on the nonempty condition variable and producer calls add.

(1) The producer adds the item to the buffer and calls nonempty signal().

(2) The Producer is immediately blocked and the consumer is allowed to continue.

(3) The consumer removes the item from the buffer and leaves the monitor.

(4) The Producer wakes up and since the signal operation wait the last statement is add, leaves the monitor.

They are easier and safer to use but less flexible. Many languages are not supported by the monitor. Java is making monitors much more popular and well known.

By making the mutual exclusion of critical regions automatic, monitors make parallel programming much less error prone than with semaphores.

Drawbacks of monitors :

(1) Major weakness of monitors is the absence of concurrency if a monitor encapsulates the resource, since only one process can be active within a monitor at a time.

(2) There is the possibility of deadlocks in the case of nested monitors calls.

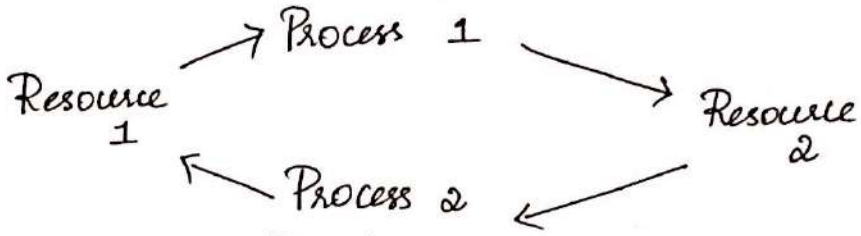
(3) Monitor concept is its lack of implementation most commonly used programming languages.

(4) Monitors cannot easily be added if they are not natively supported by the language.

| Monitors | Semaphore. |
|---|---|
| 1) Monitors are based on abstract data type. | 1) Semaphore is an operating system abstract data type. |
| 2) Monitors were derived to simplify the complexity of synchronization problems by abstracting away details. | 2) Semaphore - level synchronization Primitives are difficult to use for complex synchronization situations. |
| 3) A monitor is a programming language construct that guarantee appropriate access to critical sections. | 3) Semaphores provide a general-purpose mechanism for controlling access to critical sections. |
| 4) Monitor uses condition variables. | 4) A condition variable is like a semaphore, with two differences. |

(23) Deadlock [Feb-19]

A deadlock happens in operating system when two or more processes need some resource to complete their execution that is held by the other process.




Deadlock in OS.

In the above diagram, the process 1 has resource 1 and (needs to acquire) resource 2. Similarly process 2 has resource 2 and needs to acquire resource 1. Process 1 and Process 2 are in deadlock as each of them needs the other's resource to complete their execution but neither of them is willing to relinquish their resources.

(24) System models:

A system consists of a finite number of resources to be distributed among a number of competing processes.

A process must request a resource before using it and must release the resource after using it. A process may request as many resources ~~requested may not ex~~ as it requires to carry out its designated task. A process cannot request three printers, if the system has only two.

Under the normal mode of operation, a Process may utilize a resource in only the following sequence.

Request :

The process requests the resource. If the request cannot be granted immediately (for eg., if the resource is being used by another Process), then the requesting Process must wait until it can acquire the resource.

Use :

The Process can operate on resource (for example, if the resource is a printer), the Process can print on the printer.

Release : The process releases the resource.

(25) Deadlock characterization :

In a deadlock, Processes never finish executing, and system resources are tied up, Preventing other jobs from starting.

* Necessary conditions
* Resource - Allocation graph

* Necessary Conditions : [N/D-19]

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

* Mutual Exclusion
* Hold and wait
* No Preemption
* Circular wait

* Mutual Exclusion :

At least one resource must be held in a nonsharable 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 :

A process must be holding atleast one resource and waiting to acquise additional resources that are currently being held by other processes.

* No Preemption :

Resources cannot be preempted ; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

* Circular wait :

A set $\{P_0, P_1, \ldots P_n\}$ of waiting processes must exist such that $P_0$ is waiting for a resource held by $P_1$, $P_1$ is waiting for a resource held by $P_2, \ldots, P_{n-1}$ is waiting for a resource held by $P_n$, and $P_n$ is waiting for a resource held by $P_0$.

Resource allocation graph :

Deadlocks can be described more precisely in terms of a directed graph called a system resource - allocation graph.

This graph consists of a set of vertices V and set of edges E. The set of vertices V is Partitioned into two different types of nodes :

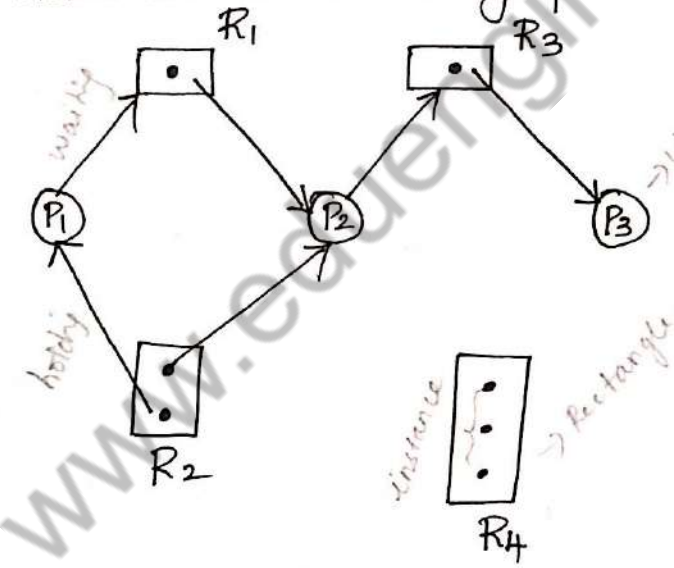
$P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the active processes in the system, and

$R = \{R_1, R_2, R_3, \ldots, R_n\}$, where $R_j$ is a set containing all resource types in the system.

A directed edge from process $P_i$ to resource type $R_j$ is denoted by $P_i \rightarrow R_j$; it signifies that Process $P_i$ has requested an instance of resource type $R_j$ and is currently waiting for that resource.

A directed edge from resource type $R_j$ to Process $P_i$ is denoted by $R_j \rightarrow P_i$; it signifies that an instance of resource type $R_j$ has been allocated to process $P_i$. A directed edge $P_i \rightarrow R_j$ is called a request edge; and a directed edge $R_j \rightarrow P_i$ is called assignment edge.

Resource - allocation graph.




Pictorially, we represent each Process $P_i$ as a circle and each resource type $R_j$ as a rectangle. Since resource type $R_j$ may have more than one instance, we represent each such instance as a dot within the rectangle.

Note that a request edge points to only the rectangle $R_j$, whereas an assignment edge must also designate one of the dots in the rectangle.

* The sets P, R and E.

$$P = \{P_1, P_2, P_3\}$$

$$R = \{R_1, R_2, R_3, R_4\}$$

$$E = \{P_1 \to R_1, P_2 \to R_3, R_1 \to P_2, R_2 \to P_2, R_2 \to P_1, R_3 \to P_3\}.$$

* Resource instances :

- One instance of resource type $R_1$.
- Two instance of resource type $R_2$.
- One instance of resource type $R_3$.
- Three instance of resource type $R_4$.

* Process states :

(i) Process $P_1$ is holding an instance of resource type $R_2$ and is waiting for an instance of resource type $R_1$.

(ii) Process $P_2$ is holding an instance of $R_1$ and an instance of $R_2$ and is waiting for an instance of $R_3$.

(iii) Process $P_3$ is holding an instance of $R_3$.

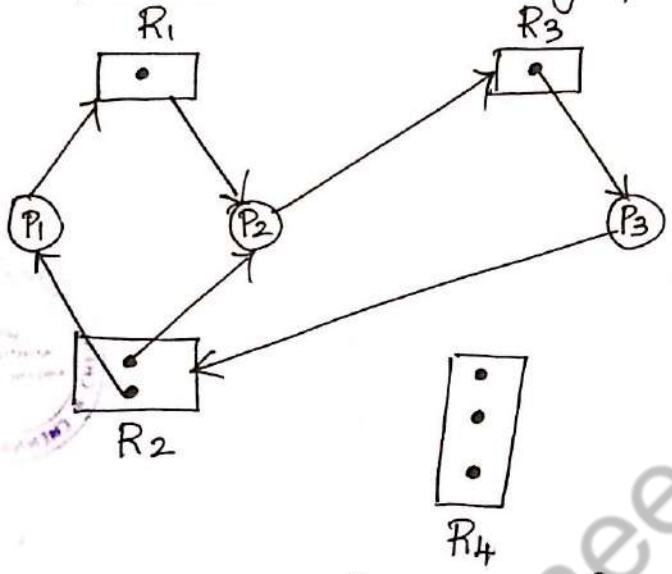
If the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred.

Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both necessary and a sufficient condition for the existence of deadlock.

In the dig., suppose that process $P_3$ requests an instance of resource type $R_2$.

Resource - allocation graph with a deadlock.




Since no resource instance is currently available, a request edge $P_3 \rightarrow R_2$ is added to the graph shown above.

Two minimal cycles exist in the system;

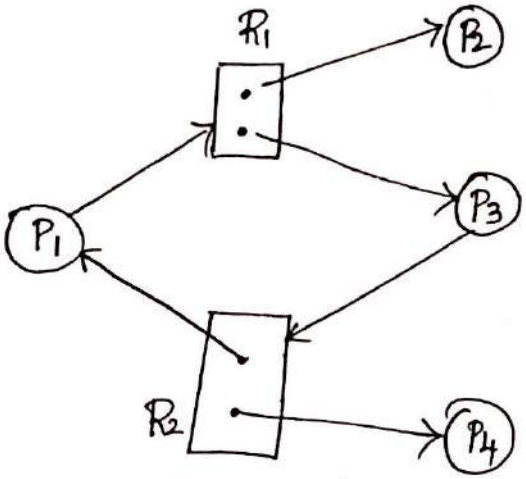
$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

$$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2.$$

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

Now consider the resource - allocation graph, in this example we have a cycle.

When a deadlock graph with a cycle but no deadlock.




$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$.

However there is no deadlock. Observe that Process $P_4$ may release its instance of resource type $R_2$. That resource can then be allocated to $P_3$, breaking the cycle.

If a resource-allocation graph does not have a cycle, then the system is not in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state. This observation is important when we deal with the deadlock problem.

## (26) Methods for Handling deadlocks:

Generally Speaking, we can deal with the deadlock Problem in one of 3 ways;

→ We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.

→ We can allow the system to enter a deadlocked State, detect it, and recover.

→ We can ignore the Problem altogether and Pretend that deadlocks never occur in the system.

27) Deadlock Prevention:

We can Prevent deadlock by eliminating any of the four conditions.

* Mutual Exclusion.
* Hold and wait
* No Preemption
* Circular wait

* Mutual Exclusion :

→ Not required for sharable resources (Read only files)

→ Must hold for non sharable resources.

→ Cannot Prevent deadlocks by denying mutual-exclusion condition (these are non sharable resources).

* Hold and Wait :

→ Must guarantee that whenever a Process requests a resources, it does not hold any other resources.

→ Requires Process to request and be allocated all its resources before it begins execution.

→ System calls requesting resources precede all other system calls.

→ Allow Process to request resources only when the Process has none.

Eg; Process copying data from a tape drive to a disk file, sort the disk file and print the results to a printer.

Low resource utilization; starvation possible.

Ex: Two resources are

* tape drive
* disk
* Printer. All the 3 resources must be used by the process. It hold the Printer for its entire execution, even through it needs the printer only at the end. (Process that copies data from Tape drive to a disk, sorts the file and prints the results to a Printer).

The second method allows the Process to request initially only the DVD drive and disk file. It copies from drive and to the disk then releases both the tape drive and disk file. Process then again request the disk and the pointer. After copying the disk file to the printer, it releases these two resources and terminates.

So both protocols has two disadvantages.

* Resource Utilization is low.
* Starvation is possible.

* No Preemption :

The third necessary condition for deadlocks is that there should be no Preemption of resources that have already been allocated.

- If a Process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released. It won't lead to deadlock but delay some time to complete.
- Preempted resources are added to the list of resources for which the Process is waiting.
- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

- If a Process requests resources, check if resources are available.
- If available, then allocate.
- If not available, check if allocated to some other Process that is waiting.
- If so, Preempt the resources from the waiting Process.
- Allocate to the requesting Process.

* Circular wait :

- Impose a total ordering of all resource types.
- Require that each process requests resources in an increasing order of enumeration.
- $R = \{R_1, R_2, \ldots R_n\}$ be the set of resource types.
- Each resource type is assigned a number.

$F : R \rightarrow N$ where,

$N \rightarrow$ Set of Natural numbers

Eg; F(tape drive) = 1
F(disk drive) = 5
F(Printer) = 12.
} The numbers are in increasing order.

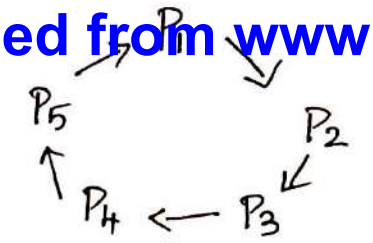
Protocol 1 : Each Process requests only in increasing Order of enumeration. After requesting $R_i$, can request $R_j$ only if $F(R_j) > F(R_i)$.

Eg ; disk drive and then printer.

For several instances of same resource type, a single request for all of them must be issued.

Protocol 2 : When an instance of $R_j$ is to be got, release all instances of $R_i$, if $F(R_i) \geq F(R_j)$

Suppose if circular wait exists in a system, then let $P_0, P_1, \ldots P_n$ be the Processes involved in the circular wait, $P_i$ waiting for a resource held by $P_{i+1}$.




$F(R_0) < F(R_1) < \ldots < F(R_n) < F(R_0)$

which is impossible.

## (28) Deadlock Avoidance :

$P_1 \rightarrow P_1 \rightarrow (P_2) \rightarrow$ wait for another resource

Deadlock is a state in which a process is waiting for the resource that is already used by another process and that another process is waiting for another resource.

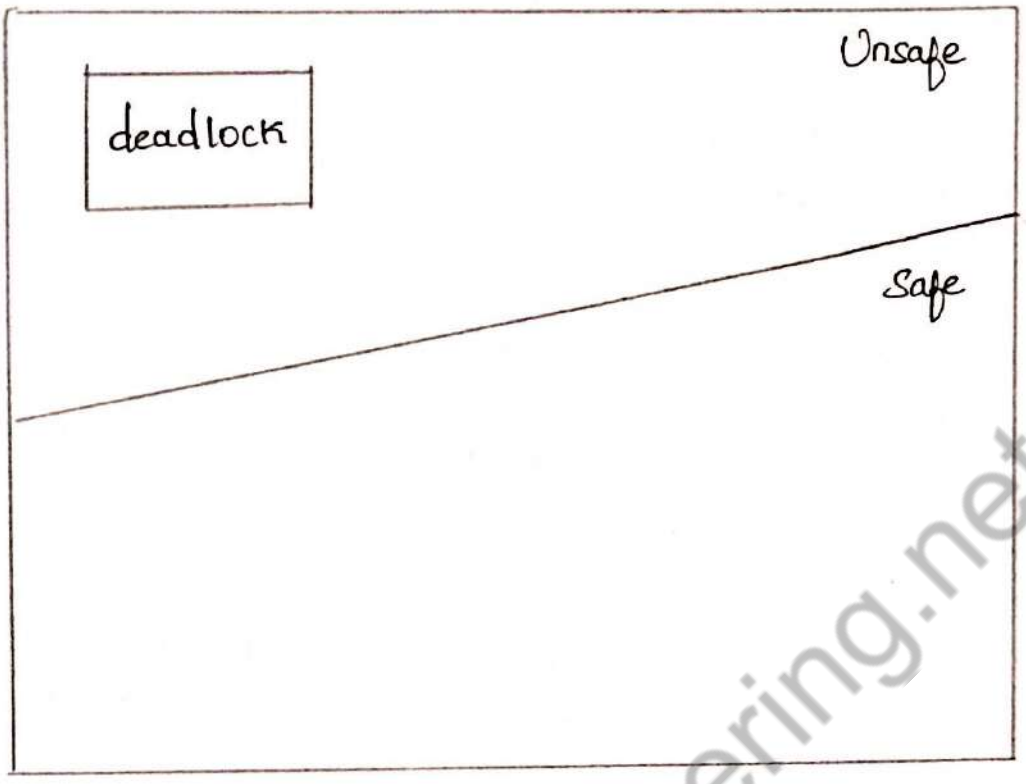
Deadlock can be avoided if certain information about processes are available to the operating system before allocation of resources. For every resource request will cause the system to enter an unsafe state that means this state could result in deadlock. The system then only grants the requests that will lead to safe states.

* Safe State
* Resource - allocation graph algorithm.
* Banker's algorithm.
  - Safety algorithm
  - Resource - Request algorithm
  - Example.

### Safe State :

A state is safe if it is has a bunch of processes and there are enough resources for the 1st process to be finished and after it releases it's resources there are enough resources for the next process to be proceed. There is no chance of deadlock.

Safe, unsafe and deadlocked state spaces.




Unsafe state : A state that may allow deadlock. It is possible for a process to be in an unsafe state, but for this not to result in a deadlock.

Deadlock : No further progress is possible.

In order to determine the condition of next state (safe / unsafe / deadlock) the system must know these information in advance.

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

To avoid deadlock we have to follow a simple rule. If a request causes the next state is to unsafe state or deadlock then we shouldn't Proceed the request.

When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state. System is in safe state if there exists a sequence $\langle P_1, P_2, \ldots P_n \rangle$ of all the processes is the systems such that for each $P_i$, the resource that $P_i$ can still request can be satisfied by currently available resources + resources held by all the $P_j$, with $j < i$.

ie., If $P_i$ resource needs are not immediately available, then $P_i$ can wait until all $P_j$ has finished. When $P_j$ is finished, $P_i$ can obtain needed resources, execute, return allocated resources, and terminate. When $P_i$ terminates, $P_{i+1}$ can obtain its needed resources, and so on.

// Avoidance Algorithms :

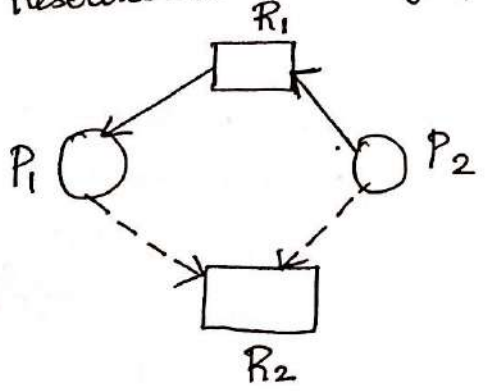
* Single Instance of a resource type. Use a resource - allocation graph.
* Multiple instance of a resource type. Use the banker's algorithm.

* Resource - Allocation - Graph Algorithm :-

In addition to request and assignment edges already described, and a new edge is introduced called a claim edge.

A claim edge $P_i \rightarrow R_j$ indicates that process $P_i$ may request resource $R_j$ at some time in the future. It is represented in graph by a dashed line.

Resource-allocation graph for deadlock avoidance.

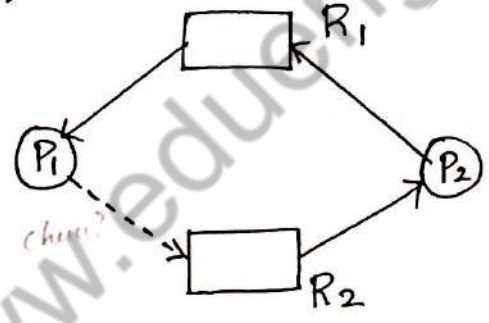


When Process $P_i$ requests resources $R_j$, the claim edge $P_i \rightarrow R_j$ is converted to a request edge. Similarly, when a resource $R_j$ is released by $P_i$, the assignment edge $R_j \rightarrow P_i$ is reconverted to a claim edge $P_i \rightarrow R_j$.

We note that the resources must be claimed a Priori in the system. That is before process $P_i$ starts executing, all its claim edges must already appear in the resource-allocation graph. We can relax this condition by allowing a claim edge $P_i \rightarrow R_j$ to be added to the graph only if all the edges associated with Process $P_i$ are claim edges.

If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. In that case, Process $P_i$ will have to wait for its requests to be satisfied.

An unsafe state in a resource-allocation graph.




Suppose that $P_2$ requests $R_2$. Although $R_2$ is currently free, we cannot allocate it to $P_2$, since this action will create a cycle in the graph. A cycle, as mentioned, indicates that the system is in an unsafe state. If $P_1$ requests $R_2$, and $P_2$ requests $R_1$, then a deadlock will occur.

Banker's Algorithm :

This algorithm is used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

Several data structures must be maintained to implement the banker's algorithm.

n → the number of processes in the system.

m → the number of resource types.

Available : Vector of length m. If available [j] = k, there are k instances of resource type $R_j$, available.

Max : n x m matrix. If Max [i,j] = k, then process $P_i$ may request at most k instance of resource type $R_j$.

Allocation : n x m matrix. If Allocation [i,j] = k then, $P_i$ is currently allocated k instances of $R_j$.

Need : n x m matrix. If need [i,j] = k, then $P_i$ may need k more instances of $R_j$ to complete its tasks.

Need [i, j] = Max [i,j] - Allocation [i, j].

Safety Algorithm :

1) Let work and finish be vectors of length m and n, respectively. Initialize work = Available and Finish [i] = false, for i = 0, 1, . . , n-1.

2) Find an index i such that both

a) Finish [i] = = false

b) Need i ≤ work.

If no such i exists, go to step 4.

3) Work = Work + Allocation i

Finish [i] = true

Goto step 2.

4) If Finish [i] = = true for all i, then the system is in a safe state.

Resource - Request Algorithm:

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

1) If $Request_i \leq Need_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.

2) If $Request_i \leq Available$, go to step 3. Otherwise, $P_i$ must wait, since the resource are not available.

3) Have the system pretend to have allocated the requested resources to process $P_i$ by modifying the state as follows;

$Available = Available + Request_i$;

$Allocation = Allocation_i + Request_i$;

$Need_i = Need_i - Request_i$;

Eg; Consider a system with five processes $P_0$ through $P_4$ and three resource types A, B and C.

| Resource type | Instance |
|---|---|
| A | 10 |
| B | 5 |
| C | 7. |

Suppose that, at time $T_0$ the following snapshot of the system has been taken:

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

Need : Max - Allocation.

| | Need. | | |
|---|---|---|---|
| | A | B | C |
| $P_0$ | 7 | 4 | 3 |
| $P_1$ | 1 | 2 | 2 |
| $P_2$ | 6 | 0 | 0 |
| $P_3$ | 0 | 1 | 1 |
| $P_4$ | 4 | 3 | 1 |

We claim that the system is currently in a safe state. Indeed the sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies the safety criteria.

Suppose now that process $P_1$ requests one additional instance of resource type A and two instances of resource type C, so $Request_1 = (1, 0, 2)$. To decide whether this request can be immediately granted, we first check that $Request_1 \leq Available$ - that is, that $(1, 0, 2) \leq (3, 3, 2)$ which is true. We arrive at the following new state.

| | Allocation | | | Need | | | Available | | |
|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | A | B | C | A | B | C |
| $P_0$ | 0 | 1 | 0 | 7 | 4 | 3 | 2 | 3 | 0 |
| $P_1$ | 3 | 0 | 2 | 0 | 2 | 0 | | | |
| $P_2$ | 3 | 0 | 2 | 6 | 0 | 0 | | | |
| $P_3$ | 2 | 1 | 1 | 0 | 1 | 1 | | | |
| $P_4$ | 0 | 0 | 2 | 4 | 3 | 1 | | | |

To check whether the new system is safe, find $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ satisfies the safety requirement.

$(3, 3, 0)$ by $P_4$ can't be granted &
$(0, 2, 0)$ by $P_0$ can't be granted even though the resources are available, since the resulting state is unsafe.

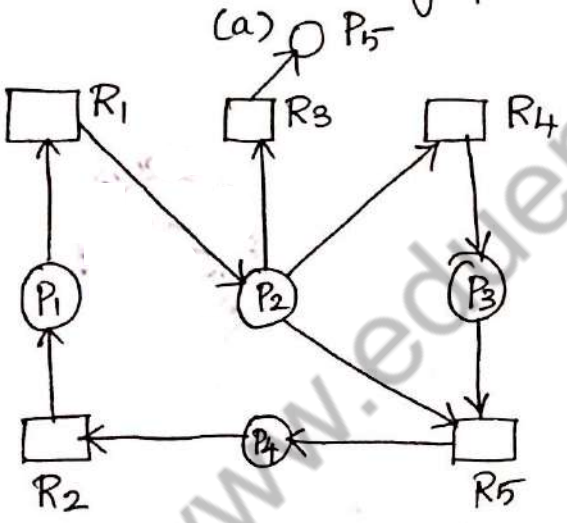
* Single Instance of each Resource type.
* Several Instance of a Resource type.
* Detection - Algorithm Usage.

* Single Instance of each Resource type :

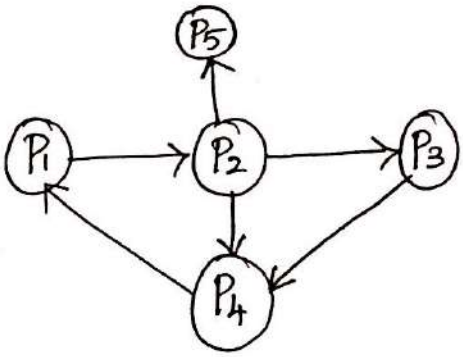
If all resources have only a single Instance, then we can define a deadlock - detection algorithm, Called wait - for graph.

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

Resource allocation graph. (a)




Corresponding wait - for graph (b)




Deadlock exists if and only if the wait - for - graph contains a cycle.

* Several Instances of a Resource type :

Available : A vector of length m indicates the number of available resources of each type.

Allocation: An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.

Request: An $n \times m$ matrix indicates the current request of each process. If Request $[i,j] = k$, then Process $P_i$ is requesting $k$ more instances of resource type. $R_j$.

Detection algorithm:

1) Let work and finish be vectors of length $m$ and $n$, respectively Initialize;
   (a) work = Available
   (b) for $i = 1, 2, \ldots, n$, if Allocation$_i \neq 0$, then Finish $[i] =$ false; Otherwise, Finish $[i] =$ true.

2) find an index $i$ such that both:
   (a) Finish $[i] ==$ false
   (b) Request$_i \leq$ work.

   If no such $i$ exists, go to step 4.

3) Work = Work + Allocation$_i$
   Finish $[i] =$ true
   goto step 2.

4) If Finish $[i] ==$ false, for some $i$, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if Finish $[i] ==$ false, then $P_i$ is deadlocked.

Eg., 5 processes $P_0$ through $P_4$; 3 resource types.
A (7 instances), B (2 instances), C (6 instances).

Snapshot at time $T_0$:

| | Allocation | Request | Available |
|---|---|---|---|
| | A B C | A B C | A B C |
| $P_0$ | 0 1 0 | 0 0 0 | 0 0 0 |
| $P_1$ | 2 0 0 | 2 0 2 | |
| $P_2$ | 3 0 3 | 0 0 0 | |
| $P_3$ | 2 1 1 | 1 0 0 | |
| $P_4$ | 0 0 2 | 0 0 2 | |

Sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in Finish[i] = true for all i.

$P_2$ requests an additional instance of type C.

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

State of System?

→ can reclaim resources held by process $P_0$, but insufficient resources to fulfill other processes, requests.

→ Deadlock exists, consisting of processes $P_1$, $P_2$, $P_3$ and $P_4$.

* Detection - Algorithm Usage:

1) How often is a deadlock likely to occur?

2) How many processes will be affected by deadlock when it happens?

## (30) Recovery from deadlock.

* Process Termination
* Resource Preemption

Process Termination:

* Abort all deadlocked processes.
* Abort one process at a time until the deadlock cycle is eliminated.

In which order should we choose to abort?

- Priority of the process.
- How long process has computed, and how much longer to completion.
- Resources process needs to complete.
- How many processes will need to be terminated.
- Is process interactive (or) batch?

Resource Preemption:

If Preemption is required to deal with deadlocks, then three issues need to be addressed.

- Selecting a victim
- Rollback
- Starvation.

Selecting a victim: Which resources and which Processes are to be Preempted? As in process termination, we must determine the order of Preemption to minimize cost.

Cost factors:-

1) Number of resources a deadlocked process is holding.
2) Amount of time the process consumed during its execution.

Rollback

We must rollback the process to some safe state and restart it from that state.

Rollback – Abort the process & restart it.

Starvation :

Same process may always be picked as victim, include number of rollback in cost factor.

# EDU ENGINEERING

PIONEER OF ENGINEERING NOTES

## TAMIL NADU'S BEST EDTECH PLATFORM FOR ENGINEERING

**CONNECT WITH US**

**WEBSITE:** www.eduengineering.net

**TELEGRAM:** @eduengineering

**INSTAGRAM:** @eduengineering

- **Regular Updates for all Semesters**
- **All Department** Notes AVAILABLE
- **Handwritten** Notes AVAILABLE
- **Past Year** Question Papers AVAILABLE
- **Subject wise** Question Banks AVAILABLE
- **Important Questions** for Semesters AVAILABLE
- Various **Author Books** AVAILABLE