UNIT - II

Process and CPU Scheduling

Process Concepts :

Program is nothing but a set of instructions whereas a process is a program which is under execution.

A process is more than a program code, which is Sometimes known as text - section.

It is also includes the current activity, as represented by the value of the program counter and the contents of the process registers.

A process is generally also includes the Process stack, which contains a temporary data.

And a Data section which contains global Variables.

A process may also includes a heap which is memory that dynamically allocated during process run time.

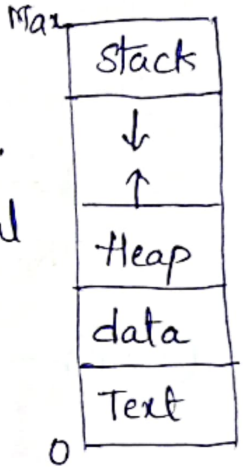


fig: Process in Memory.

We emphasize that a program by itself is not a process.

A program is a passive entity, such as file Containing a list of instructions Stored on a disk.

Whereas a process is an active entity with a program Counter specifying the address of next instruction to execute and a set of associated resources.

A program becomes process when an executable file is loaded into memory.

Process state :-

As a process, executes, it changes the 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:

1. New state : The process is being created.
2. Running state : Instructions are being executed.
3. Waiting state : The process is waiting for some event to occur.
4. Ready state : The process is waiting to be assigned to a processor.
5. Terminated state : The process has finished execution.

These names are arbitrary and vary across OS. The states that they represent found on all systems.

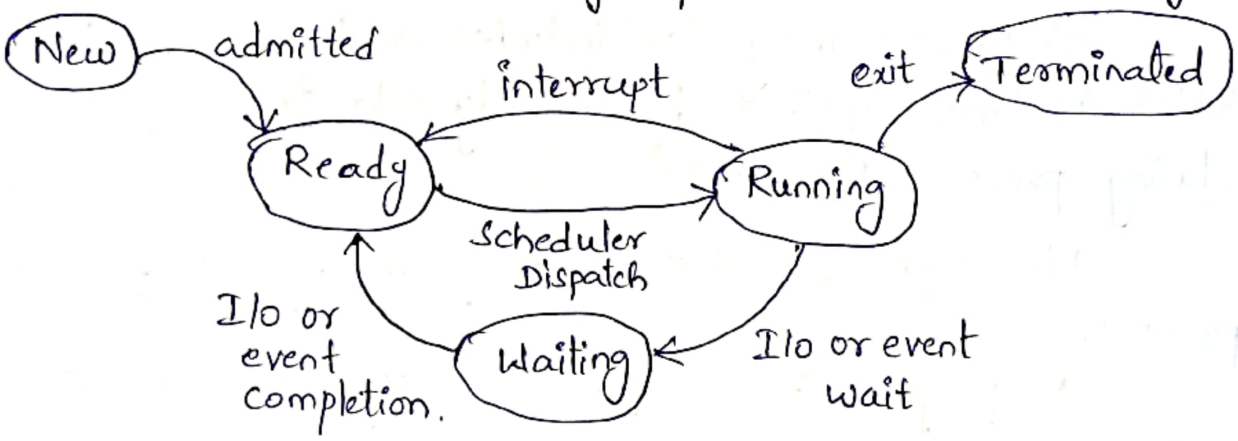


fig : Process state Diagram.

However certain OS also more finely delineate process states.

It is important to red realize that only one process can be running on any processor at any instant.

Process Control Block :-

Each process is represented in the OS by a process control block (PCB) which is also called as Task control block.

A PCB contains many pieces of information associated with a specific process.

A A PCB includes:

Process state : It may be ready running, waiting and so on.

Program Counter : It indicates the address of next instruction.

CPU Registers : Registers may vary in number and type, depending on computer architechture.

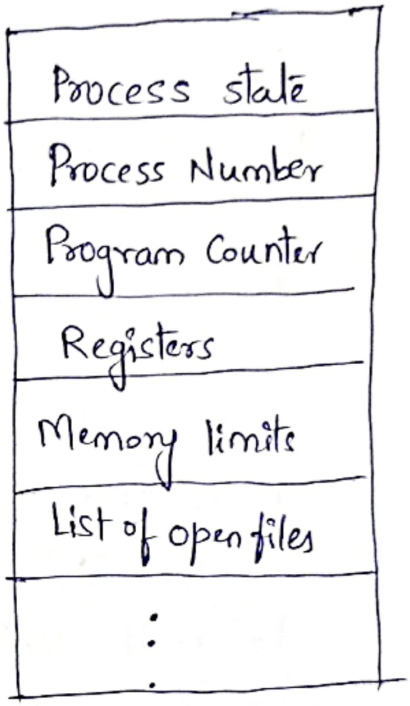


fig: Process Control block

CPU scheduling algorithms : It includes a process priority, pointers to scheduling queues and any parameters.

Memory management information : It includes such information of the value of the base and limit registers, page tables or the segment tables, depending on the memory system used by the OS.

Accounting information : It includes the amount of CPU and real time used, time limits, account numbers, job or process number and so on.

Process Scheduling :-

The objective of multiprogramming is to have some running process at all times, to maximize the CPU utilization.

The objective of time sharing is to switch the CPU among the process so frequently that the 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 :-

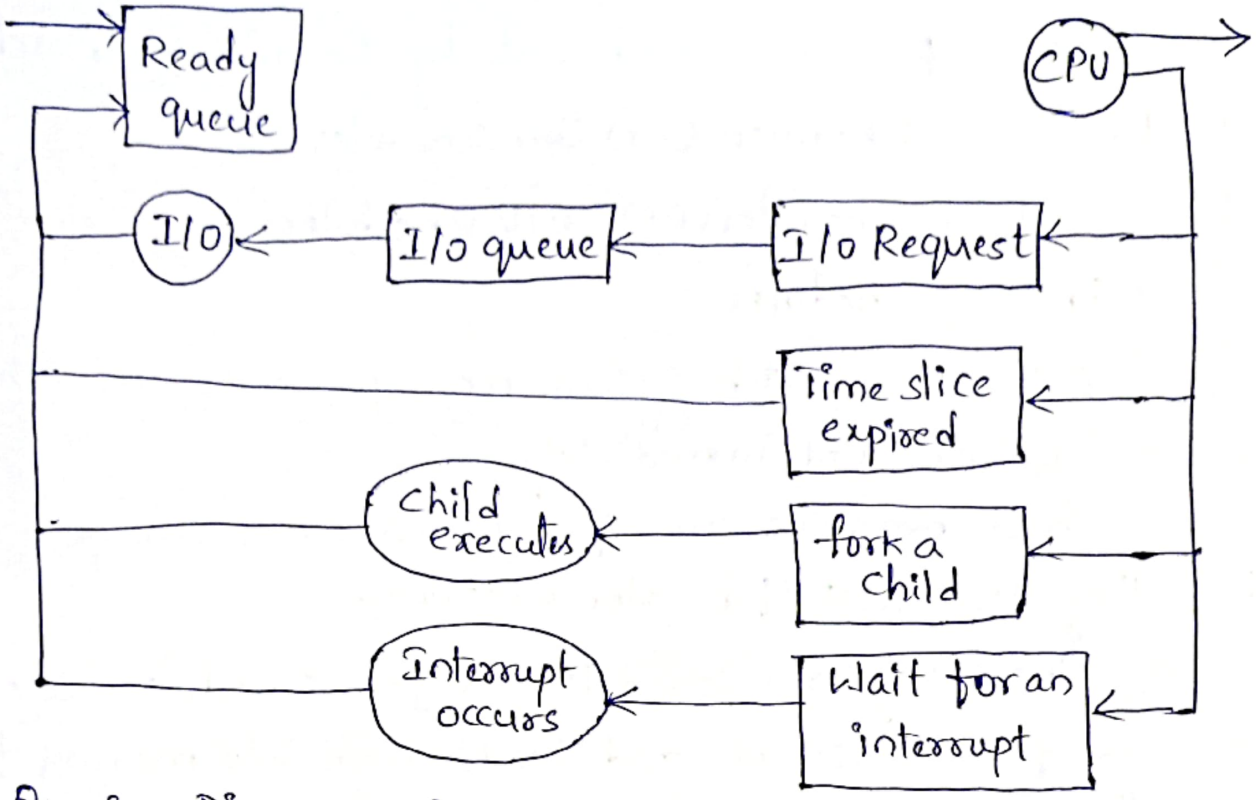
As a process enters the system, they a put into a job queue / which consist of all processes in the system.

The Processes that are residing on memory (main) are ready and waiting to execute are kept on a list called ready queue.

These queue is generally stored as a linked list.

A ready queue header contains pointer to the first and final PCB's in the list.

Each PCB includes a pointer field that points to the next PCB in the ready queue.



Queuing Diagram : Representation of Process scheduling.

Once the process is allocated and CPU is executing, one of the several events could occur:

- The process could issue an I/o request and then be placed in an queue.
- The process could create a new subprocess and wait for the subprocess termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

Schedulers :-

A process migrates among the various scheduling queues through its lifetime.

The OS must select for scheduling purposes, process from the queues in some fashion.

The selection process carried out by the appropriate Scheduler:

1. Long term scheduler (or) Job scheduler.
2. Short term scheduler (or) CPU Scheduler.
3. Mid term scheduler.

Often in a batch system, more processes are submitted than can be executed immediately.

Those programs are spooled to a mass storage device, where they are kept for later execution.

The long term scheduler or job scheduler selects processes from this pool and loads them into memory for execution.

The short term scheduler or CPU scheduler, selects from among the process that are ready to execute and allocates the CPU to one of them.

The primary distinction between the long term and short term lies in the frequency of execution.

The short term scheduler selects a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request.

The long term scheduler executes much less frequently; minutes may separate the creation of one new process to the next.

The long term scheduler may need to be invoked only when a process leaves the system.

A CPU-bound process, in contrast generates I/O requests infrequently using more of its time doing computations.

An I/O bound process is one that spends more

more of its time doing its I/o Computations.

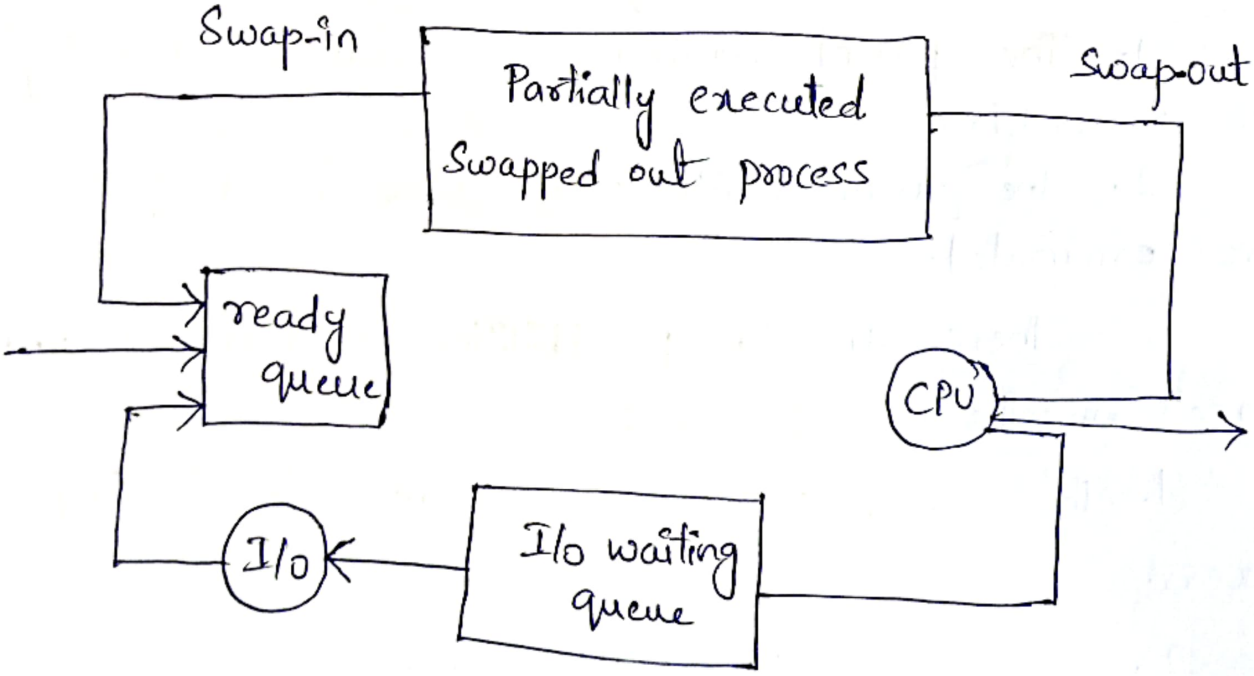
It is important to know that long term Scheduler Selects a good process mix of I/o bound and CPU bound processes.

Some OS such as time sharing systems, may introduce an additional intermediate level of scheduling i.e., Medium term Scheduler.

The key idea behind the mid-term scheduler is that sometimes it can be advantageous to remove process from memory and thus reduce the degree of multiprogramming.

Later the process can be re-introduced into memory and execution can be continued where it left off. This Scheme is called Swapping.

The process is swapped out and later swapped in by the mid term Scheduler.



Operations on Processes :-

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

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 the parent process and the new process are called the children of that process.

Each of these new process may in turn create other processes, forming a tree of processes.

When a process creates a new process, two possibilities exists 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.

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

1. The child process is a duplicate of the parent process.
2. The child process has a new program loaded into it.

Process Termination:

A process terminates when it finishes executing its final statement and ask the OS to delete it by using the exit() system call.

At that point, the process may return a status value to its parent process.

All the resources of the process including physical and virtual memory, open files and I/O buffers are deallocated by the OS.

Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call.

Usually such a system call can be invoked by the parent of a process that is to be terminated.

A parent may terminate the execution 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 OS does not allow a child to continue, if its parent terminates.

Co-operating processes, Threads and Inter-process Communication (IPC) :-

Threads :-

A thread is a basic unit of CPU utilization.

It comprises of a thread ID, a program counter, a register and a stack.

A thread is a flow of execution through process code with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains a execution history.

A Thread is also called a light weight process.

Thread provides a way to improve application performance through parallelism.

Difference between a process and a thread :-

Thread	Process
Thread is a light weight proc taking lesser resources and independent.	Process is heavy weight or Resource intensive and independent.
Thread switching does not need to interact with OS	Process switching needs interaction with OS.
All threads can share same set of open files, child process.	In multiple processing environments, each process executes the same code but has its own memory and file resources. (Different

Thread	Process
Multi-threaded process use fewer resources.	Multiprocesses without using threads use more resources.
There is no system calls involved.	System calls involved in process.
Context switching is faster.	Context switching is slower.
Blocking a thread will block entire process.	Blocking a process will not block another.

Advantages of a Thread :-

- Thread minimizes the context switching time.
- Use of threads provides Concurrency within the process.
- Efficient communication.
- It is more economical to create and context switch threads.
- Threads allow utilization of multiprocessor architechture to a greater scale and efficiency.

Types of Thread :

Threads are implemented in two ways :

1. User level threads
2. Kernel level threads.

User level threads :

Managed by user.

In this case threads are implemented by users.

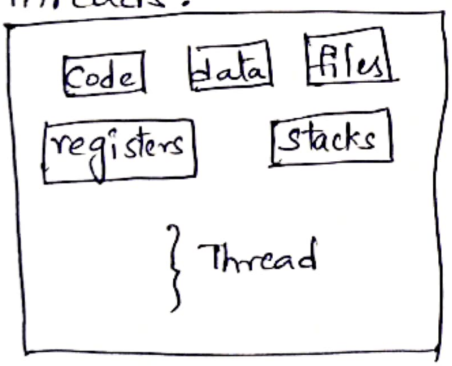
OS does not recognize user level threads.

Implementations of user threads is easy.

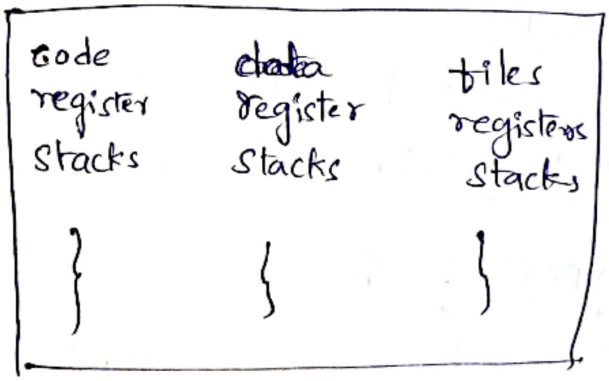
Context switch time is less.
Context switch requires no hardware support.
If one user level thread perform blocking operation the entire process will be blocked.
User level threads are designed are designed as dependent threads.

Kernel level threads :-

Kernel level threads are implemented by os.
Implementation of Kernel threads is complicated.
Context switch time is more.
Hardware support is needed.
If one kernel thread perform blocking operation then another thread can continue execution.
Kernel level threads are designed as independent threads.



Single Threaded process



Multi threaded process

Multi-Threading :-

Multithreading enables us to run multiple threads Concurrently.
The idea is to acheive paralellism by dividing a process into multiple threads.
For Example : In a browser, multiple tabs can be different threads.

Co-operating processes :-

Process executing concurrently in the OS may be either independent process or cooperating process.

A process is independent if it cannot affect or be affected by the other process executing in the system.

A process that does not share any data with other process is independent.

A process is cooperating if it can affect or affected by the other processes executing in the system.

Clearly, any process that share data with other process is a cooperating process.

There are several reasons for providing an environment that allows process co-operation:

1. Information sharing : Several users may be interested in the same place of information, we must provide an environment to allow concurrent access of such information.

2. Computation speedup : If we want to run a particular task faster, we must break it into subtasks, each of which will be executing in parallel with the others.

Notice that such a speed up can be achieved only if the computer has multiple processing elements.

3. Modularity : We may want to construct the system in a modular fashion, dividing the system

-functions into separate process or threads.

4. Convenience : Even an individual user may be editing printing and compiling in paralell.

Cooperating process requires an IPC mechanism that will allow them to exchange data and information.

Inter process Communication :-

Inter process communication in OS is the way by which multiple processes can communicate with each other.

Shared memory, Message queues, FIFO's etc are the some of the ways to acheive IPC in OS.

A system can have two types of processes i.e., Independent or cooperating processes.

Independent processes cannot affect each other and does not share any data among themselves.

IPC provides a mechanism to exchange data and information across multiple processes, which might be on single or multiple computers connected by a network.

There are different ways to implement IPC :-

1. pipes
2. FIFO'S
3. Message queues
4. Message passing
5. Shared memory
6. Direct communication
7. Indirect communication.

Scheduling Criteria :-

Different CPU scheduling algorithms have different properties and the choice of a particular algorithm may favour one class of processes over another.

Many criteria have been suggested for comparing CPU scheduling algorithms.

The criteria includes the following:

1. CPU utilization
2. Throughput
3. Turn around time
4. Waiting time
5. Response time.

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

CPU scheduling :-

CPU scheduling is a process of determining which process will own CPU for execution while another process is on hold.

The main task of CPU scheduling is to make sure that whenever the CPU remains idle, the OS atleast select one of the processes available in the ready queue for execution.

Here two kinds of CPU Scheduling methods are there : Pre-emptive & Non-preemptive.

In preemptive scheduling, the tasks are mostly assigned with their priorities. Sometimes it is important to run a task with a higher priority before another lower priority task, even if the lower priority task is running. The lower priority tasks holds for sometime and resumes when higher priority task finishes its execution.

In non-preemptive scheduling, the CPU has been allocated to a specific process. The process that keeps the CPU busy will release the CPU either by switching context or terminating. It is the only method that can be used for various hardware platforms. Because it does not need special hardware (Eg: timer) like pre emptive scheduling.

Scheduling of processes/work is done to finish the work on time:

The different times with respect to a process:

1. Arrival time: Time at which the process arrives/enters in the ready queue.
2. Completion time: Time at which process completes its execution.
3. Burst time: Time required by a process to get execute on a CPU.
4. Turn around time = Completion time − Arrival time.
5. Waiting time = Turn around time − Burst time.
6. Response time = [(The time at which a process gets CPU first time) − Arrival time].

CPU Scheduling Algorithms:-

1. First Come First Serve Scheduling (FCFS):-

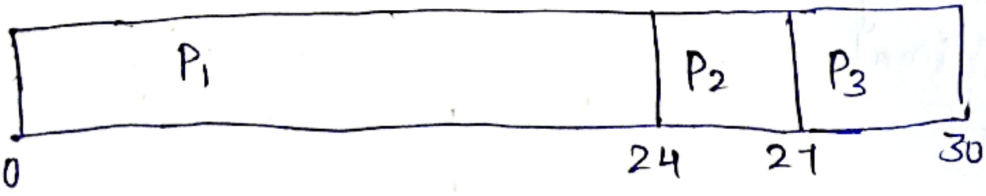
- Simplest CPU scheduling algorithm.
- The process that requests the CPU first is allocated to the CPU first.
- The implementation of the FCFS policy is easily managed with a FIFO queue.
- When a process enters the ready queue, its PCB is linked into the tail of the queue.
- When the CPU is free, it is allocated to the process at the head of the queue. the running process is then removed from the queue.
- The code to write FCFS scheduling is simple to write and understand.

Consider the following set of processes that arrive at time '0'.

Process	Burst time
P_1	24
P_2	3
P_3	3

If the process arrive in the order P1, P2, P3 and are served in FCFS order. Then we get the result as:

Gantt chart:



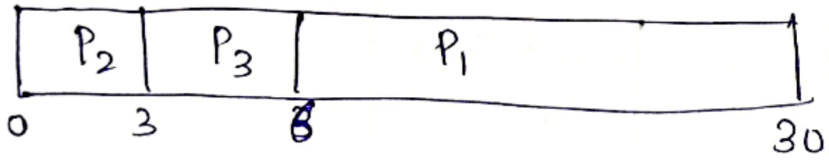
It is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes.

Process order	Arrival time	Burst time	Completion time C.T	Waiting time W.T = TAT − BT	Turn Around time TAT = C.T − A.T
P_1	0	24	24	24−24=0	24−0=24
P_2	0	3	27	27−3=24	27−0=27
P_3	0	3	30	30−3=27	30−0=30

Average Turn Around time $= \frac{24+27+30}{3} = 27$ ms

Average waiting time $= \frac{0+24+27}{3} = 17$ ms.

If the process arrive in the order P_2, P_3, P_1, however the result will be shown in the following Gantt chart:



Waiting time for P_1 = 6 ms
P_2 = 0 ms
P_3 = 3 ms

Average waiting time $= \frac{6+0+3}{3} = \frac{9}{3} = 3$ ms

The reduction is substantial. Thus the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes CPU burst times vary greatly.

The FCFS scheduling algorithm is non-preemptive.

Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or requesting I/O.

The FCFS algorithm is thus particularly troublesome for time sharing systems, where it is important that each user get a share of the CPU at regular intervals.

Convoy effect :-

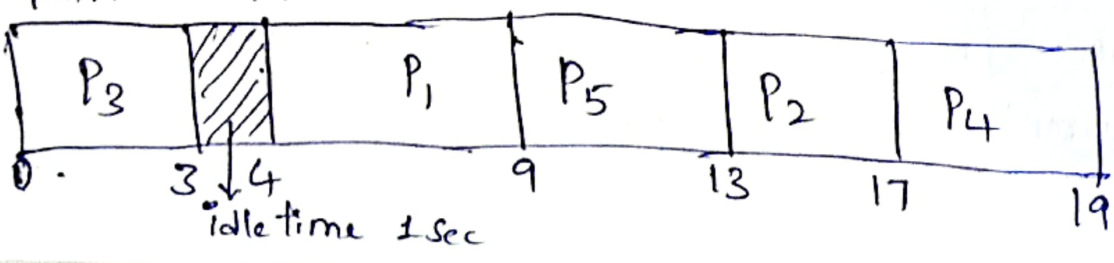
If a process with higher burst time arrived before the processes with smaller burst time, then smaller processes have to wait for a long time for longer processes to release the CPU.

Consider the set of 5 processes, whose arrival time and burst time are given below:

Process ID	Arrival time	Burst time
P_1	4	5
P_2	6	4
P_3	0	3
P_4	6	2
P_5	5	4

Calculate the average waiting time and average turn around time, if FCFS scheduling algorithm is followed.

Gantt chart:



Process ID	Arrival time A.T	Burst time B.T	Completion time C.T	TAT Around time (TAT) C.T − A.T	Waiting time (W.T) TAT − B.T
P_1	4	5	9	9−4 = 5	5−5 = 0
P_2	6	4	17	17−6 = 11	11−4 = 7
P_3	0	3	3	3−0 = 3	3−3 = 0
P_4	6	2	19	19−6 = 13	13−2 = 11
P_5	5	4	13	13−5 = 8	8−4 = 4

Average Turn around time $= \frac{5+11+3+13+8}{5} = \frac{40}{5} = 8$ units

Average waiting time $= \frac{0+7+0+11+4}{5} = \frac{22}{5} = 4$ units.

2. Shortest Job first Scheduling (SJF) 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 the two processes are the same, FCFS Scheduling is used to break the tie.

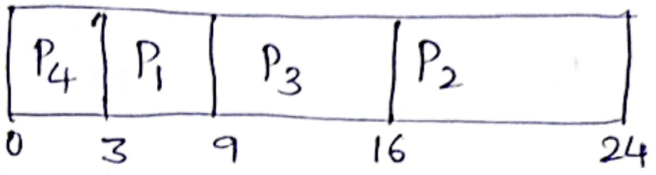
The SJF algorithm can be either preemptive or non-preemptive. The choice of algorithm arises when a new processes arrives at the ready queue while a previous process is still executing.

The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process.

Consider the set of 4 processes whose arrival time and burst time are given below (Arrival time = 0)

Process	Burst time
P_1	6
P_2	8
P_3	7
P_4	3

Gantt chart :



Process	Burst time	completion time	TAT	W.T
P_1	6	9	9-0=9	9-6 = 3
P_2	8	24	24-0=24	24-8=16
P_3	7	16	16-0=16	16-9 = 9
P_4	3	3	3-0=3	3-3 = 0

Average waiting time $= \frac{3+16+9+0}{4} = \frac{28}{4} = 7$ ms

Average turn around time $= \frac{9+24+16+3}{4} = \frac{52}{4} = 13$ ms.

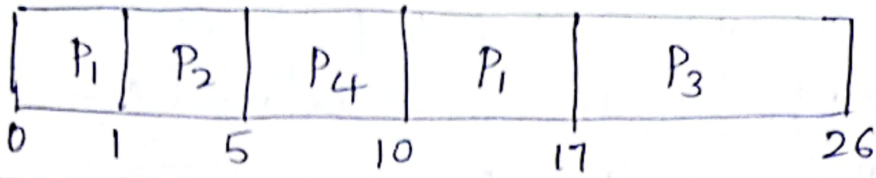
A pre-emptive SJF algorithm will preempt the currently executing process, whereas a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst.

Preemptive SJF scheduling is sometimes called Shortest Remaining time first scheduling.

Consider set of 4 processes whose arrival time and burst time are given :

Process	Arrival time	Burst time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Gantt chart :



Process	B.T	A.T	C.T	TAT = C.T − A.T	W.T = TAT − B.T
P_1	8	0	17	17 − 0 = 17	17 − 8 = 9
P_2	4	1	5	5 − 1 = 4	4 − 4 = 0
P_3	9	2	26	26 − 2 = 24	24 − 9 = 15
P_4	5	3	10	10 − 3 = 7	7 − 5 = 2

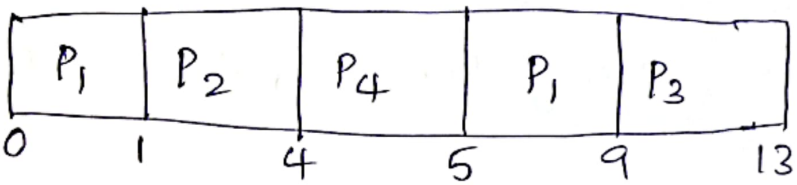
Average TAT $= \frac{17+4+24+7}{4} = \frac{52}{4} = 13$ ms

Averag W.T $= \frac{9+0+15+2}{4} = \frac{26}{4} = 6.5$ ms

SJF preemptive scheduling (SRTF) example:

Process	A.T	B.T
P_1	0	5
P_2	1	3
P_3	2	4
P_4	3	1

Gantt chart :



Process	A.T	B.T	C.T	TAT = C.T − A.T	W.T = TAT − B.T	R.T
P_1	0	5	9	9	4	0 − 0 = 0
P_2	1	4	4	3	0	1 − 1 = 0
P_3	2	3	13	11	7	9 − 2 = 7
P_4	3	1	5	1	0	4 − 4 = 0

Average TAT $= \frac{24}{4} = 6.0$

Average W.T $= \frac{11}{4} = 2.75$

3. 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.

An SJF algorithm is simply a priority algorithm where the priority is the inverse of the (predicted) next CPU burst.

The larger the CPU burst, the lower the priority and vice versa.

Priority scheduling can be either preemptive or non-preemptive.

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.

Equal priority process are scheduled in FCFS order

A major problem with priority scheduling is Indefinite blocking and starvation.

A priority scheduling can leave some low priority processes waiting indefinitely.

A solution to the problem of indefinite blockage of low priority process is aging.

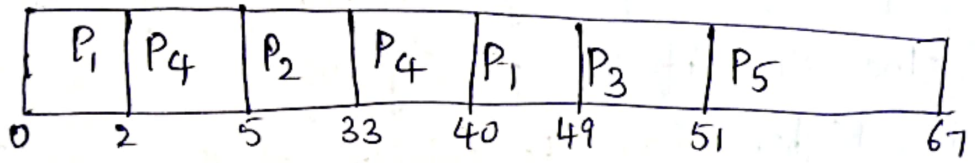
Aging is a technique of gradually increasing

the priority of processes that wait in the system for a long time.

Consider the set of processes with arrival time (in ms), CPU burst time and priority ('0' is the highest priority) shown below. None of the processes have the I/O burst time. (preemptive priority scheduling).

Process ID	Arrival time	Burst time	priority
P_1	0	11	2
P_2	5	28	0
P_3	12	2	3
P_4	2	10	1
P_5	9	16	4

Gantt chart:



Process	A.T	B.T	C.T	TAT = C.T − A.T	W.T = TAT − B.T
P_1	0	11	49	49	38
P_2	5	28	33	29	0
P_3	12	2	51	39	31
P_4	2	10	40	38	28
P_5	9	16	67	58	42

Average waiting time = $\frac{38+0+31+28+42}{5} = \frac{149}{5} = 29.8$ ms

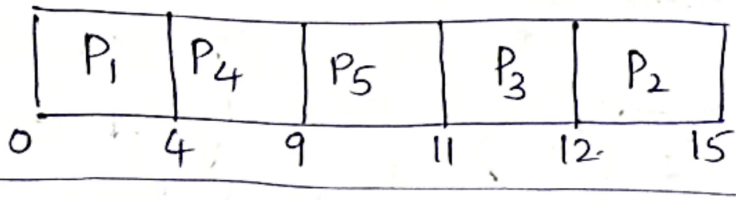
Consider the set of processes with arrival time, CPU burst time and priority shown below:

(Higher Number represents higher priority)

Process ID	Arrival time	Burst time	priority
P_1	0	4	2
P_2	1	3	3
P_3	2	1	4
P_4	3	5	5
P_5	4	2	5

If the CPU Scheduling policy is non-preemptive, Calculate the Average waiting time and Average turn around time ?

Gantt chart :



Process ID	Burst time	Arrival time	Completion time	TAT C.T − A.T	W.T. TAT − B.T
P_1	4	0	4	4−0 = 4	4−4 = 0
P_2	3	1	15	15−1 = 14	14−3 = 11
P_3	1	2	12	12−2 = 10	10−1 = 9
P_4	5	3	9	9−3 = 6	6−5 = 1
P_5	2	4	11	11−4 = 7	7−2 = 5

Average waiting time $= \frac{0+11+9+1+5}{5} = \frac{26}{5} = 5.2$ ms

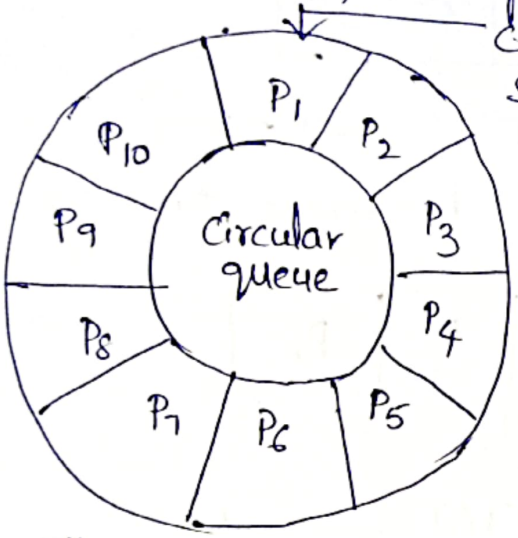
Average turn around time $= \frac{4+14+10+6+7}{5} = \frac{41}{5} = 8.2$ ms

4. Round Robin Scheduling Algorithm :-

The round robin (RR) scheduling algorithm is designed especially for time sharing systems.

It is similar to FCFS scheduling but preemption is added to switch between the processes.

A small unit of time, called a time quantum or time slice, is defined. (generally from 10 to 100 milliseconds)



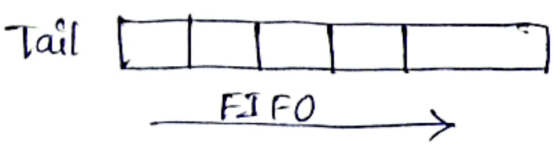
The Ready queue

* The ready queue is treated as a circular queue.

The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of upto 1 time quantum.

Implementation of Round Robin Scheduling :-

We keep the ready queue as a FIFO queue of process. New processes are added to the tail of the ready queue. The CPU Scheduler picks the first processes from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatch the process



One of the two things will happen when we set the time quantum.

I case:

The process may have a CPU burst of less than 1 time quantum.

The process itself will release the CPU voluntarily.

The CPU scheduler will then proceed to the next process in the ready queue.

II case:

The CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the OS.

A context switch will be executed and the process will be put at the tail of the ready queue.

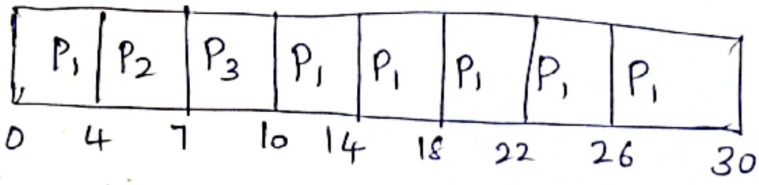
The CPU scheduler will then select the next process in the ready queue.

Example: Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in ms and time quantum taken as 4ms for RR scheduling.

Process ID	Burst time
P_1	24
P_2	3
P_3	3

Time Quantum = 4ms

Gantt chart:



Process ID	Completion time	TAT	W.T
P_1	30	30	6
P_2	7	7	4
P_3	10	10	7

Average T.A.T = $\frac{30+7+10}{3}$

$= \frac{47}{3} = 15.66$ms

Average W.T = $\frac{6+4+7}{3}$

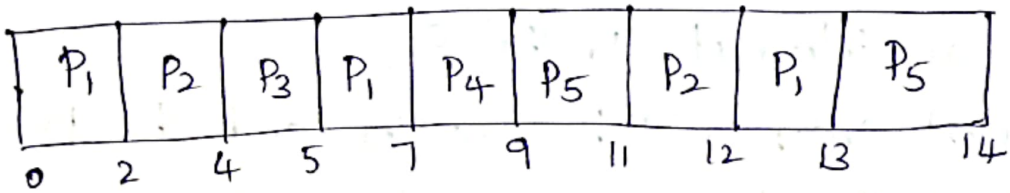
$= \frac{17}{3} = 5.66$ms

Consider the set of 5 processes whose arrival time and burst time are given below.

Time quantum = 2units.

Process ID	Arrival time	Burst time
P_1	0	5
P_2	1	3
P_3	2	1
P_4	3	2
P_5	4	3

Gantt chart :



Process ID	Completion time	TAT	W.T
P_1	13	13-0=13	13-5 =8
P_2	12	12-1 =11	11-3 = 8
P_3	5	5-2 =3	3-1 =2
P_4	9	9-3 = 6	6-2 =4
P_5	14	14-4=10	10-3 =7

Average turn around time = $\frac{13+11+3+6+10}{5}$

$= \frac{43}{5} = 8.6$ units

Average waiting time = $\frac{8+8+2+4+7}{5}$

$= \frac{29}{5} = 5.8$ units

Multi-level queue Scheduling Algorithm:-

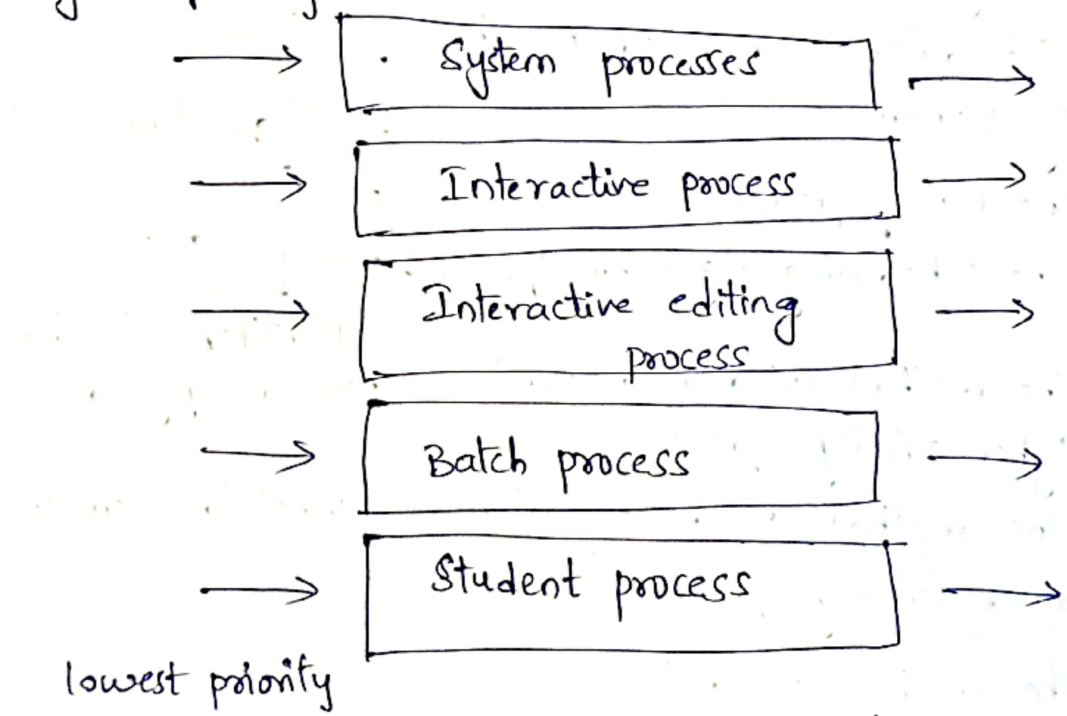
A multilevel queue scheduling algorithm partitions the ready queue into several separate queues. The processes are permanently assigned to one queue, generally based on some property of the process, such as memory, size, process priority, or process type. Each queue has its own scheduling algorithm.

For example separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.

Lets look an example of multilevel queue scheduling algorithm with the five queues.

Multilevel queue scheduling each queue has absolute priority over lower priority queues.

Another possibility is to time slice among the queues.



Here, each among the queues gets a certain portion of the CPU time, which it can then schedule among its various processes.

Multi-level feedback queue scheduling :-

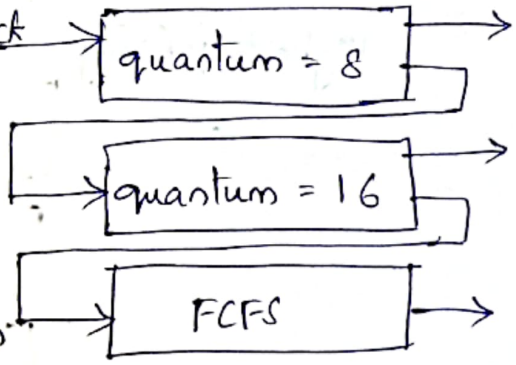
Normally when the multi-level queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system. If there are separate queues for foreground and background processes.

The multi-level feedback queue scheduling algorithms, in contrast allows a process to move between queues. The idea is to separate according to the charachterstics of their CPU bursts.

If a process uses too much CPU time it will be moved to lower priority queue. This scheme leaves I/O bound and interactive processes in the higher priority queues.

In addition, a process that waits too long in a lower priority queue may be moved to a higher priority queue. This form of aging is called starvation.

In general, a multilevel feedback queue scheduler is defined by the following parameters:



Multilevel feedback queue.

- The no. of queues.
- The scheduling algorithm for each queue.
- The method used to determine when to upgrade a process to a higher priority queue.
- The method used to determine when to demote a process to a lower priority queue.
- The method used to determine which queue a process will enter when that process needs service.

Multiple processor queue 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 to run any process in the queue which are available.

Approaches of 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 processes executes only the user code. This is simple and reduces the need of data sharing. This entire scenario is called asymmetric multiprocessing.

A second approach uses symmetric multiprocessing where each processor is self scheduling. All processes may be in common ready queue or each processor may have its own private queue for ready processes.

Processor Affinity :-

processor affinity means a processe has an affinity for the processor on which it is currently running. There are two types of processor affinity :-

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

Hard affinity : Hard affinity allows a process to specify a subset of processors on which it may run.

Load Balancing :-

It is the phenomena which keeps the work load evenly distributed all processors in an SMP system.

There are two general approaches to load balancing:

Push 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 process from overloaded to idle or less busy processors.

Pull Migration: It occurs when an idle processor pulls awaiting task from a busy processor for its execution.

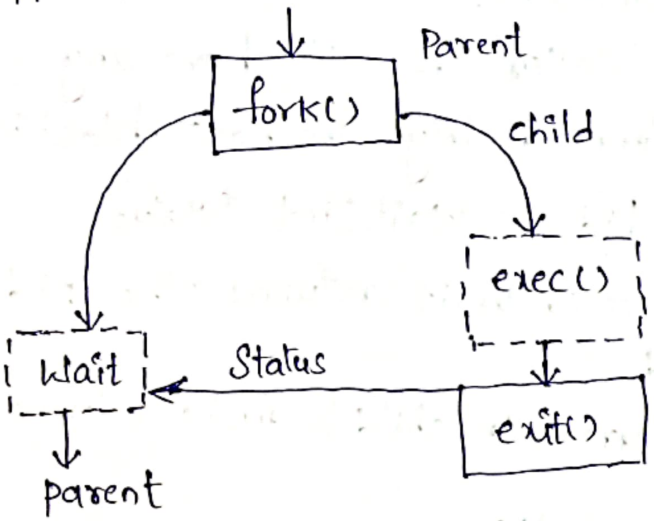
Multicore processors :

In Multi core processors multiple processors Cores are placed on the physical chip.

System call interface for process Management :-

The following system calls are used for process management in Linux / Unix :

1) fork() - to create a new process.
2) exec() - to run a new program.
3) wait() - to make the process to wait.
4) exit() - to terminate the process.
5) getpid() - to find unique processid.
6) getppid() - to find parent process id.



(1) fork() System call :-

A parent process uses fork to create a new child process.

The child process is a copy of a parent.

After fork both parent and child executes the same program but in separate process.

Syntax : Pid_t fork(void);

On success, the pid of the child process is returned in the parent and '0' returned in the child.

On failure, -1 is returned in the parent, no child process is created.

(2) exec() System call:

It replaces the program executed by a process. The child may use 'execl' after a fork to replace the process memory space with a new program executable making the child to execute a different program than parent.

(3) Wait() System call:-

The wait system call blocks the caller (parent) until one of its child process terminates.

The parent may use wait() to suspend executes until a child terminates.

If the caller doesn't have any child process, 'wait' returns immediately without blocking the caller.

Syntax: Pid_t wait(int *status)

On success, wait returns the pid of the terminated child.

on failure, (n child) waits returns -1.

(4) Exit() system call:-

Terminates the process with an exit status.

Ex: exit(EXIT_SUCCESS)

exit(EXIT_FAILURE).