UNIT-IV

# Operating System Basics :-

## Operating System :

The Operating system acts as a bridge between the user applications / tasks & the underlying system resources through a set of system functionalities & services.

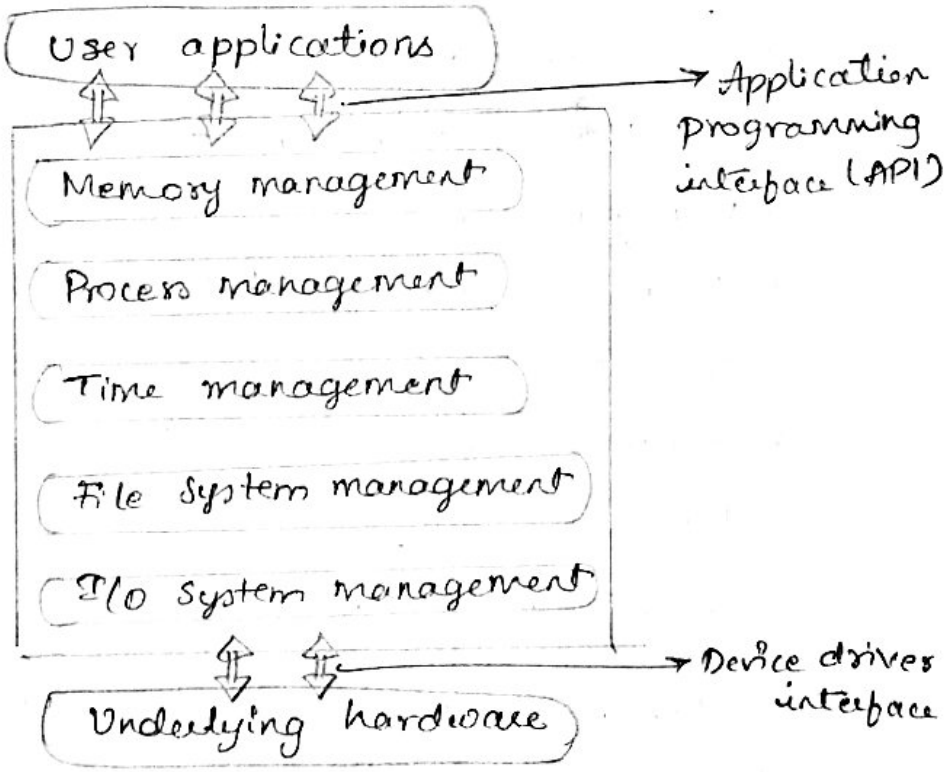


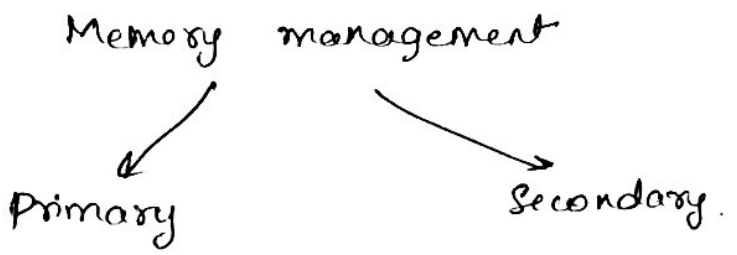

Fig: Operating System Architecture.

## Kernel Services :-

### Kernel :-

Kernel is the core of operating system & is responsible for managing the system resources & the communication among hardware & other system services. Kernel acts as the abstraction layer between system resources & user applications.

1. Memory Management :-




Primary Memory Management :-

→ The term primary memory refers to the volatile Memory (RAM) ie

→ The Memory Management Unit (MMU) of kernel is responsible for.

* Keeping track of which part of the memory area is currently used by which process.
* Allocating & De-allocating memory space on a need basis.

Secondary Storage Management :-

→ Secondary memory is used as backup medium for programs & data since the main memory is volatile.

→ The Secondary storage management service of kernel deals with

* Disk storage allocation
* Disk scheduling
* Free Disk Space management.

2. Process management :-

→ Process management deals with managing the process / tasks.

→ Process management includes setting up the memory

Space for the process, loading the process's code into the memory space, allocating system resources, scheduling & managing the execution of the process, setting up & managing the Process Control Block (PCB), Inter Process Communication & Synchronisation, process termination / deletion, etc.

3. Time Management :-

→ Accurate time management is essential for providing precise time reference for all applications.

→ The time reference to Kernel is provided by a high-resolution Real-Time Clock (RTC) hardware chip.

→ The hardware timer is programmed to interrupt the processor / controller at a fixed rate.

→ If the System time register is 32 bits wide & the 'Timer tick' interval is 1 microsecond, the system time register will reset in

$$2^{32} * 10^{6} / (24 * 60 * 60) = 49700 \text{ Days} = \sim 0.0497 \text{ Days}$$
$$= 1.19 \text{ Hours}$$

If the 'Timer tick' interval is 1 millisecond, the system time register will reset in

$$2^{32} * 10^{-3} / (24 * 60 * 60) = 497 \text{ Days} = 49.7 \text{ Days}$$
$$= \sim 50 \text{ Days}$$

4. File Management System :-

→ A file could be a program, text files, image files, word document, audio/video files etc.

→ Each of these files differ in the kind of information they hold & the way in which the information is stored.

→ The file system management service of kernel is responsible for

* The creation, deletion & alteration of files
* Creation, deletion & alteration of directories.
* Saving of files in the secondary storage memory
* Providing automatic allocation of file sy space based on the amount of free space available.
* Providing a flexible naming convention for the files.
* for example: Kernel of Microsoft DOS OS supports a specific set of file system management operations & they are not the same as the file system operations supported by UNIX kernel.

I/O System Management:

→ Kernel is responsible for routing the I/O requests coming from different user applications to the appropriate I/O devices of the system.

→ The kernel maintains a list of all the I/O devices of the system.

→ The service 'Device Manager' of the kernel is responsible for handling all I/O device related operations.

→ The kernel talks to I/O device through set of low level systems calls, which are implemented in a service, called device drivers.

→ The device drivers are specific to a device or a class of devices. The Device Manager is responsible for

* loading & unloading of device drivers.

* Exchanging information & the system specific control signals to & from the device

## Protection Systems :-

→ Most of the modern operating systems are designed in such a way to support multiple user's with different levels of access permissions.

→ Protections deals with implementing the security policies to restrict the access to both User & system resources by different applications or processes or users.

→ In multiuser supported operating systems, one User may not be allowed to view or modify the whole / portions of another user's data or profile data.

→ In addition, some application may not be granted with permission to make use of some of the system resources.

→ This kind of protection is provided by the protection services running within the kernel.

## Interrupt Handler :-

→ Kernel provider handler mechanism for all external / internal interrupts generated by system.

→ These are some of the important services offered by the kernel of an operating system. It does not mean that a kernal contains no more than components / services explained above.

→ Depending on the type of the operating system, a kernal may contain lesser number of components/ services or more number of components / services. In addition to the components / services listed above, many operating system offer a number of addtn system components / services to the kernel.

→ Network communication, network management, user interface graphics, timer services, error handler, database management etc are examples for such component/services.

→ Kernel exposes the interface to the various kernel applications / services, hosted by kernel, to the user applications through a set of standard Application Programming Interfaces (APIs).

→ User applications can avail these API calls to access the various kernel application / services.

## Kernel Space & User Space :-

### Kernel Space :-

The program code corresponding to the kernel applications / services are kept in a contiguous area of primary memory & is protected from the unauthorised access by user programs/applications. The memory space at which kernel code is located is known as 'kernel space'.

### User space :-

All the user applications are loaded to a specific area of primary memory & this memory area is referred as User Space.

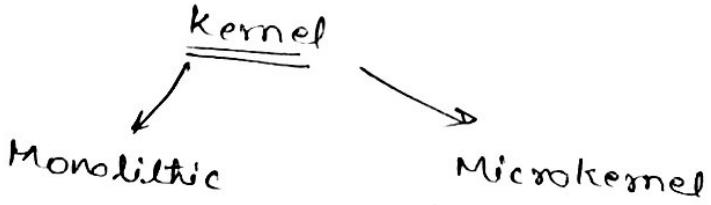
→ The partitioning of memory into Kernel & User Space is purely Operating System dependent. Some OS implements this kind of partitioning & protection whereas some OS do not segregate the Kernel & user application code storage into two separate areas.

→ The act of loading the code into & out of the main Memory is termed as swapping.

→ Swapping happens between the main memory & secondary ~~memor~~ storage memory.

→ Each process run in its own virtual memory space and are not allowed accessing the memory space corresponding to another process, unless explicitly requested by the process.

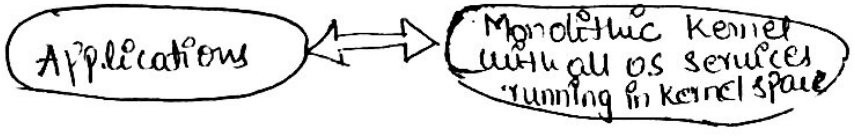
Types of Kernel




Monolithic Kernel :-

→ An monolithic kernel architecture, all kernel services run in the kernel space. Here all kernel modules run within the same memory space under a single kernel thread.

→ The tight internal integration of kernel modules in monolithic kernel architecture allows the effective utilisation of low level features of the underlying system.



Drawback:

→ Any error or failure in any one of the kernel modules leads to the crashing of entire kernel. applications.

→ examples : LINUX, SOLARIS, MS-DOS kernels

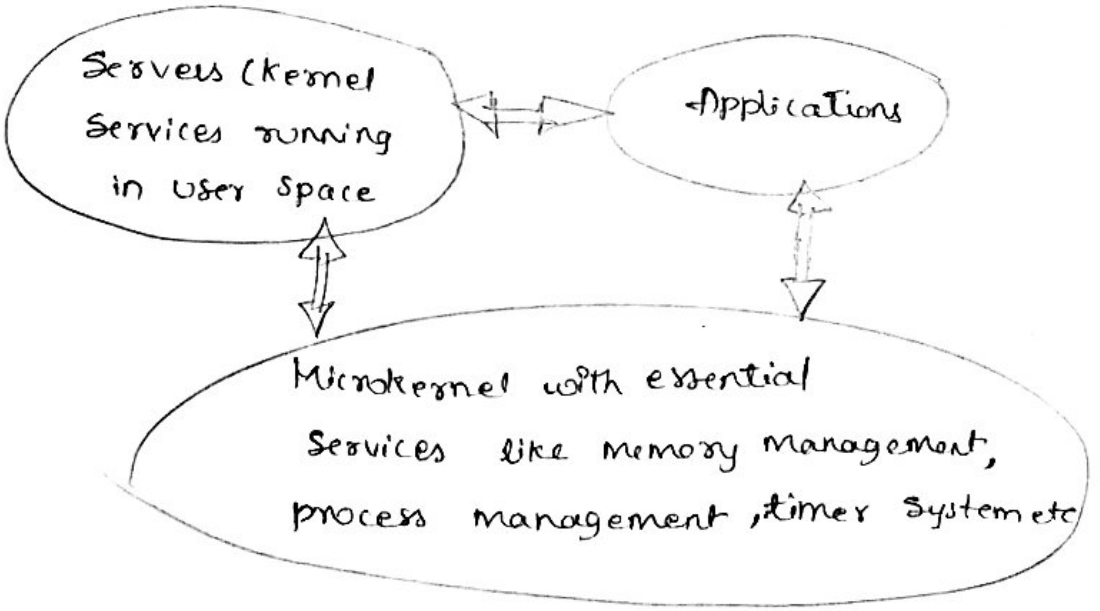
Microkernel :-

→ The Microkernel design incorporates only the essential set of OS services into the kernel.

→ The rest of OS services are implemented in programs known as 'servers' which runs in user space.

→ This provides a highly modular design & OS-neutral abstraction to the kernel.

→ Examples : Mach, QNX, Minix 3 kernels




Microkernel model

→ Microkernel based design approach offers the following benefits.

* Robustness :-

→ If a problem is encountered in any of the services,

which runs as 'server' application, the same can be reconfigured & re-started without of the services, ~~which~~ need for restarting the OS.

→ Thus, this approach is highly useful for systems, which demands high 'availability'.

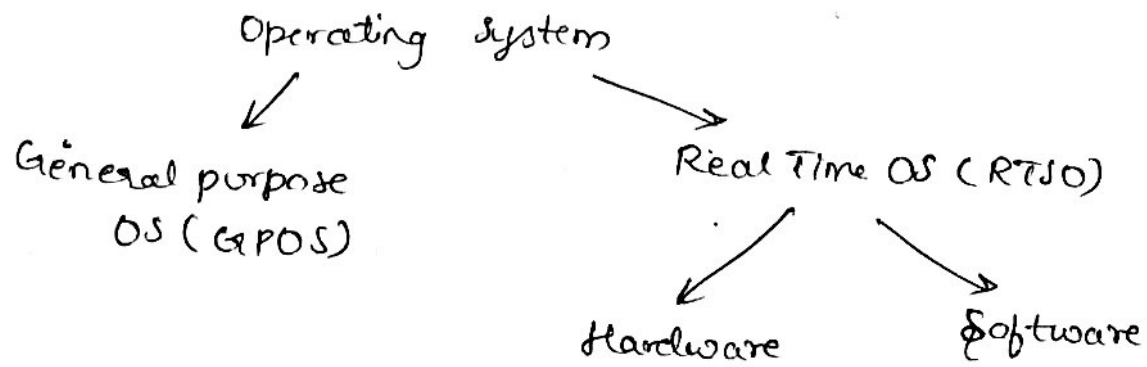
→ ~~Refer~~ eho Since the services which run as 'servers' are running on a different memory space, the chances of corruption of kernel services are ideally Zero.

* Configurability :-

→ Any services, which runs as 'server' application can be changed without the need to restart the whole system. This makes the system dynamically configurable.

TYPES OF OPERATING SYSTEMS :-

Depending on the type of kernel & kernel services, purpose & type of computing systems where the OS is deployed & the responsiveness to applications, OS are classified into different types.



## General Purpose Operating System (GPOS) :-

→ The operating systems, which are deployed in general computing systems, are refferred as General purpose operating system (GPOS).

→ The kernel of such an OS is more generalised & it contains all kinds of services required for executing generic applications.

→ General purpose OS are often quite non deterministic in behaviour.

→ Their services can inject random delays into application software & may cause slow responsiveness of an application at unexpected times.

→ GPOS are usually deployed in computing systems where deterministic behaviour is not an important criterion.

→ examples: Personal computer / Desktop system, Windows 10/8.x Windows XP / MS-DOS etc.

## Real Time Operating System :-

→ Real Time implies deterministic timing behaviour. Deterministic timing behaviour in RTOS context means the OS services consumes only known & expected amounts of time regardless the number of services.

→ A Real time OS or RTOS implements policies and rules concerning time critical allocations of a system's resources.

→ The RTOS decides which application should run in which order & how much time needs to be allocated for each application.

→ Predictable performance is the hallmark of a well designed RTOS.

→ This is best achieved by the consistent application of policies & rules.

→ policies guide the design of an RTOS. Rules implement those policies & resolve policy conflicts.

→ examples: Windows Embedded Compact, QNX, VxWorks Micro c/OS-II.

Types of RTOS :-


Hard Real Time :-

→ Real time OS that strictly adhere to the timing constraints for a task is referred as 'Hard Time systems'.

→ A Hard Time system must meet the deadlines for a task without any slippage.

→ Missing any deadline may produce catastrophic results for Hard Real Time systems, including permanent data lose & irrecoverable damages to the system/ users.

→ example; Air bag control Systems & Anti lock Brake System

→ Hard Time Real Time System does not implement the virtual memory model for handling the memory.

→ Most of the Hard Real Time systems are automatic and does not contain a human in the loop.

## Soft Real Time :-

→ RTOS that does not guarantee meeting deadlines, but offer best effort to meet the deadline are referred as Soft Real Time Systems.

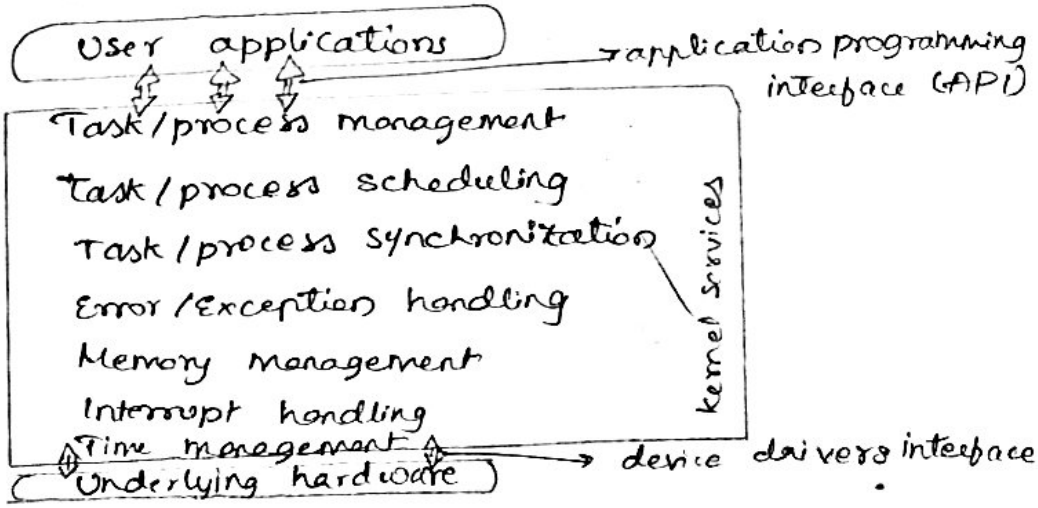
→ Missing deadlines for tasks are acceptable for a Soft Real Time system if the frequency of deadline missing is within the compliance of limit of Quality of Service (QOS).

→ examples:-

1. Automatic Teller Machine (ATM). If ATM takes a few seconds more than the ideal operation time, nothing fatal happens.
2. An audio video playback system. No potential damage arises if a sample comes late by fraction of a second, for playback.

## The Real Time Kernel :-

→ In complement to the conventional OS kernel, the Real Time Kernel is highly Specialised & it contains only the minimal set of services required for running the user applications / tasks.



## 1. Task / Process management :-

→ Deals with setting up the memory space for the tasks, loading the task's code into the memory space, allocating system resources, setting up a Task Control Block (TCB) for the task and task / process termination / deletion.

→ A TCB is used for holding the information corresponding to a task.

→ TCB usually contains the following set of information

Task ID : Task Identification number

Task State : The current state of the task.
(Eg State = Ready for task which is ready to execute)

Task Type : Task type indicates what is type of this task.
The task can be a hard real time & soft real time or background task.

Task priority : Task priority (Eg: Task priority = 1 for task with priority = 1)

Task context pointer : Context pointer for context saving.

Task Memory pointer : pointers to the code memory, data memory & stack memory for the task.

Task System Resource pointers : pointers to system resources used by the task.

Task pointers : pointers to other TCBs (TCBs for preceding, next, waiting tasks)

Other parameters : Other relevant task parameters.

→ The parameters & implementation of the TCB is Kernel dependent.

→ The TCB parameters vary across different kernels, based on task management implementation.

→ Task management service utilises the TCB of a task in the following way.

* Creates a TCB for a task on creating task.
* Delete / remove the TCB of a task when the task is terminated or deleted.
* Reads the TCB to get the state of a task.
* Update the TCB with updated parameters on need basis.
* Modify the TCB to change the priority of the task dynamically.

Task / Process Scheduling :-

→ Deals with sharing the CPU among various tasks/ processes. A kernel specifies application called 'Scheduler' handles the task scheduling.

→ Scheduler is nothing but an algorithm implementation which performs the efficient & optimal scheduling of tasks to provide a deterministic behaviour.

Task / Process Synchronisation :-

→ Deals with synchronising the concurrent access of a resource, which is shared across multiple tasks & the communication between various tasks.

Error / Exception Handling :-

→ Deals with registering & handling the errors occured/ exceptions raised during the execution of tasks.

→ Insufficient memory, timeouts, deadlocks, deadline missing, bus error, divide by zero, unknown instruction

execution etc are examples.

→ The OS kernel gives information about the error in form of a system call (API)

→ Watchdog timer is a mechanism for handling the timeouts for tasks.

→ Certain tasks may involve the waiting of external events from devices.

→ These tasks will wait infinitely when the external device is not responding & the task will generate a hang-up behaviour.

→ In order to avoid these types of scenarios, a proper timeout mechanism should be implemented.

Memory Management :-

→ In general, the memory allocation time increases depending on the size of the block of memory needs to be allocated & the state of the allocated memory block.

→ Since predictable timing & deterministic behaviour are the primary focus of an RTOS, RTOS achieves this by compromising the effectiveness of memory allocation.

→ RTOS makes use of block based memory allocation technique.

→ RTOS kernel uses blocks of fixed size of dynamic memory & the block is allocated for a task on a need basis. The blocks are stored in a 'Free Buffer Queue'.

→ To achieve predictable timing & avoid the timing overheads, most of RTOS kernels allow tasks to

acess to any of the memory blocks without any memory protection.

→ A few RTOS kernels implement Virtual Memory concept for memory allocation if the system supports secondary memory storage.

→ In the block based memory allocation, a block of fixed memory is always allocated for tasks on need basis & it is taken as a unit.

→ The block memory concept avoids the garbage collection overhead also. The block based memory allocation achieves deterministic behaviour with the trade of limited choice of memory chunk size & suboptimal memory usuage.

Interrupt Handling :-

→ Deals with the handling of various types of interrupts.

→ Interrupts inform the processor that an external device or an associated task requires immediate attention of CPU.

→ Interrupts can be either Synchronous or Asynchronous.

→ Interrupts which occur in syn with the currently executing task is known as synchronous interrupts.

→ Examples :- Divide by zero, memory segmentation error.

→ Asynchronous interrupts are interrupts, which occurs at any point of execution of any task & are not in syn with the currently executing task.

examples: timer overflow interrupts, Serial data reception/ transmission interrupts.

→ Priority levels can be assigned to the interrupts & each interrupts can be enabled or disabled individually

→ Most of the RTOS kernel implements 'Nested Interrupts' architecture. Interrupt nesting allows the preemption of an Interrupt services Routine, servicing the interrupt by a high priority Interrupt.

Time Management :-

→ Accurate time management is essential for providing precise time reference for all applications.

→ The time reference to kernel is provided by a high resolution Real Time Clock (RTC) hardware chip.

→ The hardware timer is programmed to interrupt the processor / controller at fixed rate.

→ If the System time register is 32 bits wide & the timer tick interval is 1 microsecond, the System time register will reset in

$$2^{32} * 10^{-6} / (24 * 60 * 60) = 49700 \text{ Days} = \sim 0.0497 \text{ Days}$$
$$= 1.19 \text{ Hours}$$

If 'timer tick' interval is 1 millisecond, the system time register will reset in

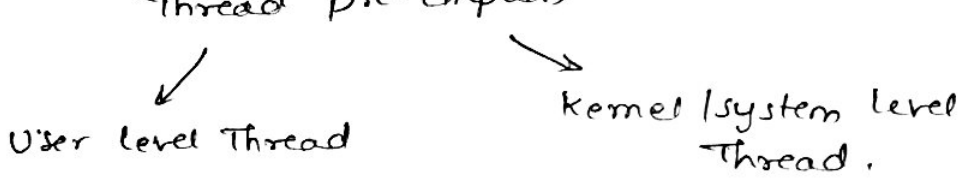
$$2^{32} * 10^{-3} / (24 * 60 * 60) = 497 \text{ Days} = 49.7 \text{ Days}$$
$$= \sim 50 \text{ Days}$$

## Thread Pre-emption :-

→ Thread Preemption is the act of pre-empting the currently running thread.

→ Thread pre-emption ability is solely dependent on the Operating system. Thread pre-emption is performed for sharing CPU time among all the threads.

→ The execution switching among threads are known as 'Thread context switching'. Thread context Switching is dependent on operating system's Scheduler & the type of thread.




## User level Thread :-

→ User level threads do not have Kernel / Operating system support & they exit soley in the running process.

→ Even if a process contains Multiple user level threads, the OS treats it as single Thread & will not switch the execution among the different threads of it.

→ User level threads of a process are non-pre-emptive at thread level from OS perspective.

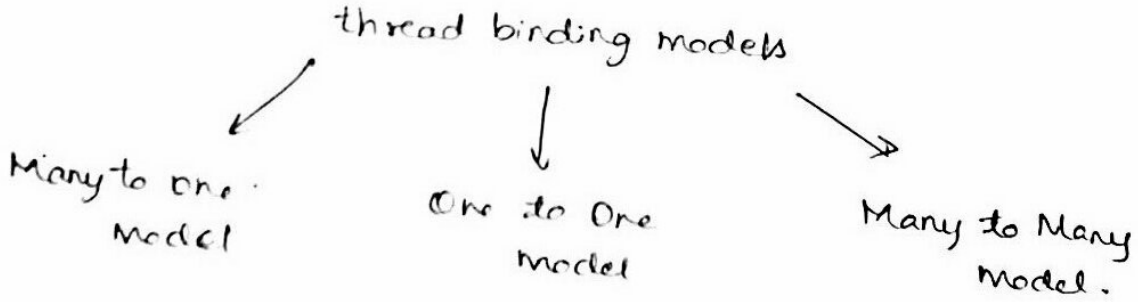
## Kernel / System level thread :-

→ Kernel level threads are individual units of execution, which the OS treats as separate threads.

→ The OS interrupts the execution of the currently running kernel threads & switches the execution to another kernel based on the scheduling policies implemented by the OS.

→ Kernel Threads are pre-emptive.

→ There are many ways for binding user levels threads with system / kernel level threads. The following sections gives an overview of various thread binding models.




### 1) Many to One Model :-

→ Many user level threads are mapped to a single kernel thread.

→ In this model, the kernel treats all user level threads as single thread & the execution switching among the user level threads happens when a currently executing user level thread voluntarily blocks itself or relinquishes the CPU.

→ examples : Solaris Green threads & GNU Portable Threads

application : 'P Thread' example given under the POSIX thread library section.

2. One to One Model :-

→ In One to One model, each user level thread is bonded to a kernel / system level thread.

→ examples : Windows NT & Linux threads

→ application : modified 'Pthread' example given under the 'Thread Pre-emption' section.

3. Many to Many model :-

→ In this model, many user level threads are allowed to be mapped to many kernel threads.

→ examples : Windows NT/2000 with Thread Fibre package.

Difference between Thread & Process :-

| Thread | Process |
|---|---|
| 1) Thread is a single unit of execution & is part of process. | 1) Process is a program in execution & contains one or more threads. |
| 2) A thread does not have its own data memory & heap memory. It shares the data memory & heap memory with other threads of the same process. | 2) Process has its own code memory, data memory & stack memory. |
| 3) A thread cannot live independently ; it lives within the process. | 3) A process contains at least one thread. |

| | |
|---|---|
| 4) There can be multiple threads in a process. The first thread calls the main function & occupies the start of stack memory of the process. | 4) Threads within a process share the code, data & heap memory. Each thread holds separate memory area for stack. |
| 5) Threads are very inexpensive ~~to fast~~ create | 5) Processes are very expensive to create. Involves many OS overhead. |
| 6) Context switching is inexpensive and fast | 6) context switching is complex & involves lot of OS over head. & is comparatively slower |
| 7) If a thread expires, its stack is reclaimed by the process. | 7) If a process dies, the resources allocated to it are reclaimed by OS & all the associated threads of the process also dies. |

# PROCESS:

A 'Process' is a program, or part of it, in Execution. Process is also known as an instance of a program in Execution.

-> Multiple instances of the same program can execute simultaneously.

-> A process requires various system resources like cpu for executing the process, memory for storing the code corresponding to the process and associated variables, I/O devices for information exchange

## The Structure of a process

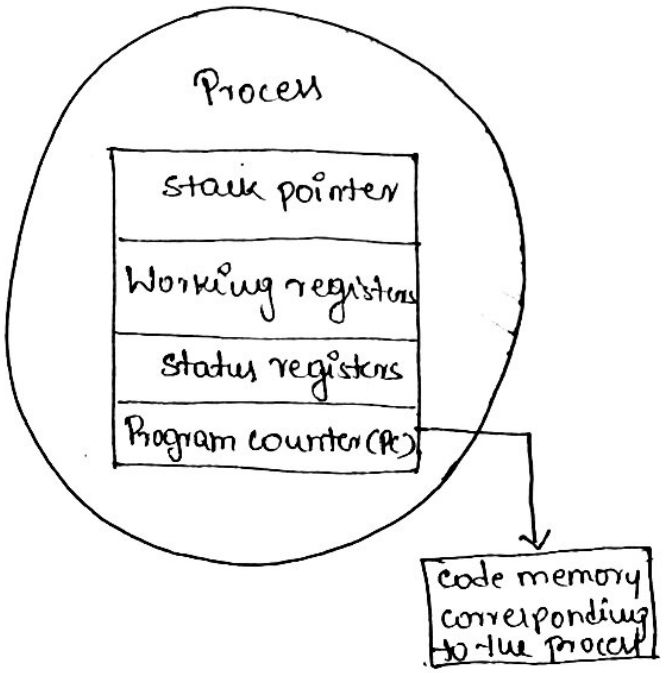



fig: Structure of a process

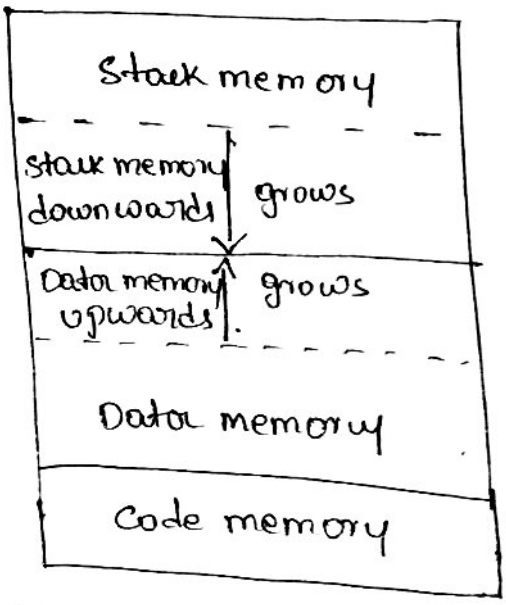



Fig: Memory organisation of a process.

number of simultaneous processes existing in the system, some of these will be system processes and others will be user processes

→ Process states:

The state of an agent defines a condition, a situation or a model of the agent. Here the state of a process means indication of the current condition and situation of the process. In fact, the program a process is executing, the values of all the data in its address space and the process context collectively determine the state of the process For the sake of the operating system, these states of processes are partitioned into a few categories These categories are themselves called as process states in operating system terminology. Each category is identified by a name and is represented by a value. The current state value of a process identifies the category of the process. As execution of a process proceeds, the process changes its state

The most commonly used state value are,

(1) created / starting state:-

This state value signifies that the process has just been created. The operating system is setting up the initial execution context and the initial private address space for the process and will be allocating it some resources.

(2) Ready state:

This state value signifies that the process is ready to run, but the CPU not yet being available, is waiting in the CPU queue, where it will wait until the operating system allocates it the CPU.

(3) Running state:

This state value indicates that the process is running on the CPU. The CPU is executing instructions from the current address space and the process execution is progressing.

(4) Blocked / waiting state:

This state value indicates that the process is currently waiting for some event (such as the completion of an I/O operation) to take place in the system. It may not resume its execution until the event occurs. Until then it sleeps in an event queue and the operating system will awaken it when the event occurs.

(5) Completed / Exiting state:

This state value indicates that the process execution is completed, either normally or abnormally. In this state, the OS will cleanup the process address space and free up resources acquired by the process.

The states ready, running, waiting/blocked are collectively called as the 'Execution state'.

# PROCESS STATE TRANSITION

As the execution of process proceeds, the process changes its states i.e., the transition of a process from one state to another is known as 'state transition'. The below figure represents the various states and a process state transition diagram

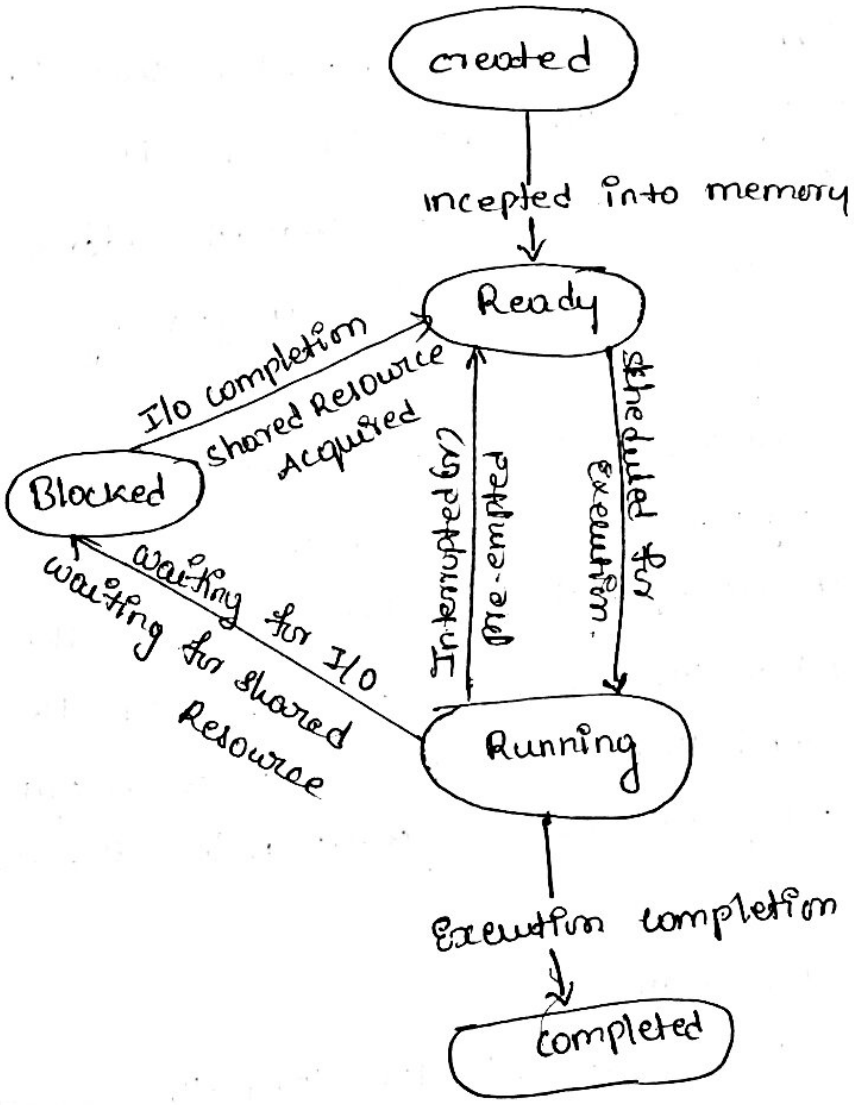



fig: Process states and state Transition Representation.

# MULTIPROCESSING AND MULTITASKING

Multiprocessing: The ability to execute multiple processes simultaneously. The systems which are capable of performing multiprocessing, are known as multiprocessor systems.

Multiprogramming: The ability of the O.S. to have multiple programs (programs) in memory which are ready for execution, is referred as multiprogramming.

Multitasking: The ability of an OS to hold multiple processes in memory and switch the processor (CPU) from executing one process to another process is known as Multitasking.

Context switching: The act of switching CPU among the process or changing the current execution context is known as 'context switching'

Context saving: The act of saving the current context which contains the context details for the currently running process at the time of CPU switching is known as 'context saving'

Context Retrieval: The process of retrieving the saved context details for a process, which is going to be executed due to CPU switching, is known as 'context retrieval'.

Multitasking involves 'context switching, context saving and context retrieval'.

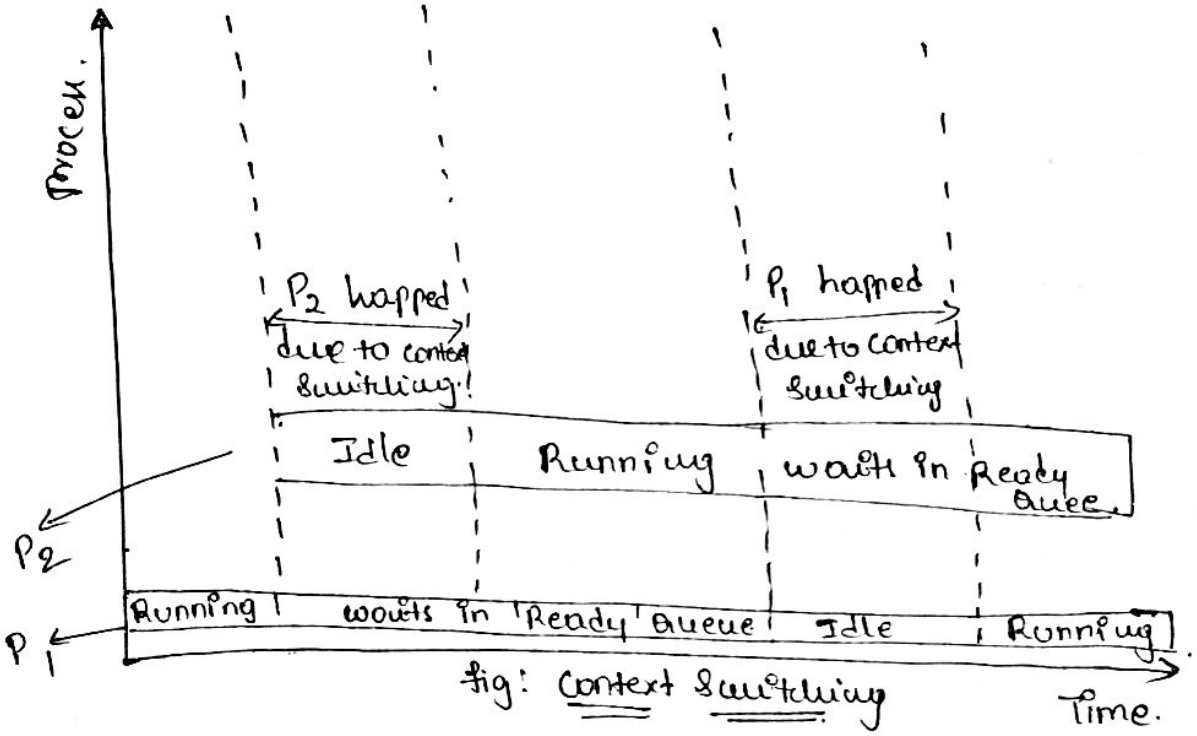



fig: Context Switching

Multitasking Example:

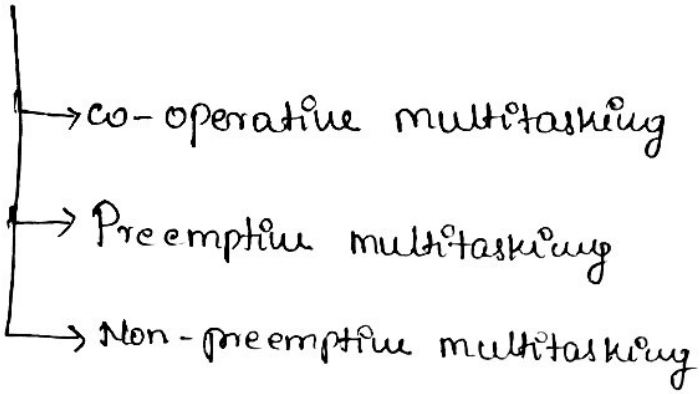
1. Real world Example

Toss Juggling: The juggler uses a no. of objects and throws them up & catches them. At any point of time, he throws only one ball & catches only one per hand. However, the speed at which he is switching the objects for throwing and catching creates the illusion, he is throwing & catching multiple balls or using more than hands simultaneously, to the spectators.

2. RT-11 (Textual O.S Example)

RT11 is a multitasking O.S system because while there is a process executing in the foreground, you can have another process executing in the background.

## TYPES OF MULTITASKING.




### Co-operative multitasking.

* Co-operative multitasking is the most primitive form of multitasking. In which a task/process gets a chance to execute only when the currently executing task/process voluntarily relinquishes the cpu.
* In this method, any task/process can hold the cpu as much time as it wants And this type of implementation involves the mercy of the tasks each other for getting the cpu time for execution, it is known as co-operative multitasking!

### Pre-emptive Multitasking:

* Preemptive multitasking ensures that every task/process gets a chance to execute. When & how much time a process gets is dependent on the implementation of the preemptive scheduling.
* As the name indicates, in preemptive multitasking, the currently running task/process is preempted to give a chance to other tasks/process to execute. The preemption of task may be based on time slots or task/process priority.

## Non-preemptive multitasking:

* In non-preemptive multitasking, the process/task, which is currently given the cpu time, is allowed to execute until it terminates (enters the 'completed' state) or enters the 'Blocked/wait' state, waiting for an I/O or system resource.
* In non-preemptive multitasking the currently executing task relinquishes the cpu when it waits for an I/O or system resource or an event to occur.

## TASK SCHEDULING:

* The process of determining which task/process is to be executed at a given point of time is known as task/process scheduling.
* The scheduling policies are implemented in an algorithm & it is run by the kernel as a service.
* The kernel service/application which implements the scheduling algorithm, is known as 'Scheduler'.
* The process scheduling decision may take place when a process switches its state to
  1. 'Ready' state from 'Running' state
  2. 'Blocked/wait' state from 'Running' state
  3. 'Ready' state from 'Blocked/wait' state
  4. 'Completed' state.

Scenario 1

A process switches to 'Ready' state from the 'Running' state when it is preempted. The type of scheduling is pre-emptive.

Scenario 2

Scheduling under this scenario can be either preemptive or non-preemptive

Scenario 3

When a high priority process in the 'Blocked/waiting' state completes its I/O & switches to the 'Ready' state, the scheduler picks it for execution if the scheduling policy used is priority based preemptive.

Scenario 4

Under this, scheduling can be preemptive, non-preemptive or co-operative

The selection of a scheduling criterion/algorithm should consider the following factors:

* cpu utilisation: The scheduling algorithm should always make the cpu utilisation high. cpu utilisation is a direct measure of how much percentage of the cpu is being utilised.

The various queues maintained by OS in association with CPU scheduling are:

1. Job Queue: Job queue contains all the processes in the system.
2. Ready Queue: Contains all the processes, which are ready for execution & waiting for cpu to get their turn for execution. The Ready queue is empty when there is no process ready for running.
3. Device Queue: Contains the set of processes, which are waiting for an I/O device.

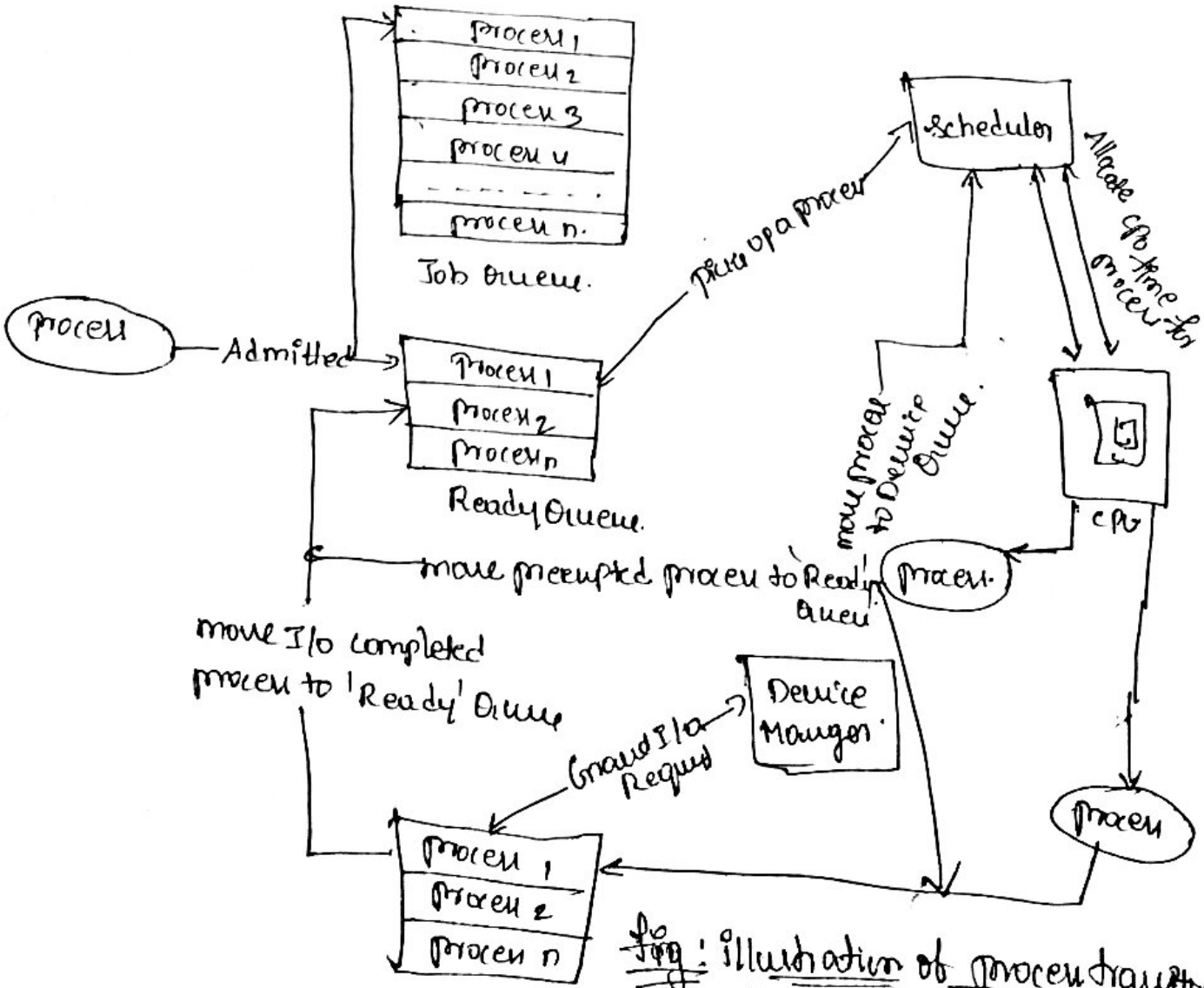



fig: Illustration of process transition through various queues.

Throughput: This gives an indication of the number of processes executed per unit of time. The throughput for a good scheduler should always be higher.

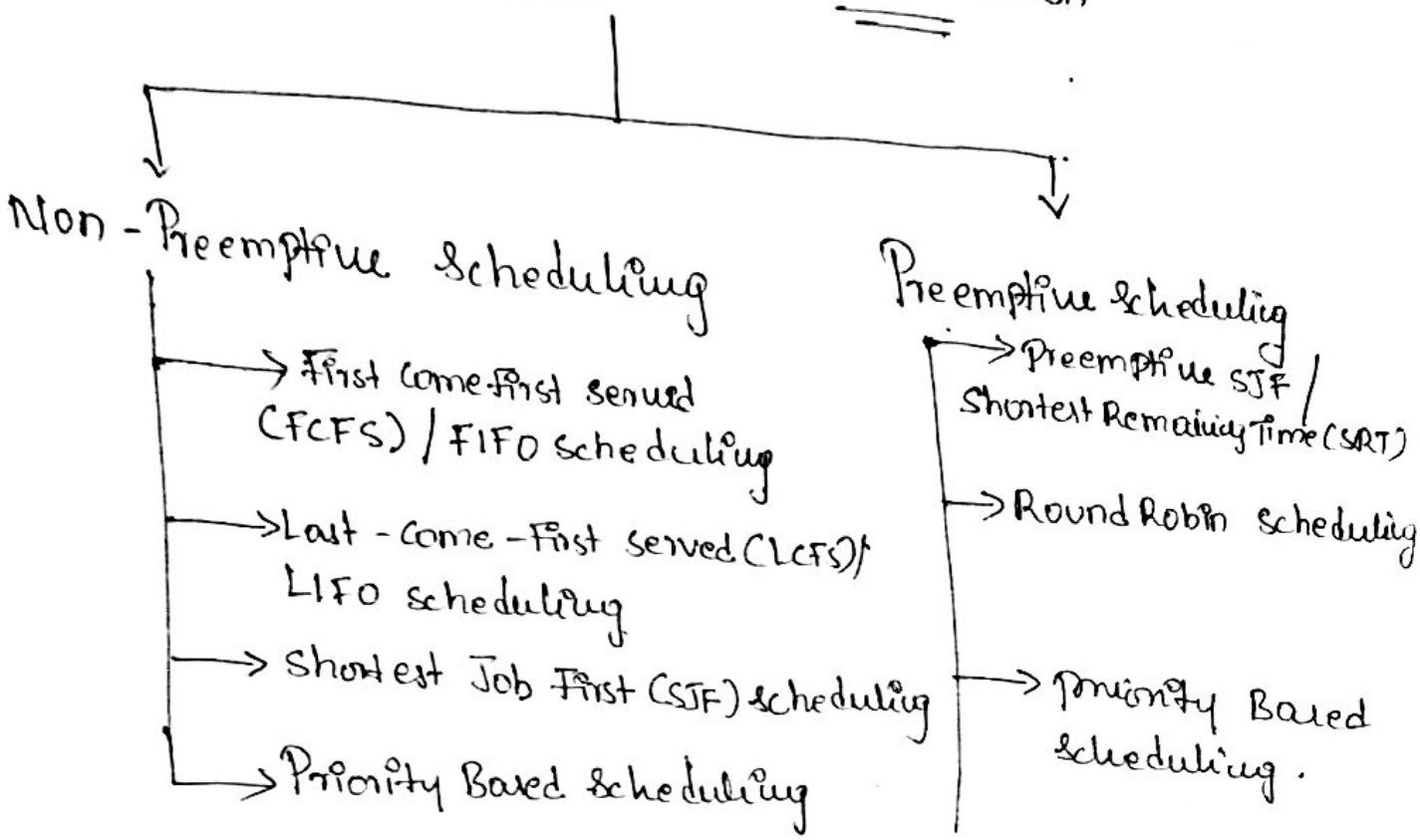
Turnaround Time: It is the amount of time taken by a process for completing its execution. It includes the time spent by the process for waiting for the main memory, time spent in the ready queue, time spent on completing the I/O operations, & the time spent in execution. The turnaround time should be a minimal for a good scheduling algorithm.

Waiting Time: It is the amount of time spent by a process in the 'Ready' queue waiting to get the cpu time for execution. The waiting time should be minimal for a good scheduling algorithm.

Response Time: It is the time elapsed between the submission of a process & the first response. For a good scheduling algorithm, the response time should be as least as possible.

Note: A good scheduling algorithm has high cpu utilisation, minimum Turn Around Time (TAT), maximum throughput & least response time.

# TASK SCHEDULING CLASSIFICATION




## Non-preemptive Scheduling

* This type of scheduling is employed in systems, which implement non-preemptive multitasking model.
* In this scheduling type, the currently executing task/process is allowed to run until it terminates or enters the 'wait' state waiting for an I/O or system resource.

⇒ First-Come-First-Served (FCFS) / FIFO Scheduling.

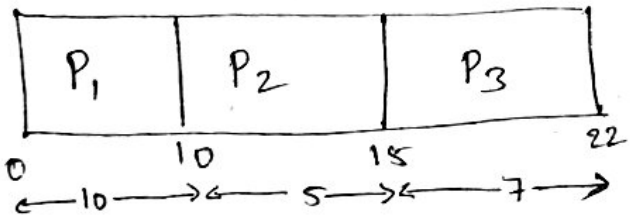
* The FCFS scheduling algorithm allocates CPU time to the process based on the order in which they enter the 'Ready' queue.

* The first entered process is serviced first.
* FCFS Scheduling is also known as First In First Out (FIFO) where the process which is put first into the 'Ready' queue is serviced first.

Example:

Three processes with Process IDs $P_1, P_2, P_3$ with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together in the order $P_1, P_2, P_3$. Calculate the waiting time & Turn Around time (TAT) for each process & the average waiting time & Turn Around Time (Assuming there is no I/O waiting for the process).

→ The sequence of execution of the processes by the CPU is represented as




Assuming the CPU is readily available at the time of arrival of $P_1$, $P_1$ starts executing without any waiting in the 'Ready' queue. Hence the waiting time for $P_1$ is zero.
The waiting time for all processes are given as:

Waiting Time (WT) for $P_1$ = 0ms ($P_1$ starts executing first)
(WT) for $P_2$ = 10ms ($P_2$ starts executing after completing $P_1$)
(WT) for $P_3$ = 15ms ($P_3$ starts executing after completing $P_1$ & $P_2$)

Average WT = (waiting time for all processes) / No. of processes
= (waiting time for ($P_1 + P_2 + P_3$))/3
= (0 + 10 + 15)/3 = 25/3
= 8.33 milliseconds.

TAT for $P_1$ = 10 ms ( Time spent in Ready Queue + Execution Time)

TAT for $P_2$ = 15 ms " "

TAT for $P_3$ = 22 ms " "

Average TAT = ( Turn Around Time for all process / No. of process

= ( TAT for ($P_1 + P_2 + P_3$)) / 3

= (10 + 15 + 22) / 3 = 47/3

= 15.66 milliseconds.

Average TAT is the sum of average waiting time & average execution time.

Average Execution Time = (Execution time for all process) / No. of process

= (Execution time for ($P_1 + P_2 + P_3$)) / 3

= (10 + 5 + 7) / 3 = 22/3

= 7.33

Average TAT = Average waiting time + Average execution time

= 8.33 + 7.33

= 15.66 milliseconds.

## Last-come-first served (LCFS) / LIFO scheduling

→ LCFS scheduling algorithm also allocates cpu time to the processes based on the order in which they are entered in the 'Ready' queue. The last entered process is serviced first.

→ LCFS scheduling is also known as last in first out (LIFO) where the process, which is put last into the 'Ready' queue, is serviced first.

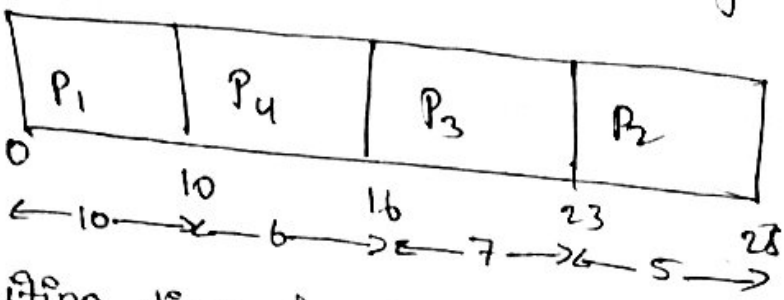
Example: Three processes with process IDs $P_1$, $P_2$, $P_3$ with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together in the order $P_1, P_2, P_3$ (Assume only $P_1$ is present in the 'Ready' queue when the scheduler picks it up & $P_2$, $P_3$ entered 'Ready' queue after that).

Now a new process $P_4$ with estimated completion time 6ms enters the 'Ready' queue after 5ms of scheduling $P_1$.

Calculate the waiting time & TAT for each process & the average waiting time & TAT (Assuming there is no I/O waiting for the processes). Assume all the processes contain only CPU operation & no I/O operations are involved.

→ Initially there is only $P_1$ available in the Ready queue & the scheduling sequence will be $P_1, P_2, P_3, P_4$ enters the queue during the execution of $P_1$

and becomes the last process entered the 'Ready' Queue. Now the order of execution changes to $P_1, P_4, P_3$ & $P_2$.




The waiting time for all the processes is given as

WT for $P_1$ = 0ms ($P_1$ starts executing first)

WT for $P_4$ = 5ms ($P_4$ starts executing after completing $P_1$. But $P_4$ arrived after 5ms of execution of $P_1$. Hence its WT = Execution start time − Arrival time = 10 − 5 = 5)

WT for $P_3$ = 16ms ($P_3$ starts executing after completing $P_1$ & $P_4$)

WT for $P_2$ = 23ms ($P_2$ starts executing after completing $P_1$, $P_4$ & $P_3$)

Average W.T = (waiting time for all process)/No. of process
= (WT for ($P_1 + P_4 + P_3 + P_2$))/4
= (0 + 5 + 16 + 23)/4 = 44/4
= 11 milliseconds.

TAT for $P_1$ = 10ms (Time spent in Ready Queue + Execution Time)

TAT for $P_4$ = 11m (Time spent in " " + " "
= (Execution start time − Arrival Time) + Estimated ET = (10 − 5) + 6 = 5 + 6)

TAT for $P_3$ = 23ms

TAT for $P_2$ = 28ms

Average (TAT) = (TAT for all process)/No. of process
= TAT for ($P_1 + P_4 + P_3 + P_2$))/4
= (10 + 11 + 23 + 28)/4 = 72/4
= 18 milliseconds.

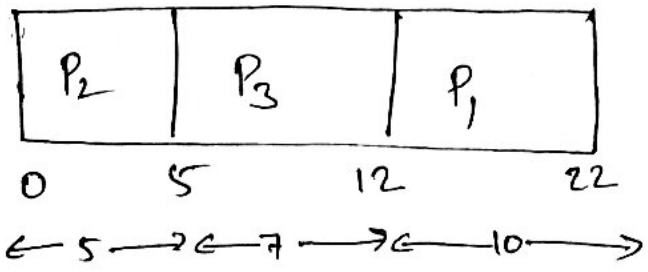
## Shortest Job First (SJF) scheduling:

→ shortest Job First (SJF) scheduling algorithm sorts the 'Ready' queue. Each time a process relinquishes the cpu (either the process terminates or enters the 'wait' state waiting for I/O or system resource) to pick the process with shortest (least) estimated completion run time.

→ In SJF, the process with the shortest ~~process and so on~~. Estimated run time is scheduled first, followed by the next shortest process, & so on.

Example:

Three processes with process IDs $P_1$, $P_2$, $P_3$ with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together. Calculate the waiting time & Turn Around Time for each process & the Average WT & TAT




The estimated execution time of $P_2$ is the least (5ms) followed by $P_3$ (7ms) & $P_1$ (10ms).

The waiting time for all processes are given as

WT for $P_2$ = 0ms ($P_2$ start executing first)

Waiting Time for $P_3$ = 5ms ($P_3$ starts executing after completing $P_2$)

Waiting Time for $P_1$ = 12 ms ($P_1$ starts executing after completing $P_2$ & $P_3$)

Average waiting time = (waiting time for all processes) / No. of processes.

= (waiting time for ($P_2 + P_3 + P_1$))/3

= (0+5+12)/3 = 17/3

= 5.66 milliseconds.

TAT for $P_2$ = 5ms (Time spent in Ready Queue + ET)

TAT for $P_3$ = 12 ms "

TAT for $P_4$ = 22 ms

Average TAT = (TAT for all process) / No. of processes

= (TAT for ($P_2 + P_3 + P_1$))/3

= (5 + 12 + 22)/3 = 39/3

= 13 milliseconds

Average TAT is the sum of average WT & average ET

The average ET = (ET for all processes) / No. of processes

= (Execution time for ($P_1 + P_2 + P_3$))/3

= (10+5+7)/3 = 22/3 = 7.33

Average TAT = ~~Aoo~~ Average WT + Average ET

= 5.66 + 7.33

= 13 milliseconds

Priority Based Scheduling:

→ The TAT & waiting-time (WT) for process in non-preemptive scheduling varies with the type of scheduling algorithm.

→ Priority based non-preemptive scheduling algorithm ensures that a process with high priority is serviced at the earliest compared to other low priority process in the 'Ready' queue.

→ The priority of a task/process can be indicated through various mechanism such as SJF and another way of priority assigning is associating a priority to the task/process at the time creation of the task/process.

→ The priority is a number ranging from 0 to the maximum priority supported by the OS.

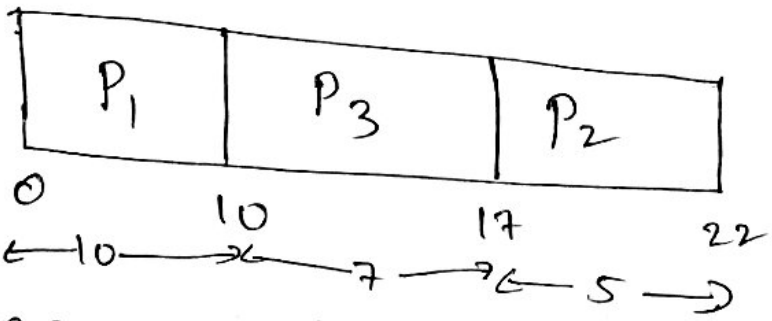
Example: windows CE supports 256 levels of priority number (0 to 255 priority numbers).

0 indicates the highest priority &

255 indicates the lowest priority.

Example: Three process with process IDs $P_1, P_2, P_3$ with estimated completion time 10, 5, 7 milliseconds & priorities 0, 3, 2 (0 – highest priority, 3 – lowest priority) respectively enter the ready queue together. Calculate the WT & TAT for each process & the ~~Avg~~ Average WT & TAT (Assuming there is no I/O waiting for the process) in priority based Scheduling algorithm.

The Scheduler sorts the 'Ready' queue based on the priority & Schedules the process with the highest priority ($P_1$ with priority no. 0) first & the next high priority process ($P_3$ with priority number 2) on second & so on. The order in which the processes are Scheduled for execution is represented as




The waiting time for all the processes are given as

WT for $P_1$ = 0ms ($P_1$ starts Executing first)

WT for $P_3$ = 10ms ($P_3$ " " " completing $P_1$)

WT for $P_2$ = 17ms ($P_2$ " " " " " & $P_3$)

Average WT = (waiting time for all process) / No. of process

= (waiting time for ($P_1 + P_3 + P_2$))/3

= (0 + 10 + 17)/3 = 27/3

= 9 milliseconds.

TAT for $P_1$ = 10 ms (Time spent in Ready Queue + Execution Time)

" " $P_3$ = 17 ms

" " $P_2$ = 22 ms

Average TAT = (TAT for all processes) / No. of processes.

= (Turn Around Time for ($P_1 + P_3 + P_2$)) / 3

= (10 + 17 + 22) / 3

= 49/3

= 16.33 milliseconds

## Preemptive Scheduling

→ Preemptive scheduling is employed in systems, which implement preemptive multitasking model.

→ In preemptive scheduling, every task in the 'Ready' queue gets a chance to execute.

→ When & how often each process gets a chance to execute is dependent on the type of preemptive scheduling algorithm used for scheduling the process.

→ The act of moving a 'Running' process/task into the 'Ready' queue by the scheduler, without the processes requesting for it is known as 'preemption'.

→ The two important approaches adopted in preemptive scheduling are time-based preemption & priority-based preemption.

## Preemptive SJF Scheduling / Shortest Remaining Time (SRT)

→ The preemptive SJF scheduling algorithm sorts the 'Ready' queue when a new process enters the 'Ready' queue & checks whether the execution time of the new process is shorter than the remaining of the total estimated time for the currently executing process.

→ If the execution time of the new process is less, the currently executing process is preempted and the new process is scheduled for execution.

→ Thus preemptive SJF scheduling always compares the execution completion time of a new process entered the 'Ready' queue with the remaining time for completion of the currently executing process & schedules the process with shortest remaining time for execution.

→ Preemptive SJF scheduling is also known as Shortest Remaining Time (SRT) scheduling.

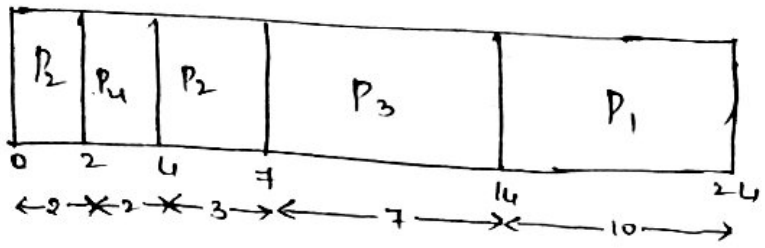
Example:

Three processes with process IDs $P_1, P_2, P_3$ with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together. A new process $P_4$ with estimated completion time 2ms enters the 'Ready' queue after 2ms. Assume all the processes contain only CPU operation and no I/O operations are involved.

At the beginning, there are only three processes ($P_1$, $P_2$ and $P_3$) available in the 'Ready' queue and the SRT scheduler picks up the process with the shortest remaining time for execution completion (In this example, $P_2$ with remaining time 5ms) for scheduling. The execution sequence diagram for this is same as that of example 1 under non-preemptive SJF scheduling.

Now process $P_4$ with estimated execution completion time 2ms enters the 'Ready' queue after 2ms of start of execution of $P_2$. Since the SRT algorithm is preemptive, the remaining time for completion of process $P_2$ is checked with the remaining time for completion of process $P_4$. The remaining time for completion of $P_2$ is 3ms which is greater than that of the remaining time for completion of the newly entered process $P_4$ (2ms). Hence $P_2$ is preempted and $P_4$ is scheduled for executed. $P_4$ continues its execution to finish since there is no new process entered in the 'Ready' queue during its execution. After 2ms of scheduling $P_4$ terminates and now the scheduler again sorts the 'Ready' queue based on the remaining time for completion of the processes present in the 'Ready' queue. Since the remaining time for $P_2$ (3ms), which is preempted by $P_4$ is less than that of the remaining time for other processes in the 'Ready' queue, $P_2$ is scheduling for execution. Due to the arrival of the process $P_4$ with execution time 2ms, the 'Ready' queue is re-stored in the order $P_2$, $P_4$, $P_2$, $P_3$, $P_1$. At the

beginning it was P2, P3, P1. The execution sequence now changes as per the following diagram




The waiting time for all the processes are given as

WT for $P_2$ = 0ms + (4-2) ms = 2ms ($P_2$ starts executing first and is interrupted by $P_4$ and has to wait till the completion of $P_4$ to get the next CPU slot)

WT for $P_4$ = 0ms ($P_4$ starts executing by preempting $P_2$ since the execution time for completion of $P_4$ (2ms) is less than that of the remaining time for execution completion of $P_2$ (Here it is 3ms))

WT for $P_3$ = 7 ms ($P_3$ starts executing after completing $P_4$ and $P_2$)

WT for $P_1$ = 14 ms ($P_1$ starts executing after completing $P_4$, $P_2$ and $P_3$)

Average waiting time = (WT for all processes) / No. of processes

= (WT for ($P_4$ + $P_2$ + $P_3$ + $P_1$)) / 4

= (0+2+7+14)/4 = 23/4

= 5.75 milliseconds

TAT for $P_2$ = 7ms (Time spent in Ready Queue + Execution Time)

TAT for $P_4$ = 2 ms (Time spent in Ready Queue + Execution time = (Execution start time - Arrival Time) + Estimated Execution Time = (2-2)+2)

TAT for $P_3$ = 14 ms (Time spent in Ready Queue + Execution Time)

TAT for $P_1$ = 24ms (Time spent in Ready Queue + Execution Time)

Average Turn Around Time = (Turn Around Time for all the processes)/
No. of processes

= (Turn Around Time for (P2 + P4 + P3 + P1))/4

= (7 + 2 + 14 + 24)/4 = 47/4

= 11.75 milliseconds.

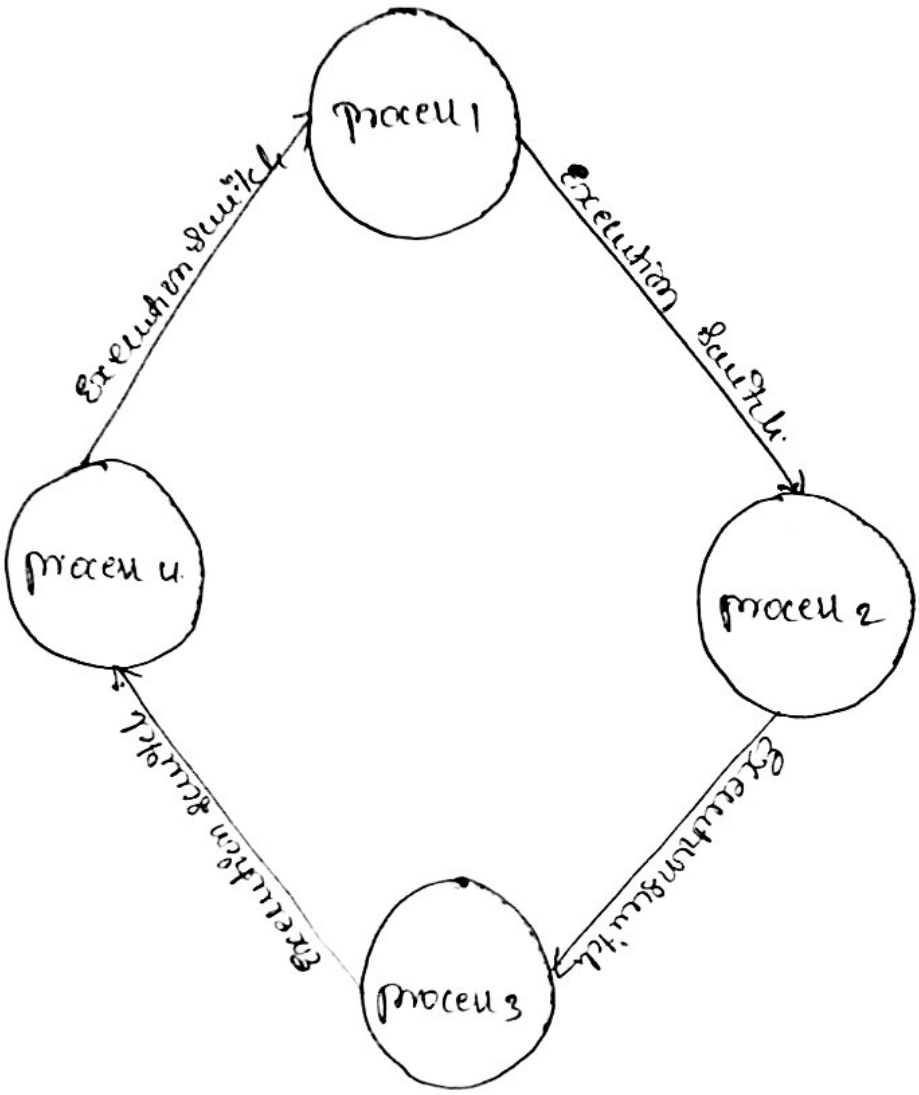
## Round Robin (RR) Scheduling:

→ The term Round Robin is very popular among the sports & game activities. It is associated with football ~~tourn~~ league.

→ In the 'Round Robin' league each team in a group gets an equal chance to play against the rest of the teams in the same group whereas in the 'knock out' league the losing team in a match moves out of the tournament.

→ In the process scheduling context also, 'Round Robin' brings the same message "Equal chance to all".
In Round Robin scheduling context, each process in the 'Ready' queue is executed for a pre defined time slot.

→ The execution starts with picking up the first process in the Ready queue. It is executed for a pre-defined time & when the pre-defined time elapses or the process completes, the next process in the 'Ready' queue is selected for execution. This is repeated for all the processes in the 'Ready' queue.




The time slice is provided by the timer tick feature of the time management unit of the OS kernel (Refer the Time management section under the subtopic 'The Real-Time kernel' for more details on Timer tick). Time slice is kernel dependent and it varies in the order of a few microseconds to milliseconds.

## Priority Based Scheduling :-

Priority based preemptive scheduling algorithm is same as that of the non-preemptive priority based scheduling except for the switching of execution between tasks. In preemptive scheduling, any high priority process entering the 'Ready' queue is immediately scheduled for execution whereas in the non-preemptive scheduling any high priority process entering the 'Ready' queue is scheduled

only after the currently executing process completes its execution or only when it voluntarily relinquishes the CPU. The priority of a task/process in preemptive scheduling is indicated in the same way as that of the mechanism adopted for non-preemptive multitasking. Refer the non-preemptive priority based scheduling discussed in an earlier section of this chapter for more details.

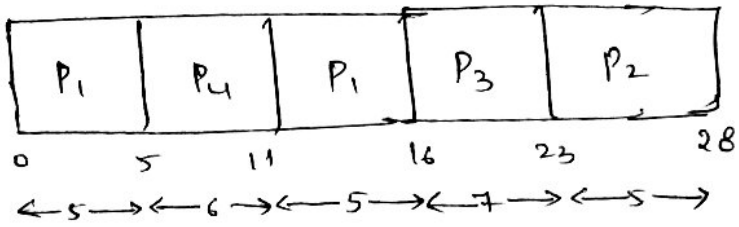
Example :-

Three processes with process IDs $P_1, P_2, P_3$ with estimated completion time 10, 5, 7 milliseconds and priorities 1, 3, 2 (0 - highest priority, 3 - lowest priority) respectively enters the ready queue together. A new process $P_4$ with estimated completion time 6ms and priority 0 enters the 'Ready' queue after 5ms of start of execution of $P_1$. Assume all the processes contain only CPU operation and no I/O operations are involved.

At the beginning, there are only three processes ($P_1$, $P_2$ and $P_3$) available in the 'Ready' queue and the scheduler picks up the process with the highest priority (On this example $P_1$ with priority 1) for scheduling.

Now process $P_4$ with estimated execution completion time 6ms and priority 0 enters the 'Ready' queue after 5ms of start of execution of $P_1$. Since the scheduling algorithm is preemptive, $P_1$ is preempted by $P_4$ and $P_4$ runs to completion. After 6ms of scheduling, $P_4$ terminates and now the scheduler again sorts the 'Ready' queue for process with highest priority. Since the priority for $P_1$ (Priority 1), which is preempted by $P_4$ is higher than that of $P_3$ (priority 2) and $P_2$ (priority 3), $P_1$ is again picked up for execution by the scheduler. Due to the arrival of the process $P_4$ with priority 0, the 'Ready' queue is restored in the order $P_1$, $P_4$, $P_1$, $P_3$, $P_2$. At the beginning if

- was $P_1, P_3, P_2$. The execution sequence now changes as per the following diagram




The WT for all the process are given as

WT for $P_1$ = 0 + (11 − 5) = 0 + 6 = 6 ms

($P_1$ starts executing first and gets preempted by $P_4$ after 5ms and again gets the CPU time after completion of $P_4$)

WT for $P_4$ = 0 ms

($P_4$ starts executing immediately on entering the 'Ready queue, by preempting $P_1$)

WT for $P_3$ = 16 ms ($P_3$ starts executing after completing $P_1$ & $P_4$)

WT for $P_2$ = 23 ms ($P_2$ starts executing after completing $P_1, P_4$ & $P_3$)

AWT = (WT for all the processes) / No. of processes

= (WT for ($P_1 + P_4 + P_3 + P_2$)) / 4

= (6 + 0 + 16 + 23) / 4 = 45/4

= 11.25 milli seconds

TAT for $P_1$ = 16 ms (Time spent in Ready Queue + execution time)

TAT for $P_4$ = 6 ms (Time spent in Ready Queue + execution time) = (execution starts time − Arrival time) + Estimated execution time = (5 − 5) + 6 = 0 + 6

TAT for $P_3$ = 23 ms (Time spent in Read Queue + Execution time)

TAT for $P_2$ = 28ms (Time spent in Ready Queue + Execution Time)

Average Turn Around Time = (Turn Around Time for all the processes) / No. of processes

= (16 + 6 + 23 + 28) / 4 = 73/4

= 18.25 milliseconds.