UNIT - 03

Process Management and Threads

- A program is a passive entity that doesnot perform any actions by itself, it has to be executed if the actions it calls for are to take place.

A process is an execution of a program. It actually performs the actions specified in a program. An operating system shares the CPU among processes.

A thread is also an execution of a program but it functions in the environment of a process i.e., it uses the code, data and resources of a process. It is possible that many threads to function in the environment of the same process; they share its code, data and resources.

- There are two views of program execution (process)

a. Programmer's view

To achieve concurrent execution of a program

b. OS view

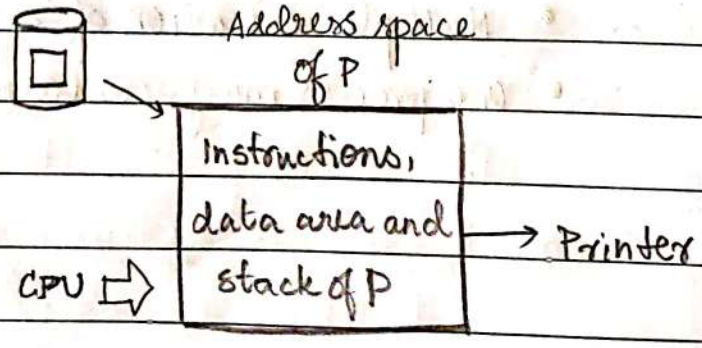
To achieve execution of a program

- A program and an abstract view of its execution.

Execution of a program P

Ex: Program P

```
file info;
int item;
open (info, "read");
.....
....
stop;
```



- A process comprises six components:
id, code, data, stack, resources, CPU state
where
id - the unique id created by the OS
code - the code of the program (text of the program)
data - the data used in the execution of the program, including data from files.
stack - contains parameters of functions and procedures called during execution of the program and their return addresses.
resources - the set of resources allocated by the OS.
CPU state - It comprises contents of the PSW and the general purpose registers of the CPU.

* Relationships between Processes and Programs:

a. One-to-one
A one-to-one relationship exists when a single execution of a sequential program is in progress.

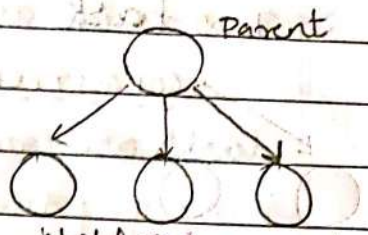
b. Many-to-one
A many-to-one relationship exists when many simultaneous executions of a program occurs or when execution of a concurrent program.
where a concurrent processes are the processes that coexist in the system at the same time.
- Multiple execution of one program creates multiple processes
- One process may contain instructions from multiple programs

* Child Processes:

The kernel initiates an execution of a program by creating a primary process which makes system calls to create other processes which becomes its child process and the primary process becomes their parent. A child process may itself create a process and so on.

A parent-child relationships can be represented in the form of a process tree which has the primary process (parent) as the root.

PROCESS TREE



The child process may inherit some of the resources of its parent; it could obtain additional resources during its operation through system calls.

- Benefits of child processes due to multitasking:

• computation speed

without a child processes the primary process would have been performed sequentially but with child processes it would be performed concurrently hence reducing the run time of the application

• Priority for critical functions

A child process that performs a critical function may be assigned a high priority it may help to meet the real-time requirements of an application.

• Guarding a parent process against error

The kernel aborts a child process if an error occurs during its operation. The parent process is not affected by this error.

- To facilitate use of child processes, the kernel provides operations for:

1. Creating a child process and assigning a priority to it.
2. Terminating a child process.

3. Determining the status of a child process.
4. Sharing, communication and synchronization between processes.

* Concurrency and Parallelism

Parallelism is the quality of ocurring at the same time. Two tasks are parallel if they are performed at the same time.

Concurrency is the illusion of parallelism. Thus two tasks are concurrent if there is an illusion that they are being performed in parallel but only one of them may be performed at any time actually.

In an OS concurrency is obtained by interleaving operation of processes on the CPU, which creates the illusion that these processes are operating at the same time. Whereas parallelism is obtained by using multiple CPUs as in a multiprocessor system and operating different processes on these CPUs.

* Implementation of Processes:

In OS view, a process is a unit of computational work. Thus the kernel's primary task is to control operation of processes to provide effective utilisation of the computer system.

The kernel allocates resources to a process, protects the process and its resources from interference by other processes and ensures that the process gets to use the CPU until it completes its operation.

The kernel is activated when an event which requires kernel's attention occurs, which leads to either a hardware interrupt or a system call.

— Functions of the kernel for controlling processes:

1. Context save:

Saving the CPU state and information concerning resources of the process whose operation is interrupted.

2. Event Handling:

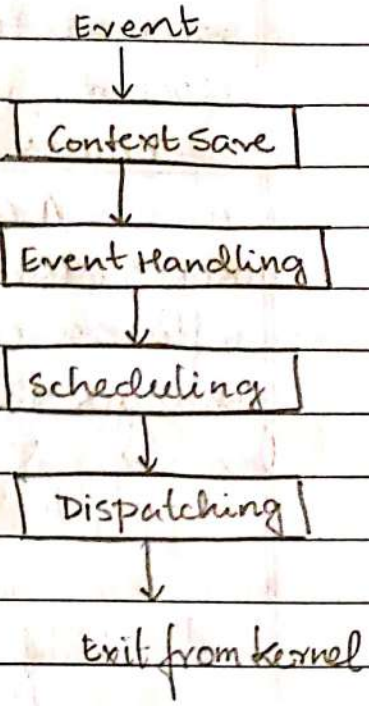
Analyzing the condition that led to an interrupt or the request by a process that led to a system call and taking appropriate actions.

3. Scheduling:

Selecting the process to be executed next on the CPU.

4. Dispatching:

Setting up access to resources of the scheduled process and loading its saved CPU state in the CPU to begin or resume the operation.



* Process States and State Transitions:

Process state indicates the nature of the current activity of a process. The four fundamental states are:

a. Ready: The process is ready for its execution, i.e., it wishes to use the CPU to continue its operation, however it has not been dispatched.

b. Running: The process is currently being executed by the CPU.

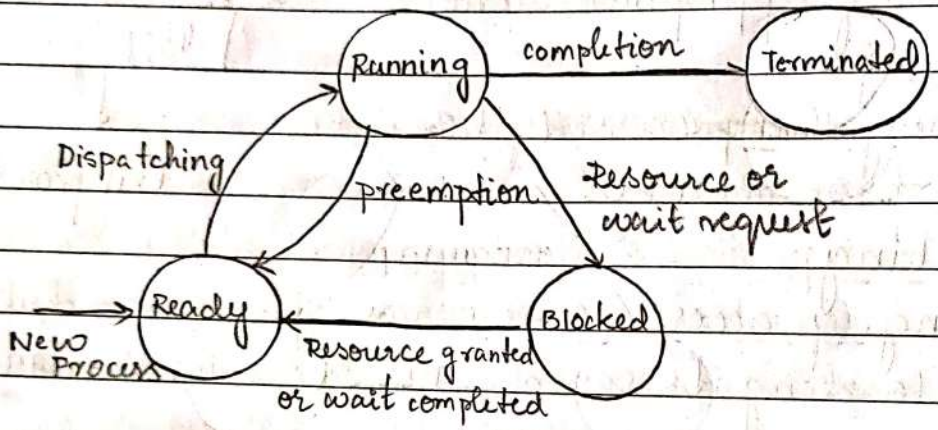
c. Blocked: The process has to wait until a resource request made by it is granted, or it wishes to wait until a specific event occurs.

d. Terminated: The execution of the process may be completed or it may be aborted by the kernel for some reason.

A conventional computer system contains only one CPU, hence only one program can be in running state whereas there can be any number of processes in the blocked, ready and terminated states.

Process State Transitions

A state transition for a process is a change in its state. When the event occurs, the kernel decides its influence on the activities in processes and accordingly changes the state of an affected process.



Fundamental State Transitions for a process.

Causes of Fundamental state transitions for a Process

- **Ready → Running**

The process is dispatched. The CPU begins or resumes execution of its instructions.

- **Blocked → Ready**

A request made by the process is granted or an event for which it was waiting occurs.

- **Running → Ready**

The process is preempted because the kernel decides to schedule some other process. This transition occurs either because a higher priority process becomes ready or because the time slice of the process elapses.

- **Running → Blocked**

The process in operation makes a system call to

indicate that it wishes to wait until some resource request made by it is granted or until a specific event occurs in the system. The five major causes of blocking are:
- Process requests an I/O operation
- Process requests a resource.
- Process wishes to wait for a specified time interval.
- Process waits for a message from another process
- Process waits for some action by another process.

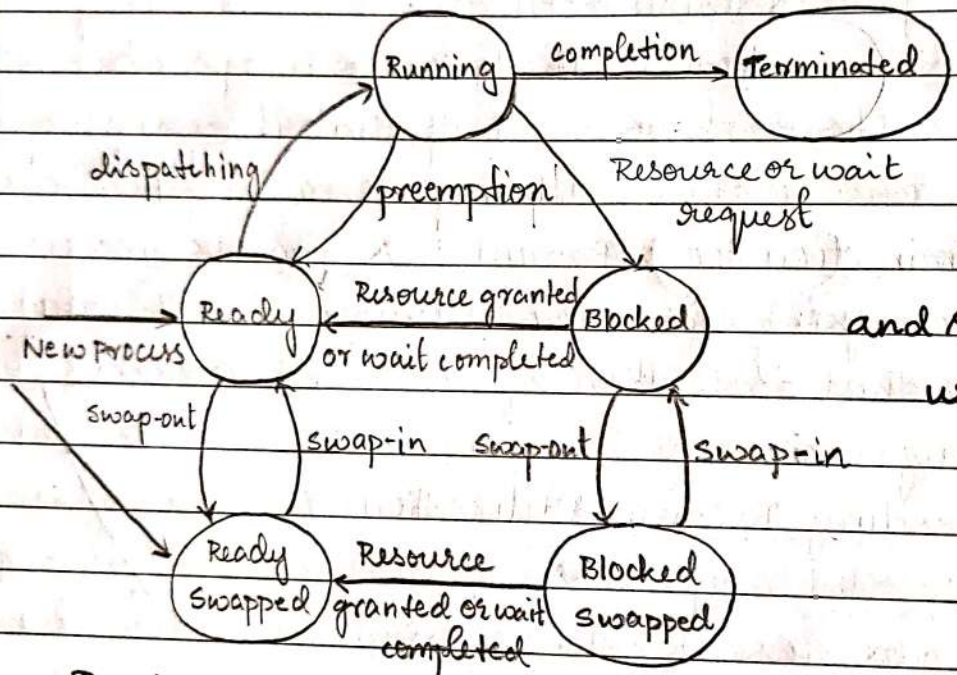
- Running → Terminated

Execution of the program is completed. Five primary reasons for process termination are:
- Self termination: The process in operation either completes its task or realises that it cannot operate further and makes a 'terminate me' system call.
- Termination by a Parent: A process makes a terminate system call to terminate a child process when it finds that execution of the child process is no longer necessary.
- Exceeding Resource Utilization: An OS may limit the resources that a process may consume thus a process exceeding a resource limit would be aborted by the kernel.
- Abnormal Conditions during operation: The kernel aborts a process if an abnormal condition arises due to the instruction being executed.
- Incorrect Interaction with other processes: The kernel may abort a process if it gets involved in a deadlock.

- Suspended Processes:

If the process present in the ready or blocked state is swapped out of memory then it has to be swapped back into the memory before it can resume its activity. Hence these processes are no longer in the ready or blocked state thus the kernel defines new state for it these processes are called the suspended processes.

There are two suspended states called
- ready swapped and
- blocked swapped.



Process states and state transitions using two swapped states.

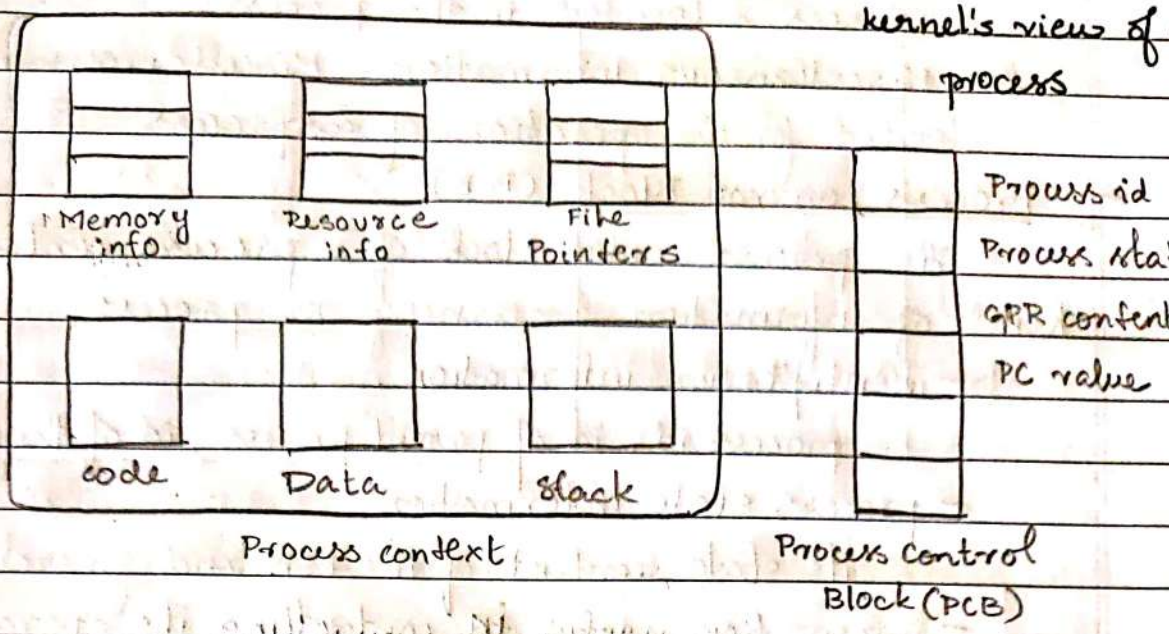
Ready → Ready Swapped
Blocked → Blocked Swapped
} swap-out action.

Ready Swapped → Ready
Blocked Swapped → Blocked
} swap-in action.

The Blocked Swapped → Ready Swapped transition takes place if the request for which the process was waiting is granted even while the process is in a suspended state.

The new process is put either in the ready state or in the ready swapped state depending on availability of memory.

* Process context and the Process control Block:



The kernel allocates resources to a process and schedules it for use of the CPU. Thus kernel's view of a process consists of two parts:

- code, data and stack of the process and information concerning memory and other resources allocated to it.
- information concerning execution of a program such as the process state, the CPU state including the stack pointer and some other items of information.

These two parts of the kernel's view are contained in the process context and process control block (PCB) respectively

- The process context consists of the following:

1. Address space of the process: The code, data and stack components of the process.
2. Memory allocation information: Information concerning memory areas allocated to a process.
3. Status of file processing activities: Information about files being used, such as current positions in the files.
4. Process Interaction Information: Information necessary to control interaction of the process with other processes.

5. Resource Information: Information concerning to resources allocated to the process.
6. Miscellaneous Information: Miscellaneous information needed for the operation of a process.

— Process control Block (PCB)

The process control block of a process contains three kinds of information concerning the process.

- identification information.
  process id, id of parent process, id of the user who created it
- process state information.
  its state, contents of the PSW and general purpose registers
- information useful in controlling its operation
  priority, its interaction with other process. It also contains a pointer field that is used by the kernel to form PCB lists for scheduling.

Fields of the Process Control Block (PCB):

	PCB Fields	Contents
1.	Process id.	The unique id assigned to the process at its creation.
2.	Parent and child ids.	These ids are used for process synchronization typically for a process to check if a child process has terminated.
3.	Priority	A process is assigned a priority at its creation.
4.	Process state	The current state of the process.
5.	PSW	Snapshot (image) of the PSW when the process last got blocked or was preempted.
6.	GPRs	contents of the general purpose registers when the process last got blocked or preempted.
7.	Event information	For a process in blocked state, this field contains information concerning the event for which the process is waiting.

8.	Signal Information	Information concerning locations of signal handlers.
9.	PCB pointer	It is used to form a list of PCB's for scheduling.

* Context Save, Scheduling and dispatching:

The context save function saves the CPU state of the interrupted process in its PCB and saves information concerning its context whenever an event occurs.

The scheduling function uses the process state information from PCB's to select a ready process for execution and passes its id to the dispatching function.

The dispatching function sets up the context of the selected process, changes its state to running and loads the saved CPU state from its PCB into the CPU.

* Event Control Block (ECB)

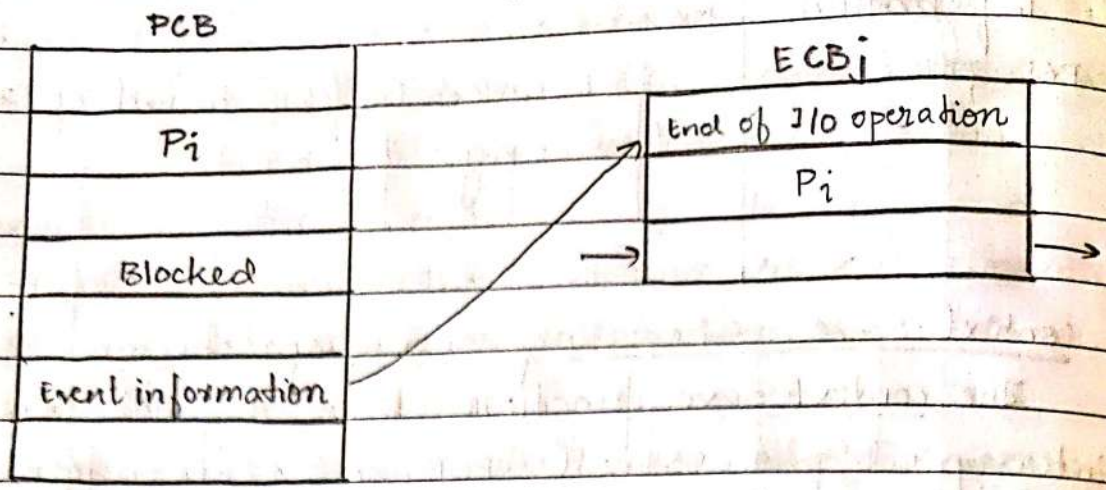
Event Description
Process id
ECB Pointer

An ECB contains three fields
- Event description field describes an event
- Process id field contains the id of the process awaiting the event.

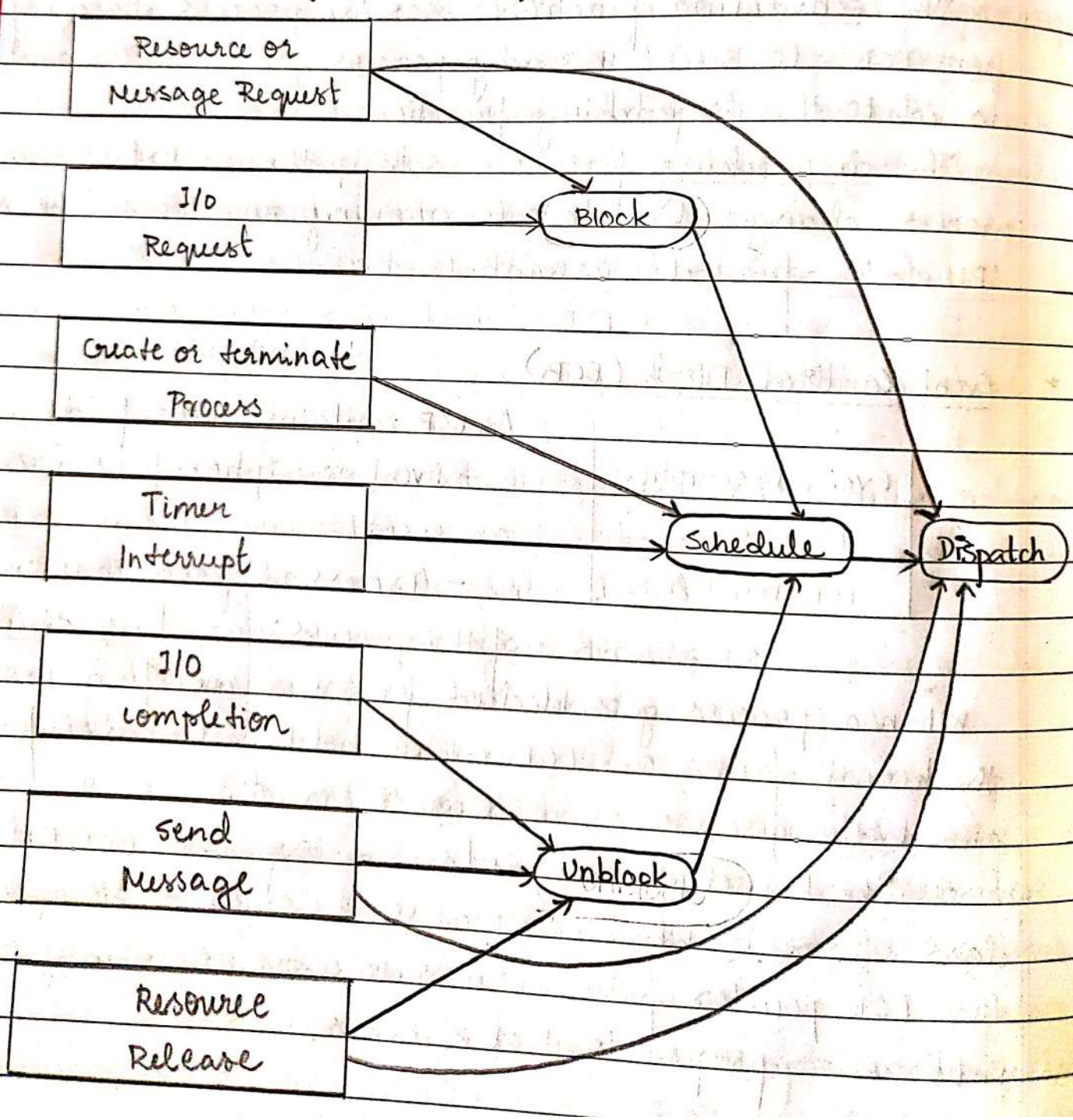
When a process gets blocked for occurance of an an event the kernel forms an ECB which holds relevent information about the process as well as the event.

- The kernel can maintain a separate ECB list for each class of events like interprocess messages or I/O operations, so the ECB pointer field is used to enter the newly created ECB into an appropriate list of ECB's.

PCB - ECB interrelationship



Event handling actions of the kernel.



The blocked action always changes the state of the process that made a system call from ready to blocked. The unblocked action finds a process whose request can be fulfilled now and changes its state from blocked to ready.

A system call for requesting a resource leads to a block action if the resource cannot be allocated to the requesting process. This action is followed by scheduling and dispatching because another process has to be selected for use of the CPU. The block action is not performed if the resource can be allocated straightaway. In this case, the interrupted process is simply dispatched again.

When a process releases a resource, an unblock action is performed followed by scheduling and dispatching if some other process is waiting for the resource release. This is because the unblocked process may have a higher priority than the process that released the resource. If no process is unblocked because of the event then scheduling is skipped.

* Threads:

Applications use concurrent processes to speed up their operation. However, switching between processes within an application incurs high process switching overhead because the size of the process state information is large. Hence a thread, an alternate model of execution of a program that could provide concurrency within an application with less overhead was developed.

Process switching overhead has two components.

- Execution related overhead:

The CPU state of the running process has to be saved and the CPU state of the new process has to be loaded in the CPU.

- Resource - use related overhead:

The process context also has to be switched. It involves switching of the information about resources allocated to the process such as memory and files and interaction of the process with other processes.

A process creates a thread through a system call. As the thread does not have any resource or context of its own it operates by using the context of the process and accesses the resources of the process through it. An application would create child processes to execute different parts of its code and each child process can create threads to achieve concurrency.

Each thread has its own stack and thread control block (TCB) which is analogous to the PCB.

The thread control block stores the following information:

1. Thread scheduling information - thread id, priority, state
2. CPU state - contents of PSW and GPR's.
3. Pointer to PCB of parent process
4. TCB pointer - to make lists of TCBs for scheduling

The use of threads splits the process state into

- the resource state:
- the execution state (CPU state)

→ Advantages of threads over Processes.

1. Lower overhead of creation and switching

Thread state consists only of the state of a computation. Resource allocation state and communication state are not a part of thread state so creation of threads and switching between them leads to a lower overhead.

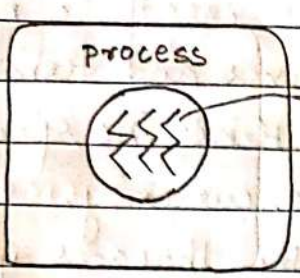
2. More Efficient communication

Threads of a process can communicate with one another through shared data, thus avoiding the overhead

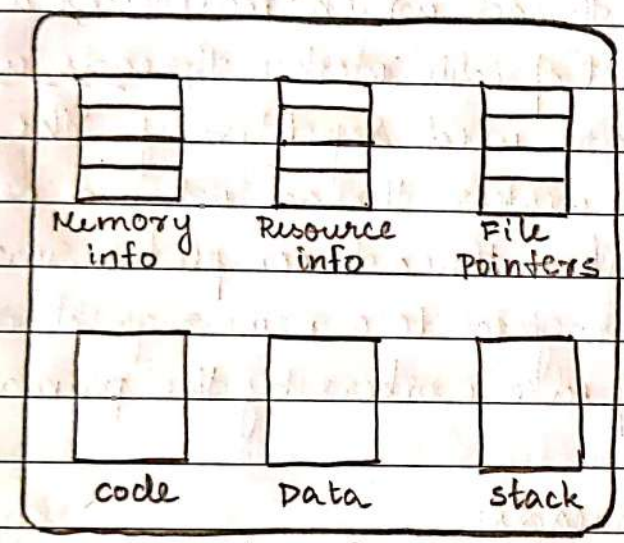
of system calls for communication.

3. Simplification of design:

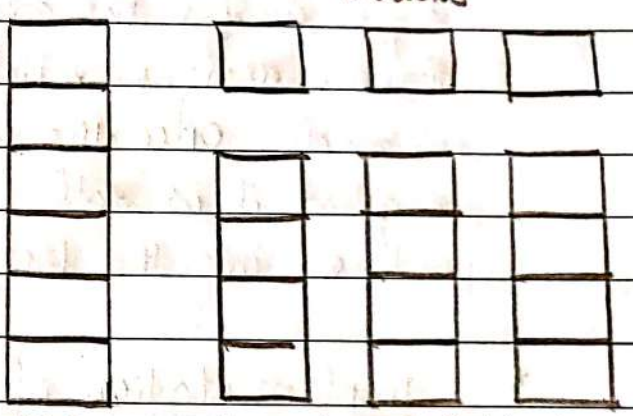
Use of threads can simplify the design and coding of applications that service requests concurrently.



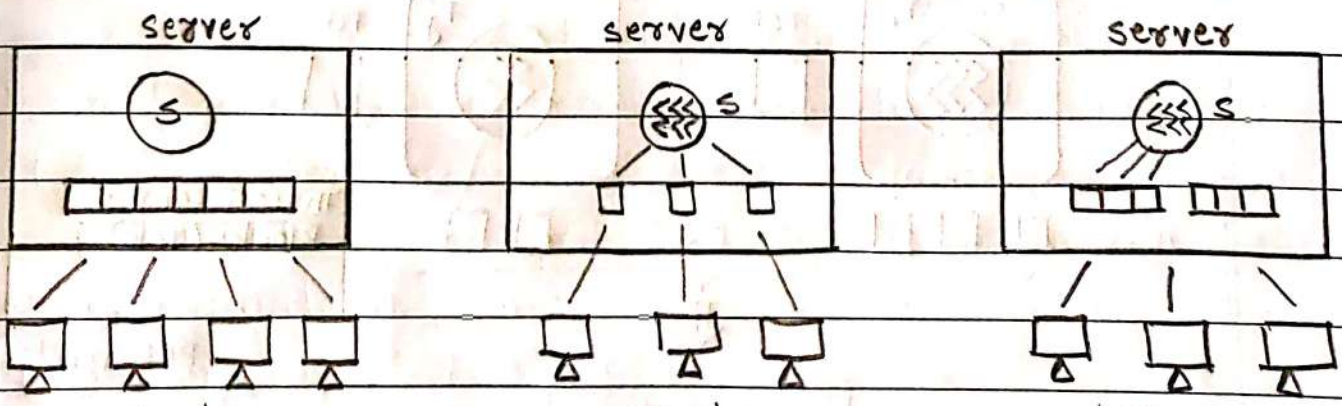
Threads in a process



Implementation of threads in process.



Applications that service requests received from users are called servers and their users are called clients.



server using sequential code

Multithreaded server

Server using a thread pool.

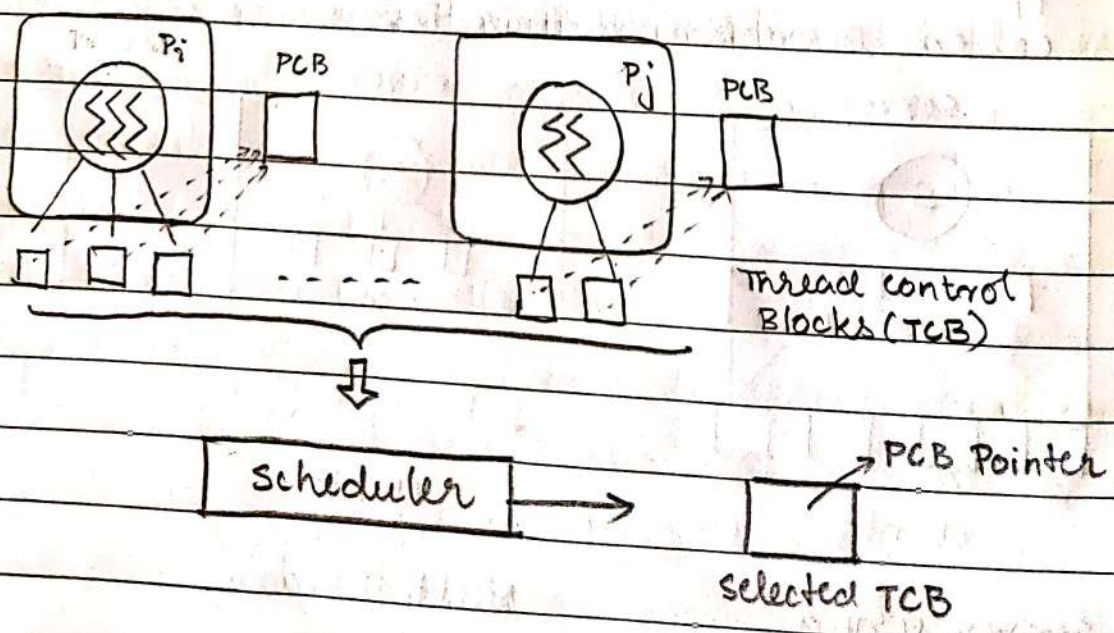
Server using sequential code - The server enters requests made by its clients in a queue and serves them one after another. If several requests are to be serviced concurrently the server would have to employ advanced I/O techniques.

Multithreaded server - The server creates a new thread for each new request it receives and terminates the thread after servicing the request. Hence no special techniques has to be employed for concurrency.

Thread Pool: - The server reuses the threads instead of destroying them after servicing requests. The thread pool consists of one primary thread which maintains a list of pending requests and list of idle worker threads where as a few worker threads are used repetitively. The primary thread assigns an idle worker thread when a new request is made. Once the worker thread completes servicing of a request it is either assigned to a new request or else it is entered into the list of idle workers by the primary thread.

* Implementation of threads:

1. Kernel-Level Threads



Kernel level threads are implemented by kernel. Hence creation and termination of kernel-level threads and checking of their status, is performed through system calls. When a process makes a 'create-thread' system call, the kernel creates a thread, assigns an id to it and allocates a thread control block (TCB). The TCB contains a pointer to the PCB of the parent process of the thread.

When an event occurs, the kernel saves the CPU state of the interrupted thread in its TCB. After event handling, the scheduler considers TCBs of all threads and selects one ready thread, the dispatcher uses the PCB pointer in its TCB to check whether the selected ~~process~~ thread belongs to a different process than the interrupted thread. If so it saves the context of the process to which the interrupted thread belongs and loads the context of the process to which the selected thread belongs. It then dispatches the selected thread i.e, changing its state to running. Hence switching between the kernel level threads of a process is ten times faster than switching between process.

- Advantages of kernel-level threads:

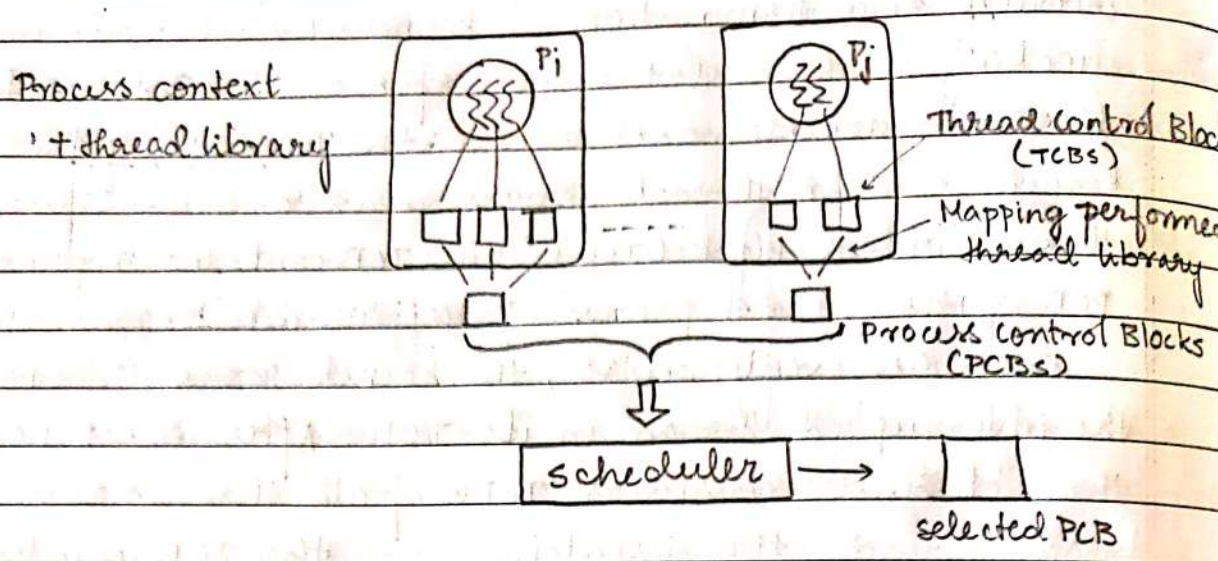
Kernel level threads provide parallelism.

It provides better computation speedup than user level threads

- Disadvantages of kernel level threads:

Switching between threads is performed by the kernel this leads to the overhead of event handling even if the interrupted thread and the selected thread belong to the same process.

2. User-level threads:



User level threads are implemented by a thread library which is linked to the code of a process.

A process invokes the library function 'create_thread' to create a new thread. The library function creates a TCB for the new thread and considers it for scheduling. When the thread in the running state invokes the library function to perform synchronization, wait until a specific event occurs, the library function performs scheduling and switches to another thread. If the thread library cannot find a ready thread in the process, it makes a 'block me' system call. The kernel now blocks the process. It will be unblocked when some event activates on of the threads and will resume execution of the thread library function which will perform scheduling and switch to newly activated thread.

- Advantages of user level threads:

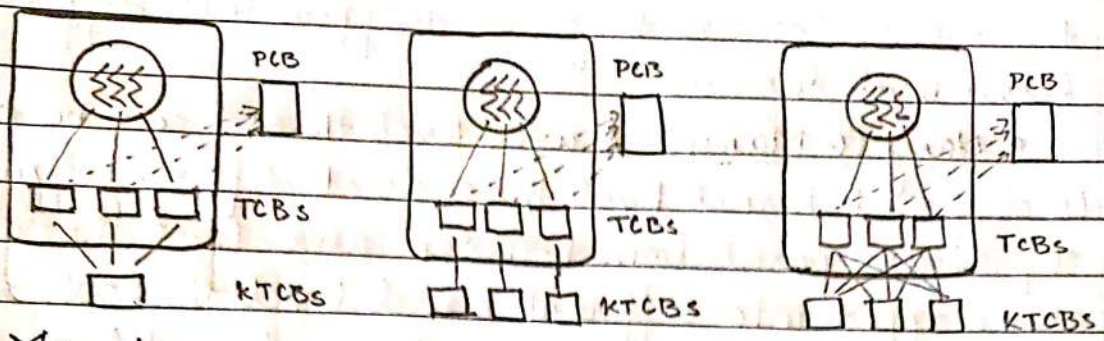
The overhead of a system call for synchronization between threads is lesser than in kernel level threads.

- Disadvantages of user level threads:

User level threads cannot provide parallelism.

Concurrency is seriously impaired if a thread makes a system call that leads to blocking.

3. Hybrid Thread Models:



Many to one One to one Many to Many

A hybrid thread model has both user-level threads and kernel level threads a method of associating user-level threads with kernel level threads. This provides low switching overhead of user level threads and the high concurrency and parallelism of kernel-level threads.

The thread library creates user level threads in a process and associates a thread control block (TCB) with each user level thread. The kernel creates kernel level threads in a process and associates a kernel thread control block (KTCB) with each kernel level thread.

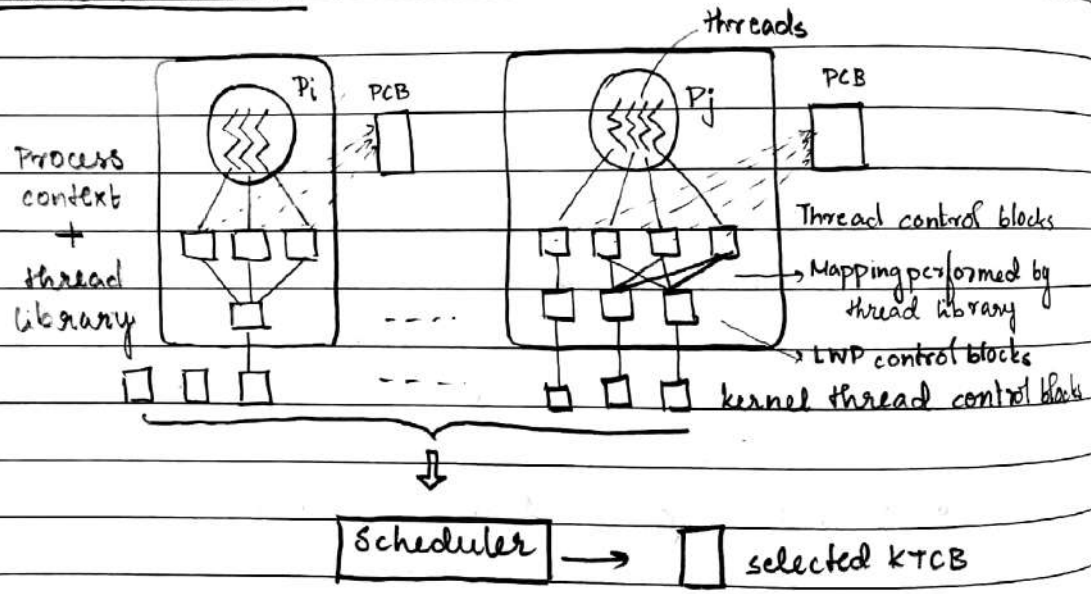
There are three methods of associating user-level threads with kernel level threads:

- Many to one: A single kernel level thread is created in a process by the kernel and all user-level threads created in a process by the thread library are associated with this kernel level thread. This is similar to user level threads: concurrent without being parallel, thread switching leads to low overhead and blocking of a user level thread leads to blocking of all threads in the process.
- One to one: Each user level thread is permanently mapped into a kernel-level thread. This is similar to kernel level threads: Threads can operate in parallel, thread switching leads to high overhead and blocking of a

user level thread does not block other user level threads of the process because they are mapped into different kernel level threads.

- Many to Many: A user level thread can be mapped to different kernel level threads at different times. It provides parallelism between user level threads that are mapped into different kernel level threads at the same time, provides low overhead of switching between user level threads that are scheduled on the same kernel level thread by the thread library.

★ Threads in Solaris:



Solaris is a UNIX 5.4 based operating system. The three kinds of entities employed to govern concurrency and parallelism within a process are:

- User threads:

User threads are analogous to user level threads. They are created and managed by a thread library, so they are not visible to the kernel. However they differ from user level threads in other aspects.

- **Light Weight Process:**

A light weight process is an intermediary between the user threads and a kernel thread. Many light weight processes may be created for a process but each of them is a unit of parallelism with in a process. User threads are mapped into light weight Processes by the thread library. This mapping can be one-to one, many-to-one, many-to-many or a suitable combination of all three. The number of light weight processes for a process and the nature of the mapping between user threads and the light weight processes is decided by the programmer.

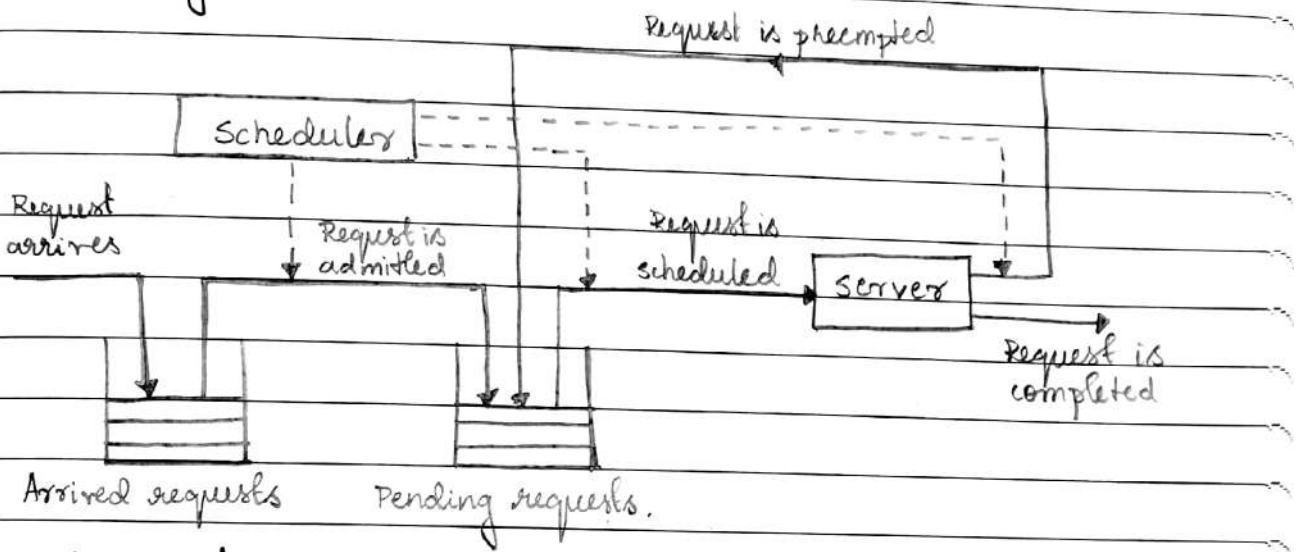
- **Kernel threads:**

A kernel thread is created by the kernel for concurrency control. The kernel creates as many kernel threads as the total number of light weight process created for all active process in the system and associates a kernel thread with each light weight process. The kernel also creates some kernel threads for its own use. A good example of this kind is a thread to handle disk or I/O in the system.

UNIT : 6

Scheduling and Message Passing

* Scheduling:



A request is the execution of a job or process and the server is the CPU. A job or a process is said to arrive when it is submitted by a user and to be admitted when the scheduler starts considering it for scheduling. An admitted job or process either waits in the list of pending requests uses the CPU or performs I/O operations. Eventually it completes and leaves the system. When a process or a job is preempted it is sent into the list of pending requests.

- Request related Scheduling terms

1. Arrival time : Time when a user submits a job or a process
2. Admission time: Time when the system starts considering job or process for scheduling.
3. Completion time: Time when a job or a process is completed.
4. Deadline: Time by which a job or a process must be completed to meet response requirement of real time application.
5. service time: the total of CPU time and I/O time required by a job to complete its operation.

6. Preemption: Forced deallocation of CPU from a job or process
7. Priority: A tie breaking rule used to select a job or a process when many jobs / processes await service.

— User Service related scheduling terms:

individual request
1. Response time (rt): Time between the submission of a subrequest for processing to the time its result becomes available. This concept is applicable to interactive process
2. Turnaround time (ta): Time between the submission of a job or a process and its completion by the system. This concept is meaningful for noninteractive jobs or processes only.
3. Weighted turnaround (w): Ratio of the turnaround time of a job or process to its own service time

$$w = ta / \text{service time}$$

4. Deadline Overrun: The amount of time by which the completion time of a job or process exceeds its deadline. Deadline overruns can be both positive or negative.
5. Fair share: A specified share of CPU time that should be devoted to execution of a process or a group of processes.
6. Response Ratio:

$$RR = \frac{\text{time since arrival} + \text{service time of a job or process}}{\text{service time of the job or process}}$$

average service
7. Mean response time ($\overline{rt}$): Average of the response times of all subrequests serviced by the system.
8. Mean turnaround time ($\overline{ta}$): Average of the turnaround times of all jobs or processes serviced by the system.

— Performance related scheduling terms:

1. Schedule length: The time taken to complete a specific set of jobs or processes.
2. Throughput: The average number of jobs, processes or subrequests completed by a system in one unit time.

Date
Page

* Fundamental Techniques of scheduling:

1. Priority based scheduling:
The process in operation should be the highest priority process requiring use of the CPU.

2. Reordering of requests:
Reordering implies servicing of requests in some other order than their arrival order.

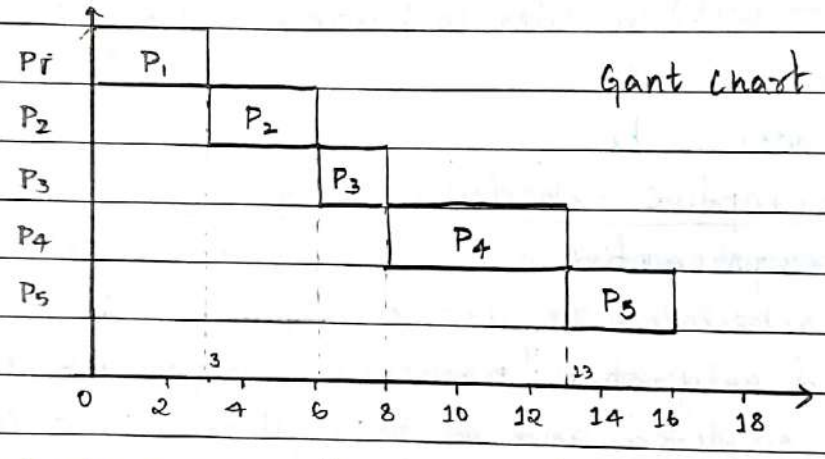
3. Variation of time slice:
Using different values of time slice for different requests (a small value for I/O bound request and a large value for CPU-bound request) in order to balance CPU efficiency and response time.

* Non Preemptive Scheduling Policies:
In non preemptive scheduling a server always services a scheduled request to completion. Thus scheduling is performed only when servicing of a previously scheduled request is completed and so preemption of request never occurs. There are three non preemptive scheduling policies:
- First come First served (FCFS) scheduling
- Shortest request next (SRN) scheduling
- Highest response ratio next (HRN) scheduling.

- First come First served Scheduling
Requests are scheduled in the order in which they arrive in the system. The list of pending requests is organised as a queue. The scheduler always schedules the first request in the list.

Q: Draw the gant chart for the following by using FCFS algorithm:

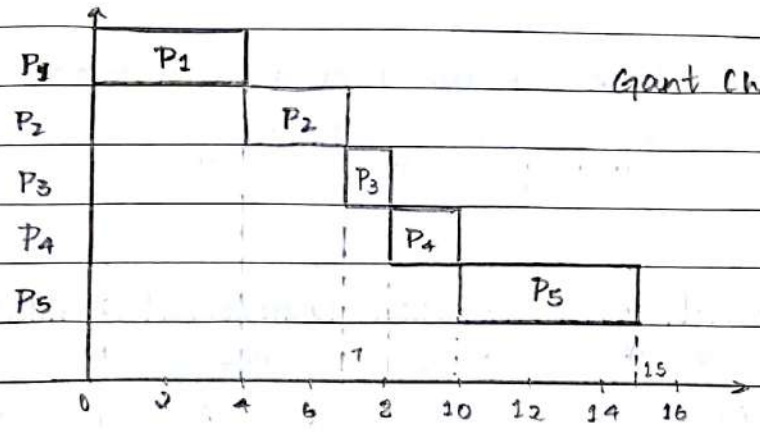
Process	Arrival time	Service time	Completion time	Turnaround time	Weighted turn around time
	AT	ST	CT	ta = CT - AT	w = ta/ST
P1	0	3	3	3	1
P2	2	3	6	4	1.33
P3	3	2	8	5	2.5
P4	5	5	13	8	1.6
P5	9	3	16	7	2.33



Q: Find the average turn around time and weighted turn around time by performing FCFS algorithm. Draw the gant chart

process	Arrival time	Service time	completion time	turn around time	Weighted turn around time
	AT	ST	CT	ta = CT - AT	w = ta/ST
P1	0	4	4	4	1
P2	1	3	7	6	2
P3	2	1	8	6	6
P4	3	2	10	7	3.5
P5	4	5	15	11	2.2

$\bar{T_a}$ = 6.8 sec $\bar{w}$ = 2.94 sec

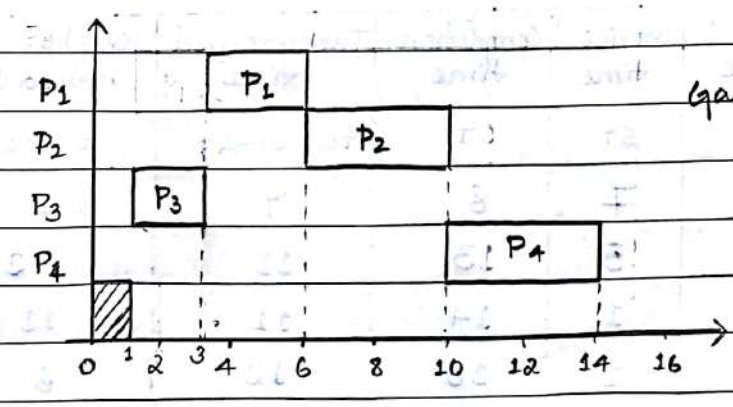


— Shortest Request Next (SRN) Scheduling:

The SRN scheduler always schedules the request with the smallest service time. Thus a request remains pending until all shorter requests have been serviced.

Q: Find the average turn around time and average weighted turnaround time by performing SRN algorithm. Draw the gant chart.

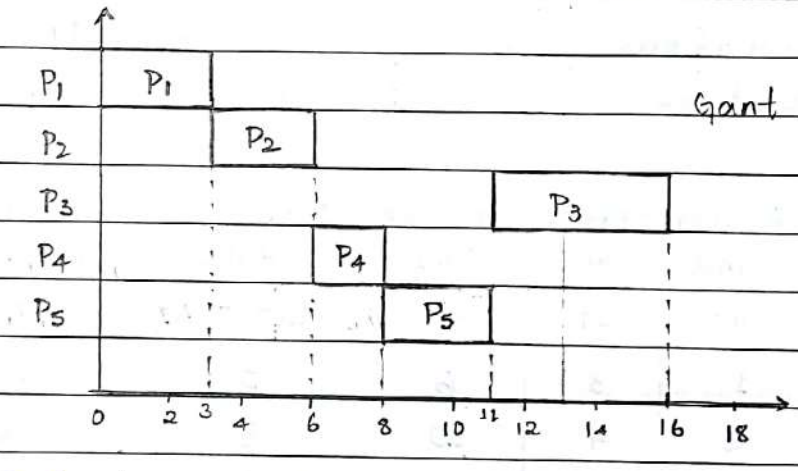
Process	Arrival time	Service time	completion time	Turnaround time	Weighted turn around time
	AT	ST	CT	ta = CT-AT	w = ta/ST
P_1	1	3	6	5	1.66
P_2	2	4	10	8	2
P_3	1	2	3	2	1
P_4	4	4	14	10	2.5
				$\overline{ta}$ = 6.25 s	$\overline{w}$ = 1.79 s



Q: Find the average turn around time and average weighted turnaround time by performing SRN algorithm. Draw the gant chart.

Process	Arrival time	service time	completion time	Turnaround time	Weighted turn around time
	AT	ST	CT	ta = CT-AT	w = ta/ST
P_1	0	3	3	3	1.
P_2	2	3	6	4	1.33
P_3	3	5	16	13	2.6
P_4	4	2	8	4	2
P_5	8	3	11	7	2.5

$\bar{ta}$ = 6.2 sec $\bar{w}$ = 1.886 sec



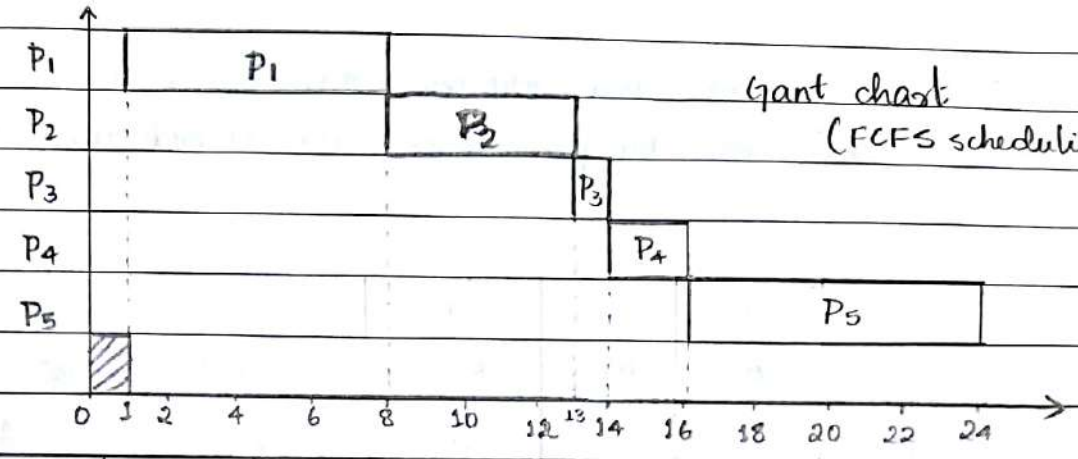
Gant Chart

Q: Find the average turn around time and average weighted turnaround time for both FCFS and SRN algorithm. Draw the gant chart.

FCFS:

Process	Arrival time	Service time	Completion time	Turnaround time	Weighted turn around time
	AT	ST	CT	ta = CT-AT	w = ta/ST
P_1	1	7	8	7	1
P_2	2	5	13	11	2.2
P_3	3	1	14	11	11
P_4	4	2	16	12	6
P_5	5	8	24	19	2.375

$\bar{ta}$ = 11 sec
$\bar{w}$ = 4.415 sec

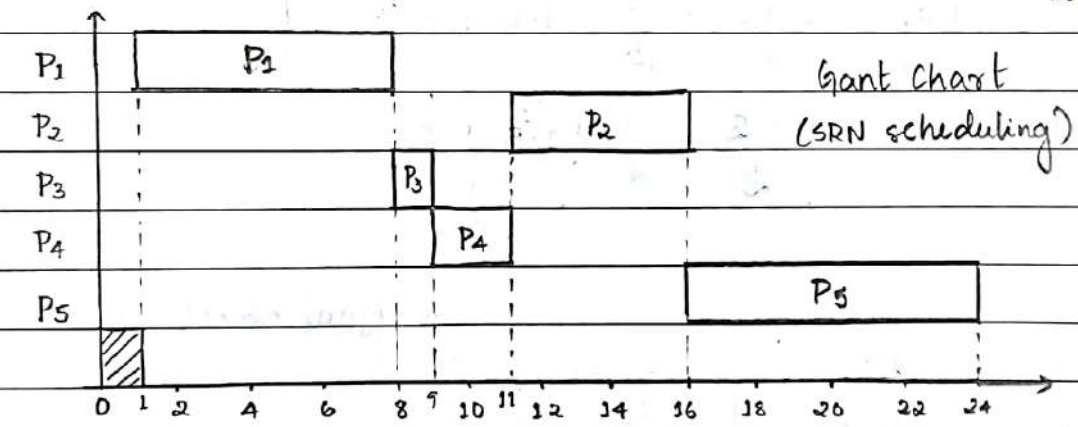


Gant chart (FCFS scheduling)

SRN:

Process	Arrival time	Service time	completion time	Turnaround time	Weighted turn around time
	AT	ST	CT	t_a = CT - AT	$w = t_a / ST$
P_1	1	7	8	7	1
P_2	2	5	16	14	2.8
P_3	3	1	9	6	6
P_4	4	2	11	7	3.5
P_5	5	8	24	19	2.375

$\bar{t_a} = 10.6$ sec $\bar{w} = 3.135$ sec //



Gant chart (SRN scheduling)

— Highest Response Ratio Next (HRRN) scheduling:

The highest Response Ratio policy computes the response ratios of all processes in the system and selects the process with the highest response ratio.

$$\text{Response Ratio} = \frac{\text{time since arrival + service time of the process}}{\text{service time of the process}}$$

Q: Find the average turn around time and average weighted turnaround time by performing HRRN scheduling.

Process	Arrival time	service time	completion time	Turnaround time	weighted turn around time
	AT	ST	CT	ta = CT - AT	w = ta/ST
P_1	0	3	3	3	1
P_2	2	6	9	7	1.166
P_3	4	4	13	9	2.25
P_4	6	5	18	12	2.4
P_5	8	2	20	12	6

$\overline{ta} = 8$ $\overline{w} = 2.1432$

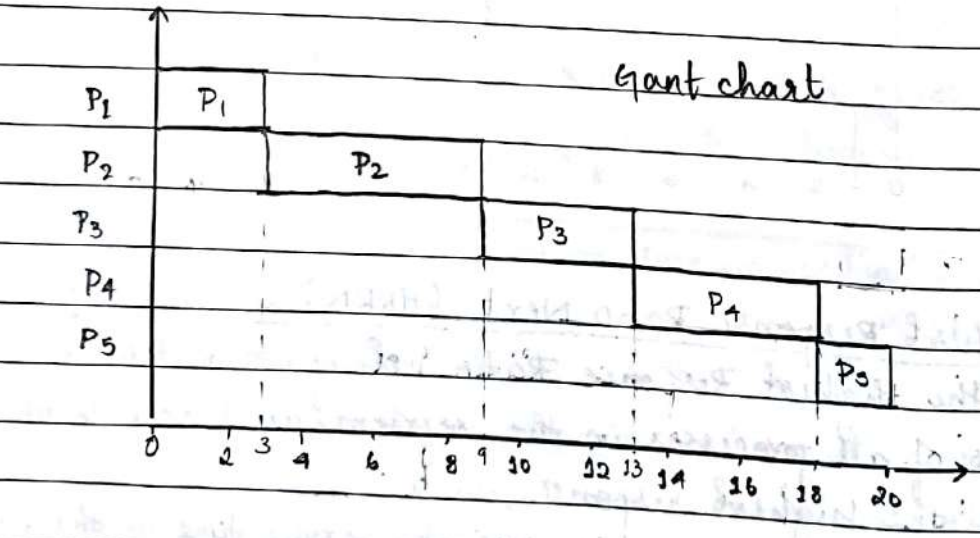
Response Ratio

$$RR = \frac{wait + ST}{ST}$$

$$RR_3 = \frac{5+4}{4} = \frac{9}{4} = 2.25 \quad (1)$$

$$RR_4 = \frac{3+5}{5} = \frac{8}{5} = 1.6 \quad (2)$$

$$RR_5 = \frac{1+2}{2} = \frac{3}{2} = 1.5 \quad (3)$$



Q: Find the average turn around time and average weighted turnaround time by performing HRRN scheduling. Also draw the gant chart.

Process	Arrival time	service time	Completion time	Turnaround time	Weighted turn around time
	AT	ST	CT	$ta = CT - AT$	$W = ta/ST$
P_1	0	3	3	3	1
P_2	2	3	6	4	1.33
P_3	3	5	13	10	2
P_4	4	2	8	4	2
P_5	8	3	16	8	2.66

$ta = 5.8$ sec $W = 1.798$ sec

Response ratio

$$RR = \frac{wait + ST}{ST}$$

I

$$RR_2 = \frac{1+3}{3} = \frac{4}{3} = 1.33 \quad (1)$$

$$RR_3 = \frac{0+5}{5} = 1$$

II

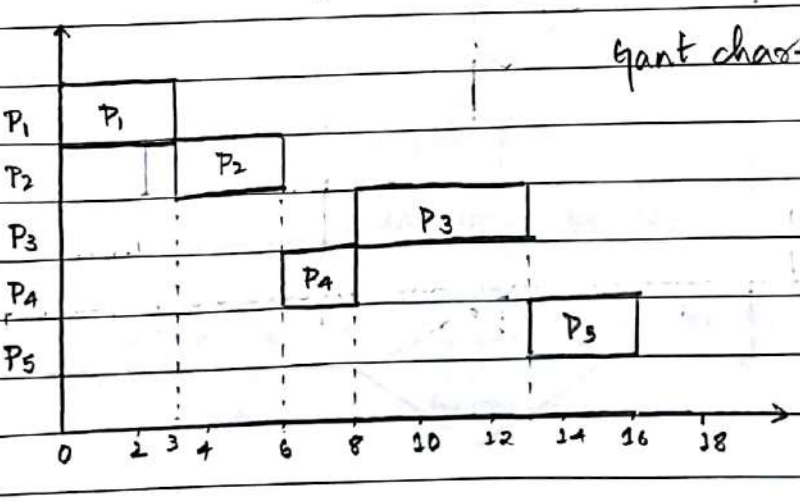
$$RR_3 = \frac{3+5}{5} = \frac{8}{5} = 1.6$$

$$RR_4 = \frac{2+2}{2} = \frac{4}{2} = 2 \quad (1)$$

III

$$RR_3 = \frac{5+5}{5} = \frac{10}{5} = 2 \quad (1)$$

$$RR_5 = \frac{0+3}{3} = 1$$



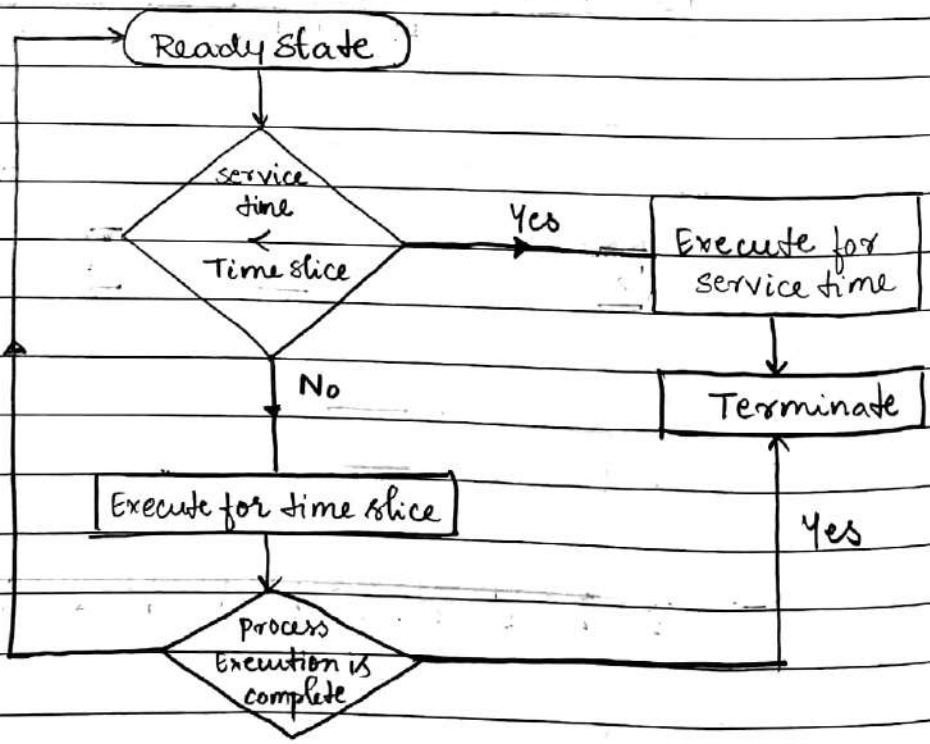
★ Preemptive Scheduling Policies:

In preemptive scheduling, the server can be switched to the processing of a new request before completion of the current request. The preempted request is put back into the list of pending requests. Its servicing is resumed when it is scheduled again. Thus a request might have to be scheduled many times before it is completed. There are three preemptive scheduling policies:

- Round-Robin scheduling with time slicing (RR)
- Least completed next (LCN) scheduling
- shortest time to go (STG) scheduling.

— Round-Robin Scheduling with time slicing (RR)

The RR scheduling policy shares the CPU among admitted requests by servicing them in turn. A request is preempted at the end of a time slice.



Date
Page

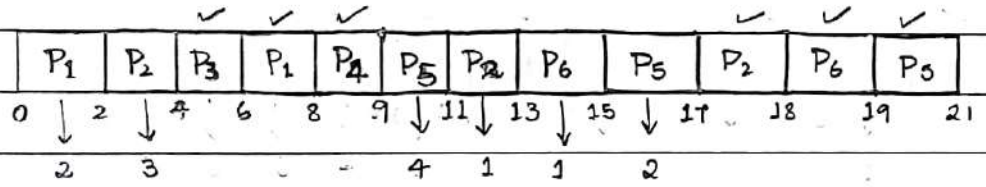
Q: Find the average turn around time and average weighted turnaround time by performing RR scheduling given a time slice of 2sec. Draw the gant chart.

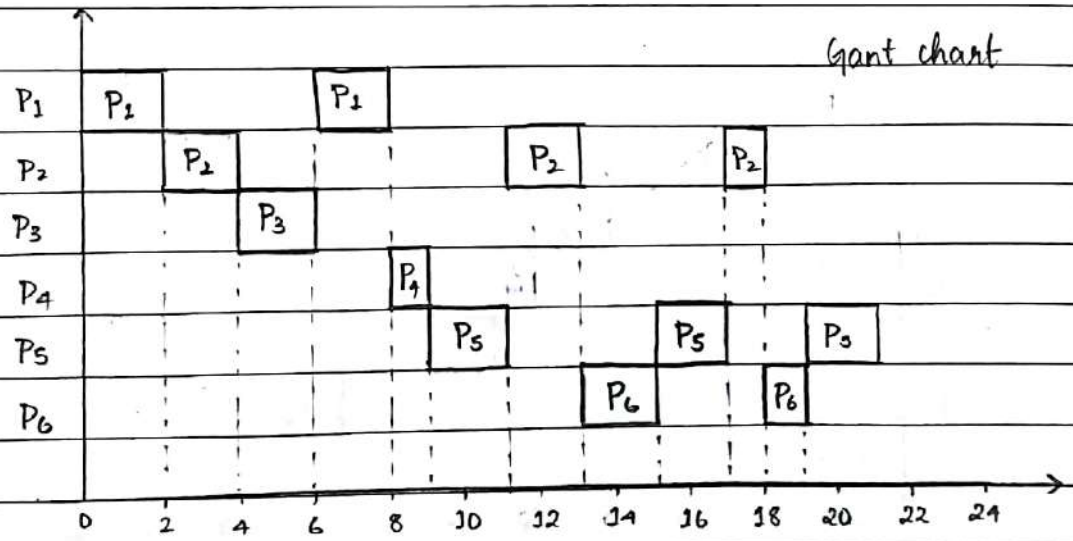
Process	Arrival time AT	Service time ST	Completion time CT	Turnaround time ta = CT-AT	Weighted turn around time W = ta/ST
P_1	0	4	8	8	2
P_2	1	5	18	17	3.4
P_3	2	2	6	4	2
P_4	3	1	9	6	6
P_5	4	6	21	17	2.83
P_6	6	3	19	13	4.33

$\overline{ta} = 10.83$ s $\overline{w} = 3.43$ s

Queue

P_1 P_2 P_3 P_1 P_4 P_5 P_2 P_6 P_5 P_2 P_6 P_5



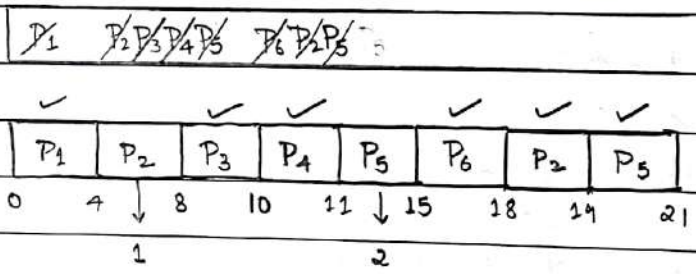


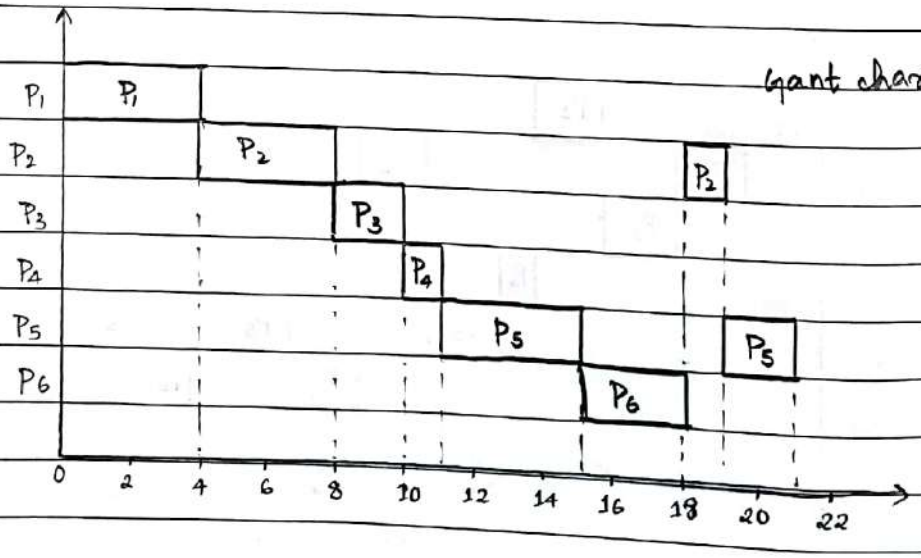
Q: Find the average turn around time and average weighted turnaround time by performing RR scheduling given a time slice of 4 sec. Draw the gant chart.

Process	Arrival time	Service time	Completion time	Turnaround time	Weighted turn around time
	AT	ST	CT	ta = CT - AT	w = ta/ST
P_1	0	4	4	4	1
P_2	1	5	19	18	3.6
P_3	2	2	10	8	4
P_4	3	1	11	8	8
P_5	4	6	21	17	2.83
P_6	6	3	18	12	4

$\overline{ta} = 11.16$ s $\overline{w} = 3.905$ s

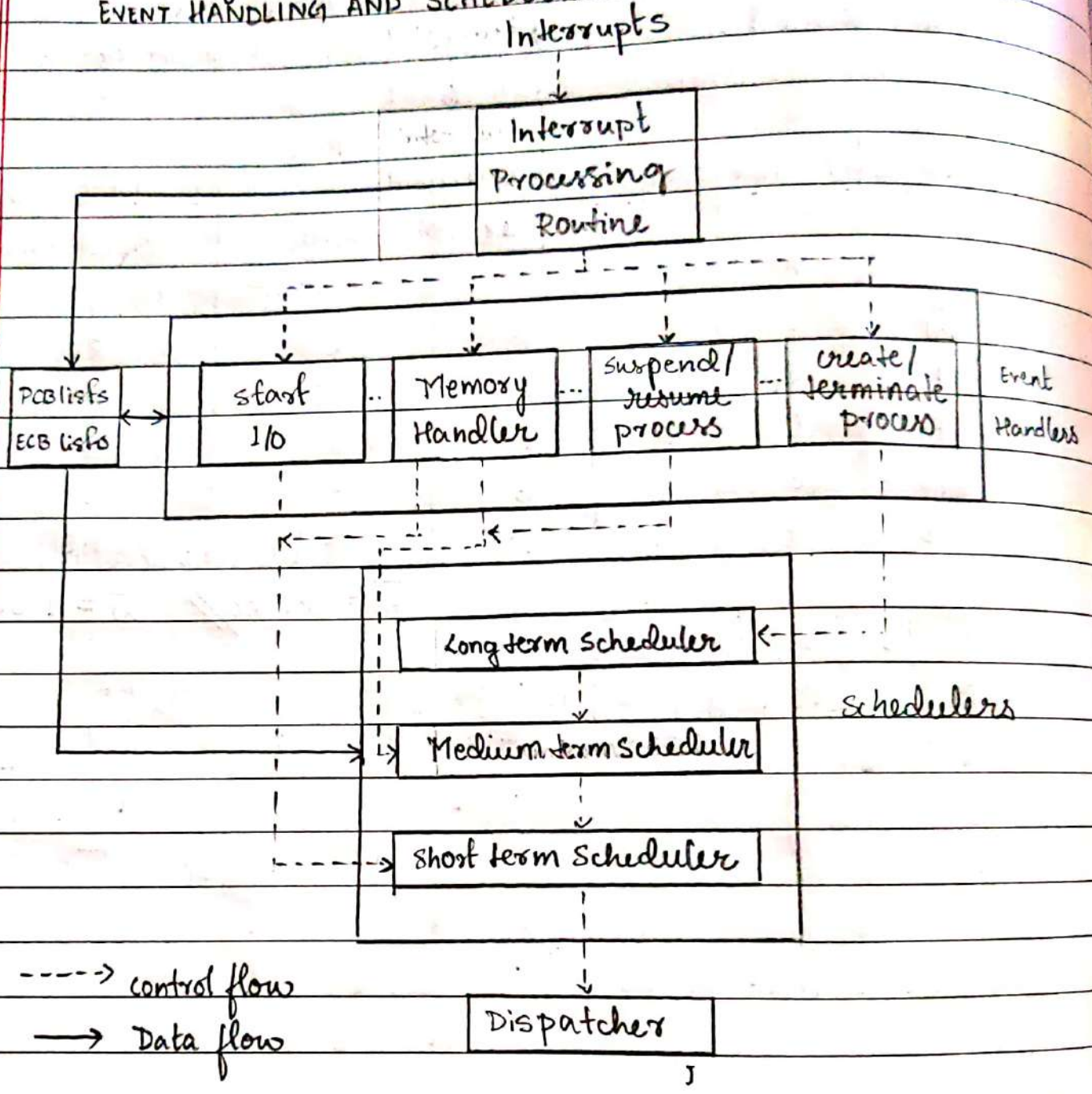
Queue





★ **Scheduling:**

EVENT HANDLING AND SCHEDULING



An operating system has to provide suitable combination of user service and system performance features. It also has to adapt its operation to the nature and number of user requests and availability of resources. A single scheduler using a single scheduling policy cannot address all these issues effectively. Therefore an operating system uses an arrangement consisting of three schedulers which are as follows:

- Long term scheduler:

It decides when to admit an arrived process for scheduling depending on its nature (whether CPU bound or I/O-bound) and on availability of resources like kernel data structures and disk space for swapping.

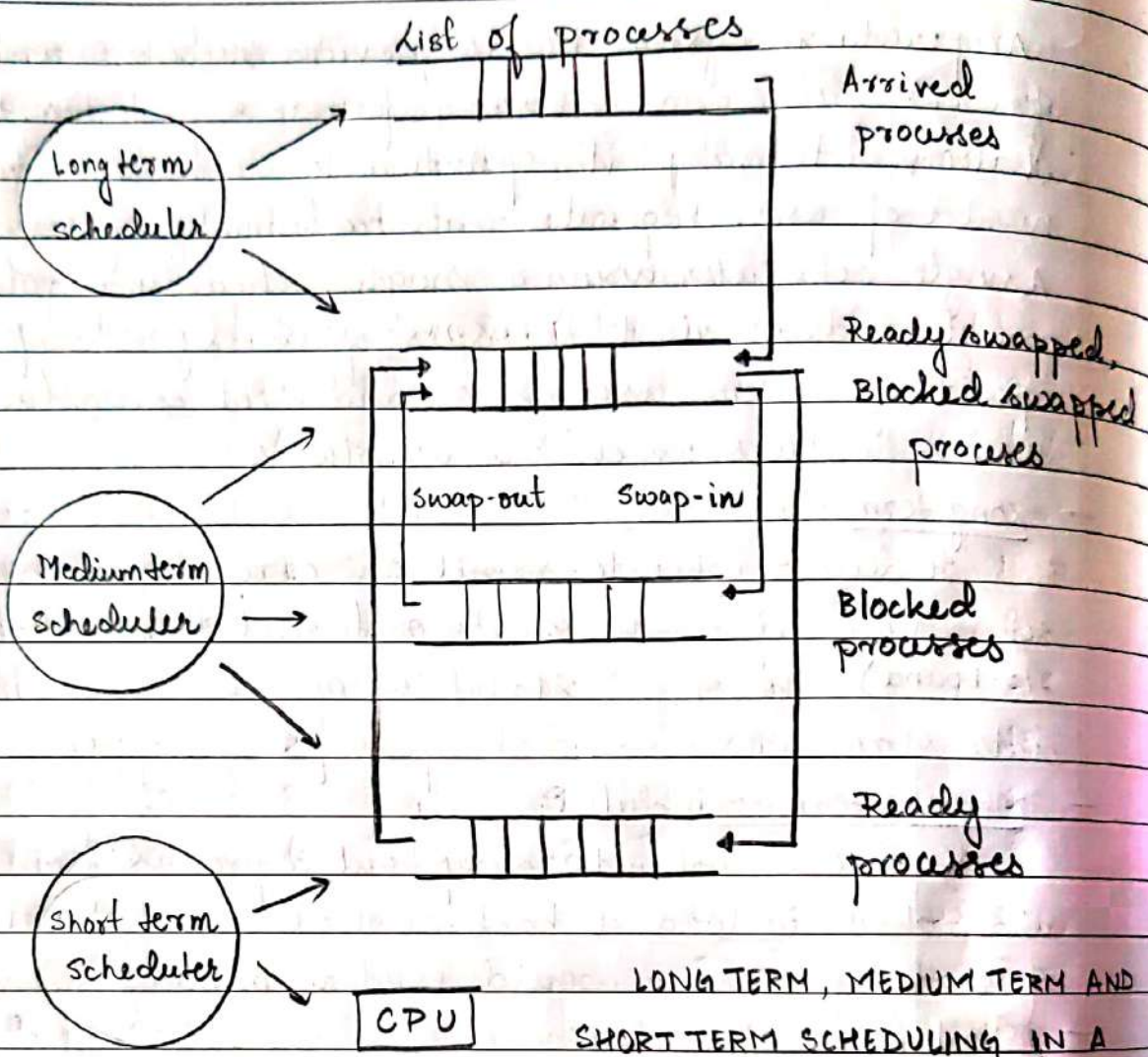
- Medium term scheduler:

It decides when to swap-out a process from memory and when to load it back so that a sufficient number of ready processes would exist in memory.

- Short term scheduler:

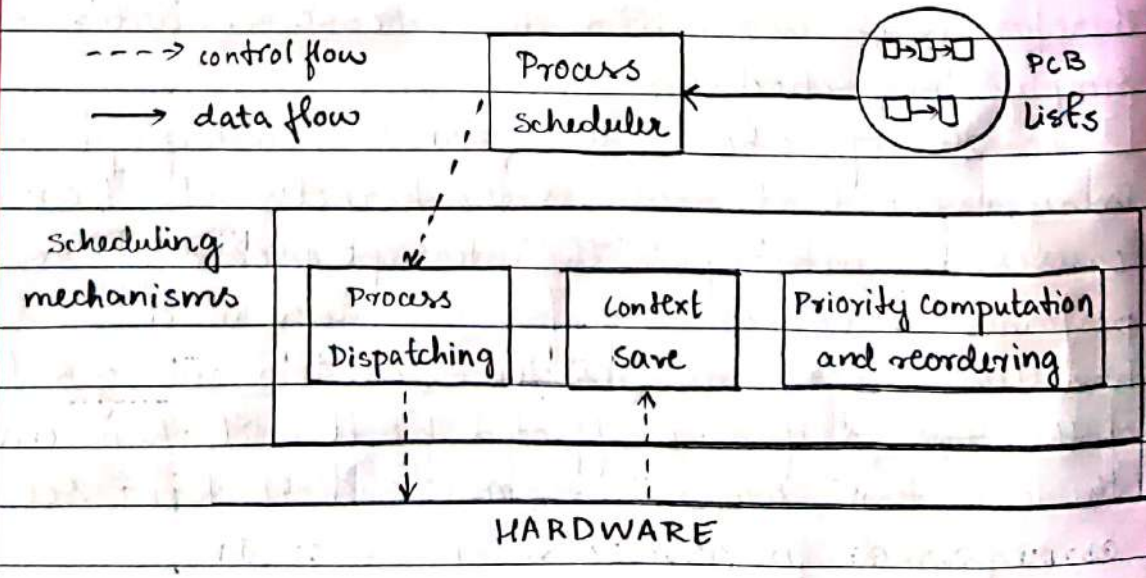
It decides which ready process to service next on the CPU and for how long. Hence short term scheduler is the one that actually selects a process for operation. Therefore it is also called the process scheduler or simply the scheduler.

The operation of the kernel is interrupt driven. Every event that requires the kernel's attention causes an interrupt. The interrupt processing routine performs a context save function and invokes an event handler which analyses the event and changes the state of the process if any affected by it. It then invokes the long-term, medium-term or short-term scheduler as required.



LONG TERM, MEDIUM TERM AND SHORT TERM SCHEDULING IN A TIME SHARING - SYSTEM

* Process Scheduler:

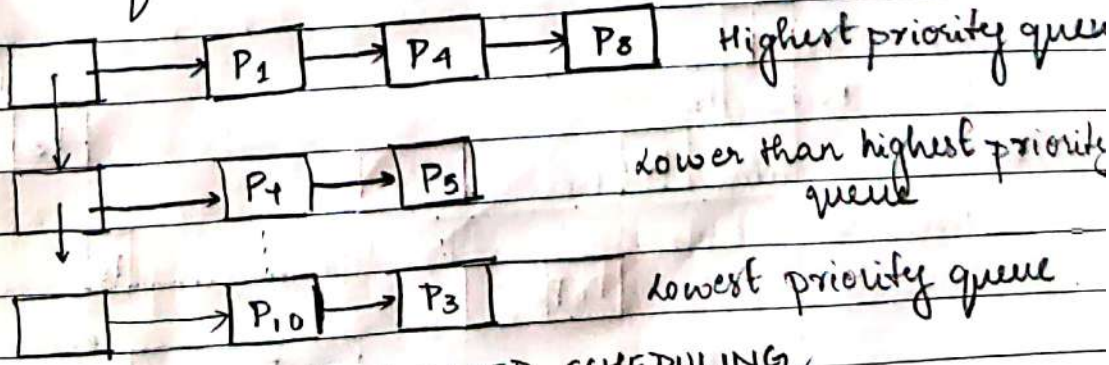


A process scheduler uses several lists of PCBs and selects one process and passes its id to the process dispatching mechanism which loads contents of PCB fields (PSW and GPRs) into the CPU to resume operation of the selected process. Thus the dispatching mechanism interfaces with the scheduler on one side and the hardware on the other side.

The context save mechanism is a part of the interrupt processing routine. When an interrupt occurs, it is invoked to save the PSW and GPRs of the interrupted process. The priority computation and reordering mechanism recomputes the priority of requests and reorders the PCB lists to reflect the new priorities based on the scheduling policy in use.

* **Priority based scheduling:**

In simple priority based scheduling a single list of PCB is maintained in the system. PCB in the list are organised in the order of decreasing priority. The PCB of newly created process is entered in the list in accordance with its priority when a process terminates its PCB is removed from the list. The scheduler scans the PCB list from the beginning and scheduler selects the first ready process that it finds.



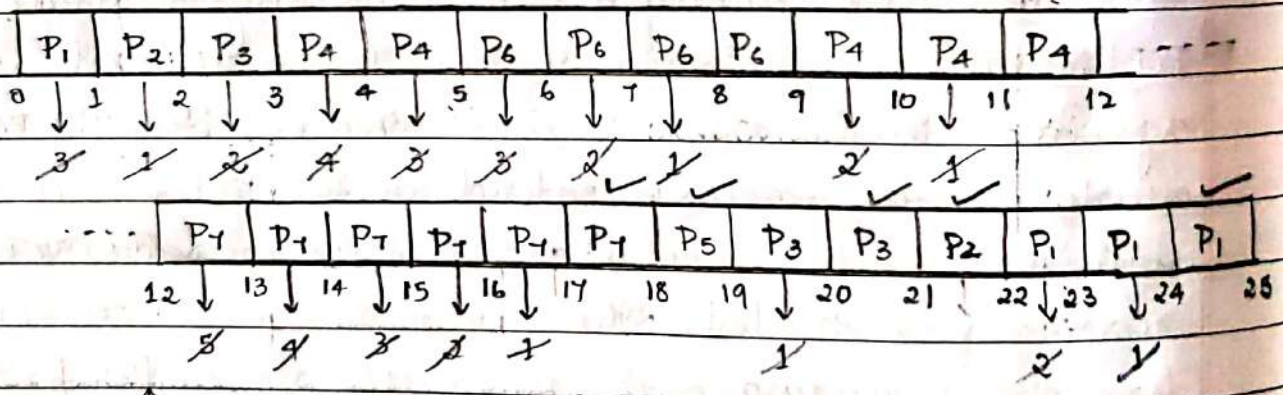
READY QUEUES IN PRIORITY BASED SCHEDULING

Q: Find the average turn around time and average weighted turn around time by performing priority based scheduling for both preemptive and nonpreemptive scheduling. Draw the gant chart.

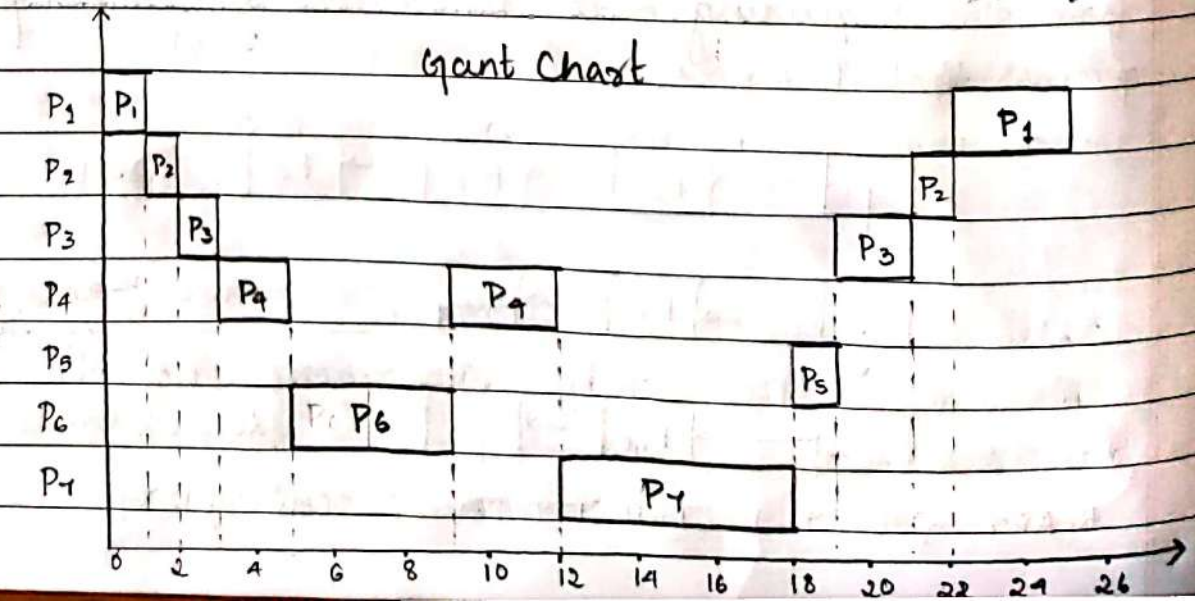
preemptive

Process	Priority	Arrival time AT	Service time ST	Completion time CT	Turnaround time ta = CT-AT	Weighted turn around time w = ta/ST
P_1	2	0	4	25	25	6.25
P_2	4	1	2	22	21	10.5
P_3	6	2	3	21	19	6.33
P_4	10	3	5	12	9	1.8
P_5	8	4	1	19	15	15
P_6	12	5	4	9	4	1
P_7	9	6	6	18	12	2

$\bar{T}a = 15$ sec $\bar{w} = 6.25$ sec

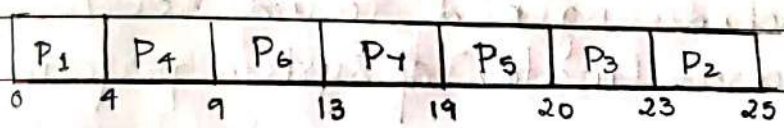


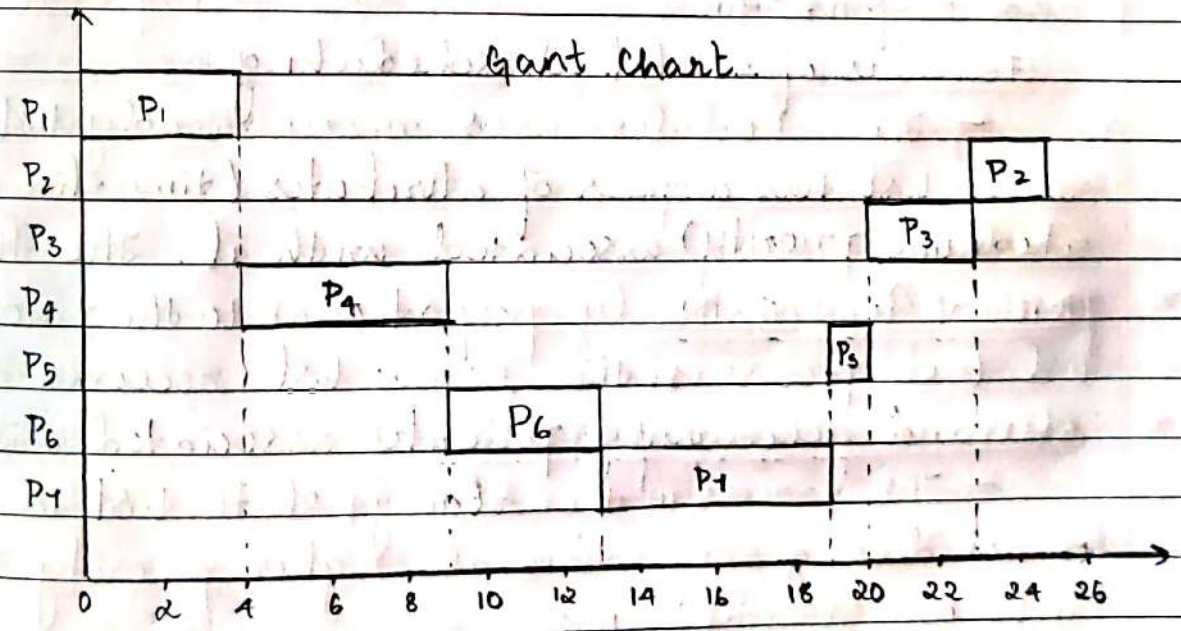
Gant Chart



Process	Priority	Arrival time AT	Service time ST	Completion time CT	Turnaround time ta = CT - AT	Weighted turn around time w = ta/ST
P_1	2	0	4	4	4	1
P_2	4	1	2	25	24	12
P_3	6	2	3	23	21	7
P_4	10	3	5	9	6	1.2
P_5	8	4	1	20	16	16
P_6	12	5	4	13	8	2
P_7	9	6	6	19	13	2.167

$\overline{ta} = 13.14$ sec $\overline{w} = 5.909$ sec





* **Round Robin scheduling with time slice:**

In round robin scheduling the scheduler maintains queue of ready process and a list of blocked process. The PCB of newly created process is added to the end of the ready queue. The scheduler always selects the PCB at the head of the ready queue. When a running process finishes its time slice it is moved to the end of the ready

queue. The event handler performs two actions:
- when a process makes I/O request or it is swapped out, its PCB is removed from the ready queue to is moved to blocked or swapped out list.
- when an I/O operation awaited by a process finishes or the process is swapped in, its PCB is removed from blocked or swapped out list and is moved to the end of the ready queue

* Multilevel Scheduling:

The multilevel scheduling policy combines priority based scheduling and round robin scheduling to provide a good combination of system performance and response times.

Features of multilevel scheduling:

- The scheduler uses many ready list. Each ready list has a pair of attributes (time slice and scheduling priority) associated with it. The time slice for the list is inversely proportional to the priority of the list. Each process in the ready list receives the time slice and scheduling priority associated with the list.
- The process in the ready state list is considered for scheduling only when all higher priority ready lists are empty.
- Round robin scheduling is performed within each list.

* Real Time Scheduling:

Real-time application often impose some special scheduling constraints that is precedence, periodicity and resource constraints in addition to familiar

need to meet deadlines.

- Static Scheduling:

They do not change with time. A schedule is prepared before execution of of the application begins.

- Dynamic scheduling:

Scheduling is performed when a request to create a process is executed. Process creation succeeds only if response requirement of the process can be satisfied in a guaranteed manner.

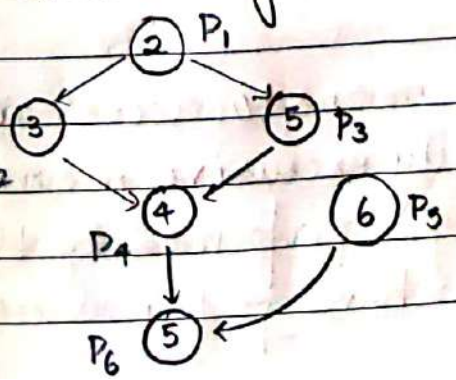
- Priority based scheduling:

Real time application is analized to assign appropriate priorities to process in it. conventional priority based scheduling is used during operation of the application.

Process Precedences

Processes of real time application interact among themselves to ensure that they perform their acts in a desired order. Thus the Process Precedence Graph (PPG) is used to depict dependences between the processes.

Process P_i is said to precede process P_j if execution of P_i must be completed before P_j can begin its execution. The notation $P_i \rightarrow P_j$ shall indicate that process P_i directly precedes process P_j. The precedence relation is transitive i.e., $P_i \rightarrow P_j$ and $P_j \rightarrow P_k$ implies that P_i precedes P_k. The notation $P_i \xrightarrow{*} P_k$ is used indicate that process P_i directly or indirectly precedes P_k.



THE PROCESS PRELEDENCE GRAPH (PPG) FOR FOR A REAL-TIME SYSTEM

- Deadline Scheduling:

Two kinds of deadlines can be specified for a process:

a. Starting deadline: the latest time by which operation of the process must begin.

b. Completion deadline: the time by which operation of the process must complete.

Deadline Estimation

$$D_i = D_{application} - \sum_{k \in descedant(i)} x_k$$

where D_i: deadline of a process

$D_{application}$: deadline of the application

x_k: service time of process

descedant(i): set of descendent of P_i in PPG

Earliest Deadline First (EDF) Scheduling:

This policy always selects the process with the earliest deadline.

★ MESSAGE PASSING:

★ Overview of message passing:

The four ways in which processes interact with one another are:

- data sharing
- message passing
- synchronization
- signals.

Data sharing provides means to access values of shared data in a mutually exclusive manner.

Process synchronization is performed by blocking a process until other processes have performed certain specific

actions. Capabilities of message passing overlap those of data sharing and synchronization. However each form of process interaction has its own application area.

Process P_i

```
send (Pj, <message>);
```

Process P_j

```
receive (Pi, msg-area);
```

message passing

Ex:

Process P_i sends a message to process P_j by executing the statement send (P_j, <message>). The compiled code of the send statement invokes the library module send. send makes a system call send, with P_j and the message as parameters. Execution of the statement receive (P_i, <msg-area), where msg-area is an area in P_j's address space, results in a system call receive.

Applications of message passing

- Message passing is employed in the client-server paradigm, which is used to communicate between the server and user processes.
- Message passing is used as the backbone of higher level protocols employed for communicating between computers or for providing the electronic mail facility.
- Message passing is used to implement communication between tasks in a parallel or distributed program.

Mailbox:

A mailbox is a repository for interprocess messages. It has a unique name. The owner of a mailbox is typically the process that created it. Only the owner process can receive messages from a mailbox. Any process that knows the name of a mailbox can send messages to it. Thus, sender and receiver processes use the name of the mailbox rather than each other's names in send and

receive statements.

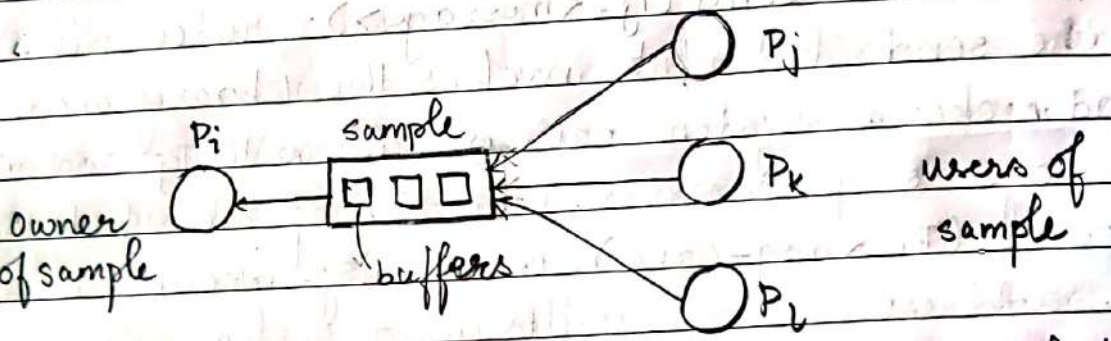
Ex: Creation and use of mailbox 'sample'

Process P_i

```
create_mailbox (sample);
...
receive (sample, '....');
```

Process P_j

```
send (sample, '....');
...
```



Process P_i creates the mailbox using the statement create_mailbox. Process P_j sends a message to the mailbox using the mailbox name in its send statement. If P_i has not already executed a receive statement, the kernel would store the message in a buffer. The kernel may associate a fixed set of buffers with each mailbox or it may allocate buffers from a common pool of buffers when a message is sent.

The kernel may provide a fixed set of mailbox names or it may permit user processes to assign mailbox names of their choice.

The kernel may require a process to explicitly connect to a mailbox before starting to use it and to disconnect when it finishes using it. This way it can destroy a mailbox if no process is connected to it or it may permit the owner of a mailbox to destroy it.