Unit - II ~~Concurrent Processes~~

CPU Scheduling

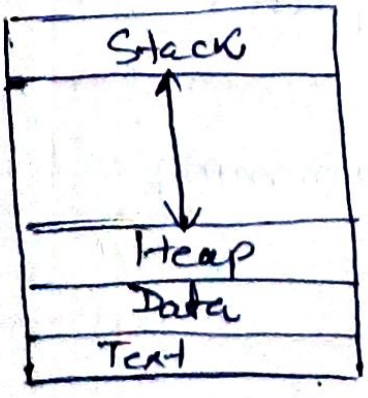
Process → "A Process is basically a Program in execution."

→ The execution of Process must Progress in a sequential fashion.

→ "A Process is defined as an entity which represents the basic unit of work to be implemented in the system."

⇒* To understand it in simple terms, we write our computer Programs in a text file and when we execute this Program, it becomes a Process which Performs all the tasks mentioned in Program.

*→ When a Program is loaded into the memory and it becomes a Process, it can be divided into four sections — stack, heap, text and data.



Stack- The Process Stack Contains the temporary data such as method/function Parameters, return address, and local variables.

Heap- This is a dynamically allocated memory to a Process during its runtime.

Text - This includes the current activity represented by the value of Program Counter and the Content of Processor registers.

Data - This section Contains the global and static variables

Program → A Program is a Piece of Code which may be a single line or millions of lines. A Computer Program is usually written by a Computer Programmer in a Programming language.

ex-

```
#include <stdio.h>
int main(){
printf("HelloWorld");
```

return 0;

}

→ A Computer Program is a collection of instructions that Perform a specific task when executed by a computer.

→ When we compare a Program with a Process, we can conclude that a Process is a dynamic instance of a computer Program

→ A Part of a computer Program that Performs a well-defined task is known as an algorithm.

→ A Collection of Computer Programs, Libraries and related data are referred to as a Software.

Process Life Cycle → When a Process executes, it Passes through different states. These stages may differ in different operating Systems, and the names of these states are also not Standardized.

In general, a Process can have one of the following five states at a time

1- Start — This is the initial state when a Process is first started/created.

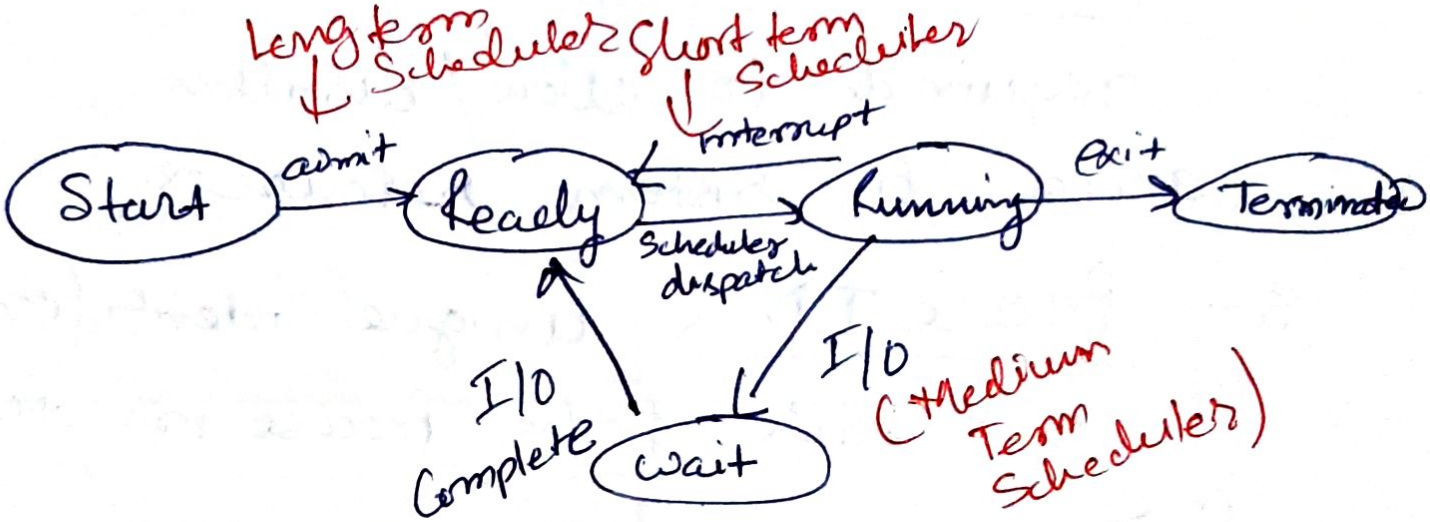
2- Ready — The Process is waiting to be assigned to a Processor.

Ready Processes are waiting to have the Processor allocated to them by O.S so that they can run. Process may come into this state after start state or while running it by but interrupted by scheduler to assign CPU to some other Process

3- Running — Once the Process has been assigned to a Processor by the OS scheduler, the Process state is set to running and Processor executes its instructions.

4- Waiting — Process moves into waiting state if it needs to wait for a resource, such as waiting for user input or waiting for a file to become available

5- Terminated or Exit - Once the process finishes its execution, or it is terminated by the O.S, it is moved to the terminated state where it waits to be removed from main memory.



Process Control Block (PCB) -

A Process control block is a data structure maintained by the O.S for every Process.

The PCB is identified by an integer Process ID (PID). A PCB keeps all the information needed to keep track of a process as listed below in the table;

1- Process State → the current state of the Process, i.e, whether it is ready, running, waiting or whatever.

2- Process Privileges → This is required to allow/disallow access to system resources.

3- Process ID → Unique identification for each of the Process in the O.S

4- Pointer — A Pointer to Parent Process.

5- Program Counter → PC is a Pointer to the address of the next instruction to be executed for this Process.

6- CPU Registers → Various CPU register where Process need to be stored for execution for running state.

7- CPU Scheduling Information →
Process priority and other scheduling information which is required to schedule the process

8- Memory management Information →
This includes the information of Page table, memory limits, Segment table depending on memory used by O.S

9- Accounting Information - This includes the amount of CPU used for Process execution, time limits, execution ID etc.

10- I O Status information →
This includes a list of I/O devices allocated to the Process

NB- * The architecture of a PCB is completely dependent on O.S and may contain different information in different O.S.

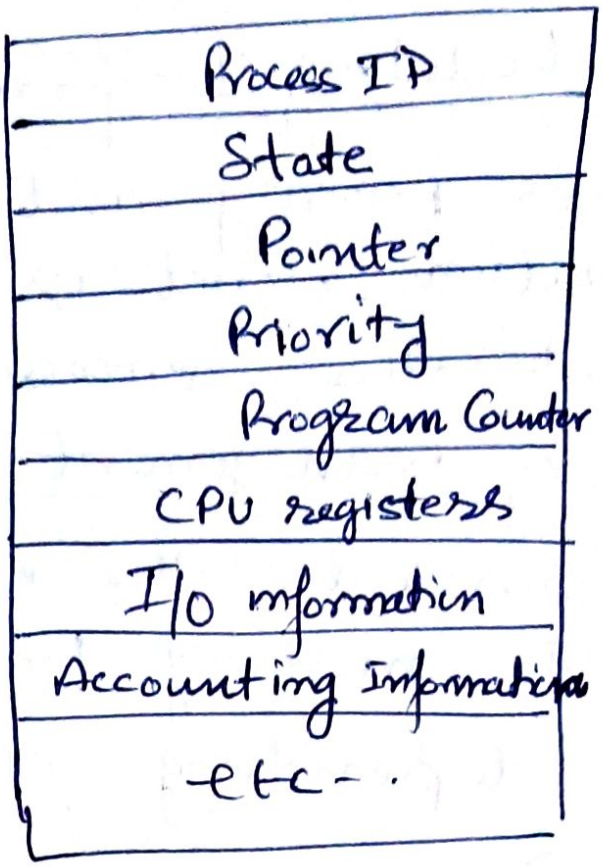


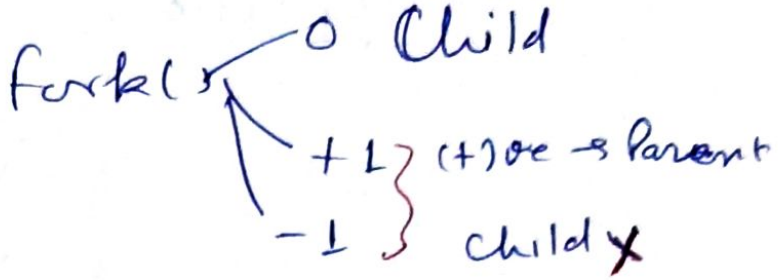
Fig- Simplified diagram of a PCB

* NB - The PCB is maintained for a Process throughout lifetime, and is deleted once the Process terminates.

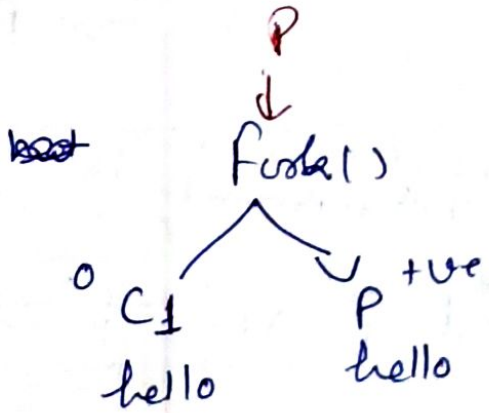
Fork() — System call —

To create a child Process

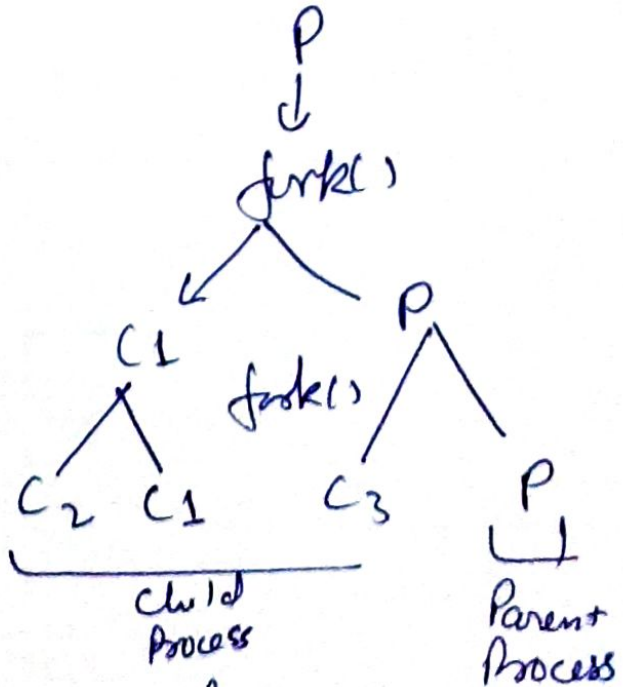
it returns




```
main() {
fork();
Printf ("hello");
}
```



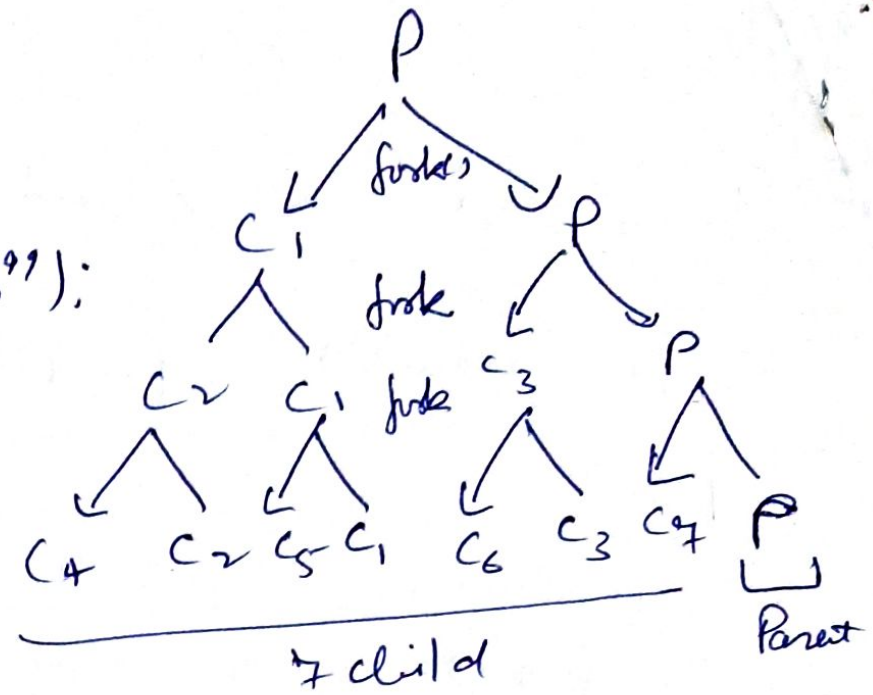
```
main() {
fork();
fork();
Printf("hello");
}
```



4 times hello will be Printed

ex

```
fork();
fork();
fork();
pf("hello");
```



8 times hellow will be Printed

So, $\Rightarrow 2^n$ (no of times Hello is Printed) (n — is the no of times fork is written)

$\Rightarrow 2^n - 1$ (no of child Process generated)

Process Scheduling

⇒ The Process scheduling is the activity of the Process manager that handles the removal of the running Process from the CPU and the selection of another Process on the basis of a Particular strategy.

⇒ Process scheduling is an essential Part of a multiprogramming O.S. Such O.S allow more than one Process to be loaded into the executable memory at a time.

Process Scheduling Queues ⇒ The O.S maintains all PCBs in Process Scheduling Queues. The O.S maintains a separate queue for each of the Process states and PCBs of all Process in the same execution state are Placed in the same queue.

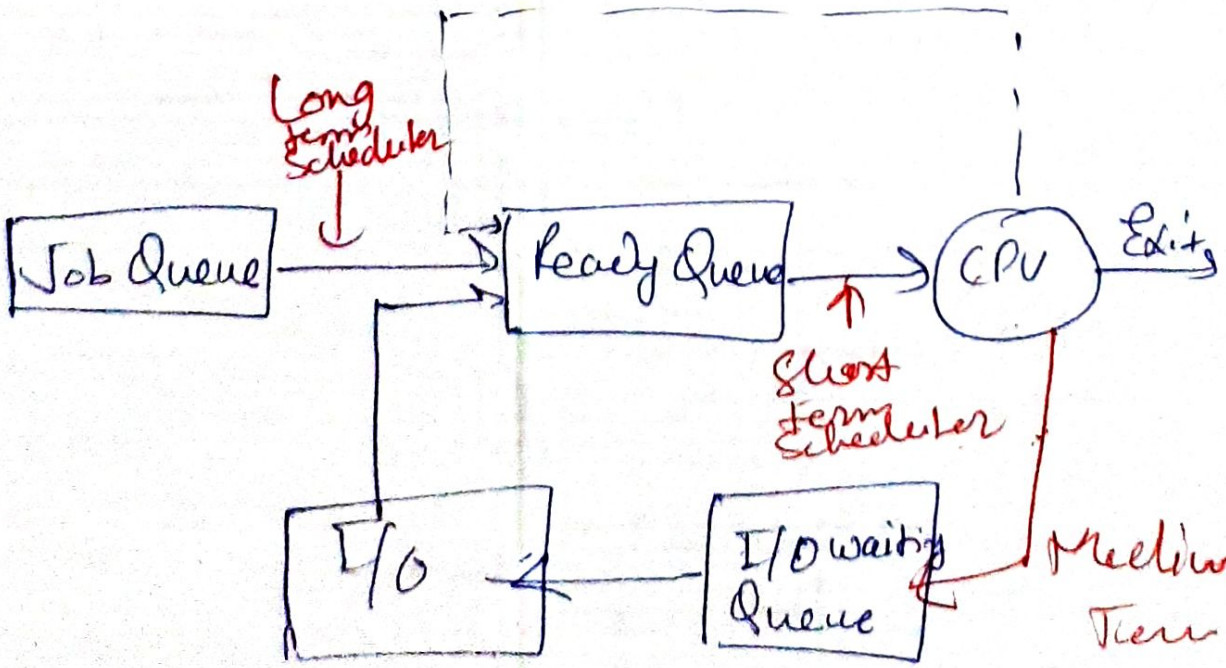
→ When the state of a Process is changed its PCB is unlinked from its current queue and moved to its new state queue.

The O.S maintains the following important Process scheduling queues:

→ Job Queue → This queue keeps all Processes in the system.

→ Ready Queue → This queue keeps a set of all Processes residing in main memory, ready and waiting to execute. A new Process is always put in this queue.

→ Device Queues → The Process which are blocked due to unavailability of an I/O device constitute this queue.



→ The OS can use different Policies to manage each queue (FIFO, Round Robin, Priority etc).

→ The O.S Scheduler determines how to move Processes between the ready and run queues which can only have one entry Per Processor Core on the system; in above diagram, it has been merged with CPU.

Two-State Process Model →

Two-state Process model refers to running and non running states which are described below.

Running → when a new Process is created, it enters into the System as in the running states.

Not Running → Processes that are not running are kept in queue, waiting for their turn to execute. Each entry in the queue is a Pointer to

- a Particular Process Queue is implemented by using linked list.

→ Use of dispatcher is as follows.

— when a Process is interrupted, that Process is transferred in the waiting queue. If the Process has completed or aborted, the Process is discarded.

→ In either case, the dispatcher then selects a Process from the queue to execute.

Schedulers → Schedulers are special system software which handle Process scheduling in various ways.

→ Their main task is to select the job to be submitted into the system and to decide which Process to run.

Schedulers are of three types:

→ Long – Term Scheduler

→ Short – Term Scheduler

→ Medium – Term Scheduler

Long-Term Scheduler :- It is also called a job scheduler.

→ A long-term scheduler determines which programs are admitted to the system for processing.

→ It selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU Scheduling

→ The primary objective of the job scheduler is to provide a balanced mix of job, such as I/O bound and Processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

→ On some system, the long-term scheduler may not be available or minimal.

→ Time sharing O.S have no long term scheduler.

→ When a Process changes the state from new to ready, then there is use of long-term scheduler.

Short-Term Scheduler :- It is also called CPU scheduler. Its main objective is to increase system performance in accordance with the chosen set of criteria.

→ It is the change of ready state to running state of the process.

→ CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them.

→ Short-term schedulers, also known as as dispatchers, make the decision of which process to execute next.

→ Short-term schedulers are faster than long-term schedulers.

Medium-Term Scheduler :- Medium-Term scheduling is a part of swapping.

→ It removes the processes from the memory.

→ It reduces the degree of multiprogramming.

→ The medium-term scheduler is in charge of handling the swapped-out Processes

→ A running Process may become suspended if it makes an I/O request. A suspended Processes cannot make any Progress towards Completion. In this condition, to remove the Process from memory and make space for other Processes, the suspended Process is moved to the secondary storage. This Process is called Swapping, and the Process is said to be swapped out or rolled out.

→ Swapping may be necessary to improve the Process mix

Comparision Among Schedulers :-

Long-Term Scheduler	Short term Scheduler	Medium Term Scheduler
1- It is a job scheduler	It is a CPU Scheduler	It is a Process Swapping scheduler
2- Speed is lesser than Short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler

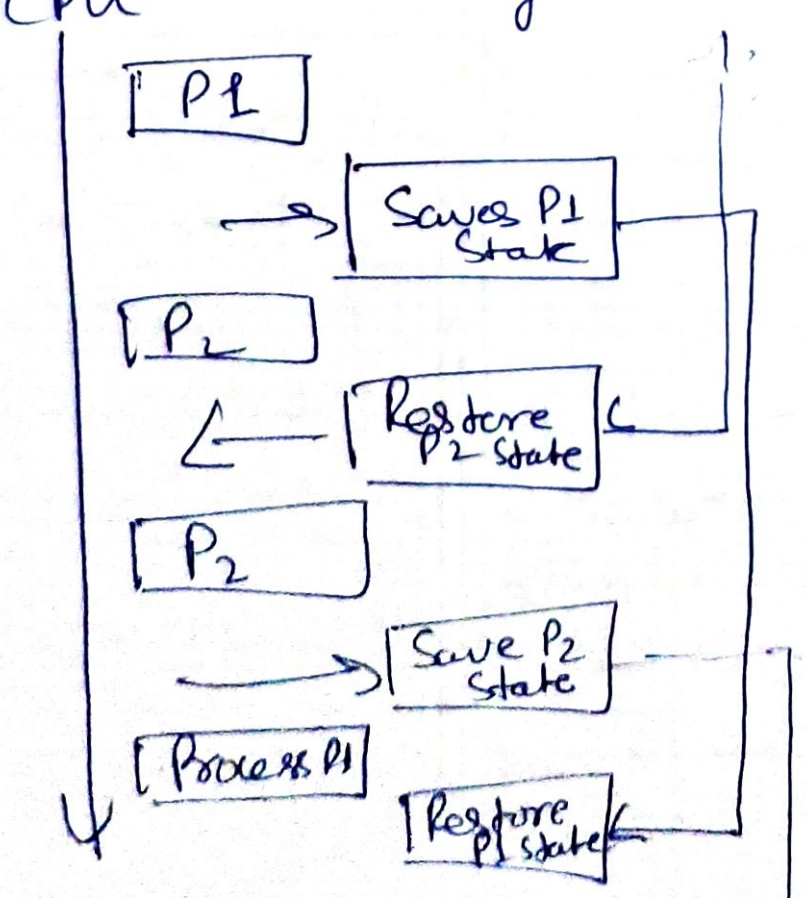
3- It Controls the degree of Multiprogramming	It Provides lesser Control over degree of Multiprogramming	It reduces the degree of Multi Programming
4- It is almost absent or minimal in time Sharing System	It is also minimal in time Sharing System	It is a Part of Time Sharing Systems
5- It selects Processes from Pool and loads them into memory for execution	It selects those Processes which are ready to execute	It can re-introduce the Process into memory and execution Can be Continued

Context Switch :- A Context Switch is the mechanism to store and restore the state or context of a CPU in PCB so that a Process execution can be resumed from the same Point at a later time.

- Using this technique, a Context Switcher enables multiple Processes to Share a Single CPU.

→ Context switching is an essential part of a multitasking O.S features.

→ When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the PCB. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.



Context switches are computationally intensive since register and memory state must be saved and restored.

→ To avoid the amount of context switching time, some hardware system employ two or more sets of processor registers.

→ When a process is switched the following information is stored for later use.

- → Program Counter
- → Scheduling information
- → Base and limit register value
- → currently used register
- → changed state
- → I/O state information
- → Accounting information.

Scheduling Algo.

→ A Process Scheduler schedules different processes to be assigned to CPU based on particular scheduling algorithms.

Scheduling Algs

Preemptive Alg

* SRTF (Shortest remaining time first)
- LRTF (Longest Remaining time first)
* Round Robin
- Priority Based

Non Preemptive Algs

- FCFS (First Come first serve)
- SJF (Shortest Job first)
- Longest Job First (LJF)
- Highest Response Ratio Next
- Multilevel Queue
- Priority

Multilevel feedback Queue

=> Non-Preemptive algorithms are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time, whereas Preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state

Different Times in CPU Scheduling

① Arrival time (Point of time) → The time at which Process enter the ready queue or state.

② Burst Time (Duration) → Time required by a Process to get execute on CPU

③ Completion Time (Point of time) → The time at which Process Complete its execution

④ Turn Around time →

$$TAT = \{Completion\ time - Arrival\ time\}$$

Defined as interval between the time of Submission and Completion of the Job.

⑤ Waiting Time →

$$W.T = TAT - Burst\ time$$

It is the sum of time intervals for which the Process has to wait in the ready Queue.

⑥ Response Time (Point of time)

$$RT = [\text{The time at which a Process get CPU first time} - \text{Arrival time}]$$

It is defined as the time interval between the Job Submission and the first response produced by the Job

Throughput :- Throughput is defined as the number of Processes that are completed per unit of time.

Types of Process :-

① I/O Bound → If it Performs lots of I/O operations as compare to other computation operations. Each I/O operation is followed by a short CPU burst to Process I/O.

② CPU Bound → If it Performs lots of Computation and has very less I/O operations to do. There will be long CPU Burst with I/O bursts of very short duration

---

FCFS (First - Come First - Served Scheduling Algorithm :-

"Allocate the CPU in the order in which the Processes arrive."

→ It assumes that ready queue is managed as FIFO (first in first out).

→ When the CPU is free, it allocated to the process, which is occupying the front of the queue.

→ Once this process goes into the running state, its PCB is removed from the queue.

→ This algorithm is non-Preemptive

Advantage :- ① It is simplest to understand and code

② Suitable for Batch Systems

Disadvantages :- ① Waiting time can be large if short requests wait behind the long once

② It is not suitable for time sharing systems where it is important that each users should get the CPU for an equal amount of time interval.

③ A proper mix of jobs (I/O based and CPU based jobs) is needed to achieve good results from FCFS scheduling

Eg

Process No	Arrival time	Burst Time	Completion Time	TAT	WT	RT
P_1	0	2	2	2	0	0
P_2	1	2	4	3	1	1
P_3	5	3	8	3	0	0
P_4	6	4	12	6	2	2

(CT − AT) (TAT − BT)

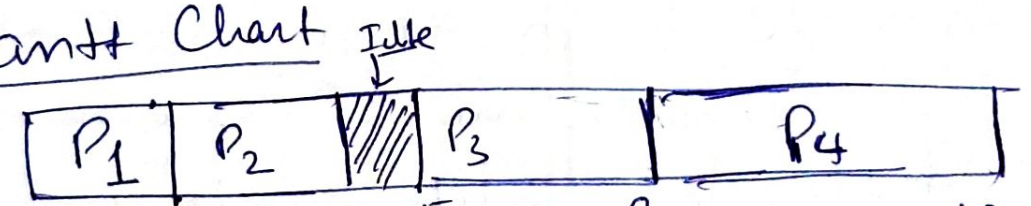
Criteria – (Arrival Time)

Mode – Non-Preemptive

Avg TAT = $\frac{14}{4} = 3.5$

Avg WT time = $\frac{3}{4} = 0.75$

Grantt Chart



Q–

Process	Burst Time
P_1	10
P_2	5
P_3	5

If orders is P_1, P_2, P_3

Avg WT time?

$= \frac{0+10+15}{3}$

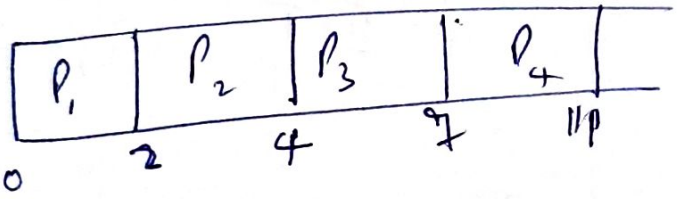
$= \frac{25}{3}$

$= 8.33$ Ans

Process No	Arrival Time	Burst Time	Completion Time	TAT	WT
P_1	0	2	2	2	0
P_2	1	2	4	3	1
P_3	2	3	7	5	2
P_4	3	4	11	8	4

Avg_WT = 1.75

Avg_TAT = 4.5



BT = Time required by a process to get execute on CPU

TAT = CT − AT (Defined as interval between the time of Submission and Completion of Job)

WT = TAT − B.T

{ It is the sum of time intervals for which the process has to wait in the ready Queue

---

Response time → [The time at which a Process get CPU first — AT]

It is defined as the time interval between the job submission and the first response produced by the job

①

P#	AT	BT
A	0	3
B	1	6
C	4	4
D	6	2

FCFS
A B C D
15
SJF
A B D C
15
SRTF
A B C D B
15

②

P#	AT	BT
A	0	4
B	2	7
C	3	2
D	3	2

FCFS
A B C D
15
SJF
A C D B
SRT
A C D B

Priority Q:

Process #	BT	Priority	AT
P_1	10	3	0
P_2	5	2	1
P_3	2	1	2

* Here 1 is highest Priority
3 is least Priority

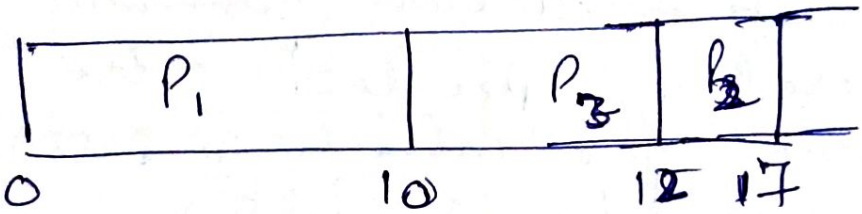
Shortest - Job first Scheduling with Preemption (SRTF)

(1)

Process	CPU - Burst	Time of Arrive
P_1	10	0
P_2	5	1
P_3	2	2

(1) non Preemptive

Shortest Job first



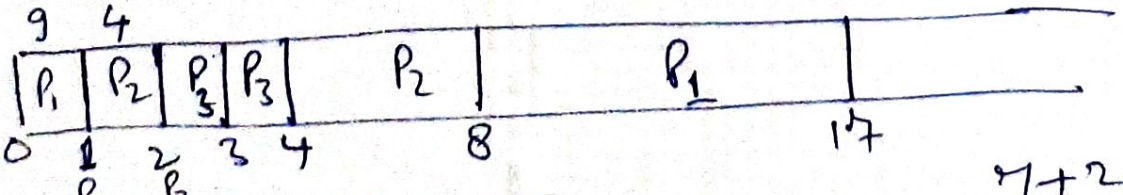
	CT	TAT(CT-AT)	WT (TAT-BT)
P_1	10	10	0
P_2	17	16	11
P_3	12	10	8

$$\text{Avg WT} = \frac{0+11+8}{3} = 6.33$$

$$\text{Avg TAT} = \frac{10+16+10}{3} = 12$$

(2) Preemptive

Shortest Job first



	CT	TAT	WT
P_1	17	17	7
P_2	8	7	2
P_3	4	2	0

$$\text{Avg WT} = \frac{7+2+0}{3} = \frac{9}{3} = 3$$

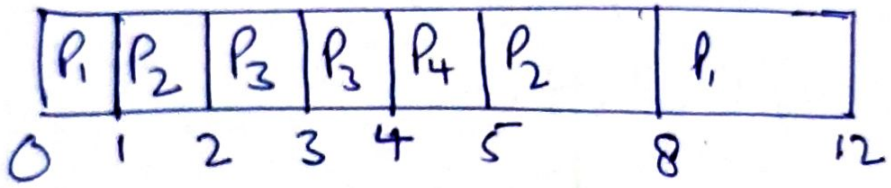
$$\text{Avg TAT} = \frac{17+7+2}{3} = 8.6$$

Advantage of SJF :- This is usually considered to be an optimal algorithm, as it gives the minimum average time

Disadvantage :- The problem is to know the length of time for which CPU is needed by a process. A prediction formula can be used to predict the amount of time for which CPU may be required by a process

Priority Scheduling with Preemption

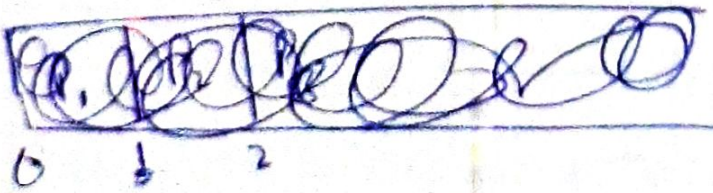
Priority	Process No	AT	BT	CT	TAT	WT
10	P_1	0	5	12	12	7
20	P_2	1	4	8	7	3
30	P_3	2	2	4	2	0
40	P_4	4	1	5	1	0



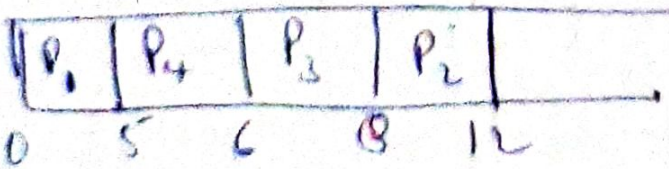
$$\text{Avg WT} = \frac{\cancel{20}}{4} = 2.5$$

$$\text{Avg TAT} = \frac{22}{4} = 5.5$$

Solve the without Preemption



{ Considering P_4 has highest Priority



	CT	TAT	WT
P_1	5	5	0
P_2	12	11	7
P_3	8	6	4
P_4	6	2	1

Avg WT = 3.00

Avg TAT = 6.00

Q-

P#	AT	BT	Priority
A	0	4	3
B	1	3	4
C	2	3	6
D	3	5	5

* A large Priority No. has High Priority

(a) Preemptive

A B C D B A
1 5

(b) Non Preemptive

A C D B
1 5

Q-

P#	AT	B-T	Priority
A	0	5	4
B	2	4	2
C	2	2	6
D	4	4	3

* Alarge No has higher Priority

(a) Preemptive

A C A D B

(b) Non Preemptive

A C D B

Problem with Preemptive Scheduling

Algo →

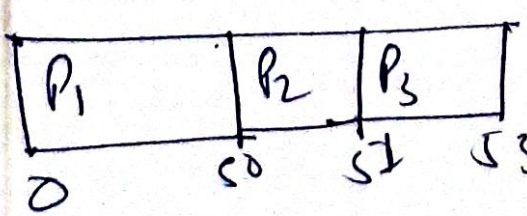
* Starvation
* If a process is in ready state but its execution is almost always Preempted by higher Priority Processes, it will starve for its execution

FCFS Serve from Convoy Effect

Case-I

	AT	BT	TAT	WT
P_1	0	50	50	0
P_2	1	1	50	49
P_3	1	2	52	50

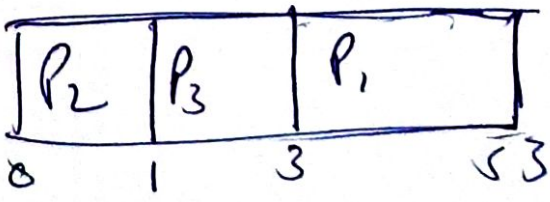
Avg = 99/3 = 33



Case-II

	AT	BT	TAT	WT
P_1	1	50	52	2
P_2	0	1	1	0
P_3	0	2	3	1

Avg = 3/3 = 1



So, we can see Avg wt. time is minimum in Case – II and maximum in Case – I because Higher B.T Processed first in Case – I — This effect is known as Convoy effect.

~~Note~~ There is no non Preemptive mode in RR

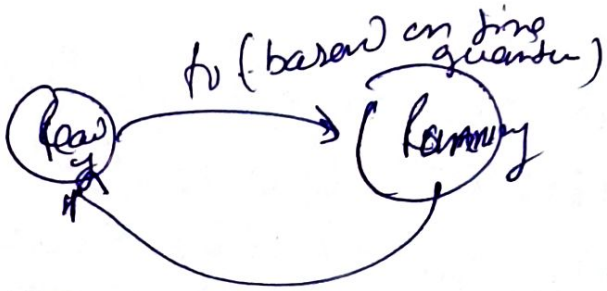
Round Robin

Criteria = time quantum
Mode = Preemptive

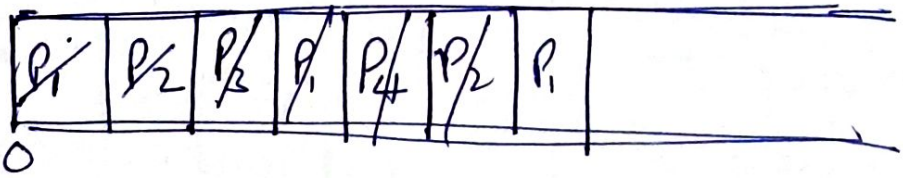
P#	A.T	B.T	C.T	TAT	WT	RT
P_1	0	~~5~~ 3 1	12	12	7	0
P_2	1	~~4~~ 2 0	11	10	6	1
P_3	2	2 0	6	4	2	2
P_4	4	~~1~~ 0	9	5	4	4

Given TQ=2

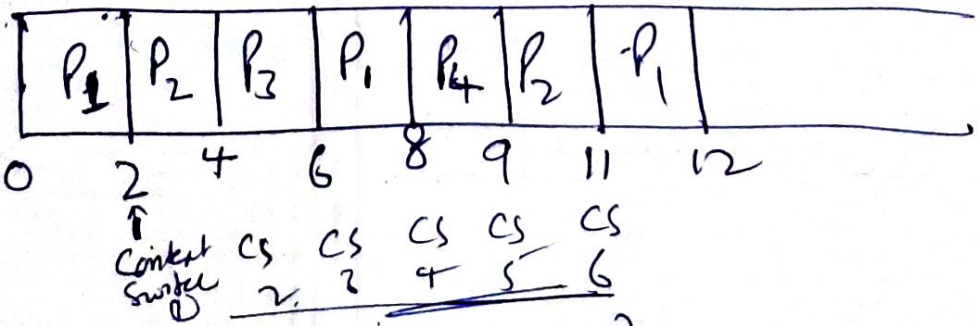
Round Robin →



Ready Queue

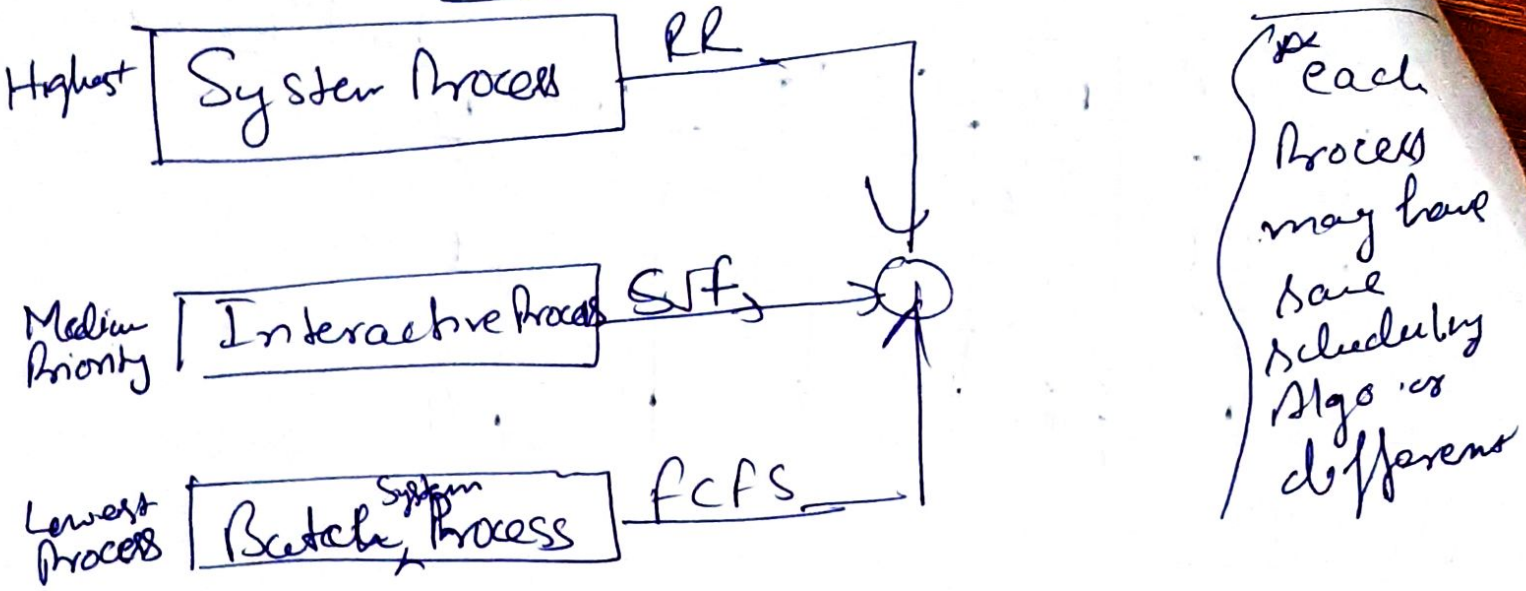


Running



RT = { CPU first time − AT }

Multilevel Queue Scheduling



There is no numerical Question exists on multilevel Queue Scheduling

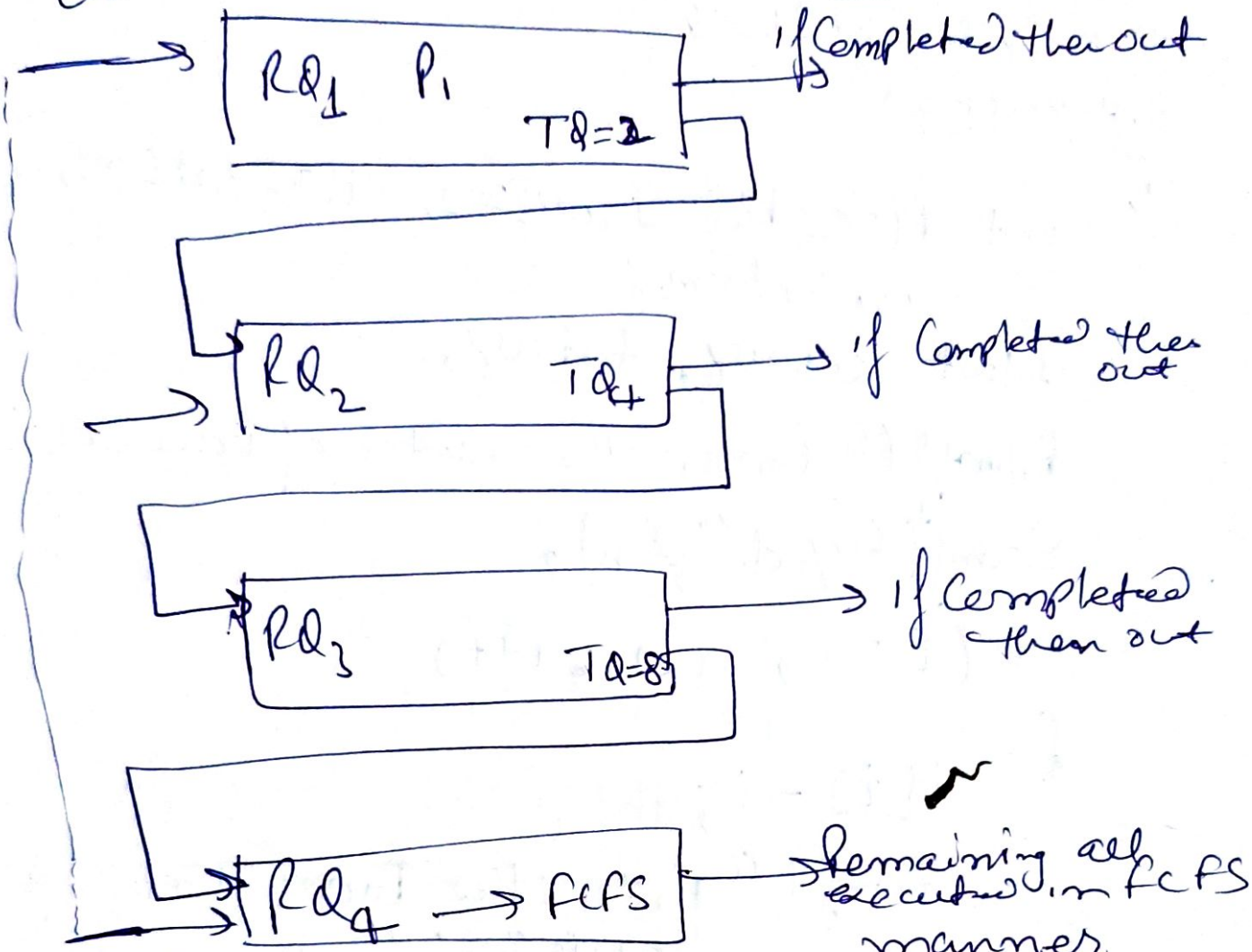
MB - Problem with this Scheduling is that if more System Processes arrives then other Processes are not getting time to execute in CPU because System Processes have highest Priority. This situation is known as Starvation.

To overcom this we use Multilevel feedback Queue Scheduling

- interactive (foreground) (higher Priority)
- Batch (Background) (lower Priority)
- Endusers Processes (lowest Priority)

Lowest Priority (Batch Process)

Multi Level Feedback



$P_1 = 19$ (Let Batch Process)

-2 ↓ move to next Queue

17 (next Queue)

-4 ↓ move to

13

-8 ↓ Like that

5 !

Q—

Process	AT	P.T
A	0	3
B	1	6
C	4	4
D	6	2

(i) RR (TQ=2)

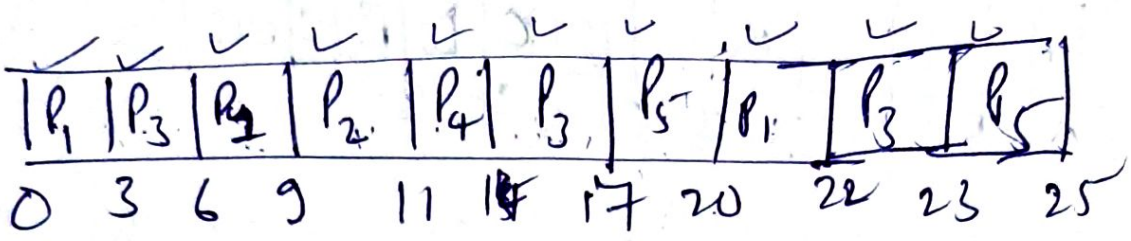
(ii) RR (TQ=1)

Q—

TQ = 3

	AT	B.T
P_1	0	8 5 2 0
P_2	5	2
P_3	1	7 4 1
P_4	6	3 0
P_5	8	5 2

$P_1 = 8$, $P_3 = 7$
5_2 4

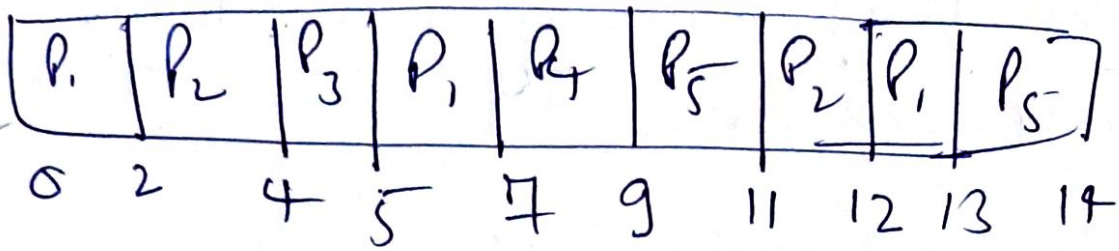


	CT	TT	WT	BT
P_1	22	22	14	0
P_2	11	6	4	4
P_3	23	22	15	2
P_4	14	8	5	5
P_5	25	13	12	9

Q

P	AT	BT
	0	5
P_1	1	3
P_2	2	1
P_3	3	2
P_4	4	3
P_5		

TQ = 2



A

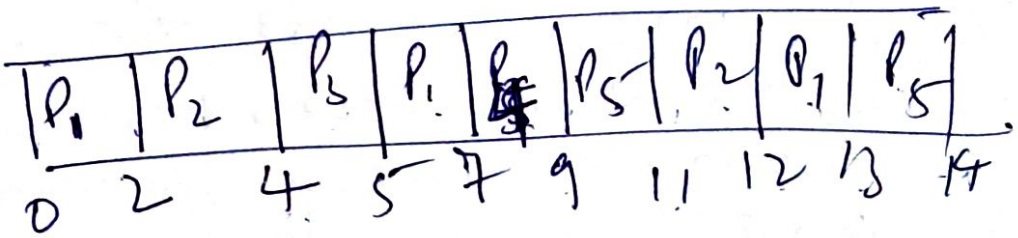
P1 - ~~5~~ ~~3~~ ~~1~~ 0

P2 - ~~3~~ ~~1~~ 0

P3 - ~~1~~ 0

P4 - ~~2~~ 0

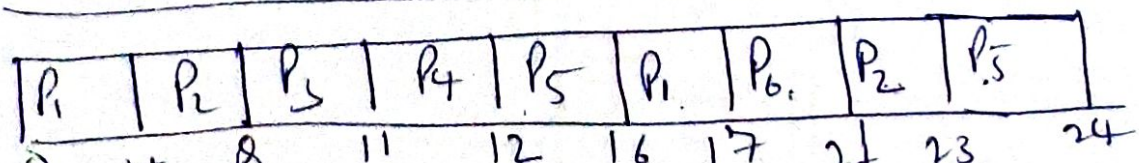
P5 - 3 1



Q - RR

Pid	AT	BT	CT	TAT	WT
1	0	5	17	17	12
2	1	6	23	22	16
3	2	3	11	9	6
4	3	1	12	9	8
5	4	5	24	20	15
6	6	4	21	15	11

Gantt chart



The Performance of Round Robin Algo depends on :-

1- Size of T.Q

- If TQ is large then this algorithm becomes same as FCFS thus Performance degrades
- If TQ size is very small then the no of Context Switches increases and TQ almost equals the time taken to switch the CPU from one Process to another. Therefore 50% of overall time spent in execution of a set of Processes due to context switching among the Processes which is not desirable at all

2- No of Context Switch —

as discussed too many Context Switches will slow down the overall execution of all Processes

Advantages of Multilevel Queue Scheduling

In a multilevel Queue algorithm, Processes are Permanently assigned to a que on entry to system. Since Processes do not change their foreground or background nature, this set up has the advantage of low scheduling overhead.

Disadvantage :- It is inflexible as the Process can never change their queues and thus may have to starve for the CPU if one or the other higher Priority queues are never becoming empty

Advantages of Multilevel feedback Queue :-

1- Multilevel feedback queue scheduling allows a Process to move between queues. Thus is fair for I/O bound Processes, they do not have to wait too long

2- A Process that waits too long in a lower Priority queue may be moved to a higher Priority queue, this form of aging Prevents starvation

3- The definition of multilevel feedback queue scheduling makes it the most general CPU scheduling algorithm. It can be configured to match a specific system under design.

4- It is more flexible.

Disadvantages:

1. It requires some means of selecting values for all the parameters to define the best scheduler.
2. Moving the process around the queues produces more CPU overheads.
3. It is the most complex scheduling algorithm.

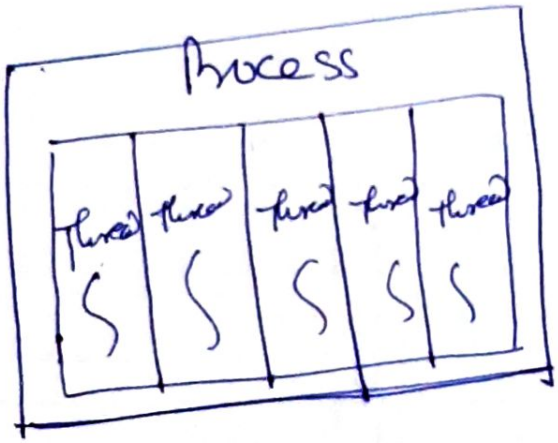
Unit-1

Process and Thread

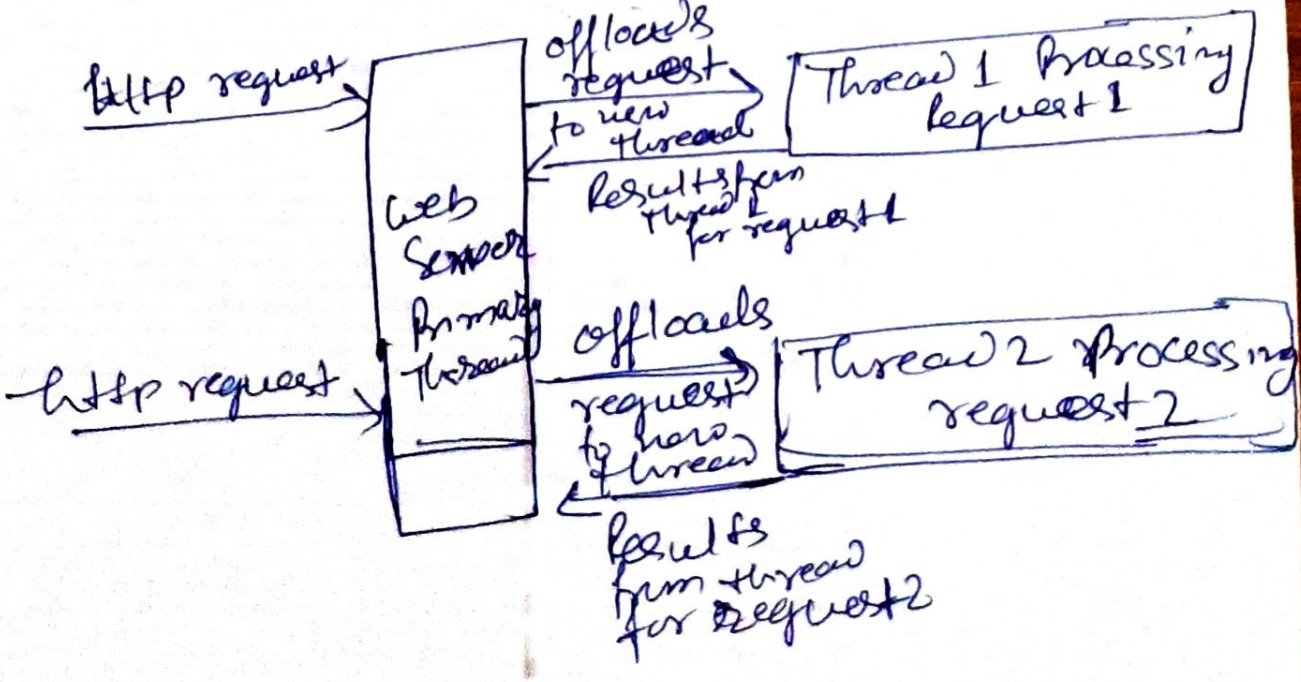
Multithreading

Process – A Process can be thought of as a Program in execution

Thread → A thread is the unit of execution within a process.
A process can have anywhere from one thread to many thread.



Ex –



Unit – 1

Multithreading

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

→ A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory, all other threads see that.

→ A thread is also called <u>lightweight Process</u>. Threads Provides a way to improve application Performance through Parallelism. Threads represent a software approach to improve performance of O.S by reducing the overhead. Thread is equivalent to a classical Process.

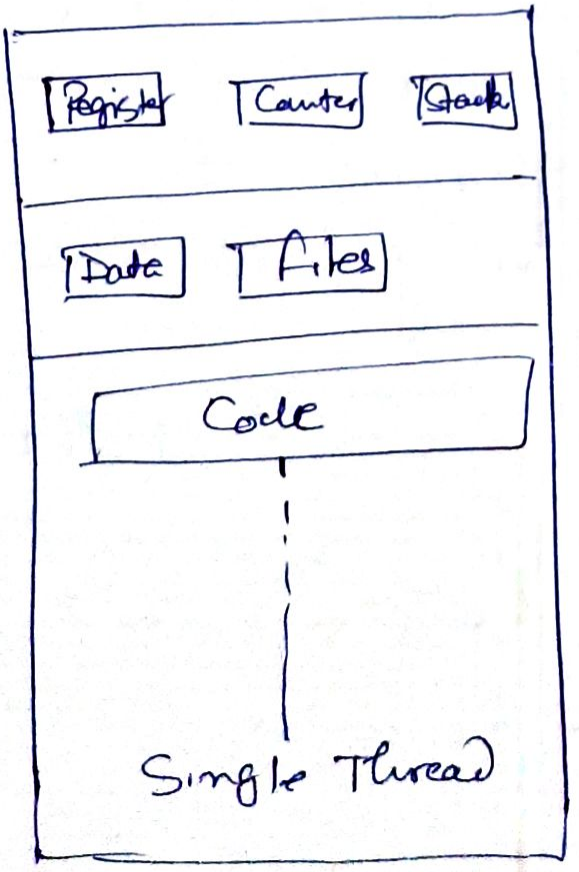
→ Each thread belongs to exactly one Process and no thread can exist outside a Process

→ Each thread represents a separate flow of control.

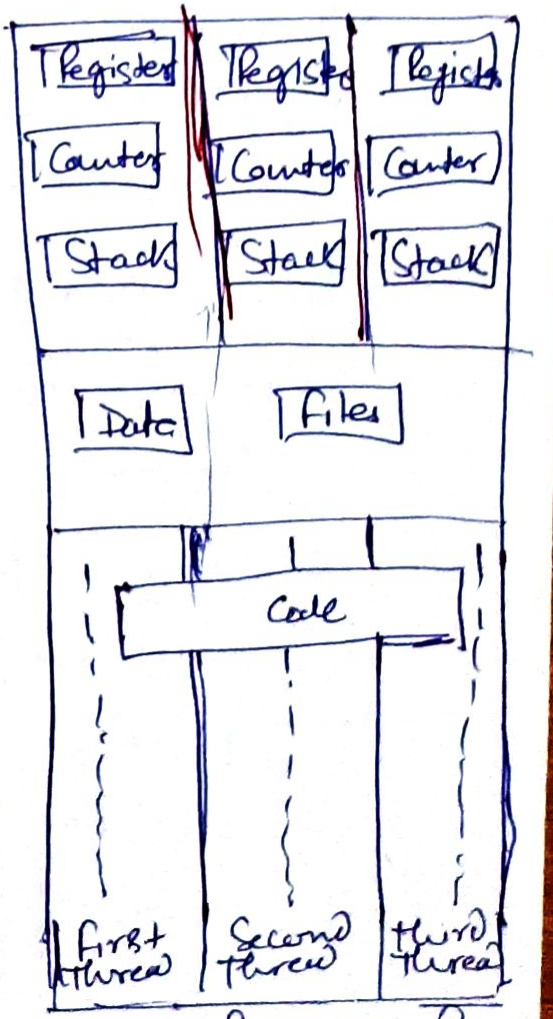
→ Threads have been successfully used in implementing network servers and web servers.

→ They also provide a suitable foundation for Parallel execution of applications on shared memory multiprocessors.

The following fig. shows the working of a single threaded and multithreaded Process



Single Process with Single thread



Single Process P with three threads

Difference between Process and Thread :-

Process	Thread
1- Process is heavy weight or resource intensive	Thread is lightweight, taking lesser resources than a Process
2- Process switching needs interaction with O.S	Thread switching does not need to interact with O.S
3- In Multi Processing environments, each Process executes the same code but it has its own memory and file resources	All threads can share same set of open files, child Processes.
4- If one Process is blocked, then other Process can execute.	While one thread is blocked, the entire Process is blocked.
5 - Multiple Processes without using threads use more resources.	Multiple threaded Processes use fewer resources
6- In multiple Processes each Process operates independently of the others	One thread can read, write or change another thread's data (Interdependent)

Advantages of Thread :-

→ Threads minimize the context switching time

→ Use of threads provides concurrency within a process

→ Efficient communication.

→ It is more economical to create and context switch threads.

→ Threads allow utilization of multiprocessor architecture to a greater scale and efficiency.

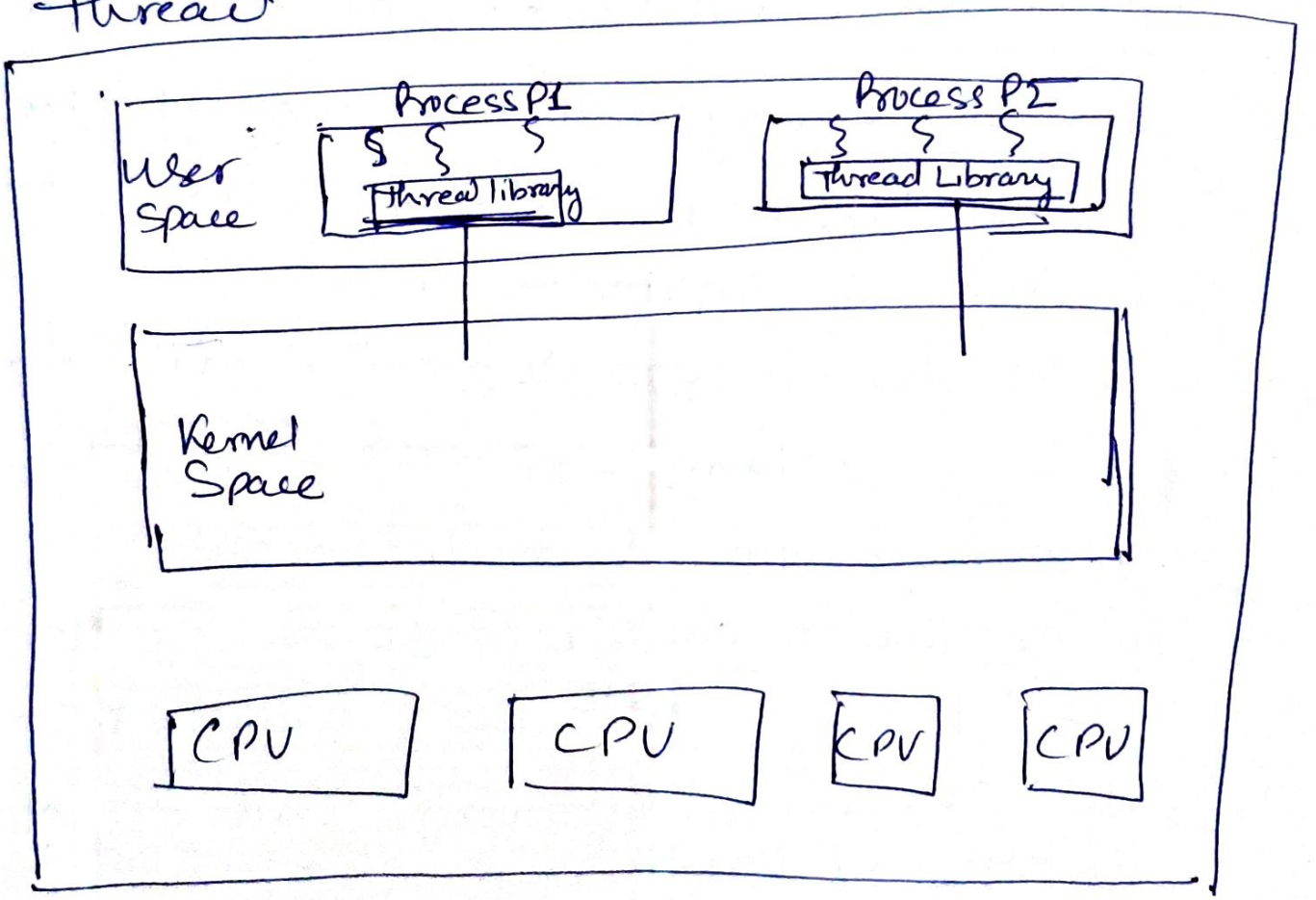
Types of Thread → Threads are implemented in following two ways.

- **User Level Threads** – User managed threads
- **Kernel level Threads** – O.S managed threads acting on Kernel, an O.S core

User Level Threads :- In this case, the thread management Kernel is not aware of the existence of threads.

→ The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts.

The application starts with a single thread.



Advantages :- Thread switching does not require Kernel mode Privileges

- User level thread can run on any O.S
- Scheduling can be application specific in the user level thread
- User level threads are fast to create and manage

Disadvantages →

- In a typical O.S, most system calls are blocking.
- Multithreaded application cannot take advantage of multiprocessing.

Kernel Level Threads → In this case, thread management is done by the Kernel. There is no thread management code in the application area.

→ Kernel threads are supported directly by the O.S. Any application can be programmed to be multithreaded.

→ All of the threads within an application are supported within a single process.

→ The Kernel maintains context information for the process as a whole and for individuals threads within the process

→ Scheduling by the Kernel is done on a thread basis.

→ The Kernel performs thread creation, scheduling and management in Kernel space.

→ Kernel threads are generally slower to create and manage than the user threads

Advantages :-

- Kernel can simultaneously schedule multiple threads from the same Process on multiple processors.
- If one thread in a Process is blocked, the Kernel can schedule another thread of same Process
- Kernel routines themselves can be multithreaded

Disadvantages :-> Kernel threads are generally slower to create and manage than the user threads.

-> Transfer of control from one thread to another within the same Process requires a mode switch to kernel.

Multithreading Models

Some O.S Provide a combined user level thread and Kernel level thread facility.

Ex- Solaris is a good example of this combined approach.

-> In combined system, multiple threads

within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process.

→ Multithreading models are three types

→ Many - to-many relationship

→ Many - to - one relationship

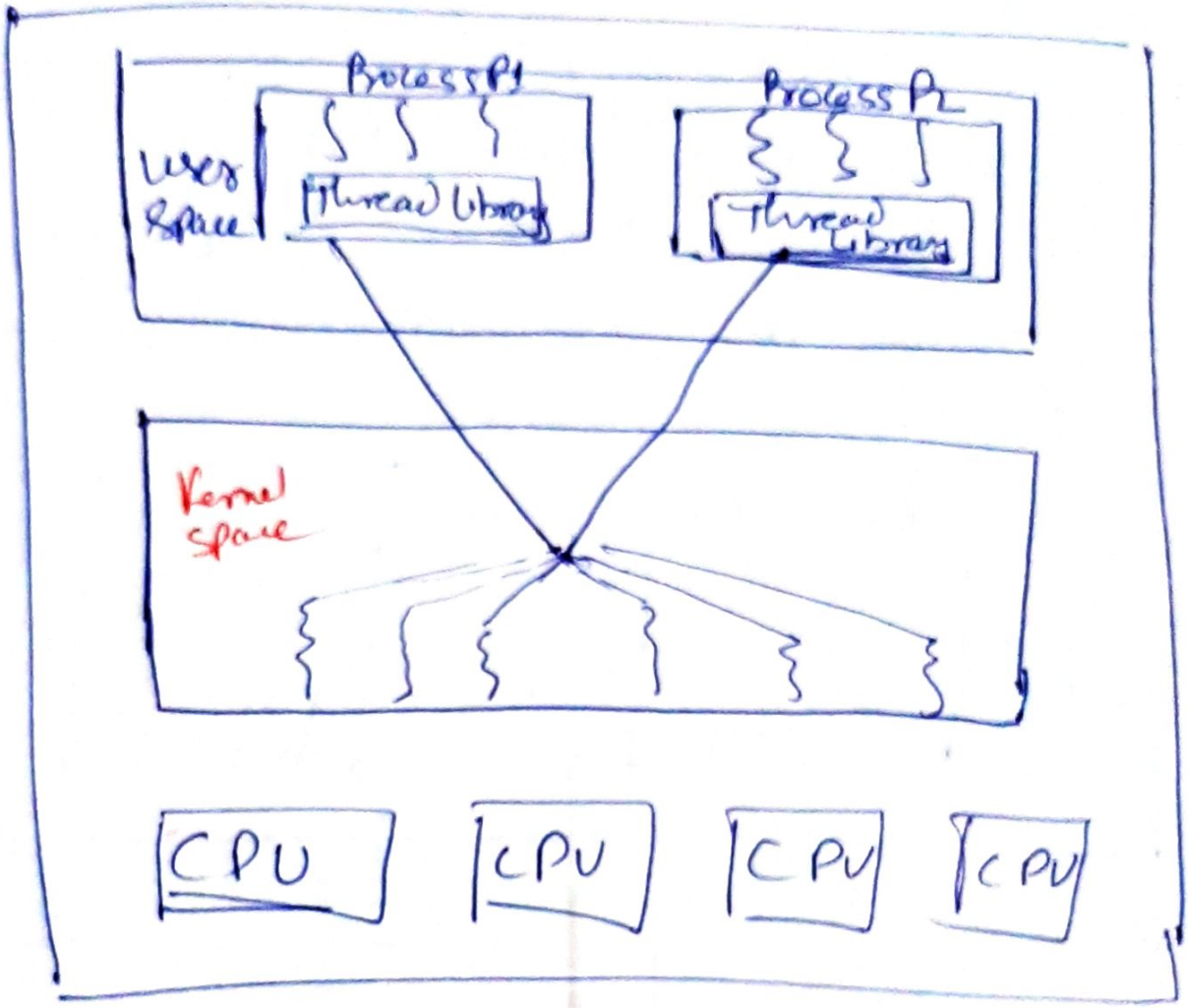
→ One to - one relationship

Many to Many Model → The many-to many model multiplexes any number of user threads onto an equal or smaller number of Kernel threads

→ The following diagram shows the many to many threading model where 6 user level threads are multiplexing with 6 Kernel level threads. In this, model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in Parallel on a multiprocessor machine.

→ This model provides the best accuracy on concurrency and when a thread performs a blocking systems call, the Kernel can

schedule another thread for execution

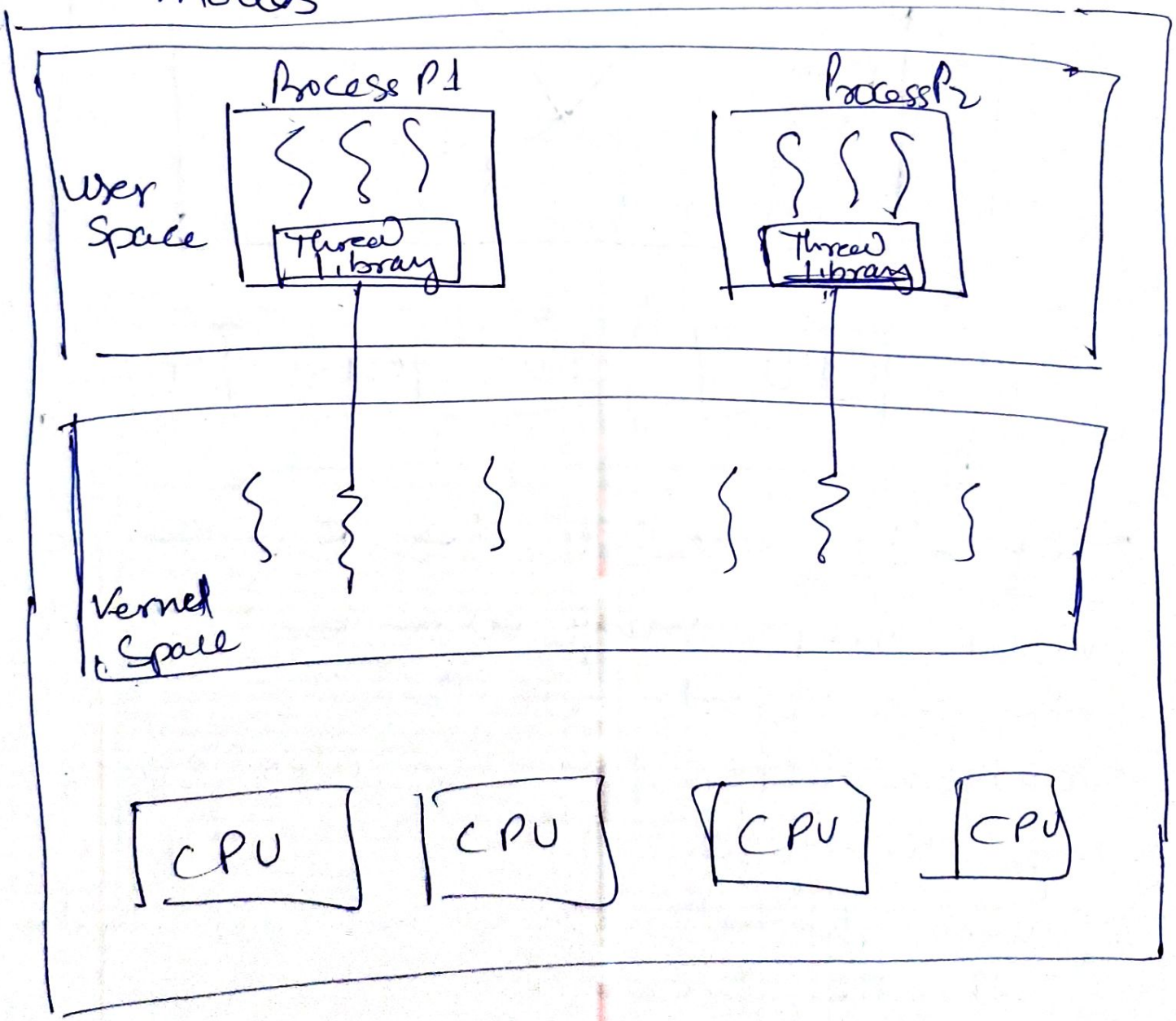


Many-to-one Model → Many to one

model maps many user level threads to one Kernel-level thread. Thread management is done in user space by the thread library.

- When thread makes a blocking system call, the entire process will be blocked. Only one thread can access the kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

→ If the user-level thread libraries are implemented in the O.S in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship model



one-to-one Model → There is one-to-one relationship of user-level thread to the Kernel-level thread.

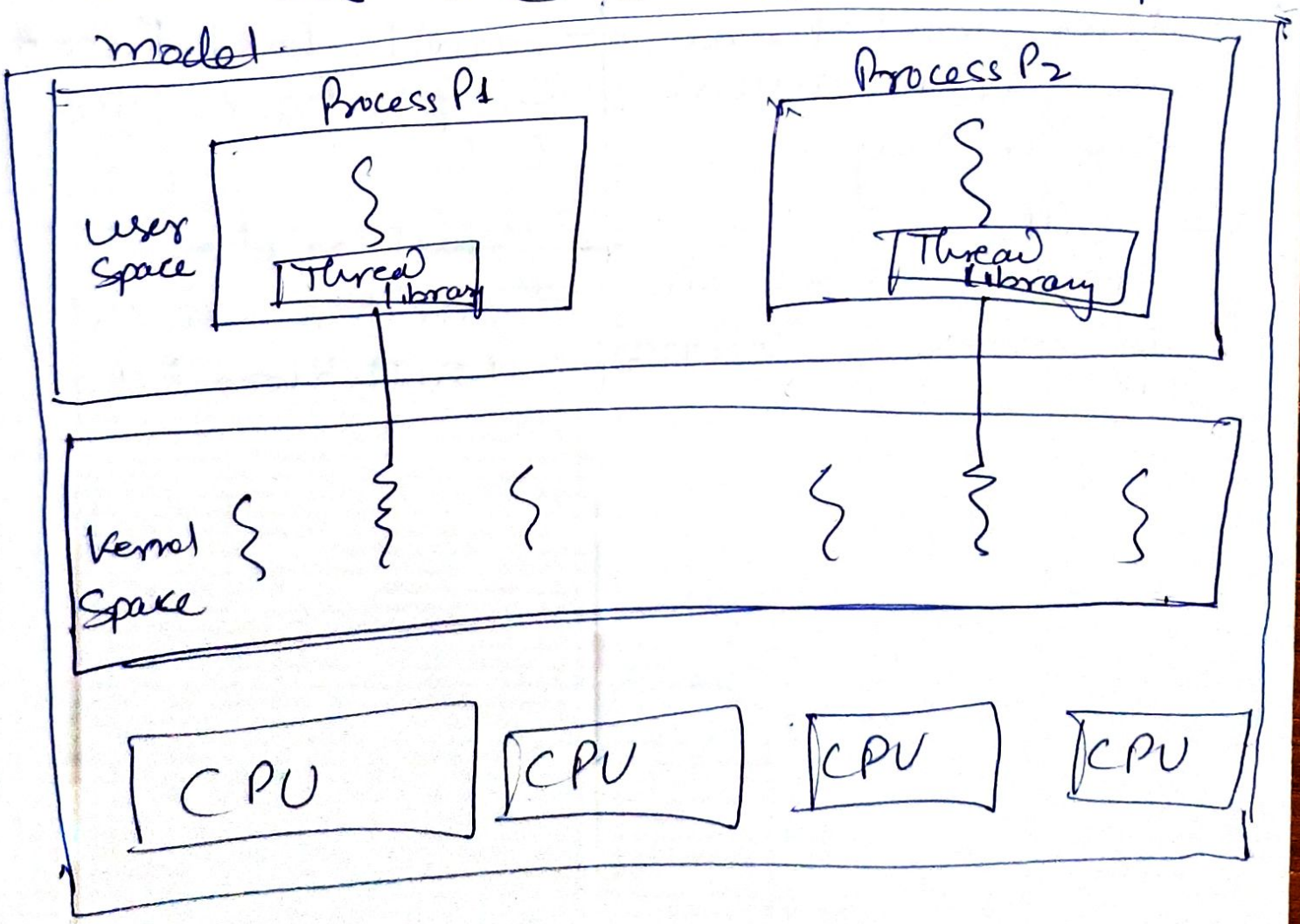
→ This model provides more concurrency than

the many to one model. It allows another thread to run when a thread makes a blocking system call.

→ It supports multiple threads to execute in parallel on microprocessors.

→ Disadvantage of this model is that creating user thread requires corresponding kernel thread.

OS/2, Windows NT and Windows 2000 use one to one relationship model.



Difference between User-Level & Kernel level thread :-

User Level Thread	Kernel-level Thread
1- User-level threads are faster to create and manage	Kernel-level threads are slower to create and manage.
2- Implementation is by a thread library at the user level	O.S Support & creation of Kernel threads
3- User-level thread is generic and can run on any O.S	Kernel-level thread is specific to the O.S
4- Multi-threaded applications can't take advantage of Multiprocessing	Kernel routines themselves can be Multithreaded.

Interprocess Communication

"Process executing concurrently in the O.S may be either independent processes or cooperating processes."

Independent Processes → They can't affect or be affected by the other processes executing in the system.

Cooperating Processes → They can affect or be affected by the other processes executing in the system

* → Any process that shares data with other processes is a cooperating process

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

① Information Sharing
② Computation Speedup.
③ Modularity
④ Convenience.

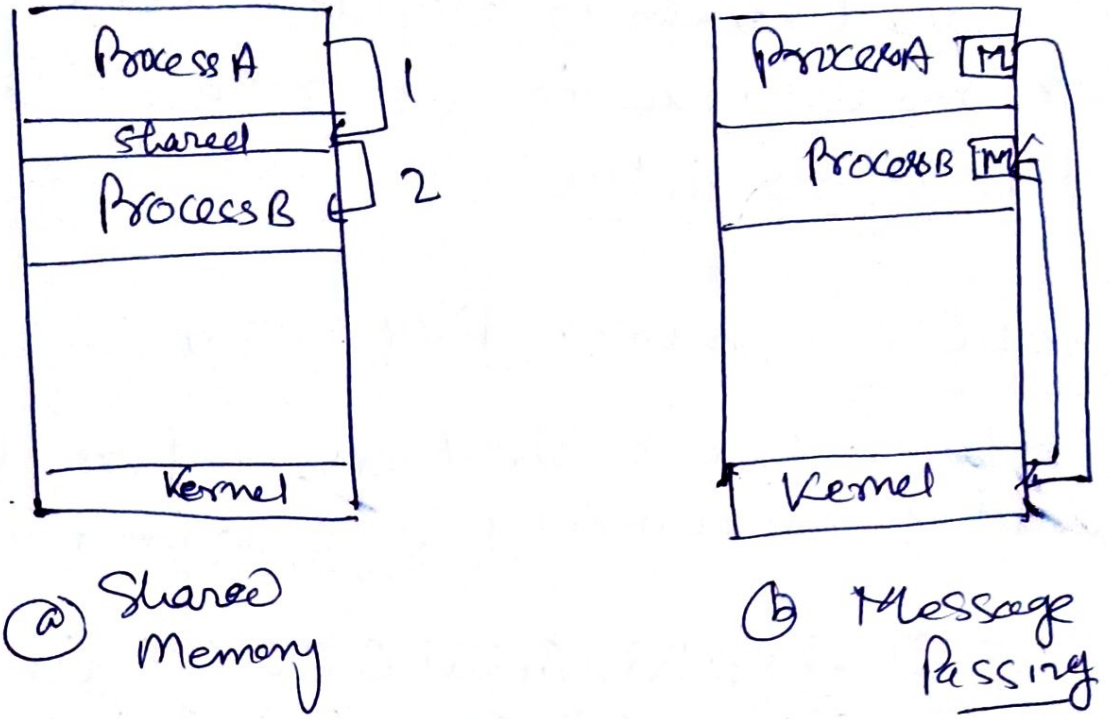
Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information

There are two fundamental models of IPC

(1) Shared memory
(2) Message Passing

(1) In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.

(2) In the message passing model, communication takes place by means of messages exchanged between the cooperating processes.



(a) Shared Memory

(b) Message Passing

Shared Memory System

* IPC using shared memory requires communicating processes to establish a region of shared memory.
* Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment.
* Other processes that wish to communicate using this shared-memory segment must attach it to their address space.
* Normally, the O.S system tries to prevent one process from accessing another process's memory.

* Shared memory requires that two or more Processes agree to remove this restriction

Producer Consumer Problem

A Producer Process Produces information that is consumed by a Consumer Process

Ex — Compiler may Produce assembly code, which is consumed by an assembler The assembler, in turn may Produce object module, which is consumed by the loader

→ One solution to the Producer-Consumer Problem uses Shared memory.

→ To allow Producer and Consumer Processes to run concurrently, we must have available a buffer of items that can be filled by the Producer and emptied by the Consumer.

→ This buffer will reside in a region of memory that is Shared by the Producer and Consumer Processes

→ A Producer can Produce one item while the Consumer is Consuming another item

→ The Producer and Consumer must be synchronized, so that the Consumer does not try to consume an item that has not yet been Produced

Two Kinds of Buffer :-

(1) Unbounded Buffer

Places no Practical limit on the size of the buffer. The Consumer may have to wait for new items but the Producer can always Produce new items

(2) Bounded Buffer → Assumes a fixed buffer size. In this case, the Consumer must wait if the buffer is empty, and the Producer must wait if buffer is full.

---

* example of Producer Consumer — Printing of documents — we can Print from several applications, i.e multiple Producers can create Printing "messages" that are enqueued by a Printer spooler and later consumed by a Printer.

Message-passing System

Message Passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.

⇒ A message-passing facility provides at least two operations

- send (message)

and

- receive (message)

⇒ messages sent by a process can be of either fixed or variable size

Fixed Size :- The system-level implementation is straightforward. But make the task of programming more difficult.

Variable Size :- Requires a more complex system level implementation But the programming task becomes simpler

⇒ If Process P and Q want to Communicate, they must send messages to and receive messages from each other.

⇒ A Communication link must exist between them

The link can be implemented in a variety of ways. There are several methods for logically implementing a link and the send()/receive() operations, like:

- → Direct or indirect Communication
- → Synchronous or asynchronous Communication
- → Automatic or explicit buffering

There are several issues related with features like:

① Naming
② Synchronization
③ Buffering

Part - 2

Naming: Processes that want to Communicate must have a way to refer to each other. They can use either direct or indirect communication

Under Direct Communication

Each Process that wants to Communicate must explicitly name the recipient or recipient or Sender of the Communication.

→ Send(P, message) → Send message to Process P.

→ Receive(Q, message) → Receive a message from Process Q.

A Communication link in this scheme has the following Properties:-

→ A link is established automatically between every pair of Processes that want to Communicate. The Processes need to know only each other's identity to communicate

→ A link is associated with exactly two Processes

→ Between each pair of Processes, there exists exactly one link.

NB — This scheme exhibits Symmetry in addressing; that is both the Sender Process and receiver Process must name each other to communicate

Another Variant of Direct Communications

Here, only the Sender names the recipient; the recipient is not required to name the Sender.

→ Send (P, message) – Send a message to Process P.

→ Receive (id, message) – Receive a message from any Process;
- the variable id is set to the name of the Process with which communication has taken Place

Ex: This scheme employs asymmetry in addressing

Disadvantage :

- The limited modularity of the resulting Process definitions.

i.e. Changing the identifier of a Process may necessitate examining all other Process definitions.

With indirect Communications the messages are sent to and received from mailboxes, or Ports

→ A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.

→ Each mailbox has a unique identification

→ Two processes can communicate only if the processes have a shared mailbox

→ Send (A, message) – Send a message to mailbox A

→ receive (A, message) – Receive a message from mailbox A

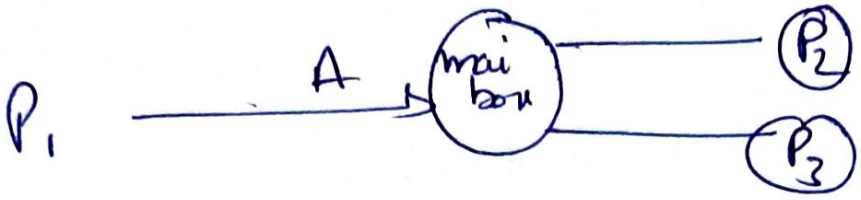
A communication link in this scheme (Indirect communication) has the following properties →

→ A link is established between a pair of processes only if both members of the pair have a shared mailbox

→ A link may be associated with more than two processes.

→ Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mailbox.

Now Suppose that Processes P_1, P_2 and P_3 all share mailbox A



Process P_1 Sends a message to A, while both P_2 and P_3 execute a receive() from A. Which Process will receive the message sent by P1?

Ans The answer depends on which of the following methods we choose:

1. Allow a link to be associated with two Processes at most.
2. Allow at most one Process at a time to execute a receive() operation
3. Allow the system to select arbitrarily which Process will receive the message (that is either P_2 or P_3, but not both, will receive the message). The System also may define an algorithm for selecting which Process will receive the message (that is, round robin where Processes take turns receiving messages). The System may identify the receiver to the Sender

* A mailbox may be owned either by a process or by the O.S.

Part-3

Synchronization → Communication between processes takes place through calls to send() and receive() primitives. There are different design options for implementing each primitive.

⇒ Message passing may be either blocking or nonblocking — also known as synchronous and asynchronous

Blocking Send — The sending process is blocked until the message is received by the receiving process or by the mailbox.

Nonblocking Send → The sending process sends the message and resumes operations.

Blocking Receive: — The receiver blocks until a message is available

Nonblocking Receive → The receiver retrieves either a valid message or a null.

Buffering — When communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

1. Zero Capacity: The queue has a maximum length of zero; Thus, the link can't have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
2. Bounded Capacity: The queue has finite length n; Thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue and the sender can continue execution without waiting. The link's capacity is finite, however. If the link is full,

the Sender must block until space is available in the queue.

③ Unbounded Capacity → the queue length is Potentially infinite, thus, any number of messages can wait in it. The Sender never blocks.

Deadlock and Starvation – (Improper use of Semaphore)

S = 1, Q = 1 } Semaphores

P_0	P_1
wait(S)	wait(Q)
I-① wait(Q)	wait(S)
I-② body	body
Signal(S)	Signal(Q)
Signal(Q)	Signal(S)

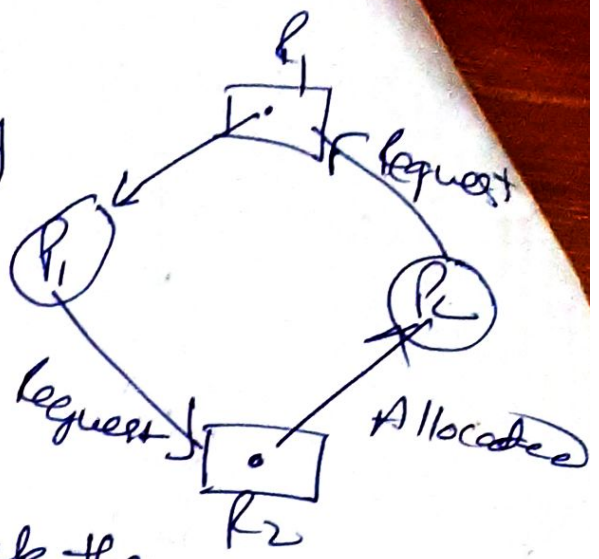
Now I-1 executed by P_0 makes S=1 to 0. Now, P_1 will execute the I-1 will make Q = 1 to 0

Now P_0 will be blocked while executing I-2. Like P_1 will be blocked while executing I-2

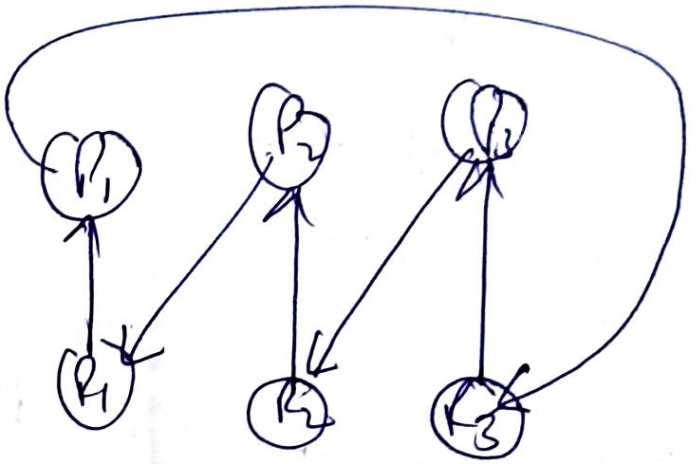
⇒ So, P_0 is blocked by P_1 and P_1 is blocked by P_0

⇒ Thus results in deadlock and Starvation (no Process can Progress)

Deadlock → if two or more Processes are waiting on happening some event, which never happens, then we say these Processes are involved in deadlock then that State is called deadlock State.



Ex



Deadlock (Circular wait)

Resource Allocation Graph (RAG)

- Vertex
  - Process vertex → P_i
  - Resource Vertex
    - Single Instance → CPU, Monitor
    - Multiple Instance → eg register
- Edge
  - Assign edge → P → R
  - Request edge → P → R

Deadlock :-

* A Process or thread is in the state Deadlock if it is waiting for a particular event that will not occur.

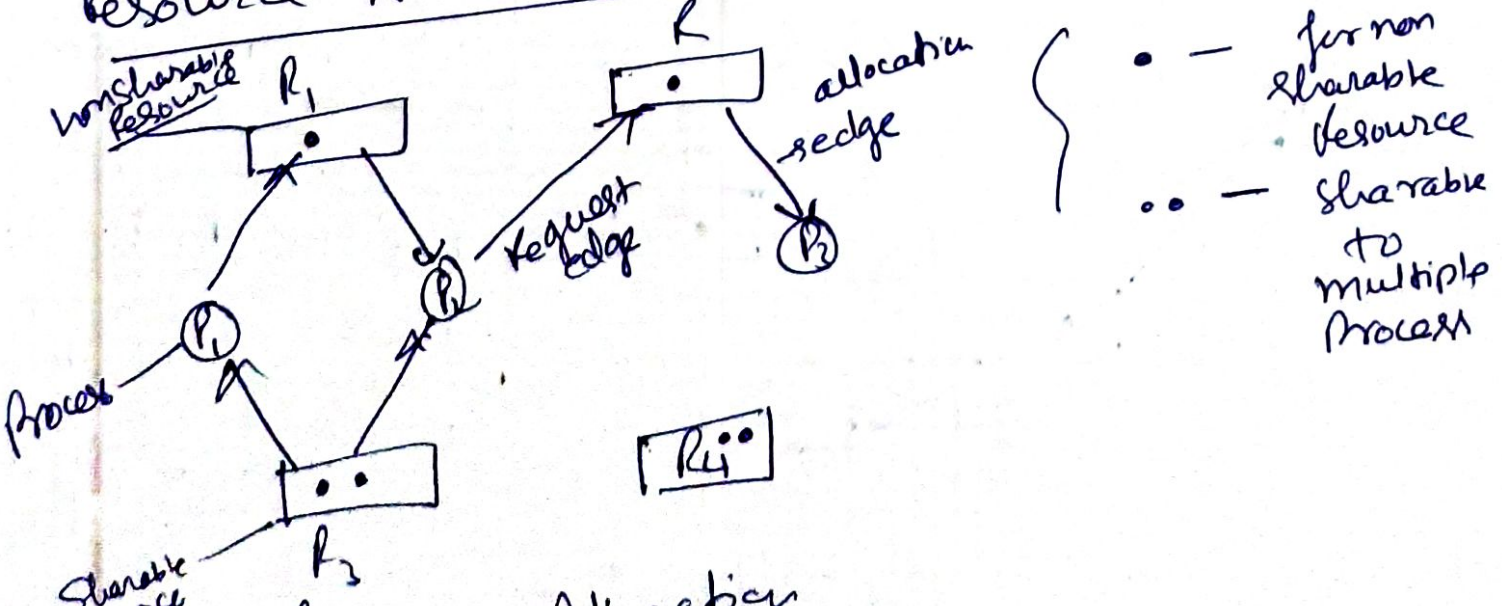
* Resource allocation sequence
  - Request
  - Use
  - Release

* Deadlock characterization: (four necessary conditions)

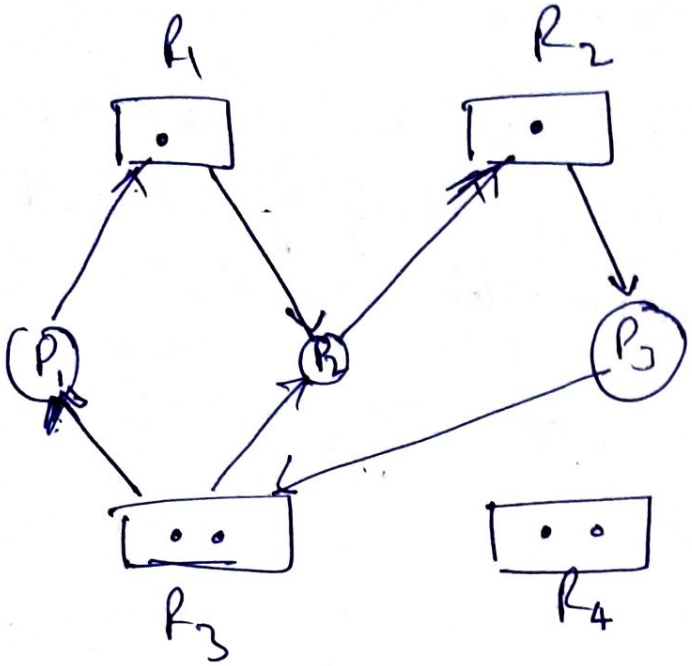
Indirect Cause
- Mutual exclusion
- Hold and wait
- No preemption

Direct Cause / mean cause
- Circular wait

Resource Allocation Graph :-



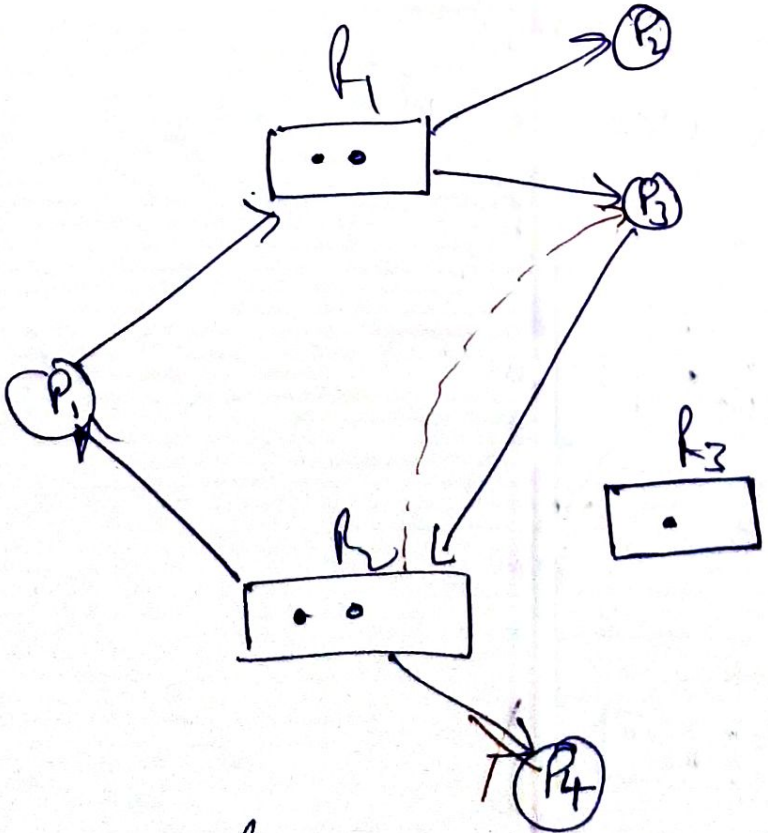
Resource Allocation Graph



Resource allocation Graph with a deadlock

i.e if Resource Allocation Graph has cycle then deadlock may or may not occur

* if a resource Allocation graph has no cycle, deadlock will not occur



Thus not not deadlock because (P4) will gets Completed because it is not Competing for any Resource than in R2 second resource can be allocated to (P3) i.e P3 to R2 a request edge can be Converted to allocation edge

* Hence No Deadlock will occur

Resource allocation Graph with a cycle but no deadlock

* Deadlock Prevention
* Deadlock avoidance
* Deadlock detection
* Deadlock recovery

Necessary Conditions for deadlock :

* Mutual Exclusion
* Hold and wait → 

* no Preemption
* Circular wait

(Various ways to handle deadlocks)

Deadlock Prevention

① (Solution) **Mutual Exclusion :-** In case of to Prevent deadlock, we must identify all resources under to head sharable & nonsharable. Now, if we have idea about sharable resources the we can use synchronization technique to Prevent from Deadlock

② **Hold and Wait :-** ① Solution is that Process should get or Provided with all resources ie in atomic manner. (ie before starting execution) during the Process if P_1 is having Resources

R_1 & R_2 and it needs another resource R_3 then first release R_1 & R_2 and the allocate R_1, R_2 and R_3 all together

Problem with both solution :

① utilization of resource is very Poor

ex — tape drive → disk → Printer

(sequential device)

So, either of any resource may be unutilize in this case.

② Starvation (demand for a resource which has a huge demand)

Solution :(of Problem)

① Just allocate only that resource which is only needed at a Particular time.

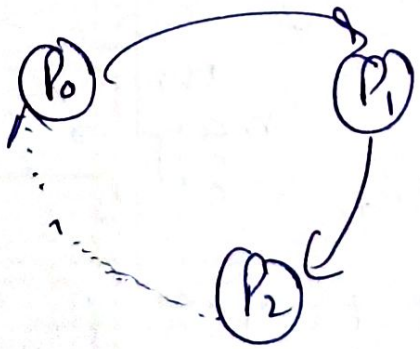
ex — Here in this case first allocate tape drive & disk, so that Process can read data from tape drive and sort the data in disk the Secondary disk & Printer, so that sorted data can be Printed

③ No Preemption (Solution) — The O.S has ~~~~ Permission to take away a resource from a waiting Process for a new Process

*** Here waiting Process is holding the current Resource, which is in demand & waiting for some other resource, & the new Process is demanding for the holded resource.

Ex Timp Stamp is also solution

④ Circular Wait :



NS ⇒ Process should request for Resources in increasing order.

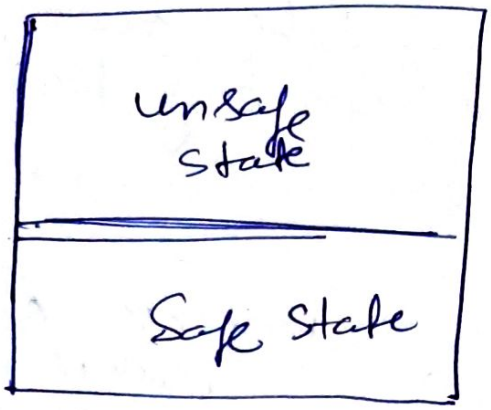
Example
- f (tape drive) = 1 — Highest Priority
- f (disk drive) = 4
- f (Printer) = 10 — lowest Priority

Priority decreasing

* So, this says that if a Process is demanding for a high Priority resource than, Enhance the Priority of Process and allocate the resource to that Process.

In this way circular wait will never occur

Deadlock Avoidance, Banker's Algo.



100 100

(45+45) = 90

both Person withdraw 45/- : 10 — 20

+ 150

Another Person may wants to withdraw 20/- (Case of deadlock)

160

but at the same time someone can deposite 150/- so that a total can be 160/- and deadlock will not occur in this case if someone is withdrawing 10/-

* So, it means an unsafe state deadlock may or may not occur

Banker's Algo

1- Work and Finish be two vector of size m and n. Initialize work with available and Finish[i] = 0 (false) for i = 1 to n

2- Find an i such that both
   (i) Finish[i] = false
   (ii) Need$_i$ ≤ work
   if no such i exists then go to step-4

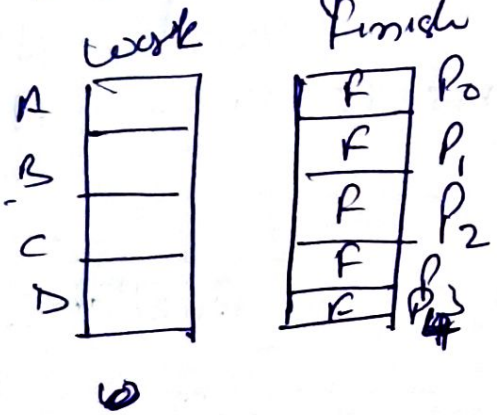
3 - Work = Work + Allocation;
Finish[i] = True
go to step-2

4 - If Finish[i] = True for all i
the system is in safe state.

Explanation

m ⇒ # no of resources (ex A B C D)

n ⇒ # no of Process (ex P_0 to P_4)



Example -

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0
P_1	1	0	0	0	1	7	5	0				
P_2	1	3	5	4	2	3	5	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				

- Calculate matrix need
- Is the system is in Safe State ?
- If P_1 arrives with request (0, 3, 2, 0) can it be granted immediately

n => no of Processcesses = 5 = P_0 to P_4

m => " " Resources = 4 = A B C D

Available [m] → { one dimensional matrix }
(Size)

Allocation [n][m] → { i.e n × m matrix
Size }

Max [n][m] → { Size n×m }

Need [n][m] → [Size n×m]

Need

(How many more resources are required)

{ need = max − Allocation }

	A	B	C	D
P_0	0	0	0	0
P_1	0	7	5	0
P_2	1	0	0	2
P_3	0	0	2	0
P_4	0	6	4	2

Available

A	B	C	D
1	5	2	0

=> P_0 is satisfying the Condition i.e $need_0 \leq work$

So, $P_0 == T$ (i.e executed)

Now if P_0 is executed then its initial allocation becomes free. i.e it will be added to work

Work	Finish
A B C D	0 F T
1 5 2 0	1 F T
+ 0 0 1 2	2 F T
1 5 3 2	3 F T
+ 1 3 5 4 (P_2)	4 F T
2 8 8 6	
+ 0 6 3 2 (P_3)	
2 1 4 1 8	
+ 0 0 1 4 (P_4)	
2 1 4 1 2 1 2	
+ 1 0 0 0 P_1	
3 1 4 1 2 1 2	

⇒ P_1 Need of $_1 \leq$ Work { not satisfied

So, P_1 is not executed

⇒ P_2 ⌐ Need of $_2 \leq$ work { satisfied

P_2 is executed

⇒ P_3 ⌐ Nee$_3 \leq$ work (satisfied

P_3 is executed

⇒ P_4 ⌐ Need$_4 \leq$ Work (satisfied)

P_4 is executed

⇒ Come back (i P_1)

P_1 Need $\leq$ work (satisfied)

So, P_1 is executed

$< P_0\ P_2\ P_3\ P_4\ P_1 >$

Safe Sequence

⇒ If P_1 arrives with (0, 3, 2, 0)

Need again Need$_1 \leq$ work (satisfied

So granted — Yes Ans

Q-

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

Question → what will happen if Process P_1 requests one additional instance of resource type A and two instance of resource type C

request = 1 0 2 (A B C)

Finish

P_0	F
P_1	F
P_2	F
P_3	F
P_4	F

Work A B C

3 3 2
+ 2 0 0 (P_1)
5 3 2
+ 2 1 1 (P_3)
7 4 3
+ 0 0 2 P_4
7 4 5
+ 0 1 0 P_0
7 5 5
+ 3 0 2
1 0 5 7

Safe Sequence

Need

	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

~~P_0~~ — not allowed
P_1 — allowed
P_2 — not allowed.
P_3 — allowed
P_4 — allowed
again P_0
P_0 — allowed
P_2 — allowed
i.e $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ Safe Sequence

Resource – request Algorithm :

1 – If $request_i \leq need_i$ then go to step 2, otherwise raise error

2 – If $request_i \leq available_i$ then go to step 3, otherwise wait until available

3 – System pretend as if request has been granted by modifying the state:

$available_i = available_i - request_i$

$need_i = need_i - request_i$

$allocation_i = allocation_i + request_i$

After this we need to check where the new request leads to a safe state or not

Ex

Allocation	A	B	C
P_0	5	4	2
	3	3	0
	4	8	5

Need

Request

Available

A	B	C
7	5	6

i.e – both 1 & 2 satisfied

Now Step – 3

Available 7 5 6

– 3 3 0 (request$_i$)

4 2 6 — new available

Need = 5 4 2

– 3 3 0

2 1 2 (new need)

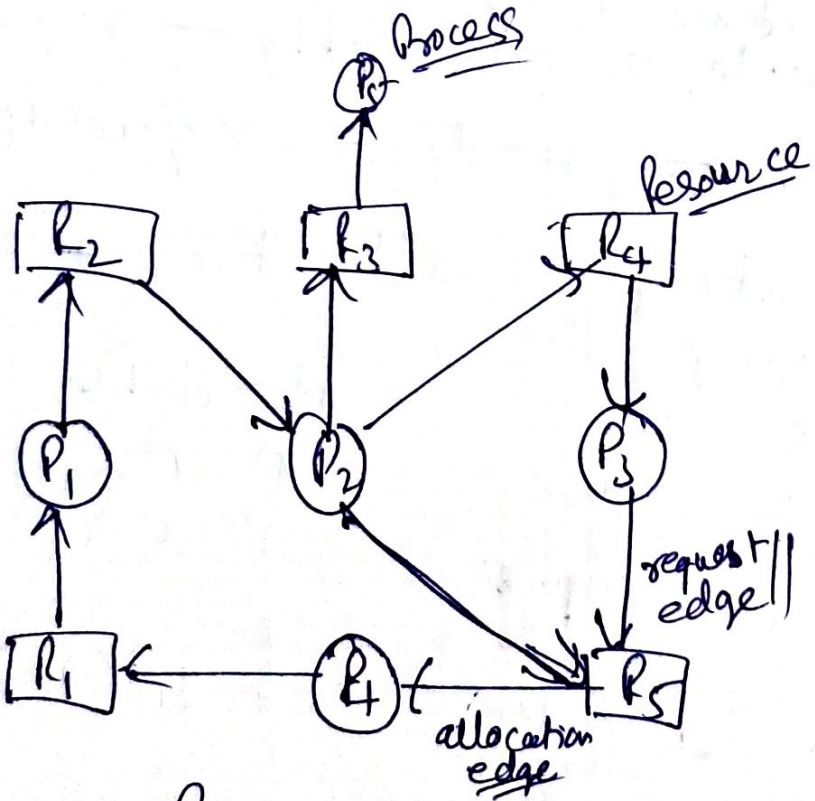
New Allocation

P_0 A B C
7 8 5
+ 3 3 0
10 11 5 (new allocation)

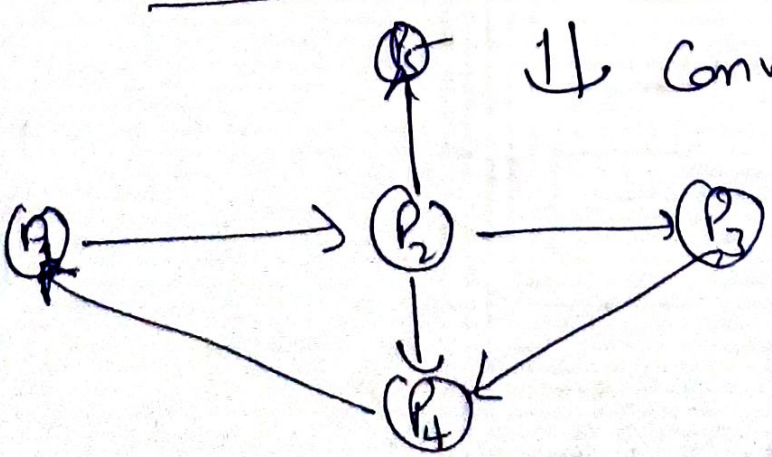
X. X.

Deadlock Detection

- Single instance (wait for graph)
- Multiple instances (Bankers algo)



Resource Allocation graph



↓↓ Convert it to wait for graph

Same graph in wait for graph

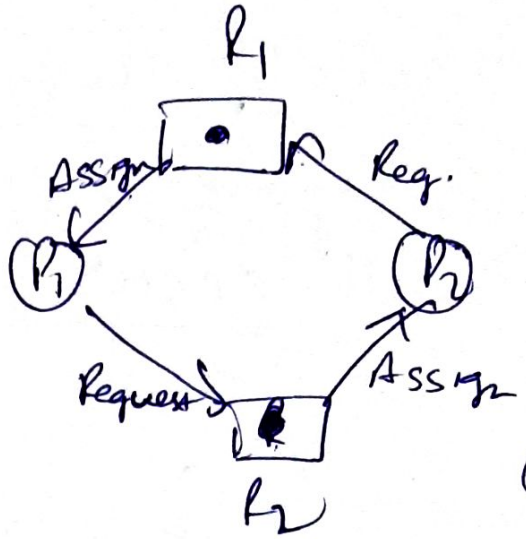
i.e if wait for graph Created a cycle [redacted], then deadlock will occur

Deadlock Recovery :

* Process termination
  - Abort all deadlocked Processes
  - Abort one Process at a time until the deadlock cycle is eliminated.
* Resource Preemption
  - Selecting a victim
  - Rollback < Partial / total
  - Starvation.
    (Finally ensure that while Selecting victim & Rollback same Process should not be selected)

Q.

Process	Allocate		Request	
	R_1	R_2	R_1	R_2
P_1	1	0	0	1
P_2	0	1	1	0



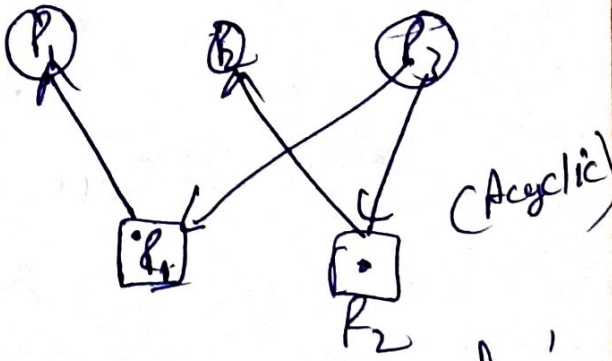
(Cycle)

Availability $\div$ $(^{R_1\ R_2}_{0,0})$ — Deadlock

So, Neither we can fulfil the request of P_1 or P_2 because Availability (0,0)

Q.

Process	Allocate		Request	
	R_1	R_2	R_1	R_2
P_1	1	0	0	0
P_2	0	1	0	0
P_3	0	0	1	1



(Acyclic)

No deadlock

Availability = (0, 0)

$P_1 P_2 P_3$ ⇒ New availability (1 0) — after P_1 gets executed

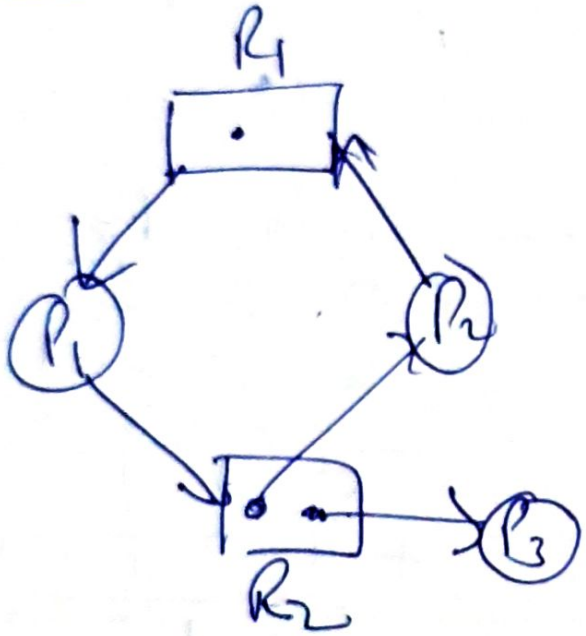
New " = (1, 1) — after P_2 gets executed

NB — If RAG has Circular wait (cycle) ⇒ Always Deadlock (True) but only in the case of single instance resource

only in case of single instance

If RAG has no cycle then no Deadlock — True
but only in case of Single instance

Multiple instances → (RAG)



	Allocation		Request	
	R_1	R_2	R_1	R_2
P_1	1	0	0	1
P_2	0	1	1	0
P_3	0	1	0	0

Current Availability
(R_1, R_2)
$(0, 0)$

$+ \begin{array}{r} 0\ 1 \\ \hline 0\ 1 \end{array}$ (P_3 get executed)

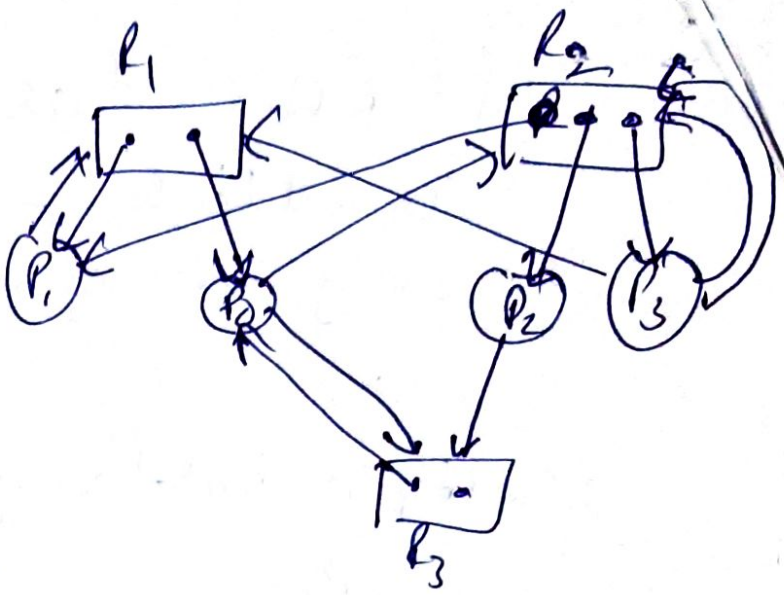
$+ \begin{array}{r} 1\ 0 \\ \hline 1\ 1 \end{array}$ (P_1 get executed)

$P_3\ P_1\ P_2$ →

no deadlock

NB, So if there is a cycle, not always deadlock
in case of Circular wait
(Multiple instance)

Q-



	Allocate			Request		
	R_1	R_2	R_3	R_1	R_2	R_3
P_0	1	0	1	0	1	1
P_1	1	1	0	1	0	0
P_2	0	1	0	0	0	1
P_3	0	1	0	1	2	0

Availability - R_1 R_2 R_3

P_2 P_0 P_1 P_3

No deadlock

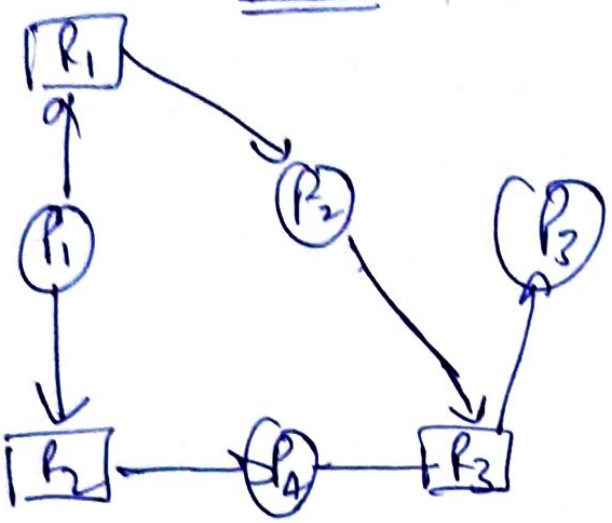
$$\begin{array}{r} (0\ 0\ 1) \\ +\ 0\ 1\ 0 \quad (P_2 \text{ get executed}) \\ \hline 0\ 1\ 1 \\ +\ 1\ 0\ 1 \quad (P_0 \text{ get executed}) \\ \hline 1\ 1\ 2 \\ +\ 1\ 1\ 0 \quad (P_1 \text{ get executed}) \\ \hline 2\ 2\ 2 \\ +\ 0\ 1\ 0 \quad (P_3 \text{ get executed}) \\ \hline 2\ 3\ 2 \end{array}$$

Curr Availability

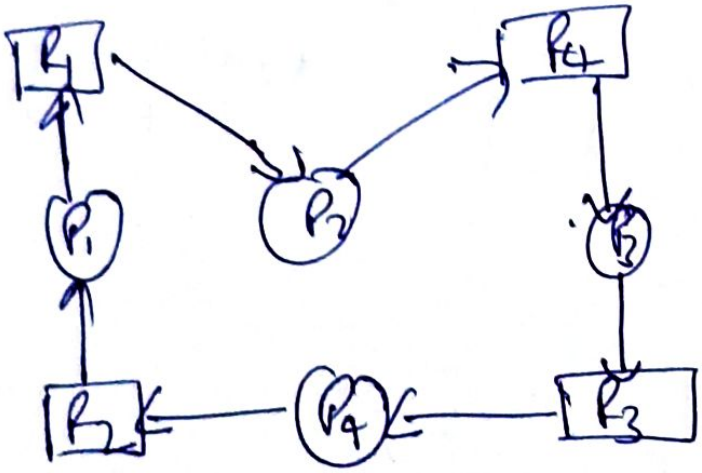
wait for Graph

RAG

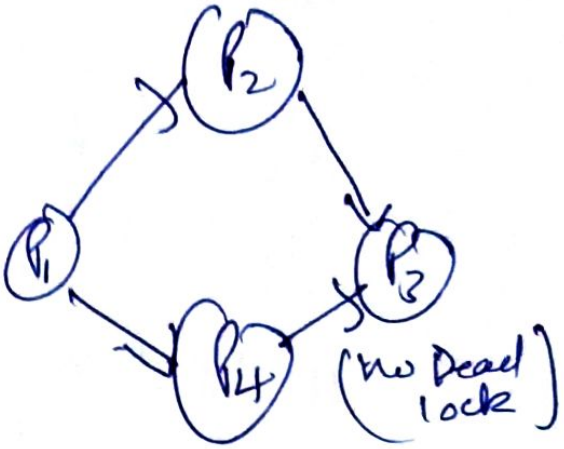
(1)



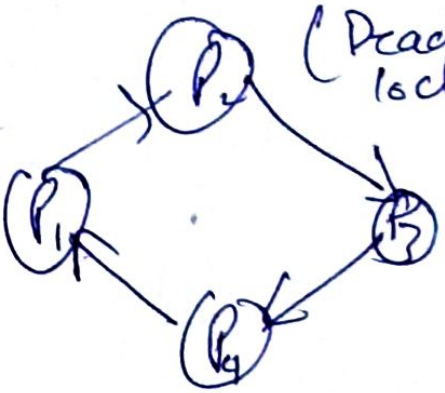
(2)



wait for graph



(No Dead lock)



(Dead lock)

Cycle

Q-

	Max A B C D	Allocation A B C D	Available A B C D
P_0	6 0 1 2	4 0 0 1	3 2 1 1
P_1	2 7 5 0	1 1 0 0	
P_2	2 3 5 6	1 2 5 4	
P_3	1 6 5 3	0 6 3 3	
P_4	1 6 5 6	0 2 1 2	

Q- ① Need matrix?
② Find if the system is in safe state?
if it is, find safe sequence
③ Determine total amount of resource of each type

Ans:
P_0, P_2, P_3, P_4, P_1

Need =

	A	B	C	D
P_0	2	0	1	1
P_1	1	6	5	0
P_2	1	1	0	2
P_3	1	0	2	0
P_4	1	4	4	4

Total amount of resources = sum of columns of allocation + Available