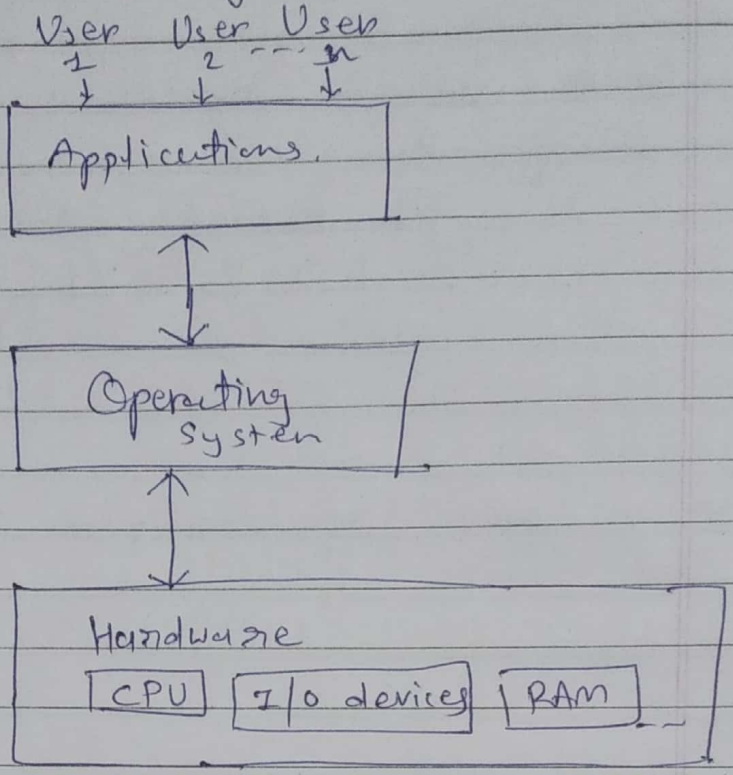
Operating System Notes

- By Rushik Pathod.

Date
Page

* Operating System Diagram :-



→ Throughput : no of tasks executed per unit time.

Ex :- Linux

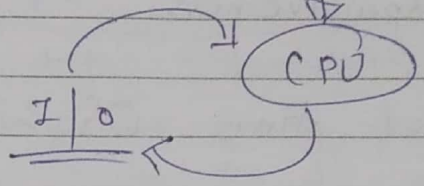
→ Functionality of OS :-

1) Resource Manager. - Parallel Processing. (Resource accessing)
2) Process Management -
   ↳ CPU Scheduling is used
3) Storage Management (HD)
   ↳ File System is used
4) Memory Management (RAM)
5) Security & Privacy

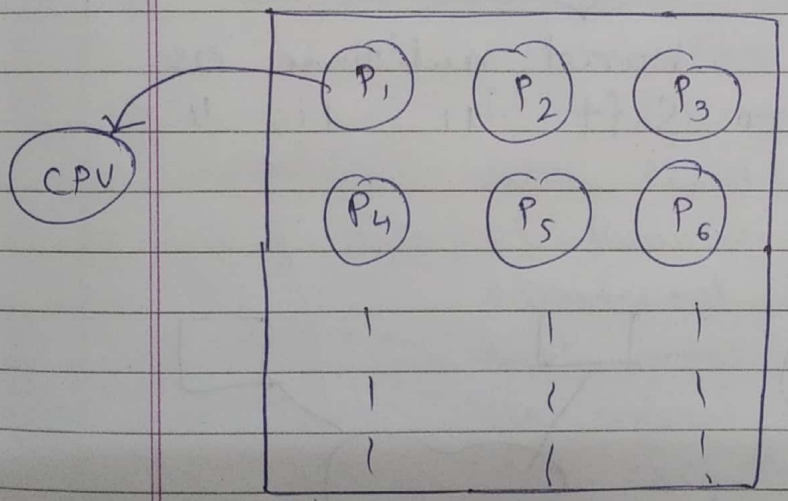
* Types of OS :-

1) Batch → Punch Cards, Paper tape, Magnetic tape → Operator
2) Multiprogrammed
3) Multi tasking
4) Real time OS
5) Distributed
6) Clustered
7) Embedded

Batches
B_1 B_2 B_3
2 3 4



* Multiprogrammed OS :-
(Non-preemptive)



→ When process is gone into CPU P_1 it will execute it completely and then CPU will fetch the next process P_2.

→ If process P_1 has to go for any other operation like I/O then CPU will become idle but in multiprogrammed CPU will fetch next process P_2.

→ Idleness ↓ of CPU.

* Multi tasking / Time sharing OS :-

→ Multi tasking OS (Preemptive) is preemptive which meeny it will execute some tasks of each process then switch to next process and it will schedule the first process in future.

→ Responsiveness ↑

* Real time OS : → Hard
→ Soft

→ delay will not be there.

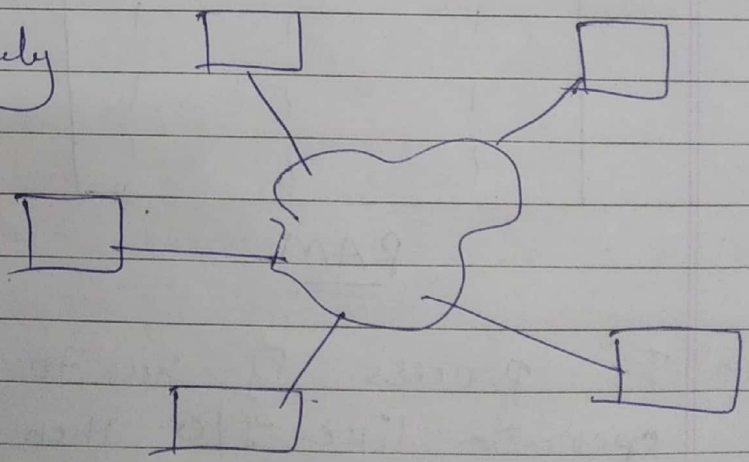
→ Ex:- Used in missile, plane, ...

Hard real time OS

Gaming live → Soft " " "

* Distributed :-

→ Environmentaly

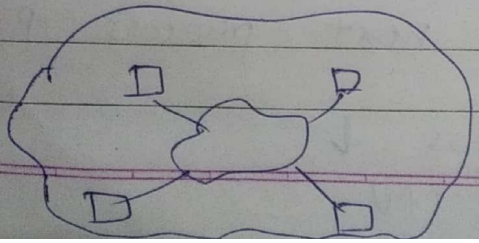


Distributed Environment

* Clustered :-

→ Availability ↑

→ Load balance

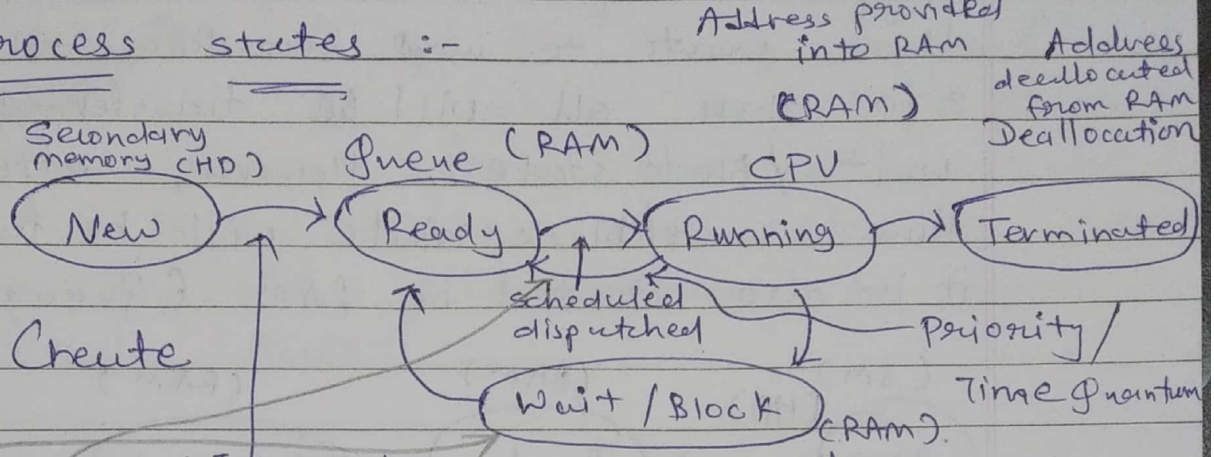


→ Computational power ↑

* Embedded :-

→ Ex :- AC, Microwave, Washing machine, ...

* Process states :-



LTS : Long term scheduler.

↳ it transfers as many processes as possible into Ready state.

↳ Uses multiprogramming concept.

STS : Short term scheduler.

↳ it transfers back some processes if high priority process comes into Ready state. The process of Running state will be transfered back to Ready state and most priority process will be moved to running state for execution.

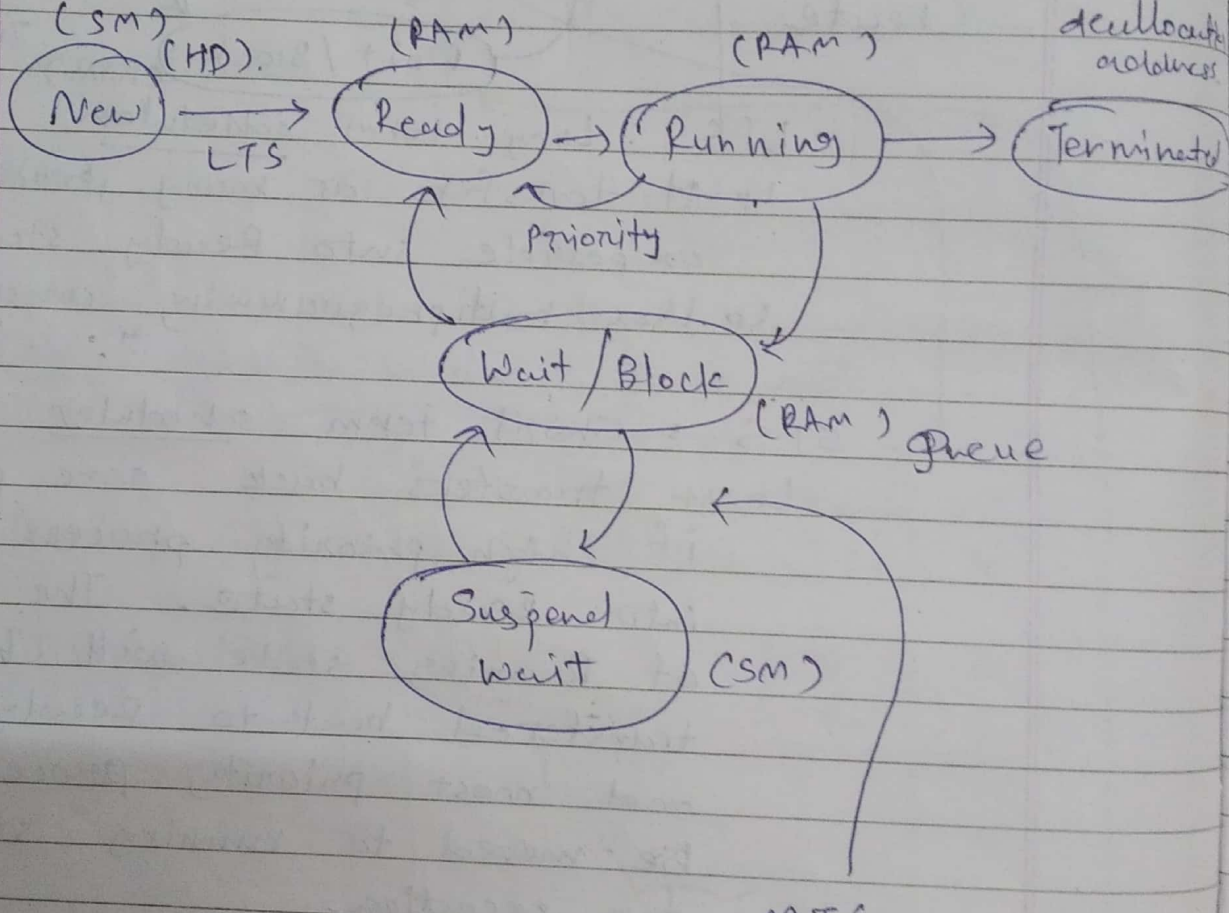
↳ it will lead to preemptive scheduling.

↳ Used multi tasking concept.

If any process demands for I/O or the duty then it will be transfered into wait / Block state.

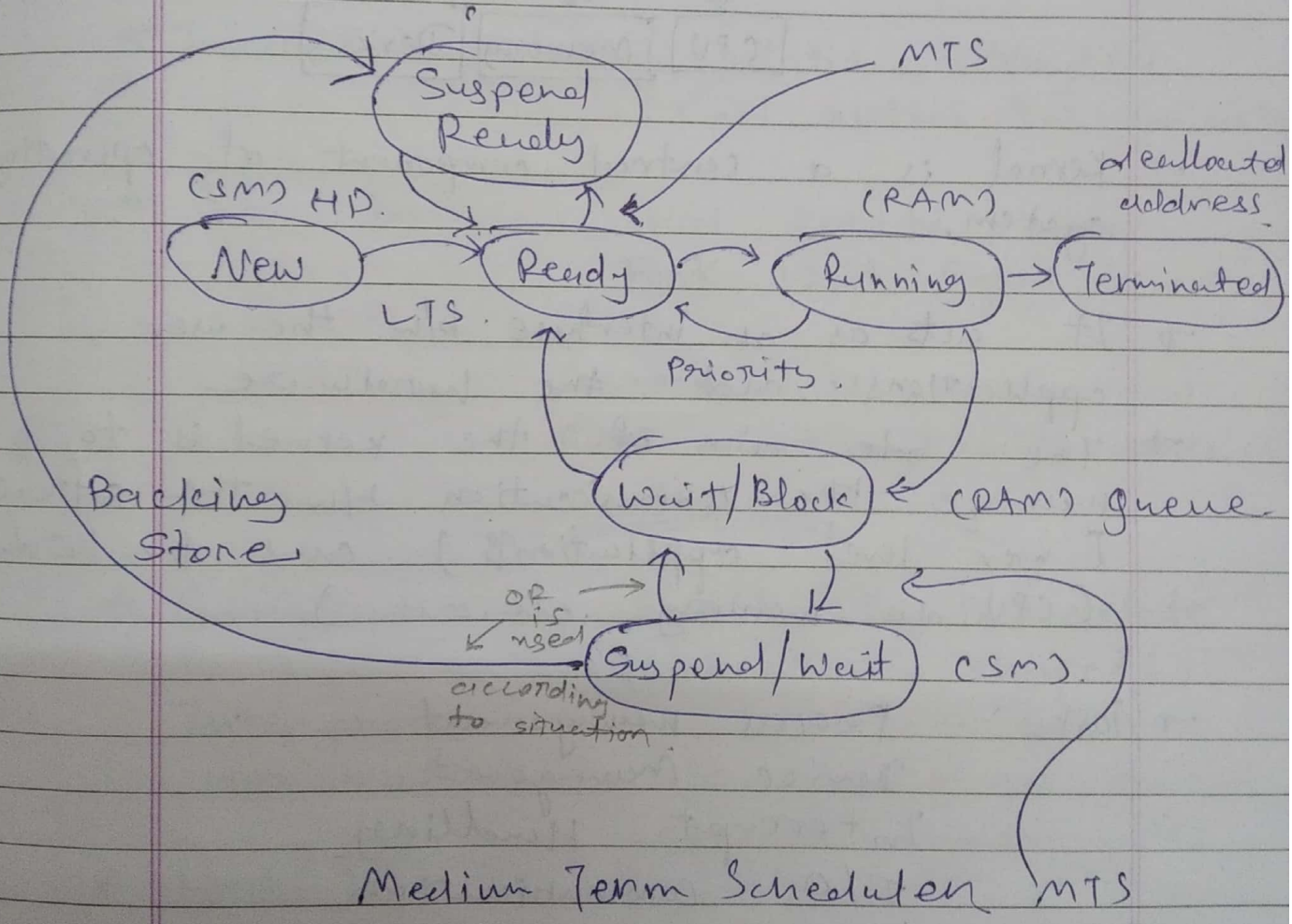
* Worst case :-

→ Suppose 10 processes are created and all of them wants to used secondary memory or I/O then all will be transferred into wait / Block state. However, after some time wait/block state will be full as it is also situated in RAM (Queue).

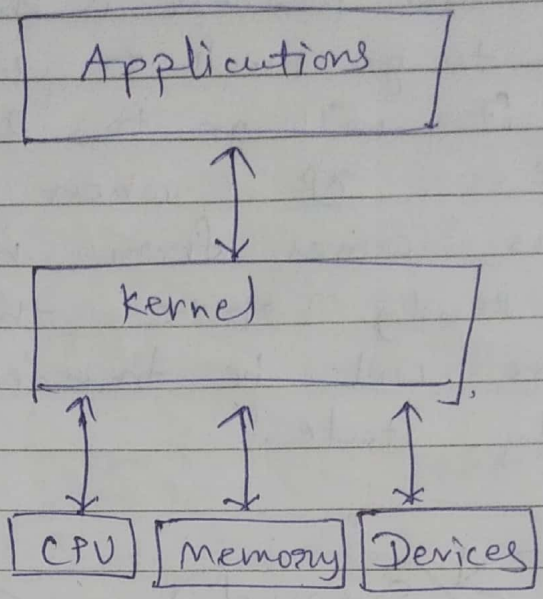


→ Medium term scheduler is used

→ if in case, suspend/wait work of any particular process is done but it is not able to go back to wait/block state then it will go to 'Ready suspend' state. OR when any high priority process comes from new create state to Ready state then already present process will be transferred to suspend Ready state.



* Kernel :-



→ Kernel is a central component of operating system.

→ It acts as an interface b/w the user applications and the hardware.

→ The sole aim of the kernel is to manage the communication b/w the software (user level applications) and the hardware (CPU, disk memory, etc.....).

→ Tasks :- Process management.
Device Management
Interrupt Handling
I/O communication
File systems, etc.....

* System calls :-

→ To switch to kernel mode from user mode.

** File related :- open(), Read(), Write(), close(), create file, etc.

** Device related :- Read, Write, Reposition, ioctl, fcntl

** Information related :- get pid, attributed, get system time and date

** Process control :- Load, Execute, abort, Fork, Wait, Signal, Allocate.
↓
Creates child process and Runs parent and child processes simultaneously.

** Communication :- Pipe(), Create / delete connections, shmget()
↑ ↓
Inter process communication to get shared memory value.

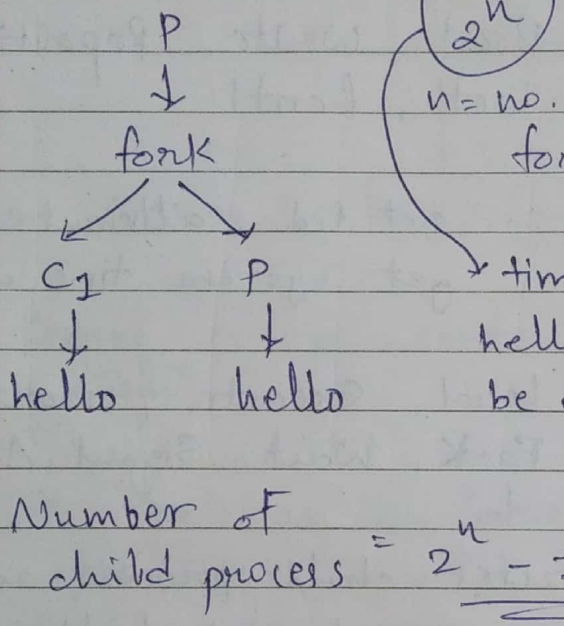
* Fork() :-

→ To create a child process.

→ Returns : 0 - child *
any positive integer → +1 - Parent / process itself.
-1 - child is not created.

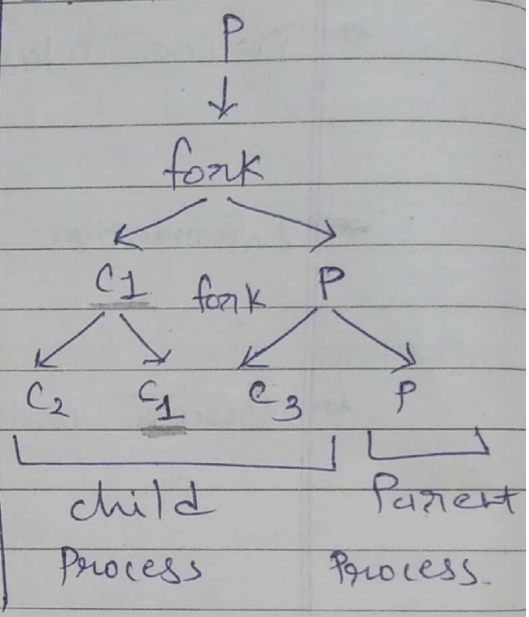
→ Ex:-

```
int main()
{
    fork();
    printf("hello");
    return 0;
}
```



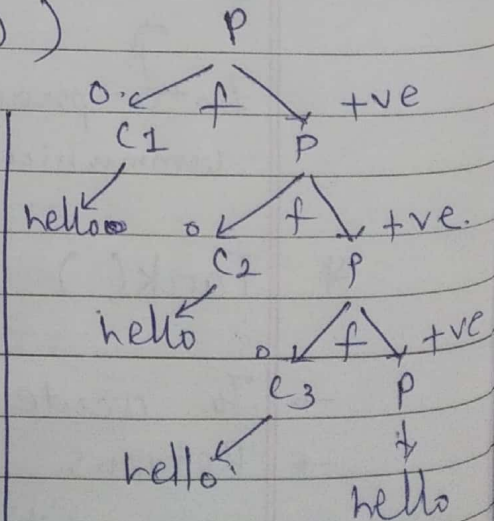
→ Number of child process $= 2^n - 1$

```
int main()
{
    fork();
    fork();
    printf("hello");
    return 0;
}
```

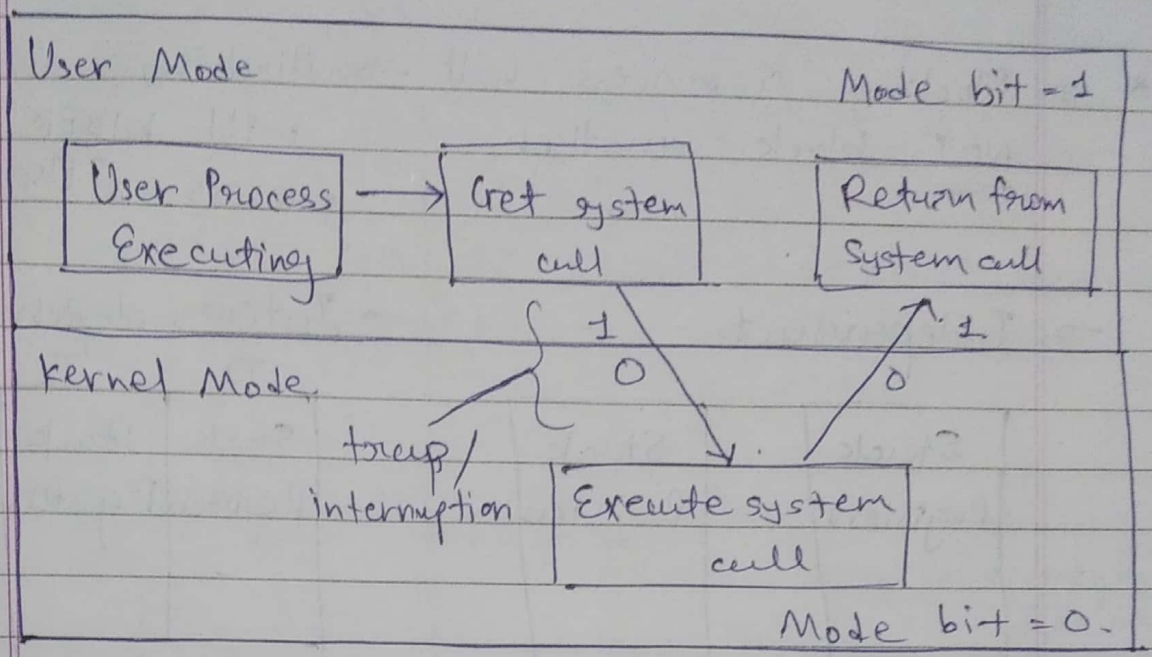


→ Ex:-

```
int main()
{
    if( fork() && fork() )
    {
        fork();
    }
    printf("hello");
    return 0;
}
```



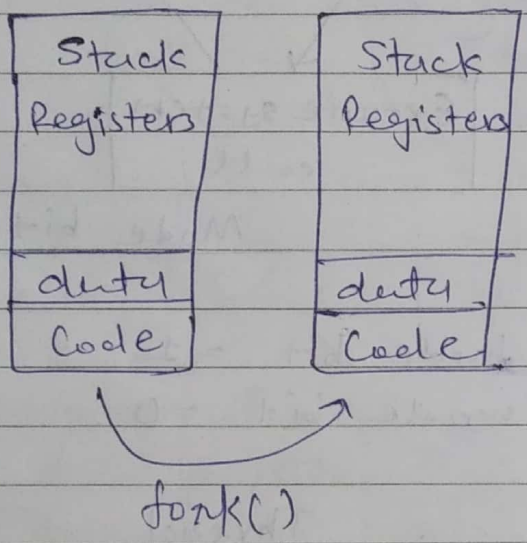
* User Mode and Kernel Mode :-

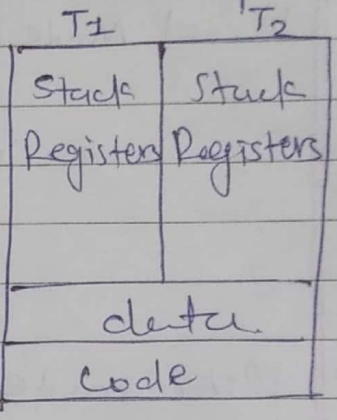


→ User mode → mode bit = 1
Kernel mode → mode bit = 0.

*	Process	Thread
→	System calls involved in process. (fork())	→ There is no system call involved. (User level)
→	OS treats different processes differently.	→ All user level threads treated as single for OS
→	Different process have different copies of stack, data, code, files.	→ Threads share same copy of code and data, (Registers & stacks will be different)

	Process	Thread
* →	Context switching is slower.	→ Context switching is faster. (User level)
* →	Blocking a process will NOT block another.	→ Blocking a thread will block entire process. (User level)
→	Independent	→ Inter-dependent





→ PCB : Process Control Block.

→ Process has to be saved before switching to the next context in case of process. Therefore, context switching is slower. While in case of thread, it is faster.

* Types of Threads :- (Light weight process)

1) User level thread
2) Kernel level thread.

*	User Level Thread	Kernel Level Thread
→	User level threads are managed by User level library.	→ kernel level threads are managed by OS system calls.
→	User level threads are typically fast.	→ kernel level threads are slower than user level.
→	Context switching is faster.	→ Context switching is comparatively slower. (∵ Works with OS system call)
→	If one user level threads perform blocking operation then the entire process gets blocked.	→ If one kernel level thread is blocked, no affects on another.

→ Context Switching Time :-

Process > KLT > ULT

KLT ↑ Kernel level thread.

ULT ↑ User level thread

* Scheduling Algorithms :-

→ It is a way of selecting a process from Ready queue and put in on the CPU.

→ Pre-emptive :- If a process P_1 is transferred into running state from Ready state. When a high priority process P_2 comes into Ready state then CPU will transfers the process (currently executing) P_1 into Ready state and will fetch high priority process P_2 and execute it. This is called pre-emptive.

→ Non-pre-emptive :- When process P_1 is in running state it will be executed completely. If in case P_1 wants any I/O or data from HD then it will go to the wait/Block state and simultaneously CPU will fetch process P_2 and will start executing it. as it is based on multi programming. After it P_1 has fetched the data then it will go back to running state and will be terminated after executing. Non-pre-emptive process once comes in running state then it will NOT go back to Ready state.

Scheduling Algorithms

Pre-emptive	Non-pre emptive
★→ Shortest Remaining Time First [SRTF]	→ FCFS (First Come first serve)
→ Longest Remaining Time First [LRTF]	→ SJF (Shortest Job First)
★→ Round Robin	→ LJF (Longest Job First)
→ Priority Based.	→ HRRN [Highest Response Ratio Next)
	→ Multilevel Queue.

✱ CPU Scheduling :-

→ Arrival time : The time at which process enters the Ready state / queue.

→ Burst time : Time required by a process to execute on CPU.

→ Completion time : The time at which process completes it's execution.

→ Turn Around Time = { Completion Time − Arrival Time }

→ Waiting Time = { Turn Around Time − Burst Time }

→ In Non-preemptive : WT = RT.
(Waiting Time = Response Time)

→ Response Time = The time at which a process gets CPU first time − Arrival time

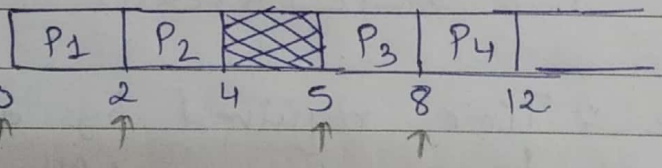
= The time at which a process gets CPU first time − Arrival Time.

✱ FCFS : First Come First Serve

Criteria: Arrival Time (Non-preemptive) (WT = RT)

Process No.	Arrival Time	Burst Time	Completion Time	TAT	WT	RT
1	0	2	2	2	0	0
2	1	2	4	3	1	1
3	5	3	8	3	0	0
4	6	4	12.	6	2	2

The time at which process gets CPU first time is situated on left



Avg TAT = 14/4 = 3.5

Avg. WT = 3/4 = 0.75

→ (TAT) Turn Around Time = CT − AT
= Completion time − Arrival time.

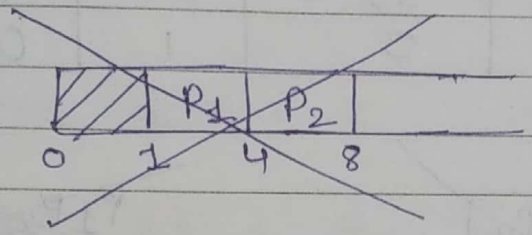
→ (WT) Waiting Time = TAT − Burst Time

→ (RT) Response Time = The time at which a process gets CPU first time − Arrival time.

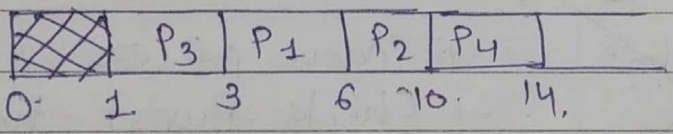
* SJF : Shortest Job First

Criteria : Burst time (Non-preemptive)

Process No.	Arrival Time	Burst Time	Completion Time	TAT	WT	RT
P_1	1	3	6	5	2	2
P_2	2	4 min	10	8	4	4
P_3	1	2	3	2	0	0
P_4	4	4	14	10	6	6



(∵ Burst time is criteria.)



P_1 P_3 P_2 P_4

Process Burst time is same.
∴ go for arrival time and put accordingly.

Avg. TAT = $\frac{25}{4}$ = 6.25

Avg. WT = $\frac{12}{4}$ = 3.

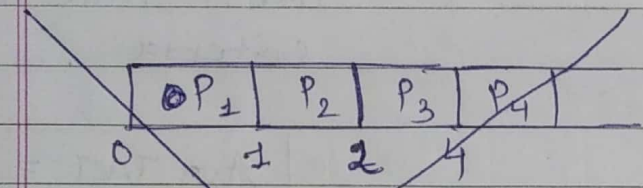
→ TAT = CT − AT. = Completion Time − Arrival Time.

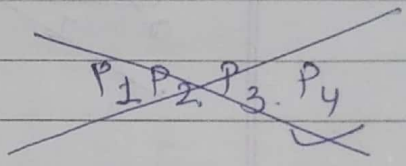
→ WT = TAT − BT = Turn Around Time − Burst Time

→ RT = The time at which Process gets CPU First time. − Arrival Time

* Shortest Remaining Time First :-
SRTF (Pre-emptive)
(Shortest Job First - pre-emptive = SRTF)
Criteria : Burst Time.

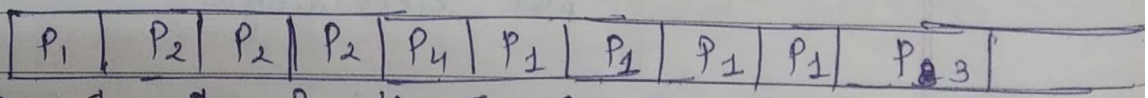
Process No.	Arrival Time	Burst Time	Completion Time	TAT	WT	RT
P_1	0	5	9	9	4	0
P_2	1	3	4	3	0	0
P_3	2	4	13	11	7	7
P_4	4	1	5	1	0	0





Burst time
P_1 ~~4~~
P_2 ~~2~~
P_3 ~~4~~ 2
P_4 1

→ Move process 1 step each time.
→ Check Burst time each time. and select the process which has minimum Burst Time.



Process	Burst Time
P_1	~~5~~ ~~4~~ ~~3~~ ~~2~~ ~~1~~ 0
P_2	~~3~~ ~~2~~ ~~1~~ 0 ← min.
P_3	4
P_4	~~1~~ 0

last value should be checked when we want to calculate completion time.
Here, P_1 is completed at 9.

→ TAT = CT - AT
= Completion Time - Arrival Time.

→ WT = TAT - BT
= Turn Around Time - Burst Time.

→ RT = The time At which process gets CPU first time - Arrival time.

→ Avg. TAT = $\frac{24}{4}$ = 6.

→ Avg. WT = $\frac{11}{4}$ = 2.75

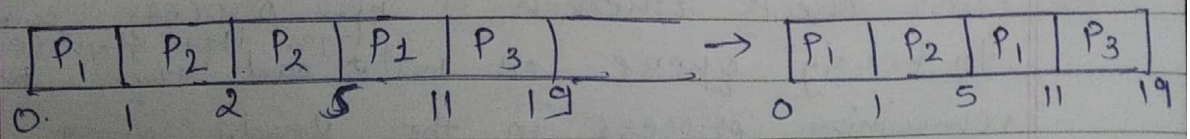
→ Avg. RT = $\frac{7}{4}$ = 1.75

Q.

Process No.	Arrival time	Completion time	TAT	WT	RT
P_1	0	11	11	4	0
P_2	1	5	4	0	0
P_3	2	19	17	9	9

→ Shortest Job First - Pre-emptive
(a) SRTF.

Process No.	Burst time
P_1	7
P_2	4
P_3	8



* Round Robin Scheduling :-

Pre-emptive

Criteria -> Time Quantum

→ Given time quantum = 2.

Process No.	Arrival Time	Burst Time	Completion Time	TAT	WT	RT
P_1	0	5	12	12	7	0
P_2	1	4	11	10	6	1
P_3	2	2	6	4	2	2
P_4	4	1	9	5	4	4

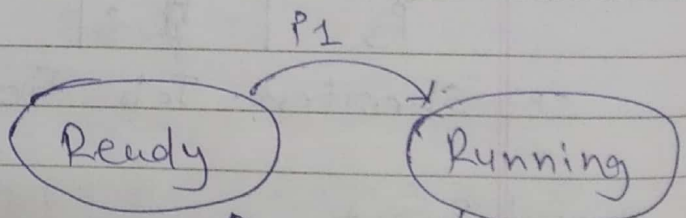
Ready queue | ~~P_1~~ | ~~P_2~~ | ~~P_3~~ | ~~P_1~~ | ~~P_4~~ | ~~P_2~~ | ~~P_1~~ |

Running queue | P_1 | P_2 | P_3 | P_1 | P_4 | P_2 | P_1 |

0 2 4 6 8 9 11 12

BT of P_4 is 1
∴ only 1 unit is allocated.

Process No.	Burst Time
P_1	~~5~~ ~~3~~ ~~1~~ 0
P_2	~~4~~ ~~2~~ 0
P_3	~~2~~ 0
P_4	~~1~~ 0



Context Switching

P_1 after 2 time quantum

→ Sequence of processes is important.

→ First check whether a new process can come to Ready queue or not → if yes then transfer it first and then move remaining process in the Ready queue.

→ TAT = CT − AT = Completion Time − Arrival Time

→ WT = TAT − BT.

→ RT = CPU gets process − AT.

→ Avg. TAT = 31/4 = 7.75.

→ Avg. WT = 19/4 = 4.75.

→ Avg. RT = 7/4 = 1.75.

→ No. of Context Switching = 6.
(Do not calculate the first and last lines. Count only of in b/w. lines).

* Qve:- Time Quantum = 3

Process No.	Arrival Time	Burst Time	Completion Time	TAT	WT	RT.
P_1	0	8	23	23	15	0.
P_2	1	4	16.	15	11	2
P_3	2	9	26	24	15	4
P_4.	3	5	21	18	13	6.

Ready Queue: P_1 | P_2 | P_3 | P_4 | P_1 | P_2 | P_3 | P_4 | P_1 | P_3

Running Queue: P_1 | P_2 | P_3 | P_4 | P_1 | P_2 | P_3 | P_4 | P_1 | P_3

0 3 6 9 12 15 16 19 21 23 26

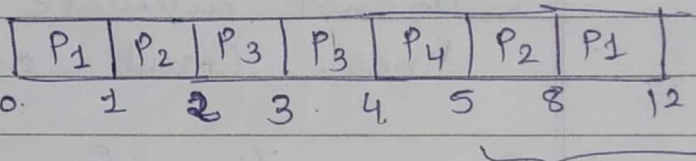
Process	BT
P_1	8 5 2 0.
P_2	4 1 0.
P_3	9 6 3
P_4.	5 2 0

* Priority Scheduling :- Criteria : Priority
Pre-emptive

Priority	Process No.	Arrival Time	Burst Time	Completion Time	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

→ Given : Higher the number, Higher the priority.

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



Lowly P_2 has the higher priority in both of them (P_2 & P_1)
∴ 5 to 8 and 8 to 12

→ TAT = CT - AT.
→ WT = TAT - BT.

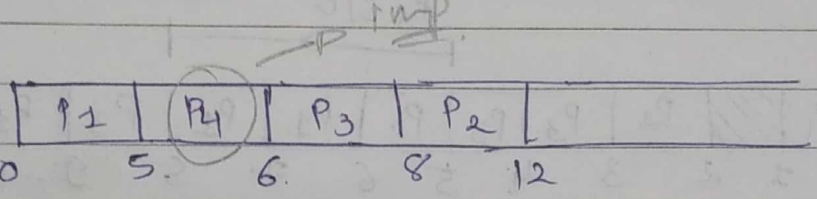
→ Run each process one time only and check whether any process arrived in Ready queue or not. if yes then check the priority and execute which has the higher priority.

→ If two or more processes having same priority and same arrival time then choose $P_1, P_2, \ldots P_n$ n - id wise.

* Priority Scheduling :- Criteria : Priority
Non- pre-emptive.

Priority	Process No.	Arrival time	Burst time	Completion time	TAT	WT
10	P_1	0	5	5	5	0.
20	P_2	1	4	12	11	7.
30	P_3	2	2	8	6	4.
40	P_4	4	1.	6	2.	1.

→ Given :- Higher the number ⇒ Higher the priority.



Process	B.T
1	~~5~~ 0.
2	~~4~~ 0.
3	~~2~~ 0.
4.	~~1~~ 0.

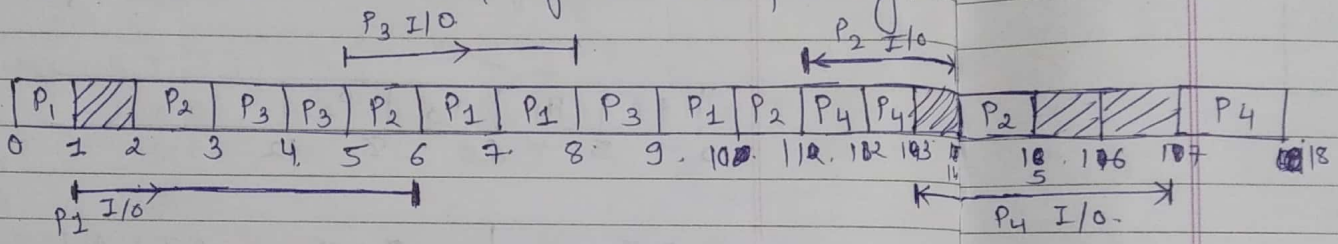
TAT = CT - AT.
WT = TAT - BT

→ In this case when Process P_1 is executing it executes completely as it is non-preemptive and simultaneously processes P_2, P_3 and P_4 comes into ready queue. Now, go according to the priority to arrange the sequence of processes.

* Mix Burst Time :- Criteria: Priority
Pre-emptive

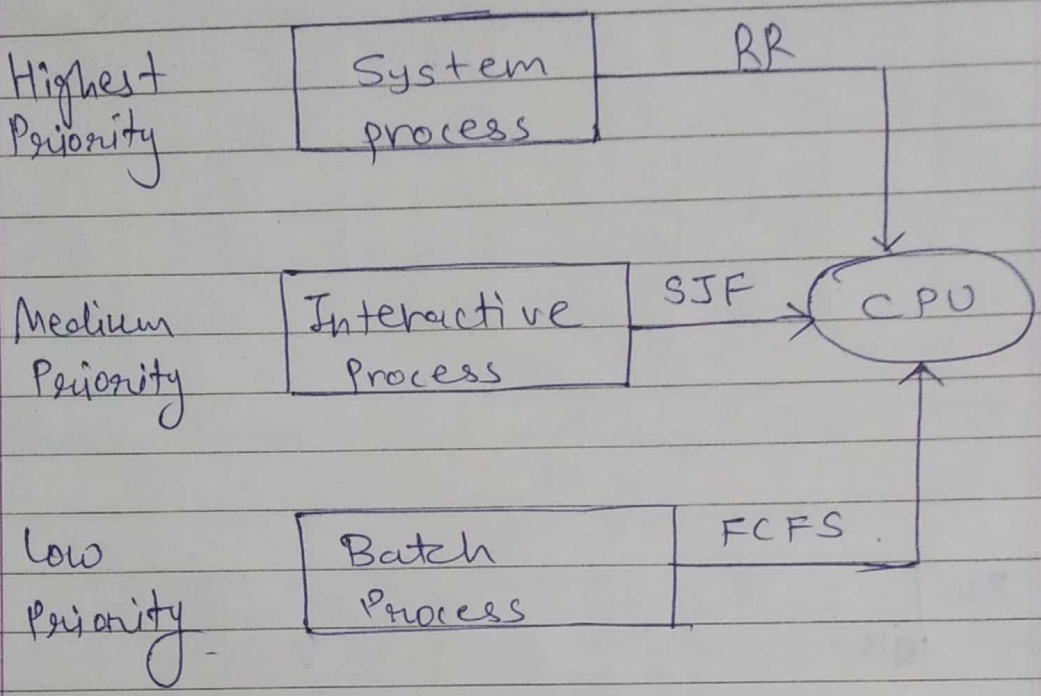
Process No.	Arrival Time	Priority	CPU	I/O	CPU	Completion Time
P_1	0	2	1	5	3	
P_2	2	3	3	3	1	
P_3	3	1	2	3	1	
P_4	3	4	2	4	1	

→ Given :- lowest the number, higher the priority.



Process	B.T/CPU	I/O	CPU
P_1	1 0	5	3 2 1 0.
P_2	3 2 1 0.	3	1 0.
P_3	2 1 0.	3.	1 0.
P_4	2 1 0	4	1 0.

* Multilevel Queue Scheduling :-

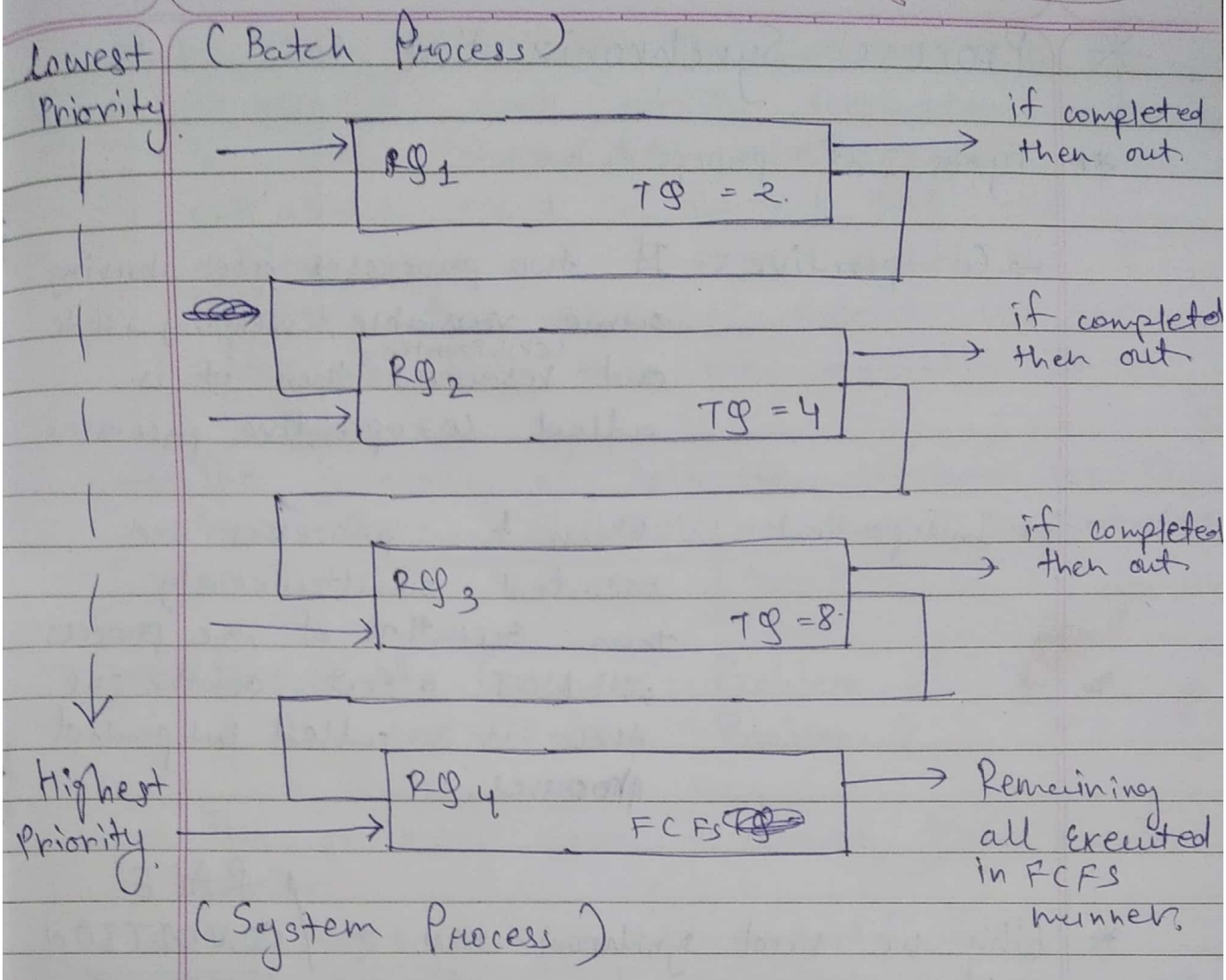


→ Problem :- When the highest priority processes will be there in system processes then other two processes have to wait for their turn to complete on execute.

→ This is called starvation problem.

* Starvation problem can be solved using Multilevel Feedback Queue Scheduling :-

→ Multilevel feedback queue scheduling is used for lower priority processes which further will be sent to next level after processing for a particular time quantum.



→ Starvation problem of implementing multilevel queue scheduling can be solved using multilevel feedback queue scheduling.

* Process Synchronization :-

** Types of processes :-

→ Co-operative :- If two processes are sharing same variable, memory, code and resources (CPU, printer, ..) then it is called co-operative processes.

→ Independent :- When two processes are executed simultaneously then execution of one process will NOT affect another one then it is called independent processes.

* Why we need synchronization ? / RACE CONDITION

int shared = 5 ~~6~~ 4

P_1	P_2
int x = shared ; (x = 5)	int y = shared ; (y = 5)
x++ ; (x = ~~5~~ 6)	y-- ; (y = ~~5~~ 4)
sleep(1) ;	sleep(1) ;
shared = x ;	shared = y ;
shared = 6	shared = 4

After executing P_1

→ In this case, both the processes are co-operative but while concluding both runs simultaneously of them shared ++ and shared -- and should result in shared the original value of shared should be 5. However, the value of shared we get after executing both the processes is 4.

→ The processing of both the processes results in Race Condition as there is no synchronization

* Producer Consumer Problem :- (Bounded Buffer Problem)

→ Two processes share a common fixed sized buffer.

→ The producer put the product / information in the buffer and consumer takes it out.

→ Trouble arises when the producer want to put a new item in the buffer but is already full.

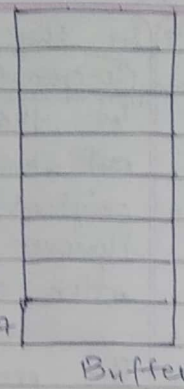
→ The solution is : the producer go to sleep to be awakened when the consumer has removed one or more items.

→ Similarly, if the consumer wants to remove item and see that buffer is empty, it goes to sleep until the producer has put the something in the buffer and wake it up.

/*

```
int count = 0 ;

void consumer (void)
{
    int itemC;
    while (true)
    {
        while ( count == 0 ) ;
        itemC = Buffer (out) ;
        out = (out + 1) mod n ;
        count = count - 1 ;
        Process_item (itemC) ;
    }
}
```



Buffer

*/

/*

```
void producer (void)
{
    int itemP ;
    while (true)
    {
        while ( count == n ) ;
        ~~itemP = Buffer [in] ;~~
        Buffer [in] = itemP ;
        in = (in + 1) mod n ;
        count = count + 1 ;
    }
}
```

*/

Sol^{n}:- Simple solution

```
int count = 0 ;   // no of items
int n = 8 ;       // no of slots in buffer

void producer (void)
{
    int itemP ;   int in = 0 ;
    while (true)
    {
        if ( count == n )   // buffer is full
            sleep() ;       // go to sleep
        Buffer[in] = itemP ;
        in = (in + 1) mod n
        count = count + 1 ;
        if (count == 1)   // buffer has item
            wakeup (consumer) ;
    }
}
```

```
void consumer (void)
{
    int itemC ;  int out = 0 ;
    while (true)
    {
        if ( count == 0 )   // buffer is empty
            sleep() ;       // go to sleep
        itemC = Buffer [out]
        out = (out + 1) mod n ;
        count = count - 1 ;
        if ( count == n-1 )   // buffer is full
            wakeup ( producer ) ; // wakeup producer
        Consume item (itemC) ;
    }
}
```

* Printer Spooler Problem :-

I_1	Load R_i, m[in]
I_2	Store SD[R_i], "F-N" ← filename
I_3	INCR R_i
I_4	Store m[in], R_i

→ Spooler directory :-

0	f_1.doc
1	f_2.doc
2	f_3.doc
3	~~f_4.doc~~ f_5.doc
4	
⋮	

* Problem occurs in this case :-

→ P_1 I_1 I_2 I_3 | P_2 I_1 I_2 I_3 I_4

Preempt

I_4 ← of P_1

P_1 → f_4.doc

P_2 → f_5.doc

In : ~~3~~ ~~4~~ 4

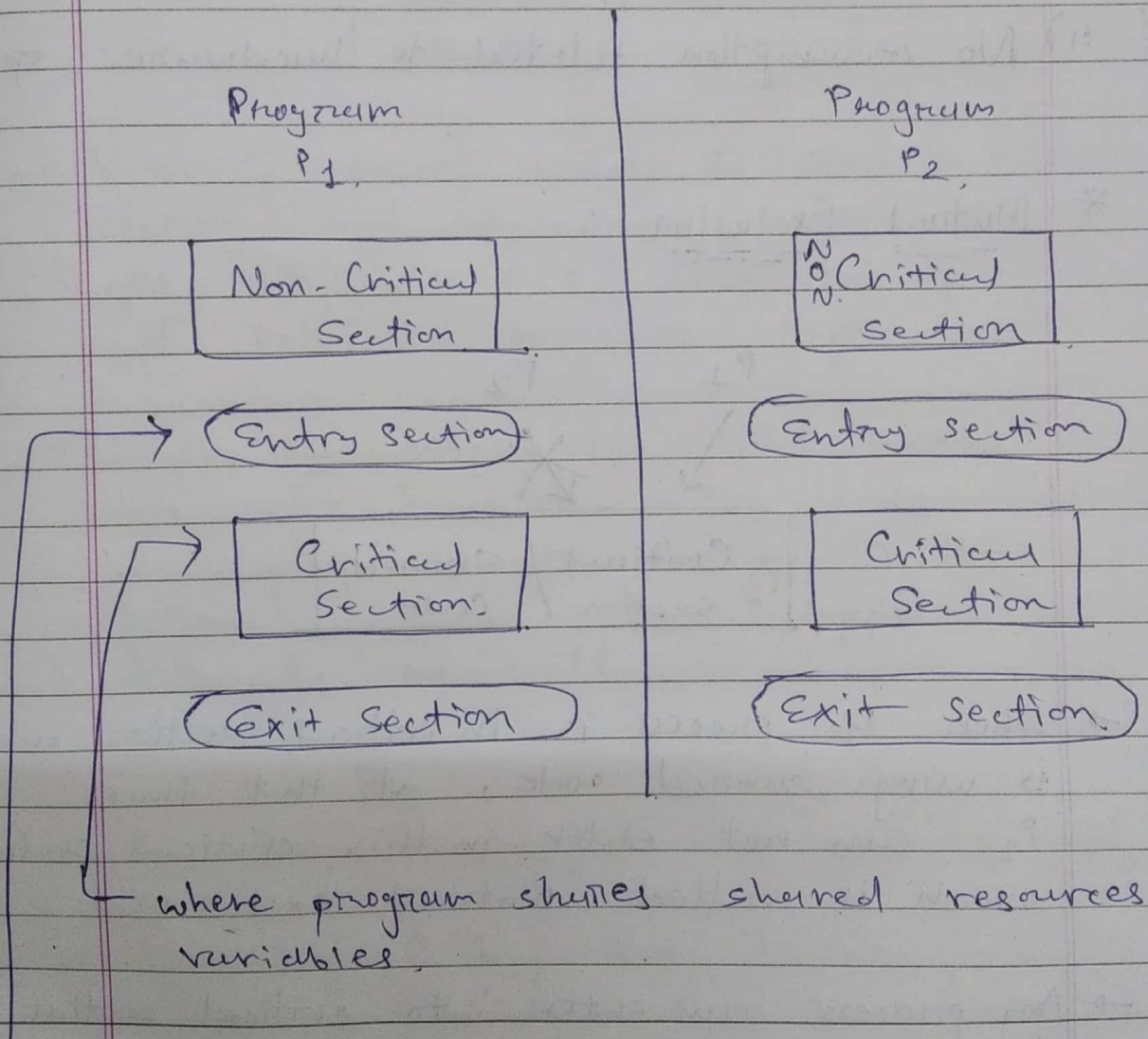
R_1	R_2
~~3~~ 4	~~3~~ 4

→ Problem occurs when P_1 process preempts and P_2 process starts executing. At that time in spooler directory f_4.doc will override with f_5.doc of process P_2. Which results in data loss.

* Critical Section Problem, Mutual Exclusion :-

* Critical section :-

It is a part [variables] of a program where shared resources are accessed by various processes (Co-operative processes).
↳ Runs simultaneously.

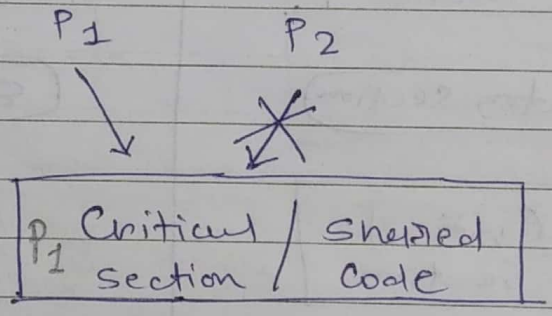


* Conditions to write synchronization mechanism:

Primary
1) Mutual Exclusion (Entry Condition of C.S.)
2) Progress

Secondary
3) Bounded wait
4) No assumption related to hardware, speed.

* Mutual Exclusion :-



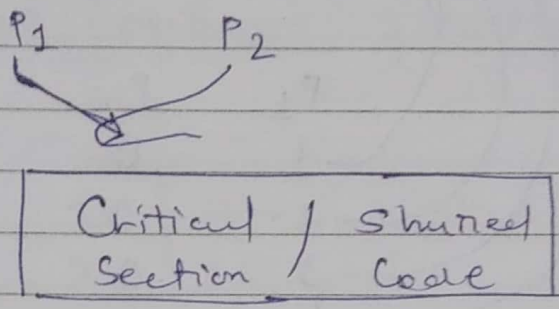
→ When P_1 process is in critical section and is using shared code, at that time P_2 can not enter in this critical section, which is called mutual exclusion.

→ Any process can enter into critical section first but when one is running in critical section, another can NOT enter.

Date
Page

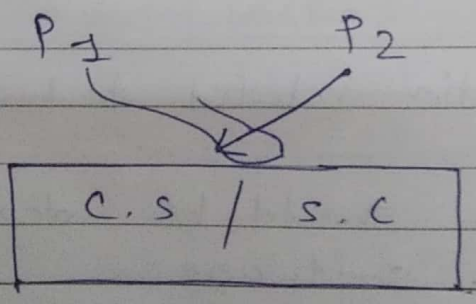
* Progress :-

→ To write synchronization mechanism there must be progress in executing the processes.



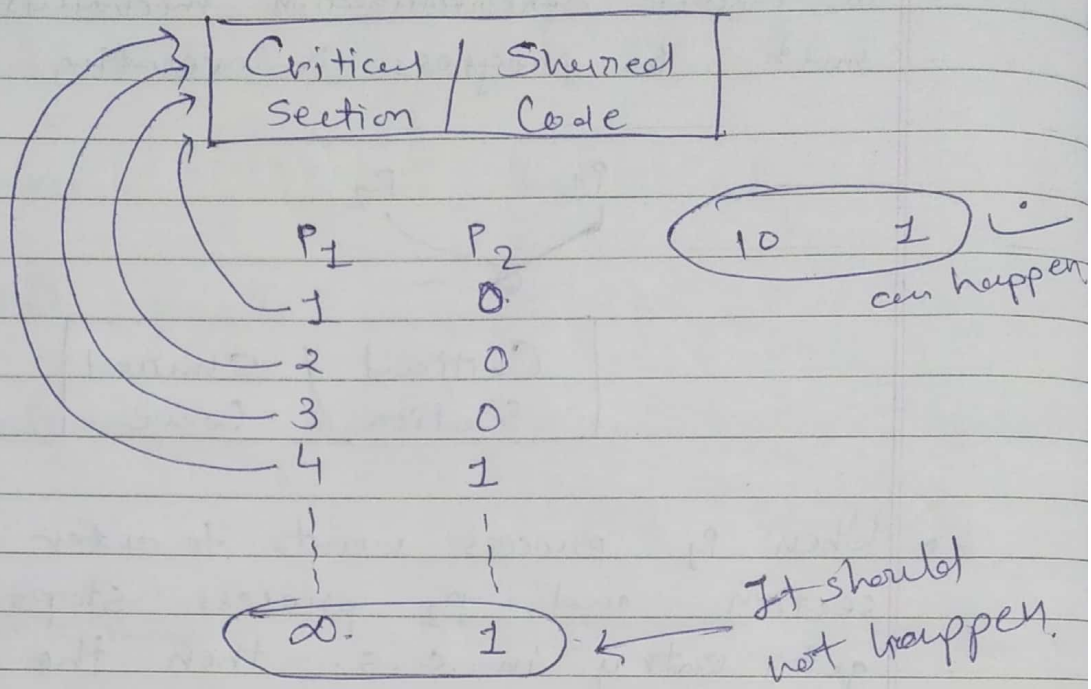
→ When P_1 process wants to enter in critical section and P_2 process stops P_1 to get entry in c.s. then the progress of both the processes stops.

→ When P_1 stops P_2 that means P_1 has some kind of entry code which is stoping P_2 to enter in the critical section. Same applies for P_1 as well.



→ In this case P_1 is stoping P_2 to enter in critical section.

* Bounded Wait :-



→ When P_1 process enters into critical section for multiple times ∞ and process P_2 waits for it's turn to enter into it. But P_1 enters into C.S ∞ times and P_2 enters for just 1 or time. This condition should not be there.

* No assumption related to hardware, speed :-

→ No solution should be dependent on hardware, software.

→ A solution should be applicable to all the hardware and software (regardless of their versions.) and compatible with

→ Universal sol^n should be there.

* LOCK Variable :-

Critical Section Solution using LOCK var:-

→ Execute is user mode
→ Multi process solution
→ No mutual exclusion gurantee

Entry Code.

```
1. while ( LOCK == 1 );
2. LOCK = 1.
3. Critical section
4. LOCK = 0
```

→ LOCK = 0 ⇒ vacant (C.S.)
LOCK = 1 ⇒ full (C.S.)

→ Case : 1 : P_1 P_2 LOCK = ~~0~~ ~~1~~ ~~0~~ ~~1~~ 0.
CS = ~~P1~~ ~~P2~~.

P_1 I_1 I_2 I_3 I_4 | P_2 I_1 I_2 I_3 I_4

★ → Case : 2 :

P_1 P_2

Preempts P_1

```
1. while (LOCK == 1);
2. LOCK = 1
3. Critical Section
4. LOCK = 0
```

P_2 P_1
LOCK : ~~0~~ ~~1~~ 1
CS : P_2

P_1 P_2
I_1 → I_1
I_2 ← I_2
I_3 I_3

→ In this case, both the processes are in Critical section.
∴ Mutual exclusion fails.

* Test and Set :- (TSL instruction)
Critical Section Solution using Test-and-Set instruction :-

→ Merged in a single instruction

```
while ( TSL (&LOCK)) ;
```

```
1. while (LOCK == 1) ;
2. LOCK = 1
3. Critical section
4. LOCK = 0
```

```
CS
LOCK = false;
```

TSL or

```
boolean test_and_set ( boolean * target )
{
    boolean rv = *target ;
    *target = TRUE ;
    return rv ;
}
```

Initially → LOCK: false (1000) ← target: 1000 ; rv: false.

P_1 : LOCK: TRUE (1000) ← target: 1000. ; rv: ~~1000~~ false

if rv returns false that means CS is empty and P_1 can enter into Critical section.

P_2 : LOCK: TRUE ← target: 1000 ; rv: true.

P_2 will fall in infinite loop.

→ Problems : Busy Waiting

* Strict Alteration Method :-

Turn Variable :-

(To achieve Mutual Exclusion and to solve the problem critical section.)

- 2 process solution :
- User mode
- Mutual Exclusion gurantee.
- Progress problem occurs.
- Bounded waiting (infinite waiting) will NOT happen
- Initially turn = 0.

	Process P_0	Process P_1
	Non - critical section	Non - critical section
Entry code	while (turn != 0);	while (turn != 1);
	Critical Section	Critical Section
Exit code	turn = 1	turn = 0.

PCB : Process Control Block.

Date
Page

* Semaphore :-

→ Semaphore is an integer variable which is used in mutual exclusive manner by various concurrent co-operative processes in order to achieve synchronization

Semaphore
- Counting: $-\infty$ to ∞
- Binary: (0, 1)

P_1 P_2 P_3

p() , down, wait ← Entry code
↓
v(), up, signal ← Critical section / Exit code

* Entry level code :-

Counting Semaphore

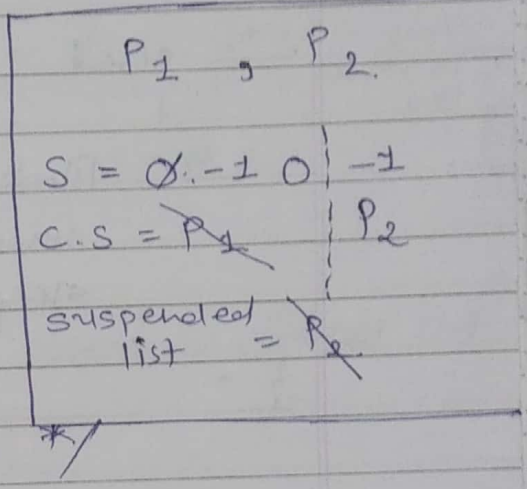
```
Down ( Semaphore S)
{
    S.value = s.value -1 ;
    if (S.value < 0 )
    {
        put process (PCB) in suspended list,
        sleep() ;
    }
    else
    {
        return ;
    }
}
```

P => - Down
V => + Up

Date
Page

* Exit level code :-

```
Up ( Semaphore S)
{
    S.value = S.value + 1;
    if (S.value ≤ 0 )
    {
        select a process from suspended list
        and wakeup();
    }
}
```




```
struct Semaphore
{
    int value;
    queue type list;
}
```

* Que :-

→ S = 0 => No. of processes in suspended list = 0

→ S = 10 => How many processes can enter into critical section ? = 10

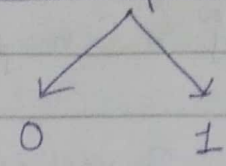
→ S = 10 , 6P , 4V.

→ Here, P = down, V = up.

∴ 10 − 6 + 4 = 8

* Entry level code :-

Binary Semaphore




```
Down ( Semaphore S)
{
    if ( s.value == 1)
    {
        s.value = 0 ;
    }
    else          // if (s.value == 0)
    {
        Block this process and place it in
        suspended list.
        sleep();
    }
}.
```

* Exit level code :-

```
Up ( Semaphore s )
{
  if ( suspended list is empty )
  {
      s.value = 1 ;
  }
  else  // is( suspended list is NOT empty )
  {
      Select a process from suspended list and
      wakeup();
  }
}
```

Put it in Ready queue.

Date
Page

Processes enters

Down → P_1 S = 1
↳ S = 0, enter into critical section.

P_2 S = 0
↳ S = 0, Block this process and place it in suspended list.

| S = 0 → 1
| S = 1 → 1

Up → S = 0
↳ ∴ suspended list is empty, S = 1.

Up is not dependent on the value of S. It will just check whether 'suspended list' is empty or not.

S = 1
↳ ∴ suspended list is NOT empty, S = 1
↓
select a process from suspended list

→ Initialize S with 1 always.

Que:- What is maximum no. of processes that may present in critical section at any point of time?

→ Each process $P_i = \{i = 1 \text{ to } 9\}$ executes following code.

repeat
Entry code → P(mutex)
[C.S.]
Exit code → V(mutex)
forever.

→ Process P_{10} executes following code.

repeat
V(mutex)
[C.S]
V.(mutex)
forever

Soln:- → Case 1 : mutex : ✗ 0

Critical section = P_1 (P_2 will not enter in C.S)

→ Case : 2 : mutex : ✗ ∅ ✗ ∅ ✗ ∅ ✗ ∅ ✗ ∅ ✗ ∅ ✗

C.S. = P_1 P_{10} P_2 P_3 P_{10} P_4 P_5 P_{10} P_6 P_7 P_{10} P_8 P_9

suspended list = P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9.

∴ Maximum processes that may present = 10.

** Producer - Consumer Problem's Solution

Using Binary Semaphore :-

→ Binary Semaphore S = 1

Empty slots = 5
full slots = 3.

out = 0
in = 3

n = 8

0	a
1	b
2	c
3	
4	
5	
6	
7	

Buffer

→ Producer (void)
{
produce-item (itemP)
1) down (empty)
2) down (s)

```
Buffer [In] = itemp
In = (In+1) mod n
```

3) up (s)
4) up (full).

→ Consumer (void).
{
1) down (full)
2) down (s)

```
itemC = Buffer [out];
out = (out+1) mod n;
```

3) up (s)
4) up (empty).

* Actual code :-

```
# define N 100.
typedef int semaphore;
semaphore mutex = 1
semaphore empty = N.
semaphore full = 0.

void producer (void)
{
    int itemP;
    while (true)
    {
        itemP = produce_item();

        down (&mutex)
        down (&empty)
Critical section → insert_item (itemP);
        up (&mutex)
        up (&full)
    }
}

void consumer (void)
{
    int itemC;
    while (true)
    {
        down(&mutex)
        down(&full)
```

no of empty slots in buffer.

no of full slots in buffer

```
    itemC = remove_item();
    up(&mutex)
    up(&empty)
    consume_item(itemC);
  }
}
```

* Mutex :-

→ When semaphore's ability to count is not needed, a simplified version of the semaphore is called mutex.

→ Mutex are good only for mutual exclusion to shared some resources.

→ This is specially useful in thread packages that are implemented entirely in user space.

→ A mutex variable can be one of two states :-

1) Unlocked 2) locked.

→ 0 → unlocked
1 → locked
or
all other ints.

→ Thread_yield is thread call which allow thread to voluntary give the CPU and let another thread run

* Implementation of lock and unlock in Mutex :-

mutex-lock :

```
TSL  R1 , mutex
CMP  R1 , 0
JZE  ok
call thread-yield
JMP  mutex_lock

OK : RET
```

mutex_unlock :

```
MOVE  MUTEX, 0
RET
```

** Reader- Writer Problem :-

→ Reader- Writer problems occur ...

$R - W$	→	Problem	On same Database.
$W - R$	→	Problem	
$W_1 - W_2$	→	Problem	
$R_1 - R_2$	→	No Problem	

```
int rc = 0.
Semaphore mutex = 1.
Semaphore db = 1.
```

Binary Semaphore

```
void Reader (void)
{
    while (true)
    {
        down (&mutex)
        rc = rc + 1
        if (rc == 1) then down (&db);
        up (&mutex)

        [ db ]

Exit    down (&mutex)
Code    rc = rc - 1
        if (rc == 0) then up (&db);
        up (&mutex)

        Process_data();
    }
}

void Writer (void)
{
    while (true)
    {
        down (&db);
        [ db ]
        up (&db);
    }
}
```

Date
Page

** Case : 1 : R - W.

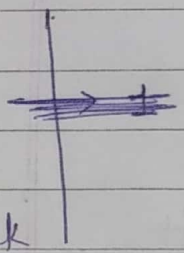
	Reader		Writer
rc = 0	1	Reader	
mutex = 1	0.	=> enters	
db = 1	0.	in c.s.	down(db) will not run

∴ Writer is blocked

→ Binary semaphore is used => mutex and db can be 0 or 1.

** Case : 2 : W - R.

	Writer		Reader.
rc = 0	→ 0	Writer	1
mutex = 1	→ 1	enters	0
db = 1	0.	in c.s.	down(db) will not work



∴ Reader can NOT enter.

** Case : 3 : W_1 - W_2.

	Writer 1		Writer 2.
rc = 0	→ 0	W_1	
mutex = 1	→ 1	enters	
db = 1	0.	in c.s.	down(db) will NOT work

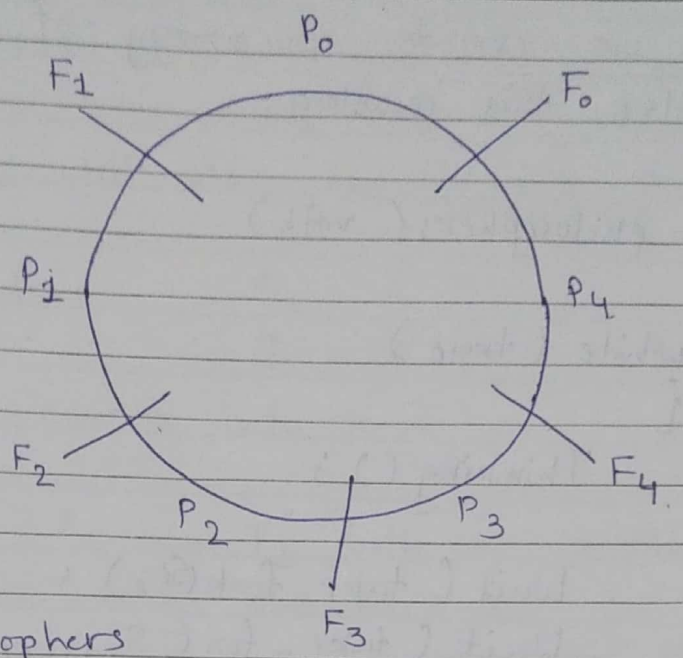
∴ W_2 can NOT enter.

** Case : 4 : R_1 - R_2

	Reader 1.			Reader 2.	
rc = 0	1		R_1 is	2	R_2 is
mutex = 1	0	1	in c.s.	0	in c.s
db = 1	0				

∴ Both the readers can be in Critical section

* Dining Philosophers Problem :-



5 - Philosophers
5 - Forks.

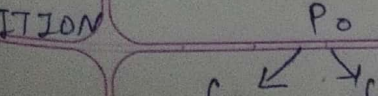
→ ~~When each philosopher~~

```
Void philosopher (void)
{
    while (true)
    {
        Thinking ();
        take_fork(i);      // left
        take_fork((i+1)%.N)  // right.
        Eat();
        Put_fork(i);
        Put_fork((i+1)%.N);
    }
}
```

(P_0, P_1)

→ When two philosophers are trying to eat with the forks then problem occurs.

RACE CONDITION



Here, P_1 will complete eating first and then the turn of P_1.

→ To solve this problem, Binary Semaphore is used.

→ Here we create an array of semaphores to solve this problem.

```
void philosophers(void)
{
    while(true)
    {
        Thinking();

        Wait(take_fork(Si));
        Wait(take_for(S(i+1)%N));

        EAT();

        Signal(Put_fork(Si));
        Signal(Put_fork(S(i+1)%N);
    }
}
```

→ Initially :- S_0 S_1 S_2 S_3 S_4

1 1 1 1 1

Philosophers/

→ Processes needs following semaphores to wait or down to take forks.

P_0 : S_0 S_1

P_1 : S_1 S_2

P_2 : S_2 S_3

P_3 : S_3 S_4

P_4 : S_4 S_0

Date
Page

* P_0 enters ∴ $S_0 = \cancel{1}\ 0$. ∴ P_0 enters into c.s.
$S_1 = \cancel{1}$

→ P_1 enters ∴ $S_1 = 0$ ← it will not be down.
$S_2 = 1$
∴ P_1 will NOT enter into c.s.

→ P_2 enters ∴ $S_2 = \cancel{1}\ 0$
$S_3 = \cancel{1}\ 0$.
∴ P_2 enters into c.s.

* Here, P_0 and P_2 both are eating which means both are in c.s. But, both the processes are independent. Therefore, both can go into c.s

∴ More than 1 philosopher can eat at the same time.

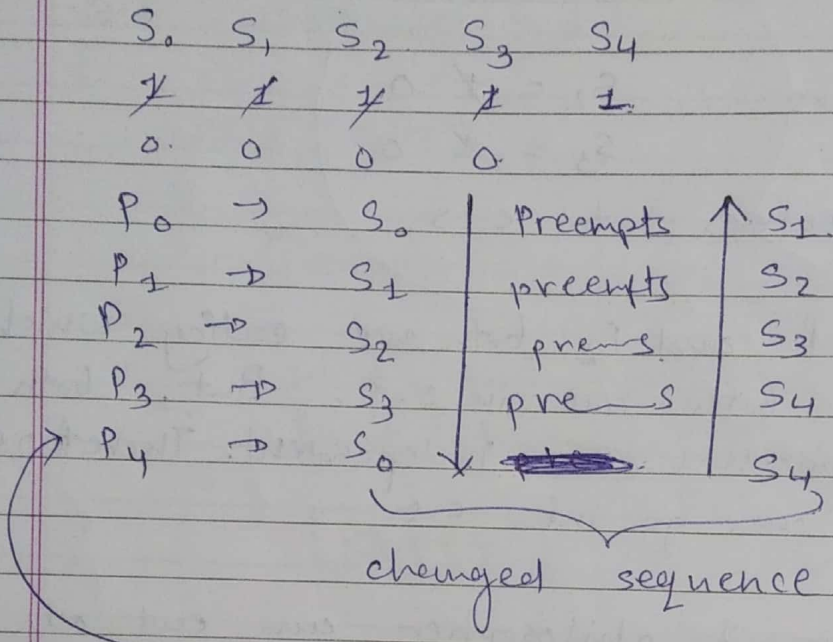
* P_0 : S_0 S_1
P_1 : S_1 S_2
P_2 : S_2 S_3
P_3 : S_3 S_4
P_4 : S_4 S_5 → NOT executed
→ executed

S_0	S_1	S_2	S_3	S_4
$\cancel{1}$	$\cancel{1}$	$\cancel{1}$	$\cancel{1}$	$\cancel{1}$
0	0	0	0	0.

→ When every process preempts before executing the second semaphores then deadlock condition occurs.

∴ After going through all the processes the array of semaphores down to 0
∴ Again when process (P_0) wants to take fork (1), it will NOT be able to take it as S1 is already downed to 0 by P_1.

** To remove Deadlock condition in Dining Philosopher problem, we just need to change the sequence of taking forks for the last philosopher / or any one philosopher.



This process will block as S_0 is already 0.

∴ We will be able to get atleast a single semaphore '1' which will execute further processes and will NOT get struck in deadlock.

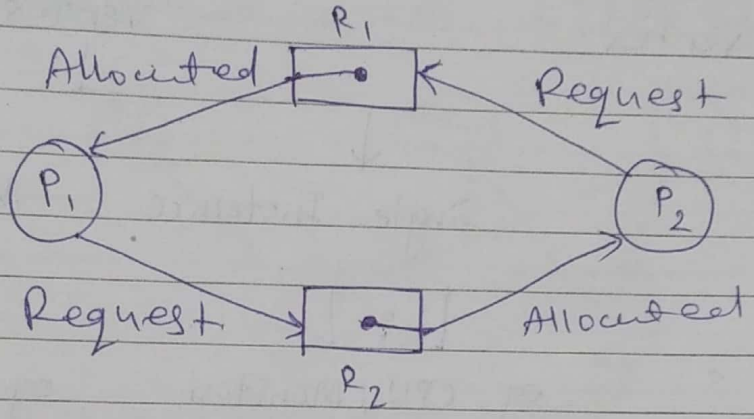
→ After the preemption of P_3 process, P_3 can resume taking the right fork as we have $S_4 = 0$.

(Likewise we can move in upwards direction to execute all the remaining processes.)

In this case, P_3 will execute completly and then will go out from C.S. and P_2 will resume taking the right fork.

* Deadlock :-

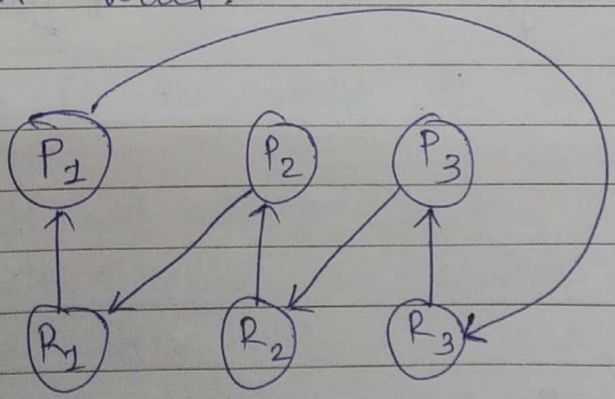
→ If two or more processes are waiting on happening of some event, which never happens, then we say these processes are involved in deadlock then that state is called Deadlock.



** Necessary conditions for occurance of deadlock :-

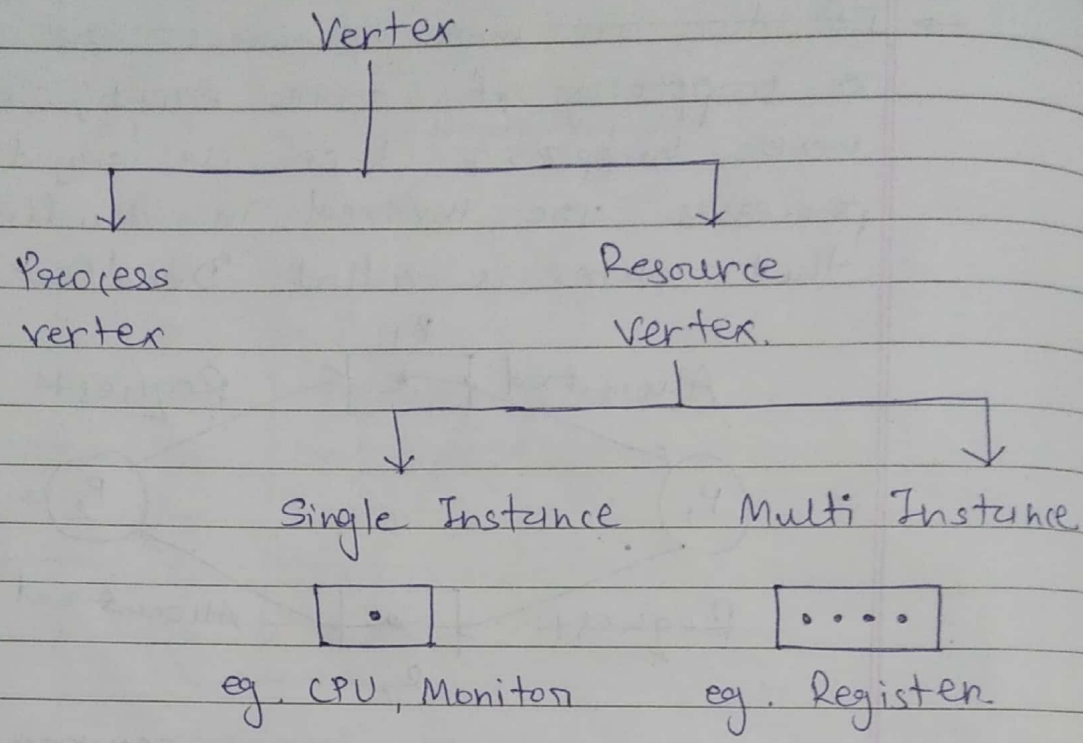
1) Mutual Exclusion
2) No Pre-emption
3) Hold and Wait
4) Circular Wait.

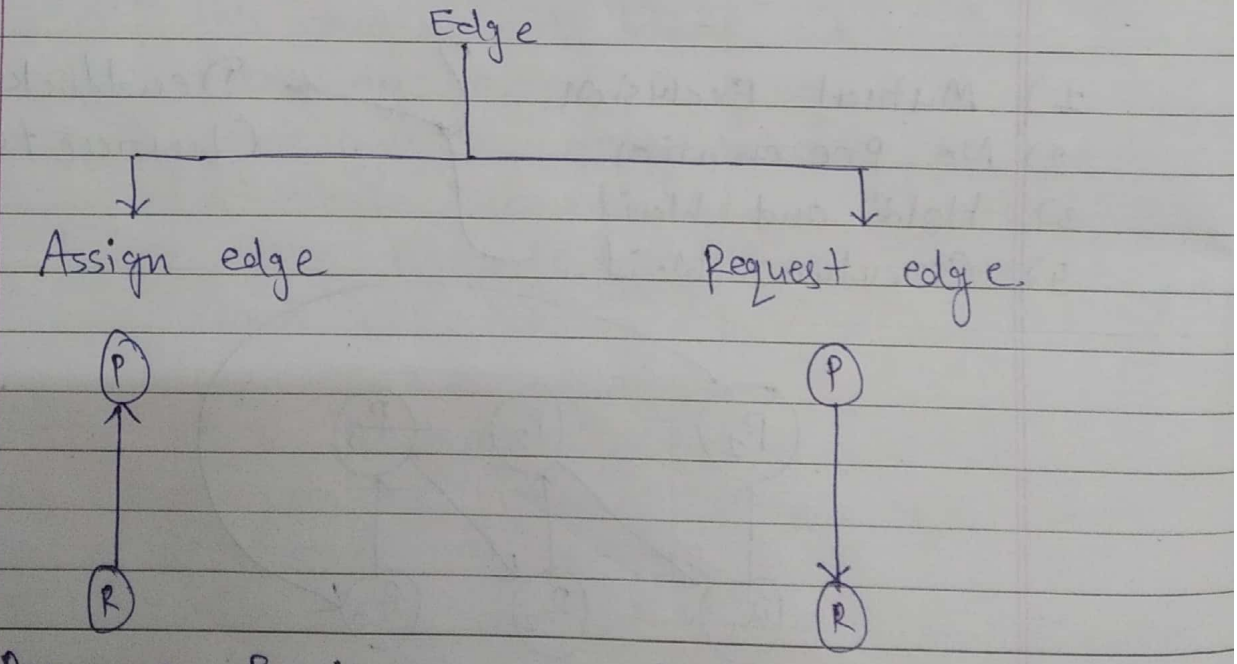
→ Deadlock Characterization



Circular wait.

* Resource Allocation Graph : [RAG] :-



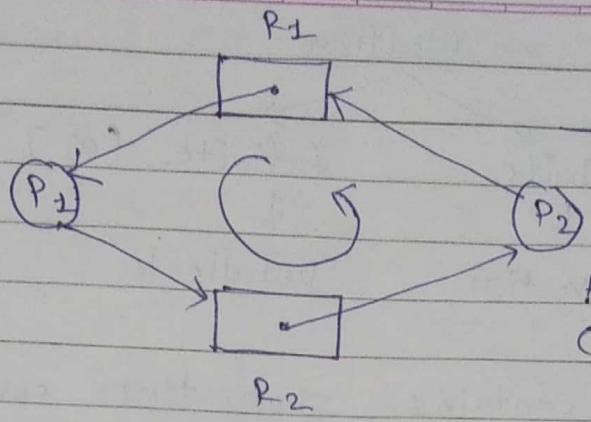


Resource R is assigned to Process P.

Process P is requesting for resource R.

→ From the availability, check whether we can fulfill the REQUEST of any process. or not.
(if not ⇒ deadlock)

Ex: 1 :-

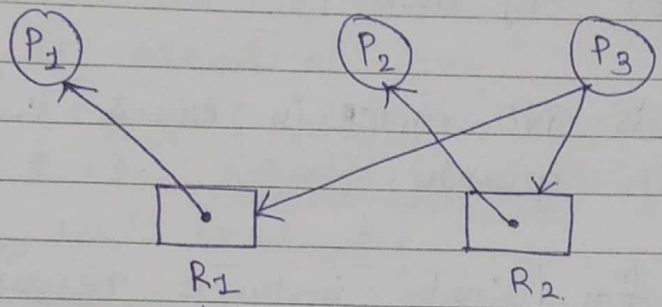


	Assigned		Request	
	R_1	R_2	R_1	R_2
P_1	1	0	0	1
P_2	0	1	1	0

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

→ Circular wait occurs ∴ RAG has deadlock.

Ex: 2 :-



Soln :-

	Assigned		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

→ Availability ⇒ $(R_1, R_2) = (0, 0)$

→ Here, P_1 is demanding for no resources
∴ P_1 will get executed and will terminate
∴ R_1 will be free.

Availability (R_1, R_2)
0 0
→ 1 0
→ 1 1

→ Same way, R_2 will execute and terminate.
∴ R_2 will be free.

→ Now, we can fulfill the request of P_3
∴ P_3 will get executed successfully.

∴ NO deadlock occurs.

Waiting

finite → Starvation

Infinite (∞) → Deadlock

→ Ex:-2 contains starvation as P_3 is waiting for finite time to get resources which were used by P_1 and P_2.

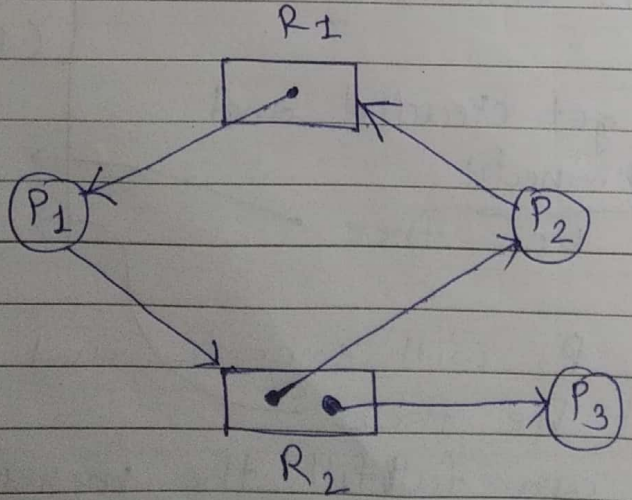
→ There is NO cycle in ex:-2 ∴ It is called Acyclic graph.

** Only for (single instance resource) if RAG has circular wait cycle then there always will be deadlock and if RAG is acyclic there there will NOT be deadlock.

* Multi Instance RAG :-

* Check whether RAG has deadlock or not.

Ex:-1:-



Soln :-

	Assigned		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

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

→ P_3 will be executed and terminated first.

∴ After termination P_3 →

Availability (R_1, R_2)
0 0
0 1
1 1

→ P_1 will be executed and terminated. as P_1 is requesting for (0, 1) which is available

Now, P_1 will free R_1.

→ P_2 will be executed and terminated as we have R_1 available which is requested by P_2.

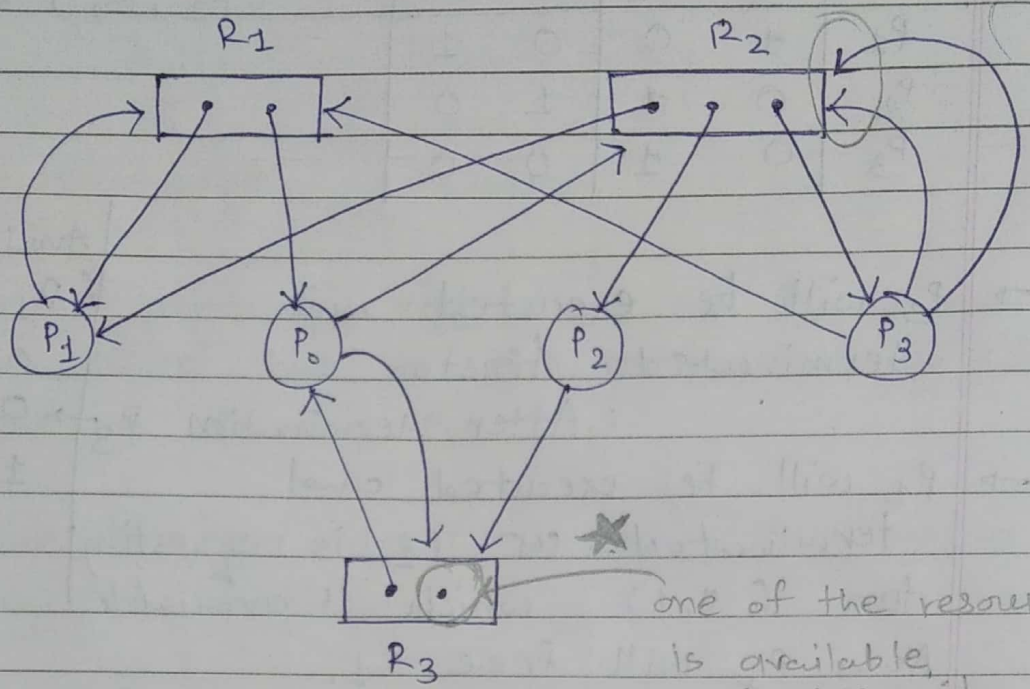
∴ Termination sequence $P_3 \rightarrow P_1 \rightarrow P_2$

∴ There is NO deadlock.

* In case of multi instance, even if a cycle is formed, ~~it's not necessary that there~~ will be deadlock there NOT

In short :→ Check for every example with table-method.

Ex:-2 :-



		Assigned			Request			Current Availability
		R_1	R_2	R_3	R_1	R_2	R_3	(R_1, R_2, R_3)
✓	P_0	1	0	1	0	1	1	(0, 0, 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)
	(0, 0, 1)
∴ P_2 will execute and terminate	→ (0, 1, 1)
∴ P_0 will execute and terminate	→ (1, 1, 2)
∴ P_1 will execute and terminate	→ (2, 2, 2)
∴ P_3 will execute and terminate	→ (2, 3, 2)

∴ Sequence : $P_2 \rightarrow P_0 \rightarrow P_1 \rightarrow P_3$

∴ There is NO DEADLOCK.

* Various Methods to handle Deadlock :-

1) Deadlock ignorance [Ostrich Method]
2) Deadlock Prevention
3) Deadlock Avoidance [Banker's Algorithm]
4) Deadlock Detection and Recovery.

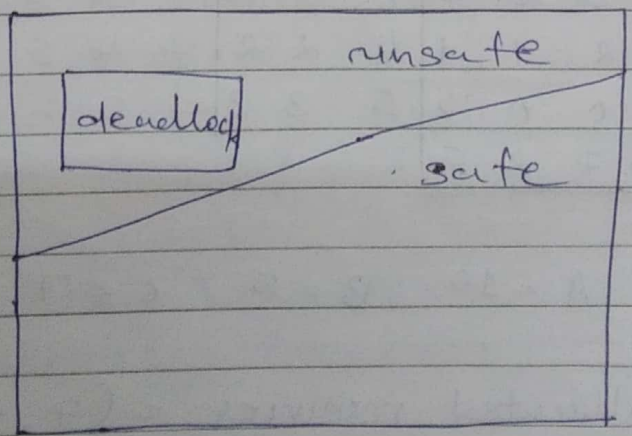
1) Deadlock ignorance [Ostrich Method] :-

→ Writing additional code for the solution of deadlock will increase the functionality but it will affect the performance adversely.

/*

→ If a system is in safe state ⇒ no deadlocks.
→ If a system is in unsafe state ⇒ possibility of deadlock
→ Avoidance ⇒ ensure that system will never enter an unsafe state.

Deadlock Avoidance Theory



→ When P_0 is requesting for R_1 which is occupied by P_1 then system will decide if allocating R_{e_1} will put the system in safe state. If yes then allocate R_1.

*/

2) Deadlock Prevention :-

→ To prevent deadlock we need to satisfy following condition :

1) No Mutual Exclusion
2) Preemption
3) No Hold and Wait
4) No Circular Wait

3) Deadlock Avoidance :-

* Banker's Algorithm :-

→ It is also used for deadlock detection.

CPU Memory Printer

Assigned/Allocation

Max - Allocation

* Process	Allocation A	B	C	Max Need A	B	C	Available A	B	C	Remaining Need A	B	C
P_1	0	1	0	7	5	3	3	3	2	7	4	3
P_2	2	0	0	3	2	2	5	3	2	1	2	2
P_3	3	0	2	9	0	2	7	4	3	6	0	0
P_4	2	1	1	4	2	2	7	4	5	2	1	1
P_5	0	0	2	5	3	3	7	5	5	5	3	1
	7	2	5									

→ Total A = 10, B = 5, C = 7. Given

Sol^n :→ Total allocated resources = (7, 2, 5) = (A, B, C)

∴ Remaining resources at a time

= (10, 5, 7) − (7, 2, 5)

= (3, 3, 2).

→ Now, calculate the remaining need of every process.
→ Remaining Need = Max need − Allocation.

→ Now, check whether (3, 3, 2) which is available can satisfy the remaining need of any process. If yes then proceed further otherwise there will be Deadlock.
Current Availability ≥ Remaining Need.

→ Process sequence = P_2, P_4, P_5, P_1, P_3
(Safe sequence)

→ You can consider any process whose remaining need can be satisfied with the current need.

∴ Ans :- $P_2 \rightarrow P_4 \rightarrow P_5 \rightarrow P_1 \rightarrow P_3$

4) Deadlock Detection and Recovery :-

→ We can use Banker's algorith to detect whether deadlock exist or not.
→ To Recove from deadlock we use following formula/ methods :-

1) Process Termination
2) Resource Pre-emption.

* Que:- Banker's Algorithm

Process	Allocation			Need. Max			Remaining Need			Available		
	E	F	G	E	F	G	E	F	G	E	F	G
P_0	1	0	1	4	3	1	3	3	0	3	3	0
P_1	1	1	2	2	1	4	1	0	2	4	3	1
P_2	1	0	3	1	3	3	0	3	0	5	3	4
P_3	2	0	0	5	4	1	3	4	1	6	4	6
	5	1	6							8	4	6

Current availability

Soln → $P_0 \rightarrow P_2 \rightarrow P_1 \rightarrow P_3 \rightarrow P_4$.

→ Here we don't need to calculate (5,1,6) as we are already given available resources as (3,3,0).

→ No deadlock ∴ Safe state.

Que:- A system is having 3 processes, each requires 2 unit of resources of 'R'. Find → The minimum no. of units of 'R' such that no deadlock will occur.

a) 3 b) 5 c) 6 d) 4.

Soln:- Suppose processes are P_1, P_2, P_3 each require 2 units of resources.

∴ Allocate the 1 less unit.

∴ P_1 P_2 P_3

Allocated (1 1 1) = 3 + 1 ← Add one to avoid Deadlock

= 4

*

P_1	P_2	P_3	
3	4	5	Required Resources.
2	3	4	Allocate 1 less resource.
↓			
9	+(1)		→ to avoid deadlock.

$= 10.$

∴ Minimum 10 Resources require to avoid deadlock condition.

*

P_1	P_2	P_3	
2	2	2	Required Resources.

Find maximum no. of resources which are allocated and still deadlock exists.

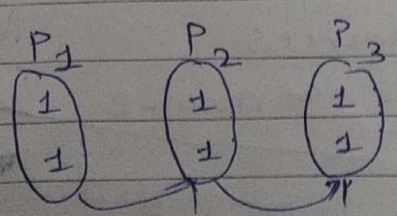
→ Ans:- Allocate 1 less to each process.

P_1	P_2	P_3	
1	1	1	→ 3

∴ Max 3 resources are needed to allocate and still there will be a deadlock situation.

Que:- Consider a system with 3 processes that share 4 instances of same resource type. Each process can request a max of K instances. The largest value of 'K' that will always avoid deadlock is ____.

→



$R = 4$

∴ $K = 2$

* Allocate Maximum Resources but still there exists a Deadlock :-

R - Resources, n - Processes.

Processes :- P_1 P_2 P_3 --- P_n

Demand :- d_1 d_2 d_3 --- d_n

$\rightarrow (d_1 - 1) + (d_2 - 1) + \dots + (d_n - 1)$

$= \sum_{i=1}^{n} d_i - n$

$\therefore$ When $R \leq \sum_{i=1}^{n} d_i - n \Rightarrow$ Deadlock

& when $R > \sum_{i=1}^{n} d_i - n \Rightarrow$ No Deadlock.

$\therefore R + n > \sum_{i=1}^{n} d_i$

Total Demand

$\therefore$ Total no. of Resources + Total no. of Processes $> \sum_{i=1}^{n} d_i$

$\Rightarrow$ No Deadlock.

(Total demand = no. of process $\times$ demand of one process)

Ex :- 4 processes, 4 Resources,

$\rightarrow 4 + 4 > 4 \times (1)$ ← Assume

$8 > 4$

$\rightarrow 8 > 4 \times (2)$

$8 > 8$ False.

Max demand $\therefore$ there always be deadlocks

$8 > 4 \times (3)$

$8 > 12$ False

Deadlock

Operating System

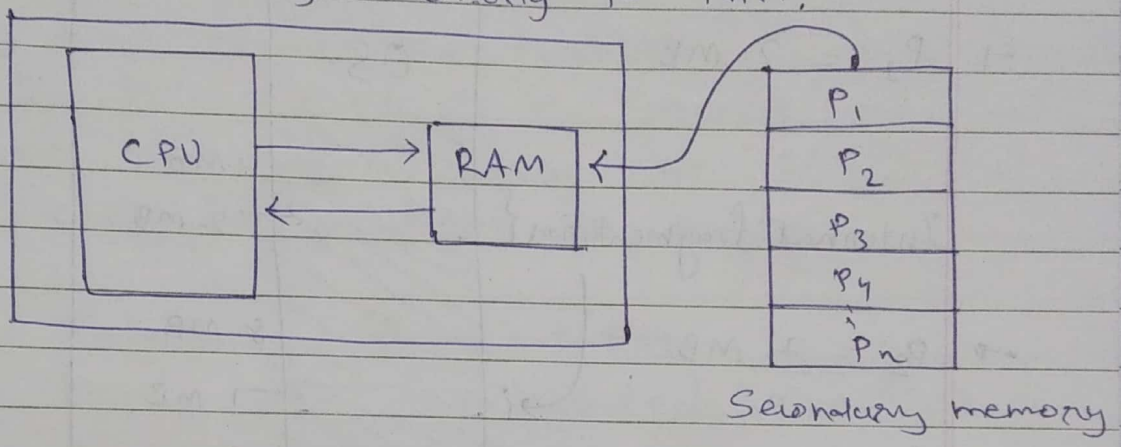
Date
Page

* Memory Management :-

→ Method of managing the primary memory.

* Multiprogramming :-

→ To transfer maximum number of processes from secondary memory to RAM.



→ Higher the degree of multiprogramming ⇒ Higher the utilization of CPU.

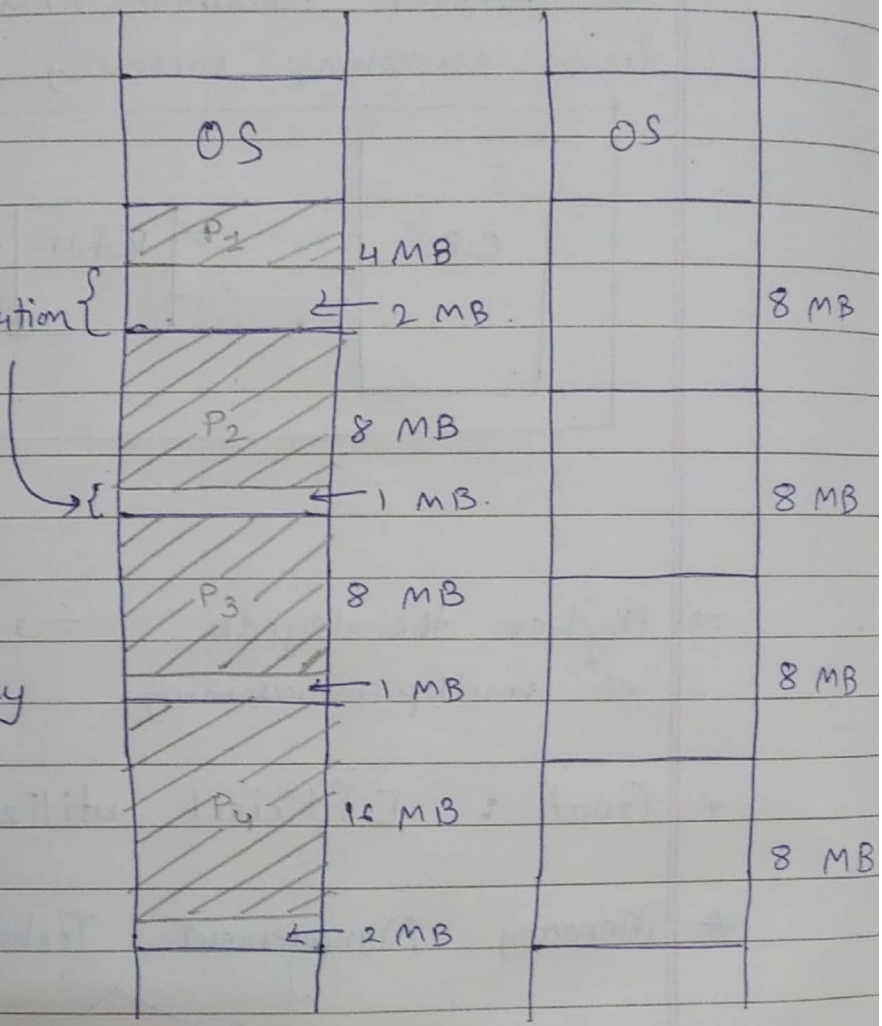
* Goal : Efficient utilization of memory. (RAM).

* Memory Management Techniques :-

- Contagious
  - Static → Fixed partition
  - Dynamic → Variable partition
- Non-contagious
  - Paging
  - Multilevel Paging
  - Inverted Paging
  - Segmentation
  - Segmented Paging

* Fixed Partitioning / Static Partition :-

→ Number of partitions are fixed.
→ Size of each partition may or may not be same.
→ Contagious allocation :. spanning is NOT allowed.



→ P_1 = 2 MB

Internal fragmentation {

→ P_2 = 7 MB

→ P_3 = 32 MB can NOT be allocated in any partition.

→ P_3 = 7 MB
→ P_4 = 14 MB
→ P_5 = 5 MB

→ Problems / limitations :-

1) Internal fragmentation :-
→ When process having lesser size then the partition then unallocated space can NOT be used further. which is called internal fragmentation.

2) Limit in process size

→ When we want to allocate the process of size n which is greater than the maximum partition size then it will NOT be allocated.

3). Limitation on Degree of Multiprogramming.

→ We can NOT allocate the memory partition if no partition is empty/vacant.

→ In this example, P_5 process can not be allocated memory as RAM is not empty / no partition is empty.

4). External fragmentation.

→ Here total unallocated memory size is 2 + 1 + 1 + 2 = 6. MB.

→ Now, even if we want to allocate memory for P_5 process of size 5, memory will NOT be allocated. (Spanning is NOT allowed) (∵ contagious allocation)
Therefore, we need to have one partition of ≥ 5 MB to allocate size for process P_5.

→ Advantage :-

1) Allocation is easy.

* Dynamic Partition / Variable Partition :-

→ Size of each process is allocated during Run time.

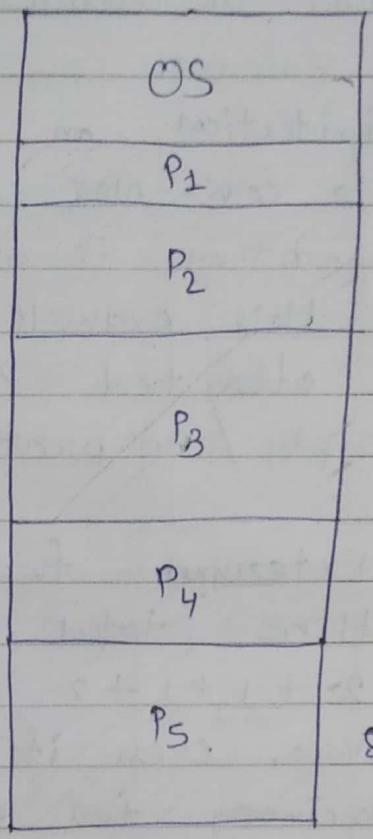
→ $P_1 = 2$ MB

→ $P_2 = 4$ MB

→ $P_3 = 8$ MB

→ $P_4 = 4$ MB

→ $P_5 = 8$ MB



→ Advantages :-

1) No internal fragmentation
→ size of each partition is allocated during Runtime. Therefore.

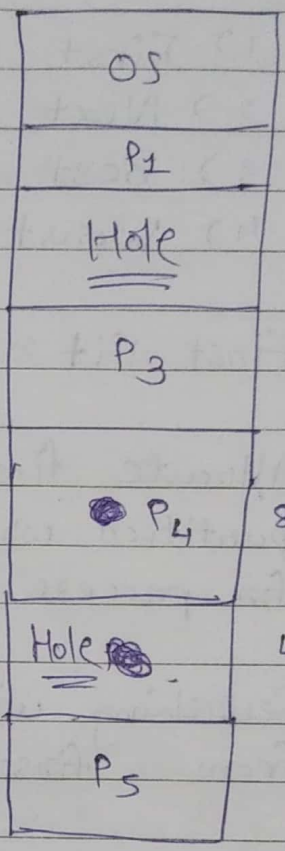
2) No limitation on number of processes.
→ Regardless of the total size of RAM, we can say that there will be allocated maximum no of processes into RAM until we reach to the end size of RAM.

3) No Limitations on process size

→ There won't be any limitations on the size of process.

→ Disadvantages :-

1) When we want to allocate memory to the process P_6 size of 4 MB and we are having empty slot of 4 MB, STILL we are unable to do it. which leads to External fragmentation.



→ It can be removed by compactation.

which means moving all the processes to fill the empty space.

:- There will be space at the end of the RAM. and we will be able to allocated that memory.

→ But, this process is time consuming which affects the performance.

2) Allocation / Deallocation is complex.

→ As after having holes it is difficult to allocate / deallocate the memory.

* Various Allocation Methods in Contagious Memory Management :-

→ 1) First Fit
2) Next Fit
3) Best Fit
4) Worst Fit.

} strategies to allocate processes into free memory partitions

1) First Fit :-

→ Allocate first free partition which is enough for process.

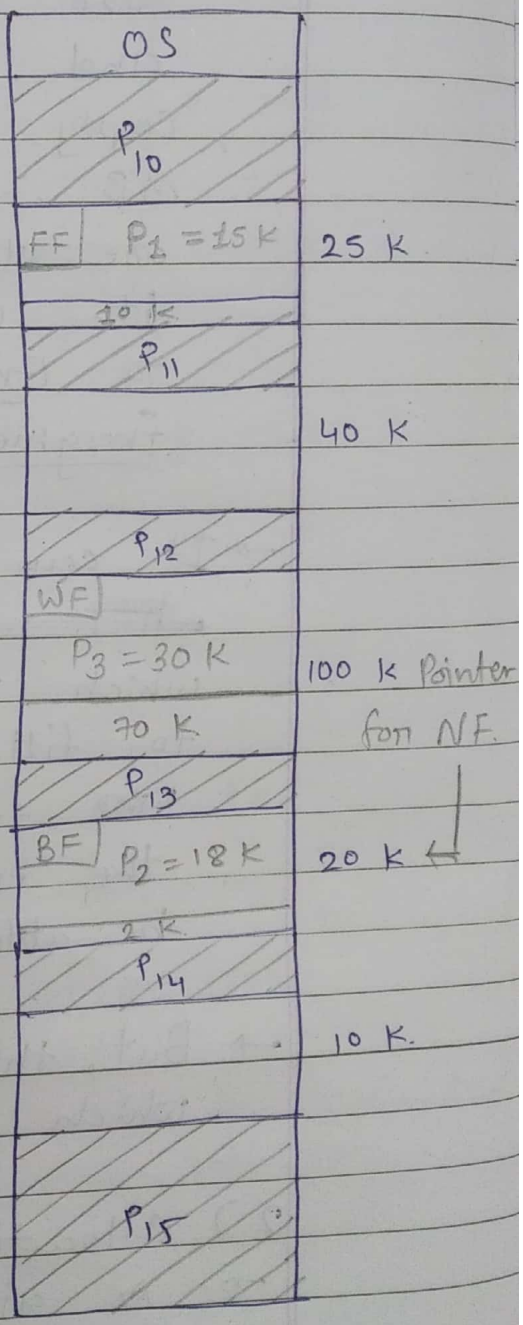
→ Searching will be started from first partition.

→ Advantages :

- → Simple
- → Tends to produce larger free blocks towards the end of the address space.

→ Disadvantages :

→ Internal Fragmentation.



2) Best Fit :-

→ Allocate the small free partition which is small enough for process.

→ All the partitions will be searched from top to bottom and allocate the partition in which internal fragmentation is less.

→ Advantages :-

→ Relatively simple
→ Work well when most allocations are of small size.

→ Disadvantages :-

→ Internal fragmentation very low (~~Advantage~~)
→ Slow allocation and deallocation.
→ Tends to produce many useless tiny fragments.

3) Next Fit

→ Same as first fit but starts searching always from the last allocated hole.

→ Advantages :-

→ Starts searching from the last allocated hole by using pointer.

→ Remaining advantages & disadvantages same as Best Fit.

4) Worst Fit

→ Allocate the largest hole. which results in high amount of internal fragmentation.
→ All partitions will be searched from top to bottom.

→ Advantages :-

→ Works best if allocations are of medium sizes.

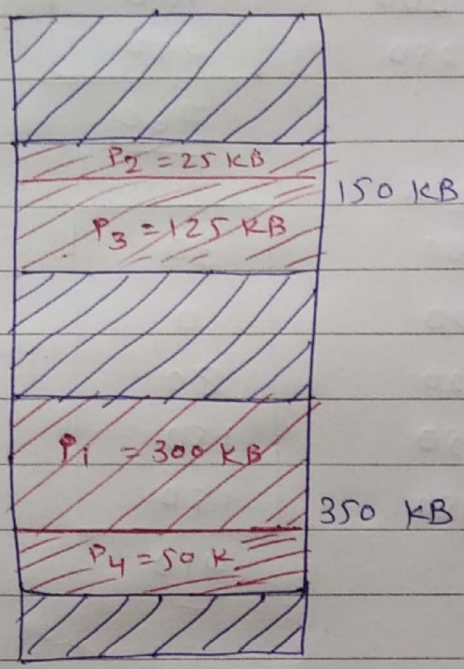
→ Disadvantages :

→ It will search entire list to find the largest hole present and then it will allocate the memory. which makes it slower.
→ Internal fragmentation.
→ Tends to break large free blocks such that large partitions can Not be allocated.

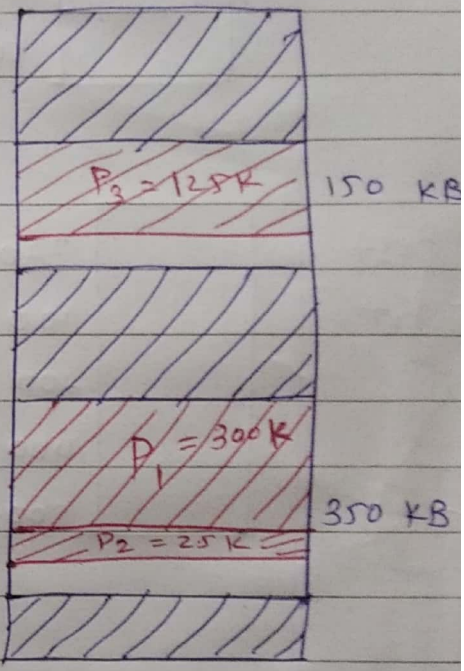
Que: Requests from processes are 300 K, 25 K, 125 K, 50 K respectively. The above request could be satisfied with ..:-

A) Best fit but not first fit
B) First fit but not best fit
C) Both
D) None.

Ans: →



First Fit.



Best Fit.

→ let P_1 = 300 KB
P_2 = 25 KB
P_3 = 125 KB
P_4 = 50 KB

↓

We can not span this process and add into Best fit case memory. As it is contagious memory allocation method

↓

Remaining space should be minimum in case of Best fit

Ex:- Calculate total internal fragmentation in each allocation strategy.

→ Processes => 120, 300, 100, 250, 350

First Fit

	Process	Internal fragmentation
330	→ 120	210
100	→ 100	0
200	Consider this →	200
400	→ 300	100
500	→ 250	250
		760

Best Fit

	Process	Internal fragmentation
330	→ 300	30
100	→ 100	0
200	→ 120	80
400	→ 250	150
500	→ 350	150
		410

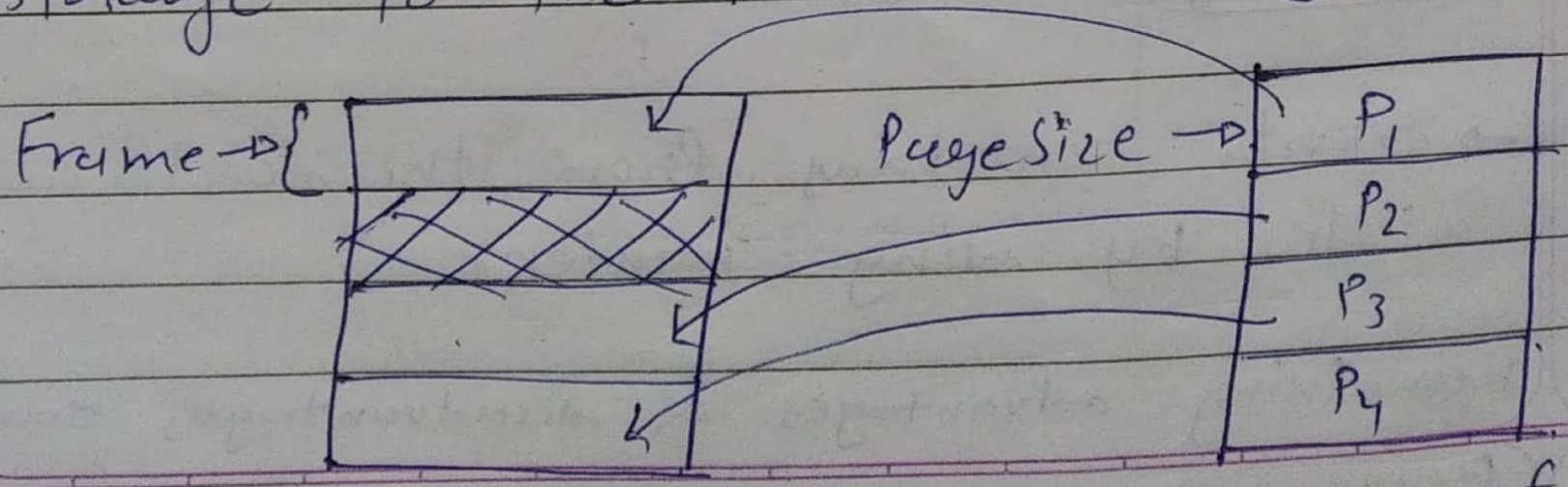
Worst Fit

	Process	Internal fragmentation
330	→ 100	200
100	→	100
200	→	200
400	→ 300	100
500	→ 120	380
		1010

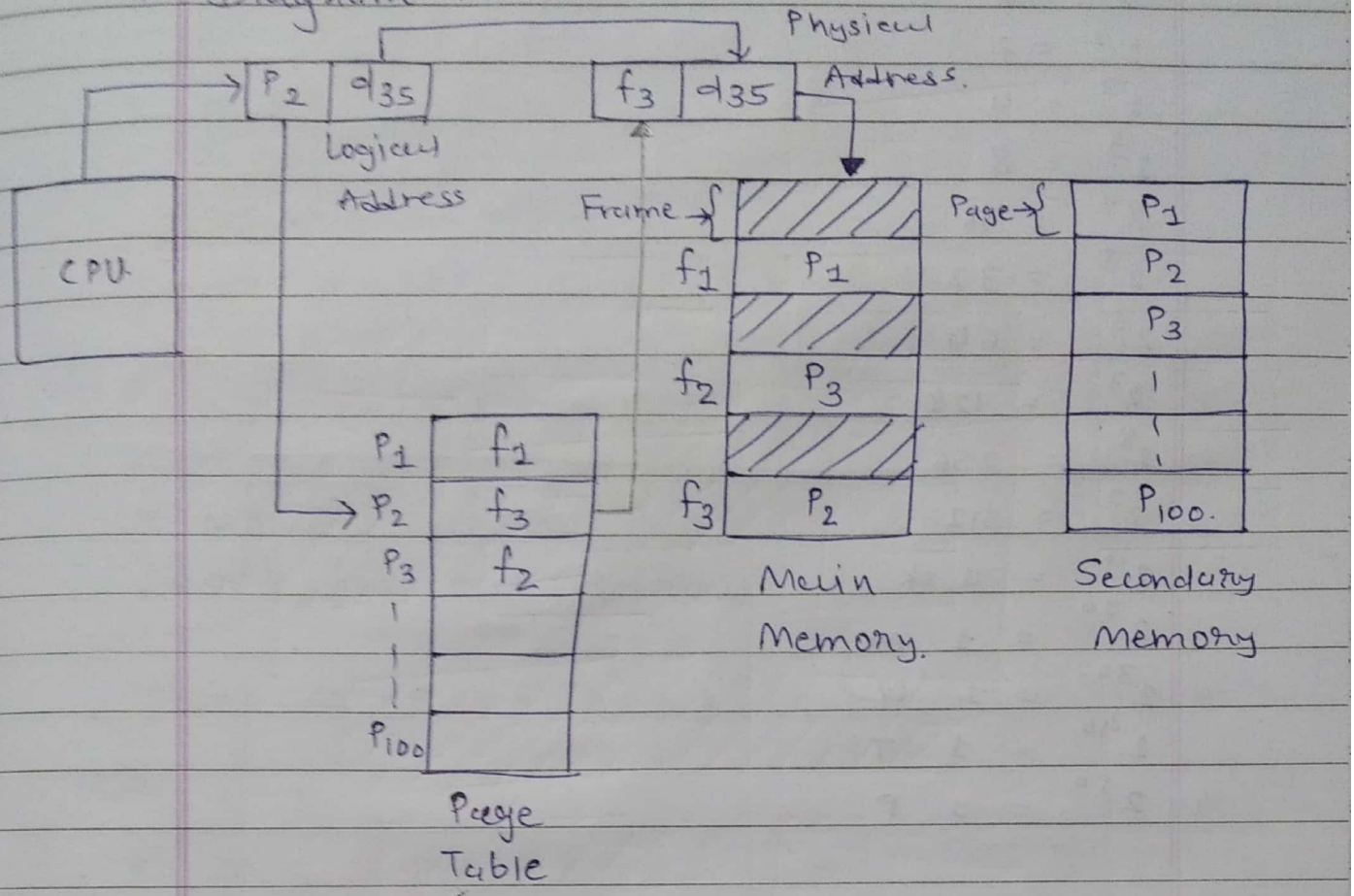
* Non Contagious Memory Allocation :-

* Paging :-

Paging is a storage mechanism used to retrieve processes from secondary storage to the main memory as pages.



* Diagram :-



→ Page table is a data structure.

→ No. of entries in a page table = No. of pages a process has in S.M.

→ For every process, page table is distinct.

→ Page table contains the base address where page is currently stored.

→ Page size = Frame size.

$2^0 = 1$

$2^1 = 2$

$2^2 = 4$

$2^3 = 8$

$2^4 = 16$

$2^5 = 32$

$2^6 = 64$

$2^7 = 128$

$2^8 = 256$

$2^9 = 512$

$2^{10} = 1\ K$

$2^{20} = 1\ M$

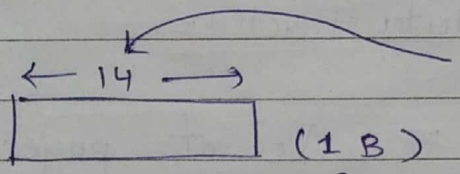
$2^{30} = 1\ G$

$2^{46} = 1\ T$

$2^{50} = 1\ P$

* Find the size of memory.

1)



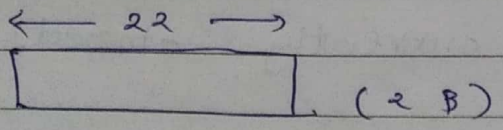
no of bits to represent address

system is 1 Byte addressable.

$= 2^{14} \times 1$

$= 16\ KB$

2)



$= 2^{22} \times 2$ Bytes

$= 2^{23}$ Bytes

$= 8\ MB$

* Find the number of bits in the address of/

1. M.S. = 64 KB.

~~$\Rightarrow 2^{64} = 2^4 \times 2^{60}$~~

$\Rightarrow$ 64 KB $= 64 \times 2^{10}$ Bytes

$= 2^6 \times 2^{10}$ Bytes

$= 2^{16}$ $\therefore$ 16 bits

2. Memory size = 32 KB.
Size of each location = 1 Byte

$\Rightarrow$ 32 KB $= \frac{2^5 \times 2^{10}}{1} = 2^{15}$ $\therefore$ 15 bits

3. M. S. = 16 GB
Size of each location = 4 B.

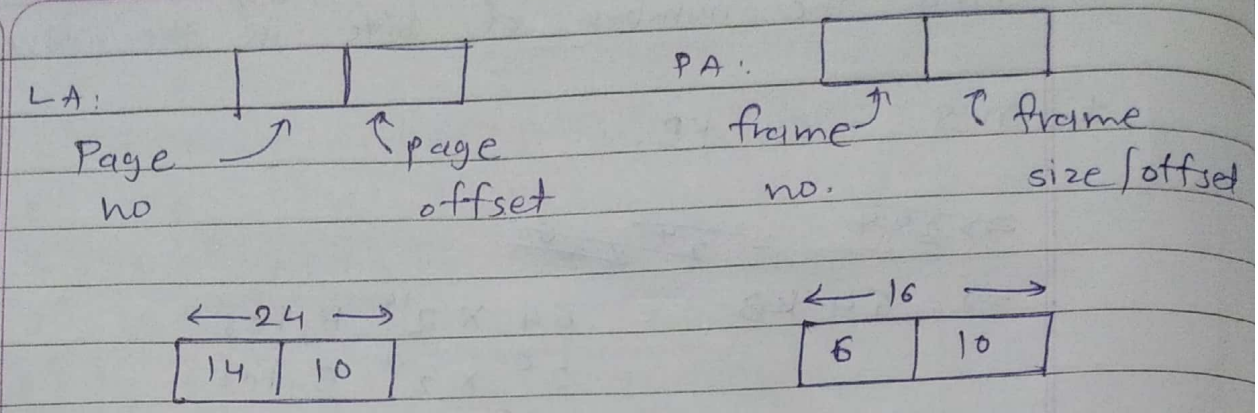
$\Rightarrow \frac{16 \text{ GB}}{4 \text{ B}} = 4 \text{ G} = 2^2 \times 2^{30} = 2^{32}$

$\therefore$ 32 bits

* Calculate no. of pages :-

1. LA = 24 bits
PA = 16 bits
Page size = 1 KB

→ Logical Address space $= 2^{24} = 16$ MB
→ Physical Address space $= 2^{16} = 32$ KB.
→ Page size = 1 kB
$= 2^{10}$ Bytes.



~~Total no. of pages = Process size~~

→ No. of pages = 2^{14}

$= 2^4 \, 2^{10}$

$= 16$ K

* Examples :-

(1) LAS = 4 GB

PAS = 64 MB

Page size = 4 KB.

No. of pages = ?

No. of frames = ?

No. of entries in a page table = ?

Size of page table = ?

Soln :- LAS = 4 GB

$= 4 \times 2^{30}$ Bytes

$= 2^{32}$ Bytes.

∴ LA = 32 bits

PAS = 64 MB

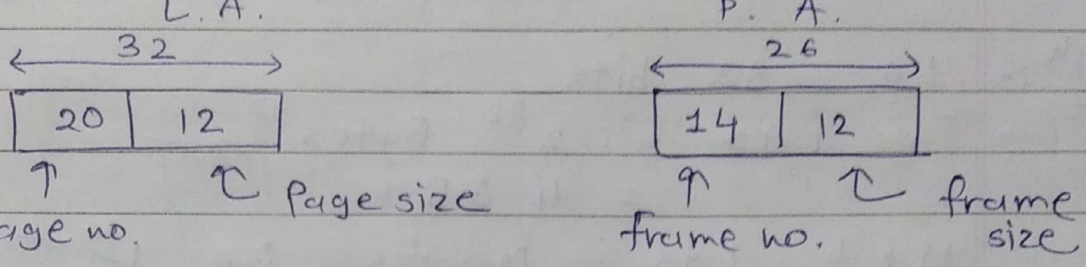
$= 2^6 \times 2^{20}$ Bytes $= 2^{26}$ Bytes

∴ PA = 26 bits.

→ Page size = 4 KB
= $2^2 \times 2^{10}$ Bytes
= 2^{12} Bytes

~~∴ page numbers = no. of frames = 12 bits~~

∴ Page size = frame size ⇒ is of 12 bits



→ Total no. of pages = 2^{20} = 1 M.

→ Total no. of frames = 2^{14} = 16 K

* → No. of entries in a page table = No. of pages in a process.

∴ No. of entries in a page table = 2^{20}
= 1 M

* → Size of page table.

= No. of entries in a page table × No. of bits required to represent frame no.
(∵ page table contains frames)

= 2^{20} × 14 bits
= 2^{20} × 16 bits (∵ approximation)
= $2^{20} \times 2$ Bytes
= 2^{22} Bytes
= 4 MB.

(2) LA = 32 bits
Page size = 4 KB
Each page table entry is of 4 Bytes.

Size of page table = ?

Soln :- LA = 32 bits
∴ LAS = 2^{32} Bytes
= 4 GB
W.K.T. LAS = Process size ✓*
∴ Process size = 4 GB

→ Size of page table,

No. of entries in a page table × Page table entry size

= $\frac{\text{Process size}}{\text{Page size}}$ × Page table entry size

= $\frac{2^{32}}{2^{12}}$ × 4 Bytes.

= $2^{20} \times 2^{2}$ Bytes.

= 2^{22} Bytes

= 4 MB

(3) PAS = 64 MB ⇒ $2^{6} \times 2^{20}$ Bytes = 2^{26} Bytes
LA = 32 bits
Page size = 4 KB

Size of page table = ?

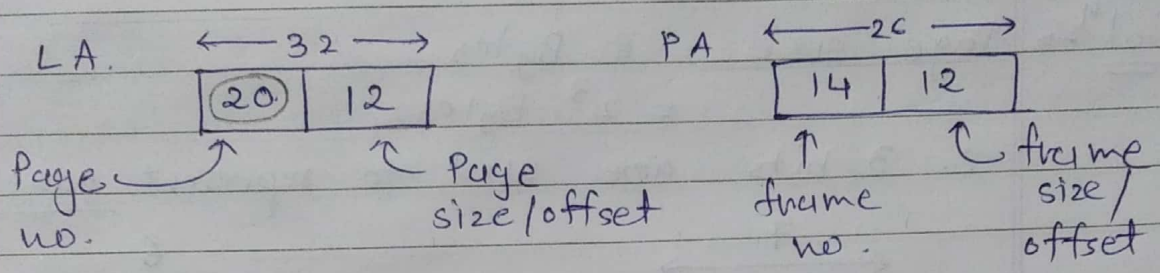
Sol^{n}:- PA = 26 bits.

LA = 32 bits.

Page size = 4 KB

= 2^{12} Bytes.

∴ page size can be represented in 12 bits.



→ Size of page table,

= No. of entries in a page table × Page table entry size

= $\frac{\text{Process size}}{\text{Page size}}$ × Page table entry size
OR
No. of bits in frame no.

= $\frac{2^{32}}{2^{12}}$ × 14 bits. (Frame no.)

= 2^{20} × 16 bits (∵ approximation)

= 2^{20} × 2 Bytes.

= 2^{22} Bytes

= 4 MB

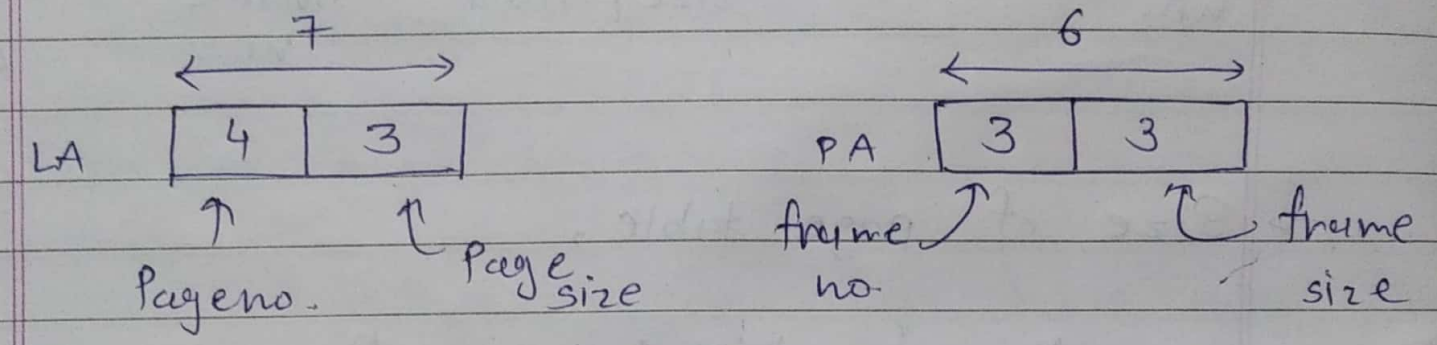
1 Byte = 8 Bits.

Date
Page

(4) LA = 7 bits
PA = 6 bits
Page size = 8 words or Bytes

No. of pages = ?
No. of frames = ?

Soln → Page size = 8 Bytes
$= 2^3$ Bytes.
∴ 3 bits are used to represent page size.



→ No. of frames = 2^3 = 8
→ No of pages = 2^4 = 16

(5) Consider a logical address space of 8 pages of 1024 words each, mapped on to a physical memory of 32 frames.

LA = ? , PA = ?

→ 1 word = 2 Bytes *

→ Here, we are given that logical address space is of 8 pages and each page is of 1024 words.

∴ 1 page contains 1024 words
= 2048 Bytes
= 2^{11} Bytes.

∴ 1 page can be represented using 20 bits.

→ Logical address space contains 8 pages.

∴ LAS = $2^{11} \times 8$
= $2^{11} \times 2^3$ = 2^{14} Bytes
= 16 MB

∴ Logical Address = 14 bits.

→ Physical memory contains 32 frames.
and we know that page size = frame size
∴ 1 frame is of 2^{11} Bytes.

∴ PAS = 32×2^{11} = $2^{11} \times 2^5$ = 2^{16}

∴ PA = 16

* Multilevel Paging :-

Ex: (1). page size = 1 MB = frame size
VA = 64 bits
page table entry size = 4 Bytes

levels of paging = ?
divided PA and VA = ?

→ Page size = 1 MB
= 2^{20} ∴ page size = 20 bits.

→ Page table entry size = 4 bytes
= 32 bits
∴ frame no = 32 bits.

∴ PA = | 32 | 20 |
frame no. ↑ ↑ frame offset.

∴ Number of frames = 2^{32}

→ Size of Main memory = No of frames x frame size
= 2^{32} X. 2^{20}
= 2^{52} Bytes.

∴ PA = ← 52 → | 32 | 20. |
frame no. ↑ ↑ frame size / offset.

→ LA = ← 64 → | 44 | 20 |
Page no. ↑ ↑ page size / offset.

→ Process size = 2^{64} Bytes

LA ← L*

(∵ process is present in logical address space)

→ Inner Page table size = No. of entries in inner page table × Page table entry size

L* = No. of pages the process is divided × Page table entry size

$= 2^{44} \times 4$ Bytes

$= 2^{46}$ Bytes. = 64 TB.

→ We can observe,

→ the size of inner page table is greater than frame size (1 MB)

→ Thus, inner page table can NOT be stored in a single frame.

→ So, inner page table has to divided into pages.

→ No. of pages of the inner page table =

* Thrashing :-

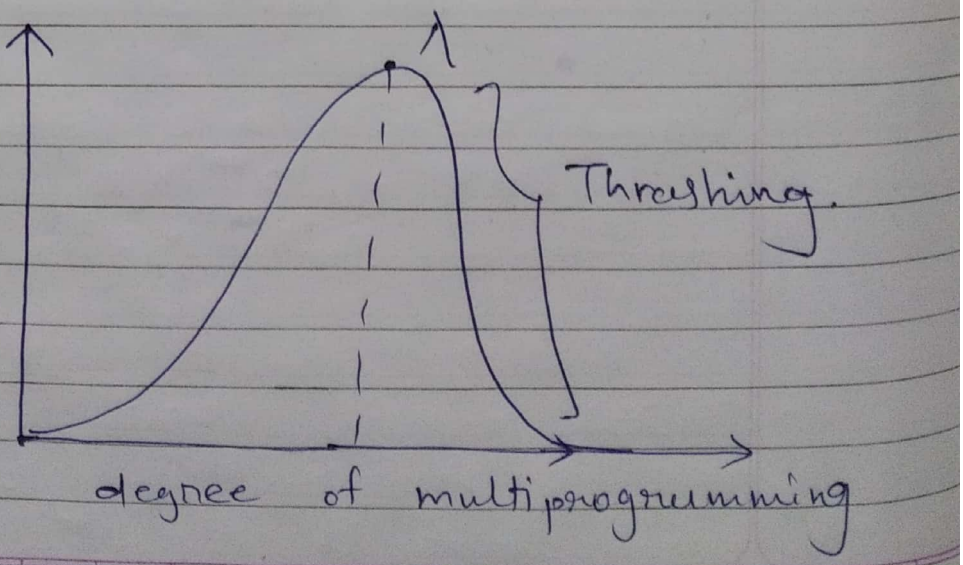
→ Thrashing is a condition in which excessive paging operations are taking place.

→ For example, we have 100 processes and to achieve degree of multiprogramming we enhance the number of processes in Main Memory.

→ For that, we transfers one page of each process to achieve highest degree of multi programming but at some point if page 2 of process 1 is required by the CPU which is not present then Page Fault occurs.

→ To solve page fault, CPU takes time which decreases the performance of CPU.

→ That decreasing performance is called Thrashing.



→ To remove thrashing,

1) Increase the size of main memory.
2) Long Term Scheduler.

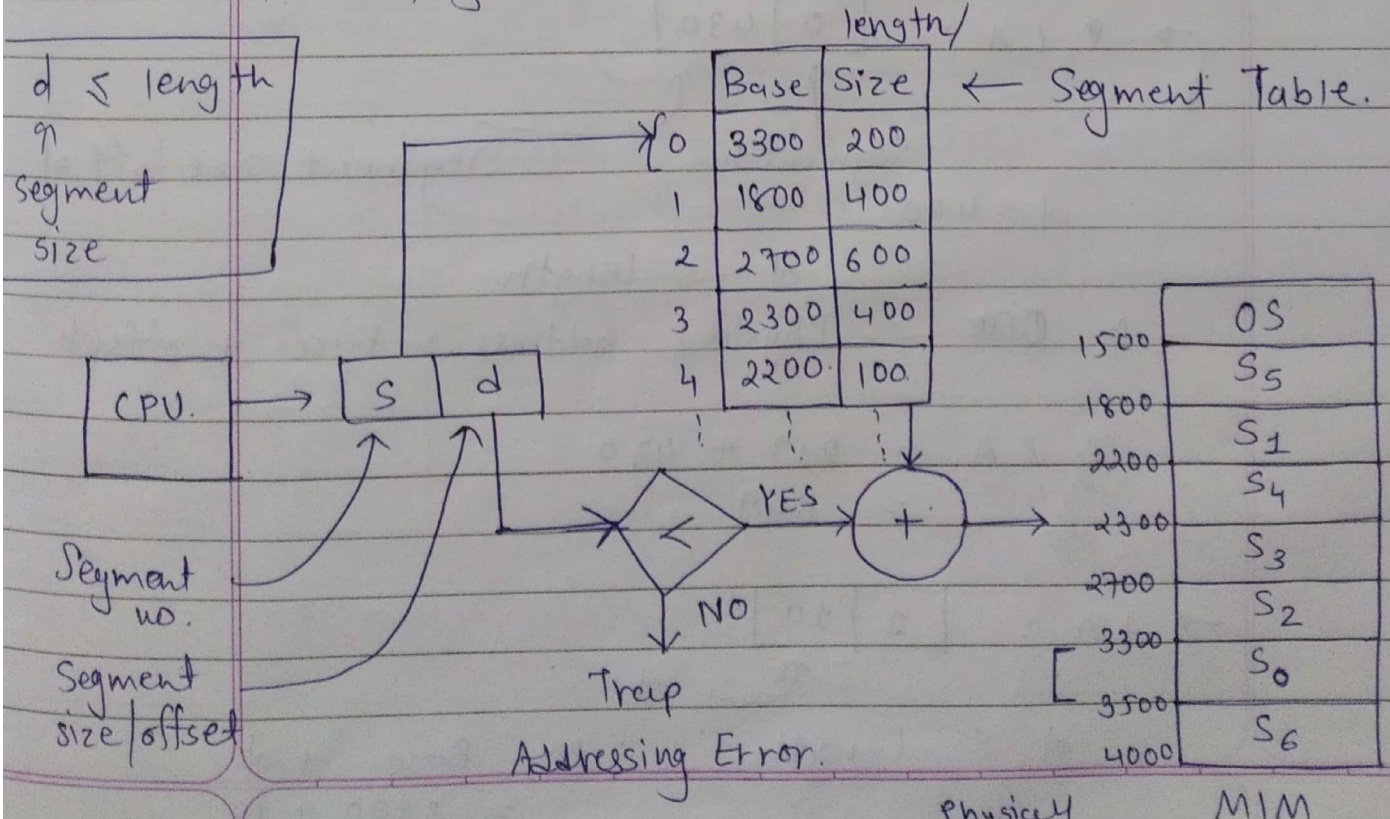
→ It transfers as many processes as possible into ready state.

* Segmentation :-

→ Segmentation is a memory management technique in which memory is divided into the variable size parts. Each part is known as a segment which can be allocated to a process.

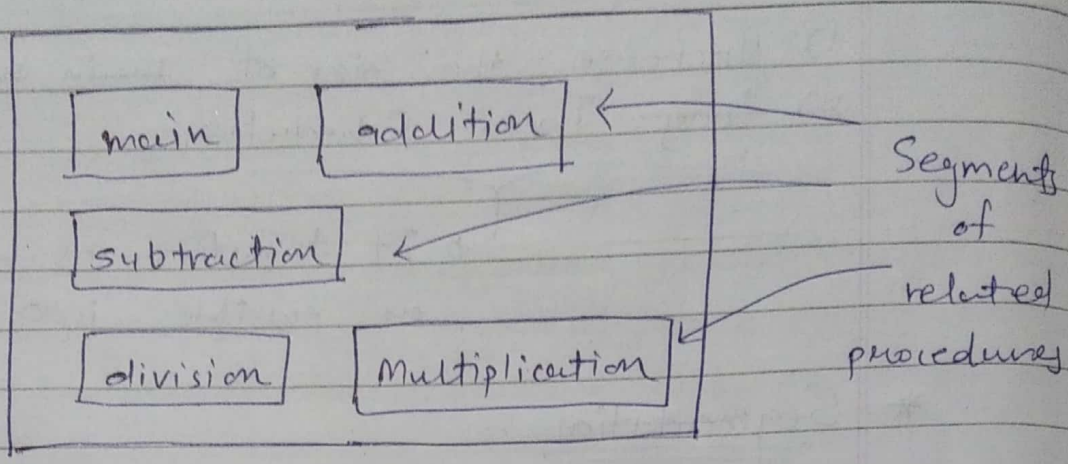
→ Segments can be of varions size.

→ MMU helps in tranforming logical address into physical address.



→ We make segments of related data.

Ex:-



Ex:- (1)

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952.	96.

What are the physical address for following logical addresses?

→ LA : | 0 | 430. |

Segment no. ↑ ↑ Segment size / offset

d = 430.

d ≤ length.

∴ Physical address = Base + offset

∴ PA = 219 + 430

= 649

→ LA : | 1 | 10 |

d ↑

d ≤ length ∴ PA = Base + d

= 2300 + 10

= 2310.

→ LA = | 1 | 11 |

d ≤ length.

∴ PA = Base + offset

= 2300 + 11

= 2311.

→ LA : | 2 | 500. |

↑ d

d ≤ length is not true.

∴ Illegal address.

(2)

Segment	Base	Length
0	3300	200
1	1800	400
2	2700	600
3	2300	400
4	2200.	100.

→ LA = | 0 | 300 |

↑ Segment size. (d)

d ≤ length is false

∴ invalid address.

→ LA = | 2 | 350 |

↑ d

d ≤ length ∴ PA = Base + d

= 1800 + 350

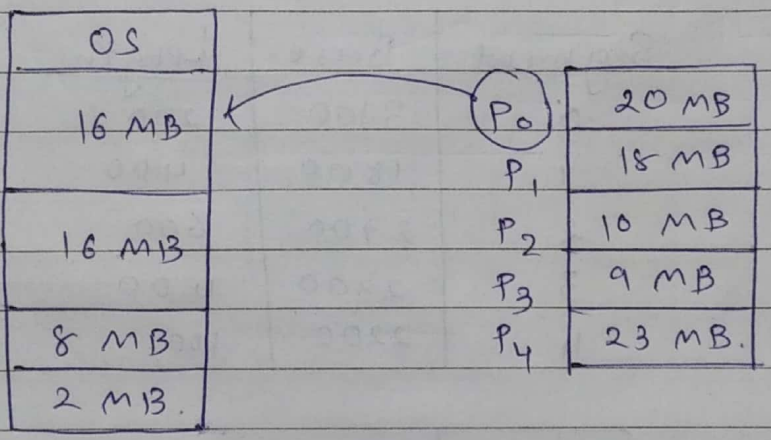
= 2150.

* Overlay :-

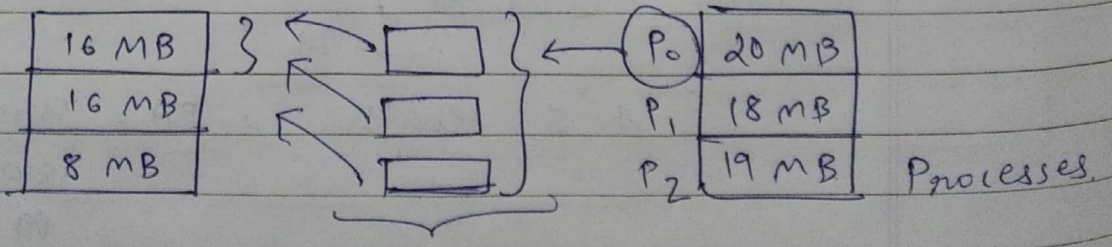
→ It is a method by which a large size process can be put into the main memory.

→ If process size is larger than the main memory then using overlay method, we can put that process into main memory.

→ It is used in embedded systems.



→ In this concept, we can put P0 process of size 20 into main memory by dividing process into small segments.



This segments should be Independent.

Date
Page

* Virtual Memory :- (Demand Paging)

→ Virtual memory is a storage scheme in which secondary memory can be addressed as though it were part of the main memory.

→ For instance, when we want to execute the process of size 16 GB and the size of main memory is 12 GB then virtual memory is used to accomodate the entire process and execute it.

→ Locality of Reference :

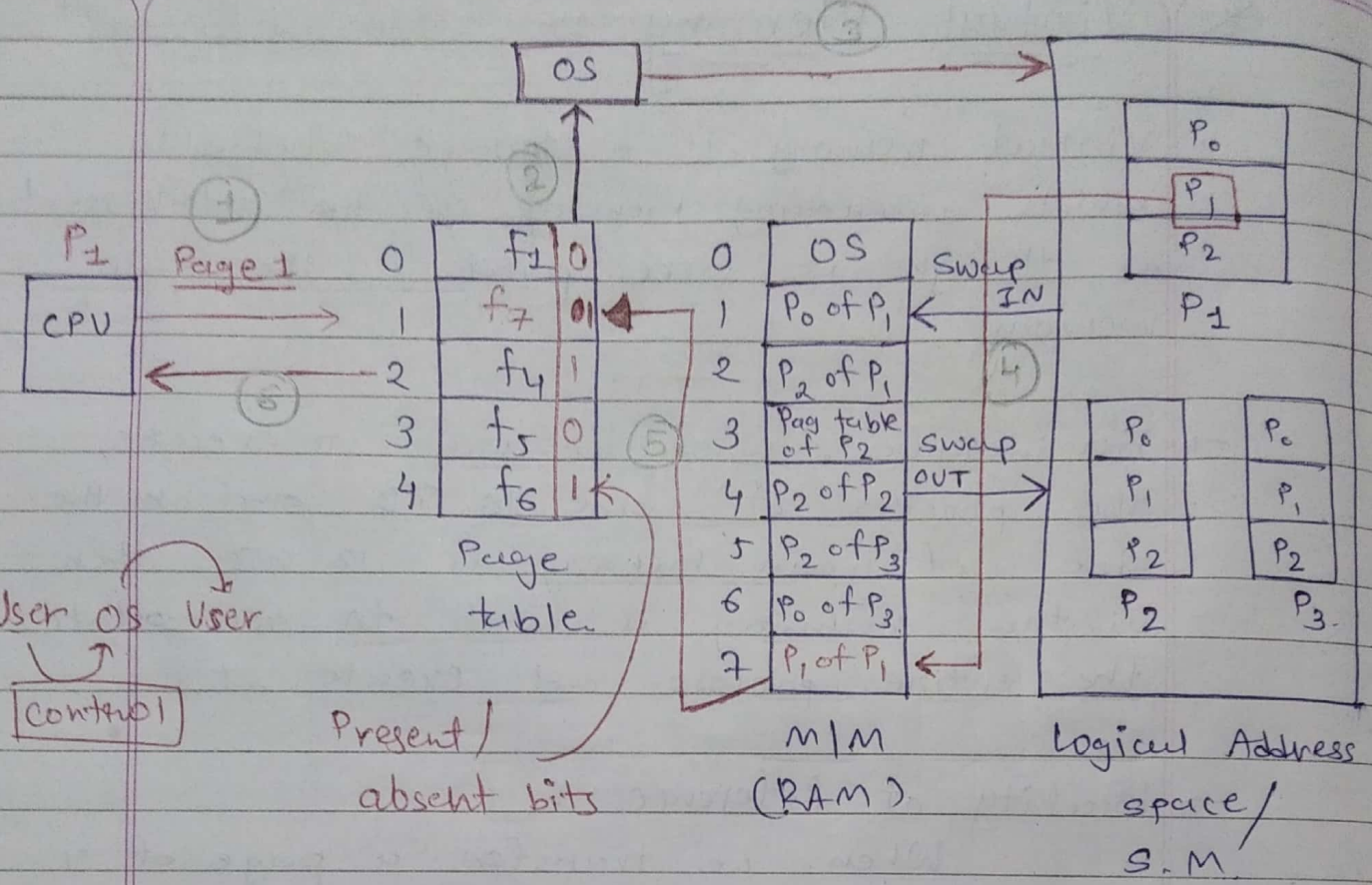
When we transfer a page of a process the we also tranfers the pages which are related to the first page transferred.

→ Advantages :-

1) No limitation of no. of process.
2) No limitation of process size.

* Page fault :-

→ A page fault is an interruption that occurs when a software program attempts to access memory blocks which is not currently stored in a main memory.



→ CPU is asking for page 1 of process 1 which is not present in main memory so there is no frame numbers in a page table.

→ Now, page fault occurs. Therefore, all the control goes to operating system and then OS finds the page of a particular process then transfers it into the RAM and so on...

→ Effective Memory Access Time :-

$$EMAT = P\left(\begin{matrix}\text{Page fault}\\ \text{service time}\end{matrix}\right) + (1-P)\left(\begin{matrix}\text{MM}\\ \text{access}\\ \text{time}\end{matrix}\right)$$

where P = probability of occuring page fault.

Ex:- VA = ~~32~~ bits Page size = 4~~0~~ KB
PA = 023 bits

No of pages (N) in VAS = ?
Page no. (p) in bits = ?
No of frames (m) in PAS = ?
Frame number (f) in bits = ?
Offset (d) = ?
Page table size = ?

Soln:- Page size = 4 KB = $2^2 \times 2^{10}$ B = 2^{12} B
∴ No of pages = 12 = no. of frames.

→ VA: | 17 | 12 | ← 29 →
PA: | 11 | 12 | ← 23 →

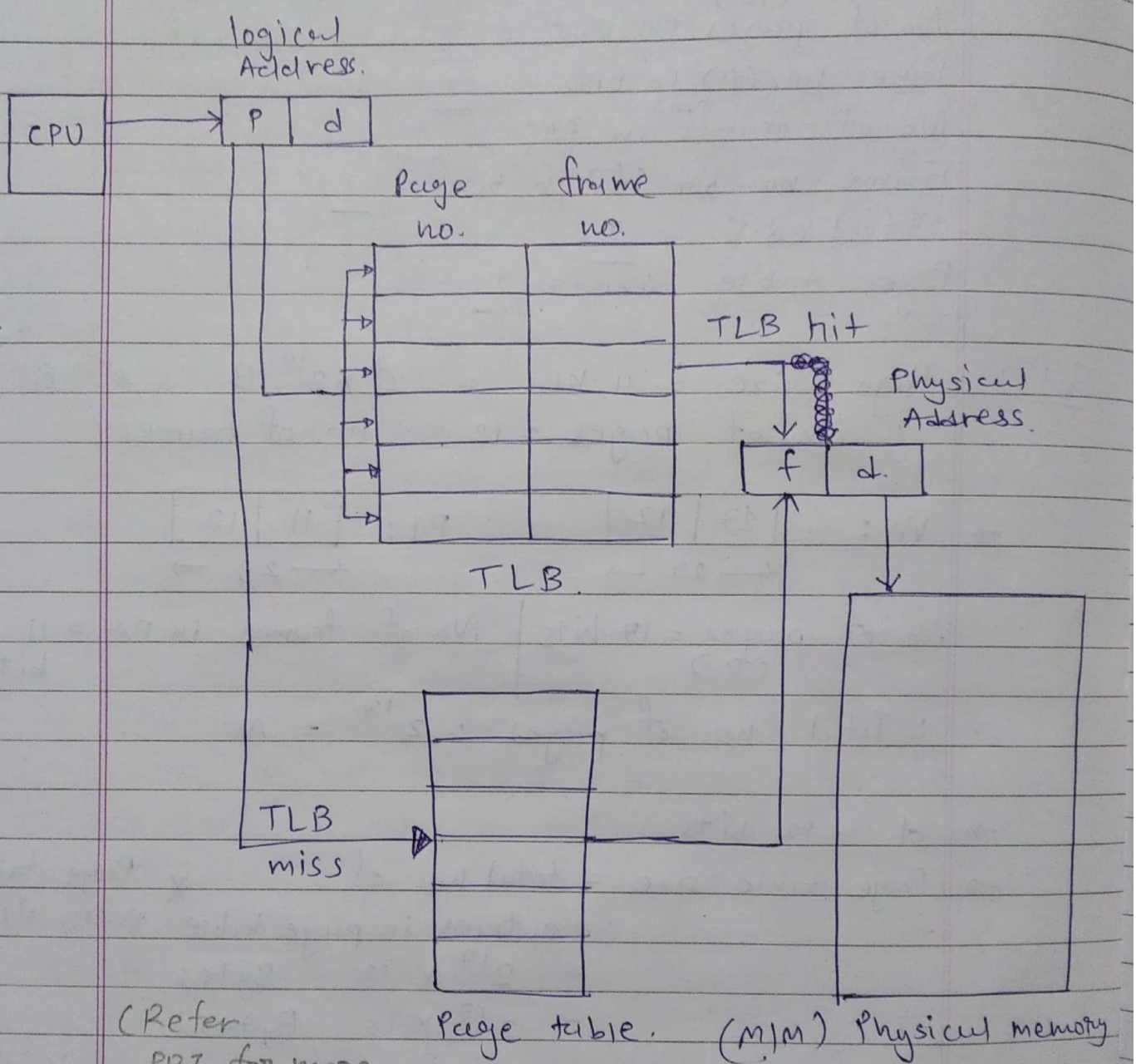
No. of pages (p) = 17 bits | No of frames in PA = 11 bits

∴ Total no. of pages = 2^{17} = N.

→ d = 12 bits.

→ Page table size = total no. of entries in page table × Page table entry size
= $2^{17} \times 12$ Bytes
= $2^{17} \times 16$ Bytes (∵ approxim)
= $2^{17} \times 2^4$
= 2^{21} Bytes
= 2 MB.

* Translation Lookaside Buffer (TLB) :-



(Refer PPT for more Information)

→ Hit ratio :- The percentage of times that a particular page number is found in the TLB is called the hit ratio.

Date
Page

** Ex: Main memory access time = m
TLB access time = c (c < m)
Hit ratio = x

→ TLB access time = 20 ns = c
Hit ratio = 95% = x
Memory access time = 100 ns = m

→ Effective Address Time = [x(c+m) + (1-x)(c+2m)]

OR

(p x hit memory time) + (1-p)(miss memory time)

→ Effective address time = x(c+m) + (1-x)(c+2m)

$= \left(\frac{95}{100}\right)(120) + \left(\frac{5}{100}\right)(220)$ ns

$= 114 + 11$

$= \boxed{125 \text{ ns}}$

$= \frac{107}{10} + \frac{110}{10}$

$= \frac{217}{10}$

$= 21.7$ ns

→ Effective address time without TLB = 2m

$= 2 \times 100$

$= \boxed{200 \text{ ns}}$

* TLB access time = 10 ns = c
Main memory access time = 50 ns = m
TLB hit ratio = 90 % = x
There is NO Page fault.

Sol^{n} → Effective memory access time = x (c+m) + (1-x) (c+2m)

$= \left(\frac{90}{100}\right)(10+50) + \left(\frac{10}{100}\right)(10+100)$

$= \left(\frac{9}{10} \times 60\right) + \left(\frac{1}{10} \times 110\right)$

= 54 + 11 ns

= 65 ns + (t) ← if page fault occurs

t — Page fault service time

** If page fault occurs that means page of a process is not present in the main memory. In this case, we have to transfer that page into main memory which takes time. which is called Page Fault Service Time.

→ Therefore, if TLB is present and page fault occurs then we have to add page fault service time extra.

* Main memory access time = 200 ns
Page hit ratio = 99.99 %
page fault service time = 8 ms
$= 8 \times 10^{-3}$ s
$= \frac{8 \times 10^{-3}}{10^{-9}}$ ns
$= 8000000$ ns

→ Effective Memory Access Time = p (page fault service time) + (1 - p) (Main memory access time)
+ swap page out
+ swap page in + Overhead

** p = probability of occuring page fault

→ p = probability of occuring page fault
$= 1 - 99.99$
$= 1 - \frac{99.99}{100}$

∴ $P = 0.0001 \Rightarrow 1 - p = 0.9999$

→ EMAT $= (0.0001)(8000000) + (200)(0.9999)$
$= \frac{1}{10000} \times 8000000 + 200 \times \frac{9999}{10000}$
$= 800 + 199.98$
$= \boxed{999.98 \text{ ns}}$

* Page Replacement Algorithms :-

* FIFO (First In First Out) :-

→ The frame which was occupied at first shold be vacant first.

→ Ex:-

(1) Reference string : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

3 frames :-

	3	3	3	2	2	2	2	4	4
2	2	2	1	1	1	1	3	3	3
1	1	4	4	4	5	5	5	5	5
*	*	*	*	*	*	*	*	*	

Total 9 page faults.

4 frames :-

		4	4	4	4	4	3	3	3
	3	3	3	3	3	2	2	2	2
2	2	2	2	2	1	1	1	5	5
1	1	1	1	1	5	5	5	4	4
✓	✓	✓	✓		✓	✓	✓	✓	✓

Page fault count →

Total 10 page faults.

For 4 frames Examples

* → Hit Ratio = $\frac{\text{No. of Hits}}{\text{Total References}}$

$= \frac{2}{12} \times 100 = 16.66\ \%$

* → Miss Ratio = $\frac{\text{No. of Miss}}{\text{Total References}}$

$= \frac{10}{12} \times 100 = 83.33\ \%$

** Belady's Anomaly :-

→ It is related to FIFO only.

→ In the above example,

Frames	Hit	Miss or faults
3	3	9
4	2	10

→ Number of frames ↑ => Page faults ↑

** Optimal Page Replacement Algorithm :-

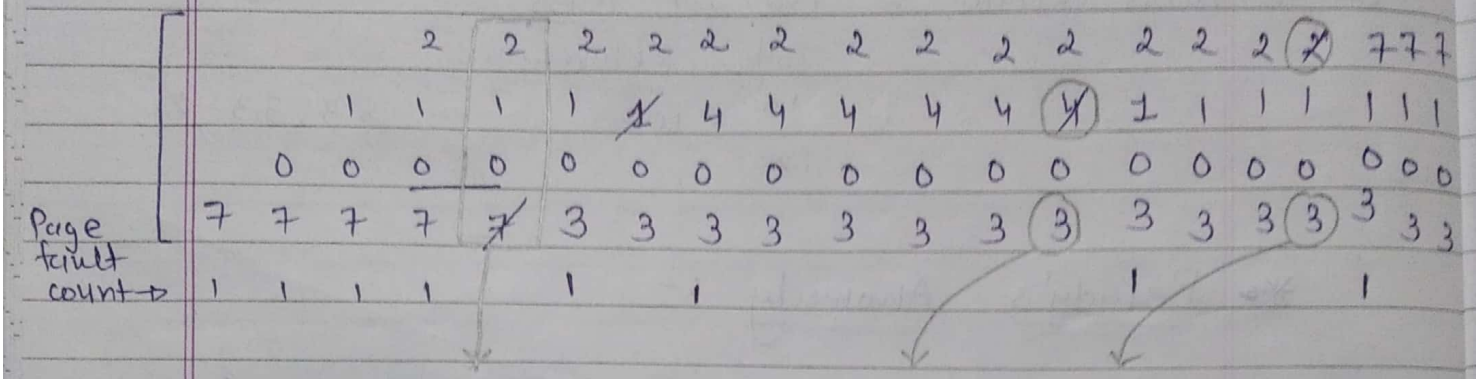
→ Replace the page which is NOT used in longest dimension of time in future.

→ If two pages are not used in future the any of them can be replaced by a new page.

→ Ex:- Ref. string,

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

4 frames :-



In this case, 7 is the lastest.
∴ Replace it.

In this case, both are not in future.
∴ select any one.

∴ Total page fault : 8 and Hit = 12.

→ Hit Ratio = $\frac{12}{20.}$ = 0.6

Miss Ratio = $\frac{8}{20.}$ = 0.4

* Least Recently Used Page Replacement Algorithm :-

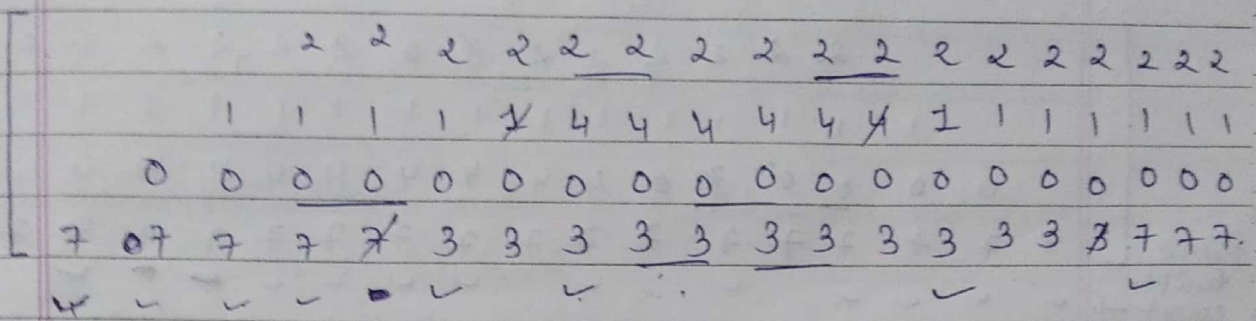
→ Replace the least recently used page in past.

→ Replace the page which was used very firstly. Check the past.

→ Ex:- Ref. string,

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

4 frames :- ← ↑ ← ↑ ← ↑ ← ↑



→ Total page faults = 8

→ Total Hits = 12

(Can compare to FIFO)

→ Speed decreases as it searches for least recently used page.

* MOST RECENTLY USED PAGE REPLACEMENT ALGORITHM :-

→ Replace the most recently used page in past.

→ Replace the page which was used lastly. Check in the past.

→ Ex:- Ref. string :-

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

4 frames :-

			2	2	2	2	2	2	3	0	3	2	2	2	0	0	0	0	0
		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	0	0	0	0	3	0	4	4	4	4	4	4	4	4	4	4	4	4	4
Page	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
fault count →	✓	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓	●	●	✓			

→ Total page faults = 12 , Hits = 8

* Hard Disk Architecture :-

→ Platters → Surface → Track → Sectors → Data

→ Disk size = P X S X T X S X D.

→ Ex:-

(1) No of platters = 8
No. of track = 256
No. of sectors = 512
Data = 512 KB.

→ Disk size = Platter x Surface x Track x Sectors x data

$= 8 \times 2 \times 256 \times 512 \times 512$ KB.

$= 2^3 \times 2^1 \times 2^8 \times 2^9 \times 2^9 \times 2^{10}$ B

$= 2^{40}$ Bytes

= 1 TB.

→ No of bits required to represent disk size = 40.

(2) Total cylinders = 480
cylinder zones = 120

Disk size

each contains different sizes
200, 240, 280, 320.
each sector contains 4096 Bytes.

→ 120 x 200 + 120 x 240 + 120 x 280 + 120 x 320.
= 124800 ← total no. of sectors.
→ each sector is of 4096 Bytes.
∴ size of disk = 124800 x 4096 = 511180800 BYTES

* Disk Access Time :-

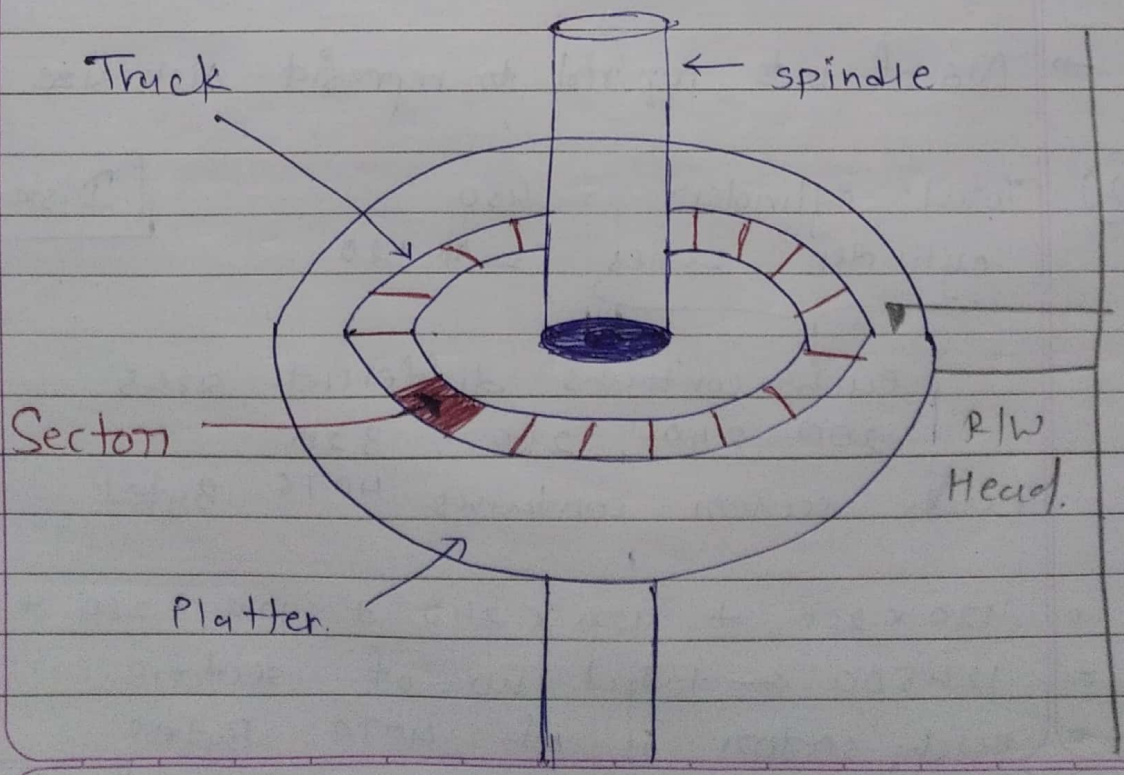
1) Seek time :- Time taken by R/W head to reach desired track.

2) Rotation time :- Time taken for one full Rotation (360°)

3) Rotational latency :- Time taken to reach to desired sector. (half of Rotation Time)

4) Transfer Time :- $\frac{\text{Data to be transfer}}{\text{Transfer Rate}}$

Transfer Rate or Data Rate = (No of Heads ↑ Surface × Capacity of one track × No. of Rotations in one second)



→ Disk Access Time = Seek time + Rotational latency + Transfer Time + Control time + Queue time.

* Disk Access Time = Seek time + Rotational latency + Transfer time

* Disk Scheduling Algorithms :-

→ To minimize (seek time)
→ Seek time ≈ seek distance.

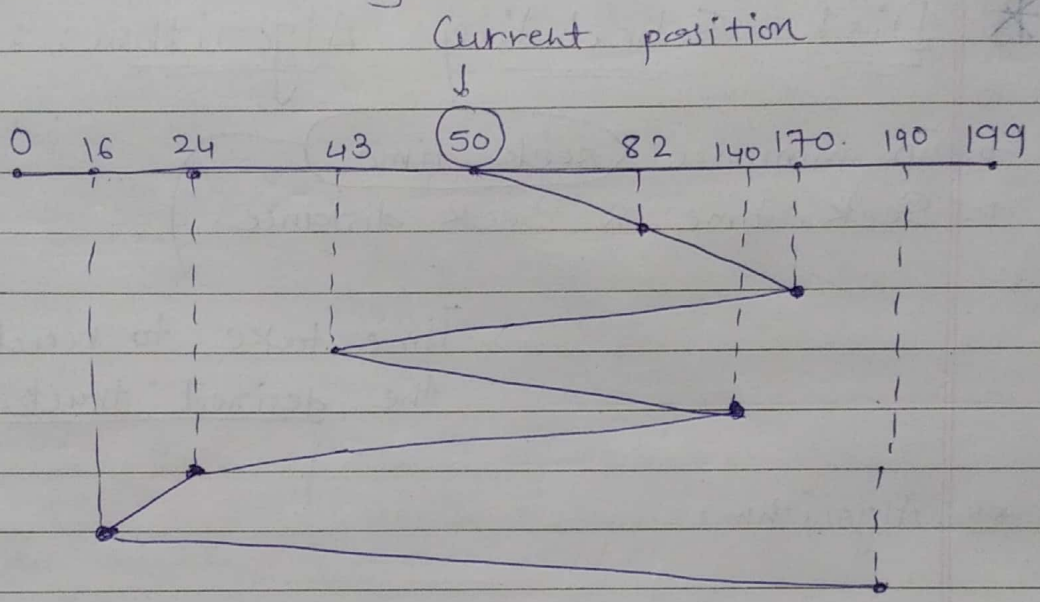
Time take to reach to the desired track.

** Algorithms :-

→ FCFS [First Come First Serve]
→ SSTF (Shortest Seek Time First]
→ SCAN
→ LOOK
→ CSCAN [Circular Scan]
→ CLOOK [Circular Look]

* FCFS : (First Come First Serve)

Ex ** A disk contains 200 tracks (0-199).
Request queue contains track no.
82, 170, 43, 140, 24, 16, 190 respectively.
Current position of R/W head = 50.
Calculate total no of tracks movement by R/W head using FCFS.



→ Total no of track movements or seek time

= (82 - 50) + (170 - 82) + (170 - 43) +
(140 - 43) + (140 - 24) + (24 - 16) +
(190 - 16)

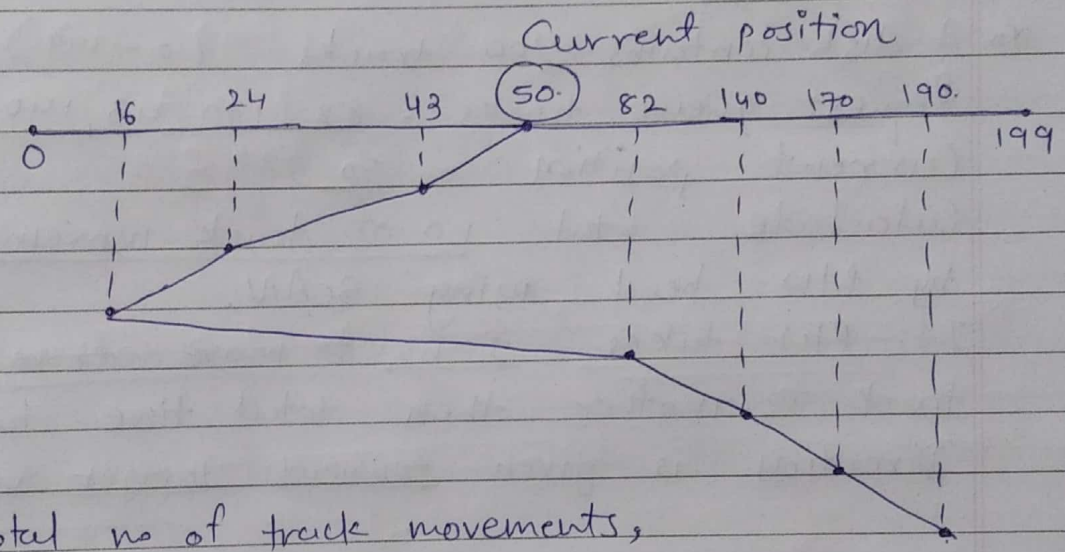
= 32 + 88 + 127 + 97 + 116 + 8 + 174
= 642

→ Advantages :- → No starvation
→ Performance decreases

* SSTF : (Shortest Seek Time First)

Ex: A disk contains 200 tracks (0 - 199). Request queue contains 82, 170, 43, 140, 24, 16, 190 respectively. Current position of R/W head = 50. Calculate the total no of track movement by R/W head using shortest seek time first. If R/W head takes 1 ns to move from one track to another then total time taken is ___.

→ Shortest Seek Time should be considered First



→ Total no of track movements,

= (50 - 16) + (190 - 16) OR

= (50 - 43) + (43 - 24) + (24 - 16) + (82 - 16) + (140 - 82) + (170 - 140) + (190 - 170)

= 34 + 174

= 208

= 7 + 19 + 8 + 66 + 58 + 30 + 20.

= 208.

→ Total track movement = 208.
1 ns is taken for one track movement
∴ total 208 ns required

→ Advantage :-

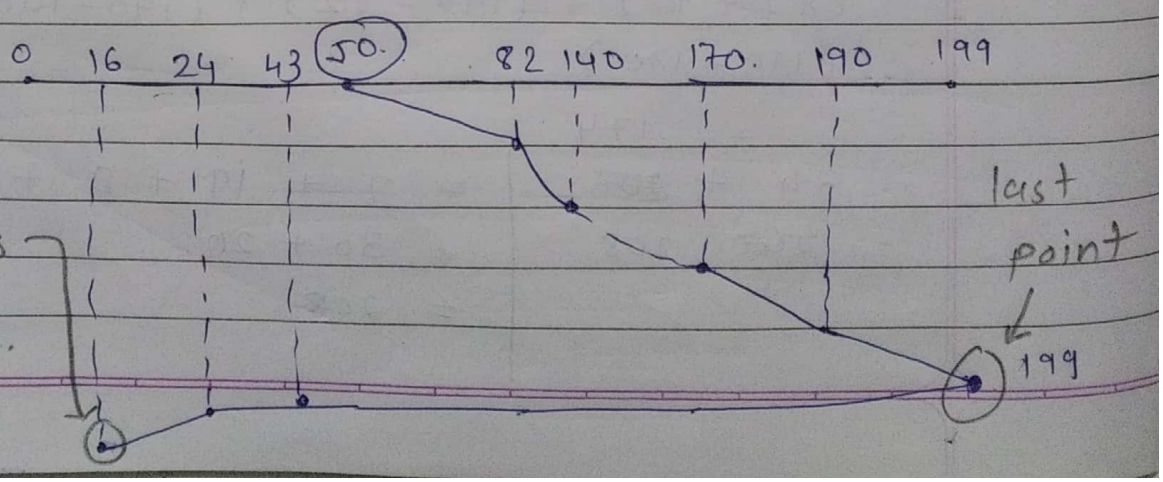
→ Tries to give optimal result. (Average case).
→ Response time increases.

→ Disadvantage :- → Starvation

* SCAN :- (Elevator)

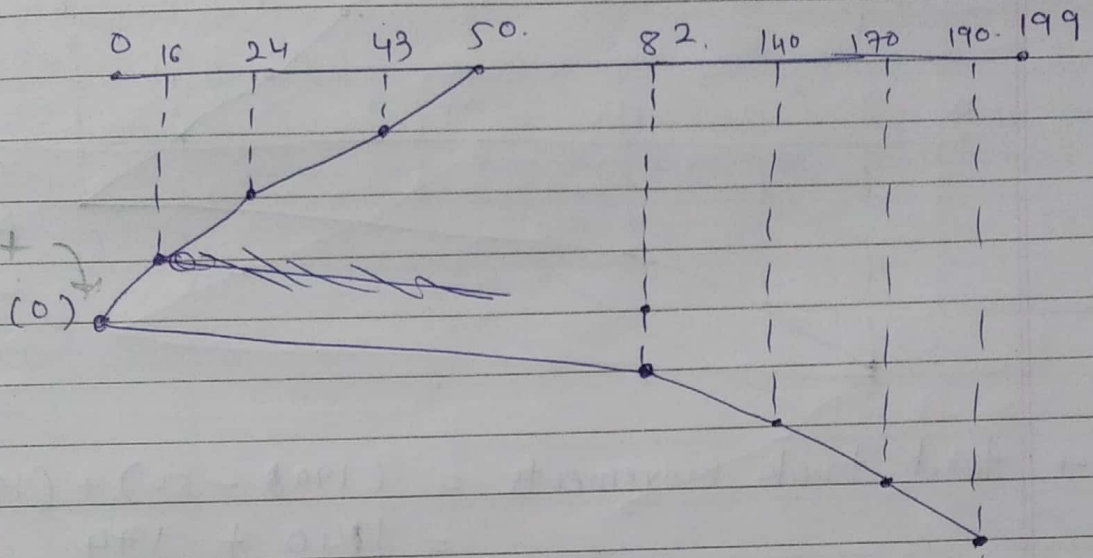
Ex:- ** A disk contains 200 tracks (0 - 199).
Request queue tracks : 82, 170, 43, 140, 24, 16, 190
Current position = 50
Calculate total no of track movement by RW head using SCAN.
If RW takes 1 ns to move from one track to another then total time take ___ ?
Direction is given towards larger value.

→ Move until you reach to the last point of any direction. (regardless of request queue tracks.) one



→ Total no. of tracks = (199 - 50) + (199 - 16)
= 149 + 183
= 332.

** If direction is given towards smaller value,



Ans = (50 - 0) + (199 - 0)
= 249

→ 249 ns

→ Disadvantage :-

→ When request queue tracks changes dynamically, starvation occurs.

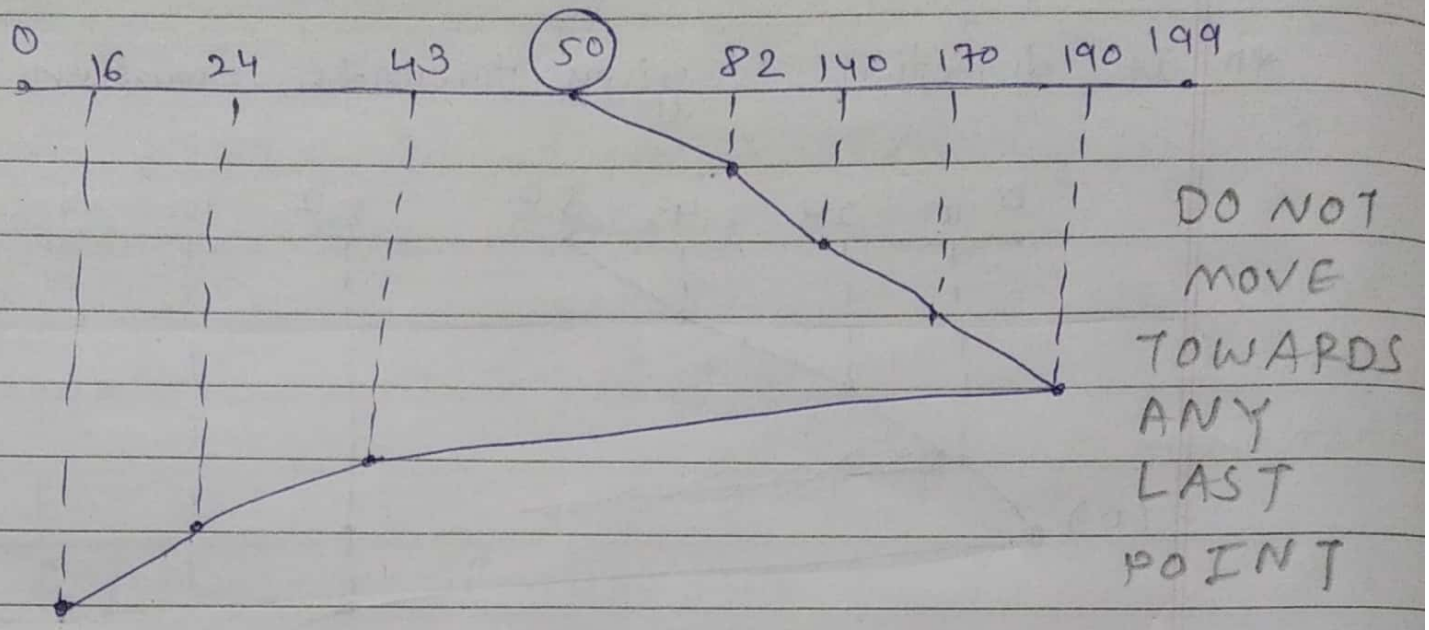
* LOOK :-

** LOOK algorithm is same as SCAN algorith but LOOK algorithm does NOT move towards the end point

Ex:- Request queue track: 82, 170, 43, 140, 24, 16, 190

Current pos :- 50

total 200 tracks, direction = larger value



→ total track movements = (190 − 50) + (190 − 16)

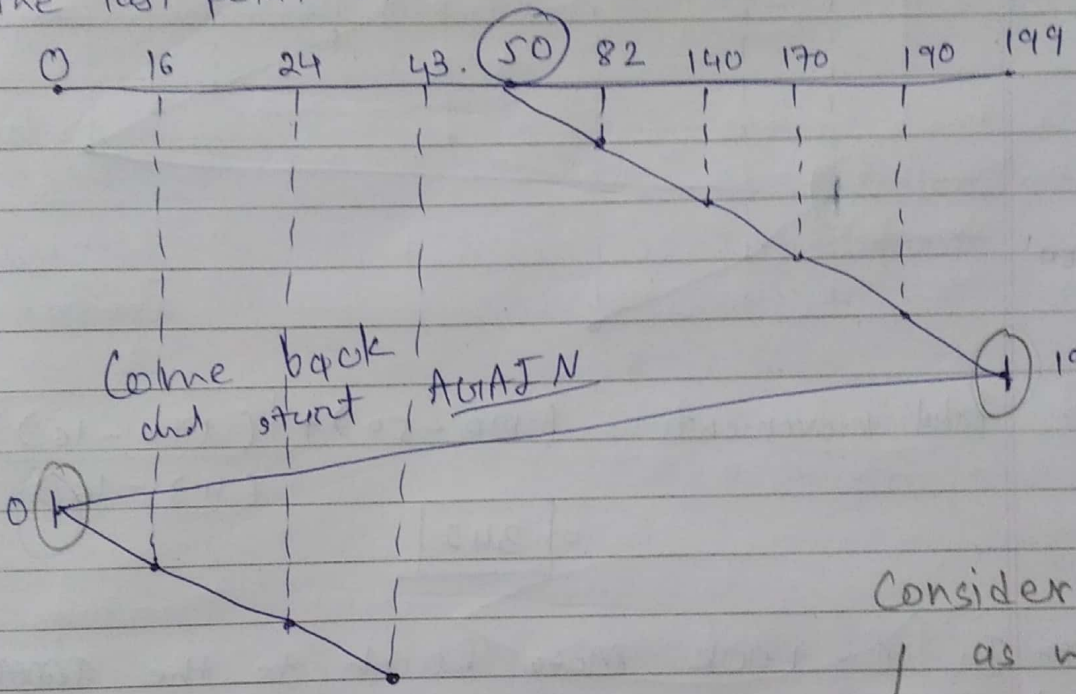
= 140 + 174

= 314

* C-SCAN : Circular SCAN.

Ex:- → Request queue track = 82, 170, 43, 140, 24, 16, 190
Current pos = 50
total 200 track, direction towards larger value.

→ Reach to the last point of any one direction (which is firstly selected) and then reach to the last point of another direction and start again.



Consider this as well

→ total movem = (199 - 50) + (199 - 0) + (43 - 0)

= 391

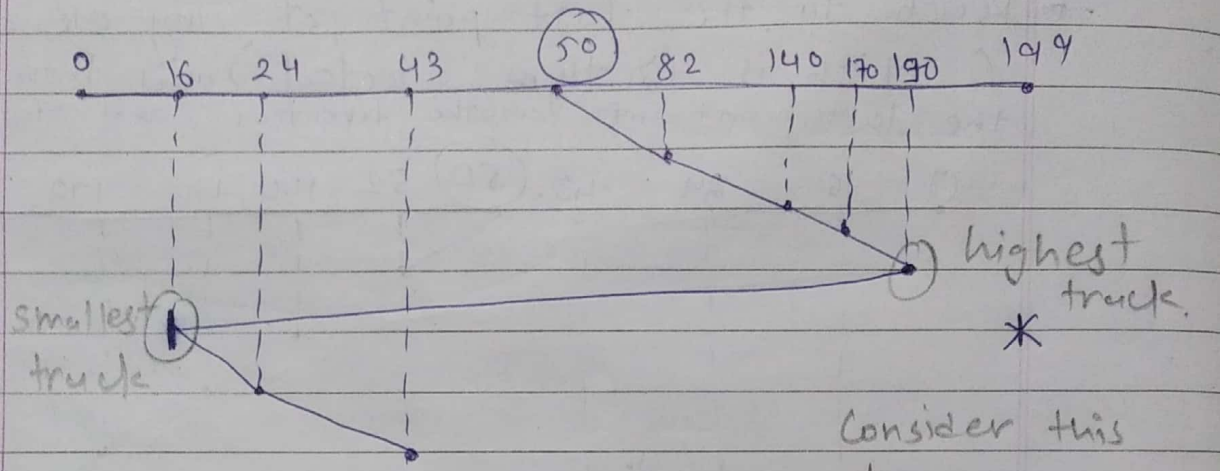
* C - LOOK : Circular LOOK

Ex:- 200 track

Request queue track = 82, 170, 43, 140, 24, 16, 190.

Current pos = 50.

direction towards large value.



total movements = (190 - 50) + (190 - 16) + (43 - 16)

= 341

** In C - LOOK Reach to the highest track and then come back to the lowest/smallest track and start again.

** If direction is smaller, then reach to the smallest no. track and then go to the higest no. track and start again.

File System :-

→ User → file → folder/directory → file system

How the data is stored
How the data is fetched

** Operations on files.

1) Creating
2) Reading
3) Writing
4) Deleting
5) Truncating
6) Repositioning

** File Attributes

1) Name
2) Extension (Type)
3) Identifier
4) Location
5) Size
6) Modified date, created date.
7) Protection / permission
8) Encryption, compression.

→ File → delete → File + Attribute (metadata)

will be deleted

→ File → Truncate → File + Attribute (metadata)

only file will be deleted.

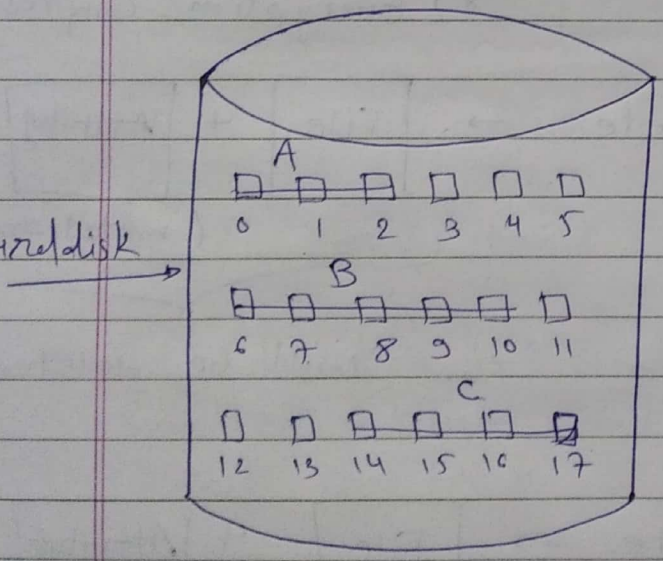
* Allocation Methods in File system :-

- Contagious Allocation
- Non Contagious Allocation
  - Linked list allocation
  - Indexed allocation.

→ Purposes :

1) Efficient disk utilization
2) Access faster

* Contagious Allocation :-



Directory

file	start	end
A	0	3
B	6	5
C	14	4

→ Advantages :

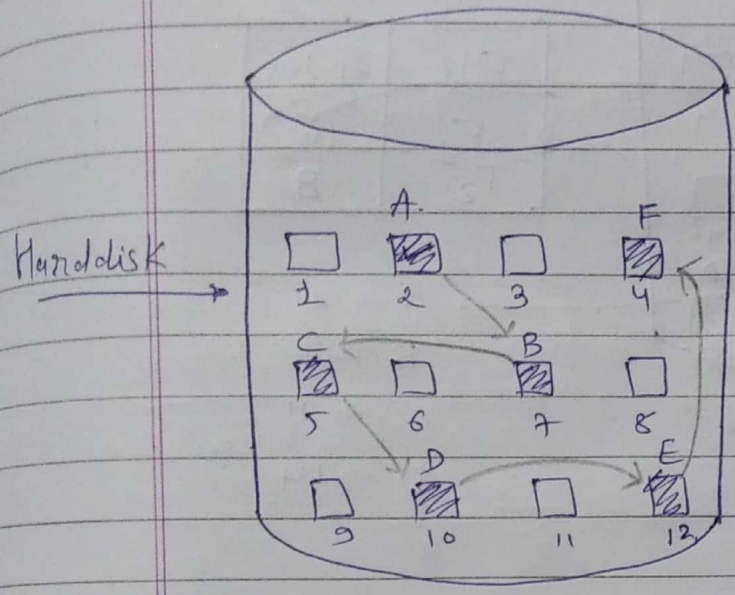
1) Easy to implement.
2) Excellent Read performance

→ Disadvantages :-

1) Disk will become fragmented (Internal & External fra.)
2) Difficult to grow file.

* Non-contagious Allocation :-

(1) Linked-list allocation :-



Directory

File	start
F_1	2
F_2	6

A
B
C
D
E
F

→ In linked list allocation we will have data and the pointer which is pointing to the next partition.

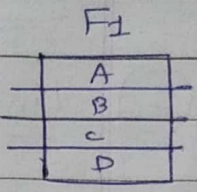
Data
Pointer

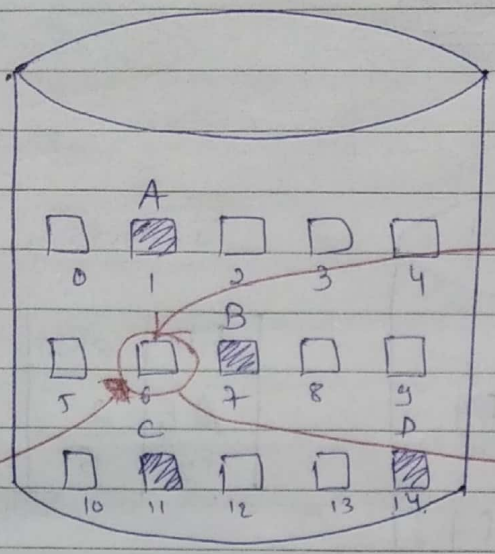
→ In last partition pointer will point to -1.

→ Advantages :- 1) No External fragmentation
2) File size can increase.

→ Disadvantages :- 1) Large seek time
2) Random access / direct access is difficult
3) Overhead of pointers

(2) Indexed allocation :-





Directory

File	Index
F_1	6
F_2	8

A → 1
B → 7
C → 11
D → 14

This block will store the index of all the partitions.

→ Advantages :

1) Supports direct access.
2) No external fragmentation.

→ Disadvantages :

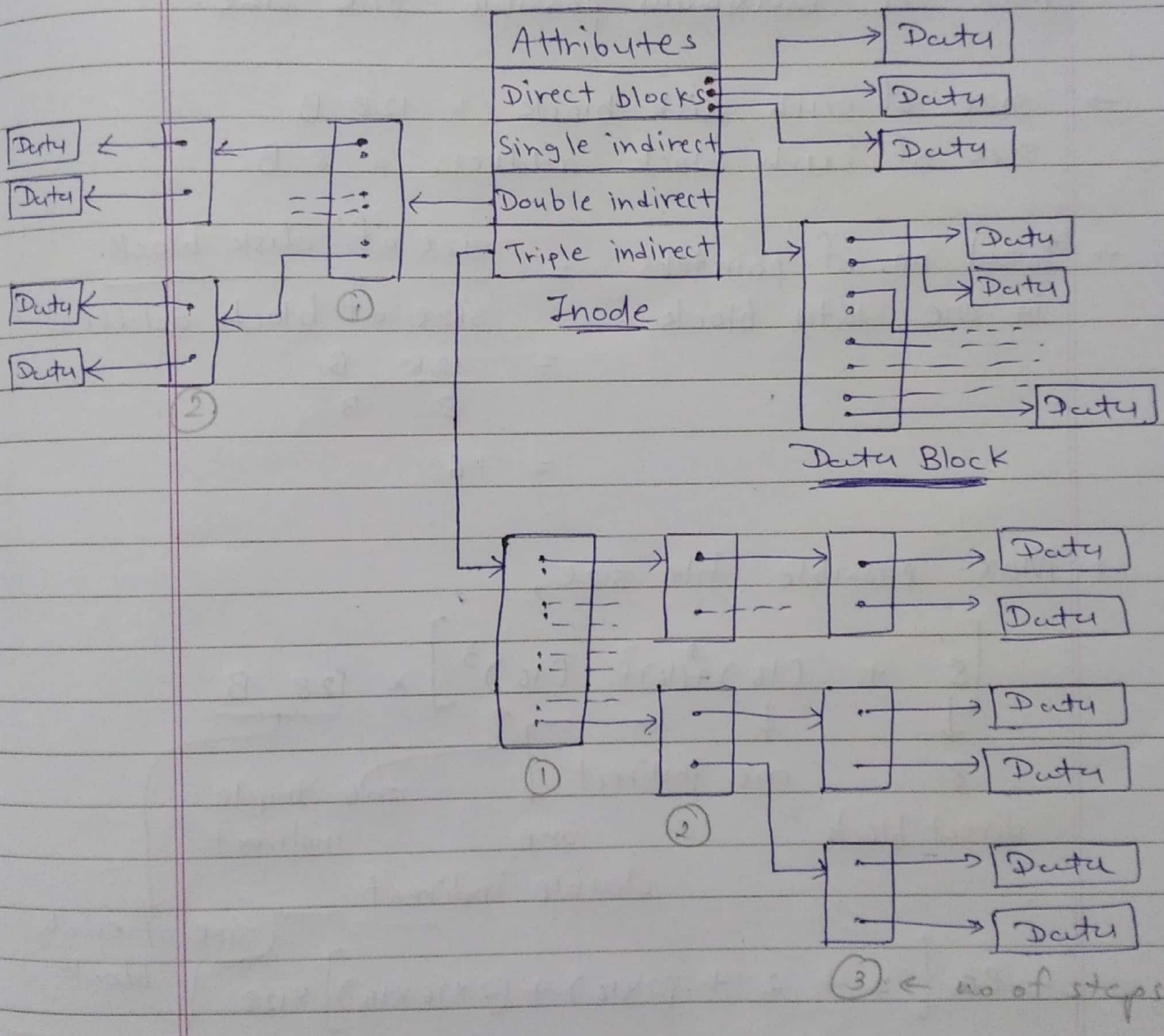
1) Pointer overhead
2) Multilevel index

* Unix Inode Structure :-

↓ ↓

OS File system

i → index , nod → block



Que: A file system uses UNIX Inode data structure which contains 8 direct block addresses, 1 indirect block, 1 double and 1 triple indirect block. The size of each disk block is 128 B and size of each block address is 8 B. Find the maximum possible file size.

→ size of each disk block = 128 B
size of each block address = 8 B

→ Total no. of pointers in one data block = $\frac{\text{size of disk block}}{\text{size of block address}}$

$= \frac{128\ B}{8\ B}$

$= 16$

→ Max. possible file size,

$$\left[8 + (16) + (16)^2 + (16)^3\right] \times 128\ B$$

8 direct blocks; one indirect; one double indirect; one triple indirect; size of each block

$= 8 \left[1 + 2 + (2 \times 16) + (2 \times 16 \times 16)\right] \times 128$

$= 2^3 (1 + 2 + 32 + 512)\ 2^7$

$= 2^{10} (547)$

$= \boxed{547\ KB}$