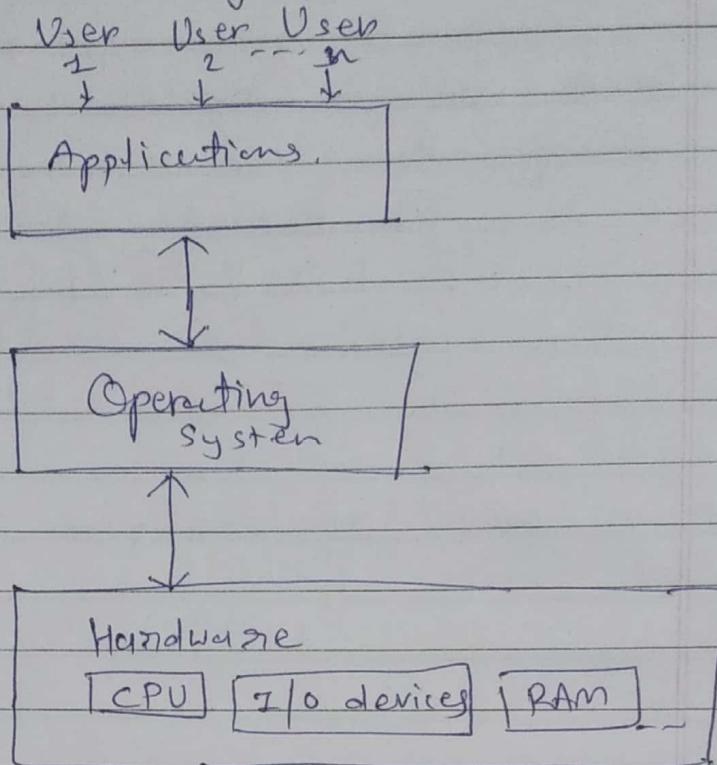


Operating System Notes

- By Rushikesh Pathan.

Date _____
Page _____

* Operating System Diagram :-



→ Throughput : no of tasks executed per unit time.

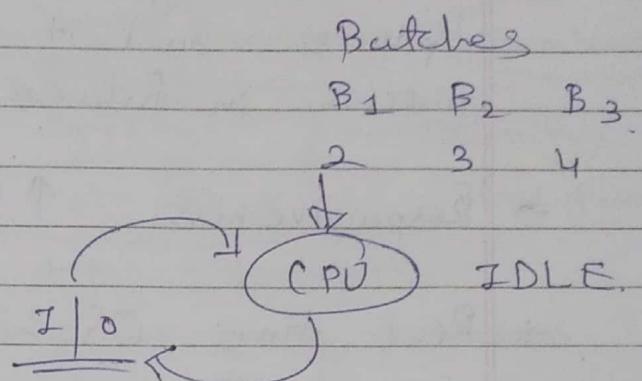
Ex:- Linux

→ 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

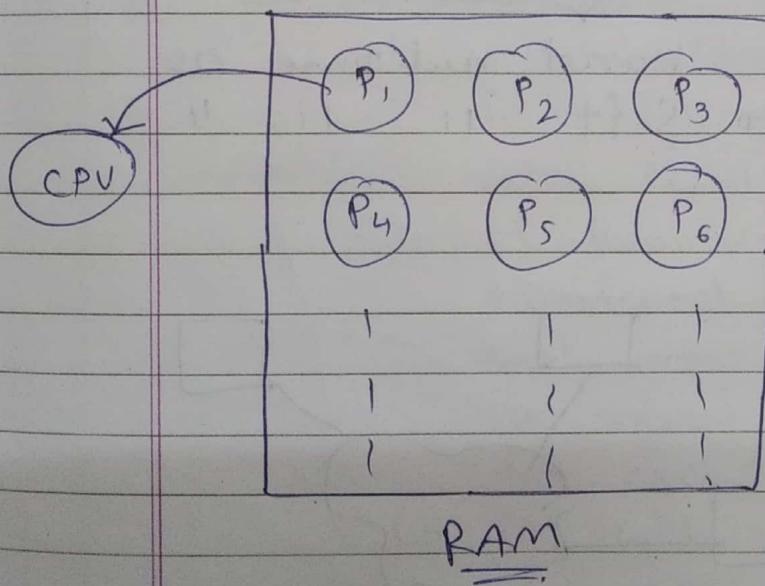
* Types of OS :-

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



* Multiprogrammed OS :-

(Non-preemptive)



→ When process P_1 is gone into CPU 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 .

→ Folliness ↓
of CPU.

* Multi tasking / Time sharing OS :-

→ Multi tasking OS (Preemptive)

is preemptive which means 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

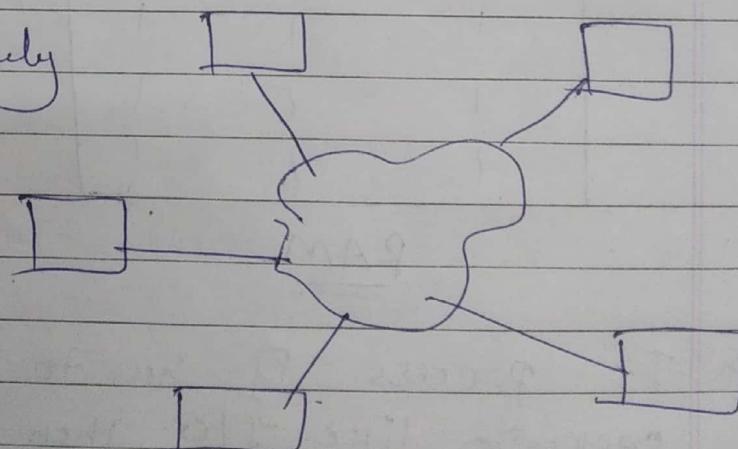
→ delay will not be there.

→ Ex:- Used in missile, plane, ...

Hard real time OS
Gaming live → Soft || || ||

* Distributed :-

→ Environmentally

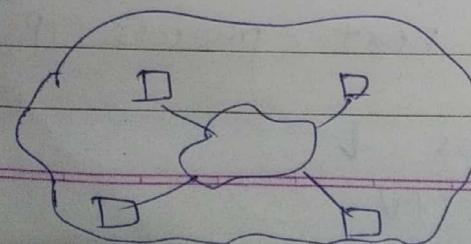


Distributed Environment

* Clustered :-

→ Availability ↑

→ Load balance

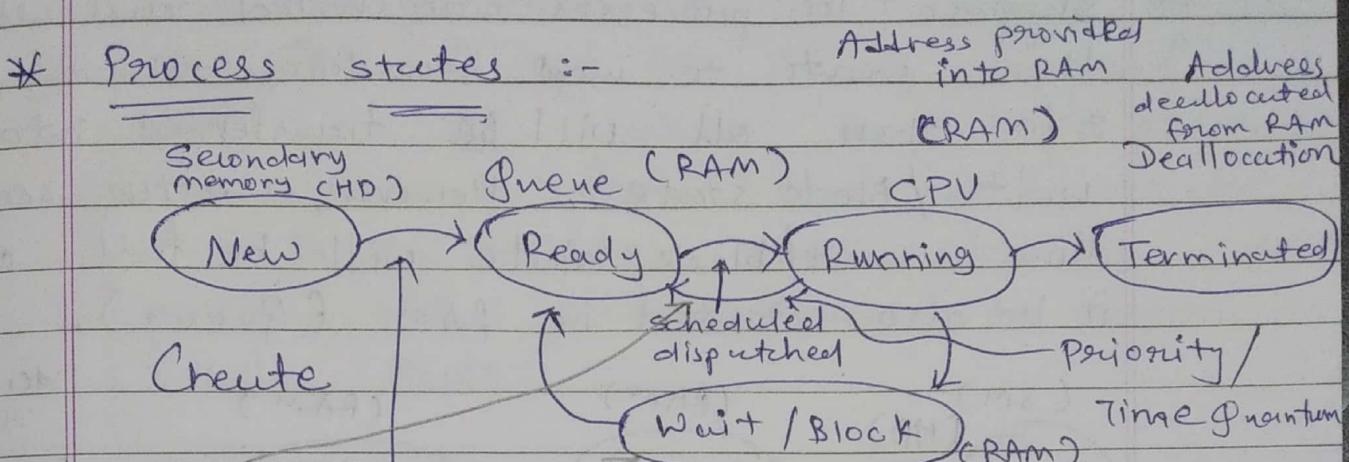


→ Computational power ↑

* Embedded :-

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

* Process states :-



LTS : Long term scheduler.

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

↳ Used 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 transferred back to Ready state and most priority process will be moved to running state for execution.

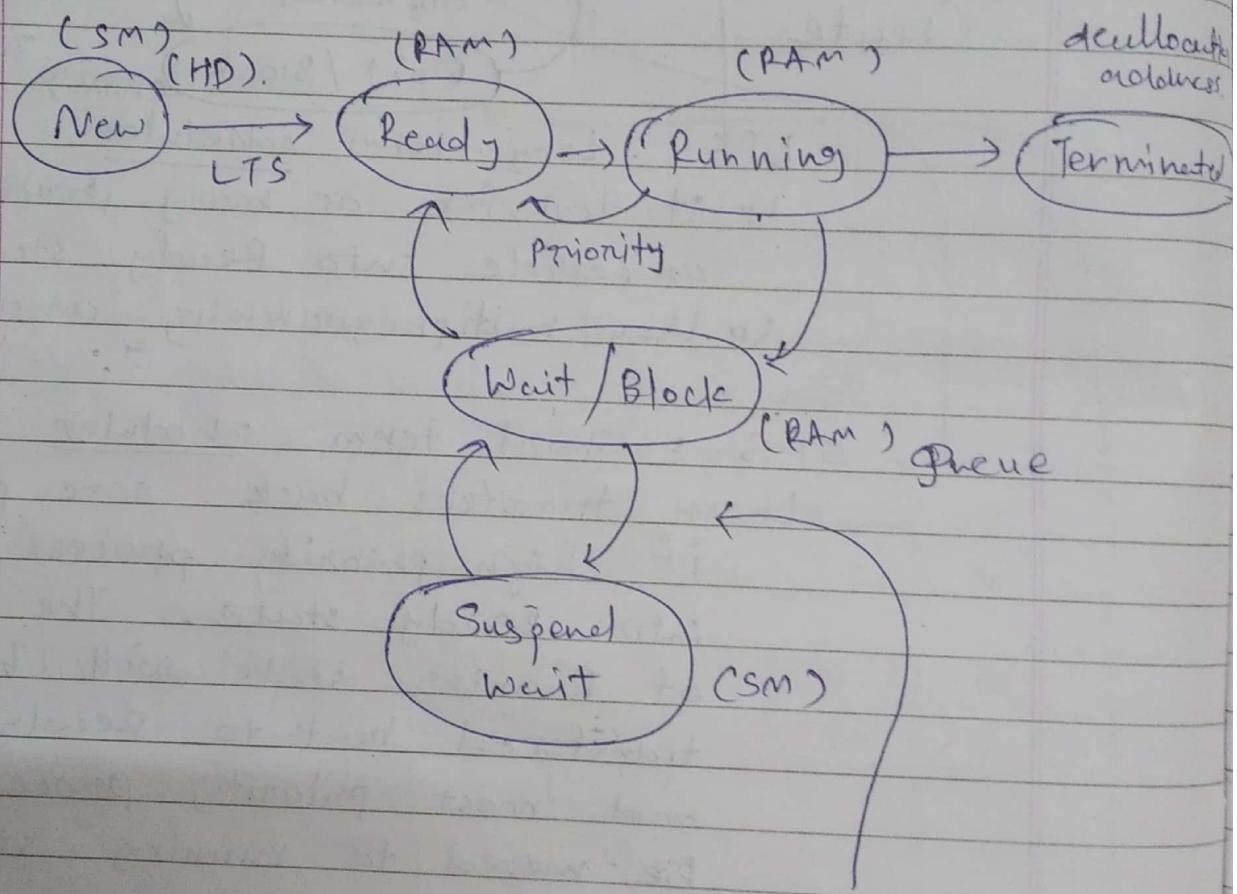
↳ It will lead to preemptive scheduling.

↳ Used multi tasking concept.

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

* Worst case -

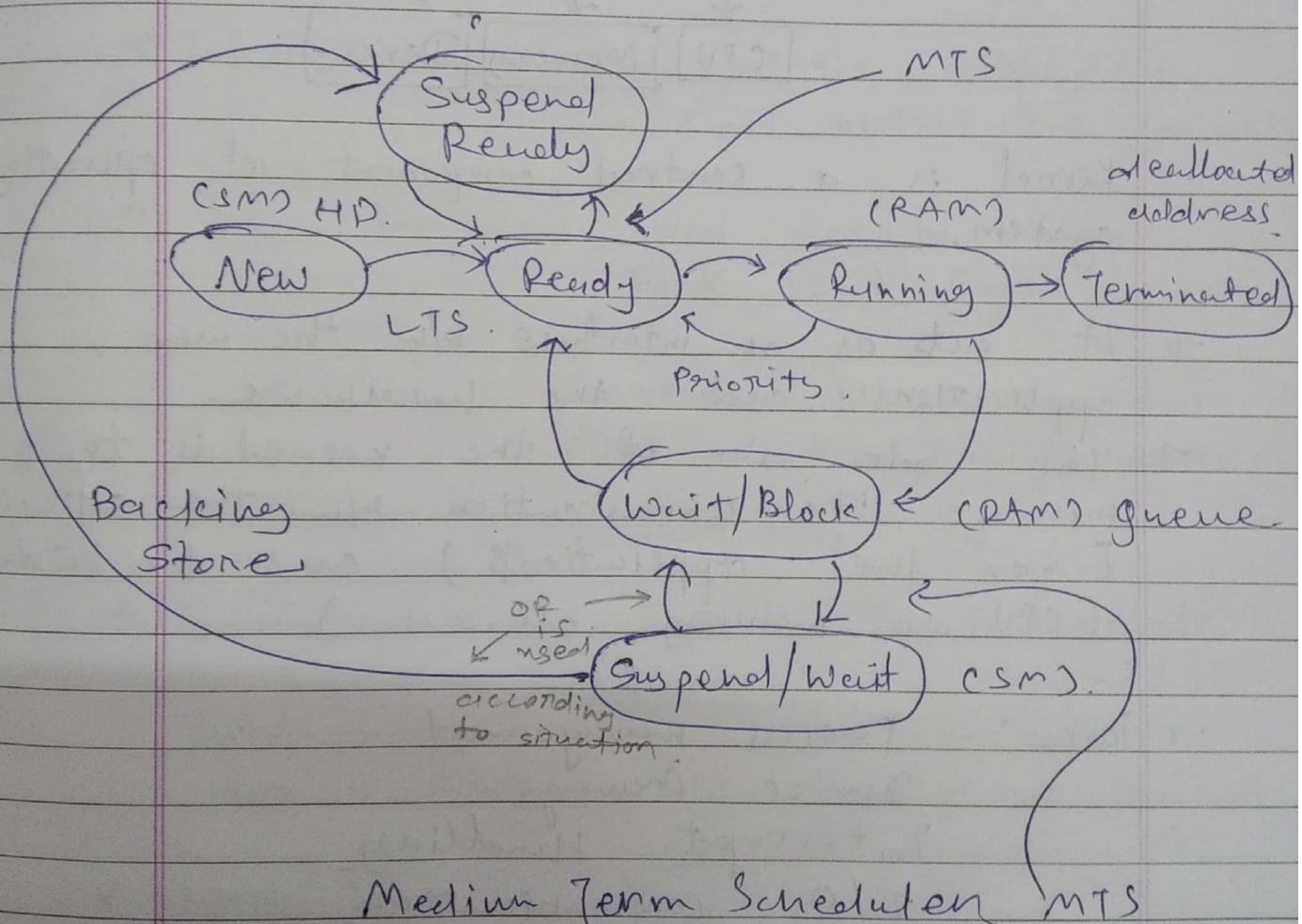
- Suppose 10 processes are created and all of them wants to used secondary memory or E/I/O then all will be transferred into wait / Block state. However, after some time wait/block state will be full as it is also situated in RAM (Queue).



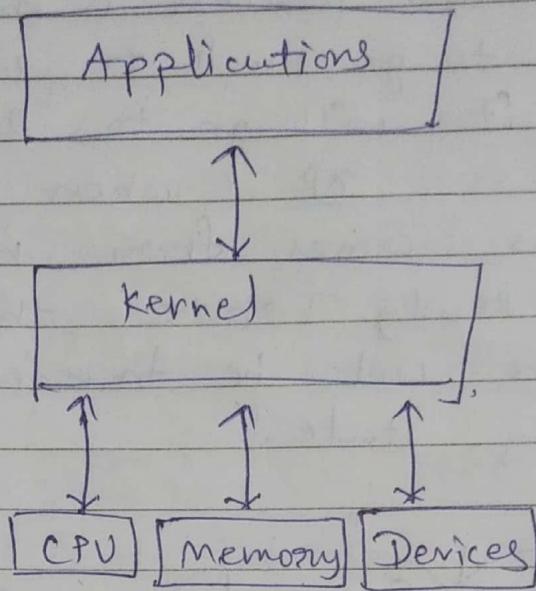
MTS .

- 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`, `Signtal`, `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() :-

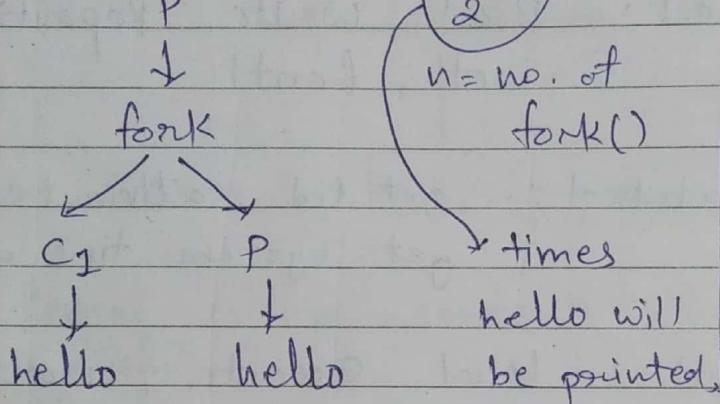
- To create a child process.
- Returns : 0 - child ✓
 ↴
 any positive integer → +1 - Parent / process itself.
 ↴
 -1 - child is not created.

→ Ex:-

```

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

```



→ Number of

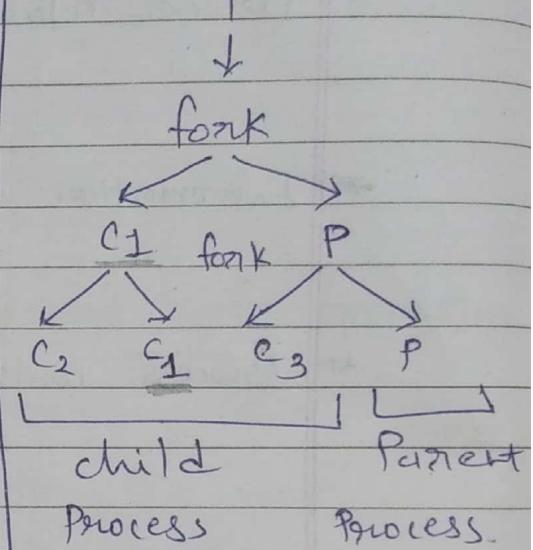
$$\text{child processes} = \underline{\underline{2^n - 1}}$$

int main()

```

{
    fork();
    fork();
    printf("hello");
    return 0;
}

```



→ Ex:-

```

int main()
{

```

```

    if(fork() && fork())

```

{

```

        fork();
    }

```

```

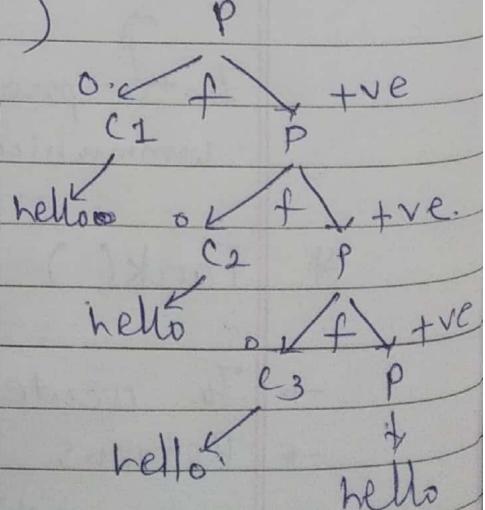
    printf("hello");

```

```

    return 0;
}

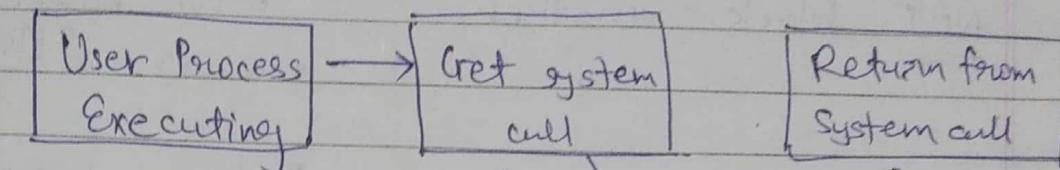
```



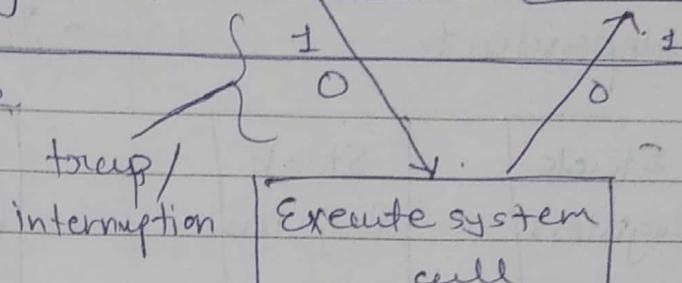
* User Mode and Kernel Mode :-

User Mode

Mode bit = 1



Kernel Mode,



Mode bit = 0.

→ 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 processes have different copies of stack, data, code, files.

→ Threads share same copy of code and data. (Registers & stack will be different)

ProcessThread

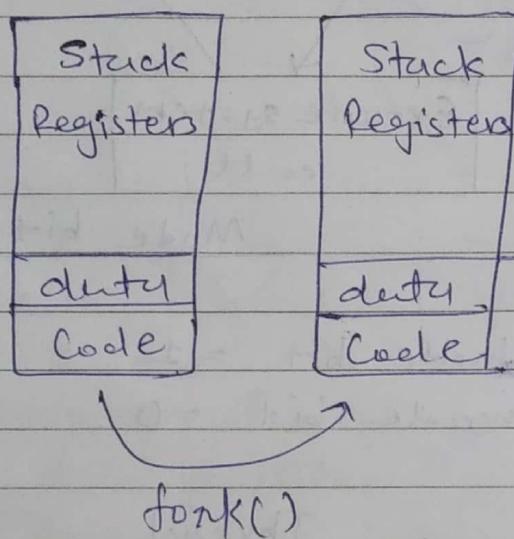
* → Context switching is slower.

→ 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~~ Thread has to ~~be~~ 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

- User level threads are managed by User level library.
- User level threads are typically fast.
- Context switching is faster.
- If one user level threads perform blocking operation then the entire process gets blocked.

Kernel Level Thread

- kernel level threads are managed by OS system calls.
- kernel level threads are slower than user level.
- Context switching is comparatively slower. (-: works with OS system calls)
- If one kernel level thread is blocked, no affects on another.

Context Switching Time :

Process > KLT > ULT
 ↓ ↓
 Kernel User level
 level thread, thread

* Scheduling Algorithms :-

- It is a way of selecting a process from Ready Queue and put it 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 stop transfer back 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 duty 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 duty then it will go back to ~~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-preemptive

- * → 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)

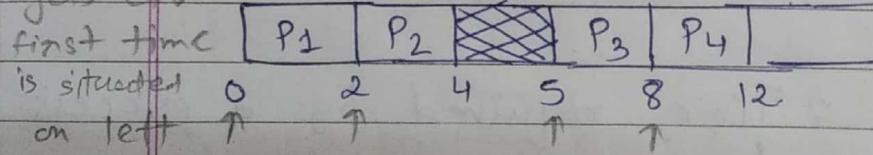
→ Response Time = $\{ \begin{array}{l} \text{The time at which} \\ \text{a process gets CPU first time} \end{array} \} - \text{Arrival time}$

= The time at which a process gets CPU - Arrival Time.
 First Time.

* FCFS : First Come First Serve
 (Criteria: Arrived Time (Non-preemptive) ($WT = RT$))

Process No.	Arrived 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



$$\text{Avg TAT} = 14/4 = 3.5$$

$$\text{Avg. WT} = 3/4 = 0.75$$

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

→ (WT) Waiting Time = TAT - Burst Time

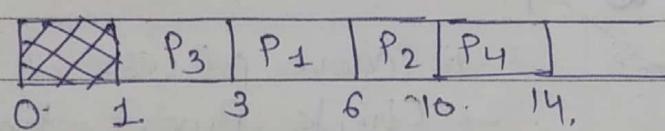
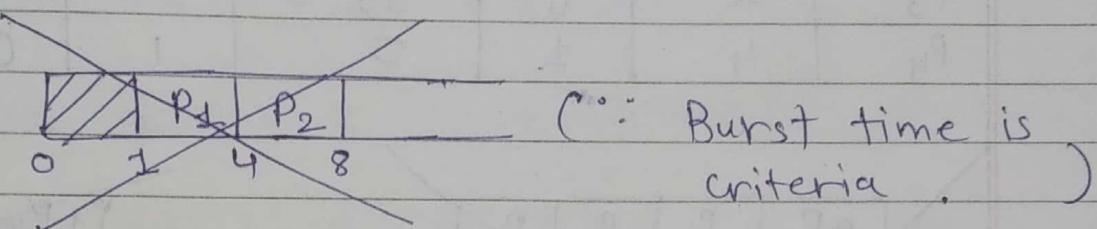
→ (RT) Response Time = The time at which a process gets CPU first time - Arrival time.

*

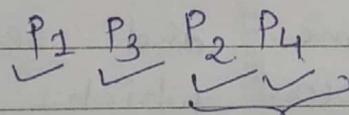
SJF = Shortest Job First
Burst time (Non-preemptive)

Criteria:

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



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



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

Process Burst time is same.

∴ go for arrival time and put accordingly.

$$\rightarrow \text{TAT} = \frac{\text{CT} - \text{AT}}{\text{Time}} = \frac{\text{Completion Time} - \text{Arrival Time}}{\text{Time}}$$

$$\rightarrow \text{WT} = \text{TAT} - \text{BT} = \frac{\text{Turn Around Time}}{\text{Time}} - \frac{\text{Burst Time}}{\text{Time}}$$

→ RT = The time at which process gets CPU - Arrival Time

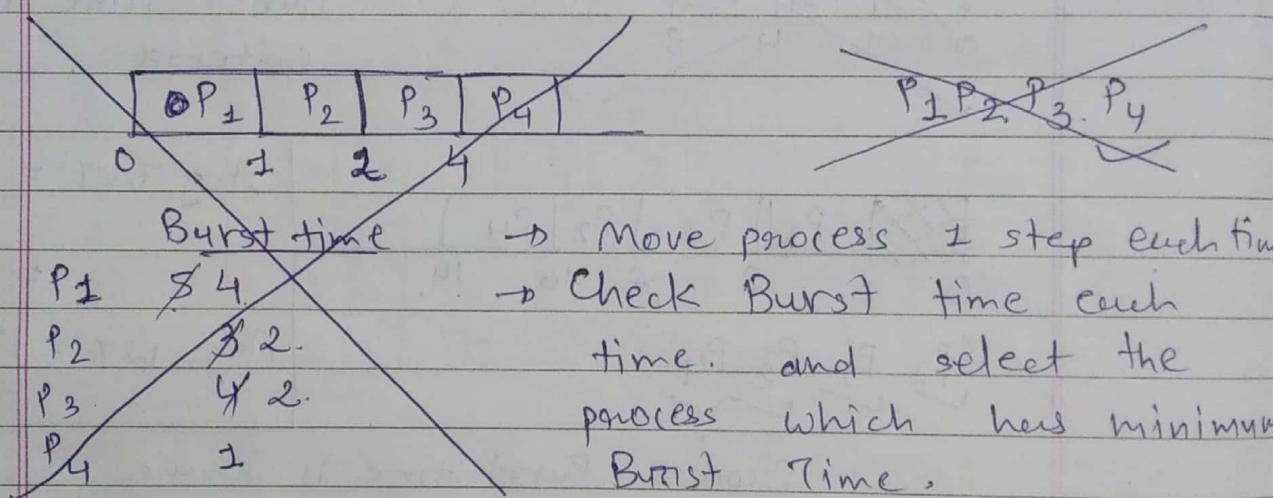
* Shortest Remaining Time, First :-

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



P ₁	P ₂	P ₂	P ₂	P ₄	P ₁	P ₁	P ₁	P ₁	P ₃
0	1	2	3	4	5	6	7	8	9

↑

Process	Burst Time
P ₁	5
P ₂	3
P ₃	4
P ₄	1

Last value should be checked when we want to calculate completion time. Here, P₁ is completed at 9.

$$\rightarrow TAT = CT - AT$$

= Completion Time - Arrival Time

$$\rightarrow WT = TAT - BT$$

= Turn Around Time - Burst Time

$\rightarrow RT =$ The time at which process gets CPU first time - Arrival time

$$\rightarrow \text{Avg. TAT} = \frac{24}{4} = 6$$

$$\rightarrow \text{Avg. WT} = \frac{11}{4} = 2.75$$

$$\rightarrow \text{Avg. RT} = \frac{7}{4} = 1.75$$

Ques:

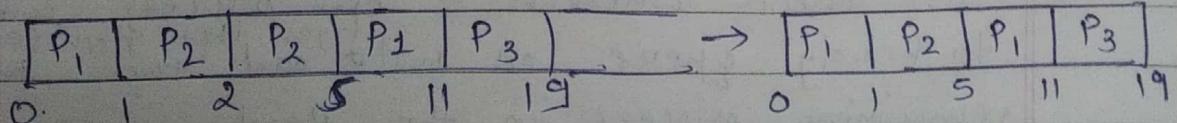
Process No.	Arrival time	Completion time	TAT	WT	RT
P ₁	0	11	11	4	0
P ₂	1	5	4	0	0
P ₃	2	19	17	9	9

\rightarrow Shortest Job First - Pre-emptive

(or) SJF

Process Burst

No.	time
P ₁	7
P ₂	4
P ₃	8



* Round Robin Scheduling :-

Pre-emptive.

Criteria → Time Quantum

→ Given time quantum = Δ .

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	5	5	4	4

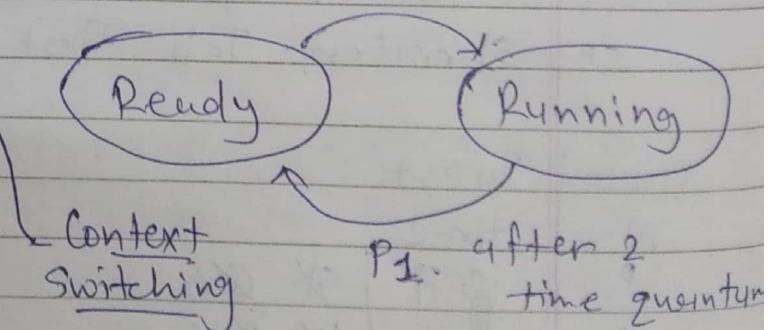
Ready queue [P₁ | P₂ | P₃ | P₁ | P₄ | P₂ | P₁]

Running queue [P₁ | P₂ | P₃ | P₁ | P₄ | P₂ | P₁]

0 2 ↑ 4 6 8 9 11 BT of P₄ is 1

∴ only 1 unit is allocated.

Process No.	Burst Time
P ₁	8 8 X 0
P ₂	X X 0
P ₃	X 0
P ₄	X 0



→ Sequence of processes is important.

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

$\rightarrow TAT = CT - AT = \frac{\text{Completion Time}}{\text{Arrival Time}}$

$\rightarrow WT = TAT - BT$.

$\rightarrow RT = \text{CPU gets process} \rightarrow AT$.

$\rightarrow \text{Avg. TAT} = 31/4 = 7.75$.

$\rightarrow \text{Avg. WT} = 19/4 = 4.75$.

$\rightarrow \text{Avg. RT} = 7/4 = 1.75$.

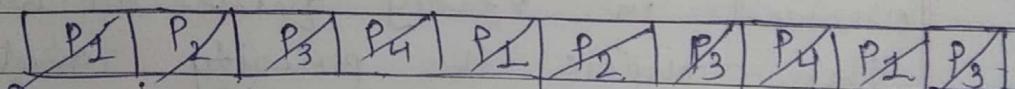
$\rightarrow \text{No. of Context Switching} = 6$.

(Do not calculate the first and last lines. Count only of in b/w. (lines)).

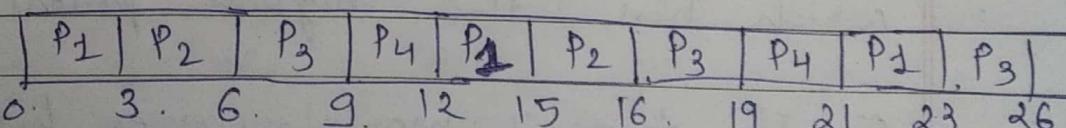
* Note:- Time Quantum = 3

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



Running Queue



Process BT

P₁ 8 8 2 0

P₂ 4 2 0

P₃ 9 6 3

P₄ 5 2 0

* Priority Scheduling :- Criteria : Priority Pre-emptive

Priority	Process	Arrival Time	Burst Time	Completion Time	TAT	WT
	No.					
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	8 X 0.	P ₁	P ₂	P ₃	P ₃	P ₄	P ₂	P ₁
P ₂	X 3 0	0	1	2	3	4	5	8
P ₃	X 2 0							
P ₄	X 0							

Lowly P₂ has the higher priority in both of them (P₂ & P₁)

∴ 5 to 8 and 8 to 12

→ Run each process

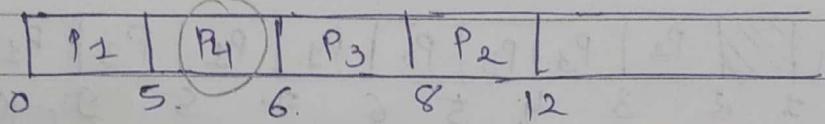
one time only and check whether any process arrived in Ready queue or not. If yes then check the priority and execute which has the higher priority.

→ If two or more processes having same priority and same arrival time then choose P₁, P₂, ..., P_n n-id wise.

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

Priority No.	Process	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 \Rightarrow Higher the priority.



Process	B.T	TAT = CT - AT
1	5	WT = TAT - BT
2	4	
3	2	
4	1	

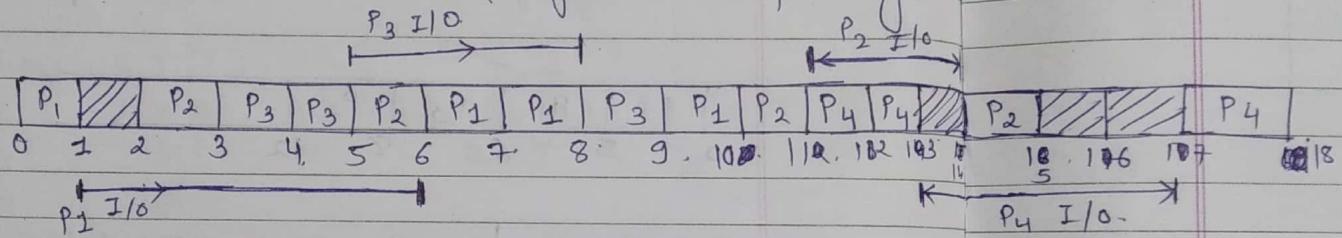
→ In this case when Process P₁ is executing it executes completely as it is non-preemptive and ~~there~~ simultaneously processes P₂, P₃ and P₄ goes into ready queue. Now, go according to the priority to arrange the sequence of processes.

* Mix Burst Time :-
Pre-emptive

Criteria: Priority

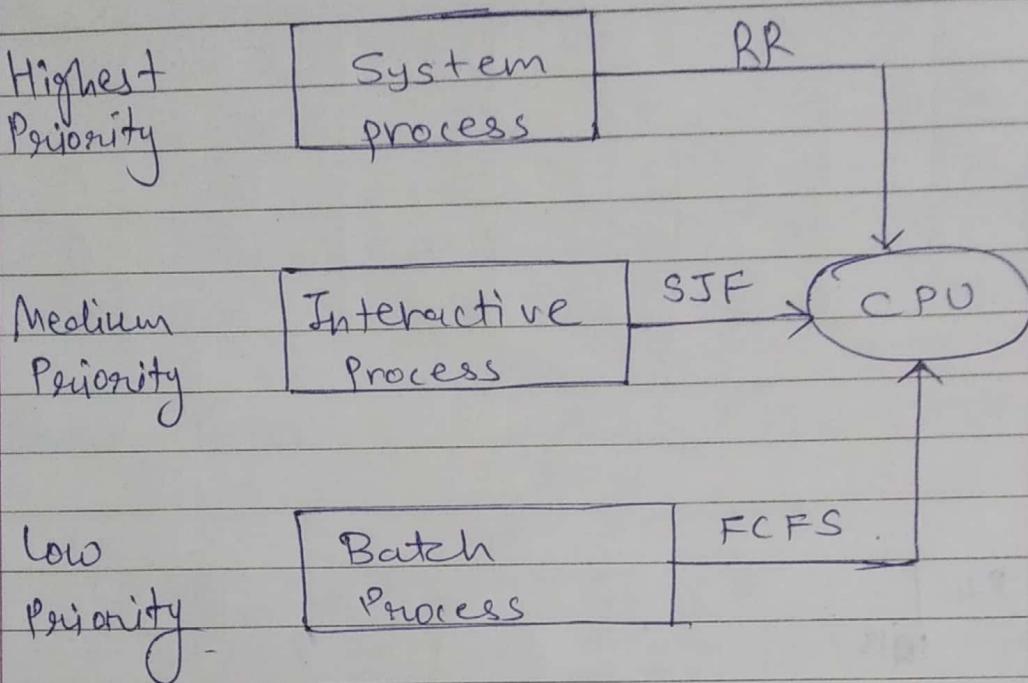
Process	Arrival	Priority	CPU	I/O	CPU	Completion Time
No.	Time					
P ₁	0	2	1	5	3	
P ₂	2	3	3	3	1	
P ₃	3	1	2	3	1	
P ₄	3	4	2	4	1	

Given: lowest the number, higher the priority.



Process	B.T./CPU	I/O	CPU
P ₁	X 0	5	X 2 0
P ₂	3 X 0	3	X 0
P ₃	X 2 0	3	X 0
P ₄	X 2 0	4	X 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 or execute.

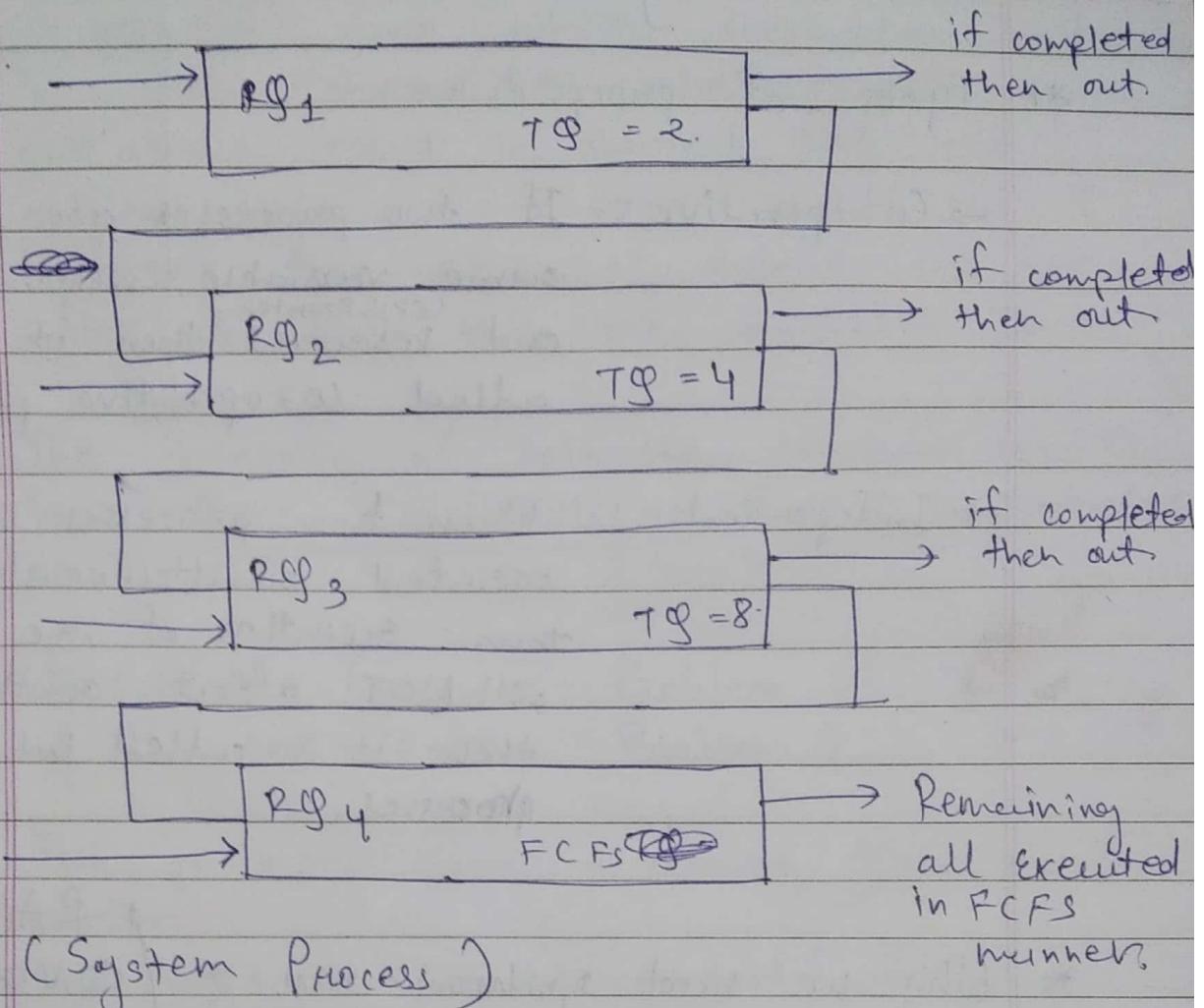
→ This is called starvation problem.

* Starvation problem can be solved using Multi level Feed back 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.

(Batch Process)

Lowest Priority.



(System Process)

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

* Process Synchronization :-

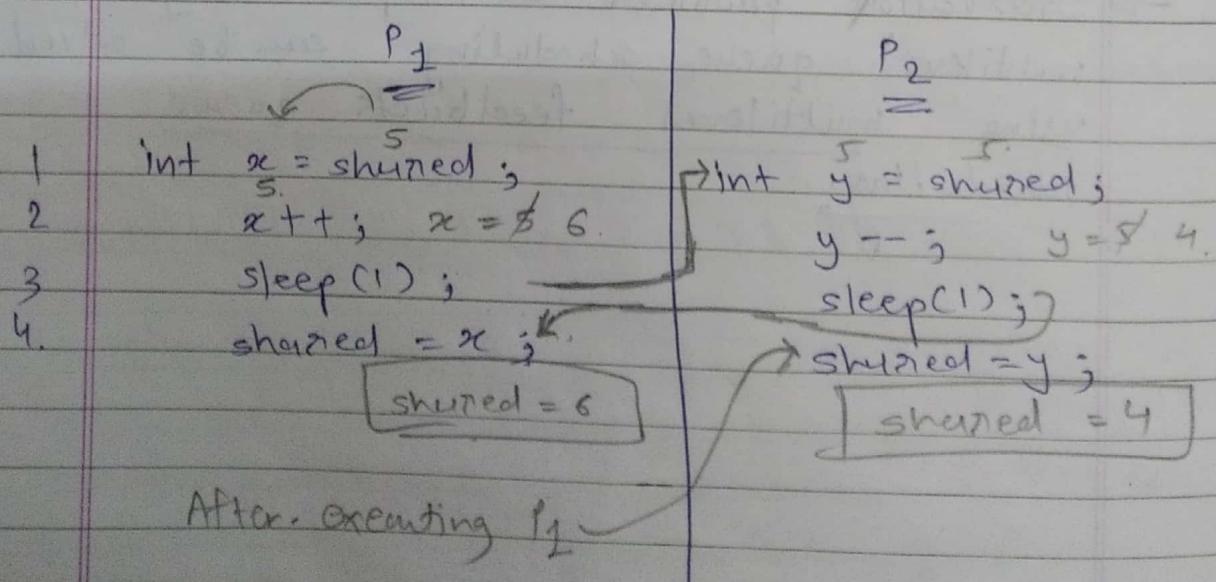
* Types of processes :-

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



- In this case, both the processes are **co-operative** but while concluding both of them shared ++ and shared -- with should result in shared ~~not~~ ^{and} 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 awokened 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;
   0
   1
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;
    }
}

```

Solⁿ:

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 );
    }
}

```

```

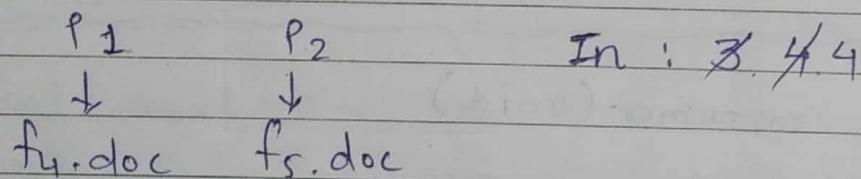
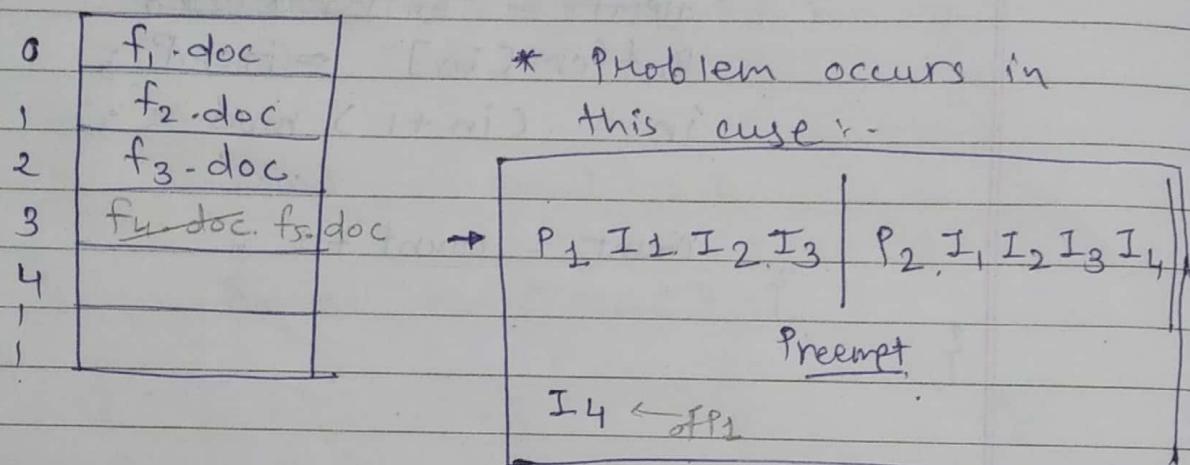
void consumer ( void )
{
    int itemC; int out = 0;
    while ( true )
    {
        if ( count == 0 ) // buffer is empty
            sleep(); // go to sleep.
        itemC = Buffer [out];
        out = (out + 1) mod n;
        count = count - 1;
        if ( count == n - 1 ) // buffer is full
    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



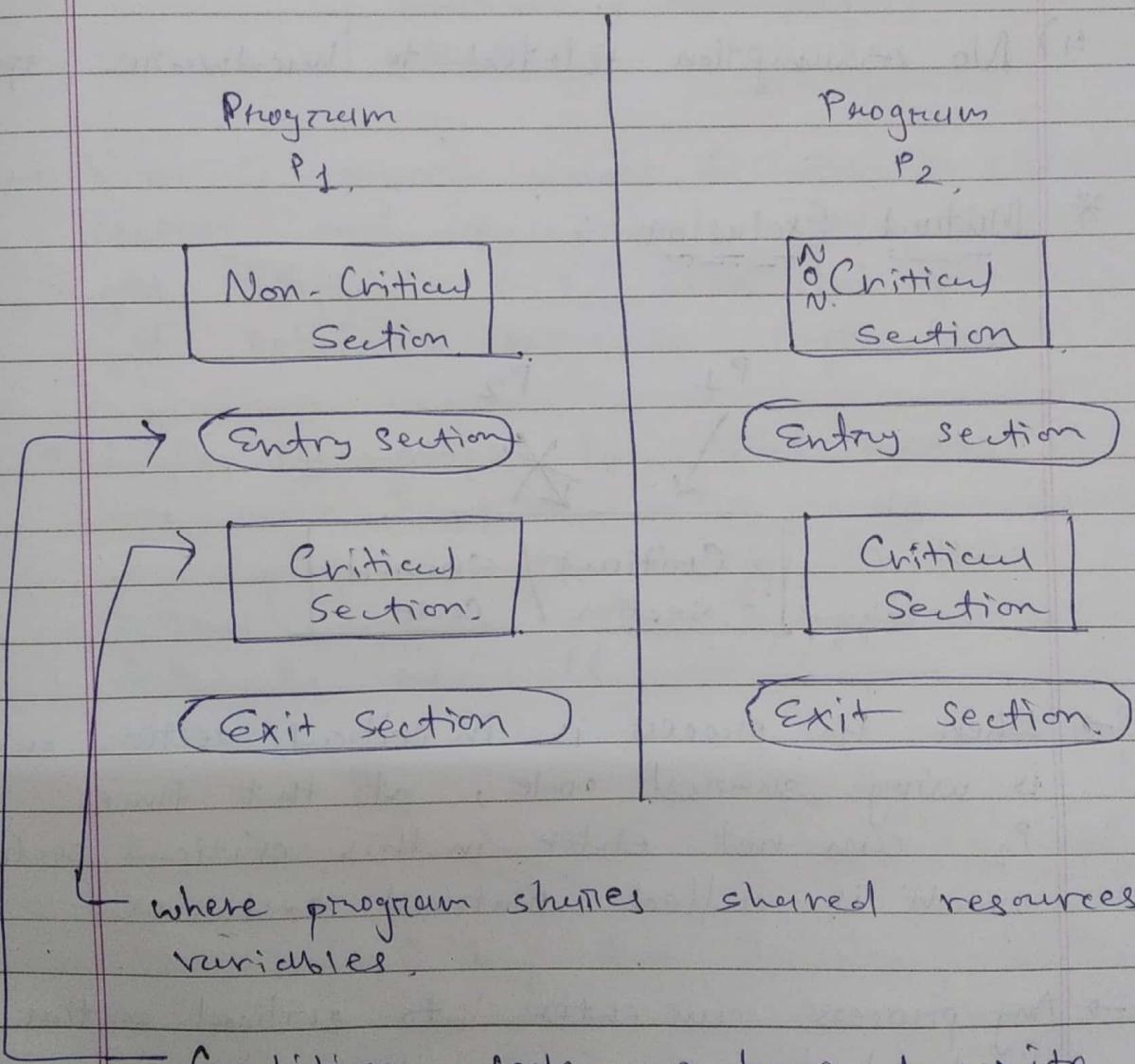
R_1 R_2
 3 4. 3 4

→ Problem occurs when P₁ process preempts and P₂ process starts executing. At that time in spooler directory f₄.doc will override with f₅.doc of process P₂. Which results in data loss.

* Critical Section Problem , Mutual Exclusion :-

* Critical section :-

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



where program shares shared resources, variables.

Condition code we have to write to enter into critical section to avoid race condition and to achieve synchronization.

* Conditions to write synchronization mechanism:

(Entry Condition)
of C.S.

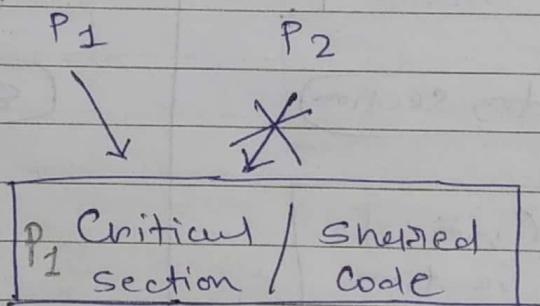
1) Mutual Exclusion

2) Progress

3) Bounded Wait

4) No assumption related to hardware, speed.

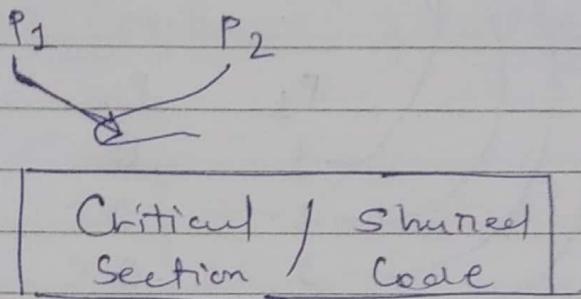
* Mutual Exclusion :-



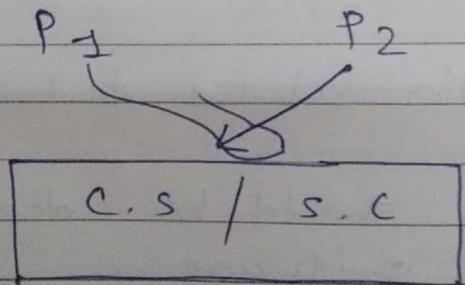
- When P_1 process is in critical section and is using shared code, at that time P_2 can not enter in this critical section, which is called mutual exclusion.
- Any process can enter into critical section first but when one is running in critical section, another can NOT enter.

* Progress :-

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

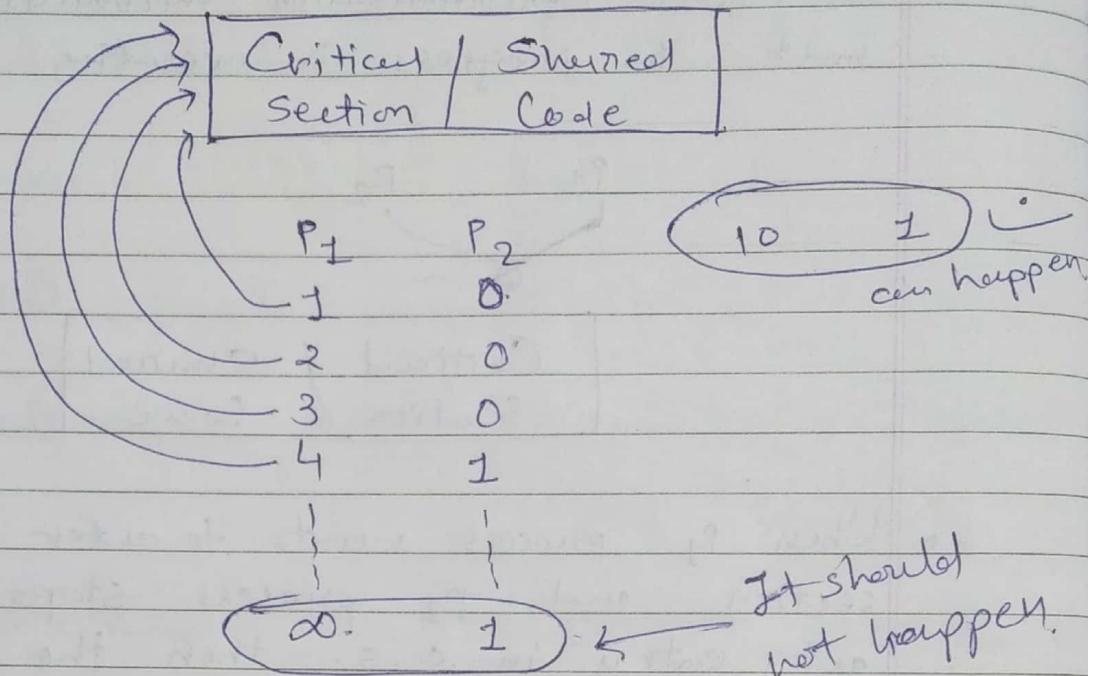


- When P_1 process wants to enter in critical section and P_2 process stops P_1 to get entry in c.s. then the progress of both the processes stops.
- When P_1 stops P_2 that means P_1 has some kind of entry code which is stopping P_2 to enter in the critical section. Same applies for P_2 as well.



- In this case P_1 is stopping P_2 to enter in critical section

* Bounded Wait :-



→ When P₁ process enters into critical section for multiple times and process P₂ waits for its turn to enter into it. But P₁ enters into C.S 20 times and P₂ 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 solⁿ should be there.

* LOCK Variable :-

Critical Section Solution using LOCK vari:

- Execute in user mode
- Multi process solution
- No mutual exclusion guarantee

Entry
Code.

1. `while (LOCK == 1);`

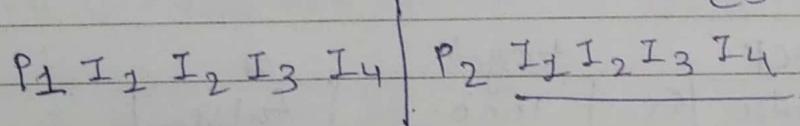
2. `LOCK = 1;`

3. Critical section

4. `LOCK = 0`

→ $LOCK = 0 \Rightarrow$ vacant (c.s.),
 $LOCK = 1 \Rightarrow$ full (c.s.).

→ Case: 1 : $P_1 \quad P_2 \quad LOCK = \cancel{0} \cancel{1} \cancel{0} \cancel{1} 0$
 $CS = P_1 \neq P_2$



* → Case: 2 :

P ₁ Preempts	1.	<code>while (LOCK == 1);</code>
	2.	<code>LOCK = 1;</code>
	3.	Critical section
	4.	<code>LOCK = 0</code>

P₂ P₂

LOCK : 0 \neq 1

CS : P₂

P₁ P₂

I₁ \rightarrow I₁

I₂ \leftarrow I₂

I₃ \rightarrow I₃

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 :-

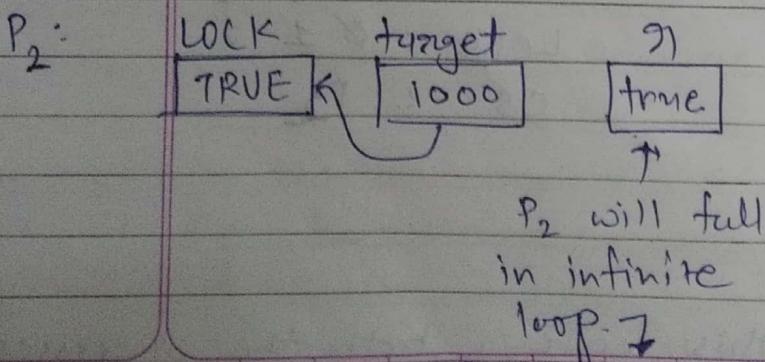
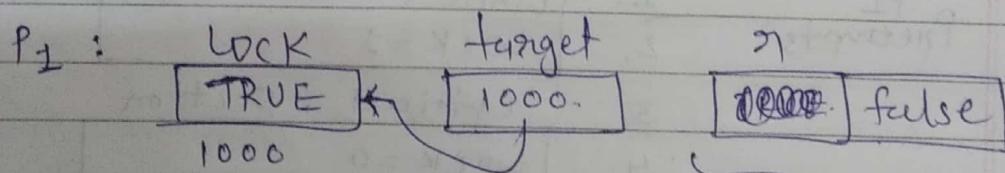
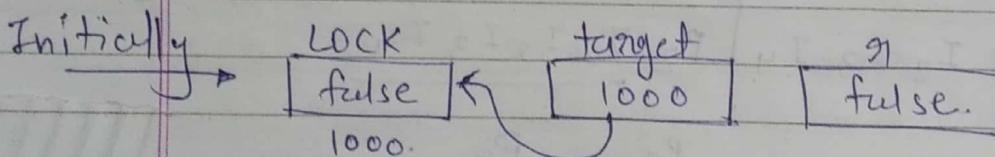
Merged in a single instruction

→ while (TSL (& LOCK)) ; { 1. while (LOCK == 1) ; } 2. LOCK = 1

CS
 LOCK = false ; 3. Critical section 4. LOCK = 0

TSL
 boolean test_and_set (boolean * target) {

boolean n = * target ;
 * target = TRUE ;
 return n ; } ✓



if n returns false that mean CS is empty and P₁ can enter into Critical section .

→ 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 guarantee.

→ Progress problem ~~can~~ occurs.

→ Bounded waiting (infinite waiting) will NOT happen

→ Initially $turn = 0$

Process P₀

Process P₁

Non - critical section

Non - critical section

entry
code

while (turn != 0);

while (turn != 1);

Critical Section

Critical Section

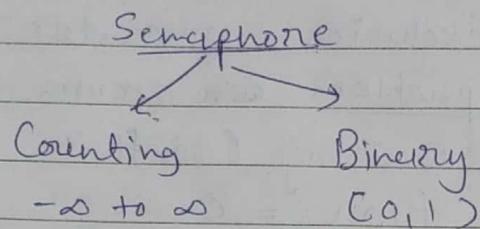
exit
code

$turn = 1$

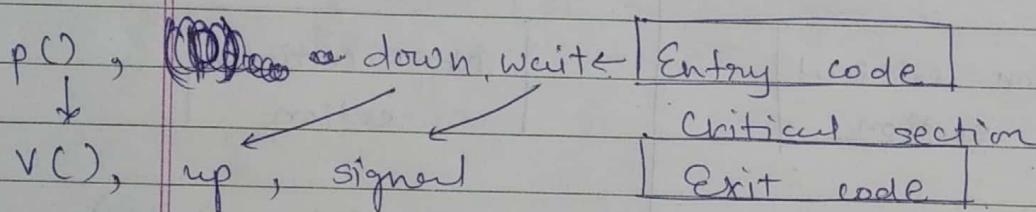
$turn = 0$.

* Semaphore :-

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



$P_1 \ P_2 \ P_3$



* 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;

}

}

$P \Rightarrow -$ Down
 $V \Rightarrow +$ 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;

}

P_1, P_2

$S = 0, -1, 0 \} -1$

C.S = P_1, P_2

suspended list = R_2

* Ques:-

→ $S = 0 \Rightarrow$ No. of processes in suspended list = 0.

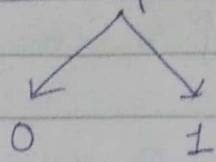
→ $S = 10 \Rightarrow$ How many processes can enter into critical section? = 10.

→ $S = 10, 6P, 4V \rightarrow +$

→ Here, $P = \text{down}$, $V = \text{up}$.
 $\therefore 10 - 6 + 4 = 8$

* 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.

Process enters
Down \rightarrow P₁

$$S = 1$$

$\hookrightarrow S = 0$, enter into critical section

$$P_2 \quad S = 0$$

$\hookrightarrow S = 0$, Block this process and place it in suspended list.

Up \rightarrow

$$S = 0$$

$$\begin{cases} S = 0 \rightarrow 1 \\ S = 1 \rightarrow 1 \end{cases}$$

$\hookrightarrow \therefore$ suspended list, $S = 1$.

is empty

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

$\hookrightarrow \therefore$ suspended list, $S = 1$

is NOT empty

+

select a process from suspended list

\rightarrow Initialize S with 1 always.

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

\rightarrow Each process P_i = {i = 1 to 9} \rightarrow Process P₁₀ executes following code.

Entry code \rightarrow P(mutex)

c.s.

Exit code \rightarrow V(mutex)

forever.

repeat

V(mutex)

c.s

V(mutex)

forever

Soln.: → Case 1 : mutex : $\neq 0$

Critical section = P_1 (P_2 will not enter in c.s.)

→ Case 2 : mutex : $\neq \emptyset \neq \emptyset$

c.s. = $P_1 \underline{P_{10}} P_2 \underline{P_3} \underline{P_{10}} P_4 P_5 \underline{P_{10}} P_6 P_7 \underline{P_{10}} P_8 P_9$

suspended list = $\cancel{P_2}, \cancel{P_3}, \cancel{P_4}, \cancel{P_5}, \cancel{P_6}, \cancel{P_7}, \cancel{P_8}, \cancel{P_9}$

∴ Maximum processes that may present = 10.

* * Producer - Consumer Problem's Solution

Using Binary Semaphore :-

→ Binary Semaphore $S = 1$

Empty slots = 5

full slots = 3.

$n=8$

out = 0

in = 3

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

$$\cdot In = (In + 1) \bmod n$$

3) up (s)

4) up (full).

→ Consumer (void).

{

1) down (full)

2) down (s)

itemC = Buffer [out];

$$out = (out + 1) \bmod n;$$

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)

no of
empty slots
in buffer

Critical section →

insert_item (itemP);

up (& mutex)

up (& full)

no of full slots
in buffer

void consumer (void) .

{

int itemC;

while (true)

{

down (& mutex)

down (& full)

itemC = remove_item();

up (& mutex)

up (& empty)

consume_item(itemC);

3.

* 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 share 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
- all other ints

- Thread - yield is thread call which allows thread to voluntary give the CPU and let another thread run

* Implementation of lock and unlock in Mutex :-

mutex-lock :

TSL R₀₁, mutex
CMP R₁, 0
JZ E OK
CALL thread-yield
JMP mutex-lock

OK : RETN RET

mutex_unlock :

MORE MUTEX, 0.
RET

* * Reader-Writer Problem :-

→ Reader-Writer problems occur ...

R - W → Problem

W - R → Problem

W - W₂ → Problem

R₁ - R₂ → No Problem

} On same Database.

int nc = 0;

Semaphore mutex = 1;

Semaphore db = 1.

Binary

Semaphore

void Render (void)

{

while (true)

{

down (& mutex)

nc = nc + 1

if (nc == 1) then down (& db);

up (& mutex)

db

exit

code

down (& mutex)

nc = nc - 1

if (nc == 0) then up (& db);

up (& mutex)

Process_duty ();

}

void Writer (void)

{

while (true)

{

down (& db);

db

up (& db);

}

* Case : 1 : R - W.

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

- Binary semaphore is used \Rightarrow mutex and db can be 0 or 1.
- Writer is blocked

* Case : 2 : W - R.

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

\therefore Reader can NOT enter.

* Case : 3 : W₁ - W₂.

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

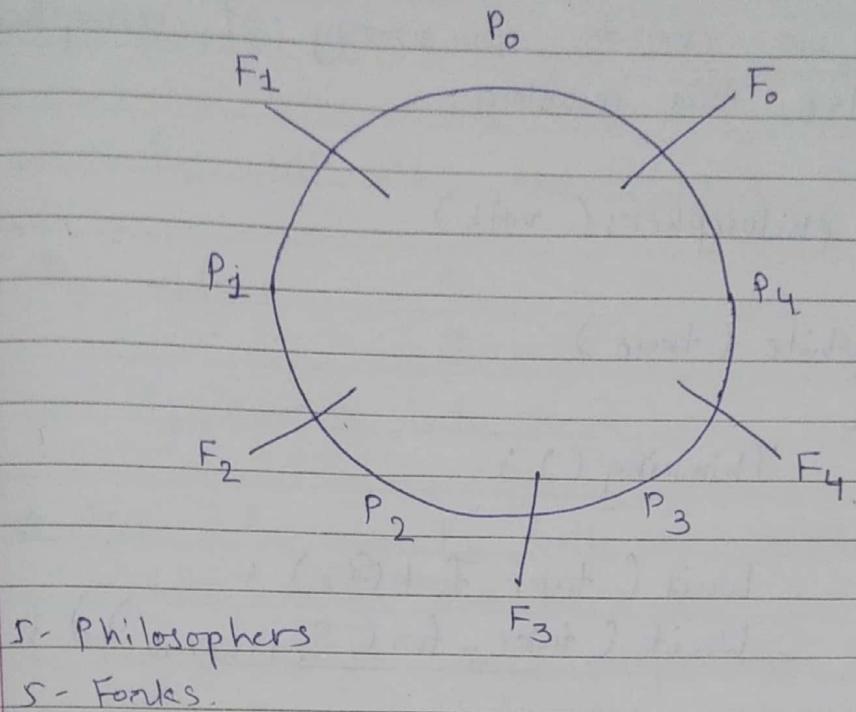
\therefore W₂ can NOT enter

* Case : 4 : R₁ - R₂

	Reader 1	Reader 2
$rc = 0$	1	R ₁ is 2
$mutex = 1$	0	1 in c.s. R ₂ is
$db = 1$	0	in c.s.

\therefore Both the readers can be in critical section

* Dining Philosophers Problem :-



→ 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);

}

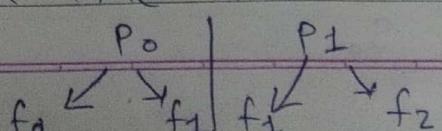
}

(P₀, P₁)

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

RACE

CONDITION



Here, P₁ will complete eating first and then the turn of P₂.

- 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 (S_i)) ;

 Wait (take_fork (S_{(i+1) % N})) ;

 EAT () ;

 Signal (Put_fork (S_i)) ;

 }

Signal (Put_fork (S_{(i+1) % N})) ;

}

- Initially :- S₀ S₁ S₂ S₃ S₄
 | | | | |

Philosophers /

- Processes needs following semaphores to wait down to take forks.

P₀ : S₀ S₁

P₁ : S₁ S₂

P₂ : S₂ S₃

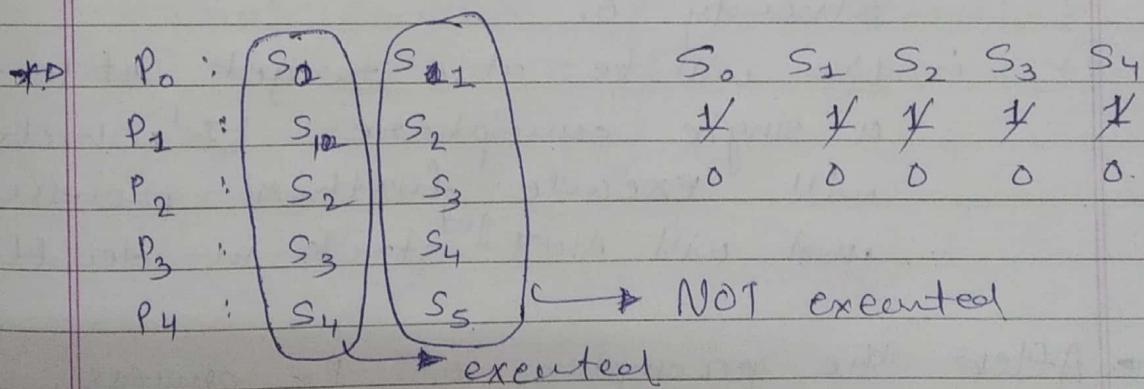
P₃ : S₃ S₄

P₄ : S₄ S₀

- * P₀ enters ∵ S₀ = X 0. ∵ P₀ enters
 S₁ = X into c.s.
- P₁ enters ∵ S₁ = 0 ← it will not be
 S₂ = 1 down.
 ∵ P₁ will NOT enter into c.s.
- P₂ enters ∵ S₂ = X 0
 S₃ = X 0.
 ∵ P₂ enters into c.s.

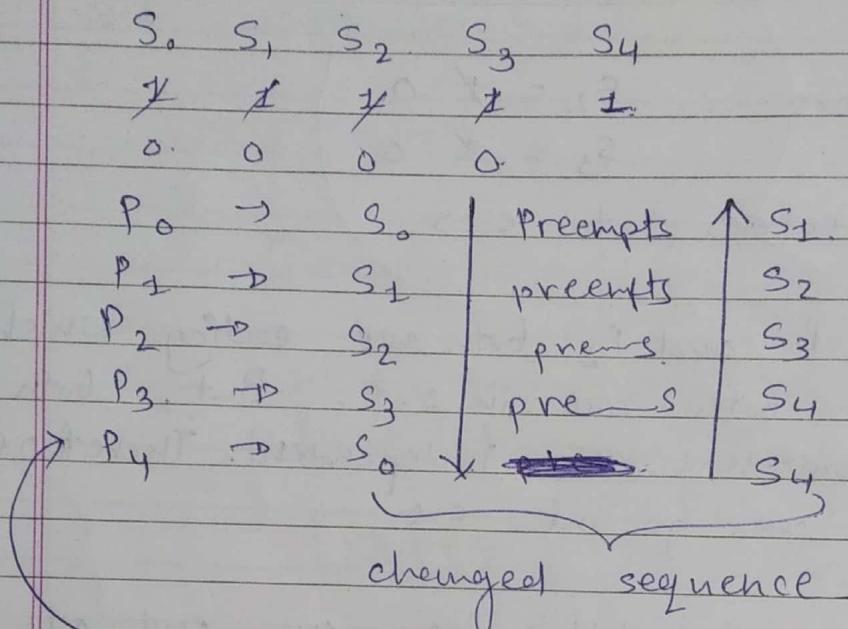
* Here, P₀ and P₂ both are eating which means both are in c.s. But, both the processes are independent. Therefore, both can go into c.s.

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



- 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 P₀ wants to take fork 1, it will NOT be able to take it as S₁ is already downed to 0 by P₁.

* 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 us as S_0 is already 0.

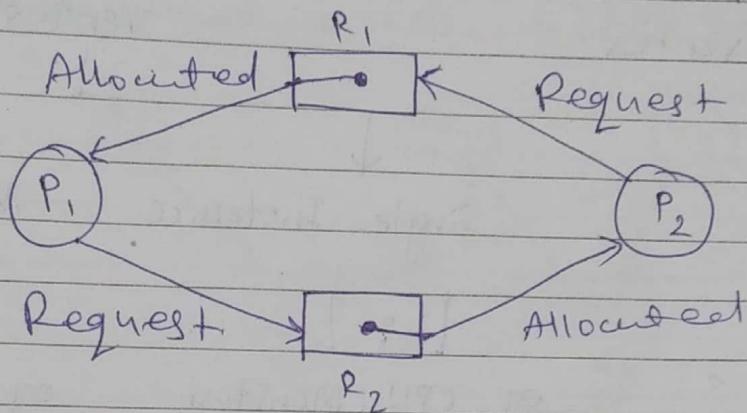
- We will be able to get at least a single semaphore '1' which will execute further processes and will NOT get stuck in deadlock.

→ 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 completely and then will go out from C.S. and P_2 will resume taking the right fork.

* Deadlock :-

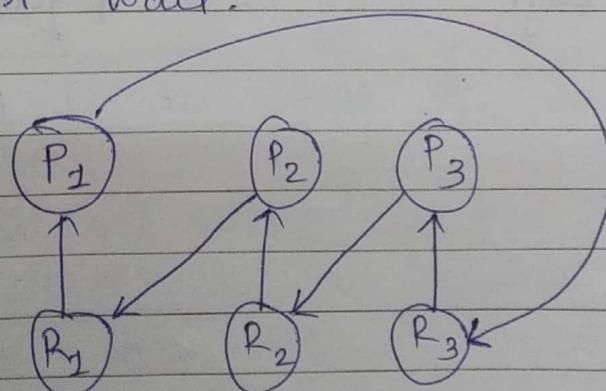
→ If two or more processes are waiting on happening of some event, which never happens, then we say these processes are involved in deadlock then that state is called Deadlock.



* Necessary conditions for occurrence of deadlock :-

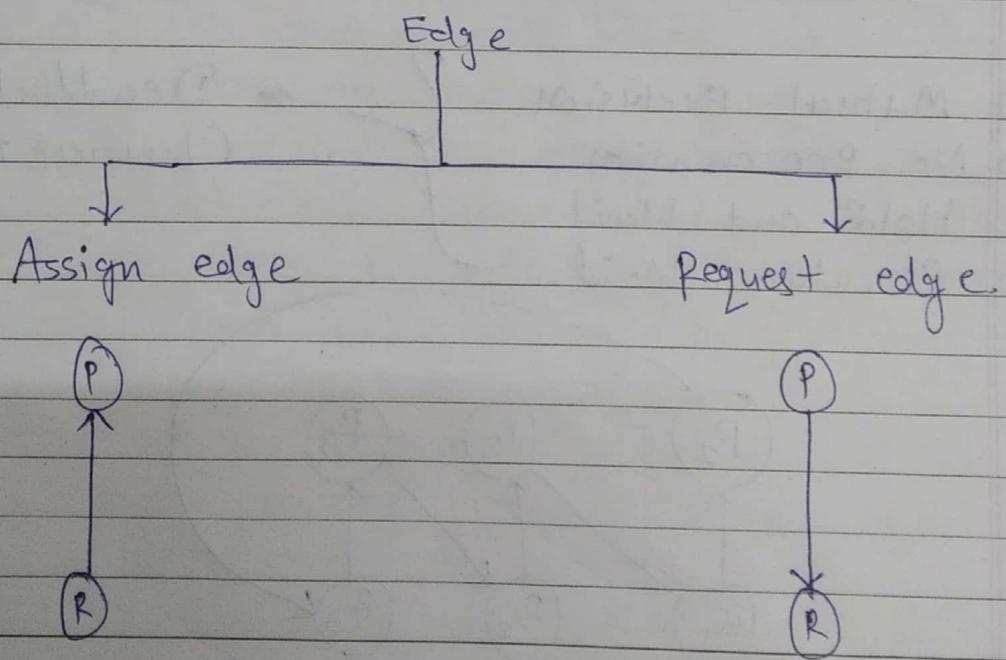
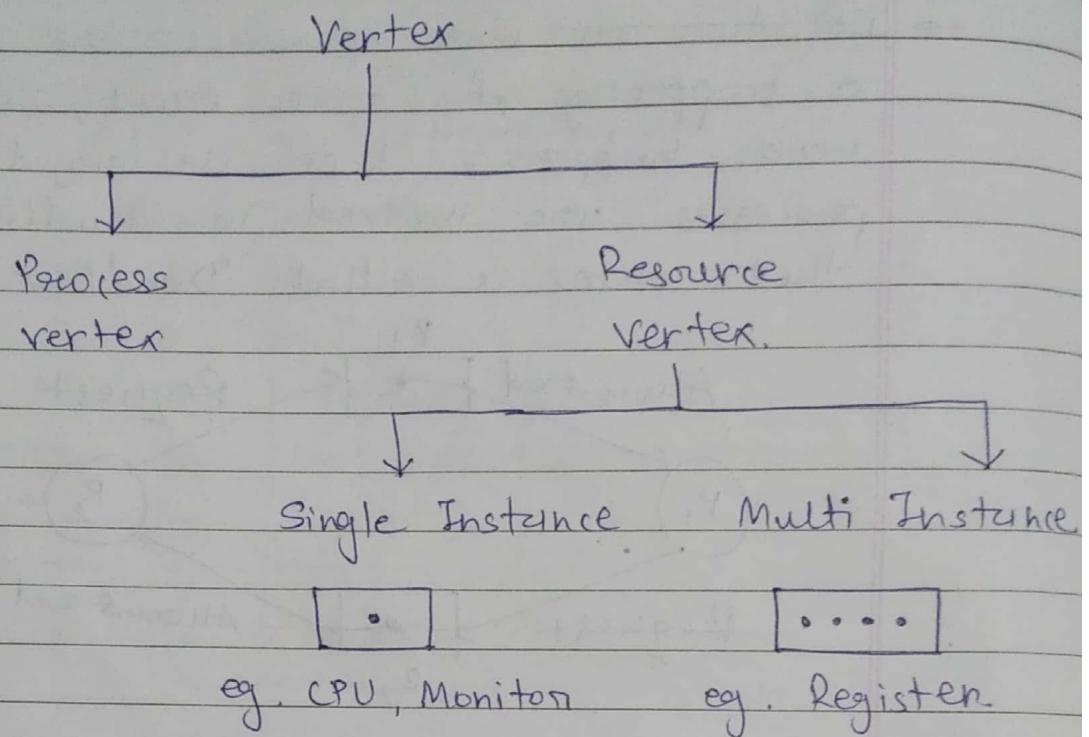
- 1) Mutual Exclusion
- 2) No Pre-emption
- 3) Hold and Wait
- 4) Circular Wait.

} **Deadlock Characterization**



Circular wait.

* Resource Allocation Graph : [RAG] :-



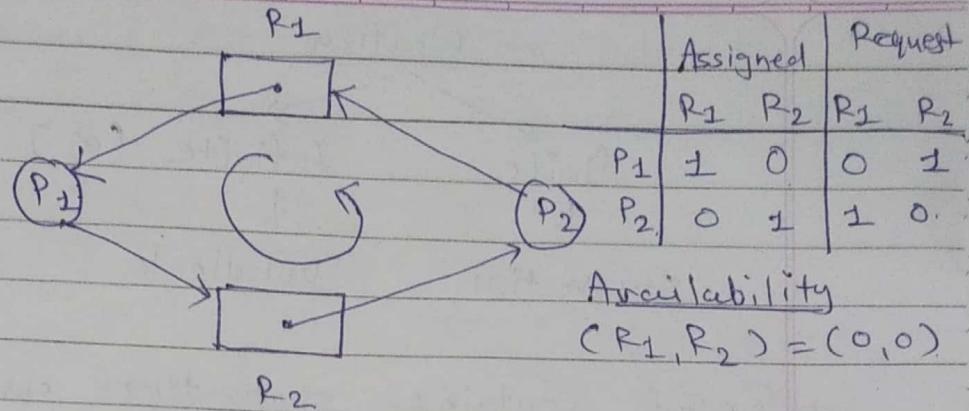
Resource R is assigned to process P.

Process P is requesting for resource R.

→ From the availability, check whether we can fulfill the REQUEST OF any process. or not.
if not \Rightarrow deadlock.

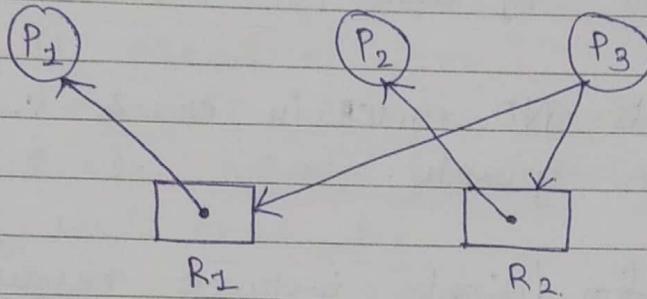
Date _____
Page _____

Ex:- 1 :-



→ Circular wait occurs \therefore RAG has deadlock.

Ex:- 2 :-



Sol'n :-

	Assigned		Request	
	R1	R2	R1	R2
P1	1	0	0	0
P2	0	1	0	0
P3	0	0	1	1

→ Availability $\Rightarrow (R_1, R_2) = (0, 0)$.

→ Here, P1 is demanding for no resources

$\therefore P_1$ will get executed and will terminate

$\therefore R_1$ will be free.

Availability

(R_1, R_2)

0 0

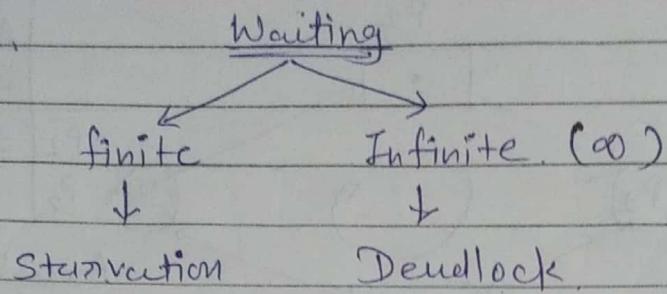
\rightarrow 1 0

\rightarrow 1 1

→ Same way, P2 will execute and terminate
 $\therefore R_2$ will be free.

→ Now, we can fulfill the request of P3
 $\therefore P_3$ will get executed successfully.

\therefore NO deadlock occurs.



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

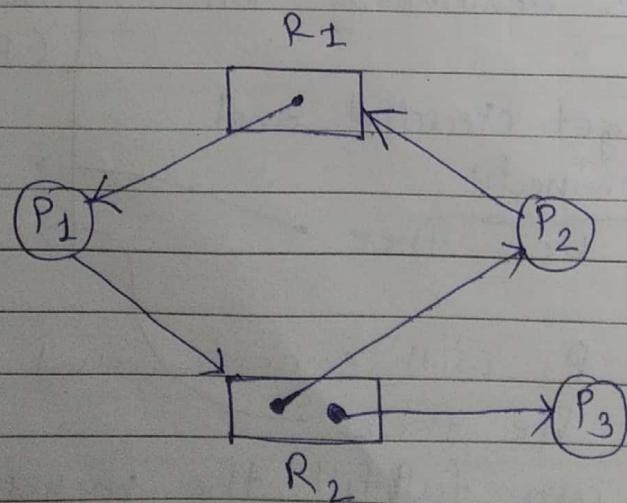
→ There is NO cycle in ex :- 2 ∴ It is called Acyclic graph.

* 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.

M u l t i I n s t a n c e R A G :-

* Check whether RAG has deadlock or not.

Ex :- 1 :-



Sol :-

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

$\rightarrow P_3$ will be executed and terminated first.

\therefore After termination $P_3 \rightarrow$ Availability $(R_1, R_2) = 0, 0$.

$\rightarrow P_1$ will be executed and terminated as P_1 is requesting for $(0, 1)$ which is available. Now, P_1 will free R_1 .

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

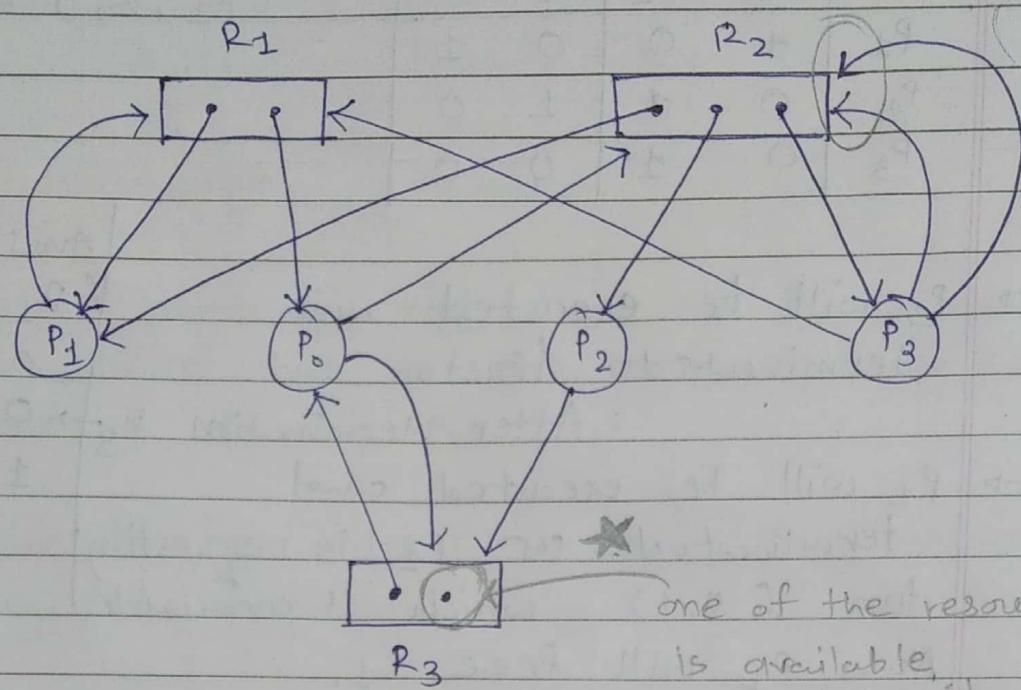
\therefore Termination sequence $P_3 \rightarrow P_1 \rightarrow P_2$.

\therefore 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	
✓ P ₀	1	0	1	0	1	1	(R ₁ , R ₂ , R ₃)
✓ P ₁	1	1	0	1	0	0	(0, 0, 1)
✓ P ₂	0	1	0	0	0	1	
P ₃	0	1	0	1	2	0	

Availability
 (R_1, R_2, R_3)
 \downarrow
 $(0, 0, 1)$
∴ P₂ will execute $\rightarrow (0, 1, 1)$
and terminate

∴ P₀ will execute $\rightarrow (1, 1, 2)$
and terminate

∴ P₁ will execute
and terminate $\rightarrow (2, 2, 2)$

∴ P₃ will execute
and terminate $\rightarrow (2, 3, 2)$

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

∴ There is NO DEADLOCK.

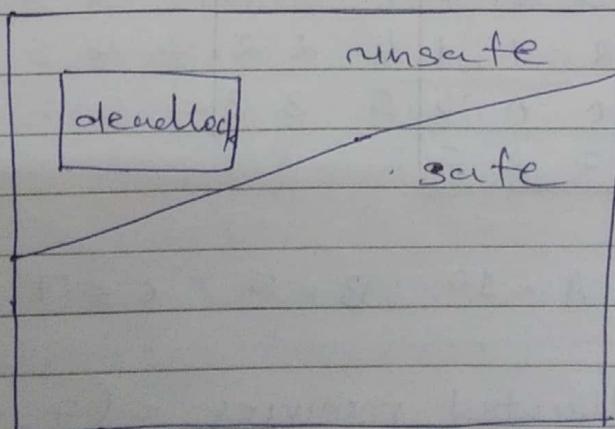
* Various Methods to handle Deadlock :-

- 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 \Rightarrow no deadlocks.
 - If a system is in unsafe state \Rightarrow possibility of deadlock.
 - Avoidance \Rightarrow 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_{01} 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 :-

* Bunker's Algorithm :-

+ It is also used for deadlock detection.

Process	CPU Memory Printer			Max - Allocation					
	Assigned			Max Need	Available	Remaining Need			
	A	B	C	A	B	C	A	B	C
P ₁	0	1	0	7	5	3	3	3	2
P ₂	2	0	0	3	2	2	5	3	2
P ₃	3	0	2	9	0	2	7	4	3
P ₄	2	1	1	4	2	2	7	4	5
P ₅	0	0	2	5	3	3	7	5	5
	7	2	5				7	4	3

→ Total A = 10, B = 5, C = 7. [Given]

Soln → Total allocated resources = (7, 2, 5) = (A, B, C).

∴ Remaining resources at a time

$$\begin{aligned}
 &= (10, 5, 7) - (7, 2, 5) \\
 &= (3, 3, 2).
 \end{aligned}$$

→ Now, calculate the remaining need of every process.

$$\rightarrow \text{Remaining Need} = \text{Max need} - \text{Allocation}$$

→ 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.
 $\text{Current Availability} \geq \text{Remaining Need}$.

→ 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. ~~and terminate~~

$$\therefore \text{Ans} \dots P_2 \rightarrow P_4 \rightarrow P_5 \rightarrow P_1 \rightarrow P_3$$

4) Deadlock Detection and Recovery :-

→ We can use Banker's algorithm to detect whether deadlock exist or not.

→ To recover 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	0	3	3	0
P ₁	1	1	2	2	1	4	✓	1	0	2	4	3
P ₂	1	0	3	1	3	3	✓	0	3	0	5	3
P ₃	2	0	0	5	4	1	✓	3	4	1	6	4
	<u>5. 1 6</u>										<u>8</u>	<u>4</u>

Solⁿ → P₀ → P₂ → P₃ → P₄.

Current availability

→ Here we don't need to calculate (S, 1; 6) as we are already given available resources by (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 P₁, P₂, P₃.
each require 2 units of resources.

∴ Allocate the 1 less unit.

$$\therefore P_1 \quad P_2 \quad P_3$$

Add one
to avoid
deadlock

Allocated (1 1 1) = 3 + 1
= [4]

★

P ₁	P ₂	P ₃
3	4	5
2	3	4
9	+ 1	= 10.

Required Resources.

Allocate 1 less resource.

$$\downarrow \quad 9 + 1 = 10. \rightarrow \text{to avoid deadlock.}$$

∴ 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.

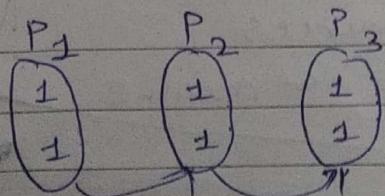
→ Ans:- Allocate 1 less to each process.

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.

Ques:- 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 _____.

→



$$R = 4$$

$$\therefore K = 2$$

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

R - Resources, n - Processes.

Processes :- $P_1 \ P_2 \ P_3 \ \dots \ P_n$

Demand :- $d_1 \ d_2 \ d_3 \ \dots \ d_n$

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

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

\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.

$$\therefore R + n > \sum_{i=1}^n d_i \quad \frac{\text{Total Demand}}{\downarrow}$$

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

\Rightarrow No Deadlock.

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

Ex :- 4 processes, 4 Resources, Max demand
 $\rightarrow 4 + 4 > 4 \times 2$ $\left(\text{Assume } 2\right)$ \therefore there always be deadlock

$8 > 4$

$\rightarrow 8 > 4 \times 2$

$8 > 8$ False

~~$8 > 7 \times 3$~~

~~$8 > 12$ False~~

Deadlock

Operating System

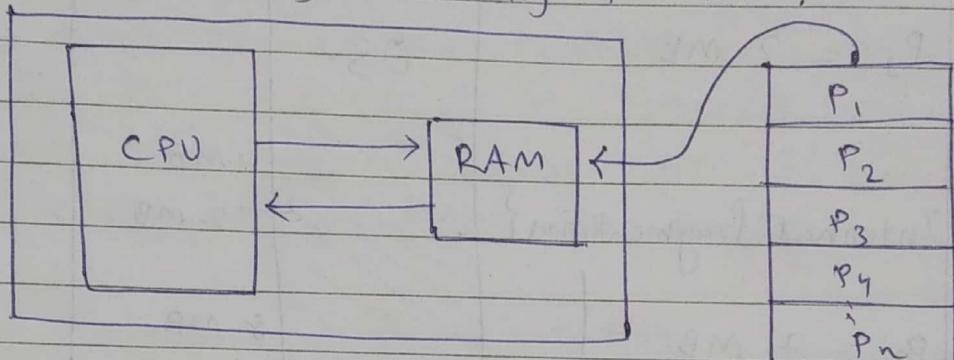
Date _____
Page _____

* Memory Management :-

→ Method of managing the primary memory.

* Multiprogramming :-

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



Secondary memory

→ Higher the degree of multiprogramming \Rightarrow Higher the utilization of CPU

* Goal : Efficient utilization of memory (RAM).

* Memory Management Techniques :-

Contiguous

Static → Fixed partition

Dynamic → Variable partition

Non-contiguous

→ 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.
- Contiguous allocation ∵ spanning is NOT allowed.

→ $P_1 = 2 \text{ MB}$

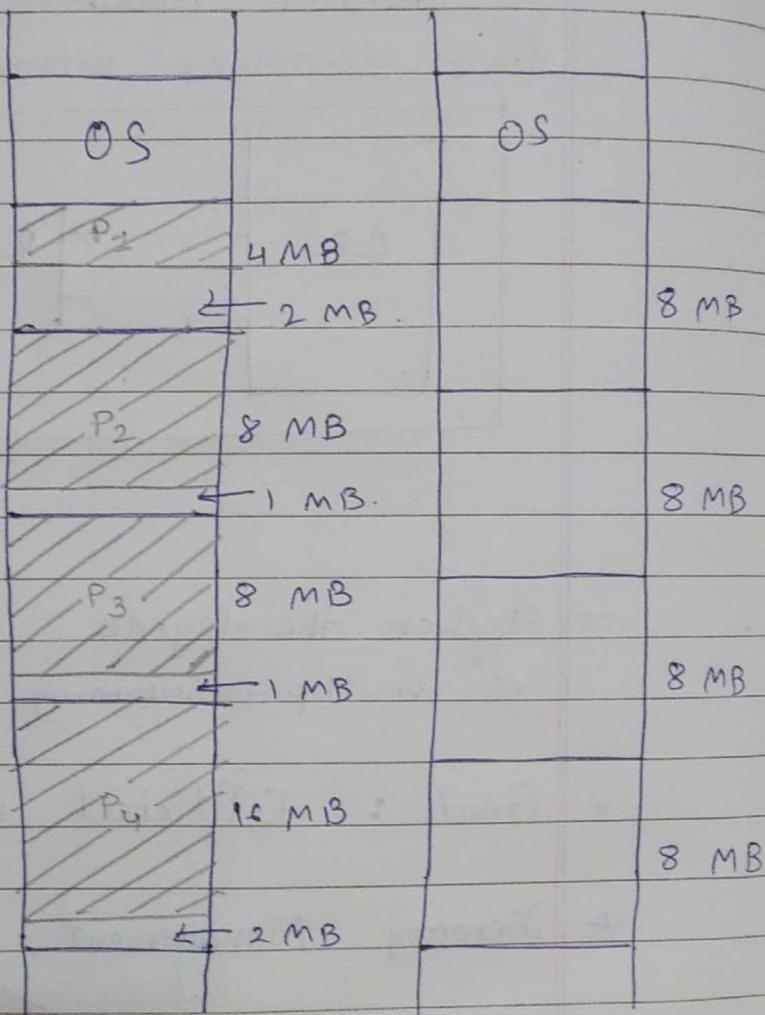
Internal fragmentation {

→ $P_2 = 7 \text{ MB}$

→ $P_3 = 32 \text{ MB}$
can Not be
allocated in any
partition.

→ $P_4 = 14 \text{ MB}$

→ $P_5 = 5 \text{ MB}$



→ Problems / limitations :-

1) Internal fragmentation :-

→ When process having lesser size than 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})
(∵ contiguous 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 Runtime.

→ $P_1 = 2 \text{ MB}$

→ $P_2 = 4 \text{ MB}$

→ $P_3 = 8 \text{ MB}$

→ $P_4 = 4 \text{ MB}$

→ $P_5 = 8 \text{ MB}$

	OS	
	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 ~~are~~ on process size

→ There won't be any limitation 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.

OS		
P_1	2 MB	
Hole	4 MB	
P_3	4 MB	
P_4	8 MB	
Hole	4 MB	
P_5	4 MB	

→ It can be removed by compactization.

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

~~also~~ → 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 Contiguous 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.

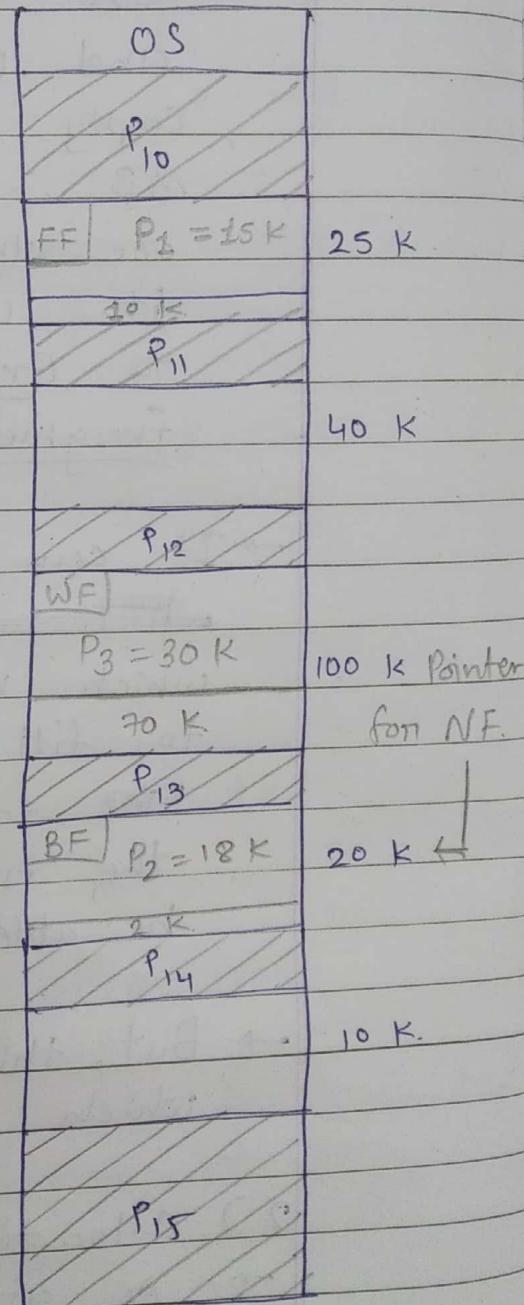
Advantages :

→ 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.

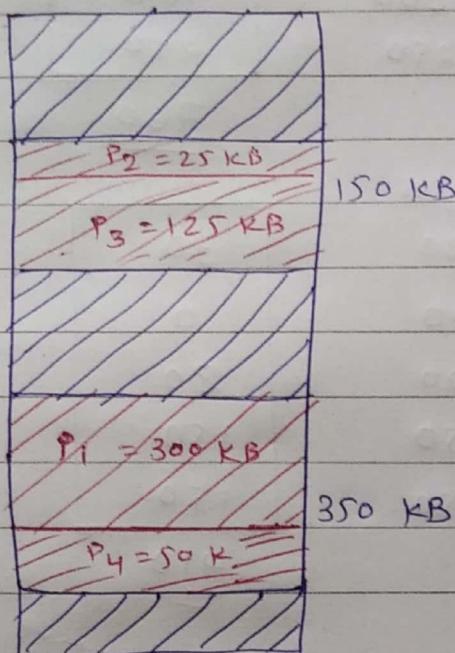
4) Worst Fit

- Allocates 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.

Puc. 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 :- →



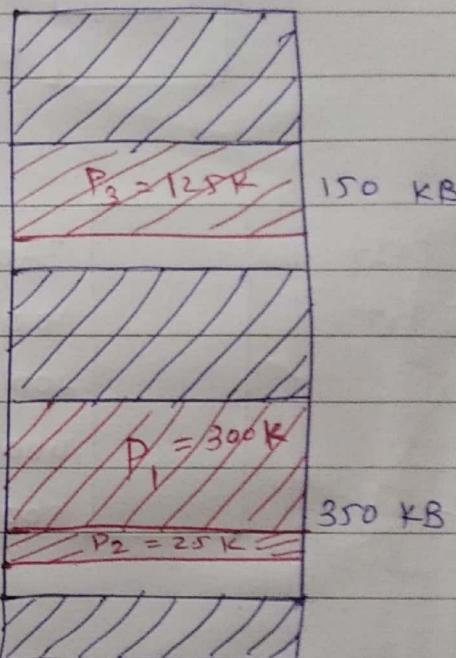
First Fit.

→ let $P_1 = 300 \text{ KB}$

$P_2 = 25 \text{ KB}$

$P_3 = 125 \text{ KB}$

$P_4 = 50 \text{ KB}$



Best Fit.



Remaining space
should be
minimum in case
of Best fit

We can not swap this
process and add into
Best fit case memory.

As it is contiguous
memory allocation method

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

→ Processes $\Rightarrow 120, 300, 100, 250, 350$

	<u>Process</u>	Internal fragmentation
First Fit	330 $\rightarrow 120$	210
	100 $\rightarrow 100$	0
	200 \rightarrow Consider this	200
	400 $\rightarrow 300$	100
	500 $\rightarrow 250$	250
		<u>760</u>

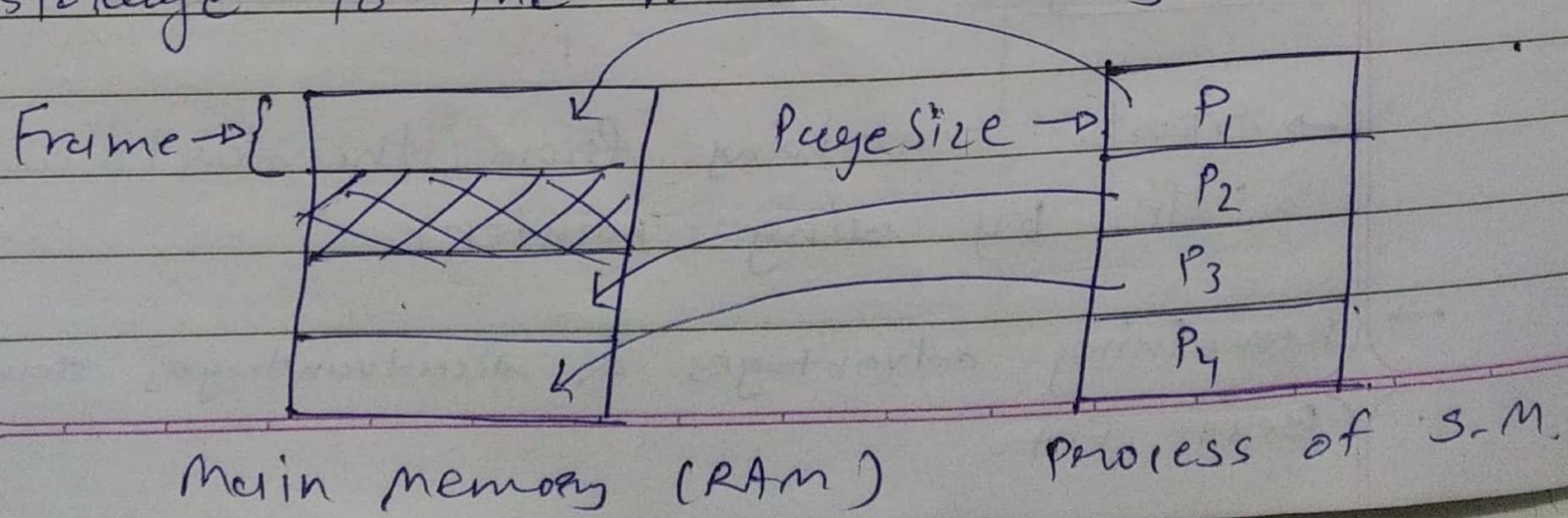
	<u>Process</u>	
Best Fit	330 $\rightarrow 300$	30
	100 $\rightarrow 100$	0
	200 $\rightarrow 120$	80
	400 $\rightarrow 250$	150
	500 $\rightarrow 350$	150
		<u>410</u>

	<u>Process</u>	
Worst Fit	330 $\rightarrow 100$	230
	100 \rightarrow	100
	200 \rightarrow	200
	400 $\rightarrow 300$	100
	500 $\rightarrow 120$	380
		<u>980</u>
		<u>1010</u>

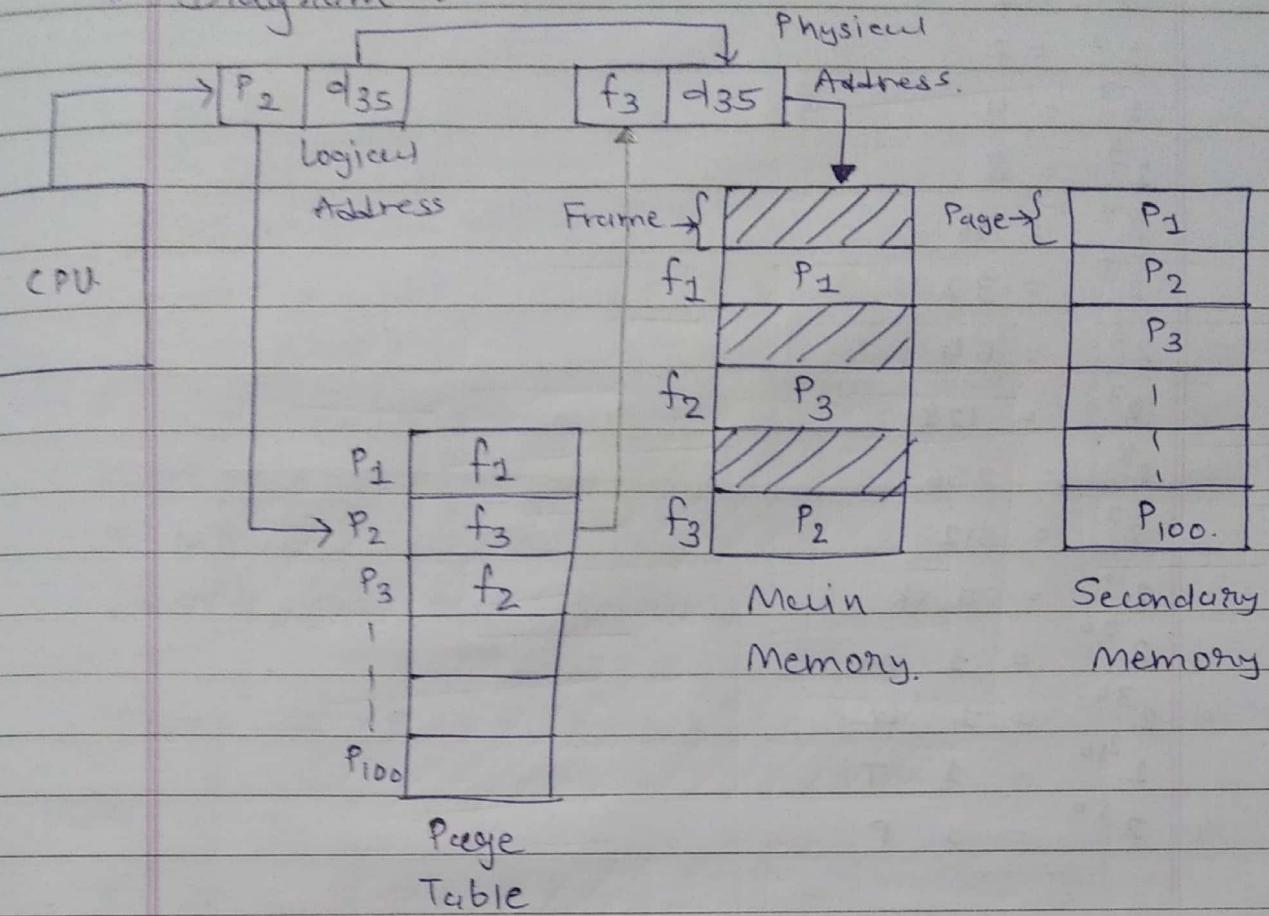
* Non Contiguous Memory Allocation :-

* 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 = 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.

$$2^0 = 1$$

$$2^1 = 2$$

$$2^2 = 4$$

$$2^3 = 8$$

$$2^4 = 16$$

$$2^5 = 32$$

$$2^6 = 64$$

$$2^7 = 128$$

$$2^8 = 256$$

$$2^9 = 512$$

$$2^{10} = 1 \text{ K}$$

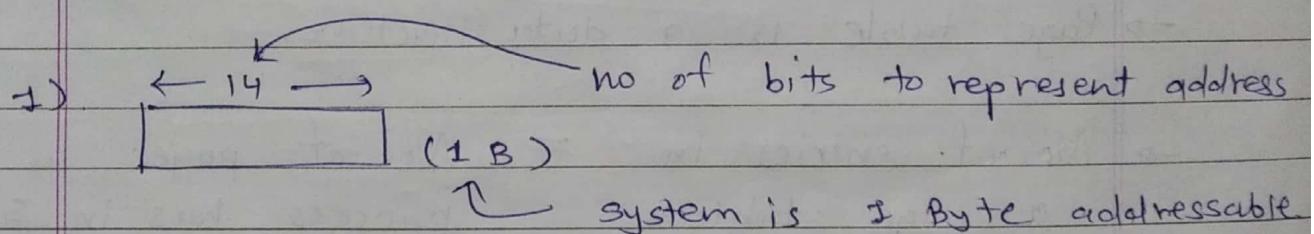
$$2^{20} = 1 \text{ M}$$

$$2^{30} = 1 \text{ G}$$

$$2^{40} = 1 \text{ T}$$

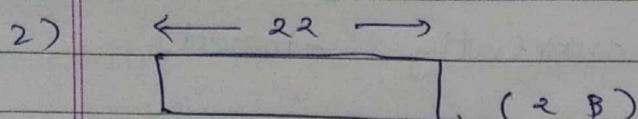
$$2^{50} = 1 \text{ P}$$

* Find the size of memory.



$$= 2^{14} \times 1$$

$$= 16 \text{ KB.}$$



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

$$= 2^{23} \text{ Bytes}$$

$$= 8 \text{ MB.}$$

* Find the number of bits in the address of

1. M.S. = 64 KB.

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

$$\Rightarrow 64 \text{ KB} = 64 \times 2^{10} \text{ Bytes} \\ = 2^6 \times 2^{10} \text{ Bytes} \\ = 2^{16} \quad \therefore 16 \text{ bits}$$

2. Memory size = 32 KB.

Size of each location = 1 Byte

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

3. M.S. = 16 GB

Size of each location = 4 B.

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

* Calculate no. of pages :-

1. LA = 24 bits

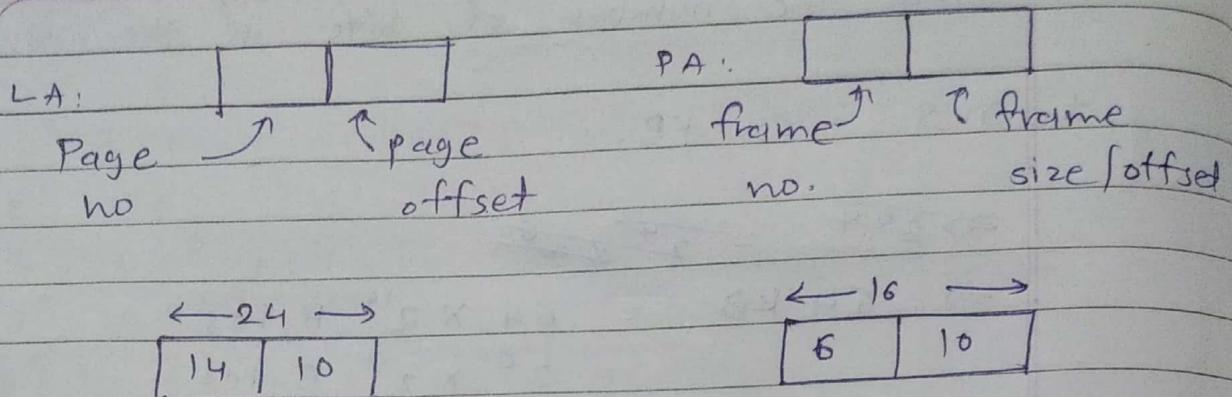
PA = 16 bits

Page size = 1 KB

→ Logical Address Space = $2^{24} = 16 \text{ MB}$

→ Physical Address Space = $2^{16} = 32 \text{ KB}$.

→ Page size = 1 KB
 $= 2^{10} \text{ Bytes.}$



$$\rightarrow \text{Total no. of pages} = \frac{\text{Process size}}{\text{Page size}}$$

$$\rightarrow \text{No. of pages} = \frac{2^{14}}{2^4 \cdot 2^{10}}$$

$$= 16 \text{ K}$$

* Examples :-

$$\textcircled{1} \quad LAS = 4 \text{ GB}$$

$$PAS = 64 \text{ MB}$$

$$\text{Page size} = 4 \text{ KB.}$$

$$\text{No. of pages} = ?$$

$$\text{No. of frames} = ?$$

$$\text{No. of entries in a page table} = ?$$

$$\text{Size of page table} = ?$$

$$\underline{\text{Soln:}} \quad LAS = 4 \text{ GB}$$

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

$$= 2^{32} \text{ Bytes.}$$

$$\therefore LA = 32 \text{ bits}$$

$$PAS = 64 \text{ MB}$$

$$= 2^5 \times 2^{20} \text{ Bytes} = 2^{26} \text{ Bytes.}$$

$$\therefore PA = 26 \text{ bits.}$$

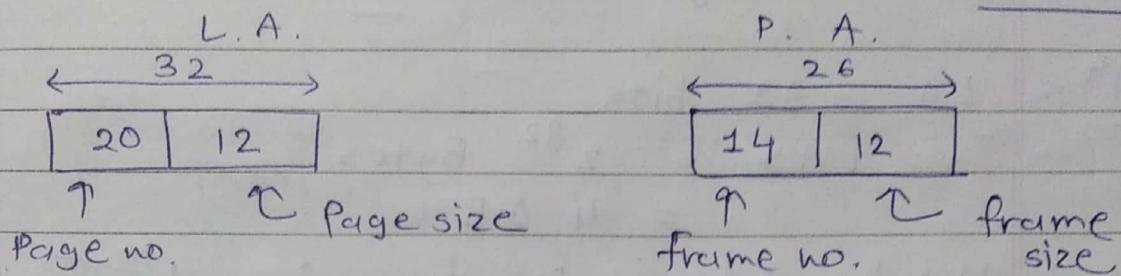
$$\rightarrow \text{Page size} = 4 \text{ KB}$$

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

$$= 2^{12} \text{ Bytes}$$

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

∴ Page size = frame size \Rightarrow is of 12 bits



$$\rightarrow \text{Total no. of pages} = 2^{20} = 1 \text{ M.}$$

$$\rightarrow \text{Total no. of frames} = 2^{14} = 16 \text{ K}$$

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

$$\therefore \text{No. of entries in a page table} = 2^{20}$$

$$= 1 \text{ M}$$

* \rightarrow Size of page table.

$$= \text{No. of entries} \times \text{No. of bits required to represent frame no.}$$

(∵ Page table contains frames)

$$= 2^{20} \times 14 \text{ bits}$$

$$= 2^{20} \times 16 \text{ bits} \quad (\because \text{approximation})$$

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

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

$$= 4 \text{ MB.}$$

(2)

$$LA = 32 \text{ bits}$$

$$\text{Page size} = 4 \text{ KB}$$

Each page table entry is of 4 Bytes.

Size of page table = ?

$$\underline{\text{Soln}}:- LA = 32 \text{ bits}$$

$$\therefore LAS = 2^{32} \text{ Bytes}$$

$$= 4 \text{ GB}$$

W.R.t. LAS = Process size \checkmark^*

$$\therefore \text{Process size} = 4 \text{ GB}$$

\rightarrow Size of page table,

No. of entries in \times Page table entry
a page table size

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

$$= \frac{2^{32}}{2^{12}} \times 4 \text{ Bytes.}$$

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

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

$$= 4 \text{ MB}$$

(3)

$$PAS = 64 \text{ MB} \Rightarrow 2^6 \times 2^{20} \text{ Bytes} = 2^{26} \text{ Bytes.}$$

$$LA = 32 \text{ bits}$$

$$\text{Page size} = 4 \text{ KB}$$

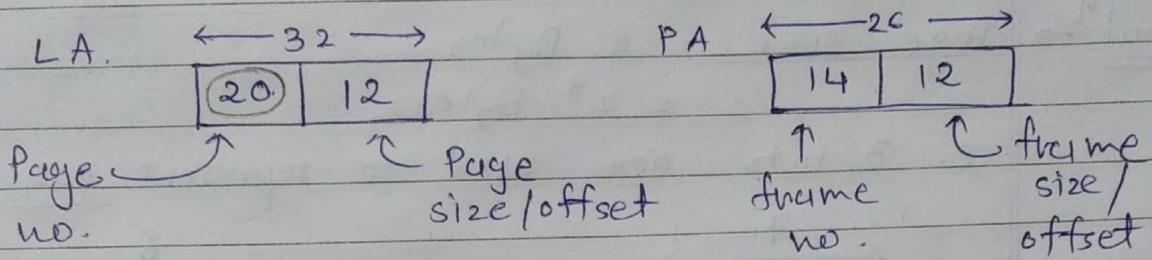
Size of page table = ?

Solⁿ: 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,

$$= \text{No. of entries in a page table} \times \frac{\text{Page table entry size}}{\text{Page size}}$$

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

$$= \frac{2^{32}}{2^{12}} \times 14 \text{ bits. Frame no.}$$

$$= 2^{20} \times 16 \text{ bits } (\because \text{approximation})$$

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

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

$$= 4 \text{ MB}$$

1 Byte = 8 Bits.

Date _____

Page _____

(4)

LA = 7 bits

PA = 6 bits

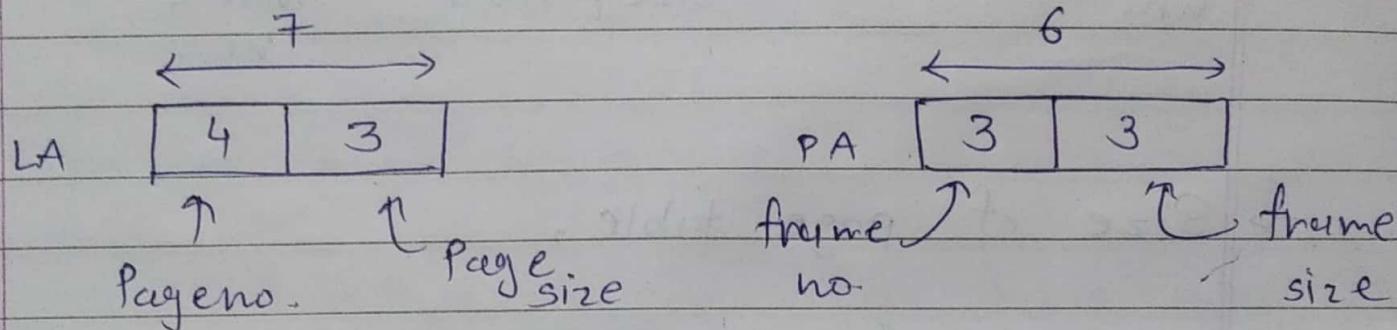
Page size = 8 words or Bytes

No. of pages = ?

No. of frames = ?

Solⁿ → Page size = 8 Bytes
= 2^3 Bytes.

∴ 3 bits are used to represent page size.



$$\rightarrow \text{No. of frames} = 2^3 = 8$$

$$\rightarrow \text{No. of pages} = 2^4 = 16$$

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

$$LA = ? \quad , \quad PA = ?$$

→ $\boxed{1 \text{ word} = 2 \text{ Bytes}} \quad *$

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

$$\begin{aligned} \therefore 1 \text{ page contains} & 1024 \text{ words} \\ & = 2048 \text{ Bytes} \\ & = 2^{11} \text{ Bytes} \end{aligned}$$

$\therefore 1 \text{ page can be represented using } 11 \text{ bits.}$

→ Logical address space contains 8 pages.

$$\begin{aligned} \therefore LAS &= 2^{11} \times 8 \\ &= 2^{11} \times 2^3 = 2^{14} \text{ Bytes} \\ &= 16 \text{ MB} \end{aligned}$$

$$\therefore \text{Logical Address} = \underbrace{14}_{\text{bits}}$$

→ Physical memory contains 32 frames.

and we know that page size = frame size

$$\therefore 1 \text{ frame is of } 2^{11} \text{ Bytes.}$$

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

$$\therefore PA = 16$$

* Multilevel Paging :-

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

levels of paging = 2
 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	0
----	---

 frame no. ↑ frame offset.

∴ Number of frames = 2^{32}

→ Size of Main memory = No of frames \times frame size
 $= 2^{32} \times 2^{20}$
 $= 2^{52}$ Bytes.

∴ PA =

32	20
----	----

 frame no. ↑ frame size / offset.

→ VA =

44	20
----	----

 Page no. ↑ page size / offset.

→ Process size = 2^{64} Bytes

LFA

✓

(∴ process is present in logical address space)

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

*.
= No. of pages page table
the process is X entry size
divided

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

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

→ We can observe,

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

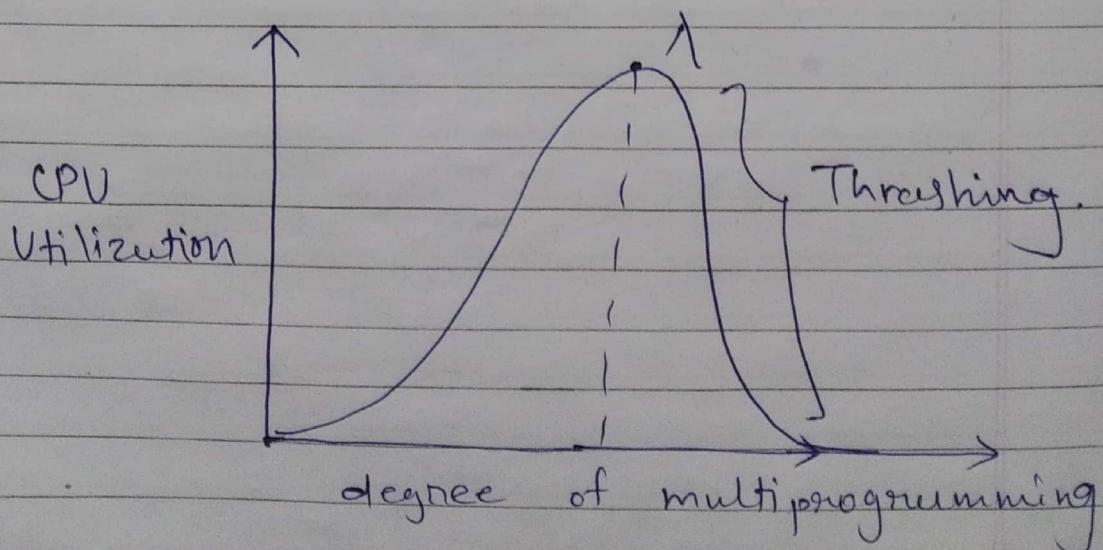
→ Thus, inner page table can NOT ~~be stored~~ be stored in a single frame.

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

→ No. of pages in 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.

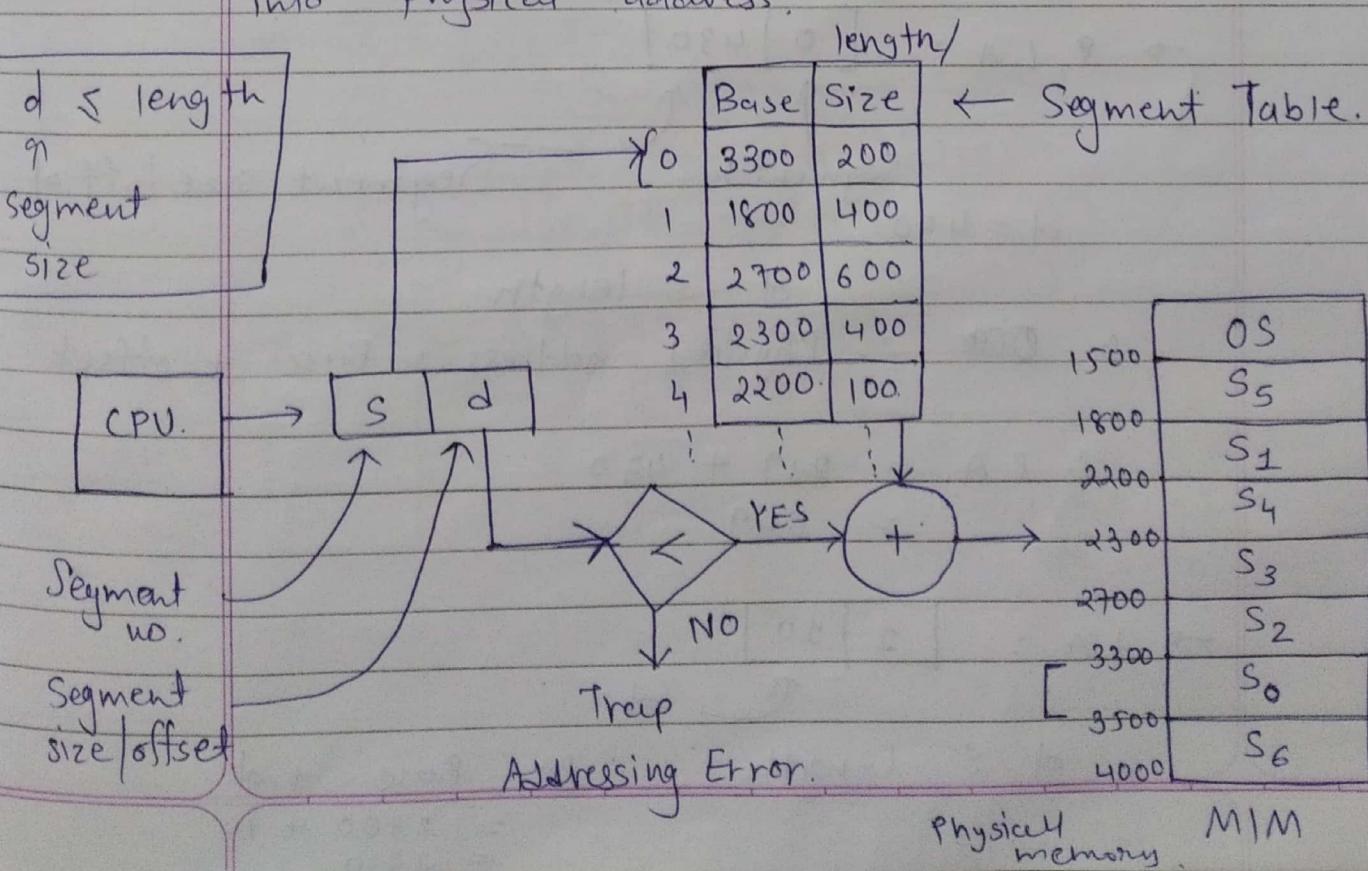
→ It transfers as many processes as possible into ready state.

* 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.

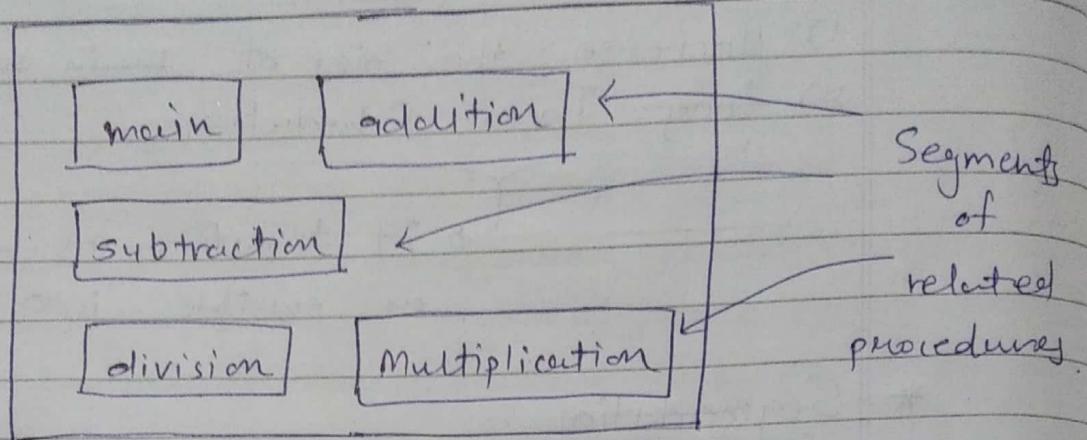
→ Segments can be of various size.

→ MMU helps in transforming 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?

→ P LA :

0	430
---	-----

↓ ↑

Segment no. Segment size/offset

$$d = 430.$$

$d \leq \text{length}$.

∴ PA : Physical address = Base + offset.

$$\begin{aligned} \therefore PA &= 219 + 430 \\ &= 649 \end{aligned}$$

→ LA :

1	10
---	----

↓ d

$d \leq \text{length}$

$$\begin{aligned} \therefore PA &= \text{Base} + d \\ &= 2300 + 10 \\ &= 2310. \end{aligned}$$

$$\rightarrow LA = [\underline{1} \quad \underline{1}]$$

$d \leq \text{length}$.

$$\begin{aligned}\therefore PA &= \text{Base} + \text{offset} \\ &= 2300 + 11 \\ &= 2311.\end{aligned}$$

$$\rightarrow LA : [\underline{2} \quad \underline{500}]$$

$\uparrow d$

$d \leq \text{length}$ is not true.

\therefore Invalid address.

(2)

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

$$\rightarrow LA = [\underline{0} \quad \underline{300}]$$

\uparrow segment size (d)

$d \leq \text{length}$ is false

\therefore invalid address.

$$\rightarrow LA = [\underline{2} \quad \underline{350}]$$

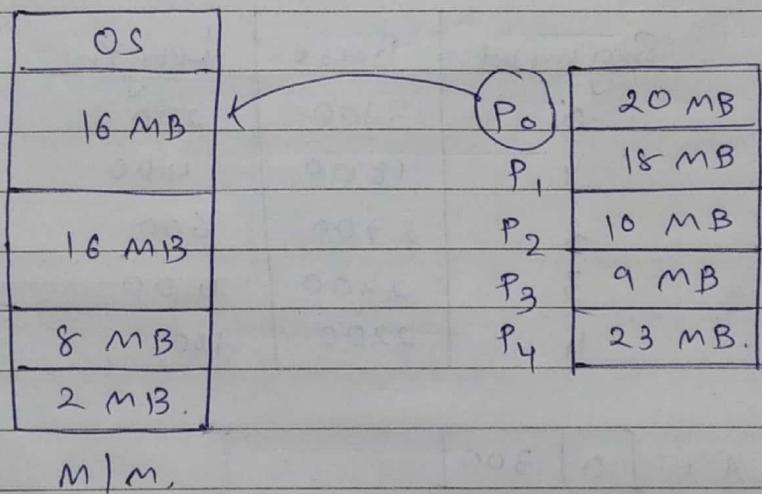
$\uparrow d$

$d \leq \text{length} \therefore PA = \text{Base} + d$

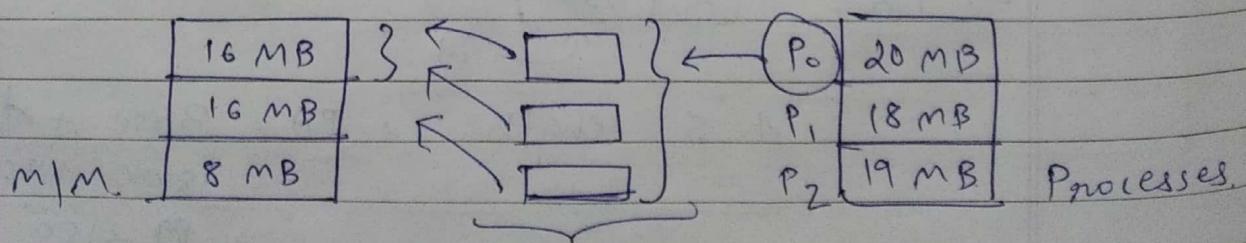
$$\begin{aligned}&= 1800 + 350 \\ &= 2150.\end{aligned}$$

* 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.

* Virtual Memory :- (Demand Paging)

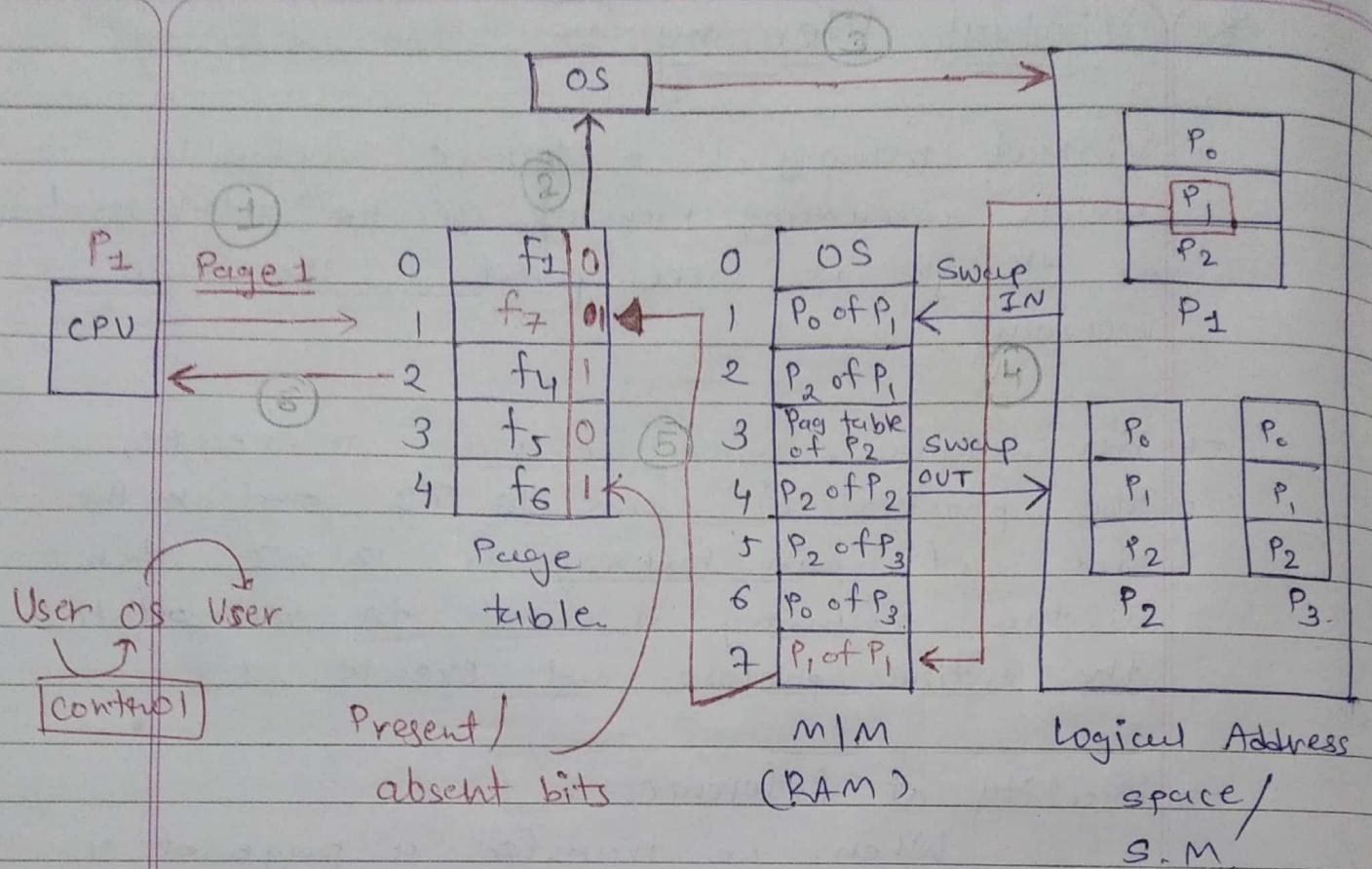
- Virtual memory is a storage scheme in which secondary memory can be addressed as though it were part of the main memory.
- For instance, when we want to execute the process of size 16 GB and the size of main memory is 12 GB then virtual memory is used to accommodate the entire process and execute it.
- Locality of Reference :
When we transfer a page of a process then we also transfers 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 block 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 number 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 :-

$$EMAT = P(\text{Page fault}) + (1 - P)(\frac{\text{MM access time}}{\text{service time}})$$

where P = probability of occurring page fault.

29

Ex:

$$\text{VA} = \underline{\underline{17}} \text{ bits} \quad \text{Page size} = 4 \text{ KB}$$

$$\text{PA} = \underline{\underline{123}} \text{ bits}$$

$$\text{No of pages in VAS} = ?$$

$$\text{Page no. (P) in bits} = ?$$

$$\text{No of frames in PAS} = ?$$

$$\text{Frame number (f) in bits} = ?$$

$$\text{Offset (d)} = ?$$

$$\text{Page table size} = ?$$

$$\underline{\underline{\text{Soln}}} : \text{Page size} = 4 \text{ KB} = 2^2 \times 2^{10} \text{ B} = 2^{12} \text{ B}$$

$$\therefore \text{No of pages} = 12 = \text{no. of frames}$$

$$\rightarrow \text{VA: } \boxed{17} \boxed{12} \qquad \text{PA: } \boxed{11} \boxed{12}$$

$\leftarrow 29 \rightarrow$ $\leftarrow 23 \rightarrow$

$$\text{No. of pages} = 17 \text{ bits} \quad \text{No of frames in PA} = 11 \text{ bits}$$

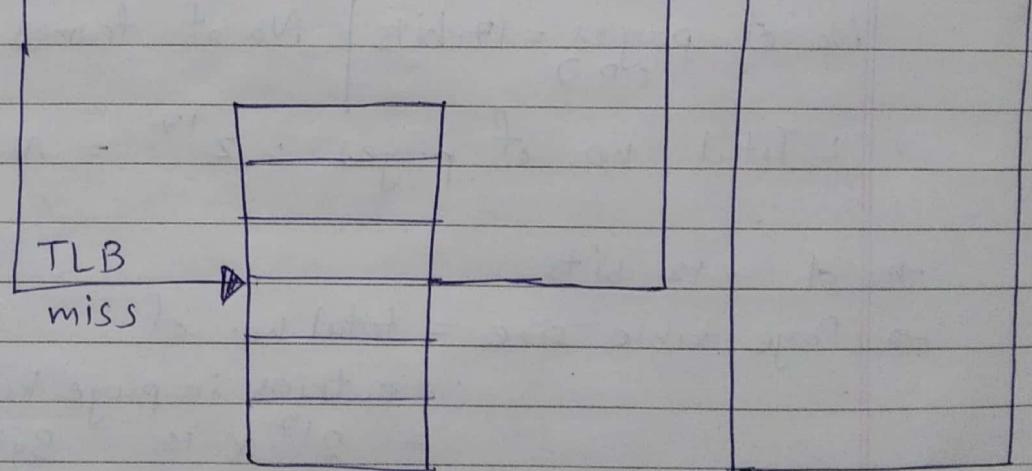
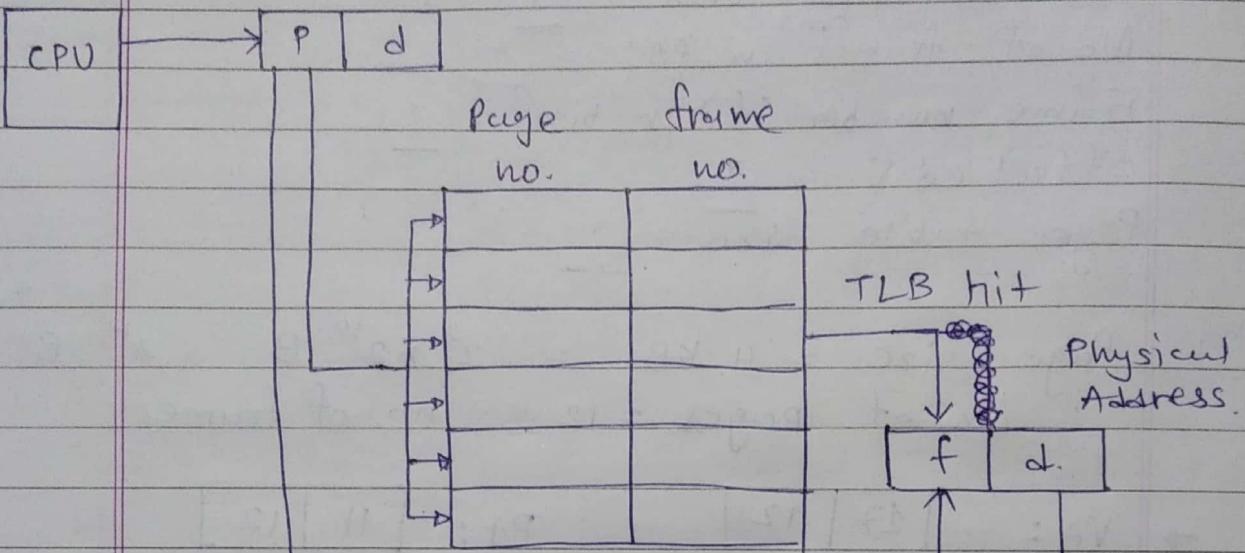
$$\therefore \text{Total no. of pages} = 2^{17} = N.$$

$$\rightarrow d = 12 \text{ bits.}$$

$$\begin{aligned} \rightarrow \text{Page table size} &= \text{total no. of entries in page table} \times \text{Page table entry size} \\ &= 2^{17} \times 11 \text{ Bytes} \\ &= 2^{17} \times 16 \text{ Bytes} \quad (\because \text{approximate}) \\ &= 2^{17} \times 2^4 \\ &= 2^{21} \text{ Bytes} \\ &= 2 \text{ MB.} \end{aligned}$$

* Translation Lookaside Buffer (TLB) :-

logical
Address.



(Refer Page table. (MIM) Physical memory
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.

* 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 = $\boxed{x(c+m) + (1-x)(c+2m)}$

OR

$$\left(\text{hit memory time} \right) + (1-p) \left(\begin{array}{l} \text{miss memory} \\ \text{time} \end{array} \right)$$

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

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

$$= 114 + 11$$

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

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

~~$= 21.7$~~

~~$= \frac{21.7}{10} \text{ ns}$~~

→ 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.

Solⁿ → Effective memory access time = $x(c+m) + (1-x)(c+2m)$

$$= \left(\frac{9}{10}\right)(10+50) + \left(\frac{1}{10}\right)(10+100)$$

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

$$= 54 + 11 \text{ ns if page fault}$$

$$= 65 \text{ ns} + \underline{t} \leftarrow \begin{array}{l} \text{occurs} \\ \text{Page fault service time} \end{array}$$

* 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

$$= 8 \times 10^{-3} \text{ s}$$

$$= \frac{8 \times 10^{-3}}{10^{-9}} \text{ ns}$$

$$= 8000000 \text{ ns}$$

→ Effective Memory Access Time = p (page fault service time)

$$+ (1 - p) (\text{Main memory access time})$$

+ swap page out

+ swap page in + Overhead.

* p = probability of occurring
page fault

→ p = probability of occurring page fault

$$= 1 - 99.99$$

$$= 1 - \frac{99.99}{100}$$

$$\therefore p = 0.0001 \Rightarrow 1-p = 0.9999$$

$$\rightarrow EMAT = (0.0001)(8000000) + (200)(0.9999)$$

$$= \frac{1000}{100000} \times 8000000 + 200 \times \frac{9999}{10000}$$

$$= 800 + 199.98$$

$$= \boxed{999.98 \text{ ns}}$$



Page Replacement Algorithms :-

FIFO (First In First Out) :-

→ The frame which was occupied at first should 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	X	4	4
•	2	2	X	1	1	1	1	X	3	3
1	1	X	4	4	4	5	5	5	5	5
A	A	A	A	A	A	A	A	A	A	A

Total 9 page faults.

4 frames :-

	4	4	4	4	4	4	4	3	3	3
•	3	3	3	3	3	3	X	2	2	2
2	2	2	2	2	X	1	1	1	X	5
1	1	1	1	1	X	5	5	5	4	4

Page fault count →

Total 10 page faults.

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

For 4 frames
Examples ↓

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

* → 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,

<u>Frames</u>	<u>Hit</u>	<u>Miss or faults</u>
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, τ is the longest.

i. Replace it.

In this case, both are not in future.

∴ Select any one.

\therefore Total page fault : 8 and Hit = 12.

$$\rightarrow \text{Hit Ratio} = \frac{12}{20} = 0.6$$

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

* Least Recently Used Page Replacement

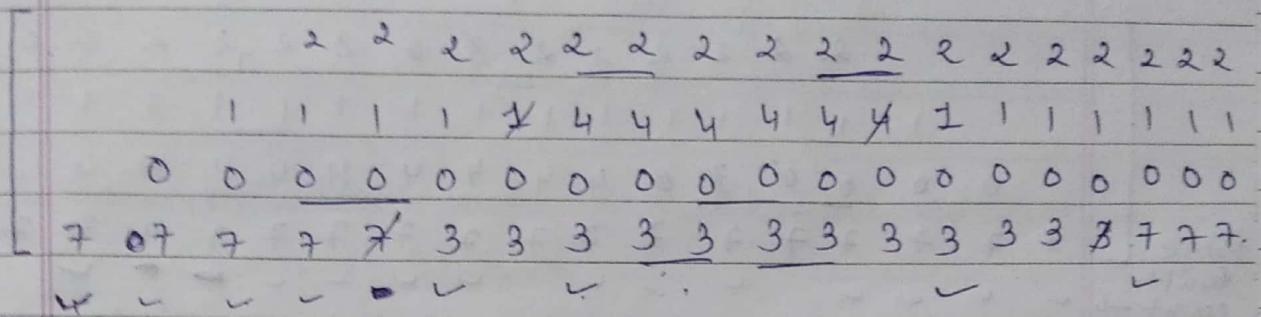
Algorithm :-

→ Replace the least recently used page in past.

→ Replace the page which ~~was~~ was used very firsty. 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

(as 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 :-

→ Total page faults = 12 , Hits = 8

* Hard Disk Architecture :-

- Platters → Surface → Track → Sectors → Data
- Disk size = $P \times S \times T \times S \times D.$
- 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

$$\begin{aligned}
 &= 8 \times 2 \times 256 \times 512 \times 512 \text{ KB.} \\
 &= 2^3 \times 2^1 \times 2^8 \times 2^9 \times 2^9 \times 2^10 \text{ B} \\
 &= 2^{40} \text{ Bytes} \\
 &= 1 \text{ TB.}
 \end{aligned}$$

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

- (2) Total cylinders = 480
- cylinder zones = ~~120~~ 120
- each contains different sizes
 200, 240, 280, 320.
 each sector contains 4096 Bytes.

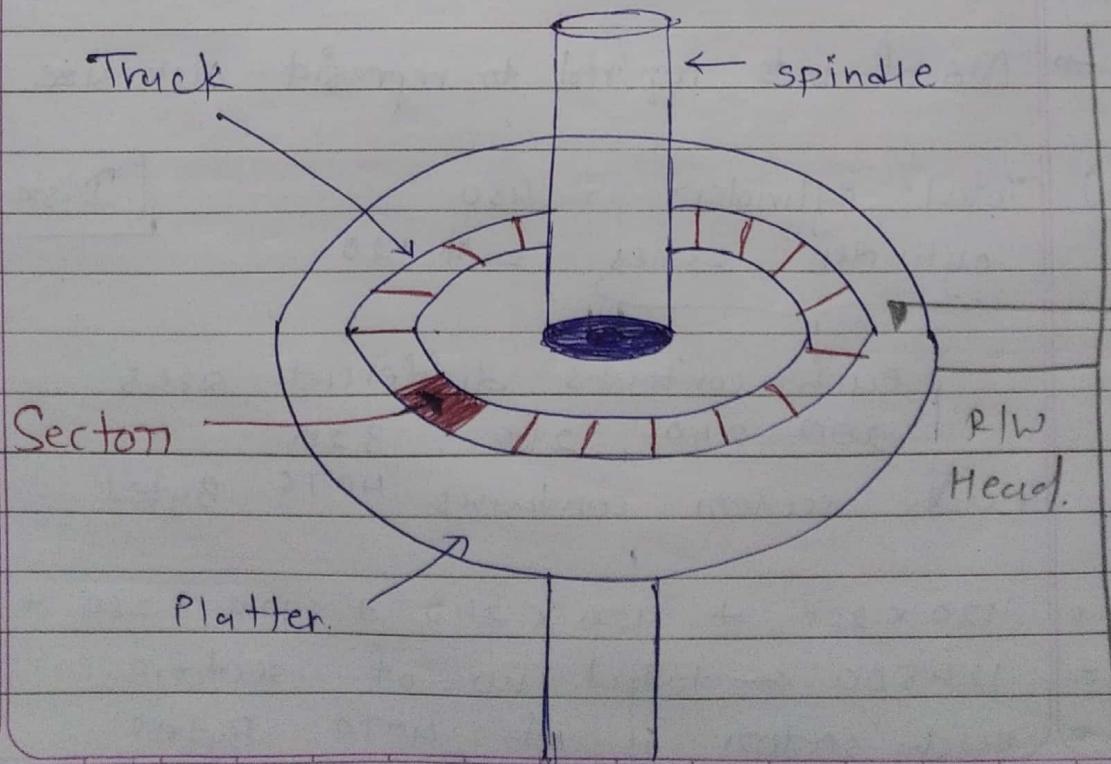
Disk Size

→ $120 \times 200 + 120 \times 240 + 120 \times 280 + 120 \times 320.$
 = 124800 ← total no. of sectors.
 → each sector is of 4096 Bytes.
 ∴ size of disk = $124800 \times 4096 = 51180800$ 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 =
$$\left(\begin{array}{l} \text{No of Heads} \times \text{Capacity of one track} \times \text{No. of Rotations in} \\ \text{Data Rate} \qquad \qquad \qquad \qquad \qquad \text{one second} \\ \uparrow \\ \text{Surface} \end{array} \right)$$



→ 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 \approx 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)

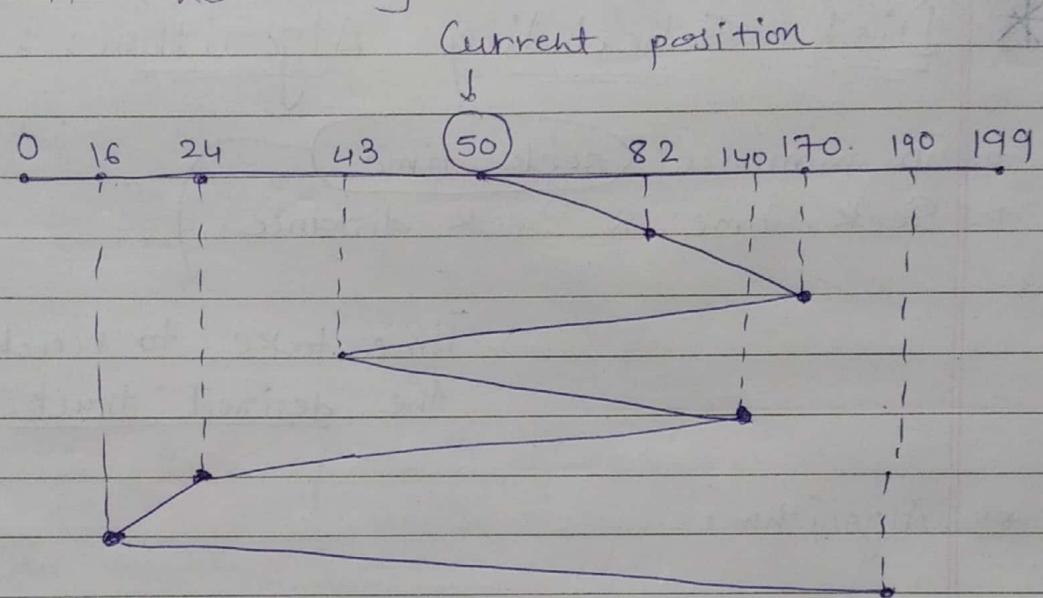
Ex → A disk contains 200 tracks (0-199).

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

$$\begin{aligned}
 &= (82 - 50) + (170 - 82) + (170 - 43) + \\
 &\quad (140 - 43) + (140 - 24) + (24 - 16) + \\
 &\quad (190 - 16)
 \end{aligned}$$

$$\begin{aligned}
 &= 32 + 88 + 127 + 97 + 116 + 8 + 174 \\
 &= 642
 \end{aligned}$$

→ Advantages :-

→ No starvation

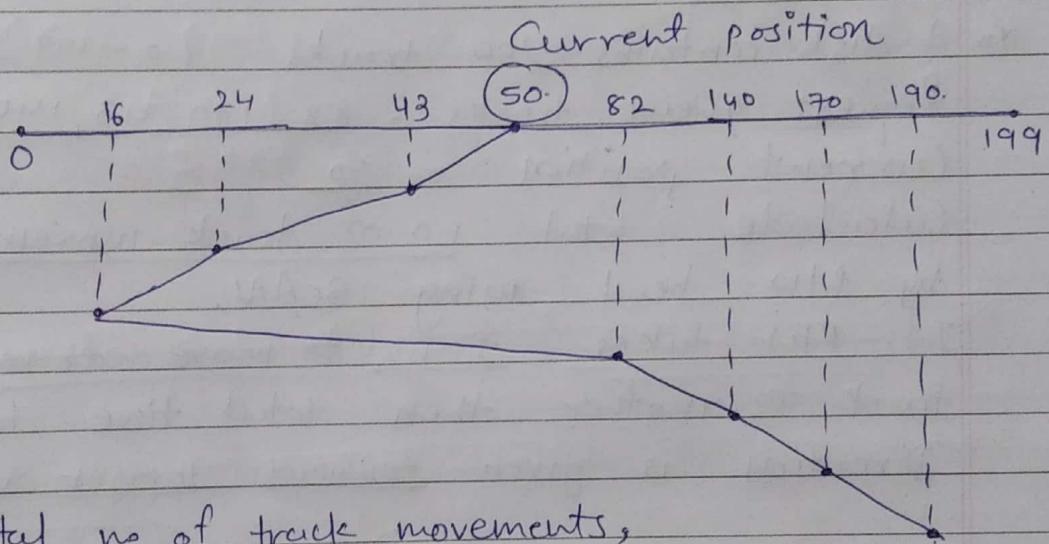
→ Performance decreases

* 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 RW head = 50. Calculate the total no of track movement by RW head using shortest seek time first. If RW head ~~takes~~ 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,

$$\begin{aligned}
 &= (50 - 16) + (190 - 16) \quad \text{OP} \\
 &= (50 - 43) + (43 - 24) + (24 - 16) + \\
 &\quad (82 - 16) + (140 - 82) + (170 - 140) + \\
 &\quad (190 - 170) \\
 &= 34 + 206 \quad \mid = 7 + 19 + 8 + 66 + 58 + \\
 &= \cancel{206} \quad 208 \quad \mid 30 + 20. \\
 &= 208.
 \end{aligned}$$

→ Total track movement = 208.

1 ns is taken for one track movement

∴ total 208 ns required

→ Advantage :-

→ Tries to give optimised 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

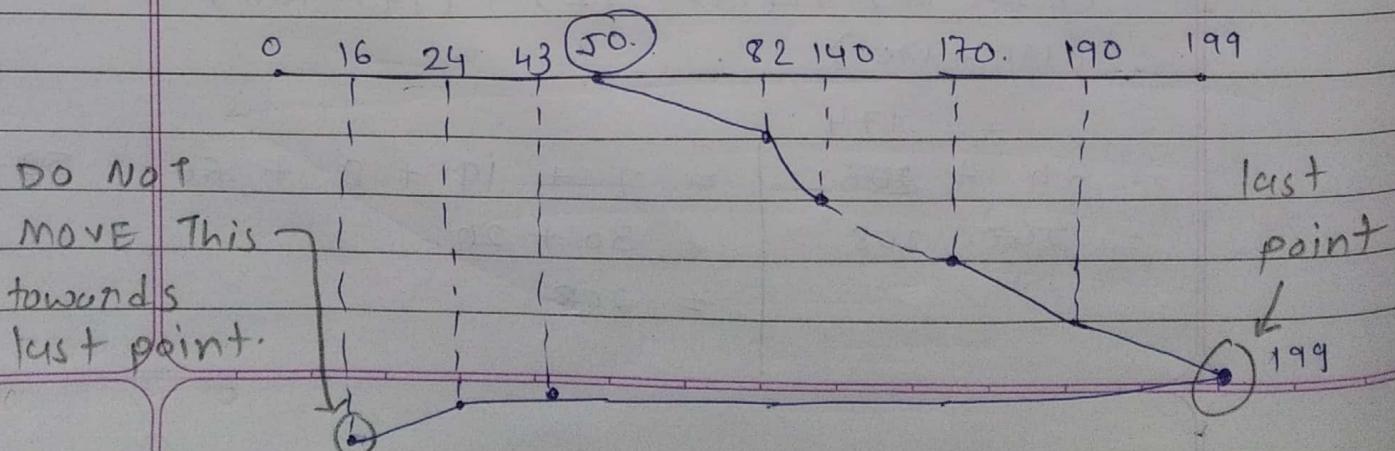
Current position = 50

Calculate total no of track movement

by HW head using SCAN.

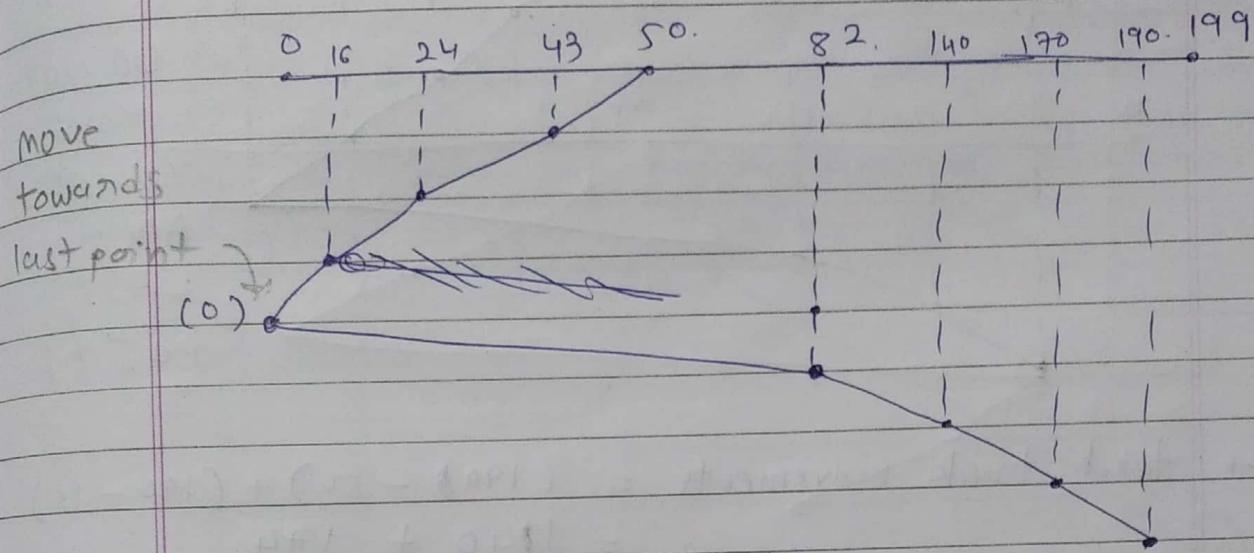
If RW takes 1 ns to move from one track to another then total time take → q
Direction is given towards larger value.

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



→ Total no. of tracks = $(199 - 50) + (199 - 160)$
 $= 149 + 189$
 $= \boxed{332}$

* If direction is given towards smaller value,



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

→ $\boxed{249 \text{ ns}}$

→ Disadvantage :-

→ When request queue tracks changes dynamically, starvation occurs.

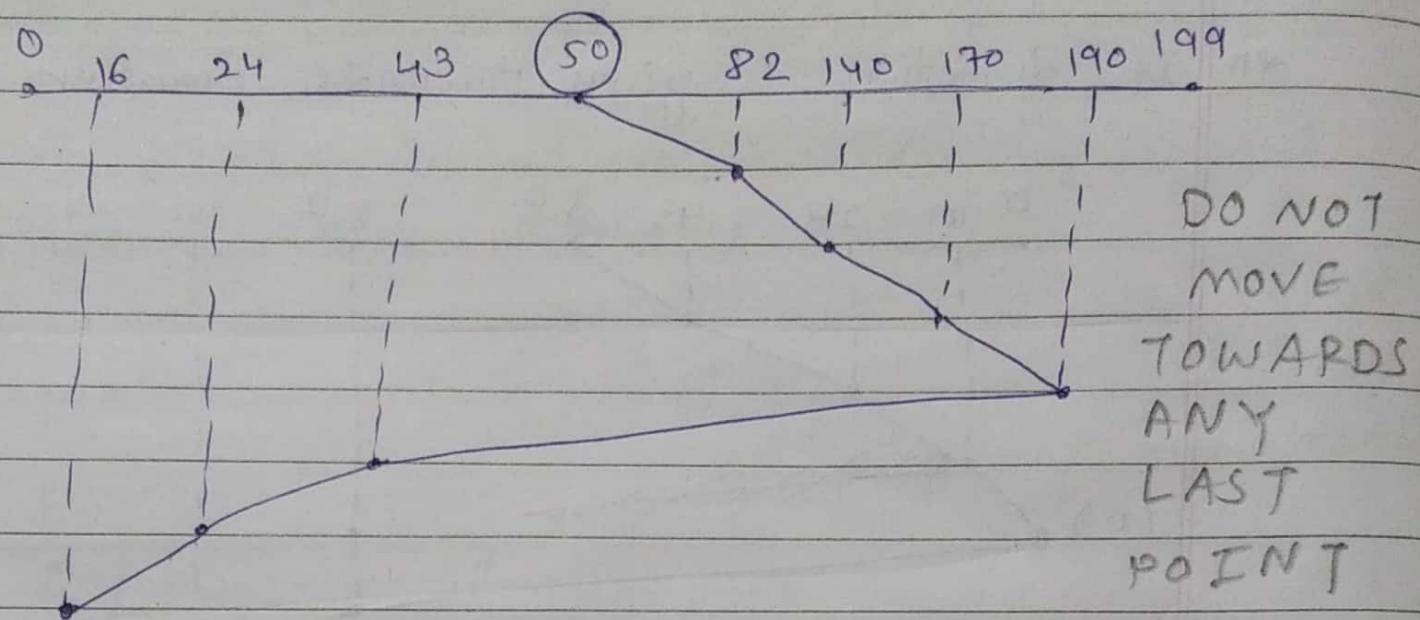
* LOOK :-

* LOOK algorithm is same as SCAN algorithm 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



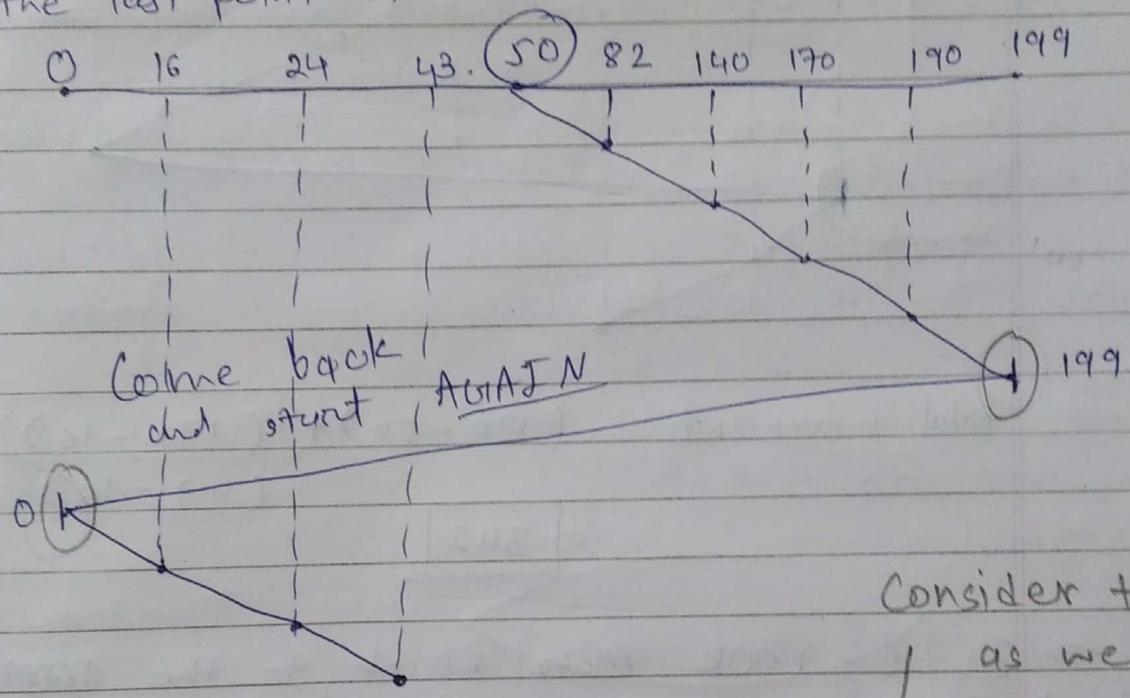
$$\begin{aligned}
 \rightarrow \text{total track movements} &= (190 - 50) + (190 - 16) \\
 &= 140 + 174 \\
 &= 314
 \end{aligned}$$

* C-SCAN:

Circular SCAN.

Ex:- Request queue track = 82, 170, 43, 140, 24, 16, 190
 Current pos = 50
 total no. of 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.



$$\rightarrow \text{total move} = (199 - 50) + [(199 - 0) + (43 - 0)] \\ = [391]$$

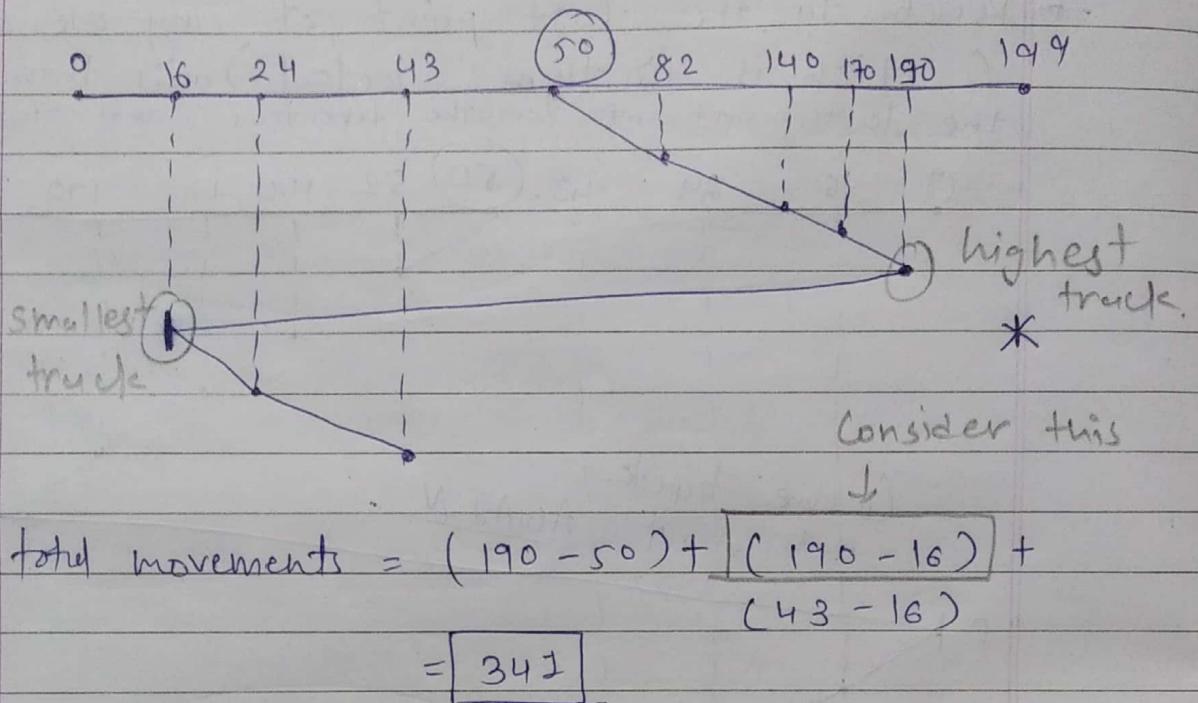
* C - LOOK : Circular LOOK

Ex: 200 track

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

Current pos = 50.

direction towards large value.



In C - LOOK ~~Move~~ Reach to the ~~next~~ highest track ~~then~~ 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 highest no. track and start again.

* File System :-

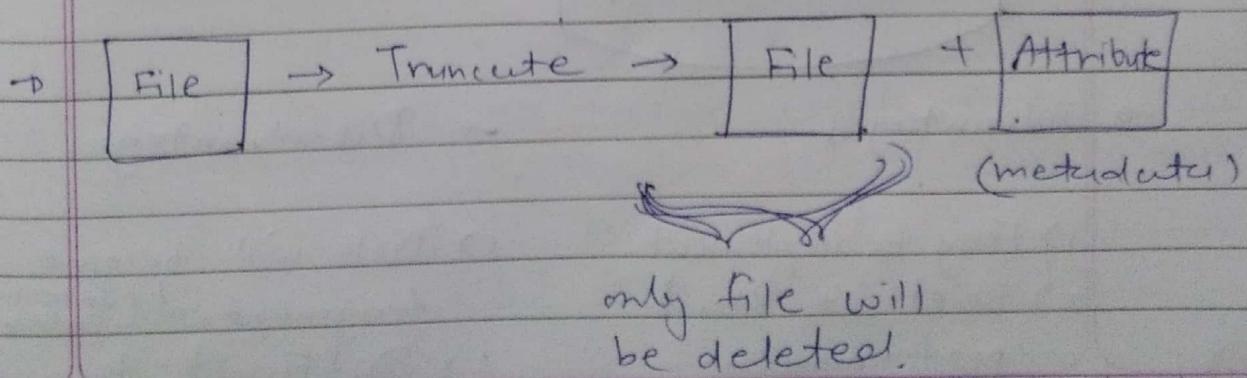
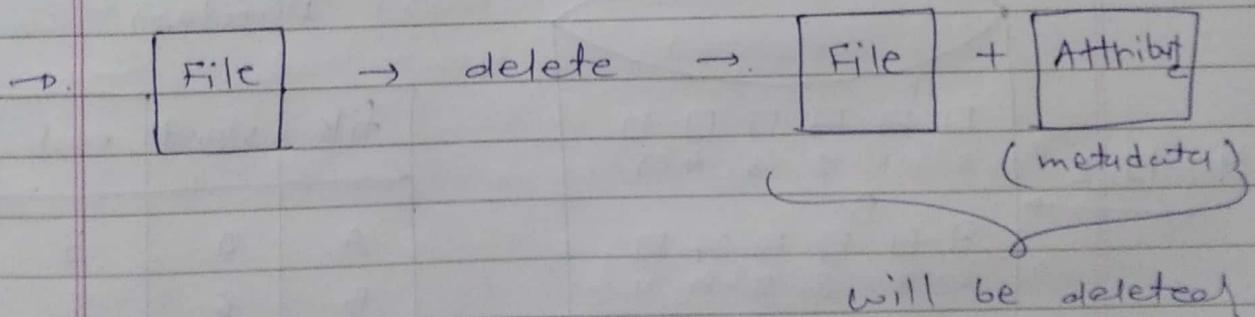
→ User → file → folder / directory → file system

How the data is stored
How the data is fetched

* Operations on files and File Attributes

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

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



* Allocation Methods in File system :-

→ Contiguous Allocation

→ Non Contiguous Allocation.

→ Linked list allocation

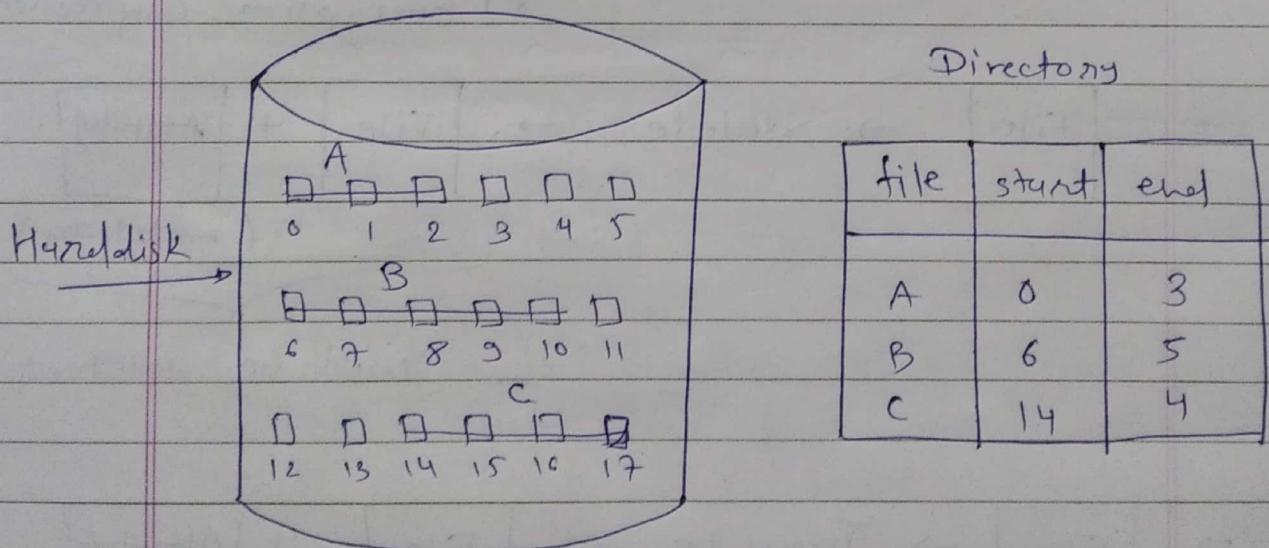
→ Indexed allocation

→ Purposes :-

1) Efficient disk utilization

2) Access faster

* Contiguous Allocation :-



→ Advantages :-

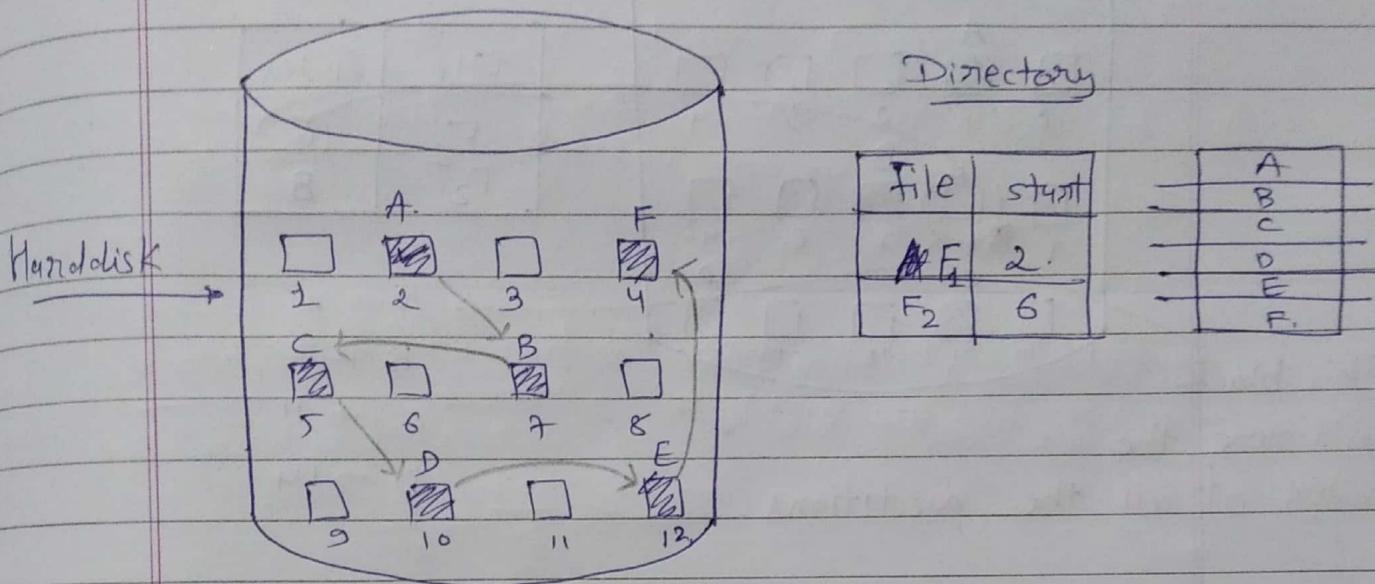
- 1) Easy to implement.
- 2) Excellent read performance

→ Disadvantages :-

- 1) Disk will become fragmented (^{internal & external fm.})
- 2) Difficult to grow file.

* Non-contiguous Allocation :-

(1) Linked-list allocation :-



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

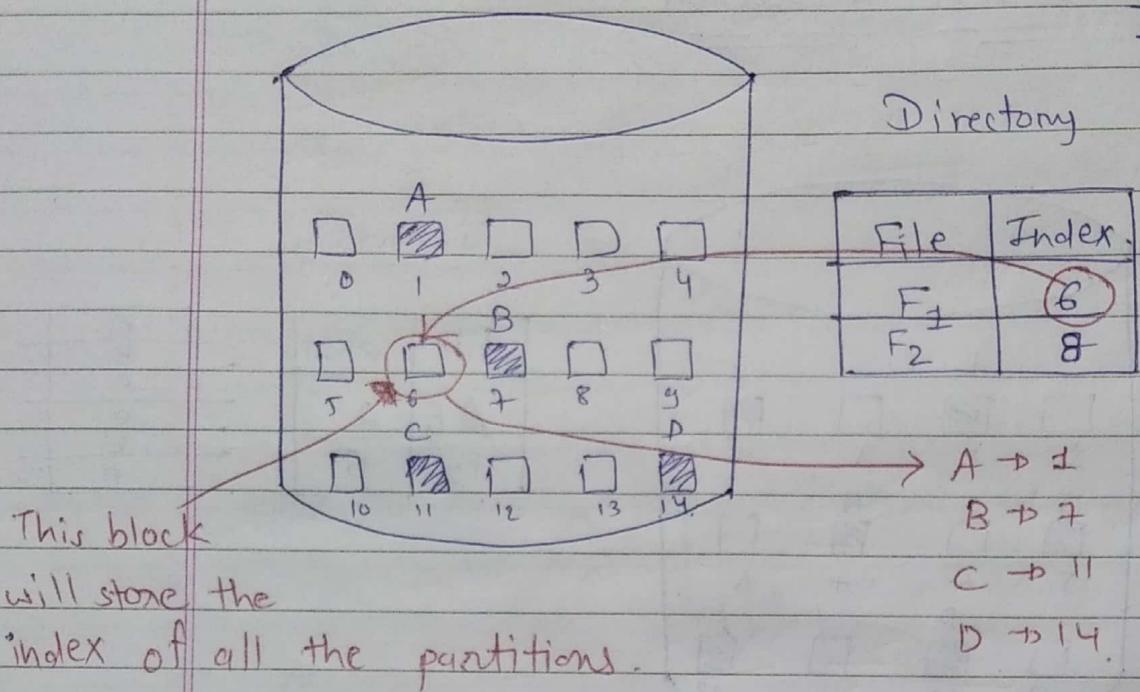
Data
Pointer

→ 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 :-



→ Advantages :

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

→ Disadvantages :

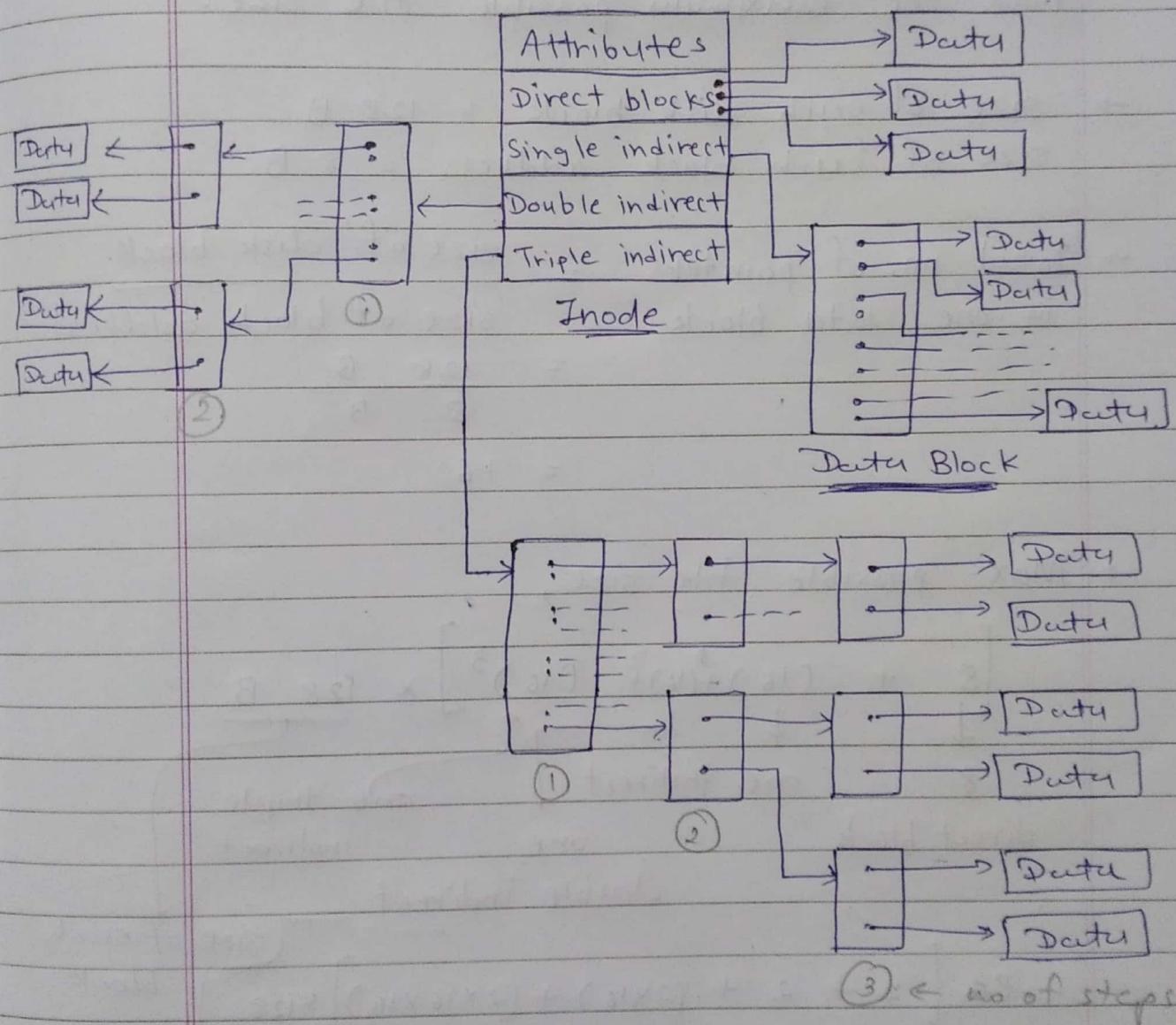
- 1) Pointer overhead
- 2) Multilevel index

* Unix Inode Structure :-

↓ ↓

OS File system

i → index , nod → block



Ques: 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.

$$\rightarrow \text{Total no. of pointers} = \frac{\text{size of disk block}}{\text{size of block address}}$$

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

$$= 16$$

→ Max. possible file size;

$$\therefore \text{Total number of ways} = 2^3 (1+2+32+512) 2^7$$

$$= 2^{10} (547)$$

$$= \boxed{547 \quad KB}$$