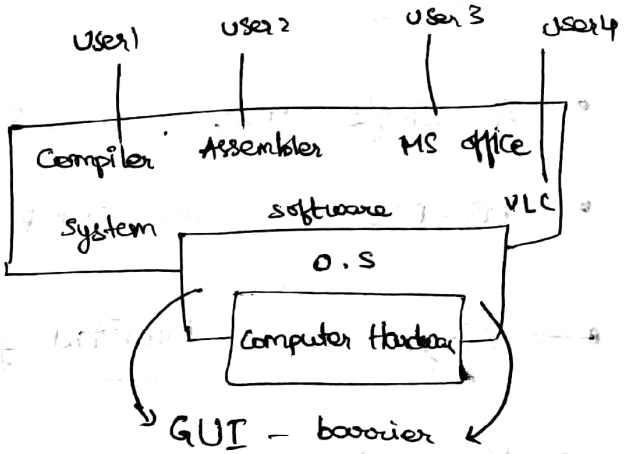
*. Operating System :

- O.S is interface between user and hardware (GUI)
- Resource Allocator – RAM, primary memory
- O.S works as Manager : Memory, process, files, security.

*. Types of O.S :

1. Batch OS
2. Multiprocessing OS
3. Multi-tasking OS
4. Multiprogramming OS
5. Real time OS

*. Abstract view of system :

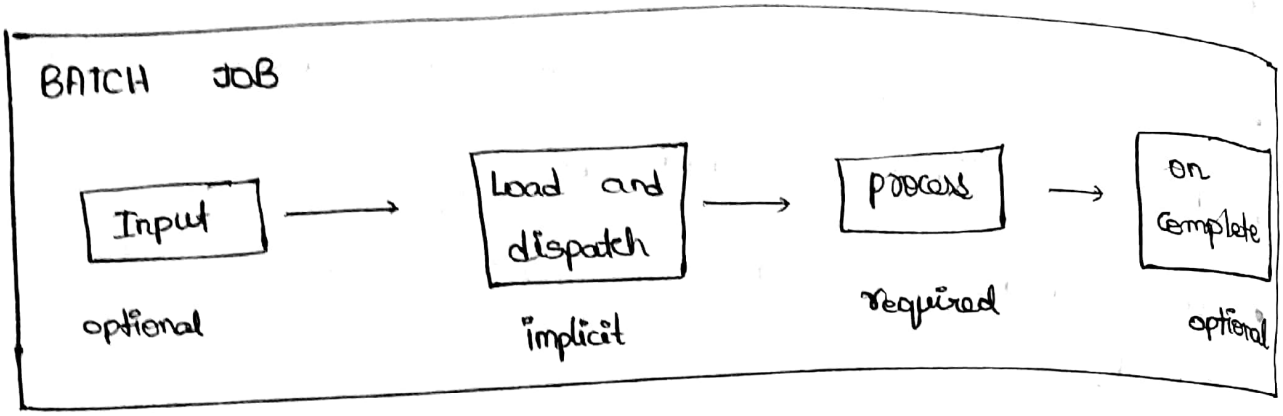


*. Goals → Convience – primary
→ Efficiency – secondary

*. Batch processing :

- It is the execution of non-interactive processing tasks, meaning tasks with no user-interface.
- It involves processing multiple data with items together as batch.
- The term is associated with scheduled processing jobs run in off-hours, known as batch window.
- It allows capital investments in computing hardware to be fully utilized and for limited processing power to be

reserved for high - priority tasks during business hours.



*. Single - user operating System :

- A single user to execute one program at a time.
- MS-DOS is an example of this kind of OS.

→ Single - user, Multitasking OS :

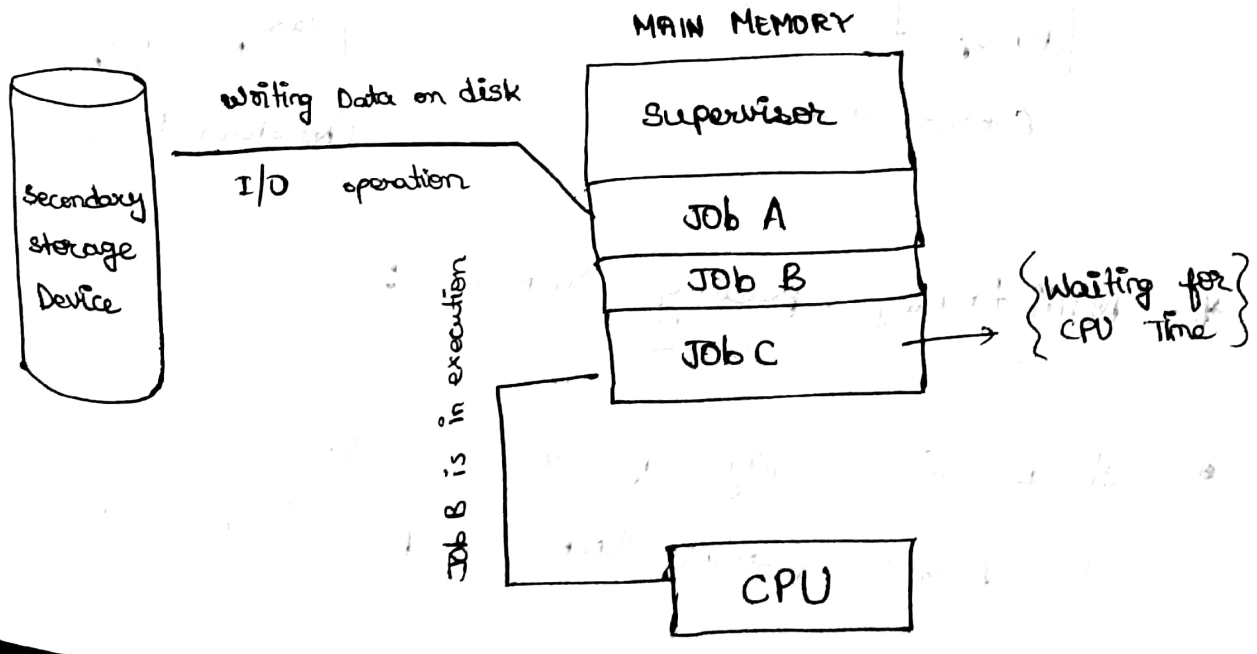
- More than one program can be executed at a time on a single computer.
- It allows a single user to execute multiple programs at the same time.
- The windows and Mac OS are examples here.

*. For example,

- In windows you can load multiple programs at a tim such as Ms-Excel, Ms-World, Ms-Access as well as you can listen the music.

*. MULTI - PROGRAMMED BATCH SYSTEM :

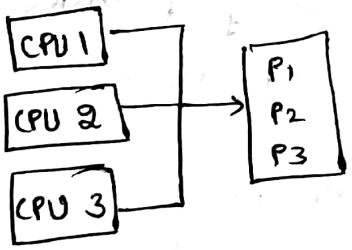
- The OS keeps multiple jobs in main memory at a time.
- There are many jobs that enter the system.
- In general, the main memory is too small to accomdate all the jobs.
- So the jobs that enter the system to be executed are kept initially on the disk in the job pool.
- When the operating system selects a job from a job pool, it loads that job into memory for execution.
- Increased throughput - Faster execution, but not 100% linear speed
- Economy of scale - Peripherals, disks, memory, shared among processors.
- Increased reliability - Failure of CPU slows system, No v crash
  - Redundant processing provides system of checks & balances.



* • Multiprocessing Systems :

- A computers capability to process more than one task simultaneously is called multiprocessing.
- It is capable of running many programs simultaneously and most modern network operating systems (NOS s) support multiprocessing.
- These operating systems include Windows NT, 2000, XP.
- The main reason why multiprocessing is more complicated than single - processing is that their operating systems are responsible for
  - → Allocating resources to competing process in a controlled environment.

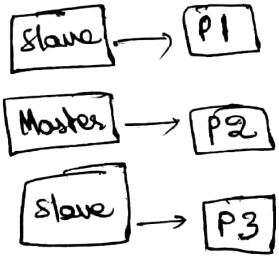
Symmetric Multiprocessing



(shared memory)

{ vs }

Asymmetric Multiprocessing



(No shared Memory)

* • Multi tasking operating Systems :

- It is the ability of an OS to execute more than one program (called; task or process) at a time.

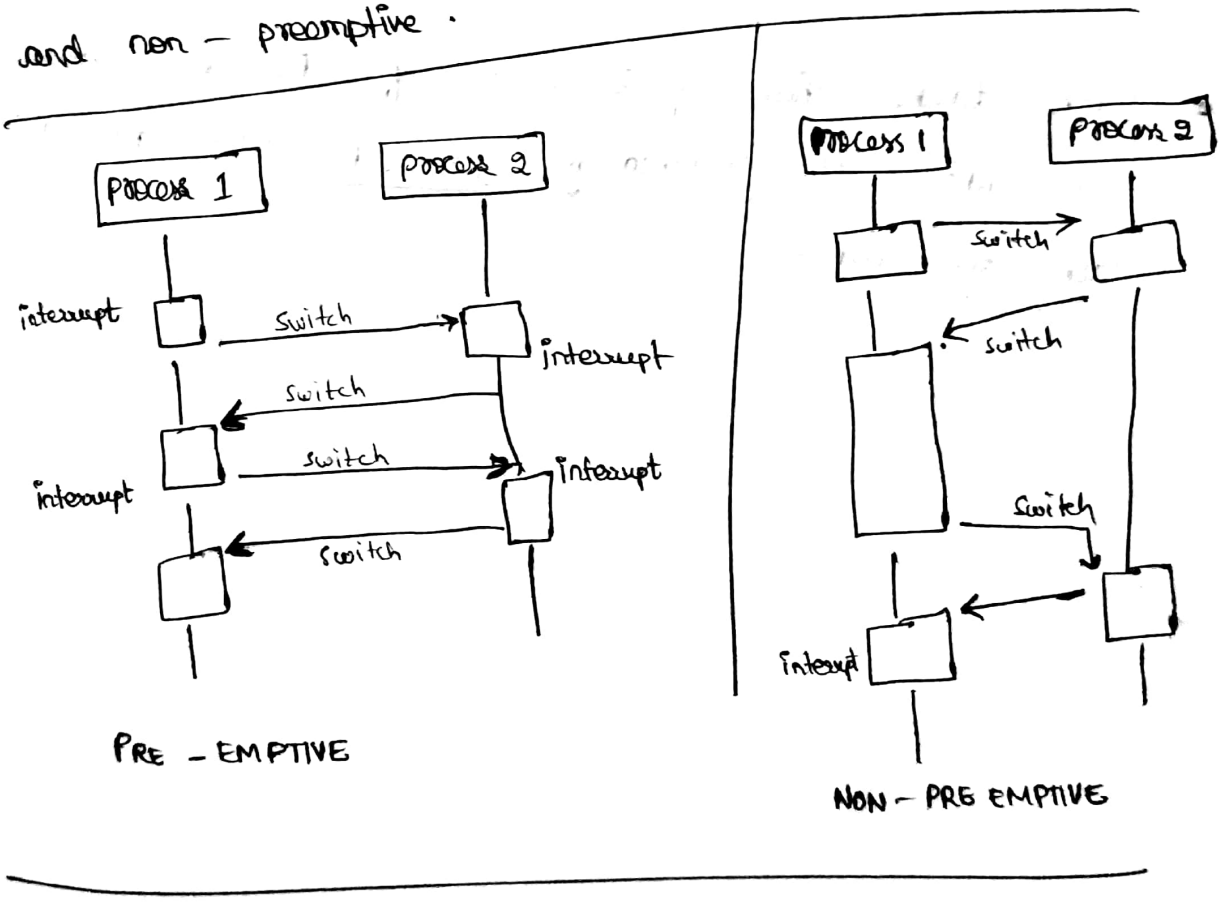
- It provide each task with its own unique address space that makes it almost impossible for one task to affect memory that belongs to another task.

- In most of the cases,

→ Each program executes as a single task or process within the memory address space allocated to the task.

→ However, a single program can also be split into several tasks. This technique is usually called multi threading, and the program's tasks are called threads.

- The two approaches to multitasking are preemptive and non-preemptive.



*. In pre-emptive multitasking,

- The operating system decides how long each task gets to execute before it should step aside so that another task can execute.
- When task's time is up, the O.S task manager interrupts the task & switches to next task in line.
- All the network operating system in wide spread use today use pre-emptive multitasking.

*. In non-pre emptive multitasking,

- Each task that gets control of CPU is allowed to run until it voluntarily gives up control to another task to run.

CPU SCHEDULING POLICIES:

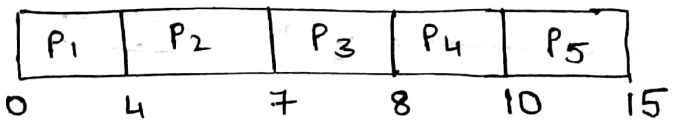
1. First Come first serve CPU scheduling Policy.
2. Shortest Remaining time first CPU scheduling Policy.
3. Round Robin CPU scheduling Policy.
4. Longest Time first CPU Scheduling Policy.
5. Longest Remaining time first CPU Scheduling Policy.
6. Highest Response Ratio next CPU Scheduling Policy.
7. Short Jobs First CPU Scheduling Policy.
8. Priority Scheduling. (preemptive, non preemptive)
9. Multilevel Queue Scheduling.
10. Multilevel Feedback Queue. Scheduling.

---

① First Come First Serve (FCFS):

Process No	Arrival time	Burst Time
1	0	4
2	1	3
3	2	1
4	3	2
5	4	5

Gantt Chart



- As the name suggests, which processes come first, it will get the CPU alloted.

P.No.	AT	BT	CT	TAT	WT
1	0	4	4	4	0
2	1	3	7	6	3
3	2	1	8	6	5
4	3	2	10	7	5
5	4	5	15	11	6

AT – Arrival Time : At what time processer arrived.

BT – Burst Time : Time taken for execution.

CT – Completion Time : Total time taken to complete execution

TAT – Turn Around Time : Time difference between Arrival & Completion

WT – Waiting Time : Time taken for processer to get CPU.

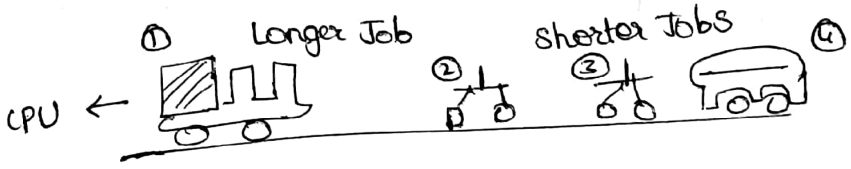
- Arrival Time – User Input
- Burst Time – User Input
- ◉ Completion Time = Burst Time of current + Completion Time of previous process
- ◉ Turn Around Time = Completion Time – Arrival Time = WT + BT
- ◉ Waiting Time = Turn Around Time – Burst Time

- First come First serve CPU Scheduling Algorithm that schedules according to arrival times of processer.

IMP
- FCFS is a non-preemptive scheduling algorithm.
- FCFS suffers from Convoy effect.

- When the arrival times are same for any processes then we go with processor ID. (which Processor number)

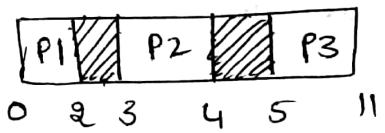
- Convoy effect : The whole operating system slows down due to arrival of processor with larger burst time followed by processor with least burst time.



- Hence in Convoy effect, one slow process downs the performance of the entire set of process, and leads to wastage of CPU time & other devices.

*. CONTEXT SWITCHING :

P#	AT	BT	CT	TAT	WT
1	0	2	2	2	0
2	3	1	4	1	0
3	5	6	11	6	0



: Gantt Chart

The CPU remains ideal for $2 \to 3$ & $4 \to 5$ out of 11 time units

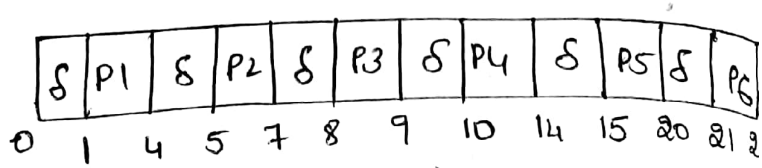
$$\text{CPU ideal} = \frac{2}{11}$$

$$\text{CPU utilization} = 1 - \text{CPU ideal} = 1 - \frac{2}{11} = \frac{9}{11} \approx 81.8\%$$

In this consider a scenario, Switching time = $\delta = 1$ unit

P#	AT	BT
1	0	3
2	1	2
3	2	1
4	3	4
5	4	5
6	5	2

Gantt chart :



- A Context switching is the process of storing the state of a process or thread, so that it can be restored & resumed execution at later point.
- This allows multiple processes to share a single CPU and is an essential feature of Multitasking O.S.

Here, in this case :

$$\eta = \left(1 - \frac{6}{23}\right) \times 100 = 73.91\ \%$$ of utilization

6 are time units when ~~CPU is busy~~ Context switching is happening and CPU is ideal.

---

② SHORTEST JOB FIRST : (SJF)

- SJF is a non-preemptive scheduling algorithm.
- It is a scheduling policy that selects the waiting process with the smallest execution time to execute next

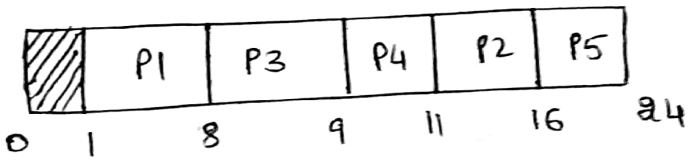
BT less

- SJF has the advantage of having a minimum Average waiting time among all (overcomes the convoy effect in FCFS.).
- This is a Greedy Algorithm.
- It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of ageing.

Ex:

P#	AT	BT	CT	TAT	WT
1	1	7	8	7	0
2	2	5	16	14	9
3	3	1	9	6	5
4	4	2	11	7	5
5	5	8	24	19	11

Gantt chart :



- In this, which processor has least Burst time, it will get the CPU allocated. (when there are 2 or more processes).

③ SHORTEST REMAINING TIME FIRST : (SRTF)

- This algorithm is the preemptive version of SJF scheduling.
- The execution of the process can be stopped after certain amount of time.

- At the arrival of every process, the short term scheduler schedules the process with the least remaining burst time among the list of available process and the running process.

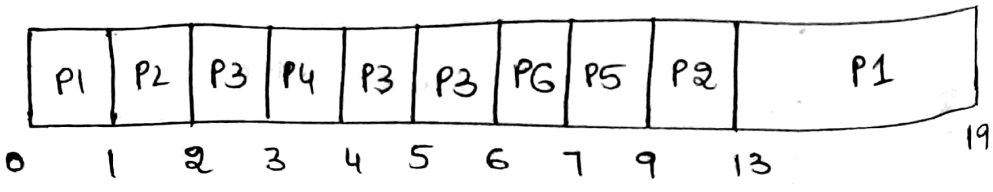
- Once all the process are available in the ready queue, No preemption will be done and the algorithm will work as SJF scheduling.

- The context of this process is saved in Process Control Block when the process is removed from the execution and the next process is scheduled. This PCB is accessed on the next execution of this process.

Ex:

P#	AT	BT	remaining BT	CT	TAT	WT
1	0	~~7~~ 6		19	19	12
2	1	~~5~~ 4		13	12	7
3	2	~~3~~ ~~2~~ 1		6	4	1
4	3	~~1~~ 0		4	1	0
5	4	2		9	5	3
6	5	1		7	2	1

Gantt chart :



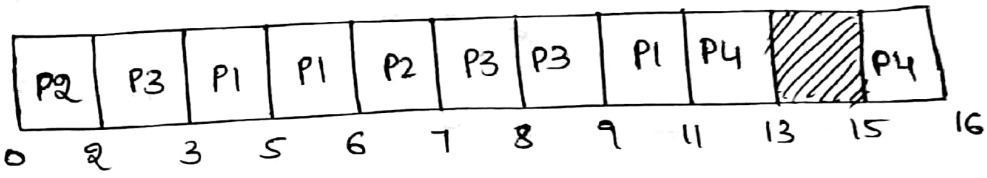
When the arrival time of other process comes in, this current process steps & compares with burst time of arrived process → less — switches to that process.
→ more — continues execution

*. SHORTEST REMAINING TIME FIRST w.r.t I/O HEADER:

P#	AT	BT_1	I/O	BT_2	CT	TAT	WT
1	0	3	2	2	11	11	6
2	0	2	4	1	7	7	4
3	2	1	3	2	9	7	4
4	5	2	2	1	16	11	8

P#	AT	BT_1	I/O	BT_2	No CPU
1	0	~~3~~ 1 0	2	2	(6-8)
2	0	~~2~~ 0	4	~~2~~ 1 0	(2-6)
3	2	~~1~~ 0	3	~~2~~ 1	(3-6)
4	5	2	2	1	

Gantt Chart:



- When I/O is into action, the CPU is not needed for that. Meanwhile CPU is taken by processor with least burst time.

④ ROUND ROBIN CPU SCHEDULING: (RR)

- Each ready task runs turn by turn only in a cyclic queue for a limited time quantum.
- Round robin is a pre-emptive scheduling algorithm.
- If time quantum ↑, Avg. TAT varies irregularly.

- The CPU is shifted to the next process after a fixed interval of time called Time Quantum / slice.
- once a process is executed for a given time period, it is preempted and other process executes for a time period.
- Context switching is used to save states of preempted process.

Ex: (Time Quantum = 4) assumption / user input

P#	AT	BT
1	0	~~4~~ 0
2	1	~~5~~ ~~1~~ 0
3	2	~~2~~ 0
4	3	~~1~~ 0
5	4	~~6~~ ~~2~~ 0
6	6	~~3~~ 0

0 ⇒ P1

4 ⇒ ~~P1~~ P2 P3 P4 P5

8 ⇒ ~~P2~~ P3 P4 P5 P6 P2

10 ⇒ ~~P3~~ P4 P5 P6 P2

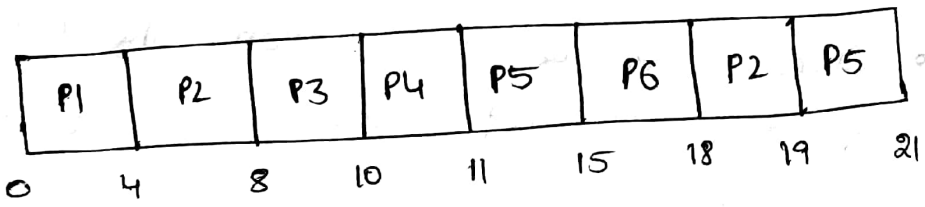
11 ⇒ ~~P4~~ P5 P6 P2

15 ⇒ ~~P5~~ P6 P2 P5

18 ⇒ ~~P6~~ P2 P5

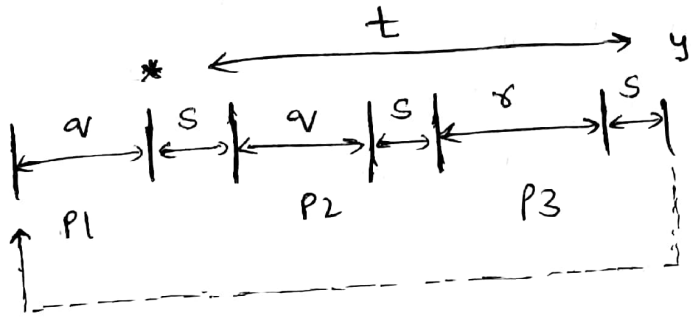
19 ⇒ ~~P2~~ P5

Gantt chart :



- Simply as FCFS with time quantum, If the burst Time,
  - BT > Quantum → append it to end of queue
  - BT ≤ Quantum → Terminates & voluntarilies release CPU
  - the circular queue repeats the process until all process are terminated.

GATE Consider 'n' process sharing the CPU in R.R. If the context switching time is 's' units. What must be the time quantum 'q' such that each process is guaranteed to get the turn at the CPU for every 't' sec.



- There is so much of context switching in Round Robin.
- R.R $\approx$ FCFS $\Rightarrow$ max (BT) $\leq$ Time Quantum
- If Time Quantum is very less, Gantt is large.
- RR is implemented using Circular Queue.
- RR is guaranteed to provide best Response time than FCFS, SJF, SRTF .... (because of equi time distribution)

$2q + 3s = t$

$q = \frac{t - 3s}{2}$

Each process is getting the turn at CPU for every 't' sec

$\therefore$ between n context switching happens

and (n-1) processes are there

(n-1) processes take n context switching.

$\Downarrow$

(n-1) time quantum.

$\Rightarrow (n-1) \cdot q + n \cdot s = t$

$* \quad \boxed{q = \frac{t - ns}{n-1}}$

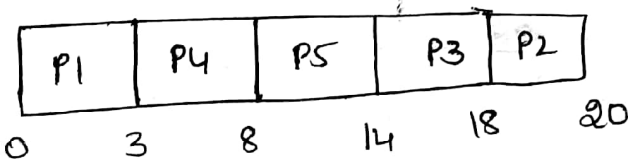
⑤ LONGEST JOB FIRST : (LJF)

- It is a non-preemptive scheduling algorithm.
- This algorithm is based on the burst time of process.
- It is a scheduling policy that selects the waiting process with the longest execution time to execute next.
  BT high
- If two process have the same burst time then the tie is broken using FCFS i.e. the process that arrived first is processed first.

Ex:

P#	AT	BT	CT	TAT	WT	RT
1	0	3	3	3	0	0
2	1	2	20	19	17	17
3	2	4	18	16	12	12
4	3	5	8	5	0	0
5	4	6	14	10	4	4

Gantt chart :



IMP
- In case of Non-preemptive, **WT = Response Time**

---

⑥ LONGEST REMAINING TIME FIRST : (LRTF)

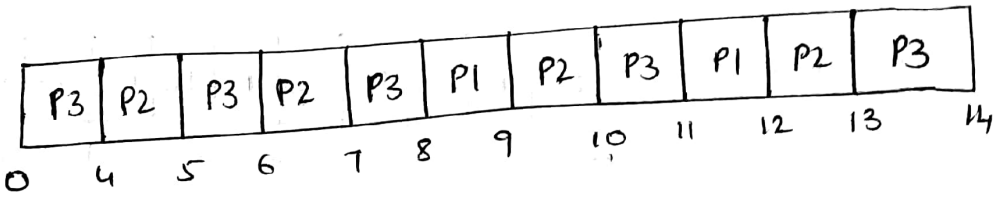
- This algorithm is preemptive version of Longest Job First
- In this algorithm, we find the process with the

maximum remaining time and then process it. We check for the maximum remaining time after some interval of time (say 1 unit each) to check if another process having more Burst Time arrived up to that time.

Ex:

P#	AT	BT	CT	TAT	WT
1	0	2	12	12	10
2	0	4	13	13	9
3	0	8	14	14	6

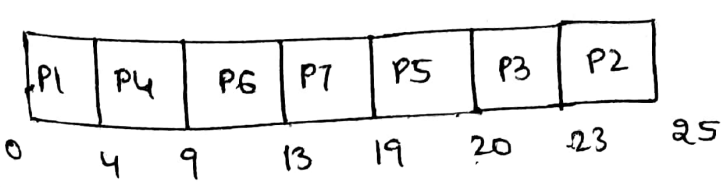
Gantt chart :



(7) PRIORITY BASED SCHEDULING : (non-preemptive)

Ex:

P#	priority	AT	BT	CT	TAT	WT	RT
1	2 (L)	0	4	4	4	0	0
2	4	1	2	25	24	22	22
3	6	2	3	23	21	18	18
4	10	3	5	9	6	1	1
5	8	4	1	20	16	15	15
6	12 (H)	5	4	13	8	4	4
7	9	6	6	19	13	7	7



: Gantt chart - [Priority + FCFS]

PRIORITY BASED SCHEDULING (mode : Pre emptive)

Ex :

P#	priority	AT	BT
1	2	0	~~4~~ 3
2	4	1	~~2~~ 1
3	6	2	~~3~~ 2
4	10	3	~~5~~ ~~4~~ 3
5	8	4	1
6	12	5	~~4~~ 3
7	9	6	6

[Priority + BT + P#]

Gantt chart :

P1	P2	P3	P4	P4	P6	P6	P4	P7	P5	P3	P2	P1

0 1 2 3 4 5 6 9 12 18 19 21 22 25

P#	Priority	AT	BT	CT	TAT	WT	RT
1	2	0	4	25	25	21	0
2	4	1	2	22	21	19	0
3	6	2	3	21	19	16	0
4	10	3	5	12	9	4	0
5	8	4	1	19	15	14	14
6	12	5	4	9	4	0	0
7	9	6	6	18	12	6	6

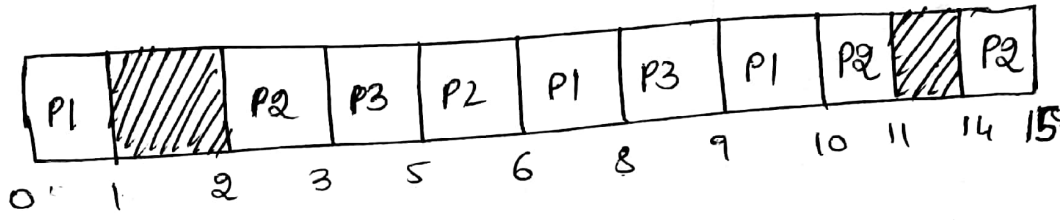
- Just compare the priorities, and arrange burst times in ascending order according to FCFS.

IMP *

PRE-EMPTIVE PRIORITY WITH I/O OVERHEAD :

P#	AT	$(CPU)_1$	I/O	$(CPU)_2$	No CPU
P1	0	~~1~~ 0	5	~~3~~ ~~2~~ 1	(1-6)
P2	2	~~3~~ ~~2~~ ~~1~~ 0	3	1	
P3	3	~~2~~ 0	3	~~1~~ 0	(5-8)

Gantt chart :



Priority	P#	AT	$(CPU)_1$	I/O	$(CPU)_2$	CT	TAT	WT	RT
(M)	1	0	1	5	3	10	10	6	
Low	2	2	3	3	1	15	13	9	
High	3	3	2	3	1	9	6	3	

C.P.U. Utilization $= \frac{11}{15} * 100 = 73.33\%$

NOTE

- SJF is the best among all CPU scheduling that produces maximum efficiency.
- To avoid starvation, implement FCFS.
- Implement R.R using circular queue data structure (best).

Example : GATE

Consider we have four processes P1 P2 P3 P4 arriving at time 0. Their respective burst times are a, b, c, d, with condition $a < b < c < d$.

P#	AT	BT
1	0	a
2	0	b
3	0	c
4	0	d

what is the average TAT when SJF CPU scheduling algorithm is used?

Answer:

P#	AT	BT	CT	TAT	WT
1	0	a	a	a	0
2	0	b	a+b	a+b	a
3	0	c	a+b+c	a+b+c	a+b
4	0	d	a+b+c+d	a+b+c+d	a+b+c

$$\text{Avg TAT} = \frac{a + (a+b) + (a+b+c) + (a+b+c+d)}{4} = \frac{4a+3b+2c+d}{4}$$

---

⑧ HIGHEST RESPONSE RATIO NEXT (HRRN):

- It is a non-preemptive algorithm in which, the scheduling is done on the basis of an extra parameter called Response ratio.

- A Response ratio is calculated for each of the available jobs and the job with the highest response ratio is given priority over others.

Response Ratio $= \frac{W+S}{S}$

W – Waiting time (current)

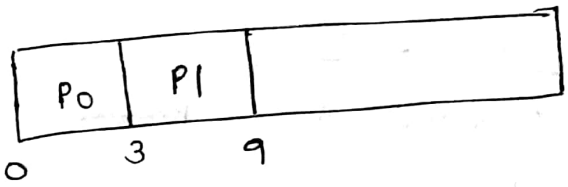
S – Burst time or Service time

- We notice that HRRN $\propto$ WT, HRRN $\propto \frac{1}{BT}$.

Ex:

P#	AT	BT
P0	0	3
P1	2	6
P2	4	4
P3	6	5
P4	8	2

Gantt chart:



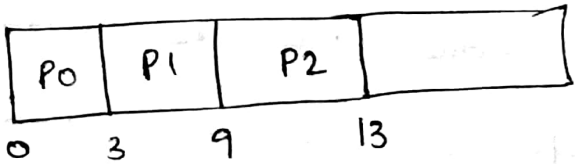
P_2: $RR_2 = \frac{(9-4)+4}{4} = 2.25$

P_3: $RR_3 = \frac{(9-6)+5}{5} = 1.6$

P_4: $RR_4 = \frac{(9-8)+2}{2} = 1.5$

So, we have schedule P_2 with highest RR = 2.25 $\Leftarrow$

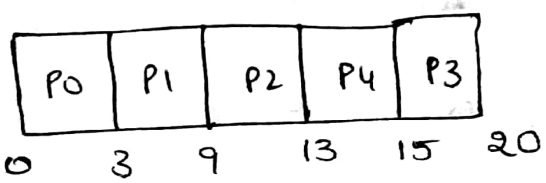
So, the Gantt chart:



P_3: $RR_3 = \frac{(13-6)+5}{5} = 2.6$

P_4: $RR_4 = \frac{(13-8)+2}{2} = 3.5$ ✓ (highest RR)

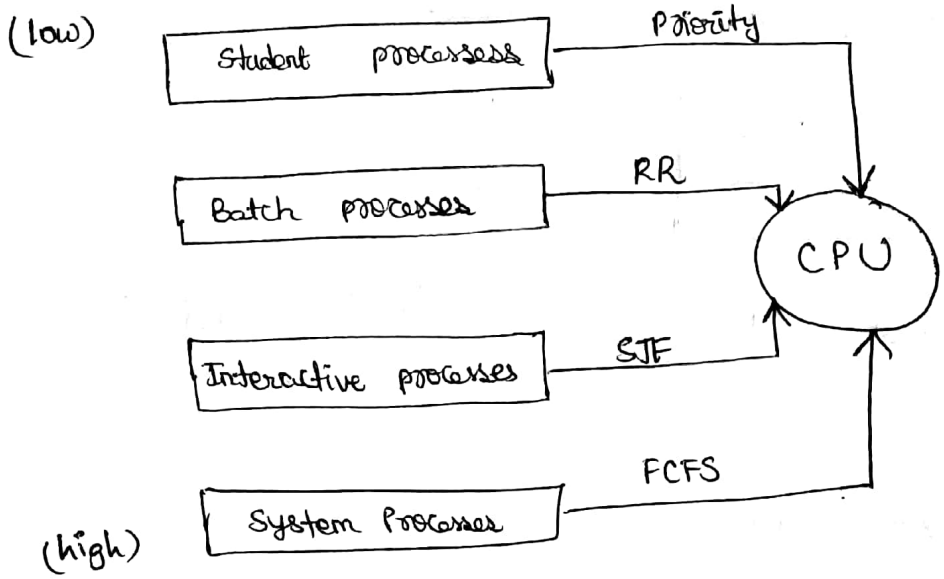
∴ Gantt chart ⇒



In HRRN, If WT is same ⇒ SJF scheduling

If BT is same ⇒ FCFS scheduling

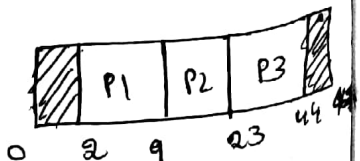
⑨ MULTILEVEL QUEUE SCHEDULING :



Problem : GATE

Consider three processes, all arriving at time zero, with total execution time of 10, 20, and 30 units respectively. Each process spends the first 20% of execution time doing I/O, the next 70% of time doing computation and the last 10% of the time doing I/O again. The O.S uses a SRTF policy and schedules a new process either when the running process gets blocked on I/O or when the running process terminates. Assume that all I/O operations can be overlapped. For what percentage of time CPU remains idle?

P#	AT	I/O	(BT) CPU	I/O
P1	0	2	7	1
P2	0	4	14	2
P3	0	6	21	3



$$\text{Idle time} = \left(\frac{2}{44} \times 100\right)$$

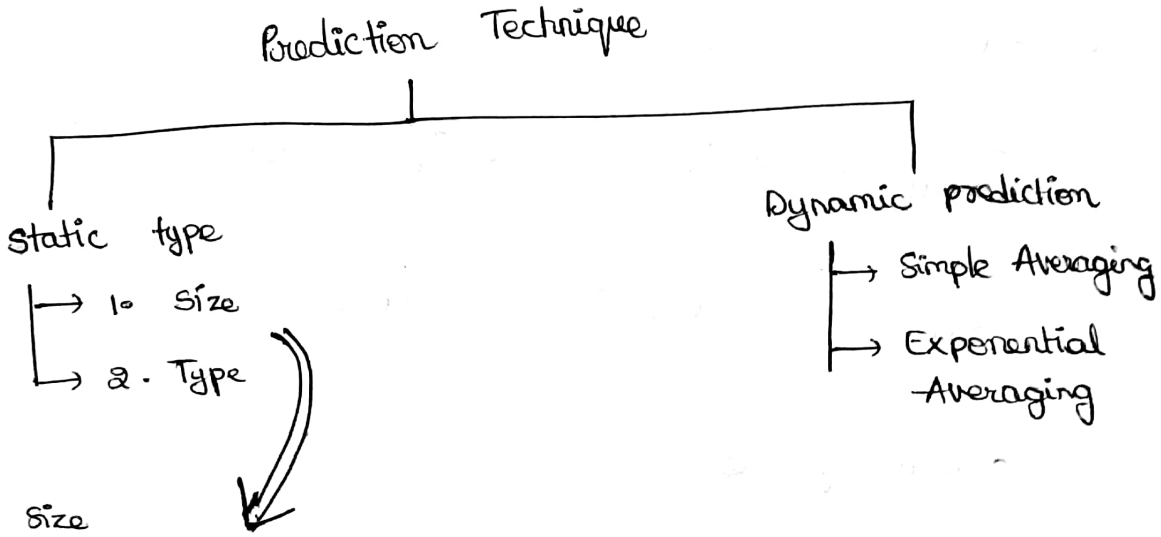
*. SHORTEST JOB FIRST with PREDICTED BURST TIME:

Advantages of SJF:

- Maximum throughput is provided. (the jobs completed per unit time)
- Avg WT, Avg TAT is minimum.

Disadvantages of SJF:

- Starvation to longer jobs.
- Not implementable because BT of a process cannot be known ahead.



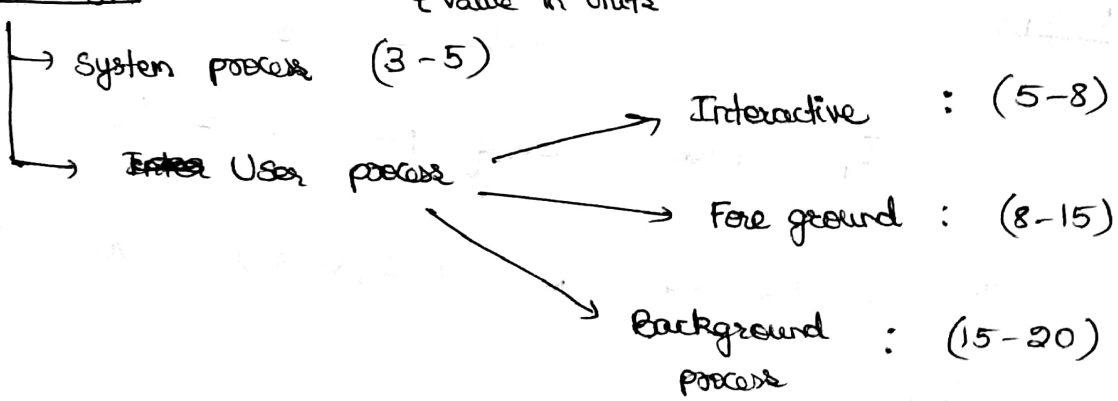
1. Process Size

P_{old} = 200 KB = 20 units

P_{new} = 201 KB ≈ P_{old} ≈ 20 units ✓ predict BT of newly arrived process

2. Process Type

t value in units



- Dynamic Prediction Technique :

1. Simple Averaging :

Given n process : $(P_1, P_2, \ldots P_n)$

t_i : actual BT of process i

τ_i : predicted BT of process i

$$\tau_{n+1} = \frac{1}{n} \cdot \sum_{i=1}^{n} t_i$$

(Simple Calculation of Average / mean)

2. Exponential Averaging :

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n \quad -(1)$$

$(0 < \alpha < 1)$

learning parameter

$$\tau_n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1} \quad -(2)$$

substituting (2) in (1),

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + (1-\alpha)^2 \tau_{n-1}$$

$$= \alpha t_n + (1-\alpha)\alpha t_{n-1} + (1-\alpha)^2 \alpha t_{n-2} + (1-\alpha)^3 \tau_{n-2}$$

Example :

α : Learning parameter = 0.5

τ_1 : predicted BT of process 1 = 10

$(t_1\ t_2\ t_3\ t_4) = (4, 8, 6, 7)$ Assumption

then $\tau_5 = ?$

Predicted BT of next process = α · Exact BT of previous process + $(1-\alpha)$ · Predicted BT of previous process

$$\boxed{\tau_{n+1} = \alpha\, t_n + (1-\alpha)\cdot \tau_n} \quad *$$

$\tau_1 = 10$, $\alpha = 0.5$, $t_1 = 4$

Put $n=1 \Rightarrow \tau_2 = (0.5)\,4 + (1-0.5) \times 10 = 7$

Put $n=2 \Rightarrow \tau_3 = (0.5)\,8 + (1-0.5) \times 7 = 7.5$

Put $n=3 \Rightarrow \tau_4 = (0.5)\,6 + (1-0.5) \times 7.5 = 6.75$

Put $n=4 \Rightarrow \tau_5 = (0.5)\,7 + (1-0.5) \times 6.75 = 6.875$

$\therefore \tau_5$: predicted BT of process 5 = 6.875

*. SYSTEM CALL :

- A system call is a way for programs to interact with the O.S.

imp
- A computer program makes a system call when it makes a request to the O.S kernel

- System calls are the only entry points into kernel.

```
# include <unistd.h>
int main()
{
    write (1, "Hello", 5);
    return 0;
}
```

output : Hello

```
# include <stdio.h>
int main()
{
    printf ("Hello");
    return 0;
}
```

output : Hello

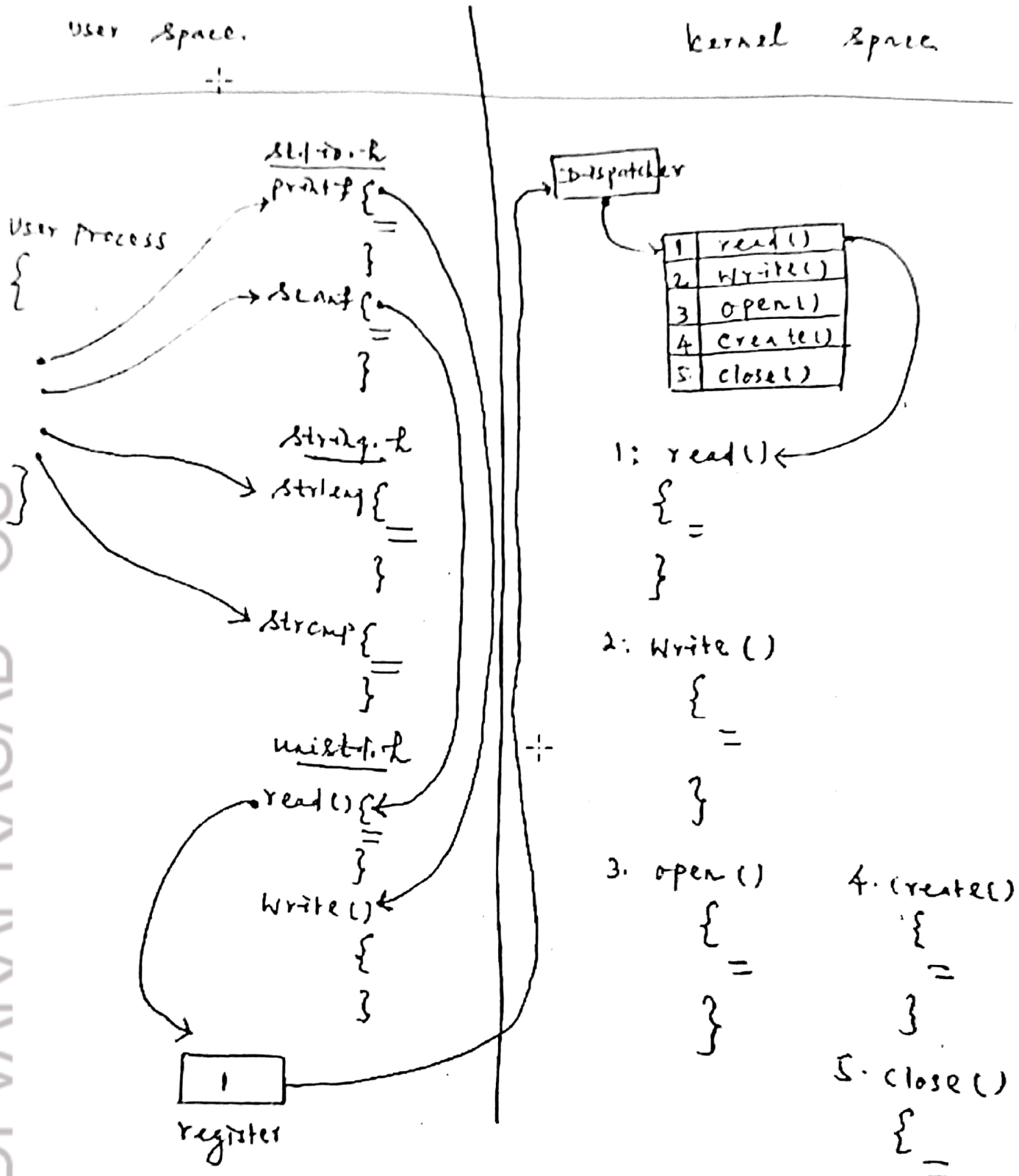
write is a wrapper function present in unix standard whereas printf is a system function present in standard input output header.

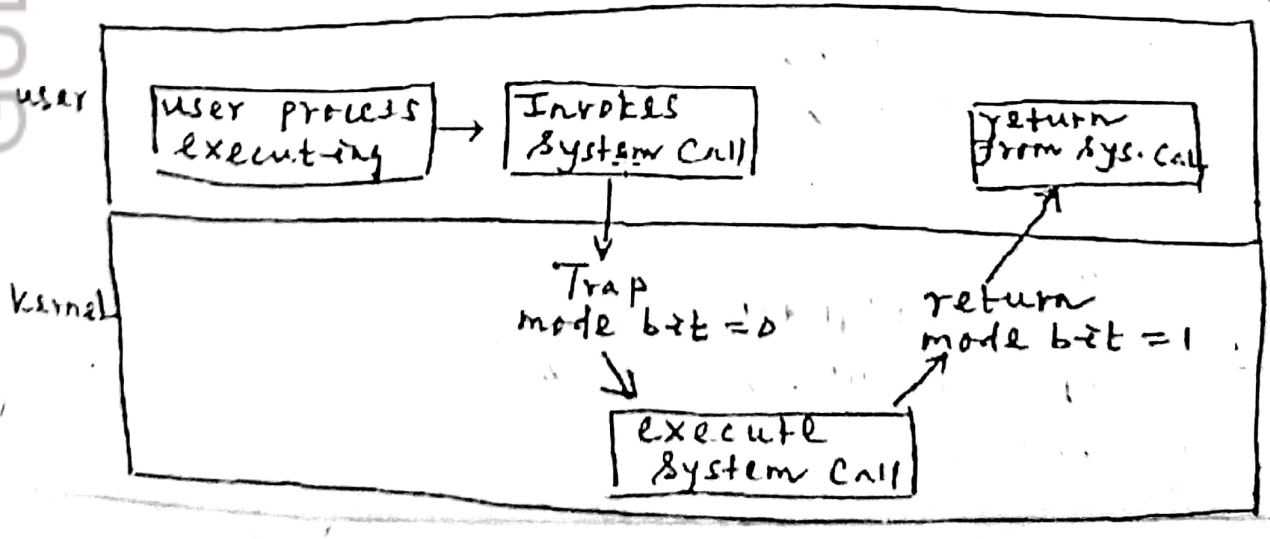
1 : File ID of standard input Monitor

"Hello" : User content to write into Monitor

} write (-,-,-)

5 : every character 1 byte H E L L O (1 2 3 4 5)



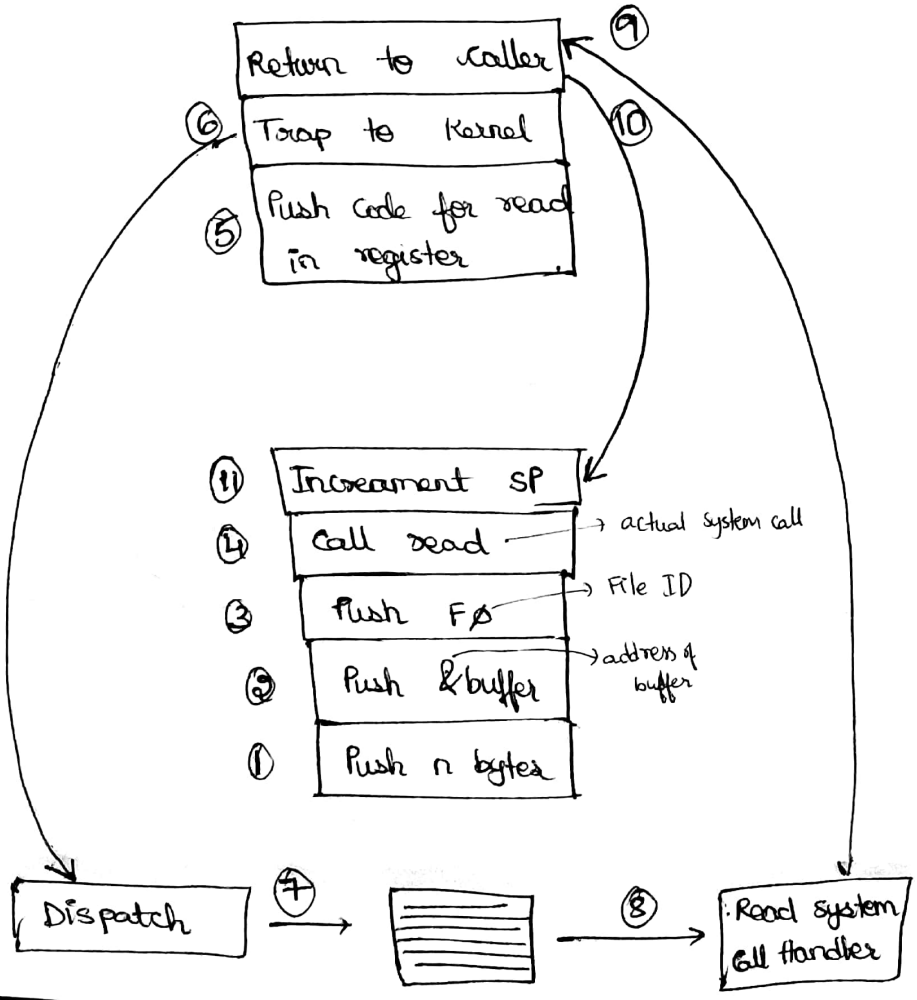


*. Types of System Calls :

1. Process Control : CreateProcess(), ExitProcess(), WaitForSignal()
2. File Manipulation : CreateFile(); ReadFile(), WriteFile(), etc
3. Device Manipulation : SetConsoleMode(), ReadConsole(), WriteConsole, etc.
4. Information Maintenance : setTimer(), Sleep(), GetCurrentProcessID(), ...
5. Communication : CreatePipe(), SetFileSecurity(), ---

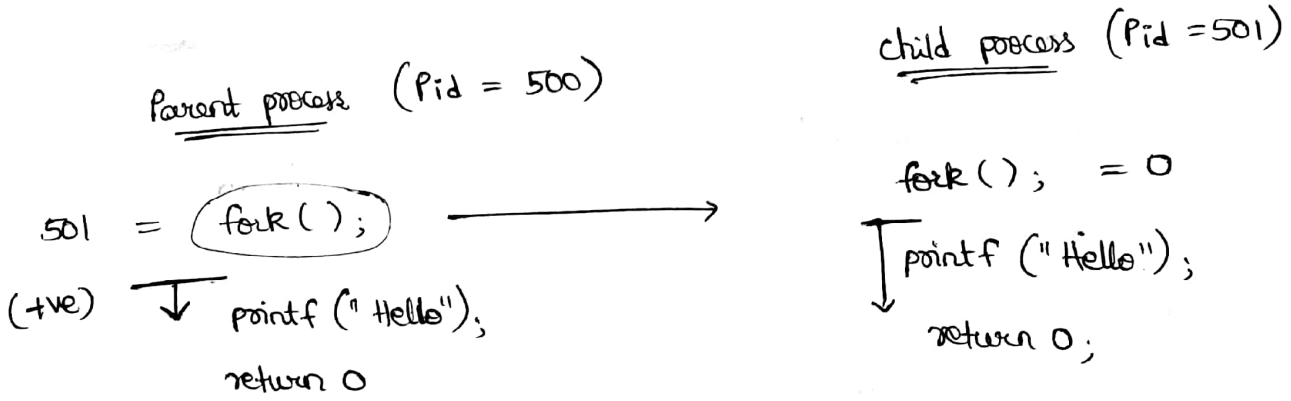
*- Parameter Passing :

1. Parameters can be passed using registers.
2. When there are more parameters than the available registers parameters are stored in block & block address can be passed as a parameter to register.
3. Parameters can be passed using stack.



Ex:

```
# include  < stdio.h >
# include  < unistd.h >
int main()
{
    fork();
    printf ("Hello");
    return 0;
}
```



Here output is : Two times "Hello" is printed.

Ex:

```
# include < stdio.h >
# include < unistd.h >
int main()
{
    if ( fork() == 0 )
        printf ("X");
    else
        printf ("Y");
    return 0;
}
```

output :

Parent process

Pid = 500

- Here fork returns a positive value
- So, if statement is not executed then else is output ⇒ "Y"

child process

Pid = 501

- Here fork of the child process always returns 0
- So, if statement executes & prints "X"

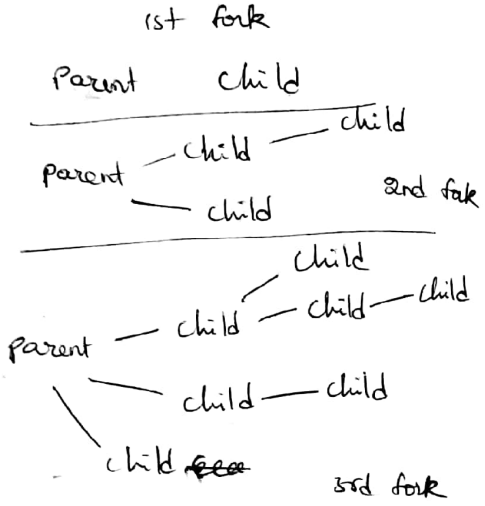
Output is : XY YX

both are possible (depends on scheduler which executes first).

Ex :

```
# include   < sys/types.h >
# include   <unistd.h >
# include   < stdio.h >

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



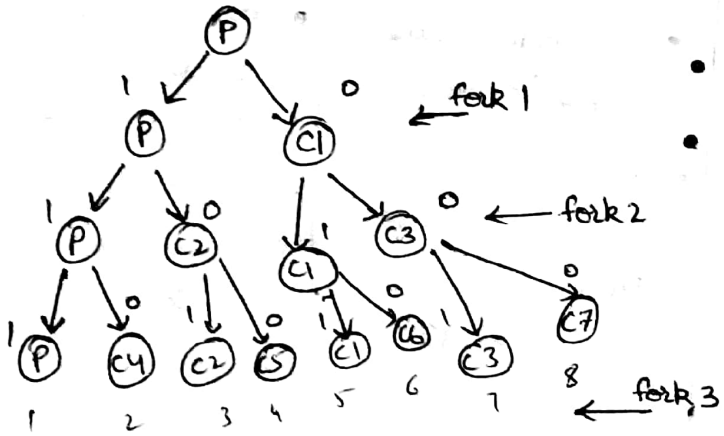
1 fork → 2 processes

2 fork → 4 processors

3 fork calls → 8 process ----

n fork calls = 2^n processors

1 parent
n-1 child ⟹ can be presented as tree structure



- Total no. of processes = 8
- Total no. of child process = 8-1 = 7

↳ "Hello" is after 2 forks
So, 2^2 outputs ⇒ 4 "Hello"

↳ "World" is after 3 forks
So, 2^3 outputs ⇒ 8 "World"

Hello World World
Hello World World
Hello World World
Hello World World

Ex: The no. of child processes created is: 31

```
# include < unistd.h >
int main ()
{
    int i;
    for (i=0; i<10; i++)
        if (i%2 == 0)
            fork();
    return 0;
}
```

At 0, 2, 4, 6, 8 → fork

fork fork fork fork

Total fork = 5

- Total processes = $2^5 = 32$
- Total child processes = $2^5 - 1 = 31$

Ex:

```
# include < unistd.h >
int main ()
{
    printf(" * ");
    for (i=0; i<n; i++)
        fork();
    printf (" * ");
}
```

fork → n → 2^n *

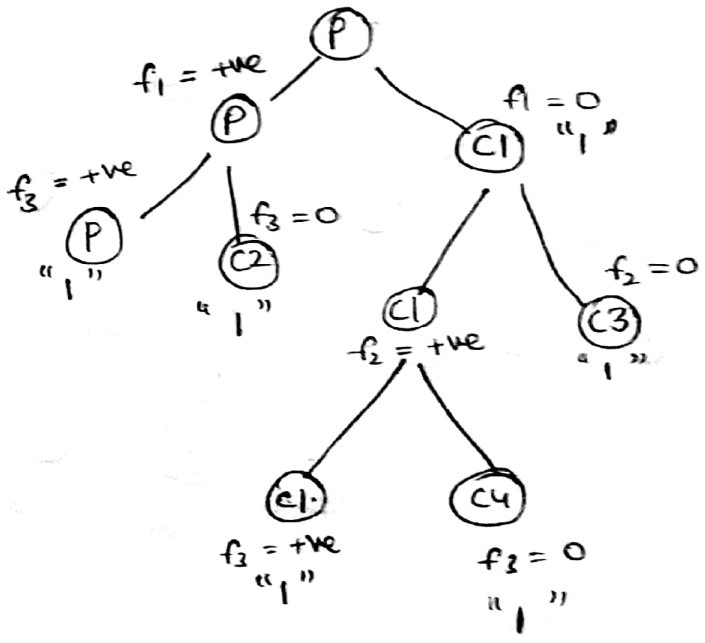
parent → *

2^n * , *

⇒ $(2^n + 1)$ * are printed.

Ex: *IMP what is o/p of the following code fragment:

```
if (①fork() || ②.fork())
{
    ③ fork();
}
printf (" 1 ");
return 0;
```



"1" is printed 5 times

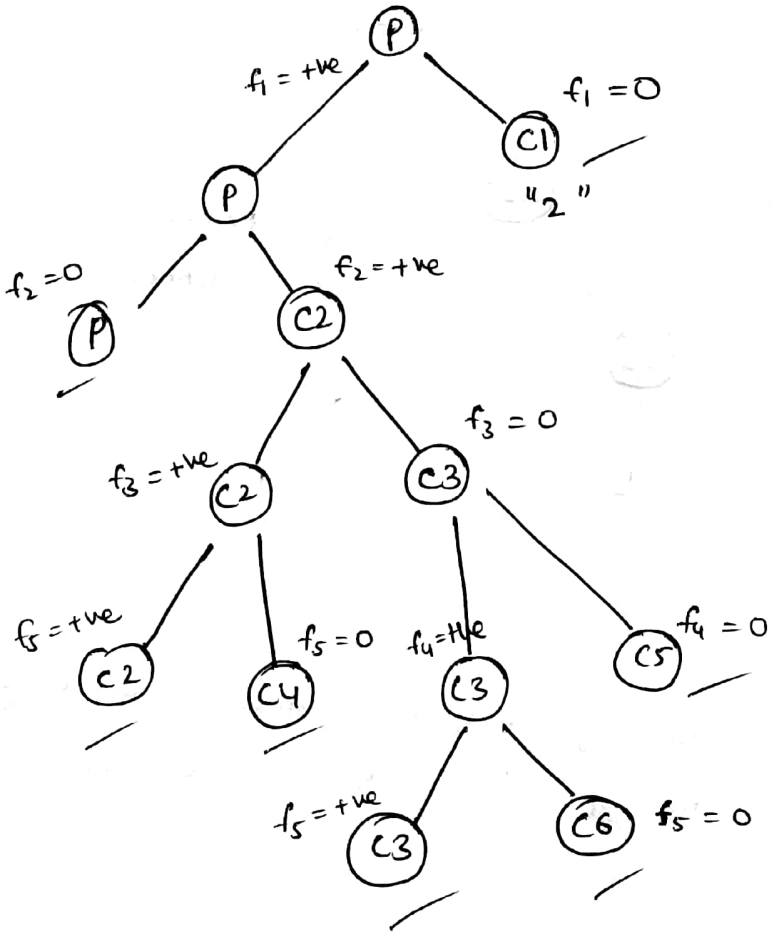
P, C1, C2, C3, C4

Ex :

```
# include   < stdio.h >
# include   < unistd.h >
int main()
{
    if ( fork()  &&  (! fork()))
    {
        if (fork() || fork())
        {
            fork()
    } }
    printf ("a");
    return 0;
}
```

! fork returns 0
&& 0
always 0

Explanation :



2 - P
2 - C1
2 - C2
2 - C3
2 - C4
2 - C5
2 - C6

total process = 7

child process = 6

*. PRECEDENCE GRAPH :

- If more than 2 processors in CPU — parallelism
- If single processor is in CPU — Concurrency
  ↓
  one or many processes
- Time Quantum / Context switching happen in case of Concurrency

*. CONCURRENT PROCESSING :

Ex :

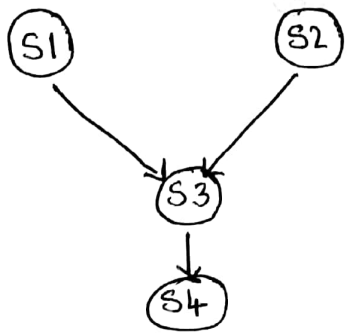
Statement S_1 : $a = x + y$ } used variables are different

Statement S_2 : $b = z + 1$

Statement S_3 : $c = a + b$ } Cannot execute unless S_1 & S_2 are done

Statement S_4 : $d = c - 1$ } Cannot execute unless S_1 S_2 S_3 are done

Level same



PRECEDENCE GRAPH

*. READ SET :

$R(S_i)$: Set of variables which are only refered in S_i, their values are not modified by S_i

*. WRITE SET :

$W(S_i)$: Set of variables which are refered and and modified, updated by S_i.

$W(S_1) = \{a\}$ $\quad W(S_2) = \{b\}$ $\quad W(S_3) = \{c\}$

$R(S_1) = \{x, y\}$ $\quad R(S_2) = \{z\}$ $\quad R(S_3) = \{a, b\}$

$W(S_4) = \{d\}$ $\quad R(S_4) = \{c\}$.

IMP ★ S_i and S_j are being executed concurrently if :

$$R(S_i) \cap W(S_j) = \emptyset$$

$$W(S_i) \cap W(S_j) = \emptyset$$

$$W(S_i) \cap R(S_j) = \emptyset$$

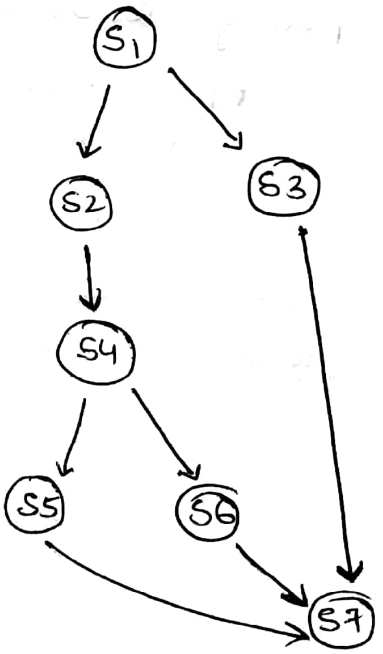
- order of Execution in Read - Read, then there is no effect on final output.

$\therefore$ From this defination ↑,

S_1, S_2 are executed concurrently.

---

*. Ex: Given a precedence Graph, Implement Precedence Graph by using fork & join statements.



P1 (parent)

```
begin          count = 3
S1
L1: fork L1    - By using fork system call
                 P1 executes S2 & S3 concurrently
S2
S4
L2: fork L2    - Parent P1 found that S5 S6
                 can be executed concurrently,
                 So, by using fork system call
                 P1 executes S5 S6 concurrently.
S5
L3: join count
S7
end
```

P2

```
L1: S3
goes to L3
```

P3

```
L2: S6
goes to L3
```

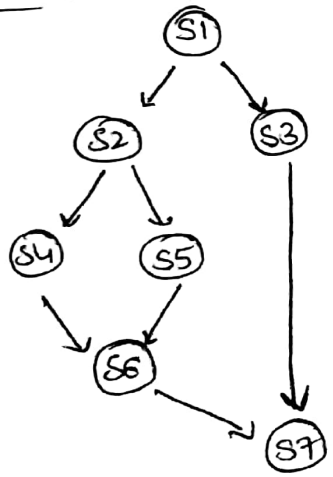
```
join count
{ Count = Count - 1
  if (Count != 0) then
    terminate();
```

Count = 3 because how many are pointing to S7 = 3

If P2 > P3 > P1 are scheduled in order → P1 continues, P2, P3 terminates

---

EX :



P1

```
begin
  count 1 = 2
  Count 2 = 2
  S1
L1: fork L1
  S2
L2: fork L2
  S4
L3: joint count 1
  S6
L4: join Count 2
  S7
end
```

P2

```
L1: S3
  goto L4
```

P3

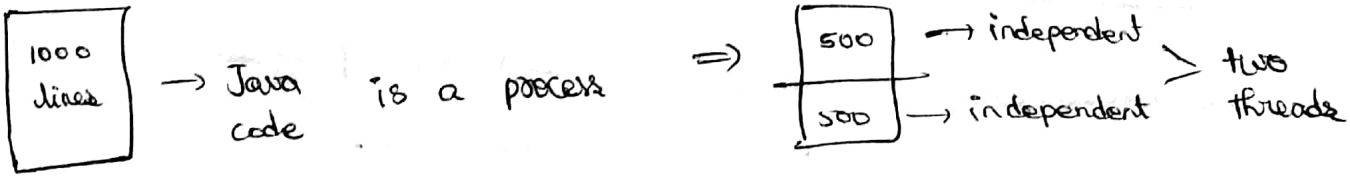
```
L2: S5
  goto L3
```

}

*. MULTI THREADING :

- Multitasking : Executing several tasks simultaneous
- Process based MT : " where each task is an independent process.
  - Ex : Typing a java code in Editor
    Listening to music in same system
    Torrent download
- Thread based MT : " where each task is an independent part of a program. Each independent flow of execution is thread.

Ex :



- Process : Heavy weight
- Thread : Light weight

*. Applications of Multithreading :

① MultiMedia ② Animation ③ Web Server

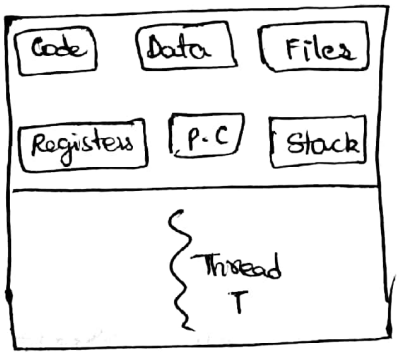
④ Video Games

- Thread : Basic unit of CPU utilization
  Independent flow of execution
  light weight process.

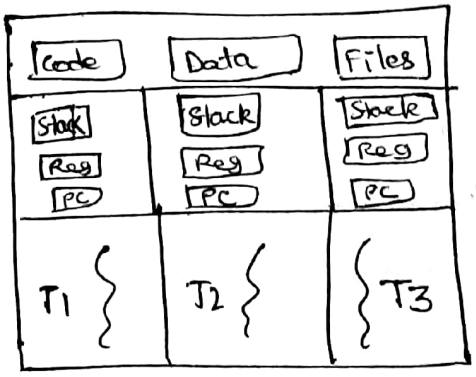
- It compares → A thread id — unique id assigned when created
  → A PC — next instruction address
  → A register set — store the outcomes of instruction
  → A Stack — For parameter passing

IMP
- It shares with other threads belonging to the same process its code, data & files.

- By default, a process is single threaded.



Single threaded process



Multi threaded process

*. Advantages :

1. Responsiveness : Allows to run a program even if a part of its is blocked.

2. Resource sharing : Different threads of activity in the same address space.

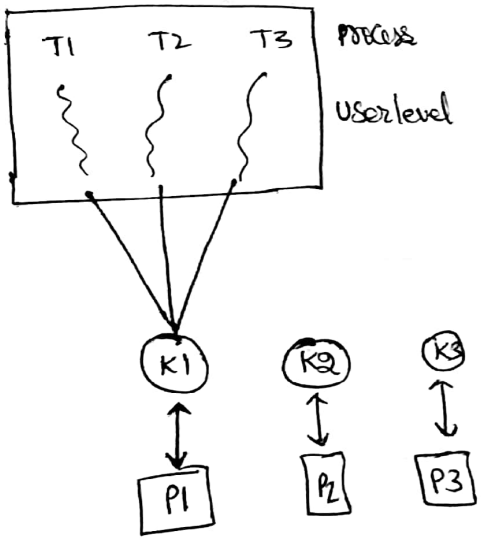
3. Economy : Allocating memory and resources for process creating in the same address space.

4. Utilization of Multiprocessing CPU.

*. MULTITHREADING MODELS :

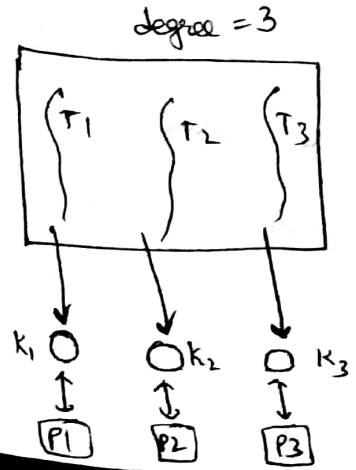
1. User level thread – creation of thread without knowledge of OS
2. Kernel level thread – created & managed by OS directly.

◉ MANY TO ONE MODEL :



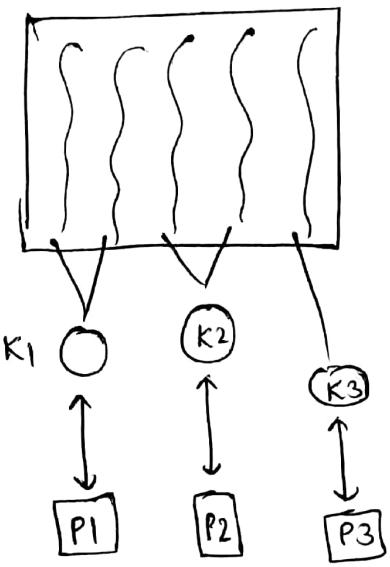
- Maps many user level threads to one kernel level thread.
- Thread management is done in user space.
- Entire process is blocked if a thread makes a blocking syscall.
- Multiple threads are unable to run in parallel on multi processors.

◉ ONE TO ONE MODEL :



- Burden on CPU performance.
- Maps each user thread to a kernal thread.
- provides more concurrency.
- Allows multiple threads to run in parallel.
- Creating an user thread results in the creation of kernal thread.

- MANY TO MANY MODEL :



- Maps many user level threads to a smaller or equal no. of Kernal threads.
- User can create as many user threads as necessary.
- Multiprocessor architecture can be explored.

- Thread of the process share global variables & heap.

*. THREADING ISSUES :

- The semantics of fork() and exec() changes in a multithreaded process.

Q) If one thread in a process calls fork() does the new process duplicates all threads ?

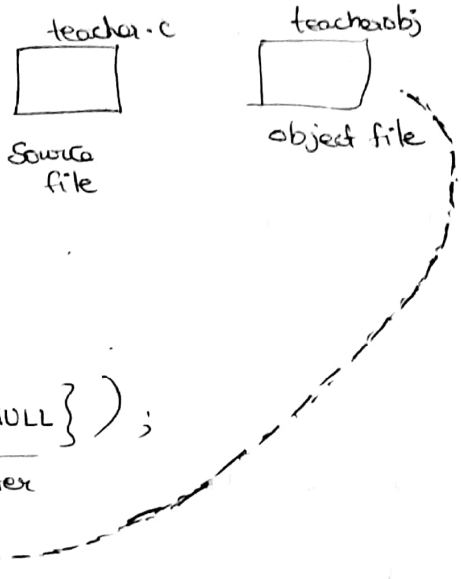
- Two versions of fork() :
  i. One that duplicates all threads.
  ii. Another that duplicates only the thread that invokes fork().
- If a thread invokes exec() system call, the process specified in the parameter to execute exec() will replace the entire process including all threads.

Ex: fork() (vs) exec()

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    printf("Teachers Day");
    return 0;
}
```

teacher.c

$ gcc teacher.c -o teacherobj.exe




```
#include <stdio.h>
#include <unistd.h>
int main()
{
    execv("teacher", {"vit", "AP", NULL});
                      1st parameter   2nd parameter
    printf("Students Day");
    return 0;
}
```

---

*. PROCESS SYNCHRONIZATION :

- Access to the shared resource must be synchronized.
- Critical section - It is a code segment that accessed shared variables and has to be executed in an atomic action. In a group of co-operating processes, at a given point of time, only one process must be executing the critical section.

process P_1

1. a = a % 10 ;
2. b = b - 1 ;
3. c = c * 10 ;
4. d = d/10 ;

atomic action

5. e = e * 2 ;
6. f = f/10 + 1 ;

process P_2

11. w = w * 2 ; } Remainder section
12. x = x + 1 ;

13. c = c * 2 ; } Critical section access to shared variables
14. d = d/2 + 1 ;

15. y = y * 10 + 1 ; } Remainder section access to independent variables
16. z = z/2 - 1 ;

Case-i :

Primary Memory ? Local Buffer

P_1 LB_1 - [10] 11

P_2 LB_2 [11] 12

Read - $R_1(a)$
$a = a+1$; } critical
Write - $W_1(a)$

$R_2(a)$
$a = a+1$;
$W_2(a)$

- Assume P_1 is followed by P_2.
- There is a consistency because atomicity property is followed.
- Atomicity property is guaranteed in serial schedule.

Case-ii:

Secondary Memory

Running parallelly
Scheduled parallel.

a [10] 11 11

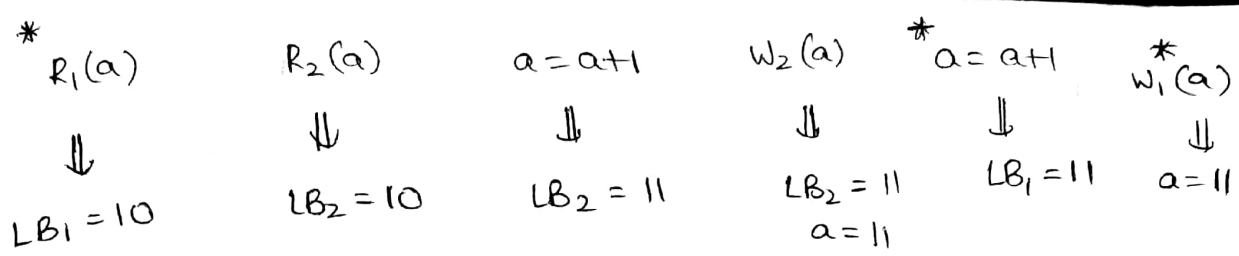
P_1 LB_1 [10] 11

$R_1(a)$
→ Content switching
$a = a+1$
$W_1(a)$

P_2 LB_2 [10] 11

$R_2(a)$
$a = a+1$
$W_2(a)$

Content switching ←



- Here atomicity is property that is not followed.
- At any instant of time, only 1 process should be there in the critical section (to follow atomicity property).

*. Need of Synchronization :

- When multiple process execute concurrently sharing some system resources.
- To avoid inconsistent result (case ii).

*. Race Condition

- The final o/p produced depends on the execution order of instructions of different processes.
- Several process compete with each other -

P_1

```
{
1.  C = B - 1 ;
2.  B = 2 * C ;
}
```

P_2

```
{
11. D = 2 * B ;
12. B = D - 1 ;
}
```

*. Solution to critical section problem using "turn" variable.

turn [0] ≠ 0

- Mutual exclusion is guaranteed.
- But there is no progress.

P_0

```
while (1)
{
1   while (turn != 0);
2   Critical section
3   turn = 1
4   remainder section
}
```

P_1

```
while (1)
{
11  while (turn != 1);
12  Critical section
13  turn = 0;
14  remainder section;
}
```

→ Initial value of turn is 0.

→ P_0 gets the first chance to enter the CSection.

→ while exiting the Critical section, P_0 makes turn = 1.

→ Then P_1 gets a chance to enter the C-section.

→ while exiting the Critical section, P_1 makes turn = 0

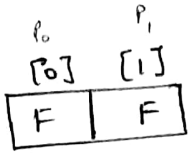
(II) Critical Section Problem using Flag variables :

P_0

```
while (1)
{
1.  flag[0] = True ;
2.  while (flag[1]);
3.  critical section
4.  flag[0] = False ;
5.  remainder section
}
```

P_1

```
while (1)
{
11. flag[1] = True ;
12. while (flag[0]);
13. critical section
14. flag[1] = False ;
15. remainder section
}
```



Array for process P_0 P_1

- Mutual exclusion property is guaranteed.
- There is Deadlock, So there is no progress, there is no bounded waiting.
- Deadlock - order of execution : 1 11 2 12 (neither P_0, nor P_1 are allowed to enter critical section).

(III) PETERSON'S SOLUTION : (BEST)

P_0

```
while (1) {
1.  flag[0] = T
2.  turn = 1
3.  while (turn==1 && Flag[1]==T);
4.  critical section
5.  flag[0] = F
}
```

P_1

```
while (1)
{
11. flag[1] = T
12. turn = 0
13. while (turn==0 && flag[0]==T)
14. critical section
15. flag[1] = F
}
```

flag | F | F | turn = 0/1 (anything)

[0] [1]

P_0 P_1

- Combination of critical section problem using turn variable and flag variables → Peterson solution
- Mutual Exclusion is guaranteed.
- IMP The portion of data where the shared data variables or shared resources or shared data will be placed is called a critical section.

  The execution before critical section is called an initial section and after is called remainder section.
- Progress property is guaranteed since critical section entry is not always done.
- Bounded Waiting is also successful here because if time quantum of one process expires, other gets chance to enter critical section. So there is limited trapping time / waiting for limited Quantum time.
- No Deadlock because either instruction 3 / instruction 13 may work as trap but not both of them since turn = 0/1.

---

* SEMAPHORES :

- Semaphore is an integer variable which is used in mutual exclusive manner by various concurrent processes inorder to achieve process synchronization.
- These solve the critical section problem by using two atomic operations { wait() signal() }.

Semaphore → Counting Semaphore ($-\infty$ to $+\infty$)

Semaphore → Binary Semaphore (0, 1)

* SECTIONS OF PROGRAM :

- Entry Section : It is part of the process which decides the entry of a particular process.
- Critical section : This part allows one process to enter and modify the shared variable.
- Exit section : Exit section allows the other process that are waiting in Entry section, to enter into the Critical Section. It also checks that a process that finished its execution should be removed through the section.
- Remainder section : All other parts of the code, which is not in Critical, Entry, Exit Sections is called Remainder section.

Note : • The entry of critical section is handled by the wait() function (or) P(), down functions.

• The exit from a critical section is controlled by the signal() (or) V(), or up() or Post(), release() etc...

(1.) COUNTING SEMAPHORE :

- Value of semaphore ranges from $-\infty$ to $+\infty$ (positive negative integers)
- Integer only (-2, 5, 0, -1, 3, 6, 7, 100...)

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

P Section

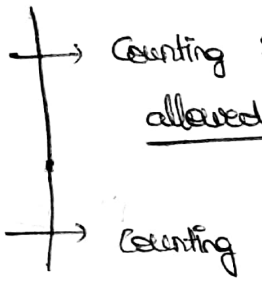
```
up (Semaphore S)
{
    S.Value = S.Value + 1 ;
    if (S.Value <= 0)
    {
        Select a process
        form suspended list ;
        Wake up() ;
    }
}
```

V section

Explaination :

- A counting semaphore has two components : 1. An integer value, 2. An associated waiting list (usually a queue)
- The value of semaphore may be positive or negative.
  - Positive value indicates the no. of process that can be present in the critical section at same time.
  - Negative value indicates the no. of process that are blocked in the waiting list.
- The waiting list of counting semaphore contains the processes that get blocked when trying to enter the critical section.
- In waiting list, the blocked process are put to sleep.
- The waiting list, is usually implemented using queue data structure.
- Using a queue or waiting list ensures bounded waiting.

- This is because, the process that arrives first in waiting queue gets the chance to enter the critical section first.
- The wait operation is executed when a process tries to enter the critical section.
- Wait operation decrements the value of counting semaphore by 1.



→ Counting semaphore value, $S >= 0$ ⇒ process is allowed to enter critical section.

→ Counting semaphore value, $S < 0$ ⇒ not allowed to enter critical section & process is put to sleep().

- The signal operation is executed when a process takes exit from the critical section
- Signal operation increments counting semaphore value by 1.

→ $S <= 0$ ⇒ choosen from waiting list ; wake up();

→ $S > 0$ ⇒ No action.

- By adjusting the value of counting semaphore, the no. of processes that can enter the critical section can be adjusted.
- If the value of counting semaphore is initialized with N, then the maximum N processes can be present in the critical section at any given time.

(Minimum may be 0)

- To implement mutual exclusion, the value of counting semaphore, $S = 1$ initialized.

Note:

- Consider 'n' units of particular non-shareable resources are available.
- Then 'n' processes can use these 'n' units at same time.
- So, the access to these units is kept in critical section.
- The value of counting semaphore is initialized to n, $(S = n)$

Example:

Consider initial value of semaphore, $\boxed{S = 12}$

Performing 5P operations, 2V operations, 4V operations, 8P operations, 10V operations, 1P operation, 3P operation, 2V operations, what is the final value of semaphore.

$S = 12$

After 5P operations, $S = 12 - 5 = 7$

After 2V operations, $S = 7 + 2 = 9$

After 4V operations, $S = 9 + 4 = 13$

After 8P operations, $S = 13 - 8 = 5$

After 10V operations, $S = 5 + 10 = 15$

After 1P operation, $S = 15 - 1 = 14$

After 3P operations, $S = 14 - 3 = 11$

After 2V operations, $S = 11 + 2 = 13 = 13$

$\therefore$ Final value of $S = 13$ (Max 13 process can access critical section at any given time.)

P ≡ Subtract 1
V ≡ add 1

Another approach to same problem is,

- P operation decrements by 1.
- V operation increments by 1.

Final value of Semaphore S is

$$= \text{initial value} + \sum \left[(-P \text{ operations}) + (V \text{ operations}) \right]$$

$$= 12 - (17 \times 1) + (18 \times 1)$$

total P operations = 17
total V operations = 18

$$= 13$$

---

*. A shared variable X, initialized to zero, is operand on by four concurrent process W, X, Y, Z as follows. Each of the process W, X reads x from memory, increments by one, stores it to memory and then terminates. Each of the process Y and Z reads x from Memory, decrements by two, stores it to memory, and then terminates. Each process before reading x invokes P operation on Counting semaphore S and invokes the V operation on semaphore S after storing x to memory. Semaphore S is initialized to two. What is the maximum possible value of x after all process complete execution.

(1) -2
(2) -1
(3) 1
(4) 2 ✓

Max = 2 → W, X
Min = -4 → Y, Z

If S = 2
& x = 0

Sol: Given,

Process W	Process X	Process Y	Process Z
Wait(s)	wait(s)	wait(s)	wait(s)
Read(x)	Read(x)	Read(x)	Read(x)
x = x+1;	x = x+1;	x = x-2;	x = x-2;
Write(x)	Write(x)	Write(x)	Write(x)
Signal(s)	Signal(s)	signal(s)	signal(s)

Initially, Counting semaphore S is initialized with value 2.

Required, the maximum possible value of x after all process complete execution.

Strategy :

- First of all, make the process W and read value of x=0
- Then preempt the process W.
- Now schedule process Y and process Z to execute one by one
- After executing them, reschedule process W.
- Now when process W gets scheduled again, it starts with value x = 0 & increments this value.
- This is because before preemption, it had read the x = 0
- The updates from the processes Y and Z gets lost.
- Later, execute process X which again increments value of x.

Logically,

S = 2 $\Rightarrow$ Max 2 processes can be there in critical section

To get maximum value (Y & Z process are not be present)

in critical section because they decrement x. So, there can be maximum of 2 process. So, if W, X are there, x shared variable incremented by 1 twice.

W → x = x+1 = 0+1 = 1

X → x = x+1 = 1+1 = 2 (max)

(or) W, X
X, W
(vice-versa)

Maximum possible value of x = 2

Minimum value of (x = 4) (because of Y, Z) x = x-2, x = x-2 } = -4

(2) BINARY SEMAPHORE : (mutex)

- The values of Binary semaphore is 0, 1.

```
Down (Semaphore S)
{
    if (S.value == 1)
    {
        S.value = 0;
        Critical Section
    }
    else
    {
        Block this process,
        Place in suspended
        list, Sleep();
    }
}
```

```
Up (Semaphore S)
{
    if (Suspended list is Empty)
    {
        S.value = 1;
    }
    else
    {
        Select a process from
        suspend list &
        wake up();
    }
}
```

- S = 1 → S = 0 → enters critical section : Down().
- S = 0 → S = 0 → sleep() : Down().
- S = 0/1 → suspended list empty → S = 1 : up().

Ex:

Assume Binary Semaphore value, S = 0 (initially)

2 P operations → 2 processes are in suspended list waiting

Making S = 0

20 V operations → For first 2V operations, 2 process are selected from the suspended list and entered into ready state. For remaining 18 V operations, the suspended list is empty.

Making S = 1.

3 P operations → 1 process gets into the critical section and other 2 processes are into suspended list waiting.

Making S = 0.

30 V operations → For first 2V operations, 2 process are selected from the suspended list and entered into ready state. For remaining 28 V operations, the suspended list is empty.

Making S = 1.

4 P operations → 1 process gets into the critical section and other 3 process into suspended list waiting.

Making S = 0.

40 V operations → for first 3V operations, 3 processes are selected from the suspended list and entered into ready state. For remaining 37 V operations, the suspended list is empty.

Making S = 1

∴ Final value of Binary Semaphore, S = 1

- A process can enter the critical section only after performing down() [P() operation] i.e. executing entry section code.
- So, the finally, the critical section has 2 process and suspended list (waiting queue) is empty having no processes waiting.

---

* SOLUTION OF PRODUCER CONSUMER PROBLEM using Semaphore

Counting Semaphore → full = 0 : No. of filled slots
→ Empty = N : No. of Empty slots

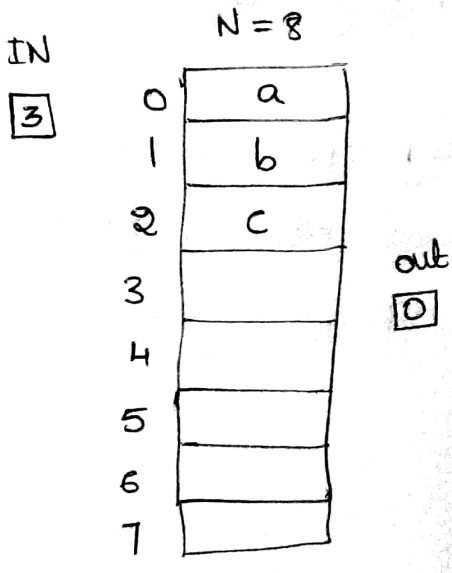
Binary Semaphore, S = 1;

Produce Item (item P);

1) down (empty);
2) down (S);

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

3) up (S);
4) up (full);



Consumer

```
down (full);
down (S);
Item c = Buffer [out];
out = (out + 1) mod n;
up (S);
up (Empty);
```

Here From Buffer diagram,

Empty = ~~5~~ 4 Full = ~~3~~ 4

S = ~~X~~ ~~0~~ 1 n = 8

In = (3+1) mod 8 = (3+1) mod 8 = 4

c = ~~0~~ X 0

out = ~~0~~ 1 mod 8 = 1

S = X ~~0~~ 1

S = 1 finally

- Consistency can be checked by { Empty + Full = n }

Case - II:

Empty = ~~5~~ 4 → producer preempted

full = ~~3~~ 2 ← consumer ← context switching

S = X 0

out = 1 ← After consumer entered critical section

← → consumer preempted

} Inconsistency

Empty ≠ n + Full

Empty = ~~5~~ 4 3

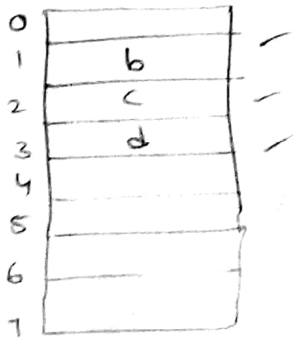
S = X ~~0~~ 0 → producer block

So, consumer executes up(S) ⇒ S = ~~0~~ 1

Now, producer gets chance.

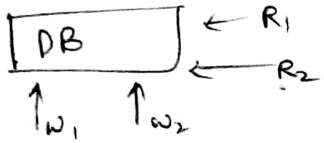
Completes Empty = 5 , Full = 3

} Consistency



* . READER - WRITER PROBLEM :

Same Data



① R - W = problem
② W - R = problem
③ W - W = problem
} Reader - Writer problem

R - R = no problem

- Creates data loss, inconsistency.
- Solved by semaphore.

```
int rc = 0                          // read count : No. of readers in buffer
Semaphore mutex = 1;                // Binary semaphore name
Semaphore dB = 1;                   // Binary semaphore Kind

Void Reader (void)
{
    while (true)
    {
        down(mutex);
        rc = rc + 1;
        if (rc == 1)
        then down(dB);
        up(mutex)
        DB
        down(mutex);
        rc = rc - 1;
        if (rc == 0) then up(dB);
        up(mutex)
        process_data
    }
}

Void Writer (void)
{
    while (true)
    {
        down(dB);                   // Entry
        DB                          // Critical section
        up(dB);                     // Exit code
    }
}
```

Entry code

Exit code

① mutex = ✗ ∅ 1
rc = ∅ 1 → DB (R1)
dB = ✗ 0
If writer comes → Blocked because dB=0

Critical section

// problem solved

② dB = ✗ 0 → dB (W1)

Reader R_1 comes

mutex = ✗ 0

rc = ⌀ 1

dB = ⌀ 0 → Reader blocked

// problem solved

③ dB = ✗ 0

Writer W_2 comes

dB = ⌀ 0

⇓

Writer blocked

// problem solved

④

R1 reader

mutex = ✗ 0

rc = ⌀ 1

dB = ✗ 0

mutex = ~~✗~~ ⌀ 1

DB (R1)

R2 reader

rc = ⌀ 1

mutex = ✗ ⌀ ✗ 0

rc = ⌀ ✗ 2 ✓

mutex = ✗ ⌀ ✗ ⌀ 1 ✓

DB (R2)

R3 reader

mutex = ✗ 0

rc = ⌀ ✗ ✗ 3

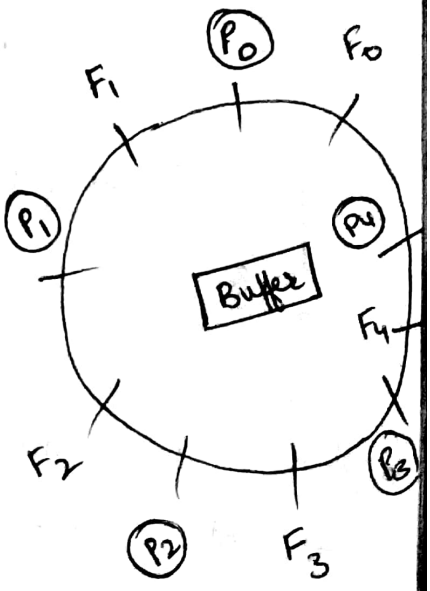
mutex = ⌀ 1

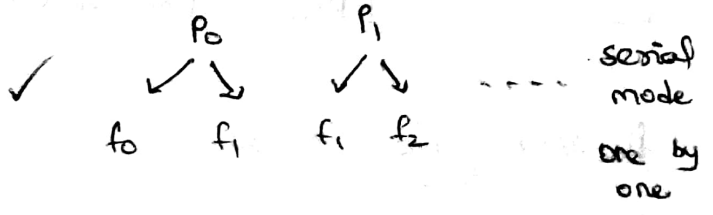
DB (R_3)

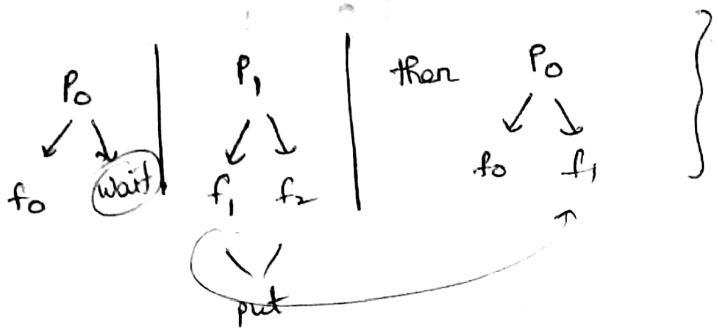
// No problem

*. DINING PHILOSOPHER PROBLEM :

```
void philosopher (void)
{
    while (true)
    {
        Thinking ();
        takeFork (i) ;          // Left Fork
        takeFork ((i+1) mod N)  // Right Fork
        Eat ();
        put fork (i);
        put fork ((i+1) mod N);
    }
}
```







There is problem of Race condition here

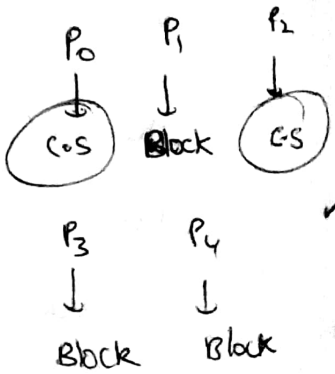
Solution using Semapheres :

- S[i] // Five Semapheres $S_0 = S_1 = S_2 = S_3 = S_4 = 1$

```
void philosopher (void)
{
    while (true)
    {
        Thinking();
        wait ( takeFork (Si))
        wait ( takeFork ( Si+1 mod N))      } Entry code
        EAT();                              } Critical section
        Signal ( Put Fork (i); )
        signal ( PutFork ( i+1 mod N))      } Exit code
    }
}
```

Each philosopher (P_i) needs 2 semapheres (S_i , $S_{(i+1) \bmod N}$)

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



- Here more than 1 process (philosopher) can enter Critical section but these processes should be independent to each other (Mutual Exclusion)

- Deadlock may occur ⟶ (GATE)

P_0 → fo / wait → preempted

P_1 → S_1 / wait → preempted

P_2 → S_2 / wait → preempted

P_3 → S_3 / wait → preempted

P_4 → S_4 / wait → preempted

All are in blocked stated for infinite state time ⟹ DEADLOCK : No progress

Bad Solution : Remove any process

Another better solution : Change sequence of one process

P_0, P_1, P_2, P_3 → left → Right

P_4 → Right → left

S_0	S_1	S_2	S_3	S_4
↓	↓	↓	↓	↓
✗	✗	✗	✗	✗

Deadlock ← $S_0 = S_1 = S_2 = S_3 = S_4 = 0$

- Better Solution :

others ← same sequence

$P_0 \rightarrow S_0\ S_1$

$P_1 \rightarrow S_1\ S_2$

$P_2 \rightarrow S_2\ S_3$

$P_3 \rightarrow S_3\ S_4$

Critical Section

Avoided deadlock

$P_4 \rightarrow S_0\ S_4$ → Blocked

any ← sequence changed : ①

- Nth philosopher :

wait (takefork ($S_{i+1 \bmod N}$));

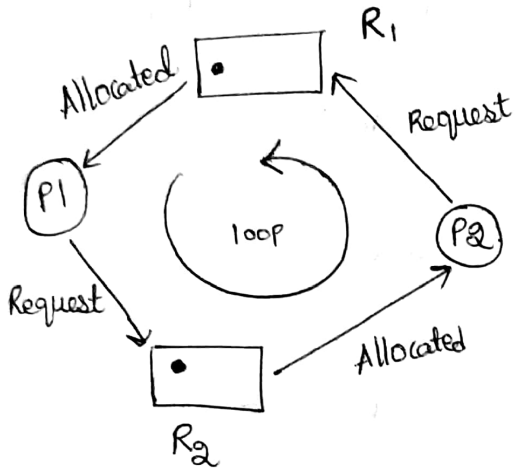
wait (takefork (S_i));

(N-1)th philosopher, (N)th philosopher → same

(GATE)

*. 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.

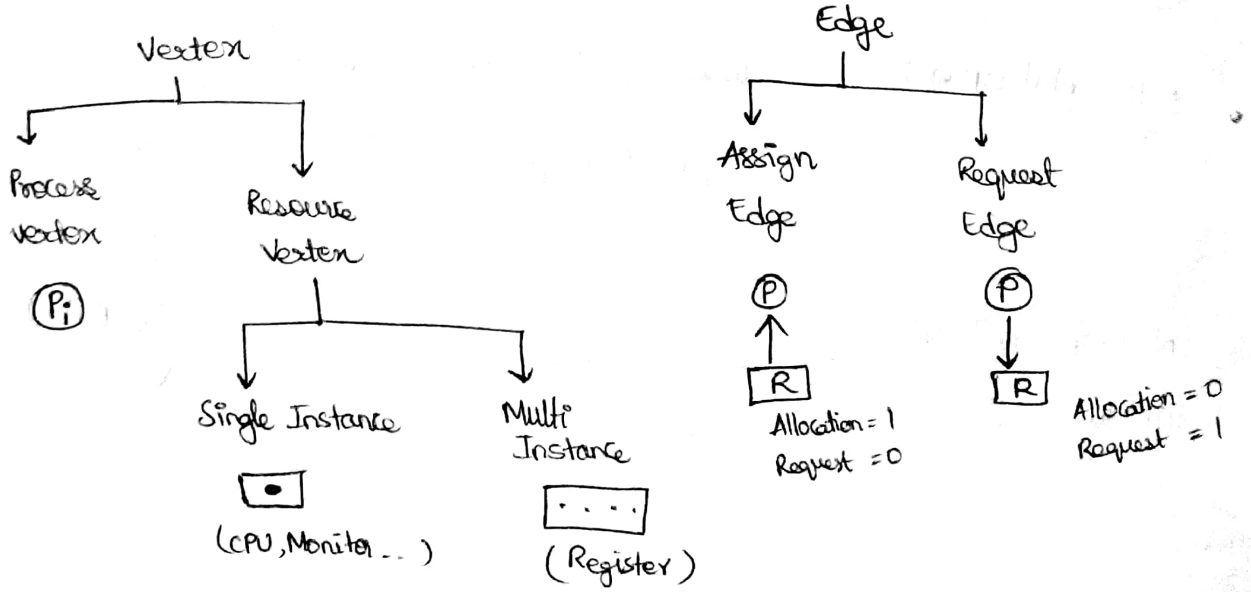


Similar situation like Deadlock. Neither P1, P2 will proceed further because one is locked by other requirement.

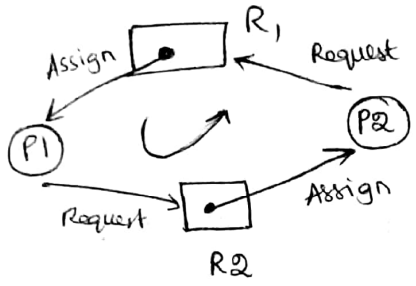
- 4 conditions for in Deadlock state :

1. Mutual Exclusion - more than 1 process can use at same time
2. No preemption - Neither of the process terminates / context switches.
3. Hold & Wait - Hold & request for another resource.
4. Circular Wait - loop in the execution schedule.

*. RESOURCE ALLOCATION GRAPH : (RAG)



- Resource of Single Instance



Circular wait ✓

P1: Holding R1 & waiting for R2 ✓

P2: Holding R2 & waiting for R1

Neither P1, P2 context switches

No preemption ✓

(DEADLOCK) ✓

	Allocate		Request	
	R1	R2	R1	R2
P_1	1	0	0	1
P_2	0	1	1	0

Availability : (0, 0)
R_1 R_2

because single Instance →

We cannot fullfill the request of either R1 (or) R2.

(DEADLOCK) ✓

Ex: Check whether in Deadlock or not ?

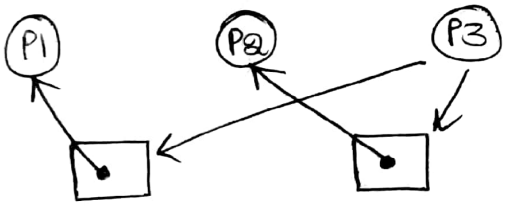


Diagram:

	Allocation		Request	
	R1	R2	R1	R2
P_1	1	0	0	0
P_2	0	1	0	0
P_3	0	0	1	1

Availability = (0, 0)
P1 P2

P_1 → not requesting anything → It executes & terminates

New Availability = (1, 0)
R1 R2

P_2 → not requesting anything → It executes & terminates

New Availability = (1, 1)

P3 → requesting R1 ✓ → executes & terminates
requesting R2 ✓

(Availability = (1, 1)
R_1 R_2
of P_1 P_2

All process are executed. So, No deadlock.

- Waiting for Resource

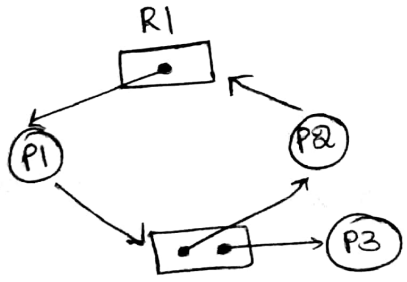
time: finite → Starvation
Infinite → Deadlock

- Cyclic – Starting & reaching same node in RA Graph
- Acyclic – Not reaching the same starting point in RA Graph.

NOTE: In Single Instance Resource, if there is circular wait then there exists Deadlock. (viceversa is true)

NOTE: In single Instance Resource, if there is no circular wait or it is in Acyclic form then there is No Deadlock. (vicevesa is also true).

EX: Check whether given RAG has deadlock?



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

Current Availability = (0, 0)
R1 R2

P3 → not requesting anything → executes & terminates

New Availability ≡ (0, 1)
R1 R2

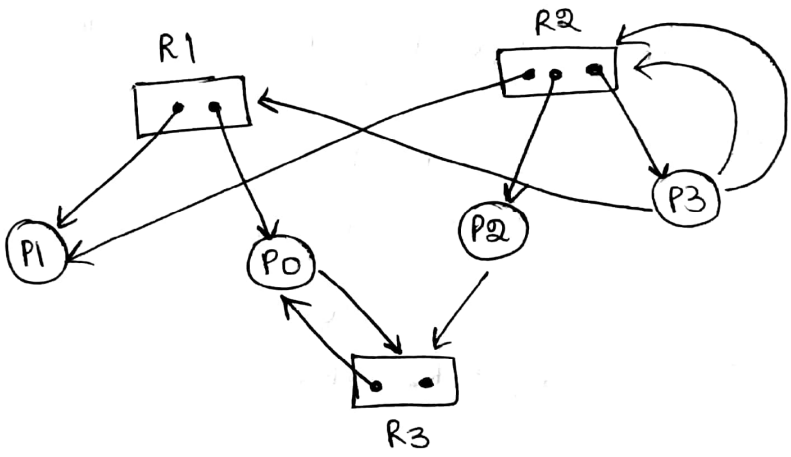
P1 → requesting R2 ✓ → executes & terminates

New Availability ≡ (1, 1)
R1 R2

P2 → requesting R1 → executes & terminates

∴ No deadlock.

GATE :



	Allocation			Request		
	R1	R2	R3	R1	R2	R3
P_0	1	0	1	0	1	1
P_1	1	1	0	1	0	0
P_2	0	1	0	0	0	1
P_3	0	1	0	1	2	0

Current Availability ≡ (0 0 1)
R1 R2 R3

001 ⇒ Request of P2

- P2 → executes & terminates

So, its Allocation (0 1 0) is added to Current Availability

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

New Availability ≡ (0 1 1)

R1 R2 R3

011 ⇒ Request of P0

- P0 → executes & terminates

So, its Allocation (1 0 1) is added to last new Availability

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

New Availability ≡ (1 1 2)

R1 R2 R3

- P1 needs only 1 of R1 ⇒ (1 0 → request; 1 1 2 → matched) ✓

⇒ P1 → executes & terminates

So, its Allocation (1 0 0) is added to Current Availability

$$\begin{array}{r} 1\ 1\ 2 \\ (+)\ 1\ 0\ 0 \\ \hline 2\ 2\ 2 \end{array}$$

(2 2 2 → Availability; 1 2 0 → Request of P3) ✓

∴ P3 executes & terminates

⇒ NO DEADLOCK

last updated Availability = 2 2 2
0 1 0
2 3 2 ✓

Sequence of execution : $P_2 \rightarrow P_0 \rightarrow P_1 \rightarrow P_3$

Multiple instance & Circular Wait $\Rightarrow$ (Always Not Deadlock) } FALSE
may be may not be

---

*. DEADLOCK HANDLING :

Various methods to handle deadlocks :

① Deadlock ignorance (ostrich method) :
- because we don't want to affect performance

② Deadlock Prevention :
- Any 1 of the conditions → Mutual Exclusion, No Preemption, Hold & Wait, Circular Wait } will be made false. So deadlock is prevented
- Preempt one of the process / priority based scheduling, Time Quantum, etc...
- remove circular wait by giving numbering to each of the resource (increasing order)

③ Deadlock Avoidance : (Banker's Algorithm)
- Dijkshtra's Algorithm
- Banker's Algorithm (check) —IMP→ [Remaining Need ≤ Current Availability]

④ Deadlock detection & Recovery :
- Kill the process
- Resource Preemption

*. BANKER'S ALGORITHM :

- It is a Deadlock Avoidance Algorithm.
- It is also used for Deadlock Detection.

EX: Assume an Example scenario as follows :

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

Resources
A : CPU
B : Memory
C : Printer

Total A = 10, B = 5, C = 7 (Given

Can Deadlock be detected ?

Safe - Deadlock doesn't occur → Safe Sequence.

No Safe - Deadlock may occurs → Need to detect & prevent

Unsafe The system may terminate Apply Banker's Algorithm.

Process	Total Allocation			Total Max			Total Availability			Remaining		
P_i	A	B	C	A	B	C	A	B	C	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

Remaining Need = |Max Need - Allocation|

Available = |Total Allocation - Given Total|

Availability current = (3 3 2)
A B C

P_2 → 1 2 2 ⟹ P_2 is given chance, it executes & terminates. It's Allocation (2 0 0) is added to Availability

Availability current → 3 3 2

✓ ✓ ✓

1 ≤ 3 2 ≤ 3 2 ≤ 2

Avai: 3 3 2

P_2: 2 0 0

5 3 2 → New Availability

P_4: 2 1 1 → P_4 is given chance & executes P_4 Allocation (2 1 1) is added to last row Availability

✓ ✓ ✓

2 ≤ 5 1 ≤ 3 1 ≤ 2 ✓

Avai: 5 3 2

P_4: 2 1 1

7 4 3 → P_5 is given chance & executes. Allocation (0 0 2) is added to last Availability

P_5: 5 3 1

✓ ✓ ✓

5 ≤ 7 3 ≤ 4 1 ≤ 3

Avai: 7 4 3

P_5: 0 0 2

7 4 5 → P_1 is given chance

7 4 5

0 1 0

7 5 5 → P_3 is given chance

⇒ 7 5 5

P_3 : 3 0 2

10 5 7 → Given (∵ Deadlock not possible)

✓

Execution order : $P_2 \rightarrow P_4 \rightarrow P_5 \rightarrow P_1 \rightarrow P_3$

GATE 2018 :

Given :

Check Deadlock occurs or not ?

Process	Allocation E	F	G	Max Need E	F	G	Available E	F	G
P_1	1	0	1	4	3	1	3	3	0
P_2	1	1	2	2	1	4			
P_3	1	0	3	1	3	3			
P_4	2	0	0	5	4	1			
Total :	5	1	6						

Sol :

Process	Allocation E	F	G	Max Need E	F	G	Availability E	F	G	Remaining Need E	F	G
P_1	1	0	1	4	3	1	3	3	0	3	3	0
P_2	1	1	2	2	1	4	4	3	1	1	0	2
P_3	1	0	3	1	3	3	5	3	4	0	~~3~~	0
P_4	2	0	0	5	4	1	6	4	6	3	4	1
	5	1	6									

Current Availability : 3 3 0 ~ P_1 Remaining Need ✓ → P_1 executes

3 3 0
+ 0 1
4 3 1

4 3 1 ~ 1 0 2 P_2 Remaining Need ✗ → P_2 fails

4 3 1 ~ 0 3 0 P_3 Remaining Need ✓ → P_3 executes

4 3 1
+ 0 3
5 3 4

5 3 4 ~ 3 4 1 P_4 remaining need ✗ → P_4 fails

5 3 4 ~ 1 0 2 P_2 Remaining need → ✓ P_2 executes

5 3 4
+ 1 1 2
6 4 6

6 4 6 ~ 3 4 1 P_4 Remaining Need → ✓ P_4 executes

Last Availability = 6 4 6
− 2 0 0
8 4 6 (Total Availability initially) ✓

* Total Availability initially = Total Allocation + Current Availability *

Execution order : $P_1 \rightarrow P_3 \rightarrow P_2 \rightarrow P_4$

∴ No Deadlock.

** IMP **

Remaining Need ≤ Current last Availability

*. Q) A system is having 3 process, each require 2 units of resources 'R'. The minimum no of units of R, such that no deadlock will occur

(GATE)

a. 3 b. 5 c. 6 d. 4 ✓

Sol: Total process = 3 (P_1 P_2 P_3)

Each requires = 2 units resources

Total units needed = Total process × Each requirement

= 3 × 2 = 6 units

But minimum is asked in question. So,

i) If Resources = 2 ⟶ P_1 = 2 ⟶ executes
then P_2 = 2 ⟶ executes
then P_3 = 2 ⟶ executes
} Sequential case of execution

✗ It is not possible if P_1 executes & its 2 allocated resources are distributed equally to P_2 P_3

1 1

then there is 1 requirement of P_2
1 requirent of P_3 ⟶ DEADLOCK occurs.

ii) If Resources = 3

P_1	P_2	P_3	
1	1	0	⟶ happens (Deadlock free)
1	0	0	

✗ & one of the case Hold & Wait ✓

P_1	P_2	P_3	
1	1	1	⟶ Not happens (Deadlock occurs)

iii. If Resources = 4

P_1	P_2	P_3	
1	1	1	→ Deadlock free (✓)
1	0	0	

Any of the Cases of Allocation ⟶ No deadlock.

If the No. of process = 3 & Resources = 4 ⟹ Deadlock doesn't occur.

Logic :

No. of process = n

Resources = r $+1$

1	2	3	4	. . .	n-1	n
r-1	r-2	r-3	r-4	. . .	r-(n-1)	r-n

then this is logic for no deadlock. $+1$ is add

What is the maximum no. of resources that are allocated but still there is deadlock exists ⟹ (Minimum resources to avoid Deadlock) -1

$=$ **Min − 1**

GATE 2018 9) Consider a system with 3 process 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 ____

If $k = 2$ ⟶ To avoid deadlock

If $k > 2$ ⟶ Deadlock occurs.

Formula :

Total no. of resources = R

Total no. of process = n (P_1 P_2 .. P_{n-1} P_n)

Demand be d_1 d_2 ... d_{n-1} d_n.

Give → d_1-1 d_2-1 .. $d_{n-1}-1$ d_n-1

Total Demand = $\sum_{i=1}^{n} d_i - n$ → $R \leq \sum_{i=1}^{n} d_i - n$: Deadlock

$R \geq \sum_{i=1}^{n} d_i - n \Rightarrow R+n > \sum_{i=1}^{n} d_i$

★ Total resources + Total process > Total demand Sum (demand) * IMP ✓ follow this

- Total demand = Total process × Maximum request

★ Total resources + Total process > Total process × Maximum request * IMP

→ No deadlock if this Condition TRUE

$6+n > 2n$

$2n-n < 6$

$n < 6$ $(n=5)$

Ex:

Consider a computer system that has N numbers of magnetic tape drives, which are shared among three concurrent process P1, P2, P3. The individual maximum demands of P1, P2, P3 are 4, 6, 9 respectively. It is added that Combined maximum demand of P1, P3 together will never exceed 10. In this scenario, what will be the minimum value of N that can ensure the system to be deadlock free?

Sol:

Total resources = N

Total processes = 3

Peak demand of P1 = 4

Peak demand of P2 = 6

Peak demand of P3 = 9

There is no condition on P2 $\Rightarrow$ P2 needs 6, give it 5.

Total resources + Total process > Total demand

let P1 needs x units

let P3 needs y units

Given, Combined maximum demand of P1, P3 never exceeds 10 $\Rightarrow$ $x + y < 10$

From the concept of Pigeonhole Principle,

$$\left[(6-1) + (x-1) + (y-1)\right] + 1 = \text{Required minimum resources}$$

$$N = 5 + (x-1) + (y-1) + 1$$

$$N = x + y + 4$$

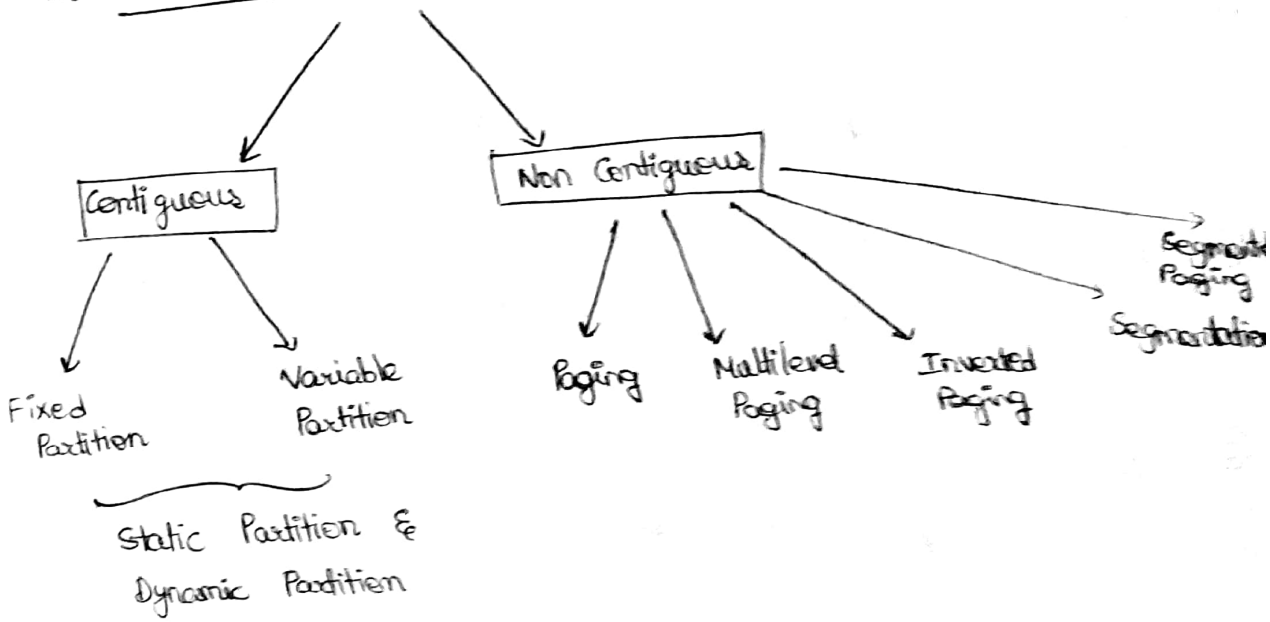
For N to be minimum $\Rightarrow$ $x + y = 10$
& deadlock free scenario

$N = (10) + 4 \Rightarrow$ Minimum value of $\boxed{N = 14}$

MEMORY MANAGEMENT

- It is a process of managing primary Memory (RAM) & any other memory resources present in the system.
- Goal : Efficient utilization of memory
- Fellest degree of Multi programming.

*. MEMORY MANAGEMENT TECHNIQUES :



*. FIXED PARTIONING :

- No. of partitions are fixed.
- Size of each partition may or may not same.
- Contiguous allocation, so spanning is not allowed.

Disadvantages :

1. Internal Fragmentation.
2. Limit in Process size.
3. Limitation on degree of multi programming.
4. External Fragmentation.

*. VARIABLE SIZE PARTIONING :

- There is no chance for internal Fragmentation
- There is no limitation on no of process.
- There is no limitation on process size
- But External Fragmentation still exists.
- Allocation/ Deallocation is complex.

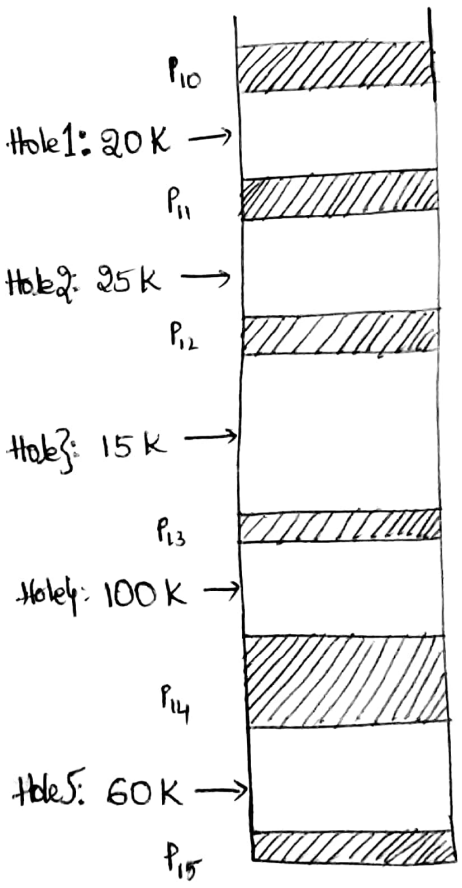
*. MEMORY ALLOCATION ALGORITHMS :

(1.) First - Fit : Allocate the first hole that is big enough.

(2.) Next - Fit : Same as first fit but always start searching from last allocated hole.

(3) Best - Fit : Allocate the smallest hole that is big enough.

(4) Worst Fit : Allocate the largest hole.



- Suppose P_1 = 22K → Hole 2 according to First Fit
- Next P_2 = 13K → Hole 3 according to Next Fit
- Next P_3 = 50K → Hole 5 according to Best Fit
- Next P_4 = 30K → Hole 4 according to Worst Fit
- Next P_5 = 12K → Hole 1 according to Next Fit

GATE:

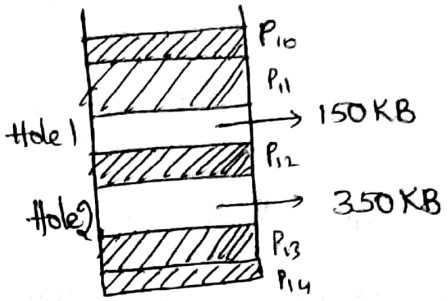
Requests from process are 300K, 25K, 125K, 50K respectively the above requests could be satisfying with:

A. Best fit but not first fit

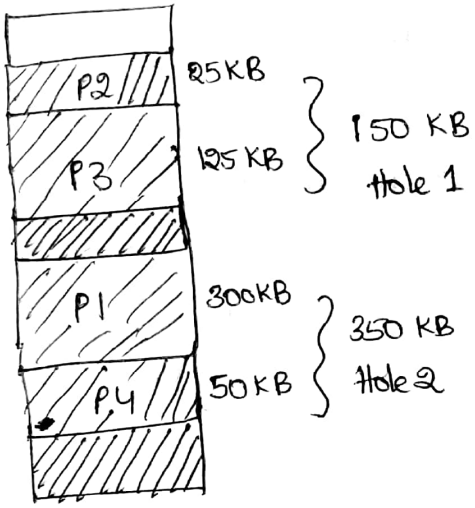
✓ B. First fit but not best fit

C. Both

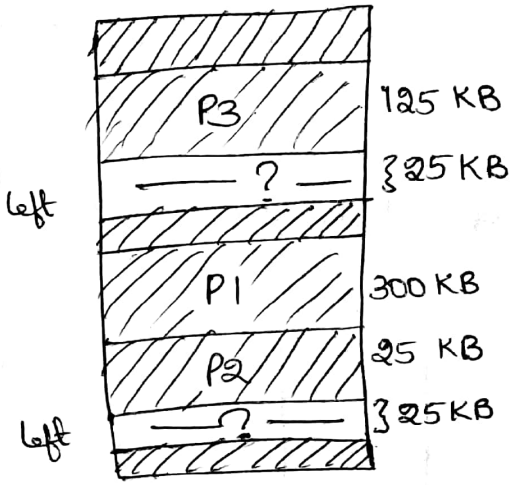
D. None



First Fit:



Best Fit:

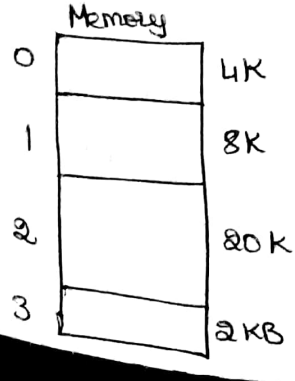


Still P_4 can't allocated

∴ First Fit allocation satisfies but not Best Fit Memory Allocation.

GATE: Given allocation using Best Fit, (sequency wise)

Request No	J_1	J_2	J_3	J_4	J_5	J_6	J_7	J_8
Request size	2K	14K	3K	6K	6K	10K	7K	20K
Usage time (Sec)	4	10	2	8	4	1	8	6



Calculate the time at which J7 will be completed ____

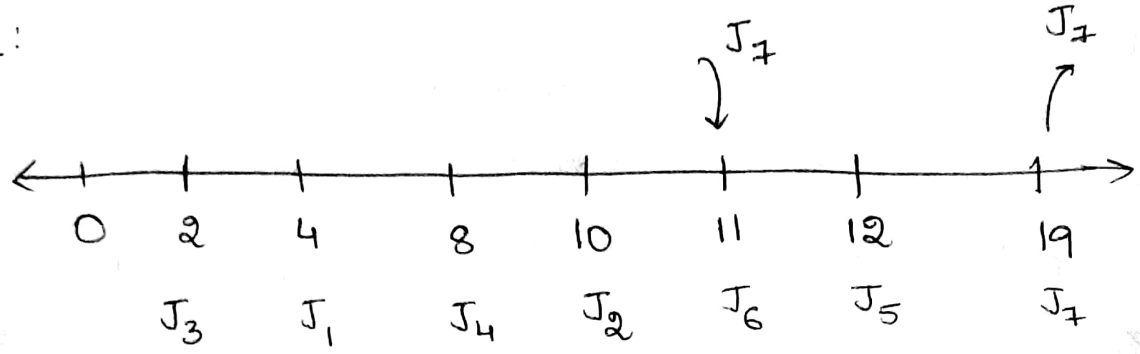
a. 17

✓ b. 19

c. 20

d. 37

Sol:



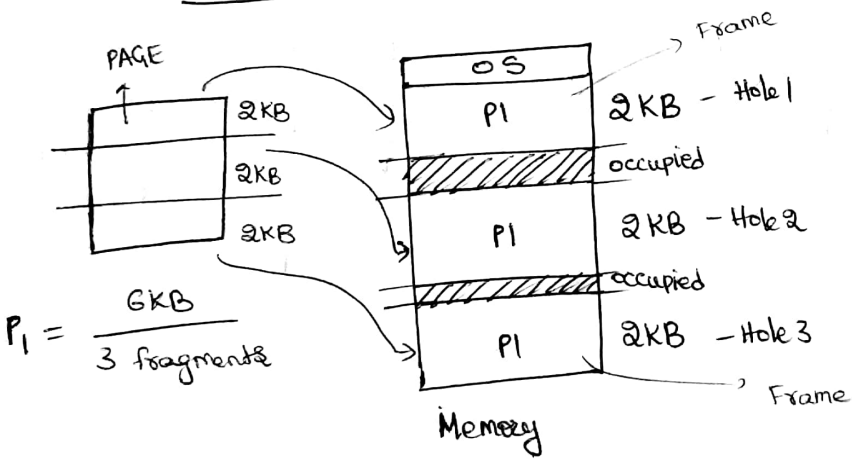
At 11th sec, J_6 leaves & makes Hole 2 of 20K empty which will be occupied by J_7.

J_7 starts at 11 & takes 8 sec $\Rightarrow$ 11 + 8 = 19 sec

J_7 completed by 19 sec. (option B)

---

* NON-CONTIGUOUS MEMORY ALLOCATION :



- Solves problem of External Fragmentation

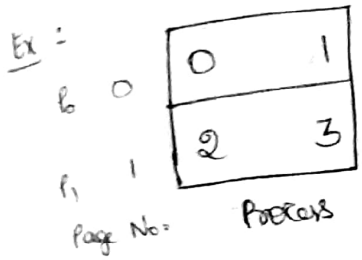
- **PAGE** : Before bringing the whole process into Memory, it is divided into fragments based on availability of Memory. These each fragment is called PAGE. Here, 6 KB is process & divided into 3 fragments each of 2KB. That 2KB (piece of code) is called PAGE. (done in secondary memory).

- **Frame** : The partitions of Main memory. (each fragment) to allocate a page.

*IMP | PAGE SIZE = FRAME SIZE |

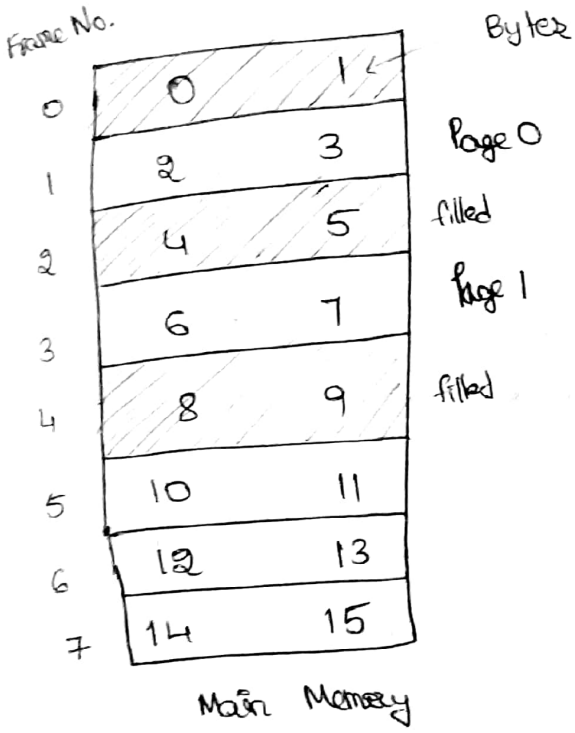
- Paging – Allocating the pages into frames of Memory.

Ex:



- Process size = 4 B
- Page size = 2 B
- No. of Pages = $\frac{\text{Process size}}{\text{Page size}}$

$= \frac{4B}{2B} = 2$ Pages

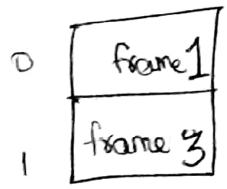


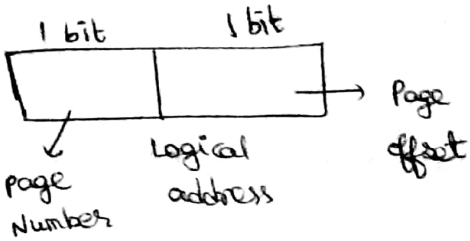
- Main Memory size = 16 B
- Frame size = 2 B

No. of Frames = $\frac{\text{Main Mem. Size}}{\text{Frame size}}$

$= \frac{16B}{2B} = 8$ frames

Page Table of P1



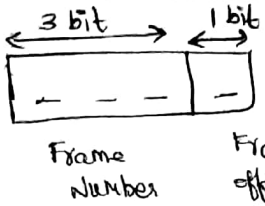
- CPU always works on logical address
- 

Suppose if CPU needs byte 3 of process P1, → It will follow the logical address of Process & Page Table of process P1

↑

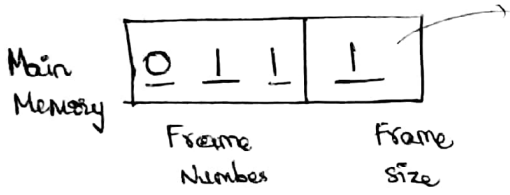
Converting Logical Address into physical address (directly associated by Main Mem.)

Physical Address



So, when CPU want Process P1's 3 byte

3 byte is in Page 1 of i.e. Frame No. = 3



0 is of byte 2 of Process P1
1 is of byte 3 of Process P1
] Inside Frame No. 3

Example : Given, the following in a Memory Allocation Scenario

Logical Address space LAS = 4GB

Physical Address space PAS = 64 MB

Page size = 4 KB then Find :

No. of Pages = ?

No. of Frames = ?

No. of Entries in Page table = ?

size of Page table = ?

Also, Memory is byte addressable.

LAS = 4GB $\Rightarrow 2^2 \times 2^{30}$ B $= 2^{32}$ B $\Rightarrow$ 32 bits are needed to represent LA.

Page size = 4 KB $= 2^{12}$ B $\Rightarrow$ 12 bits are needed to represent Page offset

* Total bits of Logical Address = Total bits needed to represent Pages + Total bits needed to represent Page offset

32 = Total bits needed to represent Pages + 12

$\Rightarrow$ Total bits needed to represent Pages = 20

$\Rightarrow$ No. of Pages $= 2^{(\text{bits needed for page})} = 2^{20}$.

PAS $= 64\,MB = 2^{26}\,B \Rightarrow$ 26 bits are needed to represent Physical Address (Main Memory size)

bits needed for Frame offset	=	bits needed for Page offset

* HINT

$\therefore$ Bits needed to represent Frame offset $= 12$

Total bits of physical Address	=	total bits needed to represent Frames	+	Total bits needed to represent Frame offset

$26 =$ Total bits of Frames $+ 12$

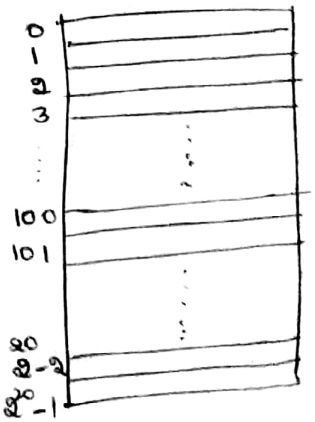
$\therefore$ Total bits needed to represent Frames $= 14$

$\Rightarrow$ No. of Frames $= 2^{(\text{bits needed for Frame})} = 2^{14}$.

No. of entries in Page Table	=	No. of Pages in process

$\Rightarrow$ No. of entries in Page Table $= 2^{20}$

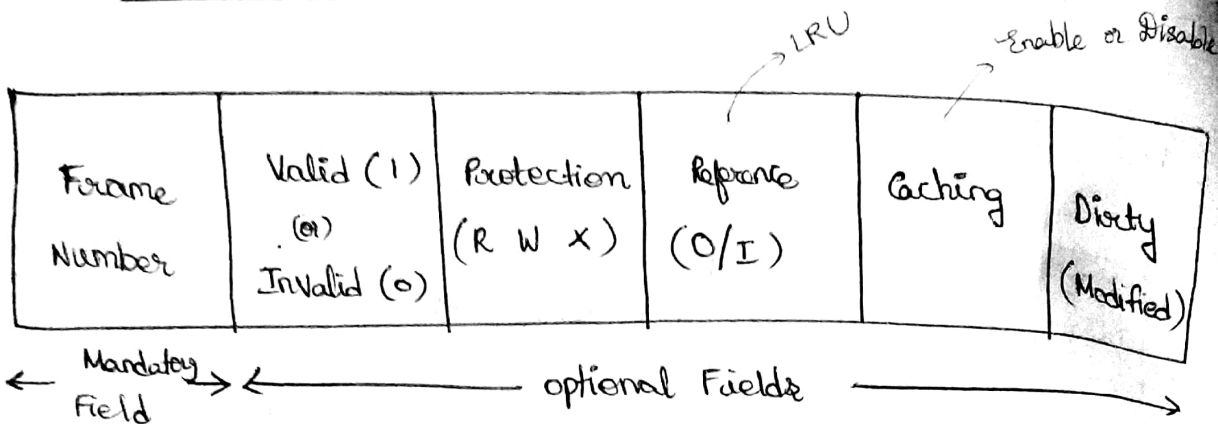
Page Table



Size of Page table	=	No. of entries in Page Table	×	Total bits for Frame

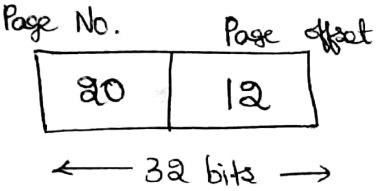
$\therefore$ Size of Page table $= 2^{20} \times 14 \approx 2\,MB$

* PAGE TABLE ENTRY :



Ex: Consider a virtual Address Space of 32 bits and Page Size of 4 KB. system is having RAM of 128 KB. Then what will ratio of page table & Inverted Page Table size if each entry in both is of size 4 B ?

Sol:



$VAS = 2^{32}$

$= 4GB$

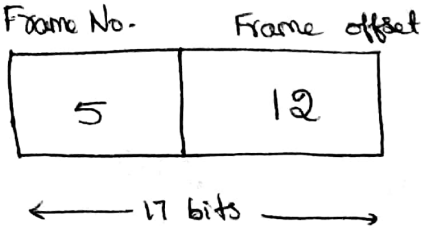
~ 32 bits

Page size = 4 KB = 2^{12} $\sim$ 12 bits

Page No. = 32 − 12 = 20 bits

Size of Page Table = No. of entries in Page Table × Total bits needed for frame

$\Rightarrow$ Size of Page Table = $2^{20} \times 4B = 2^{22} = 4MB$



PAS = RAM size

$= 128$ KB

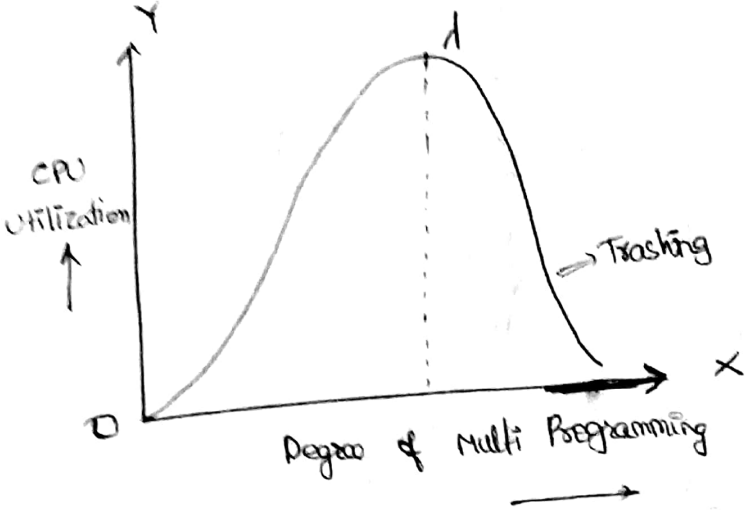
$= 2^{17}$

~ 17 bits

$\Rightarrow$ Size of Inverted Page Table = $2^5 \times 4B = 2^7 B$

ratio $\frac{\text{PT size}}{\text{IPT size}} = \frac{2^{20} \times 4B}{2^5 \times 4B} = 2^{15} : 1$

* THRASHING :

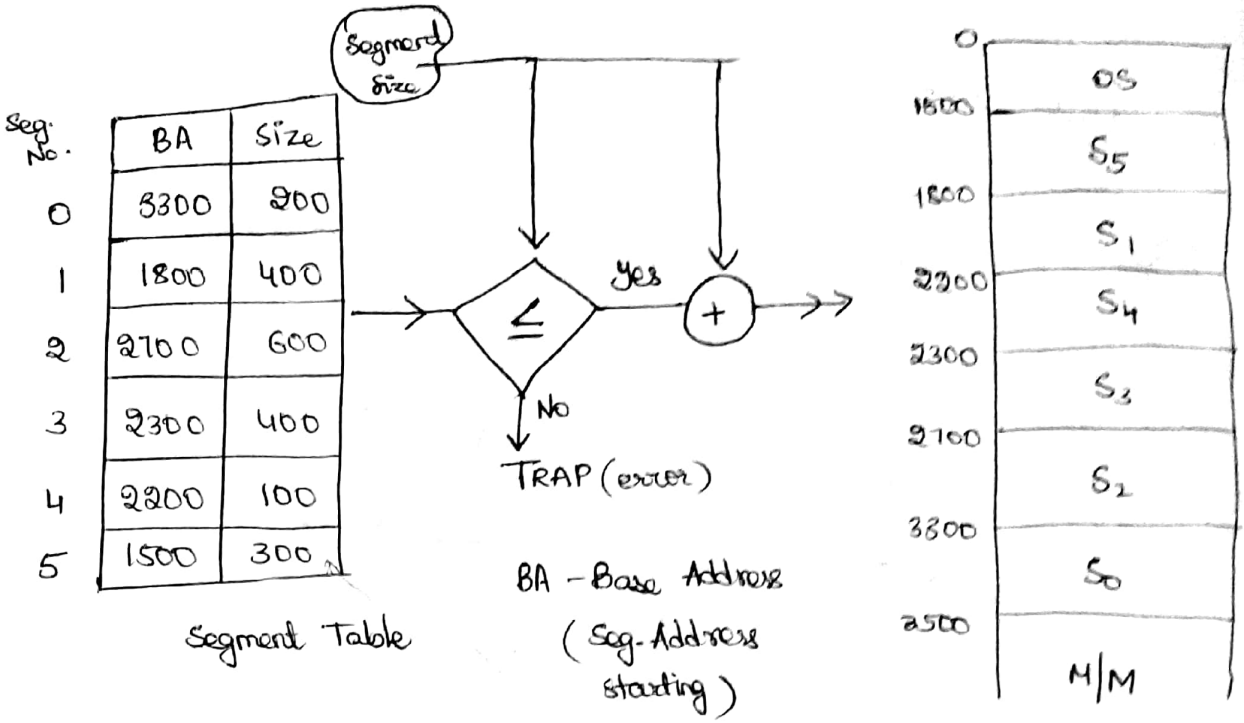


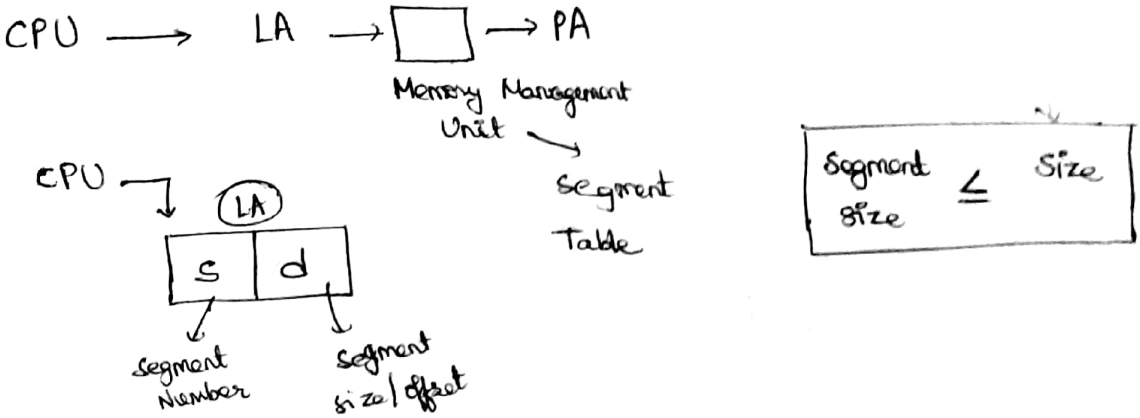
- Degree of Multi Programming - Maximum no. of pages should of a process (maximum process) should be brought into Main Memory.
- Trashing - Every one page of any process must be allocated to Main memory. After d, more no. of page faults will happen

* SEGMENTATION (vs) PAGING :

- Segmentation - The process of dividing any program / process into different parts & allocate them in main memory by proper order & Correct exact runnable code segmentation.
- Paging - Dividing any process into different parts & allocate them in main memory irrespective of code written inside it.

- Pages are of same size always. But segmentation is of different size depending on correct executable code.





*. OVERLAY :

- It is the method by which a large size process can be put into Main memory

Ex :

Consider a two pass assembler.

Pass 1 : 80 KB,

Pass 2 : 90 KB

Symbol Table : 30 KB

Common routine 20 KB

- At a time only one pass is in use. What is minimum partition size required if overlay driver is 10 KB size.

Sol:

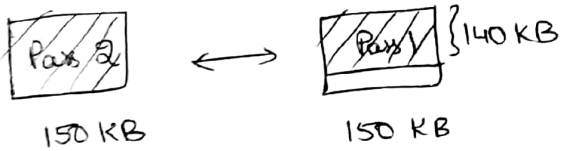
80 KB + 90 KB + 30 KB + 20 KB + 10 KB = Total size

Total size = 230 KB

For using Pass 1, Total size = 80 + 30 + 20 + 10 = 140 KB

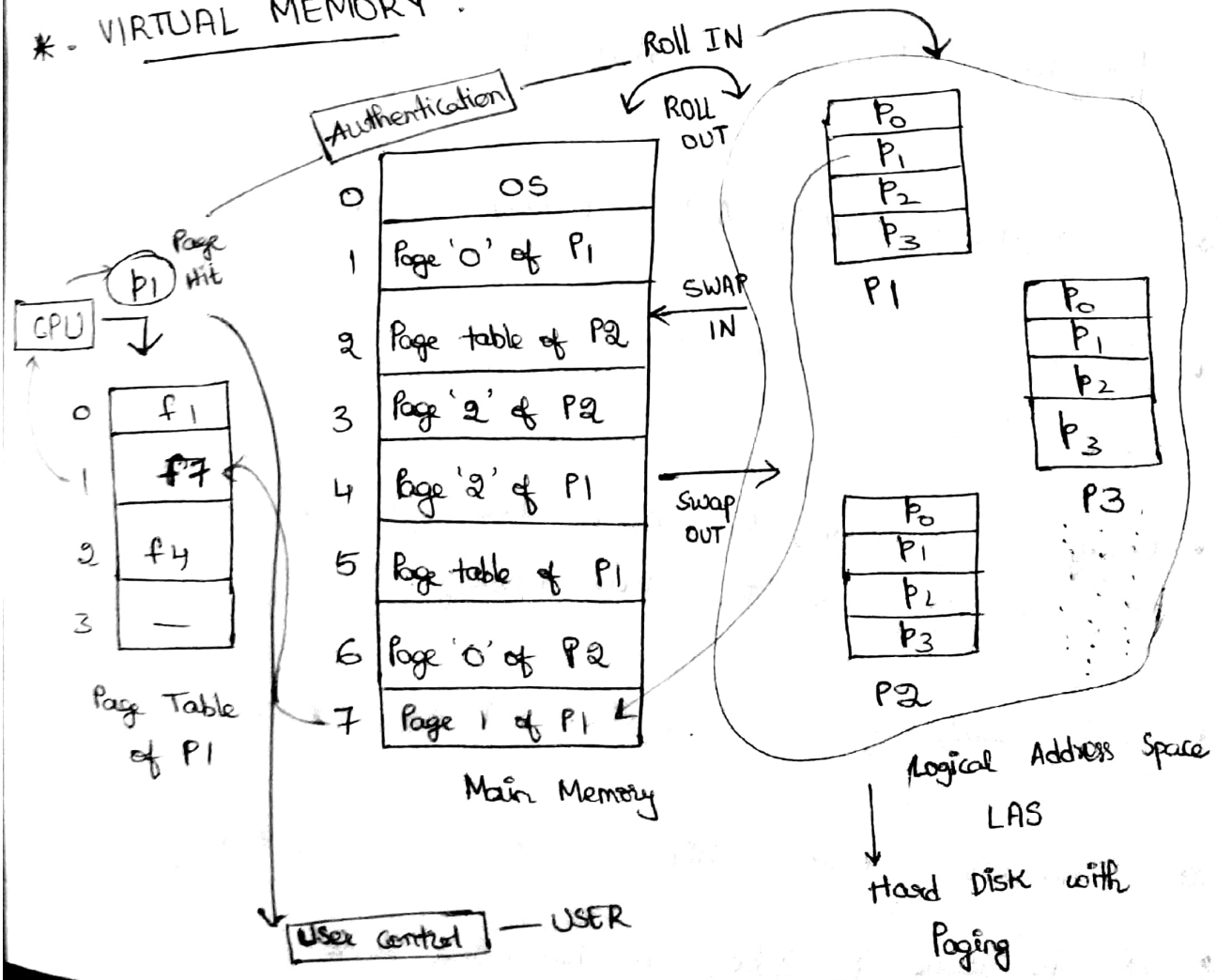
For using Pass 2, Total size = 90 + 30 + 20 + 10 = 150 KB

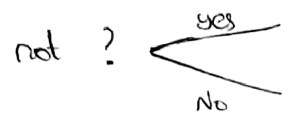
Max size (Pass1, Pass2) = 150 KB. So, if the partition is 150 KB ⟹ Pass 2 can be allocated. also Pass 1 can be.



(because only 1 pass is used at a time.)

*. VIRTUAL MEMORY :



- Degree of Multi Programming is improved.
- Page Replacement is process continued / followed.
- When Page Fault is happened, OS generates Trap & Context switching is happened from User to OS.
- Now OS checks whether US is authenticate to demand that page / or not ? 

Displays Page from LAS
Access Denied
- Effective Memory Access Time (EMAT) = P (Page fault) + (1-P)(Main)

IMP
- $EMAT = P*(PF) + (1-P)*(\text{Main Memory MA})$ *IMP

P = probability of Page Fault

1-P = probability of Page Hit

PF = Page Faullt Service Time (millsec)

Main Memory MA = Main Memory Access Time (nano sec)

---

*. TRANSLATION LOOK ASIDE BUFFER (TLB):

- It is solution that tries to reduce the effective access time.
- Being a hardware, the access time of TLB is very less as compared to the main memory.
- TLB consists of two columns. 1- Page Number 2. Frame Number

*. TRANSLATING LOGICAL ADDRESS INTO PHYSICAL ADDRESS:

- It is done by CPU is translated into physical address

step 1:

- CPU generates a logical address consisting of two parts. i.e. Page Number, Page offset.

step 2:

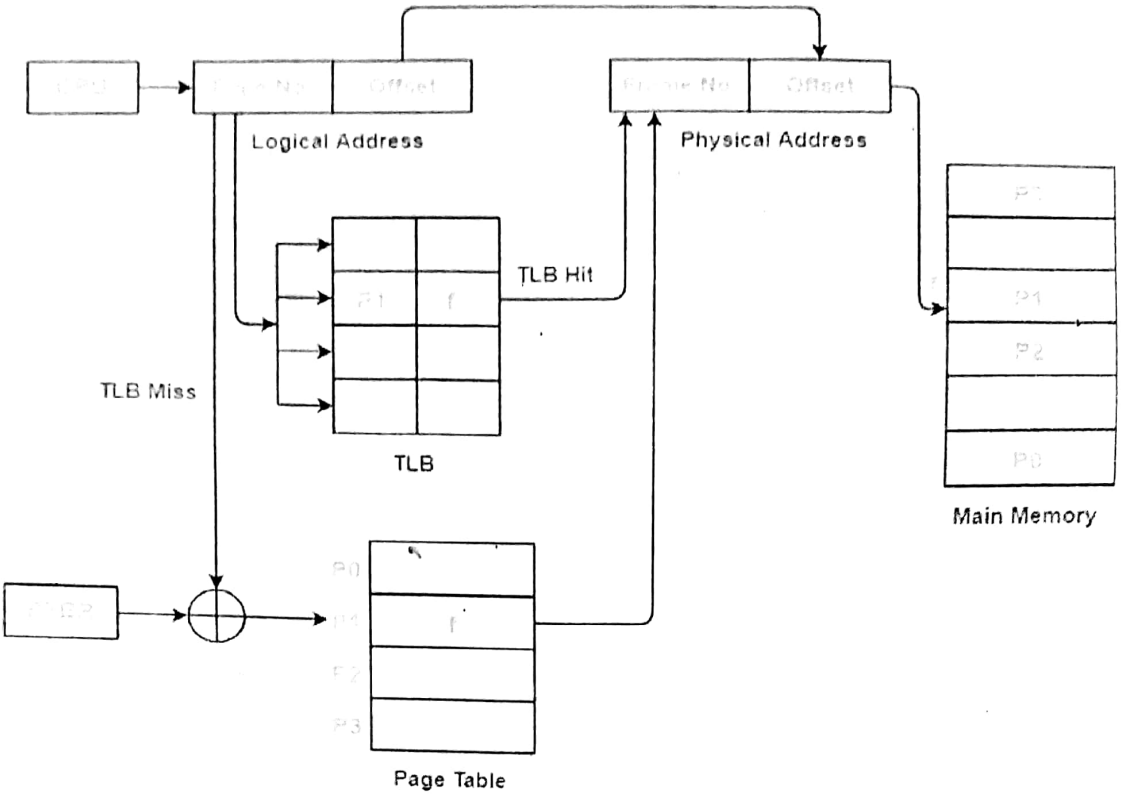
- TLB is checked to see if it contains an entry for the referred page number.
- The referenced page number is compared with the TLB entries all at once.

If there is TLB Hit or TLB miss:

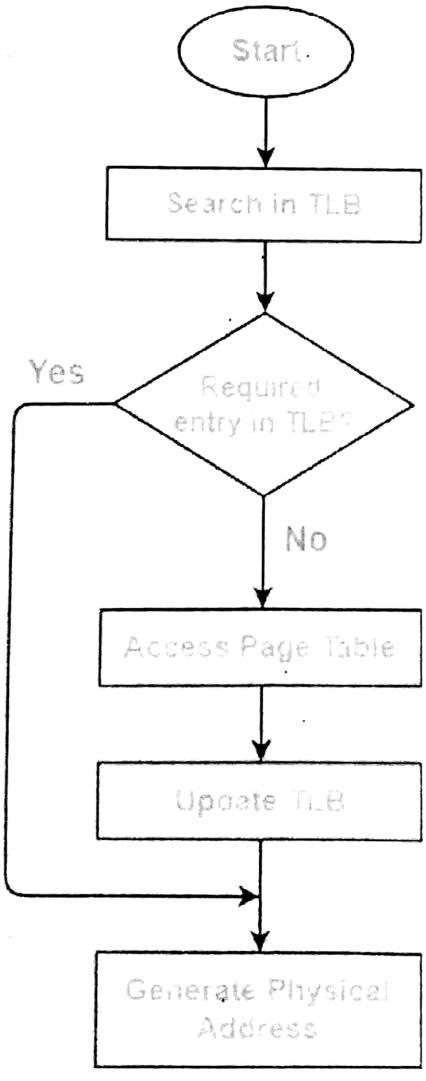
- If HIT - TLB entry is used to get the corresponding frame number for referenced frame number.
- If MISS - Page table is used to get the corresponding frame number for referenced page number. Then TLB is updated with the page number and frame number for future references.

step 3:

- After frame number is obtained, it is combined with the page offset to generate the physical address.
- Then, physical address is used to read the required word from main memory.



Translating Logical Address into Physical Address



Flowchart

* Effective Access Time :

- In a single level paging using TLB, the effective access time is given as :

$$\text{Effective Access Time} = \text{Hit ratio of TLB} \times \left(\text{Access time of TLB} + \text{Access Time of Main Memory}\right) + \text{Miss ratio of TLB} \times \left(\text{Access time of TLB} + 2 \times \text{Access Time of Main Mem}\right)$$

* Problems :

Ex: A paging scheme using TLB. TLB access time 10 ns and main memory access time taken 50 ns. What is effective memory access time (in ns) if TLB hit ratio is 90% and there is no page fault.

$$EMAT = Hit\,(TLB + MM) + Miss\,(TLB + 2MM)$$

Hit = 90% = 0.9

Miss = 1 − Hit = 0.1

TLB = 10 ns = 10^{-8} s

MM = 50 ns = 5×10^{-8} s

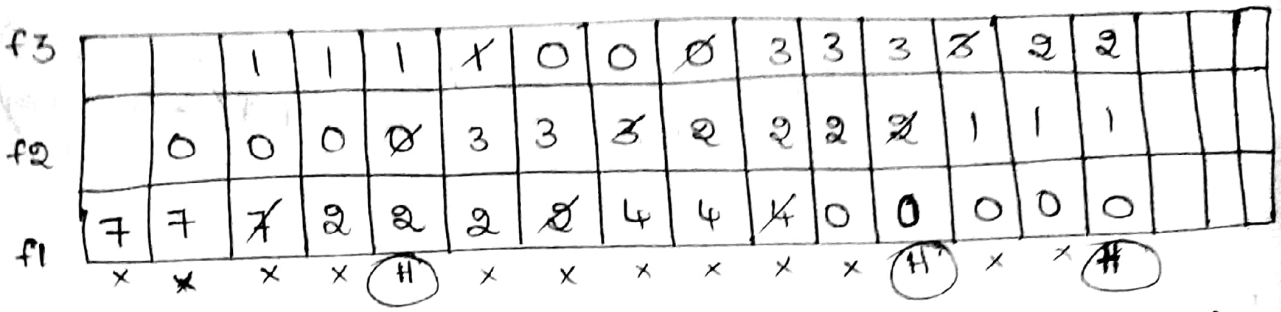
$$EMAT = 0.9\left(10^{-8} + 5\times10^{-8}\right) + 0.1\left(10^{-8} + 2\times5\times10^{-8}\right)$$

$$= 6.5 \times 10^{-8} \text{ sec}$$

$$= 65 \text{ nano sec}$$

*. PAGE REPLACEMENT ALGORITHMS : (FIFO)

Ex: Given Reference String : 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 1, 2, 0 given by CPU demand & frames = 3



- If any page number is not found in main memory, then it is Page Fault.
- First in First out page Replacement.
- Page Faults / Miss = 12
- Page Hit = 3
- Miss ratio = $\frac{\text{No. of Page Faults}}{\text{Total reference}}$ $\sim$ 80%.
- Hit ratio = $\frac{\text{No. of Hits}}{\text{Total Reference}}$ $= \frac{3}{15} = 0.2$ $\sim$ 20%.

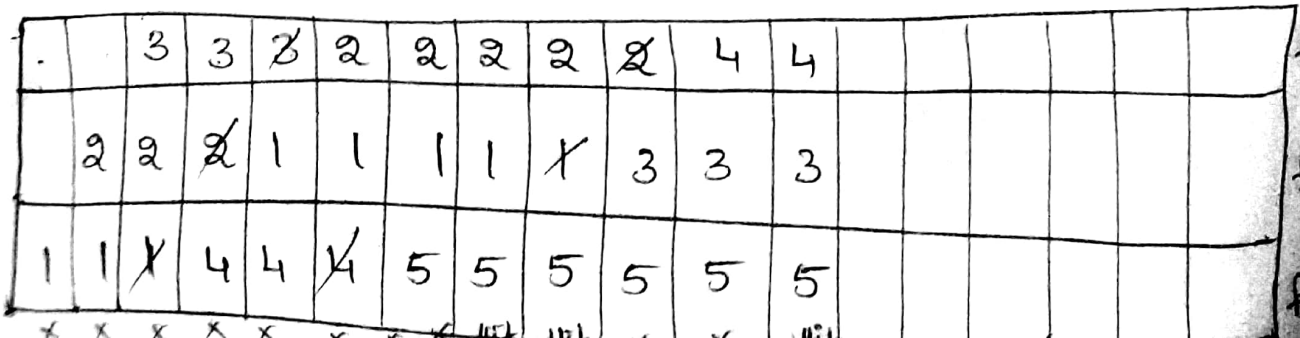
---

*. BELADY'S ANOMALY in FIFO Page Replacement :

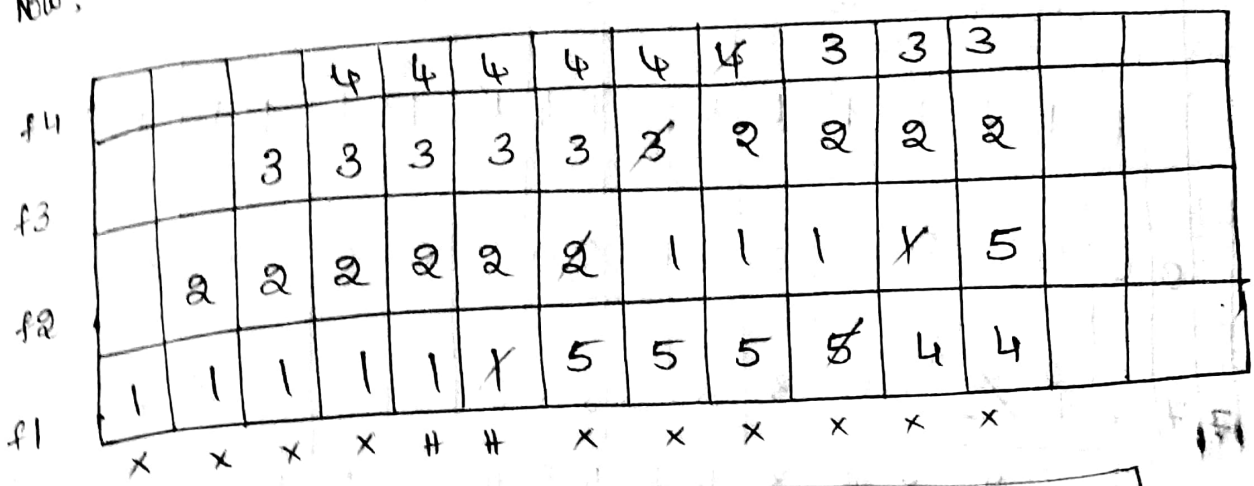
Reference string : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

No. of frames = 3

Hits = 3 Hit ratio = 25%
Faults = 9 Miss ratio = 75%



Now, If No. of frames = 4



No. of Hits = 2
No. of Page Faults = 10
Hit ratio = 16.66 %.
Miss ratio = 83.33 %.

observation :

Belady's Anomaly : In FIFO, if the no. of frames are increased and more space is allocated then there is more Page Faults & increase in Miss ratio.

---

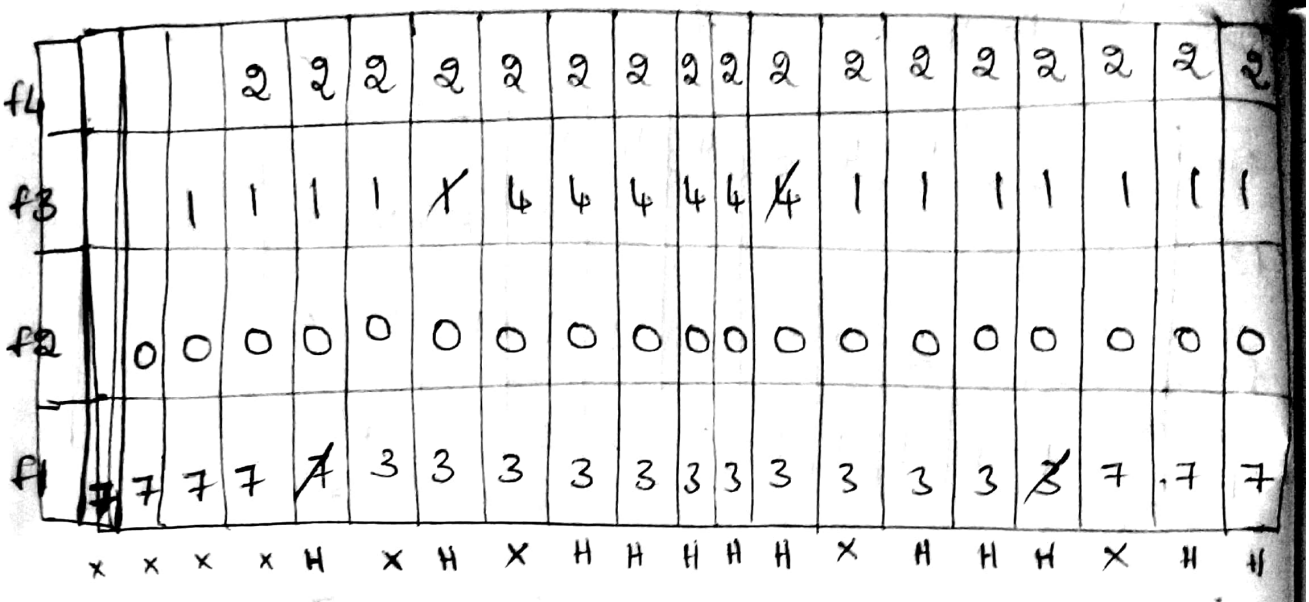
*. OPTIMAL PAGE REPLACEMENT :

- In this page replacement algorithm, if page fault occurs, we replace the page which is not used in longest dimension of time in future.

Ex :

Reference string : 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 21, 2, 0, 1, 7, 0, 1

Frames Number : 4 (f1 f2 f3 f4)



No. of Hits = 12 Hit ratio = ~~54.89~~ 60%.

No. of Miss = 8 Miss ratio = 40 %.

---

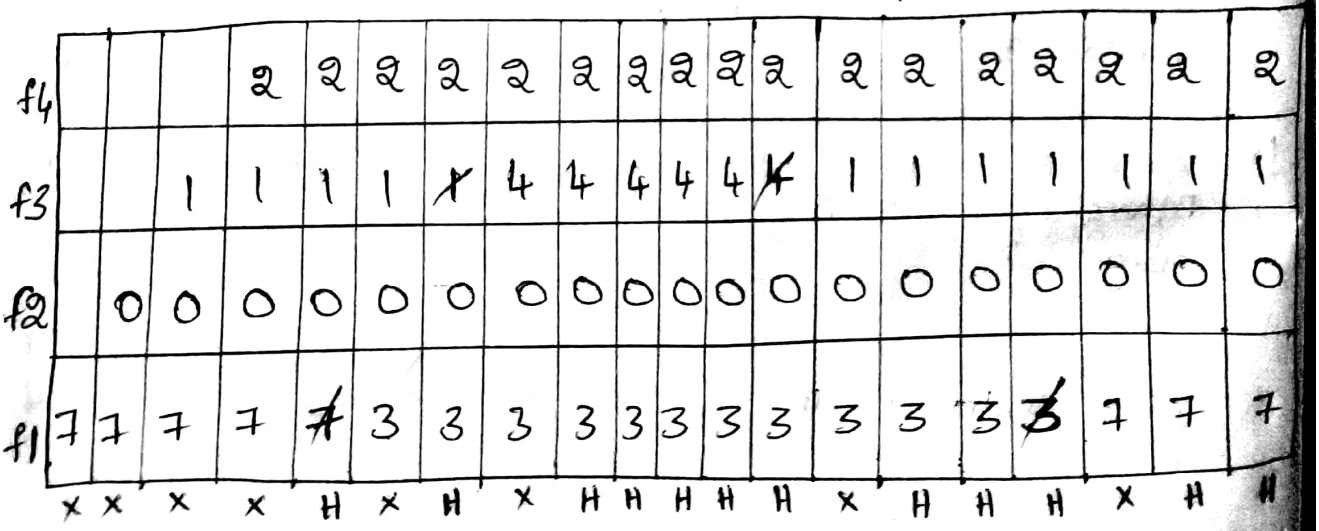
*. LEAST RECENTLY USED PAGE REPLACEMENT :

- In this algorithm, we replace the least recently used page in past.

Ex: Given Reference string : 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

No. of frames = 4

Hit = 12 ~ 60%
Fault = 8 ~ 40%



- Speed of LRU < Speed of FIFO

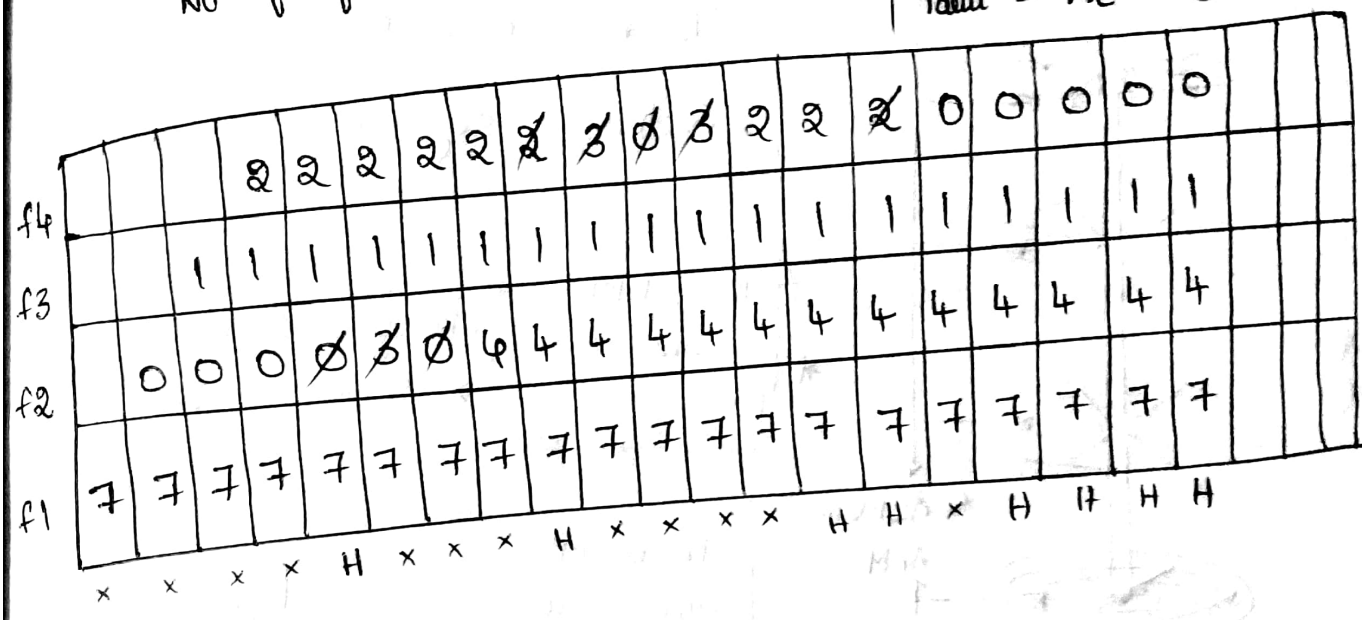
*. MOST RECENTLY USED Page Replacement :

- In this algorithm, we replace the most recently used page in past.

Ex: Given Reference string : 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

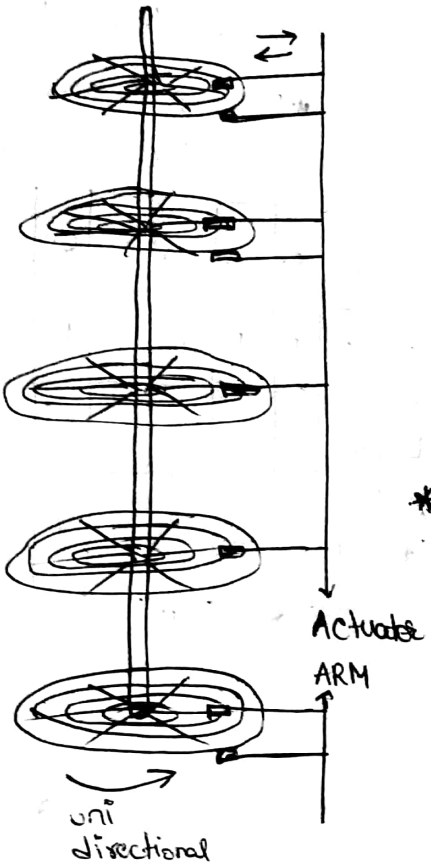
No. of frames - 4

Hit = 8 ~40%
Fault = 12 ~60%



MODULE - 5 :

* DISK ARCHITECTURE :



Platter - round shaped

Surfaces = 2 for each platter

Track = disk on a platter

Sectors = segments to store data.

* Total Disk Size = Platters x Surface x Track x Sectors x Data

* No. of bits required to represent disk size = $\log_2$ (Total Disk size)

- Platters → Surface → Track → Sectors → Data.

* DISK ACCESS TIME :

- Seek Time : Time taken by R/W Head to reach desired track.
- Rotation Time : Time taken for one full Rotation (360°)
- Rotational Latency : Time taken to reach to desired sector. (half of Rotation Time).

- 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 Relations in one sec)

- Capacity of one Track = No. of segments × Data in each segment

IMP * Disk Access Time = (Seek Time) + (Rotation latency) + (Transfer Time) + (Controller time) + (Queue Time)
(DAT)
(Controller time) ← optional
(Queue Time) ← optional .

*. DISK SCHEDULING ALGORITHM :

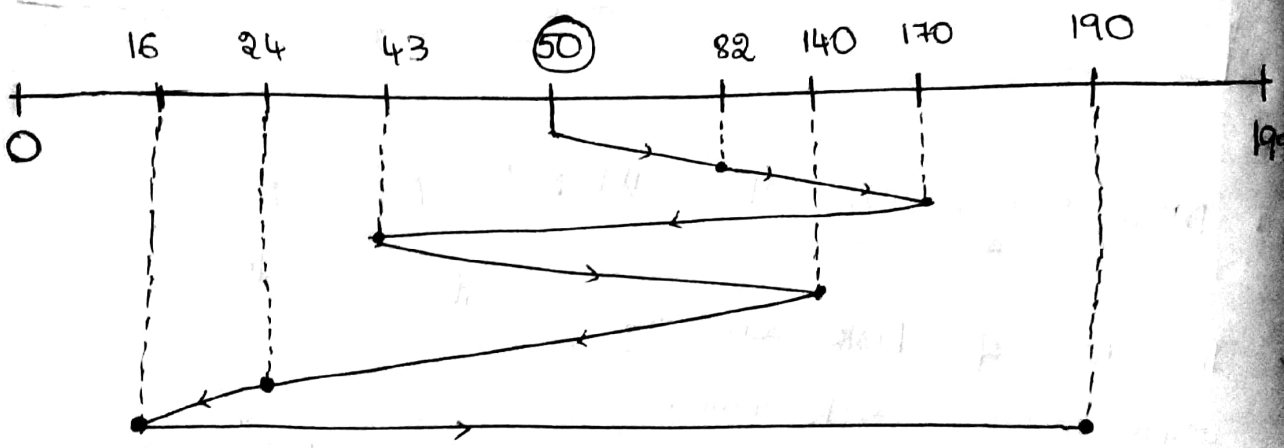
- The Goal of Disk Scheduling Algorithm is to minimize the seek time.

- Platter → Surface → Track → Sector → Data is helped by Actuator Arm. ✓

1. First Come First Serve (FCFS) .
2. Shortest Seek time First (SSTF) .
3. SCAN .
4. LOOK .
5. CSCAN (Circular SCAN) .
6. CLOOK (Circular Look) .

*. FIRST COME FIRST SERVE (FCFS) DISK SCHEDULING

Ex: A Disk Contains 200 tracks (0-199). Requests queue contains track no. 82, 170, 43, 140, 24, 16, 190, respectively. Current position of R/W head is 50. Calculate the total no. of track movements by R/W Head.



Track movements is change in directions.

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

(or)

Same direction end points

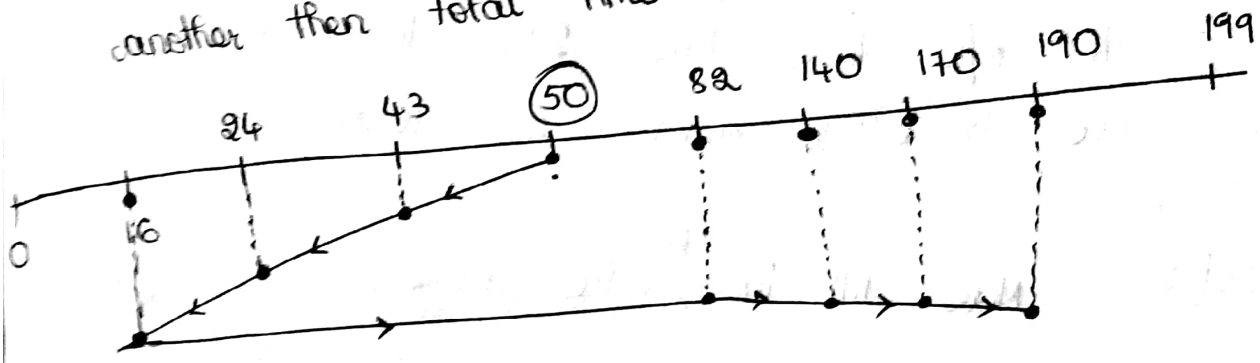
$= (170-50) + (170-43) + (140-43) + (140-16) + (190-16)$

$= 642$

- No Starvation problem. (but performance is affected)

* SHORTEST SEEK TIME FIRST (SSTF) :

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 Header is 50. Calculate the total no of tracks movements by R/W head using SSTF. Also, if R/W head takes 1 ns to move from one track to another then total time taken ——— ?



- Find the nearest value to current position of R/W header from the given queue & traverse.

$= (50-16) + (82-16) + (190-82)$

$(190-16)$ → direction change

direction change

$= 208$.

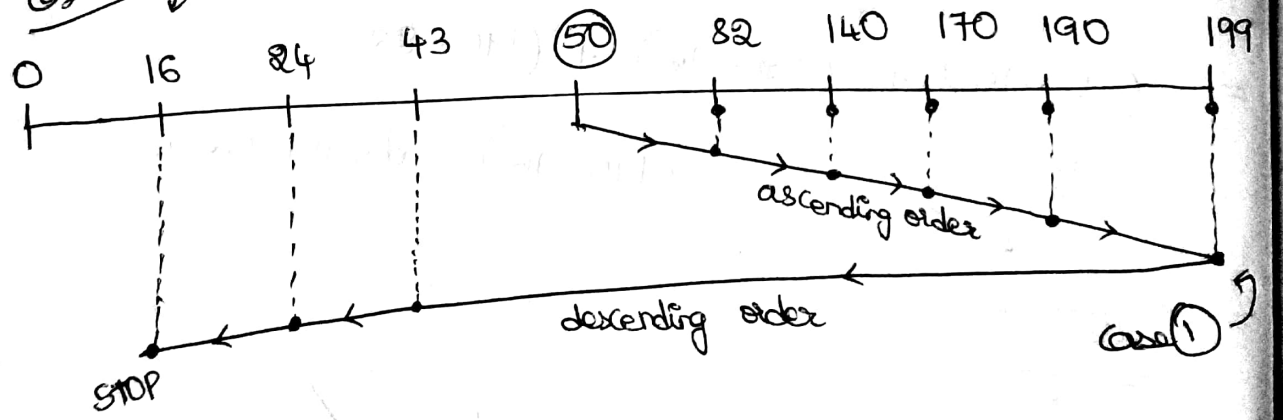
- Average Response Time is improved. ✓
- Problem is Starvation of track. ✗
- Finding nearest value of R/W head increases the complexity of the system. ✗

* SCAN ALOGORITHM IN DISK SCHEDULING :

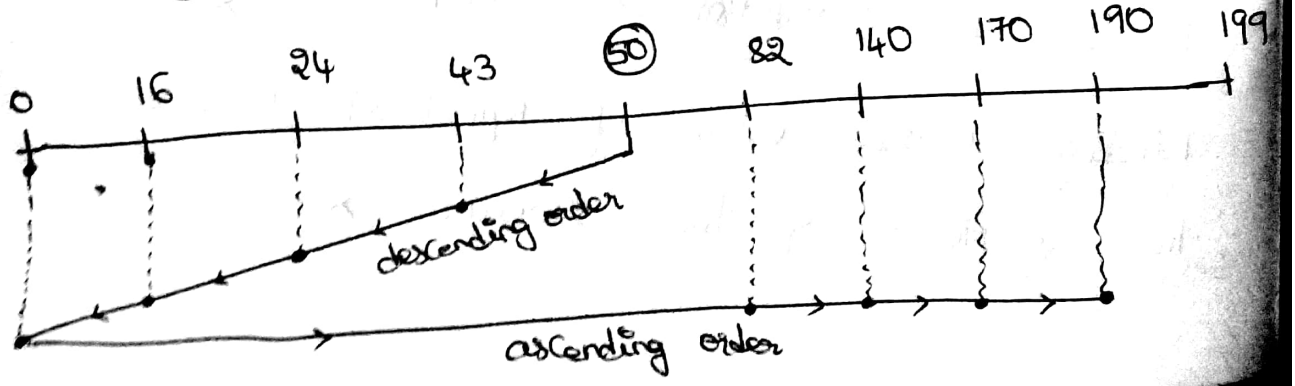
Ex: A Disk contains 200 tracks (0-199). Request Queue contains track no. 82, 170, 43, 140, 24, 16, 190 respectively. Current position of R/W header = 50. Calculate total no. of tracks movement by R/W head using SCAN? If R/W head takes 1 ns to move from one track to another then total time taken?

Sol: Move till the last value largest than given position of R/W header and then come to the values smaller than R/W header in descending order.

Case (1):



$= (199 - 50) + (199 - 16) = \underline{332}$ Case (2):



The order of which will be given in question.
And, if Ascending order then descending order,
it is case (1). If Descending order followed by
ascending order then case (2).

$= (50-0) + (190-0) = 240$

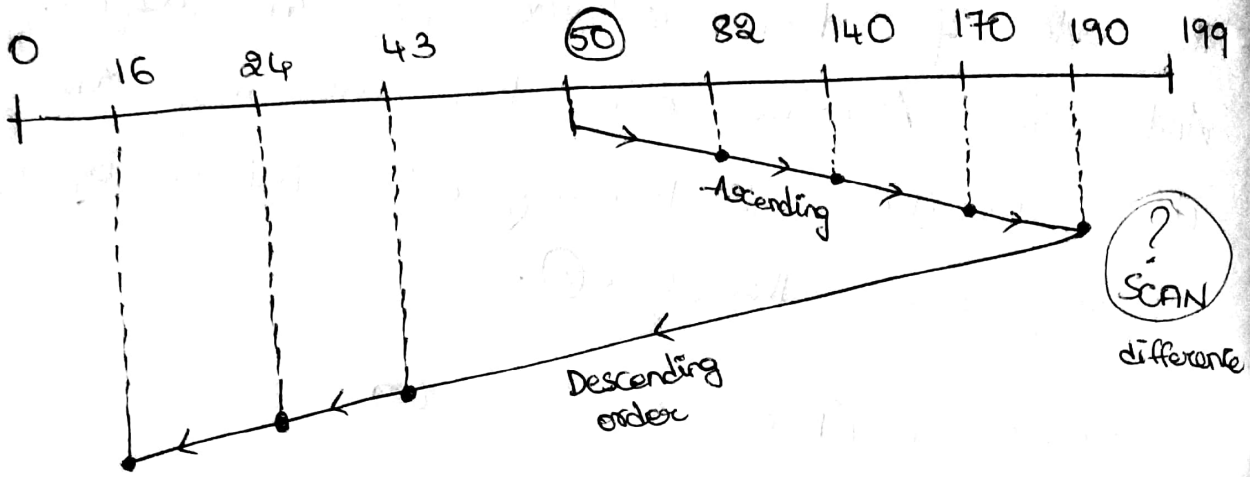
- It works like an Elevator, travels to the last point since if any input requests occurs dynamically while traversing. So, it service that.
- Problem is unidirection, if the request comes after direction change of R/W head then the performance, starvation effects.

---

*. LOOK ALGORITHM:

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 movements by R/W head using LOOK? If R/W head takes 1 ns to move from one track to another track then total time taken ____?.

- Direction is always towards larger value or smaller value. (Consider LARGER).



$= (190-50) + (190-16) = \underline{\underline{314}}$

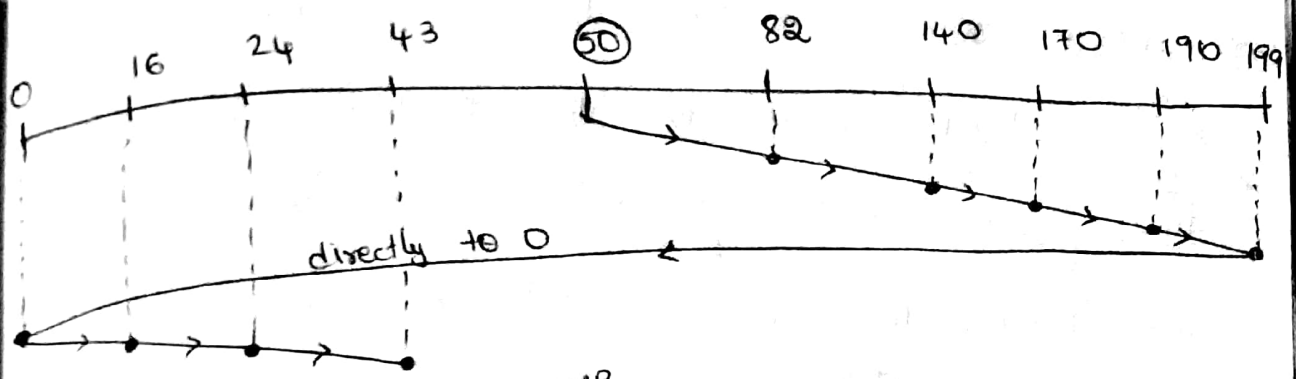
- Look algorithm compares better than scan because that extra over head loop is not there in Look. {LOOK > SCAN}

---

*. CSCAN (CIRCULAR SCAN) ALGORITHM :

Ex: A disk contains 200 tracks (0-199). Request Queue contains track no. 82, 170, 143, 140, 24, 16, 190 respectively. Current position of R/w head = 50. Calculate total no of tracks movement by R/w head using C-SCAN?

Sol: Assuming direction is towards large value.

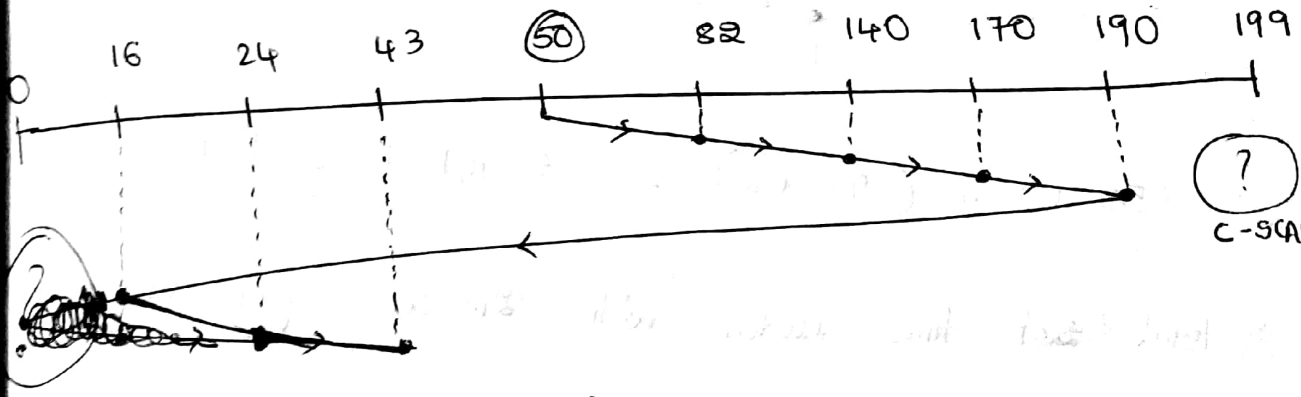


$= (199-50) + (199-0) + (43-0) = 391$ *IMP

*. GLOOK : (CIRCULAR LOOK) :

Implementing the same scenario,

Assuming direction is towards large value :

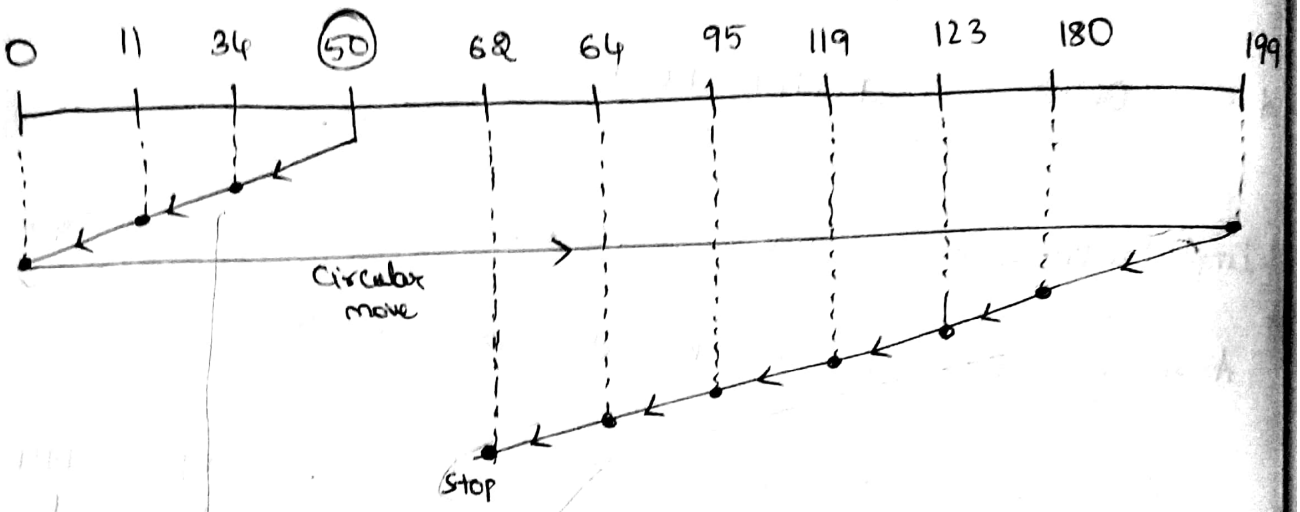


$= (190-50) + (190-16) + (43-16) = 341$ *IMP

* Questions :

- Consider the following track requests in disk queue : 95, 180, 34, 119, 11, 123, 62, 64 C-Scan algorithm is used. R/w head is at location 50. If tracks are

numbered from 0 to 199. Head is moving towards smaller track number on its servicing total seek time needed with 2m sec time to move from one track to another while servicing these requests is ____ ? Assuming moving from one end to other end is 10m sec.



$= (50-0) + (199-62) = 50 + 137 = 187$

total seek time needed with 2m sec from one track to another.

$\Rightarrow 187 \times 2ms = 374\ ms$

But also, moving from one end to another end,

$\Rightarrow 374\ ms + 10\ ms = 384\ ms$

* Consider 3 CPUS intensive processes which require 10, 20, 30, time units and arrives at 0, 2, 6 respectively. How many context switches if OS use SRTF algo? Donot count at time zero and at end?

A) 1 B) 2 C) 3 D) 4

MODULE - 6 : FILE SYSTEMS

* FILE SYSTEM :

- A software that stores and manages data in OS.
- User → Files → Folder/Directory → File System.

* ATTRIBUTES OF FILESYSTEM :

- When files are created, OS creates metadata of file having File Attributes

: operations on Files :

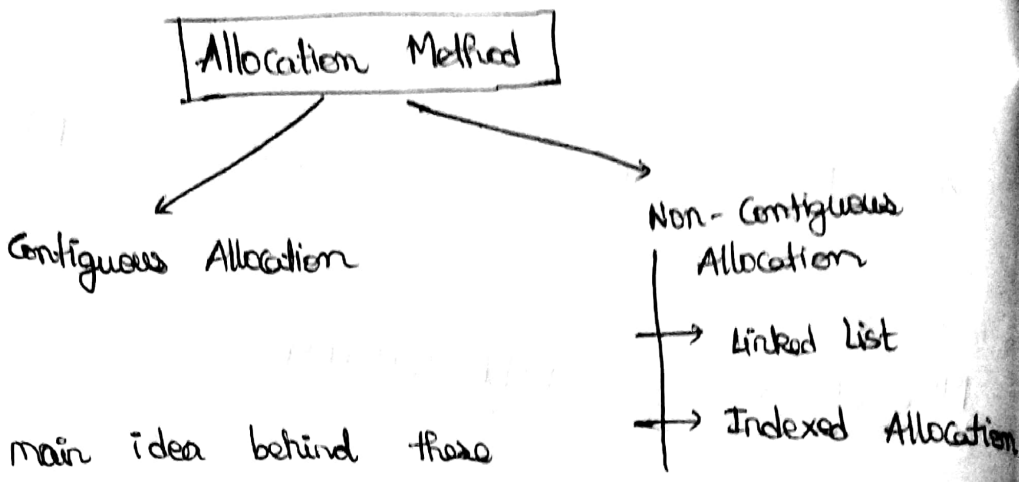
1) Creating
2) Reading
3) Writing
4) Deleting
5) Truncating
6) Repositioning

: File Attributes :

1) Name
2) extension (Type)
3) Identifier
4) Location
5) Size
6) Modified date, Created date
7) Protection / permission

✱. FILE ALLOCATION METHODS :

- logically file is divided before placing them in a Hard disk. (in physical sectors).

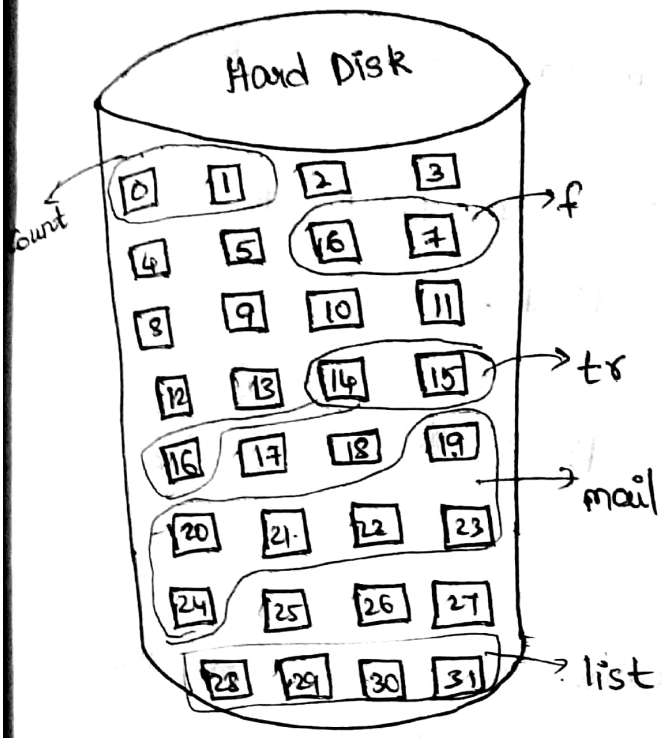


- The main idea behind these methods is to provide :
  1. Efficient disk space utilization.
  2. Fast access to the file blocks.

① CONTIGUOUS ALLOCATION :

- In this schema, each file occupies a contiguous set of blocks on the disk.
- For Example, if file requires n blocks and is given a block b as the starting location, then the blocks assigned to the file will be : b, b+1, b+2, ... b+n-1.
- The directory entry for a file with contiguous FA
  1. Address of starting block.
  2. length of allocated block.

- The file "mail" in the following figure starts from the block 19 with length = 6 blocks. Therefore it occupies 19, 20, 21, 22, 23, 24 blocks.



File	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

DIRECTORY

Advantages :

- Both the sequential & Direct Accesss are supported here
- The address of kth block of file which starts at block b is obtained as (b+k).
- Extremely fast because no. of seeks are minimum.

Disadvantages :

- Suffers from both internal & external fragmentation.
- Inefficient in terms of Memory utilization.
- Increasing file size is difficult because it depends on the availability of contiguous memory at a particular instance.

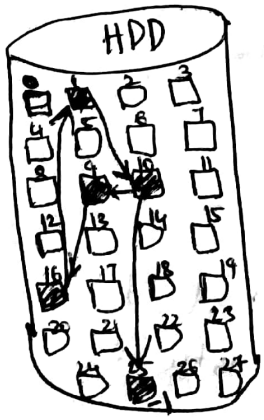
② LINKED LIST ALLOCATION :

- In this schema, each file is a linked list of disk blocks which need not be contigeous. The disk blocks can be scattered anywhere on disk.

- The directory entry contains a pointer to the starting and ending file block.

- Each block contains a pointer to the next block occupied by the file.

- The file 'jeep' in following image shows the blocks are randomly distributed. The last block (25) contains -1 indicating a null pointer and doesnot point to any other block.



file	start	end
jeep	9	25

Directory

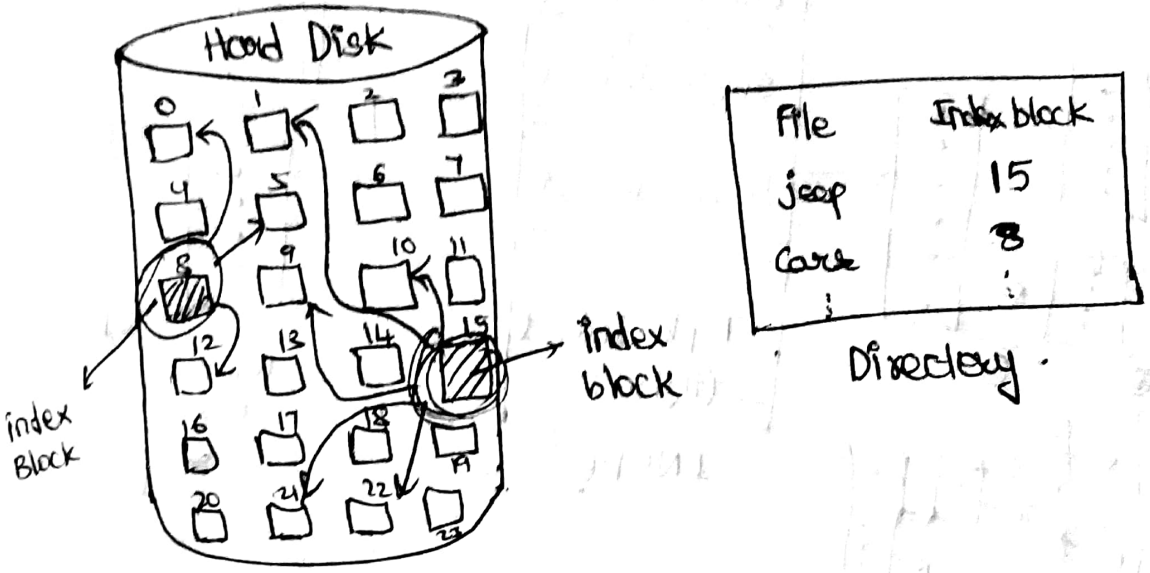
1 → next block

-1 → no block

- + Flexible in file size.
- + External Fragmentation
- - Pointers are overheaded.
- - No direct access.
- - Accessing is time consuming.

③ INDEXED FILE ALLOCATION :

- A special block known as "index block" contains the pointers to all the blocks occupied in file
- Each file has its own "index block".



Note: For files that are very large, single index block may not be able to hold all the pointers

Following mechanisms can be used to resolve this.

1. Linked Scheme :

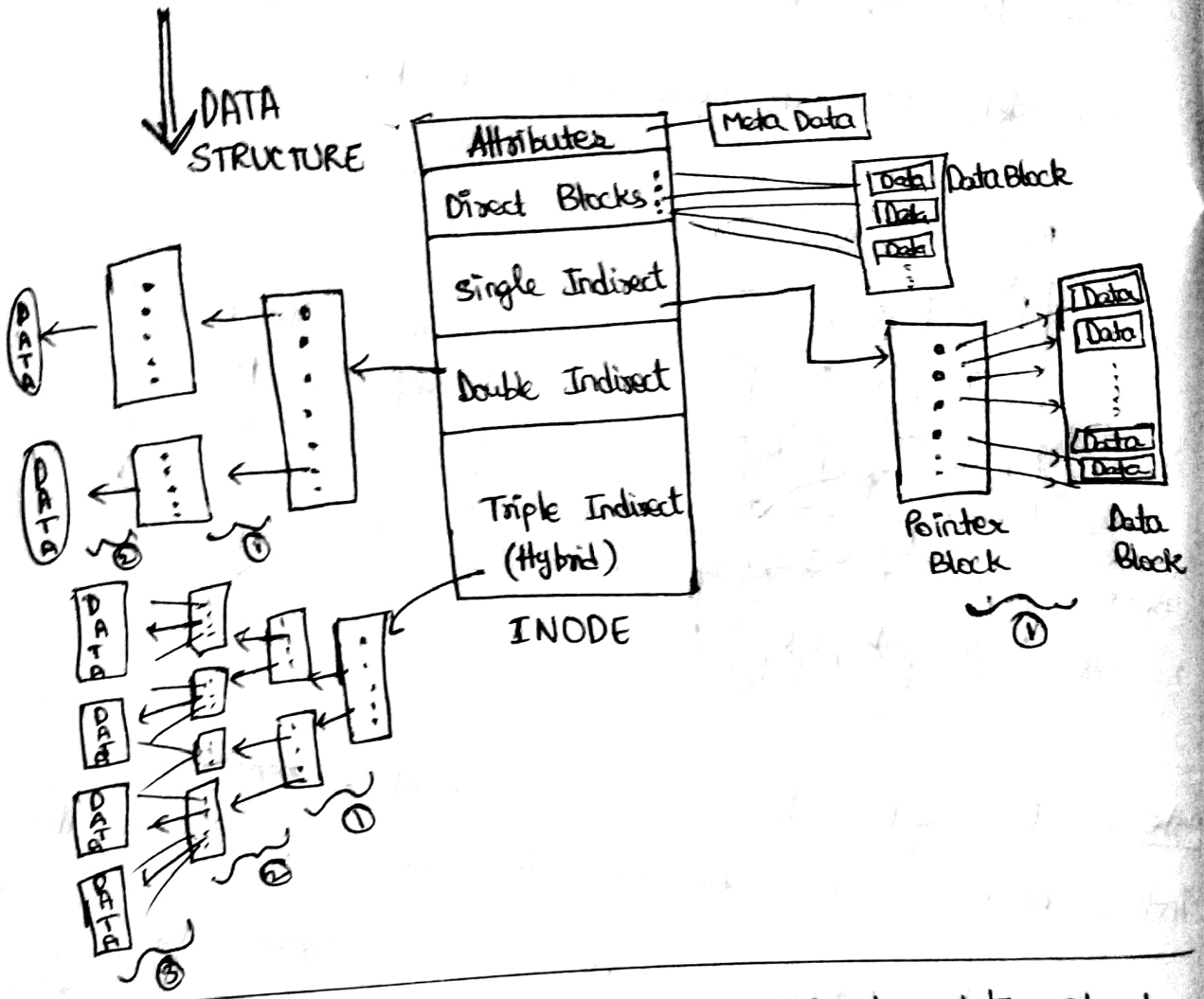
- It links two or more index blocks together for pointers
- Every index block would then contain a pointer or the address to the next index block.

2. Multilevel indexing :

3. Combined scheme :

* UNIX INODE STRUCTURE :

- Unix OS uses INODE file system (I - indexed)



Ex: A file system uses Unix inode data structure which contain 8 direct block addresses, one indirect block, one double and one triple indirect block. The size of each disk block is 128 B. and size of each block address is 8B. Find the maximum possible file size?

Sol: Total no of pointers in one disk block = $\frac{\text{size of 1 Disk Block}}{\text{size of each Block}}$

$= \frac{128\ B}{8\ B} = 16 = 2^4$

Total pointers (Addresses) in INODE $= \{ 8 + 16 + (16)^2 + (16)^3 \}$

(8 → direct, 16 → single Indirect, $(16)^2$ → double Indirect, $(16)^3$ → Triple indirect)

$= 8 + 16 + 256 + 4096 = 4376$

Maximum possible size of file $=$ (Total Pointers in INODE) $\times$ (size of each block)

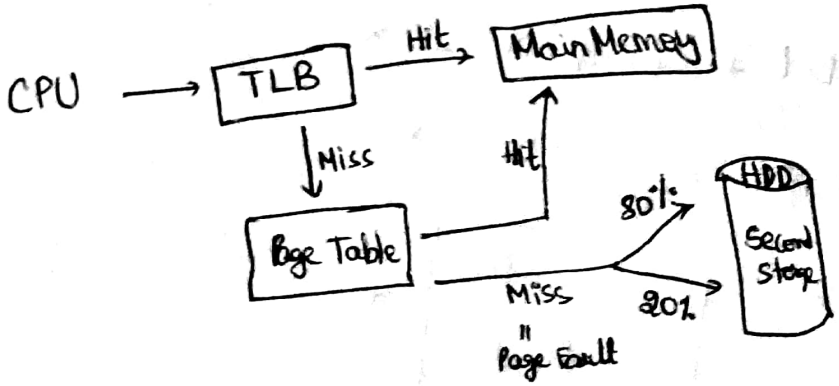
$= 4376 \times 128 \approx 560128B$

Max file size = 547 KB

PROBLEMS :

- Consider a paging system uses 1 level page table. residing in M/M and a TLB M/M access time 100 ns. TLB lookup takes 20ns. Each page transfer takes 5000 ns to/from Disk. TLB Hit ratio is 95%, Page fault rate = 10%. Assume that for 20% of total page fault a dirty page has to be written back to disk before required page is read in from disk. TLB update time is negligible. Find the avg memory Access time ?

Sol:



$= 95\%$ of TLB Hit $+$ 5% TLB Miss $\left\{ \left[\text{TLB time} + \text{Page Table Time} + \text{M/M} \right] \times 90\% \text{ no page fault} \right.$

$+ 10\% \left(\left[80\% \text{ of Not Dirty} \left(\text{TLB time} + \text{Page Table Time} + \text{Extra time} \right) \right] \right.$

$\left. \left. + \left[20\% \text{ of Dirty Page} \left(\text{TLB} + \text{PT} + \text{Extra time} + \text{Extra time} \right) \right] \right\}$

$= 0.95 (20+100) + 0.05 \left[\left((20+100+100) 90\% \right) + \right.$

$\left. 0.10 \left[(20+100+5000) \frac{80}{100} + 20\% \left(\quad \right) \right. \right.$

$(20+100+5000+500) \Big] \Big]$

$= \sim 155$ ns

- Page fault rate experienced $= P$

  Average time to access M. Memory $= M$ (page hit)

  Avg time to access M. Memory if page fault $= D$

  Total avg time to access Memory $= X$

$\Rightarrow \quad P \cdot D + (1-P) M = X$

$P \cdot D + M - PM = X$

$\Rightarrow \quad \boxed{P = \frac{X-M}{D-M}}$ * IMP

*. Let the page fault service time be 10 ms in a computer with avg memory access time being 20 ns. If one page fault is generated for every 10^6 memory accesses, what is the effective access time for the memory?

Sol:
$$\left(\frac{1}{10^6} \times 10\,ms\right) + \left(1 - \frac{1}{10^6}\right) 20\,ns$$

$$= (10^{-6} \times 10 \times 10^{-3}\,s) + (1 - 10^{-6}) \times 20 \times 10^{-9}\,s$$

$$= 10\,ns + 20\,ns - 2 \times 10^{-14}\,s \approx \underline{\underline{30\,ns}}$$

---

*. If an instruction takes i μs and a page fault takes an additional j μs, the effective instruction time if on an average a page fault occurs every k instructions is:

Sol:
$$= (k-1)\,i + 1(i+j) = ik - i + i + j$$

$$= ik + j = \text{total time}$$

$$\text{effective time} = \frac{\text{total time}}{\text{instructions}} = \frac{ik+j}{k} = \underline{\underline{i + \frac{j}{k}}}.$$

---

*. Suppose that the time to service a page fault is on the average 10 ms, while a memory access takes 1 μs. Then 99.99% ~~the~~ hit ratio results in average Memory access time of ____

Sol:

$= (99.99\% \text{ of } 1\,\mu s) + (100 - 99.99\% \text{ of } 10\,ms)$

$= (0.9999 \times 1) + (0.001 \times 10 \times 10^3) \approx 1.9999\,\mu s$

---

* A demand paging system, with the page table held in registers, takes 5ms to service a page fault if an empty page is available or if the page to be replaced is not ~~directly~~ dirty. It takes 15ms if the replaced page is dirty. Memory Access time is 1 μs. Assume we want an effective access time of 2 μs and that the page to be replaced is dirty 60% of the time. What will be the approximate page fault rate to meet this requirement?

Sol:

Let page fault rate = p

E.M.A.T = 2 μs

$= (1-p)\,1\,\mu s + p\,[0.6 \times 15\,ms + 0.4 \times 5\,ms]$

- $(1-p)$ ↓ page hit
- 1 μs ↓ MM
- p ↓ page fault
- 0.6 ↓ dirty page probability
- 15 ms ↓ dirty page time
- 0.4 ↓ not dirty page probability
- 5 ms → time

$\Rightarrow = (1-p) + p\,[0.6 \times 1500\,\mu s + 0.4 \times 5000\,\mu s]$

$2 = 1 - p + p\,[9000 + 2000] \Rightarrow p = 0.009091735\%$

$\approx 0.01\%$

11

A hard disk has the following parameters

- No. of tracks = 500
- No. of sectors / track = 100
- No. of bytes / sector = 500

Time taken by the R/W head to move from one track to another adjacent track = 1 m sec

Rotational speed = 600 RPM

what is the average time taken for transfering 250 bytes from the disk.

sol: **Worst Case**

No. of seeks needed to reach = 500 − 1 = 499

Seek Time taken = 499 × 1 msec = 499 m sec

Best Case

No. of seeks needed to reach = 0 (both R/W head & data are on same track)

Seek Time taken = 0 m sec

$$\text{Average Case} = \frac{\text{Best Case} + \text{Worst Case}}{2}$$

$$\Rightarrow \frac{499+0}{2} = 249.5 \text{ m sec} = \text{Average}^{\text{Seek}} \text{ time taken}$$

Rotational speed = 600 RPM

600 rotations ⟶ 60 sec (1 minute)

1 rotation ⟶ $\frac{60}{600}$ = 0.1 sec = 100 m sec

Average rotational delay = $\frac{0+100}{2}$ = 50 m sec

- Average seek time = 249.5 m sec
- Average Rotational delay = 50 m sec

. Total Average time to retrieve data = 249.5 m sec + 50 m sec
= 299.5 m sec

- Capacity of track = no. of sectors x no of byte/sector
= 100 x 500 = 50000 bytes

No. of times R/W head is traversing track per sec

$= \left(\frac{600}{60}\right) * 50000 \text{ bytes} = 5 \times 10^5 \text{ bytes/sec}$

↓
Transfer Capacity of the disk

- Per second the disk 5×10^5 bytes

5×10^5 bytes → 1 sec
250 bytes → ? (x)

$$x = \frac{250 \text{ bytes} \cdot \text{sec}}{5 \times 10^5 \text{ bytes}} = 0.5 \text{ m sec}$$

- Avg time taken to transfer = (Avg seek time) + (Avg rotational delay) + (time required for transfering 250B)

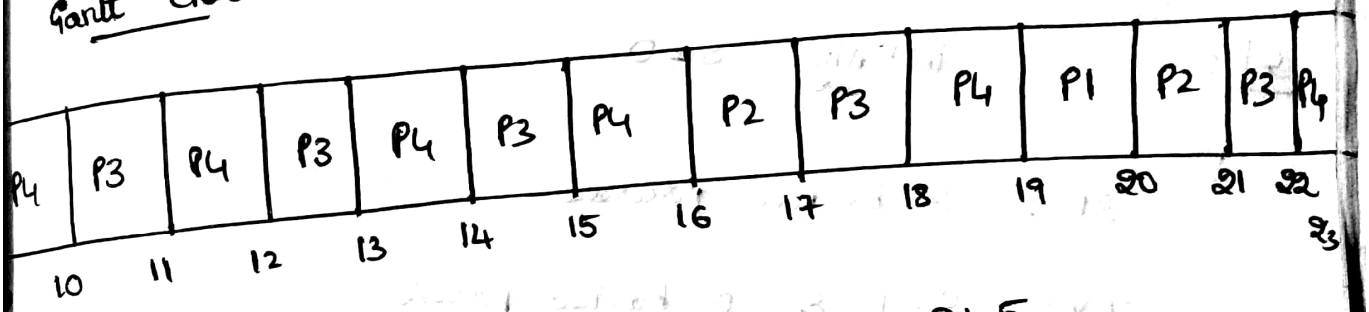
= 249.5 m sec + 50 m sec + 0.5 m sec
= 300 m sec (or) 0.3 sec

*. Longest Remaining time First

Given,

P#	AT	BT	CT	TAT	WT
P1	0	1	23	23	22
P2	0	2	21	21	19
P3	0	5	22	22	17
P4	0	15	20	20	5

Gantt chart :



Avg WAT = 15.75 , Avg TAT = 21.5

*. A system has six magnetic tape drives. 'N' concurrent processes are competing to acquire them. Every process may need upto TWO drives. The maximum value of "N" that will never put system in deadlock is:

Sol:

For a system to be deadlock free,

* Sum of max need of processes < (No. of process) + (No. of resources)

No. of processes = N

No. of resources = 6

$2n < n+6$

$2n - n < 6 \rightarrow n < 6$ or max value of $n = 5$

---

*. Suppose 's' be a binary semaphore with initial value 0. Consider that no blocked processes exist in the system initially. Then how many process will enter into blocked state after accomplishing the following set of P, V operations in sequence:

3P, 7V, 9P, 14V, 15P.

Sol: initially $S = 0$

3P: 3 blocked processes

7V: $S = 1$ & 0 blocked process

9P: 8 process blocked

14V: $S = 1$ & 0 blocked process

15P: $S = 0$ & 14 blocked process

∴ No. of process into blocked state = 14

---

*. Let a computer has a 64MB RAM with a 32-bits logical address space. Assume that a page is of size 4 KB. With these given info, what will be the approximate size of page table?

No. of entries in page table = Virtual address space size / Page size

$4\,KB = 2^{12}\,B$

$64\,MB = 2^{26}\,B$

$\Rightarrow = \frac{2^{32}}{4KB} = 2^{20}$

No. of pages $= 2^{20}$

No. of bits required to address the 64 MB ($= 2^{26}$) physical memory $= 26$

No. of page frames $= \frac{64\,MB}{4\,KB} = \frac{2^{26}\,B}{2^{12}\,B}$

$= 2^{26-12} = 2^{14}\,B$

Each table entry will have 14 bits

Total size of page table = (No. of pages) × (each table entry)

$= 2^{20} \times 14$ bits $\approx$ 2 MB

---

*. A Java Application loads 100 system defined routines during the startup. The loading of each such routine needs one disk access. The average seek time for R/W head to random a track, is 10 msec The rotational speed of disk is maintained by 6000 RPM. Given these info, estimate how long will it take to load all the 100 routines. Assume that the routines are randomly distributed in disk space.

Sol :

Avg. seek time = 10 m sec

Avg rotational latency = $\frac{1}{2}$ (total time for 1 rotation)

6000 rotations ⟶ 60 sec

1 rotation ⟶ $\frac{60}{6000}$ sec = 10 m sec

Avg rotational latency = $\frac{1}{2}$ (10 m sec) = 5 m sec

∴ | Avg disk access time = seek time + rotational latency |

⇒ = 10 m sec + 5 msec = 15 m sec

For 100 libraries ⇒ 15 m sec × 100 = 1500 m sec

---

*. Let the page fault service time be 10 ms in a computer with average memory access time being 20 ns. If one page is generated for every 10^6 memory access, what is the effective access time for memory?

Sol : Let p be the page fault rate

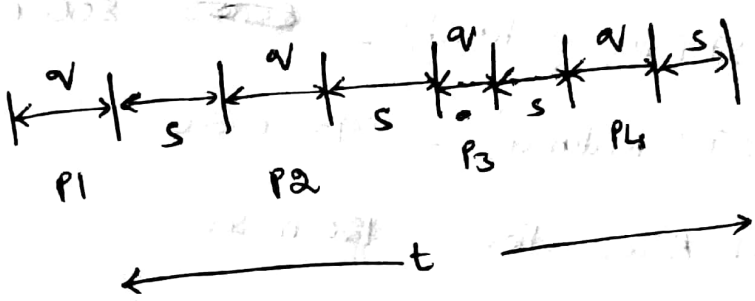
Effective Memory Access Time = p * (page fault service time) + (1-p) * (Memory access time)

⇒ = $\left(\frac{1}{10^6} \times 10 \text{ msec}\right) + \left(1 - \frac{1}{10^6}\right)(20 \times 10^{-6} \text{ msec})$

= 30 nano sec (approx)

*. Consider a multi-programming system where 4 processes P1, P2, P3, P4 share the CPU in Round-Robin ~~CPU~~ mode. Let the context switch time be 1 time unit. What is the time quantum 'q' such that every process will get its turn at the CPU in a span of every 10 time units?

Sol:



~~2q + 4s = 10~~

~~2q + 4 = 10~~

~~2q = 6~~

~~q = 3~~

$\Rightarrow$ here $t = 10$

$s = 1$

$n = 4$

~~(P2 P3 P4 P1)~~

~~q~~

$(n-1)$ process take n context switching

$(n-1) \cdot q + n \cdot s = t$

$$q = \frac{t - ns}{n-1}$$

$$\Rightarrow q = \frac{10 - 4(1)}{3}$$

$$q = 2$$

---

*. Consider a computer system that uses demand paging architecture. It takes 800 nano seconds to serve a page fault if either an empty frame is available or the page to be replaced is not dirty. It takes 950 n sec to serve a page fault if the page to be replaced is dirty.

Assume the average Memory access time is 120 n sec. Further it is estimated that 85% of time, the page to be replaced is dirty. What will be the approximate efficient Memory access time if the page fault rate is 0.2 ?

Sol :

PFST (not modified) = 800 n sec

PFST (modified) = 950 n sec

MM Access time = 120 n sec

85% of Page to be replaced is modified.

$p = 0.2$, EMAT = ?

$1-p = 0.8$

Let page fault rate = p

$$EMAT = \left[(0.2)\left[(0.85 \times 950 \text{ n sec}) + (0.15 \times 800 \text{ n sec})\right]\right] + \left[(1-0.2) \times 120 \text{ n sec}\right]$$

$$= 0.2\left[807.5 \text{ n sec} + 120 \text{ n sec}\right] + 0.8\left[120 \text{ n sec}\right]$$

$$= 0.2\,(927.5) + 0.8\,(120) = 185.5 + 96$$

$$= 281.5 \text{ n sec}$$

*. In a multi-programming system, there are two process P1, P2 in ready state. P1 uses 80% of time in executing CPU bound instructions, and remaining 20% of its time in performing I/O. P2 uses 90% of time in executing CPU bound instruction and remaining 10% of its time in performing I/O. What will be the CPU utilization rate of system?

Sol:

	CPU Execution	Non-CPU
P1	0.8	0.2
P2	0.9	0.1

For a multi process system,

CPU utilization = $\left[1 - \text{(product of all)}(\text{non-cpu time})\right] \times 100$

$= \left[1 - (0.2 \times 0.1)\right] \times 100 = (1 - 0.02) \times 100$

$= 98\%$