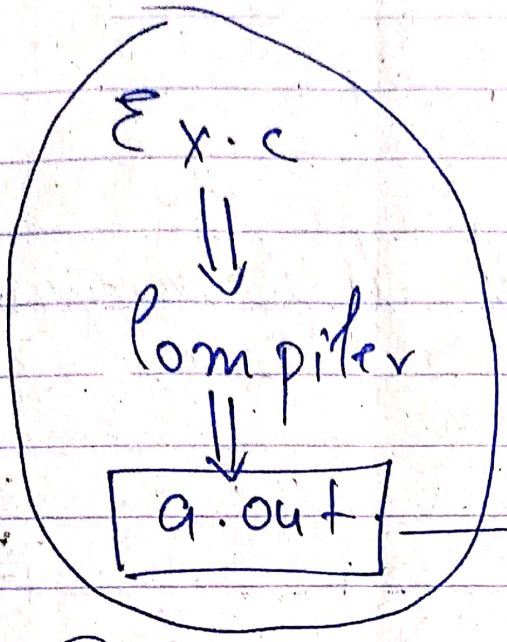
Process Management

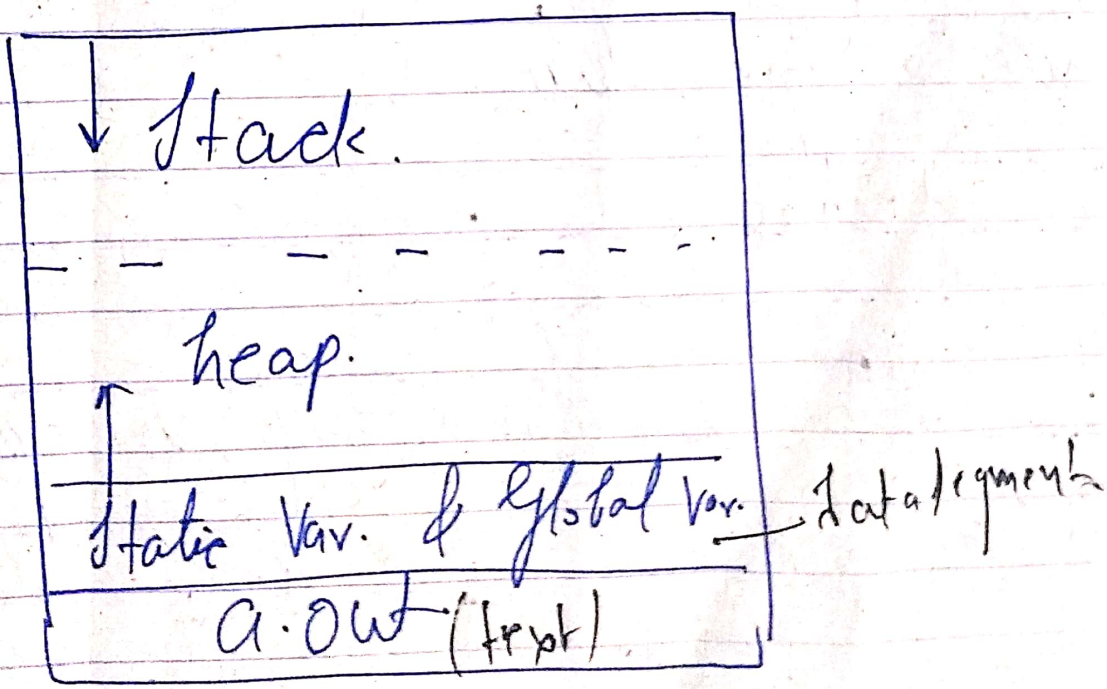


Primary memory → RAM

Secondary " → Hard disc.

Process ⇒ Body

Program ⇒ Soul.



Attributes

i) Process id (no.)

ii) Program counter (next inst. to be executed)

iii) Process state

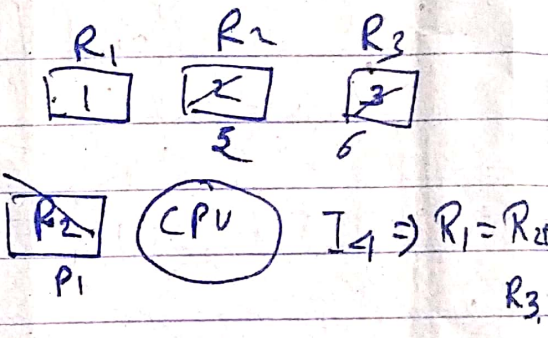
iv) Priority.

v) General purpose register

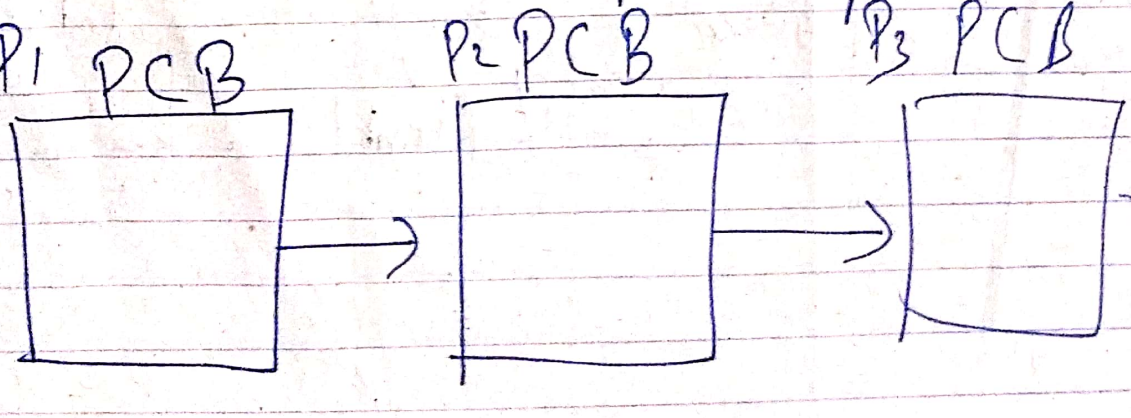
vi) List of open files

vii) List of open devices

viii) Protection.



In PCB, all these info is present.



in form of linked list.

States of a process

1) New → When picked in RAM (SM)
2) Ready → In Main memory.
3) Run → MM
4) Block or wait (for I/O again must continue)
5) Termination or completion. (PCB get deleted)

6 → Suspend ready (Move process on SM if higher priority came).

7 → Suspend wait or suspend block (Suspend waiting process & place them in SM)

Multiprogramming

- (With Preemption) (By context switch)
  - Multitasking
  - Time sharing
- Without Preemption

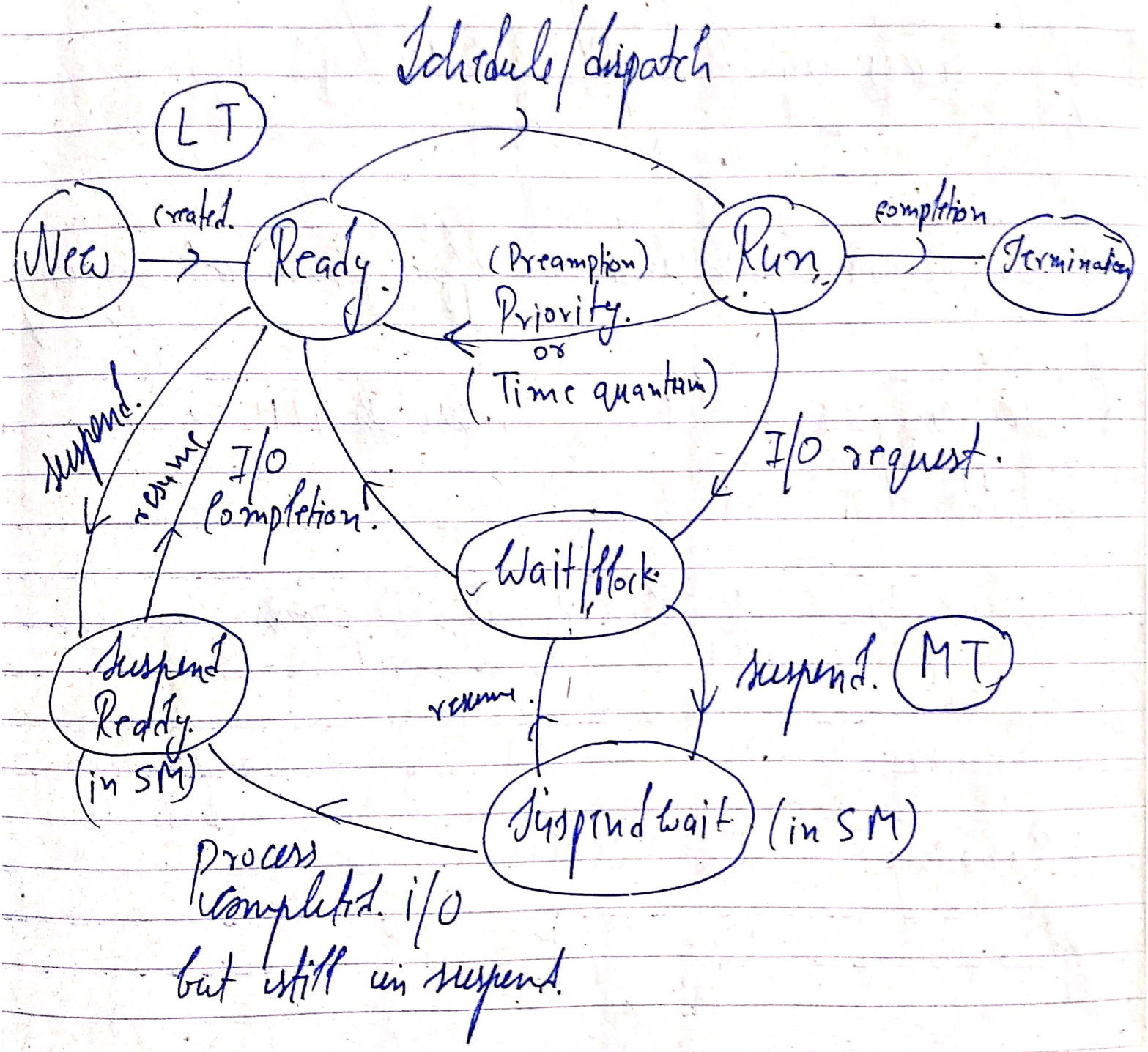
Operation on process

1) Creation
2) Scheduling
3) Executing
4) Killing / Delete

Doubt: wait to run

long term scheduler

ST → dispatcher ← to context switch.



1) Long term Scheduler → how many process to create.

2) Short term Scheduler → Next process to be executed.

3) Medium term Scheduler

Degree of multiprogramming → Max^m no. of Process in Ready. | By L.T.

* Long term Schedular → affect v. much efficiency

* Swapping ↔ Taking process b/w MM ⇔ SM.

Ex In N CPU Processor & M process, then.

	Min	Max	
Ready	0	M	No. of process.
Running	0	N	
Block.	0	M	

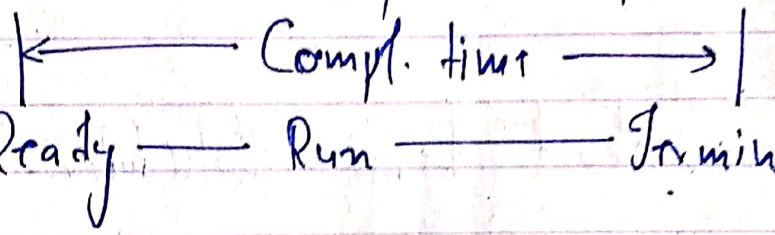
I/O → Blocked

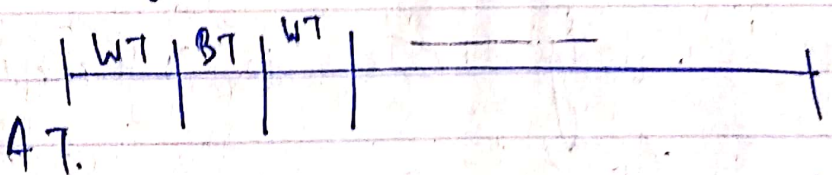
Important parameter of process

1) Arrival time → When Process becomes ready or in ready queue.

2) Burst Time → CPU Time required to get finish.

3) Completion time → When process finishes.



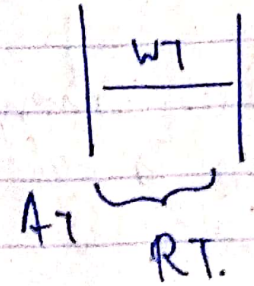


TAT = CT − AT = BT + WT. | if we ignore I/O

4) Turn around Time

5) Waiting Time ⇒ WT = TAT − BT.

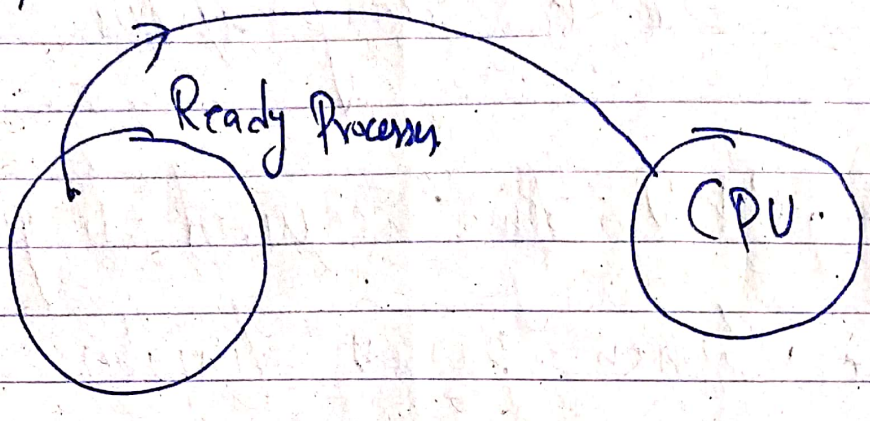
6) Response Time.



(?) → Diff. b/w Completion & TAT. :

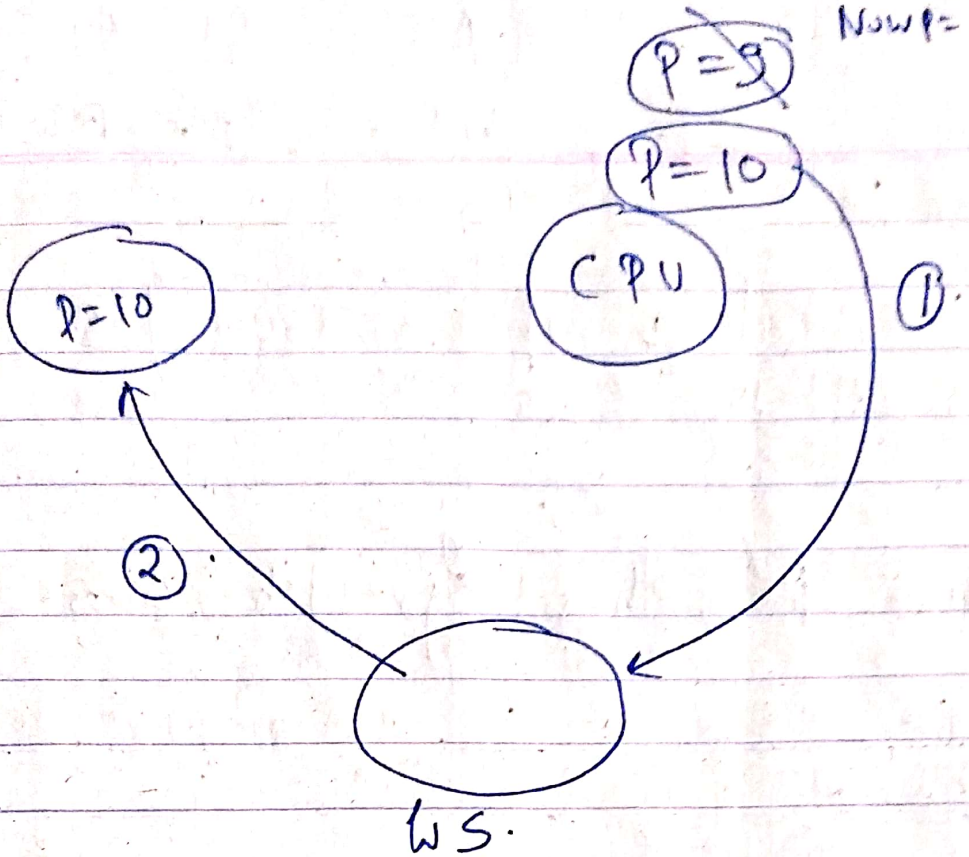
CPU Scheduling

By Short term Scheduler + Dispatcher.



* Who → Short term Scheduler.
* Where → Ready state to running.
* When → When a process moves from

1) Run → Terminate
   Run → Wait
   Run → Ready (Due to I/O) (Quantum time)
2) New → Ready i.e. when a process is just created (When higher priority process is created.
3) Wait → Ready. | Based on priority



Now P=10, again needs to schedule (as it is now ready).

FCFS

Criteria : Arrival Time

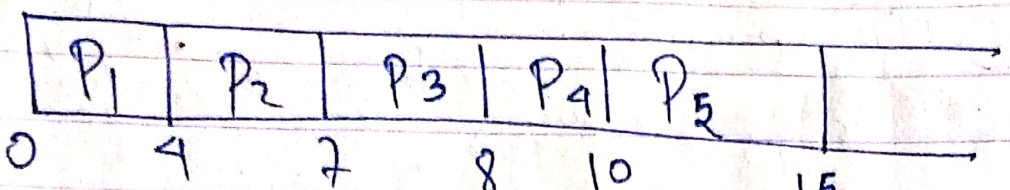
Mode : Non-Preemptive.

PNO	AT	BT
1	0	4
2	1	3
3	2	1
4	3	2
5	4	5

New → Ready → Run → Term.

TAT = CT - AT.

WT = TAT - BT.



(Gan Chot)

* if same AT, then go with lowest Process No.

CT	TAT	WT
4	4	0
7	6	3
8	6	5
10	7	5
15	11	6

WT = RT, for Non-Preemptive Algorithm.

Avg TAT = 0 + 3 + 5 +

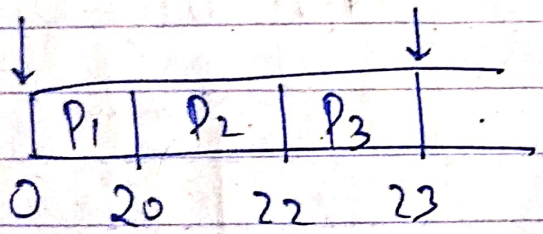
Avg TAT = $\frac{4+6+6+7+11}{5}$

Avg WT = $\frac{(0+3+5+5+6)}{5}$

(Avg WT ↑)

Convoy Effect / Starvation

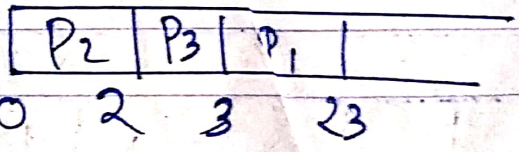
PNO	AT	BT	CT	TAT	WT
1	0	20	20	20	0
2	1	2	22	21	19
3	1	1	23	22	21



$$\text{Avg WT} = \frac{0+19+21}{3}$$

$$= \frac{40}{3}$$

PNO	AT	BT	CT	TAT
P1	1	20	23	22
P2	0	2	2	2
P3	0	1	3	3



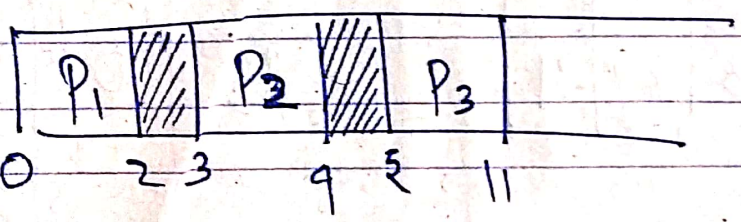
WT
2
0
2

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

* This is disadvantage of FCFS.

FCFS example

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



Avg WT = 0

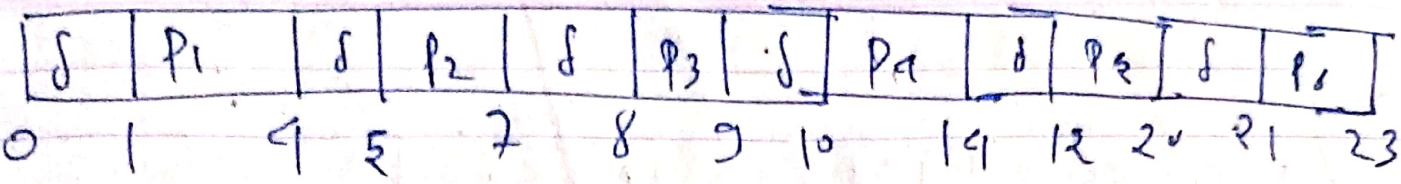
FCFS → By queue.

FCFS example

δ = 1 unit, (context switch time)

PNO	AT	BT
1	0	3
2	1	2
3	2	1
4	3	4
5	4	5
6	5	2

Schedule → Dispatch → Run



$$\text{In Efficiency} = \frac{6}{23} \times 100$$

Efficiency

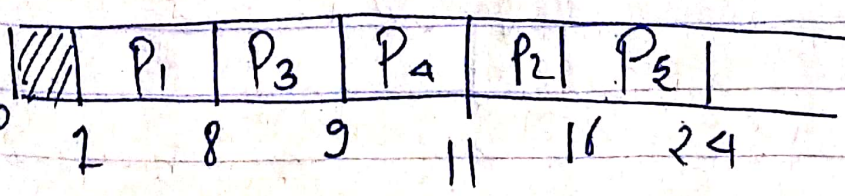
$$\text{Efficiency} = \left(1 - \frac{6}{23}\right) \times 100$$

Shortest Job First (By min heap)

criteria — Burst time

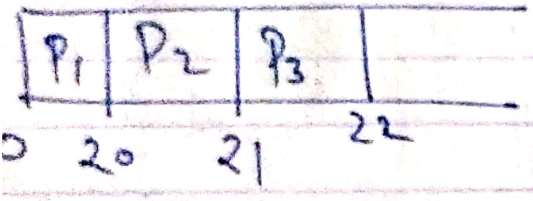
mode — Non-preemptive

PNO	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



Analysis of SJF

PNO	AT	BT	CT	WT	TAT	WT
1	0	20	20	0	20	0
2	1	1	21	20	20	19
3	2	1	22	21	21	20

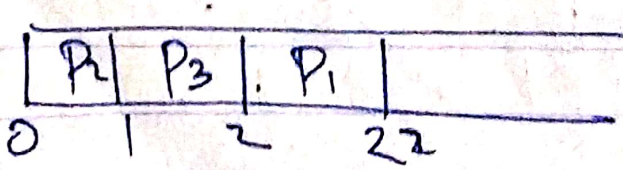


Avg WT = $\frac{39}{3} = 13$

(Convoy Effect)

But Now,

PNO	AT	BT	CT	WT	TAT	WT
1	2	20	22	2	20	0
2	0	1	1	0	1	0
3	1	1	2	1	1	0



Avg WT = 0

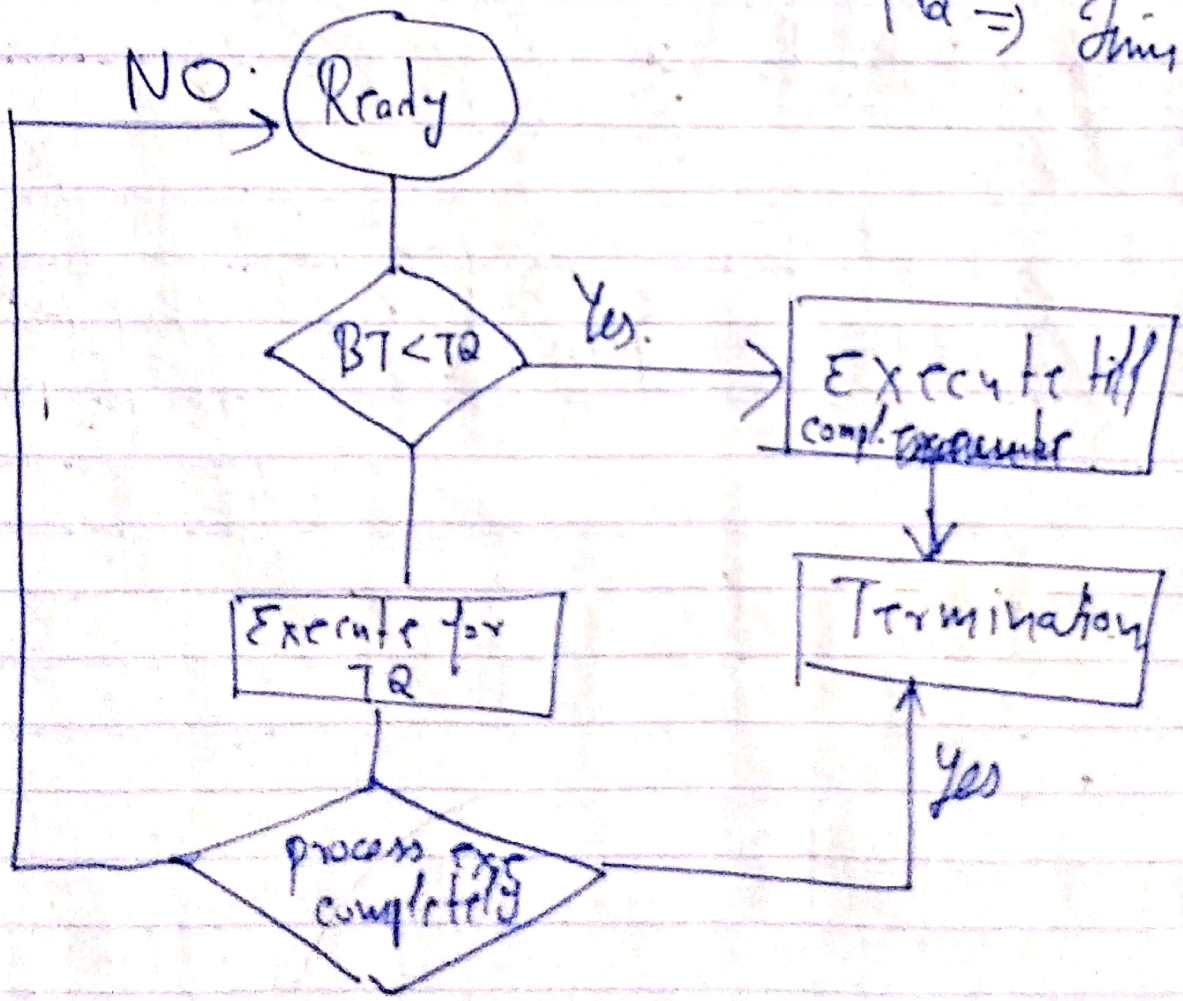
SJF is not practicle, we don't know prior about BT.

It is used for performance measure's.

Throughput = $\frac{3}{12}$, SJF give Best Throughput.

Round Robin

TQ ⇒ Time quantum.



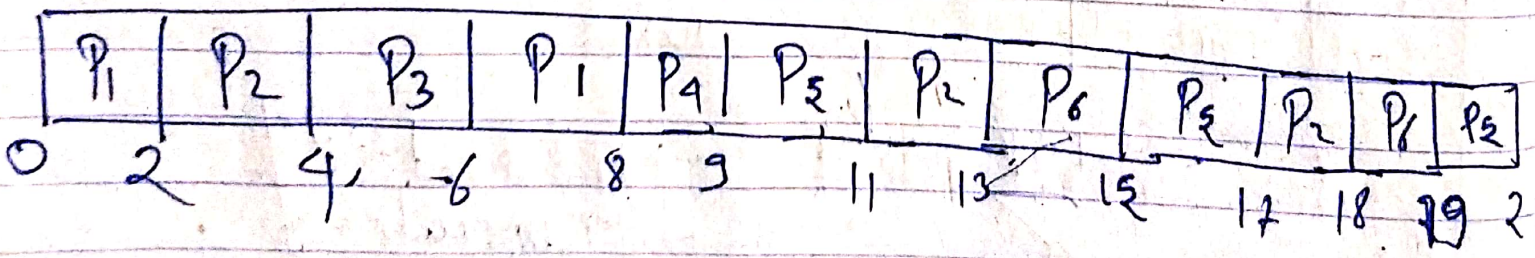
Example:

TQ = 2.

PNO	AT	BT	CT	TAT	WT
1	0	4	8	8	4
2	1	5	18	17	12
3	2	2	6	4	2
4	3	1	9	6	5
5	4	6	21	17	11
6	6	3	19	13	10

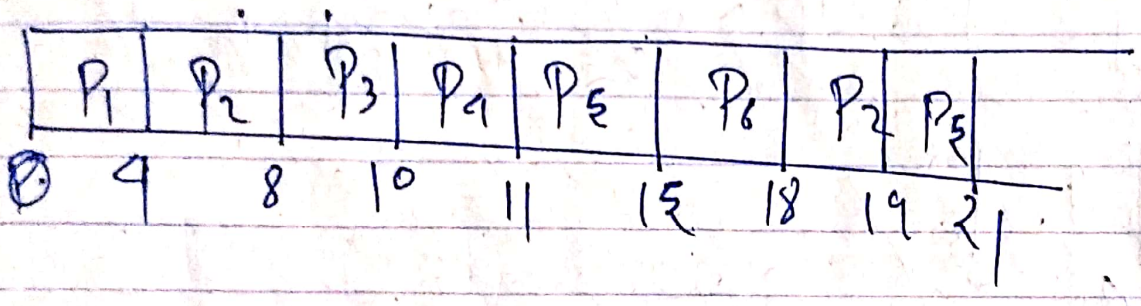
FCFS with preemption
= RR.

~~P1~~ ~~P2~~ ~~P3~~ ~~P1~~ ~~P4~~ ~~P5~~ ~~P2~~ ~~P6~~ ~~P5~~ ~~P2~~ ~~P6~~ P5



TQ = 4

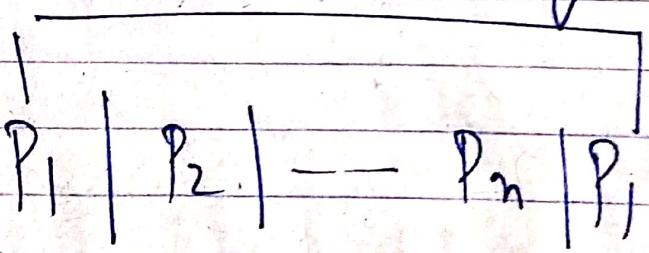
PNO	AT	BT	CT	TAT	WT
1	0	4	4	4	0
2	1	5	19	18	13
3	2	3	10	8	6
4	3	1	11	8	7
5	4	6	21	17	11
6	6	13	18	13	10

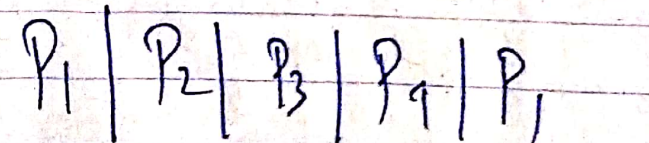


P1 P2 P3 P4 P5 P6 P2 P5

Here context switch is less due to longer TQ.

Ex: Consider n-process sharing the CPU in RR fashion. If the context switching time is 's' units. What must be the quantum time 'q' such that the no. of context switches are reduced, but at the same time each process is guaranteed to get the turn at the CPU for every 't' second.





$$ns + (n-1)q \leq t$$

$$q \leq \frac{t - ns}{(n-1)}$$

SJF

Advantage

→ maximum throughput

→ minimum Avg wt & TAT.

Disadvantage

⇒ Starvation to longer jobs

⇒ It is not implemented bcoz BT of process cannot be known.

Solution: SJF with predicted BT.

Prediction Techniques

Static

1) Process size
2) Process type

Dynamic

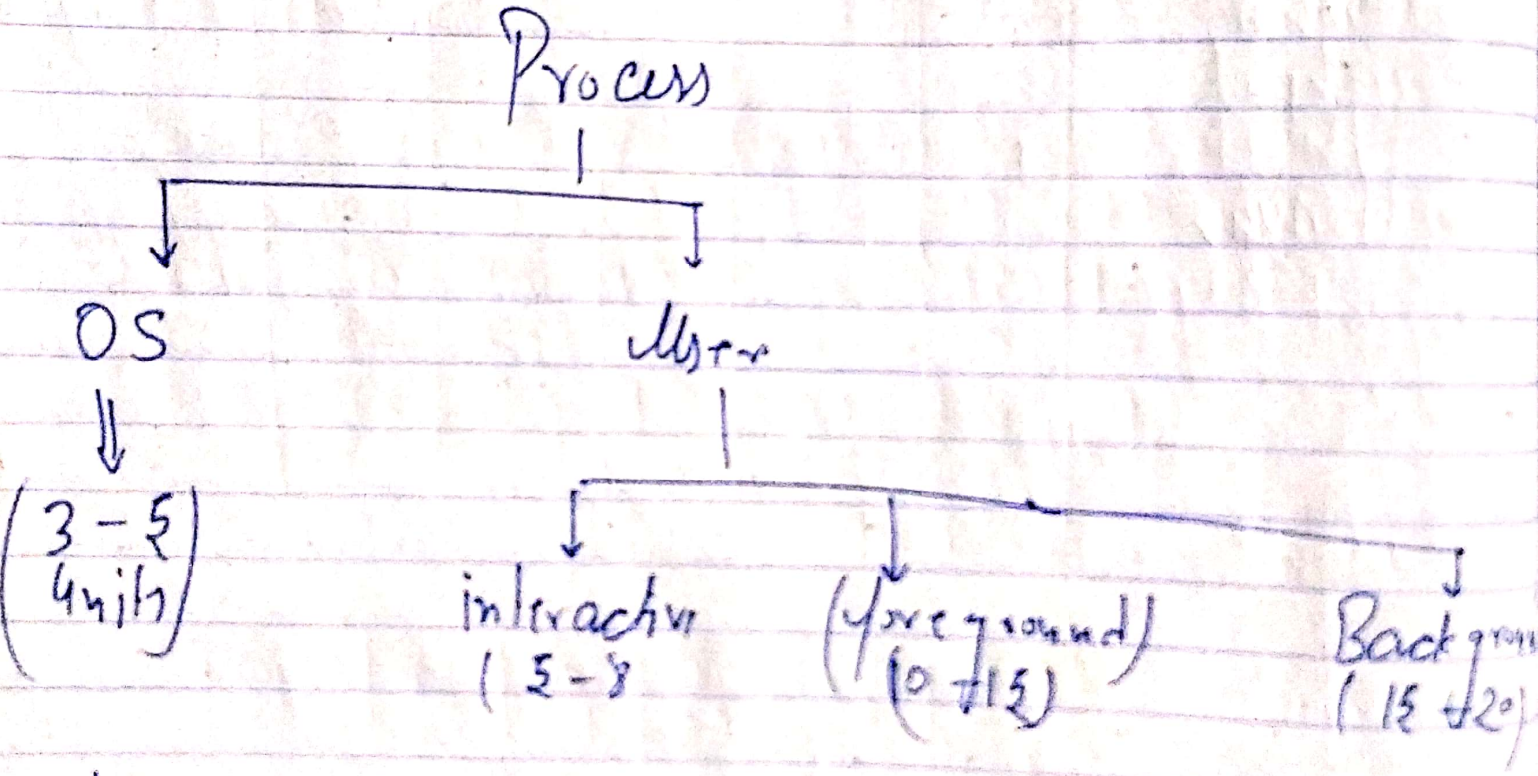
1) Simple averaging
2) Exponential.

Process size (Bytes)

Pold = 200 KB ⇒ 20 units

Pnew = 201 KB ⇒ 20 units

Process type



Simple Averaging

→ Given n-process
(P1 -- Pn)

→ Let ti be the actual BT.

⇒ Let T_i denote the predicted BT.

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

Exponential Avging

$\alpha \rightarrow$ Smoothning factor.

$$T_{n+1} = \alpha t_n + (1-\alpha) T_n, \quad 0 \le \alpha \le 1$$

Ex $\alpha = 0.5$, $T_1 = 10$.

Actual BT $(t_1, t_2, t_3, t_4) = (9, 8, 6, 7)$

Then $T_5 = ?$

T_1 & α ?.

$T_2 = 0.5 \times t_1 + (1-0.5) T_1 =$

$= 0.5 \times 9 + 0.5 \times 10 =$

$T = 2$

$T_3 = 0.5 \times t_2 + (1-0.5) T_2$

$= 0.5 \times 8 + 0.5 \times 7$

$T_4 = 0.5 \times t_3 + (1-\alpha) T_3 = 6.75$

$T_5 = \alpha t_4 + (1-\alpha) T_4 = 6.875$

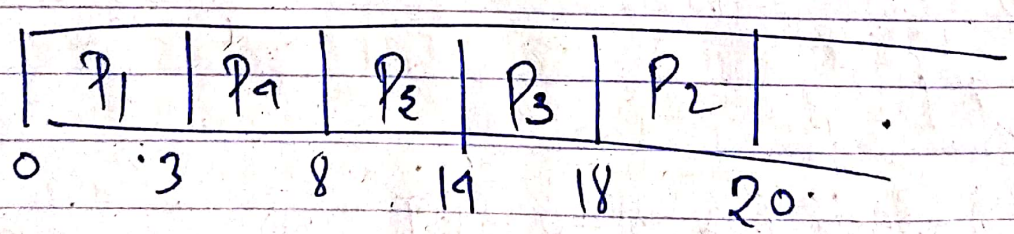
Longest Job first algorithm

Process having longest BT gets scheduled first

Criteria: BT.

mode: non-preemptive.

PNO	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



Today: HRNN

Longest Remaining Time first

PNO	AT	BT	CT	TAT	WT	RT
1	1	21	18	17	15	0
2	2	4 3 2 1	19	17	13	0
3	3	6 5 4 3 2 1	20	17	11	0
4	4	8 7 6 5 4 3 2 1	21	17	9	0

17 18 19 20

P1	P2	P3	P4

	P1	P2	P3	P4	P4	P4	P3	P4	P3	P4	P2	P3	P4	P2	P3	P4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Highest Response ratio next (HRNN)

PNO	AT	BT
0	0	3
1	2	6
2	4	4
3	6	5
4	8	2

P0	P1	P2	P4	P3
0	3	9	13	15	20

$RR_2 = \frac{5+4}{4} = 2.25$

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

$RR_4 = \frac{1+2}{2} = \frac{3}{2} = 1.5$

$RR_3 = \frac{7+5}{5} = 2.4$

$RR_4 = \frac{5+2}{2} = 3.5$

Criteria : Response Time (RR) = $\frac{W+S}{S}$

W : Waiting Time for a process so far

S : Service time of a process or BT

→ HRRN not only favours shorter job but also limits the WT of longer jobs.

→ Non-preemptive.

Priority Scheduling :

- Static
  - ↓
  - doesn't change through the execution of the process.
- Dynamic
  - ↓
  - Changes at regular interval of time

Two Types

Priority

Preemptive

Non-preemptive.

Non-Preemptive

Always same for non-preemptive

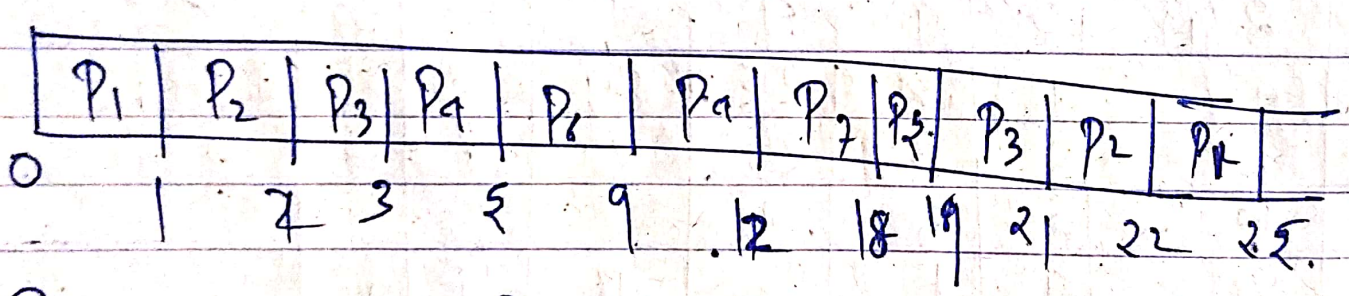
PNO	Priority	AT	BT	CT	TAT	WT	RT	
1	2 (L)	0	4	4	4	0	0	✓
2	4	1	3	25	24	22	22	
3	1	2	3	23	21	18	18	✓
4	10	3	5	9	6	1	1	✓
5	8	4	1	20	16	15	15	✓
6	12 (H)	5	4	13	8	4	4	✓
7	9	6	6	19	13	7	7	✓

P_1	P_4	P_6	P_7	P_5	P_3	P_2

0 4 9 10 13 19 20 23 25

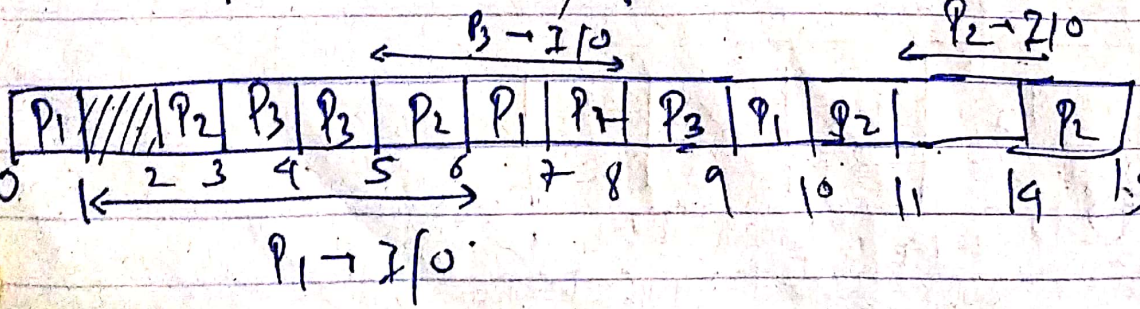
Preemptive - Priority

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



Preemptive - Priority with CPU & I/O timing

PNO	AT	Priority	CPU	I/O	CPU	CT	TAT	WT
P1	0	2	1 0	5	3 2 0	10	10	6
P2	2	3(1)	3 2 1 0	3	1 0	15	13	9
P3	3	1(4)	2 0	3	1 0	9	6	3



Preemptive.

Shortest Remaining time first (SRTF)

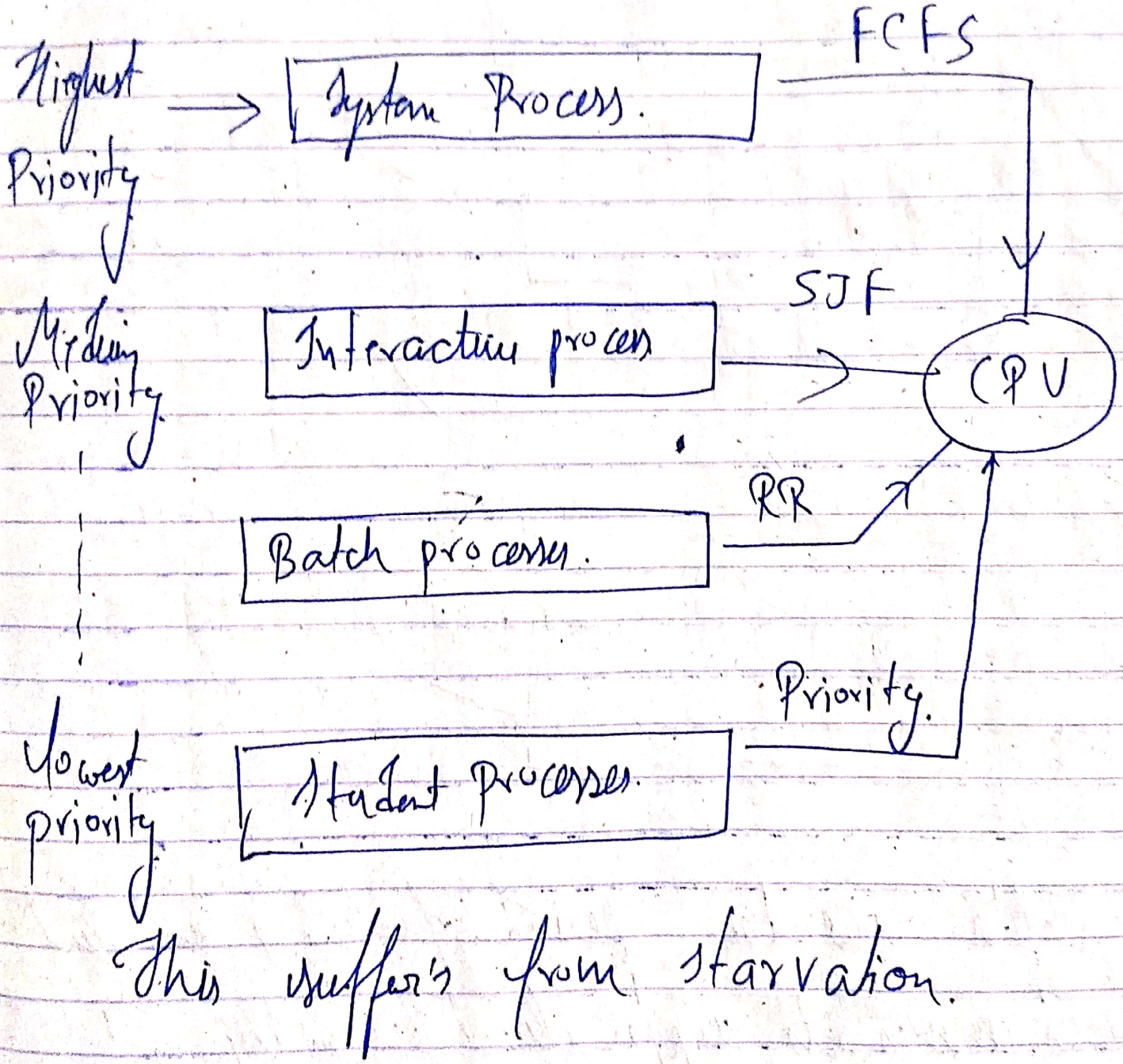
P NO	AT	BT	CT	TAT	WT
1	0	~~6~~	19	19	12
2	1	~~5~~ 4	13	12	7
3	2	~~3 2~~ 0	6	4	1
4	3	0	4	1	0
5	4	0	9	5	3
6	5	0	7	2	1

P1	P2	P3	P4	P3	P3	P6	P5	P2	P1
0	1	2	3	4	5	6	7	9	13	19

Ex

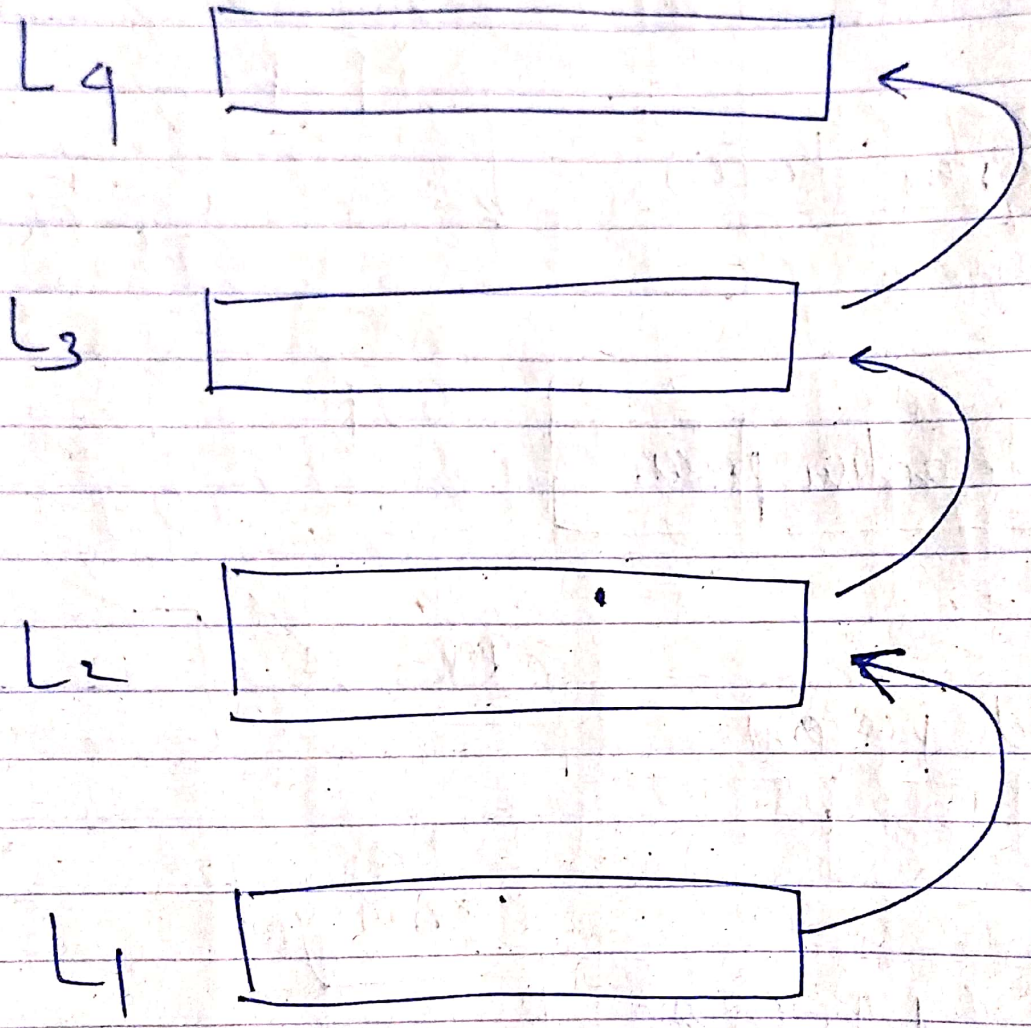
Doubt Batch, interactive, system proc.

Multilevel Queue Scheduling



This suffers from starvation.

Multilevel feedback queue

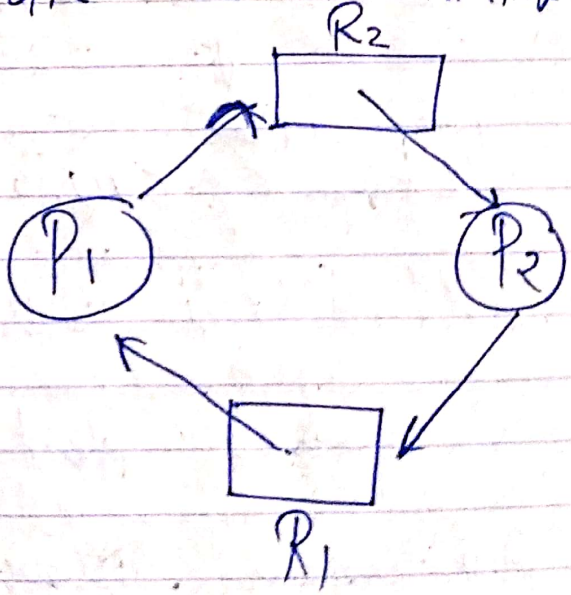


to after certain period of time, lower level process moves to upper level & finally get chance to CPU, hence starvation is removed

Deadlock

→ In a multiprogramming system a number of process competes for limited number of resources. If resources is not available at the instance then process enters into waiting state.

→ If a process unable to change its waiting state indefinitely because the resources requested by it are held by another waiting process then system is said to be deadlocked.



□ → Resource

○ → Process.

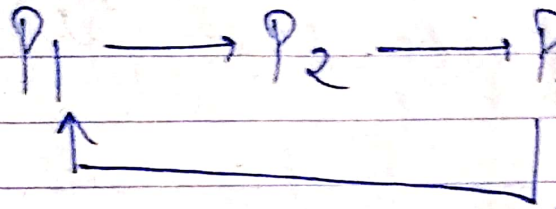
System model

→ Every process will request for the resource
→ if entertained then, process will use the resource
→ Process must to release the resource after use.

Necessary Conditions of Deadlock

1) Mutual Exclusion: one at a time i.e. printer.
2) Hold & Wait: A process is currently holding at least one resources & requesting additional resources which are being held by other processes.
3) Non-Preemption: A resource can not be pre-empted from a process by any other process resource can be released only voluntarily by the process holding it.

4) Circular Wait → Each process must be waiting for a resources which is being held by another process, which in turn waiting for the first process to release the resource.



Deadlock handling Methods (severe * frequency)

1) Prevention → means design such a system which violate at least one of four necessary condition of deadlock & ensure undergo-ing free from deadlock.

2) Avoidance → System maintains a set of data using which it takes a decision whether to maintain a new request or not, to be in safe state.

3) Detection & recovery → Here we wait until deadlock occurs & once we detect it, we recover from it

4) Ignorance/Ostrich algo → We ignore the problem as if it does not exist.

Deadlock Prevention

Mutual Exclusion

→ When environment is very secure.

→ Prevention remove one of the four condition of deadlock.

1) Mutual Exclusion.

We cannot remove mutual exclusion as it depends on h/w only.

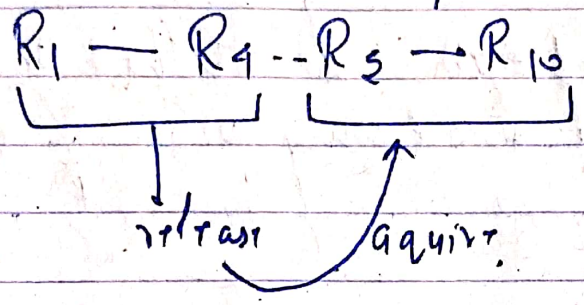
ex: printer.

2) Deadlock Prevention (Hold & Wait)

→ Conservation Approach → Process is allowed to start execution if & only if it has acquired all the resources

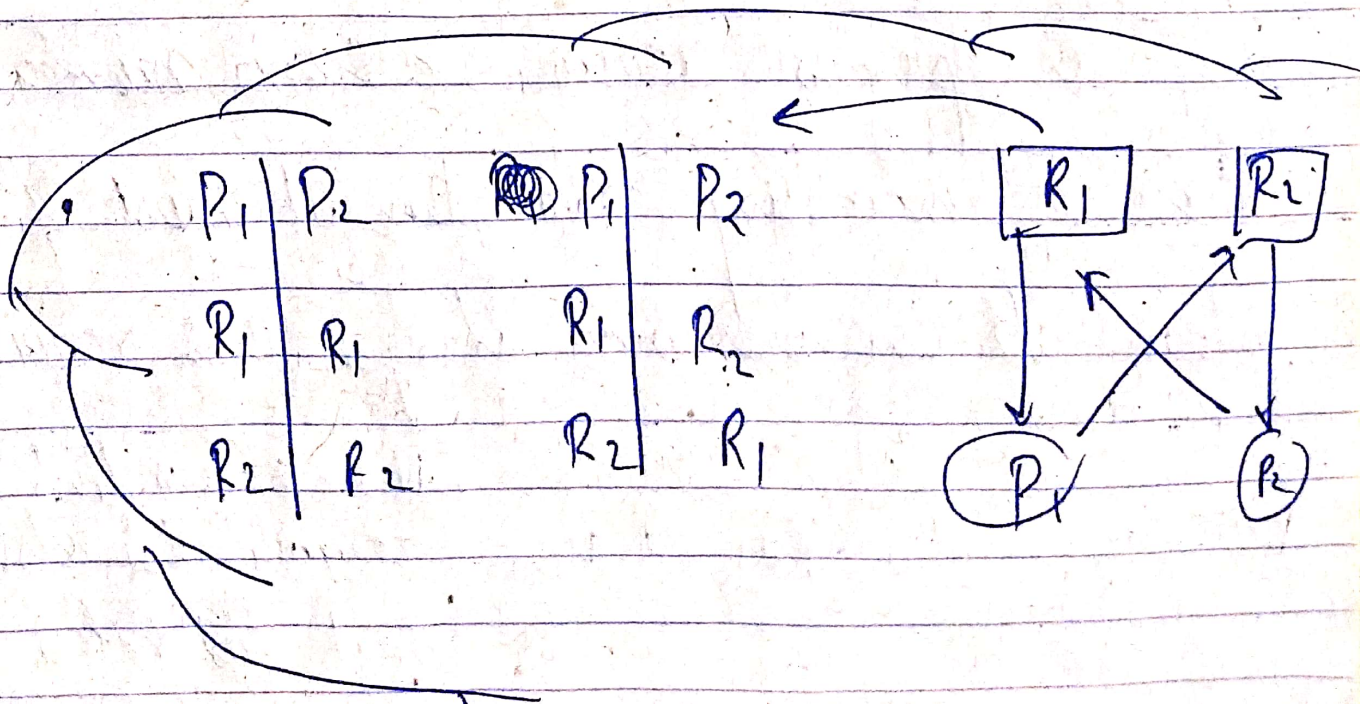
(But it is less efficient, not implementable, easy deadlock independence)

2) Do-not hold → Process will acquire only desired resources but before making any fresh request it must release all the resources that it currently hold (efficient, implementable)



3) Wait-Time-out: We place a max^m time up to which a process can wait. After which process must release all the holding resources

3) Deadlock Prevention (Circular Wait)

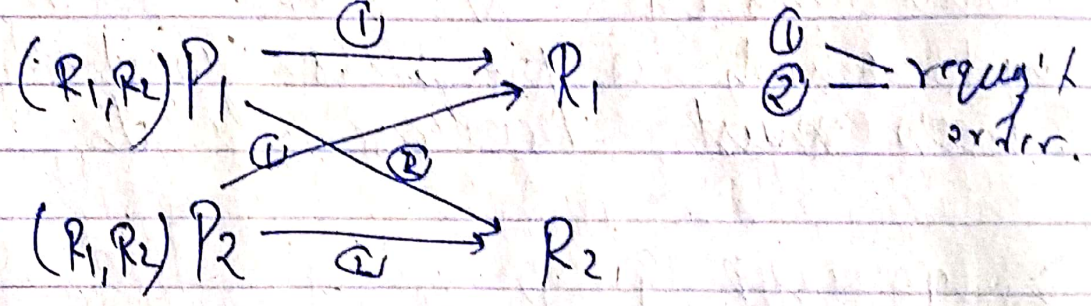


→ Circular wait can be eliminated by first giving a natural number of every resource

$$f: N \longrightarrow R$$

→ Allow every process to either only in the increasing or decreasing order of the resource number.

eg.



→ If a process requires a lesser number (in case of increasing order), then it must first release all the resources larger than required number.

For ex R11 R21 R41 (Now process finds that it requires R31 also to it first releases R41 then again request for R31)

Deadlock Avoidance

"Banker's Algorithm"

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

Process	Allocation			Max Need			Current Available			Remaining Need		
	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				6	0	0
P_4	2	1	1	4	2	2	7	4	3	2	1	1 ✓
P_5	0	0	2	5	3	3	7	4	5	5	3	3 ✓
	7	2	5				7	5	5			
							10	5	7			

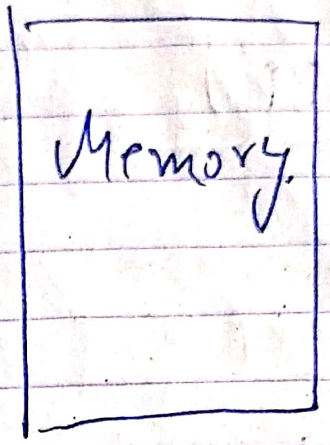
Safe State → Deadlock may occur.
↑ ↓
Unsafe State → Deadlock cannot happen → Safe sequence

$P_2 \rightarrow P_4 \rightarrow P_5 \rightarrow P_1 \rightarrow P_3$ (Safe sequence)

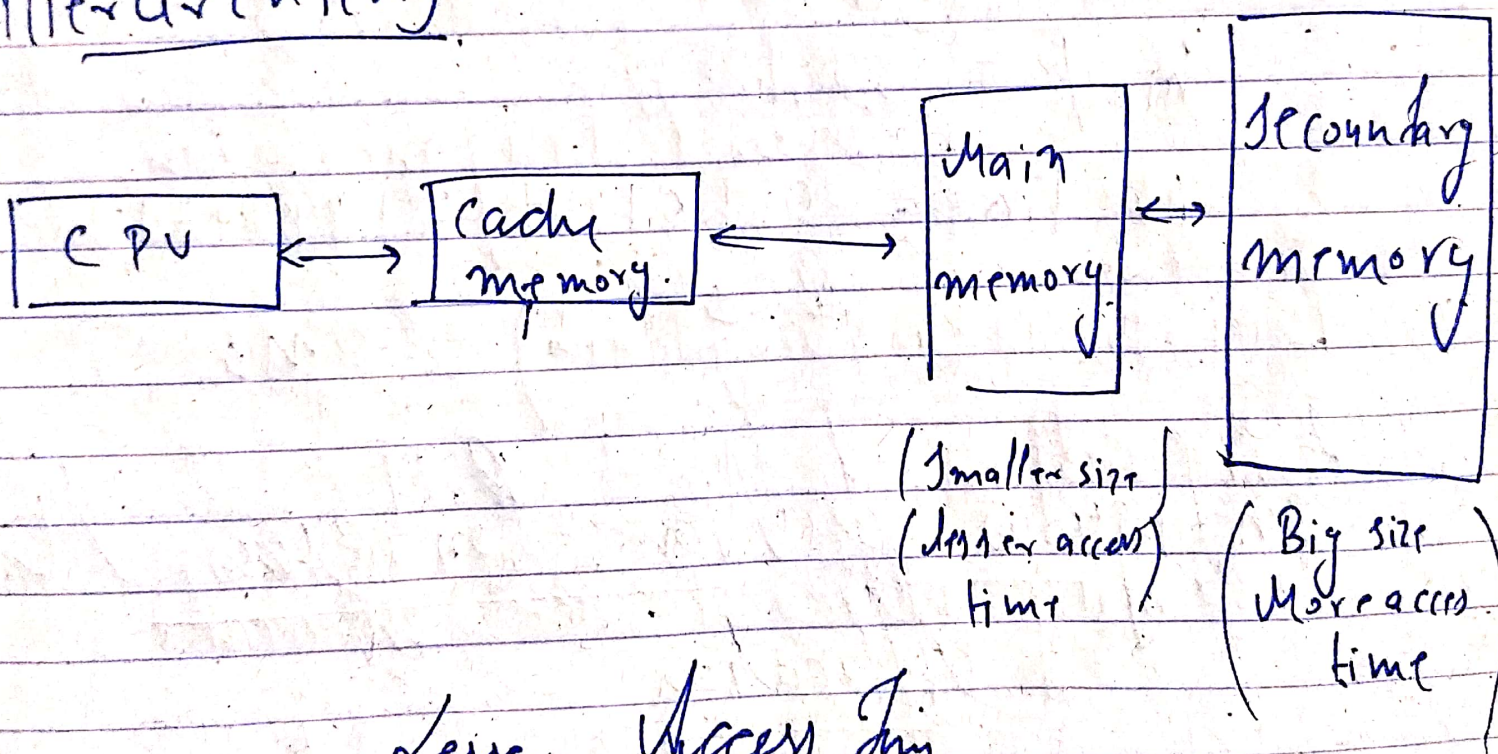
As here, Maximum Need is priority-known & hence cannot be practically used.

Basics of Memory Management

1) Size (must be more)
2) Access time (must be less)
3) Per unit cost (must be less).

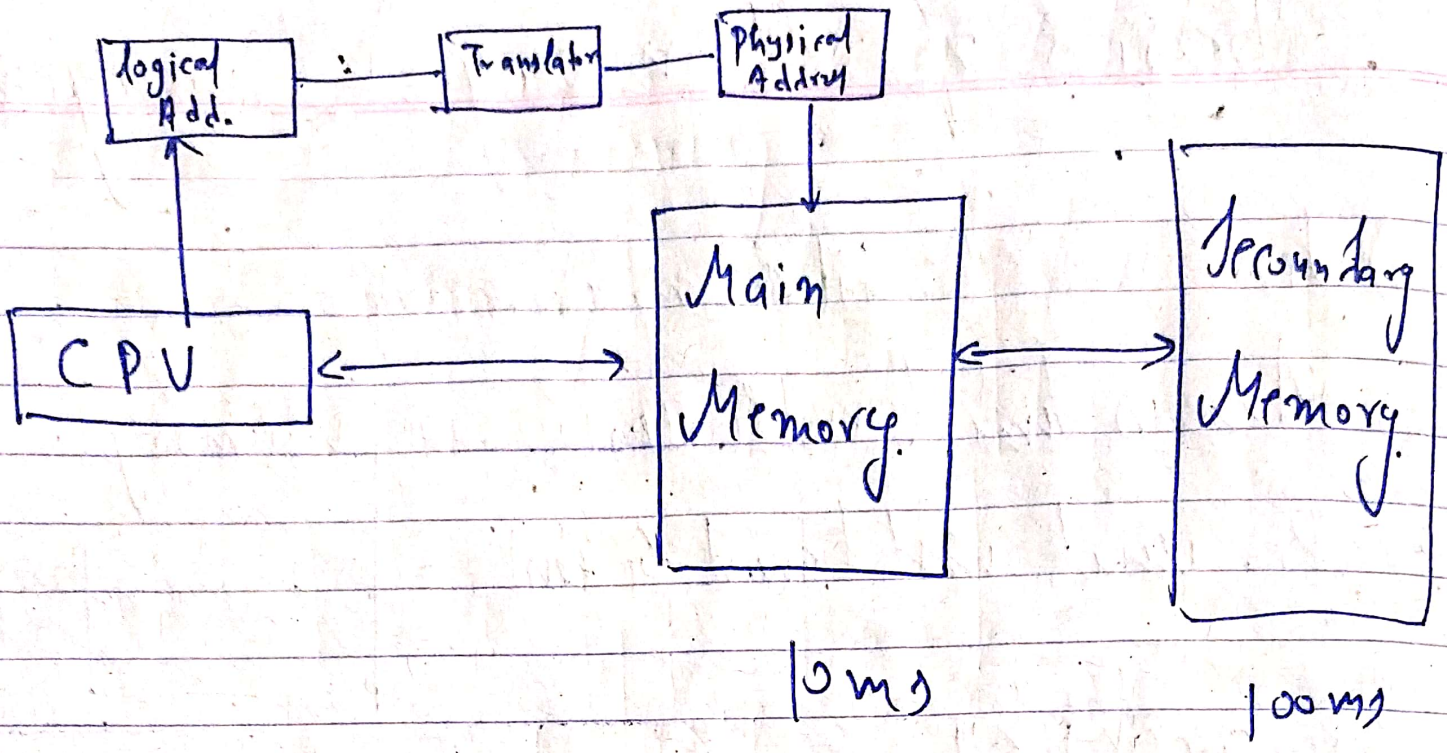


Hierarchichy



Faster Access Time

Lesser size.



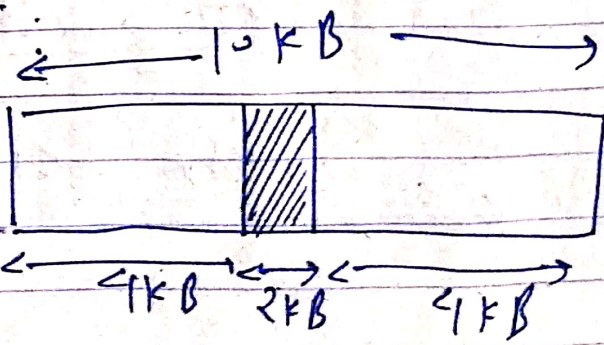
hit % = 90%

Avg time = $0.9 \times 10 + 0.1 \times (100 + 10)$

$= 9 + 19 = 20$ ms.

Contiguous & Non-Contiguous Memory Allocation

Contiguous → faster access.



Need ⇒ 2KB → cannot be allocated

↓

external fragmentation

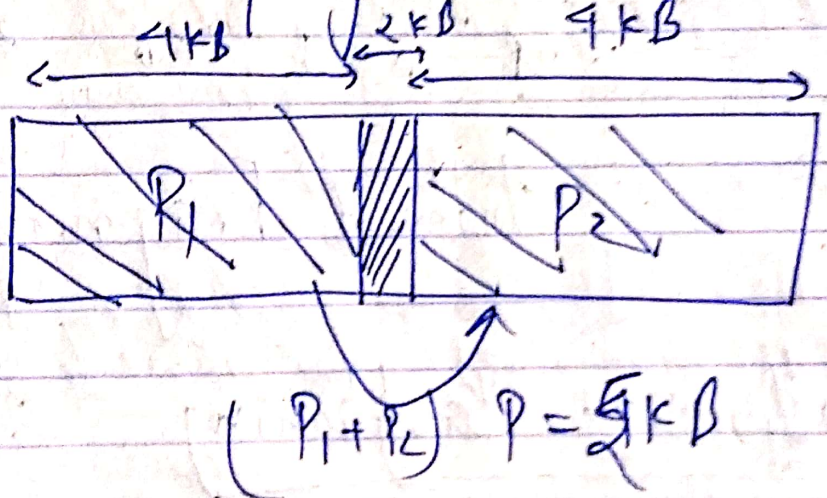
↓ Contiguous memory allocation suffers from →

Non - Contiguous

Linked list

~~memory~~ slow access.

→ No External fragmentation.

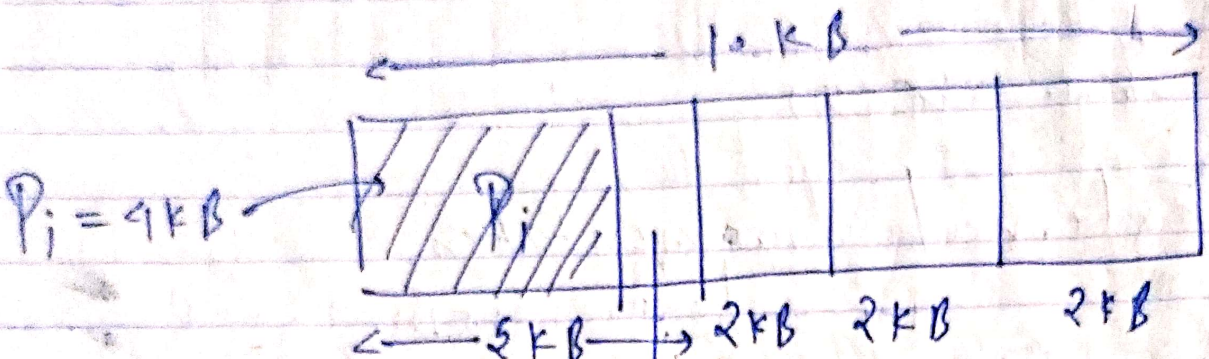


Contiguous Memory Allocation

- Fixed size Partitioning.
- Variable size Partitioning.

1) Fixed size Partitioning →

Divide memory into fixed size partitioning



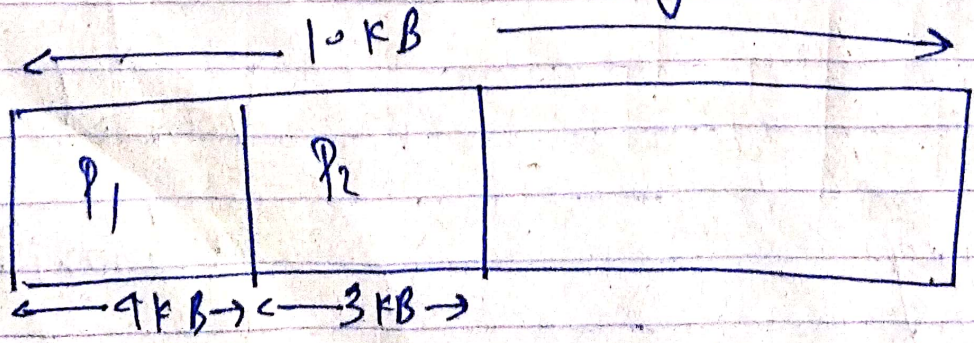
Internal Fragmentation.

2) Variable size Partitioning

→ No division

→ Memory allocated acc^n its need.

→ No Internal Fragmentation.



P_1 = 4KB ✓

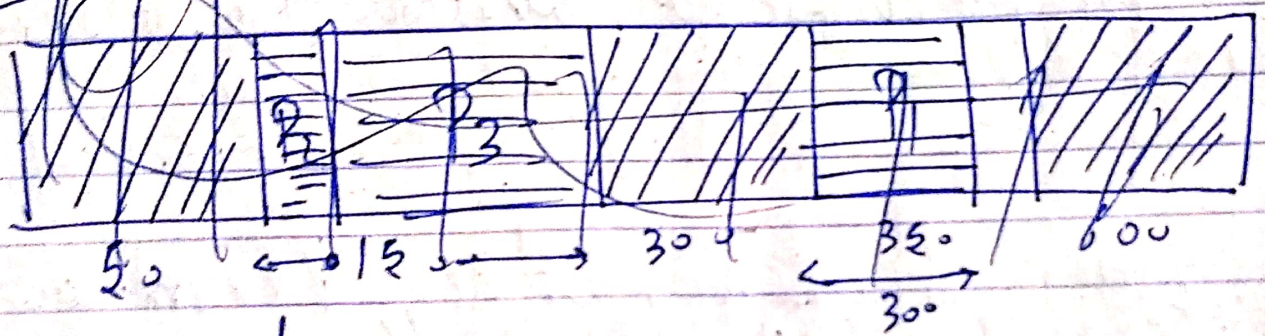
P_2 = 3KB ✓

Variable Size Partitioning

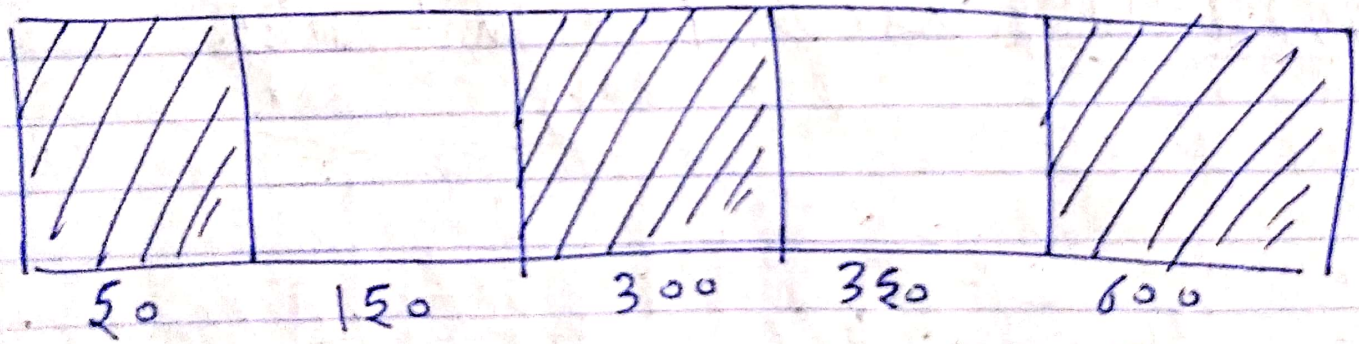
1) With Best fit
2) With Worst fit
3) With first fit.

$P_1 = 300$
$P_2 = 25$
$P_3 = 125$
$P_4 = 50$

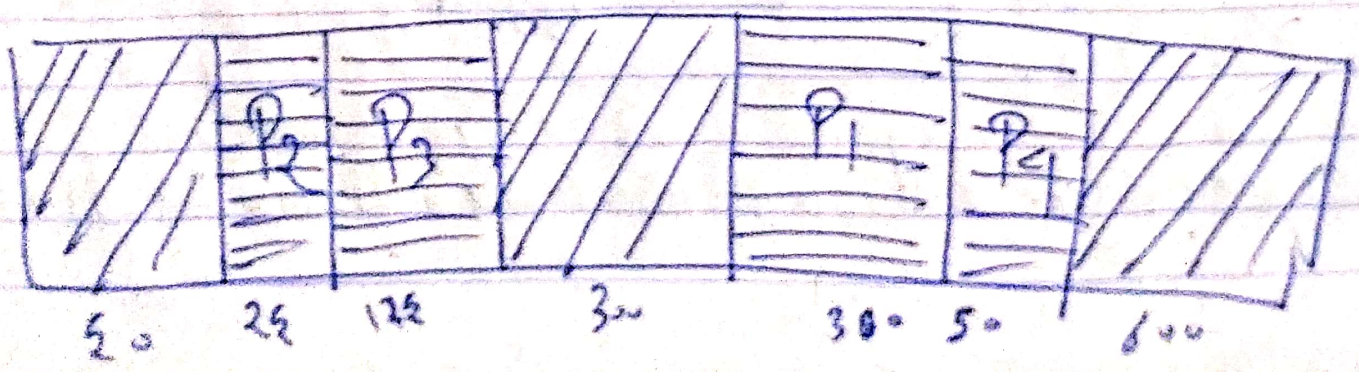
First fit



Initial Scenario

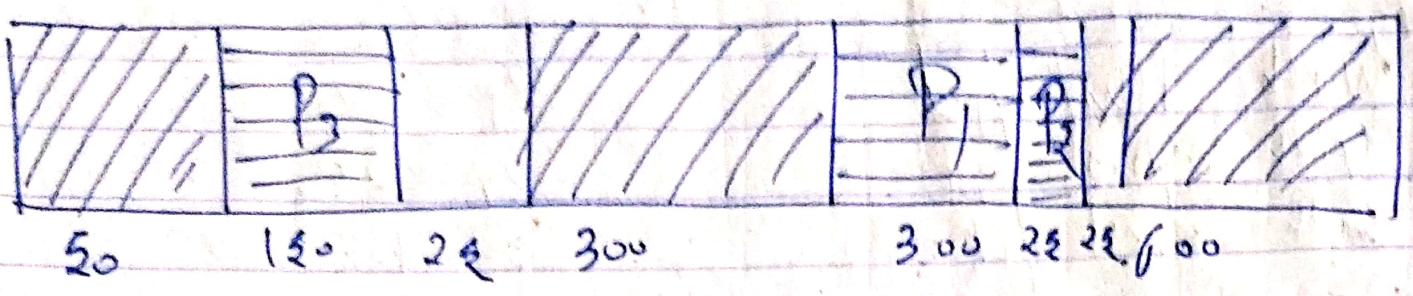


By First fit → FCFS



(Doubt) why this

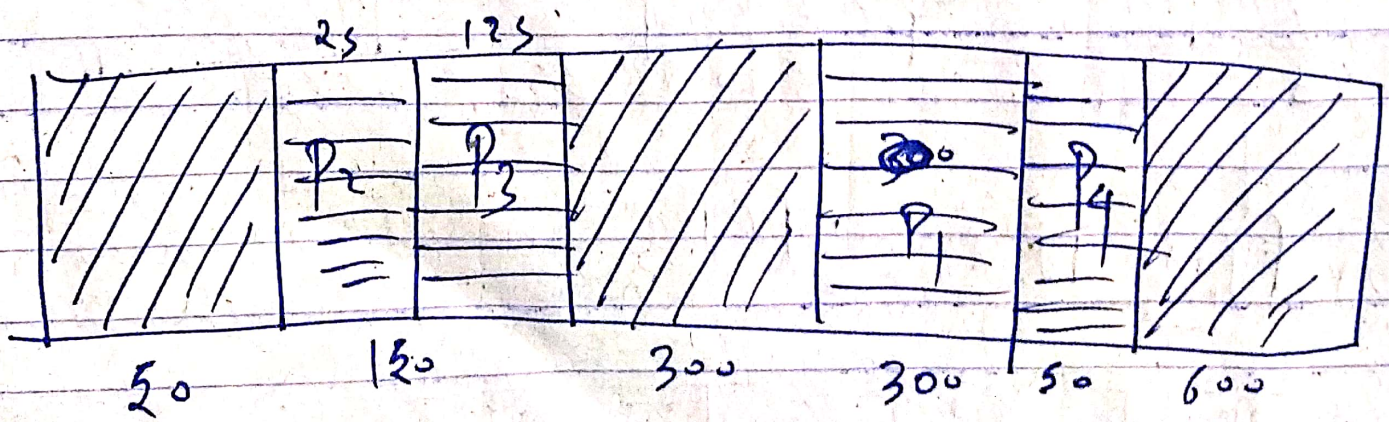
) Best Fit → smallest block that satisfies



But P4 cannot be alloted. → suffers from External fragmentation.

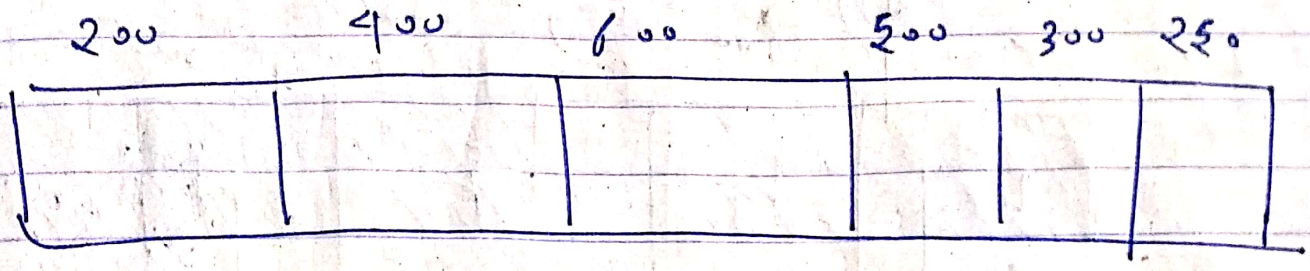
* Perform worst for variable size partitioning

3) Worst Fit → Largest Block that satisfies



+ Perform Best for variable size Partitioning.

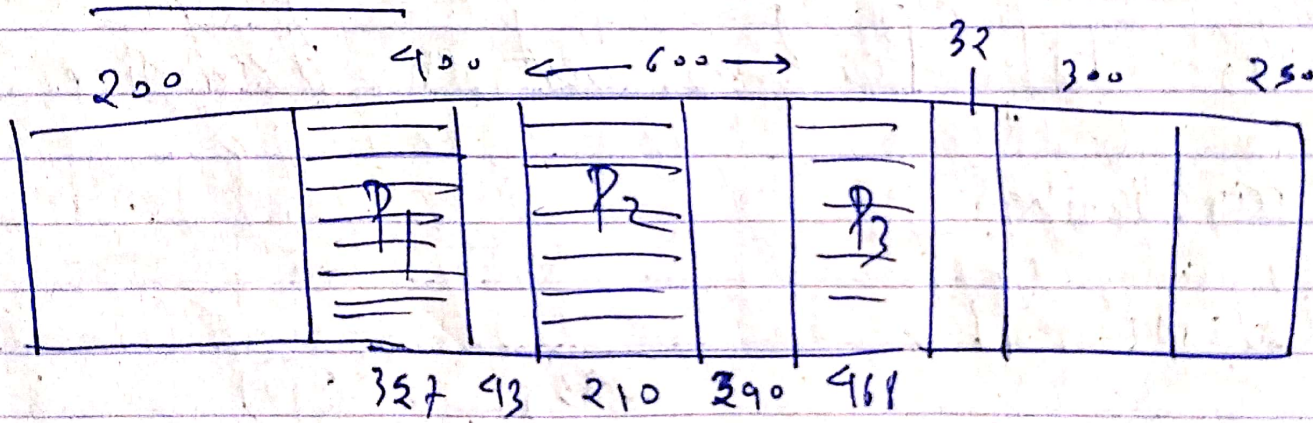
Fixed Size Partitioning



$P_1 = 357$, $P_3 = 468$

$P_2 = 210$, $P_4 = 491$

1) First Fit



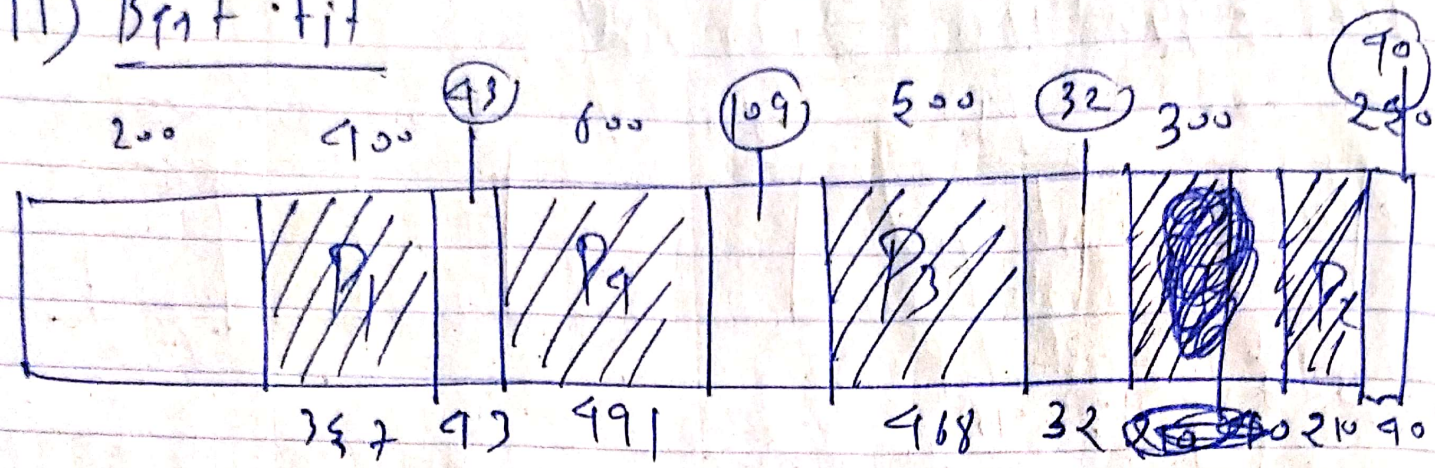
$P_4 \rightarrow$ cannot allocate

Internal fragmentation = 43 + 390 + 32

External fragmentation = 491

External fragmentation → if size is allocated is greater than smaller or equal but are not contiguous.

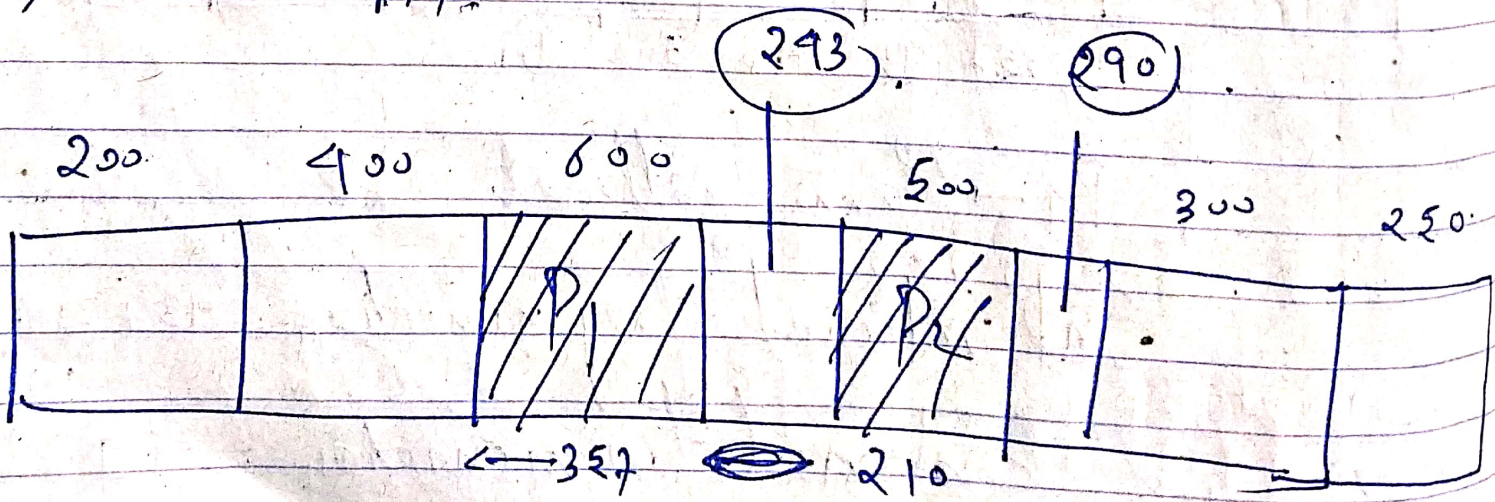
II) Best fit



No External fragmentation:

Best Fit Performs Best in fixed size Partitioning.

III) Worst fit:

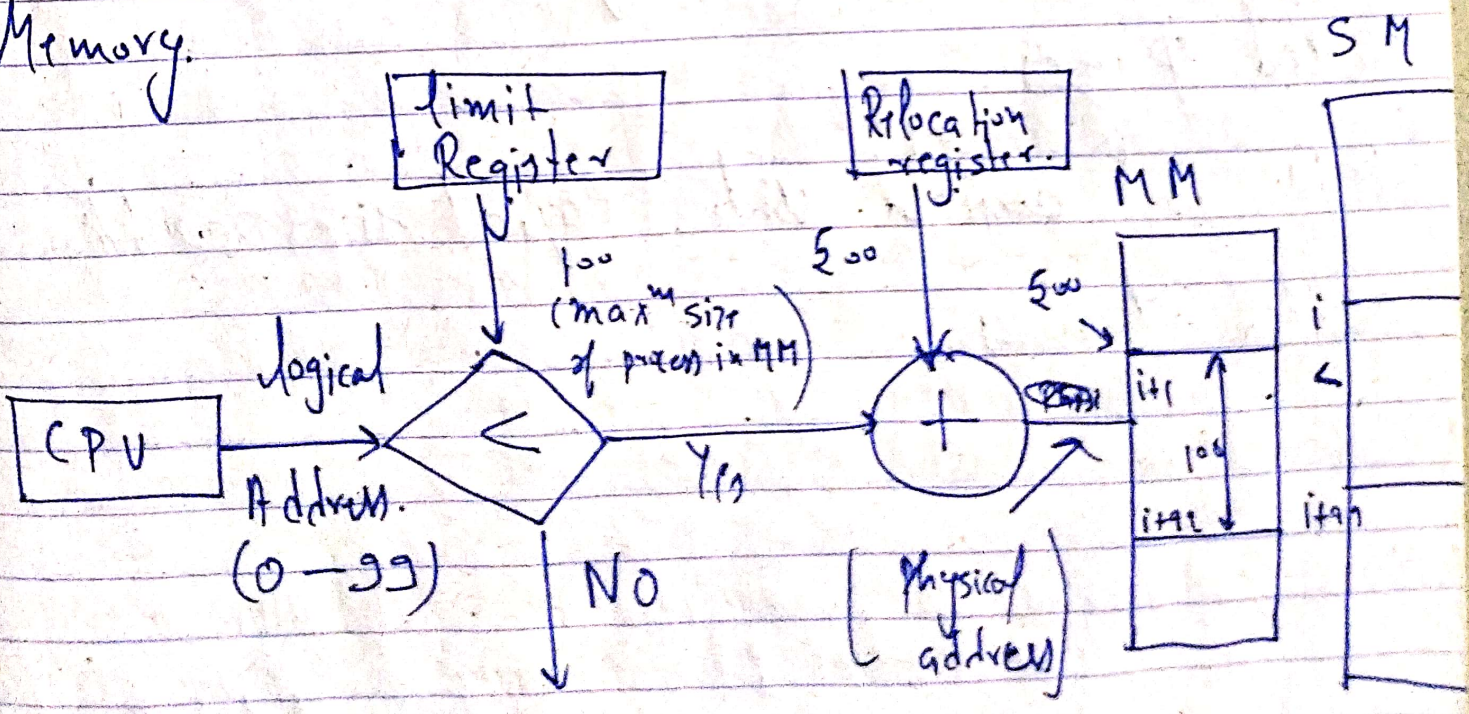


We cannot allocate P3 & P4 suffer from external fragmentation.

Worst Fit Algo — Performs Worst than Fixed Size Partitioning.

Address Translation in Contiguous Allocation

CPU generate logical add. for SM, But logical is converted and to Physical add. for Main Memory.

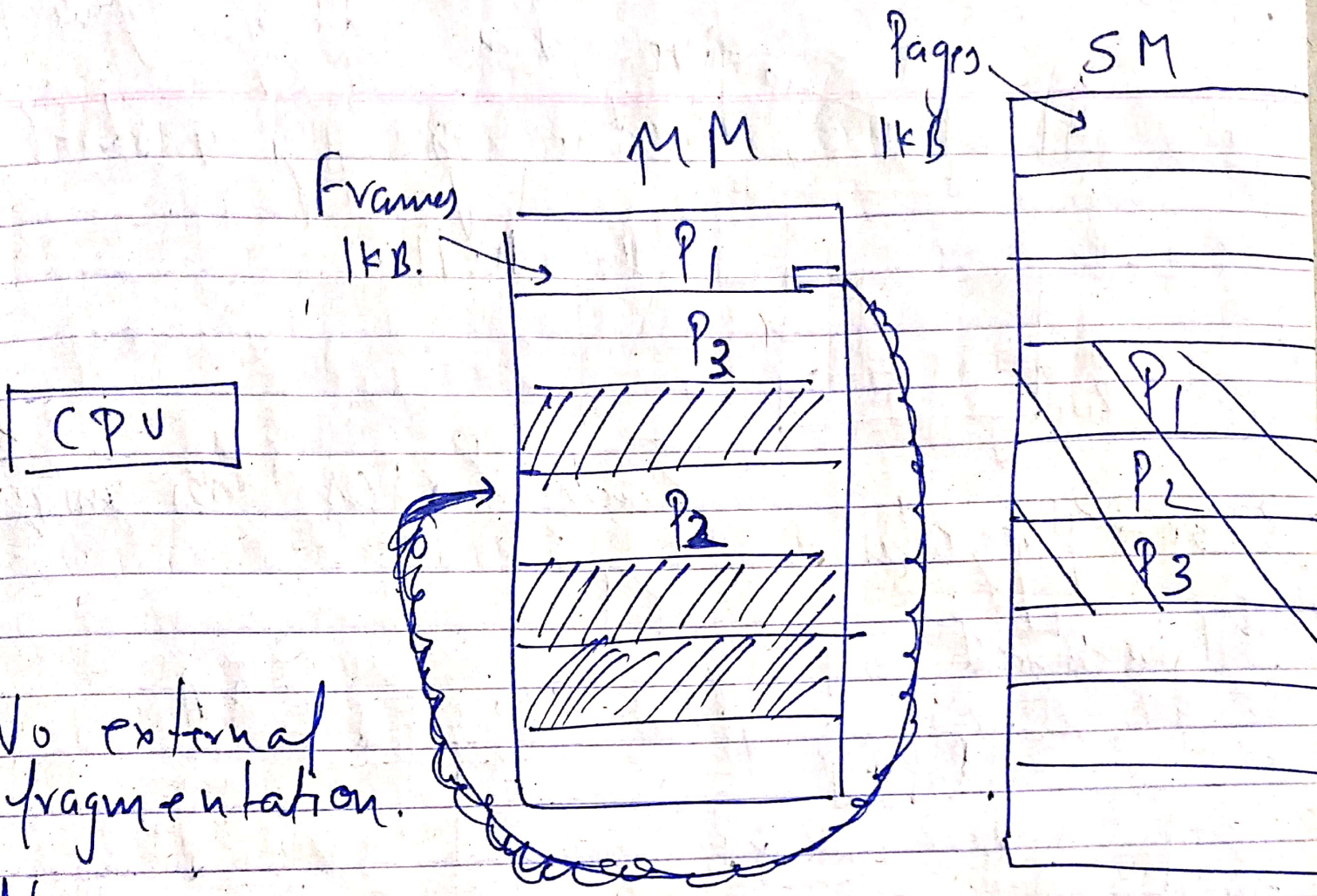


	LR	RR.	
P_0	500	1200	420 ←
P_1	275	550	300 ✗
P_2	212	880	210 ←
P_3	420	1400	450 ✗
P_4	118	200	P0 ←

Paging (Non Contiguous Allocation)

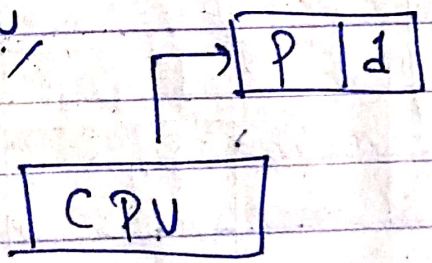
SM is devided into equal size partition called pages.

MM is devided into equal size partition called frames.



No external fragmentation.

Now,



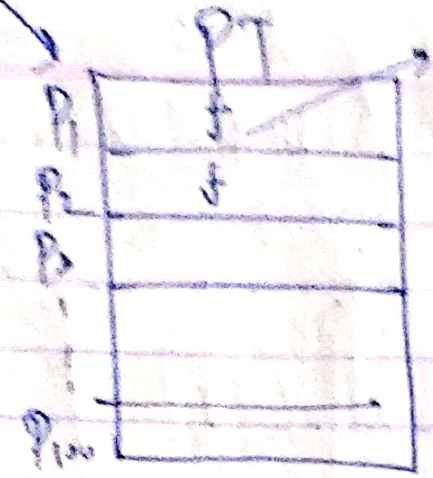
p → Page no.

d → offset (instruction offset)

Pages are stored in MM in non-contiguous method.

Now, how to access physical process in MM, → Page Table

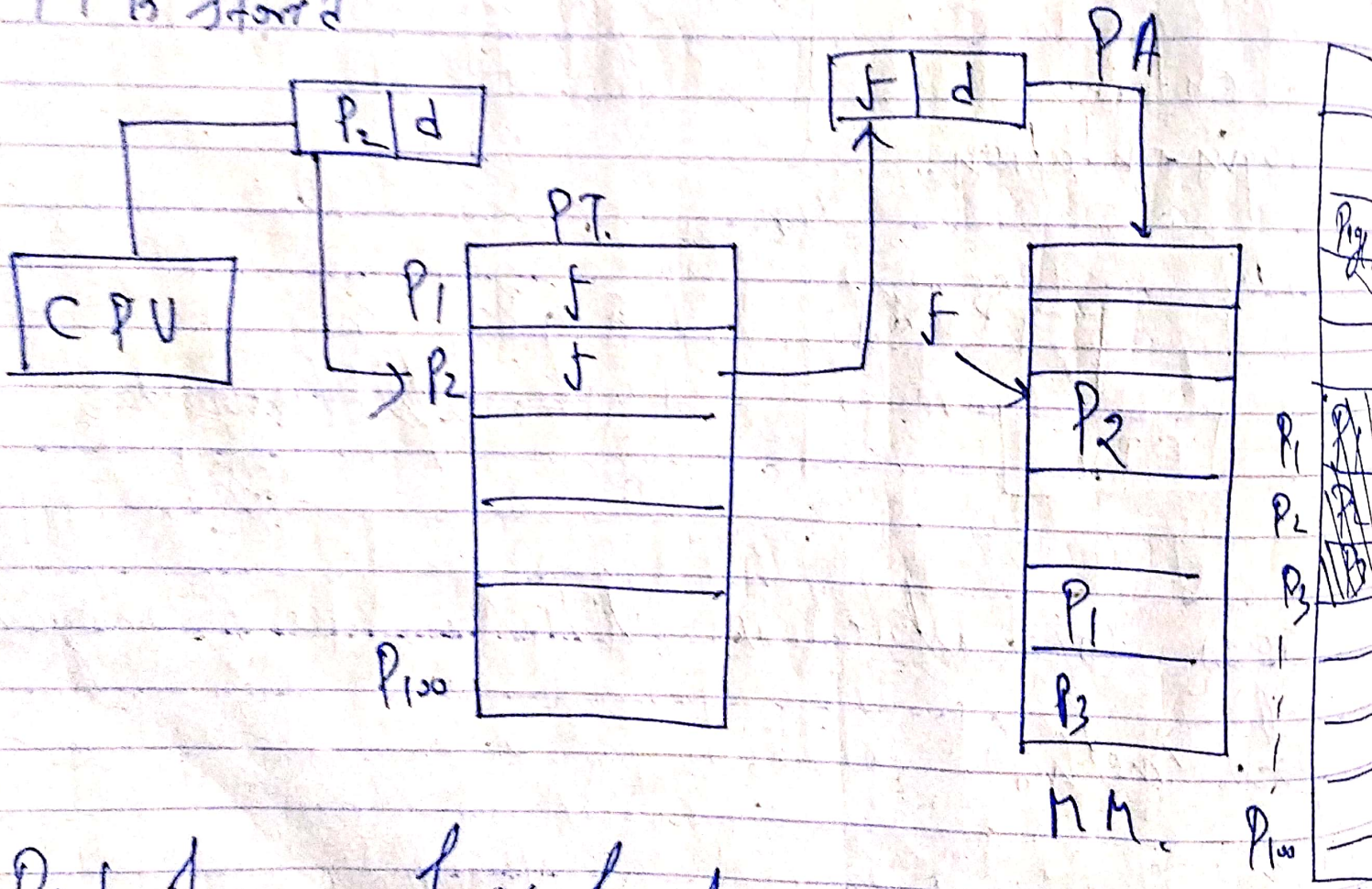
PTBR



give base add of frame where your page is stored in M.M.

Each Process has unique PT.

PTBR → give, where PT is stored



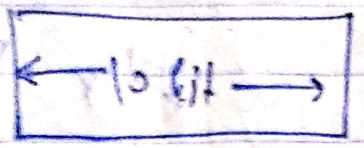
But some drawbacks

i) Access time is slow, due PT

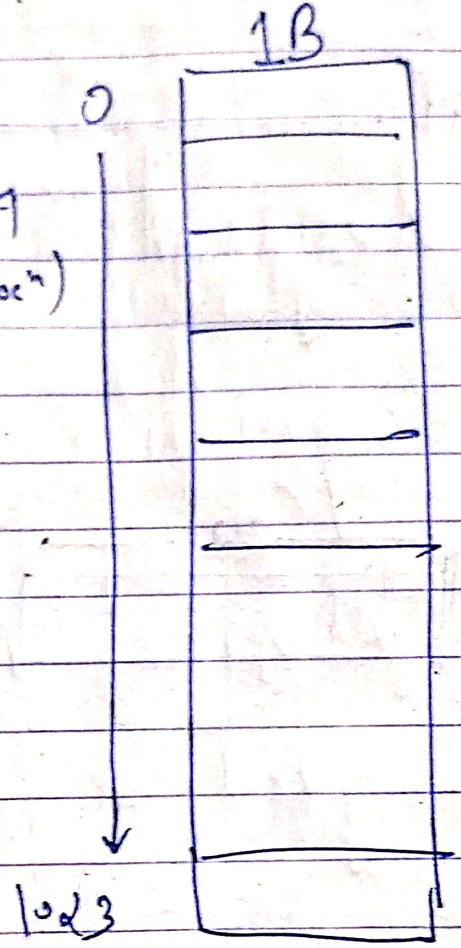
ii) Extra space for PT.

iii) Suffers from internal fragmentation due to fixed size page.

n bits ⟶ 2^n combination.



$2^{10} = 1024$ (no. of loc^n)



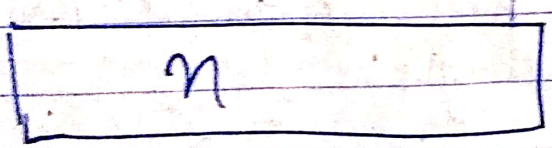
1B → size of each location.

1024 X 1B

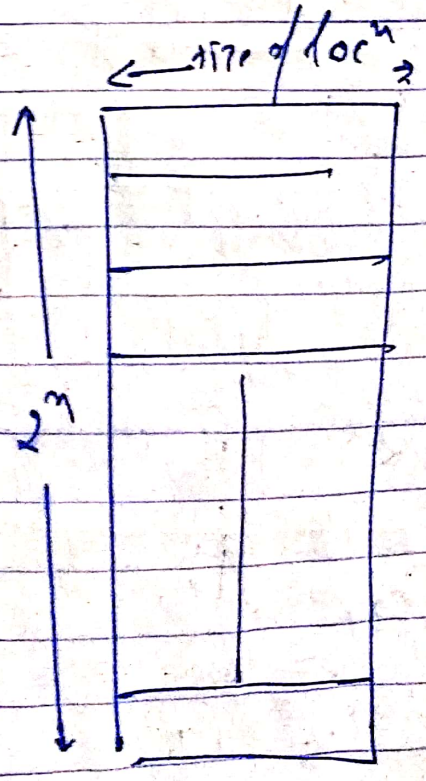
= 1024 B

= 1KB

In General,

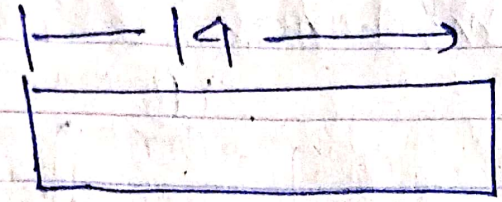


$n \longrightarrow 2^n \times 1B$.



Size of location = 1B, if not given

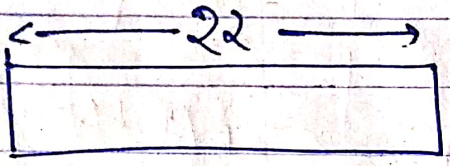
Ex:

I) 

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

$= 2^4 \times 2^{10} \times 1B$

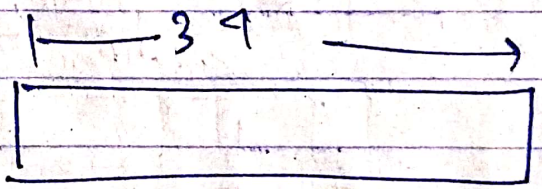
$= 16KB$

II) 

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

$= 2^2 \times 2^{20} \times 2B$

$= 8MB$

III) 

$2^{34} \times 1B$

$= 2^{30} \times 16$

$= 16GB$

$2^{10} = 1K$

$2^{20} = 1M$

$2^{30} = 1G$

$2^{40} = 1T$!!

$2^{50} = 1P$.

Now go in Reverse,

If size of memory is given, then what will be the bits.

Ex.

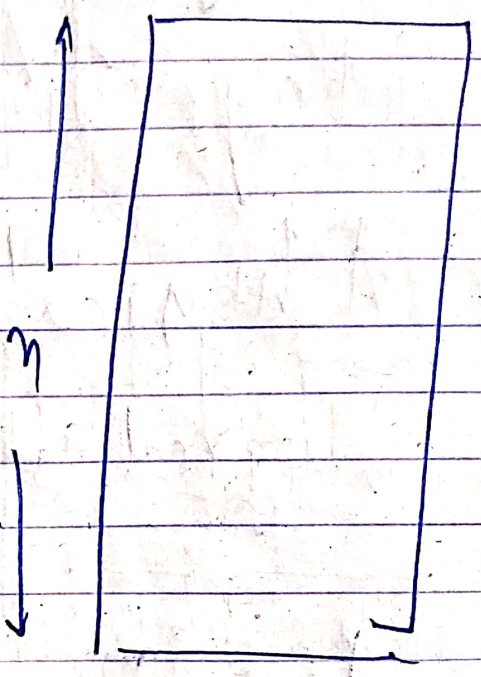
$$M.S = n \times 1B$$

$$n = \frac{64KB}{1B}$$

$$n = 64K$$

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

$$n = 2^{16} \text{ locations.}$$



Hence 16 bit.

Ex. 32 KB (1B) $= \frac{32KB}{1B} = 32K \rightarrow 2^{15}$ location

↓

15 bits.

Ex. 256 MB (1B) $= \frac{256MB}{1B} = 256M = 2^{28}$ location

↓ 28 bits

Ex. 16 GB (4B) ; $\frac{16GB}{4B} = 4GB = 2^{32}$ location

↓ 32 bits.

8 GB

#

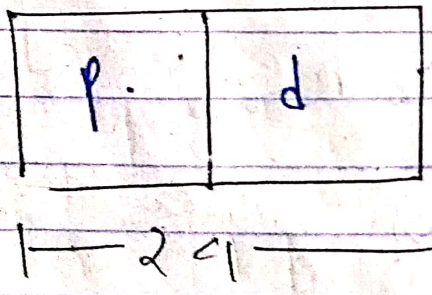
Ex:

Logical add = 24 bit

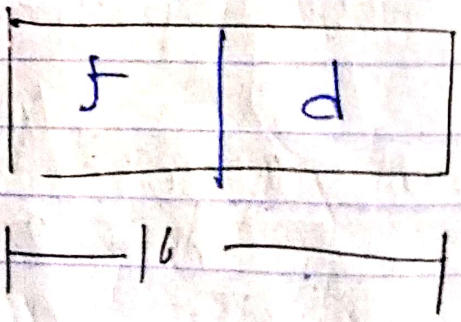
Page size = 1KB

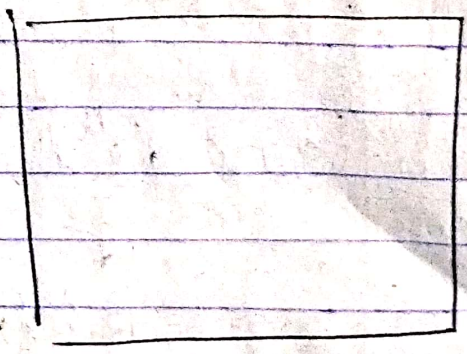
Physical Ad = 16 bit

Logical add

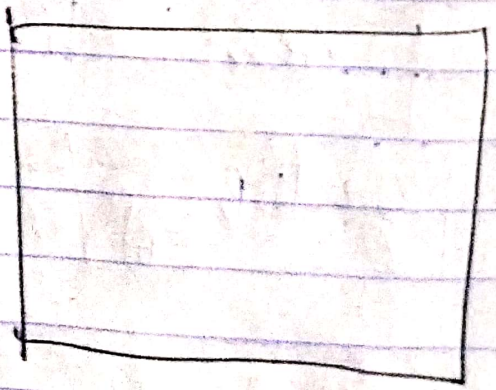


physical add





SM



MM

for SM

2^{29} locations, 16 M locations

Sup. each loc^n size is 1B.

SM → 16 MB

Page size = 1KB

each loc^n is of 1B, 1page accepts = 1K loc^n

hence $1K = 2^{10} \Rightarrow d = 10$ bits

So $p = 29 - 10 = 19$ bits.

hence 2^{19} pages.

Similarly for main memory,

MM size = $2^{16}B = 2^6 \cdot 2^{10}$

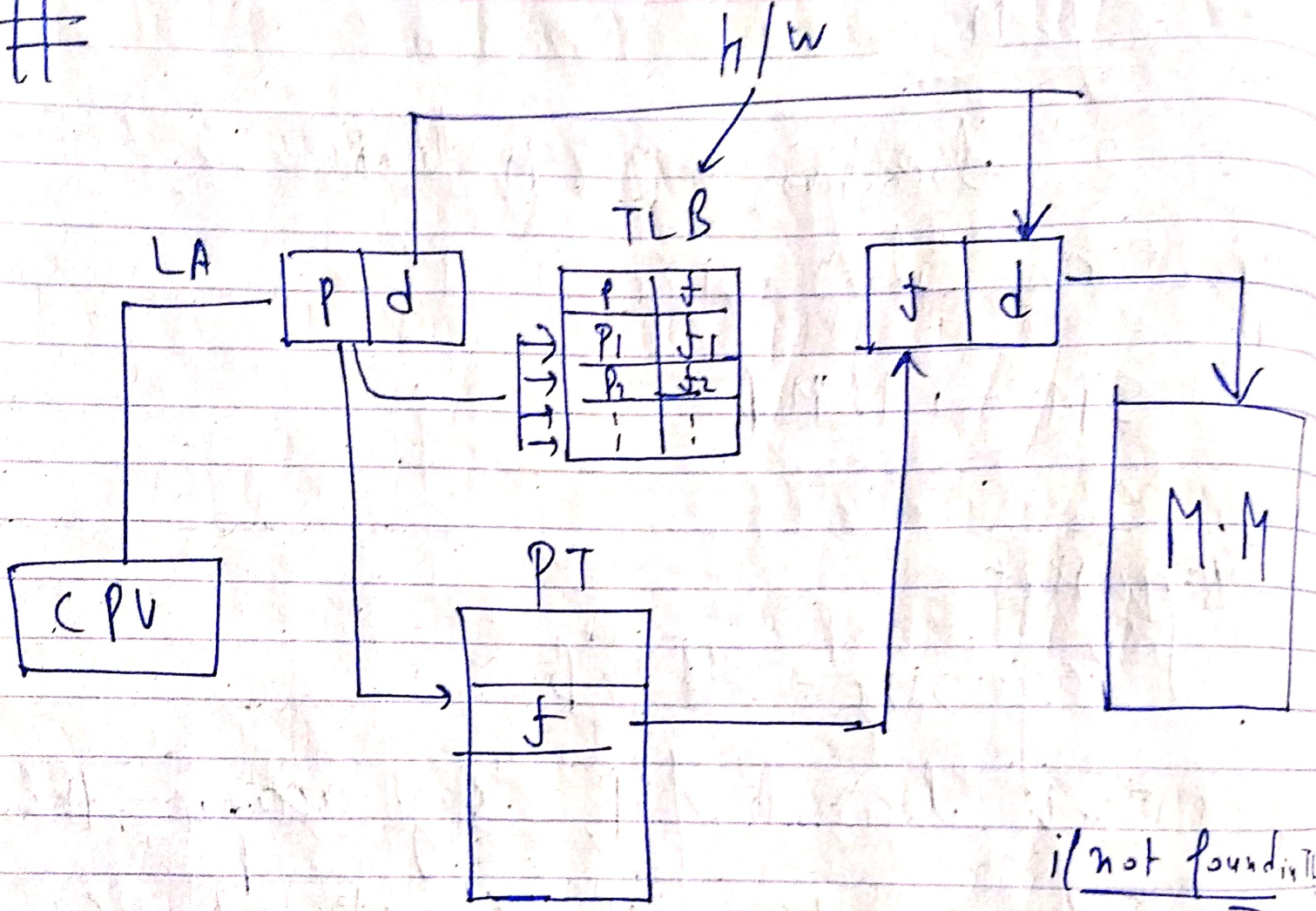
$= 64KB$

$d = 10$ bits

$f = 16 - 10 = 6$ bits. $\Rightarrow 2^6$ frames.

PT → which page in frame in which frame

#



if not found in TL

First check in TLB for page no. then PT.

↳ TLB is of very small size, so searching is very fast.

* More than 90%.

When another process came, then have to delete TLB, due to context switch

(3) With large no. of context switch, then TLB approach fails.

#

Ex

MM access time = 400 ns

TLB access time = 50 ns

Hit at TLB = 90%.

for accessing data in MM

for accessing PT in MM

$$T_{avg} = 0.9\,[50 + 400] + 0.1\,[50 + 400 + 400]$$

$$= 490 \text{ ns}.$$

General formula

$$\frac{\text{hit ratio}}{100}\,[\text{TLB} + \text{MM}] + \left[\frac{\text{po\_hit ratio}}{100}\right][\text{TLB} + \text{MM} + \text{MM}]$$

PT is meta data

Issue of context switch in TLB can be removed by -

I) By having multiple copy of TLB so that data of previous process is stored in TLB (some other TLB)

II) Some part of TLB is reserved for OS, so that if OS take charge it does not affect the currently running processes.

Ex What is the total space wasted in maintaining PT.

MM = 64MB

LA = 32 bits

PS = 4KB.

MM

64MB ⇒ 64M locn,

⇒ PA = 26 bits.

PS = 4KB

⇒ 4K locn for page, 2^{12} ⇒ d = 12.

f = 26 − 12 = 14 bit.

p = 32 − 12 = 20 bits,

No. of pages 2^{20} = 1M

$f = 14 \text{ bit} \simeq 2 B$

$\therefore$ PT size $= 1M \times 2B$

$= 2MB$

$\therefore$ 2MB wasted for PT. for all process.

ex

MM = 256MB

PS = 4KB

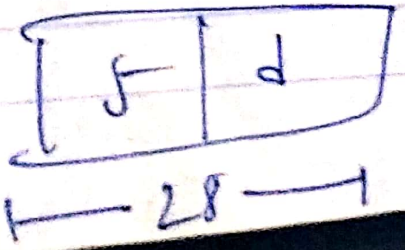
LA = 40 bit

Process size = 4MB

Find PT size for this process.

$\frac{MM}{B} \rightarrow \frac{256MB}{B} = 256M = 2^{28}$ B.

PA = 28 bit



P.S = 4KB , $\frac{4KB}{B}$ = 4K = 2^{12}

d = 12 bit

=) f = 16 bits.

≃ 2B.

No. of pages here = $\frac{4MB}{4KB}$ = 1K.

PT size of this process = 1K × 2B

= 2KB

Ex

SM = 256 GB

PS = 2KB

MM = 512KB

PT = 8KB , find size of process

SM = 256GB = 2^{38}

logical add = 38 bits.

Main Memory = 512 KB

512 KB / 1 B => PA = 19 bits

PS = 2 KB

No. of pages = 2 KB / 1 K = 2 K => 2^{11}

=> d = 11

f = 19 − 11 = 8

Size of PT = 8 KB (given)

No. of pages × size of page = 8 KB

No. of pages = 8 K.

No. of process size = 8 K × 2 KB

= 8 MB.

Ex PS = 8KB

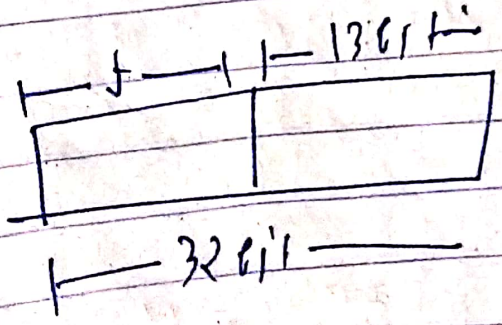
PA = 32 bit

PT = 24 MB

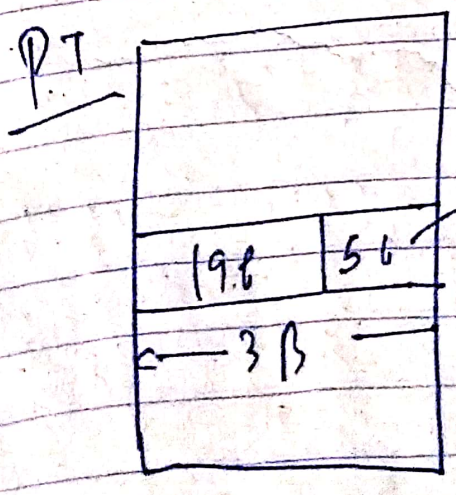
Find size of logical add.

$PS = \frac{8KB}{1B} = 8K$ pages.

$d = 13$ bits



$p = 32 - 13$
$= 19$ bits.



contiguity

No. of pages $8 \times 3B = 24MB$

No. of pages $= 8M$

$= 2^{23}$, $23 = p$

p	d

$23 + 13 = 36$ bits long.

Process Synchronization

Process

- Cooperative Process
- Independent Process.

*They share
i.e. variable
memory.
code.
Resources: cpu, printer.

int shared = 5 // 4 #Uniprocessor.

P1	P2
1. int x = shared	int y = shared.
2. x++	y--;
3. sleep(1);	sleep(1); // pause for 1 sec.
4. shared = x	shared = y.

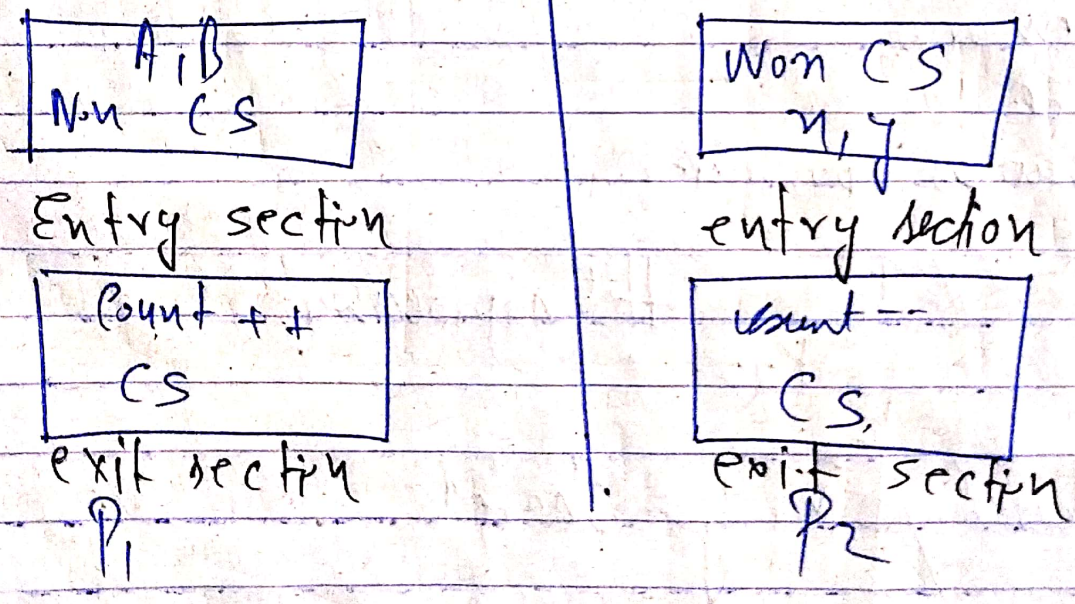
This is called Race Condition
& We need Synchronization.

Critical Section

It is part of the program where shared resources are accessed by various processes.

→ Place, where shared variable, resource are placed.

ex.



4 Condition for Synchronization.

1) Mutual Exclusion } Primary → must be followed.
2) Progress
3) Bounded Wait } Secondary
4) No assumption related to H/W speed.

2) Progress → When P_1 is not in CS, it should not allow P_2 too to enter in critical section.

① Mutual Exclusion → One at a time

③ Bounded Wait → Our process wait should not be wait for infinite time.

④ → It should not be case that it work on one OS but not in other OS

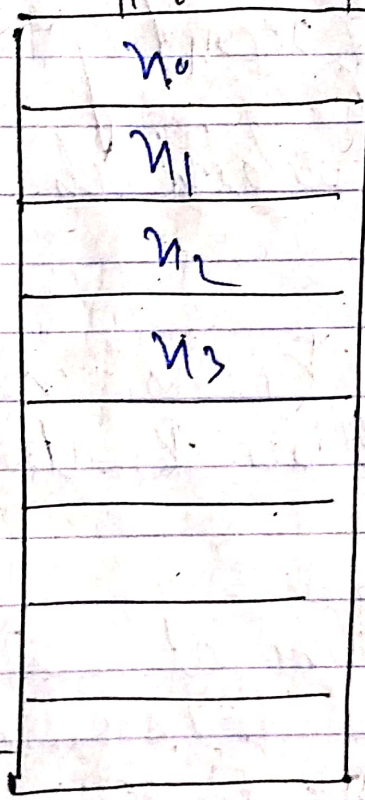
Producer - Consumer Problem

```
int count = 0.
void Producer(void)
{
    int itemp;
    while (true)
    {
        produce_item(itemp);
        while(count == n);
        Buffer[in] = itemp
        in = (in+1) % n;
        count = count + 1;
    }
}
```

```
void Consumer (void)
{
    int itemc;
    while (true)
    {
        while (count == 0);
        itemc = Buffer(out)
        out = (out+1) % n
        count = count - 1;
        process_item(itemc);
    }
}
```

n = 8

Buffer[0 - n-1]



In = 4

out = 0

count = 4

Buffer empty

1. load Rc, m[count]
2. DECR Rc
3. store m[count], Rc

Case

Producer I_1, I_2 Consumer I_1, I_2, Producer I_3, Cons I_3

Then we get wrong value of ~~count~~ count.

This is also called race condition.

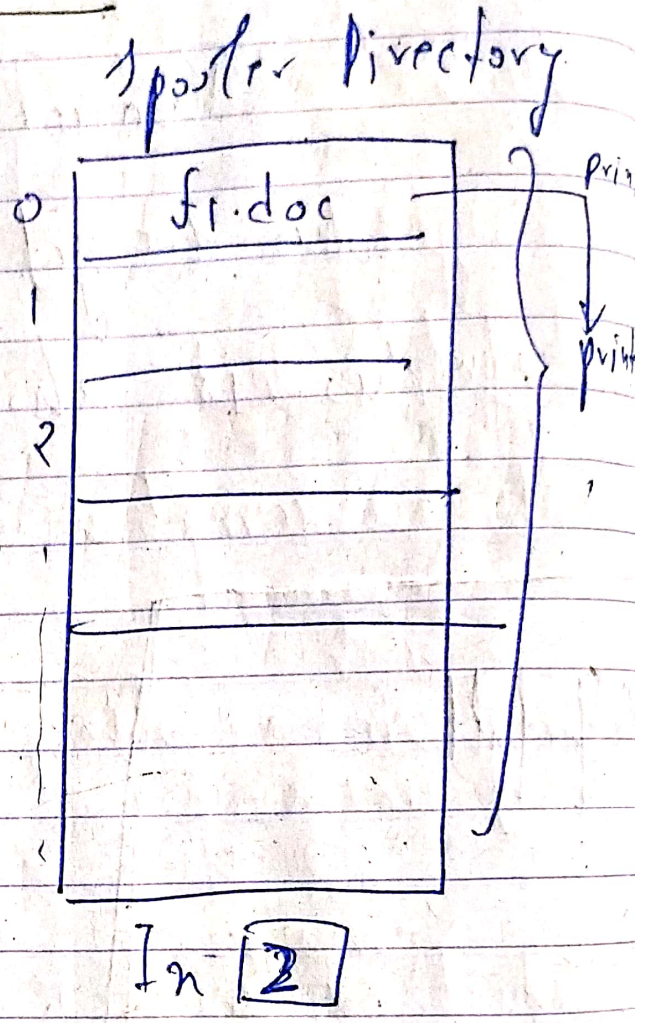
Buffer full

load R_{p_1}, m[count] I_1

INCR R_{p_1} I_2

Store m[count], R_{p_1} I_3

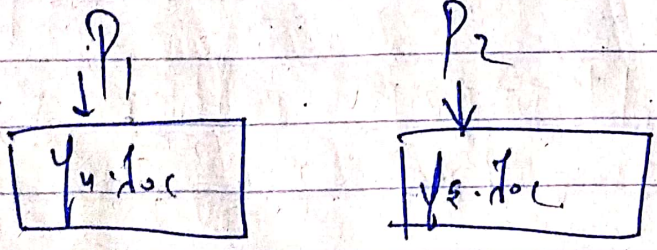
Printer Spooler Problem



1) Load Ri, m[in]
2) Store SD[Ri], "File-name"
   f1.doc
3) INCR Ri
4) Store m[in], Ri

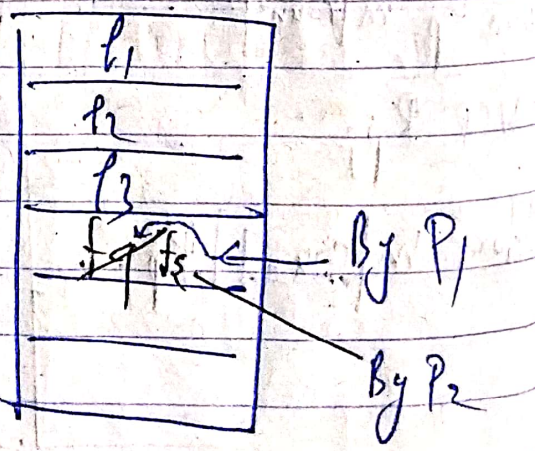
* In is shared variable

Case 1



P1 I1 I2 I3 | P2 I1 I2 I3 I4

Interrupt



hence loss of data

Semaphore

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

"Semaphore is an integer variable which is used in mutual exclusion manner by various concurrent cooperative process in order to achieve synchronization"

P(), Down, Wait	← Entry code.
V(), UP, Signal, Post, Release.	← exit code.

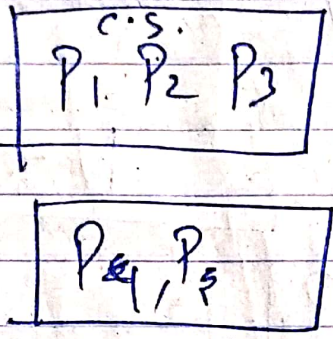
S → value from $(-\infty, 1)$

```
Down (Semaphore S)                    // Entry fn code
{
   S.value = S.value - 1
if (S.value < 0)
{  Put Process(PCB) in                // PCB → process control block.
   suspend list sleep();
}
else return;
}
```

```
Up (Semaphore S)                      // Exit fn code.
{  S.value = S.value + 1;
   if(S.value ≤ 0)
   {  Select a process
      from suspend list
      wake up();                      // place in ready queue
   }
}
```

* if S = −4, then there are 4 process in block state.

ex- if S = 3 initially



if S = 0, No process in suspend list.

ex- if S = 10, 10 process can go in C.S.

ex- if S = 10, then 6 P, & then 4 V.

10 − 6 = 4 + 4 = 8

ex- if S = 17, 5 P, 3 V, 1 P

17 − 5 + 3 − 1 = 19

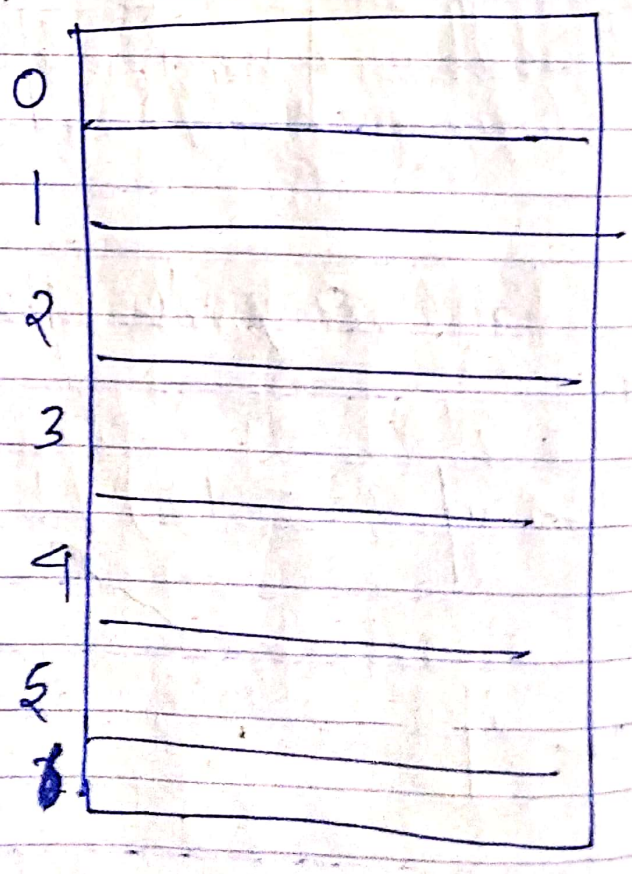
Producer, Consumer Problem

Counting semaphore — full → No. of filled slots
empty → No. of empty slots.

Binary semaphore S = 1,

Initially. full = 0
empty = N.

N = 8




```
Produce_item(itemp);
down(empty)
down(S)
Buffer[IN] = itemp;
IN = (IN+1) mod N
up(S)
up(full)
```

Consumer

down (full)

down (S)

item C = Buffer[out]

out = (out+1) mod n

Up (S)

Up (Empty)

#

Concurrency → to → Serializable using semaphore
~~Error~~

Reader - Writer Problem

Reader, writer using same database.

R - W	Problem	data loss
W R.	Problem	inconsistency
W W	"	
R R.	No Problem.	

```
int rc = 0                          // read count (no. of readers)
semaphore mutex = 1                 // Binary semaphore
semaphore db = 1
void Reader (void)
{
    while(true)
    {
        down(mutex);                // entry
        rc = rc + 1;
        if (rc == 1) then down(db);
        up(mutex)
        Data Base                   // CS
```

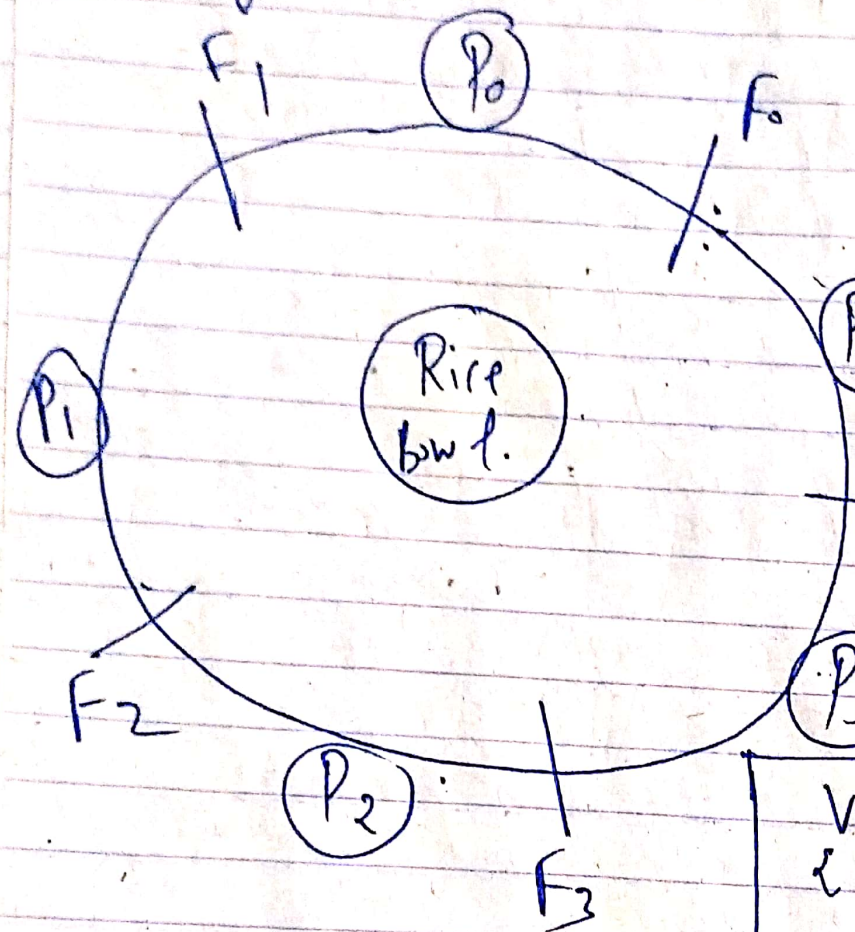
```
down (mutex)
rc = rc - 1;
if(rc == 0) then up(db);
up(mutex)
Process_data
}
}
```

exit.

```
Void writer(void)
{
  while(true)
  {
    down(db);
    DB
    up(db);
  }
}
```

Dining Philosopher Problem

They can think & eat.



W = No. of forks

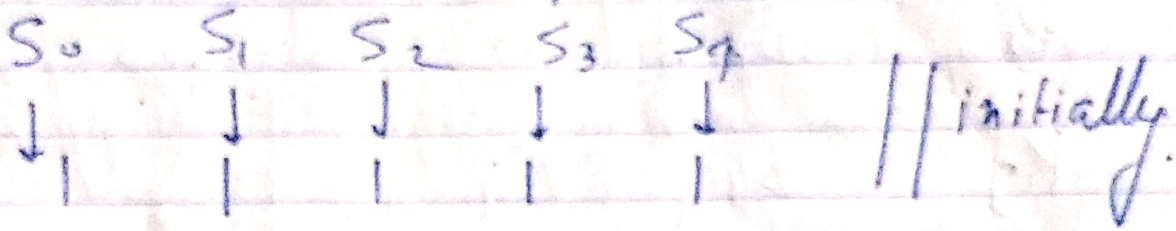
'5' philosophers

'5' Forks.

```
void philosopher (void)
{ while (true)
  { Thinking ();
    take-fork (i);
    take-fork ((i+1) % N);
    Eat ();
    put fork (i);
    put fork ((i+1) % N);
  }
}
```

To ensure deadlock free eating, we use array of semaphore.

S[i] give semaphore



⇒
```
void philosopher (void)
{
    while (true)
    {
        thinking ();
        Wait (take_fork (Si));
        Wait (take_fork (S((i+1) % N))
        Eat ();
        Signal (Put_fork (Si));
        Signal ( Put_fork (S(i+1) % 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

// Deadlock is possible here.

* When every process philosopher waiting for right fork.

To Remove Deadlock.

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_0	S_4

// First right, then left fork.

We can change the flow for any one philosopher to remove deadlock

i.e. $(N-1)^{th}$ wait (fork-for(i+1)%N)
wait (fork-for i)

Binary Semaphore

initially S = 1

0 1

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

```
UP(Semaphore S)
{
  if(S.value == 0)
  {
    S.value
  }
  if(Suspend list is empty)
  {
    S.value = 1;
  }
else
{ select a process
  from suspend list &
  wake up();
}
```

P() → Down | mutex → Binary semaphore.
V() → Up.

Ex

Each Process P_i $\{i = 1$ to $9\}$ execute the following code

repeat
entry → P(mutex)
CS
exit → V(mutex)
forever.

Process P_{10} execute the following code

repeat
V(mutex) ← entry
CS.
V(mutex) ← exit
forever.

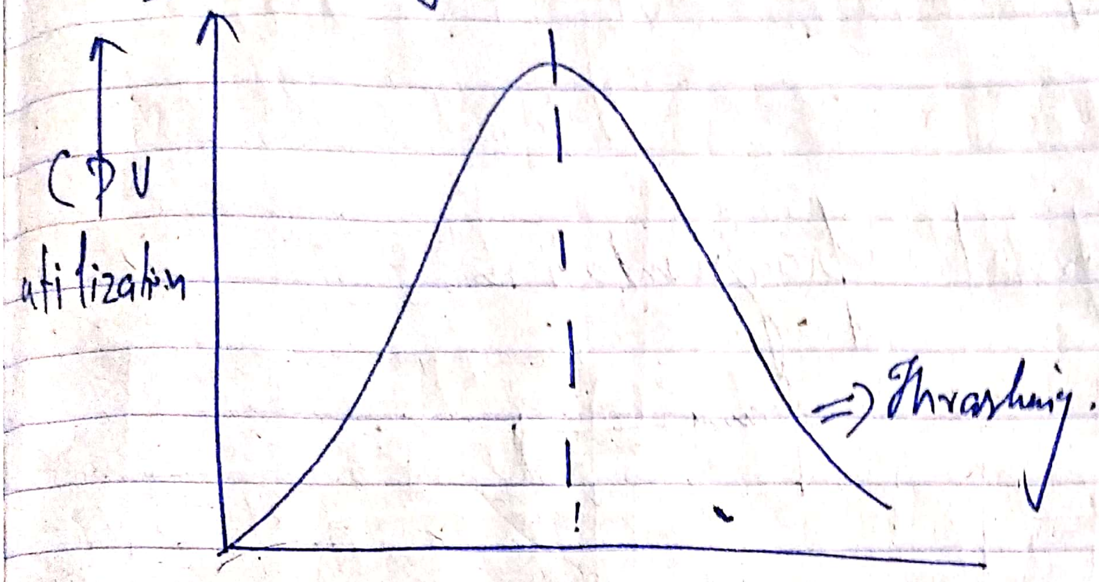
Q What is the max^m no. of process that may present in CS at any point of time.?

Ans 10

Q if exit of P_{10} is P(mutex)

Ans 3

Thrashing



Degree of multiprogramming

Thrashing → When CPU is busy more in performing ~~size~~ page fault.

P_1	1st
P_2	3rd
P_3	1st
P_4	1st

Multiprogramming will increase but due to larger page fault thrashing occur.

Sol^n

I) ↑se M/M size

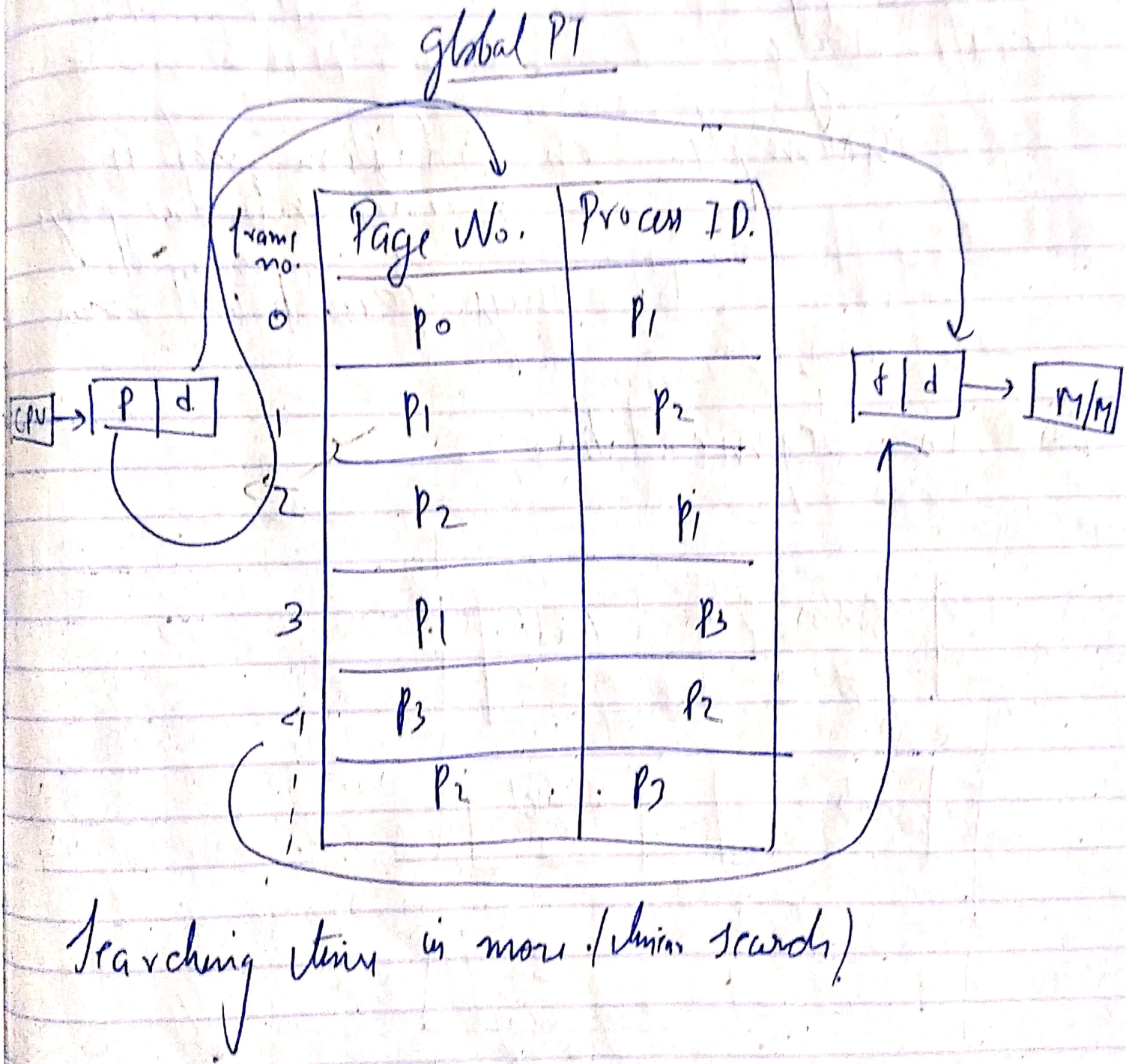
II) Long term scheduler, don't allow v. large no. of process in main memory.

No. of entries in PT = 16 K

Inverted Paging

1) Each Process has its own page table
2) PT will be in main memory.

To avoid storing PT of every process in MM
We use global PT to save space in MM.



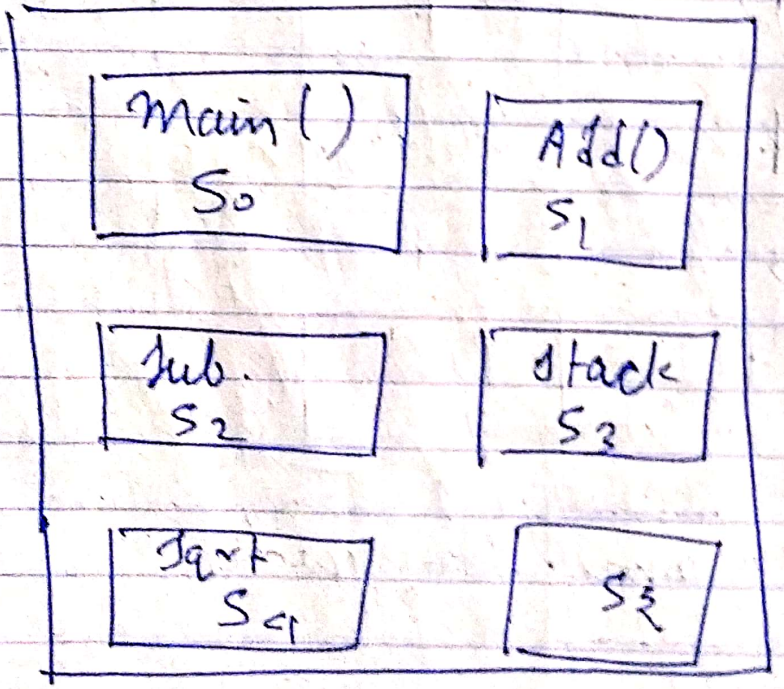
Searching time is more (linear search)

Segmentation

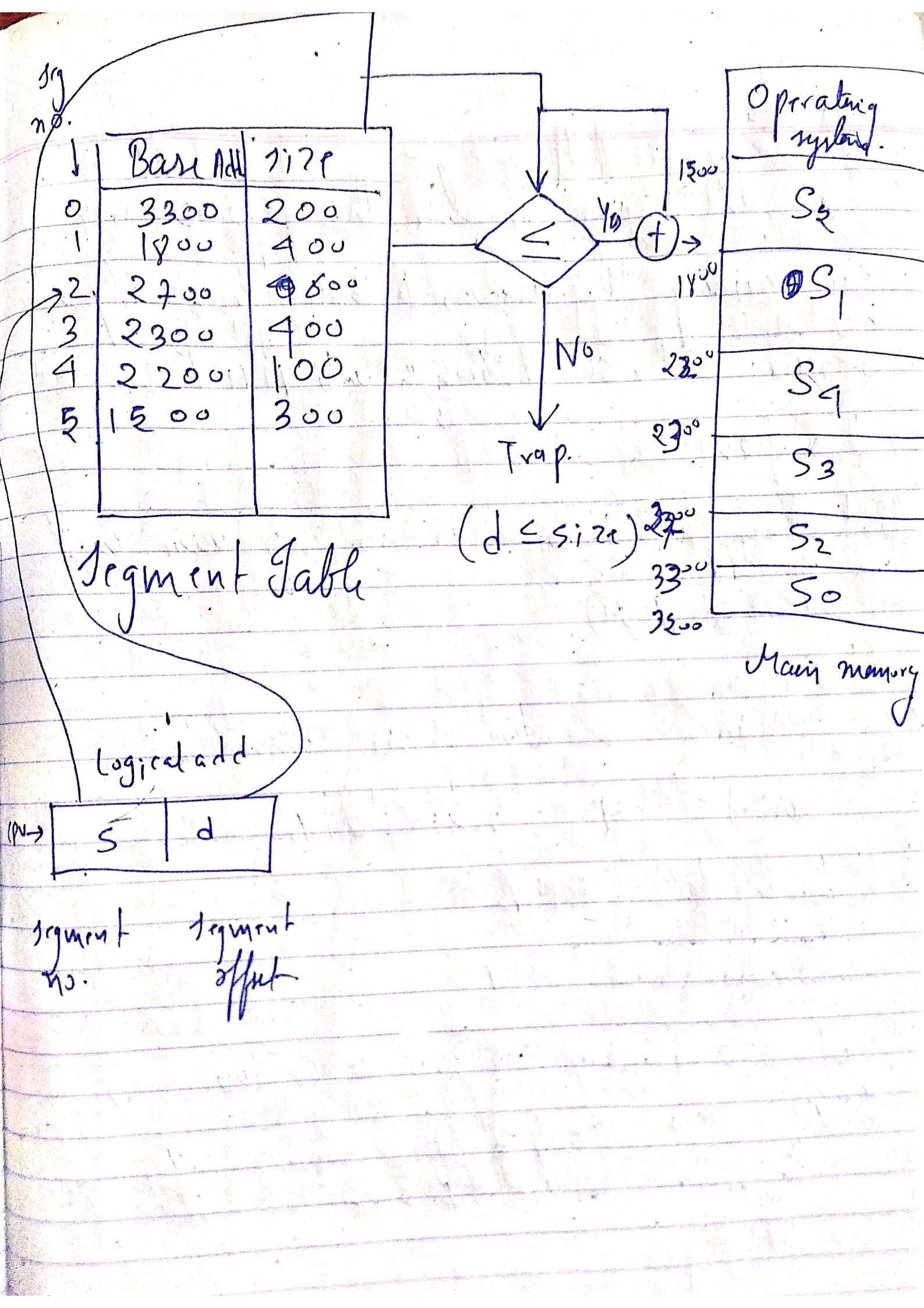
In paging, we divide code into fix no. of page

for example: In addition code is divided into two page the page fault will occur.

But in segmentation,



Size of segments may be different segments the related data.


Seg no.
Base Addr
Size
0 3300 200
1 1800 400
2 2700 600
3 2300 400
4 2200 100
5 1500 300
Segment Table
Logical add
PN
S
d
Segment no.
Segment offset
≤
Yes
+
No
Trap.
(d ≤ size)
Operating system
1500
S5
1800
S1
2300
S4
2700
S3
3300
S2
3500
S0
3900
Main memory

Overlay

When MM size is less than process size. by dividing the process & those fun^{n} of process require 1st, should brought into MM & then 2nd, 3rd & so on.

But this division is done by user, used in embedded system (limited MM).

Ex: Consider a two pass assembler

pass 1 : 80KB , pass 2 : 90KB

Symbol table : 30KB

Common routine : 20 KB

At a time only one pass is in use.

What is min partition size required if overlay driver is 10KB. size

80
90
30
20
10
230KB > (MM size) // Then we use overlay.

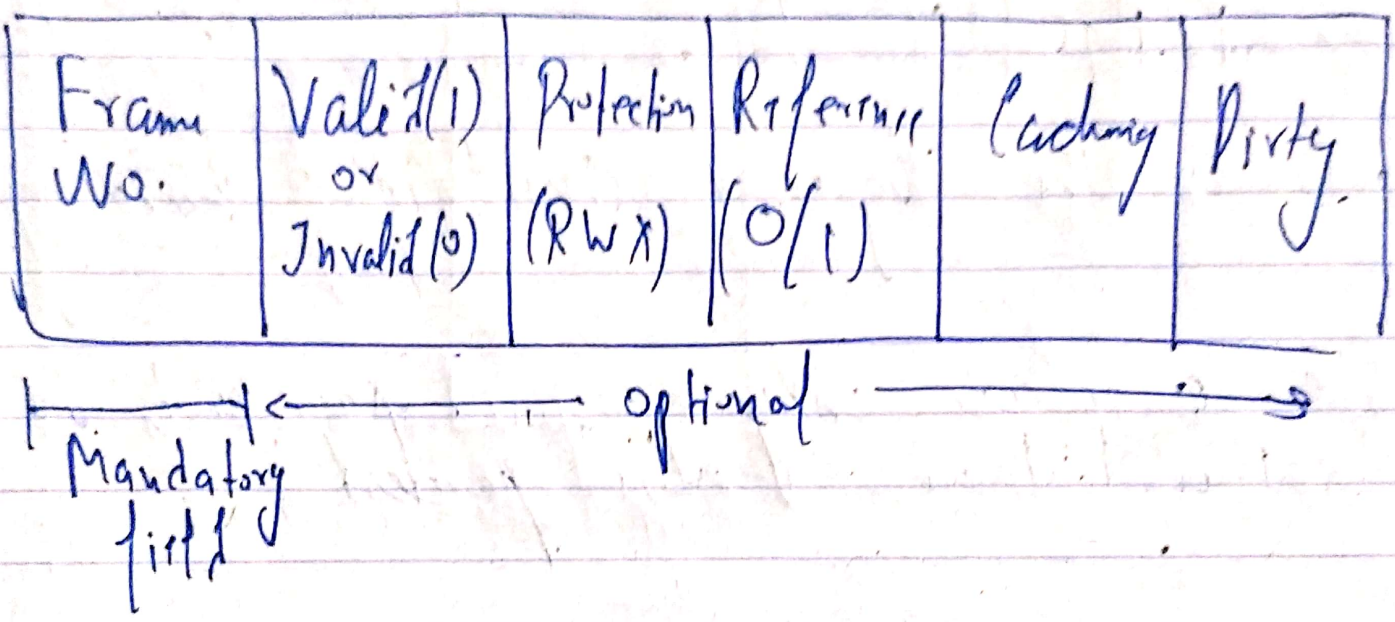
Pass 1	Pass 2
80KB	90KB
30KB	30KB
20KB	20KB
10KB	10KB
140KB	150KB

max(140KB, 150KB) = 150KB

~~Memory size~~

To 150KB partition or MM should be present.

Page Table Entry



Valid → Present, 1

Invalid → Absent, 0. (Page fault)

RWX → Read, Write, Execute.
(Permissions)

Reference (0/1) → LRU

Caching → enable / disable // for faster fetching

Dirty → modified or not.

Virtual Memory

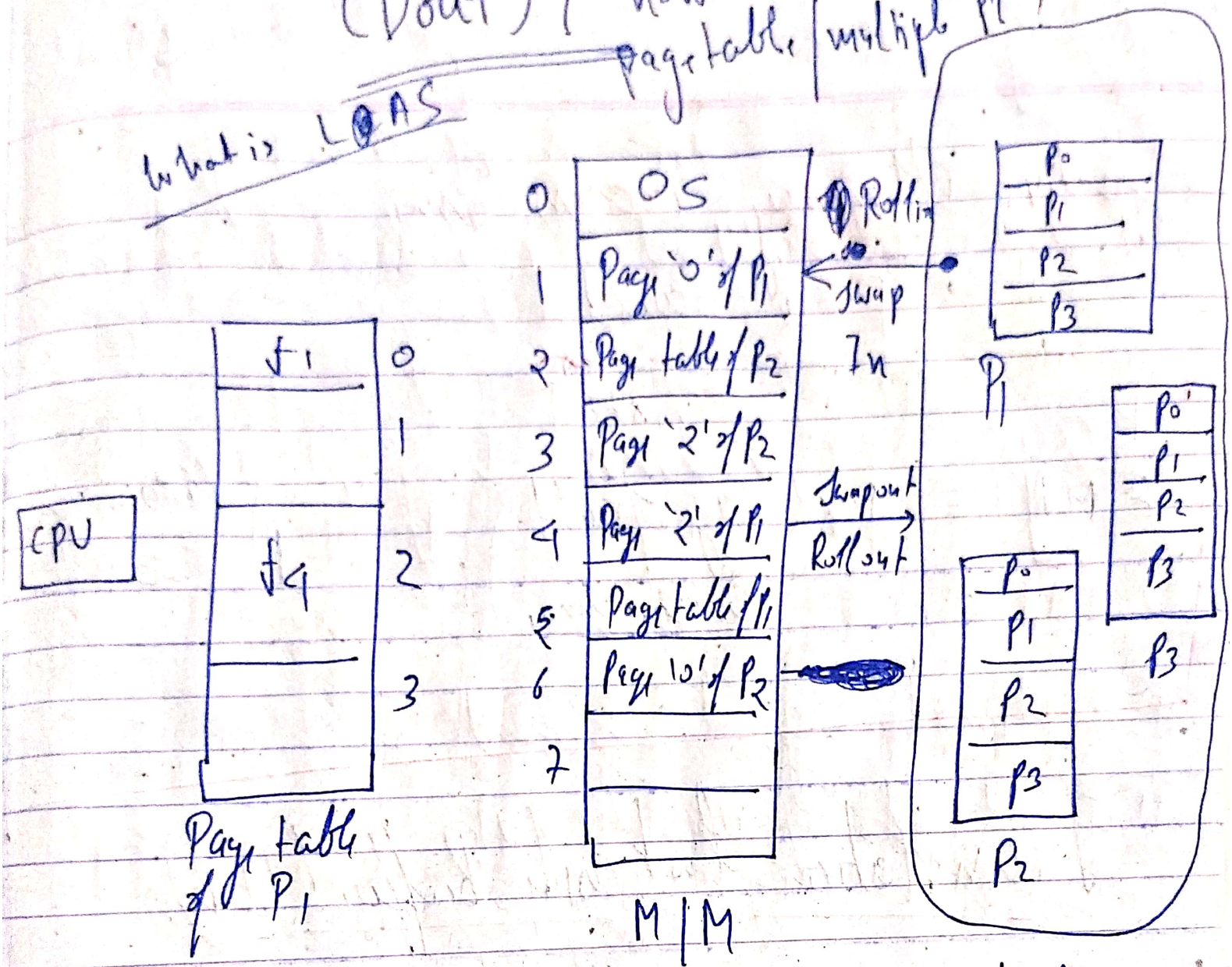
1) We can execute process, whose size is greater than MM.

2) Increase the degree of multiprogramming.

* We place only required pages into MM. rather than entire process.

(Doubt)? how stored page table / multiple PT?

What is LAS



If Page no. is absent in PT, then page fault.
In this case, OS take control, required page is fetched from LAS to MM & then again OS give control to user.

Effective Memory access time

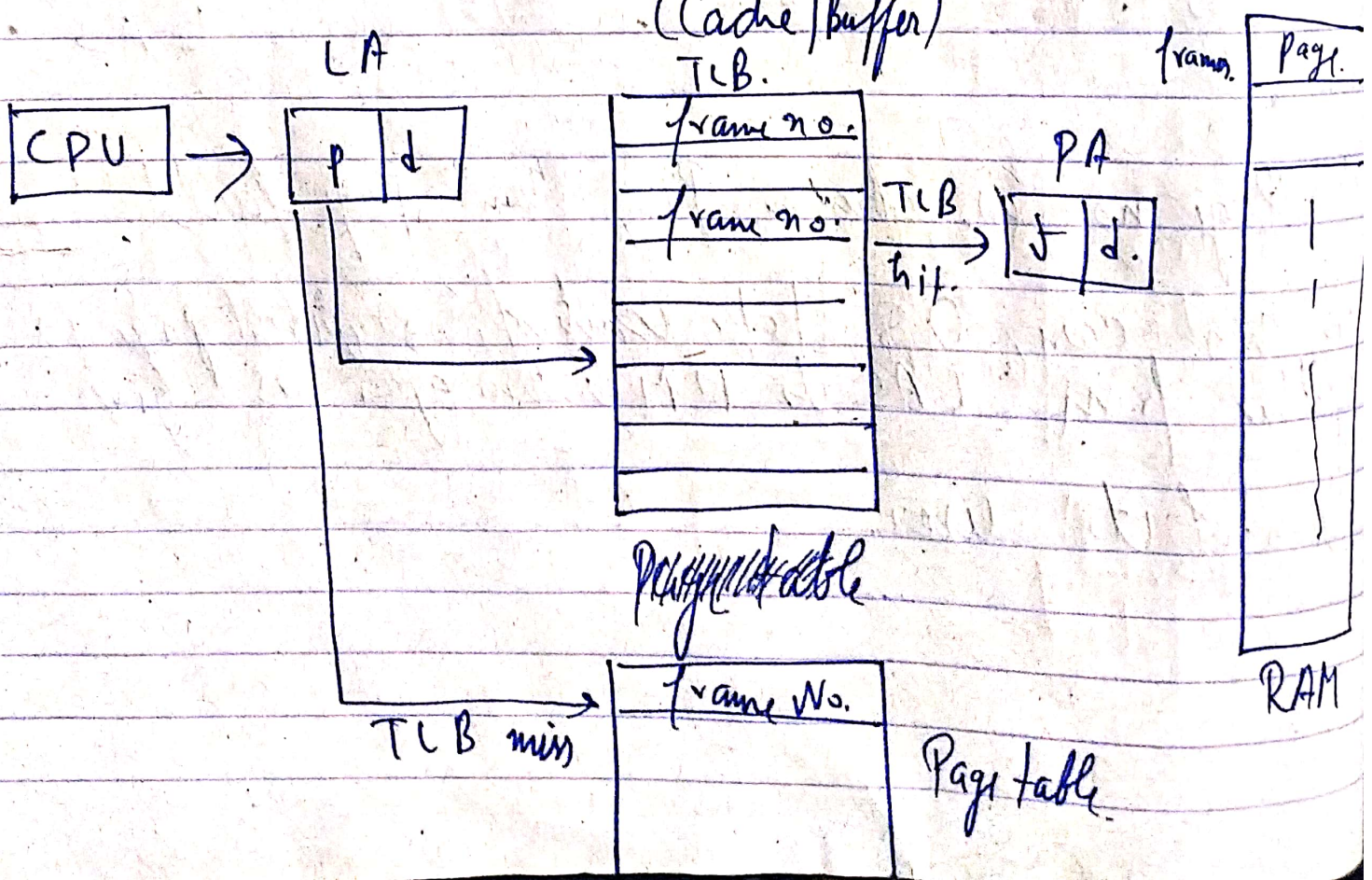
'p' → page fault

(msec)

$$EMAT = p \times (\text{page fault service time}) + (1-p)(\text{main mem access time})$$

(nsec)

Translation lookaside Buffer (TLB)



n → access time in RAM.

TLB is faster, but limited in size.

$$EMAT = p(TLB + n) + (1-p)(TLB + n + n)$$

p → hit %

(1-p) → miss %.

Ex: TLB access time = 10 ns

Main memory access time = 50 ns.

if hit ratio = 90% that is no page fault.

find EMAT.

$$EMAT = 0.90 \times |10 + 50| + 0.10 |10 + 50 + 50|$$

$$\text{hit} \times |TLB + MM| + \text{miss} |TLB + MM + MM|$$

↑ page table.

* Page fault => if Page is not present in main memory then we have to bring the page from SM

Page Replacement Algorithm's

1) FIFO
2) Optimal Page Replacement
3) Least Recently used (LRU)

FIFO Page Replacement

f3			1	1	1	1	0	0	0	3	3	3	3	3	2	2
f2		0	0	0	0	3	3	3	2	2	2	2	2	1	1	1
f1	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0
	*	*	*	*	hit	*	*	*	*	*	*	hit	hit	*	*	hit

Reference String : 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 1, 2, 0

Page fault = 12

Page hit = 3

$$\text{hit ratio} = \frac{3}{15} \times 100$$

$$\text{miss ratio} = \frac{12}{15} \times 100$$

Belady Anamaly, FIFO

F3			3	3	3	2	2	2	2	2	4	4
F2		2	2	2	1	1	1	1	1	3	3	3
F1	1	1	1	4	4	4	5	5	5	5	5	5
	*	*	*	*	*	*	*	hit	hit	*	*	hit.

hit = 3
page fault = 9

Ref. → 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.

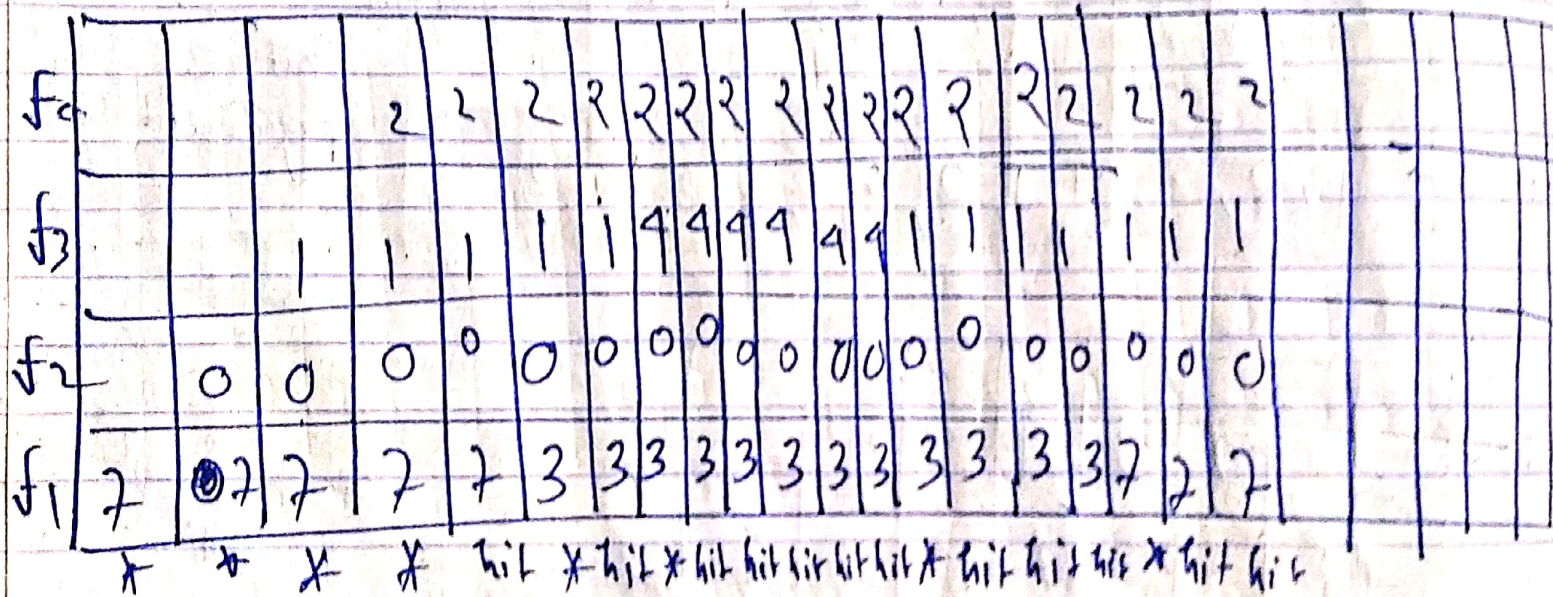
4			4	4	4	4	4	4	3	3	3	
3			3	3	3	3	3	3	2	2	2	2
2		2	2	2	2	2	2	1	1	1	1	5
1	1	1	1	1	1	1	5	5	5	5	4	4
	*	*	*	*	hit	hit	*	*	*	*	*	*

hits = 2
page fault = 10

With increasing frame no., there is ↑ in page-fault & FIFO is suffering from belady anomaly anomaly

Optimal Page Replacement

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



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

Page fault = 8 , hit ratio = $\frac{12}{20} \times 100$

hit = 12

page fault ratio = $\frac{8}{20}$ b/oo

: east Recently Used

Replace the least recently used page in past.

			2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
		1	1	1	1	1	4	4	4	4	4	4	1	1	1	1	1	1	1
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	7	7	7	7	3	3	3	3	3	3	3	3	3	3	3	3	7	7	7
*	*	*	*	*	hit	*	hit	hit	hit	hit	hit	*	hit	hit	hit	*	hit	hit	hit

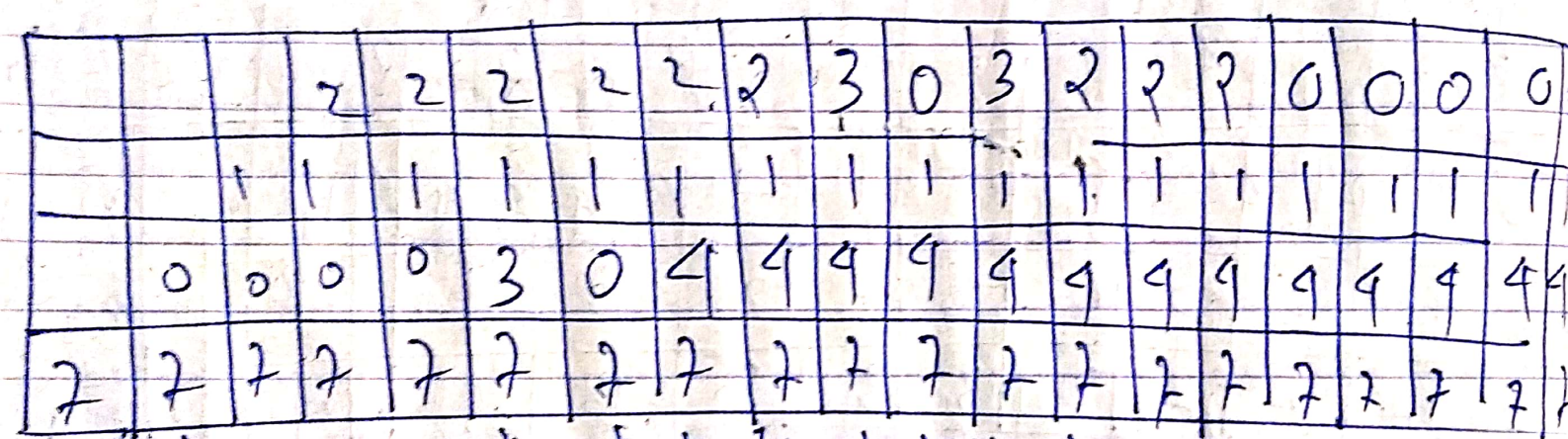
Ref. string → 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

hit = 12

miss = 8

Most Recently Used

Replace the most recently used page in past.

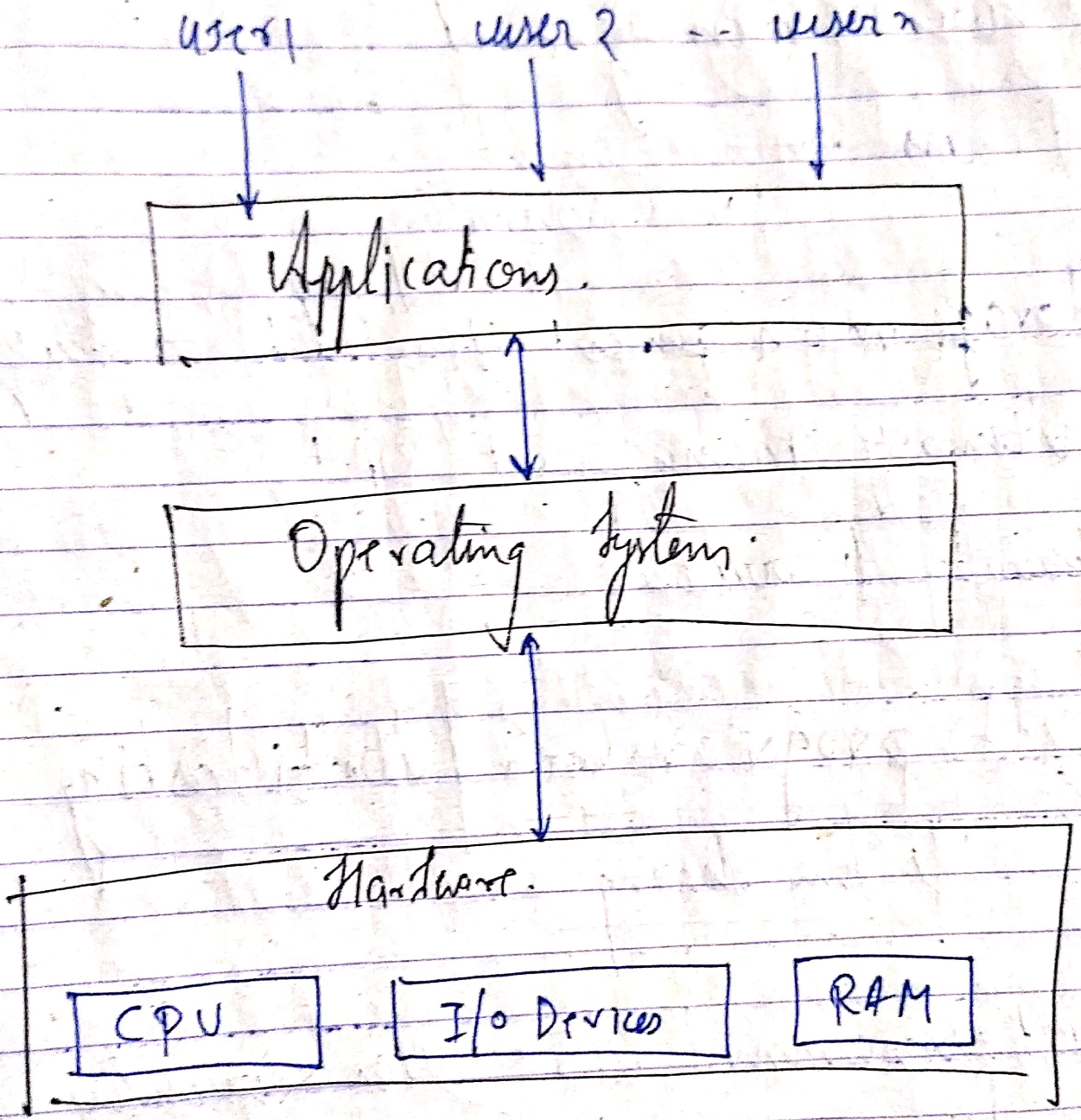


Ref String → 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0

Page fault = 12

Page hit = 8

Operating System & its function



Windows → Convinience

Unix → Throughput.

Mac

Functionality of OS

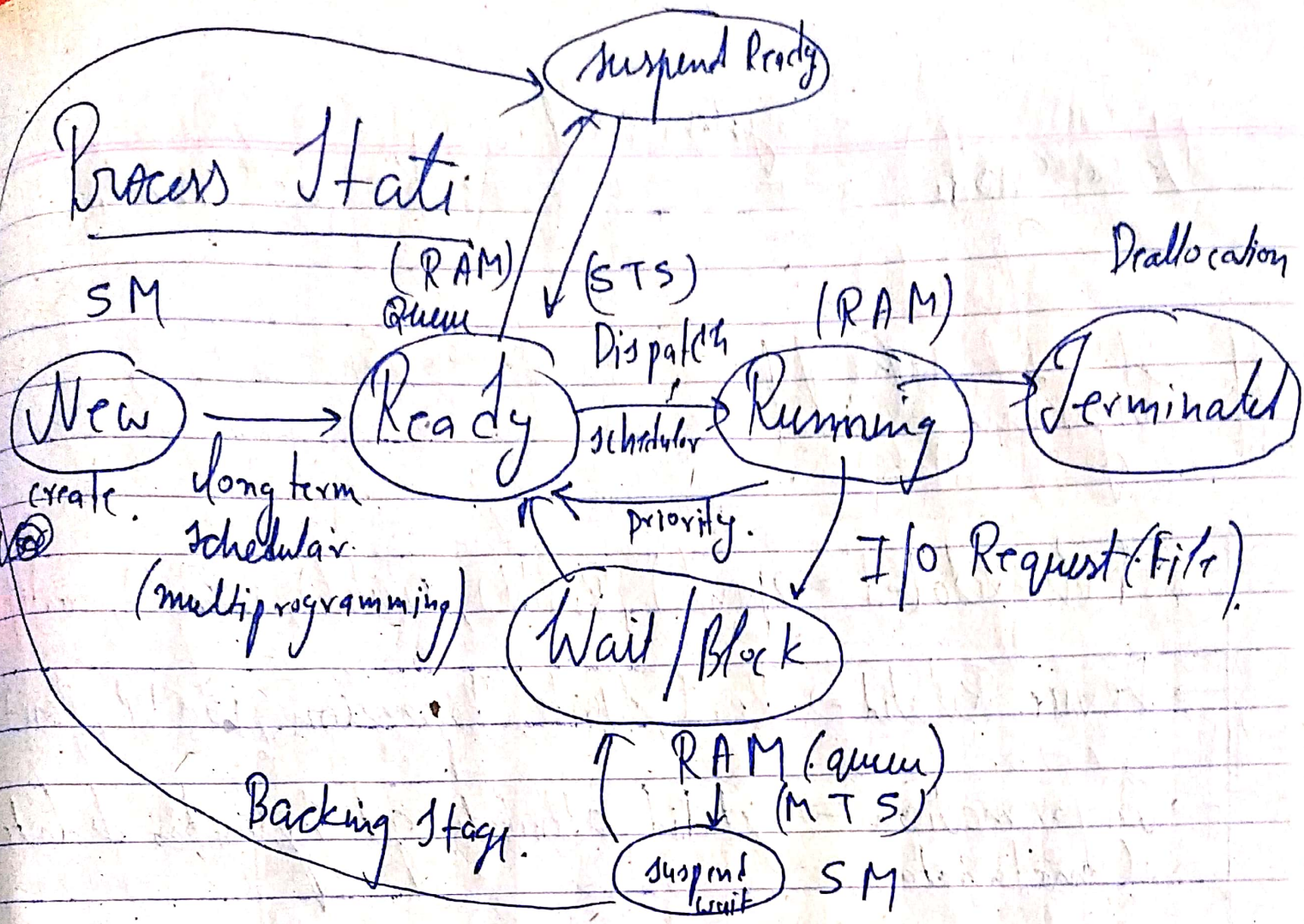
1) Resource management.
2) Process management (CPU scheduling)
3) Storage management (Hard disk) → file system.
4) Memory management (RAM)
5) Security & Privacy.

Multiprogrammed / Multitasking & Time sharing

Multiprogrammed OS → Non-preemtive
Multitasking / Time sharing OS → preemtive.

Multiprogrammed → for non IDLENESS of CPU.
Multitasking → Responsive.

Process State



New → (create) → Ready → Running → Terminated

Running → I/O Request (File) → Wait/Block → Ready

Wait/Block → (priority) → Ready

Ready → (Dispatch / scheduler) → Running

Ready → (long term scheduler (multiprogramming)) → New

Ready ⇄ (STS) ⇄ Suspend Ready

Wait/Block → (RAM (queue) → (MTS)) → Suspend wait

Suspend Ready ⇄ Backing Store ⇄ Suspend wait

Running → (Deallocation) → Terminated

User mode —system call.→ Kernel mode

System Call

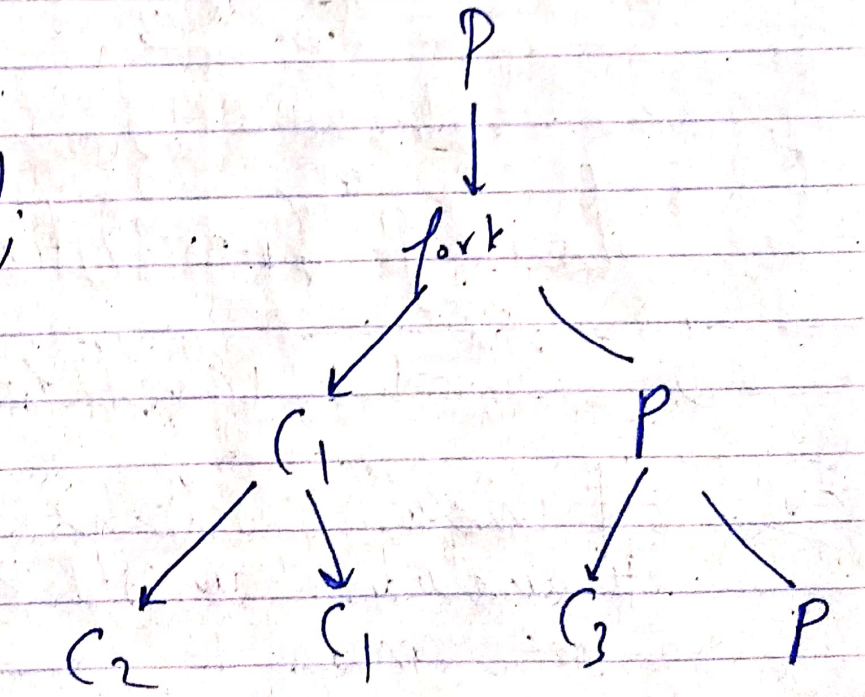
- → File Related ⇒ open(), Read(), write(), close(), create file
- → Device Related ⇒ Read, Write, Reposition, ioctl, fcntl
- → Information ⇒ get pid, attribute, get system time & date. (meta data)
- → Process Control → execute, abort, " fork, wait, load signal, allocate etc.
  - fork → create child, multitasking.
- → Communication → Pipe(), create/delete connection, signal(),

Fork()

To create child process.

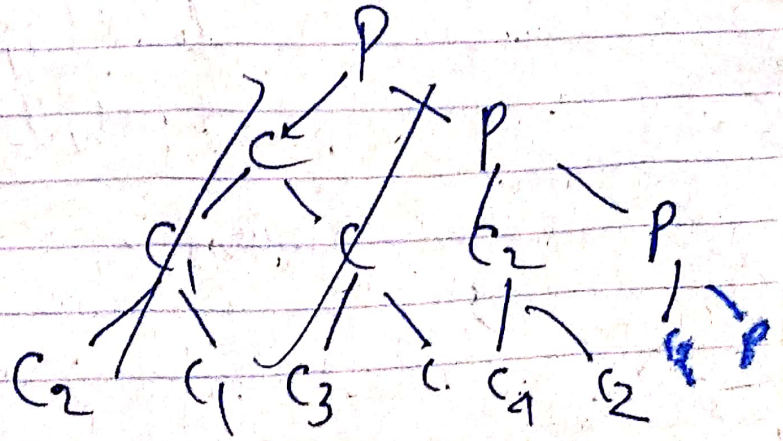
fork() → 0 child
→ +1, +ve → parent
→ -1 (child cannot be created)

Ex: main()
{
fork()
fork()
printf("hello");
}

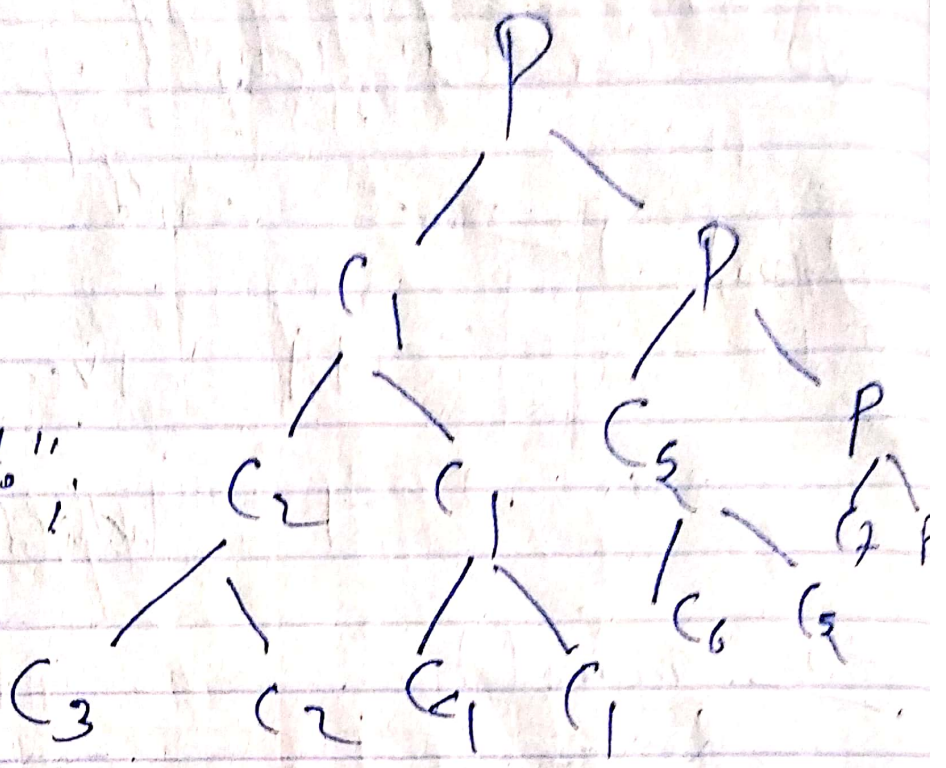


Hence 4 "hello"

Ex:
fork
fork
fork
printf("hello")



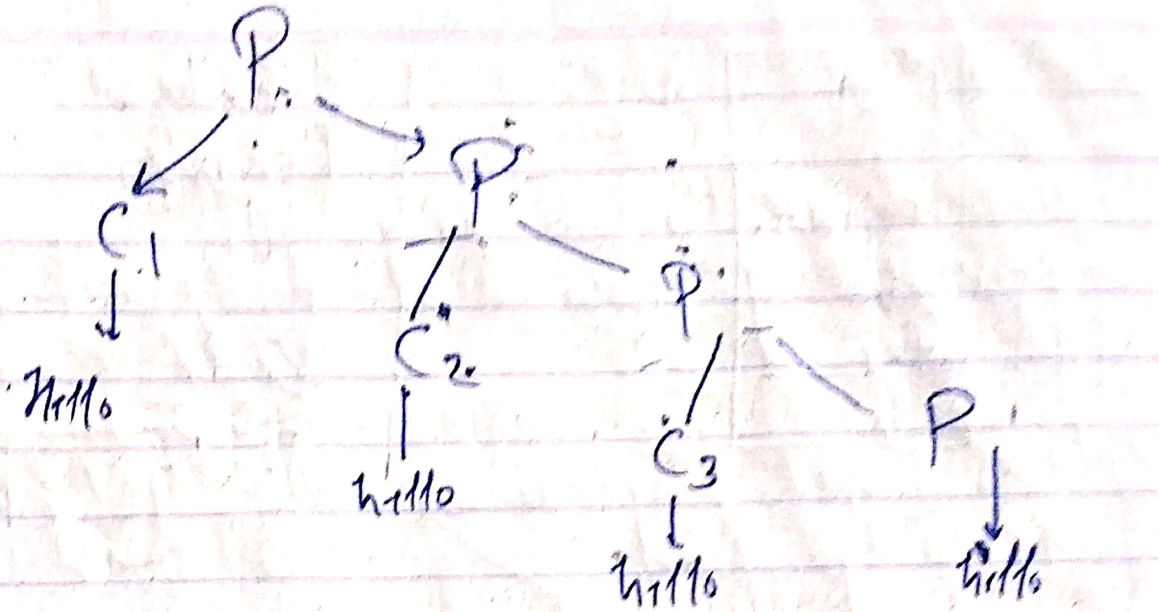
Ex
```
main()
{
    fork()
    fork()
    fork()
    printf "hello";
}
```



No. 2^n (i.e. 8) time hello will be printed

& $2^n - 1$ child will be created.

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

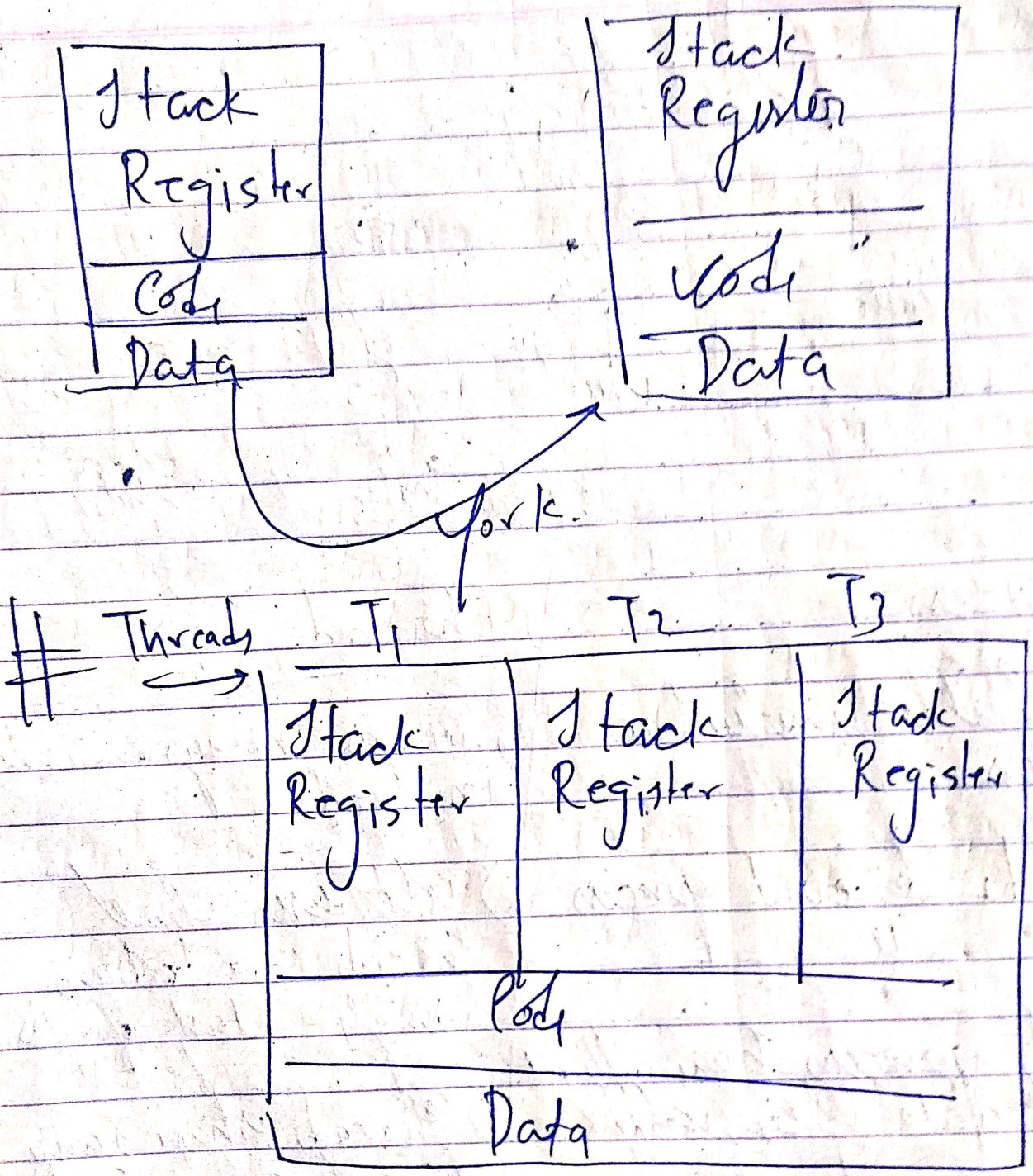


Process

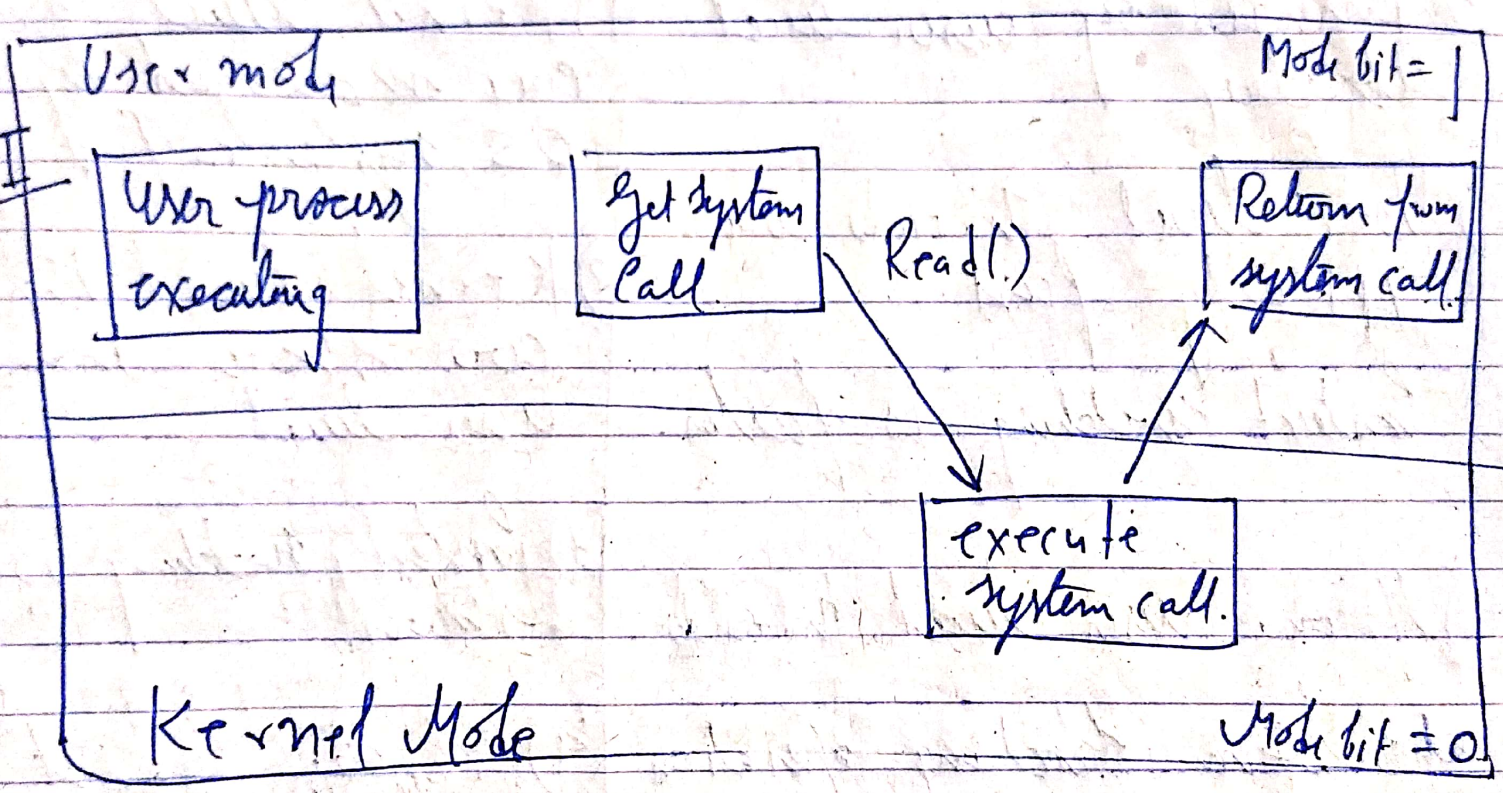
1) System Calls involved in process
2) OS treats different process differently
3) Different process have different copies of data files, code
4) Context switching is slower.
5) Blocking a process will not block another process.
6) Independent

Thread

1) There is no system call involved.
2) All user level threads treated as single task for OS
3) Threads share same copy of code & data
4) Context switching is faster
5) Blocking a thread will block entire process
6) Inter dependent


Stack Register
Code
Data
Stack Register
Code
Data
fork.
Threads
T1
T2
T3
Stack Register
Stack Register
Stack Register
Code
Data

"User mode V/s Kernel Mode"



User level Thread

1) User level threads are managed by user level library.
2) User level threads are typically fast.
3) Context switching is faster.
4) If one user level threads perform blocking operation then entire process get blocked.

Kernel Level Thread

1) Kernel level thread are managed by OS, system call.
2) Kernel level thread are slower than user level
3) Context switching is slower.
4) ~~If one~~ on kernel thread, No
4) If one kernel level thread blocked, No affect on other.

Process > KLT > ULT

(Context switching time)

1) Type of OS

2) Compiler

3) Linux Command.

~~Producer Consumer~~

~~Reader/Writer Problem~~

Peterson's Algorithm

```
while (1)
{
  flag[0] = T
  turn = 1
while (turn == 1 && flag[1] == T);
Critical Section
flag[0] = F;
}
```

```
while (1)
{
  flag[1] = T.
  turn = 0
while (turn == 0 && flag[0] == T);
Critical section
flag[1] = F;
}
```

MS
- Progress
- Bounded Wait.

Type of Process

I) Interactive
II) Batch

1) Batch OS → less CPU utilization
2) Multiprogramming OS → CPU is not idle
→ no-preemption
3) Multitasking OS → Extension of multiprogramming
(Time sharing multiprogramming with RR) → Preemption (time quantum / priority)
4) Multiprocessing OS → more than one CPU

Real time OS

I) Soft RTOS → Soft time limit.

II) Hard RTOS → Hard time bound.